60 行 NumPy 中的 GPT

发表评论 / ChatGPT, GPT, OpenAI, 信息安全文章

本文还是来自Jay Mody，那篇被Andrej Karpathy手动点赞的GPT in 60 Lines of NumPy。

LLM大行其道，然而大多数GPT模型都像个黑盒子一般隐隐绰绰，甚至很多人都开始神秘化这个技术。我觉得直接跳进数学原理和代码里看看真实发生了什么，才是最有效的理解某项技术的方法。正如DeepMind的Julian Schrittwieser所说：

> *这些都是电脑程序。*

这篇文章细致的讲解了GPT模型的核心组成及原理，并且用Numpy手搓了一个完整的实现（可以跑的那种），读起来真的神清气爽。项目代码也完全开源，叫做picoGPT(pico，果然是不能再小的GPT了)。

原文链接：GPT in 60 Lines of NumPy

译文链接：60行NumPy手搓GPT

(已获原文作者授权)

关于译文几点说明：

- 翻译基本按照原作者的表述和逻辑，个别部分译者做了补充和看法；
- 文中的个别英文术语很难翻译，算是该领域的专有名词了，因此这类术语就直接保留了，比如transformer

在本文中，我们将仅仅使用60行Numpy，从0-1实现一个GPT。然后我们将OpenAI发布的GPT-2模型的权重加载进我们的实现并生成一些文本。

注意：

- 本文假定读者熟悉Python，Numpy，还有一些训练神经网络的基本经验。
- 考虑到在保持完整性的同时让实现尽可能的简单，本文的实现故意丢弃了原始模型的大量功能和特点。目的很简单啊，就是提供一个**简单且完整的GPT的技术介绍，作为教学用途使用。**
- GPT架构只是LLM取得今时今日成就的一个小小组成部分[1]
- 本文中的所有代码都可以在这里找到:https://github.com/jaymody/picoGPT
- Hacker news上关于本文的讨论

更新(2023/2/9)：添加了"下一步呢？"部分，并且更新了介绍部分

更新(2023/2/28)：为"下一步呢？"部分又添加了一些内容

---

1. GPT是什么?

GPT代表**生成式预训练(Generative Pre-trained Transformer)**。这是一类基于Transformer的神经网络架构。Jay Alammar的"GPT3是如何工作的"一文在宏观视角下对GPT进行了精彩的介绍。但这里简单来说：

- **生成式(Generative)**：GPT可以生成文本
- **预训练(Pre-trained)**：GPT基于来自于书本、互联网等来源的海量文本进行训练
- **Transformer**：GPT是一个*decoder-only的transformer*神经网络结构译者注：Transformer就是一种特定的神经网络结构

类似OpenAI的GPT-3，谷歌的LaMDA还有Cohere的Command XLarge的大语言模型的底层都是GPT模型。让它们这么特殊的原因是**1)**它们非常的大（成百上千亿的参数）；**2)**它们是基于海量数据进行训练的（成百上千个GB的文本数据）

根本上来看，给定一组**提示(prompt)**，GPT能够基于此**生成文本(generates text)**。即使是使用如此简单的API（input = 文本，output = 文本），一个训练好的GPT能够完成很多出色的任务，比如帮你写邮件，总结一本书，给你的instagram起标题，给5岁的小孩解释什么是黑洞，写SQL代码，甚至帮你写下你的遗嘱。

以上就是宏观视角下关于GPT的概览以及它能够做的事情。现在让我们深入一些细节把。

1.1 输入/输出(Input/Output)

一个GPT的函数签名基本上类似这样：

```python
def gpt(inputs: list[int]) -> list[list[float]]:
    # inputs has shape [n_seq]
    # output has shape [n_seq, n_vocab]
    # 输入参数 'inputs' 是一个整数列表，表示序列，其形状为 [n_seq]，
    # 其中 'n_seq' 是序列的长度。

    # 函数的输出是一个二维浮点数列表，其形状为 [n_seq, n_vocab]，
    # 'n_vocab' 表示词汇表的大小，即模型能够输出的不同标签的数量。
    # 这意味着对于输入序列中的每个元素，模型都会输出一个与词汇表大小相等的概率分布。

    output = # beep boop 神经网络魔法
    # 上面的注释 "beep boop 神经网络魔法" 暗示这里是函数的核心实现，
    # 具体可能涉及到对输入数据的神经网络处理，但具体的实现细节在这段代码中并未给出。

    return output
    # 返回处理后的输出数据。
```

1.1.1 输入(Input)

输入是一些文本，这些文本被表示成**一串整数序列(sequence of integers)**，每个整数都与文本中的**tokens**对应：

```python
# integers represent tokens in our text, for example:
# 整数代表我们文本中的标记（tokens），例如：
# text   = "not  all    heroes wear capes":
#           并非 所有的 英雄   都穿 披风
# tokens = "not"  "all" "heroes" "wear" "capes"
inputs =   [1,     0,    2,       4,     6]
# 这里 'inputs' 是一个整数列表，其中的每个整数代表对应文本中的一个单词。
# 比如，'1' 可能代表 "not"，'0' 代表 "all"，以此类推。
# 这种映射关系（单词到整数的映射）通常是预先定义好的，基于某种方式进行单词的编号。
# 在自然语言处理的任务中，这样的转换是常见的第一步，
# 它将文本数据转化为模型可以处理的数值形式。
```

token是文本的小片段，它们由某种**分词器（tokenizer）**产生。我们可以通过一个**词汇表(vocabulary)**将tokens映射为整数：

```
# 词汇表中标记(token)的索引代表了该标记(token)的整数ID
# 例如，"heroes"的整数ID为2，因为 vocab[2] = "heroes"
vocab = ["all", "not", "heroes", "the", "wear", ".", "capes"]

# 一个假设的分词器，这个分词器基于空格进行分词
tokenizer = WhitespaceTokenizer(vocab)

# encode() 方法将字符串转换为整数列表
ids = tokenizer.encode("not all heroes wear") # ids = [1, 0, 2, 4]
# 这里，"not all heroes wear" 被分词并转换为对应的整数ID列表

# 我们可以通过词汇表映射来查看实际的标记(token)
tokens = [tokenizer.vocab[i] for i in ids] # tokens = ["not", "all", "heroes", "wear"]
# 通过索引访问词汇表，将整数ID列表转换回它们对应的标记(token)（单词）

# decode() 方法将整数列表转换回字符串
text = tokenizer.decode(ids) # text = "not all heroes wear"
# 将整数ID列表还原为原始的字符串文本
```

简单说：

- 我们有一个字符串
- 我们使用tokenizer将其拆解为小片段-我们称之为token
- 我们使用词汇表将这些token映射为整数

在实际中，我们不仅仅使用简单的通过空白分隔去做分词，我们会使用一些更高级的方法，比如Byte-Pair Encoding或者WordPiece，但它们的原理是一样的：

1. 有一个vocab即词汇表，可以将字符串token映射到整数索引
2. 有一个encode方法，即编码方法，可以实现str -> list[int]的转化
3. 有一个decode方法，即解码方法，可以实现list[int] -> str的转化[2]

1.1.2 输出(Output)

输出是一个**二维数组(2D array)**，其中output[i][j]表示模型的**预测概率(predicted probability)**，这个概率代表了词汇表中位于vocab[j]的token是下一个tokeninputs[i+1]的概率。比如：

```
# 定义一个词汇表，包含了一系列单词。
vocab = ["all", "not", "heroes", "the", "wear", ".", "capes"]

# 定义输入序列，这里使用整数列表表示文本中的单词，整数是词汇表中对应单词的索引。
inputs = [1, 0, 2, 4] # 对应于 "not" "all" "heroes" "wear"

# 调用gpt函数，传入上述输入序列。
output = gpt(inputs)
# 函数返回一个输出列表，每个元素是一个概率分布列表，表示下一个单词是词汇表中每个单词的概率。

# 下面是对输出的详细解释：

# output[0] =  [0.75    0.1     0.0       0.15    0.0   0.0    0.0  ]
# 给定输入"not"时，模型预测下一个单词是"all"的概率最高。

# output[1] =  [0.0     0.0      0.8     0.1    0.0    0.0   0.1  ]
# 给定序列["not", "all"]时，模型预测下一个单词是"heroes"的概率最高。

```

```
# output[-1] = [0.0     0.0     0.0     0.1     0.0    0.05  0.85  ]
# 给定完整序列["not", "all", "heroes", "wear"]时，模型预测下一个单词是"capes"的概率最高。
# 在Python中，-1作为索引表示列表中的最后一个元素。因此，output[-1]指的是output列表中的最后一个元素

# 这些输出展示了模型是如何基于给定的单词序列来预测下一个最可能的单词。
# 每个输出向量的每个元素代表词汇表中对应单词被预测为下一个单词的概率。
```

为了针对整个序列获得**下一个token预测**(**next token prediction**)， 我们可以简单的选择output[-1]中概率最大的那个token：

```
# 定义一个词汇表，包含一系列预定义的单词。
vocab = ["all", "not", "heroes", "the", "wear", ".", "capes"]

# 定义输入序列，这是一个整数列表，每个整数代表词汇表中对应单词的索引。
# 这里的输入序列代表的文本是 "not all heroes wear"。
inputs = [1, 0, 2, 4] # 对应于 "not" "all" "heroes" "wear"

# 调用gpt模型函数，传入输入序列，模型将基于这些输入预测下一个单词的概率分布。
# 'output' 是一个列表，其中每个元素代表对应于词汇表中每个单词作为下一个单词的概率。
output = gpt(inputs)

# 使用np.argmax函数从模型预测的最后一个单词的概率分布中找出概率最高的单词索引。
# 这里的 '-1' 代表列表的最后一个元素，即基于整个输入序列的预测结果。
next_token_id = np.argmax(output[-1]) # next_token_id = 6，表示概率最高的单词是词汇表中索引为6的单词

# 根据预测出的下一个单词索引（next_token_id），从词汇表中获取实际的单词。
# 这里的 "capes" 是根据模型预测，给定输入序列"not all heroes wear"的下一个最可能的单词。
next_token = vocab[next_token_id] # next_token = "capes"
```

将具有最高概率的token作为我们的预测，叫做greedy decoding或者**greedy sampling(贪婪采样)**。

在一个序列中预测下一个逻辑词(logical word)的任务被称之为**语言建模**(**language modeling**)。因此我们可以称GPT为**语言模型**(**language model**)。

生成一个单词是挺酷的（但也就那样了），但是要是生成整个句子、整篇文章呢？

1.2 生成文本(Generating Text)

1.2.1 自回归(Autoregressive)

我们可以迭代地通过模型获取下一个token的预测，从而生成整个句子。在每次迭代中，我们将预测的token再添加回输入中去：

```
# 定义一个生成文本序列的函数，接收初始输入和要生成的单词数量作为参数。
# 在Python中，下划线 _ 在for循环中作为一个变量名被使用，
# 其目的主要是表示这个变量是临时的或不重要的，即我们不打算在循环体内使用这个变量。
# 这是一种惯例，用来告诉其他阅读代码的人，这个变量在循环过程中将不会被用到
def generate(inputs, n_tokens_to_generate):
    # 自回归解码循环，根据要生成的单词数量进行迭代。
    for _ in range(n_tokens_to_generate):  # 自回归解码循环
        output = gpt(inputs)  # 模型前向传播，获取预测结果
        next_id = np.argmax(output[-1])  # 贪婪采样，选择概率最高的单词索引
        inputs.append(int(next_id))  # 将预测的单词索引添加到输入序列中，用于下一次预测
    # 函数返回新增加的单词索引，即生成的单词序列
    return inputs[len(inputs) - n_tokens_to_generate :]  # 只返回生成的单词索引

# 定义初始输入序列，代表的文本是 "not all"。
input_ids = [1, 0] # "not" "all"

```

```
# 调用generate函数，基于初始输入生成3个单词。
output_ids = generate(input_ids, 3) # output_ids = [2, 4, 6]

# 根据生成的单词索引列表，通过词汇表映射获取实际的单词。
output_tokens = [vocab[i] for i in output_ids] # 根据索引得到的单词是 "heroes" "wear" "capes"
```

这个过程是在预测未来的值（回归），并且将预测的值添加回输入中去（auto），这就是为什么你会看到GPT被描述为**自回归模型(autoregressive)**。

1.2.2 采样(Sampling)

我们可以通过对概率分布进行采样来替代贪心采样，从而为我们的生成引入一些**随机性（stochasticity）**：

```
# 定义一个输入序列，这些整数代表词汇表中对应的单词索引。
inputs = [1, 0, 2, 4] # 分别对应于 "not" "all" "heroes" "wear"

# 调用一个名为gpt的函数（假设是一个预训练的语言模型），传入输入序列。
output = gpt(inputs)

# 接下来的几行代码展示了如何使用模型的输出（最后一个单词的概率分布）来随机选择下一个单词。
# np.random.choice 函数从给定的范围内根据概率分布随机选择一个元素。
# np.arange(vocab_size) 创建一个从0到vocab_size（词汇表大小）的数组。
# p=output[-1] 参数指定了每个单词被选中的概率，这个概率来自于模型对最后一个单词的预测输出。

np.random.choice(np.arange(vocab_size), p=output[-1]) # 根据概率可能选择 "capes"
np.random.choice(np.arange(vocab_size), p=output[-1]) # 根据概率可能选择 "hats"
np.random.choice(np.arange(vocab_size), p=output[-1]) # 根据概率可能再次选择 "capes"
np.random.choice(np.arange(vocab_size), p=output[-1]) # 根据概率可能再次选择 "capes"
np.random.choice(np.arange(vocab_size), p=output[-1]) # 根据概率可能选择 "pants"

# 请注意，虽然这里给出了具体的单词作为注释，实际上np.random.choice的结果是不确定的，
# 它取决于每次调用时的概率分布。这意味着每次运行这段代码时，选择的单词可能会有所不同。
```

这样子，我们就可以基于同一个输入产生不同的输出句子啦。当我们结合更多的比如top-k，top-p和**temperature**(温度)这样的技巧的时候，（这些技巧能够能更改采样的分布），我们输出的质量也会有很大的提高。这些技巧也引入了一些超参数，通过调整这些超参，我们可以获得不同的生成表现(behaviors)。比如提高温度超参，我们的模型就会更加冒进，从而变得更有"**创造力(creative)**"。

1.3 训练(Training)

我们与训练其它神经网络一样，针对特定的**损失函数(loss function)**使用梯度下降(**gradient descent**)训练GPT。对于GPT，我们使用**语言建模任务的交叉熵损失(cross entropy loss over the language modeling task)**：

```
# 定义一个计算语言模型损失的函数。这个函数接受一个整数列表（代表文本中单词的索引）和模型参数。
def lm_loss(inputs: list[int], params) -> float:
    # 标记(token)y是输入向左移动一位的结果。这意味着，对于输入序列中的每个单词，
    # 我们使用下一个单词作为它的预测标记(token)。
    #
    # inputs = [not,     all,    heros,   wear,   capes]
    #      x = [not,     all,   heroes,   wear  ]
    #      y = [all,  heroes,     wear,   capes ]
    #
    # 例如，给定输入序列 [not, all, heroes, wear, capes]，
    # 输入x（模型的输入）将会是 [not, all, heroes, wear]，
    # 而对应的标记(token)y（我们希望模型预测的输出）将会是 [all, heroes, wear, capes]。
    #
    # 因为最后一个输入元素没有对应的下一个单词作为标记(token)，所以我们从x中排除了inputs[-1]。
    #
```

```
    # 因此，对于N个输入，我们有N-1个语言建模的样例对。
    # inputs[:-1]：这个表达式表示从列表inputs中取出从开始到倒数第二个元素的所有元素
    # inputs[1:]：这个表达式表示从列表inputs中取出从第二个元素（索引为1）到最后一个元素的所有元素
    x, y = inputs[:-1], inputs[1:]

    # 前向传播
    # 使用gpt函数和给定的参数对输入x进行处理，得到每个位置预测的下一个单词的概率分布。
    output = gpt(x, params)

    # 交叉熵损失
    # 对所有N-1个样例计算损失的平均值。
    # 这里，我们计算的是实际标记(token)y对应的概率的负对数，这是计算交叉熵损失的常见方法。
    loss = np.mean(-np.log(output[np.arange(len(y)), y]))

    return loss

# 定义一个训练函数，它接受一个文本列表（每个文本由单词列表组成）和初始模型参数。
def train(texts: list[list[str]], params) -> float:
    # 遍历文本数据集中的每个文本。
    for text in texts:
        # 使用tokenizer将文本编码为整数列表。
        inputs = tokenizer.encode(text)
        # 计算当前文本的损失。
        loss = lm_loss(inputs, params)
        # 通过反向传播计算梯度。
        gradients = compute_gradients_via_backpropagation(loss, params)
        # 使用梯度下降更新参数。
        params = gradient_descent_update_step(gradients, params)
    # 返回更新后的参数。
    return params
```

以上是一个极度简化的训练设置，但是它基本覆盖了重点。这里注意一下，我们的gpt函数签名中加入了params（为了简化，我们在上一节是把它去掉的）。在训练循环的每次迭代中：

1. 我们为给定的输入文本示例计算语言建模损失
2. 损失决定了我们的梯度，我们可以通过反向传播计算梯度
3. 我们使用梯度来更新我们的模型参数，使得我们的损失能够最小化（梯度下降）

请注意，我们在这里并未使用明确的标注数据。取而代之的是，我们可以通过原始文本自身，产生大量的输入/标签对(input/label pairs)。这就是所谓的自监督学习(**self-supervised learning**)。

自监督学习的范式，让我们能够海量扩充训练数据。我们只需要尽可能多的搞到大量的文本数据，然后将其丢入模型即可。比如，GPT-3就是基于来自互联网和书籍的**3000亿token(300 billion tokens)**进行训练的：

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.

来自GPT-3论文的Table 2.2

当然，这里你就需要一个足够大的模型有能力去从这么大量的数据中学到内容，这就是为什么GPT-3模型拥有**1750亿的参数(175 billion parameters)**，并且大概消耗了100万-1000万美元的计算费用进行训练[3]。

这个自监督训练的步骤称之为**预训练**(**pre-training**)，而我们可以重复使用预训练模型权重来训练下游任务上的特定模型，比如对文本进行分类（分类某条推文是有害的还是无害的）。预训练模型有时也被称为**基础模型(foundation models)**。

在下游任务上训练模型被称之为**微调**(**fine-tuning**)，由于模型权重已经预训练好了，已经能够理解语言了，那么我们需要做的就是针对特定的任务去微调这些权重。

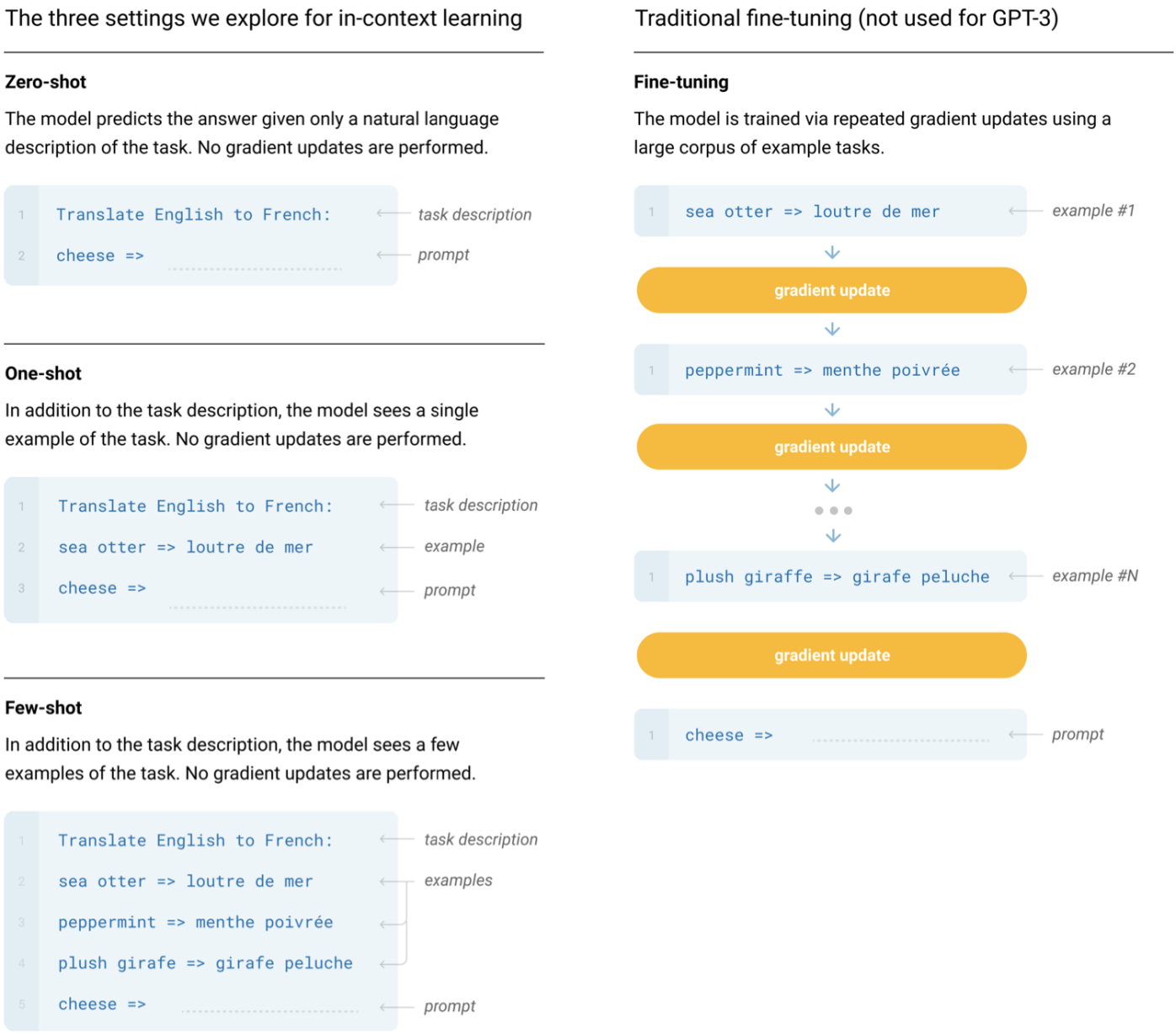
这个所谓"在通用任务上预训练 + 特定任务上微调"的策略就称之为迁移学习(transfer learning)。

1.4 提示（prompting）

本质上看，原始的GPT论文只是提供了用来迁移学习的transformer模型的预训练。文章显示，一个117M的GPT预训练模型，在针对下游任务的标注数据上微调之后，它能够在各种**NLP(natural language processing,自然语言处理)**任务上达到最优性能。

直到GPT-2和GPT-3的论文出来，我们才意识到，一个GPT模型只要在足够多的数据上训练，只要模型拥有足够多的参数，那么不需要微调，模型**本身**(**by itself**)就有能力执行各种任务。只要你对模型进行提示，运行自回归语言模型，然后你猜咋地？模型就神奇的返回给我们合适的响应了。这，就是所谓的**上下文学习(in-context learning)**， 也就是说模型仅仅根据提示的内容，就能够执行各种任务了。**上下文学习(in-context learning)**可以是**零次(zero shot)**， **一次(one shot)**， 或者是**很少次(few shot)**的：

> *译者注：我们可以简单的认为，为了执行我们的自己的任务，zero shot表示我们直接拿着大模型就能用于我们的任务了；one shot表示我们需要提供给大模型关于我们特定任务的一个列子；few shot表示我们需要提供给大模型关于我们特定任务的几个例子；*



来自GPT-3论文的图2.1

基于提示内容生成文本也被称之为**条件生成**(**conditional generation**)，因为我们的模型是基于特定的输入（*条件*）进行生成的。

当然，GPT也不仅限于自然语言处理任务(NLP)。你可以将模型用于任何你想要的条件下。比如你可以将GPT变成一个聊天机器人(即：ChatGPT)，这里的条件就是你的对话历史。你也可以进一步条件化你的聊天机器人，通过提示词进行某种描述，限定其表现为某种行为（比如你可以提示："你是个聊天机器人，请礼貌一点，请讲完整的句子，不要说有害的东西，等等"）。像这样条件化你的模型，你完全可以得到一个定制化私人助理机器人(chatbot a persona)。但是这样的方式不一定很健壮，你仍然可以对你的模型进行越狱，然后让它表现失常("jailbreak" the model and make it misbehave)。

说完了这些，现在终于要开始实际实现了。

2. 准备工作

首先将这个教程的仓库clone下来：

```
git clone https://github.com/jaymody/picoGPT
cd picoGPT

ls -l
total 32
-rwxrwxrwx 1 tony tony 1065 Apr 25  2023 LICENSE
-rwxrwxrwx 1 tony tony 2395 Apr 25  2023 README.md
-rwxrwxrwx 1 tony tony 4318 Apr 25  2023 encoder.py
-rwxrwxrwx 1 tony tony 4246 Apr 25  2023 gpt2.py
-rwxrwxrwx 1 tony tony 2330 Apr 25  2023 gpt2_pico.py
-rwxrwxrwx 1 tony tony  502 Apr 25  2023 requirements.txt
-rwxrwxrwx 1 tony tony 2745 Apr 25  2023 utils.py
```

然后安装依赖：

```
pip install -r requirements.txt
```

注意：目前代码在Python 3.9.10下测试通过。

简单介绍一下每个文件：

- encoder.py包含了OpenAI的BPE分词器的代码，这是直接从gpt-2仓库拿过来的
- utils.py：包含下载并加载GPT-2模型的权重，分词器和超参数
- gpt2.py：包含了实际GPT模型以及生成的代码，这个代码可以作为python脚本直接运行
- gpt2_pico.py：和gpt2.py一样，但是行数变少了。你问为什么？你猜

在这里，我们将从0-1复现gpt2.py，所以请先将这个文件删掉吧，我们重新建立一个新的gpt2.py文件，然后从头写起：

```
rm gpt2.py
touch gpt2.py
```

首先，将下面的代码粘贴到gpt2.py里：

```
import numpy as np
# 这行代码导入了NumPy库，并将其缩写为np。NumPy是Python的一个开源数值计算扩展库，
# 提供了大量的维度数组与矩阵运算功能，还包含了许多高级数学函数操作。
# NumPy是科学计算中常用的一个库，特别是在数据分析、机器学习等领域。
# 使用np作为NumPy的缩写是一个广泛采用的约定，这样可以在调用NumPy函数时使代码更简洁。

# 定义GPT-2模型的框架，但没有实现具体的内部逻辑。
def gpt2(inputs, wte, wpe, blocks, ln_f, n_head):
    pass # TODO: 实现这个函数

# 定义一个生成文本的函数，这个函数利用了自回归解码循环来生成指定数量的令牌。
def generate(inputs, params, n_head, n_tokens_to_generate):
    from tqdm import tqdm
    # 这行代码从tqdm库中导入了tqdm函数。tqdm是一个快速、可扩展的Python进度条库，
    # 可以在Python长循环中添加一个进度提示信息，用户只需要封装任意的迭代器 tqdm(iterator)。
    # tqdm显示当前循环的进度，预计剩余时间，以及在一行内动态更新这些信息。

    # 使用tqdm可以帮助用户了解代码的执行进度，特别是在处理需要较长时间的循环操作时，
    # 这可以提供直观的反馈信息，让用户知道程序仍在正常运行，而不是卡住或崩溃。

    # 例如，在数据处理、模型训练或文件读写等耗时操作中使用tqdm，可以改善用户的体验，
    # 通过可视化进度条明确地显示进度和预计的完成时间。

    # 使用tqdm库显示进度条，对每个生成的令牌进行迭代。
```

```
    for _ in tqdm(range(n_tokens_to_generate), "generating"):  # 自回归解码循环
        logits = gpt2(inputs, **params, n_head=n_head)  # 模型前向传播
        next_id = np.argmax(logits[-1])  # 贪婪采样，选择概率最高的下一个令牌ID
        inputs.append(int(next_id))  # 将预测的令牌ID添加到输入中，用于下一次生成

    # 返回生成的令牌ID，只返回新增加的部分
    return inputs[len(inputs) - n_tokens_to_generate :]  # 只返回生成的令牌ID

# 定义主函数，用于处理文本生成的流程。
def main(prompt: str, n_tokens_to_generate: int = 40, model_size: str = "124M", models_dir: s
    from utils import load_encoder_hparams_and_params
    # 这行代码从一个名为utils的模块中导入了load_encoder_hparams_and_params函数。
    # 这个函数的作用通常是加载编码器、超参数和参数。在上下文中，这可能指的是加载
    # 一个预训练的语言模型（如GPT-2）的相关配置和权重参数。这些参数可能包括模型的大小、
    # 用于模型训练的超参数设置，以及模型权重等。这样的设计模式使得代码结构更清晰，
    # 并且方便在不同的地方复用加载模型的逻辑。
    # 在机器学习和深度学习项目中，将模型加载和参数配置的功能模块化是一种常见的实践。

    # 从GPT-2发布的文件中加载编码器、超参数和参数
    encoder, hparams, params = load_encoder_hparams_and_params(model_size, models_dir)

    # 使用BPE分词器将输入字符串编码为令牌ID
    input_ids = encoder.encode(prompt)

    # 确保生成的令牌数量加上输入长度不会超过模型的最大序列长度
    # hparams["n_ctx"]引用的是一个超参数字典（hparams）中的n_ctx键对应的值。
    # 在神经网络模型，特别是语言模型如GPT-2中，
    # n_ctx通常指的是模型能够处理的最大上下文（或序列）长度。
    # 这个长度定义了模型一次能够接受和处理的最大令牌（如单词或字符）数量
    assert len(input_ids) + n_tokens_to_generate < hparams["n_ctx"]

    # 生成输出令牌ID
    # hparams["n_head"]：定义了在注意力机制中并行运行的独立头的数量。
    # 增加头的数量可以提升模型捕捉不同特征的能力，
    # 但同时也会增加模型的复杂度和计算要求
    output_ids = generate(input_ids, params, hparams["n_head"], n_tokens_to_generate)

    # 将令牌ID解码回字符串
    output_text = encoder.decode(output_ids)

    return output_text

# 当脚本直接运行时执行主函数
if __name__ == "__main__":
    import fire
    # 这行代码导入了Python的fire库。Fire库是一个由Google开发的开源库，用于自动生成命令行接口（CLI）。
    # 它可以将任何Python对象（无论是函数、类、模块、甚至是对象实例）转换成CLI。
    # 使用Fire，你可以很容易地将一个Python脚本变为一个可以接受命令行参数的命令行工具，
    # 而无需编写大量的解析命令行参数的代码。Fire会自动处理命令行参数的解析，并将命令行参数映射到函数的参数上。
    # 这样，开发者可以更专注于逻辑代码的开发，而不是CLI的构建和参数解析。
    # Fire非常适合快速构建和迭代开发命令行工具，使得Python脚本的命令行化变得异常简单。

    # 使用fire库使得从命令行运行脚本时能够接受参数
    fire.Fire(main)
```

我们将分为四部分进行拆解：

1. gpt2函数是我们将要实现的实际GPT代码。你会注意到函数签名中除了inputs，还有其它的参数：
   - wte, wpe, blocks, ln_f这些都是我们模型的参数
   - n_head是前向计算过程中需要的超参
2. generate函数是我们之前看到的自回归解码算法。为了简洁，我们使用贪心采样算法。tqdm是一个进度条库，它可以帮助我们随着每次生成一个token，可视化地观察解码过程。
3. main函数主要处理：
   - 1.加载分词器(encoder)， 模型权重（params）， 超参（hparams）

- 2.使用分词器将输入提示词编码为token ID
- 3.调用生成函数
- 4.将输出ID解码为字符串

4. fire.Fire(main)将我们的源文件转成一个命令行应用，然后就可以像这样运行我们的代码了：python gpt2.py "some prompt here"

我们先在notebook或者python交互界面下看看encoder, hparams, params，运行：

```
from utils import load_encoder_hparams_and_params
encoder, hparams, params = load_encoder_hparams_and_params("124M", "models")
```

上述代码将下载必要的模型及分词器文件至models/124M，并且加载encoder,hparams,params。

2.1 编码器(Encoder)

我们的encoder使用的是GPT-2中使用的BPE分词器：

```
>>> ids = encoder.encode("Not all heroes wear capes.")
>>> ids
[3673, 477, 10281, 5806, 1451, 274, 13]

>>> encoder.decode(ids)
"Not all heroes wear capes."
```

使用分词器的词汇表(存储于encoder.decoder)，我们可以看看实际的token到底长啥样：

```
>>> [encoder.decoder[i] for i in ids]
['Not', 'Ġall', 'Ġheroes', 'Ġwear', 'Ġcap', 'es', '.']
```

注意，有的时候我们的token是单词（比如：Not），有的时候虽然也是单词，但是可能会有一个空格在它前面（比如Ġall，Ġ代表一个空格），有时候是一个单词的一部分（比如：capes被分隔为Ġcap和es），还有可能它就是标点符号（比如：.）。

BPE的一个好处是它可以编码任意字符串。如果遇到了某些没有在词汇表里显示的字符串，那么BPE就会将其分割为它能够理解的子串：

```
>>> [encoder.decoder[i] for i in encoder.encode("zjqfl")]
['z', 'j', 'q', 'fl']
```

我们还可以检查一下词汇表的大小：

```
>>> len(encoder.decoder)
50257
```

词汇表以及决定字符串如何分解的**字节对组合（byte-pair merges）**，是通过*训练分词器*获得的。当我们加载分词器，就会从一些文件加载已经训练好的词汇表和**字节对组合**，这些文件在我们运行load_encoder_hparams_and_params的时候，随着模型文件被一起下载了。你可以查看models/124M/encoder.json(词汇表)和models/124M/vocab.bpe(字节对组合)。

2.2 超参数(Hyperparameters)

hparams是一个字典，这个字典包含着我们模型的超参：

```
>>> hparams
```

```
{
  "n_vocab": 50257, # 我们词汇表中的标记(token)数量
  "n_ctx": 1024, # 输入的最大可能序列长度
  "n_embd": 768, # 嵌入维度（决定了网络的“宽度”）
  "n_head": 12, # 注意力头的数量（n_embd 必须能被 n_head 整除）
  "n_layer": 12 # 网络的层数（决定了网络的“深度”）
}
```

这段代码展示了一个典型的超参数（hparams）配置字典，通常用于定义和初始化Transformer架构的神经网络模型（如GPT系列模型）：

n_vocab：定义了模型词汇表的大小，即模型能够识别的唯一标记(token)（例如单词或字符）的数量。这个参数直接影响了模型输入层和输出层的大小。

n_ctx：指定了模型可以处理的最大序列长度。这个长度限制了模型一次性可以接收和处理的输入数据的大小，对于文本生成和其他序列处理任务至关重要。

n_embd：嵌入维度，表示模型中每个标记(token)的向量表示的维度。这个参数决定了网络“宽度”，较大的嵌入维度可以让模型捕捉更复杂的特征，但也会增加模型的参数量和计算需求。

n_head：多头注意力机制中的头数。在Transformer模型中，多头注意力允许模型在不同的表示子空间中并行捕获信息，提高了模型处理复杂序列数据的能力。

n_layer：网络层数，即Transformer模型中编码器和解码器堆叠的层数。层数越多，模型的“深度”越大，理论上模型能够学习更复杂的特征和关系，但同时也增加了训练的难度和过拟合的风险。

我们将在代码的注释中使用这些符号来表示各种的大小维度等等。我们还会使用n_seq来表示输入序列的长度(即：n_seq = len(inputs))。

2.3 参数(Parameters)

params是一个嵌套的json字典，该字典具有模型训练好的权重。json的叶子节点是NumPy数组。如果我们打印params， 用他们的形状去表示数组，我们可以得到：

```
>>> import numpy as np
>>>
>>> def shape_tree(d):
>>>     # 如果d是一个NumPy数组，返回它的形状作为一个列表。
>>>     if isinstance(d, np.ndarray):
>>>         return list(d.shape)
>>>     # 如果d是一个列表，递归地对每个元素调用shape_tree，并返回形状信息的列表。
>>>     elif isinstance(d, list):
>>>         return [shape_tree(v) for v in d]
>>>     # 果d是一个字典，递归地对每个值调用shape_tree，返回一个包含形状信息的新字典。
>>>     elif isinstance(d, dict):
>>>         return {k: shape_tree(v) for k, v in d.items()}
>>>     # 如果d不是上述任何一种类型，抛出ValueError异常。
>>>     else:
>>>         ValueError("uh oh")
>>>
>>> # 印参数params的形状信息。这里假设params是一个复杂的嵌套数据结构，包含模型的参数。
>>> print(shape_tree(params))
{
    "wpe": [1024, 768],
    "wte": [50257, 768],
    "ln_f": {"b": [768], "g": [768]},
    "blocks": [
        {
```

```
            "attn": {
                "c_attn": {"b": [2304], "w": [768, 2304]},
                "c_proj": {"b": [768], "w": [768, 768]},
            },
            "ln_1": {"b": [768], "g": [768]},
            "ln_2": {"b": [768], "g": [768]},
            "mlp": {
                "c_fc": {"b": [3072], "w": [768, 3072]},
                "c_proj": {"b": [768], "w": [3072, 768]},
            },
        },
        ... # repeat for n_layers
    ]
}
```

这些是从原始的OpenAI TensorFlow checkpoint加载的：

```
>>> import tensorflow as tf
>>>
>>> # 获取指定目录下最新的检查点文件路径
>>> tf_ckpt_path = tf.train.latest_checkpoint("models/124M")
>>>
>>> # 遍历检查点文件中的所有变量
>>> for name, _ in tf.train.list_variables(tf_ckpt_path):
>>>     # 加载每个变量的值，并去除单一维度
>>>     arr = tf.train.load_variable(tf_ckpt_path, name).squeeze()
>>>     # 打印变量名和其形状
>>>     print(f"{name}: {arr.shape}")
model/h0/attn/c_attn/b: (2304,)
model/h0/attn/c_attn/w: (768, 2304)
model/h0/attn/c_proj/b: (768,)
model/h0/attn/c_proj/w: (768, 768)
model/h0/ln_1/b: (768,)
model/h0/ln_1/g: (768,)
model/h0/ln_2/b: (768,)
model/h0/ln_2/g: (768,)
model/h0/mlp/c_fc/b: (3072,)
model/h0/mlp/c_fc/w: (768, 3072)
model/h0/mlp/c_proj/b: (768,)
model/h0/mlp/c_proj/w: (3072, 768)
model/h1/attn/c_attn/b: (2304,)
model/h1/attn/c_attn/w: (768, 2304)
...
model/h9/mlp/c_proj/b: (768,)
model/h9/mlp/c_proj/w: (3072, 768)
model/ln_f/b: (768,)
model/ln_f/g: (768,)
model/wpe: (1024, 768)
model/wte: (50257, 768)
```

下述代码将上面的tensorflow变量转换为params字典。

为了对比，这里显示了params的形状，但是数字被hparams替代：

```
{
    "wpe": [n_ctx, n_embd],
    "wte": [n_vocab, n_embd],
    "ln_f": {"b": [n_embd], "g": [n_embd]},
    "blocks": [
        {
            "attn": {
                "c_attn": {"b": [3*n_embd], "w": [n_embd, 3*n_embd]},
                "c_proj": {"b": [n_embd], "w": [n_embd, n_embd]},
            },
```

```
            "ln_1": {"b": [n_embd], "g": [n_embd]},
            "ln_2": {"b": [n_embd], "g": [n_embd]},
            "mlp": {
                "c_fc": {"b": [4*n_embd], "w": [n_embd, 4*n_embd]},
                "c_proj": {"b": [n_embd], "w": [4*n_embd, n_embd]},
            },
        },
        ... # repeat for n_layers
    ]
}
```

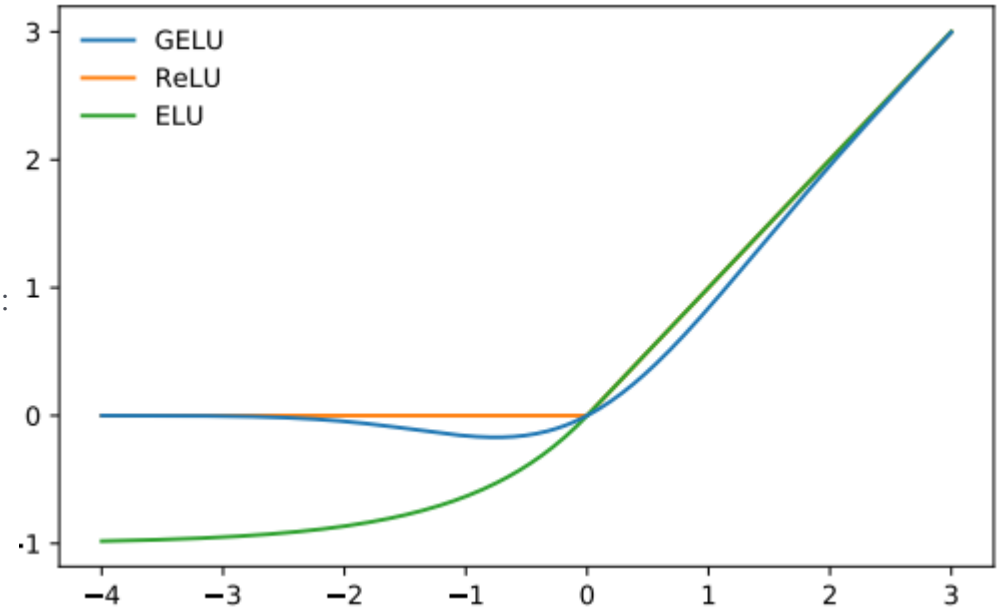
在实现GPT的过程中，你可能会需要参考这个字典来确认权重的形状。为了一致性，我们将会使代码中的变量名和字典的键值保持对齐。

3. 基础层(Basic Layers)

在进入实际GPT架构前的最后一件事，让我们来手搓几个基础的神经网络层吧，这些基础层可不只是针对GPT的，它们在各种情况下都很有用。

3.1 格鲁(GELU)

GPT-2的非线性（**激活函数，activation function**）选择是GELU（高斯误差线性单元，），这是一种类似ReLU的激活函数：



来自GELU论文的图1

下面代码定义了一个名为gelu的函数，它实现了Gaussian Error Linear Unit (GELU)激活函数。GELU是一种在深度学习模型中常用的非线性激活函数，特别是在自然语言处理（NLP）领域的Transformer模型中：

```
def gelu(x):
    # 计算GELU激活函数的值。
    # GELU激活函数提供了一种平滑的方式来权衡线性和非线性变换。
    # 公式为：0.5 * x * (1 + tanh(sqrt(2 / pi) * (x + 0.044715 * x^3)))
    # 其中：
    # - 0.5 * x * (1 + ...)：确保当x接近0时，函数接近线性变换。
    # - tanh(...)：提供非线性变换，帮助模型捕获复杂的数据特征。
    # - sqrt(2 / pi) * (x + 0.044715 * x^3)：调整x的值，增强模型的非线性能力。
    #
    # 参数：
    # x：输入数据，可以是一个数值、向量或矩阵。
```

```
    #
    # 返回值：
    # 应用GELU激活函数后的结果，与输入x具有相同的形状。
    return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))
```

和ReLU类似，GELU也对输入进行逐元素操作：

```
>>> gelu(np.array([[1, 2], [-2, 0.5]]))
array([[ 0.84119,  1.9546 ],
       [-0.0454 ,  0.34571]])
```

这意味着：

- 对于输入1，GELU激活后的输出大约为0.84119。
- 对于输入2，GELU激活后的输出大约为1.9546。
- 对于输入-2，GELU激活后的输出大约为-0.0454。
- 对于输入0.5，GELU激活后的输出大约为0.34571。

GELU激活函数能够有效地处理正数和负数输入，为负数提供平滑的非线性变换，同时保持正数的激活值相对不变，这有助于改善深度学习模型的训练和性能。

3.2 Softmax

下面是最经典的softmax:

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

下面代码定义了一个名为softmax的函数，它实现了softmax激活函数。Softmax函数是在机器学习和深度学习中广泛使用的一个函数，特别是在分类任务的输出层，用于将模型的输出转换为概率分布。

```
def softmax(x):
    # 首先，从输入x中减去x在最后一个轴（axis=-1）上的最大值，以提高数值稳定性。
    # 这个操作有助于防止在计算e^x时出现数值溢出的问题。
    # keepdims=True保持输出的维度与输入相同，以便后续操作。
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))

    # 接着，计算exp_x沿着最后一个轴的和，同样保持维度不变。
    # 这一步得到了每个元素对应的e^x的和。
    # 最后，将exp_x的每个元素除以它们所在行（或列）的e^x之和，
    # 得到的结果是一个概率分布，其中每个元素的值都在0到1之间，
    # 并且每一行（或列）的元素之和为1。
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)
```

这里我们使用了max(x)技巧来保持数值稳定性。

softmax用来将一组实数（-∞至∞之间）转换为概率（0至1之间，其求和为1）。我们将softmax作用于输入的最末轴上。

```
>>> x = softmax(np.array([[2, 100], [-5, 0]]))
>>> x
array([[0.00034, 0.99966],
       [0.26894, 0.73106]])
>>> x.sum(axis=-1)
```

```
array([1., 1.])
```

3.3 层归一化(Layer Normalization)

层归一化将数值标准化为均值为0方差为1的值：

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2}} + \beta$$

其中 μ x 是 的均值， σ^2 是 x 的方差，和 γ β 是可学习的参数。

```
def layer_norm(x, g, b, eps: float = 1e-5):
    # 计算输入x在最后一个轴（通常是特征维度）上的均值。
    # keepdims=True保持输出维度与输入相同，便于后续计算。
    mean = np.mean(x, axis=-1, keepdims=True)

    # 计算输入x在最后一个轴上的方差。
    variance = np.var(x, axis=-1, keepdims=True)

    # 使用计算出的均值和方差对x进行归一化处理，确保每个特征的均值为0，方差为1。
    # eps（一个很小的数）被加到方差中以避免除以零的情况。
    x = (x - mean) / np.sqrt(variance + eps)

    # 将归一化后的x通过缩放（乘以g）和偏移（加上b）来进行线性变换。
    # 这里的g和b是可学习的参数，分别对应于Gamma（缩放）和Beta（偏移）。
    return g * x + b
```

层归一化确保每层的输入总是在一个一致的范围里，而这将为训练过程的加速和稳定提供支持。与批归一化类似，归一化之后的输出通过两个可学习参数gamma 和 beta 进行缩放和偏移。分母中的小epsilon项用来避免计算中的分母为零错误。

我们在transformer中用层归一化来替换批归一化的原因有很多。各种不同归一化技巧的不同点在这个博客中进行了精彩的总结。

我们对输入的最末轴进行层归一化：

```
>>> # 定义一个二维数组x，模拟神经网络层的输入。
>>> x = np.array([[2, 2, 3], [-5, 0, 1]])
>>>
>>> # 对x应用层归一化。g（Gamma）设置为与x的最后一个维度相同的1的向量，b（Beta）设置为0的向量。
>>> # 这意味着归一化后的输出将不会进行缩放和偏移。
>>> x = layer_norm(x, g=np.ones(x.shape[-1]), b=np.zeros(x.shape[-1]))
>>>
>>> # 打印归一化后的x。
>>> # 可以看到，x中的每个样本都被转换成了新的值，以确保每个样本的特征均值为0，方差为1。
>>> x
>>> # 输出是归一化后的数组：
>>> # array([[-0.70709, -0.70709,  1.41418],
>>> #        [-1.397  ,  0.508  ,  0.889  ]])
>>>
>>> # 计算归一化后数据的方差，以验证它是否接近1。
>>> x.var(axis=-1)
```

```
>>> # 输出方差接近1的结果，说明归一化操作使每个样本的特征方差统一为1。
>>> # array([0.99996, 1.     ]) # 由于浮点运算的原因，结果非常接近但不完全等于1。
>>>
>>> # 计算归一化后数据的均值，以验证它是否接近0。
>>> x.mean(axis=-1)
>>> # 输出均值接近0的结果，说明归一化操作成功地将每个样本的特征均值调整为0。
>>> # array([-0., -0.])
array([[-0.70709, -0.70709,  1.41418],
       [-1.397  ,  0.508  ,  0.889  ]])
>>> x.var(axis=-1)
array([0.99996, 1.     ]) # floating point shenanigans
>>> x.mean(axis=-1)
array([-0., -0.])
```

3.4 线性（变换，Linear）

这里是标准的矩阵乘法+偏置：

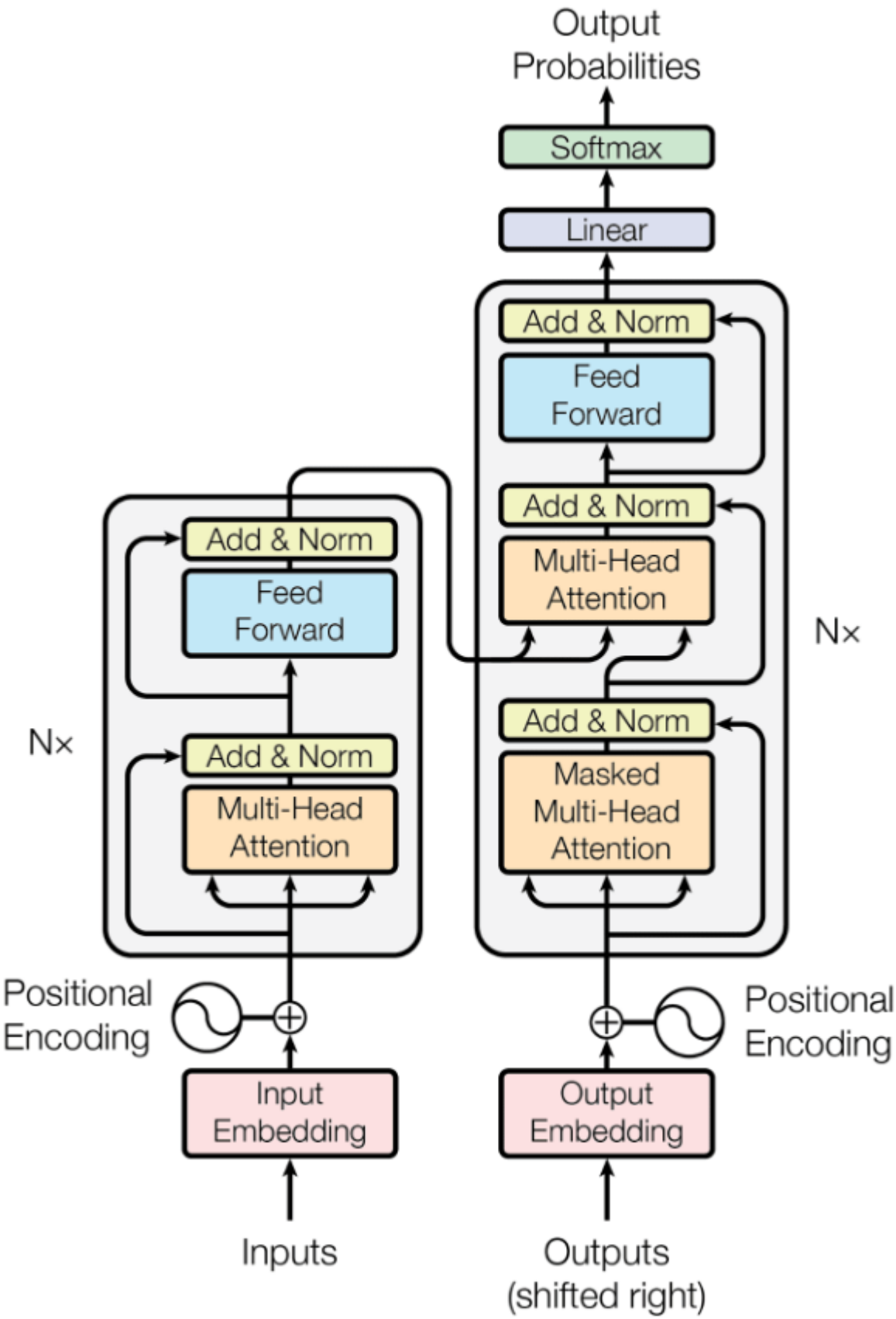
```
def linear(x, w, b):  # [m, in], [in, out], [out] -> [m, out]
    # x: 输入矩阵，形状为 [m, in]，其中 m 是批次大小，in 是输入特征的维度。
    # w: 权重矩阵，形状为 [in, out]，其中 in 是输入特征的维度，out 是输出特征的维度。
    # b: 偏置向量，形状为 [out]，其中 out 是输出特征的维度。

    # 返回值：经过线性变换后的输出矩阵，形状为 [m, out]。
    # 这个变换的计算公式为：x @ w + b，其中 @ 表示矩阵乘法，+ 表示向量加法。
    # 在这里，@ 符号用于表示矩阵乘法操作。
    return x @ w + b
```

线性层也通常被认为是**投影**操作（因为它们将一个向量空间投影到另一个向量空间）。

```
# 生成一个随机的输入矩阵x，形状为(64, 784)。
# 这里，784是输入数据的维度(dim)，64是批次大小或序列长度。
>>> x = np.random.normal(size=(64, 784)) # 输入维度(dim) = 784，批次(batch)/序列维度(sequence di

# 生成一个随机的权重矩阵w，形状为(784, 10)。
# 这里，10是输出数据的维度(dim)。
>>> w = np.random.normal(size=(784, 10)) # 输出维度(dim) = 10

# 生成一个随机的偏置向量b，长度为10。
>>> b = np.random.normal(size=(10,))

# 打印输入矩阵x的形状，确认其形状为(64, 784)。
# 这表示有64个数据点，每个数据点有784个特征。
>>> x.shape # 线性变换前的形状

# 使用linear函数对输入x进行线性变换，然后打印变换后的形状。
# 经过变换后，输出数据的形状变为(64, 10)，表示有64个数据点，
# 每个数据点经过线性层变换后输出为10个特征。
>>> linear(x, w, b).shape # 线性变换后的形状

>>> x = np.random.normal(size=(64, 784)) # input dim = 784, batch/sequence dim = 64
>>> w = np.random.normal(size=(784, 10)) # output dim = 10
>>> b = np.random.normal(size=(10,))
>>> x.shape # shape before linear projection
(64, 784)
>>> linear(x, w, b).shape # shape after linear projection
(64, 10)
```

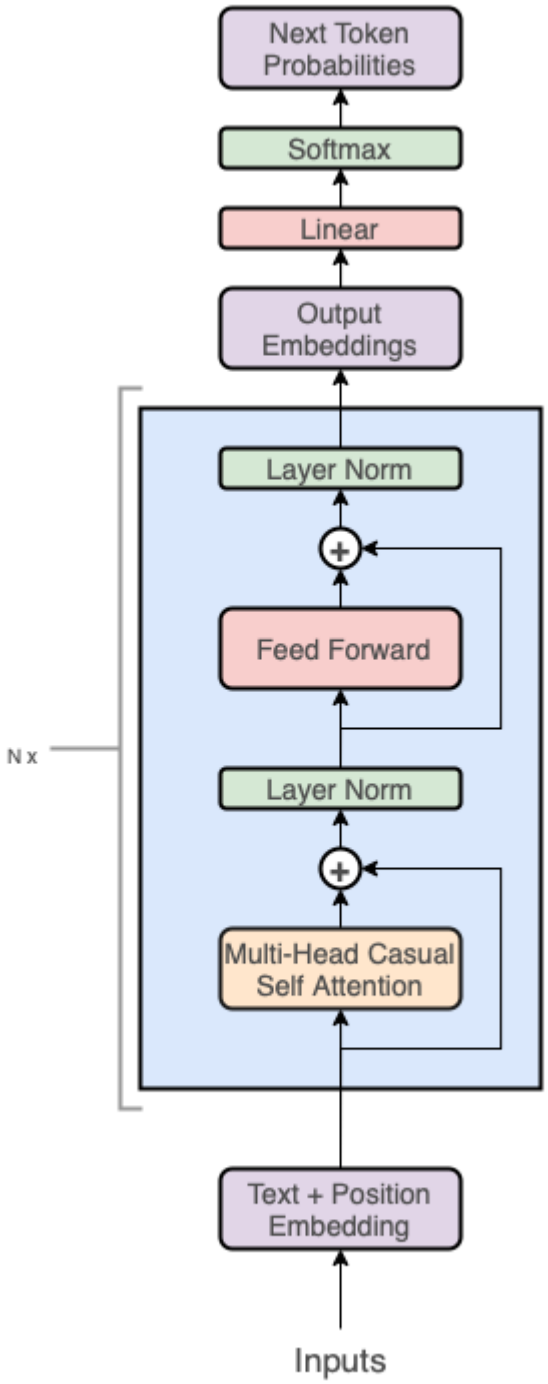
4. GPT架构

GPT的架构是基于transformer的：



来自Attention is All You Need论文的图1

但它仅仅使用了解码器层（图中的右边部分）：



GPT架构

注意，因为我们已经搞定了编码器，所以中间的"cross-attention"层也被移除了。

从宏观的角度来看，GPT架构有三个部分组成：

- 文本 + 位置**嵌入**(positional **embeddings**)
- 基于transformer的**解码器层**(**decoder stack**)
- **投影为词汇表**(**projection to vocab**)的步骤

代码层面的话，就像这样：

```python
def gpt2(inputs, wte, wpe, blocks, ln_f, n_head):  # [n_seq] -> [n_seq, n_vocab]
    # inputs: 输入序列的标记(token)索引，形状为 [n_seq]。
    # wte: 标记(token)嵌入矩阵，用于将标记(token)索引转换为嵌入向量。
    # wpe: 位置嵌入矩阵，用于给输入序列中的每个标记(token)添加位置信息。
    # blocks: Transformer模型的层（或称为block），每个block包含一组用于该层的参数。
    # ln_f: 应用于最后输出的层正则化函数的参数。
    # n_head: 注意力机制中头的数量。

    # token + positional embeddings
    # 标记(token)嵌入 + 位置嵌入
    # 使用输入索引从wte中获取标记(token)嵌入，并将其与对应的位置嵌入相加。
    # 这里，range(len(inputs))生成一个与输入序列长度相同的位置索引序列。
    x = wte[inputs] + wpe[range(len(inputs))]  # [n_seq] -> [n_seq, n_embd]

    # 通过n层Transformer block的前向传播
    for block in blocks:
        x = transformer_block(x, **block, n_head=n_head)  # [n_seq, n_embd] -> [n_seq, n_embd

    # 对输出应用层正则化
    x = layer_norm(x, **ln_f)  # [n_seq, n_embd] -> [n_seq, n_embd]

    # 投影到词汇空间
    # 使用转置的标记(token)嵌入矩阵wte.T将嵌入向量投影回词汇空间，得到最终的输出。
    return x @ wte.T  # [n_seq, n_embd] -> [n_seq, n_vocab]
```

现在我们将上面三个部分做更细致的拆解。

4.1 嵌入层(Embeddings)

4.1.1 Token 嵌入(Token Embeddings)

对于神经网络而言，token ID本身并不是一个好的表示。第一，token ID的相对大小会传递错误的信息（比如，在我们的词汇表中，如果Apple = 5，Table=10，那就意味着2 * Table = Apple？显然不对）。其二，单个的数也没有足够的**维度**喂给神经网络。

> *译者注：对于第二点补充一句，也就是说单个的数字包含的特征信息不够丰富*

为了解决这些限制，我们将利用词向量(word vectors)，即通过一个学习到的嵌入矩阵：

```python
wte[inputs] # [n_seq] -> [n_seq, n_embd]
# wte代表词嵌入矩阵，
# inputs是一个包含序列中每个标签(token)索引的数组
```

还记得吗？wte是一个[n_vocab, n_emdb]的矩阵。这就像一个查找表，矩阵中的第*ith*行对应我们的词汇表中的第*ith*个token的向量表示（学出来的）。wte[inputs]使用了数数组索引(integer array indexing)来检索我们输入中每个token所对应的向量。

就像神经网络中的其他参数，wte是可学习的。也就是说，在训练开始的时候它是随机初始化的，然后随着训练的进行，通过梯度下降不断更新。

4.1.2 位置嵌入（Positional Embeddings）

单纯的transformer架构的一个古怪地方在于它并不考虑位置。当我们随机打乱输入位置顺序的时候，输出可以保持不变（输入的顺序对输出并未产生影响）。

可是词的顺序当然是语言中重要的部分啊，因此我们需要使用某些方式将位置信息编码进我们的输入。为了这个目标，我们可以使用另一个学习到的嵌入矩阵：

```
wpe[range(len(inputs))] # [n_seq] -> [n_seq, n_embd]
# wpe代表位置嵌入矩阵，
# inputs是输入序列的标签(token)索引列表
```

wpe是一个[n_ctx, n_emdb]矩阵。矩阵的第*i*th行包含一个编码输入中第*i*th个位置信息的向量。与wte类似，这个矩阵也是通过梯度下降来学习到的。

需要注意的是，这将限制模型的最大序列长度为n_ctx[4]。也就是说必须满足len(inputs) <= n_ctx。

4.1.3 组合(Combined)

现在我们可以将token嵌入与位置嵌入联合为一个组合嵌入，这个嵌入将token信息和位置信息都编码进来了。

```
# 标签(token)嵌入 + 位置嵌入
x = wte[inputs] + wpe[range(len(inputs))]  # [n_seq] -> [n_seq, n_embd]
# x[i]代表第i个词的词嵌入加上第i个位置的位置嵌入
```

4.2 解码层(Decoder Stack)

这就是神奇发生的地方了，也是深度学习中"深度"的来源。我们将刚才的嵌入通过一连串的 n_layer transformer 解码器模块。

```
# 通过n层Transformer block进行前向传播
for block in blocks:
    # x: 当前层的输入数据，形状为[n_seq, n_embd]，其中n_seq是序列长度，n_embd是嵌入维度。
    # block: 当前Transformer层的参数，包括该层所需的所有权重和偏置等。
    # n_head: 注意力机制中的头数，这是Transformer模型的一个重要参数。
    # transformer_block: 一个函数，实现了Transformer层的计算逻辑。
    # **block: 将block字典展开为关键字参数，传递给transformer_block函数。
    # [n_seq, n_embd] -> [n_seq, n_embd]: 表示输入和输出的形状不变，依然为[n_seq, n_embd]。
    x = transformer_block(x, **block, n_head=n_head)
```

一方面，堆叠更多的层让我们可以控制到底我们的网络有多"深"。以GPT-3为例，其高达96层。另一方面，选择一个更大的n_embd值，让我们可以控制网络有多"宽"（还是以GPT-3为例，它使用的嵌入大小为12288）。

4.3 投影为词汇表(Projection to vocab)

在最后的步骤中，我们将transformer最后一个结构块的输入投影为字符表的一个概率分布：

```
# 投影到词汇表
x = layer_norm(x, **ln_f)  # [n_seq, n_embd] -> [n_seq, n_embd]
```

这里有一些需要注意的点：

1. 在进行投影操作之前，我们先将x通过**最后的层归一化层**(**final layer normalization**)。这是GPT-2架构所特有的（并没有出现在GPT原始论文和Transformer论文中）。
2. 我们**复用了嵌入矩阵(final layer normalization) wte** 进行投影操作。其它的GPT实现当然可以选择使用另外学习到的权重矩阵进行投影，但是权重矩阵共享具有以下一些优势：

- 你可以节省一些参数（虽然对于GPT-3这样的体量，这个节省基本可以忽略）
- 考虑到这个矩阵作用于**转换到词**与**来自于词**的两种转换，理论上，相对于分别使用两个矩阵来做这件事，使用同一个矩阵将学到更为丰富的表征。

3. 在最后，我们**并未使用** softmax，因此我们的输出是logits而不是0-1之间的概率。这样做的理由是：
   - softmax是单调的(monotonic)，因此对于贪心采样而言，np.argmax(logits)和np.argmax(softmax(logits))是等价的，因此使用softmax就变得多此一举。
   - softmax是不可逆的，这意味着我们总是可以通过softmax将logits变为probabilities，但不能从probabilities变为softmax，为了让灵活性最大，我们选择直接输出logits。
   - 数值稳定性的考量。比如计算交叉熵损失的时候，相对于log_softmax(logits)，log(softmax(logits))的数值稳定性就差。

投影为词汇表的过程有时候也被称之为**语言建模头（language modeling head）**。这里的"头"是什么意思呢？你的GPT一旦被预训练完毕，那么你可以通过更换其他投影操作的语言建模头，比如你可以将其更换为**分类头（classification head）**，从而在一些分类任务上微调你的模型（让其完成分类任务）。因此你的模型可以拥有多种头，感觉有点像hydra。

> *译者注：hydra是希腊神话中的九头蛇，感受一下*

好了，以上就是GPT架构的宏观介绍。那么现在我们再来看看解码器模块的细节。

4.4 解码器模块(Decoder Block)

transformer解码器模块由两个子层组成：

1. 多头因果自注意力（Multi-head causal self attention）
2. 逐位置前馈神经网络（Position-wise feed forward neural network）

```
def transformer_block(x, mlp, attn, ln_1, ln_2, n_head):  # [n_seq, n_embd] -> [n_seq, n_embd
    # x是输入的嵌入向量，形状为[n_seq, n_embd]，其中n_seq是序列长度，n_embd是嵌入维度。
    # mlp和attn分别是前馈网络和注意力机制的参数。
    # ln_1和ln_2是应用于自注意力机制和前馈网络之前的层归一化参数。
    # n_head是多头注意力机制中头的数量。
    # 多头因果自注意力机制
    # 首先，对输入x应用层归一化（layer_norm），然后传递给多头自注意力机制（mha）。
    # mha函数接受归一化后的x、注意力机制的参数(**attn)和头的数量(n_head)。
    # 然后将自注意力的输出与原始输入x相加，实现残差连接。
    # 这一步保持了x的形状不变，即[n_seq, n_embd]。
    x = x + mha(layer_norm(x, **ln_1), **attn, n_head=n_head)

    # 位置全连接前馈网络
    # 再次对x应用层归一化（layer_norm），然后传递给前馈网络（ffn）。
    # ffn函数接受归一化后的x和前馈网络的参数(**mlp)。
    # 同样地，将前馈网络的输出与输入x相加，实现另一个残差连接。
    # 这个步骤也保持了x的形状不变，即[n_seq, n_embd]。
    x = x + ffn(layer_norm(x, **ln_2), **mlp)

    # 返回经过一个Transformer层处理后的输出x。
    return x
```

每个子层都在输入上使用了层归一化，也使用了残差连接（即将子层的输入直接连接到子层的输出）。

先讲几条注意点：

1. **多头因果自注意力机制**（**Multi-head causal self attention**）便于输入之间的通信。在网络的其它地方，模型是不允许输入相互"看到"彼此的。嵌入层、逐位置前馈网络、层归一化以及投影到词汇表的操作，都是逐位置对我们的输入进行的。建模输入之间的关系完全由注意力机制来处理。
2. **逐位置前馈神经网络**（**Position-wise feed forward neural network**）只是一个常规的两层全连接神经网络。它只是为我们的模型增加一些可学习的参数，以促进学习过程。
3. 在原始的transformer论文中，层归一化被放置在输出层layer_norm(x + sublayer(x))上，而我们在这里为了匹配GPT-2，将层归一化放置在输入x + sublayer(layer_norm(x))上。这被称为**预归一化**，并且已被证明在改善transformer的性能方面非常重要。
4. **残差连接**（**Residual connections**）（由于ResNet而广为人知）这这里有几个不同的目的：
   - 1.使得深度神经网络（即层数非常多的神经网络）更容易进行优化。其思想是为梯度提供"捷径"，使得梯度更容易地回传到网络的初始的层，从而更容易进行优化。
   - 2.如果没有残差连接的话，加深模型层数会导致性能下降（可能是因为梯度很难在没有损失信息的情况下回传到整个深层网络中）。残差连接似乎可以为更深层的网络提供一些精度提升。
   - 3.可以帮助解决梯度消失/爆炸的问题。

现在我们再深入讨论一下这两个子层。

4.5 逐位置前馈网络(Position-wise Feed Forward Network)

逐位置前馈网络（Position-wise Feed Forward Network）是一个简单的 2 层的多层感知器：

```
def ffn(x, c_fc, c_proj):  # [n_seq, n_embd] -> [n_seq, n_embd]
    # 向上投影
    # 首先，使用linear函数和Gaussian Error Linear Unit (GELU)激活函数对输入x进行线性变换和非线性激活。
    # linear函数通过c_fc参数（包含权重和偏置）将每个位置的嵌入向量从n_embd维扩展到4*n_embd维。
    # 这一步的目的是增加网络的表示能力。
    a = gelu(linear(x, **c_fc))  # [n_seq, n_embd] -> [n_seq, 4*n_embd]

    # 向下投影
    # 然后，使用另一个linear函数通过c_proj参数将扩展后的嵌入向量从4*n_embd维压缩回n_embd维。
    # 这样做既可以增加模型的深度和复杂性，又能保持输入和输出的维度一致，便于堆叠多个Transformer层。
    x = linear(a, **c_proj)  # [n_seq, 4*n_embd] -> [n_seq, n_embd]

    # 返回经过前馈网络处理后的输出x。
    return x
```

这里没有什么特别的技巧，我们只是将n_embd投影到一个更高的维度4*n_embd，然后再将其投影回n_embd[5]。

回忆一下我们的params字典，我们的mlp参数如下：

```
"mlp": {
    "c_fc": {"b": [4*n_embd], "w": [n_embd, 4*n_embd]},
    "c_proj": {"b": [n_embd], "w": [4*n_embd, n_embd]},
}
```

4.6 多头因果自注意力(Multi-Head Causal Self Attention)

这一层可能是理解transformer最困难的部分。因此我们通过分别解释"多头因果自注意力"的每个词，一步步理解"多头因果自注意力"：

1. 注意力（Attention）
2. 自身（Self）
3. 因果（Causal）
4. 多头（Multi-Head）

4.6.1 注意力(Attention)

我还有另一篇关于这个话题的博客文章，那篇博客中，我从头开始推导了原始transformer论文中提出的缩放点积方程：

$$\text{attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

因此在这篇文章中，我将跳过关于注意力的解释。您也可以参考 Lilian Weng 的 Attention? Attention!和Jay Alammar的The Illustrated Transformer，这两篇也对注意力机制做了极好的解释。

我们现在只要去适配我博客文章中的注意力实现：

```
def attention(q, k, v):  # [n_q, d_k], [n_k, d_k], [n_k, d_v] -> [n_q, d_v]
    # q: 查询矩阵，形状为[n_q, d_k]，其中n_q是查询的数量，d_k是键/查询的维度。
    # k: 键矩阵，形状为[n_k, d_k]，其中n_k是键的数量，d_k是键/查询的维度。
    # v: 值矩阵，形状为[n_k, d_v]，其中n_k是值的数量，d_v是值的维度。

    # 计算查询q和键k的点积注意力得分，然后除以d_k的平方根进行缩放。
    # 这个缩放有助于控制梯度在训练初期的稳定性。
    # 使用softmax函数对注意力得分进行归一化，确保所有得分的和为1。
    # 这样可以将注意力得分解释为概率分布，表示每个键对查询的重要性。
    attention_scores = softmax(q @ k.T / np.sqrt(q.shape[-1]))

    # 将归一化的注意力得分与值v相乘，得到加权的值，然后求和。
    # 这一步骤聚合了对每个查询最重要的信息，生成输出。
    return attention_scores @ v
```

4.6.2 自身(Self)

当q，k和v来自同一来源时，我们就是在执行自注意力（即让我们的输入序列自我关注）：

```
def self_attention(x): # [n_seq, n_embd] -> [n_seq, n_embd]
    # x: 输入矩阵，形状为[n_seq, n_embd]，其中n_seq是序列长度，n_embd是嵌入维度。

    # 调用attention函数进行自注意力计算。
    # 在自注意力机制中，查询（q）、键（k）和值（v）都是同一个输入矩阵x。
    # 这意味着，对于给定的输入序列，模型将学习如何基于序列内部的其他位置来表示每个位置。
    # 这种机制使模型能够捕捉序列内的长距离依赖关系，无需依赖于传统的循环结构。
    return attention(q=x, k=x, v=x)
```

例如，如果我们的输入是“Jay went to the store, he bought 10 apples.”，我们让单词“he”关注所有其它单词，包括“Jay”，这意味着模型可以学习到“he”指的是“Jay”。

译者注：注意这里是英文的文本

我们可以通过为q、k、v和注意力输出引入投影来增强自注意力：

```python
def self_attention(x, w_k, w_q, w_v, w_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # qkv投影
    # 使用线性变换将输入x投影到查询（Q）、键（K）、值（V）空间。
    # w_k, w_q, w_v分别是对应于K、Q、V的投影矩阵。
    q = x @ w_k # 使用w_k对x进行线性变换以获得查询矩阵Q。
    k = x @ w_q # 使用w_q对x进行线性变换以获得键矩阵K。
    v = x @ w_v # 使用w_v对x进行线性变换以获得值矩阵V。

    # 执行自注意力操作
    # 调用attention函数，传入Q、K、V矩阵，计算自注意力的输出。
    x = attention(q, k, v) # 自注意力机制的输出，形状保持不变，仍为[n_seq, n_embd]。

    # 输出投影
    # 最后，使用w_proj矩阵对自注意力的输出进行线性变换，以获得最终的输出。
    x = x @ w_proj # 使用w_proj对自注意力输出进行线性变换。

    # 返回自注意力层的输出，形状为[n_seq, n_embd]，与输入形状相同。
    return x
```

这使得我们的模型为q， k， v学到一个最好的映射，以帮助注意力区分输入之间的关系。

```python
def self_attention(x, w_fc, w_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # QKV投影
    # 使用单个线性变换权重矩阵w_fc将输入x映射到一个合并了查询、键、值的空间。
    # w_fc的维度为[n_embd, 3*n_embd]，意味着输出的每个元素将被映射到一个三倍维度的空间，以便后续分割为Q、
    x = x @ w_fc # [n_seq, n_embd] @ [n_embd, 3*n_embd] -> [n_seq, 3*n_embd]

    # 分割成QKV
    # 将上一步的输出沿最后一个维度分割成三个部分，分别代表查询、键和值。
    # np.split函数实现这一分割，3表示分割成三部分，axis=-1表示沿最后一个维度进行分割。
    q, k, v = np.split(x, 3, axis=-1) # [n_seq, 3*n_embd] -> 3个[n_seq, n_embd]

    # 执行自注意力
    # 调用之前定义的attention函数，传入分割后得到的Q、K、V，进行自注意力计算。
    # 自注意力的输出维度仍为[n_seq, n_embd]，形状不变。
    x = attention(q, k, v) # [n_seq, n_embd] -> [n_seq, n_embd]

    # 输出投影
    # 使用另一个线性变换权重矩阵w_proj对自注意力的输出进行投影，以得到最终的输出。
    # w_proj的维度为[n_embd, n_embd]，保证输出的形状与输入x相同。
    x = x @ w_proj # [n_seq, n_embd] @ [n_embd, n_embd] -> [n_seq, n_embd]

    # 返回自注意力层的输出。
    return x
```

如果我们将w_q、w_k和w_v组合成一个单独的矩阵w_fc，执行投影操作，然后拆分结果，我们就可以将矩阵乘法的数量从4个减少到2个：

```python
def self_attention(x, w_fc, w_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # QKV投影
    # 使用单个线性变换权重矩阵w_fc将输入x映射到一个合并了查询、键、值的空间。
    # w_fc的维度为[n_embd, 3*n_embd]，意味着输出的每个元素将被映射到一个三倍维度的空间，以便后续分割为Q、
    x = x @ w_fc # [n_seq, n_embd] @ [n_embd, 3*n_embd] -> [n_seq, 3*n_embd]

    # 分割成QKV
    # 将上一步的输出沿最后一个维度分割成三个部分，分别代表查询、键和值。
    # np.split函数实现这一分割，3表示分割成三部分，axis=-1表示沿最后一个维度进行分割。
    q, k, v = np.split(x, 3, axis=-1) # [n_seq, 3*n_embd] -> 3个[n_seq, n_embd]

    # 执行自注意力
    # 调用之前定义的attention函数，传入分割后得到的Q、K、V，进行自注意力计算。
    # 自注意力的输出维度仍为[n_seq, n_embd]，形状不变。
    x = attention(q, k, v) # [n_seq, n_embd] -> [n_seq, n_embd]

```

```
    # 输出投影
    # 使用另一个线性变换权重矩阵w_proj对自注意力的输出进行投影，以得到最终的输出。
    # w_proj的维度为[n_embd, n_embd]，保证输出的形状与输入x相同。
    x = x @ w_proj # [n_seq, n_embd] @ [n_embd, n_embd] -> [n_seq, n_embd]

    # 返回自注意力层的输出。
    return x
```

这样会更加高效，因为现代加速器（如GPU）可以更好地利用一个大的矩阵乘法，而不是顺序执行3个独立的小矩阵乘法。

最后，我们添加偏置向量以匹配GPT-2的实现，然后使用我们的linear函数，并将参数重命名以匹配我们的params字典：

```
def self_attention(x, c_attn, c_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # QKV投影
    # 首先，通过一个线性变换（使用给定的c_attn参数）将输入x投影到一个合并了查询、键、值的空间。
    # 这一步将输入的维度从[n_seq, n_embd]扩展到[n_seq, 3*n_embd]。
    x = linear(x, **c_attn)

    # 分割成QKV
    # 然后，将上一步得到的扩展后的矩阵沿最后一个维度分割成三个相等的部分，
    # 分别代表查询（Q）、键（K）和值（V）。
    q, k, v = np.split(x, 3, axis=-1)

    # 执行自注意力
    # 使用分割得到的Q、K、V执行自注意力计算。这里的attention函数负责计算自注意力机制的输出。
    x = attention(q, k, v)

    # 输出投影
    # 最后，使用另一个线性变换（使用给定的c_proj参数）将自注意力的输出投影回原始的嵌入维度空间[n_seq, n_em
    x = linear(x, **c_proj)

    # 返回自注意力层的输出，其形状保持为[n_seq, n_embd]。
    return x
```

回忆一下，从我们的params字典中可知，attn参数类似：

```
"attn": {
    "c_attn": {"b": [3*n_embd], "w": [n_embd, 3*n_embd]},
    "c_proj": {"b": [n_embd], "w": [n_embd, n_embd]},
},
```

4.6.3 因果(Causal)

我们当前的自注意力设置存在一个问题，就是我们的输入能够“看到”未来的信息！比如，如果我们的输入是`["not", "all", "heroes", "wear", "capes"]`，在自注意力中，“wear”可以看到“capes”。这意味着“wear”的输出概率将会受到偏差，因为模型已经知道正确的答案是“capes”。这是不好的，因为我们的模型会从中学习到，输入 *i* 的正确答案可以从输入 *i*+1 中获取。

为了防止这种情况发生，我们需要修改注意力矩阵，以*隐藏*(hide)或**屏蔽**(**mask**)我们的输入，使其无法看到未来的信息。例如，假设我们的注意力矩阵如下所示：

```
        not    all    heroes wear   capes
    not 0.116  0.159  0.055  0.226  0.443
    all 0.180  0.397  0.142  0.106  0.175
heroes 0.156  0.453  0.028  0.129  0.234
   wear 0.499  0.055  0.133  0.017  0.295
  capes 0.089  0.290  0.240  0.228  0.153
```

这里每一行对应一个查询(query)，每一列对应一个键值(key)。在这个例子中，查看 “wear” 对应的行，可以看到它在最后一列以0.295的权重与 “capes” 相关。为了防止这种情况发生，我们要将这项设为0.0:

```
       not    all    heroes wear   capes
   not 0.116  0.159  0.055  0.226  0.443
   all 0.180  0.397  0.142  0.106  0.175
heroes 0.156  0.453  0.028  0.129  0.234
  wear 0.499  0.055  0.133  0.017  0.
 capes 0.089  0.290  0.240  0.228  0.153
```

通常，为了防止输入中的所有查询看到未来信息，我们将所有满足$j>i$的位置i，j都设置为0：

```
       not    all    heroes wear   capes
   not 0.116  0.     0.     0.     0.
   all 0.180  0.397  0.     0.     0.
heroes 0.156  0.453  0.028  0.     0.
  wear 0.499  0.055  0.133  0.017  0.
 capes 0.089  0.290  0.240  0.228  0.153
```

我们将这称为**掩码(masking)**。掩码方法的一个问题是我们的行不再加起来为1（因为我们在使用softmax后才将它们设为0）。为了确保我们的行仍然加起来为1，我们需要在使用softmax之前先修改注意力矩阵。

这可以通过在softmax之前将需要被掩码的条目设置为$-\infty$来实现[6]：

下面代码定义了一个名为attention的函数，它实现了带有掩码（mask）的注意力机制。掩码用于在注意力计算中排除（或减小）某些元素的影响，常见于处理变长输入序列和防止信息泄露（例如，在自回归模型中防止未来信息被提前看到）的场景

```
def attention(q, k, v, mask):  # [n_q, d_k], [n_k, d_k], [n_k, d_v], [n_q, n_k] -> [n_q, d_v]
    # q: 查询矩阵，形状为[n_q, d_k]，其中n_q是查询的数量，d_k是键/查询的维度。
    # k: 键矩阵，形状为[n_k, d_k]，其中n_k是键的数量，d_k是键/查询的维度。
    # v: 值矩阵，形状为[n_k, d_v]，其中n_k是值的数量，d_v是值的维度。
    # mask: 掩码矩阵，形状为[n_q, n_k]，用于调整或屏蔽某些键值对的注意力权重。

    # 计算查询q和键k的点积，然后除以d_k的平方根进行缩放，以控制梯度稳定性。
    # 加上掩码矩阵mask，掩码通常包含负无穷（表示完全屏蔽）或0（表示不屏蔽）。
    # softmax函数对每一行进行归一化，使得每行的元素和为1，表示概率分布。
    # 注意：掩码添加在softmax应用之前，以确保被屏蔽的项不会对最终结果产生影响。
    attention_scores = softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask)

    # 将归一化的注意力得分与值v相乘，得到加权的值，然后求和。
    # 这一步骤聚合了对每个查询最重要的信息。
    # 返回值的形状为[n_q, d_v]，每个查询对应一个输出向量，该向量是所有值的加权和。
    return attention_scores @ v
```

其中mask表示矩阵（n_seq=5）：

```
0 -1e10 -1e10 -1e10 -1e10
0   0   -1e10 -1e10 -1e10
0   0     0   -1e10 -1e10
0   0     0     0   -1e10
0   0     0     0     0
```

我们用-1e10替换-np.inf， 因为-np.inf会导致nans错误。

添加mask到我们的注意力矩阵中，而不是明确设置值为-1e10，是因为在实际操作中，任何数加上-inf还是-inf。

我们可以在NumPy中通过(1 - np.tri(n_seq)) * -1e10来计算mask矩阵。

将以上这些组合起来，我们得到：

```
def causal_self_attention(x, c_attn, c_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # QKV投影
    # 首先，使用线性变换（通过参数c_attn）将输入x映射到合并了查询Q、键K、值V的空间。
    x = linear(x, **c_attn) # [n_seq, n_embd] -> [n_seq, 3*n_embd]

    # 分割成QKV
    # 然后，将映射后的结果沿最后一个维度平均分割成三个部分，分别对应查询Q、键K和值V。
    q, k, v = np.split(x, 3, axis=-1) # [n_seq, 3*n_embd] -> 3个[n_seq, n_embd]

    # 因果掩码
    # 创建一个因果掩码，用于隐藏未来的输入，防止它们被当前位置注意到。
    # 使用numpy的tri函数创建一个下三角矩阵，然后乘以一个非常小的负数(-1e10)，
    # 以在softmax应用后将这些位置的权重接近于零。
    causal_mask = np.tri(x.shape[0], dtype=x.dtype) * -1e10  # [n_seq, n_seq]

    # 执行因果自注意力
    # 调用修改后的attention函数，传入Q、K、V和因果掩码，执行因果自注意力机制。
    x = attention(q, k, v, causal_mask) # [n_seq, n_embd] -> [n_seq, n_embd]

    # 输出投影
    # 最后，使用另一个线性变换（通过参数c_proj）将自注意力的输出映射回原始嵌入维度空间。
    x = linear(x, **c_proj) # [n_seq, n_embd] @ [n_embd, n_embd] = [n_seq, n_embd]

    # 返回因果自注意力层的输出。
    return x
```

4.6.4 多头(Multi-Head)

我们可以进一步改进我们的实现，通过进行n_head个独立的注意力计算，将我们的查询（queries），键（keys）和值（values）拆分到多个**头（heads）**里去：

```
def mha(x, c_attn, c_proj, n_head):  # [n_seq, n_embd] -> [n_seq, n_embd]
    # QKV投影
    # 首先，通过一个线性变换（使用给定的c_attn参数）将输入x映射到合并了查询Q、键K、值V的空间。
    x = linear(x, **c_attn)  # [n_seq, n_embd] -> [n_seq, 3*n_embd]

    # 分割成QKV
    # 然后，将映射后的结果沿最后一个维度分割成三个相等的部分，分别对应查询Q、键K和值V。
    qkv = np.split(x, 3, axis=-1)  # [n_seq, 3*n_embd] -> 3个[n_seq, n_embd]

    # 分割成头
    # 对Q、K、V每个进行进一步分割，按头的数量n_head分割，为每个头分配一部分维度。
    qkv_heads = list(map(lambda x: np.split(x, n_head, axis=-1), qkv))  # [3, n_seq, n_embd]

    # 因果掩码
    # 创建一个因果掩码，用于隐藏未来的输入，防止它们被当前位置注意到。
    causal_mask = (1 - np.tri(x.shape[0]), dtype=x.dtype) * -1e10  # [n_seq, n_seq]

    # 对每个头执行注意力计算
    # 遍历每个头，对Q、K、V执行带有因果掩码的注意力计算。
    out_heads = [attention(q, k, v, causal_mask) for q, k, v in zip(*qkv_heads)]  # [n_head,

    # 合并头
    # 将所有头的输出沿着最后一个维度合并起来，恢复到原始的嵌入维度。
    x = np.hstack(out_heads)  # [n_seq, n_embd]

    # 输出投影
    # 对合并后的输出执行一个线性变换（使用给定的c_proj参数），得到最终的输出。
    x = linear(x, **c_proj)  # [n_seq, n_embd] -> [n_seq, n_embd]

```

```
    # 返回多头注意力的输出。
    return x
```

这里添加了三步：

1. 拆分q， k， v到n_head个头：

```
# 分割成头
# 对查询（Q）、键（K）、值（V）的每一个进行进一步分割，以支持多头注意力机制。
# 通过在最后一个维度（axis=-1）上分割，为每个头分配一部分维度（n_embd/n_head）。
# 这里使用lambda函数和map函数来对qkv中的每个元素（Q、K、V）进行操作。
# np.split函数实现实际的分割操作，n_head指定了分割的段数。
# 结果qkv_heads是一个列表，包含三个元素（Q、K、V），每个元素的形状为[n_head, n_seq, n_embd/n_head]。
qkv_heads = list(map(lambda x: np.split(x, n_head, axis=-1), qkv))  # [3, n_seq, n_embd] -> [
```

2. 为每个头计算注意力：

```
# 对每个头执行注意力计算
# 使用列表推导式遍历qkv_heads中的每个头（Q、K、V），对每个头执行注意力函数。
# qkv_heads被解压（unpack）为Q、K、V三个组件，每个组件都是多头的集合，
# 其中包含了每个头对应的查询（Q）、键（K）和值（V）。
# 调用attention函数计算每个头的输出，这里假设attention函数已经定义，
# 能够接受单个头的Q、K、V并进行注意力计算。
# 注意：这里的调用省略了掩码参数，实际使用时可能需要考虑是否传递掩码参数以实现特定的注意力机制，如因果掩码。
# 结果out_heads是一个列表，其中包含了每个头经过注意力计算后的输出，
# 每个输出的形状为[n_seq, n_embd/n_head]，与输入形状相同，但最后一个维度被分割为头数。
out_heads = [attention(q, k, v) for q, k, v in zip(*qkv_heads)]  # [3, n_head, n_seq, n_embd/
```

3. 合并每个头的输出：

```
# 合并头
# 将多头注意力的输出从多个头的表示合并回单个表示。
# 使用np.concatenate而不是np.hstack，因为我们需要沿着最后一个维度（嵌入维度）合并头的输出，
# 而np.hstack默认操作于第一个轴（即在这个上下文中不正确）。
# 正确的合并操作应该指定axis=-1来确保沿着嵌入维度合并。
# 这里out_heads是一个形状为[n_head, n_seq, n_embd/n_head]的列表，我们想要的结果是一个
# 单一的数组，形状为[n_seq, n_embd]，即将所有头的输出合并到嵌入维度。
x = np.concatenate(out_heads, axis=-1)  # [n_head, n_seq, n_embd/n_head] -> [n_seq, n_embd]
```

注意，这样可以将每个注意力计算的维度从n_embd减少到n_embd/n_head。这是一个权衡。对于缩减了的维度，我们的模型在通过注意力建模关系时获得了额外的子空间。例如，也许一个注意力头负责将代词与代词所指的人联系起来；也许另一个注意力头负责通过句号将句子分组；另一个则可能只是识别哪些单词是实体，哪些不是。虽然这可能也只是另一个神经网络黑盒而已。

我们编写的代码按顺序循环执行每个头的注意力计算（每次一个），当然这并不是很高效。在实践中，你会希望并行处理这些计算。当然在本文中考虑到简洁性，我们将保持这种顺序执行。

好啦，有了以上这些，我们终于完成了GPT的实现！现在要做的就是将它们组合起来并运行代码。

5. 将所有代码组合起来

将所有代码组合起来，我们就得到了gpt2.py，总共的代码只有120行（如果你移除注释、空格之类的，那就只有60行）。

```
import numpy as np
from tqdm import tqdm

# 定义GELU激活函数。
```

```
def gelu(x):
    # GELU激活函数的实现。
    return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))

# 定义softmax函数。
def softmax(x):
    # 对输入x应用softmax，用于计算概率分布。
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

# 定义层归一化函数。
def layer_norm(x, g, b, eps: float = 1e-5):
    # 对输入x进行层归一化。
    mean = np.mean(x, axis=-1, keepdims=True)
    variance = np.var(x, axis=-1, keepdims=True)
    x = (x - mean) / np.sqrt(variance + eps)
    return g * x + b

# 定义线性变换函数。
def linear(x, w, b):
    # 对输入x应用线性变换。
    return x @ w + b

# 定义前馈网络。
def ffn(x, c_fc, c_proj):
    # 对输入x应用前馈网络。
    a = gelu(linear(x, **c_fc))
    x = linear(a, **c_proj)
    return x

# 定义注意力机制。
def attention(q, k, v, mask):
    # 对输入执行带掩码的注意力计算。
    return softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask) @ v

# 定义多头注意力机制。
def mha(x, c_attn, c_proj, n_head):
    # 对输入x应用多头注意力机制。
    x = linear(x, **c_attn)
    qkv = np.split(x, 3, axis=-1)
    qkv_heads = list(map(lambda x: np.split(x, n_head, axis=-1), qkv))
    causal_mask = (1 - np.tri(x.shape[0], dtype=x.dtype)) * -1e10
    out_heads = [attention(q, k, v, causal_mask) for q, k, v in zip(*qkv_heads)]
    x = np.hstack(out_heads)
    x = linear(x, **c_proj)
    return x

# 定义Transformer块。
def transformer_block(x, mlp, attn, ln_1, ln_2, n_head):
    # 对输入x应用Transformer块。
    x = x + mha(layer_norm(x, **ln_1), **attn, n_head=n_head)
    x = x + ffn(layer_norm(x, **ln_2), **mlp)
    return x

# 定义简化版的GPT-2模型。
def gpt2(inputs, wte, wpe, blocks, ln_f, n_head):
    # 对输入序列应用GPT-2模型。
    x = wte[inputs] + wpe[range(len(inputs))]
    for block in blocks:
        x = transformer_block(x, **block, n_head=n_head)
    x = layer_norm(x, **ln_f)
    return x @ wte.T

# 定义文本生成函数。
def generate(inputs, params, n_head, n_tokens_to_generate):
```

```
    # 生成文本的自回归循环。
    for _ in tqdm(range(n_tokens_to_generate), "generating"):
        logits = gpt2(inputs, **params, n_head=n_head)
        next_id = np.argmax(logits[-1])
        inputs.append(int(next_id))
    return inputs[len(inputs) - n_tokens_to_generate:]

# 定义主函数。
def main(prompt: str, n_tokens_to_generate: int = 40, model_size: str = "124M", models_dir: s
    # 主函数，加载模型参数，编码输入，生成文本，解码输出。
    from utils import load_encoder_hparams_and_params
    encoder, hparams, params = load_encoder_hparams_and_params(model_size, models_dir)
    input_ids = encoder.encode(prompt)
    assert len(input_ids) + n_tokens_to_generate < hparams["n_ctx"]
    output_ids = generate(input_ids, params, hparams["n_head"], n_tokens_to_generate)
    output_text = encoder.decode(output_ids)
    return output_text

# 入口点。
if __name__ == "__main__":
    import fire
    fire.Fire(main)
```

我们可以通过以下代码测试：

```
python gpt2.py \
    "Alan Turing theorized that computers would one day become" \
    --n_tokens_to_generate 8
```

其输出是：

```
the most powerful machines on the planet.
```

成功运行！！！

我们可以使用以下Dockerfile验证我们的实现与OpenAI的官方GPT-2仓库产生相同的结果（注意：这在M1 Macbooks上无法运行，这里涉及到TensorFlow的支持问题。还有一个警告是：这会下载所有4个GPT-2模型，而这意味着大量GB规模的文件需要被下载）：

```
docker build -t "openai-gpt-2" "https://gist.githubusercontent.com/jaymody/9054ca64eeea7fad1b5
docker run -dt "openai-gpt-2" --name "openai-gpt-2-app"
docker exec -it "openai-gpt-2-app" /bin/bash -c 'python3 src/interactive_conditional_samples.p
# paste "Alan Turing theorized that computers would one day become" when prompted
```

这里应该会给出完全相同的结果：

```
he most powerful machines on the planet.
```

6. 下一步呢？

这个实现虽然不错，但还缺少很多额外的功能：

6.1 GPU/TPU 支持

将NumPy替换为JAX：

```
import jax.numpy as np
```

搞定！现在你可以在GPU甚至是TPU上使用这个代码了！前提是你正确地安装了JAX。

译者注：JAX是个好东西：）

6.2 反向传播(Backpropagation)

如果我们用JAX替换掉了NumPy：

```
import jax.numpy as np
```

那么计算梯度也变得很简单：

```
import jax.numpy as np

# 定义语言模型的损失函数
def lm_loss(params, inputs, n_head) -> float:
    # 将输入序列分割成输入(x)和目标输出(y)。x由除了最后一个元素外的所有元素组成，
    # y由除了第一个元素外的所有元素组成，实现了向前一位的效果，用于预测下一个单词。
    x, y = inputs[:-1], inputs[1:]

    # 调用gpt2模型函数，传入输入x，模型参数params，和头的数量n_head。
    # 假定gpt2函数返回对每个可能输出的概率分布。
    output = gpt2(x, **params, n_head=n_head)

    # 计算损失，这里使用了负对数似然作为损失函数。这要求output是一个概率分布，
    # 其中output的每一行对应输入序列中每个位置的下一个单词的概率分布。
    # 注意：这里简单地使用output[y]可能并不正确，除非output确实以这种方式组织。
    # 通常需要更精确地索引output来匹配y的形状。
    loss = np.mean(-np.log(output[y]))

    # 返回计算出的平均损失。
    return loss

# 使用jax库来计算lm_loss函数关于params的梯度。
# 这里假设params是gpt2模型的参数，inputs是输入序列，n_head是多头注意力机制中头的数量。
grads = jax.grad(lm_loss)(params, inputs, n_head)
```

6.3 批处理(Batching)

还是那句话，如果我们用JAX替换掉NumPy[7]：

```
import jax.numpy as np
```

那么让gpt2函数批量化就变得很简单：

```
# 使用jax.vmap将gpt2函数批量化。in_axes参数定义了每个输入参数的批处理轴。
# 对于batched_inputs，批处理轴是0（表示每个批次的数据沿着第一个维度排列）。
# 对于其他参数（模型参数和配置），由于它们在批处理中不变，因此设置为None，表示不进行批处理。
gpt2_batched = jax.vmap(gpt2, in_axes=[0, None, None, None, None, None])

# 调用批量化后的gpt2函数处理批次化的输入。
# batched_inputs是一个形状为[batch, seq_len]的数组，其中batch是批次大小，seq_len是序列长度。
# gpt2_batched函数的输出将是一个形状为[batch, seq_len, vocab]的数组，
# 其中vocab是词汇表的大小。这表示对于批次中的每个输入序列，模型都返回了一个序列长度相同的，
# 每个位置都有一个对应词汇表大小的预测概率分布的输出。
gpt2_batched(batched_inputs) # [batch, seq_len] -> [batch, seq_len, vocab]
```

6.4 推断优化(Inference Optimization)

我们的实现相当低效。除了支持GPU和批处理之外，最快且最有效的优化可能是实现一个键值缓存。此外，我们顺序地实现了注意力头计算，而实际上我们应该使用并行计算[8]。

其实还有很多很多的推理优化可以做。我建议从Lillian Weng的Large Transformer Model Inference Optimization和Kipply的Transformer Inference Arithmetic开始学习。

6.5 训练(Training)

训练 GPT 对于神经网络来说是非常标准的行为（针对损失函数进行梯度下降）。当然，在训练 GPT 时你还需要使用一堆常规的技巧（使用 Adam 优化器，找到最佳的学习率，通过dropout和/或权重衰减进行正则化，使用学习率规划器，使用正确的权重初始化，进行分批处理等等）。

而训练一个好的GPT模型的真正秘诀在于**能够扩展数据和模型**（**scale the data and the model**），这也是真正的挑战所在。

为了扩展数据量，您需要拥有大规模、高质量、多样化的文本语料库。

- **大规模**意味着拥有数十亿的token（数百万GB的数据）。例如可以查看The Pile，这是一个用于大型语言模型的开源预训练数据集。
- **高质量**意味着需要过滤掉重复的示例、未格式化的文本、不连贯的文本、垃圾文本等等。
- **多样性**意味着序列长度变化大，涵盖了许多不同的主题，来自不同的来源，具有不同的观点等等。当然，如果数据中存在任何偏见，它将反映在模型中，因此您需要谨慎处理。

将模型扩展到数十亿个参数需要超级大量的工程（和金钱lol）。训练框架会变得非常冗长和复杂。关于这个主题的一个良好起点是Lillian Weng的How to Train Really Large Models on Many GPUs。当然，关于这个话题还有NVIDIA的Megatron Framework，Cohere的训练框架，Google的PALM，开源的mesh-transformer-jax（用于训练EleutherAI的开源模型），以及很多、很多、很多。

6.6 评估(Evaluation)

哦对了，那么要怎么评估大语言模型呢？老实说，这是一个非常困难的问题。HELM 是一个相当全面且不错的起点，但你应该始终对基准测试和评估指标保持怀疑的态度。

6.7 架构改进(Architecture Improvements)

我推荐看一下Phil Wang的X-Transformers。它包含了Transformer架构的最新最赞的研究。这篇论文也是一个不错的概述(见表格1)。Facebook最近的LLaMA论文也可能是标准架构改进的一个很好的参考（截至2023年2月）。

> *译者注：学transformer的小伙伴，看完x-transformers绝对功力大涨*

6.8 停止生成(Stopping Generation)

我们当前的实现需要事先指定要生成的确切token数量。这不是一个很好的方法，因为我们生成的文本可能会太长、太短或在句子中间截断。

为了解决这个问题，我们可以引入一个特殊的**句子结束（EOS）token**。在预训练期间，我们在输入的末尾附加EOS token（比如，tokens = ["not", "all", "heroes", "wear", "capes", ".", "<|EOS|>"]）。在生成过程中，我们只需要在遇到EOS token时停止（或者达到最大序列长度）：

```
def generate(inputs, eos_id, max_seq_len):
    # inputs: 初始输入序列的令牌索引。
    # eos_id: 结束符（End Of Sentence）的令牌索引，用于标识生成过程何时停止。
    # max_seq_len: 生成文本的最大长度限制。

    # 保存初始输入的长度，用于最后返回生成的部分。
    prompt_len = len(inputs)

    # 当最后一个令牌不是结束符且当前序列长度小于最大长度时，继续生成。
    while inputs[-1] != eos_id and len(inputs) < max_seq_len:
        # 使用gpt模型对当前输入序列进行预测，得到下一个令牌的概率分布。
        output = gpt(inputs)

        # 从最后一个位置的概率分布中选择概率最高的令牌作为下一个令牌。
        next_id = np.argmax(output[-1])

        # 将选中的令牌添加到输入序列中，为下一轮生成做准备。
        inputs.append(int(next_id))

    # 生成结束，返回除了初始输入外的生成部分。
    return inputs[prompt_len:]
```

GPT-2 没有使用 EOS token进行预训练，因此我们无法在我们的代码中使用这种方法，但是现在大多数 LLMs 都已经使用 EOS token了。

6.9 无条件生成(Unconditional Generation)

使用我们的模型生成文本需要对其提供提示**条件**（**condition**）。但是我们也可以让模型执行**无条件生成**（**unconditional generation**），即模型在没有任何输入提示的情况下生成文本。

这是通过在预训练期间在输入开头加上一个特殊的**句子开头（BOS）token**来实现的（例如 tokens = ["<|BOS|>", "not", "all", "heroes", "wear", "capes", "."]）。要进行无条件文本生成的话，我们就输入一个仅包含BOS token的列表：

```
def generate_unconditioned(bos_id, n_tokens_to_generate):
    # bos_id: 开始符号的令牌索引，表示生成文本的起始点。
    # n_tokens_to_generate: 需要生成的令牌数量。

    # 初始化输入序列，仅包含开始符号。
    inputs = [bos_id]

    # 循环指定的次数，每次生成一个令牌。
    for _ in range(n_tokens_to_generate):
        # 使用当前的输入序列调用gpt模型，获取模型的输出。
        output = gpt(inputs)

        # 从模型输出的最后一个位置（即最新生成的令牌的概率分布）中选择概率最高的令牌作为下一个令牌。
        next_id = np.argmax(output[-1])

        # 将选中的令牌添加到输入序列中，供下一次迭代使用。
        inputs.append(int(next_id))

    # 返回生成的序列，不包括初始的开始符号。
    return inputs[1:]
```

这个函数的工作原理如下：

1. 从一个仅包含开始符号（bos_id）的序列开始。
2. 在每次循环中，调用模型gpt预测下一个令牌，直到生成了指定数量的令牌。
3. 选择每次预测中概率最高的令牌作为下一个令牌，并将其加入到输入序列中。
4. 循环结束后，返回除开始符号外的所有生成的令牌序列。

这种生成方式是无条件的（unconditioned），因为它不依赖于任何先前的上下文信息，仅从一个固定的开始符号出发。这种方法通常用于测试或展示语言模型的能力，或在需要生成全新内容时使用。与条件生成相比（条件生成基于特定的上下文或提示生成文本），无条件生成可能会产生更随机和不可预测的输出。

GPT-2的预训练是带有BOS token的（不过它有一个令人困惑的名字<|endoftext|>），因此在我们的实现中要运行无条件生成的话，只需要简单地将这行代码更改为：

```
# 假定`encoder`是一个已经初始化好的编码器实例，能够将文本编码成令牌索引列表。
# `prompt`是一个字符串，表示用户输入或其他形式的提示文本。

input_ids = encoder.encode(prompt) if prompt else [encoder.encoder[""]]
```

然后运行；

```
python gpt2.py ""
```

然后即可生成：

```
The first time I saw the new version of the game, I was so excited. I was so excited to see th
```

因为我们使用的是贪心采样，所以输出结果不是很好（重复的内容较多），且每次运行代码的输出结果都是确定的。为了获得更高质量的、不确定性更大的生成结果，我们需要直接从概率分布中进行采样（最好在使用top-p之类的方法后进行采样）。

无条件生成不是特别有用，但它是演示GPT能力的一种有趣方式。

6.10 微调(Fine-tuning)

我们在训练部分简要介绍了微调。回想一下，微调是指我们复用预训练的权重，对模型在某些下游任务上进行训练。我们称这个过程为迁移学习。

理论上，我们可以使用零样本或少样本提示来让模型完成我们的任务，但是如果您可以访问一个标注的数据集，对GPT进行微调将会产生更好的结果（这些结果可以在获得更多数据和更高质量的数据时进行扩展）。

好的，以下是关于微调的一些相关主题：

6.10.1 分类微调(Classification Fine-tuning)

在分类微调中，我们会给模型一些文本，并要求它预测它属于哪个类别。以IMDB数据集为例，它包含着电影评论，将电影评为好或坏：

```
--- Example 1 ---
Text: I wouldn't rent this one even on dollar rental night.
Label: Bad
--- Example 2 ---
Text: I don't know why I like this movie so well, but I never get tired of watching it.
Label: Good
--- Example 3 ---
```

```
...
```

为了微调我们的模型，我们需要用分类头替换语言建模头，将其应用于最后一个token的输出：

```
def gpt2(inputs, wte, wpe, blocks, ln_f, cls_head, n_head):
    # wte: 令牌嵌入矩阵，用于将输入令牌编码成向量。
    # wpe: 位置嵌入矩阵，用于给输入向量添加位置信息。
    # blocks: Transformer模型的块（层）列表。
    # ln_f: 最后一层的层归一化参数。
    # cls_head: 分类头的权重矩阵，用于将模型的输出映射到类别上。
    # n_head: 多头注意力中头的数量。

    # 将输入令牌转换为嵌入向量，并加上位置嵌入。
    x = wte[inputs] + wpe[range(len(inputs))]

    # 通过所有Transformer块（层）进行前向传播。
    for block in blocks:
        x = transformer_block(x, **block, n_head=n_head)

    # 对最后的输出应用层归一化。
    x = layer_norm(x, **ln_f)

    # 使用分类头对最后一个输出向量进行投影，以得到类别预测。
    # 注意这里只取序列最后一个位置的输出进行分类，这常见于处理序列分类任务时，
    # 假设序列的最后一个输出包含了整个序列的信息。
    # [n_embd] @ [n_embd, n_classes] -> [n_classes]
    return x[-1] @ cls_head
```

这里我们只使用最后一个token的输出x[-1]，因为我们只需要为整个输入产生一个单一的概率分布，而不是像语言模型一样产生n_seq个分布。我们特别选择最后一个token（而不是第一个token或所有token的组合），因为最后一个token是唯一允许关注整个序列的token，因此它具有关于整个输入文本的信息。

同往常一样，我们根据交叉熵损失进行优化：

```
def singe_example_loss_fn(inputs: list[int], label: int, params) -> float:
    # inputs: 输入序列的令牌索引列表。
    # label: 正确类别的索引。
    # params: GPT模型的参数。

    # 调用GPT模型，获取对每个类别的logits预测。
    logits = gpt(inputs, **params)

    # 使用softmax函数将logits转换为概率分布。
    probs = softmax(logits)

    # 计算交叉熵损失：取负对数概率作为损失。
    # 这里直接索引label对应的概率，假定label是正确的类别索引。
    loss = -np.log(probs[label])

    # 返回计算的损失值。
    return loss
```

我们还可以执行**多标签分类**（即一个样本可以属于多个类别，而不仅仅是一个类别），这可以通过使用sigmoid替代softmax并针对每个类别采用二分交叉熵损失（参见这个stackexchange问题）。

6.10.2 生成式微调(Generative Fine-tuning)

有些任务无法被简单地认为是分类，如摘要的任务。我们可以通过对输入和标签拼接进行语言建模，从而实现这类任务的微调。例如，下面就是一个摘要训练样本的示例：

```
--- Article ---
This is an article I would like to summarize.
--- Summary ---
This is the summary.
```

我们就像预训练时那样训练这个模型（根据语言建模的损失进行优化）。

在预测时，我们将直到"--- Summary ---"的输入喂给模型，然后执行自回归语言建模以生成摘要。

定界符"--- Article ---"和"--- Summary ---"的选择是任意的。如何选择文本格式由您决定，只要在训练和推断中保持一致即可。

请注意，其实我们也可以将分类任务表述为生成任务（以IMDB为例）：

```
--- Text ---
I wouldn't rent this one even on dollar rental night.
--- Label ---
Bad
```

然而，这种方法的表现很可能会比直接进行分类微调要差（损失函数包括对整个序列进行语言建模，而不仅仅是对最终预测的输出进行建模，因此与预测有关的损失将被稀释）。

6.10.3 指令微调(Instruction Fine-tuning)

目前大多数最先进的大型语言模型在预训练后还需要经过一个额外的**指令微调**(**instruction fine-tuning**)步骤。在这个步骤中，模型在成千上万个由**人工标注**(**human labeled**)的指令提示+补全对上进行微调（生成式）。指令微调也可以称为**监督式微调**(**supervised fine-tuning**)，因为数据是人工标记的（即**有监督的**,**supervised**）。

那指令微调的好处是什么呢？虽然在预测维基百科文章中的下一个词时，模型在续写句子方面表现得很好，但它并不擅长遵循说明、进行对话或对文件进行摘要（这些是我们希望GPT能够做到的事情）。在人类标记的指令 + 完成对中微调它们是教导模型如何变得更有用，并使它们更容易交互的一种方法。我们将其称为**AI对齐**(**AI alignment**)，因为我们需要模型以我们想要的方式做事和表现。对齐是一个活跃的研究领域，它不仅仅只包括遵循说明（还涉及偏见、安全、意图等）的问题。

那么这些指令数据到底是什么样子的呢？Google的FLAN模型是在多个学术的自然语言处理数据集（这些数据集已经被人工标注）上进行训练的：

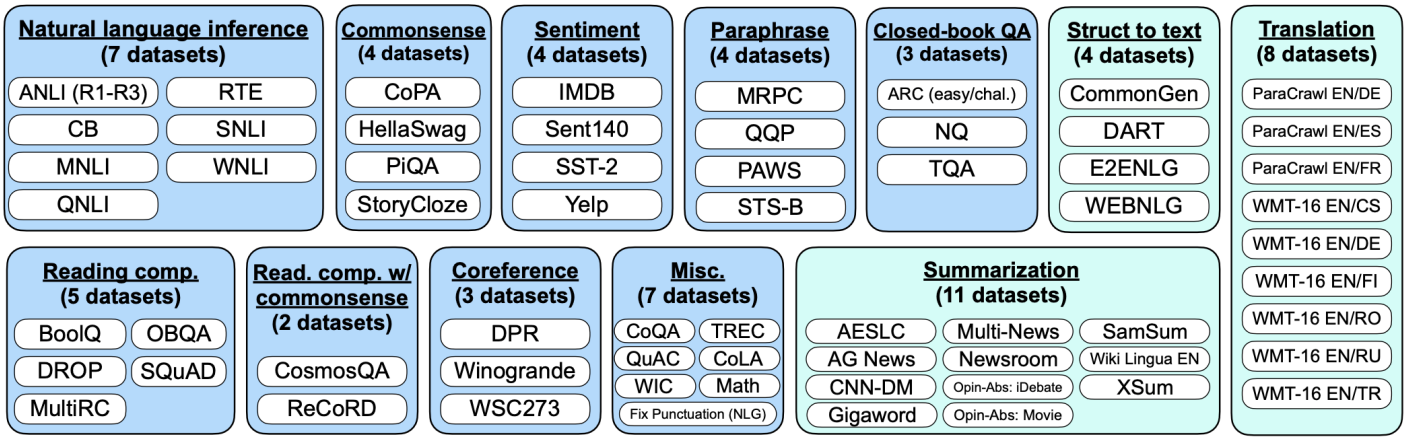


Figure 3: Datasets and task clusters used in this paper (NLU tasks in blue; NLG tasks in teal).

来自FLAN论文的图3

OpenAI的InstructGPT则使用了从其API中收集的提示进行训练。然后他们雇佣工人为这些提示编写补全。下面是这些数据的详细信息：

Table 1: Distribution of use case categories from our API prompt dataset.

Use-case	(%)
Generation	45.6%
Open QA	12.4%
Brainstorming	11.2%
Chat	8.4%
Rewrite	6.6%
Summarization	4.2%
Classification	3.5%
Other	3.5%
Closed QA	2.6%
Extract	1.9%

Table 2: Illustrative prompts from our API prompt dataset. These are fictional examples inspired by real usage—see more examples in Appendix A.2.1.

Use-case	Prompt
Brainstorming	List five ideas for how to regain enthusiasm for my career
Generation	Write a short story where a bear goes to the beach, makes friends with a seal, and then returns home.
Rewrite	This is the summary of a Broadway play: """ {summary} """ This is the outline of the commercial for that play: """

来自InstructGPT论文的表1与表2

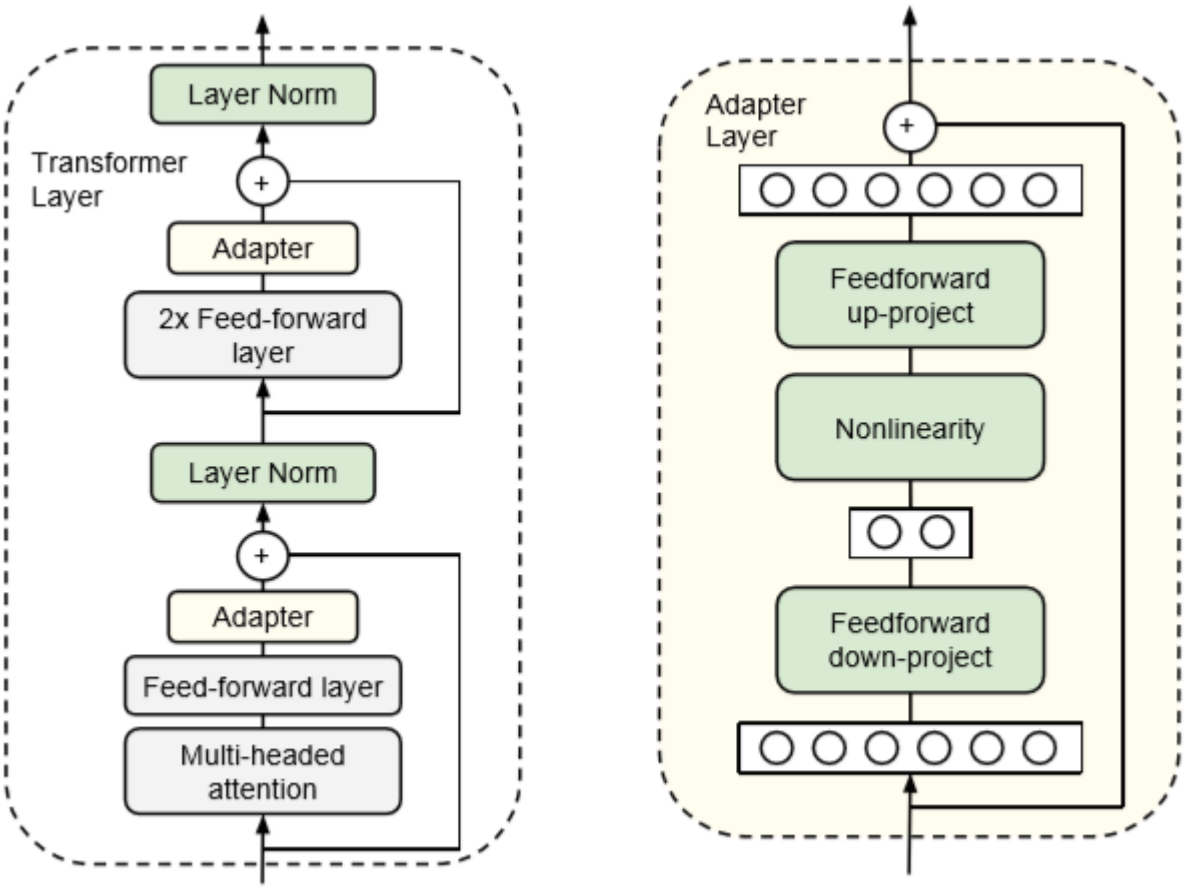
6.10.4 参数高效微调(Parameter Efficient Fine-tuning)

当我们在上面的部分讨论微调时，我们是在更新模型的所有参数。虽然这可以获得最佳性能，但成本非常高，无论是在计算方面（需要经过整个模型进行反向传播），还是在存储方面（对于每个微调的模型，您需要存储完一份全新的参数副本）。

最简单的解决方法是**只更新模型头部**(**only update the head**)并**冻结**(**freeze**)（即使其不可训练）模型的其它部分。虽然这样做可以加速训练并大大减少新参数的数量，但其表现并不好，因为某种意义上我们损失了深度学习中的**深度**(*deep*)。相反，我们可以**选择性地冻结**(**selectively freeze**)特定层（例如冻结除了最后四层外的所有层，或每隔一层进行冻结，或冻结除多头注意力参数外的所有参数），那么这将有助于恢复深度。这种方法的性能要好得多，但我们也变得不那么参数高效(parameter efficient)，同时也失去了一些训练速度的优势。

除此之外，我们还可以利用**参数高效微调(Parameter Efficient Fine-tuning)**方法。这仍然是一个活跃的研究领域，有许多不同的方法可供选择、选择、选择、选择、选择、选择、选择。

举个例子，我们可以看看Adapters论文。在这种方法中，我们在transformer模块的FFN和MHA层后添加了一个额外的"adapter"层。这里的adapter层只是一个简单的两层全连接神经网络，其中输入和输出维度是n_embd，而隐藏维度小于n_embd：



来自Adapters论文的图2

适配器方法中，隐藏层的大小是一个我们可以设置的超参数，这使我们能够在参数和性能之间进行权衡。该论文表明，对于BERT模型，使用这种方法可以将训练参数数量降低到2%，而与完全微调相比仅有少量的性能下降(<1%)。

---

1. 大规模训练、收集海量数据、提高模型速度、性能评估以及对齐模型使其为人类服务，数百名工程师/研究人员的将这视为终身事业，这些人的工作造就了今时今日的大型语言模型，绝不仅仅是因为模型的架构。GPT架构恰好是第一个具有良好的可扩展性、可在GPU上高度并行化且善于序列建模的神经网络架构。真正的秘诀来自于扩展的数据和模型规模（一如既往的重要），GPT只是让我们可以这样做而已[9]。可能Transformer的成功是刚好中了硬件彩票而已，还有一些其他的架构可能正在等待着取代Transformer。_
2. 对于某些应用程序，分词器不需要一个decoder方法。例如，如果你想要对电影评论进行分类，判断评论是说这部电影好还是不好，你只需要能够对文本进行encode，并在模型上进行前向传递，没有必要进行decode。但是对于生成文本，decode是必需的。_
3. 虽然有InstructGPT和Chinchilla的论文，我们已经意识到实际上并不需要训练那么大的模型。在经过最优训练和指令微调后，参数为13亿的GPT模型可以胜过参数为1750亿的GPT-3。_
4. 原始的transformer论文使用了预计算的位置嵌入（positional embedding），他们发现这种方法的表现和学习的位置嵌入一样好，但其有一个明显的优势，即你可以输入任意长的序列（不受最大序列长度的限制）。然而在实践中，您的模型只能表现得和它所训练的序列长度一样好。您不能只在长度为1024的序列上训练GPT，然后指望它在长度为16k的序列上表现良好。然而最近出现了一些成功的相对位置嵌入（relative positional embeddings）方法，如Alibi和RoPE。_
5. 不同的GPT模型可能选择不同的隐藏层宽度，而不必是4*n_embd，这是GPT模型的通行做法。此外，我们在推动Transformer的成功方面给予多头注意力层很多*注意*（双关了哦~），但在GPT-3的规模下，80%的模型参数包含在前馈层中。这是值得思考的事情。_
6. 如果你还没有被说服，可以看一下softmax方程，自己琢磨一下这是正确的（甚至可以拿出笔和纸进行计算）。_
7. 表白JAX_
8. 使用JAX的话，这就可以简单写为heads = jax.vmap(attention, in_axes=(0, 0, 0, None))(q, k, v, causal_mask)_

9. 实际上，我可能会争辩一下注意力模型在处理序列时的方式与循环/卷积层相比，具有内在的优越性，但现在我们已经陷入了一个注脚中的注脚了，那还是先打住吧。_