今天我们将继续我们的‘从零到英雄’系列，特别是今天我们将重新实现GPT-2模型的124百万参数版本。OpenAI在2019年发布了GPT-2，并发布了这篇博客文章。除此之外，他们还发布了这篇论文，并且还将代码发布在了GitHub上。就是OpenAI GPT-2。

现在，当我们谈论重新实现GPT-2时，我们必须小心，特别是在这段视频中，我们将重新实现124百万参数的模型。要意识到的是，这些发布总是有一个迷你系列。所以，GPT-2的迷你系列由不同大小的模型组成，通常最大的模型被称为GPT-2。但基本上，我们这么做的原因是，你可以将模型大小放在图表的x轴上，将你感兴趣的许多下游指标（如翻译、摘要、问答等）放在y轴上。你可以绘制出这些扩展规律。基本上，随着模型大小的增加，你在下游指标上会做得越来越好。

因此，特别是对于GPT-2，如果我们在论文中向下滚动，可以看到GPT-2迷你系列中的四个模型，起始为124百万，一直到1558百万。现在，我所说的数字和这个表格中的数字不一致，因为这个表格是错误的。如果你实际上去查看GPT-2的GitHub仓库，他们会说，嗯，他们在加总参数时出了个错。但基本上这是124百万参数的模型，等等。所以，124百万参数的模型有12个层，在变换器中有768个通道——768维。

我将假设你们对这些术语有一些了解，因为我在我之前的视频《让我们构建GPT-2》中已经讲解过所有这些内容。所以，我在这个播放列表中的上一期视频中讲解了这些内容。

现在，如果我们正确地做了所有事情，并且一切进展顺利，到视频的最后，我们将看到像这样的结果，我们正在观察验证损失，这基本上是衡量我们在某些模型在训练中没有见过的验证数据上，预测下一个token的准确性的指标。从一开始做得不好，因为我们是从零开始初始化，到训练结束时做得相当好。希望我们能超越GPT-2的124M模型。

现在，之前他们在做这个时，这已经是五年前的事情了，所以当时这可能是一个相当复杂的优化，而且当时的GPU和计算资源要小得多。今天，你可以在大约一个小时，甚至可能更短的时间内重新实现这个模型，并且如果你想在云端做的话，成本大约是十美元。你可以租用一种计算机，每个人都可以租用，如果你为这台计算机支付十美元，等待大约一个小时或更短，你就能实现一个和OpenAI发布的模型一样好的模型。

还有一点要提到的是，与许多其他模型不同，OpenAI确实发布了GPT-2的权重，所以这些权重都可以在这个仓库中找到。然而，GPT-2论文并不是对训练的所有细节都做得很好。除了GPT-2的论文，我们还将参考GPT-3的论文，GPT-3在许多超参数和优化设置方面更加具体，且它与GPT-2版本的模型在架构上没有太大偏差，所以我们将在重新实现GPT-2 124M时，参考GPT-2和GPT-3的论文。

所以，开始吧。首先，我想从最后开始，或者说从目标开始。换句话说，让我们加载OpenAI发布的GPT-2 124M模型，可能还试试它，生成一些token。问题是，当你进入GPT-

2的代码库，进入源代码，点击model.py时，你会发现其实它使用的是TensorFlow。所以，最初的GPT-2代码是用TensorFlow编写的，而TensorFlow现在不再那么常用了。

我们想使用PyTorch，因为它更友好、更容易，而且我个人更喜欢它。问题是，初始代码是用TensorFlow编写的，而我们想使用PyTorch。所以为了达到目标，我们将使用Hugging Face的Transformers代码，我个人更喜欢这个。进入Transformers的源代码，找到Transformer模型中的`GPT-2 modeling GPT-2.py`文件，你会看到他们在这个文件中实现了GPT-2。嗯，它是适度可读的，但并不是完全可读。它做了所有的工作，将所有这些权重从TensorFlow转换为PyTorch友好的格式。所以，它更容易加载和操作。

特别是，我们可以在这里查看GPT-2模型，并且可以使用Hugging Face Transformers加载它。所以，来看一下它是这样的：从Transformers导入GPT-2语言模型头部，然后从预训练的GPT-2加载。嗯，现在有一个尴尬的地方是，当你做GPT-2作为模型时，我们加载的实际上是124百万参数的模型。如果你想要真正的GPT-2，即15亿参数的模型，那么你其实应该选择XL。所以，这就是我们的目标——124M模型。现在，当我们实际获取这个模型时，我们会初始化PyTorch的神经网络模块，如这个类中所定义的那样。从中，我想提取出状态字典，它只是原始的张量。

所以，我们就得到了这个文件的张量。顺便提一下，这是一个Jupyter Notebook，但它是在VS Code中运行的。嗯，我喜欢在一个单一的界面中工作，所以我喜欢使用VS Code。这是VS Code中的Jupyter Notebook扩展。所以当我们获取状态字典时，它实际上只是一个字典。我们可以打印出键和值，值就是张量，我们就来看看这些形状。所以这些是GPT-2模型中的不同参数及其形状。比如说，token嵌入的W权重的大小是50257×768。这是因为GPT-2的词汇表中有50257个token。嗯，顺便说一下，这些token正是我们在我之前的视频“我的标记化系列”中讨论过的那些token。在之前的视频中，我详细讲解了标记化。GPT-2的标记化器正好有这么多token。对于每个token，我们有一个768维的嵌入。这就是代表该token的分布式表示。所以，每个token都是一个小的字符串片段，然后这768个数字就是表示该token的向量。

这是token的表格，然后这里我们有位置的查找表。因为GPT-2有一个最大序列长度为1024，所以每个token最多可以关注到过去1024个位置。GPT-2中的每个位置都有一个固定的768维向量，这是通过优化学习到的。嗯，这就是位置嵌入和token嵌入。嗯，然后这里的其他部分就是这个transformer的其他权重、偏置和一切。比如，当你仅仅拿出位置嵌入并将其展开，取出前20个元素时，你可以看到这些只是参数。这些是权重浮动，我们可以取出来并绘制它们。所以，这些就是位置嵌入，我们得到的是这样的结果。你可以看到，

这些是有结构的，它有结构是因为在这个可视化中，每一行代表的是一个不同的位置，是从0到1024范围内的一个固定绝对位置。每一行在这里代表的是该位置的表示。

它有结构是因为这些位置嵌入最终学到了这些正弦和余弦。嗯，这些正弦和余弦表示了每一个位置，每一行在这里代表的是那个位置，并且被Transformer处理来恢复所有相对位置， sort of理解哪个token在什么位置。嗯，它根据位置而不是内容来关注这些位置。所以，当我们实际查看这些中的一个列时，我随机选了三列，你会看到，举个例子，我们聚焦在每一个通道上，我们正在查看每个通道在从1到1223的位置变化。

实际上，我们可以看到这些通道中的一些基本上对位置谱的不同部分有更多或更少的反应。所以这个绿色通道实际上更喜欢在200到800之间的所有内容激活，但小于这个范围时激活较少，并且在接近零的位置有一个明显的下降。那么，谁知道这些嵌入是如何工作的，以及为什么它们是这样的呢?

你可以从中看出，举个例子，由于它们有点不规则，并且有些噪声，你可以知道这个模型并没有完全训练好。这个模型训练得越多，你会更期望它能将这些噪声平滑掉。所以，这告诉你这是一个稍微欠训练的模型。嗯，但实际上，这些曲线并不需要平滑。它们本来应该是完全的随机噪声。事实上，在优化的开始阶段，它完全是随机噪声，因为这个位置嵌入表是完全随机初始化的。所以，一开始你会看到不规则性。事实上，最终能够得到平滑的东西已经很令人印象深刻了。嗯，这只是优化的结果。

因为从原则上讲，你甚至不应该能够从中得到任何一张有意义的图。但实际上，我们得到的东西看起来有点噪声，但大部分看起来是正弦波状的。就像在原始的Transformer中一样。在原始Transformer论文《Attention is All You Need》中，位置嵌入实际上是初始化并固定的，利用不同频率的余弦和正弦值作为位置编码，它是固定的。但在GPT-2中，这些只是参数，它们从零开始训练，就像其他任何参数一样。

这个方法似乎也能很好地工作，所以它们在优化过程中恢复了这些类似正弦波的特征。我们也可以查看其他矩阵。例如，这里我查看了Transformer的第一层，并查看了它的一个权重，仅仅是300×300的第一个块，你可以看到一些结构。但同样，谁知道这些是什么呢?如果你对机械解释学感兴趣，你可能会很享受试图弄清楚这些结构是什么，它们的含义是什么。但我们在这个视频中不会做这个。不过，我们确实看到了有趣的结构，这还是挺酷的。我们最关心的是，我们已经加载了这个由OpenAI发布的模型权重，现在使用Hugging Face的Transformers，我们不仅能获取所有原始权重，还可以获得他们所说的Pipeline并从中采样。比如，前缀是“你好，我是一个语言模型”，然后我们采样了30个token，并得到5个序列。

我运行了这个，得到了这样的结果："你好，语言"模型。但实际上，我做的是生成一个可读的文档。还有其他语言，但这些是...你可以根据需要阅读它们。基本上，这些是从GPT-2 124M模型中得到的同一前缀的五个不同完成。现在，如果我去这里，我从这里拿了一个例子，遗憾的是，尽管我们固定了种子，我们得到的生成结果和他们的结果不同。所以大概是代码发生了变化。但在这一阶段，我们看到的一个重要点是，我们得到了连贯的文本。所以我们成功加载了模型。我们可以查看它的所有参数，键告诉我们这些参数来自模型的哪个部分。接下来我们想要做的是，编写我们自己的GPT-2类，这样我们就能完全理解其中发生的事情。我们不想使用像`modeling GPT-2.py`这样的东西，因为它太复杂了。

我们想从零开始编写它，所以我们将在这里并行实现GPT模型。作为我们的第一个任务，让我们将GPT-2 124M模型加载到我们将从零开发的类中。这将让我们有信心能够加载OpenAI模型，并且有一个精确匹配124M模型的权重设置。但随后，当然，我们将从零初始化模型，并尝试用我们获得的文档训练它。我们将尝试超越这个模型，所以我们将获得不同的权重，一切看起来会不同，希望能更好。但我们有足够的信心，因为我们能够加载OpenAI模型，所以我们在同一个模型家族和类别中，只需要从零重新发现一个合适的权重设置。

所以，现在让我们来编写GPT-2模型，加载权重，并确保我们也能生成看起来连贯的文本。好的？现在，让我们回到《Attention is All You Need》这篇论文，它是所有工作的起点，让我们滚动到模型架构部分。原始的Transformer。现在，记住GPT-2与原始Transformer略有修改。特别是，我们没有编码器；GPT-2是一个仅包含解码器的Transformer，我们称之为解码器-only Transformer。所以，这里的整个编码器是缺失的。除此之外，原本使用编码器的交叉注意力部分也缺失了。因此，我们删除了这一部分。其他几乎保持不变，但有一些差异我们需要在这里讨论一下。所以，主要有两个差异。当我们查看GPT-2页面下的2.3模型时，我们注意到，首先，层归一化的位置发生了重新排列，它们互换了位置；其次，在最后的自注意力块中添加了额外的层归一化。因此，基本上，所有的层归一化，现在不再在MLP或注意力之后，而是放在它们之前，并且在最终分类器之前新增了一个层归一化。

现在，让我们在我们的GPT神经网络模块中实现一些初步的模块，特别是我们将尝试匹配Hugging Face Transformers中使用的这种架构，因为这将使我们更容易从状态字典中加载这些权重。所以我们需要一种能够反映这种架构的方式。以下是我想到的：基本上，我们看到的主要容器是名为Transformer的，它包含所有的模块，所以我通过一个NN模块字典来反映这一点，这基本上是一个允许你通过键索引子模块的模块，就像一个字典一样。在其中，我们有令牌嵌入的权重WT，它是一个N维嵌入，和位置嵌入的权重，这也是一个N维嵌入。如果你还记得，嵌入其实就是一个精美的封装模块，围绕一个单一的数字数组，

或者一个数字块，像这样。它是一个单一的张量，而嵌入只是一个对张量的增强封装，允许你通过索引来访问它的元素。

除此之外，我们还看到这里有一个h，它的索引使用的是数字，而不是使用字符串，所以有h.0，1，2，等等，一直到h.11。因为Transformer中有12层。为了反映这一点，我还创建了一个H，它可能代表隐藏层。与模块字典不同，这是一个模型列表，因此我们可以通过整数来索引它，正如我们看到的那样：0，1，2，等等。这个模块列表包含n层块，而这些块还需要在模块中定义。除此之外，按照GPT-2论文的要求，我们需要一个额外的最终层归一化，我们将把它放在这里，然后我们有最终的分类器，也就是语言模型头，它将从768（这个GPT中嵌入维度的数量）投影到词汇表大小，词汇表大小为50,257。GPT-2在这个最终投影中不使用偏置。因此，这是骨架，你可以看到它反映了这一点。WTE是令牌嵌入。这里它叫做输出嵌入，但实际上它就是令牌嵌入。PE是位置编码。正如我们之前看到的，这两部分信息将会加在一起，然后进入Transformer。H是所有的块（灰色部分），而LNF是GPT-2模型中新增的这个层。LM头就是这里的这个线性部分，所以这就是GPT-2的骨架。现在我们需要实现块。好的，那么现在让我们继续到块本身。所以我们想要定义这个块。我喜欢这样写出这些块。这些是一些初始化，然后这是该块实际计算的前向传递。请注意，这里有一个与Transformer的变化，这是GPT-2论文中提到的。因此，在这里，层归一化是应用于注意力或前馈后进行的。

另外，注意到这些归一化是在残差流中进行的。你可以看到，前馈是如何应用的，这个箭头穿过并通过归一化。因此，这意味着你的残差路径内含归一化，这不是很理想或可取的。实际上，你希望残差流是干净的，一直到输入，即令牌。这样非常理想和好，因为从上面来的梯度，如果你记得，来自微分添加的部分，能够均匀地分配梯度到它的两个分支。所以加法是梯度的一个分支，这意味着来自上面的梯度直接通过残差路径传输到输入，令牌，保持不变。但除此之外，梯度也通过块流动，块会随着时间推移提供它们自己的贡献，推动优化过程的变化。但基本上，从优化的角度来看，干净的残差路径是非常理想的

这是预归一化版本，您会看到RX首先通过层归一化，然后通过注意力机制，再返回去进行第二次层归一化。多层感知机（有时也称为前馈网络，或简称FFN）接着再次进入残差流。另一个值得注意的地方是，回顾一下注意力机制，它是一种通信操作。所有的tokens——在一个序列中有1024个tokens——在这里进行交流，交换信息。因此，注意力是一种聚合函数，是加权和函数，属于归约操作，而MLP在每个token上独立发生。没有信息在tokens之间进行收集或交换。因此，注意力是归约，MLP是映射，您可以将Transformer视为一种重复应用的映射操作。可以这样理解，在注意力中它们进行通信，在MLP中它们独立思考各自收集到的信息。每一个这样的块都会迭代地细化残差流中的表示。所以，这是我们修改过的块。好了，现在我们继续到MLP部分。我实现的MLP块如

下：它相对直接，基本上我们有两个线性投影，它们夹在一个G非线性函数中间。所以，nn.G大致等于10h。现在，当我们查阅Pyro文档时，它是n.G，具有这种格式，并且有两个版本：G的原始版本，我们稍后会详细讨论，和Galo的近似版本，我们可以通过10请求。所以，您可以看到，简单预览一下，G基本上像ReLU，只是它在零附近没有完全平坦的尾部；但除此之外，它看起来非常像是一个稍微平滑的ReLU。它来自这篇论文：GAN误差线性单元，您可以逐步阅读这篇论文，其中有一些数学计算推导，解释了特定的公式。它涉及到随机径向提升器和对自适应dropout的修改期望。如果您感兴趣，可以阅读所有这些内容。这里有一点历史，解释为什么存在G的近似版本，这与当时在TensorFlow中计算精确的G所需要的地球函数非常缓慢有关。因此，他们最终开发了这个近似版本，而这个近似版本最终被BERT和GPT-2等模型采用。但今天没有特别的理由使用近似版本，您最好使用精确版本。嗯，因为我预计现在没有太大区别，这只是一个历史性的小怪癖。不过，我们正在尝试精确地重现GPT-2，而GPT-2使用了10h近似版本，所以我们更倾向于坚持使用它。另一个直观上使用G而非ReLU的理由是，之前在视频中我们提到过ReLU死神经元问题：在ReLU的尾部，如果它在零点完全平坦，任何落在这里的激活将得到完全为零的梯度。没有变化，没有适应，没有网络的进一步发展。如果这些激活结束在这个平坦区域，那么G始终贡献局部梯度，因此总会有变化——总会有适应，从经验上看，这种平滑化的做法在实际中效果更好，正如这篇论文中所示，以及它被BERT、GPT-2等论文采纳的原因。所以，基于这个理由，我们在GPT-2复现中采用了这个非线性激活。现在，在更现代的网络中，比如LLaMA 3等，这个非线性激活也进一步变更为Swiglo和其他类似变体，但对于GPT-2来说，它使用了这个近似版G。好的，最后我们来看一下注意力操作。让我贴上我的注意力代码。我知道这内容很多，所以我会稍微快一点，但不会太快，因为我们在之前的视频中已经讲过。我会建议您回顾之前的视频。那么，这是注意力操作。记得在之前的视频中，这不仅仅是注意力；这也是多头注意力。对吧？在之前的视频中，我们有一个多头注意力模块，这个实现让我们明白这些头实际上并不复杂。每个注意力块内都并行运行多个头，它们的输出仅仅是被串联起来，成为多头注意力的输出。它非常简单，并且使得头的实现变得相当直观。这里的做法是，与其使用两个单独的模块，实际上是许多模块被串联在一起，所有内容都放进一个单一的自注意力模块中。相反，我在PyTorch中非常小心地进行了很多转置、拆分张量的优化，以使其更加高效，但从算法上讲，和我们在之前的仓库中看到的实现没有什么不同。

简单提醒一下，我不想在这里花太多时间，但我们有这些tokens按顺序排列，数量是1020个。在注意力机制的这个阶段，每个token会发出三个向量：查询（query）、键（key）和值（value）。首先，在这里发生的事情是，查询和键必须进行相乘，以得到注意力值，也就是它们彼此之间的有趣程度。它们是相互乘积作用的，所以我们在这里计算QKV。我们将其拆分，然后进行一些操作，正如我在这里提到的。这个操作的方式是，我们基本上将头数（H）变成一个批处理维度，就像B一样，在后续的操作中，PyTorch将B和NH视

为批处理，并在批处理和头部之间并行应用所有操作。接下来应用的操作是，首先，查询和键的交互给我们提供了注意力。

这是自动回归遮罩，确保tokens只能关注它们之前的tokens，而永远不能关注未来的tokens。Softmax在这里对注意力进行归一化，使其总和始终为1。然后，回想一下前面的视频，进行注意力矩阵乘法与值（values）是做一个加权和的方式，计算我们在每个token中找到的有趣token的值。最后的转置和视图操作只是将所有这些重新组装起来。

实际上，它执行的是连接操作，如果你愿意，可以慢慢理解这个过程。但它在数学上与我们之前的实现是等价的，只是它在PyTorch中更高效，所以我选择了这种实现方式。

此外，我在命名变量时非常小心。例如，“cattin”和“seaten”是相同的，因此我们的键应该基本遵循Hugging Face Transformers代码的命名规范，这样可以非常方便地将所有权重移植到这种命名约定中，因为我们所有的变量都采用了相同的名称。但此时，我们已经完成了GPT-2的实现，这使得我们不再需要使用Hugging Face的这个文件，该文件相当长，大约有2000行代码。相反，我们只需要不到100行代码，这就是完整的GPT-2实现。因此，在此阶段，我们应该能够直接加载所有的权重，设置它们，然后进行生成。

现在我们来看一下这是什么样子的。好的，我还修改了GPT配置，使得这里的H参数与GPT-2 124M模型一致。最大序列长度，我称之为块大小（block size），是124。token的数量是50,257，这如果你看过我的tokenizer视频，你会知道这是50,000次合并操作，BP合并，256字节token，BP树的叶子，还有一个特殊的文本结束标记（end-of-text token），它用来区分不同的文档并可以开始生成。模型有12层，每层有12个注意力头。Transformer的维度是768。现在我们可以将Hugging Face的参数加载到我们的代码中，并使用这些参数初始化GPT类。接下来，我将复制并粘贴一大段代码。我不会太快或太慢地讲解这些代码，因为老实说，这并不那么有趣，也不那么激动人心。我们只是在加载权重，所以过程有些枯燥。但正如我之前提到的，这个关于GPT-2的迷你系列有四个模型。这是我们在右侧看到的一些Jupyter代码——嗯，就是这部分代码。它覆盖了GPT-2模型的超参数。我们正在创建配置对象，并创建我们自己的模型。接下来发生的是，我们正在为我们的模型和Hugging Face模型创建状态字典（state dict）。我们遍历Hugging Face模型的键，并将这些张量复制过来。在这个过程中，我们会忽略一些缓冲区，它们不是参数，而是缓冲区。例如，自动回归遮罩中使用的注意力偏置。因此，我们会忽略其中的一些遮罩。

就是这样。然后，还有一个额外的烦恼，那就是这些权重来自TensorFlow库，我不确定为什么这样做。这有点烦人，因为一些权重的转置方式和PyTorch所需的不一样。所以，我手动硬编码了需要转置的权重，然后如果有需要，我们会进行转置操作。然后，我们返回这个模型。`from_pretrained`是一个构造函数或类方法，它返回GPT对象，只要我们提供模型类型，在我们的案例中是GPT-2，即我们感兴趣的最小模型。这就是代码，使用方法就是这样。

“我们可以在 VS Code 里打开终端，然后用 Python 训练 GPT-2。希望一切顺利，好吗？我们没有崩溃。所以，我们可以把权重、偏置和其他所有内容加载到我们的人工神经网络模块中。

但现在让我们也更有信心确认这个模型确实在工作，我们来尝试从这个模型中生成内容。好吧，在我们实际生成内容之前，我们得先能够执行前向传播。我们实际上还没写那部分代码。所以这是前向函数。前向的输入将是我们的索引、令牌，即令牌的索引，它们的形状总是 B x T。因此，我们有批次维度 B，然后是最多 T 的时间维度。T 不能超过块大小；块大小是最大序列长度。因此，B 乘 T 的索引排列有点像二维布局。记住，基本上每一行的大小最大为块大小，这些是一个序列中的 T 个令牌。然后，我们有 B 个独立的序列堆叠在一起形成一个批次，这样就提高了效率。

现在，我们正在前向传播位置嵌入和令牌嵌入。这段代码应该和上节课的非常相似。因此，我们基本上使用一个范围（类似于 PyTorch 版本的 range），从 Z 遍历到 T，创建这些位置索引。然后我们确保它们与 idx 在同一设备上，因为我们不会仅在 CPU 上训练，这样效率太低了。我们想要在 GPU 上训练，这一点稍后会用到。接着我们有位置嵌入和令牌嵌入，并对这两个进行加法操作。

现在，请注意，位置嵌入对于每一行输入都是相同的，因此加法操作中会有广播，意味着我们需要在这里创建一个额外的维度。然后，这两个加起来是因为相同的位置信息在我们堆叠的每一行中都会应用。接下来，我们前向传播 Transformer 块，最后是层归一化和语言模型头。前向传播的输出是 logits，如果输入是 B x T 的索引，那么对于每一个 B x T，我们将计算下一个令牌在序列中出现的概率。也就是说，哪个令牌是 B_t+1，位于这个令牌的右侧？B 的大小是可能令牌的数量。因此，这是我们将获得的张量，这些 logits 只需要经过 softmax 就能转化为概率。所以，这就是网络的前向传播，现在我们可以加载模型，马上就能从模型中生成内容了。

好了，现在我们要尝试在左侧设置一个与 Hugging Face 在右侧相匹配的模型。我们从管道中采样，并且采样了五次，最多 30 个令牌，前缀为 “Hello, I'm a language model”，这些是我们生成的结果。因此，我们将尝试在左侧复制这个过程。序列的转数是五，最大长度是 30。所以，首先我们做的当然是初始化我们的模型，然后将其置于评估模式。

这是一个很好的实践，在你不打算训练模型，只是使用它时，将模型置于评估模式。我现在并不确定这是否真的有效，原因是：我们上面使用的模型不包含在训练或评估时行为不同的模块或层。例如，Dropout、批量归一化等层会有这种行为，但我们这里用的所有层在训练和评估时应该是相同的。

因此，评估模式可能没有做什么操作，但我不确定是否确实如此。也许 PyTorch 内部会根据评估模式做一些聪明的处理。

接下来的步骤是将整个模型移到 CUDA 上。因此，我们将所有的张量移到 GPU 上。我通过 SSH 连接到云端的服务器上，服务器上有很多 GPU。现在，我将整个模型和它的所有成员、所有张量等都移到 GPU 上。所有东西都将被发送到一个完全独立的计算机，它位于 GPU 上，并且 GPU 与 CPU 连接，可以进行通信。它基本上是一个完全独立的计算机，拥有自己的计算架构，且非常适合并行处理任务，比如运行神经网络。因此，我这么做是为了让模型驻留在 GPU 上——一个完全独立的计算机——这样可以让我们的代码运行得更高效，因为所有这些操作在 GPU 上运行效率要高得多。这就是模型本身的情况。

现在，下一步是我们想要在生成时使用这个前缀。因此，我们实际上创建这些前缀令牌。这是我写的代码。我们将导入 OpenAI 的 TICK token 库，然后获取 GPT-2 的编码器（tokenizer）。接着，我们将对这个字符串进行编码，并获取一个整数列表，即令牌的列表。现在，这些整数应该相当直观，因为我们可以直接复制粘贴这个字符串，并在 TICK token 库中查看它。粘贴后，这些就是我们期望得到的令牌。这些整数列表就是我们期望令牌转化后的结果。如你所见，如果你看过我的视频，你知道所有的令牌其实都是小的字符串块。

一旦我们有了这些令牌，它们就是一个整数列表。我们可以将它们转换为 PyTorch 张量。在这个例子中，有八个令牌，然后我们将这些八个令牌复制五次，得到五行八个令牌，这就是我们初始的输入 X，像我在这里所称呼的那样。它也驻留在 GPU 上，因此 X 现在是这个索引，我们可以把它传入前向函数，获取我们的 logits，从而知道在每一行的第六个令牌之后，应该是什么。抱歉，是第九个令牌。好了，我们现在准备好生成内容了。让我再粘贴一个代码块。”

“这里呢，代码块里发生的事情是，我们有一个X，它的大小是B x T，对吧？所以是batch和time。我们在每次迭代这个循环时，会向每一行中添加一个新的索引列，对吧？这些是新的索引，我们将它们添加到序列中进行采样。因此，每次循环迭代时，我们会在X中再添加一列，所有的操作都发生在PyTorch的上下文管理器`torch.no_grad()`中。这个只是告诉PyTorch，我们不会在这些操作上调用`backward`，所以它不需要缓存所有的中间张量，也不需要为可能的`backward`操作做准备，这样可以节省很多空间，也可能节省一些时间。所以我们得到的是logits，我们只在最后一个位置得到logits，丢弃其他位置的logits，我们不需要它们；我们只关心最后一列的logits。所以，虽然这种做法有些浪费，但它是采样的正确实现，只是效率不高。接着，我们将最后一列logits通过softmax得到概率，然后我这里使用的是top K采样，K值为50，这是因为这是Hugging Face的默认设置。通过查看Hugging Face的文档，我们知道他们默认使用的是top K采样，K值是50。这个操作的效果是，我们从概率分布中选择前50个概率最高的tokens，低于第50个概率的tokens就被置为零，并重新标准化。这样，我们就不会采样到非常稀有的tokens，采样的tokens总是位于前50个最可能的tokens中，这有助于保持模型的轨迹，不会东拉西扯，也不会容易跑偏。总之，这种方法可以让模型更好地保持在合理的tokens范围内。对于

PyTorch来说，应该就是这种做法。如果你愿意，可以逐步查看实现细节。我认为这部分并没有什么特别深刻的地方，所以我会快速跳过。

大体上，我们得到了新的tokens列，并将它们附加到X中，X的列数会随着循环增长，直到Y循环被触发。最后，我们得到一个完整的X，大小是5 x 30，在这个示例中，接着我们可以打印出每一行的tokens。于是，我就打印出所有的行，所有被采样的tokens，我使用Tik tokenizer的decode函数来将它们转换回字符串，然后打印出来。接下来，打开一个新的终端，运行Python训练GPT-2。好吧，这是我们得到的生成结果：

“Hello, I'm a language model, not a program.”（换行）“Hello, I'm a language model, and one of the main things that bothers me when they create languages is how easy it becomes to create something that I mean.”等等。结果就这样开始了，模型会继续生成。这些生成结果并不完全与Hugging Face的生成结果相同。老实说，我无法找到差异所在，我也没有完全检查所有的选项，但很可能在top P采样之外，还有其他因素影响结果。因此，我暂时无法与Hugging Face的结果完全匹配。但就正确性而言，下面在Jupyter Notebook中使用Hugging Face模型，执行代码之后，我得到了相同的结果。因此，模型内部并没有错，问题可能出在Hugging Face的pipeline实现上，这就是我们无法完全匹配结果的原因。否则，代码是正确的。

我们已经正确加载了所有张量，正确初始化了模型，所有的操作都成功。所以，总结一下，我们已经导入了所有的权重，初始化了GPT-2模型，这就是标准的GPT-2模型，它可以生成合理的序列。接下来，我们要做的是，从随机数开始初始化，而不是使用GPT-2的权重。我们希望训练一个模型，它生成的序列质量能和GPT-2一样好，甚至更好。接下来我们就开始这个部分。实际上，使用随机初始化模型是比较简单的，因为PyTorch默认就会用随机初始化方式来初始化模型。所以，当我们创建GPT模型时，所有的层和模块都会默认用随机初始化的方式进行初始化。当这些线性层被创建时，它们会使用类似Xavier初始化的默认方式来构造这些层的权重。因此，创建一个随机初始化的模型其实很简单，不需要额外的复杂操作。

这里，我们改成创建一个叫做GPT的模型，使用默认的GPT配置，默认配置有124M的参数。这就是随机初始化模型的实现，我们可以运行它，应该能够得到结果。这里的结果当然是完全的垃圾，因为这只是一个随机初始化的模型，得到的token字符串完全是随机的，完全是胡乱的token片段。这就是我们目前得到的结果。顺便提一句，如果你没有CUDA可用，因为没有GPU，你仍然可以跟着我们做一些操作，虽然到最后我们会用多个GPU进行真正的训练，但在此之前，你其实还是能够在一定程度上跟得上的，明白了吗？”

我喜欢在PyTorch中做的一件事是自动检测可用的设备。特别地，你可以这样做。在这里，我们试图检测一个具有最高计算能力的设备，你可以这样理解。所以，默认情况下，我们从CPU开始，CPU当然是到处都可用的，因为每台计算机都会有CPU。但接着我们可以尝试检测，是否有GPU？如果有CUDA，就使用CUDA；如果没有CUDA，至少有没有MPS？MPS是Apple Silicon的后端，如果你有一台相对较新的MacBook，里面很可能就有Apple Silicon芯片，然后它配备的GPU实际上是相当强大的，取决于你的MacBook型号。因此，你可以使用MPS，它可能比CPU更快。

现在，一旦我们获得了设备，我们就可以将其用作CUDA的替代。所以我们只需要替换它，注意，在这里，当我们调用模型处理X时，如果这个X是在CPU而不是GPU上，那么它也会正常工作。因为在前向传播中，当我们创建pose时，我们已经小心地使用idx的设备来创建这个张量。因此，不会发生设备不匹配的情况，即一个张量在CPU上而另一个在GPU上，这样你就不能将它们结合在一起。

但是在这里，我们已经小心地在正确的设备上进行初始化，如输入模型所示。这个设备会自动检测给我。这个设备将是GPU，因此使用CUDA设备。然而，正如我提到的，你也可以使用另一个设备，速度不会慢太多。如果我在这里覆盖设备，哦，如果我将设备设置为CPU，那么我们仍然会打印CUDA，但现在我们实际使用的是CPU。好，一二三四五六。好，大约六秒钟，实际上我们没有使用Torch compile等功能，这样会使一切运行得更快。但即使在CPU上，我认为你也可以在一定程度上跟得上。

关于这一点就说这么多。好，我确实想回顾一下在PyTorch中使用不同设备意味着什么，PyTorch在后台为你做了什么，当你像module.to(device)这样做时，或者当你将Torch张量传递给to(device)时，究竟发生了什么以及是如何工作的。但目前，我希望能够开始训练模型。现在我们就先说，设备使代码运行得更快。

实际上，要训练模型，我们需要一个数据集。对我来说，我喜欢使用最简单的调试数据集——Tiny Shakespeare数据集。它可以通过这个URL获取，或者你可以搜索Tiny Shakespeare数据集。我已经下载好了，它在我的文件系统中是input.txt。我已经下载它了，现在我正在读取数据集，获取前1,000个字符，并打印前100个字符。现在，记住GPT-2大约有一个压缩比。分词器的压缩比大约是3比1，所以这1,000个字符大约是300个tokens。这是前几个字符，如果你想获取更多的统计信息，我们可以对input.txt进行单词计数。我们可以看到这大约是40,000行，约200,000个单词，文件大小大约为1百万字节。知道这个文件只包含ASCII字符，没有什么复杂的Unicode字符，据我所知。

天文字符使用一个字节进行编码，所以这个数据集大约包含100万个字符。这样数据集的大小，默认是非常小的，适合用于调试，帮助我们起步。为了对数据进行分词，我们将使用Tik分词器对GPT-2进行编码。首先，处理1,000个字符，然后只打印前24个tokens。

这些是作为整数列表的tokens，如果你能读取GPT-2的tokens，你会看到198，这里你会认出来它是斜杠字符；所以这是一个换行符。然后，例如，这里我们有两个换行符，所以198在这里重复了两次，嗯，这只是前24个token的tokenization。接下来我们要做的是，我们希望实际上处理这些token序列并将它们输入到Transformer中，特别是我们想要将这些token重新排列成我们将要输入Transformer的idx变量。所以，我们不想要一个很长的一维序列；我们需要一个完整的batch，每个序列的长度最多为T，且T不能超过最大序列长度，然后我们有这些很长的token序列。我们有B个独立的序列样本，那么我们如何创建一个BYT张量来将这些一维序列输入到模型的前向传播中呢？这是我最喜欢的方式来实现这一点。所以，如果我们使用torch，我们可以从这个整数列表中创建一个tensor对象，且仅取前24个tokens。我的最爱方式是将其view为例如4x6，这样相乘得到24。所以这只是将这些tokens做成一个二维的4x6排列。当你将这个一维序列作为二维的4x6来查看时，前六个tokens（直到这里）会变成第一行，接下来的六个tokens会变成第二行，依此类推。基本上，它就是将每六个tokens堆叠成独立的行，创建一个token的batch。例如，如果我们在Transformer的第25个token时输入这个，那么这个token就会看到前面的三个tokens，它将尝试预测下一个token。通过这种方式，我们能够创建一个二维的batch，非常方便。

接下来，关于我们需要的目标标签以计算损失函数，我们该如何获取呢？嗯，我们可以在前向传播中写一些代码，因为我们知道一个序列中的下一个token，也就是标签，就是它右边的token。但是你会注意到，实际上，对于序列中的最后一个token，我们没有正确的下一个token，因为我们并没有加载它。所以，实际上我们没有足够的信息来处理这一点。接下来，我会展示我最喜欢的方式来获取这些batch。我个人喜欢不仅仅有Transformer的输入，我喜欢将其称为X，还喜欢创建一个和X大小完全相同的标签tensor，里面包含每个位置的目标。所以，下面是我喜欢做的方式。我喜欢确保我获取到下一个token，因为我们需要获取到最后一个token的ground truth。对于第13个位置的token，当我们创建输入时，我们会取到最后一个token之前的所有内容，但不包括它，并将其视为4x6。

在创建目标时，我们会将缓冲区从索引1开始，而不是从索引0开始，这样我们跳过了第一个元素，然后将其视为完全相同的大小。当我打印时，结果如下：基本上，对于这个标记25，它的目标是198，现在它就存储在目标张量的相同位置，值是198。同样，最后的标记13现在也有了它的标签，值是198，因为我们在这里加载了这个额外的标记。基本上，这是我喜欢的做法：你将长序列视为二维形式，这样就能得到一个批次，然后我们确保加载了一个额外的标记。我们基本上加载了一个标记缓冲区，大小为B * t + 1，然后我们将它们偏移并进行视图转换。接下来我们有两个张量：一个是Transformer的输入，另一个就是对应的标签。现在，我们来重新组织这段代码，创建一个非常简单的数据加载器对象，试图将这些标记加载并输入到Transformer中，计算损失。

好吧，我重新安排了代码。正如你在这里看到的，我暂时覆盖了U以运行CPU并导入了TI token，所有这些应该都很熟悉。我们加载了1000个字符。我将BT设置为4和32，仅仅是因

为我们在调试。我们只想要一个非常小的批次，所有这些现在看起来应该都很熟悉，并且遵循我们之前的做法。在这里，我们创建了模型并获取了日志，然后，正如你看到的，我已经运行过了。这只需要几秒钟，但因为我们的批次是4x32，所以我们的日志现在的大小是4x32x50257。这些是每个位置的日志。现在，我们有了存储在y中的标签，因此现在是时候计算损失了。

接下来，我们进行反向传播和优化。所以，首先我们计算损失。好吧，为了计算损失，我们将调整模型中NN模块的前向函数。具体来说，我们不仅仅会返回logits，还将返回损失，我们不仅会传入输入数据，还会传入目标数据y。

不要打印，只要返回损失函数。实际上，我们将打印损失函数并在零处退出，以便跳过一些采样逻辑。现在，让我们转到 `forward` 函数，它在那里被调用，因为现在我们也有了这些可选的目标。当我们获取到目标时，我们也可以计算损失，并且记住，我们基本上要返回日志损失。默认情况下，它是 `none`。但是，我们可以把它放在这里：如果目标不是 `none`，那么我们就计算损失。Co-pilot 已经开始计算，看起来损失是正确的，它使用的是交叉熵损失，如此处所示。这是 PyTorch 中的一个函数，位于函数式 API 下。那么，这里到底发生了什么呢？因为看起来有点吓人，基本上，`F` 这个交叉熵函数不喜欢多维输入。它不能接受 `B x T` 和词汇表大小的输入。因此，这里发生的事情是，我们将这个三维张量展平为二维。第一维将是 `B * T`，然后最后一维是词汇表大小，所以基本上是将这个三维的 logits 张量展平为二维的 `B * T`，每行的长度就是词汇表的大小。然后，它还会将目标展平，它们在这个阶段也是二维的，但我们将把它们展平，使它们变成一个 `B * T` 的单一张量，然后可以传递给交叉熵函数来计算损失，最后返回。所以，基本上到这个时候，代码应该可以运行，因为这并不复杂。让我们运行它，看看是否应该打印损失，在这里我们看到打印出的值大约是 11，嗯，大致如此。并且注意到，这个损失是一个单一元素的张量，就是这个数字 11。

现在，我们还想计算一个合理的起点，以便对一个随机初始化的网络进行检查。我们在之前的视频中讲过，词汇表的大小是 50,257。在网络初始化时，我们希望每个词汇元素的概率大致是均匀的，这样我们就不会在初始化时过于偏向任何一个 token。我们不会在初始化时就过于自信地错得离谱。所以我们希望，任何任意 token 的概率大致是 `1/50,257`。现在，我们可以进行损失的合理性检查，因为记住，交叉熵损失基本上就是负对数似然。现在，如果我们拿这个概率，经过自然对数转换后，再取负值，这就是我们在初始化时期望的损失。我们在之前的视频中讲过这个，所以我预计应该是大约 10.82，而我们看到的是大约 11。所以，它并没有偏离太多。这是我在初始化时大致期望的概率分布，它说明初始化时的概率分布大致是均匀的，这是一个很好的起点。现在我们可以进行优化，并告诉网络哪些元素应该按照正确的顺序出现。所以，到这一步，我们可以进行反向传播，计算梯度，并进行优化。

现在，让我们开始优化。这里是我们获取损失的方式。但现在，基本上我们想要一个循环。对于 `i` 在 `range` 中，我们可以做 50 步左右。让我们在 PyTorch 中创建一个优化器对象。在这里，我们使用 Adam 优化器，它是我们之前使用的随机梯度下降（SGD）优化器的替代方案。SGD 更简单；而 Adam 更复杂，我实际上特别喜欢 AdamW 变种，因为在我看来，它修复了一个 bug。所以，AdamW 就是 Adam 的 bug 修复版本。

当我们查看 AdamW 的文档时，哦天呐，我们看到它需要一堆超参数，并且比我们之前看的 SGD 更加复杂。因为，除了基本上使用梯度更新参数并根据学习率进行缩放外，它还会保留一些缓冲区，它会保持两个缓冲区：`m` 和 `v`，这两个分别叫做第一和第二矩。它有点像动量，也有点像 RMSProp，如果你了解的话。但是你不必深入了解它，它只是对每个梯度元素进行的单独归一化，并且加速了优化，尤其是在语言模型中。但是我在这里不会详细讲解。我们将把它当作一个黑箱来使用，它比 SGD 更快地优化目标，这是我们在之前的讲座中看到的。所以让我们在我们的案例中把它当作一个黑箱使用。创建优化器对象，然后进行优化。

首先，确保 Co-pilot 没有忘记清零梯度。所以，始终记住，你必须从零梯度开始。然后，当你得到损失并执行反向传播时，它会将梯度添加上去。所以，它总是对梯度执行加法操作，这就是为什么你必须先将它们设置为零，否则它会累积来自这个损失的梯度。然后，我们调用优化器的 `step` 函数来更新参数并减少损失。这里使用了损失项，因为损失是一个包含单个元素的张量。

该项操作实际上会将其转换为一个单一的浮动数值，并且这个浮动数值会保存在CPU上。所以这涉及到设备的内部实现。但是，`loss`是一个包含单一元素的张量，它保存在GPU上，因为我正在使用GPU。当你调用`item`时，PyTorch会在后台将这个一维张量传回CPU内存，并将其转换为一个浮动数值，这样我们就可以直接打印出来。因此，这是优化过程，应该可以正常工作。让我们看看会发生什么。实际上，抱歉，先不使用CPU的覆盖设置，我把它删除掉，这样运行速度会快一点，并且它会在CUDA上运行。

哦，期望所有张量都在同一设备上，但发现至少有两个设备：CUDA 0 和 CPU。所以CUDA 0是第一个GPU，因为我实际上有八个GPU在这个机器上。所以CUDA 0就是我的第一个GPU，和CPU一样，模型已经被移动到设备上。

但是，当我编写这段代码时，我实际上引入了一个bug，因为缓冲区从未移动到设备上，你必须小心，因为你不能仅仅使用`buffer.to(device)`。设备是无状态的，它并不会直接将它转换到设备上。相反，它会返回一个指向新内存的指针，该内存位于设备上。所以，你会看到我们可以使用`model.to(device)`来操作模型，但这不适用于张量。你必须使用`buf = b.to(device)`，然后这就可以正常工作了。好的，那么我们期望看到什么？我们期望一开始看到一个合理的损失值，然后继续优化这个单一的batch。所

以，我们希望看到我们能够对这个单一的batch进行过拟合。我们可以压制这个小batch，并且能够完美地预测这个batch上的索引。确实，这就是我们所看到的。

所以，我们一开始大约在10.82到11之间，然后随着我们继续对这个单一的batch进行优化，并且没有加载新的样本，我们确保我们可以对这个单一的batch进行过拟合。我们正在得到非常非常低的损失值，因此Transformer正在记忆这个单一的batch。我没有提到的另一点是，学习率是3E-4，这是大多数情况下的一个不错的默认值。在调试的早期阶段，这是我们的简单内循环，我们正在对单一batch进行过拟合。看起来很好。那么接下来呢？我们不仅仅想对单一batch进行过拟合，我们实际上想进行优化。所以，我们需要迭代这些XY批次，并创建一个小的数据加载器，确保我们总是获取到一个新批次，并且我们实际上是在优化一个合理的目标。让我们下一步做这个。

好的，这是我想出来的办法。我写了一个小数据加载器。这个数据加载器做的是，我们在这里导入token，然后读取整个文本文件`input.txt`，对其进行token化，然后我们只是打印出总共的tokens数量和在一个epoch中遍历该数据集时的批次数量。我们输出多少个独特的batch，然后再从文档的开始处重新读取。所以我们从位置0开始，然后我们就以B * T的batch大小来遍历文档。所以我们每次取B * T的块，然后总是以B * T为步长前进。

需要注意的是，我们总是以B * T精确地前进，但当我们获取tokens时，我们实际上是从当前的位置获取到B * T + 1的位置，我们需要+1是因为记得我们需要当前batch最后一个token的目标token。通过这种方式，我们就可以像之前一样做XY。如果我们用完了数据，我们就会重新回到0的位置。这个数据加载器非常简单，完全可以满足当前的目的，当然我们以后会对它进行复杂化。现在，我们想回来使用这个数据加载器。所以导入`Tiktoken`已经被移到了上面，实际上这些代码现在是没用了。我们只需要一个训练数据的训练加载器，并且我们想要使用相同的超参数。因此，B的大小是4，时间是32。然后，这里我们需要获取当前batch的XY。看看Copal是否可以处理它，因为这个足够简单。所以我们调用下一个batch，然后确保我们把张量从CPU移动到设备上。

所以，在这里，当我转换tokens时，注意到我没有实际把这些tokens移动到GPU，我把它们留在了CPU上。这是默认行为，这只是因为我尽量不浪费GPU的内存。在这个例子中，这是一个很小的数据集，完全可以适应GPU，但现在将它发往GPU进行处理是没问题的。所以我们获取下一个batch，保持数据加载器简单为CPU类，然后我们就将它送到GPU并进行所有计算。看看它是否能正常运行。好吧，运行`python train gbt2 pi`，然后我们期望看到什么？在实际运行之前，我们期望看到的是，我们现在实际获取到下一个batch，因此我们不再会对单一batch进行过拟合。我希望我们的损失值会下降，但不会下降太多，因为我仍然希望它下降；在50,257个tokens中，很多tokens在我们的数据集中是不会出现的。在优化过程中可以做一些非常简单的优化，例如，通过将从未出现的tokens的偏置驱动到负无穷大。那基本上就会是所有这些奇怪的唯一代码。

不同语言中，这些tokens从未出现过，因此它们的概率应该非常低。所以我们应该看到的提升主要是通过删除从未出现过的tokens的使用。这可能是我们在当前规模下能看到的损失-增益的主要来源。但是我们不应该看到零的损失，因为我们只做了50次迭代，而我认为这还不足以完成一个epoch。所以让我们看看结果。我们有338,000个tokens，这与我们的3:1压缩比是相符的，因为总共有100万个字符。所以，在当前B和T的设置下，一个epoch将需要2600个batch，而我们只做了50个优化batch。所以，正如预期的那样，我们从熟悉的区域开始，然后损失值似乎降到了大约6.6。因此，基本上，现在关于我们预期的结果似乎是正常的。所以这很好。

好的，接下来我想修复我们代码中的一个bug。嗯，这不是一个主要的bug，但它是一个关于GPT-2训练应该如何进行的问题。

问题如下：当我们从Hugging Face加载权重时，我们没有足够小心，实际上遗漏了一个小细节。如果我们来看这里，注意到这两个张量的形状是相同的。所以这里的第一个是Transformer底部的token嵌入，对吧？然后这个是Transformer顶部的语言模型头。这两个张量基本上是二维的，它们的形状是相同的。这里，第一个是输出嵌入，token嵌入，第二个是Transformer顶部的这个线性层，分类层。它们的形状都是50257 x 768。嗯，第一个给我们提供了底部的token嵌入，而第二个则接收Transformer的768个通道，并试图将其扩展到50257，以获取下一个token的列表。所以它们的形状是一样的，但更重要的是，实际上，如果你看比较它们的元素，嗯，在PyTorch中，这是一个逐元素相等的比较，所以我们使用`do_all`，并看到每个元素都是一样的。此外，如果我们实际上看一下数据指针，这是PyTorch中获取实际数据指针和存储的方式，我们会发现其实它们的指针是相同的。所以，不仅这两个是形状和元素都相同的独立张量，它们实际上指向的是相同的张量。那么这里发生的情况是，这是一种常见的权重绑定方式，实际上来源于原始论文。

来自“Attention is All You Need”论文，实际上在它之前的参考文献中也有提到。如果我们来看这里，论文中提到，在我们的模型中，我们在两个嵌入层和预softmax线性变换之间共享相同的权重矩阵，类似于30中的描述。所以这是一个尴尬的说法，意思是这两个是共享的，它们是绑定在一起的，实际上是同一个矩阵。30的引用指的是这篇论文，发表于2017年，你可以阅读完整的论文，但基本上它主张了这种权重绑定方案。我认为直觉上，为什么你可能会想这么做，来自于这里的这一段，基本上你可以观察到，嗯，如果两个token在语义上非常相似，比如其中一个全是小写字母，另一个全是大写字母，或者它们是同一个token在不同语言中的表现形式，或者类似的情况。如果两个token之间有相似性，那么你可以推测，它们在token嵌入空间中应该是接近的。

但是以同样的方式，你也可以预期，如果你有两个在语义上相似的tokens，你会预期它们在Transformer的输出中会得到相同的概率，因为它们在语义上是相似的。因此，Transformer中的两个位置，底部和顶部，都应该具备这种特性，即相似的token应该有相似的嵌入或相似的权重。这就是激励它们探索这一点的原因，他们也观察到，如果你看输出嵌入，它们的行为也像词嵌入。如果你仅仅试图将这些权重作为词嵌入来使用，他们观察到这种相似性。所以他们尝试将它们绑定在一起，并且观察到这种方式能够获得更好的性能。因此，这种方法被采纳并用于“Attention is All You Need”论文中，之后也被GPT-2所使用。

所以我在Transformers实现中找不到它。我不确定他们是在哪里绑定这些嵌入的，但我可以在OpenAI推出的原始GPT-2代码中找到它。这个是，嗯，OpenAI GPT-2源代码，在这里他们正向传播这个模型。虽然这是在TensorFlow中实现的，但，嗯，这没关系。我们看到他们获得了WTE token嵌入，然后这里是token嵌入和位置的编码器，接着在底部他们再次添加了WTE来计算logits。所以，当他们得到logits时，它是Transformer输出的一个数学模型，而WTE张量被重用了。基本上，它被使用了两次：一次是在Transformer的底部，一次是在Transformer的顶部。在反向传播时，我们将从两个分支得到梯度贡献，对吧？这些梯度将在WTE张量上累加。我们会得到来自分类器列表的贡献，然后在Transformer的最底部，我们会再次得到一个贡献，最终又会传递回WTE张量。因此，我们希望——我们目前的代码没有共享WTE，但我们希望这样做。

那么，权重共享方案——做这件事的一种方式是什么？让我们看看Goil是否能搞定。哦，搞定了！好的，这是一种做法。嗯，基本上，这是相对直接的。我们在这里所做的就是，我们拿到了WTE权重，然后简单地将它重定向到LM头。也就是说，它基本上复制了数据指针，对吧？它复制了引用，现在WTE权重变得孤立。它的旧值，PyTorch会清理它，Python也会清理它。所以，我们只剩下一个单独的Tensor，它将在前向传播中被使用两次。根据我的了解，这就是所有要求，所以我们应该能够使用这个。这应该能训练。我们基本上只是使用这个完全相同的tensor两次。

我们在跟踪可能性时不够小心，但根据论文和结果，你实际上应该预期这样做会得到稍微更好的结果。除此之外，另一个让这件事对我们非常有利的原因是，这涉及到了大量的参数。对吧？这里的大小是多少？是768 * 50,257，所以这是4000万个参数，而这个模型有1.24亿个参数。所以，4000万除以1.24亿，大约30%的参数是通过这个权重绑定方案被节省的。如果你没有足够长时间训练模型，这可能是它工作稍微更好的原因之一，因为权重绑定，你不需要训练那么多的参数，所以在训练过程中会更高效。因为你减少了参数，你投入了这个。

归纳偏置表明这两个嵌入应该在token之间共享相似性。所以这是时间方案的工作方式，我们节省了大量的参数。我们预计，由于这个方案，我们的模型表现会稍微更好。好的，接下来，我希望我们能在初始化时更加小心，并尽量跟随GPT-2初始化模型的方式。现在，不幸的是，GPT-2论文和GPT-3论文对初始化并没有非常明确的说明，所以我们有点需要在行间阅读。与其去阅读论文，它说得非常模糊，不如在OpenAI发布的代码中找一些信息。当我们查看model.py时，我们看到他们初始化权重时，使用了标准差0.02，这就是他们的初始化方式。这是权重的正态分布，标准差是0.02。对于偏置，他们将其初始化为零。然后，当我们向下滚动时，我们看到token嵌入被初始化为0.02，而位置嵌入则初始化为0.01，不知道为什么。

我们希望在我们的模块中模仿这一点。所以这里有一段我迅速写出来的代码。这里发生的事情是，在GPT模块的初始化器结束时，我们调用了NN模块的apply函数，它会遍历这个模块的所有子模块，并在它们上应用权重函数。于是，我们在这里遍历所有模块，如果它们是nn.Linear模块，我们就确保使用标准差为0.02的正态分布来初始化权重。如果这一层有偏置，我们会确保将其初始化为零。需要注意的是，偏置的零初始化实际上并不是PyTorch的默认行为。默认情况下，偏置是通过均匀分布初始化的，所以，这一点很有意思。所以我们确保使用零，并且对于嵌入，我们将继续使用0.02，不会将其改为0.01，除非它是位置嵌入，因为它差不多是一样的。然后如果你查看我们的模型，唯一其他需要初始化并且有参数的层是层归一化和进一步的推迟。

初始化将层归一化中的尺度设置为1，将偏移量设置为0。所以这正是我们想要的，因此我们将保持这种方式。如果我们遵循OpenAI发布的GPT-2源代码，这就是默认的初始化。我想指出，顺便提一下，通常如果你遵循Javier初始化，标准差应该是1除以进入该层的特征数的平方根。但如果你注意到，实际上0.02基本上和这个一致，因为在这些Transformer中的模型大小大约是768、1600等。所以，假如我们取768的平方根，得到0.03。如果我们插入600或1600，得到0.02。如果我们插入三倍大小，得到0.014，等等。所以基本上，0.02大致是这些初始化的合理值。因此，将0.02硬编码在这里并不是完全疯狂的事情。然而，你通常会希望一些一致性，随着模型大小的增大而增长，但我们将保持这个值，因为它是GPT-2源代码中的初始化方式。

但是，初始化还没有完全完成，因为这里还有一个附加条件。这里使用了mod初始化，它考虑了残差路径上随着模型深度的积累。我们通过一个因子1除以n的平方，来缩放残差层的初始化权重，其中n是残差层的数量。这就是GPT-2论文中提到的。我们还没有实现这一点，嗯，我们现在可以做到了。我想稍微动机一下他们在这里的意思。我觉得，嗯，大概是这样的：如果你从零开始初始化残差流，记住每个残差流都是这种形式：我们继续加它：X等于X加上一些东西，一些贡献。因此，残差网络的每个块都会贡献一定的量，并且它会被加起来。所以最终发生的事情是，残差流中激活的方差会增加。这里有一个简单

的例子：如果我们从零开始，然后加上100次，嗯，我们有一个768个零的残差流，然后100次我们加上随机噪声，这些噪声是均值为零，标准差为1的正态分布。如果我们继续加下去，那么到最后，残差流的标准差已经增长到了10，这仅仅是因为我们总是在加这些数字。

因此，他们在这里使用的缩放因子正好补偿了这种增长。如果我们取n并基本上将每个对残差流的贡献缩放为1除以n的平方根，也就是1除以n的0.5次方。对吧，因为n的0.5次方是平方根，然后1除以平方根就是n的-0.5次方。如果我们以这种方式进行缩放，那么我们看到实际得到的值是1。所以，这是一种在前向传播中控制残差流中激活增长的方法，因此我们希望以相同的方式初始化这些权重，即每个块结束时的这些权重，因此在GPT论文中提议将这些权重缩放为1除以残差层数量的平方根。

这就是接下来的内容：我不知道这是否是PyTorch官方支持的，但它有效。对于我来说，我们将在初始化中做这个；看到那个“s”标志，以处理特殊的NaN值。GPT缩放是1，因此我们为这个模块设置了一个标志。可能有更好的方法在PyTorch中实现，但我不确定。好的，所以我们基本上附加了这个标志，并确保它不会与之前的任何东西冲突。然后，当我们来到这里时，STD默认应该是0.02。但如果那个模块是这种情况，那么STD就会乘以，确保它不正确地猜测。所以我们想要根据残差层的数量来缩放标准差。残差层的数量是Saltout配置层的两倍，再乘以0.5。所以我们想要缩小标准差，这应该是正确的。我应该澄清一下，实际上“层数的两倍”来自于我们Transformer中的每一层，实际上我们有两个块加入到径向路径中。对吧，我们有注意力机制和MLP，所以这就是两个类型的来源。

另外一个需要提到的是，嗯，虽然有点尴尬，但我们不会去修正它的是，由于我们在这一版的子模块中共享了WTE和LM头，我们实际上会对这个张量进行两次初始化。第一次，我们会将它初始化为嵌入，并且用0.02来初始化，然后我们会再次回到它，在一个线性层中，并再次用0.02初始化它。它将是0.02，因为LM头当然不进行缩放，所以它不会到这里。它只是会被基本上用相同的初始化初始化两次。但这没关系。然后，滚动到这里，我添加了一些代码，以便我们能够设置随机种子来保证可重复性，现在我们应该能够用Python训练GPT-2，并让它运行。就我所知，这就是我们当前实现的GPT-2初始化方式。所以看起来这对我来说是合理的。好了，现在，我们已经有了GPT-2模型，并且有一定的信心它已正确实现。我们已经正确初始化了它，并且有一个数据加载器，正在迭代数据批次。

我们可以开始训练了，接下来是有趣的部分。我希望我们能大大加速训练，从而充分利用我们在这里使用的硬件，以确保硬件的性价比。因此，我们将大幅加速训练。

现在，你总是希望先了解你拥有什么样的硬件，它能提供什么，是否得到了充分的利用？以我的情况为例，如果我们查看Nvidia SMI，可以看到我有八个GPU，每个GPU都是A100 SXM，配备80GB显存。这就是我在这个机器上可用的GPU。

当我查看这些A100时，A100 80GB SXM是我在这里的GPU，我们有一堆数字，表示这个GPU可以执行多少计算。所以当我运行Python代码时，代码会在我们计算负载和法律之后进入。而我希望你注意到的是，当我使用dtype时，它会打印出torch.float32。默认情况下，在PyTorch中，当你创建张量时（对于所有的激活值和网络的参数等），默认所有的张量都是float32。这意味着每一个数字、激活值或权重等，都使用了32位的浮点表示，这实际上占用了相当多的内存。实际上，通过实验证明，对于深度学习这样的计算工作负载来说，这个精度是过高的。深度学习和网络的训练能够容忍显著较低的精度。并不是所有的计算工作负载都能容忍这么低的精度。

回到数据表，你会看到其实这些GPU支持的精度高达FP64，这在很多科学计算应用中是非常有用的。我理解，确实有这个需求，但我们在深度学习训练中并不需要这么高的精度。目前，我们使用的是FP32，而根据现在这段代码的设定，我们预期最多能获得19.5 TeraFLOPS的性能。这意味着我们每秒执行19.5万亿次浮点操作——通常是浮点乘加运算。那就是浮点运算。

现在请注意，如果我们愿意降低精度，那么TF32是一个较低精度的格式，我们稍后会看到，实际上你可以在这里获得8倍的性能提升。如果你愿意降低到float16或B float16，你实际上可以获得16倍的性能提升，性能可达到312 TeraFLOPS。你会看到，Nvidia喜欢引用带有星号的数字，这个星号表示“使用稀疏性”，但我们在代码中不会使用稀疏性，我也不确定这个在行业中是否得到广泛应用。

人们通常会看这个没有稀疏性的数字，你会注意到，我们本来可以得到更多的数字，但这是INT8，而INT8通常用于推理而不是训练。因为INT8有均匀的间距，而我们实际需要的是浮点数，这样可以更好地匹配训练过程中神经网络的正常分布，其中激活值和权重通常呈正态分布。所以，浮点数在这里真的非常重要，以便能够匹配这种表示方式。因此，我们通常不会在训练中使用INT8，但在推理时会使用它。

如果我们降低精度，我们可以从GPU的张量核心中获得更多的TeraFLOPS。稍后我们会讨论这个，但除此之外，如果所有的数字、权重和激活值都使用较少的比特表示，它们的存储需求将会减少，从而加速数据的传输和处理。这就是我们开始进入内存带宽和模型内存的问题。

不仅如此，我们的GPU存储的比特数量是有限的，而且访问这些内存的速度也有限，你有一定的内存带宽，这是一个非常珍贵的资源。事实上，许多深度学习训练工作负载是受内存带宽限制的。这意味着，实际上执行所有这些快速乘法操作的张量核心大部分时间处

于空闲状态，因为我们无法足够快地将数据从内存中加载到GPU中。所以如果硬件利用率达到60%，那么实际上你已经做得非常好了。在一个调优良的应用中，张量核心有一半的时间是没有进行乘法运算的，因为数据无法及时提供。

所以，内存带宽在这里也极为重要，而如果我们降低浮点数的精度，所有的数字、权重和激活值突然间需要的内存更少了，这样我们就可以存储更多，并且可以更快地访问它们。所以一切都加速了，效果非常惊人。那么，现在让我们开始收获这些好处，首先看一下张量浮点32（TensorFloat32）格式。

首先，什么是张量核心？张量核心是A100架构中的一个指令。它基本上执行的是一个4x4矩阵乘法。这里的矩阵乘法就是4x4矩阵乘法，并且在每个矩阵的精度、内部累积的精度以及输出精度等方面有多种配置。有几个开关，但它基本上是4x4的乘法。

每当我们有需要矩阵乘法的操作时，它们会被拆分成这个4x4乘法指令。所有的计算工作都会被拆解成这种指令，因为这是乘法矩阵的最快方式。实际上，我们上面做的大部分计算工作，所有工作，实际上都是矩阵乘法。大部分的计算工作发生在线性层中，嗯，线性层等等。

中间有一些东西夹在其中，包括一些残差的加法、一些G非线性激活和一些层归一化等。但是，如果你对它们进行计时，你会发现，Transformer实际上就是一堆矩阵乘法。嗯，特别是在这个小规模的124百万参数模型中，实际上最大的矩阵乘法就是位于顶部的分类层。这是一个巨大的矩阵乘法，从768维到50,257维，这个矩阵乘法在整个网络中的计算占据了主导地位，粗略来说就是如此。所以，实际上，矩阵乘法会变得更快，它们隐藏在我们的线性层中，并通过张量核心加速。

现在，我认为张量核心的最佳参考资料基本上就是去看A100架构的白皮书，它解释得非常详细。我觉得人们大多数情况下可以理解，尤其是如果你对发生的事情有一定了解的话。嗯，图9展示了张量浮点32（TF32）的解释。基本上，TF32就是在矩阵乘法中发生的事情，你会看到有很多配置选项可以选择。比如输入操作数的精度、累加器的精度，嗯，基本上就是在执行矩阵乘法时，指令内部的表示形式，比如中间结果的累加操作。

这一切都发生在FP32精度下。然后，正如我之前提到的，这是一个对操作的加速提升。因此，TF32特别是我们正在查看的这一行，它的工作原理是：通常情况下，FP32有32个比特，而TF32的比特数完全相同，包含一个符号位，8个指数位，唯一的区别就是尾数部分被裁剪了。所以，基本上我们最终只得到了19个比特，而不是32个比特，因为最后的13个比特被截断了。它们被丢弃了，而这一切发生在指令内部，所以在我们的PyTorch代码中是不可见的。

我们的PyTorch代码不会发生任何变化，所有数字看起来都是一样的。只是当你在硬件内部调用张量核心指令时，它会将这13个比特裁剪掉，这让它能够显著加速这个小矩阵的乘法——提高了8倍的速度。

当然，这种加速是有代价的，代价就是我们减少了累加的精度。累加操作仍然是FP32，输出仍然是FP32，输入也是FP32，但内部的操作数在执行操作时会被裁剪，从而更快速地进行计算，因此我们的结果开始变得稍微更为近似。然而，通过实验，当你实际进行训练时，你几乎感受不到任何区别。

所以，我喜欢TF32的原因是，如果你能容忍一些精度上的偏差，那么这实际上是免费的。你的代码不会察觉到这一点，这完全是操作内部的事情，这个操作可以让你速度提高8倍。虽然它是更近似的，但在优化中这是一个非常合适的“甜点”位置。嗯，让我们先看看这会是什么样子。所以，我已经设置好我们的代码来计时每一轮训练。我导入了时间模块，并调整了超参数，以便我们能够运行一个更符合我们需求的工作负载。因为最终我们想要进行一次相对较大的训练，所以我们将批量大小设置为16，并使用实际的GPT-2最大序列长度1024个token。这个就是配置，然后进行50轮迭代。

我在这里做的其实是一个非常懒的做法：我只是调用了时间模块来获取当前时间，然后这是优化循环的代码。现在，我想要计时这个过程需要多长时间。与GPU一起工作时的一个问题是，当你的CPU运行时，它只是调度GPU的工作，它正在排定一些任务，对吧？它发送请求，然后继续运行。有时，它可能会非常快地将很多内核任务排队到GPU上，然后CPU就像是停下来等待，但实际上GPU仍然在运行，因为它需要时间来处理那些被调度的任务。因此，你只是为GPU建立了一个队列。如果你需要等待，你想要同步一下，这样可以确保GPU完成所有排定的任务，然后再进行计时。所以基本上，我们在等待GPU完成这一轮的计算，计时后，我们就可以打印结果了。

所以，我在这里运行了训练循环，在右边的Nvidia SMI界面上观察。当我们开始时，GPU的使用率为零，我们没有使用GPU。然后，默认情况下，P会使用GPU 0，所以我们可以看到它开始被填充，使用了35GB的内存（80GB可用）。然后在左边，我们看到，因为我们增加了批量大小，现在一个小的Shakespeare训练数据集只需要20个批次就能完成一次epoch，我们看到每一轮的时间大约是1000毫秒。对吧？第一次迭代有时会比较慢，这是因为PyTorch可能会在第一次迭代时做很多初始化操作，所以它可能会初始化所有张量和缓冲区来存储梯度。我不完全确定在这里发生了什么所有的工作，但当你计时时，你总是需要小心这一点。不过基本上，我们看到每次迭代的时间大约是1000毫秒。

所以现在我们可以运行大约50秒的时间，作为我们当前的浮点32基准。我还想提到一件事，如果你的GPU内存不够，出现内存溢出错误，那就开始减少批量大小，直到适应为止。所以，改成8或者4，或者任何能够适应GPU的批量大小。如果你有更大的GPU，实

际上可以尝试32或更大。默认情况下，你应该尽量最大化适合你GPU的批量大小，并保持数值是整齐的。
所以使用2的幂次方的数字，比如16、8、24、32或者48。这些是很合适的数字，但不要使用像17这样的数字，因为它在GPU上运行效率会很低。我们稍后会看到这一点。那么现在，让我们继续使用16、24这些数字。我还添加了一个功能，并且再次运行了代码，计算了训练过程中的每秒token吞吐量，因为我们可能会随着时间改变批量大小。但每秒token吞吐量是我们真正关心的目标指标。我们在训练中处理了每秒大约163,000个token，这样的吞吐量是一个更客观的度量。好，现在让我们启用TF32。

幸运的是，PyTorch使得启用TF32变得相当容易，要启用TF32，你只需要添加一行代码。就是这行。当我们查看PyTorch文档时，这个函数基本上是告诉PyTorch运行什么类型的内核，默认情况下，我相信它是设置为矩阵乘法的最高精度。也就是说，之前所有的计算都是在浮点32（FP32）下进行的。但是如果我们将其设置为high（高），如我们现在所做，矩阵乘法将会在TF32可用的情况下使用TensorFlow 32。我的GPU是A100，所以它是安培（Ampere）系列，因此TF32是可用的。如果你使用的是旧GPU，这个选项可能不可用，但对我的GPU来说，它是可用的。

所以我期望PyTorch做的事情是，在我们看到的每一个`nn.Linear`层里面，都会执行一个矩阵乘法，我预计这些矩阵乘法现在将会使用TensorFlow，利用TF32加速。这是我认为唯一需要改变的地方，让我们再次运行它。现在，我们看到的是，吞吐量大约应该提升8倍。那么，实际发生了什么呢？我们看到我们的吞吐量大约是3倍，而不是8倍。我们从1,000毫秒减少到300毫秒，吞吐量现在大约是50,000个token每秒。

所以，我们获得了大约3倍的加速，而不是8倍。发生了什么？基本上，发生的情况是，这些工作负载实际上是内存受限的。即使TF32在原理上提供了更快的吞吐量，所有的数值仍然是浮点32（FP32）。这些FP32的数值在内存系统中被频繁地传输，导致我们在数据传输上浪费了大量时间。因此，即使我们使得矩阵乘法本身变得更快，但由于我们受限于内存，我们并没有看到这种加速所带来的全部好处。从这个角度看，虽然吞吐量提高了大约3倍，但这是免费的，一个PyTorch中的单行代码就能实现。所有的变量仍然是FP32，只是运行得更快，结果略微更近似，但我们基本上不会注意到这一点。所以，这就是TF32的效果。

好，接下来我们继续。我们已经练习了这一行代码，看到我们可以在操作内部裁剪掉一些精度。但我们也看到，我们仍然受限于内存，仍然需要传输所有这些浮点数。我们为此付出了代价。所以接下来，我们将减少我们需要传输的数据量，通过使用bfloat16。我们将只保留每个浮点数16位，并且使用bfloat16格式。我稍后会解释FP16和bfloat16的区别。

我们回到A100的文档，查看可用的精度选项。这里是原始的FP32。TF32裁剪了精度，而bfloat16则类似于TF32，但它更加激进地裁剪了浮点数的尾数。因此，bfloat16的一个重要

特点是，指数位和符号位当然保持不变。如果你熟悉浮点数，应该知道，指数决定了你能表示数字的范围，而精度决定了数字的精细度。bfloat16保持相同的指数范围，但由于尾数被裁剪，我们在这个范围内的精度较低。这意味着，bfloat16实际上非常好，因为我们拥有浮点数的原始表示范围，但精度较低。而FP16则不同，它实际上改变了范围，因此FP16无法表示FP32的完整范围，它有一个减少的范围，这会导致一些问题，比如需要使用梯度缩放器等工具。我不会在这个视频里详细讨论，因为那是另一个完整的视频内容。

bfloat16实际上是最早出现的，早在安培架构之前的Volta架构就已经有了，所以FP16是后来才被推出的，大家都开始使用FP16进行训练。然而，使用FP16时，大家必须处理那些不太方便的梯度缩放操作，这会带来额外的状态和复杂性。原因是FP16的指数范围被缩小了。所以这是IEEE的FP16规格。然后，Ampere架构发布了bfloat16，它简化了很多，因为我们只是裁剪掉尾数，保持了相同的指数范围，因此不需要使用梯度缩放器，一切变得更加简单。

然而，当我们使用bfloat16时，它确实会影响我们在PyTorch代码中看到的数字。这个变化不仅仅局限于操作本身。所以我们来看看这会如何工作。

这里有一些文档可以解释它。我觉得这可能是解释如何在PyTorch中使用混合精度的最佳页面，因为其他很多教程，甚至PyTorch文档中的一些，都会更让人困惑，所以我特别推荐这个页面。因为有五个其他版本，我不推荐。当我们来到这里时，忽略所有关于梯度缩放器的内容，只看Torch。AutoCast基本上是通过一行代码来实现的，它就是我们想要的上下文管理器，我们希望在网络中使用它，当你点击进入Torch时。AutoCast自动转换有一些额外的指南，它告诉你不要在任何张量上调用B float 16，只需使用AutoCast并仅包围模型的前向传递和损失计算。这是你应该包围的唯一两个部分。保持反向传播和优化步骤不变。这是PyTorch团队提供的指导意见。

我们将遵循这个指导，并且对于我们来说，因为损失计算是在模型的前向传递内，所以我们将这样做。我们不想使用torch float 16，因为如果我们这样做，就需要开始使用梯度缩放器。所以，我们将使用B float 16。这只能在AMP中实现。这意味着变化非常小，基本上只有这一行代码。让我在实际运行之前先中断一下。

所以，在logits之后，我想给你们展示一下，不同于我们之前看到的tf32，这实际上会影响我们的张量。现在，我们来看一下Lis张量，如果我们查看dtype，我们会突然看到它现在是B float 16了。它不再是float 32了，所以我们的激活值已经改变。激活张量现在是B float 16，但并不是所有的东西都改变了。比如模型的Transformer，WTE（权重标记嵌入表），它里面有一个权重，那个权重的dtype依然是torch float 32。所以，我们的参数似乎仍然是float 32，但我们的激活值——logits——现在是B float 16了。

显然，这就是我们得到混合精度的原因；PyTorch有些东西保持为float 32，有些东西则转换为更低精度的格式。什么东西在什么时候转换并不完全明确。我记得我滚动下去，嗯，找不到我想找的地方。好吧，找到了。这里有一些文档，讲述了当使用AutoCast时，什么会被转换为BFLOAT16，什么时候会转换。例如，只有像矩阵乘法这样的操作会被转换为float16，但很多操作仍然保持在float32中，特别是很多规范化操作，比如层归一化等，可能并不会被转换。因此，只有部分层会选择性地以BFLOAT16运行。但像softmax、层归一化、log softmax和损失函数计算等操作，很多可能仍会保持为float32，因为它们对精度的变化更为敏感。主要的矩阵乘法对精度变化比较鲁棒，因此网络的某些部分对精度变化的影响较小或较大。所以，基本上，只有模型的部分部分在运行时使用了降低精度。

让我们试试看，看看我们取得了什么样的改进。

好，现在我们曾经是333毫秒，现在是300毫秒；我们曾经每秒大约处理50,000个tokens，现在是55,000。所以，我们肯定运行得更快了，但可能没有比之前更快多少。这是因为在我们的GPT-2中仍然有许多瓶颈。我们才刚刚开始，但我们已经将精度降低到我当前的GPU（A100）能够支持的最远了。我们使用了PyTorch的AutoCast。不幸的是，我并不确切知道PyTorch AutoCast做了什么，或者说它究竟在BFloat16和Float32之间做了哪些转换。

我们可以进入并开始仔细检查，但这些规则是PyTorch内部的，遗憾的是它们没有很好地文档化。所以，我们不会详细探讨这些问题，但现在，我们已经在BFloat16中训练了，并且不需要梯度缩放器。之所以运行得更快，是因为我们能够在BFloat16上运行tensor cores。

这意味着我们走上了这条路，但我们也在精度上付出了一些代价。因此，我们预计结果相较于原始的FP32会略有不准确。但是从经验来看，在许多情况下，这种折中是值得的，因为它使得训练速度加快了。比如，你可以训练得更久，以弥补精度的损失。所以，这就是BFloat16的现状。

好，现在我们可以看到，目前每次迭代大约是300毫秒，我们接下来将尝试使用PyTorch工具箱中的一些真正强大的武器。特别是，我们将引入torch.compile。

torch.compile是PyTorch团队非常强大的基础设施，基本上是神经网络的编译器。它几乎就像C和C++代码的GCC一样。它在之前就发布了，而且使用起来非常简单。使用torch.compile的方法很简单，只需一行代码即可编译你的模型并返回它。现在，这行代码会花费你一些编译时间，但如你所料，它会让代码运行得更快。让我们实际运行一下看看，因为这将需要一些时间来执行。现在，记住我们目前是300毫秒，我们来看看会发生什么。

在这个过程中，我想稍微解释一下torch.compile到底是怎么工作的。请随意阅读PyTorch的这页文档，但基本上，如果你没有理由不使用torch.compile，我觉得你应该几乎默认使用它，除非你在调试，并且想让代码运行得更快。我发现torch.compile中有一行代码，实际上揭示了它为什么更快。加速主要来自于减少Python的开销和GPU的读写操作。让我稍微解释一下。

好了，我们现在从300毫秒变到了129毫秒。所以，这大约是通过PyTorch中一行代码实现的2.3倍的加速，真是令人难以置信！

那么，究竟发生了什么？发生了什么呢？当你将模型传递给torch.compile时，在这个NN模块中，我们实际上只是定义了网络中希望发生的算法描述。torch.compile会分析整个模型，并查看你想要使用的操作。通过完全了解将要发生什么，它就不需要在所谓的“急切模式”下运行，也不需要像Python解释器那样一层一层地执行。

在前向传播中，Python解释器会说，“好吧，我们先做这个操作”，然后再做那个操作。它会在执行过程中逐步显现所有的操作。这些计算按顺序调度并执行，而Python解释器和代码并不知道接下来会发生什么样的操作，但Torch compile会同时看到整个代码。它能够知道你打算运行什么操作，并且会对这个过程进行优化。

它首先会做的事情是完全移除Python解释器，直接编译整个神经网络，将其作为一个单独的对象，而不涉及Python解释器，这样它就能确切知道将要运行的内容。我们就运行这个，它将全部以高效的代码执行。

接下来发生的事情是读写操作，这点他们简短提到过。我认为一个很好的例子就是我们一直在看的G非线性。这里，我们使用了n和G。现在，这就是我，基本上在拆解inang Galu，你记得它有这个公式，所以这里的实现与算法层内部发生的完全一致。

它是完全相同的。现在，默认情况下，如果我们使用这个而不是结束G操作，那么如果没有Torch compile会发生什么？嗯，Python解释器会来到这里，然后说，好的，这里有一个输入，首先让我把这个输入的三次方计算出来，然后它会调度一个内核来处理这个输入，将其三次方计算出来，这个内核就会运行。当这个内核运行时，最终发生的情况是，这个输入会被存储到GPU的内存中。

这是一个帮助理解的例子，展示了发生了什么。你有你的 CPU，每台计算机都有。它有几个核心，还有内存（RAM），CPU 可以与内存通信，这些都是众所周知的。但现在我们加入了 GPU，GPU 是一个稍微不同的架构。当然，它们可以通信，不同之处在于 GPU 拥有比 CPU 更多的核心。所有这些核心的结构也更简单，但它也有内存。
对，这种高带宽内存——抱歉，如果我说错了，HBM，我甚至不知道它代表什么，突然意识到这一点，但这是内存，基本上相当于计算机中的 RAM。发生的事情是，输入数据存储在内存中，当你做输入的三次方运算时，这些数据必须传送到 GPU，传送到各个核

心，以及 GPU 上的缓存和寄存器。它必须计算所有的元素的三次方，然后把结果保存回内存。实际上，正是这个传输时间引起了很多问题。所以记住这一点——内存带宽，我们可以通信大约 2 TB 每秒，这已经很多了。但我们还需要通过这个链接，而这个链接非常慢。所以在 GPU 上，我们在芯片内部，一切都非常快，但访问内存非常昂贵，耗时极长。所以我们加载输入，进行计算，再将输出加载回来。这一往返的过程非常耗时，接着我们做乘以常数的操作。
然后发生的是，我们调度了另一个内核，结果又返回。所有的元素都被乘以常数，然后结果返回到内存。然后我们再将结果与输入相加。所以整个过程又一次地从内存传送到 GPU，进行相加操作，再写回。
所以我们做了很多次从内存到计算发生地（GPU 芯片内）的往返，因为所有的张量核心和算术逻辑单元（ALU）都存储在 GPU 的芯片上。所以我们做了很多次往返，而 PyTorch 如果不使用 Torch Compile，它并不知道如何优化这一点，因为它不知道你运行的是什么样的操作。后来，你只告诉它先做三次方，再做这个，再做那个，它就按这个顺序去执行。但 Torch Compile 会看到你整个代码，它会来判断：“等一下！所有这些都是元素级的操作。”实际上，我要做的是只将输入传送一次到 GPU，然后针对每个元素，我会在 GPU 上执行所有这些操作，或者说是对它的某些块进行操作，然后再写回。所以我们只做一次传输，而不会有这些往返操作。这就是所谓的内核融合，它是加速的主要方式之一。所以基本上，如果你清楚自己要计算的内容，就可以优化内存的往返传输，从而避免内存带宽的成本。这是为什么有些操作会变得更快的根本原因。
所谓的读写操作，意思是，我来擦掉这些，因为我们不再使用它们。是的，我们应该使用 torch compile，现在我们的代码显著加速了，我们每秒处理大约 12.5 万个标记，但我们仍然有很长的路要走。
我想补充一些讨论，给大家提供一些更多的数字，因为这是一个复杂的话题，但从高层理解它是值得的。这里发生的事情，我可能可以花一个两小时的视频来讲解，但先简单介绍一下。基本上，这个芯片是 GPU，这个芯片是所有计算主要发生的地方。但这个芯片也有一些内存在里面，但绝大多数内存是在外部的高带宽内存（HBM）中，它们是连接的，虽然是两个不同的芯片。
这是 GPU 的缩放图，首先你看到的是这个 HBM，我意识到这可能对你来说很小，但它两侧标有 HBM，这就是与 HBM 的连接。HBM，仍然是外部内存。在芯片上，有大量的流式多处理器（SMs）。每个 SM 都是一个处理单元，总共有 120 个，这是很多计算发生的地方。这是一个 SM 的放大图，它有四个象限。举个例子，张量核心是矩阵乘法发生的地方，但这里有很多其他的单元，用来进行 FP64、FP32 和整数计算等。
那么我们有所有这些逻辑来进行计算，除了这些，在芯片上还有内存分布。比如 L2 缓存就是存在芯片上的一部分内存，而在 SM 上还有 L1 缓存。我意识到这对你来说可能非常小，但这蓝色条就是 L1 缓存，也有寄存器。因此，内存存储在芯片内，但这种存储方式与 HBM 存储方式是非常不同的。这是两种完全不同的实现，关于硅芯片的设计和内存结构。这一部分使用的是 SRAM，而 HBM 则是完全不同的实现。
简而言之，芯片内是有内存的，但它并不是很多。这个关键点是，虽然芯片内的内存非常

快速，但总的来说，数量是有限的，无法与 HBM 相提并论。
这是另一个不同 GPU 的示意图。这里展示了典型的 CPU DRAM 内存，可能只有一块内存条，而访问它非常昂贵，特别是对于 GPU，必须通过 CPU 来访问。而 HBM 内存通常有几十 GB，但正如我之前提到的，访问起来非常昂贵。
总体来说，芯片内非常快速的计算，但内存非常少，只有几十 MB 的内存，存储在整个芯片的不同地方。这些内存非常昂贵，所以容量很小，但相对来说，它们访问起来极快。
所以基本上，每次执行内核操作时，我们先从全局内存中取出输入数据，接着我们在芯片上执行计算，然后将结果流回内存进行存储。如果我们不使用 Torch Compile, 我们正在将数据流经芯片，进行计算并保存到内存中，并且我们会进行多次来回的循环。但是，如果使用了Torch Compile，我们会像之前一样开始流动内存，但在芯片上，我们会有一块数据正在被处理。因此，在芯片上进行操作是非常快速的。如果我们有内核融合（kernel fusion），我们可以在这里对所有操作进行逐元素的计算，这些操作非常便宜。然后，我们只需进行一次回程，将结果写回全局内存。所以，操作符融合基本上让你能够将数据块保持在芯片上，进行大量计算，然后再写回，这样可以节省大量时间。这就是为什么Torch Compile通常会更快的原因之一；这是主要原因之一。所以，这只是一个关于内存层次结构以及Torch Compile为你做了什么的简单介绍。

现在，Torch Compile非常强大，但有些操作是Torch Compile无法发现的，一个非常好的例子就是Flash Attention，接下来我们将讨论它。Flash Attention来自2022年斯坦福大学的一篇论文，是一种非常强大的注意力算法，使得运行速度大大提高。所以，Flash Attention会来到这里，我们将去掉这四行代码。Flash Attention将这四行代码实现得非常、非常快。它是如何做到的呢？Flash Attention是一个内核融合操作。

在这个图表中，我们看到PyTorch，它包含这四个操作，包括Dropout，但我们这里没有使用Dropout，所以我们只保留这四行代码，而这些代码将被融合成一个单一的Flash Attention内核。所以它是一个内核融合算法，但这是一个Torch Compile无法发现的内核融合。而Torch Compile无法发现的原因是，它需要对注意力算法进行算法重写。在这个案例中，Flash Attention非常值得注意的一点是，尽管Flash Attention执行的浮点操作（flops）数量比常规的注意力要多，但Flash Attention实际上快得多。事实上，他们提到它可能会快7.6倍，这归功于它非常重视内存层次结构，正如我刚才所描述的。它非常注重高带宽内存中的数据、共享内存中的数据，并且非常小心地协调读取和写入高带宽内存，因此即使执行更多的浮点操作，昂贵的部分是加载和存储到HBM，而这些操作是它避免的。特别是，它永远不会将这个端到端的注意力矩阵具体化。Flash Attention的设计使得这个矩阵在任何时候都不会被具体化，也不会读取或写入到HBM。这是一个非常大的矩阵，对吧？因为这是所有查询和键相互作用的地方。我们基本上会得到每个头部、每个批次元素的T×T矩阵，注意力矩阵，这包含一百万个数字。即使是单个头部和单个批次索引，它也是一个巨大的内存占用，这个矩阵从未被具体化。

它是如何实现的呢？基本上，这个算法的重写依赖于一个在线Softmax技巧，这是之前一篇论文中提出的，我稍后会展示那篇论文。在线Softmax技巧展示了如何在不必一次性处理所有输入的情况下，增量地计算Softmax，从而实现归一化。通过使用这些中间变量M和L，它们的更新允许你以在线的方式计算Softmax。Flash Attention实际上，最近也发布了Flash Attention 2，它进一步提高了Flash Attention的计算效率。最初的Flash Attention论文基于的是在线归一化计算Softmax的方式，值得注意的是，这篇论文来自Nvidia，并且在2018年早早发布，这比Flash Attention早了四年。那篇论文中提出了如何减少内存访问来计算传统的Softmax，并假设这种减少内存访问的方式应该能提高实际硬件上的Softmax性能。事实上，他们在这个假设中非常正确，但令人着迷的是，他们来自Nvidia，早早就意识到了这一点，但直到四年后，Flash Attention才由斯坦福团队真正实现。所以，我并不完全理解这一历史背景，但他们基本上提出了对Softmax的在线更新，而这正是Flash Attention所依赖的技术。

Flash Attention将所有其他操作与在线Softmax计算融合成一个单一的内核，这就是我们将要使用的技术。所以，这是一个非常好的例子，展示了如何重视内存层次结构。浮点操作的数量并不重要，整个内存访问模式才是关键，而Torch Compile虽然非常强大，但仍然有很多优化是它无法发现的。也许有一天它能做到，但目前来看，这确实是一个很大的挑战。

接下来，我们将使用Flash Attention，基本上在PyTorch中，我们将注释掉这四行代码，然后将它们替换为一行。这行代码会调用PyTorch中的一个复合操作，称为“Scale that Product Attention”。使用这种方式时，PyTorch会调用Flash Attention。我并不完全理解为什么Torch Compile不能自动识别这四行代码应该调用Flash Attention，我们仍然需要为它做一些工作，这在我看来有些奇怪，但事情就是这样。所以，你需要使用这个复合操作，让我们等一会儿，看看Torch Compile什么时候完成。接下来，让我们记住，我们得到了6.05的损失，这是我们期望看到的结果，且这个更改之前花费了130毫秒。我们期望在第49次迭代时看到完全相同的结果，但预计会有更快的运行时间，因为Flash Attention只是一个算法重写，它是一个更快的内核，但它并没有改变计算本身，我们应该会看到相同的优化结果。

好了，我们确实快得多了，时间从130毫秒减少到约95毫秒，我们的结果是6.58，基本上与预期相同，仅仅有一个浮动的误差。所以，计算是相同的，但运行速度显著提升，从130毫秒到大约96毫秒。也就是说，时间从130毫秒降低到96毫秒，约提升了27%。这真的很有趣。

这就是Flash Attention。好了，现在我们来到了我最喜欢的优化之一，它既是最傻的，也是最聪明的优化。每次看到这个优化时，我总是觉得有点惊讶。好吧，基本上，我几分钟前提到过，一些数字是美丽的，而一些数字是丑陋的。比如64是个美丽的数字，128更美，256也是个美丽的数字。之所以这些数字美丽，是因为它们里面包含很多2的幂次，你

可以将它们不断地除以2。而丑陋的数字比如13、17之类的质数，或者其他非偶数的数字。

基本上，你总是希望在处理神经网络或CUDA的代码中使用"漂亮"的数字，因为CUDA中的一切都是基于2的幂来工作的，许多内核都是以2的幂为基础编写的。很多块大小是16、64等等，所以一切都是用这些数字编写的。当输入不是这些"漂亮"的数字时，代码里通常会有很多特殊情况的处理。我们来看一下这些是什么样的情况。

基本上，你需要扫描代码，找出"丑陋"的数字。大致来说，三的倍数是丑陋的。嗯，我不确定，这个可能可以优化，但这确实是丑陋的，不是理想的。四的倍数是漂亮的，这是好的。1024是非常漂亮的，因为它是2的幂。12有点可疑，不是很理想。768很棒。50257是一个非常丑陋的数字。首先，它是奇数，所以没有太多2的幂。所以，这真的是一个非常丑陋的数字，值得怀疑。然后，往下滚动，其他数字都很好。然后这里我们大部分是漂亮的数字，除了25。在GPT-2 XL的这个配置中，头的数量是25。那是一个非常丑陋的数字，它是一个奇数，实际上，这最近确实给我们带来了很多麻烦，特别是当我们尝试优化一些内核，以便更快运行时。它需要很多特殊情况的处理。所以，基本上这些数字有一些丑陋的数字，有些比其他的更容易修复。特别是词汇表的大小为50257，这是一个非常丑陋的数字，值得怀疑，我们希望修复它。

当你修复这些问题时，一种简单的方法是将数字增加，直到它变成你喜欢的最接近的2的幂。例如，这里有一个更漂亮的数字：50,304。为什么呢？因为50,304可以被8、16、32、64甚至128整除。我认为它是一个非常漂亮的数字。那么，我们在这里做的事情是设置GPT的配置，你看到我们初始化了B大小为50,257。让我们只覆盖这个元素，将其设置为50,304。好吧，其他一切保持不变；我们只是增加了我们的词汇表大小，所以我们几乎是在添加虚假的token。这样，词汇表的大小就包含了2的幂。

实际上，我要做的是增加网络将要执行的计算量。如果你只计算一下浮点操作数，做一下数学运算，算一下我们要执行多少浮点运算，我们会做更多的浮点操作，我们仍然需要考虑这是否会破坏任何东西。但如果我现在就运行它，我们来看看结果如何。目前，这个过程大约需要96.5毫秒每步。我大致估算一下，看看结果如何。在这个编译的过程中，我们来考虑下我们的代码是否真正有效。

好吧，当我们像这样增加词汇表大小时，让我们看看词汇表实际在哪些地方被使用。我们跳到代码中，看它被用在嵌入表中，当然，在Transformer的底部。它也被用在分类层中，Transformer的顶部有两个地方。我们来看一下，现在运行时间是93毫秒，而不是96.5毫秒。通过执行更多的计算，我们大约提高了4%的效率。原因是我们修复了，将一个丑陋的数字变成了漂亮的数字。

稍后我会解释原因，但现在让我们先相信我们在做这件事时不会破坏任何东西。首先，我们增加了嵌入表的大小；就像我们在底部引入了更多的token，而这些token从未被使用，因为GPT的tokenizer只包含最多50,257个token。所以我们永远不会索引到我们添加的那些行；因此，我们创建了从未访问过的内存，浪费了一点空间。这部分其实不完全正确，因为WTE权重最终会被共享，并且会在分类器中使用。那么，这对分类器有什么影响呢？它的影响是我们现在在分类器中预测更多的维度，并且我们预测了那些永远不会出现在训练集中的token的概率。因此，网络必须学习如何将这些概率驱动到零。所以，网络生成的logits必须将这些输出维度推向负无穷大。但这与数据集中已经没有的其他token没有区别。所以，莎士比亚的作品可能只用了大约3000个token，而不是50,257个token，大部分token通过我们刚才引入的优化已经被推向零概率。现在，多了几个token，按照类似的方式，它们永远不会被使用，需要将其概率推向零。

但从功能上讲，这一切都没有破坏。我们使用了更多的额外内存，但除此之外，按照我的看法，这是一个无害的操作。但我们增加了计算量，运行得更快了，而它之所以更快，是因为正如我在CUDA中提到的，很多内核使用的是块大小，这些块大小通常是漂亮的数字，是2的幂。计算是以64块或32块的方式进行的，当你想进行的计算无法恰好适应这些块时，就会触发额外的内核来处理剩余的部分。所以，基本上，很多内核会先处理输入的漂亮部分，然后再回到剩余部分进行处理，这个过程会非常低效。

启动这些额外的计算是非常低效的，所以你最好为输入数据添加填充，让它能更好地适应。通常情况下，这样的经验性做法实际上会让运行变得更快。这是另一个增加了4%性能的例子，而这也是Torch Compile无法为我们发现的优化。你可能希望Torch Compile在某个时刻能够识别出像这样的优化，但目前它还无法做到这一点。

我还必须指出，我们使用的是PyTorch的夜间版本，因此我们只看到4%的提升。如果你使用的是PyTorch 2.3.1或更早版本，你可能会看到30%的提升，光是这个改动就能带来这么多的改善。这个变化是从50,000改到50,304。所以，再一次，这是我最喜欢的例子之一，关于必须了解底层机制以及它是如何工作的。你需要知道哪些方面可以进行调整，以推动代码的性能。

到目前为止，我们已经把性能提高了大约11倍，对吧？因为我们从每步大约1000毫秒降到了93毫秒左右。效果非常好，我们更好地利用了我们的GPU资源。接下来，我将转向算法上的改进，进一步优化实际的计算过程。

我们想做的是遵循GPT-2或GPT-3论文中提到的超参数。不幸的是，GPT-2实际上并没有说太多。虽然他们很慷慨地发布了模型权重和代码，但论文本身对于优化细节非常模糊。它们发布的代码也是我们所查看的推理代码，并没有包含训练代码，也没有太多的超参数。所以，这对我们没有太多帮助。为了获得更多的帮助，我们必须转向GPT-3的论文。在GPT-3论文中，他们给了我们更多的超参数，而且总体而言，GPT-3论文在模型训练的各

种细节方面更加详细。然而，GPT-3的模型从未发布，所以对于GPT-2我们有权重但没有细节，对于GPT-3我们有细节但没有权重。大致来说，GPT-2和GPT-3的架构非常相似。

变化非常少。上下文长度从1024扩展到2048，这算是主要的变化。嗯，Transformer的一些超参数发生了变化，但除此之外，它们基本上还是相同的模型。只是GPT-3在更大的数据集上训练了更长时间，而且进行了更多的全面评估。而且，GPT-3模型的参数量是1750亿，而GPT-2只有16亿。嗯，简而言之，我们将参考GPT-3的论文来跟随一些超参数设置。所以，为了训练所有版本的GPT-3，我们使用Adam优化器，β1和β2分别为0.9和0.95。那么，我们过来确保β参数（在这里可以看到，默认是0.9和0.999）实际上设置为0.9和0.95。然后是epsilon参数，您可以看到默认是1e-8，这里也是1e-8。我们将其调整为这样，就可以了。

接下来，他们提到我们会将梯度的全局范数剪切到1.0。这里指的是，一旦我们计算出梯度，也就是在`.backward()`之后，我们实际上拥有了所有参数张量的梯度。人们通常做的事情是将它们剪切到某个最大范数。在PyTorch中，这非常容易做到；在计算完梯度之后，只需插入一行代码。这个工具函数的作用是计算参数的全局范数。也就是说，对于所有参数的每个梯度，首先将它们平方，累加起来，再取平方根。这就是参数向量的范数，基本上就是它的长度，您可以这么理解。我们要确保它的长度不超过1.0，并进行剪切。

人们喜欢使用这个方法，是因为有时候优化过程中可能会运气不佳，可能是由于一个坏的数据批次之类的原因。如果批次中有异常，可能会导致损失值很高，而非常高的损失值可能会导致非常高的梯度，从而使模型和优化过程遭到“震荡”。因此，大家通常使用梯度范数剪切来防止模型在梯度的幅度上受到过大的冲击。这有点像是一个补丁，解决更深层次的问题，但人们仍然相当频繁地使用它。

现在，`clip_grad_norm`返回的是梯度的范数，我总是喜欢把它可视化，因为它是有用的信息。有时候，你可以通过查看梯度的范数，如果它表现得很好，那么一切正常。如果它不断增大，那么说明情况不太好，训练过程不稳定。有时，如果范数突然飙升，可能意味着出现了某种问题或不稳定性。因此，这里的范数就是一个可以打印出来的范数。现在，让我们看看学习率调度器的细节。在这里，GPT-3使用的学习率调度器并不是固定的学习率（比如我们这里使用的3e-4），而是采用了余弦退火学习率调度。它有一个预热阶段，并且学习率以余弦形式逐渐降低，直到10%。

嗯，现在我们就来实现它。我已经在这里实现了，下面是如何工作的。首先，我稍微修改了一下训练循环，将`max steps`改成了`step`，以便让我们能够明确每一步的优化过程。接着，在这里我用一个新函数`get_learning_rate`来获取当前优化步骤的学习率。在PyTorch中，设置学习率的方法稍微有点复杂，因为你需要针对优化器中的不同参

数组进行迭代。即使我们现在只有一个参数组，也必须按照这种方式设置学习率，并在循环中完成。之后，我会打印出学习率。

这就是我对训练循环所做的所有更改。然后，当然，`get_learning_rate`就是我们的调度器。值得一提的是，PyTorch实际上提供了学习率调度器，你也可以使用它们。我相信PyTorch中有一个余弦学习率调度器，但我个人不太喜欢使用那段代码，因为老实说，它只有五行代码，而我完全理解这些代码在做什么，所以我不喜欢使用那些不太清楚其内部实现的抽象，个人风格吧。

所以，最大学习率设置为3e-4，但我们会看到在GPT-3的论文中，对于每种模型大小，最大学习率的值都有表格。例如，对于12层768的GPT-3模型，也就是GPT-3小型版本，它类似于GPT-2 124M模型。这里他们使用的学习率是6e-4，实际上我们可能想要尝试跟随这个设置，将最大学习率设置为6e-4。最小学习率是最大学习率的10%，这是论文中的描述。

接下来是预热步骤数，然后是最大优化步数，我现在也在下方的循环中使用了这些。你可以浏览一下这段代码，它其实不算太有趣。我只是根据迭代次数调整学习率，分为预热阶段、优化后阶段以及两者之间的阶段。这里是计算余弦学习率调度的地方。如果你愿意，可以详细查看这段代码，但这基本上是在实现这条曲线。我已经跑过这段代码，结果如下。当我们现在运行时，学习率从一个非常小的数字开始。请注意，我们并不会从零开始，因为学习率为零时不会进行有效的更新，因此我们在零迭代时不会使用完全为零的学习率，这就是为什么我们在零步时会设置`it+1`。

我们使用的是非常非常小的学习率。然后我们线性地将学习率提升到最大值，在我运行时，最大值为34，但现在应该是6E4，接着它开始衰减，一直到3E5，这个值是原始学习率的10%。现在有一点我们并没有完全按照原文的做法来实现，他们提到——让我看看能不能找到。我们并没有完全跟随他们的步骤，因为他们提到他们的训练总量是3000亿个token，他们在2600亿token时将学习率降低到原始学习率的10%，然后在2600亿之后继续训练，学习率保持在10%。所以，他们的衰减时间比最大步数的时间要短，而我们这里是两者完全相等。所以，虽然没有完全忠实地跟随，但这对我们来说是可以接受的。

对我们目前的目的来说，这是没问题的。我认为这不会产生太大的差异。我要指出的是，选择使用什么样的学习率调度完全取决于你自己；现在有很多不同类型的学习率调度。学习率已经通过GPT-2和GPT-3变得非常流行，但人们已经提出了各种不同的学习率调度方法。这实际上是一个活跃的研究领域，大家在探讨哪种方法最有效地训练这些网络。接下来，论文提到逐步增加批量大小。所以批量大小是线性增加的，开始时批量很小，随着时间的推移增加到较大的批量大小。然而，我们实际上跳过了这个步骤，不打算使用它。我不喜欢使用它的原因是，它会使得计算变得更加复杂，因为你在每个优化步骤中都在改变

处理的token数量。我喜欢保持这些数学计算非常简单。此外，我的理解是，这并不是一个重大的改进。

另外，我的理解是，这更多的是一种系统优化和速度改进，而不是算法优化的改进。大致来说，这是因为在优化的早期阶段，模型处于一个非常非典型的状态，主要学习的是忽略那些在训练集中不常出现的token，你在学习一些简单的偏差等。因此，你放入网络的每一个例子基本上都在告诉你使用这些token，而忽略那些不常出现的token。所以在优化的初期，所有的梯度都是高度相关的，差不多都长得一样，因为它们只是告诉你哪些token不出现，哪些token会出现。因此，在这个阶段，你为什么要用百万级的批量大小呢？如果你使用32k的批量大小，早期训练时你基本上会得到完全相同的梯度；而在优化的后期，一旦你学习了所有简单的东西，才是实际的训练工作开始的地方。那时，梯度会变得更加去相关，这才是它们真正为你提供统计能力的地方。

我们将跳过这个步骤，因为它有点复杂，而且我们已经在进行数据采样时采用了无放回的方式。也就是说，在训练过程中，直到达到一个epoch的边界。无放回的意思是，它们不是从一个固定的池中进行采样，然后拿出一个序列进行训练，再把它放回池中。而是消耗池中的数据，所以当它们抽取一个序列时，这个序列就不会再使用，直到下一个epoch开始。所以我们已经在做这件事，因为我们的数据加载器是以数据块的方式进行迭代的，因此没有放回操作。数据在当前的epoch中不会再被抽取，直到下一个epoch。因此，我们基本上已经在这样操作了。

所有模型都使用0.1的权重衰减，以提供少量的正则化。所以让我们来实现一个权重衰减。你可以看到，我已经做了一些修改，特别是，在这里我并没有直接创建优化器，而是创建了一个新的配置优化器函数，放在模型内部，我将一些超参数传递给它。现在让我们来看一下这个configure optimizers函数，它应该返回一个优化器对象。好了，虽然看起来有些复杂，但其实非常简单，我们只是非常小心地处理它。

在这个函数中，有一些设置需要注意。最重要的事宜是，你可以看到有一个权重衰减的参数，我将它传递给优化器组（optim groups），最终进入AdamW优化器。AdamW中默认使用的权重衰减值是0.01，所以它比GPT-3论文中使用的值低10倍。这个权重衰减最终会通过优化器组传递给AdamW。

这个函数里还有两件事很重要，那就是我把参数分成了应该进行权重衰减的部分和不进行权重衰减的部分。通常情况下，我们不会对偏置项以及其他一些一维张量进行权重衰减，像层归一化的尺度和偏置通常是不需要进行权重衰减的。我们主要希望对参与矩阵乘法的权重进行权重衰减，可能还会对嵌入（embeddings）进行衰减。我们在前面的视频中已经讲过，为什么对权重进行衰减是有意义的。你可以把它看作是一种正则化，当你压缩所有权重时，你迫使优化过程使用更多的权重，而不是让某个权重单独变得过大。你实际上是在迫使网络将工作分配到更多的通道上，因为权重本身受到了一种“引力”的影响。所以，

这就是我们为什么要以这种方式分离它们的原因。在这里，我们只对嵌入和主要参与计算的权重进行衰减。

我们打印出我们正在衰减的参数数量，发现大部分参数都会被衰减。然后，另一个我们在这里做的事情是，进行了一次额外的优化，这个优化在之前的AdamW中并没有这个选项，但PyTorch的后续版本添加了它，因此我在这里检查是否存在这个选项，并使用它。这个选项叫做fused，它最早并不存在，后来才加入。它的作用是，如果存在fused选项，我们会使用它，因为它能够加速优化。当你在CUDA上运行时，它通过将多个更新步骤合并为一个内核，从而减少了很多开销，提高了计算效率。

我们可以重新运行这个代码，虽然不太可能看到任何显著的不同，但我们会看到一些新的打印输出。让我们看看这些打印输出的内容。我们看到，衰减的张量数量是50个，其中大部分参数都会被衰减。没有衰减的张量数量是98个，主要是偏置项和层归一化参数，这些数量相对较少，只有10万个，所以大多数参数都会被衰减。然后，我们使用了fused实现的AdamW优化器，它会更快。如果你有这个选项，我建议你使用它。其实我不完全清楚为什么它们没有默认启用fused选项，看起来它是比较温和无害的。另外，因为我们使用了fused实现，所以我们也看到了运行时间的缩短。原来每步的时间是93毫秒，现在减少到了90毫秒，这要归功于使用了fused的AdamW优化器。所以，在这个单一的提交中，我们通过引入fused的AdamW优化器，显著改善了时间性能。

权重衰减，但我们只对二维参数进行权重衰减：即嵌入层和参与线性操作的矩阵。这样就是这样，我们可以把这个去掉，嗯，对，这一行就是这样。继续之前，我再提一下，权重衰减、学习率、批量大小、Adam优化器的β1、β2、ε等参数之间的关系。这些在优化文献中是非常复杂的数学关系。大部分时间里，在这个视频中，我只是想复制OpenAI使用的设置，但这是一个复杂的话题，内容深奥。在这个视频中，我只是想复制这些参数，因为详细讨论这些内容需要另外一个视频，给它应有的深度，而不仅仅是高层次的直观理解。

接下来，我想转到这段文字。顺便说一下，我们将在后面改进数据加载器时再回到这里；现在，我想回到这里。在表格中，你会注意到，对于不同的模型，Transformer的U超参数会有所不同，这决定了Transformer网络的大小。我们还会看到不同的学习率，比较大的网络会使用略低的学习率。我们还看到了批量大小，小网络使用较小的批量大小，而大网络使用较大的批量大小。

问题在于，我们不能仅仅使用50万个批量大小，因为如果我试着在这里设置B，在哪里调用数据来着？哦，b等于……我怎么叫这个b来着？好吧，b等于16。如果我试图设置这个值，我们得小心，它不是50万，因为这是按token计算的批量大小。我们的每一行是24个token，所以50万除以1024，大约需要488的批量大小。

问题是，我不能直接把它设置为488，因为我的GPU肯定撑不住，肯定无法容纳。即便如此，我仍然想使用这个批量大小，因为，正如我之前提到的，批量大小与其他优化超参数、学习率等都有关系。我们希望忠实地表示所有超参数，因此我们需要使用大约50万个批量大小。但是问题是，如何在只有小GPU的情况下使用50万？嗯，为此，我们需要使用梯度累积。接下来，我们将讨论这一点，它能让我们以串行方式模拟任何我们设置的批量大小。所以我们可以使用50万的批量大小，只是得运行更长时间，并且处理多个序列，基本上是把所有的梯度累积起来，从而模拟出50万的批量大小。接下来，我们来实现这一点。

好了，所以我在这里开始了实现，首先我设置了我们希望的总批量大小。这个大小是50万个，使用一个漂亮的数字，是2的幂次方，因为2的19次方是524,288，差不多就是50万了，这是一个好数字。现在，我们的微批量大小，也就是我们现在所说的大小，是16。这个大小将在进入Transformer并进行前向和反向传播时使用，但我们不会立即进行参数更新。对，我们会进行多次前向和反向传播，计算梯度，它们会累积到参数的梯度中。它们会加起来，所以我们会进行多次前向和反向传播的梯度累积步骤，然后在所有的梯度累积完后进行一次参数更新。特别是，我们的微批量大小现在控制着我们在一次前向反向传播中处理的token数量，即每次处理16*124，16乘以384个token，然后我们希望总批量大小是2的19次方，哎呀，我怎么了？2的19次方，嗯，差不多。结果，梯度累积会是32。所以梯度累积在这里是32，我们需要进行32次前向反向传播，然后进行一次更新。

现在，我们看到单次前向反向传播大约需要100毫秒，所以做32次大约需要3秒。大致的计算是这样的，但接下来我们得实现它。所以我们要去我们的训练循环，因为现在这里的前向和反向我们得重复32次，然后再做后续的操作。我们来看看怎么实现。我们过来，实际上我们确实每次都需要加载一个新的批量。把它放到这里，这时就有了内部循环。所以在梯度累积步骤的范围内，我们做这个操作。记住，`l.backward()`总是会存储梯度，所以每次做`l.backward()`时，梯度会累积到梯度张量上。嗯，我们这样做了`loss.backward()`，然后得到了所有的梯度，接下来我们就归一化，其他的操作也该接着进行。嗯，我们非常接近了，但实际上这里有一些微妙而深入的问题，这个其实不正确。我邀请你思考一下为什么它还不够充分。嗯，让我来修正它。

好吧，我把Jupyter notebook拿回来了，这样我们就可以在一个简单的示例中仔细思考，看看发生了什么。所以让我们创建一个非常简单的神经网络，它接收一个16维的向量并返回一个数值。接着，我这里创建了一些随机示例X和目标Y，然后我们使用均方误差来计算损失。所以基本上，这是四个独立的例子，我们对这四个例子做简单的回归，计算均方误差。现在，当我们计算损失时，我们执行`backward()`并查看梯度。这个就是我们得到的梯度。现在，损失目标是，注意在均方误差中，默认的损失函数是求平均值，所以我们计算的是四个例子的均方误差。均方误差在这里是对四个例子的平均。

然后我们有四个例子和它们的均方误差的平方误差，然后就是均方误差，所以我们就计算平方误差，然后归一化，使其成为对四个例子的平均。接下来是梯度累积版本，在这里我们有梯度累积步骤为4，然后我重置了梯度。我们有4个梯度累积步骤，现在我们对所有例子逐个进行前向传播，并调用`L.backward()`多次。然后我们查看梯度。当我们查看梯度时，你会注意到它们不匹配。在这里，我们执行了一个批量大小为4的前向反向传播，而在这里我们执行了4次梯度累积步骤，每次批量大小为1，结果梯度不一样。

基本上，它们不一样的原因正是因为在这里的均方误差中，这个损失的1/4的因子丢失了。发生的情况是，每次循环中的损失目标都是均方误差，在这个情况下，因为只有一个例子，所以就是这个项的值。这是零次迭代中的损失，第一、第三等也是如此。然后当你执行`loss.backward()`时，梯度累积，累积的梯度基本上就等同于对损失的求和。因此，我们在这里的损失没有1/4的系数在外面，所以我们遗漏了归一化因子。因此，解决这个问题的一种方法是，我们可以在这里将损失除以4。这样就会给每个损失项引入一个1/4的因子，然后在`backward()`时，所有这些会按1/4进行缩放，所以当我们累计这些已经归一化的损失时，我们就恢复了额外的归一化因子。

长话短说，使用这个简单的示例，当你逐步查看时，你可以看到，为什么这不正确的原因就在于，当你像这里那样使用均方误差时，模型中的损失也默认使用了均值化的减少操作。

好，现在这两部分会和原始的优化等效，因为梯度会是一样的。好的，我需要做一些微调，然后启动优化。所以，特别是，我们想要做的一个事情是，因为我们要打印输出，所以首先，我们需要创建一个累加器来计算损失。我们不能只打印最终的损失，因为那样的话我们只会打印出最后一个微步骤的损失。所以，相反，我创建了`loss_ofon`，初始化为零，然后将损失累加到其中。我使用`detach`，是因为我在将张量从计算图中分离出来，我只是想跟踪这些值，所以当我加进去时，它们会变成叶节点。然后，我们在这里打印`lakum`，而不是打印`loss`。

此外，我还需要考虑梯度累积步骤中的`tokens processed`，因为现在每步处理的tokens是`B * T * gradient_accumulation`。优化看起来合理，对吧？我们从一个不错的起点开始，计算出的梯度步骤是32，且我们在这里大约花费了3秒钟。看起来不错。如果你想验证你的优化和实现是否正确，并且你正在侧重于一边工作，那是因为我们现在有了总的批量大小和梯度累积步骤，所以我们的B设置纯粹是一个性能优化设置。如果你有一个大GPU，你可以把它提高到32，速度可能会更快。如果你有一个非常小的GPU，你可以尝试设置为8或者4，但无论如何，你应该得到完全相同的优化结果和答案，除非是浮动误差，因为梯度累积可以处理所有的序列操作。

好了，这就是梯度累积的部分。我觉得好像差不多了，现在是时候拿出重型武器了。你已经注意到，迄今为止我只使用了一个GPU进行训练，但实际上我在支付八个GPU的费

用。所以，我们应该让它们都工作，特别是它们将协作并同时优化tokens。它们会进行通信，因此它们都会在优化过程中协同工作。我们将使用PyTorch的分布式数据并行（DDP）。这里还有一个遗留的`data parallel`，我建议你不要使用那个。那种方式工作得非常简单。我们有八个GPU，所以我们将启动八个进程，每个进程分配给一个GPU。对于每个进程，训练循环和我们到目前为止做的一切基本相同。GPU对它来说，只是处理我们到目前为止构建的内容。但现在，实际上是有八个GPU，它们将处理数据的不同部分。当它们计算出梯度后，我们将进行一次梯度平均。

我们将合作处理计算工作量。为了使用所有八个GPU，我们不再仅用`PyTorch train GPT-2`命令来启动我们的脚本，而是用一个特殊的命令`torchrun`来运行PyTorch。稍后我们会看到，`torchrun`在运行我们的Python脚本时，实际上会确保它并行运行八个副本。它创建了这些环境变量，每个进程都可以查看自己是哪个进程。例如，`torchrun`会设置`rank`、`local rank`和`world size`的环境变量。这不是检测DDP是否运行的好方法，所以如果我们使用`torchrun`，并且DDP正在运行，那么我们必须确保K是可用的，因为我不知道你是否可以再用CPU运行，或者那样做是否有意义。这里有一些设置代码，最重要的一部分是，`world size`将是八——即我们将运行的进程总数。`rank`意味着每个进程将基本上同时运行相同的代码，唯一的不同是它们的DDP rank会有不同。GPU 0将有一个DDP rank为0，GPU 1将有rank为1，依此类推。否则，它们都在运行相同的脚本，只不过DDP rank是一个稍微不同的整数。这是我们协调它们不会在相同的数据上运行的方式。

`local rank`只在多节点设置中使用。我们只有一个节点的GPU，因此`local rank`是单个节点上GPU的rank，比如从0到7作为例子。但对我们来说，我们大多数时候会在单台机器上运行，所以我们关注的是`Rank`和`World size`，`world size`是八，`rank`根据GPU来决定。现在，在这里，我们确保根据`local rank`设置设备为CUDA，这表示如果有多个GPU，它将使用正确的GPU，避免不同进程使用同一个GPU。最后，我喜欢创建一个布尔变量，即`DDP rank == 0`，这是一个被任命为“主进程”的变量，通常是进程号为零的进程，它负责处理一些打印、日志记录、检查点等任务。其他进程大多数被认为是计算进程，主要执行前向和反向传递。如果我们没有使用DDP，并且这些变量都没有设置，我们就会回退到单GPU训练。

这意味着我们只有rank 0，`world size`为1，我们是主进程。我们尝试自动检测设备，这是正常的。到目前为止，我们所做的只是初始化DDP。在运行`torchrun`的情况下，稍后会有八个副本并行运行，每个副本有不同的rank。我们必须确保接下来的操作正确执行。运行多个进程时，最棘手的事情是你必须设想有八个进程并行运行。所以，当你现在读代码时，你要想象有八个Python解释器在并行运行这些代码行，唯一的不同是它们有不同的DDP rank。它们都会来这里，它们都会选择完全相同的种子，完成所有这些计算，而彼此之间并不知道其他副本的运行情况。大致来说，它们都会做相同的计算。现在，我们必须调整这些计算，考虑到世界大小和不同的rank。

特别地，这些微批次和序列长度仅仅是每个GPU的计算结果，对吧？现在它们将以进程数量并行运行。我们需要调整这个，因为现在梯度步骤将是总的B大小 / B * T * DDP大小，因为每个进程将执行B * T，而且一共有这么多个进程。此外，我们还要确保它适合总批量大小。对我们来说，16 * 124 * 8 GPU是131K，所以524288。这意味着，在当前设置下，我们的梯度将是4。对吗？

所以每个GPU将计算16 * 124个进程，然后还有8个GPU，这样我们将一次前向反向处理131,000个tokens。所以，我们要确保它能够很好地适配，以便我们可以推导出一个合适的梯度累积步骤。对了，我们在这里调整注释，乘上DDP世界大小。好的，每个GPU计算这个，现在我们开始遇到一些问题了。对吧？每个进程都将打印输出，而它们都将在打印时出现。所以，一种解决方法是使用我们之前提到的主进程变量。所以如果是主进程，那么我们就进行保护，这样就只会打印一次。否则，所有进程都会计算出完全相同的变量，没有必要打印八次。

在进入数据加载器之前，我们还需要重构它。

显然，也许此时我们应该做一些打印，试着运行一下并退出。此时，导入 sys 和 S，开始退出并打印 IM GPU。嗯，DDP，rank IM GPU DDP rank，还有那个，打印 by。那么，现在让我们尝试运行一下，看看结果如何。通常我们用来启动的是 python train gpt2 P 这样的命令。现在，我们将使用 torch run，这就是命令的样子。所以，torch run standalone。比如，我们有 8 个 GPU，那么进程数就是 8。然后是 change of2 Pi。这就是命令的格式，torch run 将会运行这 8 个进程。我们来看看会发生什么。首先，事情变得有些忙乱，很多事情同时发生。首先，有一些来自分布式的警告，我实际上不知道这些是什么意思。我猜这只是代码正在设置过程中，进程上线时出现的一些初步失败，可能是收集数据的时候出现问题。虽然我不完全确定，但我们开始看到实际的打印输出。

所有进程都退出了，第一个打印输出实际上来自进程 5，纯粹是偶然，打印出来的内容是 "I'm process on GPU 5"。然后，这些打印来自主进程。所以，进程 5 首先完成了，可能是操作系统调度的原因，导致进程 5 先执行。然后，GPU 0 完成了，接着是 GPU 3 和 GPU 2，然后大概是进程 5 之类的退出了。DDP 非常不喜欢这种情况，因为我们没有正确处理多 GPU 设置。进程组在销毁之前没有被清理，这一点 DDP 非常不喜欢。在实际应用中，我们需要调用 destroy process group 来确保 DDP 正常清理。DDP 并不喜欢这种情况。接着，剩余的 GPU 完成了任务，就这样。

进程在并行运行，但我们不希望它们打印。接下来，让我们删除这些内容。接下来，我们需要确保在创建数据加载器时，要使其适应这个多进程设置，因为我们不希望所有进程加载相同的数据。我们希望每个进程处理数据集的不同部分。所以我们来调整一下。特别简单且直观的方法是，我们需要将 rank 和 size 传递给数据加载器，然后在这里，我们保存

rank 和进程数。现在当前的位置将不再是 0，因为我们希望每个进程处理不同的数据块。一个简单的方法是，我们基本上将 B * T 乘以进程的 rank，因此进程 rank 为 0 的将从 0 开始，进程 rank 为 1 的将从 B * T 开始，进程 rank 为 2 的将从 2 * B * T 开始，以此类推。这就是初始化，现在我们依然按原来的方式进行，但是当我们推进时，我们不再按照 B * T 来推进，而是按照 B * T 乘以进程数来推进。基本上，总的 token 数量是 B * T * 进程数，每个进程都有自己的数据块，位置要按整个块来推进。然后，这里 B * T * 进程数 + 1 就会超出 token 数量，然后我们就会循环，在循环时我们会重置。好了，这是我找到的最简单的调整方法，适用于一个非常简单的分布式数据加载器。你会注意到，如果进程 rank 是 0 且进程数是 1，那么整个流程就会和之前一样，但现在我们可以有多个进程并行运行，这应该能正常工作。

接下来，一旦它们都初始化了数据加载器，它们就会在这里创建 GPT 模型。也就是说，我们在 8 个进程中创建了 8 个 GPT 模型。但由于种子是固定的，它们都会创建相同的模型。然后，每个模型都会被移到各自的 GPU 上，并且每个模型都会被编译。由于这些模型是相同的，它们会在并行中执行 8 次编译，但没关系。现在，这一部分没有变化，因为这是在每一步中进行的操作，我们目前只是工作在每一个步骤中。我们需要做的就是在构建模型时，我们实际上需要做一些工作。这里，`get loits` 已经被弃用，所以为了创建模型，我们需要将模型包装到分布式数据并行容器中。我们就是这样将模型包装进 DDP 容器的，DDP 的文档很详细，内容很多，特别是要注意很多问题，因为一旦涉及到多个进程，复杂度就会增加十倍。但大体来说，我认为这里需要传递的应该是 DDP 的本地 rank，而不是 DDP 的全局 rank。所以，我们在这里传递本地 rank，这样就能正确地将模型包装进去。DDP 所做的事情之一是，在前向传递过程中，它实际上行为是一样的。我理解的情况是，前向传递应该没有变化，但在反向传递时，当每个 GPU 完成反向传播后，每个独立的 GPU 会得到所有参数的梯度。DDP 的作用是，在反向传播完成后，它会执行一个叫做 "all reduce" 的操作，基本上会在所有 rank 上做一次平均，并将平均值存储到每个 rank 上。也就是说，每个 rank 上都会得到梯度的平均值，DDP 就是帮助进行这个同步和梯度平均。

实际上，DDP 要比这复杂一些，因为在反向传播过程中，它会在反向传播的同时调度通信，将梯度同步。这样做的好处是，它能提高效率，减少反向传播和梯度同步之间的等待时间，这就是 DDP 所做的。前向传递没有变化，反向传递大部分也没有变化，只是在其中加入了这个平均操作。好了，现在我们来看看优化部分。这里没有变化，内循环还是一样的。我们来考虑一下在 DDP 中梯度同步的处理。正如我之前提到的，默认情况下，当你进行 `l backward` 操作时，它会先执行反向传播，然后同步梯度。问题是，由于梯度累积步骤的存在，我们实际上不想在每一个 `l` 步反向传播后就进行同步，因为我们只是将梯度累积起来，而且是串行进行的，我们只希望在最后一步时进行同步。为了实现这一点，我们可以使用官方推荐的 `no sync` 上下文管理器。PyTorch 提供了这个上下文管理器来禁用跨 DDP 进程的梯度同步。在这个上下文中，梯度将会被累积，而没有任何通

信。官方建议在 `no sync` 环境下进行梯度累积。然后，我们再次执行 DDP 操作，再进行反向传播。其实，我并不太喜欢这种做法。你需要复制粘贴代码，并且使用上下文管理器，这显得有些不优雅。于是，我查看了源码，你会发现进入时，实际上只是切换了一个变量，这个变量控制着是否需要同步反向图。通过查看代码，你可以看到这个变量在执行时会被切换，用来确定是否进行梯度同步。因此，我决定直接操作这个变量。换句话说，在反向传播之前，如果我们使用了 DDP，那么我们只希望在最后一步才进行同步。在所有其他微步中，我们希望这个变量保持为 `false`。我只是直接切换这个变量，然后在最后一步执行时，反向传播将不会同步，而是等到最后一步时才同步梯度。这样做并不是官方推荐的方法，因为未来可能会有 DDP 的变化，这个变量可能会消失，但目前这种方法有效，而且它避免了代码的重复。最终，在循环结束后，每个 rank 都会得到所有梯度的平均值。接下来，我们需要考虑这个方法是否有效，并且如何与损失计算相结合。

我们现在来思考一下这个问题。我的意思是，我们已经对梯度进行了平均，这很好，但损失函数还没有被影响到，而且这部分代码是在DDP容器之外运行的，所以损失函数并没有被平均。在这里，当我们打印损失时，我们假设只会在主进程（rank 0）上打印，而它只是打印了自己进程中看到的损失。然而，我们希望打印的是所有进程的损失，并且是这些损失的平均值，因为我们已经对梯度进行了平均，因此我们也希望损失值是平均的。所以，在这之后，只需设置我过去使用过的代码。这里，我们想要使用的是loss matrix，而不是loss function。所以，如果是DDP模型，我们需要导入PyTorch分布式模块。我在哪里导入呢？哦天啊，这个文件开始变得有点乱了。所以，如果我们导入torch.distributed作为dist，那么dist会做一些事情，处理对Loss的平均。在执行all_reduce时，它会计算这些值的平均并将这个平均值传递给所有的rank。所有rank在此调用之后将包含相同的平均值。所以，当我们在主进程中打印时，所有rank的Loss值是相同的。

最后，我们要小心，因为我们不再处理任何新的tokens，所以我们要乘上DDP世界大小（这是上面处理过的token数量），其他的应该没问题。唯一要小心的就是，正如我提到的，你需要销毁进程组，以确保退出时不会对Nickel发出任何抱怨。好了，应该就这些了。让我们试着运行一下。现在，我已经启动了脚本，应该马上开始打印了。我们现在正在用8个GPU同时训练，因此梯度累积步骤不再是32，而是除以8变成4。所以，优化过程大致是这样的，哇，我们现在的处理速度非常快，每秒处理150万个token，这简直是个巨大的数字。小型的莎士比亚数据集实在是太小了，我们大概进行的epoch非常多。但大致的情况是这样的。顺便说一句，我需要修复的一个问题是：原本的模型配置优化器的代码现在不适用了，因为模型现在是DDP模型。所以，这部分需要改成原始模型配置优化器，其中原始模型是我在这里创建的。在我将模型包装成DDP模型后，我需要创建原始模型，这个原始模型是模块中的一个存储位置，里面存放了GPT-2模块，它包含了我们需要调用的配置优化器的函数。所以，这是我需要修复的一个问题。除此之外，现在看起来一切正常。

你会注意到的一点是，当你将这次运行的结果与单个GPU的运行结果进行比较时，数字不会完全一致。这个原因其实挺无聊的。问题在于，数据加载器现在的迭代方式略有不同，因为我们现在要处理所有GPU的整页数据。如果这页数据超过了token的数量，我们就会循环遍历。所以，实际上，单GPU和多GPU的过程会以略微不同的方式重置，这导致了批次略有不同，从而导致结果稍微有差异。但要让自己相信这是可以接受的，你可以尝试将总批次大小设置得小一些。我记得我用的是4 * 124 * 8，我设置的总批次大小是32,768，然后确保单GPU进行8次梯度累积步骤。通过减少数据加载器的边界效应，你会看到数字会匹配。总之，长话短说，现在我们跑得非常快。优化过程与GPT-2的三项超参数基本一致，我们的小型莎士比亚文件也快不够用了，我们想要升级它。接下来，我们来看看GPT-2和GPT-3使用了哪些数据集。GPT-2使用了这个web text数据集，它从未发布。有人尝试重现它，称之为Open Web Text。大致来说，他们的论文里提到，他们从Reddit上抓取了所有的外链，要求每个外链至少有三次karma，然后将所有网页和网页中的文本都收集起来，这个数据集包含了4500万个链接，最终是40GB的文本。这就是GPT-2数据集的基本情况，实际上就是Reddit的外链。

接下来是GPT-3，它在训练数据集部分提到了Common Crawl，这是一个更常见的使用数据集。实际上，我认为GPT-2也提到过Common Crawl。但是，Common Crawl本身并不是一个质量很高的数据集。因为它是一个完全随机的互联网子集，质量远不如你想的那样。人们为了过滤Common Crawl的数据付出了很大努力，因为其中有一些有用的内容，但大多数都是广告垃圾、随机表格、数字和股票代码，简直就是一团乱麻。所以这也是为什么人们喜欢在这些数据混合物中进行训练，过滤掉噪声，确保内容有质量。一大部分数据混合物通常会包含Common Crawl。比如在GPT-3中，50%的token来自Common Crawl，另外他们还加入了之前的web text数据集。所以，除了Reddit外链，他们还加入了图书、维基百科等其他内容。你可以加入许多其他东西。

不过，GPT-3的数据集也从未公开发布。今天我熟悉的、代表性的数据集包括：第一个是Red Pajama数据集，或者更具体地说，是Red Pajama数据集的Slim Pajama子集，它是经过清洗和去重的版本。给你一个概念，它也是Common Crawl的一部分。还有C4，它也可以算作Common Crawl，但经过了不同的处理。然后是GitHub、Books、Archive、Wikipedia、Stack Exchange等数据集，这些都是可以加入到数据混合物中的内容。特别是我最近比较喜欢的一个数据集叫做Fine Web数据集。这个数据集试图收集非常高质量的Common Crawl数据，并对其进行筛选，目前已经处理了15万亿token。最近，Hugging Face还发布了Fine Web Edu子集，其中包括了1.3万亿的教育内容和5.4万亿的高质量教育内容。因此，它们正在尽力将Common Crawl数据过滤成高质量的教育数据集，而这就是我们将要使用的数据集。

在Fine Web的网页中，有很详细的介绍，描述了他们如何处理数据，这些内容非常有趣。如果你对数据混合物以及如何处理如此大规模的数据感兴趣，我强烈推荐你阅读这篇文章。更具体地说，我们将使用Fine Web Edu，它包含了互联网上的教育内容。他们展示

了，在这些数据集上进行训练，模型的表现非常好。而我们将使用这个数据集中的10亿token子集，因为根据我过去的一些实验，实际上这个规模就足以达到接近GPT-2的表现，且足够简单，便于使用。

因此，我们将使用这个10亿token的样本。我们的目标是下载、处理它，并确保我们的数据加载器能够正常工作。好了，现在我引入了另一个脚本，它会从Hugging Face的数据集中下载Fine Web Edu数据集，进行预处理和预标记化，并将数据分片保存到本地磁盘。这个过程中，我只想简要提到，大家可以通过数据集查看器了解其中的内容。基本上，它看起来工作得还不错，比如它讲到了法国的核能、墨西哥美国、一些Mac PJs等话题，过滤效果还是挺好。至于这个脚本，我不会一一讲解，因为它和LLM关系不大，更多是关于如何加载、标记和保存数据。我们首先加载数据集，然后对数据进行标记化处理。

这些分片是NumPy文件，只存储一个NumPy数组，类似于torch张量。第一个分片，0000，是验证分片，其他所有分片都是训练分片。正如我之前提到的，每个分片恰好有1亿个Token，这使得分片文件的处理变得更加容易，因为如果我们只使用一个庞大的文件，有时在磁盘上处理起来会很困难。因此，将文件分片从这个角度来说要更方便一些。好的，接下来我们就让它运行，这大概需要30分钟左右，之后我们将开始真正的预训练。在这种情况下，这是一个很好的数据集，我们每秒处理很多Token。我们有8个GPU，代码已准备好，因此我们实际上将进行一次严肃的训练。

好了，我们回来了。现在，如果我们列出edu fine web目录，我们看到现在有100个分片。这很有意义，因为每个分片有1亿个Token，所以100个分片就是总共10亿个Token。接下来，转到主文件，我对数据加载器做了一些调整，因为我们不再使用Shakespeare数据集，而是要使用fine web分片。因此，您会看到一些额外的代码，用来加载这些分片，我们加载NumPy文件，然后将其转换为torch long张量，这是许多上层模型默认期望的格式。接着我们枚举所有分片。我还添加了数据加载器的拆分功能，这样我们就可以加载训练集拆分，也可以加载验证集拆分——即零号拆分。然后我们可以加载分片，在这里我们不仅有当前的位置，还有当前分片。这样，我们就知道在一个分片内的位置。当我们用完一个分片中的Token时，首先推进到下一个分片，如果需要的话继续循环。然后我们获取Token并重新调整位置。因此，这个数据加载器现在可以遍历所有的分片了。所以，我做了这个调整。另一个方面是，在数据处理过程中，我们的训练加载器现在确实有了训练集拆分。下面，我设定了一些数字。

我们每步处理2的9次方个Token，我们想要处理大约10亿个Token，因为这就是我们拥有的唯一Token数。所以，如果我们处理10亿个Token，再除以29，结果是1973步。就是这个来源，GPT-3论文中提到，他们在375万个Token上进行了学习率预热。因此，我来这里把375E6个Token除以2的19次方，结果是715步。所以，预热步数设置为715。这将完全匹配GPT-3使用的预热计划，我觉得715这个值非常温和，甚至可以更积极一点，可能100就足够了。但是现在没关系，暂时先保持原样，这样我们就能用GPT-3的超参数了。所以，我

修正了这一点，然后就差不多了。我们可以运行了，我们这里有我们的脚本，然后可以启动。实际上，抱歉，我还要做一件事。请原谅我，我的GPU能够处理更大的批次大小，我相信我可以在GPU上处理64的微批次大小。让我试试看。我可能记错了，但如果可以，那就意味着每个GPU会有64次乘124的批次大小，然后我们有一个GPU。所以，这意味着我们甚至不需要做梯度累积，如果这可以的话，因为这个数字会直接乘出完整的总批次大小。

所以不需要梯度累积，运行起来应该很快。如果这一切能正常工作，那么这基本上就是一次认真的预训练。我们没有进行日志记录，也没有评估验证集，我们没有运行任何评估，所以我们还没有完全搞定。但如果我们让它运行一段时间，实际上我们会得到一个相当不错的模型，甚至可能与GPT-2的124M相当，甚至更好。好了，看起来一切都很顺利。我们每秒处理150万个Token，一切看起来不错。每次迭代大约需要330毫秒，我们总共需要做1973步。所以，1973乘以0.33是这些秒数，这些分钟数。所以，这将运行大约1.7小时。就是一个半小时的运行时间，像这样，甚至不需要使用梯度累积，这很棒。你可能没有这么高效的GPU，在这种情况下，只需要减少批次大小直到可以适应。但要确保保持合理的数字。所以，这非常令人兴奋。我们现在正在对学习率进行预热，您会看到它仍然非常低，约为1到4之间。所以它会在接下来的几个步骤中逐渐增加，一直到6E负4。这里非常酷。所以现在，我想做的事是，做个全面的准备。让我们评估一下验证集，并尝试弄清楚如何运行评估、如何记录、如何可视化损失，所有这些好东西。所以，在真正开始运行之前，让我们先搞定这些。

好的，我已经调整了代码，现在我们在验证集上进行评估。通过传递split = validation来创建验证加载器，这将基本上为验证创建一个数据加载器。另一个方面是，我在数据加载器中引入了一个新的reset函数，它在初始化时被调用，基本上会重置数据加载器。这非常有用，因为当我们来到主要的训练循环时，我添加了这段代码。基本上，每100次迭代，包括第零次迭代，我们就把模型设置为评估模式，重置验证加载器。然后，不涉及梯度的操作，我们将累积20步的梯度，然后平均起来，打印出验证损失。因此，这基本上与训练循环的逻辑完全相同，只不过没有反向传播，只有推理。我们只是衡量损失，然后将其加总。除此之外，其他一切都不变，和之前一样。这个过程将每100次迭代（包括第一次迭代）打印出验证损失。所以，这很棒，它将告诉我们关于我们是否过拟合的一些信息。话虽如此，实际上我们拥有几乎无限的数据，因此我们大致上预期我们的训练损失和验证损失会非常接近。但另一个我感兴趣的原因是，我们可以从OpenAI发布的GPT-2 124M模型初始化，看看它在验证集上能达到什么样的损失，这也给了我们一些关于该模型如何推广到124M模型的指示。不过，这并不是一个非常公平的比较，因为它是在一个完全不同的数据分布上训练的，但它仍然是一个有趣的数据点。无论如何，像这样，你总是需要在训练过程中拥有验证集，这样可以确保你不会发生过拟合，尤其是在我们开始增加训练数据的epoch时。如果我们现在只进行一次epoch，但如果我们决定训练10个epoch或者更长时

间，我们就必须非常小心，避免模型过度记忆数据，如果模型足够大，验证集将是告诉我们是否发生这种情况的一种方式。

好了，除了这些，如果你记得在我们脚本的底部，我们曾经有一些关于采样的代码，我删除了那些代码并把它们移动到了这里。每隔一段时间，我们就进行一次验证；每隔一段时间，我们生成一些样本，然后每100步生成一次，而我们在每一步训练中都进行训练。

目前的结构是，我已经运行了10,000次迭代，下面是第1000次生成的样本。你好！我是一个语言模型，我不太能更具创意。你好！我是一个语言模型，语言的文件你在这里学习的是计算机的开端。

好的，所有这些都还挺——呃，这仍然是乱码，但我们才刚刚到达第1000次迭代，并且我们刚刚刚好到达最大学习率。所以，它还在学习，我们在第2000次迭代时会看到更多的样本。好的。

好了，基本上，所有的采样代码我都已经放在这里，所有这一切你应该都很熟悉，并且之前都看过。唯一的变化是，我在PyTorch中创建了一个生成器对象，这样我就能直接控制随机数的采样。我不希望影响训练中使用的全局随机数生成器状态。所以，我使用了一个特别的采样随机数生成器，并确保对它进行初始化，这样每个rank就有一个不同的种子。然后我在我们进行多项式分布采样时传递生成器对象；否则，其他部分完全相同。另一个方面是，你会注意到我们运行得稍微慢一点。这是因为为了让这个采样工作，我实际上不得不禁用了torch.compile，因此我们的速度有点慢。因为某些原因，当我在模型上使用torch.compile时，我会得到一个很可怕的PyTorch错误，至今我还不知道如何解决。所以大概等到你看到这段代码发布时，可能这个问题已经解决了。

但现在，我只好设定"end"为false，然后恢复torch.compile，而你将无法获得样本。我想等我解决了问题后再修复这个。顺便说一下，我将发布所有这些代码，实际上，我一直非常小心地在每次添加新内容时都会提交git。因此，我将发布一个从头到尾的完整仓库，所有内容都会在git提交历史中完全记录下来。我认为这会很不错。所以，希望当你去GitHub时，这个问题已经修复，而且已经工作正常。好的，我在这里运行优化，现在已经走到了第6000步，大约完成了30%的训练。

在这个过程中，我想介绍一个评估指标，用来补充验证集，这就是H-SWAG评估。H-SWAG来自2019年的一篇论文，所以它已经有5年的历史了。H-SWAG的工作方式是基于句子完成数据集。所以，对于每个问题，它是一个多项选择题。我们有一个共享的上下文：比如一个女人在外面，手里拿着一只桶，旁边有一只狗。狗在四处跑，试图避免洗澡。

她用肥皂提起水桶，给狗的头吹干。B用水管保持狗不被弄上肥皂。C把狗弄湿，它又跑开了，或者D和狗一起进浴缸。

基本上，想法是，这些多个选择是按照这样构造的：其中一个是句子的自然延续，而其他的则不是。其他的选项可能没有意义，比如用水管来保持狗不被弄上肥皂，这没有任何意义。因此，训练不好模型是无法区分这些选项的，但有世界知识的模型能够区分这些，它们能理解世界，能够创建正确的回答。这些句子来源于ActivityNet和维基百科。

在论文的底部，有一个很酷的图表，显示了维基百科中不同领域的句子，因此有很多来自计算机、电子、家庭和园艺等方面的句子，涵盖了你需要了解的世界上许多不同的知识，以便找到最可能的回答。

这个回答的识别是H-Swag的一个有趣之处：它的构造方式。错误的选项是故意对抗性地构造的，它们不仅仅是随机的句子，实际上是由语言模型生成的。这些句子以某种方式生成，使得语言模型基本上很难理解，但人类却很容易理解。人类在这个数据集上的准确度是95%，而当时最先进的语言模型仅有48%的准确度。因此，当时这是一个很好的基准。现在，你可以阅读论文的细节，了解更多。嗯，值得指出的是，这个数据集是在五年前构建的，从那时起，H-Swag的数据集已经被完全解决。所以现在语言模型的准确度已经达到了96%，因此最后的4%可能是数据集中的错误，或者是问题真的很难。因此，基本上，这个数据集对于语言模型来说已经被“攻破”。但在那时，最好的语言模型的准确率仅为50%左右。这就是事情发展的程度。不过，仍然，H-Swag之所以被大家喜欢——而且顺便说一句，GPT-2中并没有使用H-Swag——但在GPT-3中有H-Swag评估，很多人使用H-Swag。所以对于GPT-3，我们有引用的结果，我们知道GPT-3在这些不同模型检查点上达到的准确度。

人们喜欢它的原因是，H-Swag是一种平滑的评估，它提供了所谓的“早期信号”。早期信号意味着，即使是很小的语言模型，从随机概率25%开始，它们也会慢慢提高。你可以看到25%、26%、27%等等，即使模型非常小，仍然可以看到逐渐提高的趋势，且这是一个非常早期的阶段。因此，它是平滑的，提供了早期信号，并且它已经存在了很长时间。这就是为什么大家喜欢这种评估方法。嗯，现在我们将按以下方式来评估它。

如我所提到的，我们有一个共享的上下文，这有点像一个多选任务。但是，不是给模型一个多选题让它选择A、B、C或D，因为这些模型在如此小的情况下，正如我们所看到的，模型实际上无法理解多选的概念，它们不理解这一点，因此我们不得不以本地形式提供给它们。本地形式是一个标记完成，所以我们这样做：我们构造一个四行的批次和T个标记，不管T是多少。然后，共享的上下文基本上是这些选择的上下文，这些标记在所有行中都是共享的，然后我们有四个选项。所以我们把它们摆在一起，然后只有一个选项是正确的，在这个例子中是第三个选项。因此，这是正确的选项，而选项一、二和四是不正确的。现在，这些选项可能长度不同，所以我们所做的是，我们以最长的长度作为批次（B）。

然后其中一些会是填充维度，因此它们将是未使用的，我们需要这些标记、正确的标签和一个掩码，告诉我们哪些标记是有效的，掩码对于这些填充区域为零。这样我们就构建了这些批次。然后，为了让语言模型预测A、B、C或D，实际上我们就是看这些标记的概率，然后我们选择那些获得最低或最高平均概率的选项——因为这是根据语言模型最可能的完成。因此，我们就看这些标记的概率，并将它们在各选项中平均，然后选择具有最高概率的一个，粗略来说。这就是我们进行H-Swag评估的方式。

我认为这也是GPT-3的评估方式。事实上，我知道GPT-3是这样做的。但你应该注意到，其他你可能会看到H-Swag评估的地方，可能不会采用这种方式，它们可能会采用多选格式，你给定一次上下文，然后给四个选项。这时，模型在做选择时能看到所有的选项。对于模型来说，这样的任务要容易得多，因为你在做选择时可以看到其他选项。

不幸的是，我们的模型在这个方面略显“残疾”，它们无法看到其他选项，它们只能一次看到一个选项，只能为它们分配概率，正确的选项必须在这个指标中脱颖而出。那么，接下来让我们来简要实现这个，并将其合并到我们的脚本中。好的，我在这里做了一个新的文件，叫做“hell swag.py”，你可以看一下。

我不会一步步讲解所有内容，因为这不完全是深入的代码。深入的代码有点冗长，老实说。因为发生的事情是，我从GitHub下载HSAC，并将它的所有示例呈现出来，总共有10,000个示例。我将它们呈现成这种格式。在这个“render example”函数的结尾，你可以看到，我正在返回这个四行T标记数组的标记。掩码涉及哪些部分是选项，其他的都是零。标签是正确的标签，这样我们就能遍历示例并进行渲染。我在这里有一个“evaluate”函数，可以从Hugging Face加载GPT-2模型，并运行评估。在这里，它基本上计算了像我描述的那样，预测下一个标记的交叉熵损失。

然后，我们查看平均损失最小的那一行，这是我们选择的预测选项。然后我们做一些统计并打印出来，这就是评估H-Swag的方法。现在，如果你看上面，我展示了GPT-2 124M，如果你运行这个脚本，你会看到H-Swag得到29.5%的准确率。这是我们在这里得到的表现。

现在记住，随机的概率是25%，所以我们还没有走得太远。GPT-2 XL，最大的那个，大约达到了49%。因此，考虑到今天的最先进技术准确度通常是95%，这些数值相对来说是比较低的，毕竟这些模型如今已经是老款了。

接着还有一个叫做Uther Harness的非常有用的基础设施，用于运行语言模型评估，它给出了略有不同的数字。我不完全确定这种差异是什么，可能是它们实际上做了多选，而不是直接完成选择，这可能是导致差异的原因，但我不完全确定，得再看一看。现在，我们的脚本报告的是2955，因此这是我们训练GPT-2 124M时希望超越的数值。

接下来，我将把这个评估方法合并到我们主要的训练脚本中，基本上因为我们希望周期性地评估它，以便我们能够跟踪H-Swag，并观察它是如何随着时间演变的，看看我们是否跨越了2955这一区域。

接下来，我们来看一下训练GPT-2时的一些变化。首先，我在这里使用了“compile optional kind of”并默认禁用了它。问题在于，虽然compile确实能加快代码速度，但它实际上会破坏评估代码和采样代码，给我带来了非常棘手的错误信息，我不知道为什么。所以，希望等到你看到这个代码库时，我会解决这个问题，但现在我没有启用torch compile，因此运行速度稍慢一些。我们没有使用torch compile。

你应该把这段代码识别为我们在视频开始时写的古老代码，我们只是在从模型中抽样。最后，这里，这些步骤是我们在验证样本并评估 H Swag 后，实际上进行的一步训练。你应该对这些步骤都很熟悉。最后，一旦我们得到了训练损失，我们将其写入文件。唯一的变化，我真正添加的是这一整段 H Swag 评估部分。这个过程的工作原理是，我试图让所有的 GPU 协同工作进行 H Swag 评估，所以我们正在迭代所有的样本。每个进程只选择分配给它的样本。我们通过对 I 取模并让其等于 rank 来进行处理，否则我们就继续。然后我们渲染一个样本，将它放到 GPU 上，获取预测结果。

接下来，我创建了一个辅助函数，帮助我们基本上预测损失最小的选项。然后，我们进行预测。如果预测正确，我们就统计。由于多个进程正在协作处理所有这些内容，我们需要同步它们的统计数据。一种做法是将我们的统计数据打包成张量，然后调用 Alberon.Sum，然后我们把这些张量解开，从中提取出输入，最后主进程会打印并记录准确率。大致就是这样，这就是我目前运行的内容。你看到的是这个优化过程，我们刚刚进行了生成，这是 10,000 步中的一部分，整个过程大约有 20,000 步。我们已经完成了一半，这些就是我们在这一阶段得到的样本。让我们来看一下。

“你好，我是一个语言模型，所以我希望用它生成一些输出。

你好，我是一个语言模型，我为很多公司做开发工作。一个 AI 语言模型

——让我看看我能找到一个。嗯，我不知道，你可以自己试试，但显然随着训练进行，预测变得越来越不随机了。模型似乎变得更加自我意识，使用的语言也越来越具体。

你好，我是一个语言模型，我像语言一样使用语言进行沟通。

我是一个语言模型，我将会讲英语和德语。

好吧，我不确定，我们就等到这个优化完成。我们会看看能得到什么样的样本，同时我们也会查看训练验证和 Hella 的准确性，看看与 GPT-2 相比我们的表现如何。

好，早上好。现在我们集中在右边的 Jupyter Notebook 上，我创建了一个新的单元格，基本上让我们可视化训练验证和 Hela 分数。你可以浏览这个，它基本上解析了我们写入的日志文件。嗯，很多内容是一些无聊的 matplotlib 代码，但基本上就是这个样子，显示了我们的优化过程。我们已经运行了 19,731 亿个标记，也就是——哎呀！哦，我的天哪！这是用 10B 的 WebD 样本进行的一次迭代。在左边，我们看到的是损失，蓝色是训练损失，橙色是验证损失，红色是一个水平线，代表当 GPT-2 124M 模型的初始检查点仅在该验证集上进行评估时的表现。所以你可以看到，我们的验证集成绩已经超越了它，橙色的线低于红色。我们超越了这个验证集，正如我所提到的，这个数据集分布和 GPT-2 训练时的数据集分布非常不同。所以这不是一个完全公平的比较，但它是一个很好的交叉检查。现在，我们理想的情况是有一个保留集，可以比较且比较标准。嗯，对我们来说，这就是 Hella Swag 等数据集。在这里，我们看到 H Swag 的进展，从 25% 到现在这里，红色是 Open GPT-2 124M 模型的结果，达到了这个 H Swag。GPT-3 124M 模型，在训练了 3000 亿个标记后，表现更好，达到了绿色的这一部分。所以你可以看到，我们实际上已经超越了 GPT-2 124M 模型，这真的很不错。有趣的是，我们只训练了 100 亿个标记，而 GPT-2 是在 1000 亿个标记上训练的。因此，某些原因可能使得我们能用显著少得多的标记量训练成功。可能的原因之一是 Open GPT-2 使用了更广泛的数据集分布，特别是 fine web edu 完全是英文，而不是多语言的，也没有那么多数学和代码。嗯，数学和代码以及多语言可能在某种程度上消耗了 GPT-2 原始模型的容量。而且，基本上，这可能是造成这种现象的部分原因。当然，也有其他原因。比如说，H Swag 评估已经相当陈旧，大约有五年时间。也许 H Swag 的某些方面，或者甚至是完全相同的数据，已经进入了训练集。我们不能完全确定，但如果真是这样，那我们实际上是在看训练曲线，而不是验证曲线。长话短说，这个评估并不完美，还是有一些前提需要考虑。但至少我们有一定的信心，证明我们没有做错什么。而且，很可能是因为人们在创建这些数据集时，会特别确保测试集中的常见数据不包含在训练集中。例如，当 Hugging Face 创建 fine web BDU 数据集时，它使用了 H Swag 作为评估数据集，所以我希望他们能确保不重复，并且没有把 Hella Swag 包含在训练集里，但我们无法完全确定。另一个我想简要提到的内容是看这个损失曲线，它看起来真的很不正常。这里，我实际上不完全清楚这是为什么，我怀疑是因为 fine web edu 这 100 亿标记的样本没有经过适当的洗牌。这里的数据存在一些我还没有完全理解的问题，而且看起来有一些奇怪的周期性。因为我们以一种非常懒散的方式序列化所有标记，直接从头开始迭代所有标记，完全没有进行任何随机洗牌，我想我们是继承了数据集中本身的顺序。因此，这样做并不理想，但希望在你们到这个仓库时，这些问题能够得到修复。顺便提一下，我会将这个构建版本发布到 GPT 仓库中，尽管现在看起来有些丑陋且初步。但希望等你们看到的时候，它会更漂亮。不过在这里，我将展示 Ada，并谈谈一些视频之后发生的事情。我预计我们会解决这些小问题，但目前基本上，这显示出我们的训练并没有完全错，且我们能够以 10 倍的标记预算超越准确度。也有可能是数据集有所改进。原始 GPT-2 数据集是 Web 文本，可能当时在数据集上并没有付出太多关注。那时 LLM 还处于早期阶段，而现在对于良好的去重、过滤和质量筛选有了更多的关

注。可能我们正在训练的数据质量更高，每个标记的质量更好，这也可能给我们带来了提升。因此，很多前提需要考虑，但目前我们对这些结果还是比较满意。

接下来，我感兴趣的是，正如你们所见，现在是早晨，经过一夜的运行，我想看看结果能推得多远。为了进行这次过夜训练，我做了一个改变，我没有进行一个周期，而是进行了四个周期。每个周期大约需要两个小时，所以我做了四个周期，这样就需要八小时，我睡觉时完成了这四个周期，大约训练了 400 亿个标记。看看我们能达到什么样的结果。嗯，这就是唯一的改变，我重新运行了脚本，当我读取到 40B 的日志文件时，曲线看起来是这样的。

好吧，来解释一下，首先，我们看到在不同周期之间有这个周期性问题，以及 fine web edu 数据集的一些奇怪问题，这个问题待定。嗯，但其他情况下，我们看到 H Swag 实际上大幅上升，我们几乎达到了 GPT-3 124M 的准确度，虽然还没完全达到。哎呀，真可惜，我没睡得稍微长一点。如果这是五个周期的训练，我们可能就能够达到这个目标。现在有一点要指出的是，如果你在做多周期的训练，我们并没有非常小心地使用数据加载器。这个数据加载器完全以相同的格式和顺序遍历数据，这样做有点不够优化。你应该看一下扩展的部分，实际随机排列每一个新周期的每个数据块。数据块的洗牌会极大地减少复杂性，而且对优化也有好处，这样你就不会以相同的格式看到数据，而且在文档的顺序上引入一些随机性，因为你得记住，每一行中这些文档是按顺序排列的，然后是文本结尾标记，接下来是下一个文档。所以，文档目前是以完全相同的方式连接在一起的，但我们实际上希望打破这些文档，重新洗牌它们，因为文档的顺序不应该有任何关系。它们不应该——基本上，我们想打破这种依赖关系，因为这是一个虚假的相关性。我们的数据加载器目前没有这么做，这是一个可以改进的地方。另一个要注意的点是，我们几乎达到了 GPT-3 的准确度，仅仅用了 400 亿个标记，而 GPT-3 训练时使用了 3000 亿个标记。所以，再次说明，学习效率大约提高了 10 倍。还有我想提到的一点，我不完全知道怎么解释这个问题，但要解决我之前提到的一些问题。还有一个我想简要提到的是最大学习率。看到有些人已经在前一个相关的仓库里做过一些实验，结果发现你实际上可以将它几乎三倍。

所以，嗯，他们实际上是在解决它是一个语言模型的问题。几乎，呃，大家好，我是一个语言模型，我尽量做到尽可能准确。嗯，我是一个语言模型，不是编程语言。我知道如何进行沟通，呃，我使用 Python。
看看它，然后将其与只训练了 100 亿参数的模型进行比较，你会发现这些模型更加连贯，你可以自己试试。
顺便说一句，我在代码中添加了这个代码块。基本上，在我们评估验证损失之后，如果我们是主进程，除了每 5000 步记录一次验证损失外，我们还会保存检查点，这实际上就是模型的状态字典。所以，检查点保存很不错，因为你可以保存模型，之后如果你想恢复优化，也可以使用它。除了保存模型，我们还必须保存优化器的状态字典，因为记住，优化器有一些额外的缓存，尤其是在使用 Adam 时。

它有 m 和 v，你还需要正确恢复优化器。你必须小心你的 RNG 种子，随机数生成器等等。如果你想精确地恢复优化过程，你必须考虑训练过程的状态。但是如果你只想保存模型，这就是你该怎么做的一个原因。
你可能会想这么做的一个很好的原因是，因为你可能想更仔细地评估这个模型。
所以在这里，我们只是草率地进行评估，但你可能希望使用一些更好的方法，比如 LUTHER 评估硬度。
这是评估语言模型的另一种方法，可能你会想要使用不同的基础设施来更彻底地评估模型在多个任务上的表现，比较它们与开源 GPT-2 模型的表现，例如，涉及数学、代码或不同语言的任务。所以这也是一个很好的功能。
然后，我还想提到的是，我们今天所构建的一切仅仅是预训练阶段。所以，GPT 在这里是一个梦，它预测下一个词。你不能像与 Chat GPT 交流那样与它对话。如果你想与模型对话，我们需要对它进行微调，转变为聊天格式，而这并不复杂。
看一下监督微调（SFT），实际上这意味着我们只需将数据集替换为一个更加对话式的数据集，并且有一个用户助手的结构。我们只是对其进行微调，然后我们基本上填充用户令牌，采样助手令牌。没什么比这更复杂的了，基本上我们只是更换数据集并继续训练。但现在，我们将停留在预训练阶段。
我还想简要展示一下，当然，我们今天所构建的是朝着 Nanog GPT 这一仓库前进的，早些时候提到的那个，但其实还有另一个 Nanog GPT 实现，它藏在我最近在做的一个名为 LLM Doc 的项目中。LLM.C 是一个纯 CUDA 实现的 GPT-2 或 GPT-3 训练，它直接使用 CUDA，并且是用 CUDA 编写的。现在，Nanog GPT 在这里作为参考代码，与 C 实现相匹配，我们正在尝试精确对比这两者，但我们希望 C CUDA 更快，显然，目前看起来确实是这样，因为它是一个直接优化的实现。
用 LLM.C 训练 GPT-2 的实现基本上就是 NanoGPT，滚动查看这个文件，你会发现很多内容非常像我们在这堂课中所做的事情。然后，当你查看训练 GPT-2 的文档时，这就是 CUDA 实现。所以这里有很多 MPI、Nickel、GPU、CUDA，你必须对这些内容有所了解。但当这个实现完成时，我们实际上可以将这两者并排运行，并且它们会产生完全相同的结果。但 LM.C 运行得更快。
看看左侧，我有 PyTorch，类似 NanoGPT 的东西。右侧，我有 LLM.C 调用，这里我将同时启动它们。它们都将在一个 GPU 上运行，这里我将 LM.C 放在 GPU 1 上，这个会默认使用 GPU 0。
然后我们可以看到，LM.C 已经编译并分配了空间，开始了。所以基本上，与此同时，PyTorch 仍在编译，因为 Torch 编译比 LM.C 的 NBCC CUDA 编译要慢。
开始运行了，PyTorch 仍然在等待编译。当然，这个实现是 GPT-2 和 3 的非常具体实现，PyTorch 是一个非常通用的神经网络框架，所以它们并不完全可比。但如果你只关心训练 GPT-2 和 3，LM.C 确实非常快。它占用的空间更少，启动更快，每一步的速度也更快。
所以，P 开始执行了，正如你所看到的，我们在这里大约运行 223,000 个 token 每秒，而这里是大约 185,000 个 token 每秒。
嗯，速度明显慢一些，但我没有完全确认是否已经从 PyTorch 实现中榨干了所有的潜力。

但重要的是，如果我对齐这些步骤，你会看到这两者的损失和范数是完全一致的。所以左边是 PyTorch 实现，右边是 C 实现，它们唯一的不同就是这个运行得更快。这就是我想向你展示的。
另外简要提一下，LM.C 是一个并行实现，它也是你可能想要尝试的东西。看看它，嗯，这其实挺有趣的。
好了，到此为止，我应该开始总结这个视频了，因为它可能比我预期的要长很多。呃，但我们确实覆盖了很多内容，我们从头开始构建了所有东西。
简要总结一下，我们看了 GPT-2 和 GPT-3 的论文，研究了如何设置这些训练过程以及其中涉及的各种考虑因素。我们从零开始编写了所有代码，然后我们看到，在大约两小时的训练或一个过夜的训练中，我们实际上能够在很大程度上匹配 GPT-2 和 GPT-3 的 1.24 亿参数检查点。
原则上，我们编写的代码应该能够训练更大的模型，如果你有耐心或计算资源的话。呃，所以你也许可以考虑训练一些更大的检查点。
还有一些剩余问题需要解决。验证损失的变化是怎么回事，我怀疑这与数据的精细化采样有关。为什么我们不能启用 Torch 编译？目前它会破坏生成和 H swag。数据加载器的情况是怎样的？我们可能应该在达到边界时打乱数据，所以还有一些问题需要解决。我预计会在这个 GPT 仓库的构建中记录一些这些问题，并在发布这个视频时一起分享。如果你有任何问题或想讨论我们涵盖的内容，请访问讨论标签，我们可以在这里讨论，或者请访问问题或拉取请求，取决于你想要做什么。
另外，看看 Zero to Hero Discord，我会在 N GPT 上待着。
不管怎样，现在我对我们取得的进展很满意。我希望你喜欢这个视频，我们下次见。