Mamba: The Easy Way　Mamba: 小菜一碟

Oxford, UK — February 23, 2024　牛津,英国　2024.2.23

如今,几乎所有你能说出名字的语言模型都是 Transformer 模型。OpenAI 的 ChatGPT、Google 的 Gemini 和 GitHub 的 Copilot 等都是由 Transformer 驱动的。然而,Transformer 存在一个根本缺陷:它们由注意力机制驱动,而**注意力机制的计算复杂度与序列长度呈二次方关系**。简单来说,对于简短的交互(比如讲个笑话),这不是问题。但对于需要处理大量文字的查询(比如让 ChatGPT 总结一份 100 页的文档),Transformer 可能会变得 prohibitively 慢。

许多模型试图解决这个问题,但很少有模型能做得像 Mamba 那样好。由 Albert Gu 和 Tri Dao 两个月前发布的 Mamba,似乎在性能上超越了同等规模的 Transformer,同时其计算复杂度与序列长度呈线性关系。如果你在寻找对 Mamba 的深入技术解释,以及完整的 Triton 实现,那你来错地方了。《Mamba:硬核之路》已由传奇人物 Sasha Rush 写出。如果你没听说过 Mamba 或 Triton,或者你想要一个更高层次的 Mamba 核心思想概述,那我这篇文章正适合你。

一个准确且时间复杂度为线性的语言模型的前景让许多人对语言模型架构的未来感到兴奋(尤其是 Sasha,他在这上面下了赌注)。在这篇博文中,我将尝试以一种相当直白的方式解释 Mamba 的工作原理,特别是对于那些之前学过一点计算机科学的人来说应该很容易理解。让我们开始吧!

Background: S4　背景知识:S4

Mamba 的架构主要基于 S4 (Mamba 是 S6, LipLang 译注),这是一个最近提出的状态空间模型(SSM)架构。我会在这里总结重要的部分,但如果你想更详细地了解 S4,我强烈推荐阅读 Sasha 的另一篇博文《The Annotated S4》。

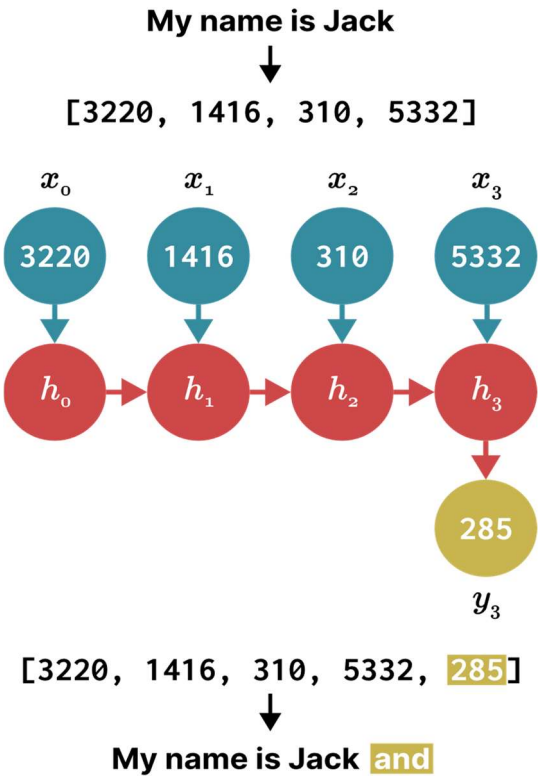
从高层次来看,S4 学习如何通过一个中间状态 h 将输入 u 映射到输出 y。这里,u, h 和 y 都是 t 的函数,因为 SSM 被设计用来很好地处理连续数据,如音频、传感器数据和图像。S4 通过三个连续参数矩阵 A, B 和 C 将它们联系在一起。这些都通过以下两个方程(Mamba 论文中的 1a 和 1b)联系在一起:

$$h'(t) = \mathbf{A}h(t) + \mathbf{B}x(t)$$
$$y(t) = \mathbf{C}h(t)$$

在实践中,我们总是处理离散数据,比如文本。这要求我们将 SSM 离散化,通过使用一个特殊的第四个参数 Δ 将我们的连续参数 A, B, C 转换为离散参数 $\bar{A}, \bar{B}, \bar{C}$。我在这里不会深入讨论离散化是如何工作的,但如果你好奇的话,S4 的作者写了一篇不错的博文介绍这个过程。一旦离散化,我们可以通过这两个方程(2a 和 2b)来表示 SSM:

$$h_t = \bar{\mathbf{A}}h_{t-1} + \bar{\mathbf{B}}x_t$$
$$y_t = \mathbf{C}h_t$$

这些方程形成了一个递归,类似于你在递归神经网络(RNN)中看到的。在每一步 k,我们将前一个时间步(k-1)的隐藏状态与当前输入结合,创建新的隐藏状态 h_k。下面,你可以看到这在预测句子中的下一个词时是如何工作的(在这个例子中,我们预测"and"跟在"My name is Jack"之后)。



通过这种方式,我们基本上可以使用 S4 作为 RNN 来一次生成一个 token。然而,S4 真正酷的地方在于你实际上也可以将它用作卷积神经网络(CNN)。在上面的例子中,让我们看看当我们展开之前的离散方程来尝试计算 h_3 时会发生什么。简单起见,我们假设 x_-1 = 0。

$$
\begin{aligned}
h_0 &= \bar{\mathbf{B}}x_0 \\
h_1 &= \bar{\mathbf{A}}(\bar{\mathbf{B}}x_0) + \bar{\mathbf{B}}x_1 \\
h_2 &= \bar{\mathbf{A}}(\bar{\mathbf{A}}(\bar{\mathbf{B}}x_0) + \bar{\mathbf{B}}x_1) + \bar{\mathbf{B}}x_2 \\
h_3 &= \bar{\mathbf{A}}(\bar{\mathbf{A}}(\bar{\mathbf{A}}(\bar{\mathbf{B}}x_0) + \bar{\mathbf{B}}x_1) + \bar{\mathbf{B}}x_2) + \bar{\mathbf{B}}x_3
\end{aligned}
$$

计算出 h_3 后,我们可以将其代入 y_3 的方程来预测下一个词。

$$
\begin{aligned}
y_3 &= \mathbf{C}(\bar{\mathbf{A}}(\bar{\mathbf{A}}(\bar{\mathbf{A}}(\bar{\mathbf{B}}x_0) + \bar{\mathbf{B}}x_1) + \bar{\mathbf{B}}x_2) + \bar{\mathbf{B}}x_3) \\
y_3 &= \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{A}}\bar{\mathbf{A}}\bar{\mathbf{B}}x_0 + \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{A}}\bar{\mathbf{B}}x_1 + \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{B}}x_2 + \mathbf{C}\bar{\mathbf{B}}x_3
\end{aligned}
$$

现在,注意到 y_3 实际上可以作为一个点积来计算,其中右侧向量就是我们的输入 u:

$$
y_3 = \begin{pmatrix} \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{A}}\bar{\mathbf{A}}\bar{\mathbf{B}} & \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{A}}\bar{\mathbf{B}} & \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{B}} & \mathbf{C}\bar{\mathbf{B}} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}
$$

由于 $\bar{A}$, $\bar{B}$ 和 C 都是常数,我们可以预先计算左侧向量并将其保存为我们的卷积核 K。这给我们留下了一种简单的方法来用卷积计算 y,如下方程所示(Mamba 论文中的 3a 和 3b):

$$
\begin{aligned}
\bar{\mathbf{K}} &= \begin{pmatrix} \mathbf{C}\bar{\mathbf{B}} & \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{B}} & \cdots & \mathbf{C}\bar{\mathbf{A}}^{\mathbf{k}}\bar{\mathbf{B}} \end{pmatrix} \\
y &= \bar{\mathbf{K}} * x
\end{aligned}
$$

重要的是,这些递归和卷积形式,我喜欢称之为"RNN 模式"和"CNN 模式",*在数学上是等价的*。这允许 S4 根据你的需求进行形态转换,而其输出没有任何差异。我们可以在 S4 论文的表 1 中比较这些"模式"之间的差异,该表显示了每种形式的训练和推理的运行时复杂度(粗体表示每个指标的最佳结果)。

	CONVOLUTION	RECURRENCE	S4
Training	$\tilde{\boldsymbol{L}}\boldsymbol{H}(\boldsymbol{B}+\boldsymbol{H})$	BLH^2	$\boldsymbol{BH}(\tilde{\boldsymbol{H}}+\tilde{\boldsymbol{L}})+\boldsymbol{B}\tilde{\boldsymbol{L}}\boldsymbol{H}$
Parallel	**Yes**	No	**Yes**
Inference	LH^2	$\boldsymbol{H^2}$	$\boldsymbol{H^2}$

注意,**CNN 模式更适合训练,而 RNN 模式更适合推理**。在 CNN 模式下,我们可以利用并行性一次性训练多个例子。在 RNN 模式下,虽然我们一次只能计算一步,但每一步需要的工作量完全相同。因为可以使用这两种模式, **S4 实现了两全其美:快速训练和更快的推理**。

创新点 1:选择性

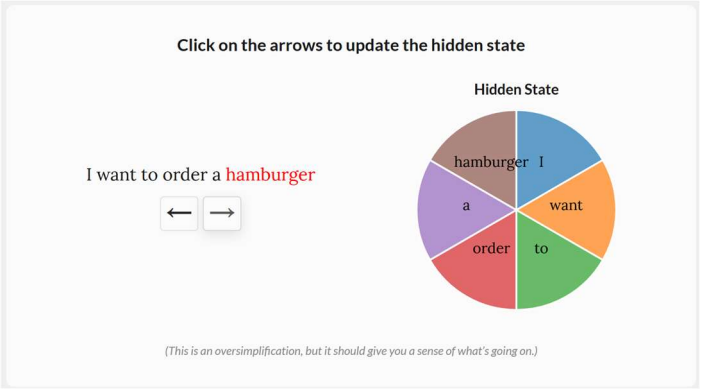
现在我们可以转向 Mamba 引入的第一个主要创新:**选择性**。让我们回顾一下定义 S4 离散形式的两个方程:

$$h_t = \bar{\mathbf{A}}h_{t-1} + \bar{\mathbf{B}}x_t$$
$$y_t = \mathbf{C}h_t$$

注意,在 S4 中,我们的离散参数 $\bar{A}, \bar{B}$ 和 C 是常数。然而,**Mamba 使 $\bar{A}, \bar{B}$ 和 C 这些参数根据输入而变化**。我们最终会得到类似这样的东西:

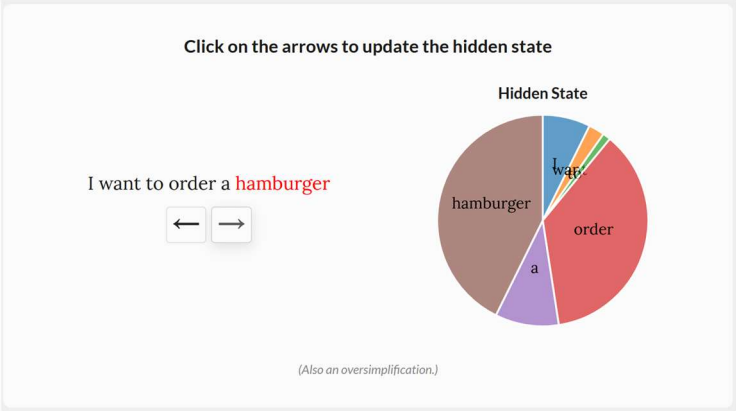
$$h_t = s_{\bar{\mathbf{A}}}(x_t)h_{t-1} + s_{\bar{\mathbf{B}}}(x_t)x_t$$
$$y_t = s_{\mathbf{C}}(x_t)h_t$$

作者认为,选择性或输入依赖性对许多任务很重要。我喜欢这样思考:因为 S4 没有选择性,它被迫完全相同地处理输入的所有部分。然而,当你在阅读一个句子时,总是会有些词比其他词更重要。想象我们有一个根据意图对句子进行分类的模型,我们给它一个句子:"I want to order a hamburger."没有选择性,S4 在处理每个词时花费相同的"努力"。点击下面的按钮,看看句子被一个词一个词处理时会发生什么。



(This is an oversimplification, but it should give you a sense of what's going on.)

但如果你是一个试图分类这个句子意图的模型,你可能想要更多地"关注"某些词而不是其他词。"want"和"to"这些词对这个句子的潜在含义真的贡献了多少价值?实际上,如果我们能够将有限的精力更多地花在像"order"这样的词上(知道用户想做什么),以及"hamburger"(知道用户在订什么)上,那就太好了。通过使模型参数成为输入的函数,Mamba 使得可以"关注"对手头任务更重要的输入部分。



(Also an oversimplification.)

然而,选择性给我们带来了一个问题。让我们回想一下我们之前计算的卷积核

$$\bar{\mathbf{K}} = \left(\mathbf{C}\bar{\mathbf{B}} \quad \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{B}} \quad \cdots \quad \mathbf{C}\bar{\mathbf{A}}^{\mathbf{k}}\bar{\mathbf{B}}\right)$$

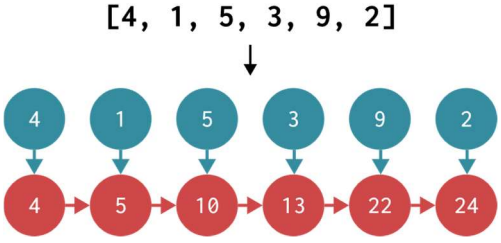
在 S4 中,我们可以预先计算这个核,保存它,然后将其与输入 u 相乘。这没问题,因为 $\bar{A}$, $\bar{B}$ 和 C 是常数。但是在 Mamba 中,这些矩阵会根据输入而变化!因此,我们无法预先计算 K,也就无法使用 CNN 模式来训练我们的模型。如果我们想要选择性,我们就需要用 RNN 模式来训练。我们可以为了戏剧性效果划掉方程 3b。

$$\cancel{y = \bar{\mathbf{K}} * x}$$

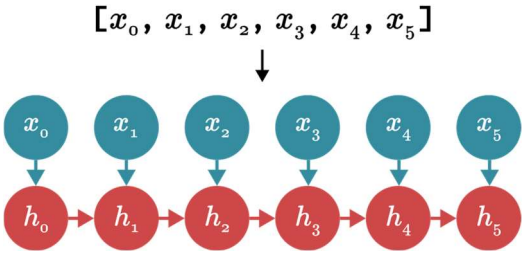
这为 Mamba 的作者们带来了一个问题:在 RNN 模式下训练真的很慢。想象我们正在用一个 1000 个 token 的序列训练我们的模型。CNN 基本上会计算其核与输入向量之间的点积,并且可以并行进行这些计算。相比之下,RNN 需要按顺序更新 1000 次其隐藏状态。RNN 这种缓慢的训练时间或多或少地阻碍了它们真正起飞,这也引导 Mamba 的作者们提出了他们的第二个大创新。

创新点 2:无需卷积的快速训练

Mamba 的第二个主要创新涉及非常非常快速地在 RNN 模式下训练。在某个时候,Gu 和 Dao 意识到他们的递归与扫描算法(也称为前缀和)非常相似。要计算前缀和,我们需要接受一个输入数组 x 并返回一个输出数组,其中每个元素是该项及其之前所有项的和。换句话说,输出的第一个元素将是 x_1,第二个元素将是 x_1 + x_2,第三个 x_1 + x_2 + x_3,依此类推。下面显示了一个例子。



现在让我们画出在 RNN 模式下更新 Mamba 隐藏状态的过程。



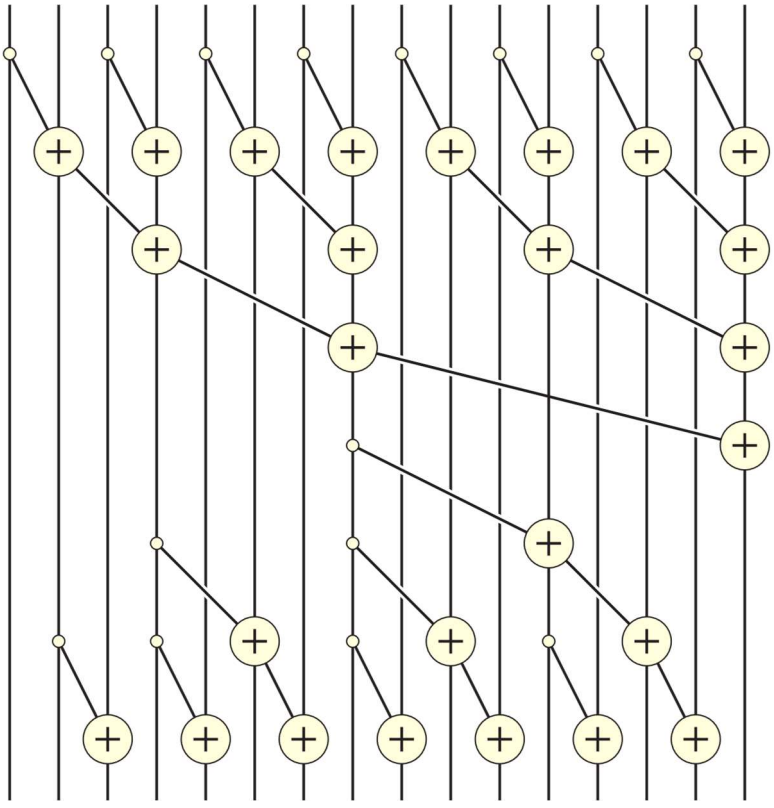
让我们思考一下。如果我们要形式化前缀和,我们可以将其写成以下方程:

$$h_t = h_{t-1} + x_t$$

这个方程形成了一个递归:在每一步,我们通过将前一个存储的值加到当前输入来计算新值。现在,让我们再看一下更新 Mamba 隐藏状态的递归。

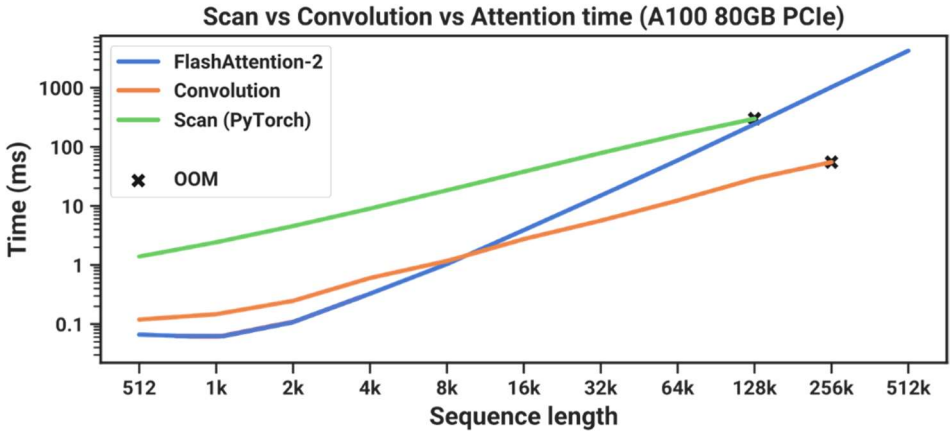
$$h_t = \bar{\mathbf{A}} h_{t-1} + \bar{\mathbf{B}} x_t$$

它们真的非常非常相似!而且这里有个很酷的部分:虽然计算前缀和在本质上可能看起来是顺序的,但我们实际上有高效的并行算法来完成这个任务!在下面的图中,我们可以看到一个并行前缀和算法是怎样工作的,其中每条垂直线代表我们数组中的一个项。



花点时间说服自己这个算法是有效的:选择任何一条垂直线,从顶部开始,向下工作,将每次加法追溯到数组的前几个项。当你到达底部时,你应该得到你的线左侧所有项的和。例如,你可以看到数组的第三个元素在最后接收到第二个元素的加值,在开始时第一个元素被加到第二个元素上。结果,当并行扫描完成时,第三个元素包含了第一、第二和第三个元素的和。

如果我们在单线程中运行这个算法,没有并行性,它会比我们按顺序加值更慢。但 GPU 有很多处理器,允许高度并行计算。因此,我们可以在大约 O(log n) 的时间内计算这个前缀和(或扫描)操作! Mamba 的作者们意识到,如果想在 RNN 模式下高效训练,他们可能可以使用并行扫描。由于 PyTorch 目前没有扫描实现,Mamba 的作者们自己写了一个,但结果并不理想。



在上图中,你可以看到他们基于 PyTorch 的扫描实现(绿色)总是比 FlashAttention-2(蓝色)慢,后者是最快可用的"精确注意力模型"的实现。在 128,000 个 token 长的序列下,扫描算法几乎赶上了运行时间,但它耗尽了内存。为了使 Mamba 实用,它需要更快。这让 Mamba 的作者们想到了 Dao 之前在 FlashAttention 上的工作。

Review: FlashAttention 回顾 FlashAttention

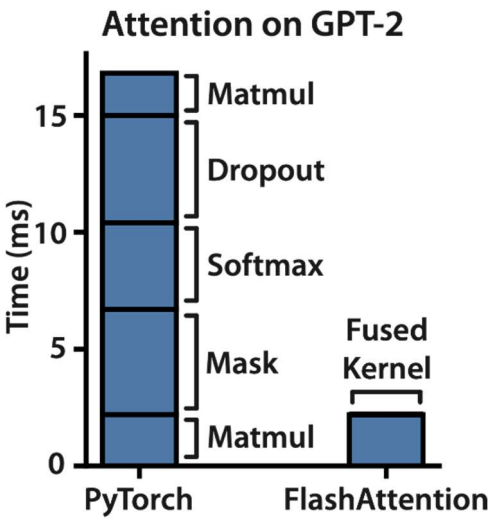
FlashAttention 是一个非常快速的注意力实现。发布时,FlashAttention 训练 BERT-large 的速度比之前最快的训练时间快 15%,比广泛使用的 HuggingFace 的 GPT-2 实现快 3 倍。简而言之,FlashAttention 的关键洞察与 GPU 上不同操作运行速度有关。他们意识到一些 GPU 操作是*计算受限*的,意味着它们受限于 GPU 执行计算的速度。然而,其他操作是*内存受限*的,意味着它们受限于 GPU 传输数据的速度。

想象你和一个朋友在玩一个游戏:你的朋友必须跑 50 米给你送两个数字,然后你需要手工将它们相乘。计时器在你的朋友开始跑时启动,在你得到答案时结束。假设你需要相乘的数字是 439,145,208 和 142,426,265。你需要一段时间才能手工相乘这些数字。你的朋友可能需要 5 秒钟来传递数字,但你(在计算器上, 译注)可能需要 60 秒钟来进行乘法。因此,你们都是计算受限的,因为大部分时间都花在了计算上。现在,想象你需要相乘的数字是 4 和 3。虽然你的朋友仍然需要 5 秒钟跑 50 米,但你(在计算器上, 译注)可以立即计算出这个结果。现在,你们都是内存受限的,因为大部分时间都花在了传输数据上。

在这个类比中,你的 GPU 本质上是在争先恐后地将数据移动到正确的位置以执行计算。例如,让我们考虑一个掩码操作。要计算掩码向量,你的 GPU 只需要在掩码等于 0 时擦除数据值(当它等于 1 时保持不变)。如果我们用 $\odot$ 表示掩码操作,一个例子如下,其中掩码强制我们将最后三个数据元素设置为零:

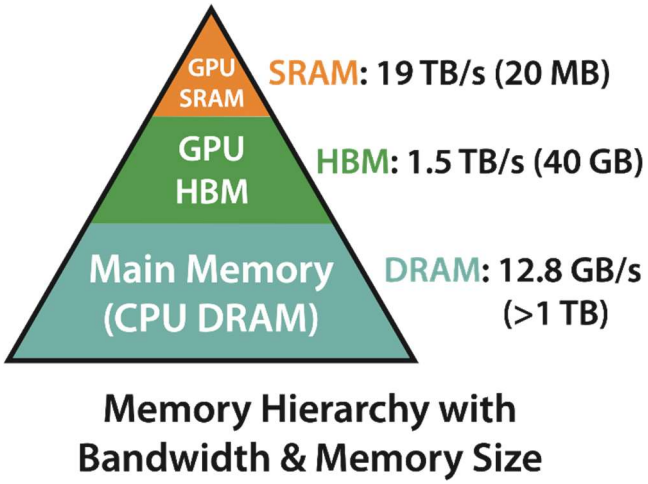
$$(4 \quad 9 \quad 4 \quad 1 \quad 2 \quad 7) \odot (1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0) = \boxed{(4 \quad 9 \quad 4 \quad 0 \quad 0 \quad 0)}$$

由于这非常容易计算,你的 GPU 最终大部分时间都花在传输内存上,将数据和掩码矩阵移动到正确的位置进行计算。这意味着*掩码是内存受限的*。另一方面,矩阵乘法涉及大量的加法和乘法。因为在计算上花费的时间比内存传输多得多,所以*矩阵乘法是计算受限的*。考虑到这一点,让我们看一下在注意力计算期间执行的计算的细分(下面的 matmul 是矩阵乘法)。



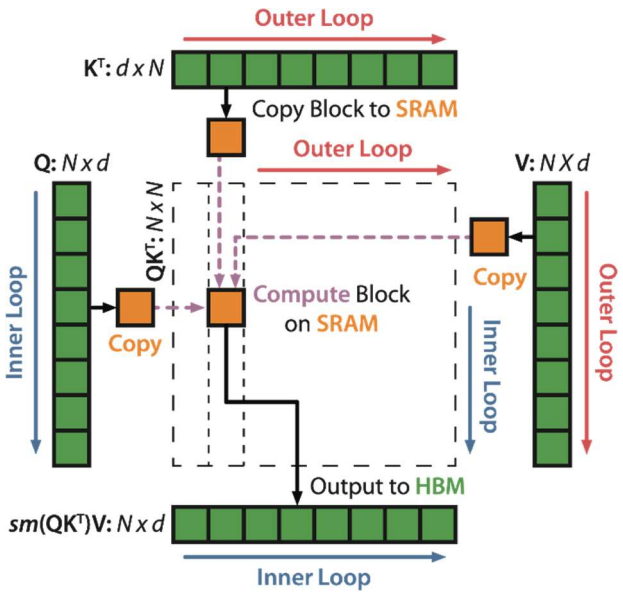
事实证明,dropout、softmax 和掩码,它们构成了注意力运行时间的大部分,都是内存受限的。这意味着我们在计算注意力时花费的大部分时间只是在等你的 GPU 移动数据。考虑到这一点, FlashAttention 的作者们想知道,我们如何加速那些受内存传输速度限制的操作?

这让他们得出了另一个关键认识:GPU 内存有两个主要区域。其中一个,高带宽内存(HBM),非常大,但非常慢。另一个,静态随机存取存储器(SRAM),非常小,但非常快。让我们分解一下 A100 GPU 上这些区域之间的差异:



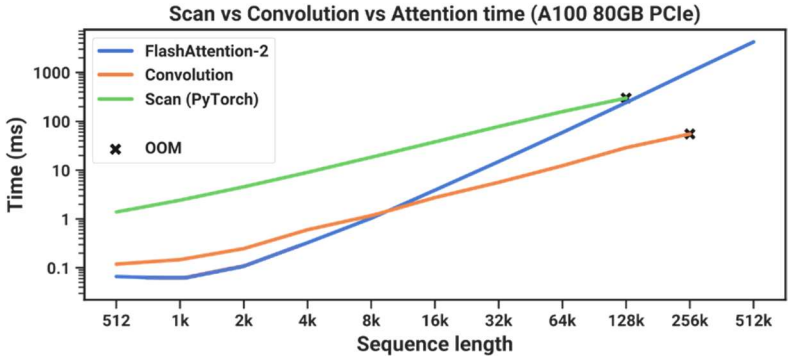
FlashAttention 的作者们意识到,如果你特别小心地使用这些 GPU 内存区域,你可以更有效地计算内存受限的操作。他们使用了一种称为 tiling 的方法,其中数据的小部分从 HBM(较慢)移动到 SRAM(较快),在 SRAM 中计算,然后再从 SRAM 移回 HBM。这使得 FlashAttention 非常非常快,同时在数值上仍然等同于注意力。

相关的工作细节非常有趣,我鼓励你查看 FlashAttention 论文以了解更多。然而,为了理解 Mamba,这基本上就是你需要知道的全部内容。

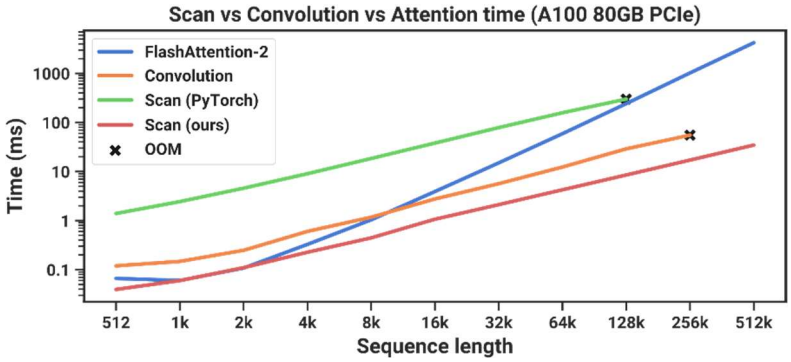


Back to Mamba　回到 Mamba

请记住,在我们开始这个关于 FlashAttention 的题外话之前,我们正试图加速我们的并行扫描实现。这里是之前的同一张图,我们可以看到 PyTorch 中的扫描实现(绿色)总是比 FlashAttention 慢,后者是最快的"精确"Transformer(蓝色)。



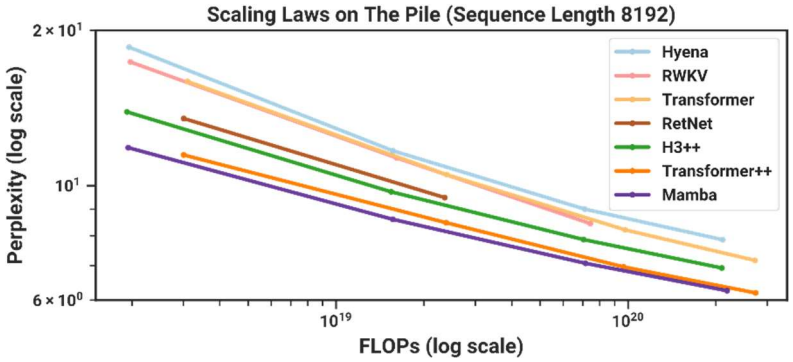
事实证明,如果你在计算扫描时采用这种内存感知的 tiling 方法,你可以大大加快速度。有了这个优化,Mamba(红色)现在在所有序列长度上都比 FlashAttention-2(蓝色)快。



这些结果表明,就速度而言,Mamba 是实用的,运行速度比最快的精确 Transformer 还要快。但它在语言建模方面表现如何呢?

Results

Gu 和 Dao 在一些涉及语言、基因组学和音频的序列建模任务上评估了 Mamba。我对后两个领域不太熟悉,但结果看起来很酷:Mamba 在对人类基因组项目的 DNA 建模和钢琴音乐数据集的音频建模方面建立了最先进的性能。然而,正是语言结果让许多人感到兴奋。关于 Mamba 的大量在线讨论都集中在下面的这张图上(图 4)。



在这个图中,模型大小向右增加,语言建模性能向下改善。这意味着最好的模型应该在左下方:小(它们更快速),同时也非常擅长建模语言。由于 Gu 和 Dao 是学者,他们没有数千个 GPU 可用来训练 GPT-4 大小的模型,所以他们通过训练一堆较小的模型(大约 125M 到 1.3B 参数)来进行这个比较。如上图所示,结果看起来非常有希望。与类似大小的其他模型相比,Mamba 似乎最擅长建模语言。

What next?

我认为 Mamba 以一种相当独特和有趣的方式创新了语言建模!不幸的是,一些审稿人并不同意:Gu 和 Dao 计划在 5 月的 ICLR 会议上展示 Mamba,但他们的论文几周前被拒绝了。我猜 Gu 和 Dao 现在正在研究论文的下一个版本,我也想象一些拥有比他们知道如何使用更多 GPU 的公司目前正试图弄清楚 Mamba 的性能是否在更大的模型规模上依然有效。如果它们能够展示良好的性能,像 Mamba 这样的线性时间模型有朝一日可能会提供一个答案。

注释:

1. 像 Gemini 1.5 这样更快的 Transformer 几乎肯定在使用注意力修改,例如 RingAttention、StreamingLLM、Linear Attention。

2. CNN 翻转核来执行卷积,这就是为什么 K 看起来与我们推导 K 时的左手向量相反。

3. 在这个表中,L 表示序列长度,B 表示批量大小,N 表示模型的隐藏大小,波浪号表示对数因子。对于这篇博文的目的,不要太关注数学。

4. 实际上比这更复杂一些:连续的 A 是常数,而我们的离散化参数 Δ 是输入依赖的。因此,$\bar{A}$ 由于离散化而是输入依赖的。

5. Mamba 的递归和前缀和之所以"相似",是因为重要的是,Mamba 的递归是其输入的线性变换。这对 RNN 不成立,这就是为什么我们不能使用并行扫描来训练 RNN。

6, 7. 如果你读了注释 1,请注意 FlashAttention/FlashAttention-2 是一种不同类型的注意力修改,因为与那些例子不同,FlashAttention 在数值上等同于标准注意力。它更快,但产生完全相同的输出。FlashAttention 的作者将此称为计算"精确注意力"。

8. 困惑度,显示在 y 轴上,是语言建模性能的常见度量。如果给你一个句子的前半部分,并要求你预测下一个词,你可以把困惑度看作是一个值,表示当你看到正确答案时你有多"困惑"。例如,如果给你序列"I went for a walk outside",下一个词是"today"时你不应该太惊讶。较低的值表示你较少困惑,因此对语言如何工作有更好的理解。