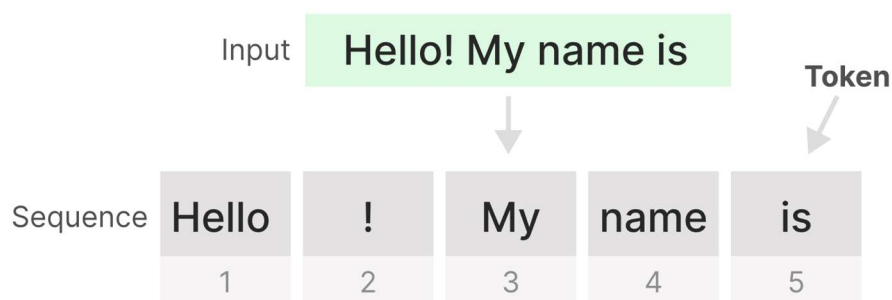


Transformer 架构是大型语言模型(LLMs)成功的主要组成部分。它被用于几乎所有当今使用的 LLMs,从开源模型如 Mistral 到闭源模型如 ChatGPT。为了进一步改进 LLMs,新的架构正在被开发,可能甚至会超越 Transformer 架构。其中一种方法是 Mamba,一种状态空间模型。Mamba 模型在论文《Mamba: 具有选择性状态空间的线性时间序列建模》中被提出。你可以在其仓库中找到官方实现。

在这篇文章中,我将在语言建模的背景下介绍状态空间模型领域,并逐一探讨相关概念以建立对该领域的直观认识。还将讨论 Mamba 如何挑战 Transformer 架构。

第一部分:Transformer 的问题

为了说明为什么 Mamba 是如此有趣的架构,让我们先简要回顾一下 Transformer,并探讨它的一个缺点。Transformer 将任何文本输入视为由标记组成的序列。



Transformer 的一个主要优势是,无论接收到什么输入,它都可以回顾序列中任何早期的标记来推导其表示。

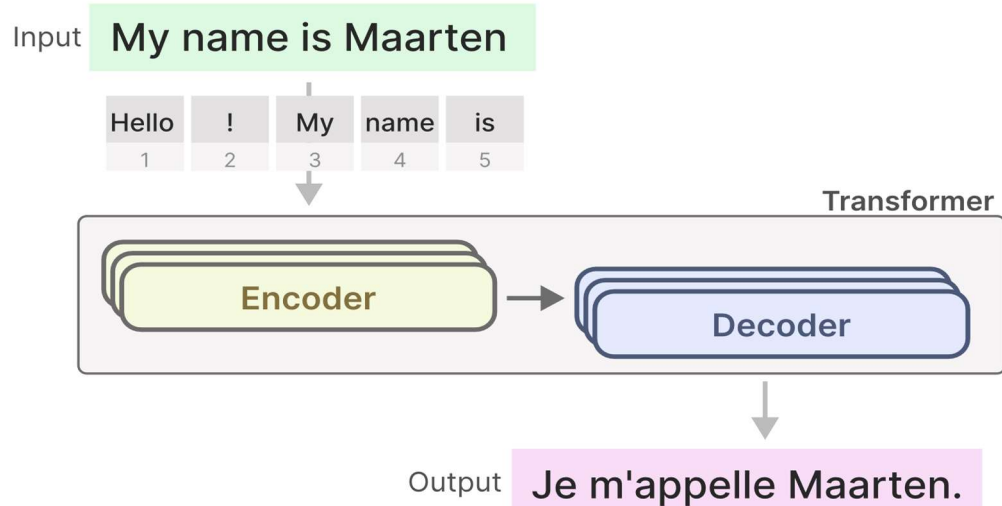
Transformer is capable of **selectively** and **individually** looking at **past tokens**.

Hello! My name is

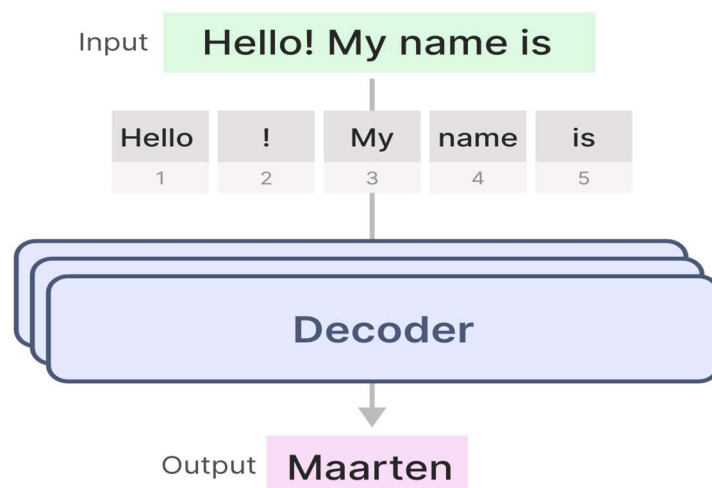
Hello! My name is

Transformer 的核心组件

请记住,Transformer 由两个结构组成,一组用于表示文本的编码器块和一组用于生成文本的解码器块。这些结构可以一起用于多个任务,包括翻译。



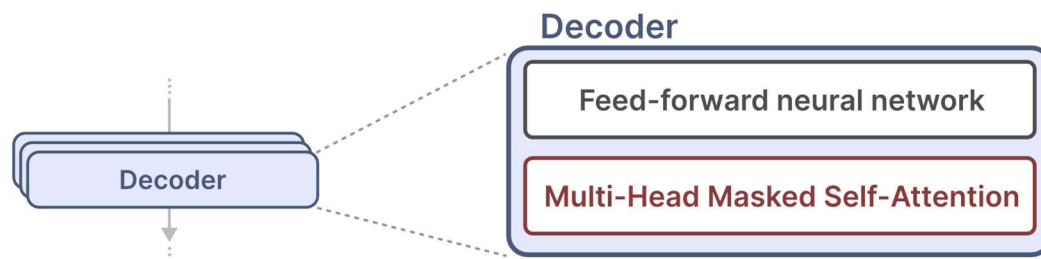
我们可以采用这种结构来创建生成模型,只使用解码器。这种基于 Transformer 的模型,即生成式预训练 Transformer(GPT),使用解码器块来完成一些输入文本。



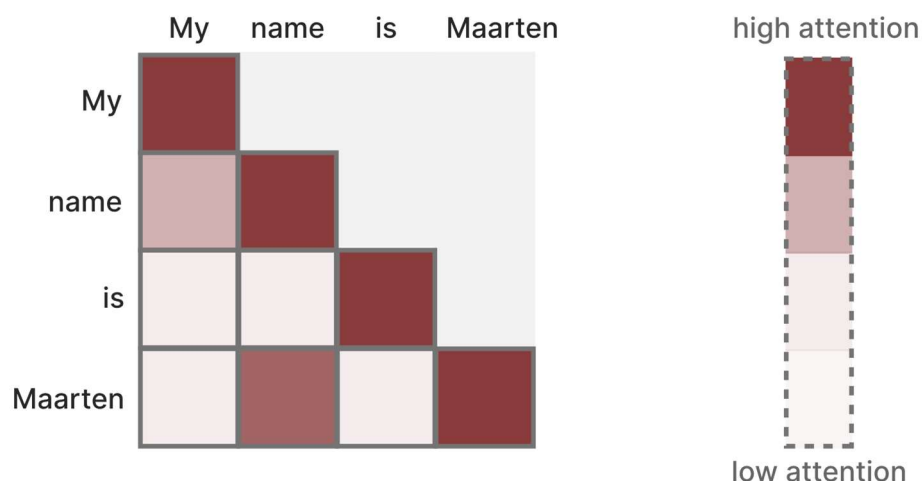
让我们看看它是如何工作的!

训练的祝福

单个解码器块由两个主要组件组成,掩码自注意力后跟前馈神经网络。



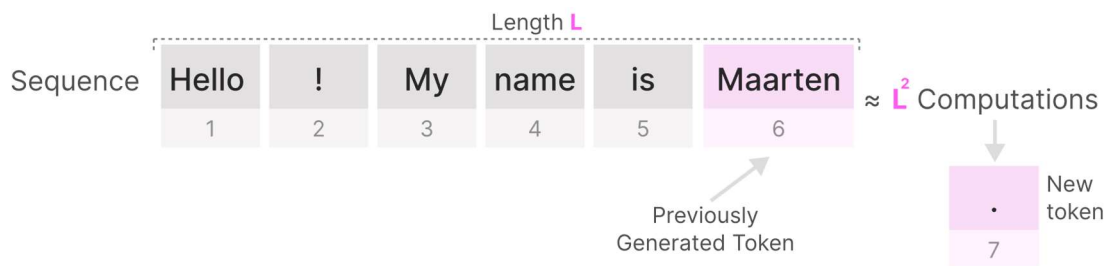
自注意力是这些模型如此有效的主要原因。它实现了对整个序列的未压缩视图,并能快速训练。那么它是如何工作的呢? 它创建了一个矩阵,比较每个标记与之前的每个标记。矩阵中的权重由标记对之间的相关性决定。



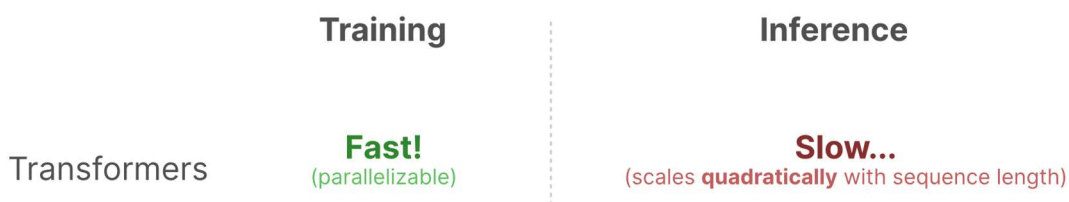
在训练过程中,这个矩阵是一次性创建的。"My"和"name"之间的注意力不需要先计算,然后再计算"name"和"is"之间的注意力。它实现了**并行化**,这极大地加快了训练速度!

推理的诅咒!

然而,这里有一个缺陷。在生成下一个标记时,我们需要重新计算整个序列的注意力,即使我们已经生成了一些标记。



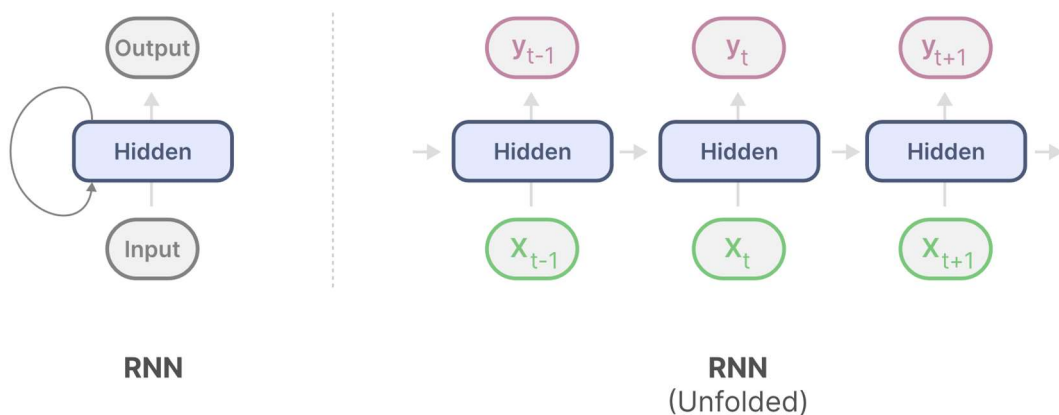
为长度为 L 的序列生成标记大约需要 L^2 次计算,如果序列长度增加,这可能会代价高昂。



这种需要重新计算整个序列的做法是 Transformer 架构的主要瓶颈。让我们看看"经典"技术递归神经网络是如何解决这个缓慢推理的问题的。

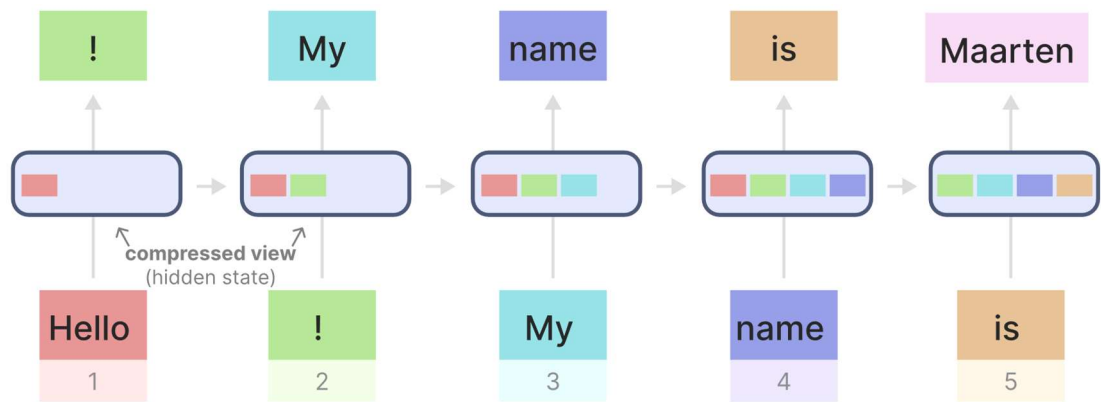
RNN 是解决方案吗?

递归神经网络(RNN)是一种基于序列的网络。在序列中的每个时间步骤,它接受两个输入,即时间步 t 的输入和前一个时间步 $t-1$ 的隐藏状态,以生成下一个隐藏状态并预测输出。RNN 有一个循环机制,允许它将信息从前一步传递到下一步。我们可以"展开"这个可视化以使其更加明确。



在生成输出时,RNN 只需要考虑前一个隐藏状态和当前输入。它避免了重新计算所有先前的隐藏状态,这是 Transformer 会做的。

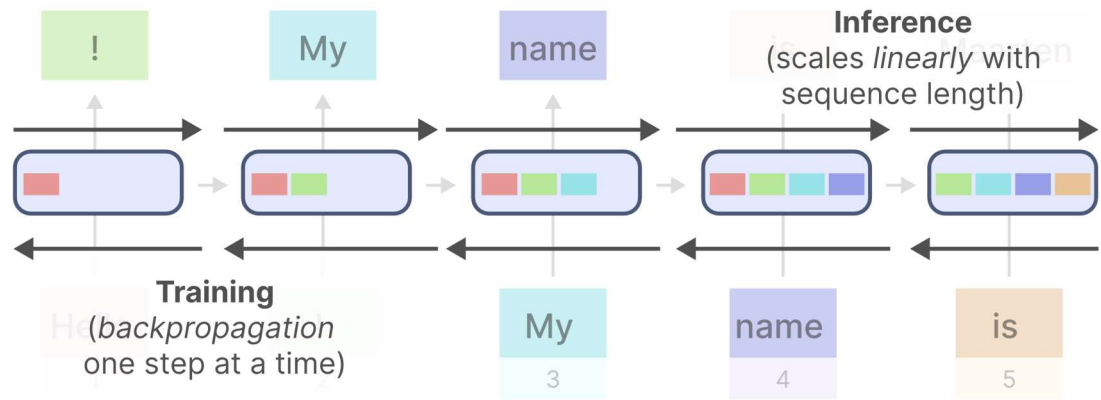
换句话说,RNN 可以快速进行推理,因为它与序列长度呈线性关系!理论上,它甚至可以有无数的上下文长度。为了说明,我们将 RNN 应用于之前使用的输入文本。



每个隐藏状态都是所有先前隐藏状态的聚合,通常是一个压缩视图。

然而,这里有一个问题...

注意,在产生名字"Maarten"时,最后的隐藏状态不再包含关于"Hello"这个词的信息。RNN 往往会随着时间忘记信息,因为它们只考虑一个先前的状态。这种顺序性质造成了一个问题。训练无法并行进行,因为它需要按顺序一次经过每个步骤。



与 Transformer 相比,RNN 的问题完全相反!它的推理速度非常快,但无法并行化。

	Training	Inference
Transformers	Fast! (parallelizable)	Slow... (scales quadratically with sequence length)
RNNs	Slow... (not parallelizable)	Fast! (scales linearly with sequence length)

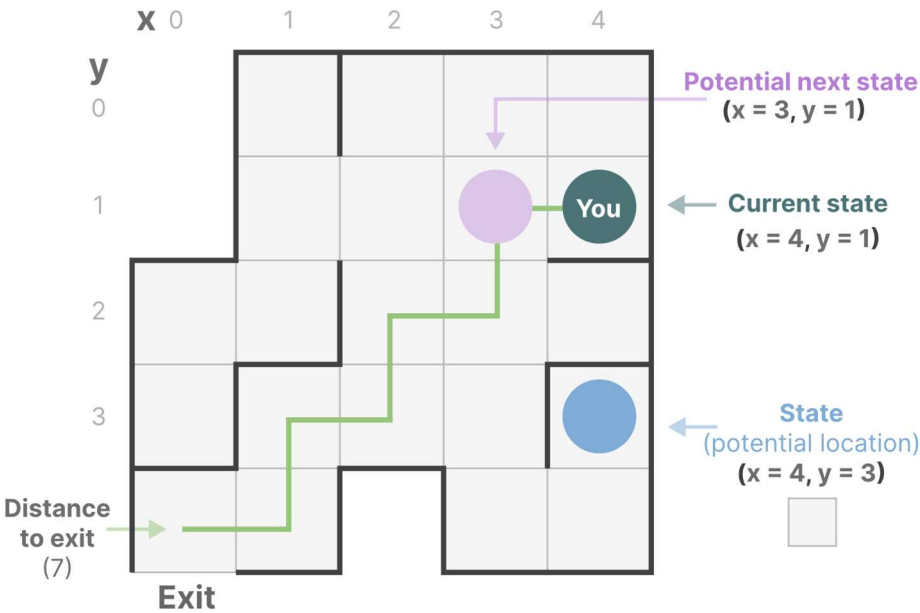
我们能否找到一种架构,既能像 Transformer 那样并行化训练,又能执行与序列长度呈线性关系的推理? 是的!这就是 Mamba 提供的,但在深入研究其架构之前,让我们先探索状态空间模型的世界。

第二部分:状态空间模型(SSM)

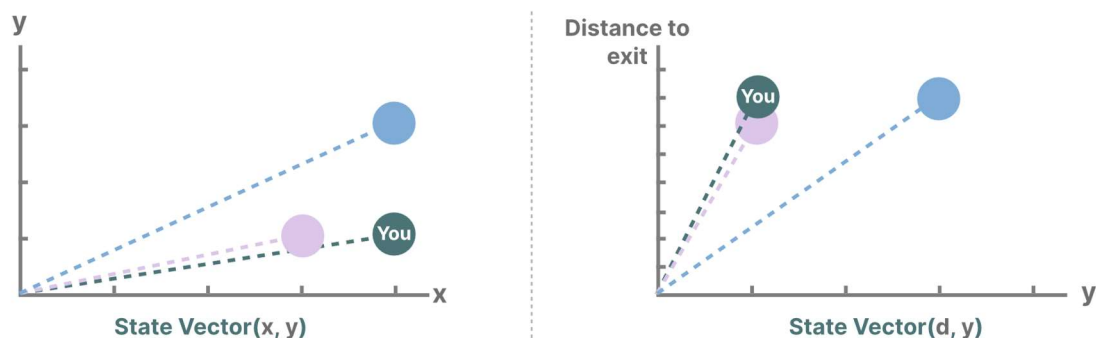
状态空间模型(SSM),像 Transformer 和 RNN 一样,处理信息序列,如文本但也包括信号。在本节中,我们将讨论 SSM 的基础知识以及它们与文本数据的关系。

什么是状态空间? 状态空间包含完全描述系统的最小变量数。它是通过定义系统的可能状态来数学地表示问题的一种方式。让我们简化一下。想象我们正在穿越迷宫。"状态空间"是所有可能位置(状态)的地图。每个点代表迷宫中的一个独特位置,具有特定的细节,比如你距离出口有多远。

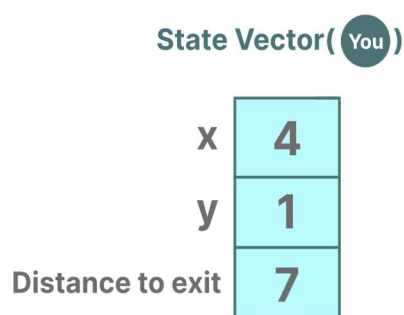
"状态空间表示"是这个地图的简化描述。它显示你在哪里(当前状态),你下一步可以去哪里(可能的未来状态),以及什么变化会让你到达下一个状态(向右或向左走)。



虽然状态空间模型使用方程和矩阵来跟踪这种行为,但它只是一种跟踪你在哪里,你可以去哪里,以及如何到达那里的方法。描述状态的变量,在我们的例子中是 X 和 Y 坐标,以及到出口的距离,可以表示为"状态向量"。



听起来熟悉吗?这是因为语言模型中的嵌入或向量也经常用来描述输入序列的"状态"。例如,你当前位置(状态向量)的向量可能看起来有点像这样:



在神经网络方面,"系统"的"状态"通常是其隐藏状态,在大型语言模型的背景下,这是生成新标记的最重要方面之一。

什么是状态空间模型?

SSM 是用来描述这些状态表示并根据某些输入预测其下一个状态的模型。

传统上,在时间 t ,SSM:

- 将输入序列 $x(t)$ 映射 - (例如,在迷宫中向左和向下移动)
- 到潜在状态表示 $h(t)$ - (例如,到出口的距离和 x/y 坐标)
- 并导出预测输出序列 $y(t)$ - (例如,再次向左移动以更快到达出口)

然而,SSM 不是使用离散序列(比如向左移动一次),而是将连续序列作为输入并预测输出序列。

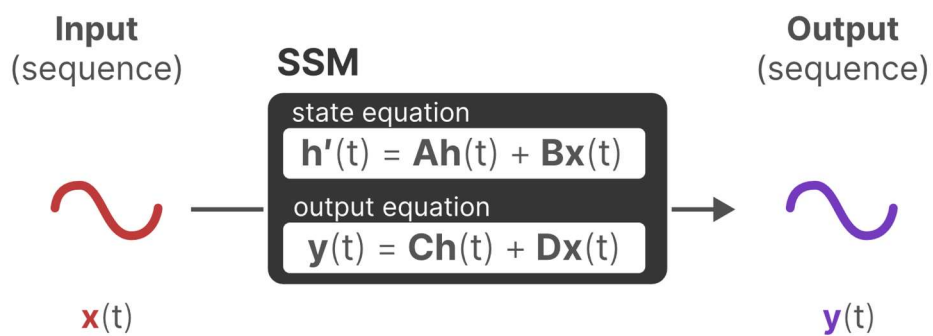


SSM 假设动态系统,如在 3D 空间中移动的物体,可以通过两个方程从其在时间 t 的状态预测。

State equation $\mathbf{h}'(t) = \mathbf{A}\mathbf{h}(t) + \mathbf{B}\mathbf{x}(t)$

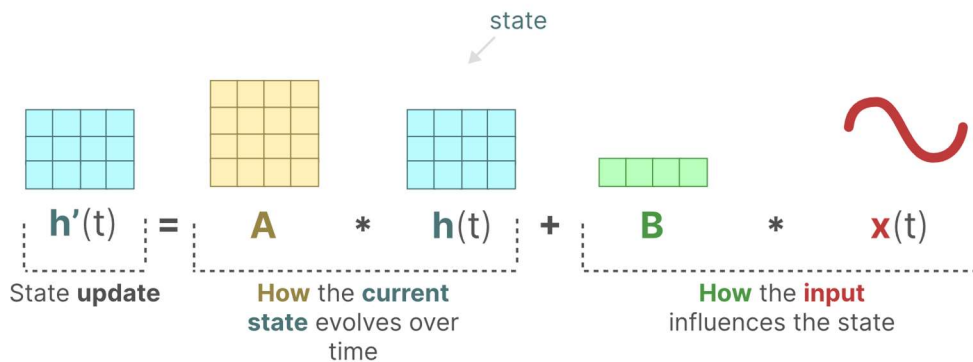
Output equation $\mathbf{y}(t) = \mathbf{C}\mathbf{h}(t) + \mathbf{D}\mathbf{x}(t)$

通过解这些方程,我们假设可以揭示统计原理,根据观察到的数据(输入序列和先前状态)预测系统的状态。其目标是找到这个状态表示 $\mathbf{h}(t)$,使我们可以从输入到输出序列。

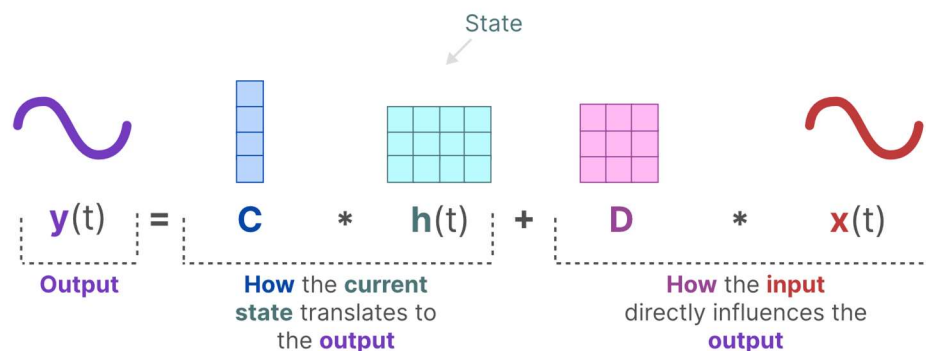


这两个方程是状态空间模型的核心。这两个方程将在本指南中多次被提及。为了使它们更直观,它们被颜色编码,以便你可以快速参考。

状态方程描述了状态如何变化(通过矩阵 \mathbf{A}),基于输入如何影响状态(通过矩阵 \mathbf{B})。

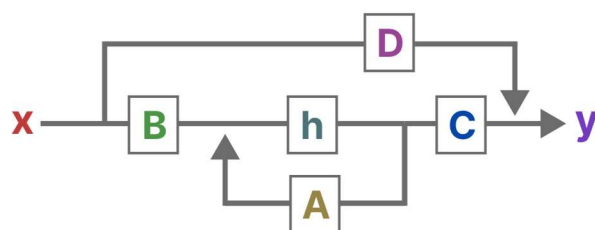


如前所述, $h(t)$ 指的是我们在任何给定时间 t 的潜在状态表示, $x(t)$ 指的是某些输入。输出方程描述了状态如何转换为输出(通过矩阵 C)以及输入如何影响输出(通过矩阵 D)。

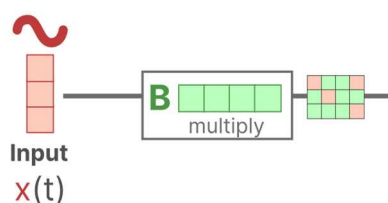


注意: 矩阵 A , B , C 和 D 也常被称为参数,因为它们是可学习的。

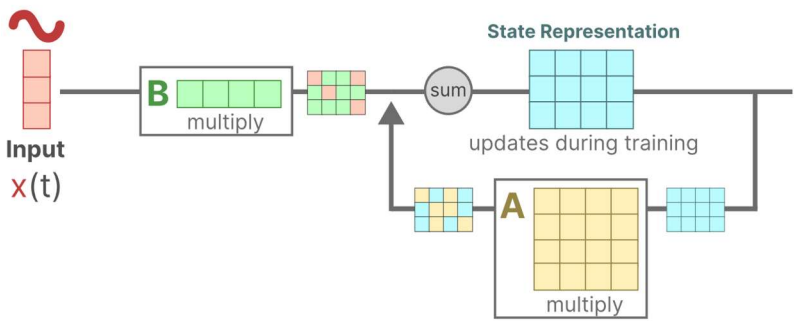
将这两个方程可视化给我们以下架构:



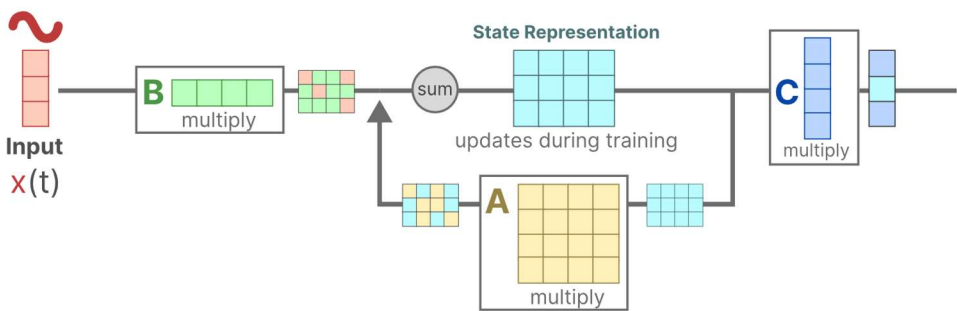
让我们逐步了解这种一般技术,以理解这些矩阵如何影响学习过程。假设我们有一些输入信号 $x(t)$,这个信号首先乘以矩阵 B ,它描述了输入如何影响系统。



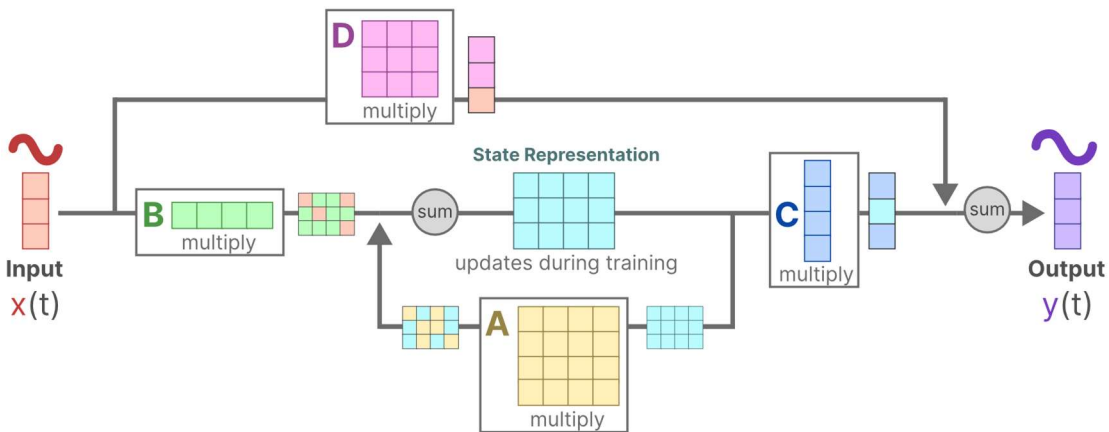
更新后的状态(类似于神经网络的隐藏状态)是一个包含环境核心"知识"的潜在空间。我们将状态乘以矩阵 A,它描述了所有内部状态如何连接,因为它们代表系统的基本动态。



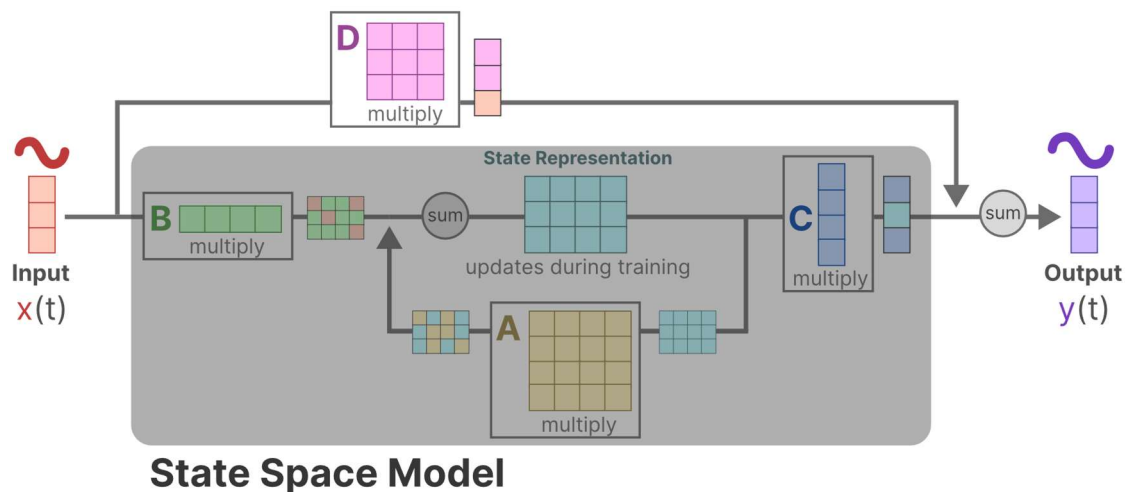
正如你可能注意到的,矩阵 A 在创建状态表示之前应用,并在状态表示更新后进行更新。然后,我们使用矩阵 C 来描述状态如何转换为输出。



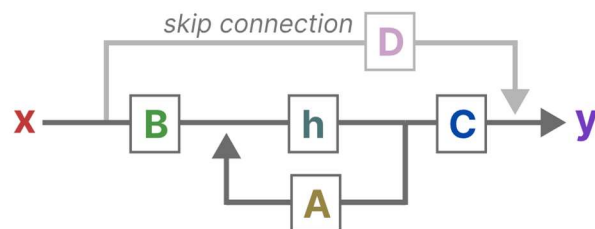
最后,我们利用矩阵 D 提供从输入到输出的直接信号。这也经常被称为跳跃连接。



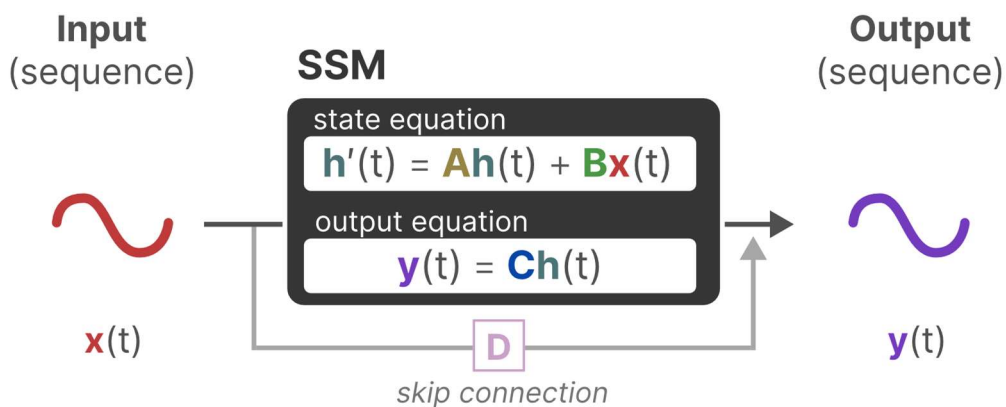
由于矩阵 D 类似于跳跃连接,SSM 通常被视为没有跳跃连接的以下形式::



回到我们简化的视角,我们现在可以专注于矩阵 A, B 和 C 作为 SSM 的核心。



我们可以更新原始方程(并添加一些漂亮的颜色)来表示每个矩阵的目的,就像我们之前做的那样。

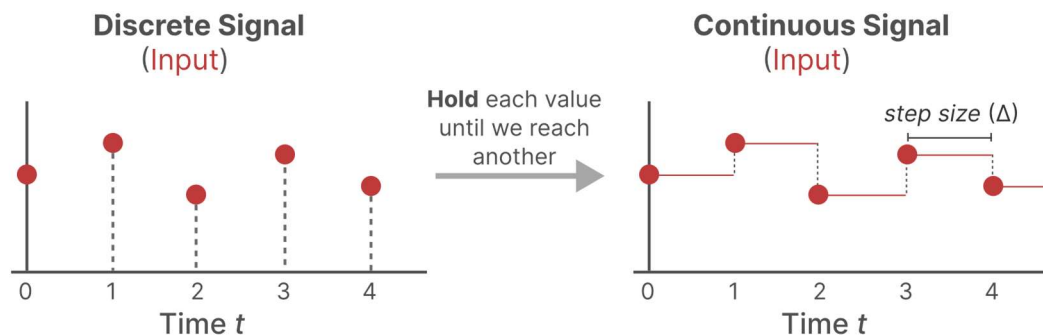


这两个方程一起旨在从观察数据预测系统的状态。由于输入预期是连续的,SSM 的主要表示是连续时间表示。

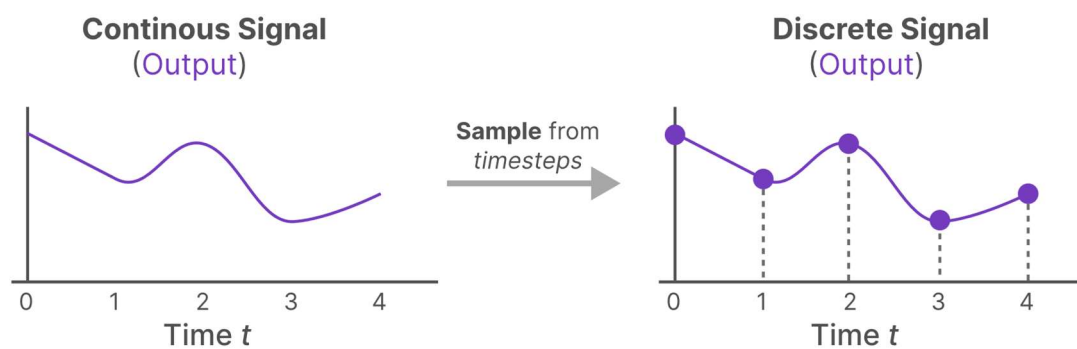
从连续到离散信号

如果你有一个连续信号,找到状态表示 $h(t)$ 在分析上是具有挑战性的。此外,由于我们通常有一个离散输入(如文本序列),我们想要离散化模型。为此,我们使用零阶

保持技术。它的工作原理如下。首先,每次我们收到一个离散信号,我们保持其值直到我们收到一个新的离散信号。这个过程创建了 SSM 可以使用的连续信号:



我们保持值的时间由一个新的可学习参数表示,称为步长 Δ 。它表示输入的分辨率。现在我们有输入连续信号,我们可以生成一个连续输出,并仅根据输入的时间步长对值进行采样。

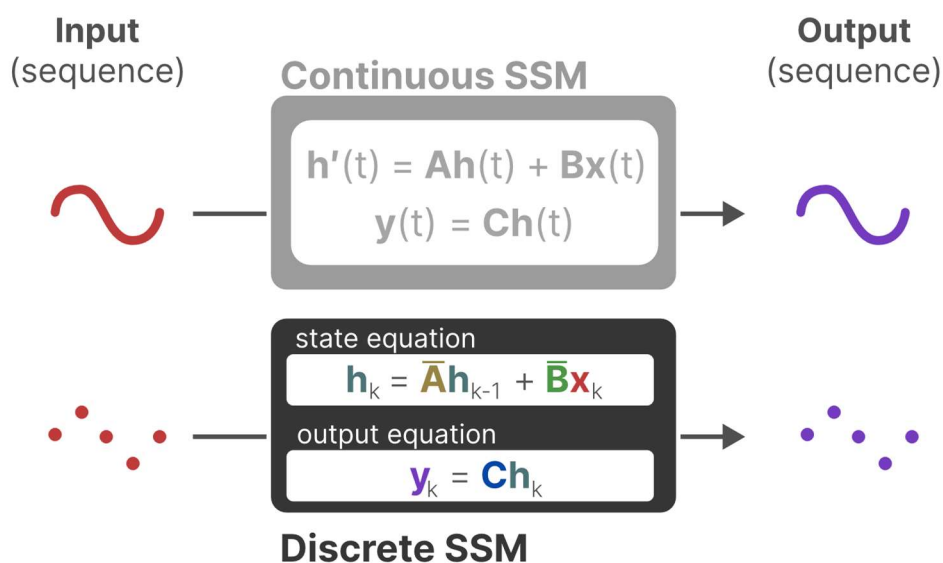


这些采样值就是我们的离散化输出! 数学上,我们可以按以下方式应用零阶保持:

Discretized matrix **A** $\bar{\mathbf{A}} = \exp(\Delta \mathbf{A})$

Discretized matrix **B** $\bar{\mathbf{B}} = (\Delta \mathbf{A})^{-1} (\exp(\Delta \mathbf{A}) - \mathbf{I}) \cdot \Delta \mathbf{B}$

它们一起允许我们从连续 SSM 转到离散 SSM,表示为一个序列到序列的公式, $x_k \rightarrow y_k$,而不是函数到函数, $x(t) \rightarrow y(t)$:



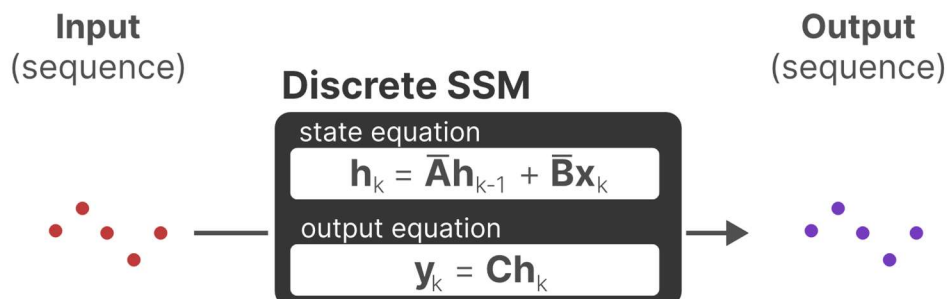
这里,矩阵 A 和 B 现在代表模型的离散化参数。我们使用 k 而不是 t 来表示离散的时间步,并使它更清楚何时我们指的是连续 SSM 与离散 SSM。

注意:在训练期间,我们仍然保存矩阵 A 的连续形式,而不是离散版本。在训练期间,连续表示被离散化。

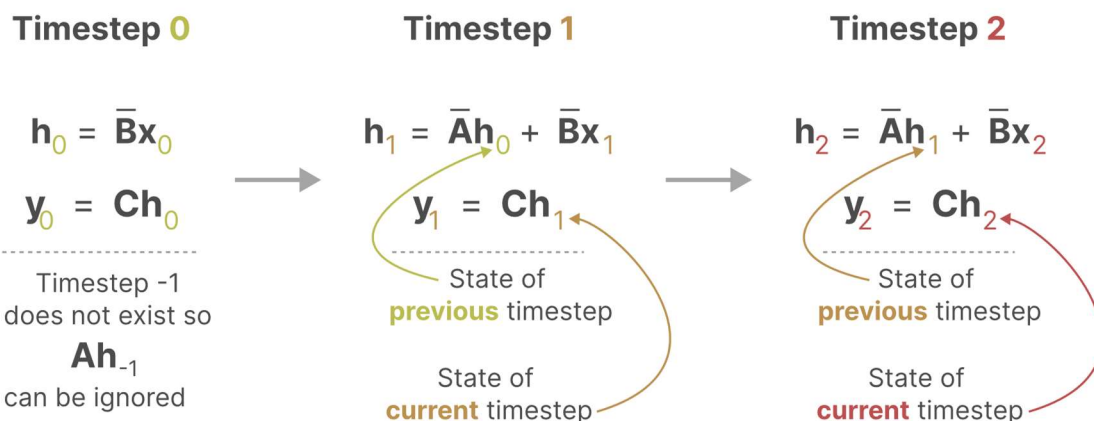
现在我们有了离散表示的公式,让我们探索如何实际计算模型。

递归表示

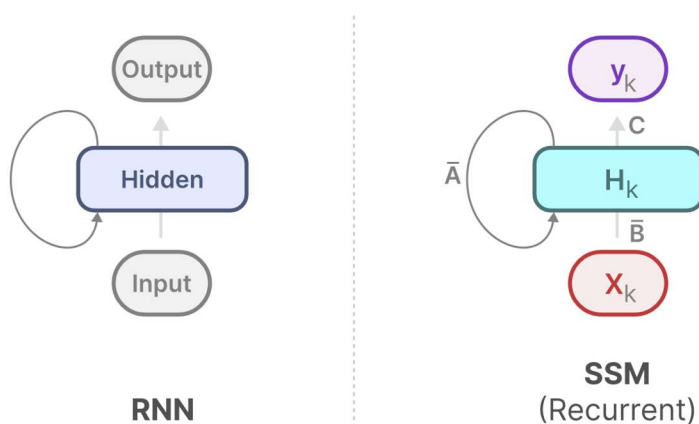
我们的离散化 SSM 允许我们以特定时间步而不是连续信号来制定问题。如我们之前在 RNN 中看到的那样,递归方法在这里很有用。如果我们考虑离散时间步而不是连续信号,我们可以用时间步重新表述问题:



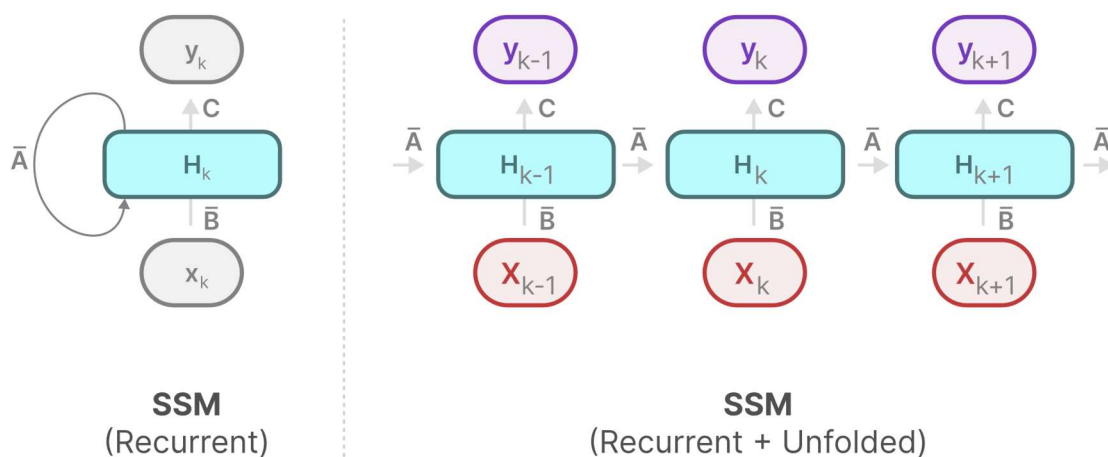
在每个时间步,我们计算当前输入(Bx_k)如何影响先前状态(Ah_{k-1}),然后计算预测输出(Ch_k)。



这个表示可能已经看起来有点熟悉!我们可以像之前处理 RNN 那样处理它。



我们可以展开(或展开)如下:

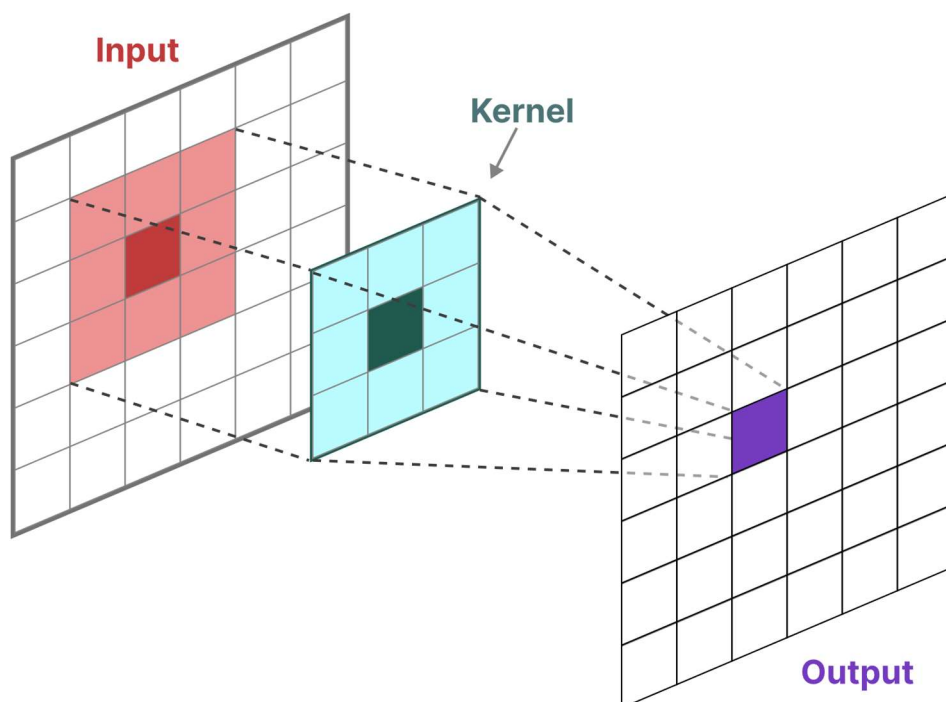


注意我们如何使用 RNN 的基本方法来使用这个离散版本。这种技术给我们带来了 RNN 的优点和缺点,即快速推理和缓慢训练。

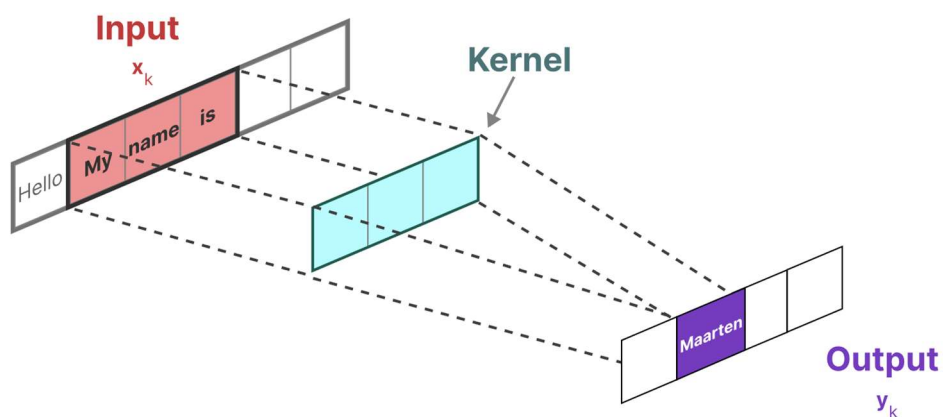
卷积表示

我们可以用于 SSM 的另一种表示是卷积。记得从经典的图像识别任务中,我们应

用过滤器(核)来导出聚合特征:



由于我们处理的是文本而不是图像,我们需要一个一维视角:



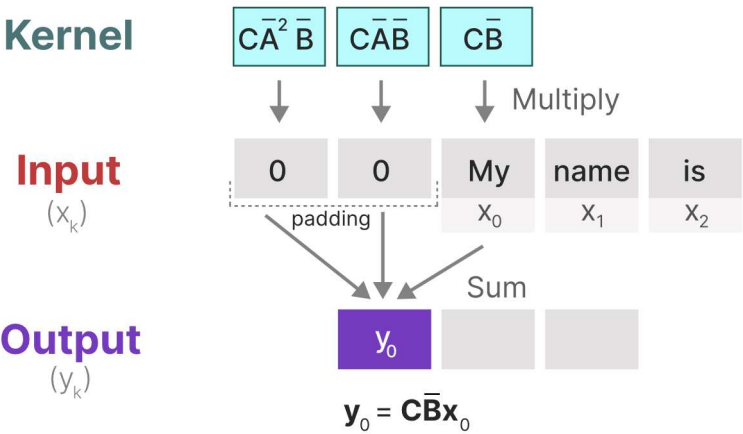
我们用来表示这个"过滤器"的核是从 SSM 公式推导出来的:

$$\text{kernel} \rightarrow \bar{\mathbf{K}} = (\bar{\mathbf{C}}\bar{\mathbf{B}}, \bar{\mathbf{C}}\bar{\mathbf{A}}\bar{\mathbf{B}}, \dots, \bar{\mathbf{C}}\bar{\mathbf{A}}^{k-1}\bar{\mathbf{B}}, \dots)$$

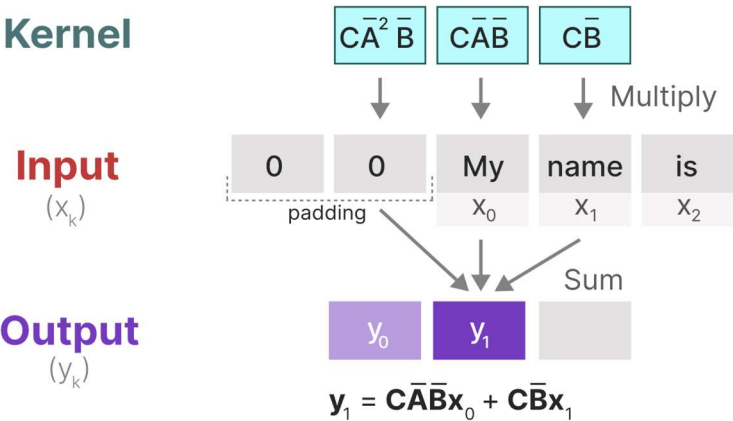
$$\mathbf{y} = \mathbf{x} * \bar{\mathbf{K}}$$

↑output ↑input ↑kernel

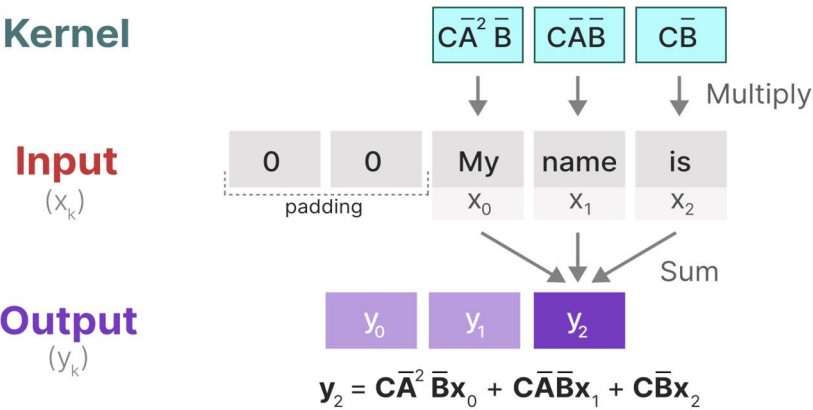
让我们探索这个核在实践中如何工作。像卷积一样,我们可以使用我们的 SSM 核来遍历每组标记并计算输出:



这也说明了填充可能对输出产生的影响。我改变了填充的顺序以改善可视化,但我们通常在句子末尾应用它。在下一步中,核向前移动一次以执行计算的下一步:



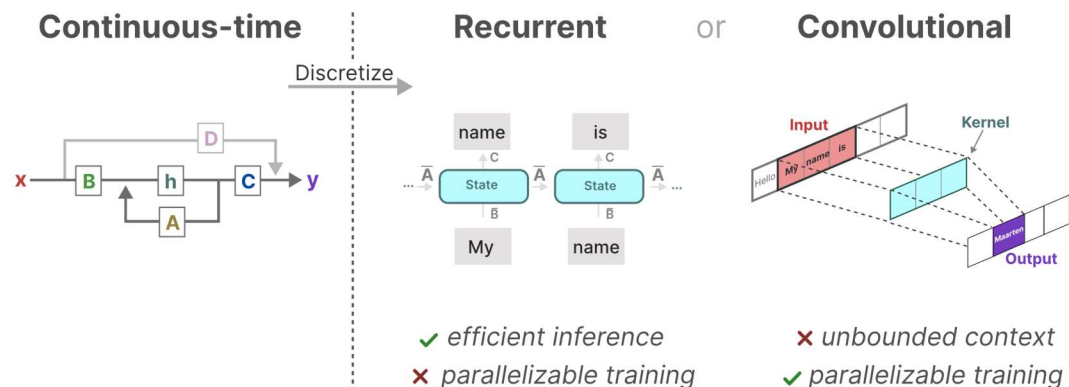
在最后一步,我们可以看到核的完整效果:



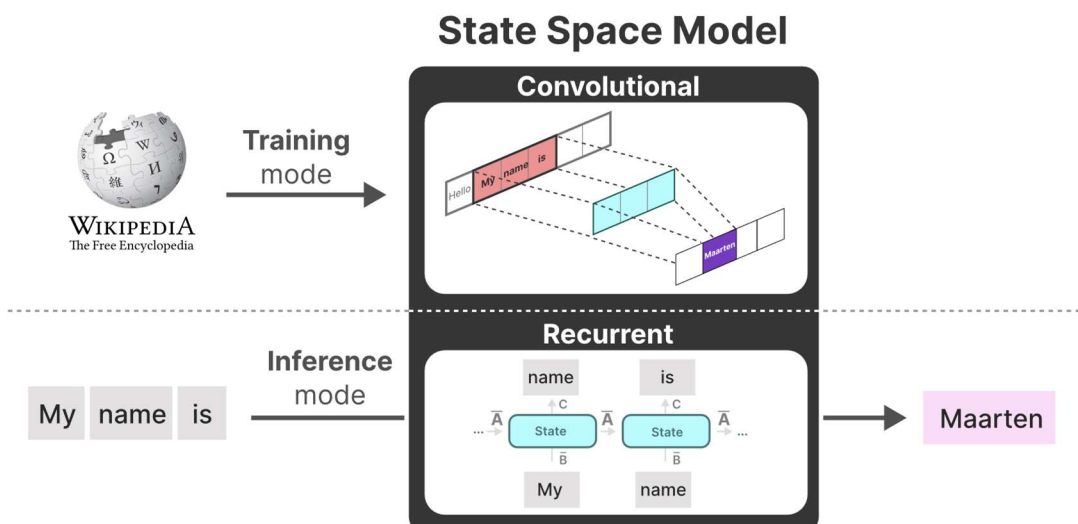
将 SSM 表示为卷积的一个主要好处是它可以像卷积神经网络(CNN)那样并行训练。然而,由于固定的核大小,它们的推理不像 RNN 那样快速和无限。

三种表示

这三种表示,连续的、递归的和卷积的,都有不同的优点和缺点集:



有趣的是,现在通过递归 SSM 实现了高效的推理,通过卷积 SSM 实现了并行化训练。有了这些表示,我们可以使用一个巧妙的技巧,即根据任务选择表示。在训练期间,我们使用可以并行化的卷积表示,而在推理期间,我们使用高效的递归表示:



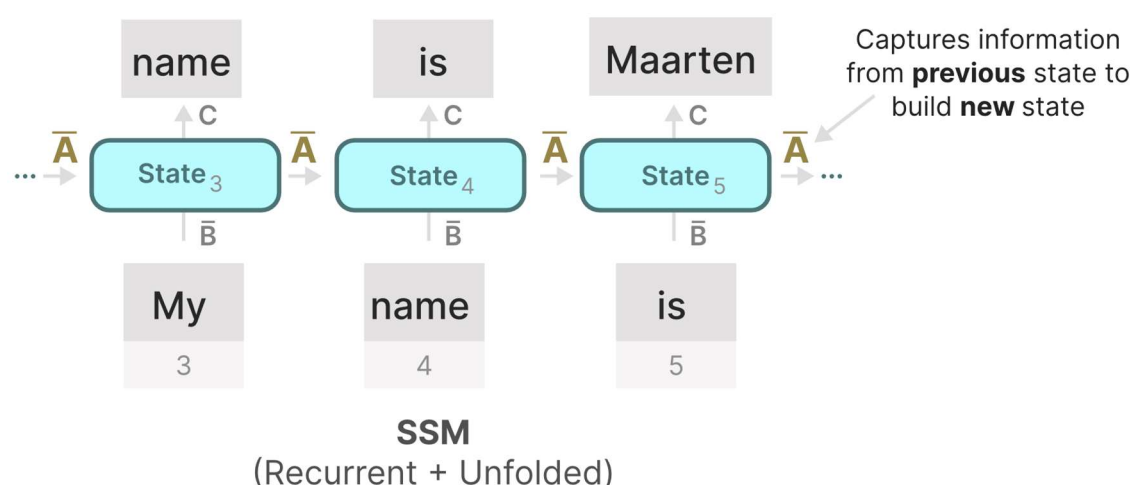
这个模型被称为线性状态空间层(LSSL)。这些表示共享一个重要的属性,即线性时间不变性(LTI)。LTI 表示 SSM 的参数 A、B 和 C 对所有时间步都是固定的。这意味着矩阵 A、B 和 C 对 SSM 生成的每个标记都是相同的。

换句话说,无论你给 SSM 什么序列,A、B 和 C 的值都保持不变。我们有一个静态表示,它不是内容感知的。

在我们探讨 Mamba 如何解决这个问题之前,让我们探讨最后一个谜题,矩阵 A。

矩阵 A 的重要性

可以说,SSM 公式中最重要的方面之一是矩阵 A。正如我们之前在递归表示中看到的,它捕获了关于先前状态的信息以构建新状态。



本质上,矩阵 A 产生隐藏状态:

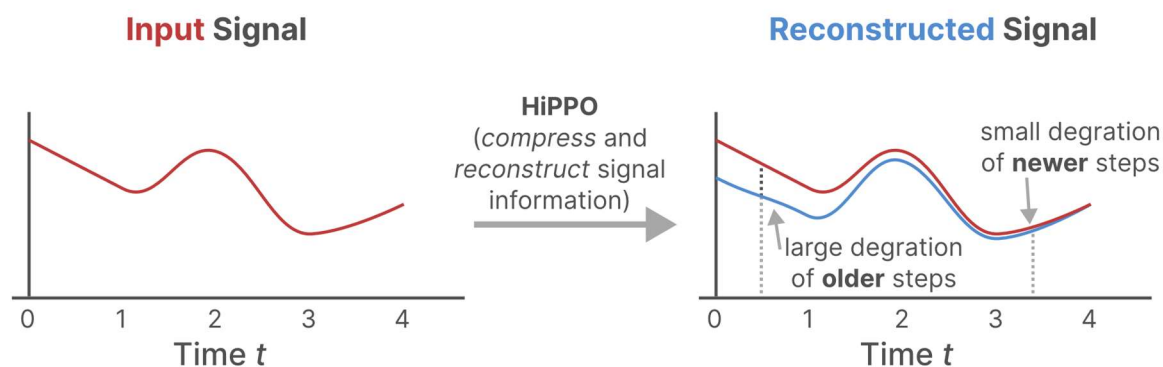
Produces hidden state

$$\mathbf{h}_k = \bar{\mathbf{A}}\mathbf{h}_{k-1} + \bar{\mathbf{B}}\mathbf{x}_k$$

$$\mathbf{y}_k = \mathbf{C}\mathbf{h}_k$$

因此,创建矩阵 A 可能是在只记住几个先前标记和捕获我们迄今为止看到的每个标记之间的区别。特别是在递归表示的背景下,因为它只回顾先前的状态。

那么我们如何以保留大内存(上下文大小)的方式创建矩阵 A 呢?我们使用 Hungry Hippo!或 HiPPO,即高阶多项式投影算子。HiPPO 试图将迄今为止看到的所有输入信号压缩成一个系数向量。



它使用矩阵 A 来构建一个能很好地捕获近期标记并衰减旧标记的状态表示。其公式可以表示如下:

$$\text{HiPPO Matrix } \mathbf{A}_{nk} \begin{cases} (2n+1)^{1/2} (2k+1)^{1/2} \leftarrow \text{everything below the diagonal} \\ n+1 \leftarrow \text{the diagonal} \\ 0 \leftarrow \text{everything above the diagonal} \end{cases}$$

假设我们有一个方阵 A , 这给我们:

HiPPO Matrix

1	0	0	0
1	2	0	0
1	3	3	0
1	3	5	4

n

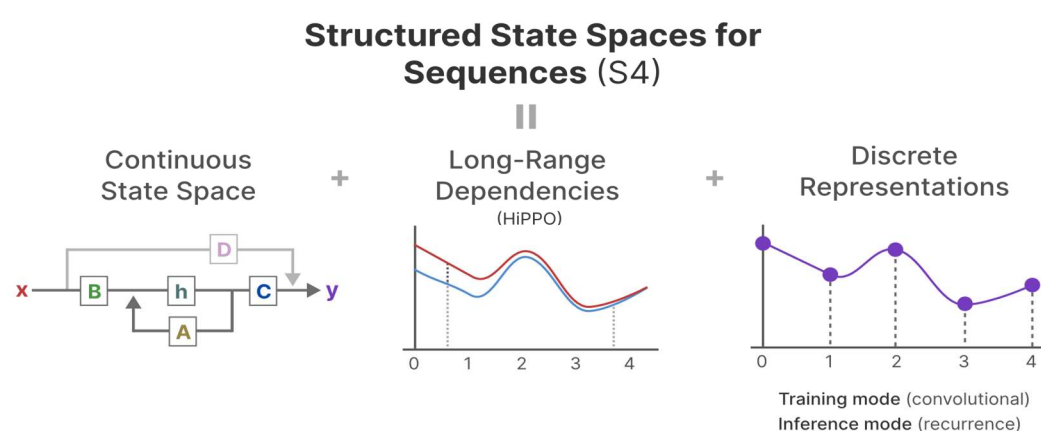
k

使用 HiPPO 构建矩阵 A 被证明比将其初始化为随机矩阵要好得多。因此,它比旧信号(初始标记)更准确地重建新信号(最近的标记)。HiPPO 矩阵背后的想法是它产生一个记住其历史的隐藏状态。从数学上讲,它通过跟踪勒让德多项式的系数来做到这一点,这允许它近似所有先前的历史。

然后,HiPPO 被应用于我们之前看到的递归和卷积表示,以处理长程依赖关系。结果是用于序列的结构化状态空间(S4),一类能有效处理长序列的 SSM。

它由三部分组成:

1. 状态空间模型
2. 用于处理长程依赖的 HiPPO
3. 用于创建递归和卷积表示的离散化



这类 SSM 有几个好处,取决于你选择的表示(递归 vs 卷积)。它还可以处理长文本序列,并通过构建在 HiPPO 矩阵之上来高效存储内存。

注意:如果你想深入了解如何计算 HiPPO 矩阵并自己构建 S4 模型的更多技术细节,我强烈建议你去看看注释的 S4。

第三部分:Mamba — 选择性 SSM

我们终于涵盖了理解 Mamba 特殊之处所需的所有基础知识。状态空间模型可以用来建模文本序列,但仍然有一些我们想要避免的缺点。

在本节中,我们将讨论 Mamba 的两个主要贡献:

1. **选择性扫描算法**,允许模型过滤(不)相关信息
2. **硬件感知算法**,通过并行扫描、核心融合和重新计算,允许高效存储(中间)结果。

它们一起创建了选择性 SSM 模型(S6),可以像自注意力一样用来创建 Mamba 块。

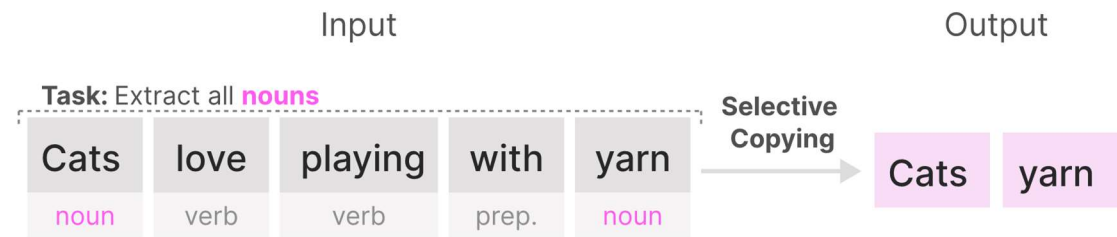
在探讨这两个主要贡献之前,让我们先探讨为什么它们是必要的。

它试图解决什么问题?

状态空间模型,甚至 S4(结构化状态空间模型),在某些对语言建模和生成至关重要的任务上表现不佳,即专注于或忽略特定输入的能力。

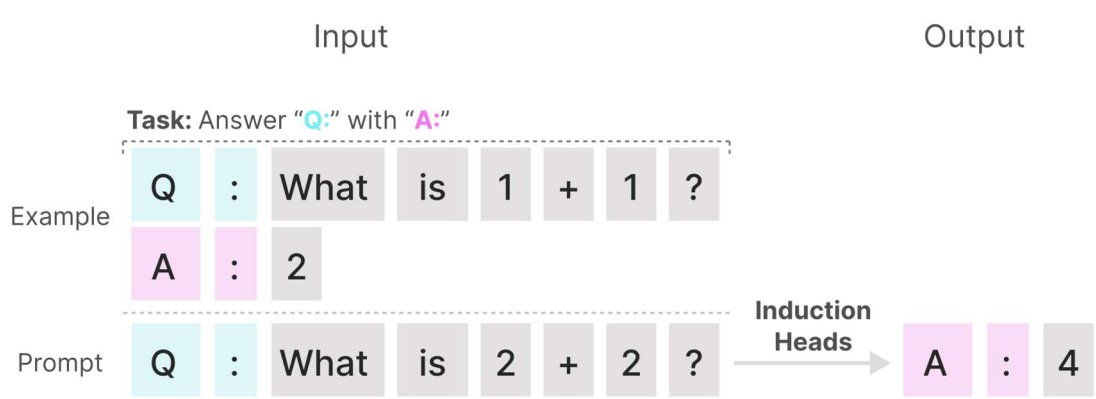
我们可以用两个合成任务来说明这一点,即**选择性复制**和**归纳头**。

在选择性复制任务中,SSM 的目标是复制输入的部分并按顺序输出它们:



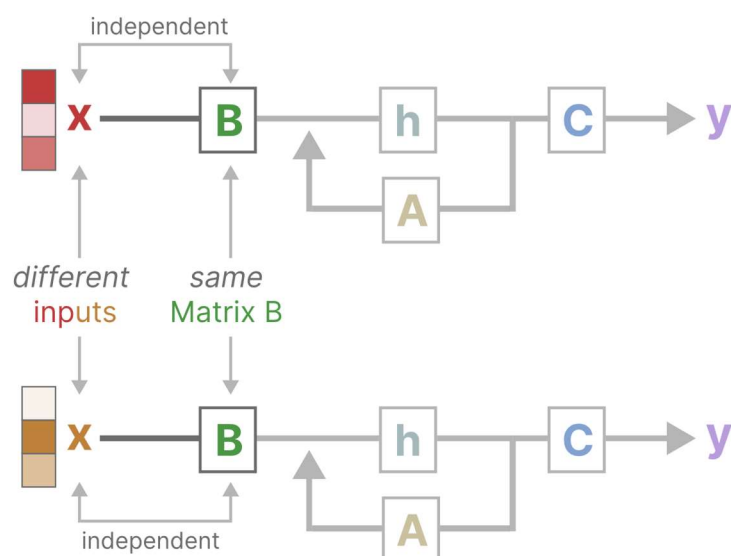
然而,(递归/卷积)SSM 在这个任务中表现不佳,因为它是线性时间不变的。如前所述,矩阵 A、B 和 C 对 SSM 生成的每个标记都是相同的。因此,SSM 无法进行内容感知推理,因为由于固定的 A、B 和 C 矩阵,它对每个标记的处理都是相同的。这是一个问题,因为我们希望 SSM 能够对输入(提示)进行推理。

SSM 表现不佳的第二个任务是归纳头,其目标是重现输入中发现的模式:



在上面的例子中,我们本质上在进行一次性提示, "教导"模型在每个"Q:"之后提供 "A:"响应。然而,由于 SSM 是时间不变的,它无法选择从其历史中回忆哪些先前的标记。让我们通过关注矩阵 B 来说明这一点。

无论输入 x 是什么,矩阵 B 保持完全相同,因此与 x 无关:



同样,无论输入如何, A 和 C 也保持固定。这展示了我们迄今为止看到的 SSM 的静态性质。

Constant regardless
of the **input**

$$h_k = \bar{A} h_{k-1} + \bar{B} x_k$$

$$y_k = C h_k$$

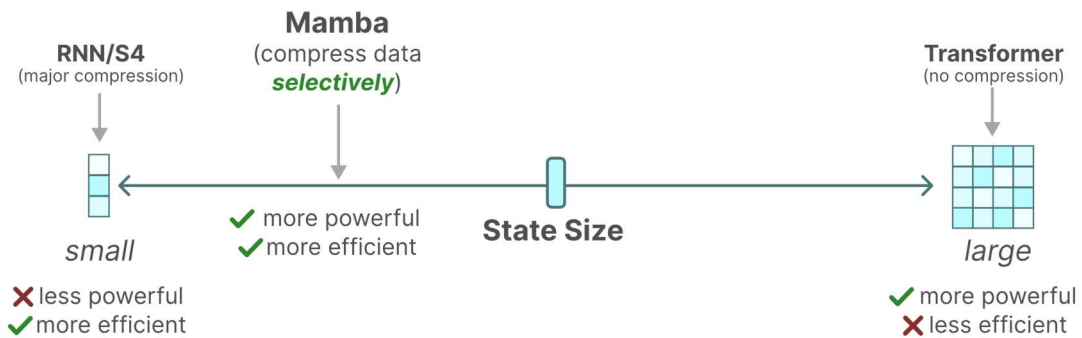
相比之下,这些任务对 Transformer 来说相对容易,因为它们根据输入序列动态地改变其注意力。它们可以选择性地"看"或"关注"序列的不同部分。

SSM 在这些任务上的糟糕表现说明了时间不变 SSM 的潜在问题,矩阵 A 、 B 和 C 的静态性质导致内容感知方面的问题。

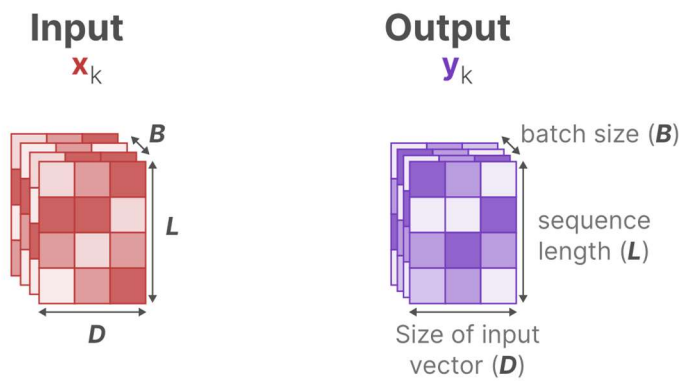
选择性保留信息

SSM 的递归表示创建了一个相当高效的小状态,因为它压缩了整个历史。然而,与不压缩历史的 Transformer 模型(通过注意力矩阵)相比,它的功能要弱得多。

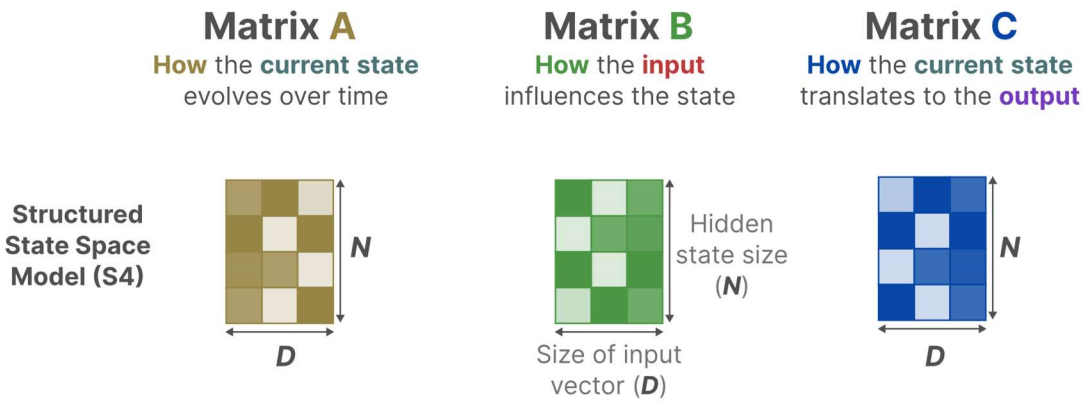
Mamba 旨在兼具两者的优点。一个小状态,但与 Transformer 的状态一样强大:



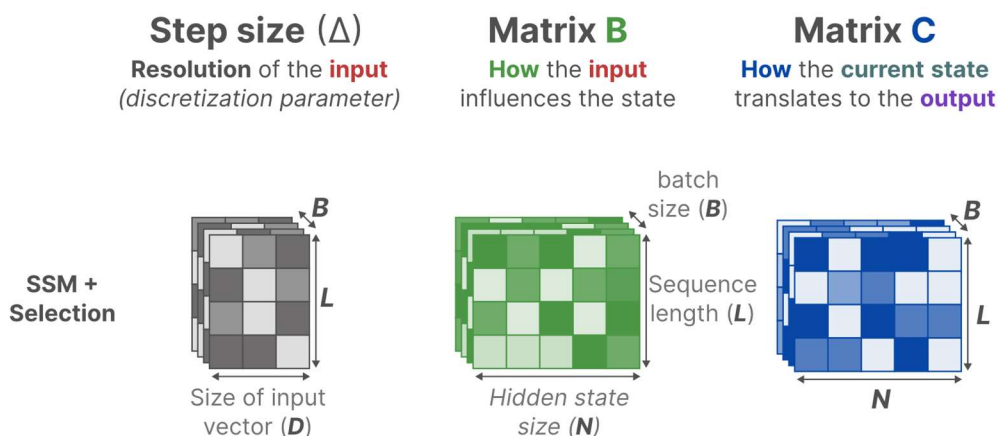
如上所示,它通过选择性地将数据压缩到状态中来实现这一点。当你有一个输入句子时,通常有一些信息,比如停用词,没有太多意义。为了选择性地压缩信息,我们需要参数依赖于输入。为此,让我们首先探讨 SSM 在训练期间输入和输出的维度:



在结构化状态空间模型(S4)中,矩阵 A、B 和 C 独立于输入,因为它们的维度 N 和 D 是静态的,不会改变。



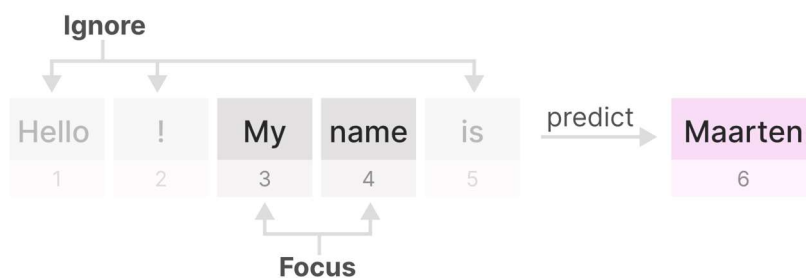
相反,Mamba 通过包含输入的序列长度和批量大小,使矩阵 B 和 C,甚至步长 Δ 依赖于输入:



这意味着对于每个输入标记,我们现在有不同的 B 和 C 矩阵,这解决了内容感知的问题!

注意:矩阵 A 保持不变,因为我们希望状态本身保持静态,但它受影响的方式(通过 B 和 C)是动态的

它们一起选择性地选择保留在隐藏状态中的内容和忽略的内容,因为它们现在依赖于输入。较小的步长 Δ 会导致忽略特定单词,而更多地使用先前的上下文,而较大的步长 Δ 则更关注输入单词而不是上下文:

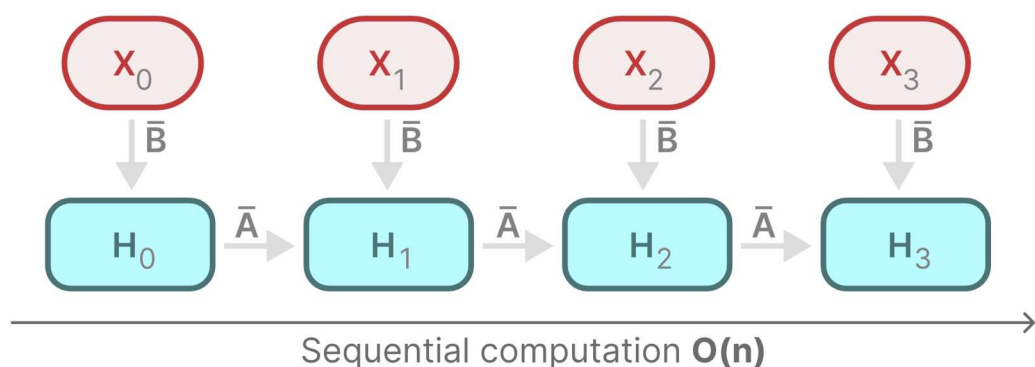


扫描操作

由于这些矩阵现在是动态的,它们不能使用卷积表示计算,因为卷积假设固定的核。

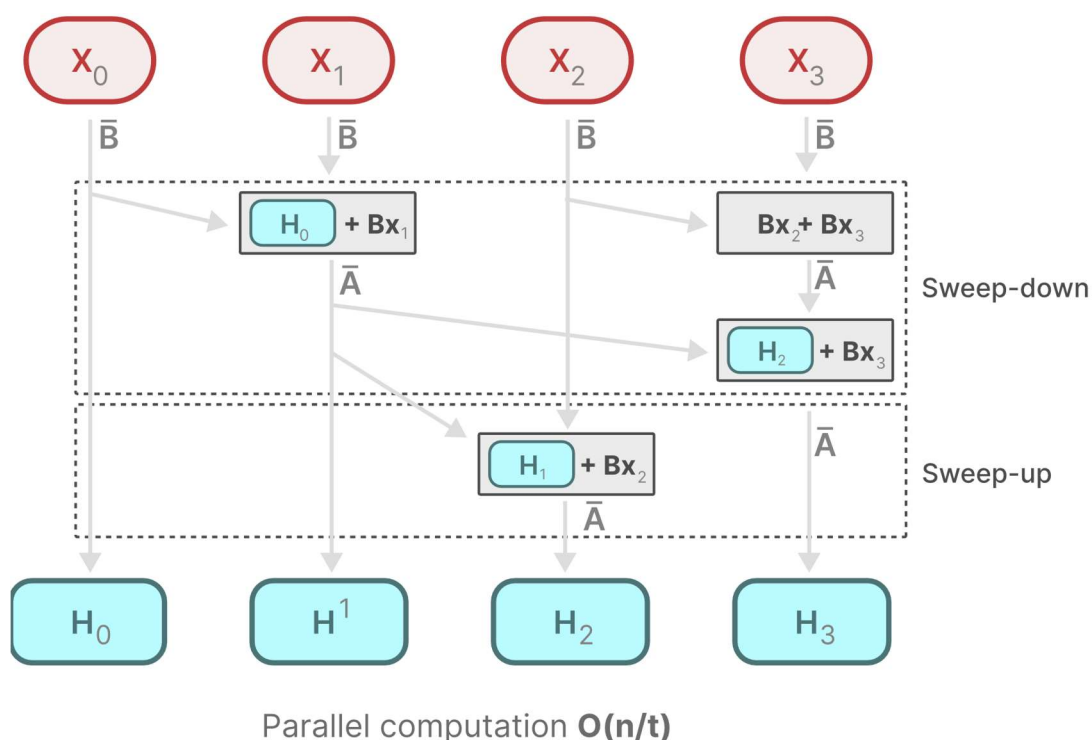
我们只能使用递归表示,并失去卷积提供的并行化。

为了实现并行化,让我们探讨如何用递归计算输出:



每个状态是先前状态(乘以 A)加上当前输入(乘以 B)的和。这被称为扫描操作,可以轻松地对 for 循环计算。相比之下,并行化似乎是不可能的,因为每个状态只有在有先前状态的情况下才能计算。

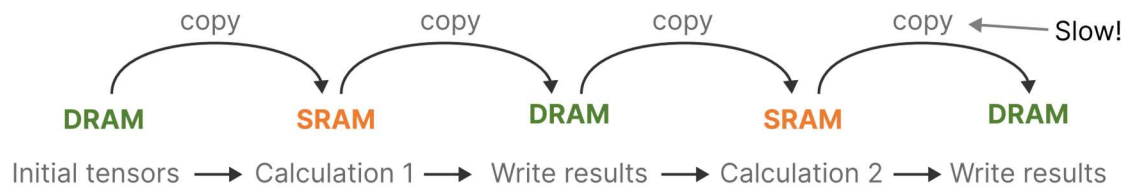
然而,Mamba 通过并行扫描算法使这成为可能。它假设我们执行操作的顺序并不重要,通过关联属性。因此,我们可以分部分计算序列,并迭代地组合它们:



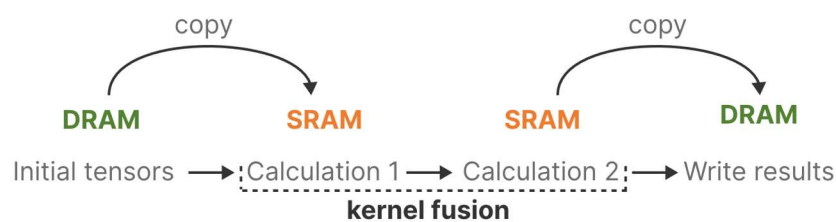
动态矩阵 B 和 C 以及并行扫描算法一起创建了**选择性扫描算法**,以表示使用递归表示的动态和快速性质。

硬件感知算法

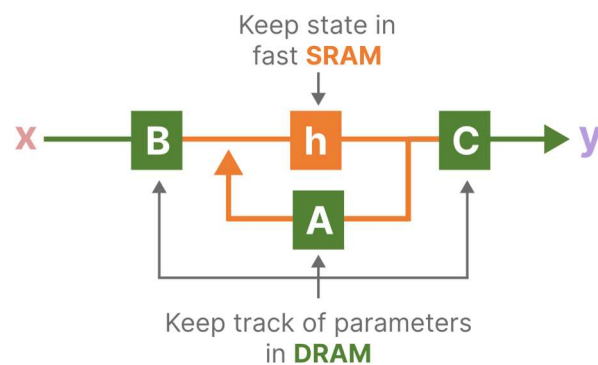
GPU 的一个缺点是它们在小而高效的 SRAM 和大而略微不那么高效的 DRAM 之间的有限传输(I/O)速度。频繁地在 SRAM 和 DRAM 之间复制信息成为瓶颈。



Mamba,像 Flash Attention 一样,试图限制我们需要从 DRAM 到 SRAM 来回的次数。它通过核心融合来实现这一点,允许模型防止写入中间结果,并持续执行计算直到完成。



可以通过可视化 Mamba 的基本架构来查看 DRAM 和 SRAM 分配的具体实例:



这里,以下内容被融合到一个核心中:

1. 带步长 Δ 的离散化步骤
2. 选择性扫描算法
3. 与 C 相乘

硬件感知算法的最后一部分是重新计算。中间状态不被保存,但在反向传播中计算梯度时是必需的。作者在反向传播过程中重新计算这些中间状态。虽然这可能看起来效率低下,但它比从相对较慢的 DRAM 读取所有这些中间状态要少得多。

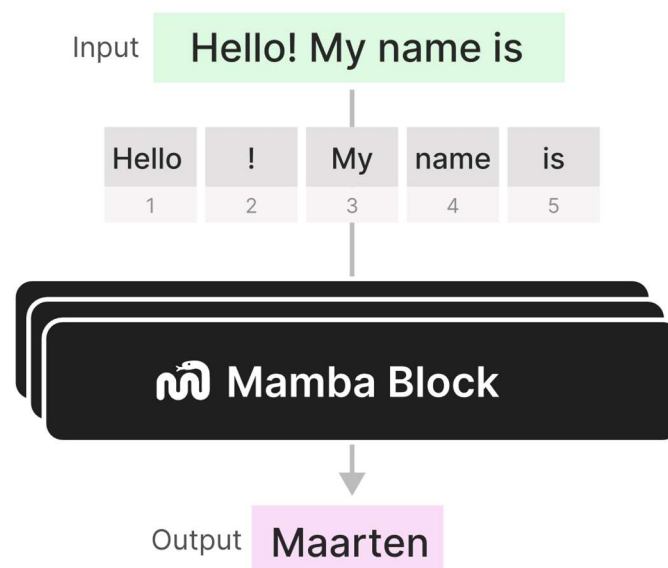
我们现在已经涵盖了其架构的所有组件,用以下来自其文章的图像描述:

Gu, Albert, Tri Dao. Mamba. (2023) preprint arXiv:2312.00752

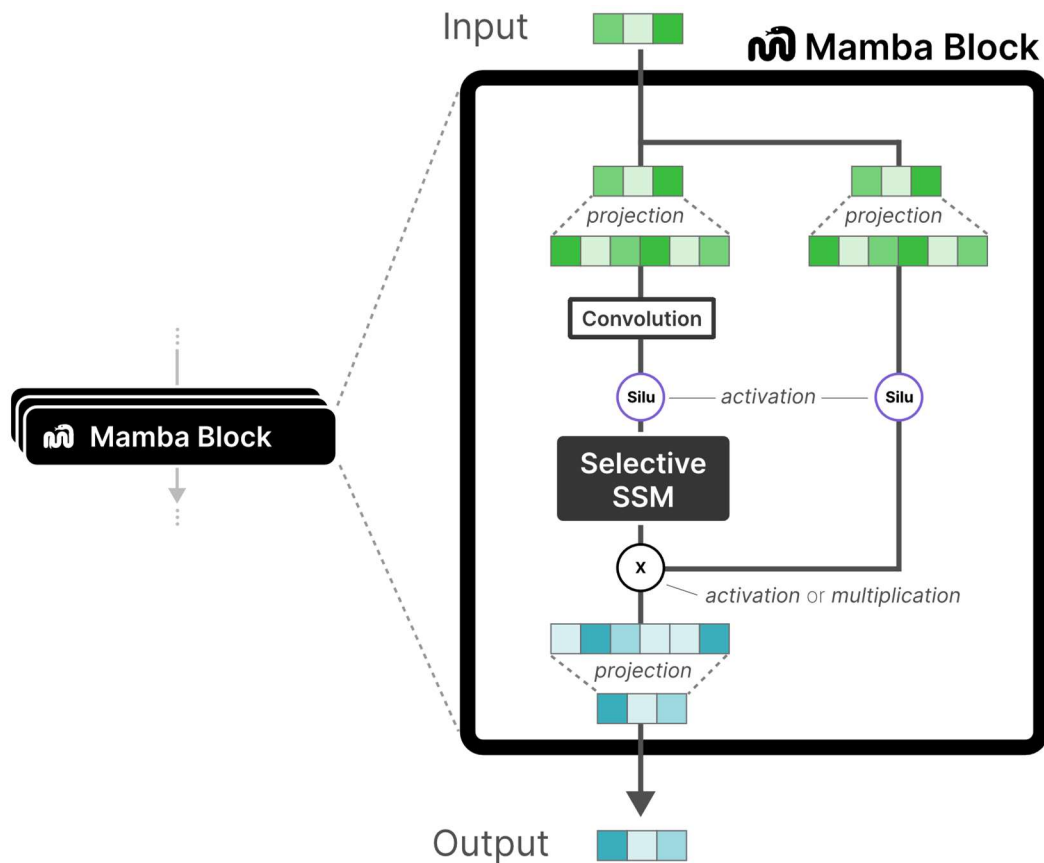
这种架构通常被称为选择性 SSM 或 S6 模型,因为它本质上是用选择性扫描算法计算的 S4 模型。

Mamba 块

我们迄今为止探讨的选择性 SSM 可以作为一个块实现,就像我们可以在解码器块中表示自注意力一样。



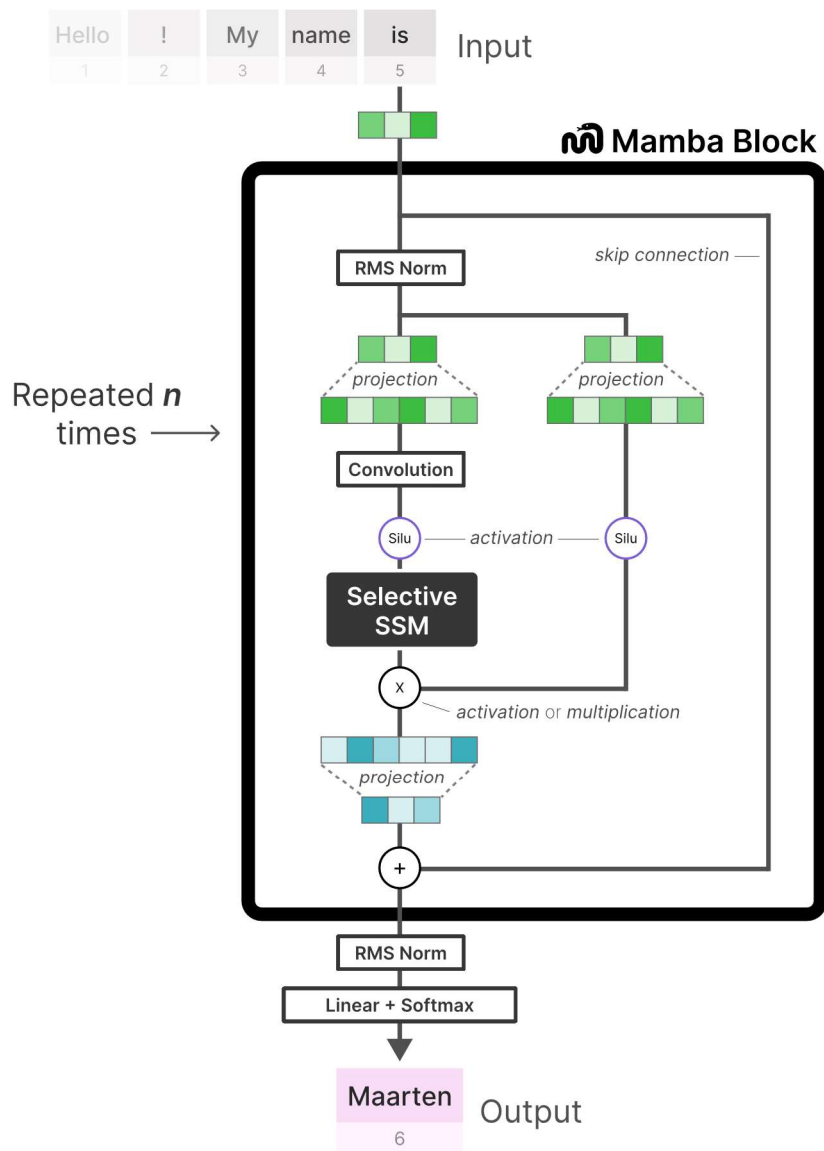
像解码器一样,我们可以堆叠多个 Mamba 块,并使用它们的输出作为下一个 Mamba 块的输入:




它从线性投影开始,以扩展输入嵌入。然后,在应用选择性 SSM 之前应用卷积,以防止独立的标记计算。选择性 SSM 具有以下属性:

1. 通过离散化创建的递归 SSM
2. 对矩阵 A 进行 HiPPO 初始化以捕获长程依赖
3. 选择性扫描算法以选择性地压缩信息
4. 硬件感知算法以加速计算

当查看代码实现时,我们可以更详细地扩展这个架构,并探索端到端示例的样子:



注意一些变化,比如包含归一化层和 softmax 以选择输出标记。当我们把所有东西放在一起时,我们得到了快速的推理和训练,甚至是无限的上下文!

	Training	Inference
Transformers	Fast! (parallelizable)	Slow... (scales quadratically with sequence length)
RNNs	Slow... (not parallelizable)	Fast! (scales linearly with sequence length)
 Mamba	Fast! (parallelizable)	Fast! (scales linearly with sequence length + unbounded context)

使用这种架构,作者发现它与相同大小的 Transformer 模型相匹配,有时甚至超过其性能!

Additional Resources[Permalink](#)

Hopefully, this was an accessible introduction to Mamba and State Space Models.

If you want to go deeper, I would suggest the following resources:

- [The Annotated S4](#) is a JAX implementation and guide through the S4 model and is highly advised!
- A great [YouTube video](#) introducing Mamba by building it up through foundational papers.
- [The Mamba repository](#) with [checkpoints on Hugging Face](#).
- An amazing series of blog posts ([1](#), [2](#), [3](#)) that introduces the S4 model.
- The [Mamba No5 \(A Little Bit Of...\)](#) blog post is a great next step to dive into more technical details about Mamba but still from an amazingly intuitive perspective.
- And of course, [the Mamba paper](#)! It was even used for DNA modeling and speech generation.

Thank you for reading! [Permalink](#)