目录

1 Transformer 和线性 RNN

Notations: $\mathbf{X}$ 表示矩阵，$\vec{x}$ 表示列向量。

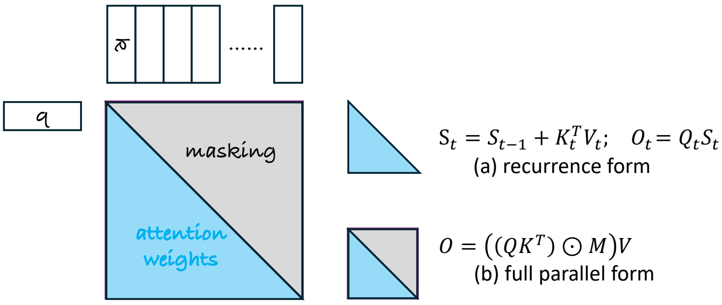


图 1: Transformer 的两种计算模式：recurrent form 和 full parallel form

训练时我们可以一次性看到全序列，如果 token 之间不存在数据流依赖，就可以让 query token [1] 全并行计算。

如图1所示，上三角部分是冗余计算，通过 masking 机制消除。Attention 的全并行方式是直接将 for 循环在空间上全展开。空间全展开情况下上图上三角部分是冗余计算，需要后续靠点乘一个 masking 矩阵消除这部分多余计算。

1.1 Linear Attention [1]

$$O = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{K}^T)\mathbf{V} \tag{1}$$

上述公式是多头注意力机制中一个头如何计算的线性代数公式。其中 $\mathbf{Q} \in \mathbb{R}^{L \times d}$，$\mathbf{K}^T \in \mathbb{R}^{d \times L}$，$\mathbf{V} \in \mathbb{R}^{L \times d}$。

目前 Transformer 模型广泛被用来处理超长序列。假设 E 是隐藏层的维度(一般 $E = 512, 1024, 2048$，在这样的量级）。如果应用了多头注意力机制，会对隐藏层分头 $d = \frac{E}{\text{num\_head}}$。于是 $L \gg d$，$\mathbf{P} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{L \times L}$ 这一步会得到一个非常大的 L^2 大小的中间结果。

仔细分析会发现**影响 attention 进一步扩展的是 softmax**。

矩阵乘满足结合律，如果没有 softmax，我们可以先计算 $\mathbf{S} = \mathbf{K}^T\mathbf{V}$：$[d \times L][L \times d] = [d \times d]$，得到中间结果大小为 $d \times d$，再计算 $\mathbf{O} = \mathbf{Q}\mathbf{S}$。由于 $d \ll L$，这样的计算过程也会近似地与序列长度呈线性复杂度。

观察 attention 的 sequential 计算公式 (换一种语言来说这里的“sequential 计算公式”，也就是把"matrixfied" 的线性代数公式展开写成 $\sum$ 求和)：

$$\text{Attention}(\vec{q}_t, \mathbf{K}, \mathbf{V}) = \frac{\sum_{i=0}^{t} \exp\left(\vec{q}_t \vec{k}_i^T\right) \vec{v}_i}{\sum_{i=0}^{t} \exp\left(\vec{q}_t \vec{k}_i^T\right)}$$

[1] 这是我们要研究的第一种计算模式。

我们可以把 attention 理解为以 $\exp\left(\vec{q}_t\vec{k}_i^T\right)$ 为权值，对 $\vec{v}_i$ 进行加权再求和。于是可以提出一个泛化版的 attention：

$$\text{Attention}\left(\vec{q}_i, \mathbf{K}, \mathbf{V}\right) = \frac{\sum_{t=0}^{L} \text{sim}\left(\vec{q}_i, \vec{k}_t\right)\vec{v}_j}{\sum_{t=0}^{L} \text{sim}\left(\vec{q}_i, \vec{k}_t\right)}$$

为了保持 attention 的输出结果是一个分布这一关键特性，**我们要求 $\mathbf{sim}(q_t, k_i) \geq 0$**[2]。基于以上分析，可扩展 attention 转换为如何设计 $\text{sim}\left(\vec{q}_i, \vec{k}_j\right)$。

寻找一个总是大于 0 的相似度函数可以被转换为许多不同的解决思路，[6] 用了一种简单的思路，给 $\vec{q}_t$ 和 $\vec{k}_i$ 加上限制了值域的激活函数，然后使用内积作为相似性度量。这种方法被解释为“kernel method"。于是我们有如下公式：

$$\vec{o}_t = \frac{\sum_{i=0}^{t} \phi\left(\vec{q}_t\right)\phi\left(\vec{k}_i\right)^T \vec{v}_i}{\sum_{i=0}^{t} \phi\left(\vec{q}_t\right)\phi\left(\vec{k}_i\right)^T} = \frac{\phi\left(\vec{q}_t\right)\sum_{i=0}^{t} \phi\left(\vec{k}_i\right)^T \vec{v}_i}{\phi\left(\vec{q}_t\right)\sum_{i=0}^{t} \phi\left(\vec{k}_i\right)^T}$$

$\mathbf{S}_t \triangleq \sum_{i=0}^{t} \phi\left(\vec{k}_i\right)^T \vec{v}_i$，$\vec{z}_t \triangleq \sum_{i=0}^{t} \phi\left(\vec{k}_i\right)^T$，于是有：

$$\begin{aligned}
\mathbf{S}_t &= \mathbf{S}_{t-1} + \phi\left(\vec{k}_t\right)^T \vec{v}_t \\
\vec{z}_t &= \vec{z}_{t-1} + \phi\left(\vec{k}_t\right)^T \\
\mathbf{O}_t &= \frac{\phi\left(\vec{q}_t\right)\mathbf{S}_t}{\phi\left(\vec{q}_t\right)\vec{z}_t}
\end{aligned} \tag{2}$$

式(2)就是一个泛化版 attention 框架，不同的工作对 ϕ 进行了探索。（1）认为 ϕ 取 indentity function in practice 也够用。(2) 有些工作 claim 归一化因子 $\vec{z}_t$ 会引起不稳定性，建议可以扔掉 $\vec{z}_t$，在输出增加归一化计算，于是会得到一个如下形式的未归一化的 linear attention Transformer：

$$\begin{aligned}
\mathbf{S}_t &= \mathbf{S}_{t-1} + \vec{k}_t^T \vec{v}_t \\
\vec{o}_t &= \vec{q}_t \mathbf{S}_t
\end{aligned} \tag{3}$$

linear-attention当作[3] RNN 式的递归式来计算时（图1(a)），模型的时间和空间复杂度是序列长度的线性关系，能够节约 FLOPS，但是会引入串行性。

如果我们把 attention 按照下式当作一个矩阵乘来计算（图1(b)），模型的时间和空间复杂度是序列长度的平方关系，优点是可以全并行计算。

$$\mathbf{O} = \left(\left(\mathbf{Q}\mathbf{K}^T\right) \odot \mathbf{M}\right)\mathbf{V}$$

[2] 每个分量 ≥ 0 是分布的一个最低要求，softmax 会得到一个严格的概率分布，求和为 0

[3] 这是我们要研究的第二种计算模式。

上式 $\mathbf{M}$ 会破坏矩阵乘的结合性，导致如果我们想通过并行方式（时间维度的计算在空间维度全展开），依然必须先算 $\mathbf{QK}^T$，再与 $\mathbf{V}$ 相乘，还是会引起需要存储 L^2 大小的中间结果。

为了让这类模型高效训练，一个折中方案是[4] Chunk-wise Block-parallel attention。

$$
\begin{aligned}
\mathbf{O}_{i+1} &= \mathbf{Q}_{i+1}\mathbf{S}_i + \left(\left(\mathbf{Q}_{i+1}\mathbf{K}_{i+1}^T\right) \odot \mathbf{M}\right)\mathbf{V}_{i+1} \in \mathbb{R}^{C\times d} \\
\mathbf{S}_0 &= \mathbf{0}
\end{aligned}
$$

1.2 Linear Attention with Decay

(3)显示地表明：**带线性 attention 的 Transformer，就是一个隐状态是矩阵的 RNN**。有许多研究指出，为 RNN 的 state 引入 decay（衰减）机制，类似于遗忘门的能力，能够有效的改善学习效果。于是引入衰减因子γ，(3)式改写成如下形式：

$$
\mathbf{S}_t = \gamma\mathbf{S}_{t-1} + \mathbf{K}_t^T\mathbf{V}_t \tag{4}
$$

对应的并行形式：

$$
\mathbf{O} = \left(\mathbf{QK}^T \odot \mathbf{D}\right)\mathbf{V}, \qquad \mathbf{D} = \begin{cases} \gamma^{n-m}, & n \geq m \\ 0, & n < m \end{cases}
$$

1.3 典型代表

1.3.1 RetNet [10]

RNN 模型相关研究表明：遗忘门对改善学习效果有着重要作用。

在 DNN 模型学习效果方面，RetNet 的一个设计要素是：在 recurrent 计算对状态的更新过程中，引入一个显示的衰减（decay term）能够极大的改善学习效果，一些情况下甚至可以超过标准的 softmax Transformer。

> adding a global decay term to the additive RNN update rule greatly improves performance, sometimes outperforming standard Transformers with softmax attention when trained at scale. [11]

1.3.2 GAU [5]

1.3.3 GLA [11]

2 SSM（state-space model）

SSM 是一个非常广泛的概念，可以认为带有隐状态的循序计算都是 SSM 模型。他的思想和和 RNN，CNN，Attention 这样的神经网络完全不同，背后的想法是让把序列看做是对一个动态系统的离散采样，SSM 模型通过微分方程组来建模动态系统状态随时间的变化。

[4] 这是我们要研究的第三种计算模式。

2.1 一个朴素的直觉解释

我们首先通过一个全标量计算模型，建立起对 state-space model 的直觉理解。

用微分方程组建模动态系统，一个极简例子

这个video（对应的slides）里 speaker 举了一个具体的例子来解释用微分方程组建模一个动态系统。假设我们有一群兔子，每只小兔子在 t 时刻会生 λ 只小兔子，λ 是一个固定的常数。令 $b(t)$ 表示 t 时刻兔子种群一共有多少只兔子，$\frac{db}{dt}$ 表示 t 时刻兔子种群数目变化的速率，于是有：

$$\frac{db}{dt} = \lambda b(t)$$

已知 $t = 0$ 时刻种群有 5 只兔子，即：$b(t) = 5$。我们希望求解出 $b(t)$ 的方程使得对任意 t，上式都成立，来告诉我们任意时刻，例如 $t = 100$，种群有多少只兔子。

连续时间系统线性 SSM 模型（continuous-time linear/latent state-space model）将输入信号 $x(t)$ 通过一个隐状态 $h(t)$ 映射到输出信号 $y(t)$。

$$h'(t) = \mathbf{A}h(t) + \mathbf{B}x(t) \tag{5}$$

$$y(t) = \mathbf{C}h(t) + \mathbf{D}x(t) \tag{6}$$

(5)和(6)已经规定了微分方程组的具体形式，参数化为 $\mathbf{A}$，$\mathbf{B}$，$\mathbf{C}$ 和 $\mathbf{D}$。这样一组微分方程可以用来建模一个动态系统其状态随时间的改变。求解目标是求解出以上微分方程的具体形式，使得给定系统在 0 时刻的初始状态，能够计算得到这个动态系统在任意时刻的状态。

为了求解上面这个微分方程组，我们需要找到 $h(t)$ 的具体形式令(5)等号左右两边相当。通常情况下找到 $h(t)$ 的解析解是困难的。因此，我们降低要求，考虑找到 $h(0)$，$h(1)$，$h(2)$，etc. 这样一个离散化的序列，用来逼近我们要研究的动态系其状态如何随时间改变。因此，我们将寻找 $h(t)$ 转化为对时间进行离散化采样，k 是采样编号，寻找一个离散化序列 $h(t_k) = h(k\triangle)$ 使以上微分方程系统近似地成立。其中，$\triangle$ 是 step size。

根据导数的定义，我们有：$h'(t) = \lim_{\triangle \to 0} \frac{h(t+\triangle)-h(t)}{\triangle}$，当 step size$\triangle$ 足够小的时候，我们可以省略极限符号：

$$h(t+\triangle) \approxeq \triangle h'(t) + h(t)$$

将(5)带入，可得：

$$\begin{aligned}
h(t+\triangle) &\approxeq \triangle\left(\mathbf{A}h(t) + \mathbf{B}x(t)\right) + h(t) \\
&= \triangle\mathbf{A}h(t) + \triangle\mathbf{B}x(t) + h(t) \\
&= \left(\mathbf{I} + \triangle A\right)h(t) + \triangle\mathbf{B}x(t) \\
&\triangleq \bar{\mathbf{A}}h(t) + \bar{\mathbf{B}}x(t) \qquad (7)
\end{aligned}$$

式(7)是 SSM 的离散化形式。

2.2 SSM

机器学习任务中隐状态均为高维，从上一节的标量形式进一步推广到高维可以参看 s4 [4] 的论文，这里我们仅关注需要知道的最基本事实。一个 SSM 模型的求解需要经过离散化（Zero-Order-

Hold（ZOH）或者 Bilinear 方法），离散化形式的 SSM 由以下两个公式表示，其中 $\vec{x}$ 是状态，$\vec{u}$ 是输入信号，$\vec{y}$ 是输出信号，

$$\vec{h}_k = \bar{\mathbf{A}}\vec{x}_{k-1} + \bar{\mathbf{B}}\vec{u}_k \tag{8}$$

$$\vec{y}_k = \bar{\mathbf{C}}\vec{x}_k + \bar{\mathbf{D}}\vec{u}_k \tag{9}$$

离散化方法会进一步给出 $\bar{\mathbf{A}}$，$\bar{\mathbf{B}}$ 和 $\bar{\mathbf{C}}$ 矩阵的进一步形式。

为了让离散化操作能快速地完成计算，会对矩阵 $\mathbf{A}$ 的形式有所约束，例如为了让矩阵的幂运算快速实现，要求 $\mathbf{A}$ 是一个对角矩阵。

在预测时候和 RNN 等价，SSM 参数化形式为一组用微分方程组描述的，带隐状态的连续时间系统。通过恰当的离散化方法，能够被转换成 CNN 或 RNN 计算。

预测时 SSM 和 RNN 计算等价，但是 SSM 这一类模型的设计来源于不同的理论框架，在训练时有可能带来不同的特性。

2.3 SSM 和 RNN 的不同之处

DeepMind 的 LRU [7] 也讨论了 SSM 和 RNN 的不同。

1. SSM 模型离散化之后根据定义在时间维度上是线性的。也就是说，SSM by definition 就是一个 linear RNN，在训练的时候适用于以 parallel scan 模型并行地进行计算；
2. 尽管式(8)是一个 RNN，但是其中 $\mathbf{A}$ 和 $\mathbf{B}$ 有更强的参数化要求。例如，ZOH 离散化方法会给出：$\bar{\mathbf{A}} = \exp(\triangle\mathbf{A})$，$\bar{\mathbf{B}} = \left(\bar{\mathbf{A}} - \mathbf{I}\right)\mathbf{A}^{-1}\mathbf{B}$，$\bar{\mathbf{C}} = \mathbf{C}$，$\bar{\mathbf{D}} = \mathbf{D}$。
3. SSM 的初始化非常特殊。recurrent 形式的全展开涉及到矩阵的幂运算。为了高效计算矩阵的幂，会要求 $\mathbf{A}$ 是对角矩阵。然而无法总是在实数域将 $\mathbf{A}$ 对角化，但是几乎所有矩阵都能在复数域上对角化，于是 SSM 将运算放在了复数域，[5]SSM 需要特殊设计的初始化计算。

2.4 典型代表

2.4.1 S4（Structured State Space Sequence models）[4]

S4 的第一个 S 代表了"structured"。具体是指：为了 SSM 能够高效地并行训练，会对矩阵 A 的结构带来一定的约束，目前使用最多的一种约束是要求 A 是一个对角矩阵。

2.4.2 S5 [9]

[5]最新的研究通过新的技术放松对这一点的要求

2.4.3 Mamba（S6）[3]

受 Transformer 整体架构的影响，目前 sequence processing 模型都是通过反复堆叠一个同构的 sequence processing block 构成。这样一个 sequence processing block 一般由归一化层, token mixer, residual connection 三个设计要素构成。mamba 中的 norm 都用 RMSnorm。

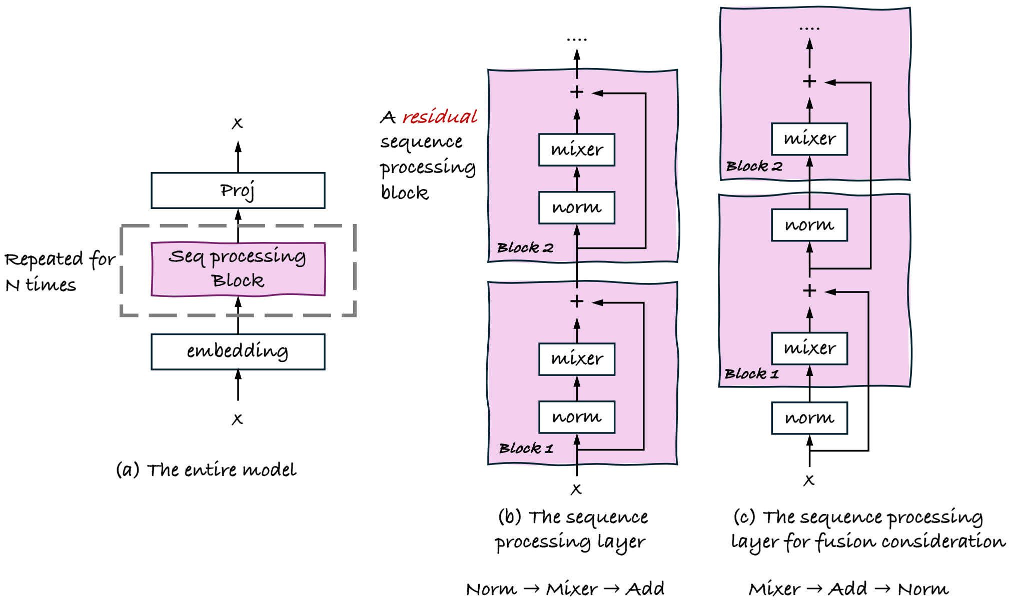
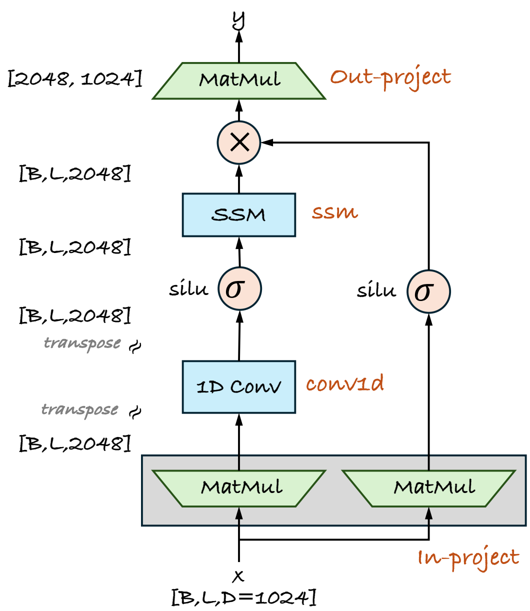


图 2: mamba 模型的整体结构

图2(b) 的 Norm → Mixer → Add 串 block 的方式更符合设计直觉。图2(a) 的 Mixer→ Add → Norm 串 block 的方式更容易在现有 PyTorch 接口下将 Add 和 Norm 进行 fuse。



Parameter Name	Shape	Module Name
In_proj	[4096,1024]	In-projct
Conv1d-kernel	[2048,1,4]	conv1d
Conv1d-bias	[2048]	
A	[2048,16]	SSM
D	[2048]	
x_proj	[96,2048]	
delta_proj	[2048,64]	
delta_proj.bias	[2048]	
out-proj	[1024,2048]	Out-project
norm	[1024]	

图 3: mamba block 作为图2中 mixer 的细节及可学习参数

Mamba block 是图2中 mixer。Mamba-370m 模型里面 mamba block 堆叠 48 次，输入序列 X 的形状 $[B, L, D]$，$D = 1024$。

图3中 SSM 部分计算公式如下，其中 $u(t)$ 是整个 sequence batch，形状为 $[B, L, 2048]$，$x(t)$ 是 SSM 内部的状态：

$$y(t) = \mathrm{SSM}(u(t)) \tag{10}$$

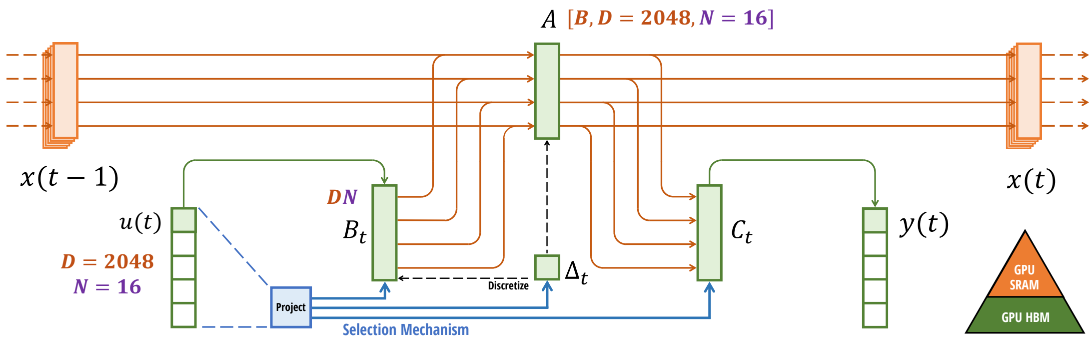


图 4: 图3中 SSM 模块的细节

SSM 内部带有隐状态 $x(t)$，式(10)通过隐状态 $x(t)$ 把输入 $u(t)$ 映射到 $y(t)$。一个离散化之后的 SSM 模型其可学习为：$\triangle$，$\mathbf{A}$，$\mathbf{B}$，$\mathbf{C}$，$\mathbf{D}$。mamba 中，$\mathbf{A}$ 和 $\mathbf{D}$ 不依赖于输入；$\triangle$，$\mathbf{B}$ 和 $\mathbf{C}$ 依赖于输入。

Listing 1中红色高亮的变量直接对应图3中右表中的可学习参数。

```
1  function SSM(u_[B,L,2048]) → y(t)_[B,L,2048]
2    v_0 = u(t) @ x_proj^T   // [B, L, 2048]@[2048, 96] → [B, L, 96]
3    v_1, B, C = split(v_0)   // [B, L, 64], [B, L, 16], [B, L, 16]
4    △ = softplus(v_1@delta_proj^T + delta_proj.bias)  // [B, L, 64]@[2048, 64] + [2048] → [B, L, 2048]
5    Ā = exp(△A)   // [B, L, 2048]@[2048, 16] → [B, L, 2048, 16]
6    V̄ = △Bu(t)   // [B, L, 2048]@[B, L, 16]@[B, L, 2048] → [B, L, 2048, 16]
7
8    // 9~12行是mamba代码中的selective_scan kernel
9    x(t) = zeros([B, 2048, 16])   // scan的状态
10   for i in [0, L):   // 把这个for循环实现成parallel scan
11     x(t) = Ā[:, i] * x(t) + V̄[:, i]   // [B, 2048, 16]
12     y(t)[:, i] = x(t)@C[:, i]   // bmm, [B, 2048, 16]@[B, 16] → [B, 2048]
13   y(t) = y(t) + D * u(t)   // [B, L, 2048]
```

Listing 1: ssm in mamba

Listing 1 中 5 至 13 行对应了离散化 SSM 的两组公式：

$$\bar{\mathbf{A}} = \exp(\triangle\mathbf{A}) \qquad [B, L, 2048, 16] \tag{11}$$

$$\bar{\mathbf{V}} = \bar{\mathbf{B}}u(t) = \triangle\mathbf{B}u(t) \qquad [B, L, 2048, 16] \tag{12}$$

$$x(t+1) = \bar{\mathbf{A}}x(t) + \bar{\mathbf{V}} \qquad [B, 2048, 16] \tag{13}$$

$$y(t) = \mathbf{C}x(t) + \mathbf{D}u(t) \qquad [B, L, 2048] \tag{14}$$

(11) 是 ZOH 离散化；(12) 中，$\bar{\mathbf{B}} = \triangle\mathbf{B}$ 是简化的欧拉离散化（关于 A 和 B 离散化方法为什么这样选择，参考这个项目给出的一些说明。）我们来看式(13)和(14)的实现。公式(13)的实现就是一直存在诸多迷思的 parallel scan。

在 Mamba 之前的 SSM 都是对一个线性时不变系统进行模拟，由于在时间维度上没有依赖，训练时可以当做卷积进行计算。

图5中 $u(t)$ 是输入信号，是一个标量，$\vec{x}$ 是系统的状态，是一个向量。$\mathbf{A}$，$\mathbf{B}$，$\mathbf{C}$ 是矩阵，形状如图所示，是作用于 scalar 输入信号的 $u(t)$ 的 SSM 模型的参数。$\mathbf{A}$ 矩阵是对角矩阵。因此，一个 SSM 模型的参数可以用 N 个数表示。

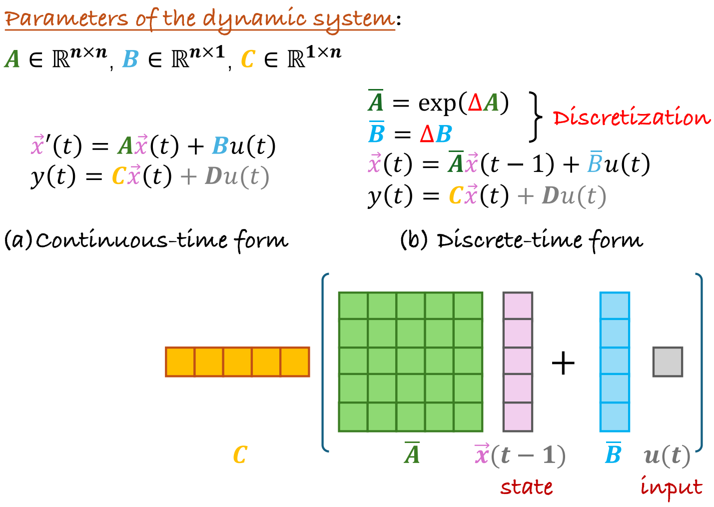


图 5: 作用于一个 scalar input $u(t)$ 的 SSM model，$\vec{x}$ 是状态

机器学习任务中输入都是向量形式，假设为 D。将图5这样一个模型应用到向量输入的方法也非常直接：引入 D 个独立的 SSM 模型，每个独立地作用于输入信号的一个维度。假设输入序列的 batch 大小为：B，序列长度 L，channel 大小为 D （hidden size 大小）。于是，一个处理向量输入的 SSM 模型的有 DN 个参数。处理完整个序列的时间和空间复杂度是：$O(BLDN)$，mamba 的贡献之一是通过模型设计，解决这个时间和空间的复杂度。

3 并行 RNN

3.1 典型代表

3.1.1 RWKV [8]

3.1.2 LRU（Linear Recurrent Unit）[7]

4 General non-linear recurrence 的并行计算问题

4.1 SOAC: scan

Second-order Array Combinator (SOAC)：***scan*** 是所有的循环神经网络（recurrent neural networks）背后的核心并行 pattern。接口和语义如下：

$$
\begin{aligned}
\textbf{scan} &:: (\alpha, \beta \to \alpha) \to \alpha \to [\beta]_n^d \to [\alpha]_n^d \\
\textbf{scan}\ \oplus\ xs &= [x_0,\ x_0 \oplus x_1, \cdots,\ x_0 \oplus x_1 \cdots \oplus x_n] \\
\textbf{scanl}\ \oplus\ I\ xs &= [I \oplus x_0,\ ((I \oplus x_0) \oplus x_1),\ \cdots,\ (((I \oplus x_0) \oplus x_1) \cdots \oplus x_{n-1})] \\
\textbf{scanr}\ \oplus\ I\ xs &= [(x_0 \oplus (x_{n-2} \oplus (x_{n-1} \oplus I))),\ \cdots,\ (x_{n-2} \oplus (x_{n-1} \oplus I)),\ x_{n-1} \oplus I]
\end{aligned}
$$

4.2 Stacked RNNs

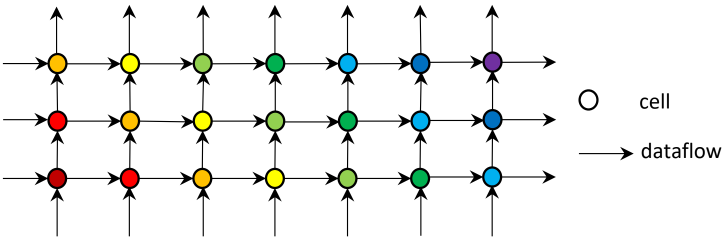


图 6: Stacked RNN 网络的数据流图

一个 optimizing compiler 最关心的问题是：检测一个算法天生的最大并行性和最大数据复用机会。如果能检测到就有可能通过调度策略的设计，在一个并行后端上达到高效执行。我们无论用什么样的语法形式去写 RNN 网络，对 optimizing compiler 检测和利用并行性最重要的是抽取出图6这样的一个数据流图结构。

```
xss: [][]float32[1, 512] = ...   // A batch of input sequences
ws: [3]float32[512, 512] = ...   // UDF的可学习参数

ysss = xss.map(xs => {  // ysss: [[[float32[1, 512]]]]
  yss = ws.scan(ss, w => {  // scan over depth
    ys = ss.scan(s, x => {  // scan over sequence length
      y = x @ w + s  // the user-defined cell function.
    }, initializer=zeros),
  }, initializer=xs),
})
```

Listing 2: 用 parallel pattern compose stacked RNN 网络

```
xss: [][]float32[1, 512] = ...   // A batch of input sequences
ws: [3]float32[512, 512]] = ...   // UDF的可学习参数

ysss = xss.map(xs => {  // ysss: [][][]float32[1, 512]
  yss = xs.scan(ss, x => {  // scan over sequence length
    ys = zip(ss, ws).scan(s0, s, w => { // scan over depth
      y = s0 @ w + s  // UDF是一个非常小的纯线性代数公式
    }, initializer=x),
  }, initializer=repeat(zeros, ws.length)),
})
```

Listing 3: Listing 2 的另一种语法等价形

这里有一个非常漂亮的特性，对并行性检测和利用至关重要，***用 map，reduce，scan 这样的 SOAC 写出来的嵌套循环程序，一定是一个可任意换序的循环嵌套***。

```
for i in range(N):  // corresponds to map
  for j in range(D):  // corresponds to fold
    for k in range(L):  // corresponds to scan
      if j == 0 and k == 0: // control region S0
        h_prev = zeros
        c_prev = zeros
        x = xss[i][k]
      elif j == 0 and k > 0: // control region S1
        h_prev = hsss[i][j][k - 1]
        c_prev = csss[i][j][k - 1]
        x = xss[i][k]
      elif j > 0 and k == 0: // control region S2
        h_prev = zeros
        c_prev = zeros
        x = hsss[i][j - 1][k]
      else:  // control region S3
        h_prev = output[i][j][k - 1]
        c_prev = output[i][j][k - 1]
        x = hsss[i][j - 1][k]

      h, c = lstm_cells[j](x, h_prev, c_prev) // the UDF
      hsss[i][j][k] = h
      csss[i][j][k] = c
```

Listing 4: Stacked RNN 的 imperative style 语法等价形, 这里把 UDF 替换成 LSTM 的 cell function

Listing 4 是 stacked RNN 网络的语法等价形式，唯一变化是这里我们把 UDF 替换成 LSTM cell：$c_t, h_t = \textbf{lstm\_cell}\left(\vec{x}_t, \vec{h}_{t-1}, \textbf{Params}\right)$，具体是由下面六个线性代数公式定义：

$$f_t = \sigma_g\left(\mathbf{W}_f \vec{x}_t + \mathbf{U}_f \vec{h}_{t-1} + \vec{b}_f\right) \tag{15}$$

$$i_t = \sigma_g\left(\mathbf{W}_i \vec{x}_t + \mathbf{U}_i \vec{h}_{t-1} + \vec{b}_i\right) \tag{16}$$

$$o_t = \sigma_g\left(\mathbf{W}_o \vec{x}_t + \mathbf{U}_o \vec{h}_{t-1} + \vec{b}_o\right) \tag{17}$$

$$\widetilde{c}_t = \sigma_c\left(\mathbf{W}_c \vec{x}_t + \mathbf{U}_c \vec{h}_{t-1} + \vec{b}_c\right) \tag{18}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \widetilde{c}_t \tag{19}$$

$$h_t = o_t \odot \sigma_h\left(c_t\right) \tag{20}$$

于是，我们会面对这样一个问题：***给定了这样一组嵌套在 loop nest 之中的线性代数公式：LSTM cell，是否有办法让一个 optimzing compiler 基于程序中的 general facts，同时给定硬件参数，自动推断出一个好的并行实现方案？*** （我们的思考甚至可以安全地忽略这个 loop nest 到底是用哪一种语法形式写出来的）。

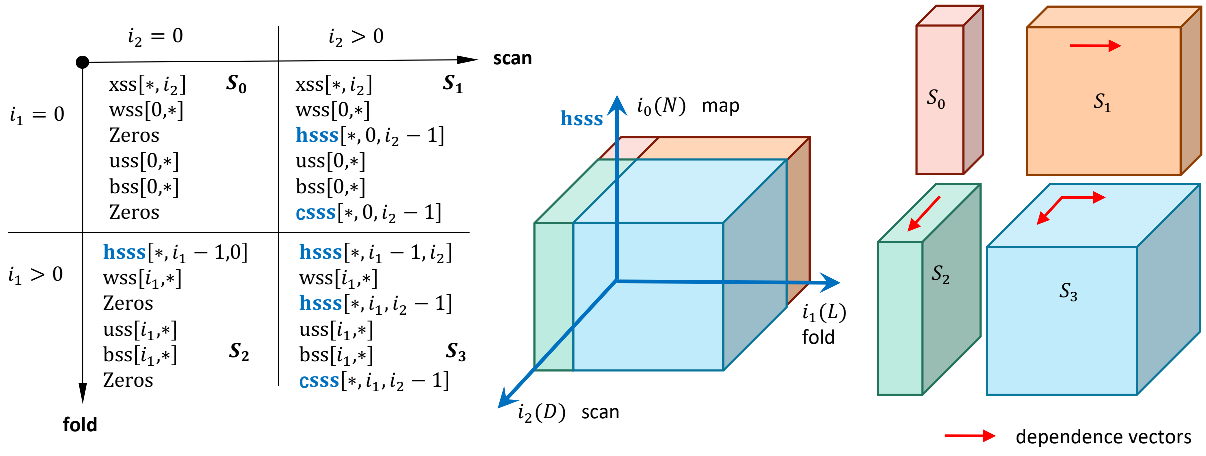


图 7: 编译 stacked RNNs 计算的复杂性

4.3 循环倾斜后循环边界的上下界的求解

约定 loop nest 深度的计数从循环最外层向循环体递减，也就是最外层层数最深，循环体深度记为 $d = 0$。

图7中数据流依赖最复杂的控制区域 s_3 用等价的 imperative for loop 写出来如下：

```
for i in range [0, N):   // map， 全并行
  for j in range [1, D):   // scan， 控制深度2携带数据流依赖
    for k in range 1, L):   // scan， 控制深度1携带数据流依赖
        UDF(...)
```

以上嵌套循环程序的迭代空间是一个三维立方体，可以用如下不等式表示：

$$\begin{aligned} 0 \le i < N \\ 1 \le j < D \\ 1 \le k < L \end{aligned} \tag{21}$$

如果我们把迭代空间看做是一个三维空间中的整数点集合，整个循环体看做一个 statement，记作 s_x。x 是迭代空间中的一个整数点，也就是 s 的一次执行。s_{i_1,j_1,k_1} 写，s_{i_2,j_2,k_2} 读同一个 buffer 位置，这两次执行之间存在数据流依赖。图6直观地揭示了另一个关于计算过程数据流依赖情况的事实：上面这个 loop nest 在迭代空间中携带了两类数据流，读和写之间的距离是：$d_1 = [0, 1, 0]$ 和 $d_2 = [0, 0, 1]$。loop nest 中的数据流依赖可以用如下 dependence distance vector 矩阵 M 描述：

$$\begin{bmatrix} \vec{d_1} & \vec{d_2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

我们要求解的一个调度问题是，给定 M 作为约束，求解一个变化矩阵 T，能够让最外层循环携带所有数据流依赖。这里我们暂时忽略矩阵 T 是如何得到的，只需要先记住 T 不唯一，T 的求解是一个 well-studied 问题，有完善的理论和工具。下面直接给出 T 的一种可能结果：

$$\delta_A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} j+k \\ i \\ j \end{bmatrix} = \begin{bmatrix} m \\ n \\ p \end{bmatrix}$$

于是有:

$$\begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} n \\ p \\ m-p \end{bmatrix} \tag{22}$$

我们会得以 m，n，p 为迭代变量的变化后的循环程序。循环体执行时，在遇到 i，j，k 的地方，带入(22)。那么下一个问题是，变化后的循环边界如何求解?（Fourier-Motzkin Elimination 算法就是专门解决这个问题的方法。isl 这样的 polyhedral 工具都集成了这一算法）

```
for m in range ?:  // map,  全并行
  for n in range ?:  // scan, 控制深度2携带数据流依赖
    for p in range ?:  // scan, 控制深度1携带数据流依赖
        UDF(...)
```

这里我们手算一下变化后的循环边界。

把(22)带入(21)我们会得到:

$$0 \leq n < N \tag{23}$$
$$1 \leq p < D \tag{24}$$
$$1 \leq m-p < L \tag{25}$$

最外层的循环边界必须是常数。将(24)和(25)相加我们能够得到最外层的循环边界:

$$2 \leq m < L+D-1$$

从(25)我们能得到: $m-L < p \leq m-1$，即: $m-L+1 \leq p < m$; 同时，p 必须满足(24)的约束。这两个不等式求交集就是 p 的上下界:

$$\max(1, m-L+1) \leq p < \min(m, D)$$

绿色标注的三个不等式就是变换后循环的上下界，我们会得到下面这样一个变化后的程序:

```
for m in range [2, L+D-1)  // 串行
  {  // 把所有可并行的数据batch成一个更大的数据, 插入对gather_nd的调用
    for n in range [0, N)  // 并行
      for p in range [max(1, m-L+1), min(m, D))  // 并行
          batched_data = gather(可并行的数据点)
  }
  UDF(batched_data)
```

Appendices

若存在可逆矩阵 P，使得一个关于矩阵 A 的如下等式成立：

$$A = (PDP)^{-1}$$

则称符合这样关系的矩阵 A 与 D 是相似矩阵，记作：$A \sim D$，则 A 的幂可以通过求矩阵 D 的幂求得

$$A^m = (PDP^{-1})^m = (PDP^{-1})(PDP^{-1})\ldots(PDP^{-1}) = PD^mP$$

如果我们能够得出 D 是一个很简单的矩阵，例如对角矩阵，那么就可以很简单的计算出 A 的幂值。然而，一般的矩阵在实数域不一定能对角化，然而几乎所有矩阵都能在复数域对角化 [2]。于是 A 总能写成：

$$A = P\Lambda P^{-1} A^m = P\Lambda^m P^{-1}$$

参考文献

[1] 线性 Attention 的探索：Attention 必须有个 Softmax 吗？. 科学网, 2020.

[2] Google 新作试图"复活"RNN：RNN 能否再次辉煌？. 科学网, 2020.

[3] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. arXiv preprint arXiv:2312.00752, 2023.

[4] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. arXiv preprint arXiv:2111.00396, 2021.

[5] Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. Transformer quality in linear time. In International Conference on Machine Learning, pages 9099–9117. PMLR, 2022.

[6] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In International conference on machine learning, pages 5156–5165. PMLR, 2020.

[7] Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. arXiv preprint arXiv:2303.06349, 2023.

[8] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. RWKV: Reinventing RNNs for the Transformer Era. arXiv preprint arXiv:2305.13048, 2023.

[9] Jimmy TH Smith, Andrew Warrington, and Scott W Linderman. Simplified state space layers for sequence modeling. arXiv preprint arXiv:2208.04933, 2022.

[10] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. arXiv preprint arXiv:2307.08621, 2023.

[11] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated Linear Attention Transformers with Hardware-Efficient Training. arXiv preprint arXiv:2312.06635, 2023.