SSM

Transformer架构虽然是当前大语言模型的主流选择,具有构建灵活、易并行、易扩展等优势,但也存在明显的局限性。其并行输入机制导致模型规模随输入序列长度呈平方增长,在处理长序列时面临严重的计算瓶颈。

前面介绍的各种KV Cache优化方法,虽然可以缓解长序列的计算瓶颈,但仍然无法从根本上解决Transformer架构在处理长序列时面临的计算瓶颈。

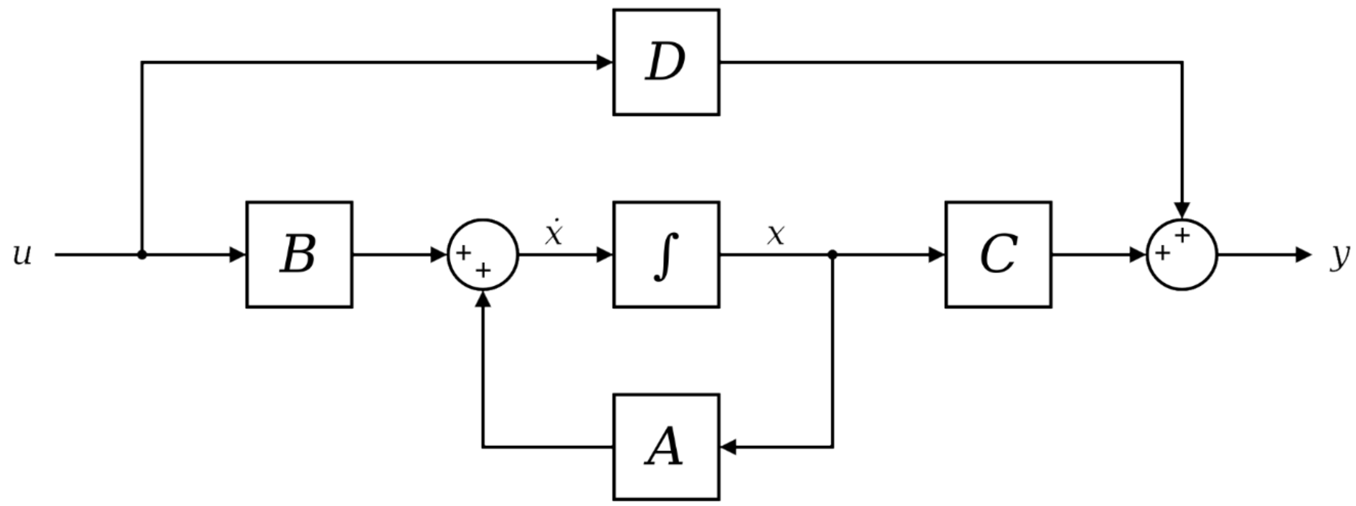
为了解决这一问题,研究者提出了基于RNN的替代方案。RNN在生成输出时只需考虑前一时刻的隐藏状态和当前输入,理论上可以处理任意长度的序列。但传统RNN模型(如GRU、LSTM)在处理长序列时往往难以捕获长期依赖关系,且容易出现梯度消失或爆炸。

针对这些挑战,近年来涌现出两类现代RNN变体:状态空间模型(State Space Model, SSM)和测试时训练(Test-Time Training, TTT)。这两种范式都实现了序列长度的线性时间复杂度,同时有效规避了传统RNN的固有问题。本节将重点介绍SSM范式及其代表性模型。

SSM原理

状态空间模型源自控制理论中的动力系统,通过一组状态变量来描述系统随时间的连续变化。这种连续时间的表示方法天然适合建模长期依赖关系。除了连续的形式，SSM还具有递归和卷积两种离散化表示形式:

- 递归形式支持高效的序列推理
- 卷积形式有助于捕获训练时的全局依赖



数学表示

设 $x(t) \in \mathbb{C}^n$ 表示n维状态向量, $u(t) \in \mathbb{C}^m$ 表示m维状态输入, $y(t) \in \mathbb{C}^p$ 表示p维输出。

系统由四个关键矩阵定义:

- 状态矩阵 $\mathbf{A} \in \mathbb{C}^{n\times n}$: 控制系统状态如何随时间演化,决定了系统的动态特性和稳定性
- 控制矩阵 $\mathbf{B} \in \mathbb{C}^{n\times m}$: 定义输入信号如何影响系统状态,控制外部输入对系统的作用方式
- 输出矩阵 $\mathbf{C} \in \mathbb{C}^{p\times n}$: 将内部状态映射到可观测的输出,决定了系统状态如何转换为最终输出
- 命令矩阵 $\mathbf{D} \in \mathbb{C}^{p\times m}$: 表示输入直接对输出的影响

SSM的系统方程包含两个部分:

状态方程：系统状态如何随时间变化:

$x'(t) = \mathbf{A}x(t) + \mathbf{B}u(t)$

其中 $x'(t)$ 表示状态向量的时间导数,等式右侧第一项 $\mathbf{A}x(t)$ 描述了系统的自然演化,第二项 $\mathbf{B}u(t)$ 描述了外部输入对系统状态的影响，可以通过积分来算出状态$x(t)$。

输出方程：如何从系统状态得到最终输出$y(t)$:

$y(t) = \mathbf{C}x(t) + \mathbf{D}u(t)$

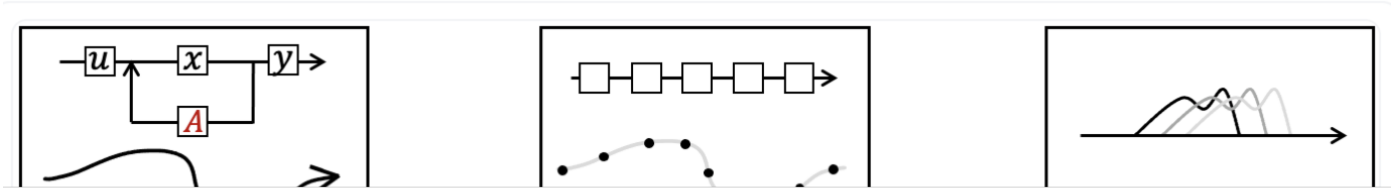
其中第一项 $\mathbf{C}x(t)$ 将内部状态映射为输出,第二项 $\mathbf{D}u(t)$ 表示输入对输出的直接影响。

其中状态方程描述系统状态的演化规律,输出方程定义状态到输出的映射。在深度学习应用中,通常可以忽略残差项 $\mathbf{D}u(t)$。

离散化表示

上面的方程是连续形式的SSM,为提高计算效率,需要将连续形式的SSM离散化。

离散化是SSM中最重要的步骤，它使我们能够从SSM的连续形式传递到其另外两个形式：递归和卷积。



> 梯形法:一种数值积分方法,通过将积分区间分成若干小区间,用梯形面积来近似曲线下的面积:
>
> $\int_a^b f(x)dx \approx \frac{h}{2}[f(a) + f(b)]$

通过梯形法代替积分操作，得到递归形式:

$$\begin{aligned} x_k &= \overline{\mathbf{A}}x_{k-1} + \overline{\mathbf{B}}u_k \\ y_k &= \overline{\mathbf{C}}x_k \end{aligned} \tag{1}$$

其中离散化矩阵与连续形式的关系为:

- $\overline{\mathbf{A}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\left(\mathbf{I} + \frac{\Delta}{2}\mathbf{A}\right)$
- $\overline{\mathbf{B}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\Delta\mathbf{B}$
- $\overline{\mathbf{C}} = \mathbf{C}$
- Δ 为时间步长

- $\mathbf{I}$ 为单位矩阵

让我们来详细推导一下离散化矩阵 $\overline{\mathbf{A}}, \overline{\mathbf{B}}, \overline{\mathbf{C}}$ 是如何得到的。

首先，对状态方程 $x'(t) = \mathbf{A}x(t) + \mathbf{B}u(t)$ 在时间区间 $[t_{k-1}, t_k]$ 上积分：

$$\int_{t_{k-1}}^{t_k} x'(t)dt = \int_{t_{k-1}}^{t_k} \mathbf{A}x(t)dt + \int_{t_{k-1}}^{t_k} \mathbf{B}u(t)dt \tag{2}$$

左边是状态的变化量：$x_k - x_{k-1}$

对右边第一项使用梯形法近似：

$$\int_{t_{k-1}}^{t_k} \mathbf{A}x(t)dt \approx \frac{\Delta}{2}(\mathbf{A}x_k + \mathbf{A}x_{k-1}) \tag{3}$$

对右边第二项，假设在一个时间步内输入保持不变，即 $u(t) = u_k$：

$$\int_{t_{k-1}}^{t_k} \mathbf{B}u(t)dt = \Delta\mathbf{B}u_k \tag{4}$$

代入原方程：

$$x_k - x_{k-1} = \frac{\Delta}{2}(\mathbf{A}x_k + \mathbf{A}x_{k-1}) + \Delta\mathbf{B}u_k \tag{5}$$

整理得：

$$x_k - x_{k-1} = \frac{\Delta}{2}\mathbf{A}x_k + \frac{\Delta}{2}\mathbf{A}x_{k-1} + \Delta\mathbf{B}u_k \tag{6}$$

$$x_k - \frac{\Delta}{2}\mathbf{A}x_k = x_{k-1} + \frac{\Delta}{2}\mathbf{A}x_{k-1} + \Delta\mathbf{B}u_k \tag{7}$$

$$(\mathbf{I} - \frac{\Delta}{2}\mathbf{A})x_k = (\mathbf{I} + \frac{\Delta}{2}\mathbf{A})x_{k-1} + \Delta\mathbf{B}u_k \tag{8}$$

最终得到：

$$x_k = (\mathbf{I} - \frac{\Delta}{2}\mathbf{A})^{-1}(\mathbf{I} + \frac{\Delta}{2}\mathbf{A})x_{k-1} + (\mathbf{I} - \frac{\Delta}{2}\mathbf{A})^{-1}\Delta\mathbf{B}u_k \tag{9}$$

因此：

- $\overline{\mathbf{A}} = (\mathbf{I} - \frac{\Delta}{2}\mathbf{A})^{-1}(\mathbf{I} + \frac{\Delta}{2}\mathbf{A})$
- $\overline{\mathbf{B}} = (\mathbf{I} - \frac{\Delta}{2}\mathbf{A})^{-1}\Delta\mathbf{B}$
- $\overline{\mathbf{C}} = \mathbf{C}$ (输出方程不需要离散化)

递归形式进一步迭代可得到卷积形式:

$y_k = \overline{\mathbf{K}}_k * u_k$

其中卷积核 $\overline{\mathbf{K}}_k = \left(\overline{\mathbf{C}}\overline{\mathbf{B}}, \overline{\mathbf{C}}\overline{\mathbf{A}}\overline{\mathbf{B}}, \ldots, \overline{\mathbf{C}}\overline{\mathbf{A}}^k\overline{\mathbf{B}}\right)$

让我们推导一下为什么递归形式可以迭代得到卷积形式。

从递归形式开始:
$x_k = \overline{\mathbf{A}}x_{k-1} + \overline{\mathbf{B}}u_k$
$y_k = \overline{\mathbf{C}}x_k$

将第一个方程迭代展开:

$$\begin{aligned} x_k &= \overline{\mathbf{A}}x_{k-1} + \overline{\mathbf{B}}u_k \\ &= \overline{\mathbf{A}}(\overline{\mathbf{A}}x_{k-2} + \overline{\mathbf{B}}u_{k-1}) + \overline{\mathbf{B}}u_k \\ &= \overline{\mathbf{A}}^2 x_{k-2} + \overline{\mathbf{A}\mathbf{B}}u_{k-1} + \overline{\mathbf{B}}u_k \\ &= \overline{\mathbf{A}}^3 x_{k-3} + \overline{\mathbf{A}}^2\overline{\mathbf{B}}u_{k-2} + \overline{\mathbf{A}\mathbf{B}}u_{k-1} + \overline{\mathbf{B}}u_k \\ &= \cdots \\ &= \overline{\mathbf{A}}^k x_0 + \sum_{i=0}^{k-1} \overline{\mathbf{A}}^i \overline{\mathbf{B}}u_{k-i} \end{aligned} \tag{10}$$

代入输出方程:

$$\begin{aligned} y_k &= \overline{\mathbf{C}}x_k \\ &= \overline{\mathbf{C}\mathbf{A}}^k x_0 + \overline{\mathbf{C}} \sum_{i=0}^{k-1} \overline{\mathbf{A}}^i \overline{\mathbf{B}}u_{k-i} \\ &= \overline{\mathbf{C}\mathbf{A}}^k x_0 + \sum_{i=0}^{k-1} (\overline{\mathbf{C}\mathbf{A}}^i \overline{\mathbf{B}})u_{k-i} \end{aligned} \tag{11}$$

假设初始状态 $x_0 = 0$，则:
$y_k = \sum_{i=0}^{k-1} (\overline{\mathbf{C}\mathbf{A}}^i \overline{\mathbf{B}})u_{k-i}$

这正是卷积形式:
$y_k = \overline{\mathbf{K}}_k * u_k$

其中卷积核 $\overline{\mathbf{K}}_k = (\overline{\mathbf{C}\mathbf{B}}, \overline{\mathbf{C}\mathbf{A}\mathbf{B}}, \ldots, \overline{\mathbf{C}\mathbf{A}}^{k-1}\overline{\mathbf{B}})$

这说明SSM的输出可以看作是输入序列与一个特定卷积核的卷积运算。这种形式特别适合并行计算，因为卷积可以通过FFT等算法高效实现。

SSM代码示例

```
import torch
import torch.nn as nn

class SSM(nn.Module):
    def __init__(self, input_dim, state_dim, output_dim):
        """
        初始化SSM模型
        Args:
            input_dim: 输入维度
            state_dim: 隐状态维度
            output_dim: 输出维度
```

```python
        """
        super().__init__()
        # 初始化状态空间参数 A[state_dim, state_dim], B[state_dim, input_dim],
C[output_dim, state_dim]
        self.A = nn.Parameter(torch.randn(state_dim, state_dim))  # 状态转移矩阵
        self.B = nn.Parameter(torch.randn(state_dim, input_dim))  # 输入矩阵
        self.C = nn.Parameter(torch.randn(output_dim, state_dim)) # 输出矩阵
        self.state_dim = state_dim

    def forward_recursive(self, x):
        """
        递归形式的前向传播
        实现公式:
        h_t = Ah_{t-1} + Bx_t
        y_t = Ch_t

        Args:
            x: 输入张量 [batch_size, seq_len, input_dim]
        Returns:
            输出张量 [batch_size, seq_len, output_dim]
        """
        batch_size, seq_len, input_dim = x.shape
        device = x.device

        # 初始化隐状态 h_0 = 0
        # h shape: [batch_size, state_dim]
        h = torch.zeros(batch_size, self.state_dim, device=device)
        outputs = []

        # 按时间步迭代
        for t in range(seq_len):
            # x_t shape: [batch_size, input_dim]
            x_t = x[:, t, :]

            # 更新隐状态: h_t = Ah_{t-1} + Bx_t
            # h shape: [batch_size, state_dim]
            h = torch.matmul(h, self.A.T) + torch.matmul(x_t, self.B.T)

            # 计算输出: y_t = Ch_t
            # y_t shape: [batch_size, output_dim]
            y_t = torch.matmul(h, self.C.T)
            outputs.append(y_t)

        # 最终输出 shape: [batch_size, seq_len, output_dim]
        return torch.stack(outputs, dim=1)

    def forward_convolutional(self, x):
        """
        卷积形式的前向传播
```

```
        实现公式：
        y_k = \sum_{i=0}^{k-1}
(\overline{\mathbf{C}}\overline{\mathbf{A}}^i\overline{\mathbf{B}})u_{k-i}

        Args:
            x: 输入张量 [batch_size, seq_len, input_dim]
        Returns:
            输出张量 [batch_size, seq_len, output_dim]
        """
        batch_size, seq_len, input_dim = x.shape
        device = x.device

        # 构建卷积核 K = (CB, CAB, CA^2B, ..., CA^{k-1}B)
        kernel = []
        A_power = torch.eye(self.state_dim, device=device)  # 初始为A^0
        for t in range(seq_len):
            # k_t shape: [output_dim, input_dim]
            k_t = torch.matmul(torch.matmul(self.C, A_power), self.B)
            kernel.append(k_t)
            A_power = torch.matmul(A_power, self.A)  # 计算下一个A幂次

        # kernel shape: [seq_len, output_dim, input_dim]
        kernel = torch.stack(kernel, dim=0)

        # 转换输入shape以便进行卷积运算
        # x_reshaped shape: [batch_size, input_dim, seq_len]
        x_reshaped = x.transpose(1, 2)

        # 执行卷积运算
        y = []
        for i in range(batch_size):
            # y_i shape: [seq_len, output_dim]
            y_i = torch.zeros(seq_len, self.C.shape[0], device=device)
            for t in range(seq_len):
                for tau in range(min(t + 1, seq_len)):
                    y_i[t] += torch.matmul(kernel[tau], x_reshaped[i, :, t-tau])
            y.append(y_i)

        # 最终输出 shape: [batch_size, seq_len, output_dim]
        return torch.stack(y, dim=0)

    def forward(self, x, mode='recursive'):
        """
        前向传播，支持递归和卷积两种模式
        Args:
            x: 输入张量 [batch_size, seq_len, input_dim]
            mode: 'recursive' 或 'convolutional'
        Returns:
            输出张量 [batch_size, seq_len, output_dim]
```

```python
        """
        if mode == 'recursive':
            return self.forward_recursive(x)
        elif mode == 'convolutional':
            return self.forward_convolutional(x)
        else:
            raise ValueError(f"不支持的模式: {mode}")

# 使用示例
if __name__ == "__main__":
    # 创建SSM模型实例
    input_dim = 4
    state_dim = 8
    output_dim = 2
    ssm = SSM(input_dim, state_dim, output_dim)

    # 生成测试数据
    batch_size = 3
    seq_len = 10
    x = torch.randn(batch_size, seq_len, input_dim)  # [3, 10, 4]

    # 使用递归模式
    y_recursive = ssm(x, mode='recursive')  # [3, 10, 2]
    print("递归模式输出形状:", y_recursive.shape)

    # 使用卷积模式
    y_conv = ssm(x, mode='convolutional')  # [3, 10, 2]
    print("卷积模式输出形状:", y_conv.shape)
```

总结

前面介绍了SSM的三种表示形式，其各有优势:

- 连续形式适合理论分析
- 递归形式适合序列推理
- 卷积形式适合并行训练

在实际应用中,通常采用"训练时卷积,推理时递归"的策略,以平衡计算效率和模型性能。

目前SSM的主要变体包括:

- S4(Structured State Space Model):使用HiPPO矩阵初始化状态矩阵,特别适合处理长序列
- RWKV和Mamba:两种广受关注的SSM架构实现

这些变体主要在离散化方式和状态矩阵定义上有所创新。下面我们将详细介绍RWKV和Mamba的具体实现。