RWKV

RWKV（Receptance Weighted Key Value）是一个基于SSM设计思想的语言模型架构创新。它巧妙地结合了RNN的高效推理和Transformer的并行训练优势,通过独特的状态更新机制来处理序列数据。让我们一起来了解这个优雅的模型设计。

RWKV的演进历程

- v1：首次提出RWKV架构,用全新的RNN序列建模机制取代了传统注意力机制
- v2：通过改进位置编码方案,让模型更好地理解序列中的位置信息
- v3：优化了激活函数和归一化层,提升了模型的稳定性和表现
- v4：在长文本处理和多语言支持方面取得重大突破,进一步提升了训练效率和性能

RWKV架构解析

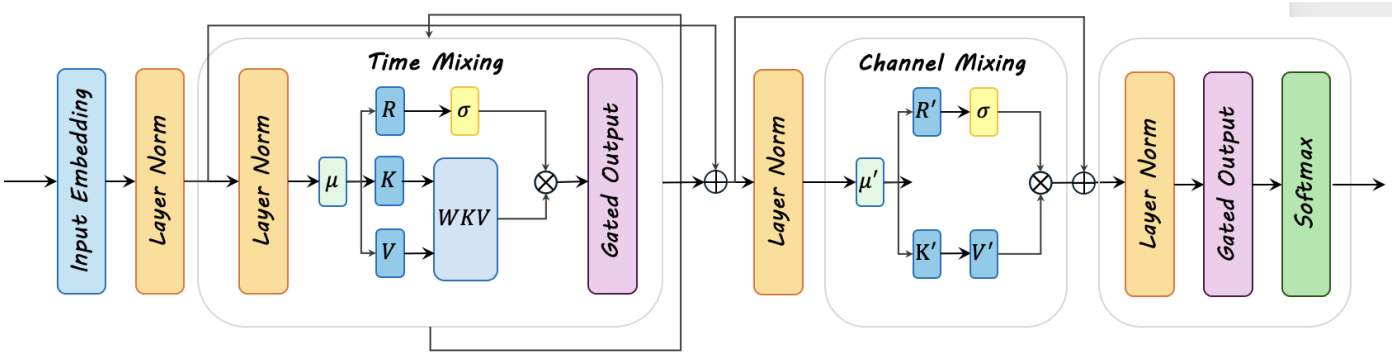
RWKV-v4的核心由两个精心设计的模块组成：

- **时间混合模块（Time Mixing）**：负责捕捉序列中不同时间步的关联,让模型能够理解长期依赖关系
- **通道混合模块（Channel Mixing）**：处理同一时刻不同特征之间的互动,增强对局部特征的理解

这两个模块采用了一种全新的状态更新机制,不再依赖传统Transformer中的查询(query)、键(key)、值(value)概念。

模块设计中的四个关键角色：

- 接收向量(Receptance Vector, R)：像一个智能门卫,控制信息的流动
- 键向量(Key Vector, K)：决定每个信息的重要程度
- 值向量(Value Vector, V)：携带实际的信息内容
- 权重(Weight, W)：平衡各个组件的贡献度



时间混合模块的工作原理

时间混合模块处理序列信息的方式和RNN类似：

1. 对于第一个时间混合模块,它直接处理模型的初始输入(通常是token的嵌入向量)
2. 后续的时间混合模块则处理前一个通道混合模块传来的信息

每个时间步都会将输入向量转换为R、K、V三个向量,然后通过精心设计的WKV机制和门控机制进行处理。

时间混合模块和通道混合模块都包含一个巧妙的Token位移操作,通过对当前和前一时间步的输入进行混合,帮助模型理解时序变化。

接收向量R负责处理历史信息。权重W决定了历史信息随时间的衰减程度,而键向量K和值向量V则分别负责评估信息的重要性和携带实际内容。模块先将当前步和前一步的输入做线性变换得到R、K、V这三个向量。接着使用WKV机制确保信息会随时间自然衰减。最后,将历史信息$\sigma(R)$和当前信息WKV通过门控机制智能地整合,传递给下一个模块。

WKV机制是RWKV最核心的部分。权重W是一个随时间衰减的向量：$w_{t,i} = -(t-i)w$。这里i表示从当前时间t往回数的步数,w是一个非负向量。这种设计让模型能够自然地理解序列中的时间依赖关系。WKV机制的巧妙之处在于使用线性时不变递归来更新状态,本质上可以看作两个SSM的比值。

具体公式为：

$$wkv_t = \frac{\sum_{i=1}^{t-1} e^{-(t-1-i)w+k_i} \odot v_i + e^{u+k_t} \odot v_t}{\sum_{i=1}^{t-1} e^{-(t-1-i)w+k_i} + e^{u+k_t}} \tag{1}$$

这个公式中：

- $\odot$ 表示向量的逐元素乘法
- u 是一个可学习的参数,用来调节当前Token的重要性。当u值大时,模型会更关注当前输入;当u值小时,则更重视历史信息
- WKV公式的分子和分母都包含了状态更新和当前输入的整合。这种结构与SSM的状态转移非常相似,所以我们可以把WKV理解为两个SSM的比值运算
- 通过权重W的指数衰减,远处的历史信息会自然淡化,而分式结构保证了梯度能稳定传播。这样既能处理长序列,又避免了传统RNN的梯度问题

让我们深入理解这个WKV公式：

1. 分子部分:
   $\sum_{i=1}^{t-1} e^{-(t-1-i)w+k_i} \odot v_i + e^{u+k_t} \odot v_t$
   包含两个重要组成：
   - 历史信息的累积：$\sum_{i=1}^{t-1} e^{-(t-1-i)w+k_i} \odot v_i$
     - $e^{-(t-1-i)w}$ 控制时间衰减
     - e^{k_i} 决定历史信息的重要性
     - v_i 携带实际信息
   - 当前信息的处理：$e^{u+k_t} \odot v_t$
     - e^{u} 是可学习的权重调节器
     - e^{k_t} 评估当前信息的重要性
     - v_t 携带当前时刻的信息
2. 分母部分:
   $\sum_{i=1}^{t-1} e^{-(t-1-i)w+k_i} + e^{u+k_t}$
   - 通过归一化确保所有权重和为1
   - 结构与分子相似,但不包含实际信息
3. 整体效果:

- 分式结构实现了智能的注意力分配
- 指数衰减让远处信息自然变小
- 可学习参数让模型能动态调整信息重要性
- 逐元素运算让每个特征通道独立选择信息

这个公式本质上是一个带时间衰减的智能加权平均,权重由key向量和时间衰减共同决定,实际信息由value向量承载。这种设计既保留了注意力机制的优点,又通过递归实现了高效计算。

计算完WKV后,模型使用sigmoid激活函数σ和接收向量R来智能控制信息流:

1. Sigmoid激活函数σ的作用:
   - 将接收向量R映射到(0,1)区间
   - 让R成为一个灵活的"软开关",控制历史信息的影响程度
   - 当$\sigma(R)$接近1时,保留大量历史信息
   - 当$\sigma(R)$接近0时,过滤掉大部分历史信息
2. 输出门控机制:
   - 通过$Output = \sigma(R) \odot WKV$整合信息
   - 这种设计让模型能:
     - 灵活调节历史和当前信息的比例
     - 在不同时刻智能选择信息
     - 有效避免梯度消失
   - 这种门控思路与LSTM有异曲同工之妙

通过σ和输出门控的配合,RWKV实现了:

- 信息流的智能调控
- 长期依赖的有效建模
- 梯度的稳定传播

通道混合模块的精妙设计

通道混合模块是RWKV的另一个关键设计,专注于特征通道间的信息交互。

核心组件

接收向量R':与时间混合模块类似,负责控制信息流动

键向量K':评估信息的重要程度

值向量V':携带实际的信息内容

这些向量的角色与时间混合模块相似,但关注点从时序转向了通道维度。

处理流程

1. 通过巧妙的线性变换得到R'和K'向量
2. 基于K'计算V',实现通道间的有效互动
3. 使用sigmoid函数$\sigma\left(\mathrm{R}'\right)$来智能控制信息流动
4. 将$\sigma\left(\mathrm{R}'\right)$与$\mathrm{V}'$结合,完成通道间的信息融合

时间感知的Softmax

时间依赖的Softmax操作:
$\mathrm{TimeSoftmax}(x_t) = \frac{e^{x_t}}{\sum_{i=1}^{t} e^{x_i}}$

这个设计有三个巧妙之处:

- 严格遵循因果关系,只考虑已有信息
- 通过指数运算突出重要特征
- 通过归一化让不同时刻的贡献可比

这种设计还能提升计算稳定性,改善梯度传播。

RWKV的突出优势

1. 模型规模的突破

- 成功扩展到14B参数规模
- 开创了非Transformer架构的新纪录
- 证明了架构的良好扩展性

2. 出色的计算效率

- 训练时享受Transformer的并行优势
- 推理时保持RNN的O(1)效率
- 内存使用与序列长度无关,非常高效

3. 强劲的性能表现

- NLP任务上媲美同规模Transformer
- 长序列处理能力仅次于S4
- 在各类任务中表现稳定出色

示例代码

以下代码为RWKV核心部分的简单实现(忽略了LayerNorm和softmax)。

```
import torch
import torch.nn as nn
import math
```

```python
class RWKVBlock(nn.Module):
    def __init__(self, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size

        # 时间混合模块参数
        self.time_decay = nn.Parameter(torch.zeros(hidden_size))  # w参数
        self.time_first = nn.Parameter(torch.zeros(hidden_size))  # u参数

        # 时间混合的线性变换
        self.time_mix_r = nn.Linear(hidden_size, hidden_size)
        self.time_mix_k = nn.Linear(hidden_size, hidden_size)
        self.time_mix_v = nn.Linear(hidden_size, hidden_size)

        # 通道混合模块参数
        self.channel_mix_r = nn.Linear(hidden_size, hidden_size)
        self.channel_mix_k = nn.Linear(hidden_size, hidden_size)

    def time_mixing(self, x, state=None):
        # 生成R、K、V向量
        r = self.time_mix_r(x)
        k = self.time_mix_k(x)
        v = self.time_mix_v(x)

        # 计算时间衰减
        time_decay = torch.exp(-torch.exp(self.time_decay))
        k = torch.exp(k)

        if state is not None:
            # 完整的WKV计算
            # 当前时刻的贡献
            current_contribution = torch.exp(self.time_first + k) * v

            # 历史信息的贡献（使用累积状态）
            numerator = state['num'] * time_decay + current_contribution
            denominator = state['den'] * time_decay + torch.exp(self.time_first + k)

            # 计算WKV
            wkv = numerator / (denominator + 1e-6)  # 添加小值避免除零

            # 更新状态
            new_state = {
                'num': numerator,
                'den': denominator
            }
        else:
            # 首个时间步的处理
            wkv = v
            new_state = {
```

```
                'num': torch.exp(k) * v,
                'den': torch.exp(k)
            }

        # 使用接收向量R进行门控
        output = torch.sigmoid(r) * wkv
        return output, new_state

    def channel_mixing(self, x):
        # 通道混合的实现
        r = self.channel_mix_r(x)
        k = self.channel_mix_k(x)

        # 使用接收向量进行门控
        return torch.sigmoid(r) * k

    def forward(self, x, state=None):
        # 时间混合
        time_mix_out, new_state = self.time_mixing(x, state)

        # 通道混合
        channel_mix_out = self.channel_mixing(x)

        # 组合输出
        out = time_mix_out + channel_mix_out

        return out, new_state

# 使用示例
def test_rwkv():
    hidden_size = 512
    rwkv_block = RWKVBlock(hidden_size)

    # 模拟输入序列
    batch_size = 1
    seq_len = 10
    x = torch.randn(batch_size, seq_len, hidden_size)

    # 初始状态为None
    state = None

    # 按序列逐步处理
    outputs = []
    for t in range(seq_len):
        out, state = rwkv_block(x[:, t:t+1, :], state)
        outputs.append(out)

    # 合并所有输出
    final_output = torch.cat(outputs, dim=1)
```

```
    print(f"输出张量形状: {final_output.shape}")

    return final_output

if __name__ == "__main__":
    test_rwkv()
```

总结

虽然RWKV表现优秀,但仍有两个值得关注的方向：提升长距离依赖的建模能力,以及增强在复杂任务上的表现。

值得期待的是,新兴的Mamba架构为解决这些挑战提供了新思路。相信随着研究的深入,RWKV在长序列建模方面会有更大突破。

RWKV通过创新的线性注意力机制,巧妙地融合了Transformer的并行优势和RNN的高效特点,为大规模语言模型开辟了一条新路。