轻量级大语言模型MiniMind源码解读（二）：为什么RMSNorm更适合大模型推理?

Original 凡希 南极Python 2025年8月14日 15:05

一、RMSNorm是什么

RMSNorm是一种简单高效的**归一化方法**，用于归一化神经网络中某一层的输出，使其数值保持稳定，常用于Transformer中。

给定输入向量x（x的shape为 `[batch_size, seq_length, embedding_dim]`），RMSNorm的计算方式为：

$$\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}$$

$$\text{RMSNorm}(x) = \frac{x}{\text{RMS}(x)} \cdot \gamma$$

其中：

- d 是token特征维度数
- ϵ是防止除以零的小常数
- γ是可训练的缩放参数

```
import torch
import torch.nn as nn

class RMSNorm(nn.Module):
    def __init__(self, dim: int, eps: float = 1e-5):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))  # 可学习的缩放参数 γ
        # print(self.weight.shape)# torch.Size([4])

    def _norm(self, x):
        # 均方根归一化：沿最后一维计算
        # torch.rsqrt返回的是x.pow(2).mean(-1, keepdim=True) + self.eps的平方根的倒数
        # 直接调用 rsqrt 比先 sqrt 再 1 / 更高效，尤其在 GPU 上
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
    def forward(self, x):
        # print(self._norm(x.float()).shape)# # torch.Size([1, 2, 4])
        return self.weight * self._norm(x.float()).type_as(x)


# 实例化测试
x = torch.tensor([[[1.0, 2.0, 3.0, 4.0],
                   [5.0, 6.0, 7.0, 8.0]]])  # shape = (1, 2, 4)
```

```
norm = RMSNorm(dim=4)
output = norm(x)
print(output.shape)# torch.Size([1, 2, 4])
```

二、RMSNorm和LayerNorm的异同点

可以看到，RMSNorm和LayerNorm一样，归一化操作都是沿着每个token内部的特征维度（也就是输入x的embedding_dim维度）进行的，而不是沿着整个batch维度。这意味着它们对每个token都进行独立的标准化，而不是跨batch的归一化。

同时，相较于LayerNorm，RMSNorm不做 **减均值**的操作。这里把LayerNorm的计算公式和实现代码搬过来做一下对比：

$$\text{LayerNorm}(\mathbf{x}) = \gamma \left(\frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta$$

```
class LayerNorm(nn.Module):
    def __init__(self, dim: int, eps: float = 1e-5):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))  # γ
        self.bias = nn.Parameter(torch.zeros(dim))   # β

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)             # 计算每个 token 的均值
        var = x.var(dim=-1, unbiased=False, keepdim=True)  # 方差
        x_norm = (x - mean) / torch.sqrt(var + self.eps)   # 标准化
        return self.weight * x_norm + self.bias


# 实例化测试
x = torch.tensor([[[1.0, 2.0, 3.0, 4.0],
                   [5.0, 6.0, 7.0, 8.0]]])  # shape = (1, 2, 4)
norm = LayerNorm(dim=4)
output = norm(x)
print(output.shape)# torch.Size([1, 2, 4])
```

可以看到，当均值为零时，LayerNorm就变成了RMSNorm。

三、为什么使用RMSNorm而不是LayerNorm?

RMSNorm的计算过程比LayerNorm更简单，因为它不涉及均值的计算，并且减少了一个可学习参数。LayerNorm在归一化时需要计算每个token的均值和方差，并使用它们来标准化输入。而RMSNorm只需要计算特征的平方和，减少了计算复杂度和内存消耗。

在处理大型模型时，输入的特征维度可能非常大，计算均值和方差的开销相对较大。RMSNorm去除了均值计算，因此可以节省计算资源，特别是在高维数据中，计算效

率更高。

作者在各种场景中实验发现，使用RMSNorm能够减少约7%~64%的计算时间。

四、引申：为什么RMSNorm和LayerNorm都在token特征维度上操作而非跨batch?

BatchNorm是在处理图像数据时常用的归一化方式。

图像数据通常有强烈的空间相关性，即相邻的像素通常会有相似的值或模式。因此，图像的像素特征在一个batch中通常有相似的分布，这使得在整个batch上做归一化是合理的。BatchNorm通过计算每个特征（比如每个通道）的均值和方差，能有效地减轻这些空间相关性带来的影响，并保证训练时每一层的输入保持一定的分布，从而加速收敛。

而在NLP任务中，每个token通常是一个具有特定语义和上下文信息的单位，比如每个token代表一个词。每个token的特征是通过模型的embedding层或Transformer层计算得到的，并包含了该token的语义信息。不同token的语义内容不同，所以它们的特征应该独立地进行归一化处理。

如果归一化操作发生在batch维度上，会导致不考虑每个token的独立性。用于归一化的数据来自不同的batch，包含不同的token内容和信息，如果跨batch进行标准化，会丢失token间的独立性，使得token之间存在耦合关系，比如一些padding token并没有实际意义，但是被加入了归一化计算，进而影响模型的学习效果。

（如果你也想从0到1看懂大语言模型的底层实现，欢迎关注我，系列文章将持续更新，我们一起把大模型“拆干净”）

修改于2025年8月14日