1 Pre-Norm Transformer Block

在我们将要实现的架构中，每个 Transformer block 都有两个 sub-layers：a multi-head self-attention mechanism and a position-wise feed-forward network。

原始的 Transformer 论文使用的是 "Post-Norm" 设计，即先进行子层计算（注意力或前馈网络），然后进行残差连接，最后才进行层归一化（Layer Normalization）。后续研究发现将层归一化移动到*每个子层的输入端（即在进入子层之前先做归一化），并在所有 Transformer 块之后再加一个最终的归一化层，可以提高训练的稳定性。这种结构被称为 "Pre-Norm" Transformer。采用 Pre-Norm 的直观理解是，它保留了一条从输入嵌入（Embedding）直通最终输出的干净的“残差流（residual stream）”，中间没有经过归一化的阻断，这被认为有助于改善梯度的流动。现代的 transformer 都会选择 pre-norm 或至少将 layernorm 放在残差流之外以获得流畅的反向传播与稳定的训练过程。

1.1 Root Mean Square Layer Normalization

1.1.1 为何 transformer 中要使用 layer-norm 而非 batch-norm?

> 一个维度为$(batch_size = 3, seq_len = 4, d_model = 5)$的三维张量，可以把它想象成一个 3 层高的蛋糕，每一层是一个 4行 5列 的矩阵。

Batch Normalization (纵向切片)

- **统计对象**：BatchNorm 关注的是“在一个特征通道上，整个批次的数据分布如何”。
- **计算方式**：它固定特征维度 C（即 d_{model} 中的某一位），在 `(Batch_Size, Seq_Len)` 这两个维度上求均值和方差。

Layer Normalization (横向切片)

- **统计对象**：LayerNorm 关注的是"在一个样本（或在一个 Token），它自身的特征分布如何"。
- **计算方式**：它固定样本索引 N 和序列位置 L，在 C（即 d_{model}）这个维度上求均值和方差。

1.1.2 为什么 Transformer 必须要用 LayerNorm?

Transformer 处理的是文本（序列数据），这与 CNN 处理图像有很大不同，导致 BatchNorm 在这里水土不服。

原因一：序列长度可变

- **问题**：在 NLP 中，一个 Batch 里的句子长短不一（如 "Hello" 和 "I am studying statistics"）。为了打包成 Tensor，短句子后面填满了 `Padding`（通常是 0）。
- **BN 的缺陷**：如果用 BatchNorm，那些大量的 `Padding 0` 会参与均值和方差的计算，导致计算出的统计量严重有偏（Bias），不能反映真实数据的分布。
- **LN 的优势**：LayerNorm 是对每个Token 独立计算的。"Hello" 算它自己的归一化，"statistics" 算它自己的。它根本不在乎同一个 Batch 里其他句子有多长，也不受 Padding 的影响。

原因二：Batch Size 的限制

- **问题**：Transformer（尤其是大语言模型）非常庞大，显存占用极高。在训练时，我们往往只能开很小的 Batch Size（有时甚至只有 1 或 2）。
- **BN 的缺陷**：BatchNorm 极其依赖 Batch Size 的大小。根据统计学的大数定律，样本太少时，估算的均值和方差非常不稳定（噪声大），这会导致训练震荡甚至无法收敛。

- **LN 的优势**：LayerNorm 的计算完全独立于 Batch Size。无论 Batch Size 是 1 还是 1000，通过 LayerNorm 算出来的结果都是一样的。这对于大模型训练至关重要。

原因三：特征的物理含义

- **CNN (适合 BN)**：在图像中，Channel 1 可能专门检测“边缘”，Channel 2 检测“圆圈”。这些特征在不同图片中的位置相对固定，跨样本归一化是有意义的。
- **Transformer (适合 LN)**：在词向量空间中，同一个维度在不同词向量中的含义非常复杂且不固定。更重要的是，我们需要保留词向量的方向（语义），通过在向量内部做归一化（LN），可以起到一种类似“把模长统一”的效果，让点积（Attention）计算更加稳定，同时保留了向量内部各维度之间的相对大小关系。

总结

- **Batch Norm**：依赖 Batch，适合固定长度、对绝对数值敏感的任务（如 CV 里的图像分类）。
- **Layer Norm / RMSNorm**：独立于 Batch，适合变长序列、对向量方向敏感的任务（如 NLP）。

1.1.3 为什么要使用 RMSNorm?

在 lecture3 中讲到：减去均值”这一步对于 Transformer 的训练并不是必须的，真正重要的是缩放（Scaling），即限制激活值的大小。

于是 RMSNorm 诞生了：它直接省去了减去均值的步骤，只通过 均方根 (Root Mean Square, RMS) 进行缩放。这样做不仅效果好，而且计算量更小。

给定一个维度为 d_{model} 的输入向量 a（即模型中的激活值），RMSNorm 的计算过程如下：

$$\text{RMSNorm}(a) = \frac{a}{\text{RMS}(a)} \cdot g_i$$

其中：$\text{RMS}(a) = \sqrt{\frac{1}{d_{model}} \sum_{i=1}^{d_{model}} a_i^2 + \epsilon}$，$g$ 是一个可学习的“增益”参数，是一个向量，长度为 d_{model}，对应每一个特征维度。

> 如何理解可学习的“增益”参数 g_i？
>
> - 恢复表达能力 (Restore Expressive Power)：归一化操作（除以 RMS）是一个“暴力”的过程，它强行把每个 Token 向量的尺度（Scale）拉回到了同一个水平（方差为 1 左右）。这虽然稳定了梯度，但也限制了模型权重的表达自由度。也许对于某些特征维度，模型希望它的数值范围大一点（表示信号强），而另一些小一点。g_i 允许模型在归一化之后，重新学习出最适合当前任务的特征缩放比例。如果没有 g_i，这层网络的输出就被“锁死”在归一化后的分布上了。
> - 维度独立性：注意文档中提到共有 d_{model} 个这样的参数。这意味着每一个特征维度都有自己独立的 g_i。比如第 1 个维度的 g_1 可以是 10，第 2 个维度的 g_2 可以是 0.5。这让模型能够独立调整每个特征通道的重要性。
> - 初始化：正如之前提到的，这个参数通常初始化为 1。这意味着在训练刚开始时，它不做任何事情，让网络先享受归一化带来的稳定性；随着训练进行，梯度下降会自动调整 g_i 的值来优化性能。

```python
class RMSNorm(nn.Module):

    # hidden_size就是d_model，在代码实现中，
    # 程序员习惯用 hidden_size 或 embed_dim 来命名这个变量
    def __init__(
        self,
        hidden_size: int
```

```
        eps: float = 1e-5,
        device=None,
    ):
        super().__init__()
        self.weight = nn.Parameter(torch.ones(hidden_size, device=device))
        self.eps = eps

    def forward(self, x):

        in_dtype = x.dtype

        x = x.to(torch.float32)
        # dim=-1: 表示在最后一个维度（即特征维度 d_model）上求均值。
        rms = torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
        x = x * rms

        # 将转换为 float32 的数值转回 float16
        return (self.weight * x).to(in_dtype)

    def extra_repr(self):
        return f"hidden_size={self.weight.shape[0]}, eps={self.eps}"
```

问题：为什么在实现 RMSNorm 时我们需要做精度变换？

float32 (单精度 Single Precision)：

- 占用 32 bit (4字节) 内存。
- **数值范围**：非常大，最大约 3.4×10^{38}。
- **精度**：大约 7 位有效十进制数字。
- *这是深度学习中默认的“标准”精度。*

float16 (半精度 Half Precision)：

- 占用 16 bit (2字节) 内存。
- **优势**：**省一半显存，算得快一倍**。这是现在训练大模型的主流选择。
- **致命弱点**：**数值范围很小**。它能表示的最大正数只有 **65504**。超过这个数，电脑就会把它变成 `inf` **(无穷大)**。

RMSNorm 的计算过程：$\sum x_i^2$。若输入数据是 `float16` 类型。某个神经元的激活值可能是 300，计算 $300^2 = 90,000$。

- **如果用 float32**：90,000 远小于 10^{38}，安全。
- **如果用 float16**：90,000 > 65,504 (float16 的上限)。电脑直接判定为 `inf`。此时RMSNorm(a) 的值会直接为0，模型训练直接崩溃。