1 Basic Building Blocks: Linear and Embedding Modules

1.1 Parameter Initialization

在深度学习中，我们通过梯度下降来优化参数。初始值的选择决定了优化的起点。如果起点选得不好（例如方差过大或过小），会导致梯度在反向传播时变得极小（梯度消失）或极大（梯度爆炸），从而使模型无法收敛 。

虽然采用了 Pre-norm 架构的 Transformer 对初始化相对不那么敏感（Robust），但好的初始化仍然能显著提升训练速度和收敛效果 。

采用如下初始值：

- Linear weights：$N(\mu = 0, \sigma^2 = \frac{2}{d_{in}+d_{out}})$ truncated at $[-3\sigma, 3\sigma]$
- Embedding：$N(\mu = 0, \sigma^2 = 1)$ truncated at $[-3, 3]$
- RMSNorm：1

1.1.1 为什么要做初始化以及截断？

参数初始化（Initialization） 的核心目的，就是让数据信号和梯度信号在经过几十上百层网络传递时，方差始终保持稳定。

在深层网络中，每一层的输出 y 大致是输入 x 和权重 W 的乘积：$y = Wx$。

- **如果权重 W 太小**（比如都接近 0）：
  - **前向传播**：信号经过层层相乘，数值会迅速衰减趋近于 0。
  - **反向传播**：梯度（Gradients）在链式法则中也会连乘这些微小的权重，导致传递到第一层时梯度几乎为 0，网络无法学习。这就是梯度消失。
- **如果权重 W 太大**：
  - **前向传播**：数值会层层放大，变成巨大的数字（NaN 或 Infinity）。
  - **反向传播**：梯度也会变成天文数字，导致更新步长过大，模型参数乱飞，无法收敛。这就是梯度爆炸。

1.1.2 前向传播：方差的指数级变化

在前向传播中，我们关注的是**激活值（Activations）的分布**。如果每一层的输出方差不稳定，随着层数 L 的增加，数值分布会迅速坍缩或发散。

假设我们有一个没有非线性激活函数的简单线性层：$y_i = \sum_{j=1}^{n} w_{ij} x_j + b_i$

其中：

- n 是输入维度（即文档中的 d_{in}）。
- x 是输入向量，假设其元素为独立同分布（i.i.d），且 $E[x] = 0, \text{Var}(x)$ 为输入方差。
- w 是权重矩阵，假设其元素为 i.i.d，且 $E[w] = 0, \text{Var}(w)$ 为权重方差。
- x 和 w 相互独立。
- 忽略偏置 b（设为 0），因为它不影响方差放大的倍数性质。

我们计算输出 y_i 的方差：

$$\begin{aligned}\mathrm{Var}(y_i) &= \mathrm{Var}\left(\sum_{j=1}^{n} w_{ij}x_j\right) \\ &= \sum_{j=1}^{n}\mathrm{Var}(w_{ij}x_j) \quad (\text{因为 } x_j, w_{ij} \text{ 独立})\end{aligned}$$

利用方差性质：若 A, B 独立且均值为 0，则 $\mathrm{Var}(AB) = \mathrm{Var}(A)\mathrm{Var}(B)$

(注：更严谨的公式是 $\mathrm{Var}(AB) = \mathrm{Var}(A)\mathrm{Var}(B) + E[A]^2\mathrm{Var}(B) + E[B]^2\mathrm{Var}(A)$，因为均值 $E[A] = E[B] = 0$，后两项消失)。

$$\begin{aligned}\mathrm{Var}(y_i) &= \sum_{j=1}^{n}\mathrm{Var}(w_{ij})\mathrm{Var}(x_j) \\ &= n \cdot \mathrm{Var}(w) \cdot \mathrm{Var}(x)\end{aligned}$$

输出方差与输入方差的关系是：$\mathrm{Var}(y) = (n \cdot \mathrm{Var}(w)) \cdot \mathrm{Var}(x)$

这里 $n \cdot \mathrm{Var}(w)$ 是一个放大系数。假设网络有 L 层，那么第 L 层的方差将是：

$$\mathrm{Var}(y^{(L)}) = \underbrace{(n \cdot \mathrm{Var}(w))^{L}}_{\text{几何级数}} \cdot \mathrm{Var}(x)$$

- **信号衰减（Vanishing）**：如果 $n \cdot \mathrm{Var}(w) < 1$，随着 $L \to \infty$，方差趋近于 0。信号几乎全变成 0，携带的信息丢失。
- **信号放大（Exploding）**：如果 $n \cdot \mathrm{Var}(w) > 1$，随着 $L \to \infty$，方差趋近于 ∞。计算机浮点数溢出（NaN）。

为了保持信号稳定，我们希望这个系数等于 1：$n \cdot \mathrm{Var}(w) = 1 \implies \mathrm{Var}(w) = \frac{1}{n} = \frac{1}{d_{in}}$

1.1.3 反向传播:

在反向传播中，我们关注的是损失函数 J 对输入 x（或权重）的梯度。这里应用的是多元微积分的链式法则。

考虑梯度的反向传递过程。对于第 l 层，梯度 $g^{(l)} = \frac{\partial J}{\partial x^{(l)}}$。

根据链式法则：

$$\frac{\partial J}{\partial x^{(l)}} = \frac{\partial J}{\partial x^{(l+1)}} \cdot \frac{\partial x^{(l+1)}}{\partial x^{(l)}}$$

其中 $\frac{\partial x^{(l+1)}}{\partial x^{(l)}}$ 是雅可比矩阵（Jacobian Matrix），对于线性层 $x^{(l+1)} = Wx^{(l)}$，其导数就是权重矩阵 W^T。

因此，反向传播的梯度模长变化取决于 W^T。

与前向传播类似，我们看梯度流过一层时的方差变化。

假设 $g^{(l+1)}$ 是上游传来的梯度，且独立同分布。$g_j^{(l)} = \sum_{i=1}^{m} w_{ij} g_i^{(l+1)}$

这里求和的上限 m 变成了输出维度（即文档中的 d_{out}，因为反向传播时，W 转置了，原来的行变成了列）。

重复上述方差推导：$\mathrm{Var}(g^{(l)}) = m \cdot \mathrm{Var}(w) \cdot \mathrm{Var}(g^{(l+1)})$

即：$\mathrm{Var}(g^{(l)}) = (d_{out} \cdot \mathrm{Var}(w)) \cdot \mathrm{Var}(g^{(l+1)})$

同样地，为了防止梯度在反向传播几十层后消失或爆炸，我们需要：

$$d_{out} \cdot \mathrm{Var}(w) = 1 \implies \mathrm{Var}(w) = \frac{1}{d_{out}}$$

1.1.4 为何是 $2/(d_{in}+d_{out})$？

现在出现了一个矛盾：

1. 为了前向传播稳定，我们需要 $\mathrm{Var}(w)=\frac{1}{d_{in}}$。
2. 为了反向传播稳定，我们需要 $\mathrm{Var}(w)=\frac{1}{d_{out}}$。

除非 $d_{in}=d_{out}$，否则无法同时满足。于是，Xavier Glorot 提出了一个折中方案，取两者的调和平均：
$\mathrm{Var}(w)=\frac{2}{d_{in}+d_{out}}$

这就是文档 3.4.1 中给出的线性层初始化公式的来源：

> Linear weights: $\mathcal{N}(\mu=0,\sigma^2=\frac{2}{d_{in}+d_{out}})$

文档使用这个特定的方差，就是为了在统计学意义上，让 Transformer 的深层网络中的信号方差在前向和反向传播中都尽可能保持在 1 附近，从而避免数值问题。

1.1.5 为什么要截断（Truncation）？

在统计学中，正态分布的定义域是 $(-\infty,+\infty)$。虽然 σ^2 控制了总体方差，但随机采样总有极小概率采到像 10σ 这种极端值。

在深度神经网络中，非线性激活函数（如 Sigmoid, Tanh, 甚至 ReLu/SiLU）对输入非常敏感。

- 如果 w 初始化出现极端大值，对于 $y=w\cdot x$，初始的神经元激活值可能直接落入激活函数的**饱和区**（导数接近 0 的区域）。
- 一旦落入饱和区，梯度 ≈ 0，这个神经元就寄了，无法学习。

因此，截断操作本质上是一个**拒绝采样（Rejection Sampling）**，它以牺牲一点点理论方差的精确度（截断后的实际方差会比理论参数略小，需要修正系数）为代价，换取了**最坏情况下的安全性**。

1.2 Linear Module

Linear layers 是 transformer blocks 中的重要组成部分，但其实现比较简单，具体代码如下：

```
class Linear(nn.Module):
    def __init__(self, d_in: int, d_out: int):
        super().__init__()
        std = math.sqrt(2 / (d_in + d_out))
        self.weight: Float[Tensor, " d_out d_in"] = nn.Parameter(
            nn.init.trunc_normal_(torch.empty(d_out, d_in), std=std, a=-3*std, b=3*std),
            requires_grad=True
        )

    def forward(self, x: Float[Tensor, " ... d_in"]) -> Float[Tensor, " ... d_out"]:
        return einsum(x, self.weight, "... d_in, d_out d_in -> ... d_out")

    def extra_repr(self):
        return f"d_out={self.weight.shape[0]}, d_in={self.weight.shape[1]}"
```

这里我们注意一下这个地方：`torch.empty(vocab_size, d_model)` 的意思是：**创建一个指定形状的张量（Tensor），但不初始化其数值（即不进行清零或填充随机数）。**

你可能会问，为什么不直接用 `torch.randn`（标准正态分布）或者 `torch.zeros`（全0）？

- **速度极快（主要原因）**：`torch.empty` 只是分配内存地址，而不进行写入操作（skip memory initialization）。当你申请非常大的显存（例如大模型的参数矩阵）时，这一步能省去一点点时间。
- **后续必有覆盖操作**：通常写这行代码的人，紧接着会立刻用专门的初始化方法（如 Xavier 初始化或 Kaiming 初始化）来覆盖这些值。如果你先用 `zeros` 填了一遍 0，再用 `Xavier` 填一遍随机数，那就等于多做了一次无用功。

`torch.nn.init.trunc_normal_(tensor, mean=0.0, std=1.0, a=-2.0, b=2.0)`，这是`.trunc_normal_`的默认初始值

注意到最后一段还写了个 `extra_repr` 函数。这也是 PyTorch 中一个非常实用但在教程中常被忽略的小功能。

这段代码的作用是：定义当你在 Python 中使用 `print(model)` 打印这个层（Layer）时，括号里显示出来的额外信息。

它是 `nn.Module` 类提供的一个钩子（Hook）方法，专门用于增强模型的可读性和调试体验。

直观的对比：有 vs 没有

假设你定义了一个自定义的层叫 `MyEmbedding`，并且实例化了一个对象 `layer`。

情况 A：如果你没有写 `extra_repr` 当你运行 `print(layer)` 时，PyTorch 默认只会输出类名：

```
MyEmbedding()
```

问题：你看不出这个层有多大，参数是多少，调试时很不方便。

情况 B：如果你写了这段 `extra_repr` 当你运行 `print(layer)` 时，PyTorch 会自动调用这个方法，把返回的字符串填进括号里：

```
MyEmbedding(vocab_size=30000, d=512)
```

优势：一目了然。你知道这个层的词表大小是 30,000，维度是 512。

1.3 Embedding Module

```
class Embedding(nn.Module):
    def __init__(self, vocab_size: int, d_model: int):
        super().__init__()
        std = 1.0
        self.weight = nn.Parameter(
            nn.init.trunc_normal_(torch.empty(vocab_size, d_model), std=std, a=-3 * std, b=3 * std),
            requires_grad=True
        )

    def forward(self, token_ids: Int[Tensor, " ..."]) -> Float[Tensor, " ... d_model"]:
        return self.weight[token_ids, :]

    def extra_repr(self):
```

```
14          return f"vocab_size={self.weight.shape[0]}, d={self.weight.shape[1]}"
```