Transformer 深度探索：项目经验总结与笔记

项目目标： 从零开始，基于 PyTorch 搭建一个用于文言文-现代文翻译的 Transformer 模型。
项目范围： 涵盖数据处理（data_preparation.py）、模型搭建（model.py, model_components.py）、训练（train.py）与高级诊断（debug_plus.py）。
核心历程： 本次项目与其说是一次“模型实现”，不如说是演变成围绕 Pre-Norm 架构数值稳定性的深度调试过程。

1. “Attention is All You Need” —— QKV 理论与注意力机制

在深入架构之前，必须理解 Transformer 的基石：**注意力 (Attention)**，特别是其 **QKV (Query, Key, Value)** 范式。

1.1. QKV 的理论意义：一个数据库查询的类比

理解 QKV 最直观的方式是将其类比为一次信息检索（或数据库查询）过程：

- **Query (Q) - 查询：** 代表当前 Token **“想要寻找什么信息”**。它是一个“查询请求”，做个比喻就相当于你假如在古代要写一篇基于《春秋》讨论当下时政的文章，你要匹配现在的时政和《春秋》里哪些内容相似，这个 Q 就是现在的时政。
- **Key (K) - 键 / 索引：** 代表序列中所有 Token **“能够提供什么信息”**。它像是数据库中每条记录的“标签”或“索引”，用于被 Q 查询和匹配，对应上面也就是《春秋》这本书的原文，但你光看《春秋》你基本看不懂这些字都什么意思，你只是初步找到一个会和你的时政相关的记载。
- **Value (V) - 值 / 内容：** 代表序列中所有 Token **“实际携带什么信息”**。它像是数据库中每条记录的“实际内容”，这就相当于《左传》、《公羊传》等这种实际讲具体发生了什么事情，事情背景是什么的记载。

注意力的工作流程（单头）：

1. **匹配（打分）：** 用你的“查询” (Q) 去和数据库中所有的“索引” (K) 进行匹配（通过矩阵点积 QK^T），得到一个“相关性分数”矩阵。
2. **归一化（缩放与 Softmax）：**
   - 将分数除以 $\sqrt{d_k}$ （后文详述为何如此），防止梯度消失。
   - 将分数通过 softmax 函数，将其转换为一组总和为 1 的“注意力权重”。这步的意义是：**“根据我的查询(Q)，我应该将百分之多少的注意力放在第 i 个 Token 上？”**
3. **加权求和（提取）：** 用这组“注意力权重”去对所有的“内容” (V) 进行加权求和。

其核心公式为：

$$\text{Attention}(Q,K,V)=\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

这个机制的本质是：**根据“查询” (Q) 和“索引” (K) 的相关性，对“内容” (V) 进行一次动态的、可微分的加权求和。** 它使得模型不再依赖固定的 RNN 隐状态，而是能动态地从序列的任意位置提取所需信息。

1.2. 多头注意力 (Multi-Head Attention)

单一的注意力（Attention(Q, K, V)）好比只有一种查询方式。模型可能希望在不同方面（例如“语法关系”、“语义关系”等）同时进行查询。

多头注意力（MultiHeadAttention） 就是为了解决这个问题：

1. **拆分 (Split)：** 它将原始的 d_{model} 维度的 Q, K, V 矩阵，通过不同的线性映射（W_q^i, W_k^i, W_v^i），拆分为 h （num_heads）个独立的、更低维度的 Q, K, V “头”（维度为 $d_k = d_{model} / h$ ）。
2. **并行计算 (Parallel Attention)：** h 个“头”**并行**地执行上述的 $\text{Attention}(Q^i, K^i, V^i)$ 计算。
3. **合并 (Concatenate)：** 将 h 个“头”各自计算得到的输出向量拼接（Concatenate）回 d_{model} 维度。
4. **投影 (Project)：** 最后通过一个线性层 W_o 将合并后的结果投影，使其“融合”不同头的信息。

这使得模型有能力在 h 个不同的“表示子空间”中同时捕获信息，极大地增强了模型的表达能力。

2. 关键组件：位置编码 (Positional Encoding)

2.1. 为什么需要位置编码？

自注意力机制（Self-Attention）有一个核心特性：**置换不变性 (Permutation Invariance)**。

这意味着，对于注意力机制来说，["我", "爱", "你"] 和 ["你", "爱", "我"] 这两个输入序列，在计算加权和后是完全等价的。注意力只关心“袋子”里有哪些词，而不关心它们的**顺序**。

这对于自然语言处理显然是致命的，因为顺序（"我爱北京天安门" vs "天安门爱我北京"）承载了绝大部分的语法和语义信息。

因此，我们必须想办法将“位置信息”注入到模型中。 位置编码（Positional Encoding, PE）就是解决方案。它不是通过 RNN 的循环结构“隐式”学习顺序，而是“显式”地将一个代表绝对位置的向量添加到词嵌入中。

$$\text{Input} = \text{Embedding}(x) + \text{PE}(pos)$$

2.2. Sin/Cos 位置编码的几何直觉

原始论文设计了一套基于 sin 和 cos 函数的固定编码。我对这套公式的理解如下，它远比"保证唯一性"要深刻：

1. 目标：构造线性无关的位置向量。
   仅仅保证 PE_{pos} 互不相同是不够的，我们需要构造一组在 d_{model} 维空间中"分布良好"的向量，理想情况下它们应线性无关，这样模型才能清晰地分辨它们。
2. 方法：基于单位圆的旋转。
   这套公式设计的精妙之处在于它如何组织维度。它将 d_{model} 维向量两两配对（$(2i, 2i+1)$）。
   - $PE_{(pos,2i)} = \sin(pos \cdot \omega_i)$
   - $PE_{(pos,2i+1)} = \cos(pos \cdot \omega_i)$

   其中，$\omega_i = 1/10000^{2i/d_{model}}$ 是**角频率**。
   正如你的理解，对于任意一个频率 ω_i，$(PE_{(pos,2i)}, PE_{(pos,2i+1)})$ 这对值构成了 $(\sin(pos \cdot \omega_i), \cos(pos \cdot \omega_i))$。
   这正是一个在二维单位圆上的点，其角度为 $pos \cdot \omega_i$。
   - 当 $pos = 0$ 时，点在 $(0,1)$。
   - 当 $pos = 1$ 时，点旋转 ω_i 弧度，到达 $(\sin(\omega_i), \cos(\omega_i))$。
   - 当 $pos = 2$ 时，点再旋转 ω_i 弧度，到达 $(\sin(2\omega_i), \cos(2\omega_i))$。
3. 核心：频率递减的设计。
   这个设计的另一个关键点，在于频率 ω_i 的递减。
   - 当 $i = 0$ 时（低维度），$\omega_0 = 1/10000^0 = 1$，频率最高，点旋转最快。
   - 当 i 增大时（高维度），$\omega_i = 1/10000^i$，分母剧增，频率 ω_i 迅速降低，点旋转得极慢（波长极长）。

总结： 整个 d_{model} 维的位置向量 PE_{pos}，可以被看作是 $d_{model}/2$ 个**不同旋转频率**的二维向量的组合。通过在 $d_{model}/2$ 个"二维子空间"中，以**指数递减的频率**进行旋转，模型为每个位置 pos 构造了一个独一无二的、且**线性无关**的"指纹"向量。

2.3. 为什么是旋转？（相对位置的线性表示）

这种基于旋转的设计（而不是其他 hash 函数）有一个至关重要的特性：**相对位置可以被线性表示**。

一个位置 pos+k 的编码，可以被表示为位置 pos 编码的**旋转**（即线性变换）。

根据三角恒等式：

$$\sin((pos+k)\omega_i) = \sin(pos \cdot \omega_i)\cos(k \cdot \omega_i) + \cos(pos \cdot \omega_i)\sin(k \cdot \omega_i)$$

$\cos((pos+k)\omega_i) = \cos(pos\cdot\omega_i)\cos(k\cdot\omega_i) - \sin(pos\cdot\omega_i)\sin(k\cdot\omega_i)$

PE_{pos+k} 在 $(2i, 2i+1)$ 子空间的值，可以由 PE_{pos} 的值（$(\sin(pos\cdot\omega_i), \cos(pos\cdot\omega_i))$）和一个只与 k 相关的旋转矩阵相乘得到。

这对模型意味着：

模型不需要去“背诵”绝对位置。注意力机制（本质上是线性投影）只需要学会一个固定的变换，就能理解“我前面/后面第 k 个词”这样的相对位置关系。这使得模型可以轻松泛化到未见过的序列长度。

3. 核心架构的实现：Encoder 与 Decoder

现在我们可以系统性地看 Encoder 和 Decoder 是如何协同工作的。

3.1. Encoder (编码器) —— 理解上下文

- **核心职责：** 将输入的源语言序列（例如 "学而时习之"）编码为一组富含上下文信息的**记忆向量**（enc_output）。
- **实现：** 由 N 层 EncoderLayer 堆叠而成（Encoder）。

EncoderLayer 的工作流程：

1. **输入：** 词嵌入 + **位置编码**（来自上一层或 Embedding 层）。
2. **子层 1: 自注意力 (Self-Attention)**
   - **QKV 来源：** Q, K, V **均来自同一来源**（前一层的输出）。
   - **作用：** 这是 Encoder 理解上下文的核心。它允许序列中的每个 Token“环顾四周”，评估它与序列中**所有**其他 Token 的关系（已包含位置信息）。例如，“之”这个词可以通过自注意力“看到”前面的“学”、“而”、“时”、“习”，从而更新自己的向量表示，使其包含“学习这件事”的含义。
3. **子层 2: 位置前馈网络 (FFN)**
   - **作用：** 在“信息融合”（自注意力）之后，对**每个** Token 的向量**独立地**进行一次非线性变换（Linear -> ReLU -> Linear），可以被理解为“消化和提炼”融合后的信息。

最终输出： 最后一层 EncoderLayer 的输出（经过最终的 encoder.norm）形成了 enc_output（形状 (batch_size, seq_len_src, d_model)）。这个张量是 Encoder 对整个源句子的“完整理解”，它将作为“只读存储器”被 Decoder 的每一层访问。

3.2. Decoder (解码器) —— 序列生成

- **核心职责：** **自回归地**（Autoregressively）生成目标语言序列（例如 "学习" -> "并" -> "时常" ...）。“自回归”意味着 T 时刻的输出，将作为 $T+1$ 时刻的输入。
- **实现：** 由 N 层 DecoderLayer 堆叠而成（Decoder）。

DecoderLayer 的工作流程：

1. **输入：** 目标序列的词嵌入（T-1 时刻的输出）+ **位置编码**。

2. **子层 1: 带掩码的自注意力 (Masked Self-Attention)**
   - **QKV 来源：** Q, K, V 均来自 Decoder 自身（前一层的输出）。
   - **作用：** 与 Encoder 的自注意力类似，但有一个**关键区别**：它使用了**前瞻掩码 (Look-ahead Mask)**。这确保了在预测第 T 个词时，注意力机制**只能**关注 $1 \ldots T$ 位置的词，而不能“偷看” $T+1$ 及未来的词。
3. **子层 2: 交叉注意力 (Cross-Attention)**
   - **作用：** 这是连接 Encoder-Decoder 的核心桥梁。
   - **QKV 来源：**
     - **Query (Q)：** 来自 Decoder（子层 1 的输出）。这代表了 Decoder 在生成第 T 个词时，发出的查询：“我需要什么样的源语言信息来辅助我决策？”
     - **Key (K) 和 Value (V)：** 均来自 Encoder 的最终输出 enc_output。
   - **流程：** 它允许 Decoder 在每一步生成时，都能用自己的 Q 去“查询”一次 Encoder 提供的完整上下文 enc_output。例如，在翻译 "说" 时，交叉注意力允许 Decoder 的 Q（“我需要翻译‘说’这个词”）去匹配 Encoder 的 K（“如‘学而时习之，不亦说乎’的‘说’字”），然后提取对应的 V（“‘说’字在‘不亦说乎’语境下的含义”）。
4. **子层 3: 位置前馈网络 (FFN)**
   - **作用：** 同样地，进行非线性“消化和提炼”。

最终输出： 经过 N 层堆叠后，最后一层 DecoderLayer 的输出被送到一个 final_proj 线性层，投影到目标词汇表大小（Logits），然后通过 softmax 得到下一个词的概率。

3.3. 掩码 (Masking) 的实现

这是 Transformer 工作的关键前提。我实现了两种掩码：

1. **Padding Mask (填充掩码)：**
   - **作用：** 屏蔽掉源序列和目标序列中 <pad> Token，使其在注意力计算中的得分为 $-\infty$，从而在 softmax 后权重为 0。
   - **实现：** (seq == pad_idx)，并调整形状用于广播。
2. **Look-ahead Mask (前瞻掩码)：**
   - **作用：** **仅用于 Decoder 的自注意力**。它确保在预测 T 时刻的词时，模型只能“看到” $1 \ldots T$ 时刻的输入，而不能“偷看” $T+1$ 时刻及未来的词。
   - **实现：** 这是一个“复合掩码”。通过一个上三角矩阵（torch.ones(...).triu(diagonal=1)）来屏蔽未来词，然后使用逻辑或（|）将其与 Padding Mask 合并。这是我学到的一个关键技巧，确保一个 Token 既不会看到未来，也不会看到 Pad。

4. 核心难题：Pre-Norm vs Post-Norm 与 $\sqrt{d_k}$

在调试中，我遇到了第一个核心难题：$\sqrt{d_{model}}$ 缩放 Bug。我最初在 Embedding 层后乘以了 $\sqrt{d_{model}}$，导致 Loss 卡在 1.5。这迫使我去入理解了 Pre-Norm 和 Post-Norm 的区别。

- **Post-Norm (原始论文架构):**
  - x = Norm(x + Dropout(Sublayer(x)))
  - 即：输入 -> 子层(MHA/FFN) -> Add -> Norm
  - **问题：** 激活值和梯度在流经 N 层后才被归一化，中途极易发生爆炸或消失，导致训练不稳定。
- **Pre-Norm (改进后的架构):**
  - x = x + Dropout(Sublayer(Norm(x)))
  - 即：输入 -> Norm -> 子层(MHA/FFN) -> Add
  - **优势：** 激活值在进入**每个**子层前都被“重置”为 $N(0,1)$ 分布，梯度流更稳定，N 这个层数就可以设计的更深。

$\sqrt{d_k}$ 缩放的推导与反思

我曾困惑为何需要 $\sqrt{d_k}$ 缩放。通过思考和推演，我理解了其数学本质：

1. 假设 Q 和 K 中的向量 q, k 均值为 0，方差为 1。
2. 点积 $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$。
3. 根据统计学，独立随机变量之和的方差等于它们方差之和：
   $Var(q \cdot k) = \sum_{i=1}^{d_k} Var(q_i k_i)$
4. 假设 q_i, k_i 独立且 $E[q_i] = E[k_i] = 0$， $E[q_i^2] = Var(q_i) = 1$， $E[k_i^2] = Var(k_i) = 1$。
   $Var(q_i k_i) = E[(q_i k_i)^2] - (E[q_i k_i])^2 = E[q_i^2]E[k_i^2] - (E[q_i]E[k_i])^2 = (1)(1) - (0)^2 = 1$
5. 因此，点积的方差为：$Var(q \cdot k) = \sum_{i=1}^{d_k} 1 = d_k$。

结论： 点积的方差是 d_k，标准差是 $\sqrt{d_k}$。

这意味着 d_k 越大，点积的结果（softmax 的输入）就越大，导致 softmax 函数过早进入其饱和区（梯度接近 0），使训练崩溃。

Post-Norm 必须使用 $\frac{QK^T}{\sqrt{d_k}}$，将方差拉回到 1。

我的 Pre-Norm 反思： 我的 Pre-Norm 架构在每次 MHA 之前都进行了 Norm(x)，这已经强制将 q 和 k 的输入源拉回 $N(0,1)$。因此，这个缩放对于 Pre-Norm 来说不再是生死攸关的（尽管保留它仍是好习惯）。

而我犯的错误（$\sqrt{d_k}$ 缩放 Bug）是在 Embedding 层进行了缩放，这导致 Embedding（幅度 ≈ 22.6）完全“淹没”了 Positional Encoding（幅度 ≈ 1.0），模型变成了“位置色盲”，无法学习序列顺序，Loss 自然卡住。

新调试

我之前犯了个大错，觉得 Pre-Norm 既然都要归一化了，那 Embedding 层后面那个乘以 $\sqrt{d_{model}}$ 就是脱裤子放屁——多此一举。
事实证明我错了，而且错得很离谱。
这里的核心矛盾在于**信号强度的博弈**：

- **位置编码 (Positional Encoding)：** 它是 sin/ cos 函数，值域雷打不动地卡在 $[-1,1]$ 之间，信号非常强势 。
- **Embedding：** 初始化的时候是很小的随机数（通常均值为 0，方差很小）。

如果不乘以 $\sqrt{d_{model}}$ 把 Embedding 的数值“放大”，**语义信息直接就被位置信息给淹了**。模型一上来根本看不清这个词是“学”还是“习”，只知道它在第几个位置。这直接导致模型变成了“位置色盲”，Loss 自然卡在 1.5 下不来 。
最终的修正操作：
我老老实实把 x = x * math.sqrt(d_model) 加了回去。这本质上不是为了防梯度消失，而是给语义信息“注入能量”，让它有资格和位置信息在同一个量级上“说话”。

5. 归一化（LayerNorm）的理解

通过这个项目，我对 LayerNorm 有了两种直觉上的认知：

$$y_i = \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

1. **统计直觉：** 在高维空间中，根据中心极限定理，向量各维趋近正态分布。LayerNorm 的作用就是将这个 $N(\mu,\sigma^2)$ 分布强行标准化为 $N(0,1)$，然后再通过可学习的 γ 和 β 参数（weight 和 bias）将其缩放和平移到一个模型认为“舒适”的 $N(\beta,\gamma^2)$ 分布上。
2. **线性空间直觉：** 归一化（减均值）本质上是将高维向量 x 投影到一个与全 1 向量 $\vec{1}$ 正交的子空间中，然后再（除标准差）将其范数（Norm）单位化。

实践中的噩梦：LayerNorm 初始化与“僵尸化”

理论归理论，实践中的 LayerNorm 初始化是我本次调试的重灾区。

- **阶段 1：标准初始化 ($\gamma = 1$)**
  - **配置：** $\gamma = 1(weight), \beta = 0(bias)$ 。
  - **现象：** 训练 Loss 能下降，但 T=1（第一个 Token）推理时，激活值范数爆炸，Logit 高达 19.16，模型坚定预测 <eos>。
  - **分析：** 6 层 Pre-Norm 堆叠对于“冷启动”（T=1）依然不稳定。
- **阶段 2：“僵尸化” ($\gamma = 0$)**
  - **配置：** 为了解决爆炸，我尝试将所有 γ 设为 0。
  - **现象：** T=1 爆炸消失。但模型完全无法训练，Loss 稳定卡在 1.5。
  - 分析： 我亲手制造了“僵尸模型”。当 $\gamma = 0$ 时，LayerNorm 的输出恒为 β（即 0）。

Pre-Norm 的残差连接变为：

$$x_{out} = x_{in} + \text{Dropout}(\text{Sublayer}(\text{Norm}(x_{in})))$$

$$x_{out} = x_{in} + \text{Dropout}(\text{Sublayer}(0))$$

$$x_{out} = x_{in} + 0$$

信号只在残差连接上“裸奔”，MHA 和 FFN 模块被完全旁路，模型学不到任何东西。

- **阶段 3：最终的初始化策略 (T-Fixup 思想)**
  - **配置：** 这是我在 debug_plus.py 中采用的最终策略：
    1. 所有**内部** LayerNorm（layer.norm_1, layer.norm_2 ...）的 $\gamma = 0$。
    2. 所有**最终** LayerNorm（encoder.norm, decoder.norm）的 $\gamma = 1$。
  - **分析：** 这是一个精妙的妥协。它使得在训练初期，内部 MHA/FFN 模块接近旁路避免爆炸，信号可以稳定地流过 N 层残差“高速公路”；同时，最终的 Norm 层（$\gamma = 1$）又能让模型输出有意义的激活值。随着训练进行，模型会慢慢学会将内部的 γ 从 0 调大。

新调试

初始化在我采用新的方法后回归最标准的 xavieruniform 初始化即可，无需细致调整初始参数了。

6. 关于注意力公式的底层重构

最终的成功测试核心依然在于那个注意力的核心公式：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

最终非常抽象的，问题被锁定到了关于这个公式的实现上，之前我们一直在使用 F.scaled_dot_product_attention 这个黑箱函数，只能说最终证明这个函数不好用。最终我们使用了大段的直接处理，代码如下：

```
# 3.1 计算注意力分数 (Scaled Dot-Product)
# Q: (B, H, L, D_k), K^T: (B, H, D_k, S) -> Scores: (B, H, L, S)
attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

# 3.2 应用掩码 (手动处理，绝对透明)
if mask is not None:
    # 你的 mask 可能是 (B, 1, 1, S) 或 (B, 1, L, S)
    # PyTorch 的广播机制会自动将 mask 广播到 (B, H, L, S)

    # 鲁棒性检查：处理 Bool 类型 (True=屏蔽) 和 Float 类型 (负无穷=屏蔽)
    if mask.dtype == torch.bool:
        # 如果是 Bool 掩码 (True=Masked)，用 -1e9 填充 (模拟 -inf)
        attn_scores = attn_scores.masked_fill(mask, -1e4)
    else:
```

```
        # 如果是 0/-inf 掩码，直接相加
        attn_scores = attn_scores + mask

    # 3.3 Softmax 和 Dropout
    # 在最后一个维度 (Seq_len_k) 上进行归一化
    attn_probs = F.softmax(attn_scores, dim=-1)

    # 应用 Dropout (仅在训练时)
    attn_probs = self.dropout(attn_probs)

    # 3.4 加权求和
    # Probs: (B, H, L, S), V: (B, H, S, D_k) -> Output: (B, H, L, D_k)
    x = torch.matmul(attn_probs, V)
```

反正在完成了这个自己底层的关于注意力公式的构建后，测试通过了，我也不明白 F.scaled_dot_product_attention 中间到底有什么黑洞导致了最终坚定预测<eos>，只能说现在手动实现注意力公式后问题解决了。

7. 我学习到的诊断技巧

这次调试迫使我建立了一套诊断工具（集中体现在 debug_plus.py），这是比模型本身更有价值的收获。

1. train() vs eval() 对比分析
   - **现象：** 这是我遇到的一个悖论。模型在 DROPOUT=0.0 时，train_one_epoch （教师强制）显示 Loss 收敛到 0.0007；但在 test_autoregressive_steps（自回归）中，model.eval() 模式下几乎 100% **失败**，model.train() 模式下却**成功**。
   - **分析：** DROPOUT=0.0 意味着 Dropout 层在两种模式下行为一致。LayerNorm 的统计量（均值/方差）在 Pre-Norm 中是即时计算的，与 eval() 模式无关。因此，train() 和 eval() 模式在逻辑上应该是**等价的**。
   - **我的假说：** 唯一的区别是 test_autoregressive_steps 在 with torch.no_grad(): 块中运行。这强烈暗示存在一个极其隐蔽的底层 Bug，可能在 torch.no_grad()、torch.amp.autocast 和 F.scaled_dot_product_attention 之间存在某种交互，导致在 T=1 时的数值计算在有/无梯度上下文时产生了差异。
2. 教师强制 (Teacher Forcing) vs 自回归 (Autoregressive)
   - 我学会了必须同时测试这两种模式。
   - **教师强制（训练时）：** 无论上一步预测对错，下一步总是喂“标准答案”。这能测试模型“背诵”数据的能力。
   - **自回归（推理时）：** T 时刻的输入是 T-1 时刻的**预测输出**。这是对模型“泛化”和“稳定性”的真正考验。我的 Bug 表明，模型“背会了”，但“不会考”。
3. T=1 Loss 独立分析

- 我在 debug_plus.py 中增加了诊断，将 T=1（第一个 token）的 Loss 与 T>1 的 Loss 分开打印。这对于定位“冷启动”数值爆炸问题（如阶段 1）至关重要。

4. 配置隔离（过拟合测试）
   - 我使用 debug_plus.py 在 20 个样本上进行过拟合测试。这是一个黄金法则：如果你的模型连 20 个样本都“背”不会（Loss 无法降到 0），那它绝对学不会 200 万个样本。
   - 在这个测试中，我关闭了所有“干扰项”：
     - DROPOUT = 0.0
     - label_smoothing = 0.0
     - 关闭 lr_scheduler
   - 这使我能确定问题出在模型架构的数值稳定性上，而不是这些训练技巧上。
5. 范数评估（Norm）
   - 范数（Norm）是一种数学函数，它将一个高维向量（比如 d_{model} 维的 x）“压缩”成一个非负的标量（单一数值），这个数值衡量该向量的“长度”或“幅度”。
   - L2 范数（最常用）： 它的工作方式是计算向量中所有元素平方和的平方根，即 $\|x\|_2=\sqrt{\sum_{i=1}^{d} x_i^2}$ 。这在几何上等于向量空间中从原点到该点的欧几里得距离。
   - LayerNorm 中的应用： 在我的项目中，LayerNorm 就利用了这个思想。它首先通过减均值和除以标准差 σ 来（在统计意义上）将向量的“范数”或“幅度”归一化（单位化）。
   - 缩放与平移： 归一化后，LayerNorm 再通过可学习的参数 γ（缩放）和 β（平移），将这个单位化的向量调整到一个模型认为最“舒适”的新幅度（分布）上 。
   - 通过检查各种层里的范数如 torch.norm(x[0, token_index, :]).item()，可以有效估计各神经层之间是坏死还是爆炸。

6. 权重共享(Weight Tying)
   把解码器最后的输出层（Linear）和编码器的 Embedding 层强行绑定。 操作很简单：model.decoder.final_proj.weight = model.encoder.embedding.weight。
   关于这个问题我理解到以下几点：
   - 输入（Embedding）和输出（Logits）其实用的是同一本字典，描述的是同一个语义空间。如果让模型分开学，相当于**进门的时候发了一张地图，出门的时候又让它自己瞎画一张**。在只有 20 条数据的过拟合测试里，这简直是难为它了，而且就算是对于更重的任务，也是要保证字典依然是同一个，不要增加考试里还把字典的顺序打乱要求它也重新学，这没什么意义。
   - 把这两层权重绑死，等于是在拓扑结构上把系统的**入口和出口强行连在了一起**。这不仅把参数量砍了一大截，更重要的是给模型施加了一个极强的几何约束——**“怎么映射进高维空间的，就得怎么映射回来”**。

8. 总结与反思

这次项目历时数天，最终解决了那个莫名其妙的 bug，也学到点 Transformer 架构的皮毛，学会了

一些诊断模型内部健康状态的一些理论与方法，也不知道是运气好还是运气差，最终居然是那个奇奇怪怪大家各种大模型 Gemini、Claude、GPT、Deepseek 等花了不少尝试都无法解决逻辑上认为都没毛病的 F.scaled_dot_product_attention，自己手动实现反而可以解决，但我也懒得看这个函数内部到底发生了什么交互导致问题的，只能说最后把那部分关键手动写出来后却无论是半精度还是全精度是都能通过的，这中间内部是什么玄学我就不太知道了。

补充：其实最后这个原因找到了，真的令人忍俊不禁，单纯就是 sdpe 的掩码逻辑和传统的 mha 的调用的掩码逻辑完全相反，就是单纯的业内行业规范的烂账而已，我无力吐槽了。