自然语言处理研讨课（实践课）

第 8 章 神经机器翻译实践

潘宇轩

2025 年 12 月 9 日

实验成果总结

模型类型	验证集 BLEU	验证集 Loss	备注
mBART-50 (预训练)	**0.51**	2.90	无需微调
Transformer (从零训练)	0.03	7.93	10 epochs

目 录

1 引言：序列建模的挑战

机器翻译的本质是序列到序列（Sequence-to-Sequence, Seq2Seq）的映射问题。给定一个源语言序列 $X = (x_1, x_2, \ldots, x_n)$，我们希望模型能够输出对应的目标语言序列 $Y = (y_1, y_2, \ldots, y_m)$。这个问题的难点在于：输入和输出的长度通常不相等，而且两种语言的语法结构、词序可能完全不同。比如英语是 SVO（主谓宾）结构，而日语是 SOV（主宾谓）结构，翻译时需要对词序进行重排。

早期的统计机器翻译（SMT）依赖于大量的人工特征工程和对齐模型，虽然在一定程度上解决了问题，但系统复杂、难以维护。2014 年前后，基于神经网络的端到端翻译模型开始崭露头角，其中最具代表性的就是基于循环神经网络（RNN）的 Encoder-Decoder 架构。然而，随着数据量的爆发和模型深度的增加，RNN 逐渐显露出其时代的局限性。

2 前世：循环神经网络 (RNN) 的局限

2.1 RNN 的基本原理

RNN 的核心设计理念是” 记忆”。它通过一个隐状态（Hidden State）h_t 来承载过去的信息，并按时间步逐步更新。其数学形式通常为：

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t + b_h) \tag{1}$$

其中 W_{hh} 是隐状态到隐状态的权重矩阵，W_{xh} 是输入到隐状态的权重矩阵。这个公式的含义是：当前时刻的隐状态由上一时刻的隐状态和当前输入共同决定。

在 Encoder-Decoder 架构中，编码器 RNN 将整个源句子压缩成一个固定长度的向量（通常是最后一个隐状态 h_n），然后解码器 RNN 以这个向量为初始状态，逐词生成目标句子。这种设计简洁优雅，但也埋下了隐患。

2.2 时间步的枷锁

这种设计虽然符合人类阅读文本的直觉（从左到右），但在工程和数学上带来了两个致命问题。

首先是**无法并行计算**。由于 h_t 的计算依赖于 h_{t-1}，这意味着无论 GPU 有多少核心，处理一个长度为 100 的句子必须串行执行 100 次运算。在深度学习时代，数据量动辄上亿，这种串行瓶颈严重限制了训练效率。相比之下，卷积神经网络（CNN）可以在一次矩阵运算中处理整个序列，速度快了不止一个数量级。

其次是**信息遗忘**（长距离依赖问题）。想象一下，如果源句子是”The cat, which was sitting on the mat and looking at the bird outside the window, suddenly jumped up.”，

当模型处理到”jumped” 时，它需要记住主语是”cat” 而不是”bird” 或”window”。但经过这么多时间步的传递，关于”cat” 的信息可能已经被稀释甚至遗忘了。

LSTM（Long Short-Term Memory）通过引入门控机制（遗忘门、输入门、输出门）来缓解这个问题，但并没有从根本上解决。实验表明，即使是 LSTM，在处理超过 50 个词的句子时，性能也会明显下降。

2.3 梯度消失与梯度爆炸

从数学角度看，RNN 的梯度需要通过时间反向传播（Backpropagation Through Time, BPTT）。假设我们要计算 $\frac{\partial L}{\partial h_1}$，需要将梯度从 h_T 一路传回 h_1：

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_T} \prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}} \tag{2}$$

如果 $\frac{\partial h_t}{\partial h_{t-1}}$ 的范数小于 1，连乘后梯度会指数级衰减（梯度消失）；如果大于 1，则会指数级增长（梯度爆炸）。这使得 RNN 很难学习到长距离的依赖关系。

2.4 RNN 结构可视化

为了直观理解这一过程，我们绘制了 RNN 的时间展开图（图 1）。

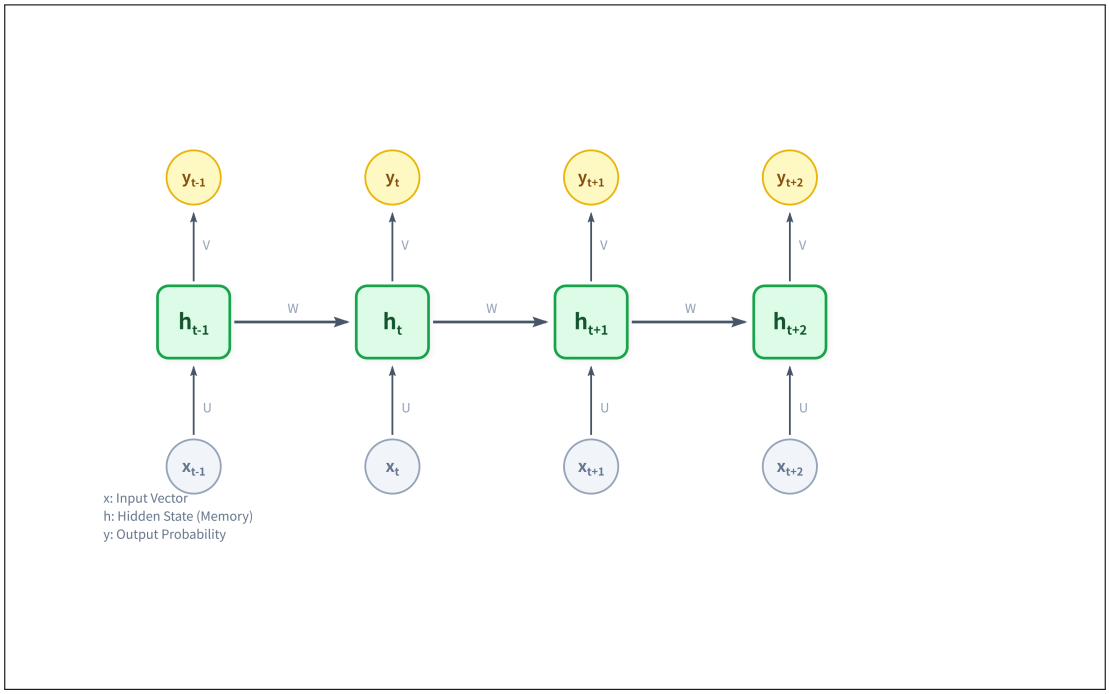

图 1: **RNN 的时间展开结构。**左侧为折叠形式，右侧为按时间步展开。可以看到数据必须沿着箭头 $h_{t-1} \to h_t \to h_{t+1}$ 逐步流动，无法并行。

如图 1 所示，信息的流动被严格限制在时间轴上，这是一条单行道。

3 今生：Transformer 的革命

2017 年，Google 团队在论文《Attention Is All You Need》中提出了 Transformer 架构。这篇论文的标题本身就是一个宣言：我们不需要循环，不需要卷积，**注意力机制就是一切**。

3.1 注意力机制的前身

其实注意力机制并不是 Transformer 的发明。早在 2014 年，Bahdanau 等人就提出了在 RNN Encoder-Decoder 中加入注意力机制的想法。当时的动机很简单：既然把整个句子压缩成一个固定长度的向量会丢失信息，那为什么不让解码器在生成每个词时，都能" 回头看" 编码器的所有隐状态呢？

这个想法效果显著，但仍然受限于 RNN 的串行计算。Transformer 的贡献在于：既然注意力机制这么好用，那我们干脆把 RNN 全部换成注意力机制，实现真正的并行计算。

3.2 核心理念：全局视野与并行计算

Transformer 的核心思想可以用一句话概括：与其按部就班地传递信息，不如让序列中的每一个词都能直接" 看" 到其他所有词。

这种机制带来了两大突破。第一是**并行计算**：由于每个位置的计算不依赖于其他位置的结果，所有词的特征提取可以同时进行。在 GPU 上，这意味着一个长度为 100 的句子可以在一次矩阵运算中完成，而不是串行执行 100 次。第二是**恒定路径长度**：在 RNN 中，位置 1 的信息要传递到位置 100，需要经过 99 个时间步；而在 Transformer 中，任意两个位置之间的交互距离都是 1，这彻底解决了长距离依赖问题。

3.3 自注意力机制 (Self-Attention) 详解

自注意力机制是 Transformer 的灵魂。为了解释它，我们可以使用" 搜索引擎" 的类比。

想象你在使用搜索引擎：你输入一个查询（Query），搜索引擎会将你的查询与数据库中所有文档的关键词（Key）进行匹配，然后返回最相关的内容（Value）。自注意力机制的工作方式与此类似：对于输入序列中的每一个词，我们将其映射为三个向量——**Query** (Q) 表示" 我想找什么信息"，**Key** (K) 表示" 我包含了什么特征" 用于被 Q 匹配，**Value** (V) 表示" 我的具体内容是什么"。

具体来说，这三个向量通过三个不同的线性变换得到：

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{3}$$

其中 X 是输入的词嵌入矩阵，W^Q, W^K, W^V 是可学习的参数矩阵。

注意力权重的计算过程分为三步：首先，计算 Q 和 K 的点积得到相似度分数；然后，除以 $\sqrt{d_k}$ 进行缩放（防止点积值过大导致 Softmax 梯度消失）；最后，通过 Softmax 归一化得到注意力权重，并将其作用在 V 上进行加权求和。

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4}$$

这个公式看起来简单，但蕴含着深刻的思想：每个词的输出表示是所有词的 Value 的加权和，权重由该词与其他词的相关性决定。这使得模型能够动态地关注句子中最相关的部分。

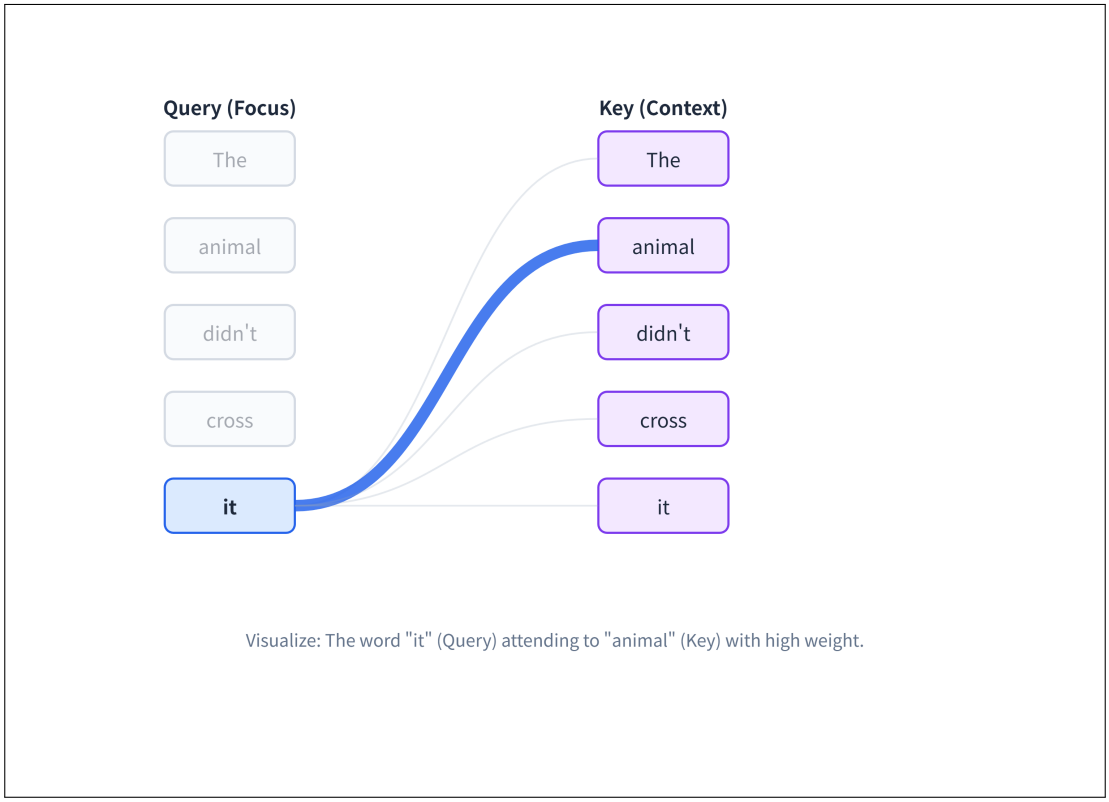

图 2: **自注意力机制可视化。**单词 ”it” (Query) 根据语义相关性，分配了不同的权重给 ”The” 和 ”animal” (Keys)。线条越粗代表关注度越高，从而解决了指代消解等复杂语义问题。

图 2 清晰地展示了这一点：模型在处理 ”it” 这个词时，不仅看到了它本身，还通过粗线条” 注意” 到了 ”animal”，从而理解 ”it” 指代的是动物。

4 模型架构深度剖析

本次实验复现的模型架构如图 3 所示，这是一个经典的 Encoder-Decoder 结构。

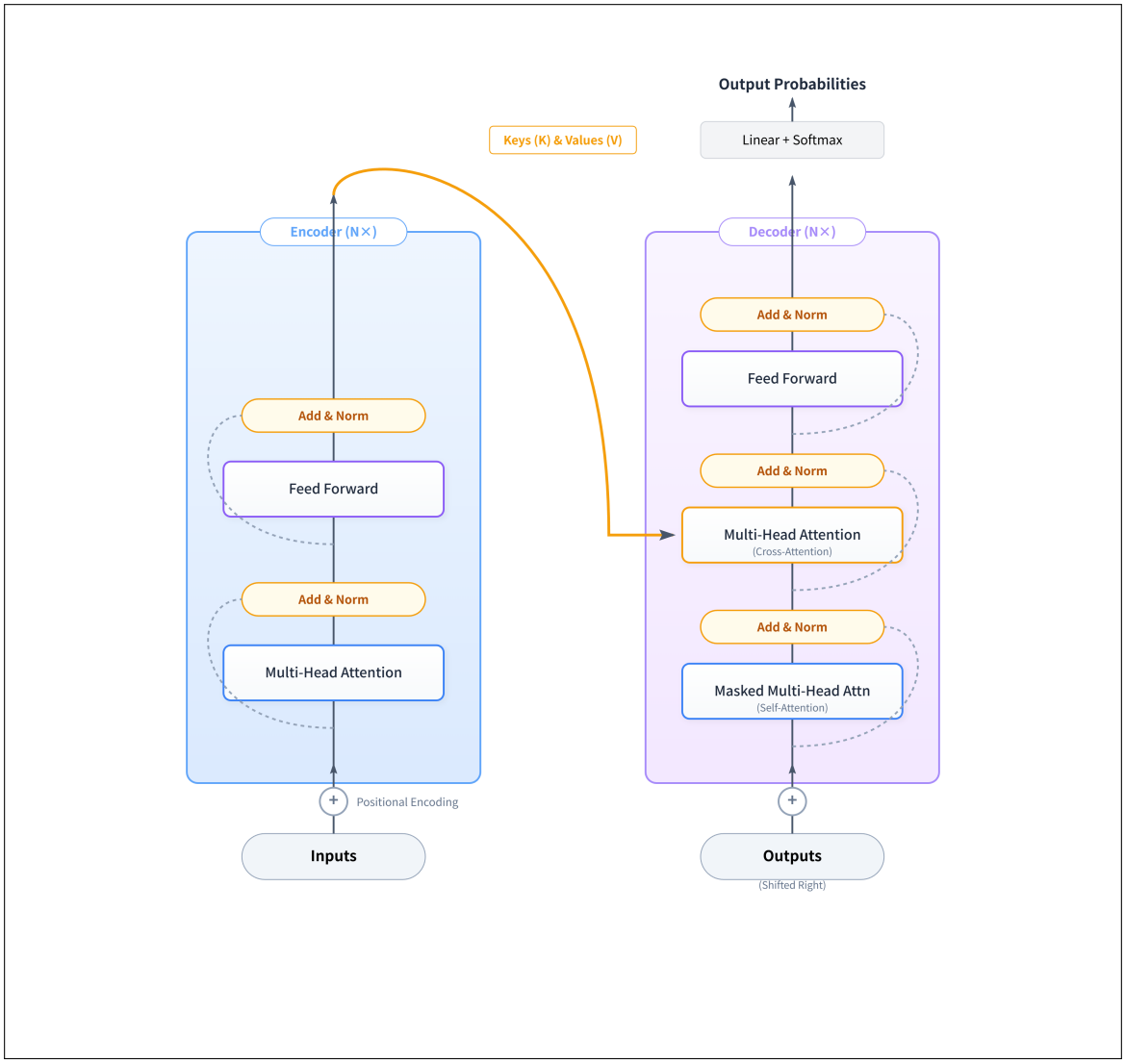

图 3: **Transformer 整体架构图。**(左) **编码器：**负责将源语言序列编码为高维语义矩阵。(右) **解码器：**利用自回归方式生成目标语言。(中) **交叉注意力：**解码器利用 Query 查询编码器的输出 (K, V)。

4.1 多头注意力 (Multi-Head Attention)

单个注意力头可能只能捕捉一种类型的关系。为了让模型同时关注不同子空间的信息，Transformer 引入了多头注意力机制。具体做法是将 Q, K, V 分别投影到 h 个不同的子空间，在每个子空间独立计算注意力，最后将结果拼接起来：

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{5}$$

其中 $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$。

直观地理解，不同的头可以学习关注不同类型的信息：有的头可能关注语法结构（主语在哪里），有的头可能关注语义指代（"it" 指的是什么），有的头可能关注位置关系（相邻的词）。这种设计大大增强了模型的表达能力。

4.2 编码器 (Encoder)：语义提取器

编码器由 N 层堆叠而成（本实验中 $N = 3$，原论文中 $N = 6$）。每一层包含两个核心子层：多头自注意力层和前馈网络层。前馈网络是一个简单的两层全连接网络，对每个位置独立应用：

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{6}$$

这里使用 ReLU 激活函数提供非线性变换能力。

每个子层都配有**残差连接**和**层归一化**，即 $\text{LayerNorm}(x + \text{Sublayer}(x))$。残差连接允许梯度直接流向底层，解决了深层网络的退化问题；层归一化则稳定了训练过程。

4.3 解码器 (Decoder)：自回归生成器

解码器的结构与编码器类似，但有两个关键差异。

第一是 **Masked Self-Attention (掩码自注意力)**。在训练时，我们将整个目标句子一次性输入解码器，但不能让模型在预测第 t 个词时看到第 $t + 1$ 及之后的词（否则就是作弊了）。解决方案是使用一个下三角掩码矩阵，将未来位置的注意力分数设为 $-\infty$，经过 Softmax 后这些位置的权重就变成了 0。

第二是 **Cross Attention (交叉注意力)**。这是连接源语言和目标语言的桥梁。在这一层中，Q 来自解码器上一层的输出，而 K, V 来自编码器的最终输出。这使得解码器在生成每个词时，都能" 查阅" 源句子的相关部分，决定应该翻译哪个词。

4.4 掩码机制可视化

为了更清楚地理解掩码的作用，图 4 展示了两种常用的掩码矩阵。

图 4: **掩码机制可视化。**左图 (Padding Mask)：展示了如何处理变长句子，矩阵会显式屏蔽掉 `<pad>` 对应的列（设为 $-\infty$），防止注意力机制关注到无意义的填充符。右图 (Causal/Look-ahead Mask)：展示了下三角矩阵，这是解码器的核心，确保预测第 t 个词时只能看到 1 到 $t-1$ 的词，上三角部分（未来信息）被屏蔽。

4.5 位置编码 (Positional Encoding)

由于 Self-Attention 机制本质上是集合运算，不具备顺序感。换句话说，如果我们打乱输入序列的顺序，Self-Attention 的输出也只是相应地打乱，不会有任何其他变化。这意味着” 我爱你” 和” 你爱我” 在纯 Attention 看来是一样的，这显然不行。

为了解决这个问题，Transformer 引入了位置编码。原论文使用正弦/余弦函数生成位置编码：

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}), \quad PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}) \tag{7}$$

这种编码方式有一个有趣的性质：对于任意固定的偏移量 k，PE_{pos+k} 可以表示为 PE_{pos} 的线性函数，这使得模型能够学习到相对位置关系。位置编码与词嵌入相加后，作为 Transformer 的输入。

5 实验设置与数据处理

5.1 数据集

实验使用的是英中平行语料，包含训练集 75,134 条、验证集 1,748 条、测试集 2,001 条。数据已经过 BPE 分词处理，但这里有一个重要的细节需要说明。

5.2 数据格式的”坑”

拿到数据后，我最初以为 `@@` 是标准 BPE 中的续接标记（表示当前 token 与下一个 token 应该拼接）。但仔细观察后发现，几乎每个 token 后面都有 `@@`，只有句末才没有。这说明在这份数据中，`@@` 实际上是**分词符号**，类似于空格的作用。

这个发现很重要，因为如果按照标准 BPE 的方式处理，会导致分词完全错误。代码中需要特别处理这一点：对于从零训练的 Transformer，我们用 `@@` 作为分隔符切分 token；对于 mBART 预训练模型，则需要先将 `@@` 去掉还原成原始文本，再交给 mBART 自带的 tokenizer 处理。

5.3 模型配置

从零训练的 Transformer 采用了相对较小的配置：$d_{model} = 256$，8 个注意力头，编码器和解码器各 3 层，前馈网络维度 512。这样的配置在 RTX 4090 上训练 10 个 epoch 大约需要几分钟。

6 实验结果与分析

6.1 从零训练的 Transformer：一场”灾难”

训练过程本身很顺利，loss 从 8.25 稳步下降到 6.88。然而，当我满怀期待地运行评估时，BLEU 分数只有 **0.03**。

更有趣的是翻译样例。我随机抽取了几条验证集数据，结果如表 1 所示。

表 1: 从零训练 Transformer 的翻译样例

类型	内容
源文	new Questions Over California Water Project
参考	加利福尼亚州水务工程的新问题
预测	我喜欢你说你 😀
源文	the $248 million in preliminary spending...
参考	这两条隧道的 2.48 亿美元初期费用支出...
预测	在 1998 年,19 年 1 月年 1 月年 1 月...
源文	on Wednesday, state lawmakers ordered...
参考	一些州议员也于周三责令...
预测	在意大利的餐厅的餐厅和花园和花园...

看到这些结果，我忍不住笑出了声。模型似乎学会了几个”万能回答”，无论你问

什么，它都会自信地回复” 我喜欢你说你” 或者开始疯狂重复某个词组。这是典型的小模型训练不足的表现：模型还没学会真正的翻译映射，只是在随机拼凑训练数据中的高频词。

6.2 交互模式的” 名场面”

为了进一步验证这个猜想，我进入了交互模式，手动输入一些句子测试。结果更加离谱：

```
源句子> new@@ Questions@@ Over@@ California@@ Water@@ Project
译文: 请求的URL:/lll

源句子> hello
译文: 我喜欢你说你

源句子> 你好帅
译文: 我喜欢你说你

源句子> 红红火火恍恍惚惚
译文: 我喜欢你说你
```

Listing 1: 交互模式测试

无论输入什么——英文、中文、甚至乱码——模型都会坚定不移地输出” 我喜欢你说你”。这大概是它在训练数据中学到的最” 安全” 的回答了 😀

顺便说一下，交互模式需要输入 `@@` 分隔的英文格式（与训练数据一致），输入中文会全部变成 `<UNK>`。不过以这个模型的水平，输入格式正确与否似乎也没什么区别...

6.3 问题分析

从零训练效果差的原因其实很明显。首先是**模型太小**，$d_{model} = 256$、3 层的配置相比生产级翻译模型差了好几个数量级。其次是**训练不足**，虽然 loss 在下降，但 10 个 epoch 远远不够让模型收敛到一个好的状态。最后是**数据量有限**，7.5 万条平行语料对于从零训练一个翻译模型来说确实偏少。

不过话说回来，这个实验的目的本来就是理解 Transformer 的原理，而不是训练一个能用的翻译系统。从这个角度看，实验是成功的——我们确实从零实现了一个能跑的 Transformer，只是它还没学会翻译而已 🙂

7 拓展：mBART-50 预训练模型

既然从零训练效果不佳，那不如试试站在巨人的肩膀上。mBART-50 是 Facebook 发布的多语言预训练模型，支持 50 种语言之间的互译，参数量约 6 亿。

7.1 一个小插曲：多语言的”惊喜”

第一次运行 mBART 评估时，我被输出惊呆了：

```
源文: new Questions Over California Water Project
参考: 加利福尼亚州水务工程的新问题
预测: neue Fragen uber das Wasserprojekt Kalifornien （德语!）

源文: critics and a state law maker say...
预测:                                          ... （印地语!）

源文: critics said the government funding...
预测: Les critiques ont dit que le gouvernement... （法语!）
```

Listing 2: mBART 的多语言输出

模型把英文翻译成了德语、印地语、法语、西班牙语... 就是不翻译成中文 🤔

原因是 mBART-50 是多语言模型，需要显式指定目标语言。我忘了设置 `forced_bos_token_id` 参数，导致模型随机选择了输出语言。修复后，一切正常了。

7.2 预训练模型的威力

修复后的评估结果令人满意：验证集 Loss 2.90，BLEU **0.51**。这个 BLEU 分数是从零训练模型的 **17 倍**！

翻译样例也终于像样了：

表 2: mBART-50 预训练模型的翻译样例

类型	内容
源文	new Questions Over California Water Project
参考	加利福尼亚州水务工程的新问题
预测	加利福尼亚水项目新问题 ✔

虽然不是完美翻译，但至少是正确的中文，而且语义基本准确。

7.3 BPE 分词痕迹的影响

不过预训练模型也不是完美的。由于数据中的 BPE 分词痕迹（空格）没有完全还原，导致一些有趣的翻译错误。比如 ”fun ding”（funding 的 BPE 切分）被翻译成了” 娱乐”（fun = 娱乐）😀，”cri tics”（critics）被翻译成了” 克里克”（音译）。

这说明数据预处理的重要性——即使是强大的预训练模型，也会被脏数据带偏。

8 总结与思考

8.1 实验心得

本次实验让我对神经机器翻译有了更深的理解。从理论上，Transformer 通过自注意力机制解决了 RNN 的并行计算和长距离依赖问题；从实践上，我体会到了从零训练小模型的艰难，以及预训练大模型的强大。

实验中遇到的各种” 翻车” 场景（” 我喜欢你说你”、多语言输出、”fun ding = 娱乐”）虽然搞笑，但都是很好的学习素材。它们让我意识到：机器翻译不仅仅是搭建一个模型那么简单，数据处理、参数设置、模型选择每一步都可能出问题。

由于数据规模很大，我没有在从零训练的 Transformer 上花太多时间调参和训练。直接用了课件中的基础配置，主要目的是验证实现的正确性。如果有更多时间和计算资源，可以尝试增加模型规模、训练轮数，或者使用更复杂的优化技巧（如学习率调度、正则化等），相信效果会有所提升。

最后，预训练模型以 17 倍的 BLEU 优势完胜从零训练的小模型，这充分说明了大规模预训练的价值。在实际应用中，除非有特殊需求，否则直接使用或微调预训练模型是更明智的选择。由于时间关系，我没有进行微调（Fine-tuning）。如果有时间和资源，微调预训练模型或许能带来更好的效果。

8.2 已知局限性

当前实现存在一些局限性。首先是**单向翻译**，代码只支持英语到中文的翻译方向，虽然 mBART 本身支持双向，但需要修改代码才能实现。其次是**解码策略**，从零训练的 Transformer 只实现了贪婪解码，没有实现 beam search，这可能影响翻译质量。最后是**数据格式依赖**，代码针对 `@@` 分隔格式做了特殊处理，不兼容标准 BPE 格式。

实验完成时间：*2025* 年 *12* 月 *9* 日