

你什么意思

目录

- 1 梳理题目要求
- 2 思路
- 3 相关知识学习
- 4 流程记录
- 5 运行成功截图

梳理题目要求

利用 **Transformer架构**，完成一个 **文本分类** 的任务。

通过训练一个**基于Transformer**的模型，你将帮助计算机像人类一样，理解文本的含义并进行 **准确分类**。

尝试搭建更高效的 Transformer 模型

尝试使用 **Hugging Face** 的 Transformer 库

比较不同模型的表现，并分析原因

思路

- 1 首先借助给出的论文链接和讲解视频，学习并理解**Transformer架构**。
- 2 尝试自己搭建一个**符合本任务情境**的Transformer架构
具体需要以下步骤：

1. 数据下载
2. 数据预处理
3. 模型训练

4. 模型评估
5. 可视化训练和评估结果
6. 输入影评文本进行测试

3 尝试搭建 **更高效的 Transformer 模型**，重复第二步

1. 限制目前模型性能的因素是什么？
2. 如何改进

4 尝试使用 **Hugging Face 的Transformers库** 中的预训练模型解决以上情感分类任务。

1. Hugging Face 是什么？
2. 如何使用

5 比较 **不同Transformer模型**（如**BERT**等）在情感分类任务中的表现，并分析原因

相关知识学习

Transformer 模型

背景

在 **Transformer** 问世之前，传统主流的 **序列到序列（Sequence-to-Sequence）模型** 都是基于 **RNN、CNN** 实现的，由于具有 **逐步处理序列** 的固有特性，无法避免地具有 **模型较为复杂、无法并行计算、长距离依赖捕捉困难、训练速度慢** 的问题。

Transformer 模型创新性地使用了 **完全基于注意力机制** 的算法，可以无需使用循环、递归结构，这使得它可以实现 **并行计算**，一次性处理整个序列，从而大幅度 **提升训练速度**，使它在 **WMT 2014 English-to-German** 和 **WMT 2014 English-to-French** 翻译任务中表现优异。

模型架构

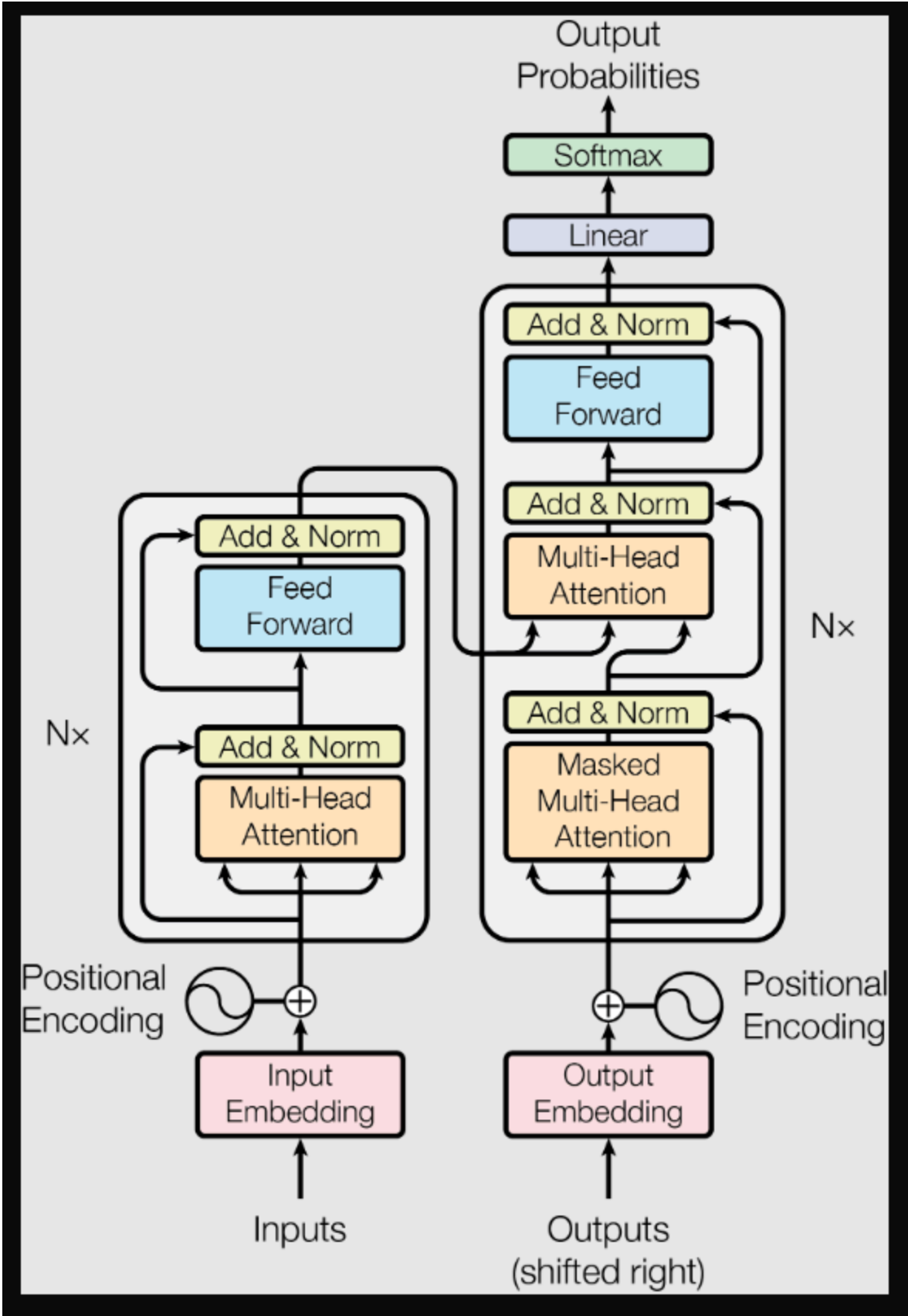


图 1: Transformer - 模型架构

Transformer 模型采用了 **编码器-解码器** 的架构:

- 编码器 (Encoder) : 主要任务是将输入序列转换为一种 **中间表示** , 下文将进行详细展开说明。原论文中包含 6 个相同结构的编码器层。
- 解码器 (Decoder) : 生成输出序列, 同样由 6 个相同结构的解码器层构成。解码器除了 **接收编码器传来的信息** 外, 还利用自注意力机制来捕捉 **已经生成的** 序列信息。

6个编码器和解码器的数量是原论文的设计, 实际应用中可调整结构。

每个编码器和解码器层都由以下的主要部分构成:

- 多头自注意力 (Multi-Head Self-Attention)
- 前馈网络 (Feed-Forward Neural Network)
- 残差连接 (Residual Connection)
- 层归一化 (Layer Normalization)

输入嵌入(Input Embedding)

1. 为什么需要输入嵌入

由于模型并不能处理 **离散** 的数据 (例如字符、ID) , 因而需要将其转化为 **可学习的向量** 形式, 这一步将通过 **输入嵌入** 完成。

输入嵌入指将序列中每个离散的 **token** (单词、字符等) **映射** 到一个连续的 **高维向量空间** 的过程。

对于 **可学习的向量**, 模型会根据任务目标不断调整他们, 使相似的词得到较为接近的向量表示。

2. 如何实现输入嵌入

- 使用 **嵌入矩阵 (Embedding Matrix)** , 其核心思想是将 **ID** 映射到对应的 **向量**。
- 构造一个形状为 $VocabSize \times d_{model}$ 的可学习矩阵 E , 其中 $VocabSize$ 是词典大小, d_{model} 是模型维度。
当一个词 **ID** (例如 w) , 就可以在矩阵中取出对应的行 $E[w]$, 得到该词的嵌入向量。
- 具体操作中, 使用到了**端到端学习**。即, 在训练开始时, 嵌入矩阵的参数是 **随机初始化** 的。在模型训练的过程中, 这些向量会随着 **反向传播** 被更新, 使得它们的数据在训练结束后 **最优**。

即嵌入矩阵的参数与模型其他部分一同通过梯度下降优化

位置编码(Positional Encoding)

1. 为什么需要位置编码

由于 Transformer **完全摒弃了循环结构**, 其输入中 **缺少位置信息**, (如“how are you”可能被理解

为“you how are”、“are you how”等排列组合形式）。

为了解决这个问题，引入了位置编码，使得模型能够获取序列中词与词之间的 **相对位置关系**。

2. 如何实现位置编码

- 《Attention is All Your Need》文章中 **使用正弦和余弦函数** 生成固定的周期性编码，这样每个位置都会获得一个 **唯一的向量**，同时这种编码方式允许模型对未见过的序列长度具有一定的泛化能力
- 具体公式如下：

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

3. 后续研究

原论文采用固定正弦编码，但后续研究（如BERT）也使用可学习的位置嵌入。

获取输入序列向量

1. 什么是输入序列向量

在实现 **输入嵌入** 以及 **位置编码** 的计算之后：

输入嵌入 将每个 token 映射为一个向量表示，主要 **捕捉词义或子词意义**。

位置编码 向每个向量中加入 **位置信息**，使模型能够区分序列中不同位置的 token。

而通过将二者进行 **叠加**，便可以得到携带二者信息的 **输入序列向量**，将这个向量输入到编码器中

2. 如何计算输入序列向量

文章中给出处理方式：

$$InputRepresentation_i = Embedding(w_i) + PositionalEncoding(i)$$

其中，

- w_i 是第 i 个 token 的 ID
- $Embedding(w_i)$ 是输入嵌入向量
- $PositionalEncoding(i)$ 是该位置的位置编码

3. 为什么是以相加的形式

- 这里的输入嵌入和位置编码的维度信息相同，二者进行相加操作，可以保证维度匹配，计算简便高效
- 对于每一个 token 其含义还是主要由输入嵌入决定的，那 $[-1,1]$ 范围的位置编码对数值改变很小，是一个轻微的调整，符合单词的语义信息是核心，位置信息仅作为辅助。

自注意力机制(Self-Attention)

1. 什么是自注意力机制

自注意力机制 (Self-Attention) 是 Transformer 模型的核心组件之一，它使得模型能够在处理序列数据时 **捕捉到序列中各个位置之间的相互依赖关系**。

在序列中，每个位置的 token 在理解其含义时，往往需要参考序列中 **其他位置的信息**。

自注意力机制通过对序列中所有位置的信息进行 **加权求和**，让每个位置的表示都能够 **融合全局信息**。这样一来，不论序列中两个词之间的距离有多远，都可以直接建立联系。

2. 工作原理和公式

自注意力机制的目标是让模型在处理每个单词时，都能 **关注到序列中其他相关的单词**，与此同时做到 **让重要的信息贡献更多，次要的信息贡献较少**。

其核心过程包括以下几个步骤：

i. 生成 **查询、键、值 (Query, Key, Value)** 向量

对于输入中的每个位置，会将其映射为三个向量：

- a. 查询向量 **Q** 表示当前需要关注信息的向量
- b. 键向量 **K** 表示每个位置的信息特征，用于与查询进行匹配。
- c. 值向量 **V** 存储实际的信息，最终会根据匹配程度被加权求和。

对于输入的每个 token，其向量表示通过不同的线性变换得到对应的 **Q、K 和 V**：

$$Q = XW^Q, K = XW^K, V = XW^V$$

其中，

- X 表示输入序列中各个 **token** 的嵌入表示
- W^Q, W^K, W^V 分别表示可以学习的权重矩阵

ii. 计算注意力权重

通过计算查询 **Q** 与键 **K** 的 **点积**，再经过 **缩放**（除以 $\sqrt{d_k}$ ，其中 d_k 是每个注意力头的维度，可以使用 $d_k = d_{model}/h$ 计算，其中 h 为多头注意力的头数）和 **softmax** 归一化，得到每个位置的权重。

- **点积** 的意义在于得到每一个查询 q_i （来自位置 i ），与所有键 k_j 得到相似度（相似度与点积后的结果成正比），即注意力得分
- **缩放** 的意义在于以避免在维度较高时点积值过大，导致 Softmax 处理之后产生梯度消失的现象

iii. **Softmax** 归一化

这里的目的是把所有单词的注意力分数变成概率分布（和为1），变成权重信息

iv. 用这些权重对值 **V** 进行加权求和，得到当前位置的输出。实际操作中可以令两矩阵相乘。

公式如下：

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

3. 优势

- i. 与 RNN 需要依次计算不同，自注意力机制可以对整个序列并行处理，极大地提高了计算效率。
- ii. 无论两个 token 的位置相隔多远，都可以通过直接计算注意力权重建立联系，有效捕捉全局信息。自注意力允许任意两个位置直接交互，彻底解决了的长期依赖衰减问题。

多头注意力机制 (Multi-Head Attention)

1. 为什么需要多头注意力？

- 单一的注意力机制虽然能够捕捉到 **部分依赖关系**，但可能难以从不同子空间中提取足够的特征信息。多头注意力通过 **并行地** 进行 **多组** 注意力计算，使得模型可以 **从多个角度**（不同的子空间）提取信息。
- 这种设计不仅增强了模型对多样化特征的捕捉能力，还提高了整体的表达能力与鲁棒性，使模型能更好地处理复杂依赖关系。

2. 实现方法

- i. 线性变换：将输入向量分别经过不同的线性变换，将 Q, K, V 拆分为 h 个头（每个头维度为 $d_k = d_{model}/h$ ）
- ii. 并行计算注意力：对每个头分别计算自注意力，再将所有头的输出 **拼接起来**，形成一个维度为 d_{model} 的向量。
- iii. 最终映射：通过一个线性层，将拼接后的结果映射到原始维度上。

将 Q, K, V 分别通过 **不同的线性变换**（采用不同的矩阵 W_q, W_k, W_v 进行变换）得到多个“头”（head），每个 head **独立计算** 自注意力。

其中，头数（heads）与维度拆分的关系是： $d_k = \frac{d_{model}}{h}$

将所有 head 的输出 **拼接起来**，变回 d_{model} 维度：

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \dots, head_h)W^O$$

W^O 是一个投影矩阵，用来把拼接后的向量映射回 d_{model} 维度。

这种设计可以使模型从 **不同角度学习信息**，更好地捕捉到序列中复杂的依赖关系。

带掩码的多头注意力 (Masked Multi-Head Attention)

1. 为什么需要

在自回归任务（文本生成、机器翻译、文本摘要）中，模型在生成的时候，不可以提前“偷看”完整答案，所以需要把未来的结果 Masked 掉（**本题不属于自回归任务**）

2. 如何实现

在普通的多头注意力的基础上，将未来单词的注意力分数赋值为负无穷，这样在经过 Softmax 计算之后，注意力权重变成了0，实现完全能忽略未来的单词

公式：

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

其中：

- M 是一个 **Mask 矩阵**，用于屏蔽未来的单词，通常：

$$M_{i,j} = \begin{cases} 0, & j \leq i \quad (\text{允许查看当前和过去的单词}) \\ -\infty, & j > i \quad (\text{屏蔽未来单词}) \end{cases}$$

前馈网络 (FFN)

前馈网络的目的在于增强模型的学习能力，能够学习到更加复杂的特征

其基本结构通常包括两个 **线性变换** 以及中间的 **激活函数**。具体来说，对每个位置的输入向量 x 来说，FFN 的计算公式通常表示为：

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

原论文中，

$d_{model} = 512$ ，中间层维度 $d_{ff} = 2048$ ，使用ReLU激活。

- **第一层线性变换**：将输入向量 x 投影到一个更高的维度。这一步的输出再经过 ReLU 引入非线性，以增强模型的表达能力。
- **第二层线性变换**：将经过非线性处理的结果再投影回原始模型维度。

特点：

- 中间层的维度要比输入和输出的维度大，这使得模型能够在非线性激活下学习到更丰富的特征表示。

残差连接 (Residual Connection) 和层归一化 (Layer Normalization)

Transformer 的每个子层（不论是自注意力子层还是 FFN 子层）中，都使用了残差连接和层归一化，这两者的组合对模型训练起到了重要作用。

残差连接

目的：

- **缓解梯度消失问题**：让信息在深层网络中更容易传播，从而避免梯度消失，提高训练效果。
- **防止信息流失**：让每一层的输入信息 "绕过" 这个层，直接加到输出上，这样即使这一层学习得不好，原始信息也能部分保留。

实现：

在每个子层（多头注意力或 FFN 子层）的输出上，加上该子层的输入，即：

$$\text{Output} = x + \text{FNN}(x)$$

$$\text{Output} = x + \text{MultiHeadAttention}(x)$$

其中：

- x 是该子层的输入

每个子层（多头注意力或FFN）的输出为输入与子层结果的残差和，再经过层归一化。

层归一化

目的：

- 在深度模型中防止梯度爆炸或梯度消失，提高训练的稳定性。
- 将子层的输出调整到一个统一的尺度，有助于缓解不同层输出分布差异带来的问题，使得模型更快收敛。

为什么不使用 **批归一化**？

- 层归一化不受批量大小的影响

实现：

在应用残差连接之后，对结果进行归一化处理。公式为：

$$\text{LayerNorm}(z) = \gamma \odot \frac{z - \mu}{\sigma + \epsilon} + \beta$$

其中：

- $z = x + \text{Sublayer}(x)$
- μ 和 σ 分别为 z 的均值和标准差,
- γ 和 β 是可学习的参数,
- ϵ 是一个很小的常数, 防止除零错误。
- 计算时, 均值和标准差是 **基于样本的特征维度**, 然后进行缩放和平移。

在 Transformer 模型架构里, 每一次残差连接之后, 都会加上 LayerNorm:

$$\text{Output} = \text{LayerNorm}(\text{Residual} + x)$$

如何使 Transformer 模型和本题相符合

在本题的任务中, 所需要的 **Transformer** 架构和 《Attention is All Your Need》中提出的 **Transformer** 架构有相同和不同之处。

首先分析两个任务的异同之处:

- 相同: 都需要接受并理解一段文本, 并对其进行处理
- 不同: 输入是电影评论文本而非句子对, 输出是二分类而非一个序列

由于他们的相同之处, 因而:

1. 都需要输入嵌入、位置编码、多头注意力、FNN、残差连接、层归一化等基本部件。
2. 需要相似的算法流程

与此同时,

1. 由于不需要输出目标序列, 因而只需要 **Encoder** 模块处理输入序列即可, 不需要 **Decoder** 模块生成目标序列。
2. 可以通过如 **BERT** 预训练模型, 这样可以提高文本识别的准确率

尝试构建更高效的 Transformer 模型

限制当前模型性能的 **制约因素** 是什么, 然后结合 **参数优化** 和 **技术优化** 两方面进行优化:

- **参数优化**
 - **EMBED_DIM** 低维度可能导致模型表达能力不足, 高维度可能引入过多噪声并增加计算量。
 - **NUM_HEADS** 影响 Transformer 提取特征的能力, 更多的头数可以捕获更丰富的模式, 但计算开销更大。
 - **HIDDEN_DIM** 过小的隐藏层可能限制模型的非线性表达能力, 过大则可能导致过拟合。
 - **NUM_LAYERS** 过少的层数可能导致模型学习不到足够的层次化信息, 过多的层数会增加计算开销和过拟合风险。
 - **N_EPOCHS** 轮数太少, 模型无法充分收敛; 轮数太多, 容易过拟合。

- max_length 过短的 max_length 可能会截断关键信息，而过长的 max_length 会增加计算成本。

- **使用更高级的优化技术**

- 使用学习率调度器

- 使用 **OneCycleLR** 学习率调度

OneCycleLR 它能够 **动态调整** 学习率，在训练过程中 **先升后降**，从而 **提升收敛速度、提高泛化能力，避免训练陷入局部最优**。

```
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-2)
```

```
scheduler = torch.optim.lr_scheduler.OneCycleLR(  
    optimizer,  
    max_lr=1e-3,          # 训练中最高学习率  
    total_steps=N_EPOCHS * len(train_loader), # 总训练步数 = epoch数 * 每个epoch的t  
    pct_start=0.3,        # warmup 30% 训练步数  
    anneal_strategy='cos', # 余弦退火策略  
    div_factor=10,         # 初始学习率 = max_lr / div_factor  
    final_div_factor=100 # 结束学习率 = max_lr / final_div_factor  
)
```

- 使用 **梯度裁剪**

训练深度 **Transformer** 时，可能出现 **梯度爆炸**

梯度裁剪 通过设置梯度最大范数，防止梯度过大。

```
def train(model, loader, optimizer, criterion):  
    model.train()  
    epoch_loss = 0  
    for text, label in loader:  
        text, label = text.to(device), label.to(device)  
        predictions = model(text)  
        loss = criterion(predictions, label)  
        optimizer.zero_grad()  
        loss.backward()  
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0) # 添加梯度裁剪  
        optimizer.step()  
        epoch_loss += loss.item()  
    return epoch_loss / len(loader)
```

使用 Hugging Face 的 Transformer 预训练模型

Hugging Face 提供了已经训练好的模型，在具体的使用场景中只需要调用即可：

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")

result = classifier(["I really loved this movie! The acting was great and the plot was fantastic",
                    "This was the worst film I have ever seen. Complete waste of time."
                    ])

print(result)
```

如此便可以得到对于文本情感的判断结果：

```
[{'label': 'POSITIVE', 'score': 0.9598045349121094}, {'label': 'POSITIVE', 'score': 0.9998813861}
```

总而言之，方便调用，并且性能强大

BERT

Transformer 的 Encoder 部分能够很好地学习语言的表示 (representation)，因此，我们可以仅使用 Encoder 进行语言建模，而无需 Decoder。通过堆叠多个 Encoder 层，即可构建出 **BERT** 的架构。

然而，没有 Decoder 进行映射的情况下，该如何在 **pre-training** 训练 BERT 呢？

BERT 采用两个训练任务来学习语言特性：

1. **MLM** 任务让模型在双向上下文信息的基础上预测被遮挡的词汇。
2. **NSP** 任务让模型判断两句话是否存在前后顺序关系，以捕捉句间语义关联。

BERT 的特性：

- BERT 的分词特性并不是简单的空格断开，而是采用 **WordPiece** 分词方法，如 "drawing" 会被拆分为 "draw" 和 "##ing"（## 标记非词首），以便更好地处理单词的不同形态变化。
- 相比 Transformer 中基于正余弦函数的固定位置编码，BERT 采用了 **可学习的参数进行位置编码**。
- 通过额外的嵌入信息标记 token 所属的句子，区分不同的输入句子。
- BERT 采用了更多的 Encoder 层数，除了参数设置不同，结构和 Transformer 的 Encoder 结构完全相同

对于 **MLM** 训练任务：

- 随机遮挡 12% 的 token（即用 [MASK] 替换），1.5% 的 token 被替换为随机单词，1.5% 保持不变但被标记，剩余 85% 维持原样。
- 通过 Encoder 提取上下文信息，最终输出一个长度为 VocabSize 的概率分布

- 计算预测结果与真实 token 的one-hot 形式之间的交叉熵损失，并通过梯度下降优化参数

对于 **NSP** 训练任务:

- 给定两段文本 A 和 B，其中 50% 的情况 B 是 A 的下一句，50% 的情况是随机选择的一句话作为 B。
- 读取 [CLS] 标记对应的表示，并通过 Encoder 计算出二分类结果（用于回答是否为下一句）。
- 计算二分类的交叉熵损失，并更新模型参数。

fine_turning 阶段:

在具体任务上微调BERT时，只需替换 Output Layer 为适用于该任务的特定层。例如，文本分类任务可将 Output Layer 替换为分类层。由于BERT已经过大规模语料的预训练，微调时只需较少的训练轮数，即可获得良好的效果。

BERT 为经典的 Transformer 架构提供了已经经过训练好的模型参数，通过调用 transformers 里的 API 使用

```
from transformers import BertTokenizer, BertForSequenceClassification

# 加载 BERT 分词器
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# 加载 BERT 预训练模型
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=1).to(dev:

# 设置训练轮数
N_EPOCHS = 3
```

运行程序证明使用 BERT 之后的结果显著提升，截图将在运行截图里呈现

流程记录

(不使用 **BERT**，但使用学习率调度器、梯度裁剪等技术优化版本)

1. 导入必要的库

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchtext.datasets import IMDB
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torch.utils.data import DataLoader, Dataset
import matplotlib.pyplot as plt
import numpy as np
```

使用支持深度学习的 `torch` 框架，用于构建模型和张量计算

使用处理文本数据的工具库 `torchtext`

使用绘图工具库 `matplotlib`

2. 数据预处理

i. 定义分词器

使用 `SpaCy` 的英文分词器将文本分割成单词和符号，方便下一步的词嵌入处理。

```
# 定义文本分词器，使用 spaCy 进行英文分词
tokenizer = get_tokenizer("spacy", language="en_core_web_sm")
```

ii. 生成词汇表

`collect_tokens` 函数将文本数据集迭代器转换为分词迭代器。

从训练数据中构建词汇表，最大词汇量为 20000，特殊符号有 `<pad>`（用于填充序列）和 `<unk>`（表示未在词汇表中的单词）。

`set_default_index` 确保遇到未在词汇表中的单词时，将其索引设置为 `<unk>` 的索引。

```

def collect_tokens(data_iter):
    """
    :param data_iter: IMDB数据集
    :return: 返回单词列表
    """
    all_tokens = []
    for _, text in data_iter:
        tokens = tokenizer(text)
        all_tokens.append(tokens)
    return all_tokens # 返回所有 tokens 的列表

# 加载 IMDB 数据集
train_iter = IMDB(split="train")
test_iter = IMDB(split="test")

# 构建词汇表
vocab = build_vocab_from_iterator(collect_tokens(train_iter), max_tokens=20000, special
vocab.set_default_index(vocab["<unk>"]) # 设置未记录的词用 <unk> 代替

```

iii. 定义预处理函数

text_pipeline 将文本转换为对应的词汇表索引序列，并截断为最大长度 100

label_pipeline 将情感标签 "pos" 和 "neg" 转换为 1 和 0。

```

def text_pipeline(text):
    """
    :param text: 欲处理的文本列表
    :return: 文本对应的索引列表
    """
    tokens = tokenizer(text)
    max_length = 100
    tokens = tokens[:max_length]
    index = []
    for token in tokens:
        index.append(vocab[token])
    return index

def label_pipeline(label):
    """
    :param label: 情感标签 pos 或 neg
    :return: 将 pos 或 neg 映射为 1 或 0
    """
    return 1 if label == "pos" else 0

```

iv. 自定义数据集类

IMDBDataset 继承自 torch.utils.data.Dataset，用于封装和操作数据。

__init__ 方法对数据进行预处理，__len__ 返回数据集的大小，__getitem__ 根据索引返回样本。

```
class IMDBDataset(Dataset):
    """自定义 IMDB 数据集类，用于加载 IMDB 电影评论数据"""
    def __init__(self, data_iter):
        """
        初始化
        :param data_iter: 同时包含 (label, text) 的数据迭代器
        """
        self.data = []
        for label, text in data_iter:
            self.data.append((text_pipeline(text), label_pipeline(label)))

    def __len__(self):
        """
        :return: 返回数据集 self.data 长度
        """
        return len(self.data)

    def __getitem__(self, idx):
        """
        :param idx: 索引
        :return: 索引对应的单词
        """
        return self.data[idx]

# 创建训练集和测试集
train_data = IMDBDataset(train_iter)
test_data = IMDBDataset(test_iter)
```

v. 定义 collate_fn

将一批样本的文本序列填充到相同长度，方便批量处理。


```

def collate_fn(batch):
    """
    处理批量数据，对文本进行填充使其长度一致
    :param batch: 批量数据
    :return: 填充处理之后的结果
    """
    texts, labels = zip(*batch)

    lengths = []
    for text in texts:
        lengths.append(len(text))

    max_len = max(lengths)

    padded_texts = []
    for text in texts:
        padded_texts.append(text + [vocab["<pad>"]] * (max_len - len(text)))

    return torch.tensor(padded_texts), torch.tensor(labels, dtype=torch.float)

```

3. 创建数据加载器

根据设备是否支持 CUDA 选择计算设备。

`DataLoader` 将数据集封装为可迭代的批量数据，训练集数据打乱，测试集数据不打乱。

```

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# 创建数据加载器
train_loader = DataLoader(train_data, batch_size=64, shuffle=True, collate_fn=collate_fn)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False, collate_fn=collate_fn)

```

4. 模型构建

i. 位置编码

由于 Transformer 本身没有位置信息，通过正弦和余弦函数计算位置编码，将其与嵌入向量相加，帮助模型捕捉序列中的位置信息。

```

class PositionalEncoding(nn.Module):
    """ 定义位置编码类，为 Transformer 提供位置信息 """
    def __init__(self, d_model, dropout):
        """
        :param d_model: 词向量的维度
        :param dropout: Dropout 的概率，用于防止过拟合。
        """
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        max_len = 5000 # 最大序列长度
        position_code = torch.zeros(max_len, d_model) # 储存位置编码

        position_item = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1) # 生成

        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-np.log(10000.0) /

        position_code[:, 0::2] = torch.sin(position_item * div_term) # 偶数位置
        position_code[:, 1::2] = torch.cos(position_item * div_term) # 奇数位置

        self.register_buffer('position_code', position_code) # 将位置编码矩阵注册为模型不

    def forward(self, x):
        """
        前向传播
        :param x: 输入张量 (batch_size, seq_len, d_model)
        :return: 添加位置编码之后的张量
        """
        x = x + self.position_code[:x.size(1)] # 只取前 seq_len 个位置编码
        return self.dropout(x)

```

ii. Transformer 分类模型

```

class TransformerClassifier(nn.Module):
    """ 定义 Transformer 分类模型 """
    def __init__(self, vocab_size, embed_dim, num_heads, num_layers, hidden_dim, dropout):
        """
        初始化
        :param vocab_size: 词汇表的大小
        :param embed_dim: 词嵌入的维度
        :param num_heads: Multi-Head 的头数
        :param num_layers: Encoder 的层数
        :param hidden_dim: FNN的隐藏层维度
        :param dropout: dropout 率
        """
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embed_dim) # 进行词嵌入，完成离散的索引
        self.pos_encoder = PositionalEncoding(embed_dim, dropout) # 完成位置信息的添加

        encoder_layers = nn.TransformerEncoderLayer(d_model=embed_dim, nhead=num_heads,

        self.transformer_encoder = nn.TransformerEncoder(encoder_layers, num_layers) #
        self.fc = nn.Linear(embed_dim, 1) # 全连接层，完成分类
        self.dropout = nn.Dropout(dropout) # Dropout 层

    def forward(self, text):
        """
        前向传播，完成计算流程
        :param text: 输入的文本张量 形状为 (batch_size, seq_len)
        :return: 分类结果，形状为 (batch_size)
        """
        embedded = self.embedding(text) # 完成词嵌入映射，得到 (batch_size, seq_len, emb
        embedded = self.dropout(embedded) # dropout

        embedded = self.pos_encoder(embedded) # 添加位置编码

        embedded = embedded.transpose(0, 1) # 调整张量的维度为 (seq_len, batch_size, emb

        output = self.transformer_encoder(embedded) # 通过 Transformer 编码器
        output = output.mean(dim=0) # 对序列维度 (seq_len) 取均值，得到句子级别的表示，形

        return self.fc(output).squeeze(1) # 通过全连接层，得到分类结果，形状为 (batch_siz

```

`nn.Embedding` 将输入的索引序列转换为密集向量。

`PositionalEncoding` 添加位置信息。

`nn.TransformerEncoderLayer` 是 Transformer 编码器层，`nn.TransformerEncoder` 由多个编码器层堆叠组成。

最后通过全连接层 `nn.Linear` 将嵌入维度转换为 1 维，用于二分类。

5. 训练和评估

i. 定义优化器和损失函数

使用 `AdamW` 优化器更新模型参数，学习率为 `1e-4`。

`nn.BCEWithLogitsLoss` 是带 `Sigmoid` 激活的二元交叉熵损失函数，用于二分类问题。

```
# 设置模型参数
embed_dim = 512
num_heads = 4
num_layers = 4
hidden_dim = 512
dropout = 0.2
epochs = 10

# 初始化模型
model = TransformerClassifier(
    vocab_size=len(vocab),
    embed_dim=embed_dim,
    num_heads=num_heads,
    num_layers=num_layers,
    hidden_dim=hidden_dim,
    dropout=dropout
).to(device)

# 定义优化器和损失函数
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-2)
scheduler = OneCycleLR(
    optimizer,
    max_lr=1e-4,          # 训练中最高学习率
    total_steps=epochs * len(train_loader), # 总训练步数 = epoch数 * 每个epoch的batch数
    pct_start=0.3,        # warmup 30% 训练步数
    anneal_strategy='cos', # 余弦退火策略
    div_factor=10,         # 初始学习率 = max_lr / 10
    final_div_factor=100   # 结束学习率 = max_lr / 100
)

criterion = nn.BCEWithLogitsLoss().to(device)
```

ii. 定义训练和评估函数

训练模型

```
def train(model, loader, optimizer, criterion):
    model.train()
    epoch_loss = 0
    for text, label in loader:
        text, label = text.to(device), label.to(device)
        predictions = model(text)
        loss = criterion(predictions, label)
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        scheduler.step()

    epoch_loss += loss.item()
    return epoch_loss / len(loader)
```

评估模型

```
def evaluate(model, loader, criterion):
    model.eval()
    epoch_loss = 0
    correct, total = 0, 0
    with torch.no_grad():
        for text, label in loader:
            text, label = text.to(device), label.to(device)
            predictions = model(text)
            loss = criterion(predictions, label)
            epoch_loss += loss.item()
            preds = torch.sigmoid(predictions) > 0.5
            correct += (preds == label).sum().item()
            total += label.size(0)
    return epoch_loss / len(loader), correct / total
```

iii. 开始训练、保存结果、绘图

a. 开始训练

```

# 训练过程
train_losses, test_losses, test_accs = [], [], []

for epoch in range(N_EPOCHS):
    train_loss = train(model, train_loader, optimizer, criterion)
    test_loss, test_acc = evaluate(model, test_loader, criterion)
    train_losses.append(train_loss)
    test_losses.append(test_loss)
    test_accs.append(test_acc)
    print(f'Epoch: {epoch + 1:02}, Train Loss: {train_loss:.3f}, Test Loss: {test_']

```

b. 保存训练后的结果

```

# 保存训练后的参数信息
torch.save(model.state_dict(), "model.pth")
# 保存词汇表信息
torch.save(vocab, "vocab.pth")

```

c. 绘图

使用 matplotlib 工具库将训练以及测试的结果以图表的形式绘制出来

```

# 绘制模型训练和评估的可视化图表
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.legend()
plt.title('Loss Curve')
plt.subplot(1, 2, 2)
plt.plot(test_accs, label='Test Accuracy')
plt.legend()
plt.title('Accuracy Curve')
plt.tight_layout()
plt.show()

```

6. 应用测试

i. 加载已经训练好的参数

```

def load_vocab():
    """加载训练好的词汇表"""
    vocab = torch.load("vocab.pth")
    vocab.set_default_index(vocab["<unk>"])
    return vocab

def load_model_and_vocab():
    """加载模型和词汇表"""
    vocab = load_vocab()
    model = TransformerClassifier(
        vocab_size=len(vocab),
        embed_dim=EMBED_DIM,
        num_heads=NUM_HEADS,
        num_layers=NUM_LAYERS,
        hidden_dim=HIDDEN_DIM,
        dropout=DROPOUT
    ).to(device)

    model.load_state_dict(torch.load("model.pth", map_location=device))
    model.eval()
    return model, vocab

```

ii. 预测给定的句子情感

```

def predict_sentiment(model, vocab, text):
    """预测单条文本的函数"""
    model.eval()
    text_tensor = torch.tensor(text_pipeline(text, vocab), dtype=torch.long).unsqueeze(
    with torch.no_grad():
        prediction = model(text_tensor)
        probability = torch.sigmoid(prediction).item()
    return "Positive" if probability > 0.5 else "Negative", probability

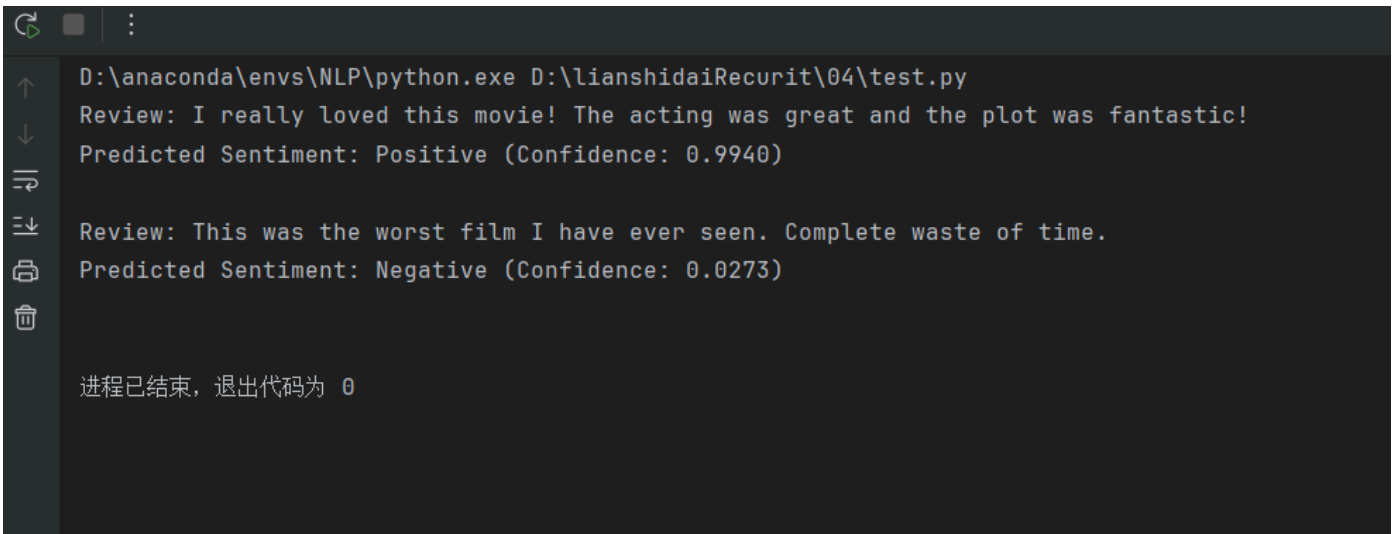
# 加载模型
model, vocab = load_model_and_vocab()

# 测试数据示例
test_texts = [
    "I really loved this movie! The acting was great and the plot was fantastic!",
    "This was the worst film I have ever seen. Complete waste of time."
]

# 进行预测
for text in test_texts:
    sentiment, prob = predict_sentiment(model, vocab, text)
    print(f"Review: {text}\nPredicted Sentiment: {sentiment} (Confidence: {prob:.4f})\n")

```

运行结果如下，证明能够成功加载已经训练的参数进行预测任务



```

D:\anaconda\envs\NLP\python.exe D:\lianshidaiRecurit\04\test.py
Review: I really loved this movie! The acting was great and the plot was fantastic!
Predicted Sentiment: Positive (Confidence: 0.9940)

Review: This was the worst film I have ever seen. Complete waste of time.
Predicted Sentiment: Negative (Confidence: 0.0273)

进程已结束，退出代码为 0

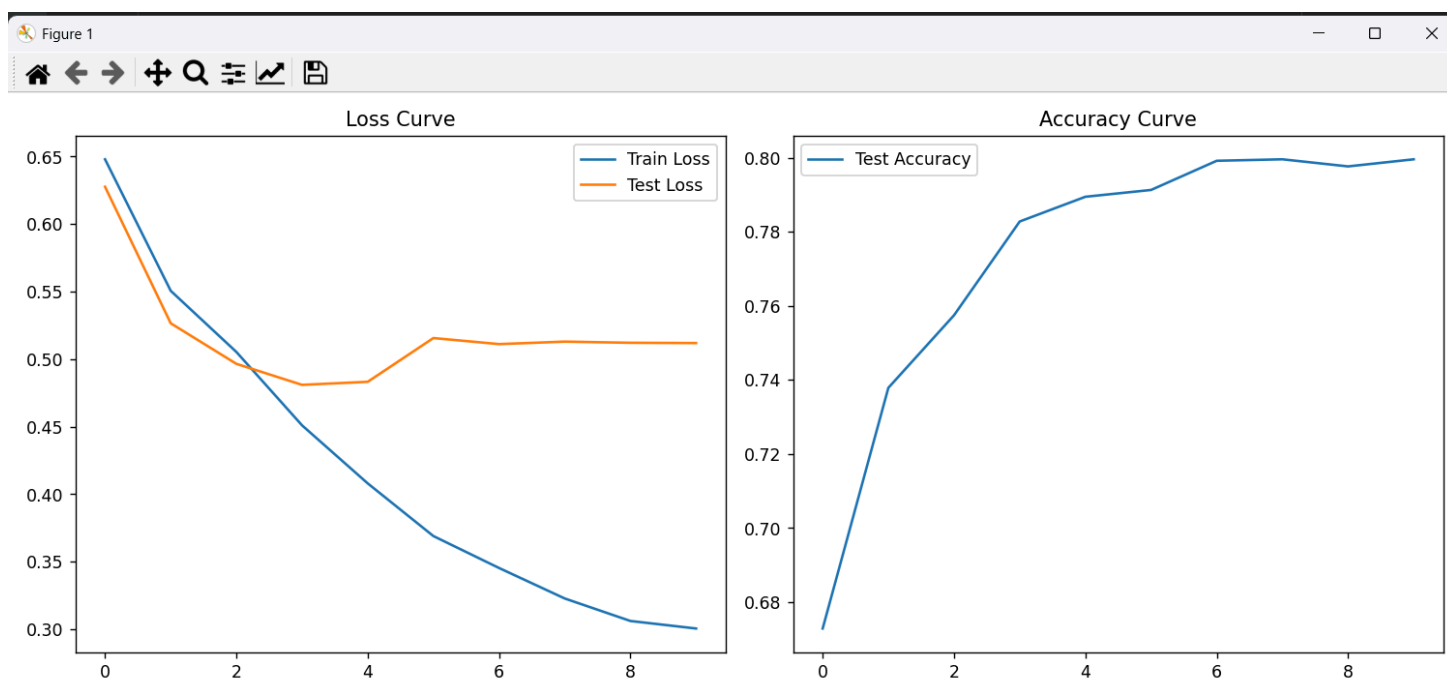
```

运行成功截图

Transformer 架构：


```
Epoch: 01, Train Loss: 0.648, Test Loss: 0.628, Test Acc: 67.28%
Epoch: 02, Train Loss: 0.551, Test Loss: 0.526, Test Acc: 73.78%
Epoch: 03, Train Loss: 0.505, Test Loss: 0.496, Test Acc: 75.74%
Epoch: 04, Train Loss: 0.451, Test Loss: 0.481, Test Acc: 78.28%
Epoch: 05, Train Loss: 0.408, Test Loss: 0.483, Test Acc: 78.94%
Epoch: 06, Train Loss: 0.369, Test Loss: 0.516, Test Acc: 79.13%
Epoch: 07, Train Loss: 0.345, Test Loss: 0.511, Test Acc: 79.92%
Epoch: 08, Train Loss: 0.323, Test Loss: 0.513, Test Acc: 79.96%
Epoch: 09, Train Loss: 0.306, Test Loss: 0.512, Test Acc: 79.76%
Epoch: 10, Train Loss: 0.301, Test Loss: 0.512, Test Acc: 79.96%
```

进程已结束，退出代码为 0



使用 BERT :

```
Epoch: 1, Train Loss: 0.439, Test Loss: 0.456, Test Acc: 80.28%
Epoch: 2, Train Loss: 0.250, Test Loss: 0.302, Test Acc: 87.58%
Epoch: 3, Train Loss: 0.092, Test Loss: 0.388, Test Acc: 87.28%
```

进程已结束，退出代码为 0

