大模型基础与应用期中作业实验报告

学生姓名: 田京雷

学号: 25140197

日期: 2025年11月

---

1. 作业说明

1.1 作业总体说明

本作业要求从零实现Transformer架构，并在小规模文本建模任务上进行训练和消融实验。

1.2 代码开源要求

本项目代码已开源，包含以下结构：

```
.
├── src/
│   ├── model.py           # Transformer核心实现
│   ├── data_loader.py     # 数据加载器
│   ├── train.py           # 训练脚本
│   └── evaluate.py        # 评估和文本生成
├── configs/
│   ├── base.yaml          # 基础配置
│   └── ablation_2heads.yaml  # 消融实验配置
├── results/               # 训练曲线图和实验结果
├── requirements.txt       # 依赖包
├── README.md              # 完整说明文档
└── scripts/run.sh         # 训练脚本
```

1.3 数据集选择

本实验使用**Tiny Shakespeare**数据集，这是一个字符级语言建模数据集：

- **来源**：Karpathy’s char-rnn项目
- **大小**：约1MB，1,115,394个字符
- **词汇表**：65个唯一字符（包括字母、标点、空格等）

- **特点**：数据集小巧，训练快速，适合验证模型实现
- **下载链接**：https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt

---

2. 引言 (Introduction)

2.1 研究背景

序列建模是自然语言处理领域的核心任务之一。传统的循环神经网络（RNN）和长短期记忆网络（LSTM）虽然能够处理序列数据，但存在以下局限性：

1. **顺序计算限制**：RNN必须按时间步顺序处理序列，无法并行化，训练速度慢
2. **长距离依赖问题**：即使使用LSTM，也难以有效捕获序列中相距较远的依赖关系
3. **梯度消失/爆炸**：深层RNN在反向传播时容易出现梯度问题

2.2 Transformer的革命性

2017年，Vaswani等人提出了Transformer架构，彻底改变了序列建模的方式：

- **自注意力机制**：允许模型直接关注序列中的任意位置，无需顺序计算
- **并行化训练**：所有位置可以同时计算，大幅提升训练效率
- **长距离依赖**：通过注意力机制直接建模任意距离的依赖关系
- **可扩展性**：为后续GPT、BERT等大语言模型奠定了基础

2.3 作业目的与贡献

本作业的目标是：

1. **深入理解Transformer**：通过从零实现，深入理解每个组件的原理和作用
2. **验证架构有效性**：在小规模数据集上训练，验证Decoder-Only Transformer的有效性
3. **消融实验**：通过系统化的消融实验，理解各个组件对模型性能的贡献

主要贡献：

- 从零实现了完整的Decoder-Only Transformer架构
- 在Tiny Shakespeare数据集上成功训练并验证了模型
- 通过消融实验（不同注意力头数）分析了模型性能
- 提供了完整的代码实现和详细的实验报告

---

3. 相关工作 (Related Work)

3.1 原始Transformer

Vaswani等人（2017）在论文“Attention is All You Need”中首次提出了Transformer架构。该架构采用Encoder-Decoder结构，主要用于机器翻译任务：

- **Encoder**：将输入序列编码为连续表示
- **Decoder**：基于编码表示和已生成的部分，自回归地生成输出序列
- **核心创新**：完全基于注意力机制，摒弃了循环和卷积结构

3.2 GPT系列：Decoder-Only架构

Radford等人（2018-2019）提出了GPT（Generative Pre-trained Transformer）系列模型，采用Decoder-Only架构：

- **架构特点**：只使用Decoder部分，通过Masked Self-Attention实现自回归语言建模
- **优势**：结构更简单，训练更高效，特别适合语言建模任务
- **应用**：GPT-1、GPT-2、GPT-3等模型都采用此架构

3.3 位置编码的变体

位置编码有多种实现方式：

- **绝对位置编码**（本项目使用）：使用正弦和余弦函数编码位置信息
- **相对位置编码**：编码相对位置关系，如Transformer-XL
- **可学习位置编码**：将位置编码作为可训练参数

3.4 相关改进

后续研究提出了多种改进：

- **Layer Normalization变体**：Pre-LN vs Post-LN
- **注意力机制优化**：Sparse Attention、Linear Attention
- **训练技巧**：Gradient Checkpointing、Mixed Precision Training

---

4. 模型架构与数学推导 (Model Architecture)

本章节详细介绍Decoder-Only Transformer的每个组件及其数学原理。

4.1 Scaled Dot-Product Attention

缩放点积注意力是Transformer的核心组件，其数学公式为：

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

符号说明：

- $Q \in \mathbb{R}^{n \times d_k}$：Query矩阵，$n$为序列长度，$d_k$为Key的维度
- $K \in \mathbb{R}^{m \times d_k}$：Key矩阵
- $V \in \mathbb{R}^{m \times d_v}$：Value矩阵，$d_v$为Value的维度
- $\sqrt{d_k}$：缩放因子，防止点积值过大导致softmax梯度消失

计算步骤：

1. 计算QK^T，得到注意力分数矩阵
2. 除以$\sqrt{d_k}$进行缩放
3. 应用softmax归一化，得到注意力权重
4. 与V相乘，得到加权后的输出

为什么需要缩放?

当d_k较大时，点积QK^T的值会很大，导致softmax进入饱和区，梯度接近0。除以$\sqrt{d_k}$可以保持梯度的稳定性。

4.2 Multi-Head Attention

多头注意力机制将输入投影到多个子空间，并行计算多个注意力头，然后拼接：

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

其中每个头定义为：

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

参数说明：

- $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$：第$i$个头的Query投影矩阵
- $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$：第$i$个头的Key投影矩阵
- $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$：第$i$个头的Value投影矩阵
- $W^O \in \mathbb{R}^{hd_v \times d_{model}}$：输出投影矩阵
- h：注意力头数

维度关系：

- 通常设置$d_k = d_v = d_{model}/h$，保持参数量不变
- 本实验中：$d_{model} = 256$，$h = 4$，因此$d_k = d_v = 64$

为什么需要多头?

不同的头可以学习不同的注意力模式：

- 一个头可能关注语法关系
- 另一个头可能关注语义关系
- 还有头可能关注长距离依赖

  通过多个头的组合，模型能够捕获更丰富的特征表示。

4.3 Position-wise Feed-Forward Network

逐位置前馈网络是一个两层全连接网络，独立应用于序列中的每个位置：

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

参数说明：

- $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$：第一层权重矩阵
- $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$：第二层权重矩阵
- b_1, b_2：偏置向量
- d_{ff}：隐藏层维度，通常$d_{ff} = 4 \times d_{model}$

特点：

- **逐位置**：每个位置独立计算，不依赖其他位置
- **非线性**：ReLU激活函数引入非线性
- **扩展表示空间**：将d_{model}维扩展到d_{ff}维，再压缩回d_{model}维

4.4 Residual Connection and Layer Normalization

残差连接和层归一化是稳定训练的关键技术：

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

残差连接的作用：

1. **缓解梯度消失**：提供梯度直接传播的路径
2. **加速收敛**：允许模型学习残差（增量），而不是完整的映射
3. **稳定训练**：即使某些层学习失败，输入仍能通过残差连接传递

Layer Normalization：

对每个样本的特征维度进行归一化：

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

其中：

- $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$：均值
- $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$：方差
- γ, β：可学习的缩放和偏移参数
- ϵ：防止除零的小常数

4.5 Positional Encoding

由于Transformer没有循环或卷积结构，需要显式地编码位置信息。本实验使用正弦位置编码：

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

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

其中：

- pos：位置索引（0, 1, 2, ...）
- i：维度索引（0, 1, 2, ..., $d_{model}/2$）
- d_{model}：模型维度

特点：

- **确定性**：位置编码是固定的，不需要训练
- **相对位置**：可以编码相对位置关系（通过三角函数性质）
- **外推性**：可以处理比训练时更长的序列（虽然性能可能下降）

4.6 Causal Mask（因果掩码）

在Decoder-Only架构中，需要确保模型在预测第t个位置时，只能看到前$t-1$个位置的信息。这通过因果掩码实现：

$$\text{mask}[i, j] = \begin{cases} 1 & \text{if } j \leq i \\ 0 & \text{if } j > i \end{cases}$$

实现方式：

使用下三角矩阵，将未来位置的注意力分数设为$-\infty$（softmax后变为0）：

```
mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1)
mask = mask == 0  # 下三角为True，上三角为False
scores = scores.masked_fill(mask == 0, -1e9)
```

这确保了自回归特性：每个位置只能attend到它之前的位置。

4.7 完整架构

Decoder-Only Transformer的完整架构如下：

```
Input Tokens
    ↓
Token Embedding + Positional Encoding
    ↓
┌─────────────────────────────────┐
│  Decoder Block 1                │
│  ├── Masked Self-Attention      │  ← Causal Mask
│  ├── Residual + LayerNorm       │
│  ├── Feed-Forward Network       │
│  └── Residual + LayerNorm       │
└─────────────────────────────────┘
    ↓
┌─────────────────────────────────┐
│  Decoder Block 2                │
│  ...                            │
└─────────────────────────────────┘
    ↓
    ... (重复n_layers次)
    ↓
Final LayerNorm
    ↓
Linear Projection → Vocabulary
    ↓
Output Logits
```

5. 实现细节 (Implementation Details)

5.1 框架和语言

- **编程语言**：Python 3.8+
- **深度学习框架**：PyTorch 2.0+
- **其他依赖**：NumPy, Matplotlib, PyYAML, tqdm

5.2 关键实现片段

5.2.1 Scaled Dot-Product Attention

```
def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    缩放点积注意力实现

    Args:
        Q: Query矩阵 [batch, seq_len, d_k]
        K: Key矩阵 [batch, seq_len, d_k]
        V: Value矩阵 [batch, seq_len, d_v]
        mask: 掩码矩阵 [seq_len, seq_len]

    Returns:
        output: 注意力输出 [batch, seq_len, d_v]
        attn_weights: 注意力权重 [batch, seq_len, seq_len]
    """
    d_k = Q.size(-1)

    # 计算注意力分数
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)

    # 应用掩码（如果是causal mask）
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    # Softmax归一化
    attn_weights = F.softmax(scores, dim=-1)
    attn_weights = self.dropout(attn_weights)

    # 加权求和
    output = torch.matmul(attn_weights, V)
```

```
    return output, attn_weights
```

5.2.2 Multi-Head Attention

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads, dropout=0.1):
        super().__init__()
        assert d_model % n_heads == 0

        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        # 投影矩阵
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

        self.dropout = nn.Dropout(dropout)

    def forward(self, Q, K, V, mask=None):
        batch_size = Q.size(0)

        # 线性投影并重塑为多头
        Q = self.W_q(Q).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)
        K = self.W_k(K).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)
        V = self.W_v(V).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)

        # 计算注意力
        attn_output, attn_weights = scaled_dot_product_attention(Q, K, V, mask)

        # 拼接多头
        attn_output = attn_output.transpose(1, 2).contiguous().view(
            batch_size, -1, self.d_model
        )

        # 输出投影
        output = self.W_o(attn_output)

        return output
```

5.2.3 Decoder Block

```
class DecoderBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, n_heads, dropout)
        self.feed_forward = PositionwiseFFN(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        # Self-Attention + Residual + LayerNorm
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))

        # FFN + Residual + LayerNorm
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))

        return x
```

5.3 模型配置

本实验使用的模型配置如下：

参数	值	说明
词汇表大小	65	字符级，65个唯一字符
嵌入维度 (d_model)	256	模型的主要维度
注意力头数 (n_heads)	4	多头注意力的头数
FFN隐藏层 (d_ff)	1024	前馈网络的隐藏层维度
Decoder层数 (n_layers)	4	Transformer层数
序列长度 (seq_len)	128	输入序列的最大长度
Dropout	0.1	Dropout率
总参数量	**3,192,897**	约3.2M参数

5.4 训练技巧

- **优化器**：AdamW (β_1=0.9, β_2=0.98, weight_decay=0.01)

- **学习率调度**：Cosine Annealing (3e-4 → 1e-6)
- **梯度裁剪**：max_norm=1.0
- **权重初始化**：Normal(mean=0, std=0.02)
- **Dropout**：0.1（训练时），0.0（推理时）

---

6. 实验设置 (Experimental Setup)

6.1 数据集

Tiny Shakespeare：

- **大小**：1,115,394个字符
- **词汇表**：65个唯一字符
- **划分**：90% 训练集 (1,003,854字符) / 10% 验证集 (111,540字符)
- **任务**：字符级语言建模（预测下一个字符）

6.2 训练配置

超参数	值	说明
Batch Size	64	每批样本数
Sequence Length	128	序列长度
Learning Rate	3e-4	初始学习率
Min Learning Rate	1e-6	最小学习率
Epochs	30	训练轮数
Weight Decay	0.01	权重衰减
Max Grad Norm	1.0	梯度裁剪阈值
Optimizer	AdamW	优化器类型
Scheduler	Cosine	学习率调度策略

6.3 评估指标

- **交叉熵损失 (Loss)**：模型预测的负对数似然，越低越好
- **困惑度 (Perplexity)**：$\mathrm{PPL} = \exp(\mathrm{Loss})$，衡量模型的不确定性
- **生成质量**：定性评估生成的莎士比亚风格文本

6.4 硬件环境

- **CPU/GPU**：英伟达5070ti（WSL2环境）
- **训练时间**：
  - 4 heads模型：140.13分钟（约2.3小时）
  - 2 heads模型：112.95分钟（约1.9小时）
- **内存**：约16GB RAM

---

7. 结果与分析 (Results and Analysis)

7.1 训练曲线

训练过程中，损失和困惑度均平稳下降，学习率按余弦调度逐渐减小。训练曲线图已保存在`results/training_curves.png`。

观察：

- 训练损失和验证损失都持续下降
- 存在一定的过拟合（验证损失高于训练损失）
- 学习率调度有效，模型能够持续学习

7.2 定量结果

7.2.1 基础实验（4 heads）

指标	训练集	验证集
损失	0.8500	1.7953
困惑度	2.34	6.02
最佳验证损失	-	1.4904

7.2.2 消融实验（2 heads）

指标	训练集	验证集
损失	0.9058	1.7330
困惑度	2.47	5.66
最佳验证损失	-	1.4801

7.2.3 对比分析

指标	4 heads	2 heads	差异
训练损失	0.8500	0.9058	+6.6% ↑
验证损失	1.7953	1.7330	-3.5% ↓
验证困惑度	6.02	5.66	-6.0% ↓
过拟合程度	2.12倍	1.90倍	-10.4% ↓

7.3 消融实验结果分析

关键发现：2 heads模型的验证性能**更好**（反直觉）

可能原因：

1. **过拟合问题**：
   - 4 heads模型训练损失很低（0.85），但验证损失较高（1.80）
   - 过拟合程度：4 heads (2.12倍) > 2 heads (1.90倍)
   - 说明4 heads模型容量相对于数据规模过大
2. **隐式正则化**：
   - 减少注意力头数起到了类似正则化的效果
   - 2 heads模型通过减少容量提高了泛化能力
3. **数据集规模限制**：
   - Tiny Shakespeare只有~1MB数据
   - 对于3.2M参数的模型，数据可能不足以充分利用多头注意力的优势
   - 在更大数据集上，4 heads的优势可能会更明显

结论：

- 在小数据集上，2 heads模型通过隐式正则化获得更好泛化
- 这**不意味着**多头机制无效，而是说明数据规模的重要性
- 建议在更大数据集上验证多头注意力的优势

7.4 生成样本

7.4.1 4 heads模型生成样本

样本1：

```
ROMEO:
There is this the duke like to their scopes;
Thou hast charged thy babes and but by the land;
For once I should vex him as I thevelith garaboush...
```

样本2:

```
ROMEO:
Nay, look on him, by the pride, and to yourself:
Never man will not shame a scrown weight of the city
of kings, which of your igranoursiesiesthande...
```

7.4.2 2 heads模型生成样本

样本1:

```
ROMEO:
Madam, which art thou, anon.
Take the tempt thousand prepare to my blood?

JULIET:
If I could not say so soon, now I do with fevithithit!
```

样本2:

```
ROMEO:
My noble lord, that mistress have merry soul,
To the grace of them are not the matience of
the root, and thou dost not so tritetenat...
```

7.4.3 生成质量分析

共同特点:

- ✅ 都能生成莎士比亚风格的对话格式（角色名 + “:”）
- ✅ 能生成基本的英语单词和短语
- ✅ 语法结构基本正确

- ⚠ 后期会出现乱码和重复字符（模型规模限制）

差异观察：

- 2 heads模型在某些样本中对话结构更清晰（如样本1中的JULIET对话）
- 4 heads模型在某些样本中语法更复杂
- 两者在长文本生成时都会出现退化

---

8. 可复现性 (Reproducibility)

8.1 代码仓库

项目代码已开源，GitHub链接：https://github.com/897599359/bjtu-1

8.2 环境配置

```
# 创建conda环境（可选）
conda create -n transformer python=3.10
conda activate transformer

# 安装依赖
pip install -r requirements.txt
```

依赖包：

- torch >= 2.0.0
- numpy >= 1.24.0
- matplotlib >= 3.7.0
- pyyaml >= 6.0
- tqdm >= 4.65.0
- requests >= 2.31.0

8.3 重现命令

基础训练实验

```
# 使用4 heads配置训练
python src/train.py --config configs/base.yaml --seed 42
```

预期输出：

- 训练时间：约140分钟（CPU）
- 最佳验证损失：~1.49
- 模型保存至：`checkpoints/best_model.pt`

消融实验

```
# 使用2 heads配置训练
python src/train.py --config configs/ablation_2heads.yaml --seed 42
```

预期输出：

- 训练时间：约113分钟（CPU）
- 最佳验证损失：~1.48
- 模型保存至：`checkpoints_2heads/best_model.pt`

文本生成

```
# 使用4 heads模型生成文本
python src/evaluate.py \
    --checkpoint checkpoints/best_model.pt \
    --prompt "ROMEO:" \
    --num_samples 3 \
    --max_len 300

# 使用2 heads模型生成文本
python src/evaluate.py \
    --checkpoint checkpoints_2heads/best_model.pt \
    --prompt "ROMEO:" \
    --num_samples 3 \
    --max_len 300
```

8.4 硬件要求

- **最低配置**：CPU，8GB RAM
- **推荐配置**：NVIDIA GPU (2GB+ VRAM)，16GB RAM

---

9. 结论与未来工作 (Conclusion and Future Work)

9.1 总结

本作业成功实现了Decoder-Only Transformer架构，并在Tiny Shakespeare数据集上完成了训练和消融实验。主要成果包括：

1. **完整实现**：从零实现了所有核心组件（Multi-Head Attention、FFN、LayerNorm、位置编码等）
2. **成功训练**：模型在字符级语言建模任务上成功训练，验证困惑度达到5.66-6.02
3. **消融实验**：通过对比4 heads和2 heads模型，深入理解了多头注意力的作用
4. **科学发现**：在小数据集上，减少头数可能通过隐式正则化获得更好泛化性能

9.2 收获与理解

通过本次实验，我们深入理解了：

- **自注意力机制**：如何通过Query、Key、Value计算注意力权重
- **多头注意力**：多个头如何并行学习不同的注意力模式
- **位置编码**：如何显式编码位置信息
- **训练技巧**：学习率调度、梯度裁剪等对训练稳定性的重要性
- **模型容量与数据匹配**：过拟合问题及其解决方案

9.3 未来工作

可能的扩展方向：

1. **更大数据集**：在WikiText-103或OpenWebText上训练，验证多头机制在充足数据下的优势
2. **相对位置编码**：实现Transformer-XL的相对位置编码
3. **注意力优化**：实现Sparse Attention或Linear Attention，提升计算效率
4. **模型规模扩展**：尝试更大的模型（更多层、更大维度）
5. **并行化优化**：实现梯度检查点、混合精度训练等
6. **系统化消融**：对比1, 2, 4, 8 heads的性能曲线，找到最优头数
7. **其他组件消融**：移除位置编码、残差连接等，分析各组件的作用

参考文献

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017).

Attention is all you need. ***Advances in neural information processing systems*****, 30.**