大模型基础与应用中期作业报告

Student 刘子赫
Student ID 25120399

1 引言

在 Transformer 提出之前，主流的序列建模方法主要依赖 RNN 及其变体（如 LSTM、GRU 等），这类模型能够捕获时间顺序信息，但是随着序列窗口大小的增加出现了问题，一方面长程依赖使得模型在处理较长序列时，对早期信息的记忆逐渐衰减，难以捕捉远距离词语之间的关联；另一方面，信息交互复杂，序列中两个位置的特征必须经过多层网络传播才能相互影响，导致训练效率低下且信息传递不稳定。

为了解决这两个问题，Transformer 提出了自注意力机制，允许序列中每个位置的表示直接与所有其他位置进行信息交互，从而显式建模全局依赖关系。在具体实现上，模型通过为每个词构建查询向量、键向量和值向量，计算它们之间的点积相似度来衡量不同词语间的相关性。这样，每个词的最终表示都能根据注意力权重聚合全局信息，实现对相关上下文的额外的关注度，从而有效克服了长程依赖问题并显著提升了序列建模能力。

本次作业的目的在于从头实现 transformer 的架构，尤其是多头注意力机制和网络结构的数学和代码实现。这对以以后对于基于 transformer 的大模型进行在结构上微调工作有所帮助，更好的找到模型的提升空间以及对应改进的结构点。同时开展消融实验探究 transformer 的相关机制对于精度的影响和作用.

2 相关工作介绍

Transformer 架构以编码器–解码器堆叠为核心，每层由多头自注意力机制与位置前馈网络构成，并辅以残差连接与层归一化以稳定训练、提升表达能力。解码器中进一步引入交叉注意力以融合编码器信息，并通过因果掩码确保自回归生成的时序约束。区别与其他神经网络，Transformer 架构完全基于注意力机制，摒弃循环与卷积结构，实现了高度并行化计算，显著提升了训练效率。为弥补模型对输入顺序不敏感的缺陷，其通过位置编码（正弦函数）或可学习嵌入——显式注入 token 的绝对与相对位置信息，从而有效建模长程依赖。得益于其模块化与可扩展性，Transformer 可通过增加层数、扩展维度或引入稀疏/局部注意力等变体，灵活适配不同任务规模与资源约束。

而改进方向主要有稀疏化、低秩/投影、核方法线性化，以及推理路径优化。稀疏化（Sparsification）方法通过限制每个 token 仅关注局部或特定模式下的子集，从而在保持模型表达能力的前提下，将注意力复杂度从 O(n2) 降至接近线性。例如 Sparse Transformer、Longformer 和 BigBird 等模型通过规则化或随机稀疏连接实现对长序列的高效建模。低秩/投影（Low-rank/Projection）方法假设注意力矩阵在高维空间中近似低秩，通过线性投影或分解技术减少计算量与存储开销，如 Linformer 提出了对键值矩阵 K，V 进行长度维度投影，从而在保持性能的同时实现线性复杂度。核方法线性化（Kernelization/Linear Attention）将 softmax 注意力解释为核函数映射，通过特征变换将二次复杂度降为线性。典型代表如 Performer（FAVOR+）使用随机特征近似 softmax 核，Linear Transformer 则通过显式核分解实现可扩展的线性注意力。推理路径优化（Inference Path Optimization）则针对自回归解码过程，通过结构重用与缓存压缩降低时延。例如使用功能 KV-cache 共享或分组键值矩阵，显著减少 KV 缓存带宽与显存占用，从而实现更高效的长文本生成。

3 模型结构和数学推导

3.1 单头注意力机制

公式如下:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

在单头注意力当中 Q 和 K 分别包含了所有的查询向量和键向量，即嵌入向量与查询矩阵和键矩阵相乘得到的小向量，因此 Q 和 K 矩阵的乘积即是键值对形成的网格，转置的效果是为了进行矩阵乘法进行维度匹配。// 而得到查询键值网格后将矩阵当中的值除以维度数量是为了数据的稳定性进行的缩放操作，当 dk 很大时，点积的数值会变得很大，而这个操作将输入的方差统一到回到大约 1 的量级，防止 softmax 饱和，使训练更加稳定。
在缩放后逐列的进行 softmax 操作得到 query 对 key 的关注程度。而在公式之外，对于值矩阵逐列求和后即是每个嵌入向量的变化量，与原本的嵌入向量相加，即得到了最终更加精准的嵌入向量。

3.2 多头注意力

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O \tag{2}$$

$$\text{where} \quad \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3}$$

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

多头注意力的机制重点在于不只计算一次注意力，而是并行计算多次，每次使用不同的线性变换（即不同的子空间），最后再拼接结果并映射回原维度。

在多头注意力当引入了具体计算时的细节，通过可训练参数的 WQ，WK，WV 和 WO 进行权重更新，但在值矩阵的实现有一些改变，将值权重进行低秩分解，分为了 2 个矩阵（由单头注意力的单一的 WV 进行分解得到了 WV，WO）并在不同阶段进行相乘，这样的操作减少了空间的占用和计算成本。其中 WV 用于将大嵌入向量映射到小向量空间,WO 用于将拼接后的多头注意力输出重新映射回原始特征空间。

而对于不同头的拼接，在同一位置（同一个 token）把各头的 dh 维向量并排接起来，形成一个 dmodel 维的向量，再使用 WO 进行重新映射，保证与残差块的维度一致

对于多头注意力模型把输入向量投影到多个子空间，每个子空间的注意力可以独立学习不同的表示，最终通过拼接和线性变换融合成整体表示。这样的操作使得模型可以关注不同的重点和上下文，同时综合考虑上下文的影响。进而提高能力。

3.3 位置感知前馈网络

$$\text{FFN}(x) = W_2\,\phi(W_1 x + b_1) + b_2 \tag{5}$$

它对每个位置的表示独立、同参数地应用一个两层 MLP 计算最终的输出结果，而不是像其他的神经网络一样将输出展平进行统一的 MLP 输出操作，，对于每一个 token 训练一个参数，因此也被称为位置感知前馈网络。

3.4 残差和层归一化

Discuss the role of residual connections and normalization in stabilizing training.

$$\text{Output} = x + \text{Dropout}\big(\text{FFN}(\text{LayerNorm}(x))\big) \tag{6}$$

增加残差之后，可以让输入直接连接到新的输出上，这样提供了一个额外的梯度传播的路径，在 transformer 较深的网络中减少梯度爆炸可能性。

$$\text{LN}(x) = \frac{x-\mu}{\sigma} \cdot \gamma + \beta \tag{7}$$

而层归一化在每个样本内部进行归一化操作，不依赖于 batch 维度。这样的设计思想与位置感知前馈网络和多头注意力机制相一致，都是针对每个样本独立地进行特征变换与计算，最终再通过样本在序列中的位置实现整体的统一与融合。

3.5 位置编码

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{8}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{9}$$

$$PE_{pos} = W_{pos} \tag{10}$$

对于 transformer 来说其本身基于自注意力截止，不具备序列顺序信息，因此需要加上位置信息进行编码，显式地加入位置信息。

正弦位置编码以固定函数形式提供连续位置表示，无需训练参数，并具备良好的序列长度扩展性。

4 代码实现细节

4.1 框架和语言

使用 pytorch2.1.2+cu121 版本，在 python3.10 上运行

4.2 关键代码实现

4.2.1 注意力机制

为了便于理解，首先先实现了单头注意力机制，其中包含了三个线性变换层：WQ, WK, WV 分别将输入投影到查询 (Q)、键 (K)、值 (V) 空间，再通过 Q × KT 计算相似度得分，然后除以 √d_k 进行缩放，然后用 Softmax 归一化，将得分转换为概率分布值加权求和，最后用注意力权重对 V 进行加权求和得到最终输出

```python
class Attention(nn.Module):
    def __init__(self, dimensional, head_dim):
        super(Attention, self).__init__()

        self.WQ = nn.Linear(dimensional, head_dim, bias=False)
        self.WK = nn.Linear(dimensional, head_dim, bias=False)
        self.WV = nn.Linear(dimensional, head_dim, bias=False)

        for module in [self.WQ, self.WK, self.WV]:
```

```
            nn.init.xavier_uniform_(module.weight)

    def forward(self, Q, K, V, mask=None):
        # 上投影
        Q = self.WQ(Q)  # [batch_size, seq_len, head_dim]
        K = self.WK(K)
        V = self.WV(V)

        # 计算注意力得分并缩放
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(Q.size(-1))

        # 掩码：避免关注到填充位置
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        attention_weights = F.softmax(scores, dim=-1)
        context = torch.matmul(attention_weights, V)

        return context, attention_weights
```

Listing 1: **单头注意力机制实现**

而多头注意力会创建多个单头注意力实例并行处理，同时计算头的维度使得总维度不变 head_dim = dimensional // num_heads **确保总维度不变，随后将各头输出在最后一维拼接，最终用** WO **将拼接结果映射回原始维度**

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dimensional, num_heads):
        super(MultiHeadAttention, self).__init__()
        self.dimensional = dimensional
        self.num_heads = num_heads
        self.head_dim = dimensional // num_heads

        self.heads = nn.ModuleList([
            Attention(dimensional, self.head_dim)
            for _ in range(num_heads)
        ])

        self.WO = nn.Linear(dimensional, dimensional, bias=False)
        nn.init.xavier_uniform_(self.WO.weight)

    def forward(self, Q, K, V, mask=None):
        head_outputs, attn_list = [], []

        for head in self.heads:
            context, attn = head(Q, K, V, mask)
            head_outputs.append(context)
            attn_list.append(attn)

        context = torch.cat(head_outputs, dim=-1)
        output = self.WO(context)

        return output, attn_list
```

Listing 2: **多头注意力机制实现**

4.2.2 Encoder / Decoder **模块**

Encoder **由自注意力层、前馈层以及归一化层组成，每个位置的前馈网络独立计算。**

```python
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.linear2(self.dropout(F.relu(self.linear1(x))))
```

Listing 3: 前馈层实现

```python
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.ffn = FeedForward(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=None):
        attn_output, attn_weights = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))

        ffn_output = self.ffn(x)
        x = self.norm2(x + self.dropout(ffn_output))

        return x, attn_weights
```

Listing 4: Encoder 层实现

以 Encoder 为例，其中包括了前馈层，网络结构和归一化实现，实现了残差链接，使用 LayerNorm 对于当前的位置进行层归一化

```python
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.ffn = FeedForward(d_model, d_ff, dropout)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_output, tgt_mask=None, memory_mask=None):
        # Masked Self-Attention
        self_attn_out, _ = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(self_attn_out))

        # Cross-Attention: Q来自decoder, K/V来自encoder
        cross_attn_out, attn_weights = self.cross_attn(
            x, enc_output, enc_output, memory_mask
        )
        x = self.norm2(x + self.dropout(cross_attn_out))

        # 前馈层
        ffn_out = self.ffn(x)
        x = self.norm3(x + self.dropout(ffn_out))

```

```
        return x, attn_weights
```

Listing 5: Decoder 层实现

其实 decoder 的结构与 Encoder 非常类似，但是对于 QKV 的来源并不相通，这也是交叉注意力的来源，使用其他解码器的查询向量输入到当中 encoder 的嵌入向量当中做解码

4.2.3 掩码函数

掩码函数用于防止模型在训练时“偷看”未来的序列位置。

```
def generate_square_subsequent_mask(sz):
    mask = torch.triu(torch.ones(sz, sz), diagonal=1)
    mask = mask.masked_fill(mask == 1, float('-inf'))
    return mask
```

Listing 6: 掩码函数实现

该函数生成上三角矩阵，是为了防止模型偷看后方信息所添加的掩码，将当前位置信息之后的值转化为 0，且掩码当中赋值负无穷是为了防止在 softmax 为 0 的位置仍旧会被分配非零权重导致模型仍旧看到了信息，赋值负无穷会使得后面的位置在 softmax 为 0，真正的起到作用。

5 实验

5.1 数据集

使用 CNNDailyMail 数据集当中的部分数据。由于算力限制和时间成本考虑，根据 GuggingFace 上的数据集统计，选择原始文本 token 数小于 1640 的文本（约占 7%，约 20000 条训练数据左右）。为保证随机性和减少计算量，在此基础上再随机选择 50% 的数据作为训练数据集。最终数据集中训练集共计条数据验证集共计条数据

5.2 数据处理

根据 transformer 原文，需要对于训练集进行 BPE Tokenization，因此需要自己维护一个语料库并且基于语料库进行词表的训练。根据使用的 CNNDailyMail 数据集其他参考训练的参数，词表大小定位 32000，属于中型大小语料库，保证新闻中出现的关键词和抽象词可以更好映射到向量空间.

其中 transformer 的要求使用 SentencePiece 通过数据集训练 BPE 模型，然后保存 Tokenizer 分词器用于训练

5.3 超参数

根据 transformer 原文，Embedding dimension 大致决定了头的数量和前馈层数量，其中 Number of heads = Embedding dimension/64，Feed-forward dimension = Embedding dimension * 4，batchsize 基于训练显卡不同（服务器）进行修改，但是保证在同一设备上 batchsize 相同（用本地电脑调试跑起来，用服务器进行跑最终模型），其他参数如下（服务器参数）：

表 1: 超参数设置

Parameter	Value
Embedding dimension	256
Number of heads	4
Feed-forward dimension	1024
Number of layers	2
Batch size	32
Learning rate	1e-4
Optimizer	Adam

6 实验结果分析

6.1 训练曲线

使用以上参数训练 300 轮得到的损失曲线如下
其中 50 轮训练损失为：0.021765843 验证损失为：0.226585318
最终训练损失为：0.006513454，验证损失为：0.177637016
从训练函数上看，训练损失和验证损失在初期快速收敛，在 50 轮左右逐渐收敛，为保证效果在初步实验

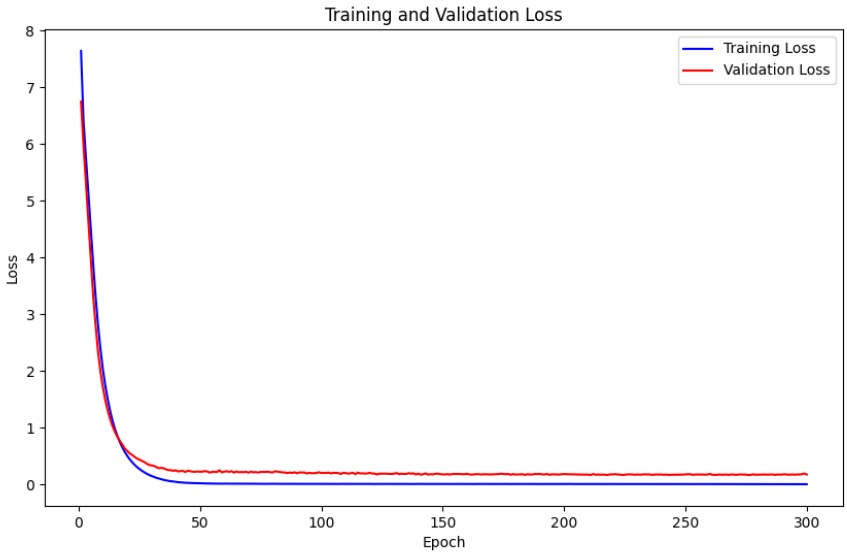

图 1: 初步 300 实验结果

当中设置了较多轮次，在 50 轮左右损失已经没有变化，后续参数实验和消融实验会以 50 轮为标准节省算力和时间。

6.2 实例输出

输入：Police and FBI agents are investigating the discovery of an empty rocket launcher tube on the front lawn of a Jersey City, New Jersey, home, FBI spokesman Sean Quinn said. Niranjan Desai discovered the 20-year-old AT4 anti-tank rocket launcher tube, a one-time-use device, lying on her lawn Friday morning, police said.

输出：ed f a w o

从输出来看还是没有形成完整的词句，仅是词表当中分词的 token，这可能既是因为数据集不够，同时模型相较于原始的 transformer 也进行了大幅的缩水和改动，在层数和和输入输出维度上均有的大幅度的缩小，同时新闻当中的专有名词和复杂语义也加剧了网络的困惑度

6.3 参数对比

对比一组缩小网络维度和层数的实验，使用 loss 进行对比，实验参数为：embedding dimension128，层数为 2，注意力头为 2，输出层数为 512。相较于之前的实验参数大幅减少参数量，在 4090 服务器上进行训练约 25 秒一轮

使用以上参数训练 50 轮得到的损失曲线如下，其中最终训练损失为：3.518277785621093，验证损失为：4.478560472789564 从实验结果来看仍旧有收敛的空间，但是在同等轮次的情况下，损失远大于大参数量的模型，说明参数量确实会极大程度的影响拟合效果。

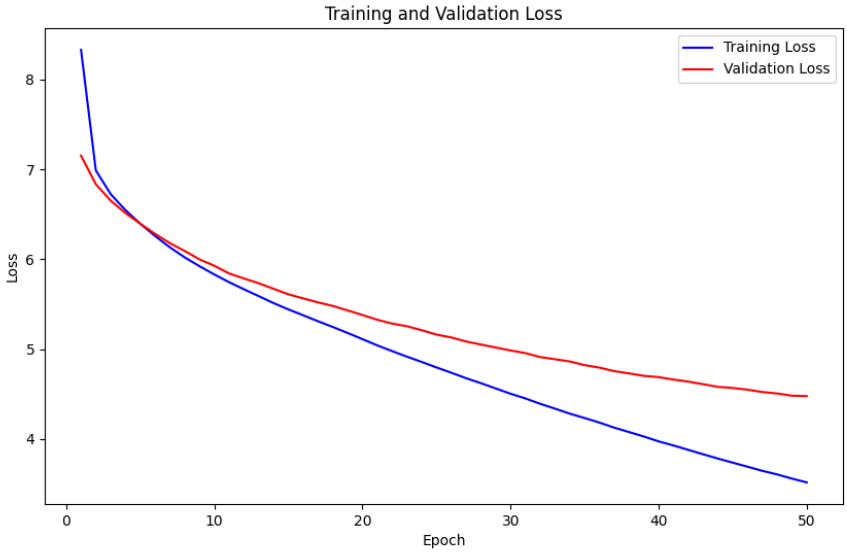

图 2: 小参数实验对比实验结果

6.4 消融实验

6.4.1 位置编码

将代码中的位置编码去掉进行一组消融实验，为保证一致性其他参数相同。实验结果如下

实验结果为 50 轮时训练损失：3.4391552213591，验证集损失：4.42082184239437 在参数一致的情况下训练损失和验证损失远大于正常有位置编码的损失，说明位置编码对于是 transformer 架构性能有极大帮助，在同参数情况下可以极大的改善训练效果，因此在之后微调 transformer 的实验中可以在这里进行调整

6.4.2 多头注意力

将实验参数中 num head 的改为即实现了单头注意力机制，为保证一致性其他参数相同。实验结果如下 实验结果为 50 轮时训练损失：0.018889540056829402，验证集损失：0.2260759700285761 在参数一致的情况下训练损失略低于多头，验证集损失略高于多头，在这个实验当中并没有明显表现出差异，这可能

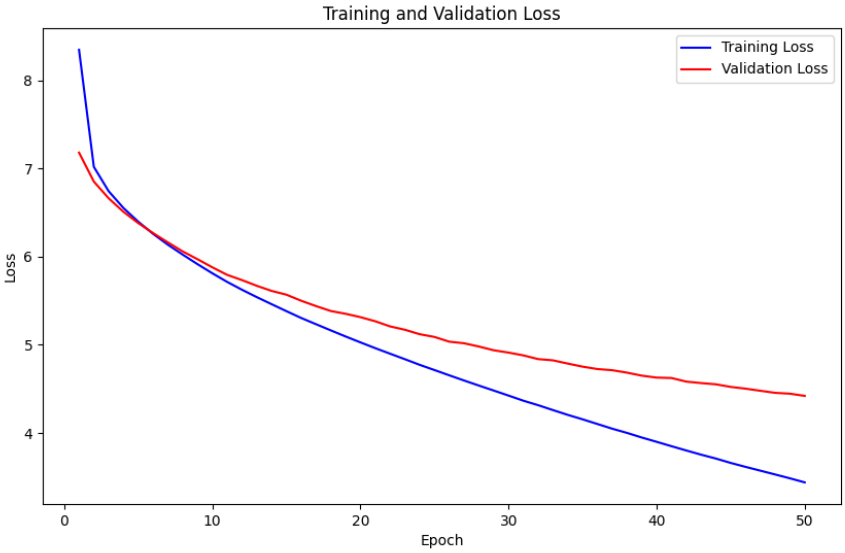

图 3: 位置编码消融实验

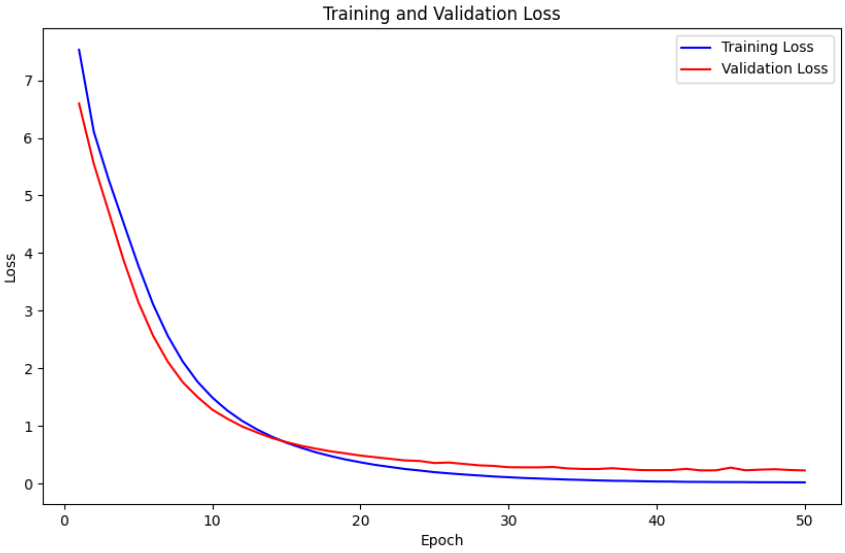

图 4: 单头消融实验

是因为层数较多，网络的复杂性弥补了单头的不足，同时数据集过于简单也可能是个因素，但是可以确定是单头的注意力机制的验证表现确实不如多头的情况

7 可重复性和代码提交

github 链接：https://github.com/lzhhe/M502082B_mid_term_experiement

7.1 文件结构

- README.md
- cnn_dailymail (数据集)
  - 3.0.0
    - test-00000-of-00001.parquet
    - train-00000-of-00003.parquet
    - train-00001-of-00003.parquet
    - train-00002-of-00003.parquet
    - validation-00000-of-00001.parquet
- requirements.txt (环境列表)
- scripts
  - run.sh (运行脚本)
- src
  - attention.py (注意力机制)
  - best_transformer.pth (生成的权重)
  - bpe_tokenization.py (tokenizer)
  - bpe_tokenizer.model (训练的词表模型)
  - bpe_tokenizer.vocab (词表)
  - corpus.txt (语料库)
  - dataset.py (处理数据集)
  - embeding.py (词嵌入)
  - encoder_decoder.py (编码解码器)
  - main.py (主程序)
  - test_sample_filtered.parquet
  - train_sample_filtered.parquet
  - transformer_model.py (transformer 模型)
  - validation_sample_filtered.parquet

7.2 实验花费时间和硬件需求

实验在本机上使用 3060ti 进行调试跑通，在 4090 服务器上对于超参数进行优化并进行训练在使用 NVIDIA 4090 上运行一轮时间在 batchsize 为 32 时约为 45 秒显存消耗约为 10-12Gb 左右，显卡占用约在 50% 左右

```
$ conda create -n transformer python=3.10
$ conda activate transformer
$ cd M502082B_mid_term_experiement
$ pip install -r requirements.txt
$ python -m src.main
```

Listing 7: 运行代码

8 实验反思与改进

8.1 代码部分部分

对于 torch 等相关函数使用不熟练；对于注意力实现不够高效，因为实际操作中会先将单头注意力创建好再进行多头注意力的计算；对于 encoder/decoder 一开始的理解有问题导致网络结构有所变动。

8.2 实验部分

对于实验结果，从训练损失和收敛情况来看，已经实现了 transformer 架构和相关训练代码，各个组件发挥了作用，但是最终的测试效果不佳，可能还是因为数据集和分词器有所欠缺；对于数据集的选择来说可能一方面是数据集过于复杂需要进行清洗和训练 tokenizer 等相关操作且最终的分词效果不佳，其实也可以尝试使用其他模型现有分词器；对于训练参数的选择，一开始实验的参数选取过大同时效果不佳，另一方面参数大小之前对于最终实验结果的影响也存在相互干扰；在训练过程中时间成本和算力消耗较大，一方面是 trasnformer 本身网络结构较大，另一方面是选择的数据集和参数也超过了本机显存和算力上限，只能另行租用服务器。；实验中手搓的相较于原生 pytorch 框架训练效率较低，同等条件下训练速度较慢，一方面是用了大量的 for 循环，无法真正的实现并行操作，另一方面也是缺少了底层代码的加速导致速度更慢。

参考文献

[1] Attention in transformers, step-by-step | deep learning chapter 6. https://www.youtube.com/watch?v=eMlx5fFNoYc. Accessed: 2025-11-08.

[2] Transformers, the tech behind llms | deep learning chapter 5. https://www.youtube.com/watch?v=wjZofJX0v4M. Accessed: 2025-11-08.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.