

Transformer 模型实验指南

项目概述

本项目实现了 Transformer 模型的训练、测试和对比实验，聚焦于不同位置编码、分词器及模型结构对翻译任务性能的影响，核心功能涵盖以下四类模型：

- 原始 Transformer 模型
- 带有 RoPE（旋转位置编码）的 Transformer
- 带有 SentencePiece 分词的 Transformer
- Mamba 模型（简化实现）

环境准备

依赖安装

执行以下命令安装项目所需核心依赖包：

```
bash
pip install torch numpy sentencepiece
```

数据集准备

项目采用 WMT14 德英翻译数据集作为实验数据。该数据集需经过语言过滤、长度筛选等预处理步骤以保证数据质量，运行以下命令即可自动完成数据集下载、预处理及增强：

```
bash
python download_dataset.py
```

脚本执行完成后，将生成包含 100 条训练样本和 10 条测试样本的增强数据集，可直接用于后续模型训练。

实验配置

模型配置

四类实验模型的核心配置差异如下表所示，其中参数数量差异主要源于分词器词汇表大小及模型结构简化程度：

模型配置	位置编码	分词器	参数数量
原始 Transformer	正弦位置编码	BPE	44,847,053
Transformer with RoPE	旋转位置编码	BPE	44,847,053
Transformer with SentencePiece	正弦位置编码	SentencePiece	48,170,047
Mamba Transformer	可学习位置编码	基础分词	24,100,479

训练配置

所有模型采用统一的基础训练参数，具体设置如下：

- 训练轮次：5 轮
- 批次大小：8（适配 CPU 训练的显存/内存需求）
- 学习率：0.0001
- 优化器：Adam
- 损失函数：交叉熵损失（适用于分类式翻译任务）
- 设备：CPU（默认）

运行实验

基础训练

执行基础训练脚本，可指定训练设备（默认 CPU），脚本将完成单模型的训练与测试流程：

```
bash
python training_and_testing.py --device cpu
```

对比实验

运行对比实验脚本，将自动执行四类模型的训练与测试，并生成性能对比结果：

```
bash
python comparison_experiments.py
```

实验结果（含训练时间、最终损失等核心指标）将自动保存到 `experiment_results.json` 文件中，便于后续分析。

实验结果分析

性能对比

四类模型在相同训练配置下的性能表现如下表所示：

模型配置	训练时间	最终损失
原始 Transformer	26.30s	2.2492
Transformer with RoPE	26.96s	2.0151
Transformer with SentencePiece	26.39s	2.0201
Mamba Transformer	17.98s	4.8251

关键发现

- RoPE 位置编码表现最佳：**RoPE 相比原始正弦位置编码能够更好地捕捉序列的相对位置信息，使模型对长文本的语义理解更准确，因此最终损失最低，翻译效果更优。
- SentencePiece 分词效果优异：**SentencePiece 作为语言无关的子词分词器，相比 BPE 能更好地处理德语复合词等复杂词汇结构，有效降低了词汇表规模并提升了模型性能。
- Mamba 训练效率高但翻译性能有限：**Mamba 模型通过选择性状态空间机制简化了计算流程，参数数量仅为传统 Transformer 的一半左右，因此训练时间显著缩短；但由于其结构更适配通用序列建模，在针对性的翻译任务上表现不如专门设计的 Transformer 架构。

自定义实验

若需开展个性化实验，可通过以下方式修改配置或扩展模型：

修改训练参数

直接在 `training_and_testing.py` 脚本中调整核心训练参数，示例如下：

```
python
# 训练参数
BATCH_SIZE = 8 # 可根据设备内存调整，GPU 环境可增大至 16 或 32
EPOCHS = 5      # 模型不收敛时可增至 10-20 轮
LEARNING_RATE = 0.0001 # 可根据优化器类型调整，如 AdamW 可适当减小
```

添加新模型

在 `transformer_model.py` 中扩展新的位置编码或注意力机制，需继承 PyTorch 的 `nn.Module` 类，示例如下（以自定义位置编码为例）：

```
python
import torch
import torch.nn as nn

class NewPositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000, dropout=0.1):
        super().__init__()
        # 实现新的位置编码逻辑（示例：基于正弦余弦的改进编码）
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len,
dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
(-torch.log(torch.tensor(10000.0)) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # 前向传播逻辑：将位置编码注入输入嵌入
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)
```

修改实验配置

在 `comparison_experiments.py` 中添加自定义模型的配置信息，即可纳入对比实验，示例如下：

```
python
configurations = [
    {
        'name': 'Custom Model',
        'pos_encoding': 'new_pos_encoding', # 对应自定义的位置编码
        'tokenizer': 'sentencepiece',      # 可选用 BPE 或
        'model_path': 'custom_transformer.py' # 若新增模型文件需指定
    }
]
```

常见问题

内存不足

CPU/GPU 内存不足时，可通过以下方式解决：

- 减小批次大小（如从 8 调整为 4）
- 减小模型维度（`d_model`），降低每一层的特征维度
- 使用更小的数据集（如减少训练样本至 50 条）

训练速度慢

训练效率低时，可尝试以下优化方案：

- 使用 GPU 加速：安装 CUDA 后，将训练命令中的 `--device cpu` 改为 `--device cuda`，多 GPU 环境可通过 `CUDA_VISIBLE_DEVICES=0,1` 指定卡号
- 减小训练轮次（如从 5 轮改为 3 轮）
- 减小数据集大小，降低单轮训练的数据量

模型不收敛

若训练过程中损失不下降（模型不收敛），可通过以下方式调整：

- 调整学习率：如将 0.0001 调整为 0.00005 或 0.0002，避免学习率过高或过低
- 增加训练轮次：延长训练时间，让模型有足够的学习周期
- 调整模型参数：如增大注意力头数、增加网络层数等，提升模型拟合能力

项目结构

项目文件组织结构如下，各文件功能清晰划分：

```
bash
.
├── download_dataset.py      # 数据集下载与预处理脚本
├── training_and_testing.py  # 基础训练与测试脚本
├── transformer_model.py     # Transformer 模型核心实现（含位置编码）
├── mamba_model.py          # Mamba 模型简化实现
├── tokenizers.py           # BPE、SentencePiece 分词器实现
├── comparison_experiments.py # 多模型对比实验脚本
├── experiment_results.json  # 实验结果存储文件
└── EXPERIMENT_GUIDE.md     # 实验指南文档（本文档）
```

加分实验-模型量化训练（INT8 量化）

实验目标

通过 INT8 量化降低模型显存占用与推理延迟，验证量化对模型性能的影响，探索高效部署方案。

核心思路

将模型权重从 32 位浮点数（FP32）量化为 8 位整数（INT8），在训练或推理阶段引入量化/反量化操作，平衡模型性能与效率。采用 PyTorch 的 `torch.quantization` 工具包实现量化逻辑。

实现步骤

- 在 `transformer_model.py` 中新增量化模型封装类，继承原有 Transformer 模型，重写 `forward` 方法，添加量化/反量化操作；
- 配置量化参数：指定需要量化的层（如线性层、注意力层），设置量化校准数据集（建议使用 10% 训练数据）；
- 修改 `training_and_testing.py`，添加量化训练参数（`--quantize int8`），支持量化与非量化模型的对比训练；
- 评估指标：除原有训练时间、最终损失外，新增显存占用（使用 `torch.cuda.memory_allocated()` 统计）、推理延迟（统计单条样本推理耗时）。