实验报告

实验任务: 垃圾邮件分类 (二分类任务)

PB22050983 胡延伸

一、数据预处理流程

1. 原始数据清洗

- **缺失值处理**: 删除包含空值的样本
- **标签过滤**: 仅保留标签为 `ham` 和 `spam` 的样本
- **文本合并**: 将 `Subject` 和 `Message` 字段拼接为完整文本

2. 文本标准化

```
# 标准化步骤：
1. 小写化: text.lower()
2. 特殊字符过滤: re.sub(r'[^a-zA-Z\s]', '', text)
3. 停用词移除: 过滤英语停用词 (nltk.corpus.stopwords)
4. 词干提取: PorterStemmer().stem(word)
```

3. 特征工程

步骤	参数	作用
词表构建	max_vocab_size=20,000	保留高频词，包含 `<pad>` 和 `<unk>`
序列截断	max_seq_len=200	保留尾部200个token (左填充)
向量化	padding_idx=0	将文本映射为词索引序列

二、模型架构对比

1. MultiHeadAttention 模型 (Transformer-based)

核心组件:

```
class MultiHeadAttentionClassifier(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        d_model: int = 128,
        n_heads: int = 4,
        num_classes: int = 2,
        max_seq_len: int = 200,
    ):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=0)
        self.pos_encoder = PositionalEncoding(d_model, max_seq_len)
        # 自定义多头注意力
        self.attention = DIYMultiHeadAttention(
            embed_dim=d_model,
            num_heads=n_heads,
            dropout=0.1
        )
        # 分类器
        # nn.Sequential是一个容器模块，可以将多个层组合在一起
        self.classifier = nn.Sequential(
            nn.Linear(d_model, d_model * 2),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(d_model * 2, num_classes),
        )
```

2. RNN 模型

核心组件:

```python
class SimpleRNNClassifier(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        embed_dim: int = 128,
        hidden_dim: int = 256,
        num_layers: int = 1,
        num_classes: int = 2,
        dropout: float = 0.2
    ):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        # 嵌入层
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)

        # 手动实现RNN参数
        # 输入到隐藏的权重和偏置
        self.Wxh = nn.ParameterList([
            nn.Parameter(torch.Tensor(embed_dim if l == 0 else hidden_dim, hidden_dim))
            for l in range(num_layers)
        ])
        # 隐藏到隐藏的权重
        self.Whh = nn.ParameterList([
            nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
            for l in range(num_layers)
        ])
        self.bh = nn.ParameterList([
            nn.Parameter(torch.Tensor(hidden_dim))
            for l in range(num_layers)
        ])

        # 初始化参数
        for layer in range(num_layers):
            # 使用Kaiming初始化输入到隐藏的权重和偏置，使用正交初始化隐藏到隐藏的权重
```

```
            # Kaiming初始化适用于ReLU激活函数，正交初始化适用于tanh激活函数
            nn.init.kaiming_normal_(self.Wxh[layer], mode='fan_in', nonlinearity='tanh')
            nn.init.orthogonal_(self.Whh[layer])
            nn.init.zeros_(self.bh[layer])

        # 分类器
        self.classifier = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, num_classes)
        )
```

三、实验分析方向

1. 模型架构对比

指标	MHA	RNN	LSTM	关键分析
训练速度	55s/batch	150s/batch	160s/batch	**MHA的并行计算优势**：Transformer自注意力机制可并行 LSTM因时间步依赖存在计算瓶颈
准确率	97.61%	97.18%	**98.30%**	**LSTM的门控优势**：输入/遗忘/ 输出门有效过滤噪声，在中等长度序列任务中（如文本分
F1	97.63%	97.21%	**98.31%**	**LSTM的均衡能力**：对类别不平衡数据（如垃圾邮件检测
Recall	**98.10%**	97.41%	97.80%	**MHA的全局感知**：自注意力机制减少信息丢失，在漏检

模型参数量比较：

Model	Parameters	Size (MB)
MHA	2.66M	10.64MB
RNN	2.63M	10.51MB
LSTM	2.69M	10.77MB

2. 超参数影响

由于 MultiHeadAttentionClassifier 训练速度最快，采用该模型做对比实验。
具体实验方法很简单，调整 `max_seq_len` 和 `max_vocab_size` 这两个超参数大小，实验结果如下：
关键参数实验:

- **词表大小**: (控制序列长度为200)

比较	2w	5w	10w
训练速度	55s/batch	70s/batch	90s/batch
预测准确率	97.61%	97.18%	97.03%
F1	97.63%	97.22%	97.03%
recall	98.10%	97.83%	96.48%

- **序列长度**: (控制词表大小 2w)

比较	100	200	500
训练速度	20s/batch	55s/batch	300s/batch
预测准确率	96.97%	97.61%	97.47%
F1	96.99%	97.63%	97.01%
recall	96.84%	98.10%	97.11%

从表中可以看出，序列长度显著影响训练速度。适当增加序列长度有助于提高模型性能，但太高反而会降低训练速度，而且对模型性能提高没太大帮助。

3. 损失函数对比

二元交叉熵 (BCEWithLogitsLoss) vs 交叉熵 (CrossEntropyLoss):

输出设计	输出维度	适用场景	实验Acc
Sigmoid + BCE	1	二分类	98.35%

输出设计	输出维度	适用场景	实验**Acc**
Softmax + CE	2	多分类	97.61%

二元交叉熵性能更好，这是因为它为单目标，直接预测正类概率，梯度仅通过单节点反向传播；并且输出层数更少，鲁棒型更强。而交叉熵函数输出层数翻倍，过拟合风险增加了。
