实验报告

实验任务一

代码

```
# TODO 1: 计算 div_term，用于控制不同维度的 sin/cos 频率
        # 要求: 使用 torch.exp() 实现 1 / 10000^(2i/d_model)
        div_term = torch.exp(torch.mul(torch.arange(0,d_model,2),-
torch.log(torch.tensor(10000)/d_model)))

        # TODO 2: 给偶数维度位置编码赋值
        # 要求: 使用 torch.sin() 完成 position * div_term，赋值给 pe 的偶数列
        pe[:, 0::2] = torch.sin(position* div_term)

        # TODO 3: 给奇数维度位置编码赋值
        # 要求: 使用 torch.cos() 完成 position * div_term，赋值给 pe 的奇数列
        pe[:, 1::2] = torch.cos(position* div_term)
```

```
# TODO 1: 计算 attention scores
        # 要求: 使用缩放点积的方式计算 (Q x K^T)，并除以 sqrt(d_k)
        scores = torch.matmul(q,k.transpose(-2,-1))/torch.sqrt(torch.tensor(self.d_k, dtype
= torch.float32))

        # mask 操作，屏蔽掉 padding 部分
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # TODO 2: 计算 attention 权重
        # 要求: 在 seq_len 维度上使用 softmax 归一化 scores
        softmax = nn.Softmax(dim = -1)
        attn =  softmax(scores)

        # TODO 3: 计算加权求和后的 context
        # 要求: 用 attn 加权 V，得到 context
        context = torch.matmul(attn, v)
```

```
# TODO 1: 计算多头自注意力输出 x2
        x2 = self.self_attn(x)

        # TODO 2: 残差连接 + 第一层 LayerNorm
        x = self.norm1(x2+x)

        # ------------------ 前馈神经网络块 ------------------ #

        # TODO 3: 前馈全连接网络（两层 Linear + ReLU）得到 x2
        x2 = self.ff(x)

        # TODO 4: 残差连接 + 第二层 LayerNorm
        x = self.norm2(x2+x)
```

实验结果:

```
Epoch 1: Loss = 0.5853, Test Acc = 0.8268
Epoch 2: Loss = 0.3723, Test Acc = 0.8255
Epoch 3: Loss = 0.3083, Test Acc = 0.8488
Epoch 4: Loss = 0.2682, Test Acc = 0.8478
Epoch 5: Loss = 0.2421, Test Acc = 0.8432
```

思考题1：为什么需要对偶数和奇数维度分别使用 sin 和 cos?

这一设计的核心目的是通过频率组合和线性关系，使模型能够灵活捕捉绝对位置与相对位置信息。

思考题2：在 Multi-Head Self-Attention 机制中，为什么我们需要使用多个 attention head?

Multi-Head Self-Attention 的设计旨在通过并行化学习不同的注意力模式，增强模型的表达能力与鲁棒性。

思考题3：为什么要用缩放因子 sqrt(d_k)?

在计算 Self-Attention 的缩放点积时，使用缩放因子sqrt(d_k) 的核心目的是控制点积结果的数值范围，避免梯度消失或爆炸。

思考题4：为什么 Transformer Encoder Layer 中要在 Self-Attention 和 Feed Forward Network 之后都使用残差连接和 LayerNorm?

1. **训练稳定性**:
    - 残差连接确保梯度有效回传，缓解消失/爆炸问题；
    - LayerNorm 抑制特征分布漂移，平滑优化路径。
2. **特征传递**:
    - 残差连接保留原始信息，防止过度扭曲；
    - LayerNorm 规范化特征，提升跨层兼容性。

思考题5：为什么在 TransformerEncoderClassifier 中通常使用 Mean Pooling？其他替代方法的优缺点分析

Mean Pooling 对序列中所有 token 的隐藏状态取平均，避免过度依赖个别 token（如首尾位置），更适用于捕捉句子的整体语义。无论序列长度如何变化，均值操作均能生成固定维度的向量，适用于动态长度的文本输入。

(1) Max Pooling

- **优点**:
    - 捕捉显著特征：关注对分类最关键的局部信息（如情感词、实体词）。
    - 适用于稀疏信号：若关键语义集中在少数 token，Max Pooling 可能更有效。
- **缺点**:
    - 信息丢失：忽略非最大值 token 的贡献，可能损失上下文信息。
    - 对噪声敏感：若最大值对应无关词汇（如广告文本中的超链接），可能引入误导。

(2) [CLS] Token 聚合

- **优点**：
    - 任务导向性：`[CLS]` token 在训练过程中可专门学习分类相关的全局表示（如 BERT 的设计）。
    - 无长度限制：直接取单个 token 的向量，计算成本极低。
- **缺点**：
    - 依赖预训练：若模型未在预训练阶段学习 `[CLS]` token 的语义（如从头训练的 Transformer），效果可能较差。
    - 容量限制：单个 token 的表示可能无法涵盖复杂句子的全部信息。

思考题6：Transformer 相比传统的 RNN/CNN，优势在哪里？为什么 Transformer 更适合处理长文本?

Transformer 的核心优势:

- 自注意力机制允许同时计算序列中所有位置的关联，利用 GPU 并行计算加速训练
- 自注意力机制直接建模任意两位置的关系，无论距离远近。例如，句子开头的词可直接影响结尾的词，无需中间传递，从而更高效捕捉全局上下文。
- ransformer 通过位置编码（如正弦函数或可学习嵌入）显式表示序列顺序，而 RNN 的顺序处理天然隐含位置信息，CNN 通过滑动窗口局部感知位置。

为什么 Transformer 更适合处理长文本?

- 对于长文本中的远距离依赖（如篇章级指代、跨段落逻辑），自注意力机制无需逐层传递信息，可直接建立关联。
- Transformer 的矩阵运算高度适配 GPU/TPU 的并行架构，即使处理长文本，训练和推理速度仍可接受。

实验任务二

代码:

```
import torch
import pandas as pd
import numpy as np
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from transformers import AutoModel, AutoTokenizer, AutoModelForSequenceClassification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from tqdm import tqdm

df = pd.read_csv("./drive/MyDrive/ag/train.csv")
df.columns = ["label", "title", "description"]
df["text"] = df["title"] + " " + df["description"]
df["label"] = df["label"] - 1
train_texts, train_labels = df["text"].tolist(), df["label"].tolist()
number = int(0.3 * len(train_texts))
```

```
train_texts, train_labels = train_texts[:number], train_labels[:number]

df = pd.read_csv("./drive/MyDrive/ag/test.csv")
df.columns = ["label", "title", "description"]
df["text"] = df["title"] + " " + df["description"]
df["label"] = df["label"] - 1
test_texts, test_labels = df["text"].tolist(), df["label"].tolist()

model_name = "./drive/MyDrive/bert-base-uncased/bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)

class AGNewsDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=50):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]
        encoding = self.tokenizer(
            text, truncation=True, padding="max_length",
            max_length=self.max_length, return_tensors="pt"
        )
        return {
            "input_ids": encoding["input_ids"].squeeze(0),
            "attention_mask": encoding["attention_mask"].squeeze(0),
            "labels": torch.tensor(label, dtype=torch.long),
        }

train_dataset = AGNewsDataset(train_texts, train_labels, tokenizer)
test_dataset = AGNewsDataset(test_texts, test_labels, tokenizer)
train_dataloader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=128, shuffle=False)

class BERTClassifier(nn.Module):
    def __init__(self, model_name, num_labels, pool_type='cls'):
        super(BERTClassifier, self).__init__()
        self.bert = AutoModel.from_pretrained(model_name)
        self.pool_type = pool_type
        self.num_labels = num_labels

        if pool_type == 'attention':
            self.attention = nn.Sequential(
                nn.Linear(self.bert.config.hidden_size, 128),
                nn.Tanh(),
                nn.Linear(128, 1)
            )
```

```
        self.classifier = nn.Linear(self.bert.config.hidden_size, num_labels)
        self.dropout = nn.Dropout(0.1)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids,
                            attention_mask=attention_mask)
        last_hidden = outputs.last_hidden_state  # [batch, seq_len, hidden_size]

        # CLS pooling (直接使用[CLS] token)
        if self.pool_type == 'cls':
            pooled = last_hidden[:, 0, :]

        # Mean pooling (平均所有token)
        elif self.pool_type == 'mean':
            mask = attention_mask.unsqueeze(-1).expand(last_hidden.size()).float()
            sum_hidden = torch.sum(last_hidden * mask, 1)
            sum_mask = torch.clamp(mask.sum(1), min=1e-9)
            pooled = sum_hidden / sum_mask

        # Attention pooling (加权平均)
        elif self.pool_type == 'attention':
            attn_weights = self.attention(last_hidden).squeeze(-1)  # [batch, seq_len]
            attn_weights = attn_weights.masked_fill(attention_mask == 0, float('-inf'))
            attn_weights = torch.softmax(attn_weights, dim=1)
            pooled = torch.sum(last_hidden * attn_weights.unsqueeze(-1), dim=1)

        pooled = self.dropout(pooled)
        return self.classifier(pooled)

def train_and_evaluate(pool_type):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = BERTClassifier(model_name, num_labels=4, pool_type=pool_type).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=2e-5)

    EPOCHS = 3
    for epoch in range(EPOCHS):
        model.train()
        total_loss = 0
        loop = tqdm(train_dataloader, desc=f"Epoch {epoch+1} [{pool_type}]")

        for batch in loop:
            optimizer.zero_grad()
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
            labels = batch["labels"].to(device)

            outputs = model(input_ids, attention_mask)
            loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()
```

```
            total_loss += loss.item()
            loop.set_postfix(loss=loss.item())

        print(f"Epoch {epoch+1}, Loss: {total_loss/len(train_dataloader):.4f}")

        model.eval()
        preds, true_labels = [], []
        with torch.no_grad():
            for batch in test_dataloader:
                input_ids = batch["input_ids"].to(device)
                attention_mask = batch["attention_mask"].to(device)
                labels = batch["labels"].cpu().numpy()

                outputs = model(input_ids, attention_mask)
                preds.extend(torch.argmax(outputs, dim=1).cpu().numpy())
                true_labels.extend(labels)

        acc = accuracy_score(true_labels, preds)
        print(f"Test Accuracy ({pool_type}): {acc:.4f}\n")

    return acc

results = {}
for pool_type in ['cls', 'mean', 'attention']:
    print(f"\n=== Evaluating {pool_type} pooling ===")
    results[pool_type] = train_and_evaluate(pool_type)

print("\n=== Final Results ===")
for k, v in results.items():
    print(f"{k} pooling accuracy: {v:.4f}")
```

实验结果:

```
=== Evaluating cls pooling ===
Epoch 1 [cls]: 100%|██████████| 282/282 [04:38<00:00,  1.01it/s, loss=0.389]
Epoch 1, Loss: 0.3219
Test Accuracy (cls): 0.9149

Epoch 2 [cls]: 100%|██████████| 282/282 [04:38<00:00,  1.01it/s, loss=0.313]
Epoch 2, Loss: 0.1707
Test Accuracy (cls): 0.9221

Epoch 3 [cls]: 100%|██████████| 282/282 [04:38<00:00,  1.01it/s, loss=0.219]
Epoch 3, Loss: 0.1193
Test Accuracy (cls): 0.9226


=== Evaluating mean pooling ===
Epoch 1 [mean]: 100%|██████████| 282/282 [04:40<00:00,  1.01it/s, loss=0.356]
Epoch 1, Loss: 0.3290
Test Accuracy (mean): 0.9180
```

```
Epoch 2 [mean]: 100%|██████████| 282/282 [04:41<00:00,  1.00it/s, loss=0.408]
Epoch 2, Loss: 0.1773
Test Accuracy (mean): 0.9203

Epoch 3 [mean]: 100%|██████████| 282/282 [04:40<00:00,  1.01it/s, loss=0.236]
Epoch 3, Loss: 0.1239
Test Accuracy (mean): 0.9236


=== Evaluating attention pooling ===
Epoch 1 [attention]: 100%|██████████| 282/282 [04:41<00:00,  1.00it/s, loss=0.0723]
Epoch 1, Loss: 0.3276
Test Accuracy (attention): 0.9186

Epoch 2 [attention]: 100%|██████████| 282/282 [04:41<00:00,  1.00it/s, loss=0.0607]
Epoch 2, Loss: 0.1694
Test Accuracy (attention): 0.9243

Epoch 3 [attention]: 100%|██████████| 282/282 [04:41<00:00,  1.00it/s, loss=0.132]
Epoch 3, Loss: 0.1181
Test Accuracy (attention): 0.9137


=== Final Results ===
cls pooling accuracy: 0.9226
mean pooling accuracy: 0.9236
attention pooling accuracy: 0.9137
```

思考题1：你觉得以上三种得到句子嵌入的方案，哪种效果会最好，哪种效果会最差？为什么?

根据实验数据，三种句子聚合方式的最终测试准确率如下：

- **CLS Pooling**: 92.26%
- **Mean Pooling**: 92.36%
- **Attention Pooling**: 91.37%

所以效果最好是Mean Pooling,效果最差的是Attention Pooling

1. 为什么 Mean Pooling 效果最好?

- 全局信息均衡
- 与预训练任务的兼容性强
- 抗噪性强

2. 为什么 Attention Pooling 效果最差?

- 可能发生过拟合
- 注意力权重失效

思考题2：如果一个文档包括多个句子，我们需要获得其中每个句子的嵌入表示。那么，我们应该怎么利用BERT得到每个句子的嵌入？

先将文档拆分为句子列表,然后将句子合并为多个块，确保每个块不超过 BERT 的最大长度，并记录每个句子在块中的位置,对每个块编码为 BERT 输入格式，并记录句子边界.

实验感想与体会

通过本次基于 BERT 的文本分类实验，我对预训练模型的应用、句子聚合方法的设计与优化有了更深刻的理解，同时也遇到了一些挑战并积累了实践经验。

本次实验我调整了学习率,发现一旦学习率过高,bert就无法学习到任何东西,无论怎样修改其他超参数,或者训练更多轮次都无法让它正常学习.