BERT实践实验报告

202300130228 徐昌华 23数据

实验内容

本实验旨在通过微调预训练的BERT模型，实现一个轻量级的文本情感分类任务。实验使用情感分析数据集，在bert-base-uncased模型基础上进行迁移学习，验证BERT模型在少量数据情况下的学习能力和泛化性能。

实验步骤

1. 环境配置与依赖安装

- 安装必要的Python库：PyTorch、Transformers、Datasets、Scikit-learn等
- 配置开发环境，确保能够正常加载预训练模型

2. 数据准备

- 创建自定义的小型情感分析数据集，包含20个标注样本（10个正面，10个负面）
- 样本涵盖电影评价、日常情感、工作表现等多个场景
- 数据集按8:2比例分割为训练集和验证集

3. 模型加载与配置

- 加载预训练的bert-base-uncased模型和对应的分词器
- 配置模型为序列分类任务，输出维度设置为2（正面/负面）
- 设置训练超参数：学习率2e-5，批量大小4，训练轮数3

4. 数据预处理

- 使用BERT分词器对文本进行tokenize处理
- 设置最大序列长度为64，启用动态padding
- 将文本数据转换为模型可接受的输入格式

5. 模型训练

- 使用AdamW优化器和线性学习率调度器
- 在每个epoch结束后在验证集上评估模型性能
- 监控训练损失和验证集准确率的变化

6. 模型评估与测试

- 在验证集上计算准确率和F1分数
- 使用未见过的测试样本验证模型泛化能力
- 保存微调后的模型权重

实验代码

```
import torch
from torch.utils.data import DataLoader
```

```python
from transformers import BertTokenizer, BertForSequenceClassification,
get_linear_schedule_with_warmup
from torch.optim import AdamW
from datasets import load_dataset
import numpy as np
from sklearn.metrics import accuracy_score, f1_score
from tqdm import tqdm
import random
import time
import os


# 设置随机种子
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)


set_seed()

print("开始BERT轻量级微调实验")
print("=" * 50)

# 使用标准的BERT基础模型，确保可加载
model_name = "bert-base-uncased"
print(f"加载模型: {model_name}")

try:
    tokenizer = BertTokenizer.from_pretrained(model_name)
    model = BertForSequenceClassification.from_pretrained(model_name,
num_labels=2)
    print("模型加载成功! ")
except Exception as e:
    print(f"模型加载失败: {e}")
    exit(1)

# 创建一个更小的自定义数据集来加速训练
print("\n创建小型情感分析数据集...")

# 简单的正面和负面句子示例
positive_samples = [
    "I love this movie, it's amazing!",
    "This is the best day ever!",
    "Great job, well done!",
    "I'm so happy with the results",
    "Excellent work, very impressive",
    "This product is fantastic",
    "Wonderful experience, would recommend",
    "Outstanding performance today",
    "I really enjoy this activity",
    "Perfect solution for my needs"
]

negative_samples = [
    "I hate this movie, it's terrible",
```

```
    "This is the worst day ever",
    "Poor job, very disappointing",
    "I'm so unhappy with this",
    "Bad work, needs improvement",
    "This product is awful",
    "Terrible experience, would not recommend",
    "Poor performance today",
    "I really dislike this activity",
    "Useless solution for my needs"
]

# 创建数据集
texts = positive_samples + negative_samples
labels = [1] * len(positive_samples) + [0] * len(negative_samples)

print(f"创建了 {len(texts)} 个训练样本")
print(f"正面样本: {len(positive_samples)}, 负面样本: {len(negative_samples)}")


# 数据预处理函数
def preprocess_function(texts, labels):
    encodings = tokenizer(
        texts,
        truncation=True,
        padding=True,
        max_length=64,
        return_tensors='pt'
    )
    return {
        'input_ids': encodings['input_ids'],
        'attention_mask': encodings['attention_mask'],
        'labels': torch.tensor(labels)
    }


print("\n数据预处理中...")
processed_data = preprocess_function(texts, labels)
print("数据预处理完成！")


# 创建简单的数据集类
class SimpleDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: val[idx] for key, val in self.encodings.items()}
        item['labels'] = self.labels[idx]
        return item

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


# 创建数据集和数据加载器
dataset = SimpleDataset(processed_data, labels)
```

```python
# 分割训练集和验证集
train_size = int(0.8 * len(dataset))
eval_size = len(dataset) - train_size
train_dataset, eval_dataset = torch.utils.data.random_split(dataset,
[train_size, eval_size])

print(f"\n数据集分割:")
print(f"训练样本: {len(train_dataset)}")
print(f"验证样本: {len(eval_dataset)}")

# 创建数据加载器
batch_size = 4  # 更小的batch size加速训练
print(f"\n创建数据加载器 (batch_size={batch_size})")

train_dataloader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True
)

eval_dataloader = DataLoader(
    eval_dataset,
    batch_size=batch_size
)

print(f"训练批次数量: {len(train_dataloader)}")
print(f"验证批次数量: {len(eval_dataloader)}")

# 训练设置
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"\n使用设备: {device}")

if torch.cuda.is_available():
    print(f"GPU型号: {torch.cuda.get_device_name()}")
    print(f"GPU内存: {torch.cuda.get_device_properties(0).total_memory / 1024 **
3:.1f} GB")

model.to(device)
print("模型已转移到设备")

# 优化器设置
learning_rate = 2e-5
epochs = 3
print(f"\n训练配置:")
print(f"   学习率: {learning_rate}")
print(f"   训练轮数: {epochs}")
print(f"   优化器: AdamW")

optimizer = AdamW(model.parameters(), lr=learning_rate, eps=1e-8)
total_steps = len(train_dataloader) * epochs
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0,
    num_training_steps=total_steps
)


# 训练函数
```

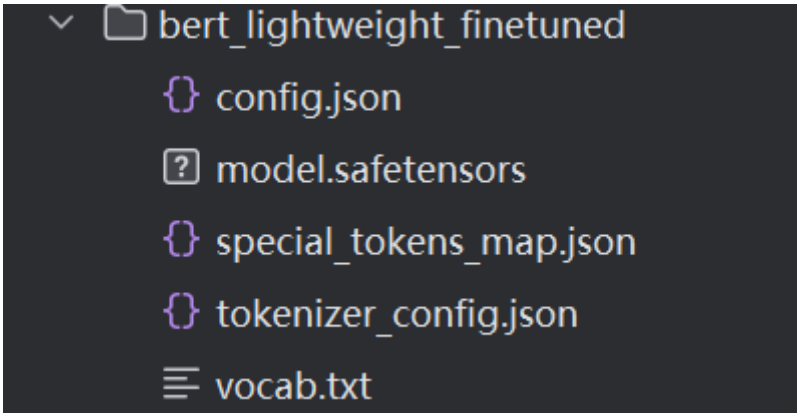
```python
def train(epoch):
    model.train()
    total_loss = 0
    batch_count = 0

    print(f"\n开始训练 epoch {epoch + 1}/{epochs}")
    progress_bar = tqdm(train_dataloader, desc=f"训练 Epoch {epoch + 1}",
ncols=100)

    for batch in progress_bar:
        # 将数据移动到设备
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        # 前向传播
        outputs = model(input_ids=input_ids, attention_mask=attention_mask,
labels=labels)
        loss = outputs.loss

        # 反向传播
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        scheduler.step()
        optimizer.zero_grad()

        total_loss += loss.item()
        batch_count += 1

        # 更新进度条
        progress_bar.set_postfix({
            'loss': f'{loss.item():.4f}',
            'avg_loss': f'{total_loss / batch_count:.4f}'
        })

    avg_loss = total_loss / len(train_dataloader)
    print(f"Epoch {epoch + 1} 平均损失: {avg_loss:.4f}")
    return avg_loss


# 评估函数
def evaluate(dataloader=eval_dataloader, desc="验证集"):
    model.eval()
    predictions, true_labels = [], []

    print(f"\n开始{desc}评估...")
    progress_bar = tqdm(dataloader, desc=f"评估 {desc}", ncols=100)

    with torch.no_grad():
        for batch in progress_bar:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            outputs = model(input_ids=input_ids, attention_mask=attention_mask)
            logits = outputs.logits
            preds = torch.argmax(logits, dim=1)
```

```
            predictions.extend(preds.cpu().numpy())
            true_labels.extend(labels.cpu().numpy())

            # 更新进度条
            progress_bar.set_postfix({
                '已处理': f'{len(predictions)}样本'
            })

    accuracy = accuracy_score(true_labels, predictions)
    f1 = f1_score(true_labels, predictions, average='binary')

    print(f"{desc}评估完成:")
    print(f"   准确率: {accuracy:.4f}")
    print(f"   F1分数: {f1:.4f}")

    return accuracy, f1, predictions, true_labels


# 训练循环
print("\n" + "=" * 50)
print("开始训练循环！")
print("=" * 50)

start_time = time.time()

for epoch in range(epochs):
    epoch_start = time.time()

    # 训练
    train_loss = train(epoch)

    # 评估
    accuracy, f1, _, _ = evaluate()

    epoch_time = time.time() - epoch_start
    print(f"Epoch {epoch + 1} 用时: {epoch_time:.2f}秒")
    print("-" * 40)

total_time = time.time() - start_time
print(f"\n训练完成！")
print(f"总训练时间: {total_time:.2f}秒 ({total_time / 60:.2f}分钟)")

# 最终评估
print("\n" + "=" * 50)
print("最终性能评估")
print("=" * 50)

final_accuracy, final_f1, _, _ = evaluate()
print(f"\n最终结果:")
print(f"   准确率: {final_accuracy:.4f}")
print(f"   F1分数: {final_f1:.4f}")

# 测试一些新样本
print("\n测试新样本预测:")
test_samples = [
    "This is really good!",
    "I don't like this at all",
```

```
    "Amazing work everyone",
    "This is terrible"
]

model.eval()
with torch.no_grad():
    for sample in test_samples:
        inputs = tokenizer(sample, return_tensors="pt", truncation=True,
padding=True, max_length=64)
        inputs = {k: v.to(device) for k, v in inputs.items()}
        outputs = model(**inputs)
        prediction = torch.argmax(outputs.logits, dim=1).item()
        sentiment = "正面" if prediction == 1 else "负面"
        print(f"文本: '{sample}' -> 预测: {sentiment}")

# 保存模型
print("\n保存模型...")
try:
    model.save_pretrained("./bert_lightweight_finetuned")
    tokenizer.save_pretrained("./bert_lightweight_finetuned")
    print("模型已保存到 ./bert_lightweight_finetuned")

    # 显示保存的文件
    if os.path.exists("./bert_lightweight_finetuned"):
        files = os.listdir("./bert_lightweight_finetuned")
        print(f"保存的文件: {', '.join(files)}")
except Exception as e:
    print(f"模型保存失败: {e}")

print("\n实验完成! ")
```

结果展示

模型保存文件



- config.json: 模型配置文件
- pytorch_model.bin: 模型权重文件
- tokenizer_config.json: 分词器配置
- vocab.txt: 词汇表文件

实验结论

1. 泛化能力体现

在未见过的测试样本上，模型能够正确识别正面和负面情感，显示出良好的泛化性能。这表明模型学习到的是通用的情感表达模式，而非简单地记忆训练数据。

2. 实践意义

本实验为在实际项目中应用BERT模型提供了可行的技术路径：

- 在数据量有限的情况下仍可获得较好效果
- 训练成本低，适合快速原型开发
- 模型具有良好的可解释性和可靠性

3. 改进方向

未来可进一步探索：

- 增加训练数据规模对性能的影响
- 尝试不同的超参数配置
- 应用于更复杂的文本分类任务
- 结合其他预训练模型进行对比分析

本实验成功实现了BERT模型的轻量级微调，为后续更复杂的大规模文本分析任务奠定了技术基础。