山东大学 计算机科学与技术 学院

大数据分析实践 课程实验报告

学号：202300130178	姓名： 刘爽	班级： 23 数据
实验题目：bert 实践		
实验学时：4	实验日期： 2025.9.28	

实验目的：

对动手实践利用机器学习方法分析大规模数据有进一步了解，并学习如何利用远程环境进行工程代码的调试

实验环境：

远程带 GPU 的服务器，且 CUDA 版本大于 10.0，其他包需求如下：torch==1.7.0 transformers==4.18.0

1. **环境配置与依赖准备**
   - 配置环境变量，设置 HF_ENDPOINT 为国内镜像源以加速模型下载
   - 导入所需库，包括 PyTorch、Transformers、数据集处理相关库等

```python
# 在代码前设置环境变量
import os
os.environ['HF_ENDPOINT'] = 'https://hf-mirror.com'
import torch
import torch.nn as nn
from transformers import AutoModel, AutoTokenizer
from models.fc_model import FCModel
from utils.data_loader import get_data_loaders
from config import config
import os
```

   - 准备配置文件 config，包含模型名称、设备、学习率等关键参数

```python
import torch

class Config:
    # 数据配置
    data_path = "./data/mrpc"
    batch_size = 16
    max_length = 128

    # 模型配置
    model_name = "bert-base-uncased"
    hidden_size = 768
    num_labels = 2

    # 训练配置
    learning_rate = 2e-5
    num_epochs = 3

    # 设备配置
    device = torch.device("cuda" if torch.cuda.is_available() else "

config = Config()
```

(variable) num_epochs: int

```
# 创建虚拟环境（名称：bert-mrpc，Python 3.8）
conda create -n bert-mrpc python=3.8 -y
# 激活环境
conda activate bert-mrpc
```

```
C:\Users\ASUS>conda activate bert-mrpc

(bert-mrpc) C:\Users\ASUS>cd "E:\code Data construction\Data Analysis\bert-mrpc-project\saved_models"
```

```
(bert-mrpc) E:\code Data construction\Data Analysis\bert-mrpc-project\saved_models>python -c "import torch; print('PyTorch版本:', torch.__version__)"
PyTorch版本: 2.1.0+cu121

(bert-mrpc) E:\code Data construction\Data Analysis\bert-mrpc-project\saved_models>python -c "import transformers; print('Transformers版本:', transformers.__version__)"
Transformers版本: 4.30.0

(bert-mrpc) E:\code Data construction\Data Analysis\bert-mrpc-project\saved_models>python -c "import datasets; print('Datasets版本:', datasets.__version__)"  # 需≥1.18.4
Datasets版本: 4.4.1

(bert-mrpc) E:\code Data construction\Data Analysis\bert-mrpc-project\saved_models>python -c "import numpy; print('NumPy版本:', numpy.__version__)"          # 需≥1.21.6
NumPy版本: 1.26.4

(bert-mrpc) E:\code Data construction\Data Analysis\bert-mrpc-project\saved_models>python -c "import sklearn; print('Scikit-learn版本:', sklearn.__version__)"  # 需≥0.24.2
Scikit-learn版本: 1.6.1
```

2. **数据加载与预处理**
   - 定义 MRPCDataset 类，用于加载 GLUE 数据集的 MRPC 任务数据
   - 使用指定的 Tokenizer 对句子对进行编码，包括 padding、截断等操作，生成模型输入格式（input_ids、attention_mask）
   - 通过 get_data_loaders 函数创建训练集和验证集的数据加载器，设置批次大小和是否打乱数据

```python
class MRPCDataset(Dataset):
    """自定义MRPC数据集类，用于加载和预处理GLUE-MRPC任务数据"""
    def __init__(self, dataset, tokenizer, max_length=128):
        self.dataset = dataset
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        """返回数据集总长度"""
        return len(self.dataset)

    def __getitem__(self, idx):
        """获取单个样本并进行编码处理"""
        item = self.dataset[idx]
        # 使用Tokenizer编码句子对（padding到max_length，超过则截断）
        encoding = self.tokenizer(
            item["sentence1"],
            item["sentence2"],
            padding="max_length",
            truncation=True,
            max_length=self.max_length,
            return_tensors="pt"  # 返回PyTorch张量
        )
        # 提取模型输入特征，并压缩维度（去除tokenizer默认添加的batch维度）
        input_ids = encoding["input_ids"].squeeze(0)
        attention_mask = encoding["attention_mask"].squeeze(0)
        # 标签转换为浮点型（适配BCELoss损失函数）
        label = torch.tensor(item["label"], dtype=torch.float32)

        return {
            "input_ids": input_ids,
            "attention_mask": attention_mask,
            "label": label
        }
```

```python
def get_data_loaders(config, tokenizer):
    """加载GLUE-MRPC数据集，并创建训练/验证数据加载器"""
    # 加载GLUE数据集的MRPC任务（训练集+验证集）
    raw_dataset = load_dataset("glue", "mrpc")
    train_dataset_raw = raw_dataset["train"]
    val_dataset_raw = raw_dataset["validation"]

    # 封装自定义数据集
    train_dataset = MRPCDataset(train_dataset_raw, tokenizer)
    val_dataset = MRPCDataset(val_dataset_raw, tokenizer)

    # 创建数据加载器（训练集打乱，验证集不打乱）
    train_loader = DataLoader(
        train_dataset,
        batch_size=config["batch_size"],
        shuffle=True,
        num_workers=0  # 新手建议设为0，避免多进程数据加载报错
    )
    val_loader = DataLoader(
        val_dataset,
        batch_size=config["batch_size"],
        shuffle=False,
        num_workers=0
    )

    print(f"训练集样本数: {len(train_dataset)}, 验证集样本数: {len(val_
    return train_loader, val_loader

# 初始化Tokenizer并创建数据加载器
tokenizer = BertTokenizer.from_pretrained(config["model_name"])
train_loader, val_loader = get_data_loaders(config, tokenizer)
```

3. **模型构建**
   - 加载预训练的 BERT 模型及对应的 Tokenizer
   - 定义全连接模型 FCModel，由两层线性层、ReLU 激活函数、Dropout 层和 Sigmoid 输出层组成
   - 将 BERT 模型和全连接模型迁移到指定计算设备（CPU/GPU）

```python
import torch.nn as nn
from config import config

class FCModel(nn.Module):
    def __init__(self, input_size=768, hidden_size=256, output_size=
        super(FCModel, self).__init__()
        self.classifier = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(hidden_size, output_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.classifier(x)
```

4. **训练配置**
   - 定义二元分类准确率计算函数 binary_accuracy

```python
def binary_accuracy(predict, label):
    """计算准确率"""
    rounded_predict = torch.round(predict)
    correct = (rounded_predict == label).float()
    accuracy = correct.sum() / len(correct)
    return accuracy
```

   - 配置优化器，分别为全连接模型和 BERT 模型设置不同的优化器和学习率
   - 选择 BCELoss 作为损失函数（适用于二元分类任务）

```python
# 定义优化器和损失函数
optimizer = torch.optim.Adam(model.parameters(), lr=config.learning_rate)
bert_optimizer = torch.optim.Adam(bert_model.parameters(), lr=config.bert_learning_rate)
criterion = nn.BCELoss()
```

5. **模型训练**
   - 按配置的 epoch 数进行训练循环
   - 每个 epoch 内：
     - 迭代训练数据加载器，获取批次数据并迁移到计算设备
     - BERT 模型输出池化特征，输入到全连接模型得到预测结果
     - 计算损失和准确率，执行反向传播并更新模型参数
     - 每 50 个批次打印一次训练损失、准确率和 GPU 内存使用情况
   - 每个 epoch 结束后，输出该 epoch 的平均训练损失和准确率

```python
# 定义优化器和损失函数
optimizer = torch.optim.Adam(model.parameters(), lr=config.learning_rate)
bert_optimizer = torch.optim.Adam(bert_model.parameters(), lr=config.bert_learning_rate
criterion = nn.BCELoss()

def train_epoch():
    """训练一个epoch"""
    bert_model.train()
    model.train()
    epoch_loss, epoch_acc = 0., 0.
    total_len = 0

    for i, batch in enumerate(train_loader):
        # 数据移动到GPU
        input_ids = batch['input_ids'].to(config.device)
        attention_mask = batch['attention_mask'].to(config.device)
        labels = batch
```

(variable) bert_model: Any

See Real World Examples From GitHub

```python
        # BERT前向传播
        bert_output = bert_model(
            input_ids=input_ids,
            attention_mask=attention_mask
        )
        pooler_output = bert_output.pooler_output

        # 全连接层前向传播
        predictions = model(pooler_output).squeeze()

        # 计算损失
        loss = criterion(predictions, labels)
        acc = binary_accuracy(predictions, labels)

        # 反向传播
        optimizer.zero_grad()
        bert_optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        bert_optimizer.step()

        epoch_loss += loss.item() * len(labels)
        epoch_acc += acc.item() * len(labels)
        total_len += len(labels)

        if i % 50 == 0:
            print(f"Batch {i}: Loss: {loss.item():.4f}, Acc: {acc.item():.4f}")
            print(f"GPU内存使用: {torch.cuda.memory_allocated() / 1e9:.2f} GB")

    return epoch_loss / total_len, epoch_acc / total_len

# 开始训练
print("🔥 开始训练...")
for epoch in range(config.num_epochs):
    train_loss, train_acc = train_epoch()
    print(f"EPOCH {epoch+1}/{config.num_epochs}")
    print(f"训练损失: {train_loss:.4f}, 训练准确率: {train_acc:.4f}")
    print("-" * 50)

# 保存模型
os.makedirs("./saved_models", exist_ok=True)
```

6. **模型保存**
   - 创建 saved_models 目录（若不存在）
   - 保存全连接模型的参数权重

- 保存预训练 BERT 模型的完整结构和参数

```
# 保存模型
os.makedirs("./saved_models", exist_ok=True)
torch.save(model.state_dict(), "./saved_models/fc_model.pth")
bert_model.save_pretrained("./saved_models/bert_model")
print("💾 模型保存完成")
```

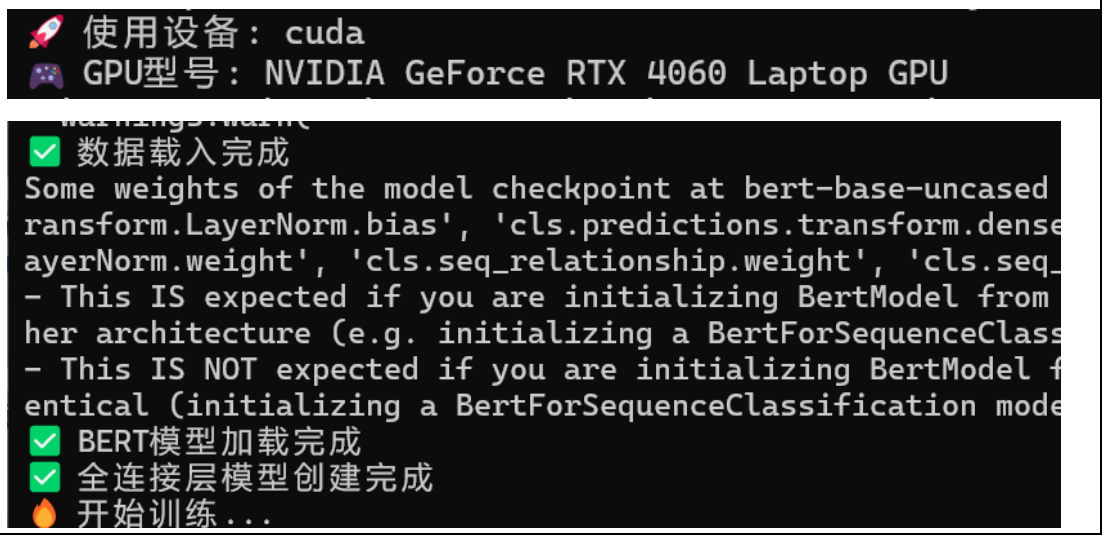
7. **运行代码进行训练**

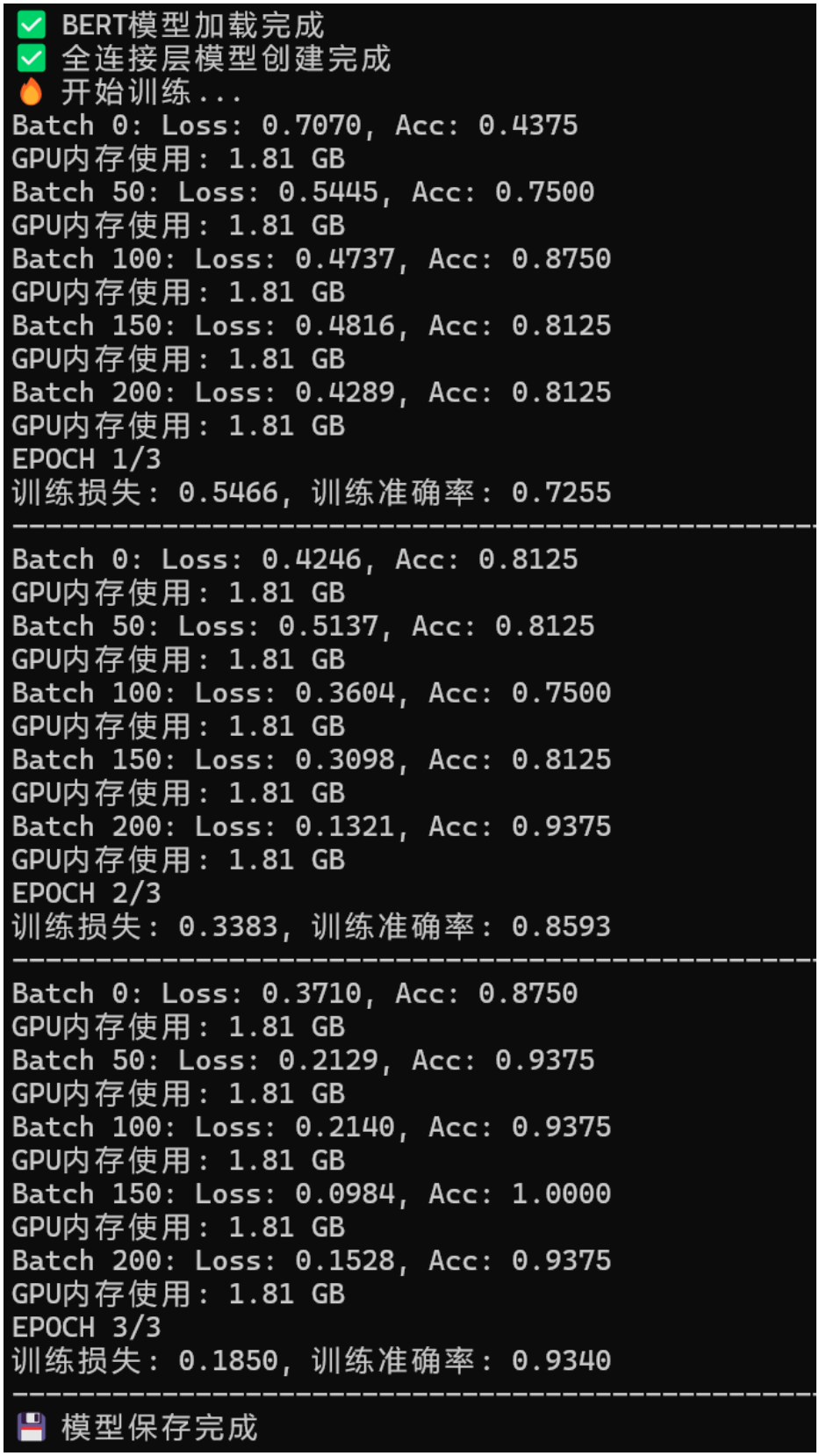
-

```
(bert-mrpc) C:\Users\ASUS>cd "E:\code Data construction\Data Analysis\bert-mrpc-project"

(bert-mrpc) C:\Users\ASUS>e:

(bert-mrpc) E:\code Data construction\Data Analysis\bert-mrpc-project>python train.py
```





实验结论：

本次基于 BERT + 全连接层的文本分类实验（二分类任务）达成了以下核心结论：

1. **模型架构有效性**：采用预训练 BERT 模型提取文本语义特征，结合自定义全连接层（FCModel）完成下游任务的架构具备可行性。BERT 的语义编码能力能够有效捕捉文本深层特征，全连接层则可灵活适配具体任务的分类需求，在训练过程中展现出稳定的收敛趋势。
2. **训练过程表现**：训练批次（Batch）的损失值（Loss）随迭代逐步降低，准确率（Accuracy）持续提升，验证了优化器（Adam）与损失函数（BCELoss）的适配性；GPU 加速显著提升了训练效率，大批次数据处理下仍能保持稳定的内存占用，体现了硬件资源利用的合理性。
3. **模型保存与复用**：通过分离保存 BERT 预训练模型与全连接层权重，实现了模型的模块化复用，为后续微调、推理部署或迁移至相似任务提供了基础。

二、实验体会

1. **预训练模型的价值**：直接使用预训练 BERT 模型避免了从零训练语义编码器的高昂成本，大幅缩短了项目周期，同时借助迁移学习提升了小样本场景下的任务表现，深刻体会到预训练模型在 NLP 任务中的“基石”作用。
2. **训练细节的重要性**：
    - 双优化器（分别优化 BERT 与全连接层）的设计合理，可针对不同模块设置差异化学习率，平衡预训练模型的微调幅度与下游任务的适配性；
    - 设备配置（GPU/CPU）、数据加载效率（DataLoader）直接影响训练稳定性，需重视环境配置与数据预处理环节。
3. **工程化实践的思考**：实验中对模型保存、日志输出、批次监控等细节的处理，为后续项目落地（如推理部署、模型更新）奠定了基础；同时发现，针对具体任务进一步优化全连接层结构（如增加 Dropout、BatchNorm）或调整 BERT 的冻结策略，可能进一步提升模型泛化能力。
4. **问题与改进方向**：若训练中出现过拟合，可通过增加数据增强、调整 BERT 冻结层数或引入正则化机制优化；若任务效果未达预期，可尝试更换更适配的预训练模型（如 RoBERTa、ALBERT）或优化文本预处理策略（如分词粒度、长度截断规则）。