

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: 数据清洗		
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		
实验步骤: (1) 安装 Anaconda, 创建并激活虚拟环境:		
<pre># 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</pre>		
<pre>C:\Users\ASUS>conda activate bert-mrpc (bert-mrpc) C:\Users\ASUS>cd "E:\code Data construction\Data Analysis\bert-mrpc-project\saved_models"</pre>		
(2) 安装依赖库, 验证环境:		
<pre>(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</pre>		
(3) 加载代码进行训练:		

```
Code Data Construction > Data Analysis > Bert fine-tune project > train.py > train
# 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

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

def train():
    # 设置设备
    print(f"🚀 使用设备: {config.device}")
    print(f"GPU型号: {torch.cuda.get_device_name(0)}")
    #
    # 加载数据
    train_loader, val_loader = get_data_loaders()
    print("✅ 数据载入完成")

    # 加载BERT模型
    tokenizer = AutoTokenizer.from_pretrained(config.model_name)
    bert_model = AutoModel.from_pretrained(config.model_name)
    bert_model.to(config.device)
```

```
# 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['label']  # See Real World Examples From GitHub
        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)

```

```

# 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")

```

```

if __name__ == "__main__":
    train()

```

```

(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

```

 使用设备: cuda

 GPU型号: NVIDIA GeForce RTX 4060 Laptop GPU

warnings.warn

✓ 数据载入完成

Some weights of the model checkpoint at bert-base-uncased
transform.LayerNorm.bias', 'cls.predictions.transform.dense
ayerNorm.weight', 'cls.seq_relationship.weight', 'cls.seq_

- This IS expected if you are initializing BertModel from
her architecture (e.g. initializing a BertForSequenceClass

- This IS NOT expected if you are initializing BertModel f
entical (initializing a BertForSequenceClassification mode

✓ BERT模型加载完成

✓ 全连接层模型创建完成

🔥 开始训练 ...

BERT模型加载完成
 全连接层模型创建完成
🔥 开始训练 ...

Batch 0: Loss: 0.7070, Acc: 0.4375
GPU内存使用: 1.81 GB
Batch 50: Loss: 0.5445, Acc: 0.7500
GPU内存使用: 1.81 GB
Batch 100: Loss: 0.4737, Acc: 0.8750
GPU内存使用: 1.81 GB
Batch 150: Loss: 0.4816, Acc: 0.8125
GPU内存使用: 1.81 GB
Batch 200: Loss: 0.4289, Acc: 0.8125
GPU内存使用: 1.81 GB
EPOCH 1/3
训练损失: 0.5466, 训练准确率: 0.7255

Batch 0: Loss: 0.4246, Acc: 0.8125
GPU内存使用: 1.81 GB
Batch 50: Loss: 0.5137, Acc: 0.8125
GPU内存使用: 1.81 GB
Batch 100: Loss: 0.3604, Acc: 0.7500
GPU内存使用: 1.81 GB
Batch 150: Loss: 0.3098, Acc: 0.8125
GPU内存使用: 1.81 GB
Batch 200: Loss: 0.1321, Acc: 0.9375
GPU内存使用: 1.81 GB
EPOCH 2/3
训练损失: 0.3383, 训练准确率: 0.8593

Batch 0: Loss: 0.3710, Acc: 0.8750
GPU内存使用: 1.81 GB
Batch 50: Loss: 0.2129, Acc: 0.9375
GPU内存使用: 1.81 GB
Batch 100: Loss: 0.2140, Acc: 0.9375
GPU内存使用: 1.81 GB
Batch 150: Loss: 0.0984, Acc: 1.0000
GPU内存使用: 1.81 GB
Batch 200: Loss: 0.1528, Acc: 0.9375
GPU内存使用: 1.81 GB
EPOCH 3/3
训练损失: 0.1850, 训练准确率: 0.9340

 模型保存完成

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）。