

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：202300130236	姓名： 陈德康	班级： 数据 23
实验题目： BERT 实践		
实验学时： 2	实验日期： 2025.11.7	
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		
硬件环境： 计算机一台		
软件环境： Linux		
实验步骤与内容： 在实验 4 中，已经配置好了 BERT 微调相关的环境，使用 VSCode 的 SSH REMOTE 插件远程连接服务器，并把实验代码上传至服务器； 在代码中，我使用了预训练的 BERT 模型在 MRPC 数据集上进行微调，判断两个句子是否为语义等价的复述关系； 实验代码如下：		
<pre>import torch from torch.utils.data import DataLoader from transformers import BertTokenizer, BertForSequenceClassification, AdamW, get_linear_schedule_with_warmup from datasets import load_dataset import numpy as np from sklearn.metrics import accuracy_score, f1_score from tqdm import tqdm import random # 设置随机种子 def set_seed(seed=42): random.seed(seed) np.random.seed(seed) torch.manual_seed(seed) torch.cuda.manual_seed_all(seed) set_seed() # 加载模型和分词器 model_name = "bert-base-uncased"</pre>		

```
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name, num_labels=2)
# 加载 MRPC 数据集
print("加载 MRPC 数据集...")
dataset = load_dataset("glue", "mrpc")
# 数据预处理函数
def preprocess_function(examples):
    # 使用动态 padding, 在 collate 函数中统一处理
    return tokenizer(
        examples['sentence1'],
        examples['sentence2'],
        truncation=True,
        padding=False, # 在 collate 中统一 padding
        max_length=128,
    )
# 预处理数据集
tokenized_datasets = dataset.map(preprocess_function, batched=True)
# 设置数据格式
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
# 自定义 collate 函数来处理动态 padding
def collate_fn(batch):
    # 提取所有输入
    input_ids = [torch.tensor(item['input_ids']) for item in batch]
    attention_mask = [torch.tensor(item['attention_mask']) for item in batch]
    token_type_ids = [torch.tensor(item['token_type_ids']) for item in batch]
    labels = torch.tensor([item['labels'] for item in batch])

    # 动态 padding 到批次中的最大长度
    input_ids = torch.nn.utils.rnn.pad_sequence(input_ids, batch_first=True, padding_value=tokenizer.pad_token_id)
    attention_mask = torch.nn.utils.rnn.pad_sequence(attention_mask, batch_first=True, padding_value=0)
    token_type_ids = torch.nn.utils.rnn.pad_sequence(token_type_ids, batch_first=True, padding_value=0)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'token_type_ids': token_type_ids,
        'labels': labels
    }
# 创建数据加载器
train_dataloader = DataLoader(
    tokenized_datasets['train'],
```

```
        batch_size=16,
        shuffle=True,
        collate_fn=collate_fn
    )
eval_dataloader = DataLoader(
    tokenized_datasets['validation'],
    batch_size=16,
    collate_fn=collate_fn
)
# 训练设置
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")
model.to(device)
# 优化器和学习率调度器
optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)
epochs = 3
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
def train():
    model.train()
    total_loss = 0

    progress_bar = tqdm(train_dataloader, desc="训练")
    for batch in progress_bar:
        batch = {k: v.to(device) for k, v in batch.items()}

        outputs = model(**batch)
        loss = outputs.loss

        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        optimizer.step()
        scheduler.step()
        optimizer.zero_grad()

        total_loss += loss.item()
    progress_bar.set_postfix({'loss': loss.item()})
```

```

        return total_loss / len(train_dataloader)
# 评估函数
def evaluate():
    model.eval()
    predictions = []
    true_labels = []

    with torch.no_grad():
        for batch in tqdm(eval_dataloader, desc="评估"):
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)

            logits = outputs.logits
            preds = torch.argmax(logits, dim=1)

            predictions.extend(preds.cpu().numpy())
            true_labels.extend(batch['labels'].cpu().numpy())

    accuracy = accuracy_score(true_labels, predictions)
    f1 = f1_score(true_labels, predictions)

    return accuracy, f1, predictions, true_labels
# 训练循环
print("开始训练...")
for epoch in range(epochs):
    print(f"\nEpoch {epoch + 1}/{epochs}")

    train_loss = train()
    print(f"训练损失: {train_loss:.4f}")

    accuracy, f1, _, _ = evaluate()
    print(f"验证集准确率: {accuracy:.4f}, F1 分数: {f1:.4f}")
# 最终测试
print("\n 在测试集上最终评估...")
test_accuracy, test_f1, test_predictions, test_true_labels = evaluate()
print(f"测试集准确率: {test_accuracy:.4f}")
print(f"测试集 F1 分数: {test_f1:.4f}")
# 保存模型
model.save_pretrained("./bert_mrpc_finetuned")
tokenizer.save_pretrained("./bert_mrpc_finetuned")
print("模型已保存到 ./bert_mrpc_finetuned")

```

示例运行结果如下图所示:

```
开始训练...

Epoch 1/3
训练: 100% | 230/230 [02:09<00:00, 1.77it/s, loss=0.411]
训练损失: 0.5284
评估: 100% | 26/26 [00:03<00:00, 8.33it/s]
验证集准确率: 0.8211, F1分数: 0.8636

Epoch 2/3
训练: 100% | 230/230 [02:06<00:00, 1.81it/s, loss=0.354]
训练损失: 0.3216
评估: 100% | 26/26 [00:03<00:00, 8.25it/s]
验证集准确率: 0.8578, F1分数: 0.8986

Epoch 3/3
训练: 100% | 230/230 [02:04<00:00, 1.84it/s, loss=0.776]
训练损失: 0.1840
评估: 100% | 26/26 [00:03<00:00, 8.61it/s]
验证集准确率: 0.8554, F1分数: 0.8988

在测试集上最终评估...
评估: 100% | 26/26 [00:03<00:00, 7.97it/s]
测试集准确率: 0.8554
```

最终输出的结果如下图所示:

```
▼ bert_mrpc_finetuned
  {} config.json
  ≡ model.safetensors
  {} special_tokens_map.json
  {} tokenizer_config.json
  ≡ vocab.txt
```

输出包含了:

- **config.json**: 模型配置文件, 包含 BERT 模型的架构参数、隐藏层大小等信息;
- **model.safetensors**: 模型权重文件, 包含微调后的所有参数权重;
- **tokenizer_config.json**: 分词器配置;
- **special_tokens_map.json**: 特殊 token 映射 (如[CLS]、[SEP]、[PAD]等)
- **vocab.txt**: BERT 词汇表

结论分析与体会:

本次实验通过微调 BERT 模型在 MRPC 语义等价任务, 成功在远程服务器上完成了 BERT 模型的微调任务, 并且将结果文件拉取保存在了本地; 实验结果表明, 即使仅用 3 个 epoch 的微调, BERT 就能有效学习句子间的语义关系, 这印证了预训练过程中获得的语言理解能力可以高效适配到特定任务。这为我进行进一步的学习和进行更复杂的实验奠定基础