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：202300130051	姓名： 汤冉	班级：数据班
实验题目：实验 5		
实验学时： 2	实验日期：2025.11.27	
硬件环境：Windows11		
软件环境：vscode		

实验步骤与内容：

（1）BertMRPCClassifier 模型类：BertMRPCClassifier 这个类把“预训练 BERT 主体”和“自定义两层 MLP 分类头”打包成了一个完整的文本分类模型：构造函数里先通过 BertModel.from_pretrained 加载了 bert-base-uncased，负责从句子对中提取语义表示；然后定义了一个 nn.Sequential 的两层全连接网络（768→256→2，加上 ReLU），用来根据 BERT 输出的句子向量做最终的同义/不同义分类。在 forward 里，模型接收 input_ids 和 attention_mask，先用 BERT 得到池化后的句子表示 pooler_output，再送入 MLP 得到 logits；如果同时传入了标签 labels，就顺便用 CrossEntropyLoss 计算分类损失，并把 loss 和 logits 一起返回，方便训练时直接使用。

```
class BertMRPCClassifier(nn.Module):
    def __init__(self, backbone_name: str = "bert-base-uncased",
                 hidden_size: int = 768,
                 mlp_hidden: int = 256,
                 num_labels: int = 2):
        super().__init__()
        # 预训练 BERT 主体
        self.bert = BertModel.from_pretrained(backbone_name)
        # 两层 MLP 分类头
        self.classifier = nn.Sequential(
            nn.Linear(hidden_size, mlp_hidden),
            nn.ReLU(),
            nn.Linear(mlp_hidden, num_labels)   # CrossEntropyLoss 里做 softmax
        )

    def forward(self, input_ids, attention_mask, labels=None):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled = outputs.pooler_output              # [batch, 768]
        logits = self.classifier(pooled)            # [batch, num_labels]

        loss = None
        if labels is not None:
            loss_fn = nn.CrossEntropyLoss()
            loss = loss_fn(logits, labels)

        return {"loss": loss, "logits": logits}
```

（2）build_tokenize_fn 函数：build_tokenize_fn 的作用是生成一个“批量分词预处理函数”，专门用来喂给 datasets 的 map 方法。它内部返回的 tokenize_batch 会对一个 batch 的 MRPC 样本（字典中有 "sentence1" 和 "sentence2"）调用 BERT 的 tokenizer，把成对的句子编码成模型可以接受的张量形式：自动添加 [CLS] 和 [SEP] 标记、按照最大长度进行截断、统一 padding 到固定长度 max_length，最后产出 input_ids、attention_mask 等字段。这样做的结果是，GLUE 原始的文本数据在 map 之后就被转换成了标准的 BERT 输入格式。

```
def build_tokenize_fn(tokenizer, max_len=128):
    def tokenize_batch(batch):
        return tokenizer(
            batch["sentence1"],
            batch["sentence2"],
            truncation=True,
            padding="max_length",
            max_length=max_len
        )
    return tokenize_batch
```

（3）设备、分词器和原始数据集:首先用 torch.device("cuda" if torch.cuda.is_available() else "cpu") 自动选择当前机器上是否有 GPU，并打印出使用的设备；然后通过 BertTokenizer.from_pretrained("bert-base-uncased") 加载与 BERT 模型配套的分词器；接着使用 load_dataset("glue", "mrpc") 一行直接从 GLUE 官方源下载并加载 MRPC 数据集，得到包含 train / validation / test 划分的 DatasetDict，为后续预处理和训练做好准备。

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("使用设备:", device)

# -------- 分词器 / 数据集 --------
model_name = "bert-base-uncased"
tokenizer = BertTokenizer.from_pretrained(model_name)

raw_datasets = load_dataset("glue", "mrpc")

tokenize_fn = build_tokenize_fn(tokenizer, max_len=128)
encoded_datasets = raw_datasets.map(tokenize_fn, batched=True)
```

（4）对 MRPC 做 map 预处理和列清理：这部分代码先用前面构造的 tokenize_fn 对 raw_datasets 进行 map(tokenize_fn, batched=True)，把每条 MRPC 的 (sentence1, sentence2) 都编码成了固定长度的 input_ids、attention_mask 等张量字段，相当于完成了“文本 → BERT 输入”的转换；随后通过 remove_columns(["sentence1", "sentence2", "idx"]) 删除不再需要的原始字符串列和索引列，只保留模型相关字段；然后用 rename_column("label", "labels") 把标签列改名为更通用的 "labels"，最后调用

set_format("torch")，让所有字段都自动转成 PyTorch Tensor。经过这一步，数据集已经是“可以直接丢进 DataLoader 再喂到模型里”的形式。

```
encoded_datasets = encoded_datasets.remove_columns(["sentence1", "sentence2", "idx
encoded_datasets = encoded_datasets.rename_column("label", "labels")
encoded_datasets.set_format("torch")
```

（5）DataLoader:构造 train_ds 和 valid_ds 后，使用 DataLoader 分别创建了训练和验证的迭代器：训练集的 DataLoader 设置了 batch_size=16, shuffle=True，表示每次从训练集里随机抽取 16 条样本组成一个 batch，用于打乱数据顺序、提升训练效果；验证集的 DataLoader 设为 shuffle=False，按固定顺序遍历验证集，用来在不打乱的前提下稳定评估模型表现。在训练/验证循环中，从这些 DataLoader 取出来的每个 batch 都是一个包含 input_ids、attention_mask 和 labels 等张量的字典，这些张量被移动到 GPU 或 CPU 上后直接输入模型进行前向计算。

```
train_ds = encoded_datasets["train"]
valid_ds = encoded_datasets["validation"]

batch_size = 16
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_ds, batch_size=batch_size, shuffle=False)
```

（6）模型、优化器和评价指标:在 DataLoader 之后，代码实例化了一个 BertMRPCClassifier 模型并移动到指定设备上，这个模型内部已经包含了预训练的 BERT 和两层 MLP 分类头；然后用 AdamW(model.parameters(), lr=5e-5) 创建优化器，指定了一个适合 BERT 微调的学习率，用于在训练中更新所有参数

```
model = BertMRPCClassifier(backbone_name=model_name).to(device)
optimizer = AdamW(model.parameters(), lr=5e-5)
```

（7）训练循环部分通过 for epoch in range(num_epochs) 多次遍历训练集，每一轮首先调用 model.train() 切换到训练模式，然后在进度条 tqdm(train_loader, desc="训练") 中依次取出 batch。对每个 batch，代码先用 {k: v.to(device) for k, v in batch.items()} 将输入和标签统一搬到 GPU/CPU 上，接着 optimizer.zero_grad() 清空上一轮的梯度，然后调用 model(...) 完成一次完整的前向传播，得到包含分类损失和 logits 的输出字典；取出其中的 loss，对它调用 loss.backward() 反向传播梯度，再执行 optimizer.step() 用

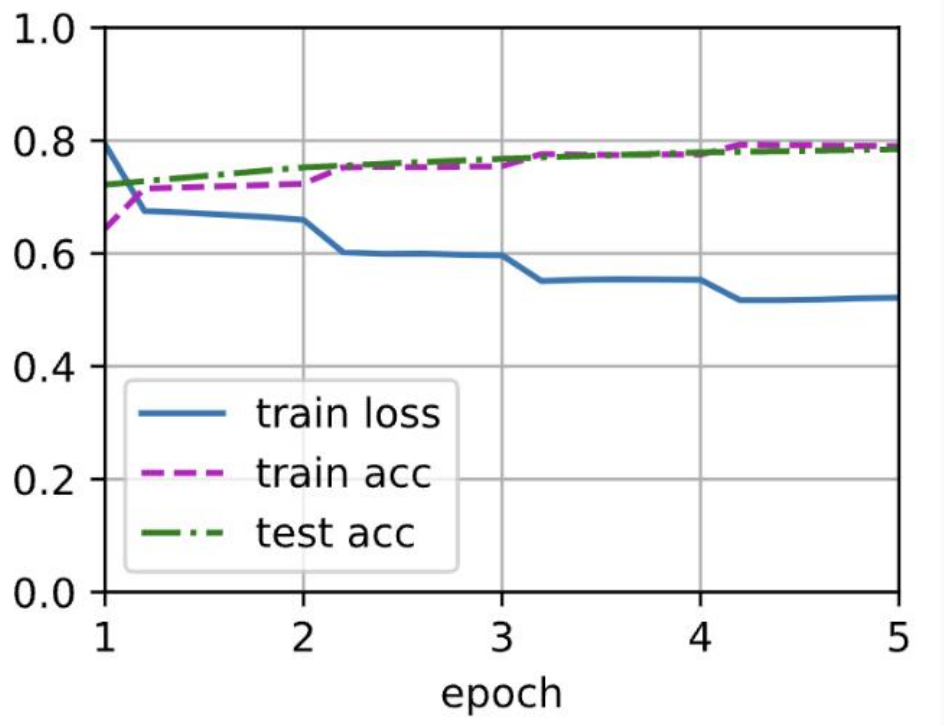
AdamW 更新模型参数；同时把每个 batch 的 loss 累加起来，最终除以 batch 数得到这一轮的平均训练损失，从而观察模型在训练过程中的收敛情况。整个过程就是典型的“对预训练 BERT + MLP 做端到端微调”的训练框架。

```
model.train()
running_loss = 0.0

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

    optimizer.zero_grad()
    out = model(
        input_ids=batch["input_ids"],
        attention_mask=batch["attention_mask"],
        labels=batch["labels"]
    )
    loss = out["loss"]
    loss.backward()
    optimizer.step()

    running_loss += loss.item()

avg_train_loss = running_loss / len(train_loader)
print(f"  训练集平均损失: {avg_train_loss:.4f}")
```

（8）在每个 epoch 的训练结束后，代码进入验证阶段：先调用 model.eval() 切换到推理模式，并在循环外把 eval_loss、all_preds 和 all_refs 清零；随后在 tqdm(valid_loader, desc="验证") 中遍历验证集的所有 batch，每个 batch 内部用 torch.no_grad() 禁用梯度计算以节省显存和加速推理，再像训练时一样将 batch 搬到设备上并调用 model(...) 得到损失和 logits。验证过程中不会做反向传播和参数更新，只是把每个 batch 的验证损失累加起来，同时对 logits 做 argmax 得到每条样本的预测类别，收集到 all_preds 中，与对应的真实标签一起存入 all_refs。整个验证集跑完后，先计算平均验证损失 avg_eval_loss，再调用 metric.compute(predictions=all_preds, references=all_refs) 用官方的 MRPC 指标计算准确率，并打印出来。这样就完整实现了“在验证集上进行推理并用标准评价指标衡量模型效果”的流程。

```python
model.eval()
eval_loss = 0.0
all_preds = []
all_refs = []

for batch in tqdm(valid_loader, desc="验证"):
    with torch.no_grad():
        batch = {k: v.to(device) for k, v in batch.items()}

        out = model(
            input_ids=batch["input_ids"],
            attention_mask=batch["attention_mask"],
            labels=batch["labels"]
        )
        loss = out["loss"]
        logits = out["logits"]

        eval_loss += loss.item()

        preds = torch.argmax(logits, dim=-1)
        all_preds.extend(preds.cpu().numpy())
        all_refs.extend(batch["labels"].cpu().numpy())

avg_eval_loss = eval_loss / len(valid_loader)
scores = metric.compute(predictions=all_preds, references=all_refs)

print(f"  验证集平均损失: {avg_eval_loss:.4f}")
print(f"  验证集准确率: {scores['accuracy']:.4f}")
```

将结果可视化，可得到：



结论分析与体会：
本次基于 BERT 的 MRPC 文本分类微调实验，让我对预训练语言模型的实际应用有了直观且深入的体会。也让我意识到实际工程中，批大小、学习率、训练轮次等超参数调优，以及数据预处理的细节（如截断、填充策略），都会直接影响模型性能，为后续更复杂的 NLP 任务实践积累了宝贵的经验。

注：实验报告的命名规则：学号_姓名_实验 n_班