人工智能实验报告

实验9　自然语言推理

陳日康　信息与计算科学　22336049

一、实验任务

使用给定数据集（QNLI 限制长度所得的子集）完成识别文本蕴含任务，训练过程中某次测试准确率达到 55% 以上（不要求 loss 收敛），分析运行时间和其他指标。

二、算法原理

自然语言推理（Natural Language Inference，NLI）是一种重要的自然语言处理任务，它的目的是判断两个文本之间的推理关系。即是给定一个前提（premise）和一个假设（hypothesis），NLI 模型需要判断前提是否支持假设、反驳假设或者与假设无关。NLI 算法通常通过以下几个步骤来实现：

1. **文本表示**：首先，需要将前提和假设转换为模型能够处理的表示形式。现代 NLI 算法通常使用预训练语言模型来将文本转化为高维向量表示。这些表示捕捉了文本中的语义信息和上下文关系。
2. **特征提取**：在获取了前提和假设的表示后，模型会提取更多的特征。常见的方法包括将前提和假设的向量表示进行拼接、点乘、减法等操作，生成一个包含两者关系的特征向量。
3. **推理模型**：接下来，将特征向量输入到一个分类模型中。这个分类模型通常是一个神经网络，如全连接层或者LSTM。模型的输出是三个类别的概率分布，分别表示 “支持”（entailment）、“反驳”（contradiction）和 “无关”（neutral）。
4. **损失函数和训练**：为了训练 NLI 模型，使用交叉熵损失函数来衡量模型预测的概率分布与真实标签之间的差异。通过反向传播和梯度下降算法，不断调整模型参数，使损失函数最小化。
5. **预测**：在训练完成后，模型可以对新的前提和假设对进行预测。根据输出的概率分布，选择概率最高的类别作为最终的推理结果。

通过 NLI 模型，可以更好地理解和处理自然语言中的逻辑关系，从而提高这些任务的性能和准确性。

三、流程图

```
开始
|
|---> 导入所需库
|       |
|       |---> 导入Numpy、PyTorch、Sklearn、Matplotlib等库
|
|---> 设置设备
|       |
|       |---> 使用GPU或CPU
|
|---> 数据准备阶段
|       |
|       |---> 读取数据
|       |       |
```

```
|       |       |---> 从文件中读取数据
|       |
|       |---> 构建词汇表
|       |       |
|       |       |---> 将文本转化为词汇表
|       |
|       |---> 载入预训练词向量
|       |       |
|       |       |---> 根据词汇表载入词向量
|       |
|       |---> 对标签进行编码
|               |
|               |---> 使用LabelEncoder编码标签
|
|---> 数据集和数据加载器
|       |
|       |---> 定义自定义数据集类
|       |
|       |---> 定义自定义collate函数
|       |
|       |---> 创建数据加载器
|
|---> 模型定义
|       |
|       |---> 定义LSTM分类器模型
|               |
|               |---> 嵌入层
|               |
|               |---> 双向LSTM层
|               |
|               |---> 全连接层
|
|---> 训练和验证
|       |
|       |---> 训练阶段
|       |       |
|       |       |---> 设置模型为训练模式
|       |       |
|       |       |---> 混合精度训练和梯度裁剪
|       |       |
|       |       |---> 记录训练损失和准确率
|       |
|       |---> 验证阶段
|               |
|               |---> 设置模型为评估模式
|               |
|               |---> 计算验证损失和准确率
|
|---> 结果展示
|       |
|       |---> 绘制训练准确率和损失曲线
|
结束
```

四、关键代码展示

`load_data` 函数用于从给定的文件路径读取数据，预期数据格式是四列（序号、查询、上下文和答案）。通过读取每行并分割，可以获取到查询、上下文和答案的列表，这是后续文本处理的基础。

```
def load_data(file_path):
    # 读取文件内容
    with open(file_path, 'r', encoding='utf-8') as file:
        lines = file.readlines()
    # 初始化三个列表用于存储查询、上下文和答案
    queries, contexts, answers = [], [], []
    for line in lines:
        parts = line.strip().split('\t')
        # 检查行是否包含期望的4个部分
        if len(parts) == 4:
            queries.append(parts[1])
            contexts.append(parts[2])
            answers.append(parts[3])
    return queries, contexts, answers
```

`build_vocab` 函数遍历所有文本数据，为每个独立的词分配一个唯一的索引。这个词汇表是文本数据转换为模型可以处理的数字格式的关键。

```
def build_vocab(sentences):
    vocab = {'<pad>': 0}  # 初始化词汇表，<pad>用于填充
    for sentence in sentences:
        for word in sentence.split():
            if word not in vocab:
                vocab[word] = len(vocab)  # 给每个新词分配一个新的索引
    return vocab
```

`load_pretrained_embeddings` 继承自 PyTorch 的 `Dataset` 类，负责将文本转换为索引序列，并处理标签，使其适合模型训练。这种处理是训练机器学习模型的标准步骤。

```
def load_pretrained_embeddings(file_path, vocab, emb_dim):
    embeddings = np.zeros((len(vocab), emb_dim))  # 初始化嵌入矩阵
    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            parts = line.split()
            word = parts[0]
            if word in vocab:
                idx = vocab[word]
                try:
                    vector = np.array(parts[1:], dtype=float)
                except ValueError:
                    vector = np.random.uniform(-0.25, 0.25, emb_dim)  # 随机初始化
无法解析的向量
                embeddings[idx] = vector
    return embeddings
```

`pad_collate` 在加载数据时，为了能够以批量方式处理，需要将不同长度的序列填充到相同长度。这个自定义函数确保所有文本数据在送入模型前具有统一的形状。

```
def pad_collate(batch):
    texts, labels = zip(*batch)
    # 对批次中的文本进行填充
    padded_texts = pad_sequence(texts, batch_first=True, padding_value=0)
    return padded_texts, torch.tensor(labels)
```

`LSTMClassifier` 定义了模型架构，包括嵌入层、双向 LSTM 层和全连接层。嵌入层使用预训练的 glove 词向量初始化，这有助于模型理解单词的语义，通常可以提高模型的学习效率和性能。双向 LSTM 允许模型不仅学习从前到后的语序信息，还能从后到前捕捉上下文，这通常能提高文本处理任务的准确率。全连接层将 LSTM 层的输出转换为最终分类标签，是生成预测结果的最后阶段。

```
# 定义LSTM分类器模型
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, emb_dim, hidden_dim, num_layers, num_classes,
embeddings):
        super(LSTMClassifier, self).__init__()
        # 嵌入层，使用预训练词向量
        self.embedding = nn.Embedding(vocab_size, emb_dim)
        self.embedding.weight.data.copy_(torch.from_numpy(embeddings))
        # LSTM层，双向
        self.lstm = nn.LSTM(emb_dim, hidden_dim, num_layers, batch_first=True,
bidirectional=True)
        # 全连接层，将LSTM的输出映射到分类结果
        self.fc = nn.Linear(hidden_dim * 2, num_classes)

    def forward(self, x):
        x = self.embedding(x)  # 嵌入层
        x, _ = self.lstm(x)  # LSTM层
        x = x[:, -1, :]  # 取最后一个时间步的输出
        x = self.fc(x)  # 全连接层
        return x
```

`train_and_evaluate` 封装了整个训练和验证流程，包括误差反向传播和参数更新。它还使用了自动混合精度来提高训练的效率和速度。

性能监控：通过记录每个 epoch 的训练准确率和损失，可以观察到模型训练的进展，这对于调整超参数和早期停止策略至关重要。

调度器：学习率调度器在训练过程中调整学习率，有助于模型在接近最优解时更细致地调整权重，通常能提高模型的最终性能。

```
# 训练与验证函数
def train_and_evaluate(model, train_loader, val_loader, criterion, optimizer,
scheduler, num_epochs=10):
    scaler = GradScaler()  # 用于混合精度训练
    train_accuracies, train_losses, epoch_times = [], [], []
```

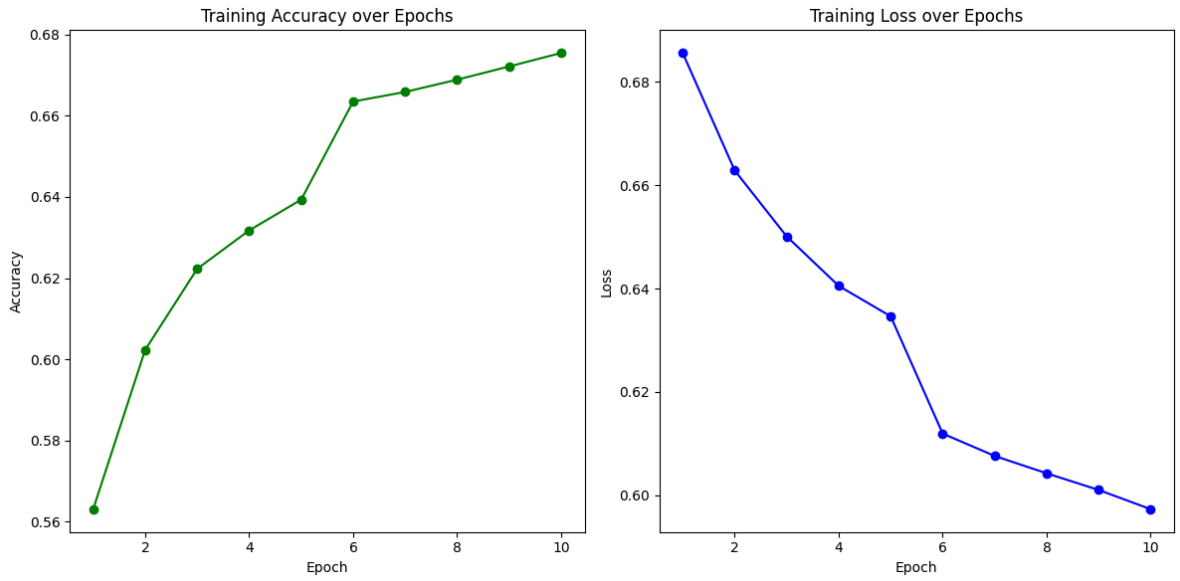
```
    for epoch in range(num_epochs):
        model.train()  # 模型设为训练模式
        start_time = time.time()
        epoch_loss, correct_predictions, total_samples = 0, 0, 0

        for texts, labels in train_loader:
            texts, labels = texts.to(device), labels.to(device)
            optimizer.zero_grad()
            with autocast():  # 使用自动混合精度
                outputs = model(texts)
                loss = criterion(outputs, labels)
            scaler.scale(loss).backward()  # 反向传播
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
# 梯度裁剪
            scaler.step(optimizer)
            scaler.update()

            epoch_loss += loss.item()
            _, predicted_labels = torch.max(outputs, 1)
            correct_predictions += (predicted_labels == labels).sum().item()
            total_samples += labels.size(0)

        epoch_accuracy = correct_predictions / total_samples
        train_accuracies.append(epoch_accuracy)
        train_losses.append(epoch_loss / len(train_loader))
        epoch_time = time.time() - start_time
        epoch_times.append(epoch_time)
        print(f'Epoch {epoch + 1}, Loss: {epoch_loss / len(train_loader):.4f},
Accuracy: {epoch_accuracy:.4f}, Time: {epoch_time:.2f}s')
        scheduler.step()  # 更新学习率

    # 验证阶段
    model.eval()
    val_loss, val_correct, val_total = 0, 0, 0
    with torch.no_grad():
        for texts, labels in val_loader:
            texts, labels = texts.to(device), labels.to(device)
            outputs = model(texts)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted_labels = torch.max(outputs, 1)
            val_correct += (predicted_labels == labels).sum().item()
            val_total += labels.size(0)

    val_accuracy = val_correct / val_total
    avg_val_loss = val_loss / len(val_loader)
    print(f'Validation Accuracy: {val_accuracy:.4f}, Validation Loss:
{avg_val_loss:.4f}')

    # 绘制训练准确率和损失曲线
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(range(1, num_epochs + 1), train_accuracies, 'go-')
    plt.title('Training Accuracy over Epochs')
    plt.xlabel('Epoch')
```

```
    plt.ylabel('Accuracy')

    plt.subplot(1, 2, 2)
    plt.plot(range(1, num_epochs + 1), train_losses, 'bo-')
    plt.title('Training Loss over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')

    plt.tight_layout()
    plt.show()

    return train_accuracies, train_losses, epoch_times
```

五、实验结果展示

5.1 实验结果展示示例

训练曲线：



输出结果：

```
Epoch 1, Loss: 0.6856, Accuracy: 0.5631, Time: 68.37s
Epoch 2, Loss: 0.6629, Accuracy: 0.6023, Time: 65.54s
Epoch 3, Loss: 0.6501, Accuracy: 0.6223, Time: 65.75s
Epoch 4, Loss: 0.6405, Accuracy: 0.6318, Time: 66.15s
Epoch 5, Loss: 0.6346, Accuracy: 0.6393, Time: 66.13s
Epoch 6, Loss: 0.6119, Accuracy: 0.6635, Time: 65.40s
Epoch 7, Loss: 0.6076, Accuracy: 0.6659, Time: 66.23s
Epoch 8, Loss: 0.6042, Accuracy: 0.6689, Time: 65.39s
Epoch 9, Loss: 0.6010, Accuracy: 0.6722, Time: 65.71s
Epoch 10, Loss: 0.5973, Accuracy: 0.6755, Time: 66.10s
Validation Accuracy: 0.6149, Validation Loss: 0.6613
```

由上图可见准确率达到 55% 以上。

5.2 评价指标展示及分析

1. 验证指标：

   在验证阶段，模型的验证准确率和验证损失如下：验证准确为 0.6149；验证损失是 0.6613。

2. 训练损失和准确率：

   从训练损失来看，随着训练周期的增加，损失值逐渐下降。这表明模型在训练集上的拟合能力不断提高。训练准确率也在逐渐提高，从最初的 56.31% 提高到 67.55%，显示出模型的学习效果逐步改善。

3. 验证损失与准确率：

   验证损失为 0.6613，相比于训练损失略高，这是预期的，因为验证集上的样本是模型未见过的，模型在这些样本上的表现通常会略差于训练集。验证准确率为 61.49%，相比训练准确率 67.55% 略低。这表明模型在验证集上的泛化能力还需要进一步提升。

4. 时间开销：

   每个训练周期大约需要 65 到 68 秒，总体来说时间开销相对稳定。这说明代码实现的效率较高，没有出现明显的性能瓶颈。

六、参考资料

实验课 ppt