中山大学计算机学院
人工智能
本科生实验报告

（2023 学年春季学期）

课程名称：Artificial Intelligence

教学班级	刘咏梅班级	专业（方向）	信息与计算科学
学号	22336313	姓名	郑鸿鑫

一、 实验题目

自然语言推理这个实验的主要任务是自然语言推理（NLI），特别是处理 Quora Question Pairs（QNLI）数据集。NLI 任务要求模型判断两个句子之间的逻辑关系，例如是否一个句子可以从另一个句子推理出来（entailment）。

二、 实验内容

1. 算法原理

主要的算法原理如下：

- **词向量表示**：使用 Gensim 的 Word2Vec 模型将词语转换为固定长度的向量。对每个句子进行分词，并将每个词转换为对应的词向量。如果词在词向量模型中不存在，则用零向量表示。

- **自定义数据集类**：创建一个 PyTorch 的数据集类 QNLIDataset，用于处理和加载数据。在数据集类中，实现了将句子分词并转换为词向量的功能。

- **LSTM 模型**：定义一个 LSTM 模型，该模型包括 LSTM 层和全连接层。LSTM 用于处理序列数据（句子中的词向量序列），捕捉词与词之间的依赖关系。最后的全连接层用于输出两个类别（entailment 和 not entailment）的预测。

2. 伪代码

```
Procedure Main
    # 下载 NLTK 需要的资源
    DownloadNLTKResources('punkt')
    # 预处理数据文件，移除多余的逗号
    PreprocessFile('path_to_train_data', 'path_to_train_data_clean')
    PreprocessFile('path_to_dev_data', 'path_to_dev_data_clean')
    # 读取预处理后的训练和验证数据
    TrainData <- ReadCSV('path_to_train_data_clean', sep='\t', names=['index', 'sentence',
'question', 'label'])
    DevData <- ReadCSV('path_to_dev_data_clean', sep='\t', names=['index', 'sentence',
```

```
'question', 'label'])
    # 训练 Word2Vec 模型并创建句子的词向量
    Sentences <- Tokenize(TrainData['sentence'] + TrainData['question'])
    Word2VecModel <- TrainWord2Vec(Sentences, vector_size=300, window=5,
min_count=1, workers=4)
    # 设置最大句子长度和创建数据加载器
    MaxLen <- 50
    TrainDataset <- QNLIDataset(TrainData, Word2VecModel, MaxLen)
    DevDataset <- QNLIDataset(DevData, Word2VecModel, MaxLen)
    TrainLoader <- DataLoader(TrainDataset, batch_size=32, shuffle=True)
    DevLoader <- DataLoader(DevDataset, batch_size=32, shuffle=False)
    # 初始化 LSTM 模型参数
    InputSize <- 300
    HiddenSize <- 128
    NumLayers <- 2
    NumClasses <- 2
    # 定义并初始化 LSTM 模型、损失函数、优化器
    Model <- LSTMModel(InputSize, HiddenSize, NumLayers, NumClasses)
    CrossEntropyLoss <- nn.CrossEntropyLoss()
    AdamOptimizer <- optim.Adam(Model.parameters(), lr=0.001)
    # 训练模型
    For Epoch From 1 To NumEpochs
        Model.train()
        For Batch Of TrainLoader
            Premises, Hypotheses, Labels <- Batch
            Premises, Hypotheses, Labels <- ConvertToTensor(Premises, Hypotheses,
Labels).to(device)
            Outputs <- Model(Concatenate(Premises, Hypotheses))
            Loss <- CrossEntropyLoss(Outputs, Labels)
            AdamOptimizer.zero_grad()
            Loss.backward()
            AdamOptimizer.step()
        EndFor
    EndFor
    # 评估模型
    Model.eval()
    AllPredictions, AllLabels <- [], []
    For Batch Of DevLoader
        Premises, Hypotheses, Labels <- Batch
        Premises, Hypotheses, Labels <- ConvertToTensor(Premises, Hypotheses,
Labels).to(device)
        Outputs <- Model(Concatenate(Premises, Hypotheses))
        Predictions <- torch.max(Outputs.data, 1)
        AllPredictions.extend(Predictions.cpu().numpy())
        AllLabels.extend(Labels.cpu().numpy())
    EndFor
    Accuracy <- CalculateAccuracy(AllLabels, AllPredictions)
    Print(f'Accuracy: {Accuracy:.2f}')
    Print("Training complete!")
EndProcedure
```

3. 关键代码展示

下载必要资源和数据的预处理部分代码：

```
# 下载 nltk 必要资源
nltk.download('punkt')#下载 NLTK 的 punkt 模块，该模块用于句子和单词的分割
# 定义数据预处理函数
def preprocess_file(input_file, output_file):
    with open(input_file, 'r', encoding='utf-8') as f:
        lines = f.readlines()
    with open(output_file, 'w', encoding='utf-8') as f:
        for line in lines:
            # 移除行内多余的逗号
            cleaned_line = re.sub(r'(?<!"),(?!")', '', line)
            f.write(cleaned_line)
# 预处理训练和验证数据文件
preprocess_file('C:\\Users\\26618\\Desktop\\人工智能实验\\实验
9\\QNLI\\train_40.tsv',
                'C:\\Users\\26618\\Desktop\\人工智能实验\\实验
9\\QNLI\\train_40_clean.tsv')
preprocess_file('C:\\Users\\26618\\Desktop\\人工智能实验\\实验
9\\QNLI\\dev_40.tsv',
                'C:\\Users\\26618\\Desktop\\人工智能实验\\实验
9\\QNLI\\dev_40_clean.tsv')
# 读取数据，设置 on_bad_lines='skip' 忽略有问题的行
train_data = pd.read_csv('C:\\Users\\26618\\Desktop\\人工智能实验\\实验
9\\QNLI\\train_40_clean.tsv', sep='\t',
                         names=["index", "sentence", "question", "label"],
on_bad_lines='skip')
dev_data = pd.read_csv('C:\\Users\\26618\\Desktop\\人工智能实验\\实验
9\\QNLI\\dev_40_clean.tsv', sep='\t',
                       names=["index", "sentence", "question", "label"],
on_bad_lines='skip')
```

数据集类的代码实现：

```
# 定义自定义数据集类，用于预处理和词嵌入
class QNLIDataset(Dataset):
    def __init__(self, data, word2vec, max_len):
        #data 是传入的数据，word2vec 是用于词向量转化的模型，max_len 是设定好
句子的最大长度
        self.data = data
        self.word2vec = word2vec
        self.max_len = max_len
    def __len__(self):
        return len(self.data)#返回样本的数量
    def __getitem__(self, idx):
        row = self.data.iloc[idx]
```

```
        # 处理句子和问题，转换为词向量
        premise = self.process_sentence(row['sentence'])
        hypothesis = self.process_sentence(row['question'])
        label = 1 if row['label'] == 'entailment' else 0
        return premise, hypothesis, label
    #getitem 函数根据索引返回一条数据，并进行处理，返回句子及其对应的标签
    def process_sentence(self, sentence):
        tokens = word_tokenize(sentence.lower()) #将字母小写并分词，得到词
列表。
        # 将每个词转换为词向量，如果词不存在于词向量模型中，则用零向量表示
        vecs = [self.word2vec.wv[token] if token in self.word2vec.wv
else np.zeros(300) for token in tokens]
        # 处理后的词向量长度固定为 max_len，不足则补零，超出则截断。
        vecs = vecs[:self.max_len] + [np.zeros(300)] * (self.max_len -
len(vecs))
        return np.array(vecs)
    #该方法将句子进行分词，并将每个词词转换为对应的词向量，最终返回一个固定长度
的词向量数组
```

定义 LSTM 模型的代码实现：

```
# 定义 LSTM 模型，用于处理句子对并进行分类
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
num_classes):
        super(LSTMModel, self).__init__()
        # 定义 LSTM 层
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
        # 定义全连接层
        self.fc = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
        # 初始化 LSTM 的初始隐藏状态和细胞状态
        h0 = torch.zeros(num_layers, x.size(0), hidden_size).to(device)
        c0 = torch.zeros(num_layers, x.size(0), hidden_size).to(device)
        # LSTM 前向传播
        out, _ = self.lstm(x, (h0, c0))
        # 取 LSTM 最后一个时间步的输出，输入到全连接层
        out = self.fc(out[:, -1, :])
        return out
```

模型的初始化及训练：

```
# 初始化模型、损失函数和优化器
model = LSTMModel(input_size, hidden_size, num_layers,
```

```
num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
# 训练模型
for epoch in range(num_epochs):
    model.train()  # 设定模型为训练模式
    for premises, hypotheses, labels in tqdm(train_loader):
        # 将输入和标签转换为 tensor 并移动到设备（CPU 或 GPU）
        premises = premises.clone().detach().float().to(device)
        hypotheses = hypotheses.clone().detach().float().to(device)
        labels = labels.clone().detach().long().to(device
        # 前向传播
        outputs = model(torch.cat((premises, hypotheses), dim=1))
        loss = criterion(outputs, labels)
        # 反向传播和优化
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    model.eval()  # 设定模型为评估模式
    all_preds = []
    all_labels = []
    with torch.no_grad():  # 禁用梯度计算
        for premises, hypotheses, labels in dev_loader:
            # 将输入和标签转换为 tensor 并移动到设备（CPU 或 GPU）
            premises = premises.clone().detach().float().to(device)
            hypotheses = hypotheses.clone().detach().float().to(device)
            labels = labels.clone().detach().long().to(device)
            # 前向传播
            outputs = model(torch.cat((premises, hypotheses), dim=1))
            _, predicted = torch.max(outputs.data, 1)
            # 收集预测结果和真实标签
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    # 计算验证集上的准确率
    accuracy = accuracy_score(all_labels, all_preds)
    print(f'Epoch [{epoch + 1}/{num_epochs}], Accuracy: {accuracy:.2f}')
```

4. 创新点&优化（如果有）

加入 Attention 机制进行训练。

加入正则化手段如 droppout 防止过拟合

调整学习率，隐藏层大小等参数

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

见下面的分析

2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

首先使用上文代码展示中的模型来训练并测试得到结果如下：

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\26618\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
100%|█████████████| 2261/2261 [00:50<00:00, 44.53it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [1/10], Accuracy: 0.53
100%|█████████████| 2261/2261 [00:51<00:00, 44.33it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [2/10], Accuracy: 0.53
100%|█████████████| 2261/2261 [00:52<00:00, 43.25it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [3/10], Accuracy: 0.53
100%|█████████████| 2261/2261 [00:51<00:00, 43.53it/s]
Epoch [4/10], Accuracy: 0.53
100%|█████████████| 2261/2261 [00:51<00:00, 43.50it/s]
Epoch [5/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:52<00:00, 43.17it/s]
Epoch [6/10], Accuracy: 0.63
100%|█████████████| 2261/2261 [00:52<00:00, 42.96it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [7/10], Accuracy: 0.63
100%|█████████████| 2261/2261 [00:58<00:00, 38.64it/s]
Epoch [8/10], Accuracy: 0.64
100%|█████████████| 2261/2261 [00:53<00:00, 42.41it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [9/10], Accuracy: 0.65
100%|█████████████| 2261/2261 [00:50<00:00, 44.49it/s]
Epoch [10/10], Accuracy: 0.64
Training complete!
```

可以看到最后的结果可以达到 65%的准确度，然后我们尝试加入 Attention 机制来训练模型：

代码如下所示：

```
# 定义带有 Attention 机制的 LSTM 模型
class LSTMWithAttention(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
num_classes):
        super(LSTMWithAttention, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
        self.attention = nn.Linear(hidden_size, 1)
        self.fc = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
        h0 = torch.zeros(num_layers, x.size(0), hidden_size).to(device)
        c0 = torch.zeros(num_layers, x.size(0), hidden_size).to(device)
```

```
        out, _ = self.lstm(x, (h0, c0))
        # Apply attention
        attn_weights = torch.softmax(self.attention(out), dim=1)
        attn_applied = torch.bmm(attn_weights.transpose(1, 2), out)
        out = self.fc(attn_applied.squeeze(1))
        return out
```

加入 Attention 机制后得到结果如下所示：

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\26618\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
100%|█████████████| 2261/2261 [00:59<00:00, 37.96it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [1/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:50<00:00, 44.70it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [2/10], Accuracy: 0.63
100%|█████████████| 2261/2261 [00:50<00:00, 44.42it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [3/10], Accuracy: 0.63
100%|█████████████| 2261/2261 [00:53<00:00, 42.50it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [4/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:54<00:00, 41.85it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [5/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:54<00:00, 41.75it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [6/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:53<00:00, 42.08it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [7/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:53<00:00, 41.99it/s]
Epoch [8/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:54<00:00, 41.79it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [9/10], Accuracy: 0.61
100%|█████████████| 2261/2261 [00:55<00:00, 40.47it/s]
Epoch [10/10], Accuracy: 0.60
Training complete!
```

我们可以看到，加入了 Attention 机制后，模型的稳定性好了不少，基本维持在 60%以上，但是加入后准确率的最大值反而下降，不加 attention 时最高能达到 65%，但是加上以后最高只达到 63%。

因此加上定期衰减的学习率来训练：

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)  # 每 5 个 epoch 后学习率减半
```

得到结果如下：

```
D:\Aconda\envs\torch\python.exe D:\Code\Python\Lab9\NLP\NLP.py
100%|█████████████| 2261/2261 [00:54<00:00, 41.46it/s]
Epoch [1/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:55<00:00, 40.80it/s]
Epoch [2/10], Accuracy: 0.63
100%|█████████████| 2261/2261 [00:55<00:00, 40.45it/s]
Epoch [3/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:55<00:00, 40.48it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [4/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:56<00:00, 40.37it/s]
Epoch [5/10], Accuracy: 0.63
100%|█████████████| 2261/2261 [00:55<00:00, 40.48it/s]
Epoch [6/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:55<00:00, 40.84it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [7/10], Accuracy: 0.61
100%|█████████████| 2261/2261 [00:55<00:00, 40.93it/s]
Epoch [8/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:55<00:00, 41.01it/s]
  0%|             | 0/2261 [00:00<?, ?it/s]Epoch [9/10], Accuracy: 0.62
100%|█████████████| 2261/2261 [00:54<00:00, 41.79it/s]
Epoch [10/10], Accuracy: 0.61
Training complete!
```

可以看到一样可以保持比较稳定的准确率，最高准确率仍然是 63%，但是平均准确率比上一版本升高了一些。

四、 参考资料

1）自然语言推理-文本蕴含识别简介_识别文本蕴含数据集-CSDN 博客

2）解决：pandas.errors.ParserError: Error tokenizing data. C error: Expected 2 fields in line 18, saw 4-CSDN 博客

3）pandas.read_csv 参数详解 - 李旭 sam - 博客园 (cnblogs.com)