中山大學

中山大学计算机学院人工智能

实验报告

题　　目	:	自然语言推理
教学班级	:	20230349
姓　　名	:	张超
学　　号	:	22336290
专　　业	:	计算机科学与技术（系统结构）
日　　期	:	2024/06/20

实验题目

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

实验内容

LSTM算法原理

长短期记忆网络（LSTM）是一种特殊类型的递归神经网络（Recurrent Neural Network, RNN），旨在解决标准RNN在处理长序列数据时容易遗忘早期信息的问题。LSTM通过引入一系列门控机制，能够更有效地捕捉和保持长时间跨度的信息。以下是LSTM的算法原理：

1. LSTM单元的组成

一个典型的LSTM单元由以下部分组成：

- **细胞状态（Cell State）**：表示单元的记忆，通过时间步长传递信息。
- **输入门（Input Gate）**：决定当前输入信息对细胞状态的影响程度。
- **遗忘门（Forget Gate）**：决定细胞状态中哪部分信息需要被遗忘。
- **输出门（Output Gate）**：决定当前单元输出什么信息。

2. LSTM单元的计算

在时间步t，一个LSTM单元接收输入(x_t)和上一个时间步的隐藏状态(h_{t-1})以及细胞状态(C_{t-1})。以下是详细的计算步骤：

2.1 遗忘门

遗忘门决定细胞状态中哪些部分将被遗忘。它的计算公式为：

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

其中，W_f 和 b_f 分别是遗忘门的权重矩阵和偏置向量，(σ) 是 sigmoid 函数。

2.2 输入门和候选细胞状态

输入门控制新信息的写入。它包括两个部分：输入门和候选细胞状态。

- 输入门的计算公式为：

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

其中，W_i 和 b_i 分别是输入门的权重矩阵和偏置向量。

- 候选细胞状态的计算公式为：

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

其中，W_C 和 b_C 分别是候选细胞状态的权重矩阵和偏置向量，$\tanh$ 是双曲正切函数。

2.3 更新细胞状态

通过结合遗忘门和输入门的信息来更新细胞状态：

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

2.4 输出门和隐藏状态

输出门决定哪些部分的细胞状态将被输出：

- 输出门的计算公式为：

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

其中，(W_o) 和 (b_o) 分别是输出门的权重矩阵和偏置向量。

- 隐藏状态的更新公式为：

$$h_t = o_t \cdot \tanh(C_t)$$

3. 总结

通过上述步骤，LSTM单元能够有效地在长时间序列中捕捉和保留关键信息。其核心在于利用门控机制（遗忘门、输入门和输出门）来控制信息的流动和存储，从而克服标准RNN的长期依赖问题。

LSTM的整体架构

LSTM网络由多个LSTM单元按照时间步长依次连接而成，形成一个完整的时间序列模型。输入序列通过这些单元进行处理，最终输出预测结果或分类结果。

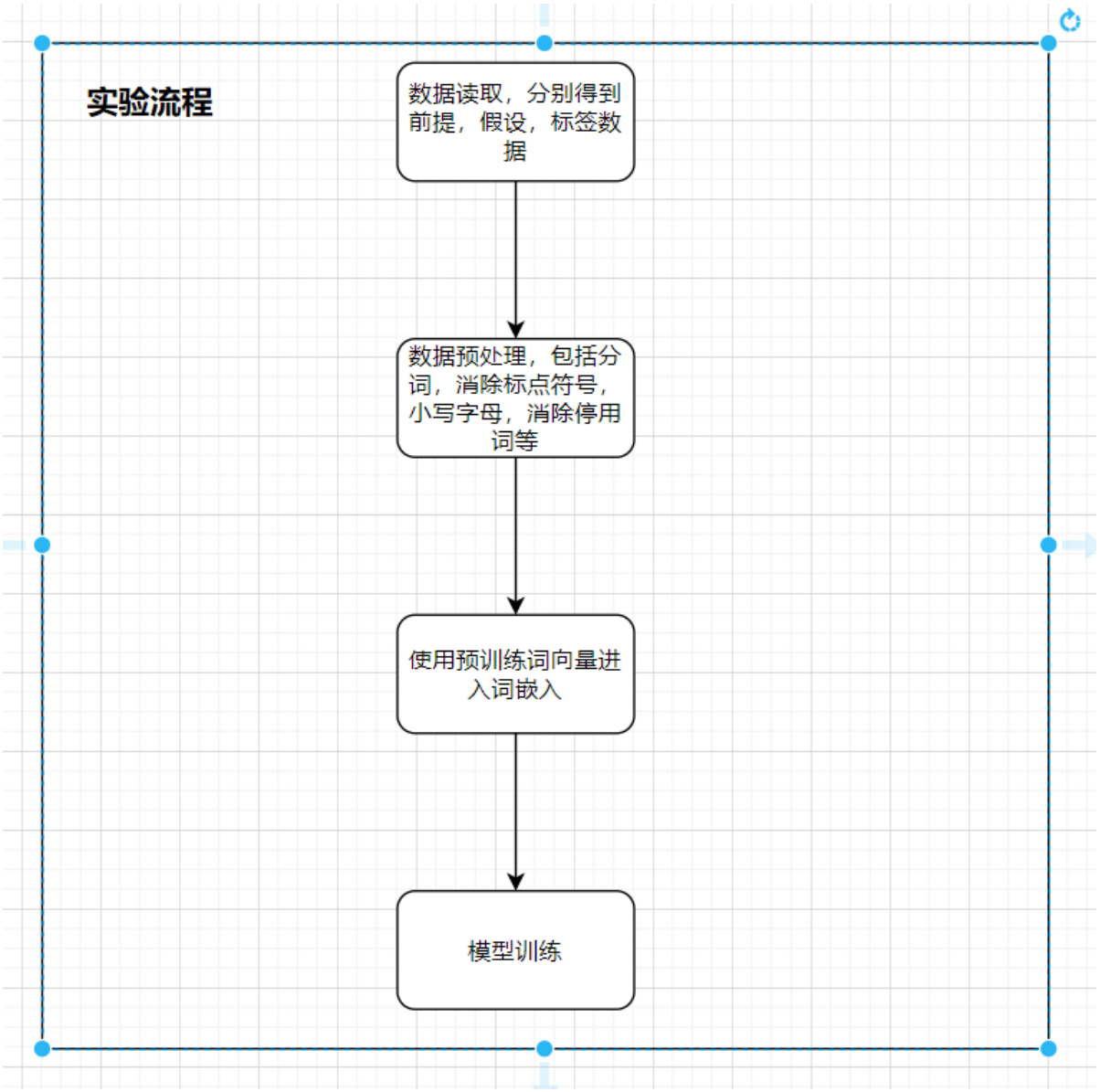
这种设计使LSTM在处理长时间依赖问题上表现优异，广泛应用于自然语言处理（如文本生成、翻译）、时间序列预测（如股票预测、天气预报）等领域。

流程图

模型架构

网络层号	网络层类型	网络层具体参数
第一层	双向LSTM（带dropout）	输入维度100，输出维度128
第二层	全连接层（带激活函数，dropout）	输入样本大小1024，输出样本大小128
第三层	全连接层（带激活函数，dropout）	输入样本大小128，输出样本大小128
第四层	全连接层（不带激活函数和dropout）	输入样本大小128，输出样本大小2

实验流程



关键代码展示

数据读取

```
def get_data(train):
    index = []
    question = []
    sentence = []
    label =[]
    if train:                                                    #区分数据集
和训练集
        path = "lab9\\train_40.tsv"
    else:
        path = "lab9\\dev_40.tsv"
    with open(path,encoding='utf-8') as f:
        next(f)
        for line in f:
            line = line.strip('\n').split('\t')                  #按'\t'划
分数据
            index.append(line[0])
```

```
            question.append(line[1])
            sentence.append(line[2])
            label.append(line[3])
    label_dict = {'entailment':0,'not_entailment':1}
    label = [label_dict[_] for _ in label]                                  #将标签转为
布尔值
    return index,question,sentence,torch.LongTensor(label).to('cuda')
```

数据预处理

```
def process_data(lines,vocab_):
    tokenizer  = get_tokenizer('basic_english')                 #分词器
    stop_words = set(stopwords.words('english'))                #停用词集合
    lines = [line.replace(',','').replace('?','').replace('.','').replace(':','')
for line in lines] #删除标点符号
    processed_lines = [torch.LongTensor([vocab_[word.lower()] for word in
tokenizer(line) if (not word.lower() in stop_words) and word.isalnum()]) for line
in lines] #分词并删除停用词
    return processed_lines
```

模型构建

```
class NLI(nn.Module):
    def __init__(self,embedding_matrix,embedding_dim, hidden_dim, output_dim):
        super().__init__()
        self.embedding =
nn.Embedding.from_pretrained(embedding_matrix,freeze=False)
        self.lstm =
nn.LSTM(embedding_dim,hidden_size=hidden_dim,num_layers=2,batch_first=True,bidire
ctional=True)
        self.fc1 = nn.Linear(hidden_dim*4,hidden_dim)
        self.fc2 = nn.Linear(hidden_dim,128)
        self.fc3 = nn.Linear(128,output_dim)
        self.dropout = nn.Dropout(0.2)
    def forward(self,premise,hypothesis):
        embedded_premise = self.embedding(premise)
   # 前提词嵌入
        embedded_hypothesis = self.embedding(hypothesis)
   # 假设词嵌入
        # LSTM
        _, (hidden_premise, _) = self.lstm(embedded_premise)
   #最终时间步的隐藏状态
        _, (hidden_hypothesis, _) = self.lstm(embedded_hypothesis)

        hidden_premise = torch.cat((hidden_premise[-2,:,:],
hidden_premise[-1,:,:]), dim=1)          #拼接前向和后向的隐藏状态
        hidden_hypothesis = torch.cat((hidden_hypothesis[-2,:,:],
hidden_hypothesis[-1,:,:]), dim=1)

        combined = torch.cat((hidden_premise, hidden_hypothesis), dim=1)
   #拼接前提和假设
        combined = self.dropout(nn.functional.relu(self.fc1(combined)))
  # 全连接
```

```
        combined = self.dropout(nn.functional.relu(self.fc2(combined)))
    # 全连接
        return self.fc3(combined)
```

实验结果展示及分析

实验结果展示示例

```
[1:71168]---准确率: 58.98%      loss: 0.6737
[1:71680]---准确率: 58.59%      loss: 0.6702
[1:72192]---准确率: 58.20%      loss: 0.6689
[1:72704]---准确率: 58.01%      loss: 0.6673
[1:73216]---准确率: 60.35%      loss: 0.6621
[1:73728]---准确率: 56.45%      loss: 0.6787
[1:74240]---准确率: 55.52%      loss: 0.6761
--------test begin--------
QNLI准确率: 58.75%
```

评测指标分析

进行了多次实验并修改了随机种子，对测试集的准确率都在55%之上。

并且修改超参数epochs，可以大大提高在训练集上的准确率，loss也更加低，但是在测试集上效果比较普通

如图为epochs=5的训练结果：

```
[5:71168]---准确率: 80.47%      loss: 0.3687
[5:71680]---准确率: 79.10%      loss: 0.4174
[5:72192]---准确率: 81.84%      loss: 0.3726
[5:72704]---准确率: 78.32%      loss: 0.3933
[5:73216]---准确率: 81.64%      loss: 0.3747
[5:73728]---准确率: 78.71%      loss: 0.4550
[5:74240]---准确率: 82.47%      loss: 0.3826
--------test begin--------
QNLI准确率: 56.61%
```