中山大学计算机学院人工智能本科生实验报告（2022学年春季学期）

课程名称：Artificial Intelligence

教学班级	专业（方向）	学号	姓名
2班	计算机科学与技术	21307174	刘俊杰

一、实验题目

Week 16 自然语言推理任务

二、实验内容

实验内容：

- 使用给定数据集（QNLI限制长度所得的子集）进行训练和验证。
- 数据集四列使用'\t'分开

实验要求：

- 对文本进行embedding
- 使用lstm（或者更复杂的模型）进行分类
- 准确率达到55%以上（不要求loss收敛，训练过程中某次评估达标即可）
- 分析运行时间

基准参考：

- 设计：使用glove.6B.50d.txt，直接对齐所有数据，batch_size=32，torch.nn.LSTM设置隐藏层为64，后接nn.Linear(64,2)，使用交叉熵计算损失函数。
- 准确率：5-10个epoch能达到57%以上，每个epoch用时10s左右。（此处每个epoch指将整个训练集分成的N/batch_size组全部训练一次）

1. 算法原理

自然语言推理是一种涉及理解和推理两个句子之间关系的任务。结合LSTM和词嵌入可以帮助实现自然语言推理，并提高模型的性能。

首先，词嵌入技术可以将句子中的每个单词转换为连续向量表示。训练好的词嵌入模型可以为每个单词学习到具有上下文相似性的词向量。这些词向量捕捉了单词的语义特征，可以用来表示句子中的单词。

(本实验，将两个句子分别作为模型的输入：前提和假设。通过词嵌入，可以将每个句子中的单词转换为对应的词向量。)

然后创建lstm的模型，定义前向传播

(LSTM（Long Short-Term Memory，长短期记忆）是一种循环神经网络（RNN）的变体，特别适用于处理序列数据。在文本分类任务中，我们可以使用LSTM模型来捕捉句子中的上下文信息。LSTM模型由多个LSTM单元组成，每个LSTM单元都有输入门、遗忘门和输出门。通过这些门控制机制，LSTM能够选择性地保留或遗忘一些信息，并传递给下一个时间步。）

每轮训练在训练集训练LSTM模型，在测试集上通过LSTM模型预测前提和假设之间的关系。

以下为实验具体步骤

(1)构建词嵌入的词表

通过读取glove.6B.50d.txt，获得已经可以使用的词表。

(2)数据预处理：

首先，加载并读取文本数据集。然后，对于每个样本，将premise（前提）和hypothesis（假设）两个句子合并成一个句子。为了统一输入的长度，对句子进行填充或截断操作，使其具有相同的长度。最后词嵌入，通过词表将文本数据转换为数值表示形式，如将每个单词映射到一个词向量。

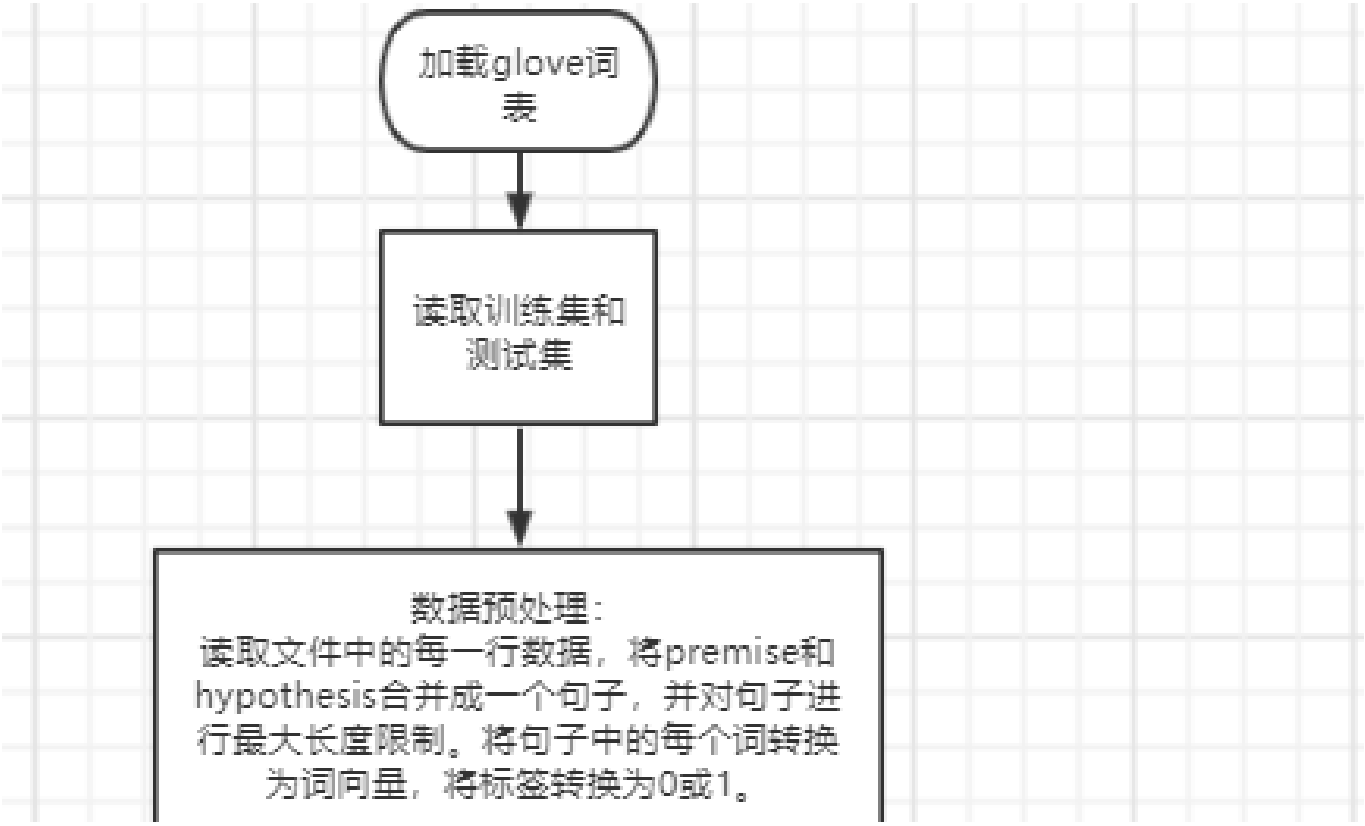
(3)构建LSTM模型：

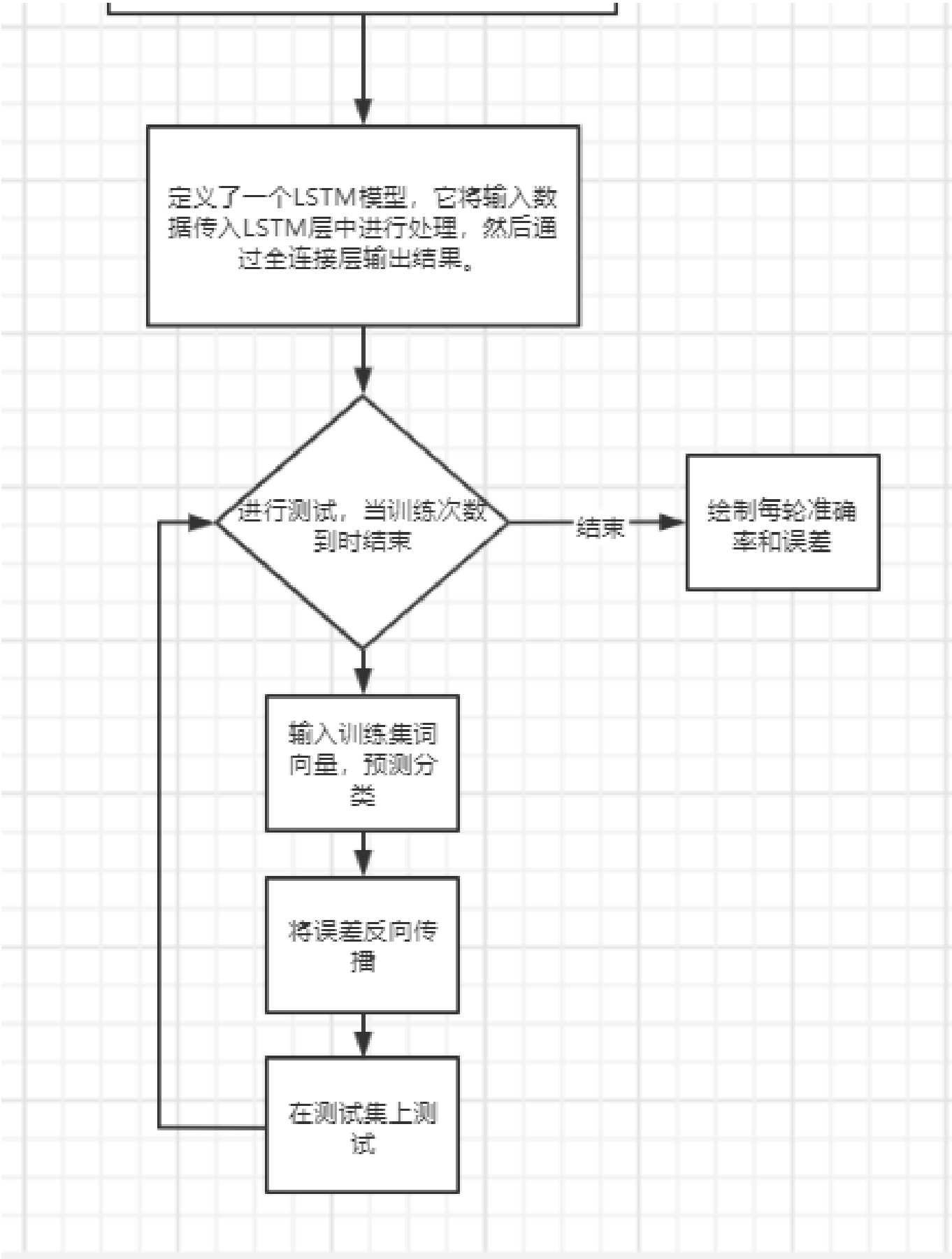
构建LSTM层的分类模型，定义好前向传播，模型的输入是文本数据的词向量表示，经过LSTM层的处理后，输出结果传递给全连接层进行分类。

(4)训练模型并进行测试

(5)绘制每轮的准确率和损失

2.流程图





3.关键代码展示

(1)词嵌入准备:加载GloVe词向量模型

```
embedding_model = {}
with open('E13\glove.6B.50d.txt', 'r', encoding='utf-8') as f:
    for line in f:
        values = line.strip().split()
        word = values[0]
        vector = np.array(values[1:], dtype='float32')
        embedding_model[word] = vector
embedding_dim = len(embedding_model[next(iter(embedding_model.keys()))])
```

(2)定义一个自定义数据集类，用于加载并预处理数据集

首先，将前提句子和假设句子分别转换为由单词组成的列表，并符将它们合并成一个句子。为了使得句子的长度一致，将其限制在max_length之内。

接下来，每个单词，将其转换为对应的词向量。如果单词存在于embedding_model中，转为对应的词向量；否则，会使用维度相同的零向量。

为了保证输入的句子长度统一，对句子进行填充（padding）。如果句子长度小于max_length，会将零向量添加到句子末尾，直到达到max_length长度。

```
class QNLIDataset(Dataset):
    def __init__(self, file_path, max_length, embedding_model):
        self.sentences = []
        self.labels = []
        self.max_length = max_length
        self.embedding_dim =
len(embedding_model[next(iter(embedding_model.keys()))])

        with open(file_path, 'r', encoding='utf-8') as f:
            lines = f.readlines()

        for line in lines[1:]:
                # 读取每一行数据，格式为：premise sentence  hypothesis sentence
label
                items = line.strip().split('\t')
                premise = items[1].split()
                hypothesis = items[2].split()
                # 将premise和hypothesis合并成一个句子
                sentence = premise + hypothesis
                # 限制句子最大长度为max_length
                sentence = sentence[:self.max_length]
                # 将句子中的每个词转换为词向量

                sentence_vectors = []
                for word in sentence:
                    if word in embedding_model:
                        sentence_vectors.append(embedding_model[word])
                    if len(sentence_vectors) == 0:
                        sentence_vectors.append(np.zeros(self.embedding_dim))
                # 将句子的长度补齐到max_length，并将句子向量作为样本
```

```
                while len(sentence_vectors) < self.max_length:
                    sentence_vectors.append(np.zeros_like(sentence_vectors[0]))
                self.sentences.append(np.array(sentence_vectors))
                # 将标签转换为0或1
                self.labels.append(int(items[3] == 'entailment'))

    def __len__(self):
        return len(self.sentences)

    def __getitem__(self, idx):
        return torch.from_numpy(self.sentences[idx]).float(),
torch.tensor(self.labels[idx])
```

(3)定义LSTM模型和前向传播

创建一个LSTM层连接一个全连接层，用于将LSTM的最后一个隐藏状态映射到结果类别。

```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.lstm.num_layers, x.size(0), self.lstm.hidden_size)
        c0 = torch.zeros(self.lstm.num_layers, x.size(0), self.lstm.hidden_size)
        out, _ = self.lstm(x, (h0, c0))
        out = out[:, -1, :]
        out = self.fc(out)
        return out
```

(4)训练及测试

```
# 定义训练函数和验证函数
def train(model, data_loader):
    model.train()
    running_loss = 0.
    correct = 0
    total = 0
    for inputs, labels in data_loader:
        inputs = inputs
        labels = labels
        optimizer.zero_grad()
        #预测
        outputs = model(inputs)
        #计算损失
        loss = criterion(outputs, labels)
```

```
            #反向传播损失
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * inputs.size(0)
            #计算在训练集上的准确率
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        epoch_loss = running_loss / len(data_loader.dataset)
        epoch_acc = correct / total
        return epoch_loss, epoch_acc

    def evaluate(model, data_loader):
        model.eval()
        running_loss = 0.
        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in data_loader:
                inputs = inputs
                labels = labels
                #预测
                outputs = model(inputs)
                #计算损失
                loss = criterion(outputs, labels)
                running_loss += loss.item() * inputs.size(0)
                #计算准确率
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
        epoch_loss = running_loss / len(data_loader.dataset)
        epoch_acc = correct / total
        return epoch_loss, epoch_acc
```

4.创新点&优化

(1)预处理操作

为了使得句子的长度一致，将其限制在max_length之内。

超过max_length进行截取，小于的对句子进行填充（padding）。如果句子长度小于max_length，会将零向量添加到句子末尾，直到达到max_length长度。

(2)dropout避免过拟合

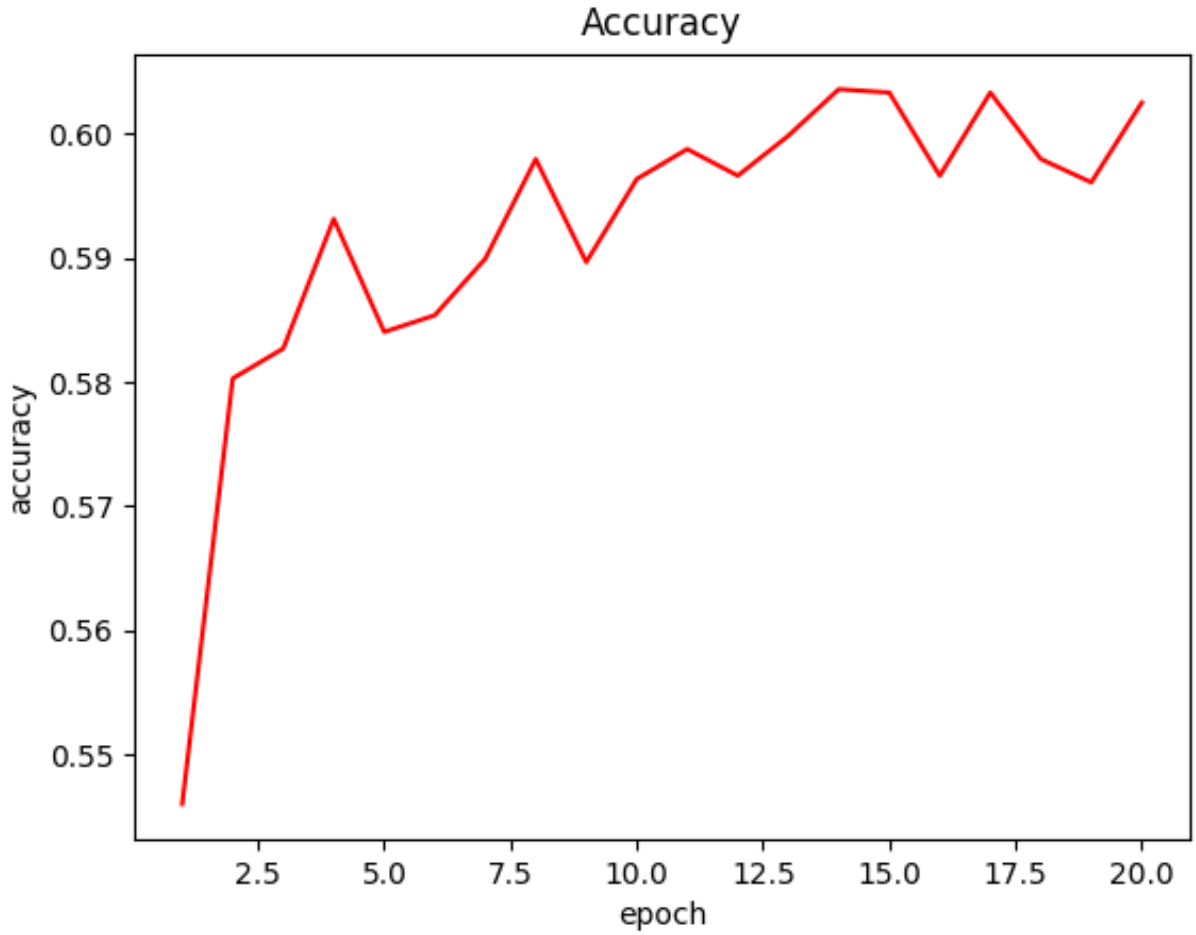
询问有的同学，他们的代码出现了过拟合，使用了dropout避免

虽然我的代码没出现过拟合，但也在模型中添加了dropout层，比较两者的性能。

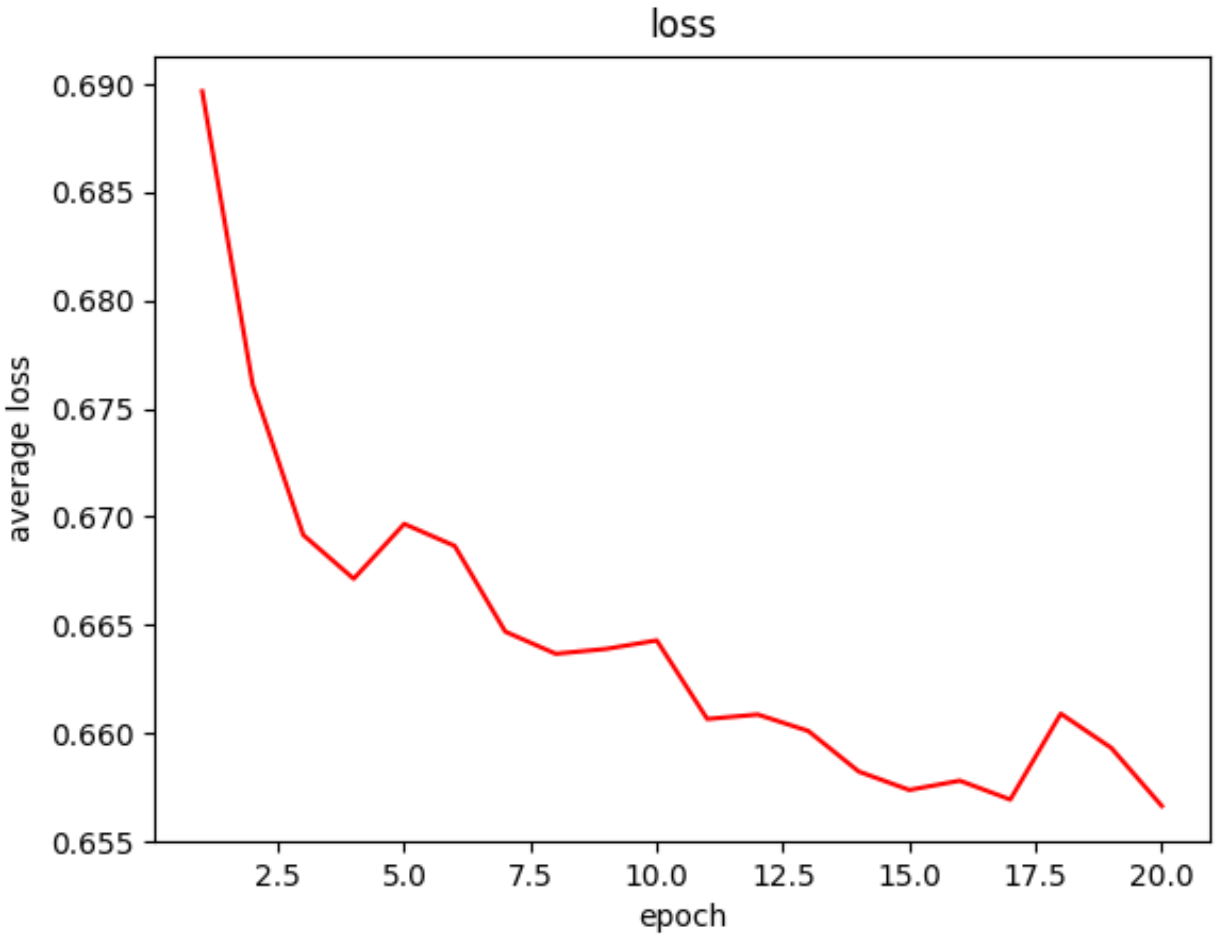
三、实验结果及分析

1. 实验结果展示示例(实验结果放入result文件夹中)

每一轮的准确率:



每一轮的平均损失:



2、评价指标展示及分析

未设置遗忘层:

```
Epoch [1/20], Train Loss: 0.6931, Train Acc: 0.5047, Valid Loss: 0.6932, Valid
Acc: 0.4807
Elapsed Time: 19.826775312423706 s
Epoch [2/20], Train Loss: 0.6838, Train Acc: 0.5529, Valid Loss: 0.6726, Valid
Acc: 0.5907
Elapsed Time: 20.614581823349 s
Epoch [3/20], Train Loss: 0.6721, Train Acc: 0.5825, Valid Loss: 0.6674, Valid
Acc: 0.5867
Elapsed Time: 19.832038164138794 s
Epoch [4/20], Train Loss: 0.6692, Train Acc: 0.5876, Valid Loss: 0.6615, Valid
Acc: 0.6028
Elapsed Time: 20.398427486419678 s
Epoch [5/20], Train Loss: 0.6673, Train Acc: 0.5939, Valid Loss: 0.6633, Valid
Acc: 0.5961
Elapsed Time: 20.720221996307373 s
Epoch [6/20], Train Loss: 0.6659, Train Acc: 0.5944, Valid Loss: 0.6591, Valid
Acc: 0.6001
Elapsed Time: 21.51720929145813 s
Epoch [7/20], Train Loss: 0.6651, Train Acc: 0.5958, Valid Loss: 0.6591, Valid
```

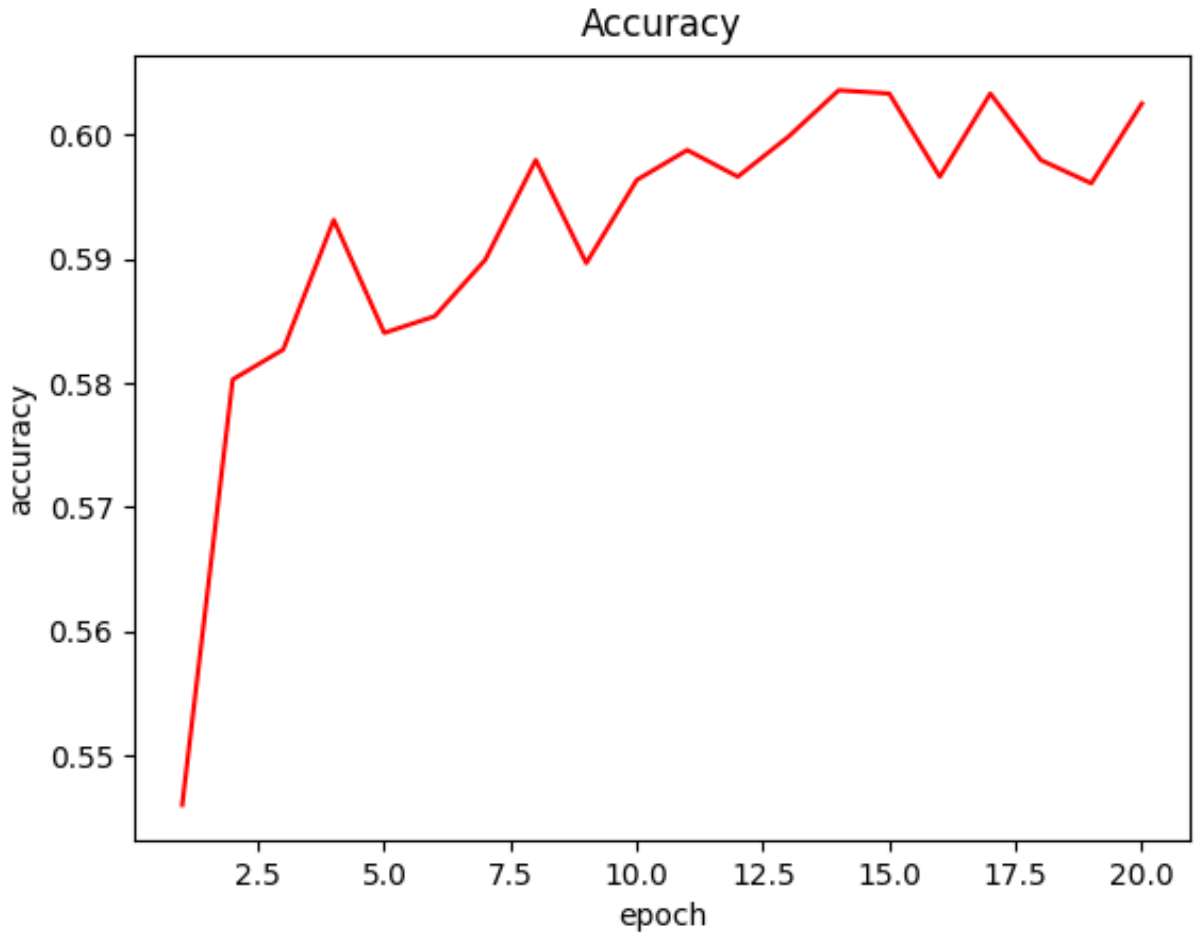
```
Acc: 0.6076
Elapsed Time: 23.299988269805908 s
Epoch [8/20], Train Loss: 0.6640, Train Acc: 0.5977, Valid Loss: 0.6591, Valid
Acc: 0.6073
Elapsed Time: 24.799240112304688 s
Epoch [9/20], Train Loss: 0.6628, Train Acc: 0.5997, Valid Loss: 0.6566, Valid
Acc: 0.6057
Elapsed Time: 21.96651005744934 s
Epoch [10/20], Train Loss: 0.6619, Train Acc: 0.6016, Valid Loss: 0.6546, Valid
Acc: 0.6087
Elapsed Time: 21.736791372299194 s
Epoch [11/20], Train Loss: 0.6603, Train Acc: 0.6047, Valid Loss: 0.6576, Valid
Acc: 0.6006
Elapsed Time: 21.446046113967896 s
Epoch [12/20], Train Loss: 0.6593, Train Acc: 0.6048, Valid Loss: 0.6545, Valid
Acc: 0.6105
Elapsed Time: 21.698827028274536 s
Epoch [13/20], Train Loss: 0.6586, Train Acc: 0.6068, Valid Loss: 0.6552, Valid
Acc: 0.6097
Elapsed Time: 22.535269737243652 s
Epoch [14/20], Train Loss: 0.6573, Train Acc: 0.6085, Valid Loss: 0.6531, Valid
Acc: 0.6079
Elapsed Time: 21.88309144973755 s
Epoch [15/20], Train Loss: 0.6559, Train Acc: 0.6111, Valid Loss: 0.6590, Valid
Acc: 0.6036
Elapsed Time: 21.964349269866943 s
Epoch [16/20], Train Loss: 0.6548, Train Acc: 0.6129, Valid Loss: 0.6525, Valid
Acc: 0.6097
Elapsed Time: 21.564936637878418 s
Epoch [17/20], Train Loss: 0.6534, Train Acc: 0.6134, Valid Loss: 0.6556, Valid
Acc: 0.6087
Elapsed Time: 22.16382622718811 s
Epoch [18/20], Train Loss: 0.6525, Train Acc: 0.6141, Valid Loss: 0.6531, Valid
Acc: 0.6130
Elapsed Time: 21.549935817718506 s
Epoch [19/20], Train Loss: 0.6516, Train Acc: 0.6161, Valid Loss: 0.6532, Valid
Acc: 0.6140
Elapsed Time: 22.046547412872314 s
Epoch [20/20], Train Loss: 0.6500, Train Acc: 0.6186, Valid Loss: 0.6546, Valid
Acc: 0.6031
Elapsed Time: 22.145246505737305 s
```

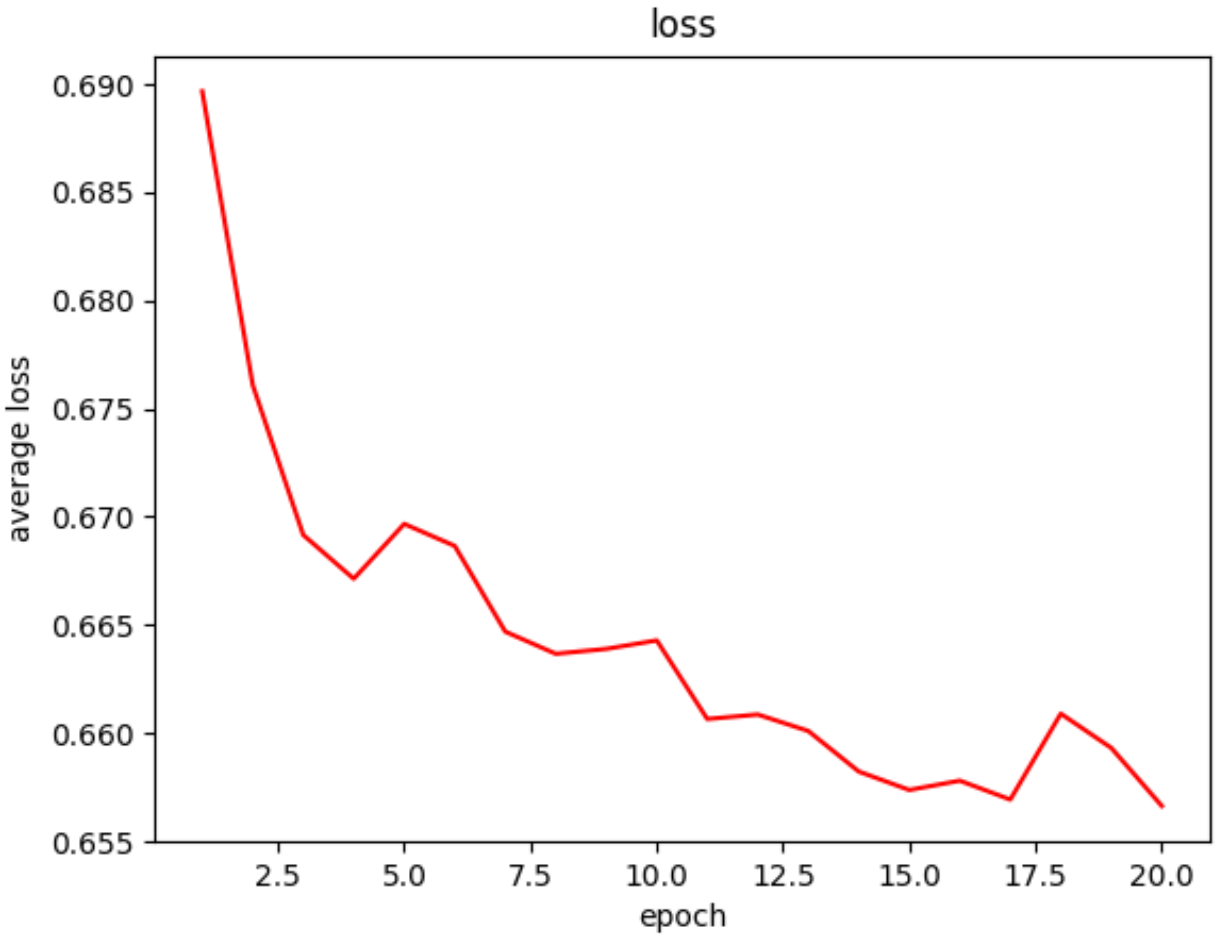
设置遗忘层:

```
Epoch [1/20], Train Loss: 0.6933, Train Acc: 0.5018, Valid Loss: 0.6897, Valid
Acc: 0.5460
Elapsed Time: 36.27997016906738 s
Epoch [2/20], Train Loss: 0.6862, Train Acc: 0.5516, Valid Loss: 0.6761, Valid
Acc: 0.5803
Elapsed Time: 36.601069688797 s
Epoch [3/20], Train Loss: 0.6763, Train Acc: 0.5786, Valid Loss: 0.6692, Valid
Acc: 0.5827
```

```
Elapsed Time: 35.40084195137024 s
Epoch [4/20], Train Loss: 0.6729, Train Acc: 0.5870, Valid Loss: 0.6671, Valid
Acc: 0.5931
Elapsed Time: 34.57430386543274 s
Epoch [5/20], Train Loss: 0.6709, Train Acc: 0.5883, Valid Loss: 0.6697, Valid
Acc: 0.5840
Elapsed Time: 26.85925841331482 s
Epoch [6/20], Train Loss: 0.6693, Train Acc: 0.5906, Valid Loss: 0.6687, Valid
Acc: 0.5854
Elapsed Time: 20.080960035324097 s
Epoch [7/20], Train Loss: 0.6682, Train Acc: 0.5919, Valid Loss: 0.6647, Valid
Acc: 0.5899
Elapsed Time: 21.230495929718018 s
Epoch [8/20], Train Loss: 0.6670, Train Acc: 0.5965, Valid Loss: 0.6637, Valid
Acc: 0.5980
Elapsed Time: 20.71807026863098 s
Epoch [9/20], Train Loss: 0.6657, Train Acc: 0.5975, Valid Loss: 0.6639, Valid
Acc: 0.5897
Elapsed Time: 21.05164909362793 s
Epoch [10/20], Train Loss: 0.6644, Train Acc: 0.5982, Valid Loss: 0.6643, Valid
Acc: 0.5964
Elapsed Time: 21.132972478866577 s
Epoch [11/20], Train Loss: 0.6634, Train Acc: 0.5996, Valid Loss: 0.6607, Valid
Acc: 0.5988
Elapsed Time: 21.54642963409424 s
Epoch [12/20], Train Loss: 0.6622, Train Acc: 0.6018, Valid Loss: 0.6609, Valid
Acc: 0.5966
Elapsed Time: 21.28011441230774 s
Epoch [13/20], Train Loss: 0.6609, Train Acc: 0.6014, Valid Loss: 0.6601, Valid
Acc: 0.5998
Elapsed Time: 21.283547401428223 s
Epoch [14/20], Train Loss: 0.6595, Train Acc: 0.6071, Valid Loss: 0.6582, Valid
Acc: 0.6036
Elapsed Time: 22.040310621261597 s
Epoch [15/20], Train Loss: 0.6583, Train Acc: 0.6096, Valid Loss: 0.6574, Valid
Acc: 0.6033
Elapsed Time: 22.83413314819336 s
Epoch [16/20], Train Loss: 0.6573, Train Acc: 0.6100, Valid Loss: 0.6578, Valid
Acc: 0.5966
Elapsed Time: 21.98153567314148 s
Epoch [17/20], Train Loss: 0.6560, Train Acc: 0.6116, Valid Loss: 0.6569, Valid
Acc: 0.6033
Elapsed Time: 21.864758491516113 s
Epoch [18/20], Train Loss: 0.6550, Train Acc: 0.6129, Valid Loss: 0.6609, Valid
Acc: 0.5980
Elapsed Time: 21.32651686668396 s
Epoch [19/20], Train Loss: 0.6542, Train Acc: 0.6165, Valid Loss: 0.6593, Valid
Acc: 0.5961
Elapsed Time: 20.636079788208008 s
Epoch [20/20], Train Loss: 0.6527, Train Acc: 0.6151, Valid Loss: 0.6566, Valid
Acc: 0.6025
Elapsed Time: 21.543195962905884 s
```

分析:


Accuracy
accuracy
0.60
0.59
0.58
0.57
0.56
0.55
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
epoch



(1)可以看到在第二次模型的准确率就提高到了57%以上，说明，损失值也在下降，后续有较小的波动，在十次左右的训练就可以达到较好的水平。

(2)每一轮的时间为20s左右，分为训练和测试的，即大概每轮10s左右，符合要求;时间只要与文本的数量和每条文本的长度有关。同时一开始读取glove的词表也需要一定的时间。

(3)分析使用了遗忘层和没有使用的模型效果差距不大,优化效果不明显，可能是以下原因:

- 数据集大小：如果数据集非常小，dropout的效果可能会受到限制，因为模型无法从足够多的样本中学习到有效的模式。较小的数据集容易造成过拟合，而dropout的主要目的是减少过拟合。
- 模型复杂性：有时，模型本身可能不太复杂或容易欠拟合，这种情况下，dropout可能并不能提供明显的优化效果。dropout通常在大型深层网络中发挥更大作用，以帮助防止过拟合。
- 已经进行了其他正则化方法：如果已经应用了其他形式的正则化，如L1/L2正则化或批归一化等，dropout可能不会对模型性能产生显著影响。这是因为其他正则化方法已经在一定程度上减少了过拟合的风险。

四、参考资料

https://blog.csdn.net/qq_52785473/article/details/122800625

https://blog.csdn.net/Bat_Reality/article/details/128509050