# 中山大学计算机学院

# 人工智能

## 本科生实验报告

## (2024学年春季学期)

课程名称：Artificial Intelligence

| 教学班级 | 人工智能(22信计+系统结构) | 专业（方向） | 信息与计算科学 |
| --- | --- | --- | --- |
| 学号 | **22336044** | 姓名 | **陈圳煌** |

## 一、实验题目

自然语言推理

## 二、实验内容

### 1.算法原理

**自然语言推理：**

自然语言推理（Natural Language Inference，NLI）是自然语言处理（NLP）领域的一个任务，其目标是判断两个句子之间的逻辑关系。给定一个前提（premise）和一个假设（hypothesis），NLI 系统需要判断这两个句子之间的关系，在这个实验中，包含两个关系：entailment和not_entailment。

**步骤如下：**

**预处理：**

将pandas读取的文件进行预处理，去重，统一成小写，将标签转换为bool值。

**自定义数据类：**

设计数据类，读入文本和标签，使用GloVe词向量模型处理得到tensor张量。

**RNN模型：**

包含输入层将预处理后的文本数据转换为词向量、LSTM层处理序列数据中的时间依赖关系、全连接层，输出为每个样本（entailment和not_entailment）的类别概率分布。

***选择RNN原因：**

CNN在处理图像等数据时非常高效，因为它们能够利用局部空间信息，但在处理序列数据时通常不直接适用。RNN在处理序列数据时表现出色，因为它们能够捕捉序列中的时间依赖关系。RNN在处理序列数据时表现出色，因为它们能够捕捉序列中的时间依赖关系。

## 2.伪代码

```
使用pd.read_csv()读入训练集和测试集数据
数据预处理（去重、转换小写、标签转换为bool值）

定义数据集类TextDataset:
    def __init__:
    初始化实例变量:
        df: 赋值为提供的 dataframe
        tokenizer: 赋值为提供的 tokenizer 函数
        glove: 赋值为提供的 GloVe 模型

    len方法：求dataframe的长度

    def __getitem__:
        item: 获取指定索引的数据帧行
        text: 从 item 中提取 'combined' 文本
        label: 从 item 中提取 'label'
        text_vector: 使用 text_to_vector 方法将文本转换为向量
        返回一个词向量和标签元组

    def text_vector:
        输入text要处理的文本
        tokens=用tokenizer分词文本
        for token in tokens:
            if token 在 glove模型的词汇表中：加入相应词向量
            else：加入一个零向量

        返回向量

class RNN类:
    初始化:
    self.rnn=LSTM实例化（输入维度为 embedding_dim，隐藏维度为 hidden_dim）
    self.fc全连接层
    self.dropout随机丢弃

    def forward前向传播:
        packed_input=pack_padded_sequence对 x 进行打包，以忽略填充部分
        将打包后的序列输入到 LSTM 层
        self.rnn(packed_input)获取 LSTM 层的输出和最后时刻的隐藏状态
        hidden=self.dropout应用 Dropout 层到隐藏状态
        将隐藏状态传递给全连接层
        返回全连接层的输出

主函数（main）:
    加载词向量
    glove=GloVe
    #训练
    train_dataset->train_dataloader加载训练集并导入
    model=RNN(……)初始化模型
    初始化critetion损失函数和optimizer优化器
    for epoch : epochs
        for 每个批次的数据 in train_dataloader:
```

```
            output=model(数据)
            预测、求损失函数
            反向传播、更新优化器
            loss.backward()
            optimizer.step()

    #测试
    test_dataset->test_dataloader加载测试集数据并导入
    for 每个批次的数据 in test_dataloader:
        output=model(数据)
        predicted预测
        记录正确率
        correct+=(predicted==label)
```

## 3.关键代码展示（带注释）

```
#读入数据以及预处理
df=pd.read_csv('data/train_40.tsv',sep='\t',header=0,quoting=csv.QUOTE_NONE)
df2=pd.read_csv('data/dev_40.tsv',sep='\t',header=0,quoting=csv.QUOTE_NONE)
#去除重复值
df=df.drop_duplicates()
#数据类型转换，将标签转换为数值
label_map={'entailment':1,'not_entailment':0}
df['label']=df['label'].map(label_map)
df2['label']=df2['label'].map(label_map)
#转化成小写
df['question']=df['question'].str.lower()
df['sentence']=df['sentence'].str.lower()
df2['question']=df2['question'].str.lower()
df2['sentence']=df2['sentence'].str.lower()
#将问题和回答结合
df['combined']=(df['question']+df['sentence']).astype(str)
df2['combined']=(df2['question']+df2['sentence']).astype(str)

#自定义数据集类
class TextDataset(Dataset):
    def __init__(self,dataframe,tokenizer,glove):
        """
        初始化 TextDataset 类。
        参数:
        - dataframe (DataFrame): 包含 'combined'（文本）和 'label' 列的 pandas
DataFrame。
        - tokenizer: 用于将文本分词为标记的函数。
        - glove: GloVe 词向量模型（假设具有 `stoi` 表示词汇到索引的映射和 `vectors` 表示
词向量的属性）。
        """
        self.df=dataframe
        self.tokenizer=tokenizer
        self.glove=glove
```

```
    def __len__(self):
        return len(self.df)

    def __getitem__(self,idx):
        item=self.df.iloc[idx]#获取指定索引的 DataFrame行
        text=item['combined']
        label=item['label']
        text_vector=self.text_to_vector(text)
        return text_vector,label#返回元组

    def text_to_vector(self,text):
        tokens=self.tokenizer(text)#使用分词器分词
        vectors=[]
        for token in tokens:
            if token in self.glove.stoi:# 如果词在 GloVe 词汇表中
                vectors.append(self.glove.vectors[self.glove.stoi[token]])
            else:
                vectors.append(torch.zeros(100))#使用全0向量表示未知词
        # 保证返回一个有效的张量
        return torch.stack(vectors) if vectors else torch.zeros(1,100)

#处理批次数据
def collate_fn(batch):
    #解包批次数据，得到文本向量和标签的列表
    texts,labels=zip(*batch)
    #计算每个文本向量的长度
    lengths=torch.tensor([len(t) for t in texts])
    #对文本向量进行填充，使它们具有相同的长度
    texts_pad=pad_sequence(texts,batch_first=True)
    labels=torch.tensor(labels)

    return texts_pad,lengths,labels
#RNN模型
class RNN(nn.Module):
    def
__init__(self,embedding_dim,hidden_dim,output_dim,n_layers=1,bidirectional=False,
dropout=0.5):
        """
        初始化 RNN 模型。
        参数:
        - embedding_dim (int): 词向量的维度。
        - hidden_dim (int): RNN 隐藏层的维度。
        - output_dim (int): 输出层的维度，通常等于类别的数量。
        - n_layers (int): RNN 层的数目，默认为 1。
        - bidirectional (bool): 是否使用双向 RNN,默认为 False。
        - dropout (float): Dropout 概率，默认为 0.5。
        """
        super(RNN,self).__init__()

 self.rnn=nn.LSTM(embedding_dim,hidden_dim,num_layers=n_layers,bidirectional=bidi
rectional,batch_first=True,dropout=dropout)
        self.fc=nn.Linear(hidden_dim*2 if bidirectional else
hidden_dim,output_dim)
        self.dropout=nn.Dropout(dropout)

    def forward(self,x,lengths):
```

```
        #使用 pack_padded_sequence 对输入序列进行打包，以忽略填充部分
        #lengths.cpu()：涉及索引或特定功能时，将长度张量移到 CPU 上可以提高兼容性和稳定性

 packed_input=pack_padded_sequence(x,lengths.cpu(),batch_first=True,enforce_sorte
d=False)
        #将打包后的序列输入到 LSTM
        packed_output, (hidden, cell) = self.rnn(packed_input)
        #使用 pad_packed_sequence 将 LSTM 的输出解包，恢复到原始的 batch_size x
sequence_length 形状
        output,_=pad_packed_sequence(packed_output,batch_first=True)
        #如果是双向 RNN，将两个方向的最后时刻的隐藏状态拼接起来
        if self.rnn.bidirectional:
            hidden=self.dropout(torch.cat((hidden[-2,:,:],hidden[-1,:,:]),dim=1))
        else:
            #如果是单向 RNN，只使用最后一个时刻的隐藏状态
            hidden=self.dropout(hidden[-1,:,:])
        #将隐藏状态传递给全连接层，得到最终的输出
        return self.fc(hidden)

#主函数
def main():
    #开始时间
    time1=time.time()
    device=torch.device("cuda" if torch.cuda.is_available() else "cpu")  # 检测GPU
是否可用
    #加载GloVe词向量
    glove=GloVe(name='6B',dim=100)
    tokenizer=get_tokenizer('basic_english')
    #加载训练集
    train_dataset=TextDataset(df,tokenizer,glove)

 train_dataloader=DataLoader(train_dataset,batch_size=256,collate_fn=collate_fn,s
huffle=True)
    #初始化模型
    #双向LSTM

 model=RNN(embedding_dim=100,hidden_dim=128,output_dim=2,n_layers=2,bidirectional
=True,dropout=0.5).to(device)
    #损失函数和优化器
    criterion=nn.CrossEntropyLoss()
    optimizer=optim.Adam(model.parameters(),lr=0.001)
    #训练
    epochs=12
    loss_set=[]
    correct_set=[]
    #训练模式
    model.train()
    for epoch in range(epochs):
        total_loss=0
        correct=0
        time3=time.time()
        for texts_pad,lengths,labels in train_dataloader:

 texts_pad,lengths,labels=texts_pad.to(device),lengths.to(device),labels.to(devic
e) #将数据移动到GPU上
            #优化器的梯度清零
```

```
            optimizer.zero_grad()
            #前向传播得到结果
            outputs=model(texts_pad,lengths)
            #求预测值
            _,predicted=torch.max(outputs,1)
            #累加正确的数据量
            correct+=(predicted==labels).sum().item()
            #求损失函数
            loss=criterion(outputs,labels)
            total_loss+=loss.item()
            #反向传播
            loss.backward()
            #计算梯度，更新优化器
            optimizer.step()

        loss_set.append(total_loss/len(train_dataloader))
        correct_set.append(correct/len(train_dataloader.dataset))
        time4=time.time()
        print(f"Epoch {epoch+1:<3} Loss: {total_loss/len(train_dataloader):>8.6f}
Acc: {correct*100/len(train_dataloader.dataset):5>.2f}%  Time: {time4-
time3:>5.2f}sec")

    draw_picture(loss_set,correct_set)

    #模型评估
    #评估模式
    model.eval()
    #导入测试数据
    test_dataset=TextDataset(df2,tokenizer,glove)

 test_dataloader=DataLoader(test_dataset,batch_size=128,collate_fn=collate_fn,shu
ffle=True)
    correct=0
    total=0
    #语句块内的所有操作都不会计算梯度
    with torch.no_grad():
        for t,le,la in test_dataloader:
            #与训练过程相似
            t,le,la=t.to(device),le.to(device),la.to(device)
            output=model(t,le)
            _,pre=torch.max(output,1)
            total+=la.size(0)
            correct+=(pre==la).sum().item()

    time2=time.time()
    print(f"Accuracy: {correct /total*100:>5.2f}%   Total Time: {time2-
time1:>6.2f}sec")
```

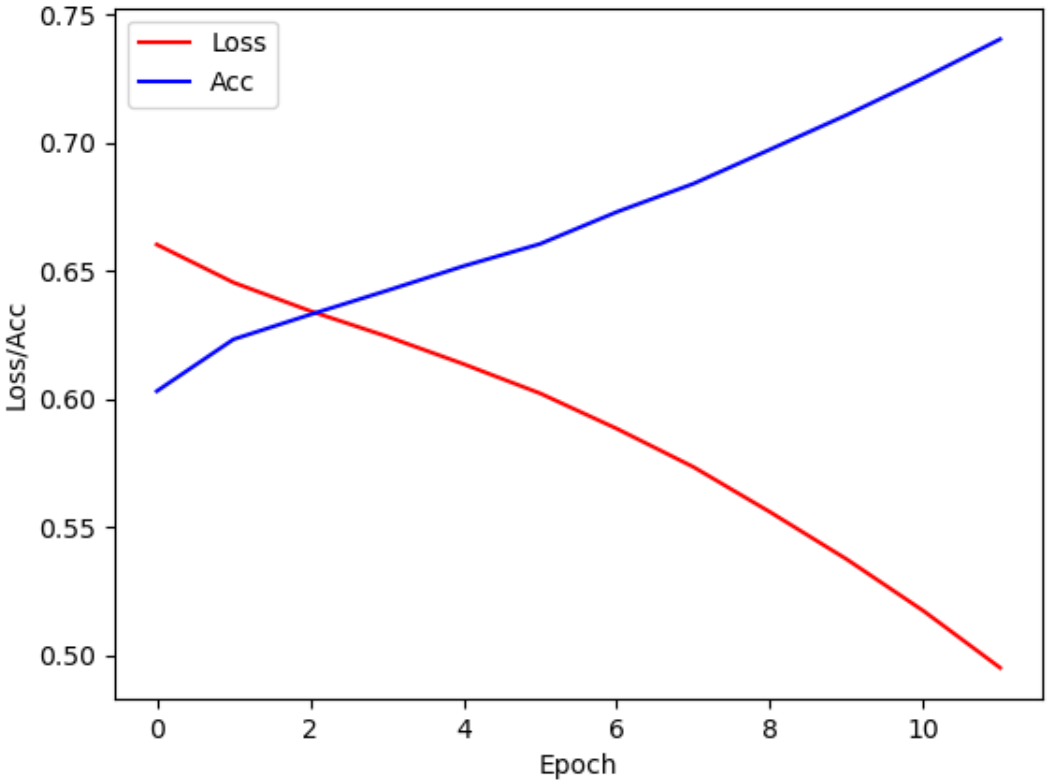
# 三、实验结果及分析

## 1.实验结果展示示例

文件夹result/第4次

参数设置：

批次大小batch_size=128, 学习率lr=0.001,隐藏层大小hidden_size=128




```
Epoch 1    Loss: 0.660123  Acc: 60.29%  Time: 37.68sec
Epoch 2    Loss: 0.645307  Acc: 62.31%  Time: 36.24sec
Epoch 3    Loss: 0.634198  Acc: 63.28%  Time: 36.05sec
Epoch 4    Loss: 0.624288  Acc: 64.21%  Time: 35.83sec
Epoch 5    Loss: 0.613509  Acc: 65.17%  Time: 35.46sec
Epoch 6    Loss: 0.602058  Acc: 66.04%  Time: 35.00sec
Epoch 7    Loss: 0.588285  Acc: 67.28%  Time: 36.49sec
Epoch 8    Loss: 0.573330  Acc: 68.39%  Time: 37.13sec
Epoch 9    Loss: 0.555740  Acc: 69.72%  Time: 35.49sec
Epoch 10   Loss: 0.537308  Acc: 71.07%  Time: 35.25sec
Epoch 11   Loss: 0.517188  Acc: 72.50%  Time: 36.93sec
Epoch 12   Loss: 0.494988  Acc: 74.02%  Time: 39.09sec
Accuracy: 64.29%   Total Time: 467.50sec
```

由结果中发现，最终测试的准确率能达到60%以上，且训练过程中的训练集损失函数和准确率符合。

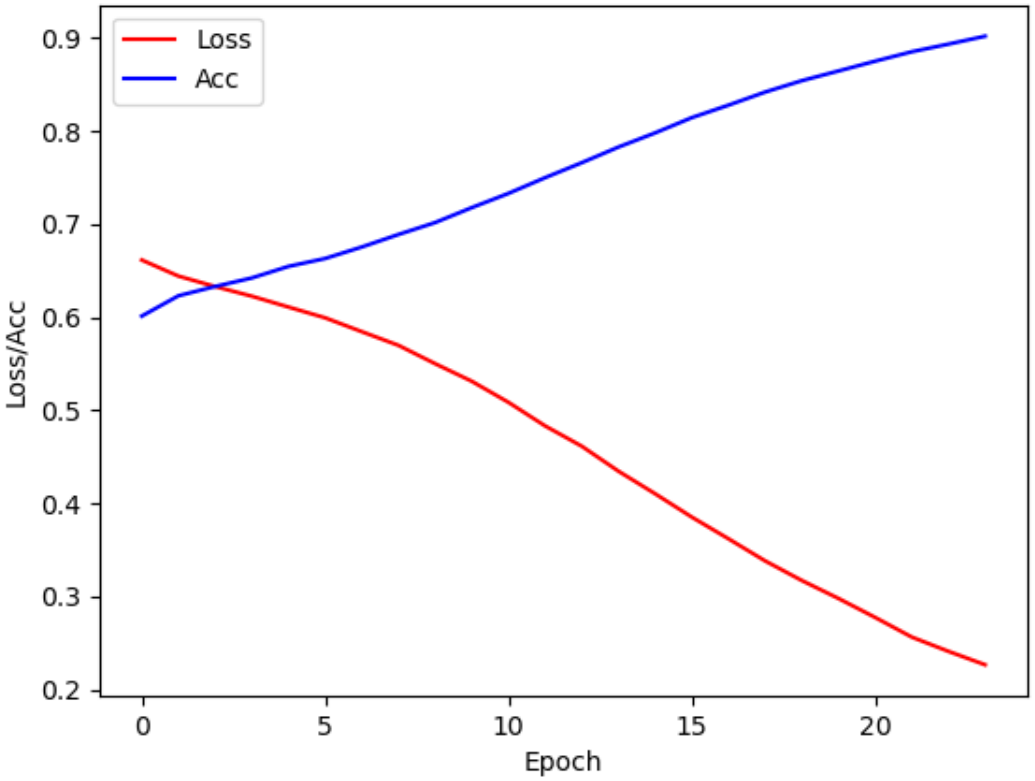
## 2.评测指标展示及分析

在这个实验中采用了GPU进行运算，使用CPU计算时，以batch_size=128为例，每一个epoch需要300sec左右，采用了GPU后每一个epoch需要35sec左右，速度有了很大的提升。

接着通过调整参数batch_size、epoch以及lr来测试训练的效果。

（1）batch_size=128，epoch=24，lr=0.001,，hidden_size=128

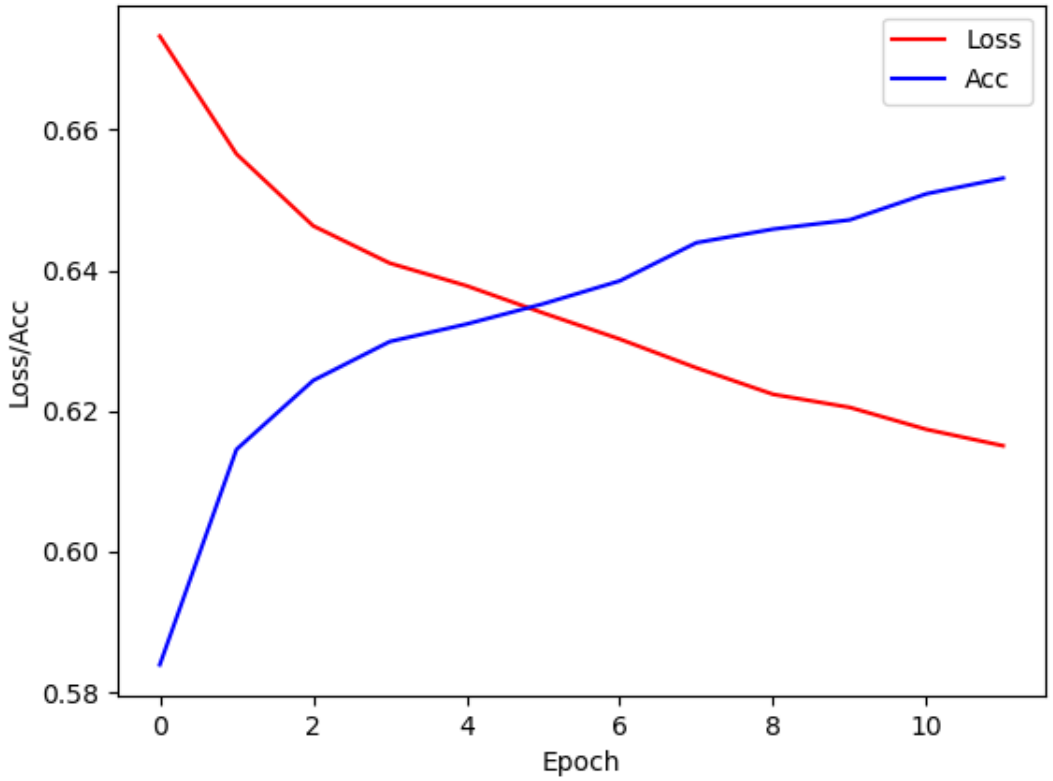
文件夹result/第1次



```
Epoch 1    Loss: 0.661516  Acc: 60.12%
Epoch 2    Loss: 0.644213  Acc: 62.31%
Epoch 3    Loss: 0.632996  Acc: 63.31%
Epoch 4    Loss: 0.622626  Acc: 64.22%
Epoch 5    Loss: 0.610944  Acc: 65.46%
Epoch 6    Loss: 0.599380  Acc: 66.31%
Epoch 7    Loss: 0.584474  Acc: 67.55%
Epoch 8    Loss: 0.569785  Acc: 68.89%
Epoch 9    Loss: 0.550062  Acc: 70.18%
Epoch 10   Loss: 0.531159  Acc: 71.78%
Epoch 11   Loss: 0.508527  Acc: 73.31%
Epoch 12   Loss: 0.483165  Acc: 75.00%
Epoch 13   Loss: 0.461325  Acc: 76.61%
Epoch 14   Loss: 0.434497  Acc: 78.29%
Epoch 15   Loss: 0.410238  Acc: 79.82%
Epoch 16   Loss: 0.385150  Acc: 81.46%
Epoch 17   Loss: 0.361841  Acc: 82.80%
Epoch 18   Loss: 0.338047  Acc: 84.20%
Epoch 19   Loss: 0.316989  Acc: 85.43%
Epoch 20   Loss: 0.297683  Acc: 86.47%
Epoch 21   Loss: 0.277294  Acc: 87.53%
Epoch 22   Loss: 0.256153  Acc: 88.54%
Epoch 23   Loss: 0.240802  Acc: 89.36%
Epoch 24   Loss: 0.226368  Acc: 90.21%
Accuracy: 63.60%
```

（2） batch_size=128，epoch=12，lr=0.01，hidden_size=128

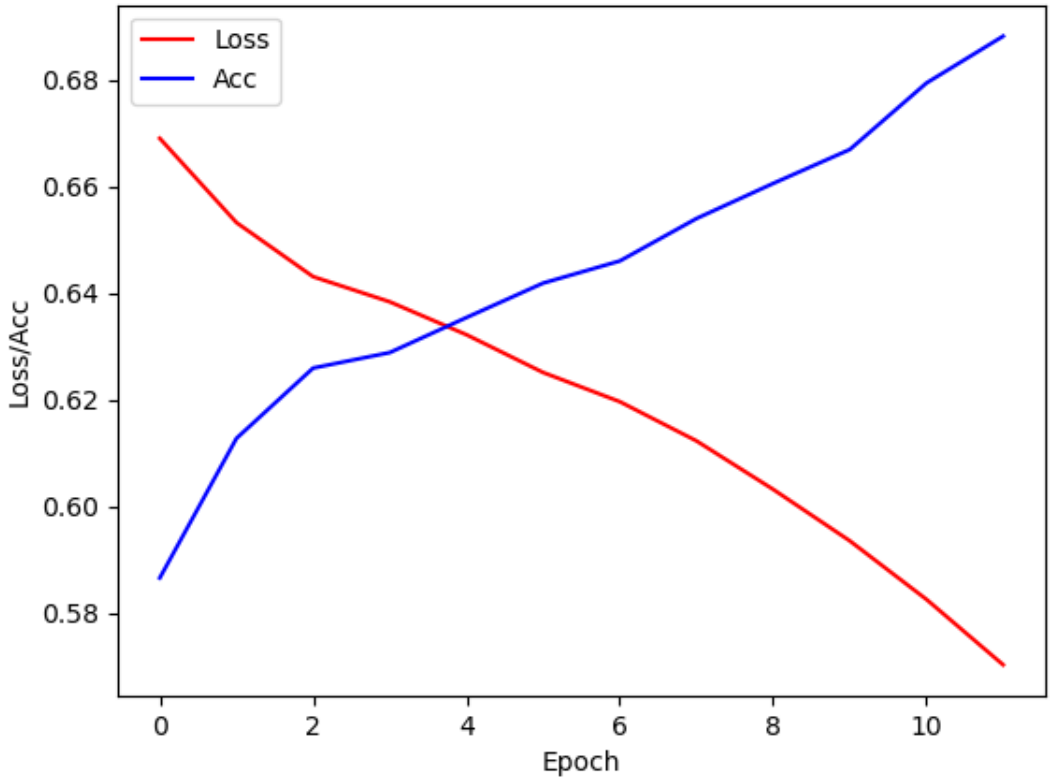
文件夹result/第5次




```
Epoch 1    Loss: 0.673299  Acc: 58.39%  Time: 35.75sec
Epoch 2    Loss: 0.656549  Acc: 61.45%  Time: 35.19sec
Epoch 3    Loss: 0.646334  Acc: 62.43%  Time: 37.61sec
Epoch 4    Loss: 0.641018  Acc: 62.99%  Time: 35.55sec
Epoch 5    Loss: 0.637807  Acc: 63.24%  Time: 36.11sec
Epoch 6    Loss: 0.633938  Acc: 63.52%  Time: 35.76sec
Epoch 7    Loss: 0.630221  Acc: 63.85%  Time: 35.04sec
Epoch 8    Loss: 0.626125  Acc: 64.39%  Time: 35.33sec
Epoch 9    Loss: 0.622365  Acc: 64.59%  Time: 36.13sec
Epoch 10   Loss: 0.620507  Acc: 64.72%  Time: 35.18sec
Epoch 11   Loss: 0.617377  Acc: 65.09%  Time: 35.69sec
Epoch 12   Loss: 0.615052  Acc: 65.31%  Time: 37.72sec
Accuracy: 62.31%   Total Time: 448.89sec
```

（3） batch_size=1024，epoch=12，lr=0.001，hidden_size=256

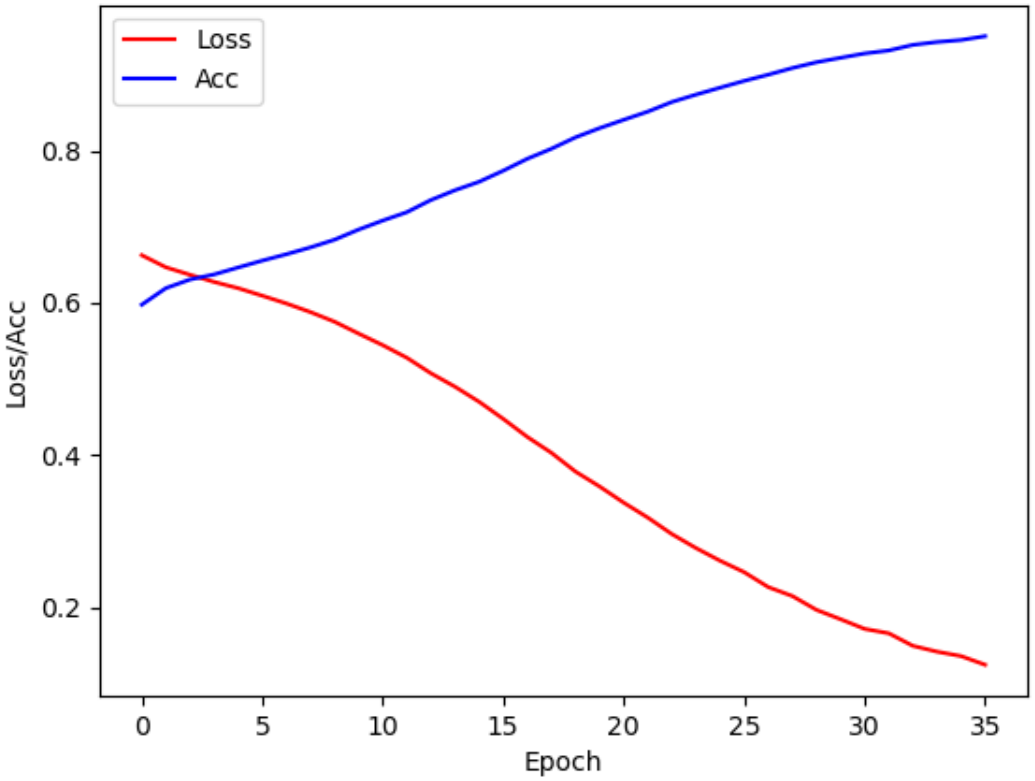
文件夹result/第6次




```
Epoch 1    Loss: 0.669070  Acc: 58.67%  Time: 24.28sec
Epoch 2    Loss: 0.653279  Acc: 61.28%  Time: 23.41sec
Epoch 3    Loss: 0.643138  Acc: 62.60%  Time: 22.54sec
Epoch 4    Loss: 0.638422  Acc: 62.89%  Time: 23.90sec
Epoch 5    Loss: 0.632257  Acc: 63.55%  Time: 23.72sec
Epoch 6    Loss: 0.625175  Acc: 64.19%  Time: 25.24sec
Epoch 7    Loss: 0.619706  Acc: 64.61%  Time: 24.34sec
Epoch 8    Loss: 0.612352  Acc: 65.40%  Time: 37.86sec
Epoch 9    Loss: 0.603271  Acc: 66.06%  Time: 64.57sec
Epoch 10   Loss: 0.593614  Acc: 66.70%  Time: 64.52sec
Epoch 11   Loss: 0.582641  Acc: 67.94%  Time: 64.57sec
Epoch 12   Loss: 0.570381  Acc: 68.82%  Time: 64.91sec
Accuracy: 65.07%   Total Time: 506.88sec
```

（4） batch_size=1024，epoch=36，lr=0.001，hidden_size=256

文件夹result/第7次


Loss
Acc
0.8
0.6
0.4
0.2
Loss/Acc
0
5
10
15
20
25
30
35
Epoch

```
Epoch 1   Loss: 0.662731  Acc: 59.76%  Time: 22.71sec
Epoch 2   Loss: 0.646812  Acc: 61.94%  Time: 21.93sec
Epoch 3   Loss: 0.636982  Acc: 63.07%  Time: 22.14sec
Epoch 4   Loss: 0.627582  Acc: 63.74%  Time: 22.03sec
Epoch 5   Loss: 0.619363  Acc: 64.66%  Time: 21.76sec
Epoch 6   Loss: 0.609449  Acc: 65.56%  Time: 22.50sec
Epoch 7   Loss: 0.599239  Acc: 66.40%  Time: 21.96sec
Epoch 8   Loss: 0.587959  Acc: 67.30%  Time: 22.13sec
Epoch 9   Loss: 0.575255  Acc: 68.31%  Time: 23.18sec
Epoch 10  Loss: 0.559653  Acc: 69.64%  Time: 21.96sec
Epoch 11  Loss: 0.544474  Acc: 70.83%  Time: 22.30sec
Epoch 12  Loss: 0.527919  Acc: 71.93%  Time: 22.13sec
Epoch 13  Loss: 0.507472  Acc: 73.53%  Time: 22.57sec
Epoch 14  Loss: 0.489818  Acc: 74.81%  Time: 22.42sec
Epoch 15  Loss: 0.470047  Acc: 75.93%  Time: 24.11sec
Epoch 16  Loss: 0.447576  Acc: 77.38%  Time: 22.36sec
Epoch 17  Loss: 0.423838  Acc: 78.95%  Time: 23.11sec
Epoch 18  Loss: 0.402953  Acc: 80.26%  Time: 22.25sec
Epoch 19  Loss: 0.378274  Acc: 81.77%  Time: 22.65sec
Epoch 20  Loss: 0.358666  Acc: 82.97%  Time: 22.62sec
Epoch 21  Loss: 0.337428  Acc: 84.07%  Time: 22.70sec
Epoch 22  Loss: 0.317803  Acc: 85.15%  Time: 22.66sec
Epoch 23  Loss: 0.296134  Acc: 86.41%  Time: 22.13sec
Epoch 24  Loss: 0.277751  Acc: 87.38%  Time: 22.50sec
Epoch 25  Loss: 0.261110  Acc: 88.29%  Time: 22.32sec
Epoch 26  Loss: 0.246058  Acc: 89.18%  Time: 22.20sec
Epoch 27  Loss: 0.226446  Acc: 90.00%  Time: 22.30sec
Epoch 28  Loss: 0.214649  Acc: 90.88%  Time: 22.43sec
Epoch 29  Loss: 0.196486  Acc: 91.64%  Time: 22.20sec
Epoch 30  Loss: 0.184199  Acc: 92.22%  Time: 22.30sec
Epoch 31  Loss: 0.171478  Acc: 92.79%  Time: 22.36sec
Epoch 32  Loss: 0.165716  Acc: 93.18%  Time: 22.41sec
Epoch 33  Loss: 0.149233  Acc: 93.92%  Time: 22.27sec
Epoch 34  Loss: 0.141550  Acc: 94.29%  Time: 22.63sec
Epoch 35  Loss: 0.135730  Acc: 94.56%  Time: 22.30sec
Epoch 36  Loss: 0.124313  Acc: 95.06%  Time: 22.80sec
Accuracy: 62.07%   Total Time: 832.47sec
```

基于以上的结果，发现随着训练层数的增加，损失函数逐渐收敛且训练集准确率逐渐接近100%，但是对于测试集的数据预测准确率始终保持在60%-65%之间。且微调学习率、批次大小以及隐藏层大小等参数，也并不会有很大的影响。

推测可能原因有：

（1）模型过拟合，但是发现在仅仅训练一轮的情况下也有接近60%的准确率，这个猜想不合理；

（2）数据集中标签为entailment（bool值设为1）的占多数，因此导致了预测偏差，但是计算了entailment出现的次数，大约一半左右，因此也可以排除；

```
36654 74036
```

（3）GloVe词向量库包含丰富的语义信息，且是预训练好的，使得模型在初始化时就能够利用这些预先学习到的知识，提高了模型的性能的同时也导致了模型的学习并没有很好的效果。

# 四、参考资料

[1] 在 PyTorch 中借助 GloVe 词嵌入完成情感分析 - 知乎 (zhihu.com)

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

[3] week16 NLI.pdf