神经网络应用实践报告——语义相似度计算

姓名: 孙华山
学号: 1120192305
专业: 人工智能
指导老师：鉴萍

2022 年 4 月 11 日

说明：本次实验由于数据处理和模型无法训练原因，找出问题后重新跑了模型并实验，导致迟交。

目录

1 实验简介 **3**

2 实验目标 **3**

3 任务描述 **3**

4 相关算法原理 **3**

4.1 基于深度学习的文本语义相似 (分类) 流程 . . . 3

4.2 实验编码器原理 . . . 4

5 实验环境 **4**

6 实验过程 **4**

6.1 导入相关库函数并定义参数字典 . . . 4

6.1.1 导入相关库函数 . . . 4

6.1.2 定义参数字典 . . . 5

6.2 数据加载器构建以及文本预处理 . . . 5

6.2.1 构建词表 . . . 5

6.2.2 词嵌入 . . . 7

6.2.3 文本预处理 . . . 7

6.2.4 定义数据集类 . . . 8

6.2.5 定义数据加载器 . . . 9

6.3 编码器构建 . . . 11

6.3.1 CNN . . . 11

6.3.2 LSTM、RNN . . . 11

6.3.3 Transformer . . . 11

6.3.4 Bert and Ernie . . . 13

6.4 模型组网构建 . . . 13

6.4.1 构建模型 . . . 14

6.4.2 可视化模型参数 . . . 16

6.5 相关函数构建 . . . 16

6.5.1 预测函数构建 . . . 16

6.5.2 验证函数构建 . . . 17

6.5.3 训练函数构建 . . . 18

6.6 使用不同的模型进行实验 . . . 19

6.6.1 加载数据集 . . . 19

6.6.2 定义模型 . . . 20

6.6.3 定义优化器和损失函数 . . . 20

6.6.4 训练、验证、预测 . . . 20

6.6.5 保存数据 . . . 21

6.7 数据可视化与实例预测 . . . 21

7 实验结果与分析 **22**

7.1 文本相似分类模型构建实验 . . . 22

7.1.1 实验结果 . . . 22

7.1.2 实验分析 . . . 22

7.2 不同编码器对比试验 . . . 22

7.2.1 实验结果 . . . 23

7.2.2 实验分析 . . . 23

8 收获与感想 **26**

9 遇到的问题与解决方法 **26**

1 实验简介

文本语义相似度计算在文章查重、信息检索、图像检索、智能机器问答、词义消歧和搜索引擎等多个领域有着非常广泛的应用。如何提升文本匹配的准确度，是自然语言处理领域的一个重要挑战。

本实验主要研究的问题是自然语言处理中的文本理解中的句子语义相似度问题，本质为分类问题：“相似”和“不相似”，利用 RNN、LSTM、Transformer、Bert 等模型对文本进行表征，并分类；通过实验结果分析模型之间的优缺点并解释可能的原因。

2 实验目标

(1) . 通过文献阅读等方法了解 NLP 语义相似度计算的任务与意义。

(2) . 通过实验，实现可进行语义相似度分类的模型，掌握模型搭建过程以及基本原理。

(3) . 通过实现 RNN(LSTM)、Transformer、预训练模型 (eg.Bert) 三种方式的编码，并对比分析实验结果，进一步理解各种模型的原理和优缺点。

(4) . 进一步可实现 CNN、GNN 编码方式。

3 任务描述

(1) . 任务目标：

1 . 实现语义相似度计算模型。

2 . 实现 RNN(LSTM)、Transformer、Bert 三种编码方式并进行对比试验分析。

(2) . 数据介绍：本次实验使用 AFQMC 蚂蚁金融语义相似度数据集，部分数据展示如下：‘sentence’ 表示两个标签，‘label’


```
{"sentence1": "蚂蚁借呗等额还款可以换成先息后本吗", "sentence2": "借呗有先息到期还本吗", "label": "0"}
{"sentence1": "蚂蚁花呗说我违约一次", "sentence2": "蚂蚁花呗违约行为是什么", "label": "0"}
{"sentence1": "帮我看一下本月花呗账单有没有结清", "sentence2": "下月花呗账单", "label": "0"}
{"sentence1": "蚂蚁借呗多长时间综合评估一次", "sentence2": "借呗得评估多久", "label": "0"}
```


图 1: 部分 AFQMC 数据展示

为对应标签:0：不相似，1：相似。

4 相关算法原理

4.1 基于深度学习的文本语义相似 (分类) 流程

整体实验流程：

1. 文本预处理：语料库分词，制作词表，word2id 映射，将语料库线下转换为 token-id 序列，序列长度排序。
2. 构建数据集类、构建数据加载器。
3. 编码器构建
4. 分类器构建 + 模型组网
5. 模型训练、验证、预测

6. 实验结果可视化并分析

基于深度学习的文本语义相似度分类流程如图 2 所示。

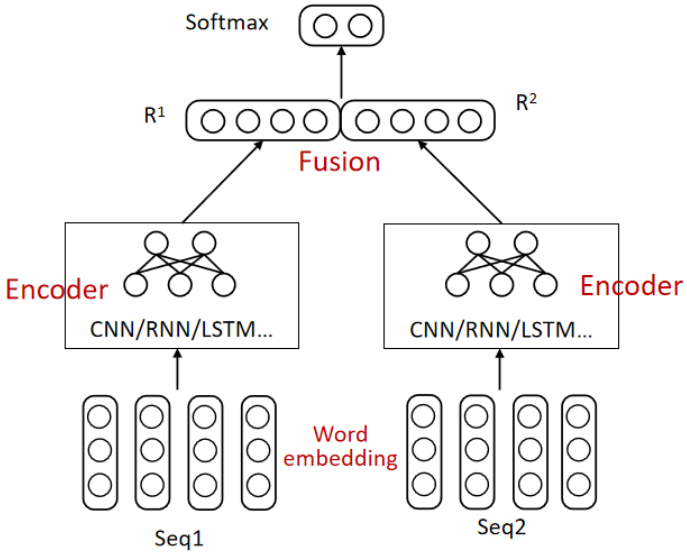


图 2: 语义相似分类流程

注意：若是计算语义相似度，则使用编码器输出的实向量进行指定的相似度计算。

4.2 实验编码器原理

实验所用编码器包括 LSTM、Transformer、Bert 等，其原理在此不再赘述。注意编码器的输入输出，Transformer 需要进行 wordEmbedding 和 positionEmbedding 加和作为输入；Bert 需要转换为 wordEmbedding 和 positionEmbedding、segmentEmbedding 分别作为输入。

其中，LSTM、Transformer 通过对词嵌入的结果进行最大池化得到句子的表征；Bert 认为第一个输出的 CLS 向量包含了整个句子的信息；故本实验使用对应两者作为对应编码器对句子的编码。

5 实验环境

本实验在百度 AI Studio 平台进行部署，使用 Paddle 的深度学习框架。

同时，实验使用了 paddlenlp 中部分集成的模型，如：分词器、预训练的词嵌入模型。

6 实验过程

6.1 导入相关库函数并定义参数字典

6.1.1 导入相关库函数

```
import paddle
import paddlenlp
import json
import paddle.nn as nn
import numpy as np
import jieba
import time
import os
import paddle.nn.functional as F
from functools import partial
```

```
from paddlenlp.embeddings import TokenEmbedding
from paddlenlp.data import JiebaTokenizer
from paddlenlp.data import Stack, Tuple, Pad, Dict
from paddle.io import Dataset
from tqdm import tqdm
from paddlenlp.data import JiebaTokenizer
```

导入相关库函数，在使用 paddlenlp 中的训练好的 TokenEmbedding 函数，直接使用 word2vector，利用 paddle 的 API 快速进行词嵌入。

6.1.2 定义参数字典

定义参数字典，便于调参和实验：

```
paramparameters_dict={'maxlength':5,'learning_rate':5e-4,'max_epoch':20,'embedding_dim':300,'output_dim':2,\
                'eval_step':200,'batch_size':64,'log_step':70,'save_step':2000,'dropout_p':0.3,\
                'nhead':10,'transformer_layers':2,'lstm_hidden_size':400,'encode_dim':400,'lstm_layer':2,\
                'CNN_num_filter':128, 'ngram_filter_sizes':(3, ),\
                'dim_feedforward':100,'save_path':'model/','data_savepath':'data_save/','encoder_name':'transformer'}
```

6.2 数据加载器构建以及文本预处理

本实验使用了 PaddlNLP 中的部分预训练模型和 API，如 TokenEmbedding、JiebaTokenizer。

6.2.1 构建词表

构建词表的类，便于日后使用：

```
#加载paddle预训练的词嵌入模型,并不训练
token_embedding = TokenEmbedding(embedding_name="w2v.baidu_encyclopedia.target.word-word.dim300")

#制作词表的类
class ToWordDict():
    def __init__(self,data_path,token_embeding=token_embedding,embedding_dim=300):
        self.data_path=data_path#载入数据的地址
        self.token_embeding=token_embeding#预训练的词嵌入模型
        self.tokenizer = JiebaTokenizer(vocab=token_embedding.vocab)#预训练的分词器
        self.word_freq_dict = dict()#记录词频
        self.word2id_dict = dict()# 每个词到id的映射关系：word2id_dict
        self.word2id_freq = dict()# 每个id出现的频率：word2id_freq
        self.id2word_dict = dict()# 每个id到词的映射关系：id2word_dict
        self.embedding_dim=embedding_dim#词嵌入的维度300

    #根据需要将词汇转换为预训练的词向量或者id
    def word2id_vector(self,words,mode='id'):
        #输入：单词列表；返回id序列/词向量序列
        if mode=='id':
            return self.tokenizer.encode(words)
        else:
            return self.token_embeding.search(words)

    def sentence2word(self,sentence):
        words=self.tokenizer.cut(sentence)
        return words
```

```
27
28      #将joson文件转换为句子
29      def json2sentence(self):
30          self.data=[]
31          files=os.listdir(self.data_path)
32          #print(files)
33          for i in files:
34              with open(self.data_path+'/'+i,'r',encoding='utf-8') as f:
35                  for line in f.readlines():
36                      d=json.loads(line)
37                      self.data.append(d['sentence1'])
38                      self.data.append(d['sentence2'])
39              f.close()
40
41      #建立词典，统计词频，建立id号和映射关系
42      def ToWordDict(self,data=None):
43          if data is not None:
44              self.data=data
45          for sentence in self.data:
46              #切词
47              words=self.sentence2word(sentence)
48              #统计并制作词典
49              for word in words:
50                  if word not in self.word_freq_dict:
51                      self.word_freq_dict[word] = 0
52                  self.word_freq_dict[word] += 1
53          #加入padding
54          self.word_freq_dict['PAD'] = 0
55          # 一般来说，出现频率高的高频词往往是：I，the，you这种代词，而出现频率低的词，往往是一些名词，如：nlp
56          self.word_freq_dict = sorted(self.word_freq_dict.items(), key = lambda x:x[1], reverse = True)
57          #构建三大词典
58          # 按照频率，从高到低，开始遍历每个单词，并为这个单词构造一个独一无二的id
59          for word, freq in self.word_freq_dict:
60              curr_id = len(self.word2id_dict)
61              self.word2id_dict[word] = curr_id
62              self.word2id_freq[self.word2id_dict[word]] = freq
63              self.id2word_dict[curr_id] = word
64
65          # 把语料转换为id序列
66      def convert_corpus_to_id(self,corpus):
67          # 使用一个循环，将语料中的每个词替换成对应的id，以便于神经网络进行处理
68          corpus = [self.word2id_dict[word] for word in corpus]
69          return corpus
70
71          #建立id和词嵌入的映射和词表
72      def create_vocab_embedding(self):
73          self.vocab_embeddings=np.zeros((len(self.word_freq_dict), self.embedding_dim))
74          #print(self.word_freq_dict)
75          for ind, data in enumerate(self.word_freq_dict):
76              word,_=data
77              embedding=self.token_embeding.search(word)
78              self.vocab_embeddings[ind, :] = embedding
79
```

```
twd=ToWordDict(data_path='data/data')
#josn2sentence
twd.json2sentence()
#todict
twd.ToWordDict()
#制作词表
twd.create_vocab_embedding()
#获得词表和单词-id表
vocab_embedding=twd.vocab_embeddings
word2id=twd.word2id_dict
```

6.2.2 词嵌入

词嵌入方法使用 word2vector 的方法将中文词汇转换为词向量，定义转换函数便于数据加载类使用：（函数说明，作用等见相关注释）

注意：在使用 Bert 的时候，预处理部分直接使用 Bert 的分词器以及 Bert 本身进行词嵌入即可。

```
def word_embedding(words_id_list,vocab_embedding):
    #输入：词列表、词向量表；输出：ID序列
    embeddings=[vocab_embedding[i] for i in words_id_list]
    return np.array(embeddings)
embedding_func=partial(word_embedding,vocab_embedding=vocab_embedding)
```

通过词嵌入函数将每个单词转换为 300 维的词向量，对于一个 token 列表输入，输出则为一个矩阵维度分别为 token 数目、词向量长度（示例将于 6.2.3 展示）。

此函数的作用是将输入的词汇列表、token_id 列表进行词嵌入，返回嵌入后的矩阵。

6.2.3 文本预处理

文本预处理操作包括：分词、停词处理，实验考虑到部分数据较短，因此不做停词处理，定义文本预处理操作函数便于数据加载类使用：（函数说明，作用等见相关注释）

```
#分词停词处理
#考虑到句子可能比较短，停词后可能是空，故不做停词，直接

#将sample中的句子分词并转换为token_id序列
def convert_function(sample,token_embedding,word2id_dict):
    #输入：sample,分词器、词-ID映射表
    #输出：sample,其中的句子转换为id序列
    sentence1=sample["sentence1"]
    sentence2=sample["sentence2"]
    #使用paddleAIP进行分词，词汇为预训练词表
    tokenizer = JiebaTokenizer(vocab=token_embedding.vocab)
    words_id_1,words_id_2=[],[]
    for word in sentence1:
        if word in word2id_dict.keys():
            words_id_1.append(word2id_dict[word])
        else:
            words_id_1.append(word2id_dict['PAD'])
    for word in sentence2:
        if word in word2id_dict.keys():
            words_id_2.append(word2id_dict[word])
        else:
```

```
            words_id_2.append(word2id_dict['PAD'])
    sample["sentence1"]=words_id_1
    sample["sentence2"]=words_id_2
    if 'label' in sample.keys():
        sample['label']=np.array(sample['label'],dtype="float32")
    return sample
#分词、去停词、转换为id预处理
trans_func = partial(
        convert_function,
        token_embedding=token_embedding,word2id_dict=word2id)
```

此函数是使用预训练的词向量模型 word2vector 进行分词、词嵌入，转换为对应词表的 token-id 序列（输入为样本字典，实际输出是一个样本对应的字典，展示将于 6.2.3 部分展示）。

后采用线下转换的方法：把句子线下转换为 token-id 序列，这样避免了在线转换，很大程度上提升了训练速度和效率，减少了训练时间。

```
original_path='data/data/train.json'
target_path='data_new/train.json'
with open(original_path,'r',encoding='utf-8') as f1:
    with open(target_path,'w+') as f2:
        for line in tqdm(f1.readlines()):
            d=json.loads(line)
            d=trans_func(d)
            label=d['label']
            d['label']=int(label)
            f2.write(json.dumps(d, ensure_ascii=False)+'\n')
    f2.close()
f1.close()
```

6.2.4 定义数据集类

定义数据数据集类，读取数据并可更具具体要求返回原数据、分词处理后 token-id 和词嵌入后的数据，具体可根据实际使用的模型进行选取：

```
class Dataset(paddle.io.Dataset):
    def __init__(self,data_path='data/data/',mdoe='train',trans_fun=None,word_embeding_func=None):
        super(Dataset, self).__init__()
        self.data=[]
        self.mode=mdoe
        self.trans_func=trans_fun #句子转ID
        self.word_embedding_func=word_embeding_func#词嵌入
        #读取数据
        data_path=data_path+self.mode+'.json'
        with open(data_path,'r',encoding='utf-8') as f:
            for line in f.readlines():
                d=json.loads(line)
                self.data.append(d)
        f.close()

    def __getitem__(self,index,cut_embedding='all'):
        #更具具体需求返回分词与否、词嵌入与否的数据
        if cut_embedding == 'all':
            sample=self.data[index]
```

```
            #print(sample)
            #分词去停词
            sample_words=self.trans_func(sample)
            #进行词嵌入
            sample_embedding=dict()
            sample_embedding['sentence1']=self.word_embedding_func(sample_words['sentence1'])
            sample_embedding['sentence2']=self.word_embedding_func(sample_words['sentence2'])
            if 'label' in sample.keys():
                sample_embedding['label']=sample_words['label']
            return sample_embedding
        elif cut_embedding=='cut':
            sample=self.data[index].copy()
            #分词去停词
            sample_words=self.trans_func(sample)
            return sample_words
        else:#cut_embedding=='no'
            sample=self.data[index]
            return sample

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

将部分数据输出展示：

```
#展示部分数据
test_dataset=Dataset(data_path='data/data/',mdoe='train',trans_fun=trans_func,word_embeding_func=embedding_func)
print('原始数据：')
print(test_dataset.__getitem__(0,cut_embedding='no'))
print('分词、去停词后的对应数据：')
print(test_dataset.__getitem__(0,cut_embedding='cut'))
print('分词、去停词并进行词嵌入后的对应数据：')
sampletest=test_dataset.__getitem__(0,cut_embedding='all')
print("词嵌入后的维度：",sampletest['sentence1'].shape)
print(sampletest)
```

```
原始数据：
{'sentence1': '蚂蚁借呗等额还款可以换成先息后本吗', 'sentence2': '借呗有先息到期还本吗', 'label': '0'}
分词、转换为token_id的对应数据：
{'sentence1': [2990, 3378, 2, 0, 766, 317, 11, 91, 304, 117, 145, 838, 427, 299, 50, 955, 8], 'sentence2':
'label': array(0., dtype=float32)}
分词、去停词并进行词嵌入后的对应数据：
词嵌入后的维度： (17, 300)
```

图 3: 部分数据不同输出对比展示

通过数据集类，我们可以按照需求将数据输出为句子、token-id 列表或者词向量矩阵。

6.2.5 定义数据加载器

定义数据加载器，返回指定 Bath 和形式的数据，便于训练：**注意**：将 Pad 填充在句子的左边，得到的信息更多，效果更好。

```
#构建数据加载器
def create_dataloader(dataset, mode='train',batch_size=1,batchify_fn=None):
    shuffle = True if mode == 'train' else False
```

```
    if mode == 'train':
        batch_sampler = paddle.io.DistributedBatchSampler(
            dataset, batch_size=batch_size, shuffle=shuffle)
    else:
        batch_sampler = paddle.io.BatchSampler(
            dataset, batch_size=batch_size, shuffle=shuffle)
    return
        paddle.io.DataLoader(dataset=dataset,batch_sampler=batch_sampler,collate_fn=batchify_fn,return_list=True)

#整合Batch的数据
#需要根据是否使用Bert进行调整：非Bert返回token-id，Bert则需返回token-id\token_type等为字典。
def collate_func(batch_data,if_bert=False,max_length=15):
    batch_size = len(batch_data)
    # 如果batch_size为0，则返回一个空字典
    if batch_size == 0:
        return {}
    sentence1_list,sentence2_list,label_list=[],[],[]
    input_ids,token_type_ids=[],[]
    #判断是否是预测数据
    instance1=batch_data[1]
    is_test=1
    if 'label' in instance1.keys():
        is_test=0
    for instance in batch_data:
        if if_bert:
            token_type_id=instance['token_type_ids']
            input_id=instance['input_ids']
            input_ids.append(paddle.to_tensor(input_id, dtype="int64"))
            token_type_ids.append(paddle.to_tensor(token_type_id, dtype="int64"))
        else:
            sentence1=instance["sentence1"]
            sentence2=instance["sentence2"]
            #由于是token-id因此返回的是int64形
            if len(sentence1)>max_length:
                sentence1=sentence1[:max_length]
            if len(sentence2)>max_length:
                sentence2=sentence2[:max_length]
            sentence1_list.append(paddle.to_tensor(sentence1, dtype="int64"))
            sentence2_list.append(paddle.to_tensor(sentence2, dtype="int64"))
        if is_test==0:
            label=instance['label']
        if is_test==0:
            label_list.append(label)
    # 对一个batch内的数据，进行padding,padding的值为词表中的[pad]的Index:635964
    if if_bert:
        if is_test==0:
            return {'input_ids':Pad(axis=0, pad_val=tokenizer.pad_token_id,pad_right=False)(input_ids), # input_ids
                'token_type_ids':Pad(axis=0, pad_val=tokenizer.pad_token_type_id,pad_right=False)(token_type_ids),
                "labels":Stack(dtype="int64")(label_list),
                }
        else:
            return {'input_ids':Pad(axis=0, pad_val=tokenizer.pad_token_id,pad_right=False)(input_ids),
                'token_type_ids':Pad(axis=0, pad_val=tokenizer.pad_token_type_id,pad_right=False)(token_type_ids),
```

```
            }
    else:
        if is_test==0:
            return {"sentence1s": Pad(pad_val=10218, axis=0,pad_right=False)(sentence1_list),
                "sentence2s": Pad(pad_val=10218, axis=0,pad_right=False)(sentence2_list),
                "labels": Stack(dtype="int64")(label_list),
                }
        else:
            return {"sentence1s": Pad(pad_val=10218, axis=0,pad_right=False)(sentence1_list),
                "sentence2s": Pad(pad_val=10218, axis=0,pad_right=False)(sentence2_list),
                }
```

通过定义数据加载器，每个 Batch 返回的就是组装好的字典（collate_func 的返回形式），便于输入模型进行训练。

6.3 编码器构建

构建编码器，对句子进行编码：

(1) .LSTM 和 Transformer 使用最后一个单词对应的输出或者加作为句子表征。

(2) .Bert 使用 CLS 输出的向量作为整个句子到的编码

构建不同的编码器，便于模型组网：

6.3.1 CNN

```
#使用集成的CNNEncoder
cnn_encoder_api=seq2vec.CNNEncoder(emb_dim=paramparameters_dict['embedding_dim'],\
              num_filter=paramparameters_dict['CNN_num_filter'],\
              ngram_filter_sizes=paramparameters_dict['ngram_filter_sizes'])
```

6.3.2 LSTM、RNN

```
#LSTM
#Lstm使用最后一个输出编码作为整体句子的编码，对编码的句子的长度没有要求
lstm_encoder=nn.LSTM(input_size=paramparameters_dict['embedding_dim'],\
              hidden_size=paramparameters_dict['lstm_hidden_size'],num_layers=2,\
              dropout=paramparameters_dict['dropout_p'])
#使用集成的LSTMEncoder
lstm_encoder_api=seq2vec.LSTMEncoder(input_size=paramparameters_dict['embedding_dim'],\
              hidden_size=paramparameters_dict['encode_dim'],\
              num_layers=paramparameters_dict['lstm_layer'],direction='bidirect'
#RNN集成API
rnn_encoder_api=seq2vec.RNNEncoder(input_size=paramparameters_dict['embedding_dim'],\
          hidden_size=paramparameters_dict['encode_dim'],\
          num_layers=paramparameters_dict['lstm_layer'],)
```

6.3.3 Transformer

Transformer 编码器首先需要将输入的 token-id 进行词嵌入和位置信息嵌入二者加和作为编码器的输入，最后通过最大池化得到句子的信息。

```python
#定义Transformer编码器类
from paddlenlp.transformers import WordEmbedding,PositionalEmbedding
class Transformer_encoder(nn.Layer):
    def __init__(self,vocab_size,max_length,emb_dim,encode_dim,\
        trans_layer_num,trans_head_num,hidden_dim,normalize_before=False,\
        if_pre_embeding=False,token_embedding=vocab_embedding ):
        super(Transformer_encoder,self).__init__()
        #输入：词嵌入大小，最大长度，词嵌入维度，句子编码维度，transformer层数，\
        # 多头机制头数目，FNN隐藏层大小,是否在子层输入前层归一化

        #定义wordembedding和positionembeding，作为transformer的输入
        self.if_pre_embeding=if_pre_embeding
        #选择是否使用预训练的词嵌入
        if self.if_pre_embeding:
            print('使用预训练的词向量')
            pretrained_attr = paddle.ParamAttr(
                 initializer=paddle.nn.initializer.Assign(token_embedding),
                                trainable=True)
            self.wordembedding=nn.Embedding(num_embeddings=token_embedding.shape[0],
                embedding_dim=token_embedding.shape[1],
                weight_attr=pretrained_attr)
        else:
            print('使用可学习的词向量')
            self.wordembedding=WordEmbedding(vocab_size=vocab_size,emb_dim=emb_dim,bos_id=10218)
        self.positionembedding=PositionalEmbedding(emb_dim=emb_dim,max_length=max_length)
        self.positionembedding=PositionalEmbedding(emb_dim=emb_dim,max_length=max_length)
        #定义编码层
        self.encoderlayer=nn.TransformerEncoderLayer(d_model=emb_dim,nhead=trans_head_num,\
            dim_feedforward=hidden_dim,normalize_before=normalize_before)
        #定义编码器:
        self.tansformer_encoder=nn.TransformerEncoder(self.encoderlayer,num_layers=trans_layer_num)
        #定义线性层，将transformer的输出映射到编码输出
        self.liner=nn.Linear(in_features=emb_dim,out_features=encode_dim)

    #采用平均池化的方法进行句子特征融合
    def avrage2out(self,code):
        encode=paddle.zeros((code.shape[0],code.shape[2]))
        for i in range(code.shape[1]):
            encode=code[:,i,:]+encode
        return encode/code.shape[1]

    #采用最大池化的方法进行句子特征融合
    def maxpooling2out(self,code):
        encode=paddle.max(code,axis=1)
        return encode

    def forward(self,token_ids):
        #[batch_size,sequence_length]
        batch_size,sequence_length=token_ids.shape
        #首先得到对饮的位置id
        pos=paddle.tile(paddle.arange(start=0, end=sequence_length), repeat_times=[batch_size, 1])
        #进行位置编码
```

```
        pos_emb = self.positionembedding(pos)
        #进行词嵌入
        src_emb = self.wordembedding(token_ids)
        #加和作为transformer输入
        trans_input=pos_emb+src_emb
        code=self.tansformer_encoder(trans_input)
        #print(code)
        code=self.liner(code)
        #得到句子表征
        out=self.maxpooling2out(code)
        return out

transformer_encoder=Transformer_encoder(vocab_size=vocab_embedding.shape[0],max_length=256,\
        emb_dim=paramparameters_dict['embedding_dim'],\
        encode_dim=paramparameters_dict['encode_dim'],\
        trans_layer_num=paramparameters_dict['transformer_layers'],\
        trans_head_num=paramparameters_dict['nhead'],\
        hidden_dim=paramparameters_dict['dim_feedforward'],normalize_before=True )
```

6.3.4 Bert and Ernie

需要注意，在 Bert 和 Ernie 预训练模型，需要构建针对性的 sample_transfunction 用于将样本转换为包含 label、input_ids、token_type_ids 的字典返回。

```
#Bert
from paddlenlp.transformers import BertModel,BertTokenizer

#加载预训练的Bert模型
tokenizer_bert = BertTokenizer.from_pretrained('bert-wwm-chinese')
bert_encode = BertModel.from_pretrained('bert-wwm-chinese')
#Ernie
tokenizer_ernie = paddlenlp.transformers.ErnieGramTokenizer.from_pretrained('ernie-gram-zh')
Ernie_encode = paddlenlp.transformers.ErnieGramModel.from_pretrained('ernie-gram-zh')

tokenizer=tokenizer_bert

#对于Bert需要特有的转换方式等预处理方式,输入sample字典，输出是每个句子对应的input_id\token_type_id的字典
def for_Bert_transfunc(sample,tokenizer,max_seq_length=512):
    query=sample['sentence1']
    title=sample['sentence2']
    encoded_inputs = tokenizer(
        text=query, text_pair=title, max_seq_len=max_seq_length)
    if 'label' in sample.keys():
        encoded_inputs['label']=np.array(sample['label'],dtype="float32")
    return encoded_inputs

bert_transfunc=partial(for_Bert_transfunc,tokenizer=tokenizer,max_seq_length=512
```

6.4 模型组网构建

构建模型组网，模型可以根据指定参数选择性构建 Embedding 层：

(1) . 构建词嵌入层，且使用预训练的词向量：word2vector、Glove 等

(2) . 构建词嵌入层，且不适用预训练的词嵌入，改为可训练的词嵌入。

(3) . 不构建词嵌入层，直接输入词向量。

(4) . 不够构建词嵌入层，输入句子——Bert 作为 encoder 时候

同时可以选择不同的模型作为编码器。

forward 函数先通过词嵌入层获得词嵌入向量，然后再通过 encoder 层（编码器），再经过一个 FNN+softmax 进行分类。

其中：choose_last_one 函数是为了选取编码后的最后一个词对应的向量，认为其嵌入了整个句子的信息，Bert 则选择 CLS 向量。sumtosentence 函数是将编码器的每个输出进行加和，认为其嵌入了整个句子的信息。

6.4.1 构建模型

```
class Classifer(nn.Layer):
    def __init__(self,encoder=None,embedding_dim=paramparameters_dict['embedding_dim'],\
                encoder_dim=paramparameters_dict['encode_dim'],\
                class_dim=paramparameters_dict['output_dim'],is_embedding=False,\
                token_embedding=vocab_embedding,is_pre_embedding=False,mode='LSTM',init_scale=0.1,\
                vocab_size=vocab_embedding.shape[0],):
        #参数分别的意义：编码器、词嵌入维度、编码维度、分类数目、是否进行词嵌入、##词嵌入的词表、是否使用预训练词嵌入、
        编码器名称、词嵌入层初始化、词表大小
        super(Classifer,self).__init__()
        print('使用的编码器是：'+mode)
        self.is_pre_embedding=is_pre_embedding
        self.is_embedding=is_embedding
        self.embedding_dim=embedding_dim
        self.encoder_dim=encoder_dim
        self.mode=mode
        if self.is_pre_embedding and self.is_embedding:
            #构建预训练的词嵌入层
            print('使用预训练的词嵌入')
            pretrained_attr = paddle.ParamAttr(
                                initializer=paddle.nn.initializer.Assign(token_embedding),
                                trainable=False)
            self.embedding_layer=nn.Embedding(num_embeddings=token_embedding.shape[0],
                                  embedding_dim=token_embedding.shape[1],
                                  weight_attr=pretrained_attr)
        elif self.is_embedding:
            #使用非预训练的词嵌入层
            print('使用非预训练的词嵌入，训练词嵌入层')
            self.embedding_layer=nn.Embedding(num_embeddings=vocab_size, embedding_dim=self.embedding_dim,
                sparse=False,
                                 weight_attr=paddle.ParamAttr(initializer=nn.initializer.Uniform(low=-init_scale,
                                     high=init_scale)))

        self.encoder=encoder
        if self.mode!='Bert' and self.mode!="Ernie":
            self.fnn=nn.Sequential(nn.Linear(in_features=2*self.encoder_dim,out_features=400),nn.Tanh(),\
                   nn.Linear(in_features=400,out_features=class_dim))
            print('使用普通的Embedding层')
        else:
            self.fnn=nn.Sequential(nn.Linear(in_features=self.encoder.config["hidden_size"],out_features=400),nn.ReLU(),\
                   nn.Linear(in_features=400,out_features=class_dim))
```

```
        print('使用Transformer-based model')

    def forward(self,input_data):
        first_sentence_id,second_sentence_id=input_data[0],input_data[1]
        #如果不是Bert,那么直接就是输入词向量或者token-id
        #对于Bert编码器，两个sentence_id对应的是各自句子的Bert:
        #input_id和token_type_id，需要进一步处理转换为Bert的输入，且使用Bert时，不需要embedding。
        if self.is_embedding:
            #进行词嵌入
            first_sentence=self.embedding_layer(first_sentence_id)
            second_sentence=self.embedding_layer(second_sentence_id)
        else:
            first_sentence=first_sentence_id
            second_sentence=second_sentence_id
        #计算编码
        if self.mode=='Bert' or self.mode=='Ernie':
            (sequence_output,cls)=self.encoder(input_ids=first_sentence,\
                                token_type_ids=second_sentence)
            #得到的CLS就可以用于文本分类
            input_v=cls
        elif self.mode=='Transformer':
            code1=self.encoder(first_sentence)
            code2=self.encoder(second_sentence)
            #合并向量:
            input_v=paddle.concat(x=[code1,code2], axis=-1)
        elif self.mode=='LSTM_API':
            seq_len1=paddle.to_tensor([first_sentence.shape[1] for i in range(first_sentence.shape[0])])
            seq_len2=paddle.to_tensor([second_sentence.shape[1] for i in range(second_sentence.shape[0])])
            code1=self.encoder(first_sentence,seq_len1)
            code2=self.encoder(second_sentence,seq_len2)
            #合并向量:
            input_v=paddle.concat(x=[code1,code2], axis=-1)
        elif self.mode=='CNN':
            code1=self.encoder(first_sentence)
            code2=self.encoder(second_sentence)
            #合并向量:
            input_v=paddle.concat(x=[code1,code2], axis=-1)
        elif self.mode=='LSTM':
            codes1,(h,c)=self.encoder(first_sentence)
            codes2,(h,c)=self.encoder(second_sentence)
            #选择句子表征
            code1=self.sumtosentencevecter(codes1)
            code2=self.sumtosentencevecter(codes2)
             #合并向量:
            input_v=paddle.concat(x=[code1,code2], axis=-1)
        #进一步融合特征进行分类
        out=self.fnn(input_v)
        return out

    #选择编码最后一个单词时的输出作为句子表征
    def choose_last_one(self,code):
        index=np.array(range(code.shape[0])).astype('int32')
        #生成选择的索引并选择
```

```
        index=paddle.to_tensor(code.shape[1]-1)
        encode=paddle.index_select(x=code,index=index,axis=1)
        #resize到二维
        encode=paddle.reshape(encode,[encode.shape[0],encode.shape[2]])
        return encode

    #将编码器的所有输出加和作为句子表征，避免信息遗漏
    def sumtosentencevecter(self,code):
        encode=paddle.zeros((code.shape[0],code.shape[2]))
        for i in range(code.shape[1]):
            encode=code[:,i,:]+encode
        return encode
```

6.4.2 可视化模型参数

```
paddle.set_device('gpu:0')
model=Classifer(encoder=lstm_encoder,token_embedding=vocab_embedding,is_embedding=True,mode='LSTM')
#打印网络查看网络结构
params_info = paddle.summary(model, (2, 64,5),dtypes='int64')
print(params_info)
```

```
---------------------------------------------------------------------------------------------------------
 Layer (type)       Input Shape                      Output Shape                      Param #
=========================================================================================================
  Embedding-1        [[64, 5]]                       [64, 5, 300]                    3,065,700
    LSTM-1         [[64, 5, 300]]    [[64, 5, 200], [[2, 64, 200], [2, 64, 200]]]     723,200
   Linear-20        [[64, 400]]                      [64, 600]                        240,600
    ReLU-1          [[64, 600]]                      [64, 600]                            0
   Linear-21        [[64, 600]]                      [64, 300]                        180,300
    ReLU-2          [[64, 300]]                      [64, 300]                            0
   Linear-22        [[64, 300]]                      [64, 100]                         30,100
    ReLU-3          [[64, 100]]                      [64, 100]                            0
   Linear-23        [[64, 100]]                       [64, 2]                           202
=========================================================================================================
Total params: 4,240,102
Trainable params: 1,174,402
Non-trainable params: 3,065,700
---------------------------------------------------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 2.59
Params size (MB): 16.17
Estimated Total Size (MB): 18.77
---------------------------------------------------------------------------------------------------------

{'total_params': 4240102, 'trainable_params': 1174402}
```

图 4: 模型 (LSTM) 参数可视化

模型包含 4240102 个参数，其中可训练的有 1174402 个，如图 4。

6.5 相关函数构建

构建相关函数，便于不同的模型训练:

6.5.1 预测函数构建

定义预测函数，用于预测所有的数据——test.joson 文件:

```
def alltest(model,data_loader_test):
```

```
    # 测试函数，需要完成的任务有：根据测试数据集中的数据，逐个对其进行预测，生成预测值。
    with open('predict_labels_1120192305.txt','w+',encoding='utf-8') as f:
        model.eval()
        mode=model.mode
        for sample in tqdm(data_loader_test):
            if mode=="Bert" or mode=='Ernie':
                sentence1=sample['input_ids']
                sentence2=sample['token_type_ids']
            else:
                sentence1=sample['sentence1s']
                sentence2=sample['sentence2s']
            #预测
            input_data=[sentence1,sentence2]
            pres=model(input_data)
            pres=np.argmax(pres.numpy(),axis=-1)
            for pre in pres:
                f.write(str(pre)+'\n')
    f.close()
```

6.5.2 验证函数构建

定义验证函数，在迭代一定次数后，在模型上进行验证，查看模型训练效果：

```

def evaluate(model, criterion, metric, data_loader):
    model.eval()
    metric.reset()
    losses = []
    acces=[]
    f1score=[]
    mode=model.mode
    for sample in data_loader:
        if mode=="Bert" or mode=='Ernie':
            sentence1=sample['input_ids']
            sentence2=sample['token_type_ids']
        else:
            sentence1=sample['sentence1s']
            sentence2=sample['sentence2s']
        labels=sample['labels']
        input_data=[sentence1,sentence2]
        logits = model(input_data)
        loss = criterion(logits, labels)
        losses.append(loss.numpy())
        logits = F.softmax(logits, axis=1)
        correct = metric.compute(logits, labels)
        #f1分数
        y_pred=np.argmax(logits.numpy(),axis=-1)
        y_ture=labels.numpy()
        f1=f1_score(y_pred=y_pred,y_true=y_ture)
        f1score.append(f1)
        metric.update(correct)
        accu = metric.accumulate()
    print("eval loss: %.5f, accu: %.5f,f1:%.5f" % (np.mean(losses),accu,np.mean(f1score)))
```

```
    model.train()
    metric.reset()
    #返回验证的损失和准确率便于数据展示
    return (np.mean(losses), accu,np.mean(f1score))
```

6.5.3 训练函数构建

定义训练函数，对模型进行训练，并记录训练过程：

```
def do_train( model, data_loader, vali_data_loader, criterion, optimizer,metric , scheduler=None ):
    #paddle.set_device('gpu:0')
    model.train()
    global_step = 0

    tic_train = time.time()
    num_train_epochs=paramparameters_dict['max_epoch']
    log_steps=paramparameters_dict['log_step']
    mode=model.mode#使用的编码器
    eval_step=paramparameters_dict['eval_step']
    save_step=paramparameters_dict['save_step']
    train_loss,train_acc,train_f1,train_iters=[],[],[],[]
    eval_loss,eval_acc,eval_iters,eval_f1=[],[],[],[]
    for epoch in range(num_train_epochs):
        for step,sample in enumerate(data_loader):
            if mode=="Bert" or mode=='Ernie':
                sentence1=sample['input_ids']
                sentence2=sample['token_type_ids']
            else:
                sentence1=sample['sentence1s']
                sentence2=sample['sentence2s']
            truelabels=sample['labels']
            input_data=[sentence1,sentence2]
            outputs = model(input_data)
            #计算损失
            loss = criterion(outputs, truelabels)
            #print(loss)
            outputs = F.softmax(outputs, axis=1)
            correct = metric.compute(outputs, truelabels)
            metric.update(correct)
            acc = metric.accumulate()
            #f1分数
            y_pred=np.argmax(outputs.numpy(),axis=-1)
            y_ture=truelabels.numpy()
            f1=f1_score(y_pred=y_pred,y_true=y_ture)
            #反向传播
            loss.backward()
            global_step += 1
            # 每间隔 log_steps 输出训练指标
            if global_step % log_steps == 0:
                print("global step %d, epoch: %d, batch: %d, loss: %.5f, accuracy: %.5f,F1-score:%.5f, speed: %.2f
                    step/s"
                % (global_step, epoch, step, loss, acc,f1,
                    log_steps / (time.time() - tic_train)))
```

```
            train_iters.append(global_step)
            train_acc.append(acc)
            train_f1.append(f1)
            train_loss.append(loss.numpy())
        if global_step%eval_step==0 :
            evalloss,evalacc,evalf1=evaluate(model, criterion, metric, vali_data_loader)
            eval_acc.append(evalacc)
            eval_loss.append(evalloss)
            eval_f1.append(evalf1)
            eval_iters.append(global_step)
        #优化器迭代一步
        optimizer.step()
        optimizer.clear_grad()
        if scheduler:
            scheduler.step()
        if global_step % save_step == 0:
            save_path=paramparameters_dict['save_path']+
            paramparameters_dict['encoder_name']+str(global_step)+'.pdparams'
            paddle.save(model.state_dict(),save_path)
    metric.reset()
save_path=paramparameters_dict['save_path']+paramparameters_dict['encoder_name']+'fc_finall.pdparams'
paddle.save(model.state_dict(),save_path)
return (train_loss,train_acc,train_f1,train_iters),(eval_loss,eval_acc,eval_f1,eval_iters)
```

6.6 使用不同的模型进行实验

使用不同的编码器进行实验:

6.6.1 加载数据集

```
#构建数据集
batch_size=paramparameters_dict['batch_size']
train_ds=Dataset(data_path='data/data/',mdoe='train',trans_fun=trans_func,word_embeding_func=embedding_func)
eval_ds=Dataset(data_path='data/data/',mdoe='dev',trans_fun=trans_func,word_embeding_func=embedding_func)
test_ds=Dataset(data_path='data/data/',mdoe='test',trans_fun=trans_func,word_embeding_func=embedding_func)
print('训练集大小: ',train_ds.__len__())
print('验证集大小: ',eval_ds.__len__())
print('测试集大小',test_ds.__len__())
train_data_loader = create_dataloader(train_ds,mode='train',batch_size=batch_size,batchify_fn=collate_func)
eval_data_loader = create_dataloader(eval_ds,mode='dev',batch_size=batch_size,batchify_fn=collate_func)
test_data_loader = create_dataloader(test_ds,mode='test',batch_size=batch_size,batchify_fn=collate_func)
print('训练一轮步数: ',train_ds.__len__()//batch_size)
```

相关信息输出:

```
训练集大小:  34334
验证集大小:  4316
测试集大小 3861
Batch_size: 128
训练一轮步数:  268
```

图 5: 数据部分信息输出

6.6.2 定义模型

```
#定义分类器模型
'''model=Classifer(encoder=bert_encode,is_embedding=False,mode='Bert')'''
#transformer
model=Classifer(encoder=transformer_encoder,is_embedding=False,mode='Transformer')
#lstm
'''model=Classifer(encoder=latm_encoder_api,is_embedding=True,is_pre_embedding=True,mode='transformer')'''
```

6.6.3 定义优化器和损失函数

优化器使用 Adam 优化器；损失函数使用交叉熵损失，具体参数见 6.1.2。**注意：**对于预训练模型使用了 warmup 的方法，与 LSTM、CNN、RNN 的优化方式不同。

```
#定义损失函数：交叉熵损失
criterion=nn.loss.CrossEntropyLoss()
#定义评估标准：准且率、二分了可以使用F1分数
acc_metric=paddle.metric.Accuracy()
#定义优化器——预训练模型
from paddlenlp.transformers import LinearDecayWithWarmup
num_training_steps = len(train_data_loader) * paramparameters_dict['max_epoch']
lr_scheduler = LinearDecayWithWarmup(paramparameters_dict['learning_rate'], num_training_steps, 0.0)
decay_params = [
    p.name for n, p in model.named_parameters()
    if not any(nd in n for nd in ["bias", "norm"])
]
optimizer = paddle.optimizer.AdamW(
    learning_rate=lr_scheduler,
    parameters=model.parameters(),
    weight_decay=0.0,
    apply_decay_param_fun=lambda x: x in decay_params)

#定义优化器——普通模型
optimizer =
    paddle.optimizer.AdamW(learning_rate=paramparameters_dict['learning_rate'],parameters=model.parameters())
#输出学习率
print(paramparameters_dict['learning_rate'])
```

6.6.4 训练、验证、预测

训练模型并保存实验结果：

```
#训练验证
start=time.perf_counter()
paddle.set_device('gpu:0')
train_data,eval_data=do_train(model,train_data_loader,eval_data_loader,criterion,optimizer,acc_metric)
end=time.perf_counter()
use_time=end-start
print('训练使用了：{:.5f}s'.format(use_time))
#测试
alltest(model,test_data_loader)
```

部分输出如下：

```
eval loss: 0.50131, accu: 0.74444
global step 1560, epoch: 2, batch: 485, loss: 0.44867, accuracy: 0.77031, speed: 0.02 step/s
global step 1570, epoch: 2, batch: 495, loss: 0.36407, accuracy: 0.78906, speed: 0.02 step/s
global step 1580, epoch: 2, batch: 505, loss: 0.38071, accuracy: 0.78229, speed: 0.02 step/s
global step 1590, epoch: 2, batch: 515, loss: 0.40021, accuracy: 0.78281, speed: 0.02 step/s
global step 1600, epoch: 2, batch: 525, loss: 0.46887, accuracy: 0.78844, speed: 0.02 step/s
eval loss: 0.50118, accu: 0.74444
global step 1610, epoch: 2, batch: 535, loss: 0.42877, accuracy: 0.76875, speed: 0.02 step/s
训练使用了：625.71518
```

图 6: 训练过程 log 输出

6.6.5 保存数据

```
np.save(paramparameters_dict['data_save_path']+paramparameters_dict['encoder_name']+'train_loss.npy',train_data[0])
np.save(paramparameters_dict['data_save_path']+paramparameters_dict['encoder_name']+'train_acc.npy',train_data[1])
np.save(paramparameters_dict['data_save_path']+paramparameters_dict['encoder_name']+'train_iter.npy',train_data[2])

np.save(paramparameters_dict['data_save_path']+paramparameters_dict['encoder_name']+'eval_loss.npy',eval_data[0])
np.save(paramparameters_dict['data_save_path']+paramparameters_dict['encoder_name']+'eval_acc.npy',eval_data[1])
np.save(paramparameters_dict['data_save_path']+paramparameters_dict['encoder_name']+'eval_iter.npy',eval_data[2])
```

6.7 数据可视化与实例预测

绘图比较简单，此处不再给出所有代码。预测结果展示如图 8。

```
model.eval()
label_dict={0:'不相似',1:'相似'}
test_example={"sentence1": "本月花呗为什么不能分期", "sentence2": "花呗分期为什么不能红包付款"}
ture_label='不相似'
inputs=trans_fun(test_example)
#print(inputs)
inputs_=[paddle.to_tensor([inputs['input_ids']]),paddle.to_tensor([inputs['token_type_ids']])]
pre=model(inputs_)
pre=F.softmax(pre,axis=1)
pre_label=np.argmax(pre.numpy()[0])
print('句子1为：'+test_example['sentence1'])
print('句子2为：'+test_example['sentence2'])
print('两个句子实际'+ture_label)
print('-----------------------------------------------')
print('编码器为:'+paramparameters_dict['encoder_name'])
print('两个类别概率为:',pre.numpy()[0])
print('预测为：'+label_dict[pre_label])
```

7 实验结果与分析

7.1 文本相似分类模型构建实验

7.1.1 实验结果

训练过程举例如下：

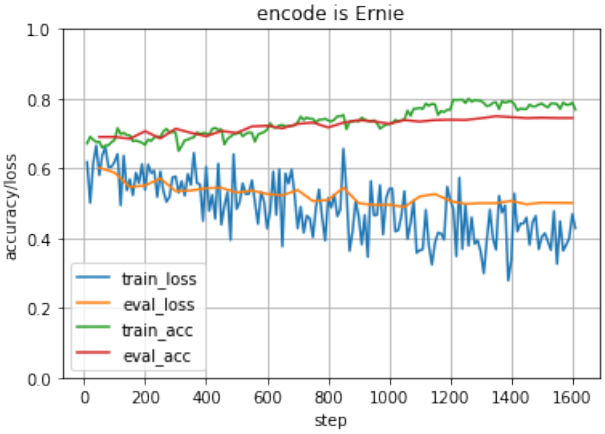


图 7: 训练过程 loss,accurancy,F1score 变化

模型在测试集上最佳准确率达到了 74.77%,F1 分数最佳为 0.6。

预测实例展示如图 8

```
句子1为：本月花呗为什么不能分期
句子2为：花呗分期为什么不能红包付款
两个句子实际不相似
-----------------------------------------------
编码器为:Bert
两个类别概率为： [9.9999106e-01 8.9411787e-06]
预测为：不相似
```

(a) Bert

```
句子1为：本月花呗为什么不能分期
句子2为：花呗分期为什么不能红包付款
两个句子实际不相似
-----------------------------------------------
编码器为:Ernie
两个类别概率为： [9.9985838e-01 1.4162147e-04]
预测为：不相似
```

(b) Ernie

```
句子1为：本月花呗为什么不能分期
句子2为：花呗分期为什么不能红包付款
两个句子实际不相似
--------------------------------------
编码器为:LSTM
两个类别概率为： [0.9484553  0.05154472]
预测为：不相似
```

(c) LSTM

```
句子1为：本月花呗为什么不能分期
句子2为：花呗分期为什么不能红包付款
两个句子实际不相似
--------------------------------------
编码器为:transformer
两个类别概率为： [0.8450979 0.1549021]
预测为：不相似
```

(d) Transformer

图 8: 预测实例输出

7.1.2 实验分析

通过实验结果，说明本实验成功建立了基于 AFQMC 蚂蚁金融语义相似度数据集的语义相似分类模型，并能够准确地对两个句子相似与否进行判断，说明实验基本任务成功。

同时可以看到，预训练模型输出的置信度更高，LSTM 较低；**分析认为**，预训练模型由于模型大、同时预训练的数据多，特征提取能力非常强，从而能够较好提取到语义信息，进行置信度很高的判断。

7.2 不同编码器对比试验

实验参数设置：

Bert and Ernie 参数设置（使用 5e-4 更大的学习率，预训练模型均无法学习）

最大学习率	Warmup	Batch size	max_length	max_epoch	优化器	学习准则
5e-5	线性	64	512	10	AdamW	交叉熵损失

表 1: 实验参数设置

LSTM、CNN、Transfromer 参数设置

学习率	Batch size	Encoder_dim	max_epoch	优化器	学习准则
5e-4	64	400	15(transformer:30)	AdamW	交叉熵损失
LSTM 层数	**LSTM_direction**	**Transformer 编码器层数**	**Transformer 头数**	**CNN 卷积核数目**	
2	bidirection	6	10	128	

表 2: 实验参数设置

7.2.1 实验结果

使用不同编码器后 (*** 表示模型使用了 word2vector 预训练词向量作为词嵌入**) 得到的模型性能评估结果如下:

编码器	LSTM*	CNN*	Bert	Ernie	LSTM	CNN	Transformer
训练集最佳准确率	0.9396	0.9446	0.9914	0.9492	0.8443	0.9736	0.9888
验证集最佳准确率	0.6907	0.6886	0.7335	0.7477	0.6902	0.6902	0.6914
训练集最佳 F1 分数	0.9677	0.9756	1.0000	0.9787	0.7742	1.0000	1.0000
验证集最佳 F1 分数	0.3236	0.3625	0.5756	0.5989	0.2640	0.3102	0.4634

表 3: 不同编码器实验结果

不同编码器训练过程可视化如图 9 图 10:

7.2.2 实验分析

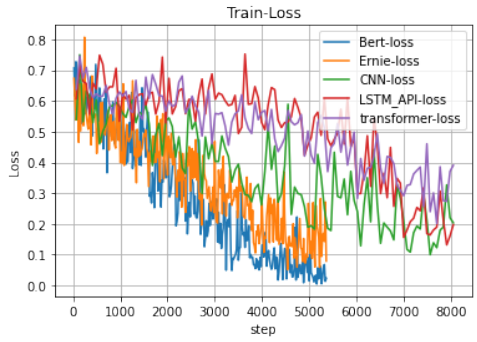
表格实验结果分析

(1) 通过观察表 3，在训练集上，预训练模型和 CNN、LSTM 作为编码器的拟合能力都比较强，准确率和 F1 分数都非常高，Bert 较为突出，训练集上的准确率和 F1 分数均是最高，同时也能看到 CNN 编码器和 Ernie 在训练集上的拟合效果非常接近，LSTM 相。说明 CNN、LSTM、Bert 等作为编码器，都具有很强的拟合能力，有效提取到了训练集的特征。**分析认为**，CNN 拟合效果接近预训练模型的部分原因可能是其可以卷积局部信息的能力在本实验中短文本中表现了出来且很突出。

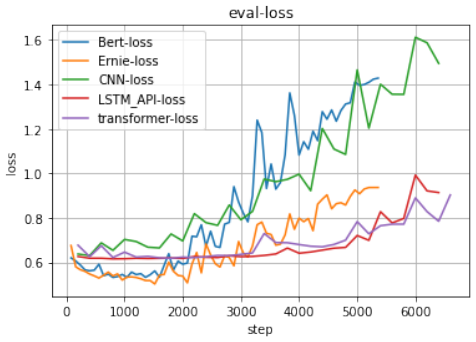
(2) 在验证集上，预训练模型和 CNN、LSTM 就表现出了较大的差距，预训练模型的准确率和 F1 分数非常高，Ernie 和 Bert 分别为 0.7477、0.5989 和 0.7335、0.5756；而 LSTM 和 CNN 验证集准确率为 0.69 左右，F1 分数为 0.3236 和 0.3625，和预训练模型相差较大；**分析认为**，这是由于预训练模型很大，同时可以对输入的文本进行动态编码，很好地提取的输入文本的上下文信息和语义，使得预训练模型具有较强的稳定性和更强的特征提取能力，而在验证集上泛化性能更好。

(3) 预训练模型之间的比较，Ernie 使用了更加多的 Mask 机制来训练模型，这也一定程度上提高了模型预见的问题和提取特征的能力和泛化能力，使得模型较于 Bert 在验证集上有更好的效果，准确率更好，F1 分数高出 Bert1% 和 2% 左右。

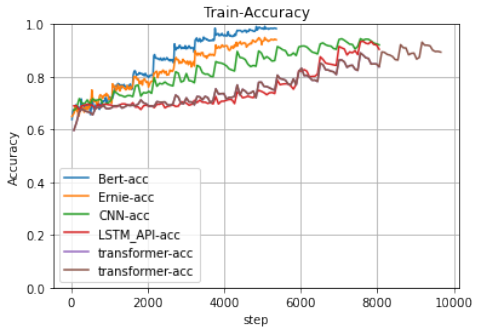
(4) LSTM 和 CNN 之间，通过对比，CNN 在数据集拟合上比 LSTM 好一点，但相差不多，但是在验证集上 LSTM 的准确率更高，CNN 的 F1 分数更好，一定程度上可认为 CNN 得到的效果更佳；**分析认为**，使用的数据集是较短的语句，CNN 的局部信息提取能力强，能够很好适应这种短文本语料，使得模型 F1 分数比 LSTM 高出了 4% 左右；而 LSTM 更加适合长文本，解决长序列问题，其优点在这里没有很好体现。因此，在短文本上，CNN 的编码能力也非常突出，LSTM 也很好，但是其在长序列问题上会更加优秀。



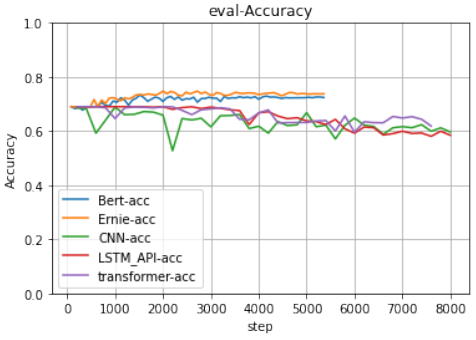
(a) 训练集损失变化



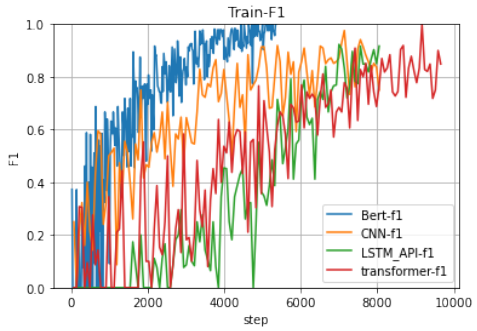
(b) 测试集损失变化



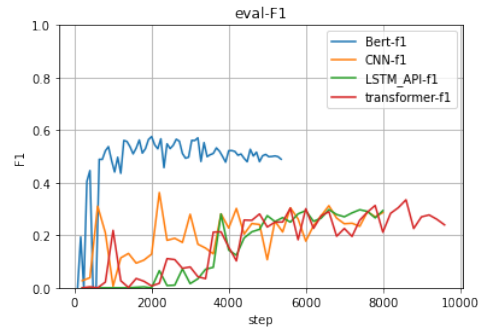
(c) 训练集准确率变化



(d) 测试集准确率变化

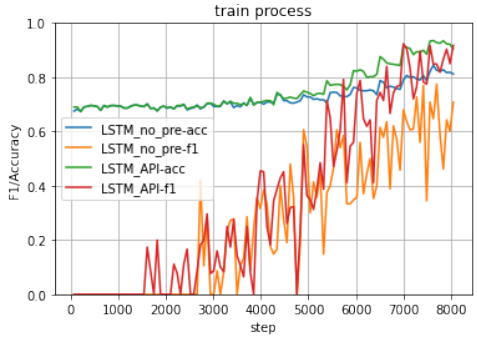


(e) 训练集 F1 变化

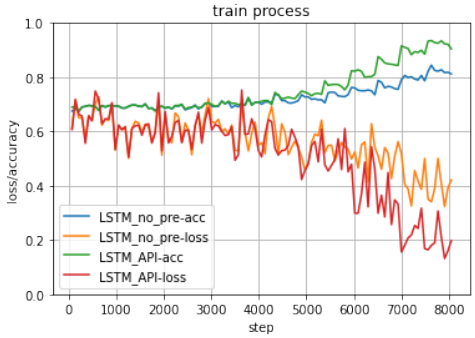


(f) 测试集 F1 变化

图 9: 不同编码器训练过程变化



(a) 训练集 F1/Accuracy 变化



(b) 训练集 Loss/Accuracy 变化

图 10: 是否使用预训练的 LSTM 训练过程变化

(5) 再观察 LSTM 使用预训练的词嵌入与否，能够发现，对于 LSTM，使用预训练的词嵌入效果更好，训练集上的准确率提高了 10% 左右，F1 分数也提高了 20%。**分析认为**，预训练的词向量一定程度上已有部分通用的语义表征，再使用 LSTM 在此任务上进行训练，相当于在预训练的词向量周围进行适当的调整，使得模型整体更好地提取了语义信息，给模型一个更好的初始化参数；而未使用预训练词嵌入的模型，很可能由于随机初始化而未找到较优点。

(6) 同时，也能在注意到，CNN 模型使用预训练与否，在训练集拟合上相差不大，说明 CNN 在短文本上的特征提取能力非常强；但是在验证集上 F1 分数使用了预训练词嵌入更好，这说明：预训练的词嵌入一定程度上具有文本的通用表征，提高了编码的泛化能力；而随机初始化的词嵌入，由于其可训练程度大，加之 CNN 特征提取能力强，就导致了其过分学习到了训练集的特征，泛化能力降低，导致验证集 F1 分数低了 5%。

(7) Transformer 经过较 LSTM 更长的时间训练， 模型在训练集上拟合得非常好，准确率和 F1 分数分别为 0.9888 和 1.000，基本接近效果最好得预训练模型 Bert（0.9914，1.0），与未使用预训练词嵌入的 CNN、LSTM 比较，其分别在准确率和 F1 分数上高出二者分别 1.52%,0(CNN) 和 14.45%，22.58%，即使是使用了预训练的词嵌入的 LSTM、CNN，训练集上也均没有 Transformer 好；一定程度上能够说明，Tansfromer 的特征提取能力非常强大，在数据集上的拟合能力比 LSTM 更好。 在验证集上，Transformer、CNN、LSTM 相比，虽然准确率相差不大，但是 Transformer 的 F1 分数达到了 0.4634，比 CNN 高 15.32%，比 LSTM 高出了 19.94%，说明 Transformer 的泛化能力更强，同时进一步说明，其提取特征的有效性比 CNN、LSTM 更好。 **分析认为**：Transformer 通过**自注意力机制**能够很好地关注到整个句子序列的信息；通过**多头机制**，在子空间对文本进行了更加丰富的特征提取与表达，并提升了模型注意其他位置的能力；Transformer 也具有更**深的结构**，利于提取更高层次的特征。Transformer 的这些特点使得其对文本整体的语义信息提取更加充分，对文本理解更加深入，因此其拟合能力和泛化性能均比 CNN、LSTM 更好，同时其还具有处理长序列问题的能力。

图像实验结果分析

(1) 首先通过观察不同编码器在训练集上的 loss、accuracy、f1score 的变化，可以很直观感受到预训练模型特征提取能力的强大，在训练集上很快学习了任务的特征，使得 loss 下降很快，准确率和 f1 迅速提升，快速拟合了数据集，同时 Bert 的拟合能力比 Ernie 更强。

(2) LSTM 和 CNN 之间，CNN 可以更快拟合数据集特征，loss 下降更快，准确率等提升也更加迅速，而 LSTM 前期相对缓慢，后期迅速提升，和 CNN 最终基本达到一致；**分析认为**，一反面是数据集为短文本数据集，CNN 很有效地提取了局部信息，使得模型性能快速增加；另一方面，我**认为**是 LSTM 的设计原因，遗忘门、输入门、输出门等等机制虽然让 LST 的性能带来了提升，但是一定程度上会带来学习上的问题——训练初期，这些‘门’的开关机制模型是不清楚的，需要学习，而基本学习到了‘门’的‘开关’后，就能够快速更加有效地学习如何更好提取文本信息，这就导致了 LSTM 前期学习相对缓慢，而后期其性能则迅速提升。

(3) 通过观察不同编码器在验证集上的变化过程，能够得到和表 3 基本一直的结论，同时在模型 loss 逐渐上升的过程中，Bert 等预训练模型在验证集上的准确率和 F1 分数基本保持不变，而 LSTM、CNN 则逐渐下降，这也一定程度上再次证明了预训练模型的强大泛化能力。同样，CNN 特征提取比 LSTM 强，这也导致其更加容易过拟合，使得后期模型泛化能力不佳。

(4) 观察图 10 使用预训练词向量与否，能够发现，除了之前提到的 LSTM 前期需要摸索基本的‘门控机制’的过程，使用了预训练的模型性能提升速度更快，loss 下降更快。**分析认为**，预训练的词向量一定程度上表征了语义，给模型一个较好的初始化参数，这可以使得模型快速学习到门控机制以及与词嵌入之间的衔接，然后迅速提升效果；而未使用的预训练的词嵌入的模型则需要花费更多时间调整词嵌入和门控机制之间的相适应的问题，使得学习过程相对较慢。同时预训练的词向量相当于给模型提供了一个通用语义基础，使得模型在特征提取上能力更精进，而最终效果也比未使用的更好。

(5) Transformer 的在 训练集上性能变化整体比较均匀，和 LSTM 较接近，同时前期较优于 LSTM，后期 LSTM 提升更快。**分析认为**，主要原因是 Transformer 模型比 LSTM 更大，达到更好的效果需要训练的时间更长，而 LSTM 的变化以上述一样，不再说明。 在验证集上，同样过拟合，但是 Transformer 在验证集的表现比 CNN 和 LSTM 更加稳定，损失增加更慢，准确率和 F1 分数保持更久。这也进一步说明了 Transformer 的有效特征提取的能力更强，泛化性能更好。

8 收获与感想

这是算是第一次较完整地进行一个 NLP 的实验项目，在实验过程中遇了很多问题，但收获是非常多的，我认为这次实验非常有意义，总结了很多经验，不仅是对于日后进行更多的 NLP 相关实验；也进一步加深了我对 NLP 的兴趣，了解了更多相关知识方法。

(1) . 首先，通过此实验，进一步了解到了 NLP 中文本理解双语句分类的基本流程，特别是进一步掌握好了 NLP 领域相关数据预处理的方法，例如：构建词表、线下将句子转换为 token-id 便于后期模型训练等等，在此前的 NLP 相关作业中都是在线转换，实际会降低很多速度；同时制作词表也是一种规范化的方式；还有就是对 token 序列进行 padding 时，Pad 位置的问题，在前会比在后效果好以及对序列继续宁排序，减少冗余的 pad 无效信息等；本次实验在此处收获挺多，更多是规范性以及 NLP 处理流程方面的经验等，通过此次实验对日后 NLP 相关实验有很大的帮助。

(2) . 同时，本次实验在 Transformer 和 LSTM 编码器对句子表征的时候出现了问题，使用了最大池化解决了问题，进一步学习到了 sequence2vector 的方法，同时和 CNN 卷积的时候池化进行对比，进一步学习到了其中的思想：不同特征维度的整体信息进一步提取，更能够表现出不同句子的差异性而利于分类（**个人认为**）。

(3) . 其次，本次实验是第一次使用 RNN、LSTM、Transformer 模型，首先通过理论学习基本掌握了原理，后特别是通过代码实践掌握了其原理、输入输出等；特别是实验过程中遇到的由于 pad 位置不对导致模型无法训练的问题，进一步体现出了 LSTM、RNN 的对时序序列建模的特点，同时也进一步展现了 LSTM 接受、遗忘等门控单元的作用，让我对其模型设计原理有了更多了解，同时对其可解释性和其作用有了深刻体会。

(4) . 同时，通过对实验结果的比较分析，进一步理解到了各个模型的原理、优缺点以及适应场景等等，对模型的特点也有了进一步的认识，对各个模型背后的思想也有了一定感受。

(5) . 本次实验也进一步提高了我的代码实践能力和代码规范化和复用的思想，如构建不同的类以便日后使用；在构建模型组网的时候，也加入了很多参数，将使用不同方法的模型集成在一起等等。

(6) . 同时感受到了与老师交流的重要性，此前实验，想通过自己的努力进行问题求解，但是一定不能一直这样，还是要与老师多交流，才能学习到更多的知识经验，也会少走弯路。

9 遇到的问题与解决方法

(1) . 本实验遇到最大的问题是自行构建的 LSTM、Transformer 以及调用的 paddle 的 API 均无法正常训练，出现模型 “偷懒”，将所有输出均为 0 得到.07 左右的准确率，但是 F1 分数一直为 0；起初认为是数据分布不均匀，降采样和调整预测阈值均无效；后认为是模型选取 LSTM、Transformer 最后一个输出作为句子表征有问题，尝试拼接所有输出后依旧有问题；同时尝试了学习率等等的参数调整也没有效果。后来，向老师咨询了相关问题，在老师帮助下，开始注意模型的输入是否有问题，发现出现**问题的原因**是**在整合 batch 时，对 token-id 序列进行 padding 时，将 PAD 的 id 号放在了句子后面，同时没有将 token 序列长度进行排序，而直接按照最长的进行填充**，**我认为**这样就会导致问题：某些短句子后出现很多 PAD 序号，PAD 的信息多分类作用是较小的，同时 LSTM 预先设计提取最后一个输入对应的输出作为句子表征，很大程度上，由于 LSTM 的遗忘功能等，就将序列初期的有效信息忽略掉或者覆盖了，这就导致编码器的输出是无效信息或有效信息较少，模型接受不到有效信息就无法进行正确分类；在使用 API 使用拼接作为句子表征时，同样的道理，过多的 PAD 会降低有效性息的比重，也会导致模型无法接收有效性息而无法训练。**解决方法是：** 对数据长度进行排序，使得每个 batch 的数据长度差距不大，p 使得 ad 的数量较少，不足以挤掉有效信息； 将 pad 填充在序列的前面，这样 LSTM、RNN 等时序模型就会保留更多后方的有效信息。虽然是最大的问题，困扰了很久，发现还是要及时向老师求助，毕竟自己经验不足，但这也是我实验中收获较大的地方，虽然这是一个问题，但是实际也一定程度上说明了，LSTM、RNN 等模型提取特征、句子表征的机理，对其有了进一步的认识。

(2) . 在编码器代码实践部分，第一次使用 paddle 的 LSTM 等 API，不熟练，通过查阅 API 文档的参数解释以及示例解决了问题；其次，在进行选择句子表征时遇到了困难，通过查阅 API 文档，使用 paddle.index_select 函数并不断尝试实验解决了问题。

(3) . 在进行数据预处理时耗时较长，主要想根据选择加载是否词嵌入的数据或者 token-io，原始句子等；在设计过程汇总设计到 Dateset、dataloader、模型的改动设计，同时遇到了一些数据类型不对口、数据输入维度不对口等问题，后来

使用了参数字典进行统一，逐渐解决了问题；同时由于 API 说明不充分，在设计预训练模型的多语句输入时，采用了逐一切词，再定义函数将 token_id 融合为预训练的输入；后来再网络上查找发现，已经有集成的 API 可以达到这种效果，但是之前看的同一个函数的 API 说明中并没有提到，在这些地方设计函数等等花费了一些时间，但是最后发现都是有集成函数的，但是很大程度上也提高了代码实践能力，算法设计能力，规范化的思想，对框架、任务流程更加熟悉，积累了很多经验。

(4) . 数据预处理问题。本实验在 paddle 上使用了 GPU, 但是训练速度依旧非常慢。起初认为是词嵌入时使用没有在模型内部而使用了 CPU 导致速度变慢，后在模型中加入了 Embedding 层，使用的词表时预训练的参数，很大，速度依旧很慢；将进一步尝试了自己制作词表，依旧训练较慢；最终基本判断为，在数据处理（语料转 token-id）时，使用了 online 的机制，在数据集中 ___getitem___ 时进行转换，会消耗很多时间（一开始这么做是图方便）；后来使用了 offline 的机制，事先将数据转换为 token-id（基本证明了确实是这个原因，仅训练集数据 (34334 条) 转换就使用了 1.5 小时左右）, 线下与处理数据后速度非常快，在此总结经验：NLP 转换 tokenid 等作为数据预处理部分线下进行，从而保证模型训练速度。

(5) .**Transformer 训练不了的问题**，自己搭建的 LSTM、Transformer 等模型一开始无法训练，一开始通过老师的帮助发现在输入出现了问题如 (1)，处理后使用 API 可以正常进行训练；后来认为应该是出现在句子表征出有问题，直接加和过于简单，通过老师的帮助，知道了一般使用句子的表征会使用 Maxpooling 和 Averagepooling 的方法（类似 CNN），通过更改使用最大池化层，成功训练了 Transformer 模型。**自己的疑问:**，CNN 中的卷积在词向量方向移动是没有意义的，因此会使用一维卷积的方法捕获 n-gram 信息，但是为什么池化在 token 方向是有用的；这里**我认为**可能是和图像卷积中的不同通道进行池化的思想相同，将不同特征维度的最显著的句子局部特征（多层编码后也包括了全局信息）提取出来，组合为具有句子本身特点显著的特征，进行分类，从而提高了分类能力，同样 Averagepooling，也是一个道理（后期查看了相关论文，确实如此）。