案例三：基于迁移学习的新闻分类

1.案例目的

• 掌握文本预处理方法；

• 掌握基于大型预训练模型进行迁移学习的方法

• 掌握使用pytorch深度学习框架搭建Bert和ERNIE模型进行文本分类。

• 掌握文本分类模型的评估

2.案例内容

迁移学习近年来在图形领域中得到了快速的发展，主要在于某些特定的领域不具备足够的数据，不能让深度模型学习的很好，需要从其它领域训练好的模型迁移过来，再使用该模型进行微调，使得该模型能很好地拟合少量数据的同时又具备较好的泛化能力（不过拟合）。

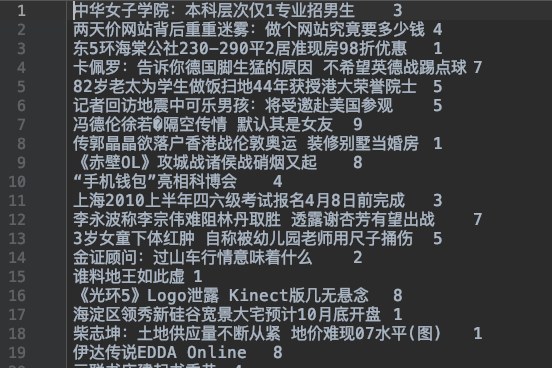
在迁移学习任务中，需要事先定义一组源数据集合，使用该集合训练得到预训练好的模型，该模型具备了源数据集合中的一些知识，在目标数据集合上微调该预训练的模型，使得模型能够很好地完成目标数据集合定义的任务，即完成了迁移学习。

本案例使用的数据是从THUCNews新闻数据集中中抽取的20万条新闻标题，使用近年来非常强大的预训练模型Bert和ERNIE模型进行迁移学习分类，研究大型预训练模型在实际工业场景中的应用。

数据集的文本长度在20到30之间。一共10个类别，每类2万条。数据以字为单位输入模型。类别：财经、房产、股票、教育、科技、社会、时政、体育、游戏、娱乐。

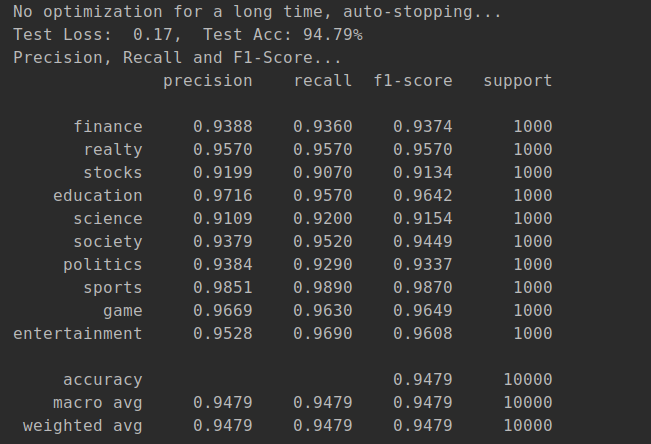


数据如下图所示:

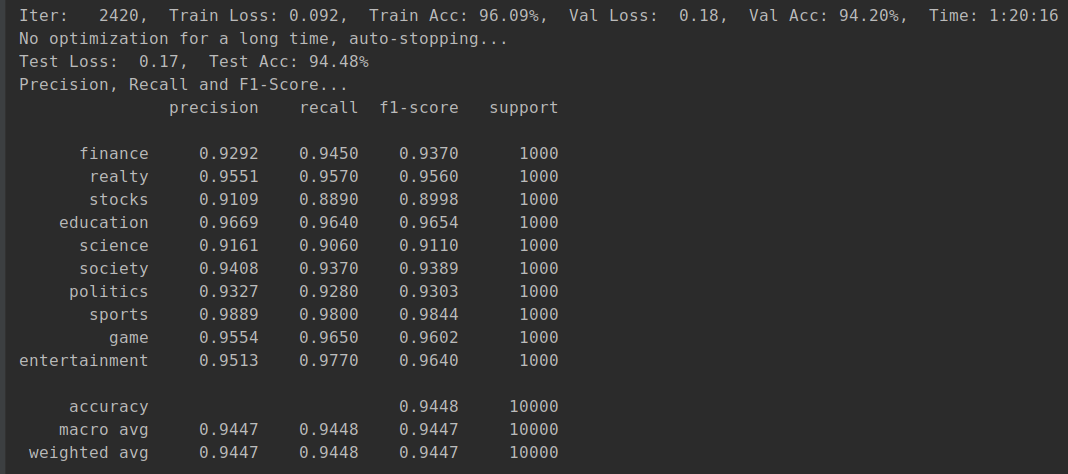


本案例使用2种模型做分类，这2种模型分别是Bert模型和ERNIE模型。案例中已经给出了基于海量数据训练好的预训练模型bert\_pretrain和ERNIE\_pretrain。

Bert预训练模型迁移学习训练结果的分类报告如下图所示:



ERNIE预训练模型迁移学习训练结果如下图所示:



从上面的图中可以看出,在该数据集上,ERNIE预训练模型性能相比bert预训练模型结果稍微差一些。.

3.案例知识点

• pytorch\_pretrained框架的使用；

• 文本分词和词表构建；

• 使用pytorch\_pretrained搭建Bert模型；

• 使用pytorch\_pretrained搭建ERNIE模型；

• 使用pytorch进行模型训练。

• 使用pytorch进行文本分类预测；

4.案例时长

共4学时，学时分配如下：

• 数据预处理（1学时）；

• 模型搭建（1学时）。

• 模型训练（2学时）

5.案例实验环境

• Windows 10 操作系统；

• Pycharm

• Python3.6

• pytorch 1.6.0

• tqdm

• importlib

6.案例分析

1）项目准备

2）数据预处理

3）Bert和ERNIE模型搭建

• 搭建Bert模型；

• 搭建ERNIE模型；

4）搭建模型训练、测试和预测流程

5）总装并完成训练和测试;

6）模型预测

7.案例实验过程

7.1项目准备

1) 创建项目

使用Pycharm工具，创建一个 Project，项目名称为“Bert-Classification”。

2) 创建“95598”目录

在Bert-Classification项目中目录上点右键，在弹出的菜单中选择“NewDirectory” ，创建一个新目录，如图4所示。

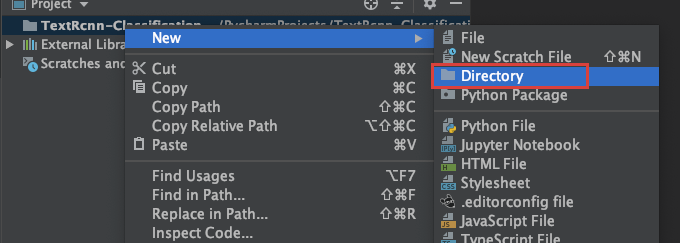


图4 创建目录菜单

在“New Directory”向导对话框，如图5所示，在对话框中需要进行以下操作：

1）创建目录“news”；

4）点击“OK”按钮。

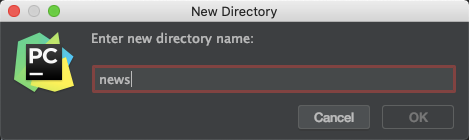


图5 创建目录向导

3) 创建其他目录

按照同样的方法创建目录“models”、“bert\_pretrain”和“ERNIE\_pretrain”，并在“95598”目录下创建“data”、“save\_dict”等目录，最终结果如图6所示：

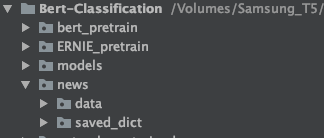


图6 创建目录结果

4) 复制文件

将案例提供的数据文件“train.txt”、“dev.txt”、“test.txt”和“class.txt”复制到“news-data”目录下，结构如图7所示：

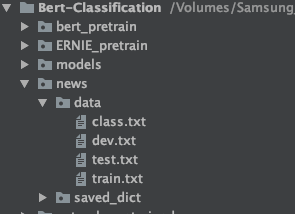


图7 复制数据文件结果

其中，各个文件的作用如下：

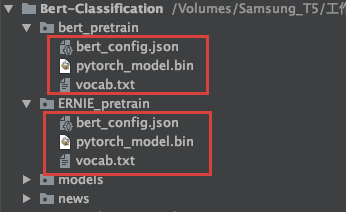
• train.txt：训练数据

• dev.txt：验证数据

• test.txt：测试数据

• class.txt：类别标签

然后，将试验提供的预训练文件分别复制到“bert\_pretrain”和“ERNIE\_pretrain”目录下，结构如下图所示：



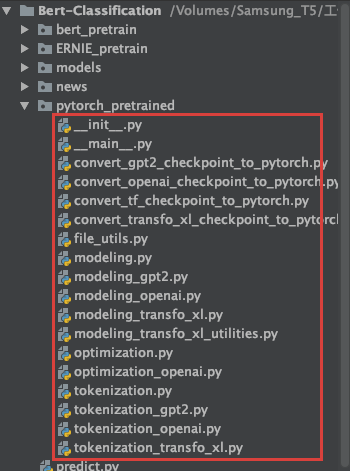
从图中可以看到，“bert\_pretrain”和“ERNIE\_pretrain”目录中均有三个文件，这就是前文提到的预训练模型的文件。各文件意义如下：

• bert\_config.json：模型的配置参数文件

• pytorch\_model.bin：预训练模型的参数

• vocab.txt：模型的词表

最后，将案例提供的目录“pytorch\_pretrained”整个目录复制到根目录下，如下图所示：



“pytorch\_pretrained”目录中有很多py脚本文件，均为pytorch\_pretrained中定义好的各种模型，本案例的Bert模型和ERNIE模型都是基于此框架实现，使用该框架实现Bert模型和ERNIE模型非常简单，只需要几行代码即可实现。

至此，项目准备工作已经完成，下面我们开始编写代码。

7.2数据预处理

深度学习模型训练时，由于计算量巨大，我们不能一次性将所有数据送入模型进行训练，因此我们需要对数据集进行批次迭代管理。主要步骤为：

• 对文本进行分词处理

• 长度短语规定长度的文本在后面填充<PAD>，长度长于规定长度的文本做截断处理，保证文本长度相等

• 返回词id序列，标签和真实长度作为数据集

• 对数据集进行分批，构建迭代器对象

在Bert-Classification项目中新建Python File文件“utils”。

1）导入需要使用的包。

【代码1】utils.py

import torch

from tqdm import tqdm

import time

from datetime import timedelta

PAD, CLS = '[PAD]', '[CLS]' # padding符号, bert中综合信息符号

2）编写实现数据集加载的函数build\_dataset。

build\_dataset函数负责加载训练集、验证集和测试集数据。

首先该函数接收一个参数config，这个参数是一个庞大的字典，其中保存的是模型的所配置参数，将在后面模型搭建的时候实现，这里我们先知道一下即可。我们在build\_dataset函数内部定义一个函数load\_dataset，函数接收两个参数：

• path：数据文件路径

• pad\_size：文本填充长度，即每个文本都被处理成这个长度，长截短补

该函数功能是根据给定的数据文件路径加载文本数据得到训练需要的文本特征向量和对应的真实标签。逻辑如下：

• 打开数据文件；

• 遍历每一行文本，对每一行文本执行如下操作：

• 获取content和label

• 对content进行分词

• 添加bert句子分类信息标志CLS

• 利用bert词表将每个token转换成对应的id，

• 计算词的个数并和pad\_size比较，长就截断，短就补'<PAD>'，同时生成对应的mask

• 最终，返回所有文本处理的结果，形式是一个元组构成的列表，形式为（（所有词的id，标签），真实长度，mask）

【代码2】utils.py

def build\_dataset(config):

def load\_dataset(path, pad\_size=32):

contents = []

with open(path, 'r', encoding='UTF-8') as f:#读取文件

for line in tqdm(f):#遍历每一行

lin = line.strip()

if not lin:

continue

content, label = lin.split('\t')#获取数据和标签

token = config.tokenizer.tokenize(content)#调用分词器进行文本分词

token = [CLS] + token#添加分类信息标志

seq\_len = len(token)

mask = []

token\_ids = config.tokenizer.convert\_tokens\_to\_ids(token)#转id

if pad\_size:

if len(token) < pad\_size:#句子长度小于pad\_size

#生成mask

mask = [1] \* len(token\_ids) + [0] \* (pad\_size - len(token))

#句子进行pad填充

token\_ids += ([0] \* (pad\_size - len(token)))

else:

mask = [1] \* pad\_size

token\_ids = token\_ids[:pad\_size]#截断

seq\_len = pad\_size

contents.append((token\_ids, int(label), seq\_len, mask))

return contents

最后，调用刚刚定义的函数load\_dataset，分别加载并转换训练集、验证集合测试集数据并返回。

最终，整个build\_dataset函数返回值为：训练集数据，验证集数据，测试集数据

【代码3】utils.py

#加载训练集

train = load\_dataset(config.train\_path, config.pad\_size)

#加载验证集

dev = load\_dataset(config.dev\_path, config.pad\_size)

#加载测试集

test = load\_dataset(config.test\_path, config.pad\_size)

return train, dev, test

3）编写实现数据批次迭代的类DatasetIterater。

DatasetIterater函数负责根据数据集构建一个迭代器，每次吐出一个批次的数据。首先我们实现DatasetIterater类的\_\_init\_\_初始化方法，该方法接收三个参数：

• batches：所有数据

• batch\_size：批次大小，就是没一小批的数据个数

• device：设备，CPU还是GPU

在该方法中，我们主要完成以下工作：

• 属性赋值，批次大小，数据，索引设备等；

• 计算批数，批数=总数据量/批次大小，注意这里可能不能整除；

• 定义一个属性residue，代表“总数据量/批次大小”是否是整数，是为False，不是为True；

【代码4】utils.py

class DatasetIterater(object):

def \_\_init\_\_(self, batches, batch\_size, device):

"""

根据数据集构建一个迭代器，每次吐出一个批次的数据

:param batches: 数据集

:param batch\_size: 批次大小

:param device: 设备 CPU还是GPU

"""

self.batch\_size = batch\_size

self.batches = batches

#计算总批次

self.n\_batches = len(batches) // batch\_size

self.residue = False # 记录batch数量是否为整数

if len(batches) % self.n\_batches != 0:

self.residue = True

self.index = 0

self.device = device

编写方法\_to\_tensor，该方法功能是将特征x，标签y，序列长度seq\_len和mask定义为torch的tensor。

【代码5】utils.py

def \_to\_tensor(self, datas):

#为x和y定义Tensor

x = torch.LongTensor([\_[0] for \_ in datas]).to(self.device)

y = torch.LongTensor([\_[1] for \_ in datas]).to(self.device)

# pad前的长度(超过pad\_size的设为pad\_size)

seq\_len = torch.LongTensor([\_[2] for \_ in datas]).to(self.device)

mask = torch.LongTensor([\_[3] for \_ in datas]).to(self.device)

return (x, seq\_len, mask), y

编写迭代器迭代方法\_\_next\_\_，该方法负责返回截取的一个批次的数据，其中的逻辑为：

• residue为True（代表数据不能整除批次）且index= n\_batches，表示要返回最后一批（余数）的数据，那么就截取到最后一条；

• index>= n\_batches,表示所有批次数据取完，那么就终止迭代；

• index<n\_batches，表示在中间，那么就截取一个批次大小的数据返回；

【代码6】utils.py

def \_\_next\_\_(self):

#判断批次数不为整数且正好是最后一个批次

if self.residue and self.index == self.n\_batches:

#截取到最后一条数据

batches = self.batches[self.index \* self.batch\_size: len(self.batches)]

self.index += 1

batches = self.\_to\_tensor(batches)

return batches

#index超过了 表示一个轮次完了

elif self.index >= self.n\_batches:

self.index = 0

raise StopIteration

else:

#截取一个批次长度数据

batches = self.batches[self.index \* self.batch\_size: (self.index + 1) \* self.batch\_size]

self.index += 1

batches = self.\_to\_tensor(batches)

return batches

最后，定义两个小方法：

• \_\_iter\_\_：重写的一个方法，返回对象本身self；

• \_\_len\_\_：重写的方法，返回迭代器的长度，其逻辑是：

• residue为真，表示数据不能整除，那么长度=n\_batches+1，因为要加上最后那个不能整除批次

• residue为假，表示能够整除，长度就等于n\_batches

【代码7】utils.py

def \_\_iter\_\_(self):

return self

def \_\_len\_\_(self):

if self.residue:

return self.n\_batches + 1

else:

return self.n\_batches

3）编写两个工具函数build\_iterator（负责创建数据迭代器对象）和get\_time\_dif（负责计算时间差）。

【代码8】utils.py

def build\_iterator(dataset, config):

iter = DatasetIterater(dataset, config.batch\_size, config.device)

return iter

def get\_time\_dif(start\_time):

"""获取已使用时间"""

end\_time = time.time()

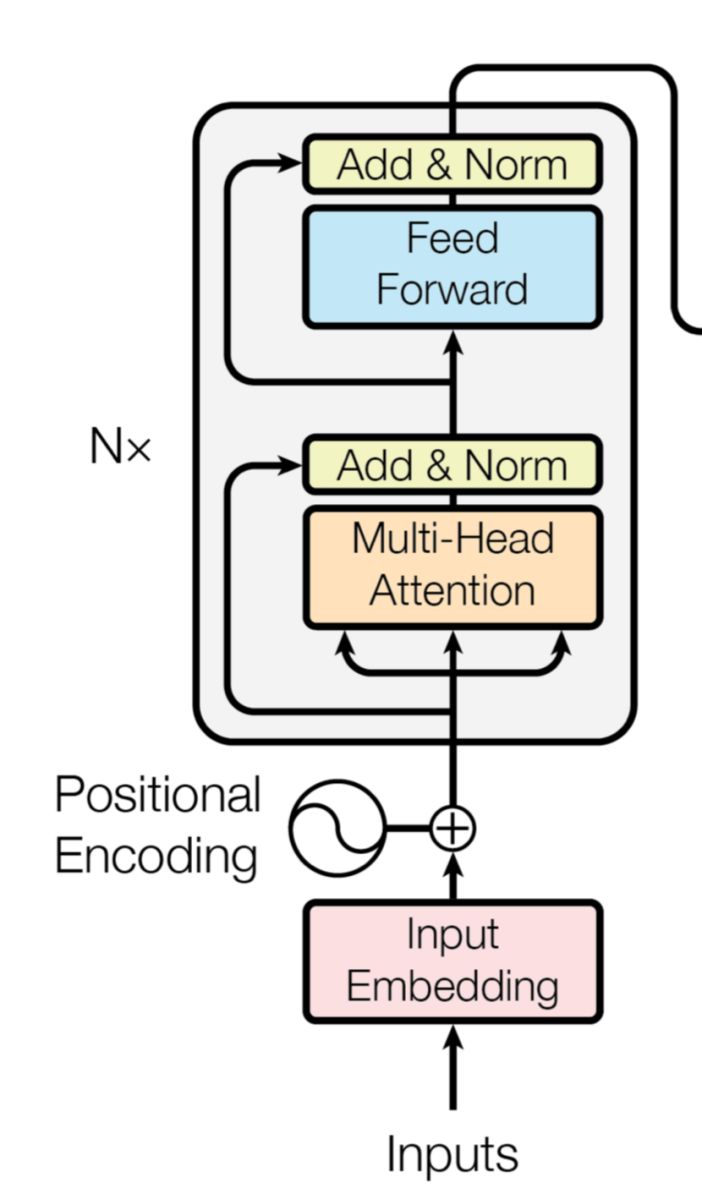
time\_dif = end\_time - start\_time

return timedelta(seconds=int(round(time\_dif)))

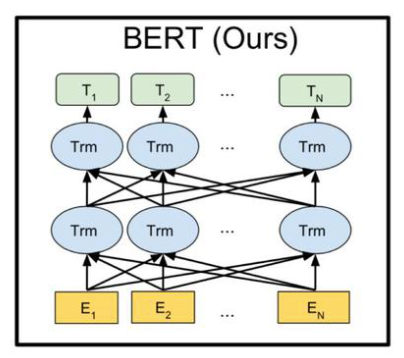
7.3搭建Bert模型和ERNIE模型

Bert和ERNIE是近年来出现的非常强大的基于Transformer的预训练模型。本案例目的是分别实现基于这两个模型的迁移学习分类。因此我们需要实现这两个模型的模型结构。

BERT的网络架构使用的是《Attention is all you need》中提出的多层Transformer结构，其最大的特点是抛弃了传统的RNN和CNN，通过Attention机制将任意位置的两个单词的距离转换成1，有效的解决了NLP中棘手的长期依赖问题。Transformer的结构在NLP领域中已经得到了广泛应用。BERT模型沿袭了GPT模型的结构，采用Transfomer的编码器作为主体模型结构。Transformer舍弃了RNN的循环式网络结构，完全基于注意力机制来对一段文本进行建模。Transformer的编码器结构如下图所示:



BERT采用跟GPT一样的“Fine-Tuning Approaches”预训练模式，分两个阶段：第一阶段采用双层双向Transformer模型通过MLM和NSP两种策略进行预训练；第二阶段采用Fine-Tuning的模式应用到下游任务。有人戏称：Word2Vec + ELMo + GPT = BERT，不过也并无道理，BERT吸收了这些模型的优点：“完形填空”的学习模式迫使模型更多依赖上下文信息预测单词，赋予了模型一定的纠错能力；Transformer模型相比LSTM模型没有长度限制问题，具备更好的能力捕获上下文信息特征；相比单向训练模式，双向训练模型捕获上下文信息会更加全面；其结构如下图所示：



我们使用pytorch深度学习框架基于pytorch\_pretrained搭建Bert网络结构，该部分我们主要实现两个类：

• Config：负责对Bert模型进行配置

• Model：实现Bert模型的网络结构

在Bert-Classification项目中models目录下新建Python File文件“bert”。

1）导入需要使用的包。

这里主要是从pytorch\_pretrained框架中导入实现好的两个类：

• BertModel：Bert网络模型的实现

• BertTokenizer：Bert分词器

【代码24】models/ bert.py

import torch

import torch.nn as nn

from pytorch\_pretrained import BertModel, BertTokenizer#导入

2）定义用来配置参数的配置类Config，该类用来保存配置模型的一些参数。每个参数的作用已经在代码中注释，这里不再做解释。

【代码25】models/ bert.py

class Config(object):

"""配置参数"""

def \_\_init\_\_(self, dataset):

self.model\_name = 'bert'#模型名称

self.train\_path = dataset + '/data/train.txt' # 训练集

self.dev\_path = dataset + '/data/dev.txt' # 验证集

self.test\_path = dataset + '/data/test.txt' # 测试集

self.class\_list = [x.strip() for x in open(

dataset + '/data/class.txt').readlines()] # 类别名单

self.save\_path = dataset + '/saved\_dict/' + self.model\_name + '.ckpt' # 模型训练结果

self.device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu') # 设备

self.require\_improvement = 1000 # 若超过1000batch效果还没提升，则提前结束训练

self.num\_classes = len(self.class\_list) # 类别数

self.num\_epochs = 3 # epoch数

self.batch\_size = 128 # mini-batch大小

self.pad\_size = 32 # 每句话处理成的长度(短填长切)

self.learning\_rate = 5e-5 # 学习率

self.bert\_path = './bert\_pretrain'#预训练模型的存储路径

self.tokenizer = BertTokenizer.from\_pretrained(self.bert\_path)#实例化分词器

self.hidden\_size = 768#预训练模型的隐藏层神经元数

3) 实现Bert模型。

这里我们使用pytorch\_pretrained来搭建Bert模型。具体步骤如下：

• 使用pytorch\_pretrained和预训练模型文件构建Bert层

• 设置bert模型中的参数求导，这样模型才能微调更新

• 定义全连接层 进行分类

【代码26】models/ bert.py

#继承nn.Module

class Model(nn.Module):

def \_\_init\_\_(self, config):

super(Model, self).\_\_init\_\_()

#使用pytorch\_pretrained和预训练模型文件构建Bert层

self.bert = BertModel.from\_pretrained(config.bert\_path)

for param in self.bert.parameters():#设置bert模型中的参数求导，这样模型才能微调更新

param.requires\_grad = True

#定义全连接层 进行分类

self.fc = nn.Linear(config.hidden\_size, config.num\_classes)

接着我们来编写前向传播方法forward，该方法中使用定义好的两个层来完成整个神经网络的前向计算，计算流程如下：

• 获取输入x中的句子id序列和对应的mask序列；

• 使用bert层计算句向量；

• 使用全连接层对句向量进行映射分类

【代码27】models/bert.py

def forward(self, x):

context = x[0] # 输入的句子

mask = x[2] # 对padding部分进行mask，和句子一个size，padding部分用0表示，如：[1, 1, 1, 1, 0, 0]

#bert计算出句向量

\_, pooled = self.bert(context, attention\_mask=mask, output\_all\_encoded\_layers=False)

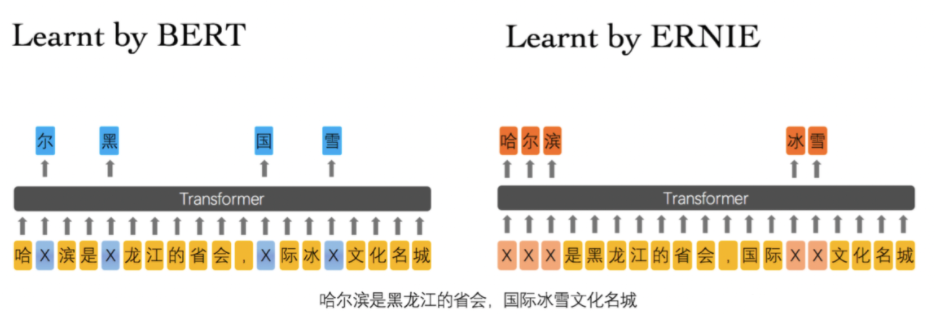
#使用全连接层对句向量进行映射分类

out = self.fc(pooled)

return out

2019年上半年，百度的开源深度学习平台PaddlePaddle发布了知识增强的预训练模型ERNIE，ERNIE通过海量数据建模词、实体及实体关系。相较于BERT学习原始语言信号，ERNIE直接对先验语义知识单元进行建模，增强了模型语义表示能力。

简单来说，百度ERNIE采用的Masked Language Model是一种带有先验知识Mask机制。可以在下图中看到，如果采用BERT随机mask，则根据后缀“龙江”即可轻易预测出“黑”字。引入了词、实体mask之后，“黑龙江”作为一个整体被mask掉了，因此模型不得不从更长距离的依赖（“冰雪文化名城”）中学习相关性。



我们使用pytorch深度学习框架基于pytorch\_pretrained搭建ERNIE网络结构，该部分我们主要实现两个类：

• Config：负责对ERNIE模型进行配置

• Model：实现ERNIE模型的网络结构

在Bert-Classification项目中models目录下新建Python File文件“ERNIE”。

1）导入需要使用的包。

这里主要是从pytorch\_pretrained框架中导入实现好的两个类：

• BertModel：Bert网络模型的实现

• BertTokenizer：Bert分词器

【代码24】models/ ERNIE.py

import torch

import torch.nn as nn

from pytorch\_pretrained import BertModel, BertTokenizer#导入

2）定义用来配置参数的配置类Config，该类用来保存配置模型的一些参数。每个参数的作用已经在代码中注释，这里不再做解释。

【代码25】models/ ERNIE.py

class Config(object):

"""配置参数"""

def \_\_init\_\_(self, dataset):

self.model\_name = 'ERNIE'#模型名称

self.train\_path = dataset + '/data/train.txt' # 训练集

self.dev\_path = dataset + '/data/dev.txt' # 验证集

self.test\_path = dataset + '/data/test.txt' # 测试集

self.class\_list = [x.strip() for x in open(

dataset + '/data/class.txt').readlines()] # 类别名单

self.save\_path = dataset + '/saved\_dict/' + self.model\_name + '.ckpt' # 模型训练结果

self.device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu') # 设备

self.require\_improvement = 500 # 若超过500batch效果还没提升，则提前结束训练

self.num\_classes = len(self.class\_list) # 类别数

self.num\_epochs = 3 # epoch数

self.batch\_size = 128 # mini-batch大小

self.pad\_size = 32 # 每句话处理成的长度(短填长切)

self.learning\_rate = 5e-5 # 学习率

self.bert\_path = './ERNIE\_pretrain'#预训练模型的存储路径

self.tokenizer = BertTokenizer.from\_pretrained(self.bert\_path)#实例化分词器

self.hidden\_size = 768#预训练模型的隐藏层神经元数

3) 实现Bert模型。

这里我们使用pytorch\_pretrained来搭建Bert模型。具体步骤如下：

• 使用pytorch\_pretrained和预训练模型文件构建Bert层

• 设置bert模型中的参数求导，这样模型才能微调更新

• 定义全连接层进行分类

【代码26】models/ ERNIE.py

#继承nn.Module

class Model(nn.Module):

def \_\_init\_\_(self, config):

super(Model, self).\_\_init\_\_()

#使用pytorch\_pretrained和预训练模型文件构建Bert层

self.bert = BertModel.from\_pretrained(config.bert\_path)

for param in self.bert.parameters():#设置bert模型中的参数求导，这样模型才能微调更新

param.requires\_grad = True

#定义全连接层 进行分类

self.fc = nn.Linear(config.hidden\_size, config.num\_classes)

接着我们来编写前向传播方法forward，该方法中使用定义好的两个层来完成整个神经网络的前向计算，计算流程如下：

• 获取输入x中的句子id序列和对应的mask序列；

• 使用bert层计算句向量；

• 使用全连接层对句向量进行映射分类

【代码27】models/ ERNIE.py

def forward(self, x):

context = x[0] # 输入的句子

mask = x[2] # 对padding部分进行mask，和句子一个size，padding部分用0表示，如：[1, 1, 1, 1, 0, 0]

#bert计算出句向量

\_, pooled = self.bert(context, attention\_mask=mask, output\_all\_encoded\_layers=False)

#使用全连接层对句向量进行映射分类

out = self.fc(pooled)

return out

7.4搭建模型训练、测试和预测流程

Bert和ERNIE网络实现后，就可以开始搭建训练流程了，但是考虑到训练和测试流程的代码较多，为了项目整体的易读性，我们将训练、验证和测试过程分别进行封装，下面我们将实现5个主要函数，分别为：

• init\_network：该函数负责在训练开始时，对模型中的参数进行初始化

• train：该函数接收模型、训练集迭代器、验证集迭代器和测试集迭代器并完成整个训练和测试过程

• evaluate：该函数负责在验证集上进行验证

• test：该函数负责在测试集上进行评估

• predict：该函数实现模型的推理预测

在Bert-Classification项目中新建Python File文件“train\_eval”。

1）导入需要使用的包。

【代码28】train\_eval.py

import numpy as np

import torch

import torch.nn as nn

import torch.nn.functional as F

from sklearn import metrics

import time

from utils import get\_time\_dif

from pytorch\_pretrained.optimization import BertAdam

2）编写初始化权重函数init\_network，该函数负责在构建模型后对模型的参数进行初始化。主要参数如下：

• model：需要进行参数初始化的模型；

• method：参数初始化的方法，就是用什么初始化器对参数进行初始值赋值；

• exclude：指定哪些参数不进行初始化；

• seed：初始化一般都是随机数，这个就是设定随机数种子；

该函数中的逻辑如下：

• for循环遍历模型的参数；

• 判断参数名字是不是不在排除（exclude）的参数中；

• 判断method，根据不同的method设置不同的初始化器进行初始化；

【代码29】train\_eval.py

# 权重初始化，默认xavier

def init\_network(model, method='xavier', exclude='embedding', seed=123):

#遍历模型中的每一个参数

for name, w in model.named\_parameters():

#排除embedding参数，这个加载预训练词向量

if exclude not in name:

if len(w.size()) < 2:#参数长度小于2

continue

#如果是w参数

if 'weight' in name:

if method == 'xavier':

nn.init.xavier\_normal\_(w)

elif method == 'kaiming':

nn.init.kaiming\_normal\_(w)

else:

nn.init.normal\_(w)

#如果是b参数

elif 'bias' in name:

nn.init.constant\_(w, 0)

else:

pass

3） 编写执行训练流程的函数train。该函数负责训练，并每隔100次迭代在验证集上进行验证，所有的epochs走完后在测试集上进行评估。由于该函数代码较长，我们分几个部分进行讲解。

第一部分，我们先对训练过程中需要使用的一些变量和工具进行初始化赋值，主要步骤如下：

• 指定模型为训练（train）模式，在该模式下，模型会自动计算梯度并更新参数；

• 提取模型中的所有参数，过滤掉不需要训练的参数['bias', 'LayerNorm.bias', 'LayerNorm.weight']

• 指定优化器optimizer，作用是在训练的过程中对参数进行更新，这里我们使用的是BertAdam优化器；

• 定义一些记录训练过程的变量

在train\_eval.py中编写如下代码：

【代码30】train\_eval.py

def train(config, model, train\_iter, dev\_iter, test\_iter):

start\_time = time.time()

model.train()#设置为训练模式

param\_optimizer = list(model.named\_parameters())#提取模型所有参数

no\_decay = ['bias', 'LayerNorm.bias', 'LayerNorm.weight']#不需要训练的参数

#过滤掉不需要训练的参数

optimizer\_grouped\_parameters = [

{'params': [p for n, p in param\_optimizer if not any(nd in n for nd in no\_decay)], 'weight\_decay': 0.01},

{'params': [p for n, p in param\_optimizer if any(nd in n for nd in no\_decay)], 'weight\_decay': 0.0}]

#指定BertAdam优化器，对需要训练的参数进行训练更新

optimizer = BertAdam(optimizer\_grouped\_parameters,

lr=config.learning\_rate,

warmup=0.05,

t\_total=len(train\_iter) \* config.num\_epochs)

total\_batch = 0 # 记录进行到多少batch

dev\_best\_loss = float('inf')

last\_improve = 0 # 记录上次验证集loss下降的batch数

flag = False # 记录是否很久没有效果提升

第二部分，我们主要实现模型训练，主要步骤如下：

• for循环num\_epochs，即将训练集重复迭代num\_epochs次；

• 每次循环进来，首先让学习率衰减一次

• 然后遍历训练集的所有批次数据，对每一个批次的数据作如下几步计算：

• 计算模型输出值

• 梯度清零，准备计算本批次的梯度

• 计算loss损失函数的值

• 损失函数反向传播计算梯度

• 优化器根据梯度更新参数，完成一次迭代

【代码31】train\_eval.py

#遍历num\_epochs次

for epoch in range(config.num\_epochs):

print('Epoch [{}/{}]'.format(epoch + 1, config.num\_epochs))

for i, (trains, labels) in enumerate(train\_iter):

outputs = model(trains)#计算输出值

model.zero\_grad()#清零导数

loss = F.cross\_entropy(outputs, labels)#计算损失

loss.backward()#反向传播求导

optimizer.step()#更新参数

第三部分，这一部分我们主要完成模型在验证机上的效果检验，通过观察模型在验证集上的效果，我们能够更好的了解模型优化的过程，并可以通过早停技术来防止模型过度训练导致过拟合。主要的步骤如下：

• 计算模型在训练机上的准确率；

• 调用evaluate函数（后面马上实现）使用验证集评估模型效果，计算出验证集准确率和损失值；

• 判断验证集损失值（dev\_loss）是否小于历史验证集损失值最小值（dev\_best\_loss），如果小于，就保存模型并更新dev\_best\_loss和last\_improve；这样做的目的是保存在验证集行最好的模型；

【代码32】train\_eval.py

if total\_batch % 10 == 0:

# 每多少轮输出在训练集和验证集上的效果

true = labels.data.cpu()#提取真实值

predic = torch.max(outputs.data, 1)[1].cpu()#提取预测值

train\_acc = metrics.accuracy\_score(true, predic)#计算训练集准确率

#计算验证集准确率和损失

dev\_acc, dev\_loss = evaluate(config, model, dev\_iter)

if dev\_loss < dev\_best\_loss:#如果本轮loss小于历史最小loss

dev\_best\_loss = dev\_loss

#保存模型

torch.save(model.state\_dict(), config.save\_path)

improve = '\*'

last\_improve = total\_batch

else:

improve = ''

time\_dif = get\_time\_dif(start\_time)

msg = 'Iter: {0:>6}, Train Loss: {1:>5.2}, Train Acc: {2:>6.2%}, Val Loss: {3:>5.2}, Val Acc: {4:>6.2%}, Time: {5} {6}'

print(msg.format(total\_batch, loss.item(), train\_acc, dev\_loss, dev\_acc, time\_dif, improve))

model.train()

第四部分，在验证集评估之后，我们可以使用早停技术来防止过拟合，其思想是记录每次保存最好模型的步数last\_improve，然后使用当前的总步数total\_batch-last\_improve，如果超过设定的阈值1000次，也就意味着模型在1000次迭代中都没有在验证集上取得更好的效果，就终止训练；

最后，在模型所有的epochs迭代完毕后，在测试集上评估模型。

【代码33】train\_eval.py

total\_batch += 1

if total\_batch - last\_improve > config.require\_improvement:

# 验证集loss超过1000batch没下降，结束训练

print("No optimization for a long time, auto-stopping...")

flag = True

break

if flag:

break

# 最后在测试集上评估

test(config, model, test\_iter)

4）编写执行验证集验证的函数evaluate。该函数接收参数有以下四个：

• config：配置参数字典，该参数之前我们已经介绍过

• model：训练的网络模型

• data\_iter：数据集迭代器，负责接收数据

• test：控制时候是测试，是测试我们就输出分类报告

该部分代码较长，我们分两个部分进行讲解。

第一部分，我们完成以下步骤：

• 指定模型为eval模式，即评估模式，这样模型就不会计算梯度和更新参数；

• 遍历数据的所有批次数据，对每一个批次我们做以下操作：

• 计算模型输出值

• 计算损失并加入总损失列表中，后续用来计算总损失值

• 计算预测值，并将预测值和真实值加入列表，后续用来评估整个数据集的结果

• 遍历完所有数据后，计算准确率

【代码34】train\_eval.py

def evaluate(config, model, data\_iter, test=False):

model.eval()

loss\_total = 0

predict\_all = np.array([], dtype=int)

labels\_all = np.array([], dtype=int)

with torch.no\_grad():#不计算梯度

for texts, labels in data\_iter:#遍历数据集

outputs = model(texts)#计算输出值

loss = F.cross\_entropy(outputs, labels)#计算损失

loss\_total += loss

labels = labels.data.cpu().numpy()

predic = torch.max(outputs.data, 1)[1].cpu().numpy()#计算预测值

labels\_all = np.append(labels\_all, labels)

predict\_all = np.append(predict\_all, predic)

#计算准确率

acc = metrics.accuracy\_score(labels\_all, predict\_all)

第二部分，数据集评估完成后，我们判断是否是测试集，如果是测试集，我们就输出分类报告（classification\_report）和混淆矩阵（confusion），最终返回准确率、平均损失、分类报告（测试才返回）和混淆矩阵（测试才返回）。

【代码35】train\_eval.py

if test:#如果是test测试

#输出分类报告

report = metrics.classification\_report(labels\_all,

predict\_all,

target\_names=config.class\_list,

digits=4)

#输出混淆矩阵

confusion = metrics.confusion\_matrix(labels\_all, predict\_all)

return acc, loss\_total / len(data\_iter), report, confusion

return acc, loss\_total / len(data\_iter)

5）编写执行测试集评估的函数test。该函数接收参数有以下四个：

• config：配置参数字典，该参数之前我们已经介绍过

• model：训练的网络模型

• data\_iter：数据集迭代器，负责接收数据

该函数主要作用是在训练过程完成后，使用测试集数据对训练的模型进行评估。主要流程为：

• 加载模型

• 指定模型为验证模式

• 调用evaluate函数计算准确率、损失、分类报告和混淆矩阵

• 输出信息

【代码36】train\_eval.py

def test(config, model, test\_iter):

# 加载模型

model.load\_state\_dict(torch.load(config.save\_path, ,map\_location=torch.device(config.device)))

# 指定模型为验证模式

model.eval()

start\_time = time.time()

#测试集评估

test\_acc, test\_loss, test\_report, test\_confusion = evaluate(config, model, test\_iter, test=True)

msg = 'Test Loss: {0:>5.2}, Test Acc: {1:>6.2%}'

print(msg.format(test\_loss, test\_acc))

print("Precision, Recall and F1-Score...")

print(test\_report)

print("Confusion Matrix...")

print(test\_confusion)

time\_dif = get\_time\_dif(start\_time)

print("Time usage:", time\_dif)

5）编写实现预测过程的函数predict。该函数接收参数有以下四个：

• config：配置参数字典，该参数之前我们已经介绍过

• model：训练的网络模型

• data\_iter：数据集迭代器，负责接收数据

该函数主要作用是使用训练好的模型，对给定的数据进行预测，实现模型推理过程。主要流程为：

• 加载模型

• 遍历给定数据

• 计算模型输出值并提取预测类别下标

• 根据预测类别下标得到对应的类别标签

【代码36】train\_eval.py

def predict(config,model,data\_iter):

#加载模型

model.load\_state\_dict(torch.load(config.save\_path, ,map\_location=torch.device(config.device)))

#验证模式

model.eval()

predict\_all = np.array([], dtype=int)

with torch.no\_grad():#不计算梯度

for texts, labels in data\_iter:#遍历数据

outputs = model(texts)#计算输出值

predic = torch.max(outputs.data, 1)[1].cpu().numpy()#计算预测类别下标

predict\_all = np.append(predict\_all, predic)

results=[config.class\_list[i] for i in predict\_all]#根据类别下标返回对应的类别标签

return results

7.5总装并完成训练和测试

编写好了训练、验证和测试评估的流程函数之后，我们来将整个训练过程进行总装。整个训练的流程为：

• 加载数据

• 加载并配置模型

• 进行训练

在Bert-Classification项目中新建Python File文件“run”。

1）导入需要使用的包。

【代码37】run.py

import time

import torch

import numpy as np

from train\_eval import train, init\_network

from importlib import import\_module

import argparse

from utils import build\_dataset, build\_iterator, get\_time\_dif

2）构建ArgumentParser参数对象。该对象可以接收命令行参数，这里我们设定了一个参数：

• --model：该参数指定使用的模型是“bert”还是“ERNIE”

【代码38】run.py

parser = argparse.ArgumentParser(description='Chinese Text Classification')

parser.add\_argument('--model', type=str, default='bert',help='choose a model: bert, ERNIE')

args = parser.parse\_args()

3）写main函数，实现训练。这个函数中，我们使用之前编写的类和函数来实现完整的模型训练流程，下面我们详细介绍一下整个环节：

• 指定数据集名称为news，这样模型就会去访问对应文件夹的数据；

• 导入对应的模型文件为x，并调用该文件的Config类生成参数配置对象

• 调用build\_dataset函数加载训练集、验证集和测试集

• 构建训练集、验证集、测试集迭代器，为训练做数据准备

• 利用参数配置对象config配置并实例化model对象

• 调用之前实现的train函数完成模型训练和评估

【代码39】run.py

if \_\_name\_\_ == '\_\_main\_\_':

dataset = 'news' # 指定数据集文件目录

model\_name = args.model # bert

x = import\_module('models.' + model\_name)#导入模型

config = x.Config(dataset)#生成配置文件

np.random.seed(1)

torch.manual\_seed(1)

torch.cuda.manual\_seed\_all(1)

torch.backends.cudnn.deterministic = True # 保证每次结果一样

start\_time = time.time()

# 加载数据集 训练集、验证集、测试集

print("Loading data...")

train\_data, dev\_data, test\_data = build\_dataset(config)

# 构建训练集、验证集、测试集迭代器

train\_iter = build\_iterator(train\_data, config)

dev\_iter = build\_iterator(dev\_data, config)

test\_iter = build\_iterator(test\_data, config)

time\_dif = get\_time\_dif(start\_time)

print("Time usage:", time\_dif)

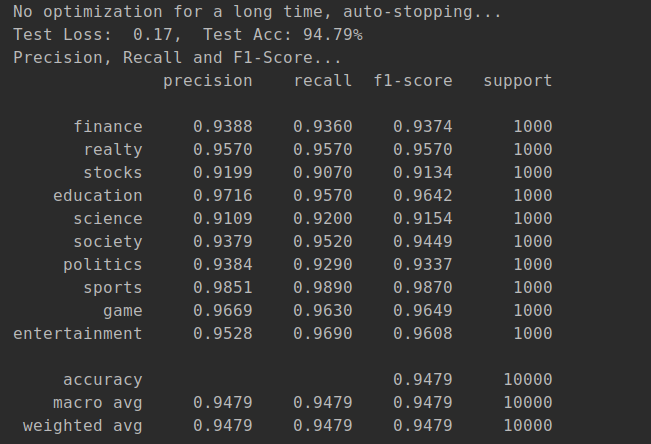
#利用配置参数实例化模型

model = x.Model(config).to(config.device)

#模型训练

train(config, model, train\_iter, dev\_iter, test\_iter)

4）运行run.py文件，模型开始训练，模型训练完毕后会输出如下图所示的分类报告结果：



由于Bert模型和ERNIE模型非常庞大，在配备了GTX1070TI的深度学习服务器上也需要2个小时的训练时间，因此在这里同学们只需要呼起训练感受一下即可，后续步骤中我们将提供训练好的模型文件。

注意：如果在运行过程中出现“cuda out of memory”错误，如下图所示：



该错误表明我们的的显存不足，说明我们训练的批次batch\_size过大，我们需要到“models/bert.py”中将batch\_size改小，如下图所示：



7.6模型预测

模型训练完成后，我们可以使用模型对给定的投诉文本进行预测分类，这里我们使用测试集数据进行预测。整个预测的流程为：

• 加载数据

• 加载并配置模型

• 读取预测文本

• 使用模型进行预测

• 展示预测结果

在Bert-Classification项目中新建Python File文件“predict”。

1）导入需要使用的包。

**【代码37】predict.py**

from train\_eval import predict

from importlib import import\_module

import argparse

from utils import build\_dataset, build\_iterator

from tqdm import tqdm

2）构建ArgumentParser参数对象。该对象可以接收命令行参数，这里我们设定了一个参数：

• --model：该参数指定使用的模型是“bert”还是“ERNIE”。

【代码37】predict.py

parser = argparse.ArgumentParser(description='Chinese Text Classification')

parser.add\_argument('--model', type=str, default='bert',help='choose a model: bert, ERNIE')

args = parser.parse\_args()

3）写main函数，实现训练。这个函数中，我们使用之前编写的类和函数来实现完整的模型训练流程，下面我们详细介绍一下整个环节：

• 指定数据集名称为news，这样模型就会去访问对应文件夹的数据；

• 导入对应的模型文件为x，并调用该文件的Config类生成参数配置对象

• 调用build\_dataset函数加载测试集

• 构建测试集迭代器，为预测做数据准备

• 读取测试集数据原始的文本

• 利用参数配置对象config配置并实例化model对象

• 调用之前实现的predict函数完成预测

• 遍历文本和预测结果进行展示

【代码39】predict.py

if \_\_name\_\_ == '\_\_main\_\_':

dataset = 'news' # 指定数据集

model\_name = args.model #指定模型

x = import\_module('models.' + model\_name)#加载模型

config = x.Config(dataset)#生成配置

\_, \_, test\_data = build\_dataset(config)#读取数据集

test\_iter = build\_iterator(test\_data[:200], config)#构建迭代器

texts=[]#读取原始文本

with open(config.test\_path, 'r', encoding='UTF-8') as f:

for line in tqdm(f):

lin = line.strip()

if not lin:

continue

content, label = lin.split('\t')

texts.append(content)

#实例化模型对象

model = x.Model(config).to(config.device)

#调用predict函数实现模型加载和预测

preds=predict(config,model,test\_iter)

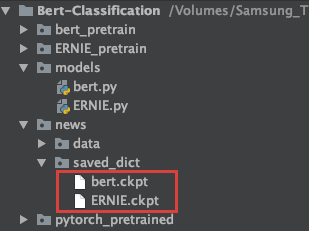
#展示预测结果

for text,label in zip(texts[:200],preds):

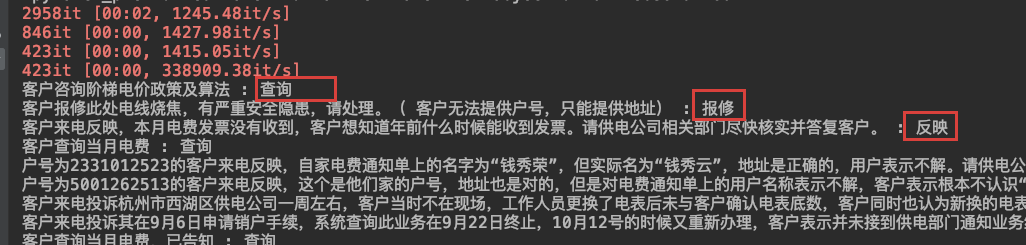
print(text,':',label)

4）拷贝案例提供的模型文件

我们将案例提供的两个训练好的模型文件bert.ckpt和ERNIE.ckpt拷贝到目录“news/saved\_dict”中，如下图所示：



5）运行predict.py文件，模型开始预测，输出结果如下图所示：

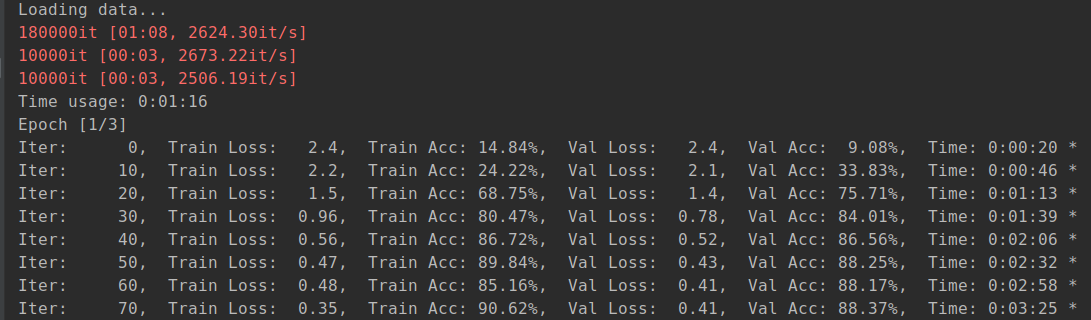


8.案例实验结果（结论）

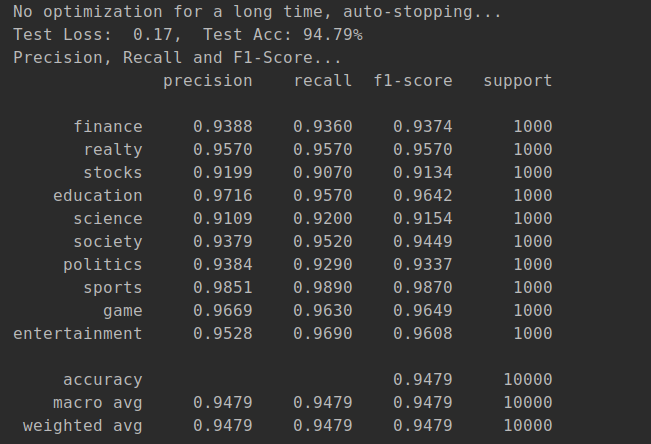
1）模型训练。

本次案例中，我们分别使用了Bert和ERNIE两种大型预训练模型来进行迁移学习文本分类训练，因此我们需要分别使用Bert和ERNIE来进行训练。

首先，我们使用Bert预训练模型进行训练，因为在代码中我们已经指定模型的名称是“bert”，因此我们直接运行run.py文件即可，训练过程如下图所示：

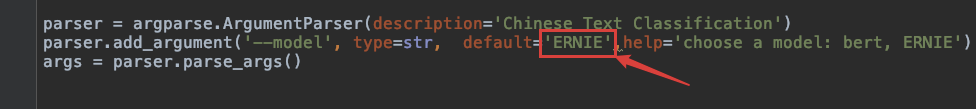


训练完成后，会自动在测试集上进行评估，最终解雇如下图所示：

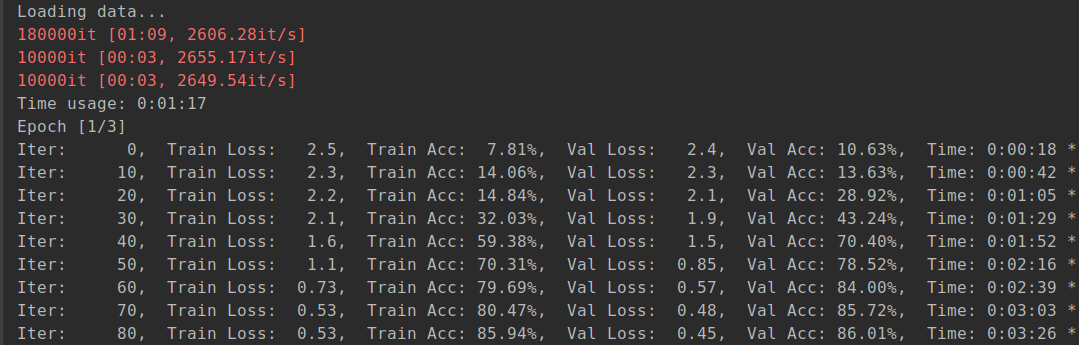


从结果中可以看到，模型的分类准确率为0.9479，效果非常好。下面我们来训练ERNIE模型。

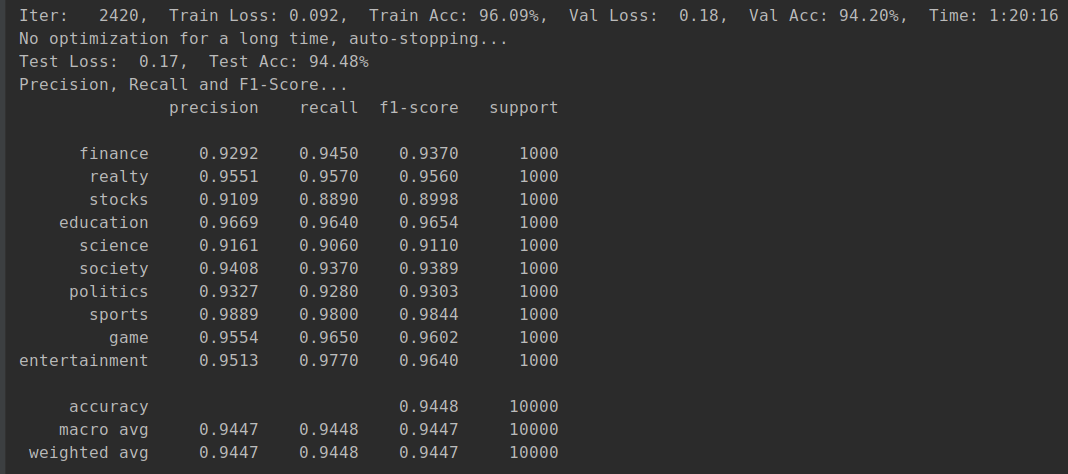
我们将run.py文件中的argparse参数中的—model参数由“bert”修改为“ENRIE”，如下图所示：



然后，运行run.py文件，训练过程如下图所示：



训练完成后，会自动在测试集上进行评估，最终解雇如下图所示：

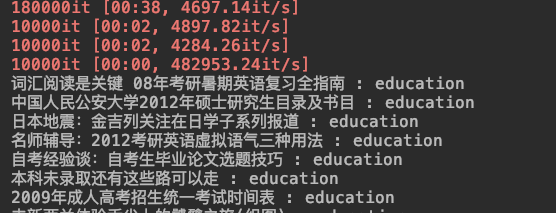


使用ERNIE预训练模型迁移学习的准确率为94.48%，效果与bert模型相差无几。

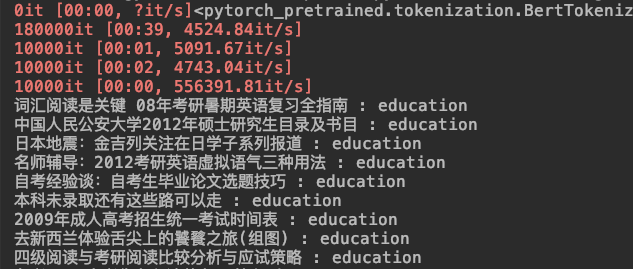
2）分类预测。

训练好模型后，我们就可以使用模型来进行文本分类预测。

首先我们使用bert模型预测，确保predict.py文件中的argparse参数中的—model参数值为“bert”。运行predict.py文件，模型开始启动预测，最终结果如下图所示。



然后，我们在使用ERNIE模型进行预测，将predict.py文件中的argparse参数中的—model参数由“bert”修改为“ENRIE”，运行predict.py文件，模型开始启动预测，最终结果如下图所示。



9.案例代码

【案例代码】utils.py

# coding: UTF-8

import torch

from tqdm import tqdm

import time

from datetime import timedelta

PAD, CLS = '[PAD]', '[CLS]' # padding符号, bert中综合信息符号

def build\_dataset(config):

def load\_dataset(path, pad\_size=32):

contents = []

with open(path, 'r', encoding='UTF-8') as f:

for line in tqdm(f):

lin = line.strip()

if not lin:

continue

content, label = lin.split('\t')#获取数据和标签

token = config.tokenizer.tokenize(content)#调用分词器进行文本分词

token = [CLS] + token#添加分类信息标志

seq\_len = len(token)

mask = []

token\_ids = config.tokenizer.convert\_tokens\_to\_ids(token)#转id

if pad\_size:

if len(token) < pad\_size:#句子长度小于pad\_size

#生成mask

mask = [1] \* len(token\_ids) + [0] \* (pad\_size - len(token))

#句子进行pad填充

token\_ids += ([0] \* (pad\_size - len(token)))

else:

mask = [1] \* pad\_size

token\_ids = token\_ids[:pad\_size]#截断

seq\_len = pad\_size

contents.append((token\_ids, int(label), seq\_len, mask))

return contents

train = load\_dataset(config.train\_path, config.pad\_size)

dev = load\_dataset(config.dev\_path, config.pad\_size)

test = load\_dataset(config.test\_path, config.pad\_size)

return train, dev, test

class DatasetIterater(object):

def \_\_init\_\_(self, batches, batch\_size, device):

"""

根据数据集构建一个迭代器，每次吐出一个批次的数据

:param batches: 数据集

:param batch\_size: 批次大小

:param device: 设备 CPU还是GPU

"""

self.batch\_size = batch\_size

self.batches = batches

#计算总批次

self.n\_batches = len(batches) // batch\_size

self.residue = False # 记录batch数量是否为整数

if len(batches) % self.n\_batches != 0:

self.residue = True

self.index = 0

self.device = device

def \_to\_tensor(self, datas):

# 为x和y定义Tensor

x = torch.LongTensor([\_[0] for \_ in datas]).to(self.device)

y = torch.LongTensor([\_[1] for \_ in datas]).to(self.device)

# pad前的长度(超过pad\_size的设为pad\_size)

seq\_len = torch.LongTensor([\_[2] for \_ in datas]).to(self.device)

mask = torch.LongTensor([\_[3] for \_ in datas]).to(self.device)

return (x, seq\_len, mask), y

def \_\_next\_\_(self):

# 判断批次数不为整数且正好是最后一个批次

if self.residue and self.index == self.n\_batches:

# 截取到最后一条数据

batches = self.batches[self.index \* self.batch\_size: len(self.batches)]

self.index += 1

batches = self.\_to\_tensor(batches)

return batches

# index超过了 表示一个轮次完了

elif self.index >= self.n\_batches:

self.index = 0

raise StopIteration

else:

# 截取一个批次长度数据

batches = self.batches[self.index \* self.batch\_size: (self.index + 1) \* self.batch\_size]

self.index += 1

batches = self.\_to\_tensor(batches)

return batches

def \_\_iter\_\_(self):

return self

def \_\_len\_\_(self):

if self.residue:

return self.n\_batches + 1

else:

return self.n\_batches

def build\_iterator(dataset, config):

iter = DatasetIterater(dataset, config.batch\_size, config.device)

return iter

def get\_time\_dif(start\_time):

"""获取已使用时间"""

end\_time = time.time()

time\_dif = end\_time - start\_time

return timedelta(seconds=int(round(time\_dif)))

【案例代码】models/bert.py

# coding: UTF-8

import torch

import torch.nn as nn

from pytorch\_pretrained import BertModel, BertTokenizer

class Config(object):

"""配置参数"""

def \_\_init\_\_(self, dataset):

self.model\_name = 'bert'#模型名称

self.train\_path = dataset + '/data/train.txt' # 训练集

self.dev\_path = dataset + '/data/dev.txt' # 验证集

self.test\_path = dataset + '/data/test.txt' # 测试集

self.class\_list = [x.strip() for x in open(

dataset + '/data/class.txt').readlines()] # 类别名单

self.save\_path = dataset + '/saved\_dict/' + self.model\_name + '.ckpt' # 模型训练结果

self.device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu') # 设备

self.require\_improvement = 1000 # 若超过1000batch效果还没提升，则提前结束训练

self.num\_classes = len(self.class\_list) # 类别数

self.num\_epochs = 3 # epoch数

self.batch\_size = 128 # mini-batch大小

self.pad\_size = 32 # 每句话处理成的长度(短填长切)

self.learning\_rate = 5e-5 # 学习率

self.bert\_path = './bert\_pretrain'#预训练模型的存储路径

self.tokenizer = BertTokenizer.from\_pretrained(self.bert\_path)#实例化分词器

self.hidden\_size = 768#预训练模型的隐藏层神经元数

class Model(nn.Module):

def \_\_init\_\_(self, config):

super(Model, self).\_\_init\_\_()

# 使用pytorch\_pretrained和预训练模型文件构建Bert层

self.bert = BertModel.from\_pretrained(config.bert\_path)

for param in self.bert.parameters():

param.requires\_grad = True

self.fc = nn.Linear(config.hidden\_size, config.num\_classes)

def forward(self, x):

context = x[0] # 输入的句子

mask = x[2] # 对padding部分进行mask，和句子一个size，padding部分用0表示，如：[1, 1, 1, 1, 0, 0]

\_, pooled = self.bert(context, attention\_mask=mask, output\_all\_encoded\_layers=False)

out = self.fc(pooled)

return out

【案例代码】models/ERNIE.py

# coding: UTF-8

import torch

import torch.nn as nn

from pytorch\_pretrained import BertModel, BertTokenizer

class Config(object):

"""配置参数"""

def \_\_init\_\_(self, dataset):

self.model\_name = 'ERNIE'

self.train\_path = dataset + '/data/train.txt' # 训练集

self.dev\_path = dataset + '/data/dev.txt' # 验证集

self.test\_path = dataset + '/data/test.txt' # 测试集

self.class\_list = [x.strip() for x in open(

dataset + '/data/class.txt').readlines()] # 类别名单

self.save\_path = dataset + '/saved\_dict/' + self.model\_name + '.ckpt' # 模型训练结果

self.device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu') # 设备

self.require\_improvement = 500 # 若超过500batch效果还没提升，则提前结束训练

self.num\_classes = len(self.class\_list) # 类别数

self.num\_epochs = 3 # epoch数

self.batch\_size = 128 # mini-batch大小

self.pad\_size = 32 # 每句话处理成的长度(短填长切)

self.learning\_rate = 5e-5 # 学习率

self.bert\_path = './ERNIE\_pretrain'#预训练模型的存储路径

self.tokenizer = BertTokenizer.from\_pretrained(self.bert\_path)#实例化分词器

print(self.tokenizer)#预训练模型的隐藏层神经元数

self.hidden\_size = 768#预训练模型的隐藏层神经元数

class Model(nn.Module):

def \_\_init\_\_(self, config):

super(Model, self).\_\_init\_\_()

#使用pytorch\_pretrained和预训练模型文件构建Bert层

self.bert = BertModel.from\_pretrained(config.bert\_path)

for param in self.bert.parameters():#设置bert模型中的参数求导，这样模型才能微调更新

param.requires\_grad = True

#定义全连接层 进行分类

self.fc = nn.Linear(config.hidden\_size, config.num\_classes)

def forward(self, x):

context = x[0] # 输入的句子

mask = x[2] # 对padding部分进行mask，和句子一个size，padding部分用0表示，如：[1, 1, 1, 1, 0, 0]

#bert计算出句向量

\_, pooled = self.bert(context, attention\_mask=mask, output\_all\_encoded\_layers=False)

#使用全连接层对句向量进行映射分类

out = self.fc(pooled)

return out

【案例代码】train\_eval.py

# coding: UTF-8

import numpy as np

import torch

import torch.nn as nn

import torch.nn.functional as F

from sklearn import metrics

import time

from utils import get\_time\_dif

from pytorch\_pretrained.optimization import BertAdam

# 权重初始化，默认xavier

def init\_network(model, method='xavier', exclude='embedding', seed=123):

for name, w in model.named\_parameters():

if exclude not in name:

if len(w.size()) < 2:

continue

if 'weight' in name:

if method == 'xavier':

nn.init.xavier\_normal\_(w)

elif method == 'kaiming':

nn.init.kaiming\_normal\_(w)

else:

nn.init.normal\_(w)

elif 'bias' in name:

nn.init.constant\_(w, 0)

else:

pass

def train(config, model, train\_iter, dev\_iter, test\_iter):

start\_time = time.time()

model.train()#设置为训练模式

param\_optimizer = list(model.named\_parameters())#提取模型所有参数

no\_decay = ['bias', 'LayerNorm.bias', 'LayerNorm.weight']#不需要训练的参数

#过滤掉不需要训练的参数

optimizer\_grouped\_parameters = [

{'params': [p for n, p in param\_optimizer if not any(nd in n for nd in no\_decay)], 'weight\_decay': 0.01},

{'params': [p for n, p in param\_optimizer if any(nd in n for nd in no\_decay)], 'weight\_decay': 0.0}]

#指定BertAdam优化器，对需要训练的参数进行训练更新

optimizer = BertAdam(optimizer\_grouped\_parameters,

lr=config.learning\_rate,

warmup=0.05,

t\_total=len(train\_iter) \* config.num\_epochs)

total\_batch = 0 # 记录进行到多少batch

dev\_best\_loss = float('inf')

last\_improve = 0 # 记录上次验证集loss下降的batch数

flag = False # 记录是否很久没有效果提升

# model.train()

for epoch in range(config.num\_epochs):

print('Epoch [{}/{}]'.format(epoch + 1, config.num\_epochs))

for i, (trains, labels) in enumerate(train\_iter):

outputs = model(trains)#计算输出值

model.zero\_grad()#清零导数

loss = F.cross\_entropy(outputs, labels)#计算损失

loss.backward()#反向传播求导

optimizer.step()#更新参数

if total\_batch % 10 == 0:

# 每多少轮输出在训练集和验证集上的效果

true = labels.data.cpu()#提取真实值

predic = torch.max(outputs.data, 1)[1].cpu()#提取预测值

train\_acc = metrics.accuracy\_score(true, predic)#计算训练集准确率

#计算验证集准确率和损失

dev\_acc, dev\_loss = evaluate(config, model, dev\_iter)

if dev\_loss < dev\_best\_loss:#如果本轮loss小于历史最小loss

dev\_best\_loss = dev\_loss

#保存模型

torch.save(model.state\_dict(), config.save\_path)

improve = '\*'

last\_improve = total\_batch

else:

improve = ''

time\_dif = get\_time\_dif(start\_time)

msg = 'Iter: {0:>6}, Train Loss: {1:>5.2}, Train Acc: {2:>6.2%}, Val Loss: {3:>5.2}, Val Acc: {4:>6.2%}, Time: {5} {6}'

print(msg.format(total\_batch, loss.item(), train\_acc, dev\_loss, dev\_acc, time\_dif, improve))

model.train()

total\_batch += 1

if total\_batch - last\_improve > config.require\_improvement:

# 验证集loss超过1000batch没下降，结束训练

print("No optimization for a long time, auto-stopping...")

flag = True

break

if flag:

break

# 最后在测试集上评估

test(config, model, test\_iter)

def evaluate(config, model, data\_iter, test=False):

model.eval()

loss\_total = 0

predict\_all = np.array([], dtype=int)

labels\_all = np.array([], dtype=int)

with torch.no\_grad():#不计算梯度

for texts, labels in data\_iter:#遍历数据集

outputs = model(texts)#计算输出值

loss = F.cross\_entropy(outputs, labels)#计算损失

loss\_total += loss

labels = labels.data.cpu().numpy()

predic = torch.max(outputs.data, 1)[1].cpu().numpy()#计算预测值

labels\_all = np.append(labels\_all, labels)

predict\_all = np.append(predict\_all, predic)

#计算准确率

acc = metrics.accuracy\_score(labels\_all, predict\_all)

if test:

#输出分类报告

report = metrics.classification\_report(labels\_all,

predict\_all,

target\_names=config.class\_list,

digits=4)

#输出混淆矩阵

confusion = metrics.confusion\_matrix(labels\_all, predict\_all)

return acc, loss\_total / len(data\_iter), report, confusion

return acc, loss\_total / len(data\_iter)

def test(config, model, test\_iter):

# test

# 加载模型

model.load\_state\_dict(torch.load(config.save\_path, ,map\_location=torch.device(config.device)))

# 指定模型为验证模式

model.eval()

start\_time = time.time()

# 测试集评估

test\_acc, test\_loss, test\_report, test\_confusion = evaluate(config, model, test\_iter, test=True)

msg = 'Test Loss: {0:>5.2}, Test Acc: {1:>6.2%}'

print(msg.format(test\_loss, test\_acc))

print("Precision, Recall and F1-Score...")

print(test\_report)

print("Confusion Matrix...")

print(test\_confusion)

time\_dif = get\_time\_dif(start\_time)

print("Time usage:", time\_dif)

def predict(config,model,data\_iter):

#加载模型

model.load\_state\_dict(torch.load(config.save\_path, ,map\_location=torch.device(config.device)))

#验证模式

model.eval()

predict\_all = np.array([], dtype=int)

with torch.no\_grad():#不计算梯度

for texts, labels in data\_iter:#遍历数据

outputs = model(texts)#计算输出值

predic = torch.max(outputs.data, 1)[1].cpu().numpy()#计算预测类别下标

predict\_all = np.append(predict\_all, predic)

results=[config.class\_list[i] for i in predict\_all]#根据类别下标返回对应的类别标签

return results

【案例代码】run.py

# coding: UTF-8

import time

import torch

import numpy as np

from train\_eval import train, init\_network

from importlib import import\_module

import argparse

from utils import build\_dataset, build\_iterator, get\_time\_dif

parser = argparse.ArgumentParser(description='Chinese Text Classification')

parser.add\_argument('--model', type=str, default='bert',help='choose a model: bert, ERNIE')

args = parser.parse\_args()

if \_\_name\_\_ == '\_\_main\_\_':

dataset = 'news' # 指定数据集文件目录

model\_name = args.model # bert

x = import\_module('models.' + model\_name)#导入模型

config = x.Config(dataset)#生成配置文件

np.random.seed(1)

torch.manual\_seed(1)

torch.cuda.manual\_seed\_all(1)

torch.backends.cudnn.determiniustic = True # 保证每次结果一样

start\_time = time.time()

# 加载数据集 训练集、验证集、测试集

print("Loading data...")

train\_data, dev\_data, test\_data = build\_dataset(config)

# 构建训练集、验证集、测试集迭代器

train\_iter = build\_iterator(train\_data, config)

dev\_iter = build\_iterator(dev\_data, config)

test\_iter = build\_iterator(test\_data, config)

time\_dif = get\_time\_dif(start\_time)

print("Time usage:", time\_dif)

#利用配置参数实例化模型

model = x.Model(config).to(config.device)

#模型训练

train(config, model, train\_iter, dev\_iter, test\_iter)

【案例代码】predict.py

# coding: UTF-8

from train\_eval import predict

from importlib import import\_module

import argparse

from utils import build\_dataset, build\_iterator

from tqdm import tqdm

parser = argparse.ArgumentParser(description='Chinese Text Classification')

parser.add\_argument('--model', type=str, default='bert',help='choose a model: bert, ERNIE')

args = parser.parse\_args()

if \_\_name\_\_ == '\_\_main\_\_':

dataset = 'news' # 指定数据集

model\_name = args.model #指定模型

x = import\_module('models.' + model\_name)#加载模型

config = x.Config(dataset)#生成配置

\_, \_, test\_data = build\_dataset(config)#读取数据集

test\_iter = build\_iterator(test\_data[:200], config)#构建迭代器

texts=[]#读取原始文本

with open(config.test\_path, 'r', encoding='UTF-8') as f:

for line in tqdm(f):

lin = line.strip()

if not lin:

continue

content, label = lin.split('\t')

texts.append(content)

#实例化模型对象

model = x.Model(config).to(config.device)

#调用predict函数实现模型加载和预测

preds=predict(config,model,test\_iter)

#展示预测结果

for text,label in zip(texts[:200],preds):

print(text,':',label)