# Pytorch神经网络气温预测

环境配置:

*<https://pypi.tuna.tsinghua.edu.cn/simple>*

Python3.6 对应的Pandas、Numpy的版本

pip install matplotlib==3.3.4

pip install pyparsing==3.1.2

pip install pandas==0.25.3

pip install numpy==1.19.5

pip install scikit-learn==0.24.2

pip install torch==1.3.0+cu92 -f <https://download.pytorch.org/whl/torch_stable.html>

补充:

1.python基本操作:

1).压包和解包

zip,压包操作,用于将可迭代的对象(列表、元组、集合)作为参数,将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表(Python3 返回的是迭代器对象)

\*、\*\*解包操作,\*args、\*\*kwargs通常作为赋值变量、参变量;\*args可将迭代对象(字符串、元组、列表、集合)解包为单独的值,并分配给相应的列表变量arg; \*\*kwargs可将字典解包为关键字参数和对应的值,并分配给相应的字典变量kwargs;

# 解包  
ls = 1, 2, 3, 4, 5, 6  
print(type(ls)) # <class 'tuple'>  
  
a, \*b, c = 1, 2, 3, 4, 5  
print('a=>', a, 'b=>', b, 'c=>', c)  
  
  
def Print(\*arg, \*\*kwargs):  
 print(arg)  
 print(kwargs)  
  
  
Print(1, 2, 3, 4, key=10, value=20)  
  
# 压包  
years = [2022, 2023, 2024]  
months = [10, 11, 12]  
days = [1, 2, 3]  
for year, month, day in zip(years, months, days):  
 print('year=>', year, 'month=>', month, 'day=>', day)

2).迭代器对象

A.要成为一个可迭代对象,需要实现\_\_iter\_\_()、\_\_next\_\_()方法；

B.\_\_next()\_\_方法,返回下一个数据,若没有数据了,则需要抛出StopIteration异常;

C.迭代器对象支持python内置函数next()取值,支持for循环中的in关键字(in+iter(迭代器对象))

# 1.迭代器使用  
ls = [1, 2, 3, 4, 5, 6]  
ls = iter(ls)  
print(next(ls))  
print(next(ls))  
  
  
# 2.自定义迭代器  
class MyIterator:  
 def \_\_init\_\_(self, count):  
 self.tmp = 0  
 self.count = count  
  
 def \_\_iter\_\_(self):  
 return self  
  
 def \_\_next\_\_(self):  
 if self.tmp < self.count:  
 self.tmp += 1  
 else:  
 raise StopIteration  
 return self.tmp  
  
  
my\_iter = MyIterator(10)  
my\_iterator = iter(my\_iter)  
print(next(my\_iterator))  
print(next(my\_iterator))  
  
for item in MyIterator(10):  
 print(item)

3).datetime/time模块

A.datetime模块提供各种用于操作日期和时间的类,该模块侧重于高效率的格式化输出。

datetime.fromtimestamp(time\_stamp):根据时间戳,生成datetime对象

datetime.strptime('2001.6.21', '%Y.%m.%d'):根据格式化字符串,生成datetime对象

datetime.timedelta(days=0,seconds=int(round(time\_dif))):根据时间戳,返回timedelta对象

import time  
from datetime import datetime  
  
time\_stamp = time.time()  
date = datetime.fromtimestamp(time\_stamp)  
print(date)  
print(type(date))  
print(date.year)  
print(date.month)  
print(date.day)  
  
date = datetime.strptime('2001.6.21', '%Y.%m.%d')  
print(date)  
print(type(date))  
print(date.year)  
print(date.month)  
print(date.day)

B.time模块:

time():返回当前时间的时间戳;

sleep():推迟调用进程运行;

strftime():用于格式化时间,返回以可读字符串表示的当地时间;

localtime():格式化时间戳为本地的时间

Eg:日志的保存形式

1).time模块

writer = SummaryWriter(log\_dir=config.log\_path + '/' + time.strftime('%m-%d\_%H.%M', time.localtime()))

2).datetime模块

import datetime  
  
now = datetime.datetime.now()  
print(now)  
strtime = now.strftime('%Y-%m-%d')  
print(strtime)

2.numpy基本操作:

1).ndarray对象:

numpy.array(object,dtype=None,copy=True,order=None,subok=False,ndmin=0)

2).ndarray切片和索引

ndarray对象的内容可以通过索引或切片来访问和修改,切片遵循左闭右开原则;

3).dataframe、tensor转化成ndarray

dataframe转化成ndarray通过.to\_numpy()、np.array()方法

tensor转化成ndarray通过.numpy()、np.array()方法

import numpy  
import numpy as np  
import pandas as pd  
import torch  
  
# 1.ndarray对象  
features = np.array([1, 2, 3], dtype=numpy.float, ndmin=3)  
print(features)  
print(type(features)) # <class 'numpy.ndarray'>  
  
# 2.ndarray切片索引  
features = np.array([[[1, 2, 3], [4, 5, 6]],  
 [[4, 5, 6], [7, 8, 9]],  
 [[7, 8, 9], [1, 2, 3]]], dtype=numpy.float)  
print(features)  
print(features.size) # 18  
print(features.shape) # (3, 2, 3)  
print(type(features)) # <class 'numpy.ndarray'>  
  
print(features[2, 1, :]) # [1. 2. 3.]  
print(features[1, :, :]) # [[4. 5. 6.],[7. 8. 9.]]  
print(features[1, :, 1]) # [5. 8.]  
  
# 3.dataframe、tensor转化成ndarray  
pd\_array = pd.DataFrame(data=[[1, 2, 3], [4, 5, 6], [7, 8, 9]], columns=['idx1', 'idx2', 'idx3'])  
np\_array\_1 = pd\_array.to\_numpy()  
print(np\_array\_1)  
np\_array\_2 = np.array(pd\_array)  
print(np\_array\_2)  
  
tensor = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=torch.float)  
np\_array\_3 = tensor.data.numpy()  
print(np\_array\_3)  
np\_array\_4 = np.array(tensor.data)  
print(np\_array\_4)

3.pandas基本操作

Pandas 可以从各种文件格式比如 CSV、JSON、SQL、Microsoft Excel 导入数据,并返回DataFrame对象。

1).DataFrame对象

DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)

data:DataFrame 的数据部分，可以是字典、二维数组、Series、DataFrame 或其他可转换为 DataFrame 的对象。如果不提供此参数，则创建一个空的 DataFrame。

index:DataFrame 的行索引,用于标识每行数据。可以是列表、数组、索引对象等。如果不提供此参数，则创建一个默认的整数索引。

columns:DataFrame的列索引,用于标识每列数据。可以是列表、数组、索引对象等。如果不提供此参数，则创建一个默认的整数索引。

2).Series对象

Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)

data:Series的数据部分,可以是列表、数组、字典、标量值等。如果不提供此参数,则创建一个空的 Series。

Index:Series的索引部分,用于对数据进行标记。可以是(字符串/整数)列表等。如果不提供此参数,则创建一个默认的整数索引。

3).常见方法:

A.get\_dummies()

将离散型特征的每一种取值都看成一种状态，若你的这一特征中有N个不相同的取值，那么我们就可以将该特征抽象成N种不同的状态，one-hot编码保证了每一个取值只会使得一种状态处于“激活态”，也就是说这N种状态中只有一个状态位值为1，其他状态位都是0。

B.drop(labels=None,axis=0,inplace=False)

labels:字符串格式,指明要删除的行索引名、列索引名;

axis:删除行,axis=0;删除列,axis=1;

inplace:默认情况为False,表示保持原来的数据不变;True,则表示在原来的数据上改变；

import pandas as pd  
  
features = pd.read\_csv('./temps.csv')  
features = pd.get\_dummies(features)  
print(features)  
  
features = features.drop(labels='actual', axis=1)  
print(features)

4.tensor基本操作

1).常见属性

.T:表示对tensor张量进行转置

.data:表示生成无梯度的纯张量

取消张量和计算图的关联,从而使得对张量进行的操作不会影响到梯度计算;.data属性与.detach()方法效果相同,.detach()方法线程更安全(require\_grad=True, 意为张量纳入计算图中,梯度反传时对该 tensor 计算梯度,并存入 tensor.grad 中)

weight1.data.add\_(-weight1.grad.data \* learning\_rate)

bias1.data.add\_(-bias1.grad.data \* learning\_rate)

weight2.data.add\_(-weight2.grad.data \* learning\_rate)

bias2.data.add\_(-bias2.grad.data \* learning\_rate)

.shape:表示tensor张量的维度(N,C,H,W)

.shape属性与.size()方法效果相同,.size()方法线程更安全

2).常见方法

torch.addmm(input,mat1,mat2,beta=1,alpha=1,out=None)

out= (同Tensor.addmm\_(mat1,mat2,beta=1,alpha=1))

torch.add(input,other,alpha=1,out=None)

out= (同Tensor.add\_(other,alpha=1))

torch.rand(size,dtype=None,device=None,requires\_grad=False)

根据标准[0,1]上的均匀分布,随机初始化张量

torch.randn(size,dtype=None,device=None,requires\_grad=False)

根据标准正态分布,随机初始化张量

3).ndarray、dataframe转化成tensor

ndarray转化成tensor通过torch.tensor(ndarray)

dataframe转化成tensor通过torch.tensor(dataframe)

5.plt基本操作

1).plot()

plt.[plot](https://so.csdn.net/so/search?q=plot&spm=1001.2101.3001.7020" \t "_blank)(\*args, scalex=True, scaley=True, data=None, \*\*kwargs) 用于绘制线图和散点图

\*args:x轴数据、y轴数据,数据可以是列表(list)或数组(ndarray)和fmt线条颜色或样式;

\*\*kwargs:设置指定属性,如线的标签(label搭配plt.legend()一起使用)、线的宽度(linewidth)等等

2).show()

plt.show()用于在新窗口中显示图像

3).xlabel()、ylabel()、title()

xlabel()、ylabel():设置x、y轴标签;

title():设置图像标题

plt.rcParams['font.sans-serif'] = ['SimHei'],解决中文乱码问题。

4).xtick()、ytick()、tight\_layout()

xtick()、ytick():通过rotation指定x、y轴各个单位刻度旋转角度;

tight\_layout():图的自动布局,解决轴标签、标题超出图形区域和不同轴域标签折叠在一起

5)subplots()、subplot()

subplots(nrows=1, ncols=1, sharex=False, sharey=False):用于在一张图中展示多个子图;

nrows:子图行数;

ncols:子图列数;

sharex: 子图是否共享x轴;

sharey:子图是否共享y轴;

subplot(numRows, numCols, plotNum):用于指定绘制子图的区域

numRows:绘图区域所分行数;

numCols:绘图区域所分列数;

plotNum:指定子图所在区域

fig.autofmt\_xdate(rotation=45) 不仅适用于 date 对象,而且适用于普通标签文本字符串,指定x轴各个单位刻度旋转角度,并且sharex=True;

import matplotlib.pyplot as plt  
  
# 1.plt绘制线图  
plt.rcParams['font.sans-serif'] = ['SimHei']  
x = [1, 2, 3, 4]  
y = [2, 4, 6, 8]  
plt.plot(x, y, 'r-', label='dot')  
plt.legend()  
plt.xlabel('axis-x轴')  
plt.ylabel('axis-y轴')  
plt.title('y=2x')  
plt.xticks(rotation=60)  
plt.tight\_layout()  
plt.show()  
  
# 2.plt subplots()绘制子图  
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True)  
# fig.autofmt\_xdate(rotation=45)  
ax1.plot([1, 2, 3], [1, 2, 3], 'b--')  
ax1.set\_xlabel('x轴')  
ax1.set\_ylabel('y轴')  
ax1.set\_title('subplot\_1')  
ax2.plot([1, 2, 3], [2, 4, 6], 'c-')  
ax2.set\_xlabel('x轴')  
ax2.set\_ylabel('y轴')  
ax2.set\_title('subplot\_2')  
ax3.plot([1, 2, 3], [1, 2, 3], 'r-.')  
ax3.set\_xlabel('x轴')  
ax3.set\_ylabel('y轴')  
ax3.set\_title('subplot\_3')  
ax4.plot([1, 2, 3], [2, 4, 6], 'b--')  
ax4.set\_xlabel('x轴')  
ax4.set\_ylabel('y轴')  
ax4.set\_title('subplot\_4')  
plt.tight\_layout()  
plt.show()  
  
# 3.plt subplot()绘制子图  
plt.subplot(2, 2, 1)  
plt.plot([1, 2, 3], [1, 2, 3], 'b--')  
plt.xlabel('x轴')  
plt.ylabel('y轴')  
plt.title('subplot\_1')  
plt.subplot(2, 2, 2)  
plt.plot([1, 2, 3], [2, 4, 6], 'ro')  
plt.xlabel('x轴')  
plt.ylabel('y轴')  
plt.title('subplot\_2')  
plt.subplot(2, 1, 2)  
plt.plot([1, 2, 3], [1, 2, 3], 'b--')  
plt.xlabel('x轴')  
plt.ylabel('y轴')  
plt.title('subplot\_3')  
plt.show()

6.经验总结

1).损失函数无法收敛的原因:1.学习率设置得过大,导致了损失函数loss振荡,进而导致模型无法收敛;2.数据分布较为复杂,没有进行归一化处理,导致每次迭代模型都往不同的方向上优化,

进而需要较多的迭代次数;3.训练过程中出现了梯度消失、梯度爆炸

2).batch\_size越大越能体现GPU的计算性能;batch\_size一般取16，32，64，128，256（2的整数次幂）等

3).随机梯度下降法(SGD),迭代速度较快,但并不是每次迭代都向着整体最优化方向(相同迭代次数下不稳定且准确性下降)

4).模型训练基本步骤:A.数据处理 B.定义模型 C.模型训练

数据处理注意事项:

1).提取特征、标签,特征记得进行归一化处理,归一化处理的作用:1.数量级的差异将导致数量级较大的属性占据主导地位,依赖于样本距离的算法对于数据的数量级非常的敏感;2.数量级的差异将导致迭代收敛速度减慢

2).DataSet类中,\_\_init()\_\_需要得到tensor格式特征、标签,\_\_getitem()\_\_需要返回tensor格式特征、标签的第item个数据

定义模型注意事项:

1).继承nn.Module类,类中\_\_init\_\_()方法中记得super().\_\_init\_\_()

模型训练注意事项:

1). for (trains/imgs/texts/xx),(labels/yy) in dataloader:

2). 使用cuda时,记得(data/labels)要.to(device)

# LSTM文本分类实战

环境配置:

pip install numpy==1.19.5

pip install tqdm==4.64.1

pip install scikit-learn==0.24.2

pip install tensorboard==2.10.1

pip install torch==1.3.0+cu92 -f <https://download.pytorch.org/whl/torch_stable.html>

重点:

1.embedding层

将词汇表中的词或短语映射为固定长度向量的技术,通过词嵌入(降维算法)可以将One-Hot编码表示的高维稀疏矩阵转为低维且连续的向量;语义相近的词语,词语对应的向量位置也更接近。词嵌入向量,不仅可以表达语义的相似性,还可以通过向量的数学关系描述词语之间的语义关联。

词嵌入算法——word2vec、fasttext、glove等,可以通过特定的词嵌入算法,训练一个通用的嵌入矩阵;嵌入矩阵的行,是语料库中词语的个数,嵌入矩阵的列,是表示词语的维度;将OneHot矩阵和嵌入矩阵相乘可以得到句子的嵌入向量(嵌入矩阵一般作为神经网络中的一层,并通过训练得到;FastText进行文本分类的同时也会产生词的embedding,一般不使用pre\_trained的embedding)

预训练pre\_trained和微调fine\_turning:就是指预先训练的一个模型或者指预先训练模型的过程;微调 就是指将预训练过的模型(可以是裁剪后的模型)作用于自己的数据集，并使参数适应自己数据集的过程。预训练可以加快模型训练、加速模型收敛;可以使知识从一个任务转移到另外一个任务。

1).随机Embedding

class torch.nn.Embedding(num\_embeddings, embedding\_dim, padding\_idx=None)

num\_embeddings:语料表长度;

embedding\_dim:每个词/字所需的维度(自定义);

padding\_idx=None:用于填充的参数索引,训练过程不进行参数更新;

2).预训练Embedding

classmethod  from\_pretrained(embeddings, freeze=True, padding\_idx=None)

embeddings:预训练的embedding;

freeze:布尔值,True: embedding.weight.requires\_grad = False(默认);

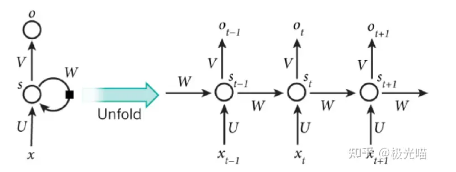
padding\_idx:用于填充的参数索引

import torch  
from torch import nn  
  
print('==========random===========')  
embedding = nn.Embedding(10, 5)  
print(embedding.weight)  
x = torch.LongTensor([0, 1, 2])  
x = embedding(x)  
print(x)  
  
embedding = nn.Embedding(10, 5, padding\_idx=9)  
print(embedding.weight)  
x = torch.LongTensor([0, 1, 2])  
x = embedding(x)  
print(x)  
  
print('==========pre\_trained===========')  
pretrained\_embedding = torch.randn((10, 5))  
print(pretrained\_embedding)  
embedding = nn.Embedding.from\_pretrained(embeddings=pretrained\_embedding, padding\_idx=9)  
print(embedding.weight)  
x = torch.LongTensor([1, 3, 2, 9, 9])  
x = embedding(x)  
print(x)

2.LSTM模型(RNN模型的变形)

文本分类模型:RNN(较多,适合短序列预测,不适合长序列预测),CNN

基于门控的循环神经网络,为了改善循环神经网络的长程依赖问题,引入门控机制(输入门、遗忘门、输出门)来控制信息的累积速度,包括有选择地加入新的信息,并有选择地遗忘之前累积的信息。



class torch.nn.LSTM(input\_size, hidden\_size, num\_layers=1, bias=True, batch\_first=False, dropout=0.0, bidirectional=False,device=None)

input\_size: 每个词/字所需的维度(自定义,某一时刻Xt的神经网络的输入);

hidden\_size:隐藏层神经元的维度;(输出层神经元的维度==隐藏层神经元的维度)

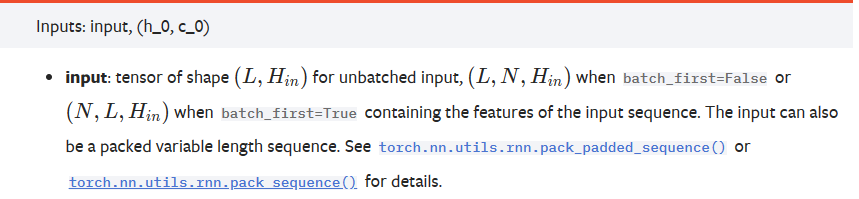
num\_layers:RNN的层数;

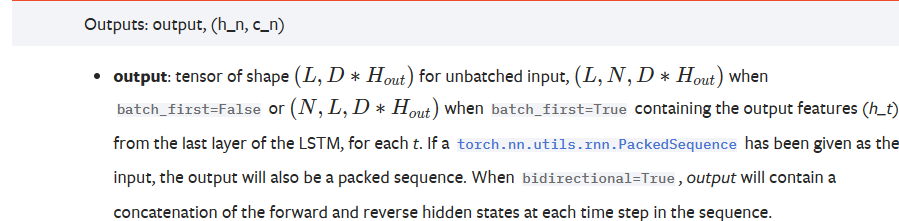
batch\_first:batch\_size是否为第一维度,默认输入数据的input\_shape=(seq\_len,batch\_size

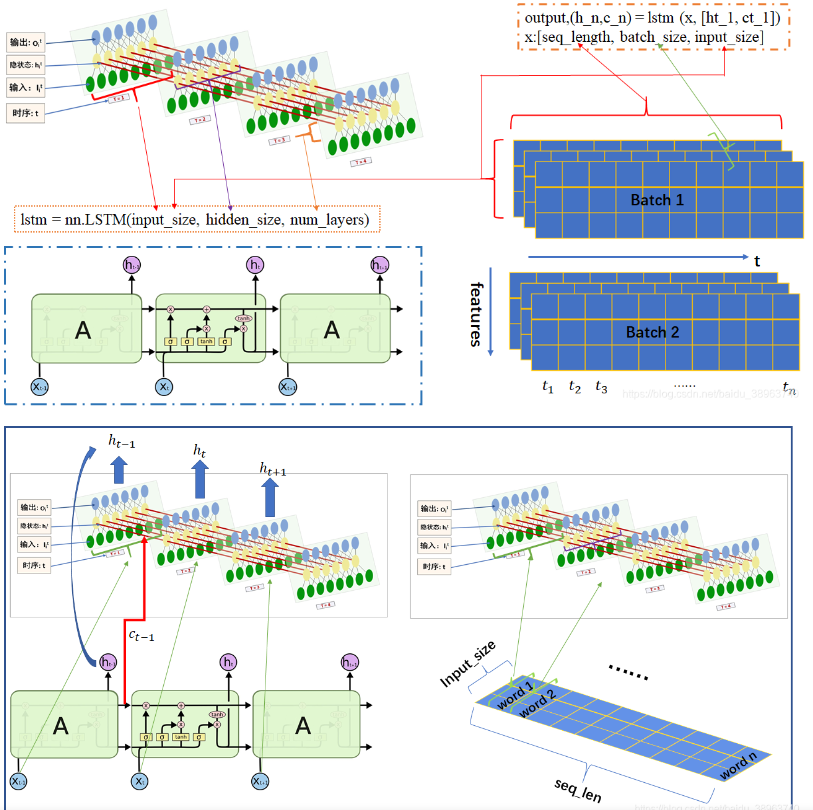
,embedding\_dim);

dropout:防止网络过拟合;

bidirectional:布尔值(默认为False),是否启用双向LSTM;

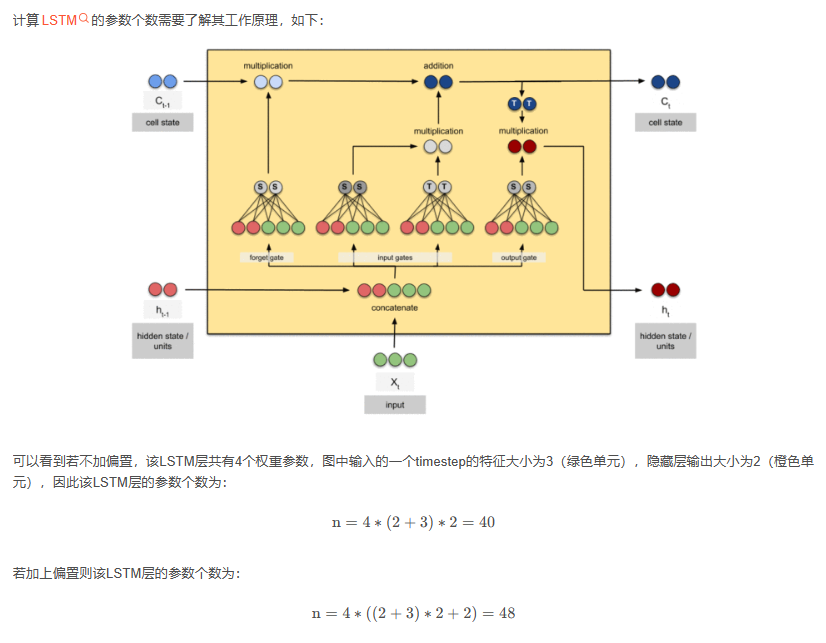






import torch  
from torch import nn  
  
print('=======batch\_first=False=======')  
# output 维度 batch\_size seq\_len output\_size  
rnn = nn.LSTM(3, 5, 2, batch\_first=True)  
x = torch.randn(3, 5, 3)  
output, (h\_n, c\_n) = rnn(x)  
print(output)  
print(output[:, -1, :])  
print(h\_n)  
  
print('=======batch\_first=True=======')  
# output 维度 seq\_len batch\_size output\_size  
rnn = nn.LSTM(3, 5, 2)  
x = torch.randn(3, 5, 3)  
output, (h\_n, c\_n) = rnn(x)  
print(output)  
print(output[-1, :, :]) # 第一维度为具体值时,会降维  
print(h\_n)

LSTM的参数个数:



补充:deepthought lstm源码解读

补充:

1.python基本操作

1).argparse模块

A.创建一个解析器

parser=argparse.ArgumentParser()

ArgumentParser对象,包含将命令行解析成python基本数据类型所需的全部信息;

B.加入参数或选项

parser.add\_argument()

name or flags -命名或者选项字符串的列表,例如 model或 -m, --model;

action -当参数在命令行中出现时使用的动作基本类型,例如store\_true, store\_false, store\_const(需要指定const的值);

default -当参数未在命令行中出现时使用的值;

required -此命令行选项是否可省略(仅选项可用);

type -命令行参数应当被转换成的类型(str/bool);

help -一个此选项作用的简单描述;

dest -添加到parse\_args() 所返回对象上的属性名(默认为选项字符串)

import argparse  
  
parser = argparse.ArgumentParser()  
parser.add\_argument('--model', required=True, help='choose a model:TextCNN、TextRNN')  
parser.add\_argument('--embedding', default='pre\_trained', help='choose random or pre\_trained')  
parser.add\_argument('--vocab', default=True, help='use vocab or not')  
args = parser.parse\_args()  
print(args.model)// TextRNN  
print(args.embedding)// pre\_trained  
print(args.vocab)// True

2).importlib模块

import\_module(name:str,package:str) 根据需要动态导入相应配置文件、库文件等依赖;

name:绝对路径/相对路径(相对于package)

package:可选选项,文件包名

x = import\_module(model\_name)//根据模型名称导入不同模块

3).os模块

.getcwd() 获取当前工作路径

.listdir(path) 列举指定目录中的文件名和目录名

.mkdir(path) 创建单层目录

.remove(path) 删除文件

.path.abspath(path) 返回绝对路径

.path.exists(path) 返回布尔值,路径path是否存在

os.path.join(path1[, path2[, ...]]) 把目录和文件名合成一个路径

import os  
  
path = 'THUCNews/'  
dir\_list = os.listdir(path)  
print(dir\_list) // ['data', 'log', 'saved\_dict']  
if not os.path.exists(path + 'model'):  
 os.mkdir(path + 'model')  
os.rmdir(path + 'model')  
  
relative\_path = os.path.join(path, 'data/class.txt')  
with open(relative\_path, 'r') as f:  
 class\_list = f.readlines()  
print(class\_list) // ['finance\n', 'realty\n', 'stocks\n', 'education\n', 'science\n', 'society\n', 'politics\n', 'sports\n', 'game\n', 'entertainment']  
print(os.getcwd()) //E:\py\_conda\nlp  
print(os.path.abspath('model')) //E:\py\_conda\nlp\model

补充:

1. os.path.join与os.path.abspath一般一起使用:

import os  
  
file = os.path.join(os.path.dirname(\_\_file\_\_), "../release/Base.so")  
print(file)// E:\py\_conda\openke\openke\data\../release/Base.so  
base\_file = os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), "../release/Base.so"))  
print(base\_file)// E:\py\_conda\openke\openke\release\Base.so

4).pickle模块

pickle模块实现了用于序列化和反序列化Python对象结构的二进制协议

import os.path  
import pickle  
  
user\_dict = {'name': '林泽源', 'age': 18}  
if not os.path.exists('user\_dict.pkl'):  
 pickle.dump(user\_dict, open('user\_dict.pkl', 'wb'))  
  
user\_dict = pickle.load(open('user\_dict.pkl', 'rb'))  
print(user\_dict) //{'name': '林泽源', 'age': 18}

5).tqdm模块

tqdm是 [Python](https://so.csdn.net/so/search?q=Python&spm=1001.2101.3001.7020" \t "_blank) 进度条库,可以在 Python长循环中添加一个进度提示信息。

.tqdm(iterable:Iterable)加载一个可迭代对象,并以进度条的形式实时显示该可迭代对象的加载进度。

iterable:可迭代对象list、str、tuple、dict、file

.tqdm按行读取文件

import time  
from tqdm import tqdm  
  
with open('./THUCNews/data/test.txt', encoding='utf-8') as f:  
 for line in tqdm(f):  
 print(type(line))  
 print(line)  
 time.sleep(2)

6).关键字

is 关键字 判断两个变量的地址是否相同(==用于数值比较)

not 关键字的返回值是一个布尔值(bool),该关键字会对参数进行取反操作。

None关键字用于定义 null 值,或根本没有值。

print(not False)// True  
  
temp = None  
print(id(None))// 1410100240  
print(id(temp))// 1410100240  
print(temp is None)// True  
print(temp is not None)// False

6).内置函数

A.sorted() 函数,对所有可迭代的对象进行排序操作

sorted(iterable, cmp=None, key=None, reverse=False)

iterable:可迭代对象;

cmp:比较的函数,这个具有两个参数,参数的值都是从可迭代对象中取出,此函数必须遵守的规则为,大于则返回1,小于则返回-1,等于则返回0;

key:主要是用来进行比较的元素,只有一个参数,具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序;

reverse:排序规则,reverse = True 降序,reverse = False升序(默认);

(cmp和reverse一般不同时使用)

dic1 = {'height': 178, 'weight': 140, 'age': 18}  
print(dic1)  
print(dic1.items())  
print(dic1.keys())  
print(dic1.values())  
dic1 = sorted(dic1.items(), key=lambda x: x[1], reverse=False)  
print(dic1)  
  
dic2 = dict([('height', 178), ('weight', 140), ('age', 18)])  
print(dic2)  
print(dic2.items())  
print(dic2.keys())  
print(dic2.values())  
print(dic2)  
dic2 = sorted(dic2.items(), key=lambda x: x[1], reverse=True)  
print(dic2)

B. enumerate() 函数,用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中。

enumerate(sequence, [start=0]) sequence:迭代器对象;start:下标起始位置的值;

C. 基本运算/ 、//

/:常规除,取浮点数部分;

//:向下取整除,只取整数部分;

2.numpy基本操作

1).numpy的文件存储:

.save():数组是以未压缩的原始二进制格式保存在扩展名为.[npy文件](https://so.csdn.net/so/search?q=npy%E6%96%87%E4%BB%B6&spm=1001.2101.3001.7020" \t "_blank)中;

.savez():多组数组是以压缩的原始二进制格式保存在扩展名为.[npz文件](https://so.csdn.net/so/search?q=npy%E6%96%87%E4%BB%B6&spm=1001.2101.3001.7020" \t "_blank)中;

.load():读取.npz格式文件,返回NpzFile类型(def \_\_getitem\_\_类似字典结构);读取.npy格式文件,返回numpy.ndarray类型

import numpy as np  
  
  
class User(object):  
 def \_\_init\_\_(self, name: str, age: int):  
 self.name = name  
 self.age = age  
  
 def \_\_str\_\_(self):  
 return "%s的年龄为%d" % (self.name, self.age)  
  
  
user1 = User('林泽源', 18)  
print(user1) //林泽源的年龄为18  
dic = {'filename': 'user', 'data': user1}  
print(dic)// {'filename': 'user', 'data': <\_\_main\_\_.User object at 0x0000021990B5CB70>}  
  
np.save('dict', dic)  
dict\_np = np.load('dict.npy', allow\_pickle=True)  
print(type(dict\_np))// <class 'numpy.ndarray'>  
dict\_np = dict\_np.item()  
print(dict\_np['data'])// 林泽源的年龄为18  
  
array1 = np.arange(3)  
array2 = np.arange(4)  
array3 = np.arange(5)  
np.savez('arr', arr1=array1, arr2=array2, arr3=array3)  
file\_np = np.load('arr.npz')  
print(file\_np)// <numpy.lib.npyio.NpzFile object at 0x0000021990B612B0>  
print(file\_np['arr1'])// [0 1 2]

补充: 1.\_\_str\_\_()返回object的字符串形式2. allow\_pickle 是否允许对数组、类进行序列化

3.torch基本操作:

1).torch切片操作：

取torch中的部分元素(降维操作):

import torch  
  
tensor = torch.randn(size=(3, 4, 5))  
print(tensor)  
print(tensor[:, -1, -1])  
print(tensor[:, :, -1])

2).torch 函数：

针对所有元素返回的是标量，针对给定维度返回的是降维矢量

.argmax(input,dim=None) 返回输入张量中所有元素(给定维度dim)的最大值的索引

.max(input,dim=None) 返回一个元组(values, indexes)，其中values是所有元素(给定维度dim)输入张量的每一行的最大值;而indexes是找到的每个最大值的索引位置

.mean(input,dim=None):返回输入张量中所有元素(给定维度dim)中的平均值;输入必须是浮点数或复数

.std(input,dim=None):计算dim指定的维度上的标准偏差

import torch  
import numpy as np  
  
tensor = torch.randn(3, 6)  
print(tensor)  
tensor\_1 = torch.argmax(tensor, dim=1)  
print(tensor\_1)  
tensor\_2 = torch.max(tensor, dim=1)  
print(tensor\_2)  
tensor\_3 = torch.mean(tensor, dim=1)  
print(tensor\_3)  
tensor\_4 = torch.std(tensor, dim=1)  
print(tensor\_4)  
  
ndarr = tensor.numpy()  
print(ndarr)  
ndarr\_1 = np.argmax(ndarr, axis=1)  
print(ndarr\_1)  
ndarr\_2 = np.max(ndarr, axis=1)  
print(ndarr\_2)  
ndarr\_3 = np.mean(ndarr, axis=1)  
print(ndarr\_3)  
ndarr\_4 = np.std(ndarr, axis=1)  
print(ndarr\_4)

.sum(input,dim=None) 返回输入张量中所有元素(给定维度dim)的和

import torch  
  
tensor = torch.tensor([[False, False, True, False], [True, False, True, False]])  
print(tensor.sum())//3

3).模型参数

.named\_parameters():将会打印每一次迭代元素的名字和param。

.parameters():将会打印每一次迭代元素的param而不会打印名字。

4.numpy、torch的对比

1).numpy的默认dtype=int32/float64;torch的默认dtype=torch.int64/torch.float32

tensor = torch.tensor(data=[[1, 2, 3], [4, 5, 6]])  
print(tensor.dtype)// torch.int64  
ndarray = np.array([1, 2, 3])  
print(ndarray.dtype)// int32

tensor = torch.tensor(data=[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])  
print(tensor.dtype)  
ndarray = np.array([1.0, 2.0, 3.0])  
print(ndarray.dtype)

2).numpy更改数据类型astype();torch更改数据类型type()

tensor = torch.tensor(data=[[1, 2, 3], [4, 5, 6]]).type(torch.float32)  
print(tensor.dtype)// torch.float32  
ndarray = np.array([1, 2, 3]).astype(np.float)  
print(ndarray.dtype)// float64

补充: astype(np.float32)等价于astype(‘float32’)

3).numpy、torch的基本运算(+、-、\*、/)

数组与标量的运算、数组与数组的运算:广播机制+向量元素乘

import torch  
import numpy as np  
  
ndarray1 = np.arange(1, 13).reshape(4, 3) # type:np.ndarray  
print(ndarray1)  
  
ndarray2 = np.arange(1, 4).reshape(1, 3)  
print(ndarray2)  
  
ndarray3 = ndarray1 \* ndarray2  
# ndarray3 = ndarray1.\_\_mul\_\_(ndarray2)  
print(ndarray3)  
  
tensor1 = torch.arange(1, 13).reshape(4, 3)  
print(tensor1)  
  
tensor2 = torch.arange(1, 4).reshape(3, )  
print(tensor2)  
  
tensor3 = tensor1 \* tensor2  
# tensor3 = tensor1.mul(tensor2)

# tensor3 = tensor1.\_\_mul\_\_(tensor2)  
print(tensor3)

4).numpy、torch的转换维度、坐标系变换

numpy维度变化:

.transpose():坐标系变换

import torch  
import numpy as np  
  
ndarray = np.array([[[8, 5, 5, 4],  
 [5, 4, 4, 0],  
 [2, 6, 7, 0]],  
  
 [[6, 4, 7, 0],  
 [8, 0, 6, 6],  
 [6, 0, 7, 2]]])  
print(ndarray)  
ndarray = ndarray.transpose((2, 1, 0))  
print(ndarray)

.reshape():转换维度

import torch  
import numpy as np  
  
ndarray = np.array([[[8, 5, 5, 4],  
 [5, 4, 4, 0],  
 [2, 6, 7, 0]],  
  
 [[6, 4, 7, 0],  
 [8, 0, 6, 6],  
 [6, 0, 7, 2]]])  
print(ndarray)  
ndarray = ndarray.reshape(3, 8)  
print(ndarray)

torch:

.permute():坐标系变换(相当于多次transpose操作)

import torch  
import numpy as np  
  
tensor = torch.randint(0, 10, size=(3, 2, 4))  
print(tensor)  
tensor = tensor.permute(dims=(2, 1, 0))  
print(tensor)

.reshape():转换维度

import torch  
import numpy as np  
  
tensor = torch.randint(0, 10, (2, 3, 4))  
print(tensor)  
tensor = tensor.reshape(3, 8)  
print(tensor)

5.sklearn基本操作

1).预处理preprocessing

preprocessing.StandardScaler().fit\_transform()

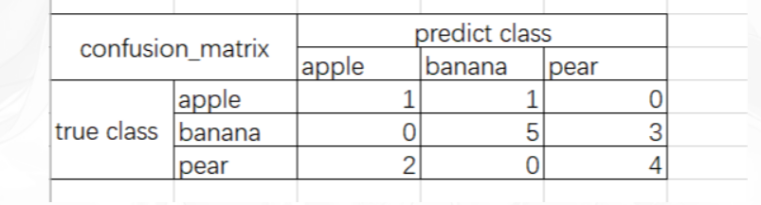
不仅计算训练数据的均值和方差,还会基于计算出来的均值和方差来转换训练数据,从而把数据转换成标准的正太分布

# 1.数据归一化  
ndarr = np.array([[0, 1, 2], [3, 4, 5]])  
scale = preprocessing.StandardScaler().fit(ndarr)  
print(scale.\_\_dict\_\_)  
print(scale.mean\_)// [1.5 2.5 3.5]  
print(scale.var\_)// [2.25 2.25 2.25]  
ndarr = scale.transform(ndarr)  
print(ndarr)// [[-1. -1. -1.],[ 1. 1. 1.]]  
  
# 2.数据归一化  
ndarr = np.array([[0, 1, 2], [3, 4, 5]])  
ndarr = preprocessing.StandardScaler().fit\_transform(ndarr)  
print(ndarr) // [[-1. -1. -1.],[ 1. 1. 1.]]

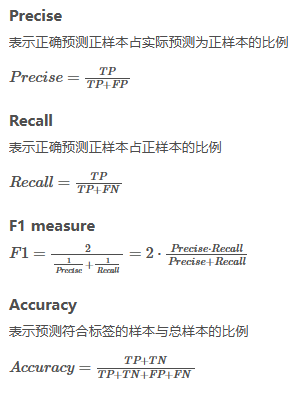
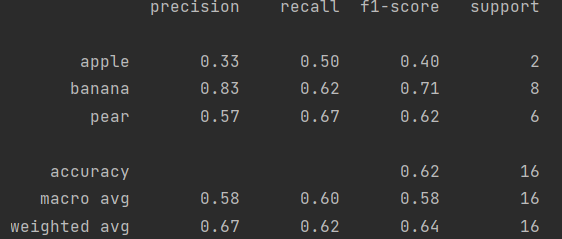
2).衡量指标metric

准确率: 所有分类正确的百分比

混淆矩阵:



分类报告:



TP——预测为 P (正例), 预测对了 TN——预测为 N (负例),预测对了

from sklearn.metrics import accuracy\_score  
from sklearn.metrics import confusion\_matrix  
from sklearn.metrics import classification\_report  
  
y\_test = ["a", "b", "p", "b", "b", "b", "b", "p", "b", "p", "b", "b", "p", "p", "p", "a"]  
y\_pred = ["a", "b", "p", "p", "p", "p", "b", "p", "b", "p", "b", "b", "a", "a", "p", "b"]  
  
score = accuracy\_score(y\_test, y\_pred)  
print(score)  
  
matrix = confusion\_matrix(y\_test, y\_pred, labels=["a", "b", "p"])  
print(matrix)  
  
report = classification\_report(y\_test, y\_pred, labels=["a", "b", "p"], target\_names=["apple", "banana", "pear"])  
print(report)

6.经验总结

1).torch实验复现技巧

设置numpy、cpu、gpu随机种子,并设置cudnn为确定性算法，保证每次得到的随机数序列都是一样的

torch.backends.cudnn.deterministic=True, deterministic确定性算法网络前馈结果无差异，相对应的为benchmark基准算法,benchmark模式会提升计算速度,但是由于计算中有随机性,网络前馈结果略有差异。

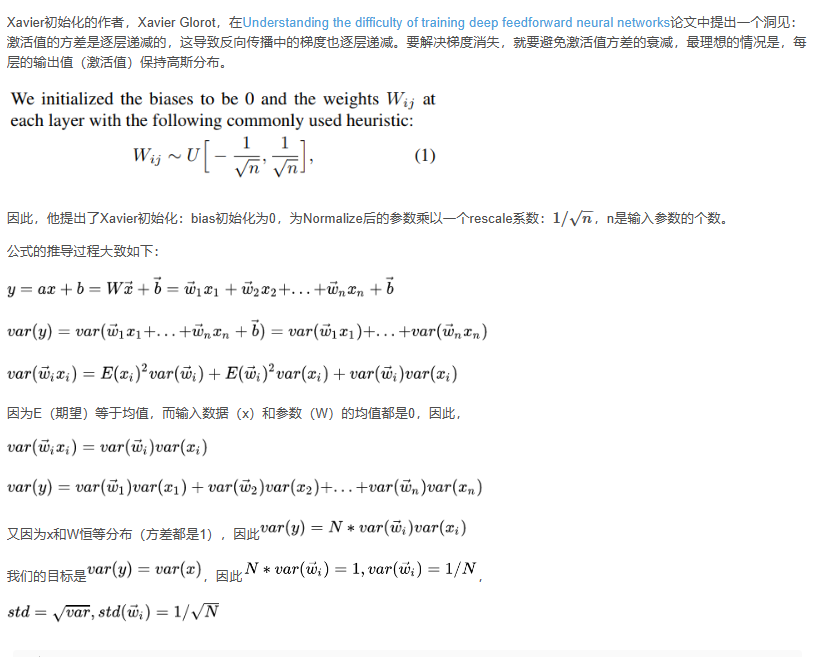
np.random.seed(1)  
torch.manual\_seed(1)  
torch.cuda.manual\_seed(1)  
torch.backends.cudnn.deterministic = True

2).初始化网络权重参数

在网络模型训练之前，对各个节点的权重和偏置进行初始化赋值的过程;embedding层权重参数(随机/预训练),无需初始化。

目的:防止模型在通过深度神经网络的正向传递过程中爆炸或消失。如果发生这种情况，损失梯度将太大或太小而无法向后流动，并且网络将需要更长的时间来收敛。

激活值的方差是逐层递减的，这导致反向传播中的梯度也逐层递减。要解决梯度消失，就要避免激活值方差的衰减，最理想的情况是，每层的输出值（激活值）保持高斯分布。



Xavier初始化,存在的问题只适用于线性激活函数;对于深层神经网络来说,线性激活函数是没有价值的;神经网络需要非线性激活函数来构建复杂的非线性系统。

针对relu的kaiming初始化，因为relu会抛弃掉小于0的值，对于一个均值为0的data来说，这就相当于砍掉了一半的值，这样一来，均值就会变大，前面Xavier初始化公式中E(x)=mean=0的情况就不成立了。根据新公式的推导，最终得到新的rescale系数：√2/n

def init\_network(model, method='xavier', exclude='embedding'):  
 for name, weight in model.named\_parameters():  
 if exclude not in name:  
 if 'weight' in name:  
 if method == 'xavier':  
 nn.init.xavier\_normal\_(weight)  
 elif method == 'kaiming':  
 nn.init.kaiming\_normal\_(weight)  
 else:  
 nn.init.normal\_(weight)  
 elif 'bias' in name:  
 nn.init.constant\_(weight, 0)  
 else:  
 pass

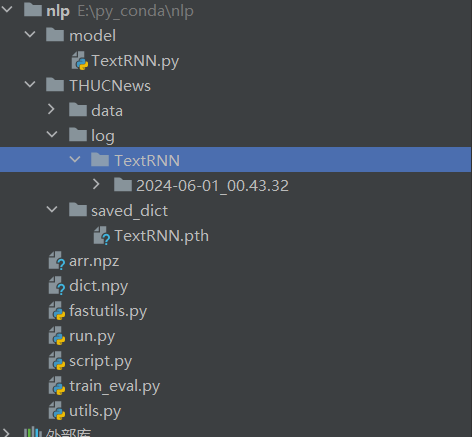
3).迭代次数优化——提前早停策略

验证集经过require\_improvement次迭代后,loss没有下降(或acc没有上升),则提前停止迭代（或减小学习率）。

4).合理设置模型参数获取——每N轮迭代所需时间

def get\_time\_diff(start\_time):  
 end\_time = time.time()  
 time\_dif = end\_time - start\_time  
 return timedelta(seconds=int(round(time\_dif)))

关键代码:



1. Config配置类:

配置类只能在Config类中进行静态修改(运行前修改参数、路径),在主函数main中进行动态修改;在其他类中允许只读;

class Config(object):  
 def \_\_init\_\_(self, dataset, embedding, use\_word):  
 self.model\_name = 'TextRNN'  
  
 self.num\_epoch = 10  
 self.batch\_size = 128  
 self.learning\_rate = 0.001  
 self.require\_improvement = 1000  
 self.last\_improvement = 0  
 self.embedding\_pretrained = torch.tensor(np.load(dataset + '/data/' + embedding)[  
 'embeddings'].astype('float32')) if embedding != 'random' else None  
 self.hidden\_size = 128  
 self.num\_layer = 2  
 self.drop\_out = 0.5  
 self.device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
  
 self.train\_path = dataset + '/data/train.txt'  
 self.dev\_path = dataset + '/data/dev.txt'  
 self.test\_path = dataset + '/data/test.txt'  
 self.vacab\_path = dataset + '/data/vocab.pkl'  
 self.n\_vocab = 0  
 self.pad\_size = 32  
 self.embed = self.embedding\_pretrained.size(1) if self.embedding\_pretrained is not None else 300  
 self.save\_path = dataset + '/saved\_dict/' + self.model\_name + '.pth'  
 self.log\_path = dataset + '/log/' + self.model\_name  
 self.class\_list = [x.strip() for x in open(dataset + '/data/class.txt').readlines()]  
 self.num\_class = len(self.class\_list)  
  
 self.use\_word = use\_word

1. 制作语料表

def build\_vocab(config: Config):  
 vocab\_dict = {}  
 if os.path.exists(config.vacab\_path):  
 vocab\_file = open(config.vacab\_path, 'rb')  
 vocab\_dict = pickle.load(vocab\_file)  
 return vocab\_dict  
 if not config.use\_word:  
 tokenizer = lambda x: [\_ for \_ in x] # 分字  
 else:  
 tokenizer = lambda x: x.split(" ") # 分词  
 with open(config.train\_path, 'r', encoding='utf-8') as f:  
 for line in tqdm(f):  
 lin = line.strip()  
 if not lin and lin != '': # 避免空行、文件末尾EOF  
 continue  
 text = tokenizer(lin.split('\t')[0])  
 for elem in text:  
 vocab\_dict[elem] = vocab\_dict.get(elem, 0) + 1  
 vocab\_list = sorted(vocab\_dict.items(), key=lambda x: x[1], reverse=True)  
 vocab\_dict = {item[0]: idx for idx, item in enumerate(vocab\_list)}  
 vocab\_dict.update({'<UNK>': len(vocab\_dict), '<PAD>': len(vocab\_dict) + 1})  
 vocab\_file = open(config.vacab\_path, 'wb')  
 pickle.dump(vocab\_dict, vocab\_file)  
 return vocab\_dict

1. 制作数据集

def build\_dataset(config: Config):  
 vocab\_table = build\_vocab(config)  
  
 embed\_from\_vocab = lambda content: [vocab\_table.get(\_, vocab\_table.get('<UNK>')) \  
 if \_ != PAD else vocab\_table.get('<PAD>') for \_ in content]  
  
 def load\_dataset(path, pad\_size=32):  
 data\_list = []  
 with open(path, 'r', encoding='utf-8') as f:  
 for line in tqdm(f):  
 lin = line.strip()  
 if not lin and lin != '':  
 continue  
 text, label = lin.split('\t')  
 seq\_len = len(text)  
 if seq\_len < pad\_size:  
 text = text + (pad\_size - seq\_len) \* PAD  
 else:  
 text = text[:pad\_size]  
 seq\_len = pad\_size  
 data\_list.append((embed\_from\_vocab(text), seq\_len, int(label)))  
 return data\_list  
  
 train\_dataset = load\_dataset(config.train\_path)  
 dev\_dataset = load\_dataset(config.dev\_path)  
 test\_dataset = load\_dataset(config.test\_path)  
  
 return vocab\_table, train\_dataset, dev\_dataset, test\_dataset

1. 制作数据迭代器

def build\_iterator(dataset, batch\_size, device):  
 iterator = DataIterator(dataset, batch\_size, device)  
 return iterator

自定义迭代器(一)

class DataIterator(object):  
 def \_\_init\_\_(self, dataset, batch\_size, device):  
 self.dataset = dataset  
 self.length = len(self.dataset)  
 self.batch\_size = batch\_size  
 self.n\_batch = self.length // self.batch\_size  
 self.residue = True if self.length % self.batch\_size != 0 else False  
 self.start = 0  
 self.device = device  
  
 def \_to\_tensor(self, datas):  
 x = torch.LongTensor([\_[0] for \_ in datas]).to(self.device)  
 y = torch.LongTensor([\_[2] for \_ in datas]).to(self.device)  
 seq\_len = torch.LongTensor([\_[1] for \_ in datas]).to(self.device)  
 return (x, seq\_len), y  
  
 def \_\_iter\_\_(self):  
 return self  
  
 def \_\_next\_\_(self):  
 if self.start == self.length:  
 self.start = 0  
 raise StopIteration  
 else:  
 end = self.start + self.batch\_size if self.start + self.batch\_size < self.length else self.length  
 item = self.dataset[self.start:end]  
 item = self.\_to\_tensor(item)  
 self.start = end  
 return item  
  
 def \_\_len\_\_(self):  
 if self.residue:  
 return self.n\_batch + 1  
 else:  
 return self.n\_batch

自定义迭代器(二)

class DatasetIterator(object):  
 def \_\_init\_\_(self, batches, batch\_size, device):  
 self.batches = batches  
 self.n\_batches = len(batches) // batch\_size  
 self.batch\_size = batch\_size  
 self.residue = False if len(batches) % self.n\_batches == 0 else True  
 self.device = device  
 self.index = 0  
  
 def \_to\_tensor(self, datas):  
 x = torch.LongTensor([\_[0] for \_ in datas]).to(self.device)  
 y = torch.LongTensor([\_[2] for \_ in datas]).to(self.device)  
 seq\_len = torch.LongTensor([\_[1] for \_ in datas]).to(self.device)  
 return (x, seq\_len), y  
  
 def \_\_next\_\_(self):  
 if self.residue and self.index == self.n\_batches:  
 batches = self.batches[self.index \* self.batch\_size:len(self.batches)]  
 self.index += 1  
 batches = self.\_to\_tensor(batches)  
 return batches  
 elif self.index >= self.n\_batches:  
 self.index = 0  
 raise StopIteration  
 else:  
 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

1. 定义模型

class Model(nn.Module):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_()  
 if config.embedding\_pretrained is not None:  
 self.embedding = nn.Embedding.from\_pretrained(embeddings=config.embedding\_pretrained, freeze=False)  
 else:  
 self.embedding = nn.Embedding(num\_embeddings=config.n\_vocab, embedding\_dim=config.embed,  
 padding\_idx=config.n\_vocab - 1)  
 self.lstm = nn.LSTM(input\_size=config.embed, hidden\_size=config.hidden\_size, num\_layers=config.num\_layer,  
 batch\_first=True, dropout=config.drop\_out, bidirectional=True)  
 self.fc = nn.Linear(in\_features=config.hidden\_size \* 2, out\_features=config.num\_class)  
  
 def forward(self, x):  
 x, \_ = x  
 out = self.embedding(x)  
 out, \_ = self.lstm(out)  
 out = self.fc(out[:, -1, :])  
 return out

1. 模型训练

import time  
  
import torch  
from sklearn import metrics  
from torch import nn  
from torch import optim  
from torch.nn import functional as F  
from torch.utils.tensorboard import SummaryWriter  
  
from utils import get\_time\_diff  
from model.TextRNN import Model, Config  
  
  
def init\_network(model, method='xavier', exclude='embedding'):  
 for name, weight in model.named\_parameters():  
 if exclude not in name:  
 if 'weight' in name:  
 if method == 'xavier':  
 nn.init.xavier\_normal\_(weight)  
 elif method == 'kaiming':  
 nn.init.kaiming\_normal\_(weight)  
 else:  
 nn.init.normal\_(weight)  
 elif 'bias' in name:  
 nn.init.constant\_(weight, 0)  
 else:  
 pass  
  
  
def train(config, model, train\_iterator, dev\_iterator, test\_iterator, writer: SummaryWriter):  
 start\_time = time.time()  
 optimizer = optim.Adam(model.parameters(), lr=config.learning\_rate)  
  
 total\_train = 0  
 dev\_best\_loss = float('inf')  
 last\_improve = 0  
 flag = False  
 for epoch in range(config.num\_epoch):  
 print('epoch[{}/{}]'.format(epoch, config.num\_epoch))  
 for i, (trains, labels) in enumerate(train\_iterator):  
 model.train()  
 optimizer.zero\_grad()  
 output = model(trains)  
 train\_loss = F.cross\_entropy(output, labels).to(config.device)  
 train\_loss.backward()  
 optimizer.step()  
 if total\_train % 100 == 0:  
 true = labels.data.cpu()  
 predict = torch.argmax(output.data.cpu(), dim=1)  
 train\_acc = metrics.accuracy\_score(true, predict)  
 dev\_acc, dev\_loss = evaluate(config, model, dev\_iterator)  
 if dev\_loss < dev\_best\_loss:  
 dev\_best\_loss = dev\_loss  
 torch.save(model.state\_dict(), config.save\_path)  
 improve = '\*'  
 last\_improve = total\_train  
 else:  
 improve = ''  
 time\_diff = get\_time\_diff(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\_train, train\_loss.item(), train\_acc, dev\_loss.item(), dev\_acc, time\_diff, improve))  
 writer.add\_scalar('train\_loss', train\_loss.item(), total\_train)  
 writer.add\_scalar('train\_acc', train\_acc, total\_train)  
 writer.add\_scalar('dev\_loss', dev\_loss.item(), total\_train)  
 writer.add\_scalar('dev\_acc', dev\_acc, total\_train)  
 total\_train += 1  
 if total\_train - last\_improve > config.require\_improvement:  
 print('No optimization for a long time, auto-stopping...')  
 flag = True  
 break  
 if flag:  
 break  
 writer.close()  
 print('------complete train start test---------')  
 test(config, model, test\_iterator)  
  
  
def evaluate(config, model, data\_iterator, test=False):  
 model.eval()  
 predicts\_all = torch.tensor([], dtype=torch.int64)  
 labels\_all = torch.tensor([], dtype=torch.int64)  
 loss\_total = 0  
 with torch.no\_grad():  
 for i, (devs, labels) in enumerate(data\_iterator):  
 outputs = model(devs)  
 loss\_total += F.cross\_entropy(outputs, labels).to(config.device)  
 predicts = torch.argmax(outputs, dim=1)  
 predicts\_all = torch.cat((predicts\_all, predicts.cpu()))  
 labels\_all = torch.cat((labels\_all, labels.cpu()))  
 acc = metrics.accuracy\_score(predicts\_all, labels\_all)  
 if test:  
 test\_confusion = metrics.confusion\_matrix(y\_true=labels\_all, y\_pred=predicts\_all)  
 test\_report = metrics.classification\_report(y\_true=labels\_all, y\_pred=predicts\_all,  
 target\_names=config.class\_list)  
 return acc, loss\_total / len(data\_iterator), test\_confusion, test\_report  
 return acc, loss\_total / len(data\_iterator)  
  
  
def test(config, model: nn.Module, test\_iterator):  
 model.load\_state\_dict(torch.load(config.save\_path))  
 start\_time = time.time()  
 test\_acc, test\_loss, test\_confusion, test\_report = evaluate(config, model, test\_iterator, test=True)  
 msg = 'Test Loss: {0:>5.2}, Test Acc: {1:>6.2%}'  
 print(msg.format(test\_loss, test\_acc))  
 print("Confusion Matrix...")  
 print(test\_confusion)  
 print("Precision, Recall and F1-Score...")  
 print(test\_report)  
 time\_dif = get\_time\_diff(start\_time)  
 print("Time usage:", time\_dif)

1. 运行文件

import argparse  
import time  
import numpy as np  
from importlib import import\_module  
  
import torch.cuda.random  
  
from train\_eval import init\_network, train  
from torch.utils.tensorboard import SummaryWriter  
  
parser = argparse.ArgumentParser()  
parser.add\_argument('--model', required=True, help='choose a model:TextCNN、TextRNN、FastText')  
parser.add\_argument('--embedding', default='pre\_trained', help='choose random or pre\_trained')  
parser.add\_argument('--word', default=False, help='True for word,False for char')  
args = parser.parse\_args()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 dataset = 'THUCNews'  
 embedding = 'embedding\_SougouNews.npz'  
 if args.embedding == 'random':  
 embedding = 'random'  
 model\_name = args.model  
 if model\_name == 'FastText':  
 from fastutils import get\_time\_diff, build\_dataset, build\_iterator  
  
 embedding = 'random'  
 else:  
 from utils import get\_time\_diff, build\_dataset, build\_iterator  
 use\_word = args.word  
  
 x = import\_module('model.' + model\_name)  
 config = x.Config(dataset, embedding, use\_word)  
 np.random.seed(1)  
 torch.manual\_seed(1)  
 torch.cuda.manual\_seed(1)  
 torch.backends.cudnn.deterministic = True  
  
 # loading data  
 print('loading data')  
 start\_time = time.time()  
 vocab\_table, train\_dataset, test\_dataset, dev\_dataset = build\_dataset(config)  
 config.n\_vocab = len(vocab\_table)  
 train\_iterator = build\_iterator(train\_dataset, config.batch\_size, config.device)  
 dev\_iterator = build\_iterator(dev\_dataset, config.batch\_size, config.device)  
 test\_iterator = build\_iterator(test\_dataset, config.batch\_size, config.device)  
 tme\_diff = get\_time\_diff(start\_time)  
  
 # train  
 model = x.Model(config).to(config.device)  
 writer = SummaryWriter(log\_dir=config.log\_path + '/' + time.strftime('%Y-%m-%d\_%H.%M.%S', time.localtime()))  
 if model\_name != 'Transformer':  
 init\_network(model)  
 print(model.parameters)  
 train(config, model, train\_iterator, dev\_iterator, test\_iterator, writer)

# RE-GCN论文代码

## 论文理解

## numpy操作

1. **array和asArray的区别:**

array: 深拷贝 asArray:浅拷贝

Eg:

import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])

arr2 = np.asarray(arr1)

arr3 = np.array(arr1)

arr2[1, :] = [6, 7, 9]

print(arr1)

print(arr2)

arr3[:, 2] = [-1, -1]

print(arr1)

print(arr3)

Tip:

1). id返回对象在内存中的地址

1. **arange**

用于生成一个具有特定步长的等差数列数组

Eg:

import numpy as np

arr = np.arange(1, 10, 2, dtype=float)

print(arr)

Tip:

1.前向传播输入:前向传播中的输入数据类型应该是 float（通常是 float32），因为神经网络的计算依赖于浮点数。

2.标签:对于分类任务中的标签数据，通常使用 long 类型（int64），例如用于 CrossEntropyLoss 等损失函数。

1. **transpose**

用于对数组进行转置操作(交换数组的轴)

Eg:

import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]],

[[4, 5, 6], [7, 8, 9]],

[[7, 8, 9], [1, 2, 3]]])

print(arr.transpose())

print(arr.transpose(2, 1, 0))

print(arr.transpose((2, 0, 1)))

Tip:

1. arr.transpose()==arr.transpose(2, 1, 0)
2. **concatnate**

用于沿指定轴连接多个数组

Eg:

import numpy as np

vec1 = np.array([[1, 2, 3]])

vec2 = np.array([[4, 5, 6]])

matrix\_1 = np.concatenate((vec1, vec2), axis=0)

print(matrix\_1)

matrix\_2 = np.concatenate((vec1, vec2), axis=1)

print(matrix\_2)

Tip:

1. concatnate指定维度为指定维度之和，其余维度保持不变;
2. **unique**

用于找到数组中的唯一元素，并返回这些唯一元素的排序(升序排序)列表

return\_index:返回唯一元素在原数组中的索引

return\_inverse:返回原数组元素在唯一元素中的索引

return\_counts:返回每个唯一元素的出现次数

axis：沿着哪个轴查找唯一元素。默认情况下（axis=None），会对整个数组进行操作

Eg1:

import numpy as np

arr1 = [1, 2, 3, 4, 5]

arr2 = [2, 3, 4, 5, 6]

res, index = np.unique((arr1, arr2), return\_inverse=True)

print(res, index)

Eg2:

arr1 = [1, 2, 3, 4, 5]

arr2 = [5, 3, 4, 5, 6]

arr3 = [1, 2, 3, 4, 5]

res, index = np.unique((arr1, arr2, arr3), return\_inverse=True, axis=0) // axis=1没意义

print(res)

## pytorch 操作

1. **nonzero**

用于返回一个张量中所有非零元素的索引

torch.nonzero(input, as\_tuple=False) → Tensor

Input：输入的张量，类型可以是任意类型的张量

as\_tuple：如果设置为 True，返回的结果将是一个包含每个维度索引的元组；如果为 False，则返回一个形状为 (N, D) 的二维张量，其中 N 是非零元素的数量，D 是张量的维度

返回值：返回一个张量，包含所有非零元素的索引(坐标)

Eg:

import torch

matrix1 = torch.tensor([0, 1, 2, 3, 0])

matrix2 = torch.tensor([[0, 1, 2], [1, 2, 0], [3, 2, 0]])

print(torch.nonzero(matrix1))

print(torch.nonzero(matrix1, as\_tuple=True))

print(torch.nonzero(matrix2))

print(torch.nonzero(matrix2, as\_tuple=True))

Tip:

1. Tensor索引取值?

import torch

matrix = torch.tensor([[0, 1, 2], [1, 2, 0], [3, 2, 0]])

index = torch.tensor([[0, 1], [1, 2], [0, 2]])

print(matrix.shape)

print(index.shape)

print(matrix[index])

tensor索引取值 — 依次遍历index索引，按最高维度从目标Tensor中取值;

1. **index\_select**

是一个用于根据给定的索引选择张量元素的操作

torch.index\_select(input, dim, index, out=None) → Tensor

input：输入的张量。

dim：指定要选择的维度，0 代表行维度，1 代表列维度，依此类推

index：一个包含索引的张量。它可以是一个一维张量，包含要选择的索引

返回值：返回一个新的张量，包含在指定维度（dim）上根据给定索引选择的元素（除指定维度外，其余维度大小与原始张量维度大小一致）

Eg:

import torch

features = torch.randn(6, 2)

print(features)

index = torch.randint(0, 6, (6,))

print(index)

print(features.index\_select(0, index))

1. **masked\_select**

是一个根据布尔掩码选择张量元素的操作

torch.masked\_select(input, mask) → Tensor

input：输入的张量

mask：一个与 input 形状相同的布尔张量，指定哪些元素会被选择（True 选择，False 不选择）

返回值：返回一个一维张量，包含 input 张量中所有在 mask 中为 True 的元素。返回的张量的形状与 mask 中为 True 的元素个数相同。

Eg:

import torch

features = torch.randn(6, 2)

print(features)

mask = torch.randint(0, 2, (6, 2))

print(mask)

print(features.masked\_select(mask == 1))

1. **masked\_fill**

用于根据布尔掩码将张量中的元素替换为指定的值

torch.masked\_fill(input, mask, value) → Tensor

input：输入的张量，任意形状

mask：一个与 input 张量相同形状的布尔张量，用来指示哪些元素需要被替换。True 表示该位置的元素将被替换

value：替换的值，可以是标量或与 input 张量形状匹配的张量。它会替换所有对应位置为 True 的元素

返回值：返回一个新的张量，其中掩码为 True 的位置的元素被替换为 value，其余位置的元素保持不变

Eg:

import torch

features = torch.randn(6, 2)

print(features)

mask = torch.randint(0, 2, (6, 2))

print(mask)

print(features.masked\_fill(~(mask == 1), 0))

1. **cat**

## dgl操作

1. **创建图**

Eg:

import dgl

src = [0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 1, 2]

dst = [1, 2, 3, 4, 5, 0, 0, 0, 0, 0, 2, 1]

g = dgl.graph((src, dst))

1. **创建无向图**

Eg:

import dgl

g = dgl.to\_bidirected(g)

1. **可视化**

Eg:

import networkx as nx

from matplotlib import pyplot as plt

nx.draw(g.to\_networkx(), with\_labels=True)

plt.show()

1. **dgl 消息传递范式(消息+聚合+更新)**

Eg1：

import torch

import dgl

import networkx as nx

from matplotlib import pyplot as plt

src = [0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 1, 2]

dst = [1, 2, 3, 4, 5, 0, 0, 0, 0, 0, 2, 1]

g = dgl.graph((src, dst))

nx.draw(g.to\_networkx(), with\_labels=True)

plt.show()

g.ndata['feat'] = torch.randn(6, 2)

print(g.ndata['feat'])

print(g.ndata['feat'][1:])

# 计算结点0聚合后的特征

print(torch.sum(g.ndata['feat'][1:], dim=0))

def message\_func(edges):

return {'msg': edges.src['feat']}

def reduce\_func(nodes):

""" reduce\_func(nodes) 会被多次调用，并行计算

return [批量大小, 所有公共边相同, 特征维度]

1.批量大小 == 所有公共边相同的数量 2.所有公共边相同（升序排序）"""

print("shape{}".format(nodes.mailbox['msg'].shape))

feat = torch.mean(nodes.mailbox['msg'], dim=1, keepdim=True)

return {'feat': feat}

def apply\_func(nodes):

feat = nodes.data['feat']

return {'feat': feat}

g.update\_all(message\_func, reduce\_func, apply\_func)

print(g.ndata['feat'])

Eg2:

import torch

import dgl

import dgl.function.reducer as fr

import networkx as nx

from matplotlib import pyplot as plt

src = [0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 1, 2]

dst = [1, 2, 3, 4, 5, 0, 0, 0, 0, 0, 2, 1]

g = dgl.graph((src, dst))

nx.draw(g.to\_networkx(), with\_labels=True)

plt.show()

g.ndata['feat'] = torch.randn(6, 2)

print(g.ndata['feat'])

# 计算结点0聚合后的特征

print(torch.sum(g.ndata['feat'][1:], dim=0))

def message\_func(edges):

return {'msg': edges.src['feat']}

def apply\_func(nodes):

feat = nodes.data['feat']

return {'feat': feat}

g.update\_all(message\_func, fr.sum('msg', 'feat'), apply\_func)

print(g.ndata['feat'])

# 多GPU并行训练

1.常见的gpu使用方法（model parallel、data parallel）

图示

AI 生成的内容可能不正确。

2.数据集如何在不同设备间分配

3.误差梯度如何在不同设备之间通信

[!!!important]:调整梯度的同时，需要适当调大学习率; 简单粗暴的方法，使用几块gpu，就对应将学习率扩大几倍;

4.BatchNormalization如何在不同设备间同步

[!!!important]:

batch\_size越大,每个batch数据的均值和方差越接近整体数据的均值和方差; 同步BN精度会更高，但训练的时间相对也会较长;

5. pytorch并行训练API – DistributedDataParallel

pytorch中使用多gpu训练的常用启动方式:

1. . torch.distributed.launch 代码量少点，启动速度快点

python -m torch.distributed.launch --help [手动终止，关注显存的占用]

python -m torch.distributed.launch --use\_env（os.environ)

**torch >= 2.0**

torchrun --nproc\_per\_node=2 /home/ps/lzy/vista/demo/multi\_gpu\_demo.py

1. . torch.multiprocessing 拥有更好的控制和灵活性

6. 代码实现:

1). init\_distributed\_mode函数:

def init\_distributed\_mode(args):

if 'RANK' in os.environ and 'WORLD\_SIZE' in os.environ:

args.rank = int(os.environ["RANK"])

args.world\_size = int(os.environ['WORLD\_SIZE'])

args.gpu = int(os.environ['LOCAL\_RANK'])

elif 'SLURM\_PROCID' in os.environ:

args.rank = int(os.environ['SLURM\_PROCID'])

args.gpu = args.rank % torch.cuda.device\_count()

else:

print('Not using distributed mode')

args.distributed = False

return

args.distributed = True

torch.cuda.set\_device(args.gpu)

args.dist\_backend = 'nccl' # 通信后端，nvidia GPU推荐使用NCCL

print('| distributed init (rank {}): {}'.format(

args.rank, args.dist\_url), flush=True)

dist.init\_process\_group(backend=args.dist\_backend, init\_method=args.dist\_url,

world\_size=args.world\_size, rank=args.rank)

dist.barrier()

2). 数据集在不同设备间进行分配

# 平分数据给每块gpu

train\_sampler = torch.utils.data.distributed.DistributedSampler(train\_data\_set)

val\_sampler = torch.utils.data.distributed.DistributedSampler(val\_data\_set)

# 设置每块gpu所对应的batch\_size

train\_batch\_sampler = torch.utils.data.BatchSampler(

train\_sampler, batch\_size, drop\_last=True)

nw = min([os.cpu\_count(), batch\_size if batch\_size > 1 else 0, 8]) # number of workers

if rank == 0:

print('Using {} dataloader workers every process'.format(nw))

train\_loader = torch.utils.data.DataLoader(train\_data\_set,

batch\_sampler=train\_batch\_sampler,

pin\_memory=True,

num\_workers=nw,

collate\_fn=train\_data\_set.collate\_fn)

val\_loader = torch.utils.data.DataLoader(val\_data\_set,

batch\_size=batch\_size,

sampler=val\_sampler,

pin\_memory=True,

num\_workers=nw,

collate\_fn=val\_data\_set.collate\_fn)

3). 模型加载（保存第一块gpu下的模型参数，复制模型参数到其他gpu）

# 如果存在预训练权重则载入

if os.path.exists(weights\_path):

weights\_dict = torch.load(weights\_path, map\_location=device)

load\_weights\_dict = {k: v for k, v in weights\_dict.items()

if model.state\_dict()[k].numel() == v.numel()}

model.load\_state\_dict(load\_weights\_dict, strict=False)

else:

checkpoint\_path = os.path.join(tempfile.gettempdir(), "initial\_weights.pt")

# 如果不存在预训练权重，需要将第一个进程中的权重保存，然后其他进程载入，保持初始化权重一致

if rank == 0:

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

dist.barrier()

# 这里注意，一定要指定map\_location参数，否则会导致第一块GPU占用更多资源

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

4). 转化模型为DDP模型

model = torch.nn.parallel.DistributedDataParallel(model, device\_ids=[args.gpu])

[!!!important]DistributedDataParallel (DDP) 会在多个进程中各自维护一份模型副本，并在 每个 GPU 上分别计算梯度，然后使用 NCCL（默认后端）在不同 GPU 之间同步梯度;

DDP会自动执行 all-reduce 操作，确保梯度正确聚合。

5). 同步BatchNorm

if args.syncBN:

# 使用SyncBatchNorm后训练会更耗时

model = torch.nn.SyncBatchNorm.convert\_sync\_batchnorm(model).to(device)

6). 模型训练过程

I.数据集shuffle

train\_sampler.set\_epoch(epoch)

if device != torch.device('cpu'):  
 torch.cuda.synchronize(device)

II.误差梯度通信

model = torch.nn.parallel.DistributedDataParallel(model, device\_ids=[args.gpu])

III.损失函数计算

def reduce\_value(value, average=True):

world\_size = get\_world\_size()

if world\_size < 2: # 单GPU的情况

return value

with torch.no\_grad():

dist.all\_reduce(value)

if average:

value /= world\_size

return value

IV.评估函数

def reduce\_value(value, average=True):

world\_size = get\_world\_size()

if world\_size < 2: # 单GPU的情况

return value

with torch.no\_grad():

dist.all\_reduce(value)

if average:

value /= world\_size

return value

7). 删除临时缓存文件

# 删除临时缓存文件

if rank == 0:

if os.path.exists(checkpoint\_path) is True:

os.remove(checkpoint\_path)

补充:

1. linux杀死某个进程pid

lsof -i:端口号

kill -9 pid

# 注释规范

1. ”“” ”“”: 类功能、方法功能注释;
2. ””: 参数含义注释;
3. #: 代码可修改注释(Torch形状、论文公式、特定网络);

# SSH连接

1. 原106（2张3090+1张4060Ti）

ip：172.17.169.70

port：34

用户：ps

密码：1qazse45

1. 原110（2张3090）

ip：172.17.174.152

port：22

用户：ps

密码：`1qazse45