**中国矿业大学计算机学院**

**2019 级本科生课程作业**

|  |  |
| --- | --- |
| 课程名称 | 机器学习及优化——实验三 |
| 报告时间 | 2023年4月25日 |
| 学生姓名 | 许万鹏 |
| 学 号 | 05191643 |
| 班 级 | 信息安全2019-01班 |
| 任课教师 | 李政伟 |

摘要

情感分析是对文本中所表达情感进行分析和识别的过程，我们可以使用机器学习技术更好地分析自然语言所表达的情感。在本次实验中，本人首先使用机器学习库Scikit-Learn实现了一种简洁且有效的预测方式。为了更深入地理解各机器学习算法的原理，将理论知识工程化，本人基于目前最常用的深度学习框架PyTorch，从零实现了本次任务中用到的所有机器学习算法，如SGD、Adam等，并将它们组合成了一个完整的框架——XuTorch，最后调用该框架完成了中文的情感分析任务。

目 录

[摘要 II](#_Toc133707131)

[1 概述 1](#_Toc133707132)

[1.1 数据集介绍 1](#_Toc133707133)

[1.2 基于特征的自然语言情感分析 1](#_Toc133707134)

[2 基于SkLearn的简洁实现 2](#_Toc133707135)

[3 基于PyTorch的从零实现 4](#_Toc133707136)

[3.1 调用XuTorch的实现 4](#_Toc133707137)

[3.2 基于Pytorch实现XuTorch 9](#_Toc133707138)

[3.2.1 xutorch.dataset.prep\_dataloader 10](#_Toc133707139)

[3.2.2 xutorch.dataset.Sentiment 11](#_Toc133707140)

[3.2.3 xutorch.misc.set\_seeds 13](#_Toc133707141)

[3.2.4 xutorch.misc.get\_device 14](#_Toc133707142)

[3.2.5 xutorch.model.MLP 14](#_Toc133707143)

[3.2.6 xutorch.model.EmbeddingMLP 15](#_Toc133707144)

[3.2.7 xutorch.optim.SimpleAdam 15](#_Toc133707145)

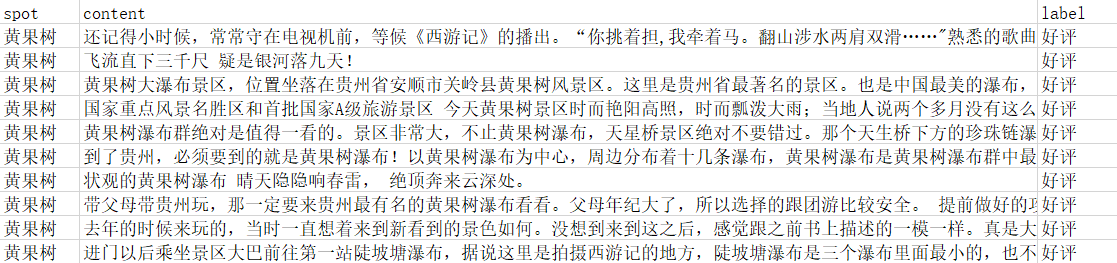
[3.2.8 xutorch.loss.CE 17](#_Toc133707146)

[3.2.9 xutorch.plot.\* 18](#_Toc133707147)

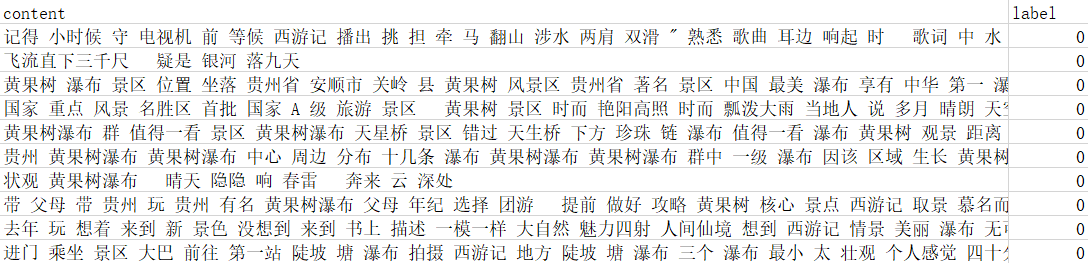
# 概述

## 数据集介绍

情感分析作为NLP领域内比较热门的方向，其数据集规模很大。出于理解基本机器学习技术而不是对SOTA方法改进的目的，我们不使用大规模数据集，而是从大众点评网站上爬取6000条左右的好评/差评信息（如下图）做二分类任务。当然对于多分类任务也是一样的，只需要更改网络的输出维度即可。这种中文的情感二分类任务也是有实际用途的，例如我国的AI辅助监管系统，就可以通过用户评论的正面/负面来标记用户是否经常发送违背社区规则的言论。



通过jieba库进行简单的分词预处理，可以得到如下图所示的数据。



## 基于特征的自然语言情感分析

这里介绍了基于特征的自然语言情感分析与传统机器学习任务的区别和联系，以及其中出现问题的解决方法。

NLP领域作为人工智能的热门子领域，已经有多种针对自然语言进行情感预测的方法，如Transform和Bert。但我们在这里用的是一种基于特征的方法，这种方法属于机器学习方法，大致思路是将句子分词为单词，将单词作为特征，然后像传统机器学习方法一样通过特征和标签进行模型学习。这种方法的好处是简单，快速，模型小；但也有不好的地方，就是没有语法和前后文关系，这些需要使用循环神经网络，如LSTM，或更高级的Transform或Bert实现。

“将句子分为数个词汇作为特征，然后进行传统机器学习。”这个地方听起来简单，但是自然语言中的词汇全部都是离散的，且数量庞大，其中不乏有许多没有情绪意义的词/数字/符号，如“而且”、“一个”、“一般”、“211”、“（）”等。

对于离散的特征，我们有解决方法，就是将其one-hot化，一个特征对应一个位置，如[0, 0, 0, 1]。在NLP中有一种类似的方法——Bag-of-Words Model，即词袋模型，同样是将离散特征（单词）数值化，但不是一个单词对应一个位置，而是一个单词对应一个数，如3。这次任务我们会使用词袋模型。

对于数量庞大的词汇们，我们也有一种简单有效的处理办法——去除低频单词。将单词数值化放入词袋的同时，我们会统计它们的频率，因此我们可以通过设置最低频率阈值min\_df来对低频单词进行去除。当然，这是中文的情况，因为我们主要处理中文词汇，所以对于其中夹杂的英文词汇，我们可以直接通过其长度进行去除，毕竟大部分英文词汇长度越短，其自身所包含的信息量越少。

对于没有意义的词，我们可以将其标记为停用词，保存在字典里面，当处理时碰到位于停用词字典里的词，忽略它即可。

即便这样，处理后的词汇数量仍然达到了5000个以上！这和前两个任务中的13个、30个特征并不在一个数量级，要知道我们仅仅爬取了约6000条数据！

而这样又带来了两个新的问题。

一：从每条句子中提取的特征数量都不同，同一个batch内甚至无法将输入进行堆叠，更无法将变长的输入放入模型。

二：从一条句子中提取的特征远小于从句子全集中提取的特征，这是因为自然语言的词汇本就是离散的。例如一条句子可能仅有20个特征，要怎么放到一个宽度为5000条特征的网络呢？

首先，对于第一个问题，我们可以强制规定句子的特征数量num\_sentence\_word，多裁少补，将规定长度外的特征去除，或用某个固定的数（如<PAD>: 0）来补足特征。这样一般不会带来问题，因为超长的特征带来的细微情感变化对于我们预测大体情感是多余的，另外添加的0也会被深度神经网络学习到它并没有帮助分析情感的作用。

然后，对于第二个问题，对于小模型，我们可以不设置全体特征数为层的尺寸，毕竟宽度为5000的网络不那么容易训练，我们将特征数设置得比num\_sentence\_word大即可，这样的网络更方便训练，对理解NLP任务的帮助更大，至于为什么可以呢？我们可以认为神经网络将部分类似情感的词汇进行了合并，完成了训练。

另外，如果想要训练更精确的模型，也可以设置全体特征数为层的尺寸，我们可以将输入层设置为Embedding层，这个层会自动将低维度的输入映射到高维度上。

# 基于SkLearn的简洁实现

首先，导入sklearn中已实现的函数。

import numpy as np  
import pandas as pd  
from scipy.sparse import coo\_matrix  
from sklearn.neural\_network import MLPClassifier  
from sklearn.metrics import classification\_report  
from sklearn.model\_selection import train\_test\_split  
from sklearn.feature\_extraction.text import CountVectorizer  
from sklearn.feature\_extraction.text import TfidfTransformer

然后，按照机器学习的一般流程：准备数据、数据分析、特征工程、模型选择及训练、模型评估，逐个调用函数。这里与一般的机器学习任务有些不同，但大致上是一样的。

*# 加载黄果山好评数据集*df = pd.read\_csv('huangguo\_mountain.csv', encoding='UTF-8')  
df = df.dropna()  
  
contents = df['content']  
labels = df['label']  
  
*#将文本中的词语转换为词频矩阵 矩阵元素a[i][j]表示j词在i类文本下的词频*vectorizer = CountVectorizer(min\_df=5)  
  
*#该类会统计每个词语的tf-idf权值*transformer = TfidfTransformer()  
  
*#第一个fit\_transform是计算tf-idf 第二个fit\_transform是将文本转为词频矩阵*tfidf = transformer.fit\_transform(vectorizer.fit\_transform(contents))  
  
*# 获取词袋模型中的所有词语*word = vectorizer.get\_feature\_names\_out()  
print("词汇数量:", len(word))  
  
*#将tf-idf矩阵抽取出来，元素w[i][j]表示j词在i类文本中的tf-idf权重*X = coo\_matrix(tfidf, dtype=np.float32).toarray() *#稀疏矩阵 注意float  
  
# 随机划分训练集和测试集*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, labels, test\_size=0.3, random\_state=1)  
  
*# 创建MLP*classifier = MLPClassifier()  
  
*# 训练*classifier.fit(X\_train, y\_train)  
  
print('模型的准确度: ', classifier.score(X\_test, y\_test))  
  
y\_preds = classifier.predict(X\_test)  
print(classification\_report(y\_test, y\_preds))

得到输出如下。

词汇数量: 4923

模型的准确度: 0.9890230515916575

precision recall f1-score support

0 1.00 0.99 0.99 1527

1 0.94 1.00 0.97 295

accuracy 0.99 1822

macro avg 0.97 0.99 0.98 1822

weighted avg 0.99 0.99 0.99 1822

可以看出，基于sklearn的代码可以很简洁地（不超过30行）得到较好的结果（精确率Acc 0.989）。

# 基于PyTorch的从零实现

## 调用XuTorch的实现

导入PyTroch和XuTroch库

import torch  
import xutorch

设置参数

args = {  
 'min\_df': 5,  
 'num\_sentence\_word': 100,  
 'lr': 0.0005,  
 'num\_epochs': 200,  
 'batch\_size': 8,  
 'early\_stop': 50,  
 'save\_path': 'weights/hw3\_model.pth'  
}

创建数据加载器

train\_loader, train\_dataset = xutorch.dataset.prep\_dataloader(  
 dataset\_name='Sentiment',  
 download=False,  
 train=True,   
 batch\_size=args['batch\_size'],   
 test\_ratio=0.25,   
 min\_df=args['min\_df'],   
 num\_sentence\_word = args['num\_sentence\_word']  
)  
val\_loader, val\_dataset = xutorch.dataset.prep\_dataloader(  
 dataset\_name='Sentiment',  
 download=False,  
 train=False,   
 batch\_size=args['batch\_size'],   
 test\_ratio=0.25,   
 min\_df=args['min\_df'],   
 num\_sentence\_word = args['num\_sentence\_word']  
)

输出如下。

Finished reading the train set of Text Dataset (4573 samples found), each dim = 5044

Finished reading the val set of Text Dataset (1500 samples found), each dim = 5044

创建模型、优化器、损失函数

xutorch.misc.set\_seeds(42)  
  
device = xutorch.misc.get\_device()  
  
*# model = xutorch.model.EmbeddingMLP(input\_dim=train\_dataset.dim, hidden\_dim=8192, output\_dim=2).to(device)*model = xutorch.model.MLP(input\_dim = args['num\_sentence\_word'], hidden\_dim1=128, hidden\_dim2=256, output\_dim=2).to(device)  
  
*# 可读取模型权重接续训练  
# state\_dict = torch.load(args['save\_path'])  
# model.load\_state\_dict(state\_dict)  
  
# 创建一个CE损失函数*criterion = xutorch.loss.CrossEntropy()  
  
*# 创建一个Adam优化器*optimizer = xutorch.optim.SimpleAdam(model.parameters(), lr=args['lr'])

训练模型

loss\_record = {  
 'train': [],   
 'val': []  
}  
  
best\_val\_acc = 0.0  
best\_val\_acc\_epoch = 0.0  
  
early\_stop\_cnt = 0  
  
for epoch in range(args['num\_epochs']):  
 train\_acc = 0.0  
 train\_loss = 0.0  
  
 model.train()  
 for batch\_idx, (inputs, labels) in enumerate(train\_loader):  
 inputs, labels = inputs.to(device), labels.to(device)  
   
 outputs = model(inputs)  
 train\_pred = torch.sign(outputs)  
  
 loss = criterion(outputs, labels)  
  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 *# print(train\_pred.cpu(), labels.cpu())* train\_pred\_labels = torch.argmax(train\_pred, dim=1)  
 train\_acc += (train\_pred\_labels.cpu() == labels.cpu()).sum().item()  
 train\_loss += loss.item()  
 loss\_record['train'].append(train\_loss / len(train\_dataset))  
  
 val\_acc = 0.0  
 val\_loss = 0.0  
  
 model.eval()  
 for batch\_idx, (inputs, labels) in enumerate(val\_loader):  
 inputs, labels = inputs.to(device), labels.to(device)  
 with torch.no\_grad():  
 outputs = model(inputs)  
 val\_pred = torch.sign(outputs)  
  
 loss = criterion(outputs, labels)  
  
 val\_pred\_labels = torch.argmax(val\_pred, dim=1)  
 val\_acc += (val\_pred\_labels.cpu() == labels.cpu()).sum().item()  
 val\_loss += loss.item()  
 loss\_record['val'].append(val\_loss / len(val\_dataset))  
  
 if epoch == 0 or val\_acc > best\_val\_acc:  
 best\_val\_acc = val\_acc  
 best\_val\_acc\_epoch = epoch  
 print(f'Saving model (epoch = {epoch + 1 : 4d}, acc = {best\_val\_acc / len(val\_dataset) : .4f})')  
 torch.save(model.state\_dict(), args['save\_path'])  
 early\_stop\_cnt = 0  
 else:  
 early\_stop\_cnt += 1  
   
 if early\_stop\_cnt > args['early\_stop']:  
 print('EARLY STOP')  
 break  
  
 *# if epoch % 10 == 9:* print('[{:03d}/{:03d}] Train Acc: {:3.6f} Loss: {:3.6f} | Val Acc: {:3.6f} loss: {:3.6f}'.format(  
 epoch + 1, args['num\_epochs'],   
 train\_acc / len(train\_dataset), train\_loss / len(train\_dataset),   
 val\_acc / len(val\_dataset), val\_loss / len(val\_dataset)))

输出如下。

Saving model (epoch = 1, acc = 0.8320)

Saving model (epoch = 2, acc = 0.8547)

Saving model (epoch = 3, acc = 0.8733)

Saving model (epoch = 4, acc = 0.8853)

Saving model (epoch = 6, acc = 0.8900)

Saving model (epoch = 10, acc = 0.8947)

[010/200] Train Acc: 0.895692 Loss: 0.039401 | Val Acc: 0.894667 loss: 0.092232

Saving model (epoch = 11, acc = 0.9093)

Saving model (epoch = 16, acc = 0.9120)

Saving model (epoch = 20, acc = 0.9187)

[020/200] Train Acc: 0.918872 Loss: 0.014198 | Val Acc: 0.918667 loss: 0.053652

Saving model (epoch = 22, acc = 0.9220)

[030/200] Train Acc: 0.912311 Loss: 0.011373 | Val Acc: 0.911333 loss: 0.038753

Saving model (epoch = 35, acc = 0.9240)

[040/200] Train Acc: 0.908157 Loss: 0.042239 | Val Acc: 0.910000 loss: 0.063794

[050/200] Train Acc: 0.916029 Loss: 0.038269 | Val Acc: 0.908667 loss: 0.069276

Saving model (epoch = 51, acc = 0.9267)

[060/200] Train Acc: 0.920621 Loss: 0.012680 | Val Acc: 0.922667 loss: 0.018538

[070/200] Train Acc: 0.915592 Loss: 0.019676 | Val Acc: 0.910667 loss: 0.048084

[080/200] Train Acc: 0.919965 Loss: 0.009358 | Val Acc: 0.914667 loss: 0.088475

Saving model (epoch = 86, acc = 0.9300)

[090/200] Train Acc: 0.915810 Loss: 0.014332 | Val Acc: 0.917333 loss: 0.041101

[100/200] Train Acc: 0.919965 Loss: 0.014983 | Val Acc: 0.909333 loss: 0.105510

[110/200] Train Acc: 0.915154 Loss: 0.012200 | Val Acc: 0.912667 loss: 0.050361

[120/200] Train Acc: 0.917122 Loss: 0.012642 | Val Acc: 0.916667 loss: 0.055190

[130/200] Train Acc: 0.919309 Loss: 0.011532 | Val Acc: 0.906000 loss: 0.034873

Saving model (epoch = 134, acc = 0.9313)

[140/200] Train Acc: 0.920840 Loss: 0.013111 | Val Acc: 0.914000 loss: 0.067818

[150/200] Train Acc: 0.913623 Loss: 0.013278 | Val Acc: 0.912000 loss: 0.032541

[160/200] Train Acc: 0.898972 Loss: 0.024381 | Val Acc: 0.896667 loss: 0.048209

[170/200] Train Acc: 0.888038 Loss: 0.023985 | Val Acc: 0.891333 loss: 0.043248

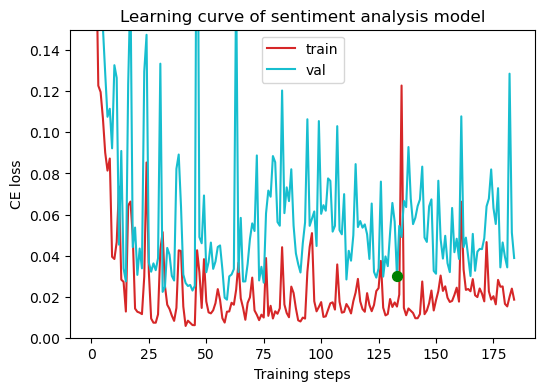
[180/200] Train Acc: 0.900503 Loss: 0.024974 | Val Acc: 0.902000 loss: 0.046386

\*\*EARLY STOP\*\*

通过最后保存的Acc可以看出，其略低于同随机种子下的基于sklearn的实现。因为数据和模型部分进行了许多取舍，但是0.931的准确率也十分高了。

绘制学习曲线

xutorch.plot.plot\_learning\_curve(loss\_record, title='sentiment analysis model', loss\_name='CE', bottom=0.0, top=0.15, min\_loss\_x=best\_val\_acc\_epoch, min\_loss\_y=loss\_record['val'][best\_val\_acc\_epoch])



## 基于Pytorch实现XuTorch

这里介绍**本次任务所用到的**XuTorch中的机器学习算法的实现过程。

XuTorch

│ \_\_init\_\_.py

│

├─dataset

│ │ boston.py

│ │ breast\_cancer.py

│ │ **prep\_dataloder.py**

│ │ **sentiment.py**

│ │ \_\_init\_\_.py

│ │

│ └─data

│ boston.txt

│ **huangguo\_mountain.csv**

│ wdbc.data

│

├─loss

│ **cross\_entropy.py**

│ hinge.py

│ mean\_squared\_error.py

│ \_\_init\_\_.py

│

├─misc

│ **get\_device.py**

│ **r2\_score.py**

│ **set\_seeds.py**

│ \_\_init\_\_.py

│

├─model

│ **embedding\_mlp.py**

│ linear.py

│ **mlp.py**

│ svm.py

│ \_\_init\_\_.py

│

├─optim

│ **adam.py**

│ sgd.py

│ \_\_init\_\_.py

│

└─plot

**plot.py**

\_\_init\_\_.py

### xutorch.dataset.prep\_dataloader

这里定义了一个数据加载函数prep\_dataloader，其目的是根据数据集名称生成相应数据集对象并将其放入数据加载器中。

from xutorch.dataset.boston import BOSTON  
from xutorch.dataset.breast\_cancer import BreastCancer  
from xutorch.dataset.sentiment import Sentiment  
from torch.utils.data import DataLoader  
  
  
def prep\_dataloader(dataset\_name, download, train, batch\_size, num\_workers=0, test\_ratio=0.25, \*\*kwargs):  
 dataset = None  
 if dataset\_name == 'BOSTON':  
 dataset = BOSTON(download=download, train=train, test\_ratio=test\_ratio, \*\*kwargs)  
 elif dataset\_name == 'BreastCancer':  
 dataset = BreastCancer(download=download, train=train, test\_ratio=test\_ratio, \*\*kwargs)  
 elif dataset\_name == 'Sentiment':  
 dataset = Sentiment(download=download, train=train, test\_ratio=test\_ratio, \*\*kwargs)  
  
 dataloader = DataLoader(dataset, batch\_size, shuffle=train, drop\_last=False, num\_workers=num\_workers, pin\_memory=True)  
 return dataloader, dataset

### xutorch.dataset.Sentiment

这里实现了一个基于特征的文本处理类 Sentiment，用于处理文本数据进行情感分析任务。

\_\_init\_\_方法的参数含义如下：

download：是否下载数据集，默认为 False；

path：数据集的路径，默认为 data/huangguo\_mountain.csv；

train：是否用于训练集，默认为 True；

test\_ratio：测试集占总数据集的比例，默认为 0.25；

min\_df：单词至少出现的次数，默认为 5；

num\_sentence\_word：每个句子包含的最大单词数量，默认为 100。

在\_\_init\_\_方法中，首先读取数据集，并删除其中的空值。然后将数据集的内容和标签分别存储到data和labels中。接下来，读取停用词表stopwords.txt，并将数据集中的单词进行统计，去除停用词后，对单词进行统计，并存储到word\_counts字典中。

然后根据min\_df参数筛选出单词数大于等于min\_df的单词，并将其与对应的索引存储到word\_to\_idx字典中。最后在字典中加入了一个<PAD>单词，其对应的索引为0。

根据train和test\_ratio参数，选出相应的数据集作为当前数据集，并将其长度和字典大小存储到dim中。最后输出数据集的基本信息，如数据集类型、数据集大小和字典大小。

因为NLP任务的数据量大，最好不要一次性地对全集进行预处理，而是在通过\_\_getitem\_\_取出时对每一个样本单独进行预处理。

在\_\_getitem\_\_方法中，根据给定索引获取相应的文本内容和标签，并将文本内容转换成 PyTorch 中的FloatTensor类型返回。在转换过程中，先根据单词索引进行转换，然后进行多裁少补，若单词数量大于 num\_sentence\_word，则只取前 num\_sentence\_word 个单词，若单词数量小于 num\_sentence\_word，则在末尾加上若干个 0，使其长度达到 num\_sentence\_word。

import os  
import torch  
from torch.utils.data import Dataset  
import pandas as pd  
  
  
*# 基于特征的文本处理方法*class Sentiment(Dataset):  
 data\_url = None  
 current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  
  
 def \_\_init\_\_(self, download=False, path=os.path.join(current\_dir, 'data', 'huangguo\_mountain.csv'), train=True, test\_ratio=0.25, min\_df=5, num\_sentence\_word = 100, stopword\_path=os.path.join(current\_dir, 'data', 'stopwords.txt')):  
 if download:  
 path = self.data\_url  
  
 df = pd.read\_csv(path, encoding='UTF-8')  
 df = df.dropna()  
  
 data = df['content']  
 labels = df['label']  
  
 with open(stopword\_path, 'r', encoding='utf-8') as f:  
 stop\_words = [stop\_word.strip() for stop\_word in f.readlines()]  
  
 self.word\_counts = {}  
 for sentence in data:  
 words = sentence.split()  
 for word in words:  
 if word not in stop\_words and len(word) > 1:  
 self.word\_counts[word.lower()] = self.word\_counts.get(word.lower(), 0) + 1  
  
 self.num\_sentence\_word = num\_sentence\_word  
  
 self.word\_to\_idx = {word: idx + 1 for idx, (word, count) in enumerate(self.word\_counts.items()) if count >= min\_df}  
 self.word\_to\_idx['<PAD>'] = 0  
  
 pivot = int(100 \* (1 - test\_ratio))  
 if train:  
 indices = [i for i in range(len(data)) if i % 100 < pivot]  
 else:  
 indices = [i for i in range(len(data)) if i % 100 >= pivot]  
  
 self.data = data.iloc[indices].values  
 self.labels= labels.iloc[indices].values  
  
 self.dim = len(self.word\_to\_idx)  
  
 print('Finished reading the {} set of Text Dataset ({} samples found), each dim = {}'.format(  
 'train' if train else 'val', len(self.data), self.dim))  
  
 def \_\_len\_\_(self):  
 return len(self.data)  
  
 *# NLP任务数据量大，取出时再进行处理* def \_\_getitem\_\_(self, index):  
 content = self.data[index]  
 label = int(self.labels[index])  
 return self.convert\_to\_tensor(content), torch.tensor(label)  
  
 def convert\_to\_tensor(self, sentence):  
 content\_idx = [self.word\_to\_idx[word.lower()] for word in sentence.split() if word.lower() in self.word\_to\_idx]  
 if len(content\_idx) >= self.num\_sentence\_word:  
 content\_idx = content\_idx[:self.num\_sentence\_word]  
 else:  
 content\_idx[len(content\_idx):self.num\_sentence\_word] = [0] \* (self.num\_sentence\_word - len(content\_idx))  
 return torch.FloatTensor(content\_idx)  
 *# return torch.LongTensor(content\_idx) # EmbeddingMLP*

### xutorch.misc.set\_seeds

固定随机种子以复现相同结果。

import os  
import torch  
import random  
import numpy as np  
  
  
def set\_seeds(seed):  
 torch.manual\_seed(seed)  
 os.environ['PYTHONHASHSEED'] = str(seed)  
 random.seed(seed)  
 np.random.seed(seed)  
 if torch.cuda.is\_available():  
 torch.cuda.manual\_seed(seed)  
 torch.cuda.manual\_seed\_all(seed)  
 *# if torch.backends.cudnn.is\_available():  
 # torch.backends.cudnn.benchmark = False  
 # torch.backends.cudnn.deterministic = True*

### xutorch.misc.get\_device

检查计算设备：是否有可用的GPU，如果有则使用GPU，否则使用CPU。

import torch  
  
  
def get\_device():  
 return torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

### xutorch.model.MLP

这里定义了MLP类，与线性模型并没有太大区别，只是在层间加入了ReLU作为激活函数。

import torch.nn as nn  
  
  
class MLP(nn.Module):  
 def \_\_init\_\_(self, input\_dim, hidden\_dim1, hidden\_dim2, output\_dim):  
 super(MLP, self).\_\_init\_\_()  
 self.fc1 = nn.Linear(input\_dim, hidden\_dim1)  
 self.relu1 = nn.ReLU()  
 self.fc2 = nn.Linear(hidden\_dim1, hidden\_dim2)  
 self.relu2 = nn.ReLU()  
 self.fc3 = nn.Linear(hidden\_dim2, output\_dim)  
  
 def forward(self, x):  
 x = self.fc1(x)  
 x = self.relu1(x)  
 x = self.fc2(x)  
 x = self.relu2(x)  
 x = self.fc3(x)  
 return x.squeeze(1)

### xutorch.model.EmbeddingMLP

这里定义了EmbeddingMLP类，其本质是一个将输入层从Linear换为了Embedding的单隐层网络。

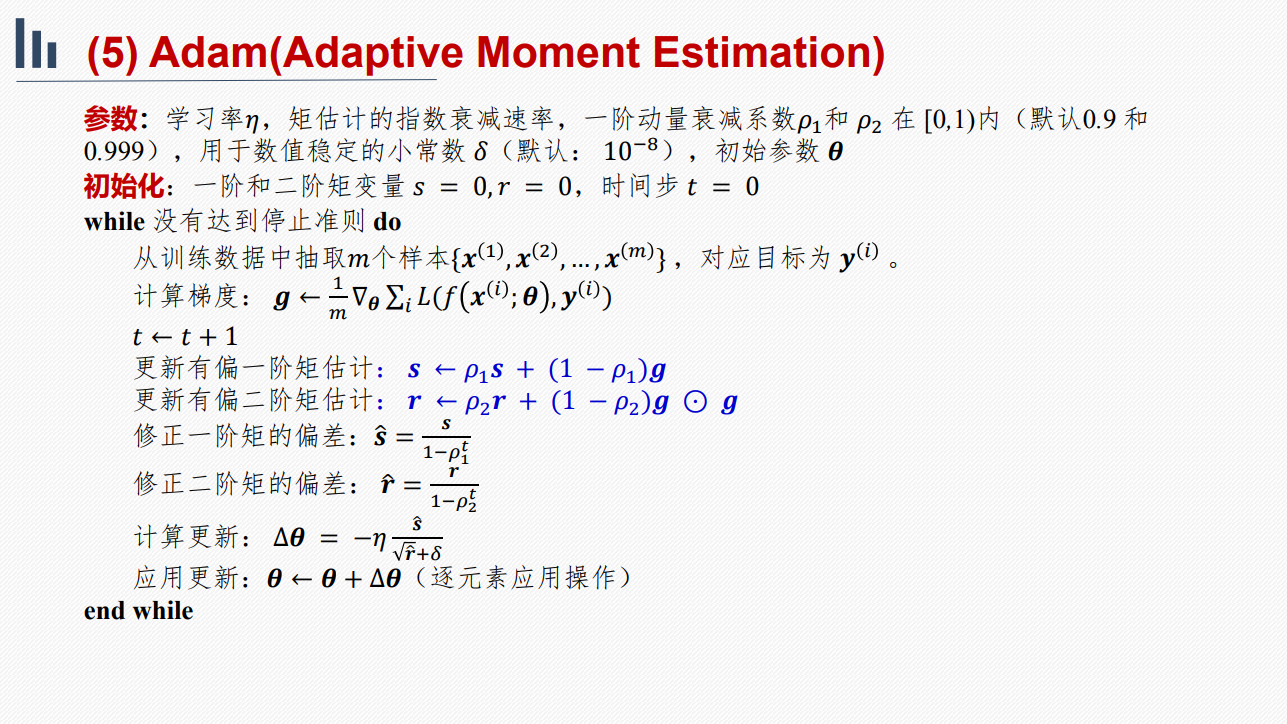
import torch.nn as nn  
  
  
class EmbeddingMLP(nn.Module):  
 def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):  
 super(EmbeddingMLP, self).\_\_init\_\_()  
 self.fc1 = nn.Embedding(input\_dim, hidden\_dim)  
 self.relu1 = nn.ReLU()  
 self.fc2 = nn.Linear(hidden\_dim, output\_dim)  
  
 def forward(self, x):  
 x = self.fc1(x)  
 x = self.relu1(x)  
 x = self.fc2(x)  
 return x.squeeze(1)

### xutorch.optim.SimpleAdam

这里实现了简单的随机梯度下降（SGD）算法。为了省略一些不重要的操作（如zero\_grad和另存参数），我令该类继承自PyTorch的Optimizer类。

构造函数\_\_init\_\_中，接收五个参数：params表示要更新的参数组成的列表，lr是学习率（默认为0.01），betas是指数衰减率（默认为(0.9, 0.999)），eps是小常数（默认为1e-8）, weight\_decay是正则化系数（默认为0）。

类中的step方法是该优化器的核心方法，用于更新参数，**参考自课程PPT中的Chap3 P68 Adam**。



在这个方法中，对于每个参数param，如果其梯度为None，则跳过此参数。否则，根据当前的学习率、动量参数beta1和beta2以及梯度grad，更新其对应的状态state，包括步数step、一次动量的指数平均值exp\_avg和二次动量的指数平均值exp\_avg\_sq。然后，利用偏差修正项对指数平均值进行修正，计算步长step\_size，并更新参数的值。如果设置了weight\_decay，则还需要对参数进行权值衰减。注意，如果不在step内部清空梯度（p.data=None），则一定要在代码外部显式地书写optimizer.zero\_grad()

需要注意的是，对于原地方法buf.mul\_和p.data.add\_中的alpha参数，它们的含义分别是乘数和加数的系数，用于对乘积或加和进行缩放。

import torch  
import torch.optim as optim  
  
  
*# extend from torch.optim.Optimizer, 'zero\_grad' method can be omitted*class SimpleAdam(optim.Optimizer):  
 def \_\_init\_\_(self, params, lr=0.001, betas=(0.9, 0.999), eps=1e-8, weight\_decay=0):  
 defaults = dict(lr=lr, betas=betas, eps=eps, weight\_decay=weight\_decay)  
 super(SimpleAdam, self).\_\_init\_\_(params, defaults)  
  
 for group in self.param\_groups:  
 for param in group['params']:  
 state = self.state[param]  
 state['step'] = 0  
 state['exp\_avg'] = torch.zeros\_like(param.data)  
 state['exp\_avg\_sq'] = torch.zeros\_like(param.data)  
  
 def step(self):  
 for group in self.param\_groups:  
 for param in group['params']: *# 这个params其实是weights而不是模型的超参数* if param.grad is None:  
 continue  
 grad = param.grad.data  
  
 state = self.state[param]  
 state['step'] += 1  
  
 beta1, beta2 = group['betas']  
 exp\_avg = state['exp\_avg']  
 exp\_avg\_sq = state['exp\_avg\_sq']  
  
 exp\_avg.mul\_(beta1).add\_(grad, alpha=1 - beta1)  
 exp\_avg\_sq.mul\_(beta2).addcmul\_(grad, grad, value=1 - beta2)  
  
 bias\_correction1 = 1 - beta1 \*\* state['step']  
 bias\_correction2 = 1 - beta2 \*\* state['step']  
  
 step\_size = group['lr'] \* (bias\_correction2 \*\* 0.5) / bias\_correction1  
   
 if group['weight\_decay'] != 0:  
 param.data.add\_(-group['lr'] \* group['weight\_decay'], param.data)  
  
 param.data.add\_(exp\_avg / (exp\_avg\_sq.sqrt() + group['eps']), alpha=-step\_size)

### xutorch.loss.CE

该类用于计算交叉熵（CrossEntropy）损失函数。

forward方法实现了接受两个tensor作为输入参数outputs和targets，首先使用nn.LogSoftmax对网络的输出结果进行log softmax操作，接着使用nn.NLLLoss计算log softmax的负对数似然损失，并将其返回。

import torch.nn as nn  
  
  
class CrossEntropy(nn.Module):  
 def \_\_init\_\_(self):  
 super(CrossEntropy, self).\_\_init\_\_()  
  
 def forward(self, outputs, targets):  
 log\_softmax = nn.LogSoftmax(dim=1)  
 log\_probs = log\_softmax(outputs)  
 loss = nn.NLLLoss()(log\_probs, targets)  
 return loss

### xutorch.plot.\*

这里包含了三个用于绘图的函数，其中：

plot\_gt\_and\_pred 用于绘制预测值与真实值的散点图（预测-真实值散点图）；

plot\_gt\_vs\_pred 用于绘制真实值与预测值的曲线图（拟合曲线）；

plot\_learning\_curve 用于绘制训练曲线。

import torch  
import matplotlib.pyplot as plt  
from matplotlib.pyplot import figure  
  
  
def plot\_gt\_and\_pred(targets=None, preds=None, bottom=0.0, top=35.0):  
 import os  
 os.environ["KMP\_DUPLICATE\_LIB\_OK"] = "TRUE"  
  
 preds = torch.cat(preds, dim=0).numpy()  
 targets = torch.cat(targets, dim=0).numpy()  
  
 figure(figsize=(5, 5))  
 plt.scatter(targets, preds, c='r', alpha=0.5)  
 plt.plot([bottom, top], [bottom, top], c='b')  
 plt.xlim(bottom, top)  
 plt.ylim(bottom, top)  
 plt.xlabel('ground truth value')  
 plt.ylabel('predicted value')  
 plt.title('Ground Truth v.s. Prediction')  
 plt.show()  
  
  
def plot\_gt\_vs\_pred(y\_true, y\_pred):  
 y\_pred = torch.cat(y\_pred, dim=0).numpy()  
 y\_true = torch.cat(y\_true, dim=0).numpy()  
 plt.plot(y\_true, label="gt")  
 plt.plot(y\_pred, label="predict")  
 plt.xlabel("index")  
 plt.ylabel("price")  
 plt.legend(loc="best")  
 plt.title("gt v.s. predict")  
 plt.show()  
  
  
def plot\_learning\_curve(loss\_record, loss\_name='', title='', bottom=0.0, top=100.0, min\_loss\_x=None, min\_loss\_y=None):  
 x\_1 = range(len(loss\_record['train']))  
 x\_2 = x\_1[::len(loss\_record['train']) // len(loss\_record['val'])]  
 figure(figsize=(6, 4))  
 plt.plot(x\_1, loss\_record['train'], c='tab:red', label='train')  
 plt.plot(x\_2, loss\_record['val'], c='tab:cyan', label='val')  
 if min\_loss\_x and min\_loss\_y:  
 plt.scatter(min\_loss\_x, min\_loss\_y, c='g', alpha=1, marker='o', s=50, zorder=10)  
 plt.ylim(bottom, top)  
 plt.xlabel('Training steps')  
 plt.ylabel('{} loss'.format(loss\_name))  
 plt.title('Learning curve of {}'.format(title))  
 plt.legend()  
 plt.show()