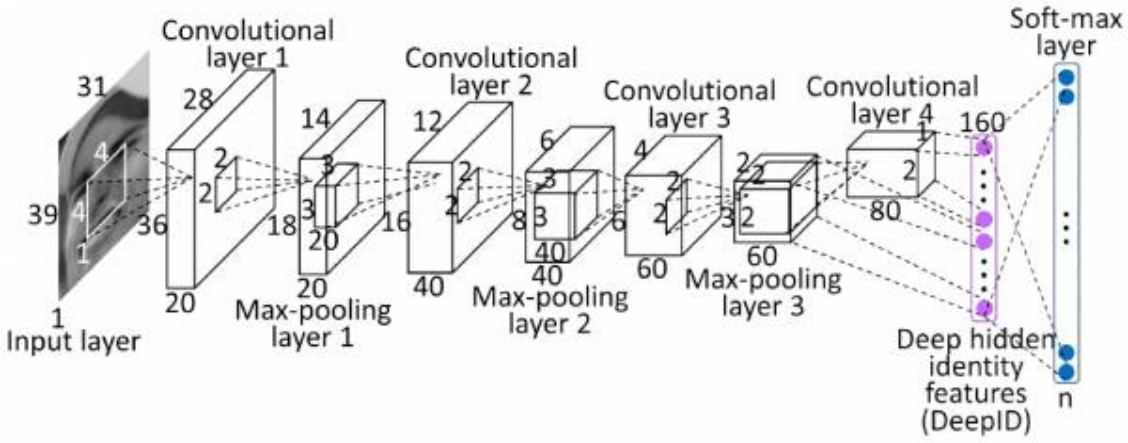
1. Pytorch环境配置

anaconda:python3.6+pytorch1.6+cuda10.1

2. CNN简述

- **大致架构**：卷积层【提取特征】+池化层【降维，减少运算，避免过拟合】+全连接层【分类】



- **常见运用**：CNN适合用于具有局部空间相关性的数据，经常用于图像的特征提取和分类等
- **CNN用于时间序列异常检测**：其实并不是很合适，CNN进行空间扩展，输入和输出都是静态的，没有划分独立文本的能力。因为时间序列中序列的时间性也体现在位置上，所以可以用CNN。做时间序列异常检测时经常将CNN与RNN结合使用，处理图像时间序列。

3. CNN用于时间序列异常检测模型详解

3.1 整体框架

3.1.1 数据集

- 由于CNN没有划分独立文本的能力，所以CNN的数据输入是一个个长度为3653的数字序列

```
[-0.          0.00955149  0.03315829 ...  5.49407832  5.51484617
  5.50371358]
3653
```

- CNN模型的判断是针对这样一串序列判断其是否异常，而非对单个数据值进行判断(利用序列空间上的特征)
- 导入数据代码(完整代码见后)：

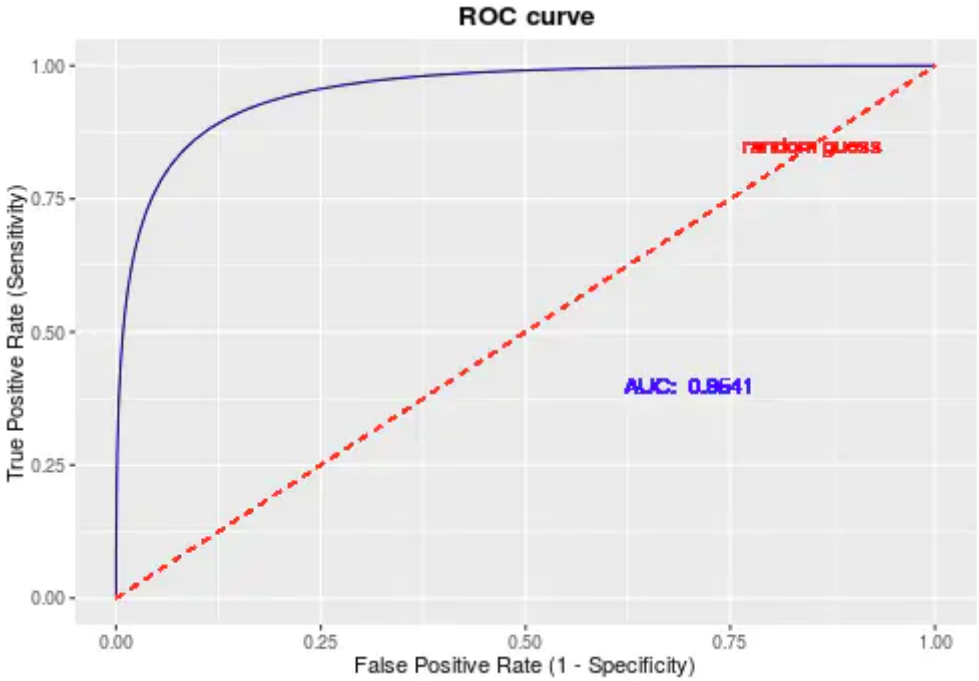
```
#具体数据集输入
dataset_train = TimeSeriesFolder(root='dataset/train/', loader=npy_loader,
transform=transformation, extensions='.npy')
dataset_val = TimeSeriesFolder(root='dataset/val/', loader=npy_loader,
extensions='.npy')
```

3.1.2 CNN模型的构建和训练

- 构建的是一个二分类分类器，经过了两层卷积池化和三层全连接。
- 模型训练过程中需要进行反向传播和更新模型参数，然后计算每一次训练完成后的准确度并显示训练结果。

3.1.3 CNN模型的验证

- 验证数据输入到模型之中，但是仅仅通过模型的计算给出一个判断结果，不改变模型的参数（即不进行反向传播和梯度计算）等。
- 根据模型的判断结果和实际值的差别，可以得出模型在验证集上的准确度。
- 本模型采用了ROC曲线可视化模型的性能。
    - **ROC曲线**：ROC的全称是Receiver Operating Characteristic Curve，中文名字叫"受试者工作特征曲线"，图样示例如下，



该曲线的横坐标为假阳性率（False Positive Rate, FPR），N是真实负样本的个数，FP是N个负样本中被分类器预测为正样本的个数。

纵坐标为真阳性率（True Positive Rate, TPR），

$$TPR = \frac{TP}{P}$$

P是真实正样本的个数，TP是P个正样本中被分类器预测为正样本的个数。

通过曲线可以对模型有一个定性的分析，如果要对模型进行量化的分析，此时需要引入一个新的概念，就是AUC（Area under roc Curve）面积，即曲线沿着横轴积分的值。

3.2 完整代码及注释

```
# -*- coding: utf-8 -*-
"""
CNN上的时间序列分类器，通过监管的方式训练得到将数据分为正常和异常两类的CNN分类器
从而可以在新数据出现时判断其正常与否
判断结果为一个标签（正常或异常）
"""
```

```
import numpy as np
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torch.utils.tensorboard import SummaryWriter
from torchvision import datasets, transforms, utils
from sklearn.metrics import confusion_matrix, classification_report, roc_curve,
auc
import matplotlib.pyplot as plt


##数据集创建
#创建数据集类以从文件夹加载
#DatasetFolder假设所有的文件按文件夹保存好，每个文件夹下面存贮同一类别的数据，文件夹的名字为分
类的名字
class TimeSeriesFolder(datasets.DatasetFolder):
    def __getitem__(self, index):
        path, target = self.samples[index]
        sample = self.loader(path)
        if self.transform is not None:        #若对数据的变换函数不为空，则变换
            sample = self.transform(sample)
        return sample, target

#npy文件加载器
def npy_loader(path):
    #因为存在多个尺寸为（1，3564）的系列?
    data = np.load(path)[0:3653]
    #最小最大缩放
    normalized = (data - np.amin(data)) / (np.amax(data) - np.amin(data))
    sample = torch.from_numpy(normalized) #从numpy数组创建一个张量（tensor和numpy区
别）
    return sample

#增加噪音以提高模型性能
transformation  = transforms.Compose([transforms.Lambda(lambda x: x +
(torch.rand(x.shape)/3).double())])

#具体数据集输入
dataset_train = TimeSeriesFolder(root='dataset/train/', loader=npy_loader,
transform=transformation, extensions='.npy')
dataset_val = TimeSeriesFolder(root='dataset/val/', loader=npy_loader,
extensions='.npy')

#为WeightedRandomSampler准备权重系数的函数
"""
大概思想是根据两类数据的量的多少为其分配权重
先得到一类数据的权重
再下分到每一数据
"""
def make_weights_for_balanced_classes(examples, nclasses):

    count = [0] * nclasses

    for item in examples:

        count[item[1]] += 1
    weight_per_class = [0.] * nclasses         #[0.]是什么意思?
```

```
    n = float(sum(count))

    for i in range(nclasses):
        weight_per_class[i] = n/float(count[i])
    weight = [0] * len(examples)

    for idx, val in enumerate(examples):

        weight[idx] = weight_per_class[val[1]]

    return weight

print('训练样本:', dataset_train, '\n')
print('验证样本:', dataset_val, '\n')

#训练数据集的权重
weights_train = make_weights_for_balanced_classes(dataset_train.samples,
len(dataset_train.classes))
#WeightedRandomSampler是加权随机采样器
"""
由于我们不能将大量数据一次性放入网络中进行训练，所以需要分批进行数据读取
这一过程涉及到如何从数据集中读取数据，采样器应运而生
常见的有：随机采样、顺序采样等
"""
sampler_train = torch.utils.data.sampler.WeightedRandomSampler(weights_train,
len(weights_train))

#验证数据集的权重
weights_val = make_weights_for_balanced_classes(dataset_val.samples,
len(dataset_val.classes))
sampler_val = torch.utils.data.sampler.WeightedRandomSampler(weights_val,
len(weights_val))

batch_size_train = 10  #每一批次训练的数据量
batch_size_val = 525

#训练数据生成器
train_loader = DataLoader(dataset_train, batch_size=batch_size_train,
sampler=sampler_train)
val_loader = DataLoader(dataset_val, batch_size=batch_size_val,
sampler=sampler_val)


##CNN模型的构建
class CNN_classifier(nn.Module):
    def __init__(self):
        super().__init__()

        """
        nn.Conv1d就是定义一层卷积层，原数据是一维的，所以是Conv1d
        卷积层nn.Conv1d(1, 64, 6)
        第一个参数值1，表示输入一个一维数组；
        第二个参数值64，表示提取64个特征，得到64个feature map
        第三个参数值6，表示卷积核是一个6*6的矩阵
        """
        self.layer1 = nn.Sequential(nn.Conv1d(1, 64, kernel_size=6),
                                    nn.ReLU(True),
                                    nn.MaxPool1d(kernel_size=6, stride=2))
```

```
        self.layer2 = nn.Sequential(nn.Conv1d(64, 32, kernel_size=6),
                                    nn.ReLU(True),
                                    nn.MaxPool1d(kernel_size=6, stride=2))

        # Расчет входных признаков по формуле Lout=((Lin+2*pading - dilation*
(kernel - 1) - 1)/stride) + 1
        self.classifier = nn.Sequential(nn.Linear(in_features=906*32,
out_features=512, bias=True),
                                        nn.ReLU(True),
                                        nn.Dropout(),
                                        nn.Linear(in_features=512,
out_features=512, bias=True),
                                        nn.ReLU(True),
                                        nn.Dropout(),
                                        nn.Linear(512, 2))
    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.classifier(out)
        return out

model = CNN_classifier()
model = model.float()      #训练过程中张量类型为float


##CNN模型的训练
num_epochs = 3                #迭代次数
num_classes = 2               #分类类别数
learning_rate = 0.001         #学习率

criterion = nn.CrossEntropyLoss()  #交叉熵损失函数，做分类很有用
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)   #优化器，改变
模型参数

total_step = len(dataset_train)
loss_list = []
acc_list = []

#可视化实例
writer = SummaryWriter()
#tensorboard迭代?
tf_iter = 0

tcorrect = 0    # 定义预测正确的数据数，初始化为0
ttotal = 0      # 总共参与训练的数据数，也初始化为0

#每一次迭代
for epoch in range(num_epochs):
    for i, (series, labels) in enumerate(train_loader):
        series = series[:,None,:]   #?不是很理解

        #开始训练
        outputs = model(series.float())
        loss = criterion(outputs, labels)
        loss_list.append(loss.item())
```

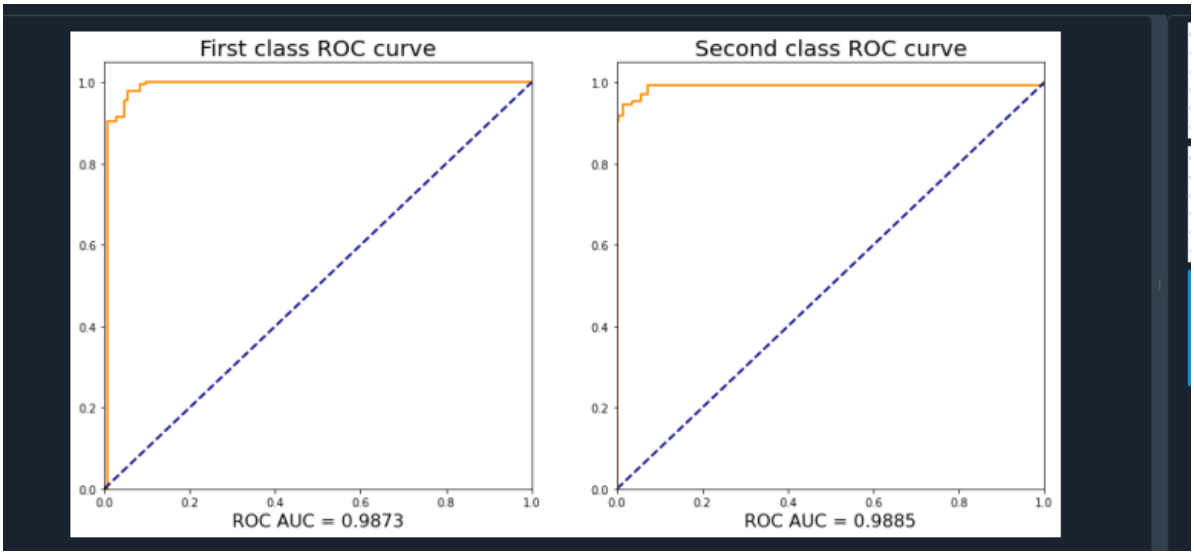
```
        #反向传播和优化器
        optimizer.zero_grad()  #清空过往梯度
        loss.backward()        #反向传播，计算当前梯度
        optimizer.step()       #根据梯度更新网络参数

        #准确度计算
        ttotal += labels.size(0)
        """
         _, predicted = torch.max(outputs.data, 1)这条语句中
         _, predicted表示函数有两个返回值而第一个值我们不关心
         torch.max函数返回outputs.data中的最大值及其索引，我们只需要得到索引就好了，所以第一个参数不在意
         torch.max函数的第二个参数为1，表示从每一行中找最大值，为0表示从每一列中找最大值
        """
        _, predicted = torch.max(outputs.data, 1)
        tcorrect += (predicted == labels).sum().item()
        acc_list.append(tcorrect / ttotal)
        accuracy = (tcorrect / ttotal) * 100

        #结果存到tensorboard
        writer.add_scalar('Accuracy/train',  accuracy,  tf_iter)
        writer.add_scalar('Loss/train',  loss.item(),  tf_iter)
        tf_iter += 1

        #显示结果到console
        if (i + 1) % 30 == 0:      #每到每次迭代下训练的数据量达到500的倍数时显示一次
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Accuracy: {:.2f}%'
                  .format(epoch + 1, num_epochs, (i + 1)*batch_size_train,
                  total_step, loss.item(),
                  accuracy))

print("----------------The model is trained!-------------------- ")


##验证集
actuals_0_class, actuals_1_class = [], []
probabilities_0_class, probabilities_1_class  = [], []
probabilities_0_class_list, probabilities_1_class_list,   = [], []


"""
model.eval和with torch.no_grad都是做测试的时候不可少的两函数
model.eval使得输入的测试数据不会改变模型的权值且层中的dropout函数失效
no_grad中的过程不需要计算梯度也不需要反向传播【不构建计算图】，只是通过模型的计算得到一个结果
"""
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for series, labels in val_loader:
        #这些步骤和前面训练集一样
        series = series[:,None,:]
        outputs = model(series.float())
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        actuals_0_class.extend(labels.view_as(predicted) == 0)
```

```
            actuals_1_class.extend(labels.view_as(predicted) == 1)

            #每一类的概率
            probabilities_0_class.extend(np.exp(outputs[:, 0]))
            probabilities_1_class.extend(np.exp(outputs[:, 1]))

            actuals_0_class_list = [i.item() for i in actuals_0_class]
            actuals_1_class_list = [i.item() for i in actuals_1_class]

            probabilities_0_class_list = [i.item() for i in probabilities_0_class]
            probabilities_1_class_list = [i.item() for i in probabilities_1_class]

            #结果存到tensorboard
            writer.add_scalar('Accuracy/validation',  (correct / total) * 100)
        #显示结果到console
        print('验证集准确率: {:.2f} '.format((correct / total) * 100))

#ROC曲线
fpr_0, tpr_0, _0 = roc_curve(actuals_0_class_list, probabilities_0_class_list)
fpr_1, tpr_1, _1 = roc_curve(actuals_1_class_list, probabilities_1_class_list)

#AUC面积
roc_auc_0 = auc(fpr_0, tpr_0)
roc_auc_1 = auc(fpr_1, tpr_1)

#ROC曲线绘制
fig, axes = plt.subplots(1,2, figsize=(15,7))
lw = 2

axes[0].set_title('First class ROC curve', fontsize=20)
axes[0].set_xlim([0.0, 1.0])
axes[0].set_ylim([0.0, 1.05])
axes[0].set_xlabel('ROC AUC = %0.4f' % roc_auc_0, fontdict={'size': 16})
axes[0].plot(fpr_0, tpr_0, color='darkorange', lw=lw)
axes[0].plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')


axes[1].set_title('Second class ROC curve', fontsize=20)
axes[1].set_xlim([0.0, 1.0])
axes[1].set_ylim([0.0, 1.05])
axes[1].set_xlabel('ROC AUC = %0.4f' % roc_auc_1, fontdict={'size': 16})
axes[1].plot(fpr_1, tpr_1,  color='darkorange', lw=lw)
axes[1].plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')

MODEL_STORE_PATH = "saved_model/"
torch.save(model.state_dict(), MODEL_STORE_PATH +
'time_series_CNN_binary_classifier.ckpt')
```

3.3 运行结果截图

```
训练样本: Dataset TimeSeriesFolder
    Number of datapoints: 1531
    Root location: dataset/train/
    StandardTransform
Transform: Compose(
               Lambda()
           )

验证样本: Dataset TimeSeriesFolder
    Number of datapoints: 525
    Root location: dataset/val/
```

```
Epoch [1/3], Step [300/1531], Loss: 0.1667, Accuracy: 78.00%
Epoch [1/3], Step [600/1531], Loss: 0.0036, Accuracy: 85.67%
Epoch [1/3], Step [900/1531], Loss: 0.1362, Accuracy: 88.44%
Epoch [1/3], Step [1200/1531], Loss: 0.0212, Accuracy: 90.67%
Epoch [1/3], Step [1500/1531], Loss: 0.0019, Accuracy: 92.13%
Epoch [2/3], Step [300/1531], Loss: 0.0022, Accuracy: 93.45%
Epoch [2/3], Step [600/1531], Loss: 0.0001, Accuracy: 93.99%
Epoch [2/3], Step [900/1531], Loss: 0.0045, Accuracy: 94.49%
Epoch [2/3], Step [1200/1531], Loss: 0.0092, Accuracy: 94.87%
Epoch [2/3], Step [1500/1531], Loss: 0.0176, Accuracy: 95.05%
Epoch [3/3], Step [300/1531], Loss: 0.1323, Accuracy: 95.36%
Epoch [3/3], Step [600/1531], Loss: 0.0025, Accuracy: 95.63%
Epoch [3/3], Step [900/1531], Loss: 0.0789, Accuracy: 95.86%
Epoch [3/3], Step [1200/1531], Loss: 0.0009, Accuracy: 96.15%
Epoch [3/3], Step [1500/1531], Loss: 0.0000, Accuracy: 96.38%
----------------The model is trained!--------------------
验证集准确率: 94.67
```



3.3 反思

CNN做时间序列异常检测有诸多不便之处，本模型实际上是将时间序列的时间属性转化为空间属性再经由CNN进行分类，模型的这种异常检测方式只能检测出某段序列里有异常而无法指明具体的异常数据。CNN做异常检测对输入和输出均有限制，无法实时、动态地检测异常，后续考虑采用其他模型继续研究。

4. 时间序列异常检测数据集

- Alibaba/clusterdata
- Azure/AzurePublicDataset
- Google/cluster-data
- The Numenta Anomaly Benchmark(NAB)
- Yahoo: A Labeled Anomaly Detection Dataset
- 港中文loghub数据集