实验四、声纹识别系统实现及验证

一、 实验目的

本次实验利用公开数据库实现声纹识别系统并进行验证。该实验旨在贯通课程学习中声纹识别的相关内容。通过实现前端特征提取、深度神经网络、池化层、损失函数等必要步骤，掌握声纹识别的整体流程，并了解部分优化方法。

二、 实验平台及编程环境

python3，pytorch，librosa，numpy，pandas

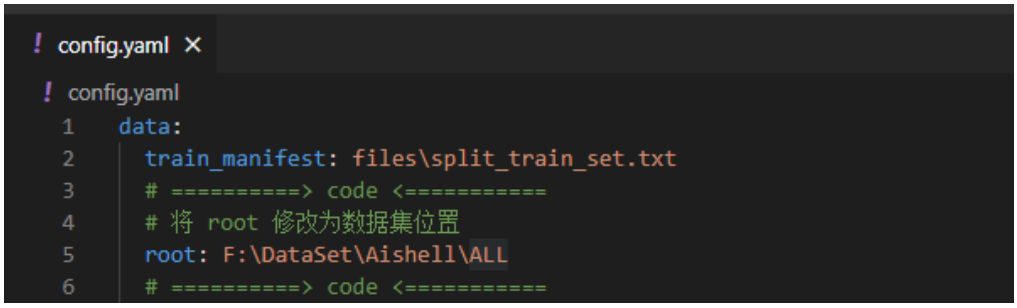
三、 实验内容（标注“*”内容为选做实验）：

1、数据准备：

实践内容基于中文语料库 AiShell（https://www.openslr.org/33/）实现。该数据集包含 340 位说话人的训练数据，40 位说话人的开发数据，以及 20 位说话人的测试数据。本次实验共选取 300 位说话人，每位说话人 10 条语音作为训练数据；20 位说话人，每位说话人 10 条语音作为测试数据。

1.1 数据配置

将 config.yaml 中的 root 选项修改为具体的数据集位置。



2、实现基于 x-vector 的声纹识别系统（2 学时）：

补全 xvector.py、pooling.py、loss.py 中的缺失代码，使其可以正常运行；以课程中所学习的声纹识别训练过程为依据，调用必要函数，训练基本的声纹识

别系统。（第一部分实验采用前端特征：mfcc，深度神经网络：TDNN，池化层：global average pooling，损失：softmax）

2.1 全局平均池化层（GlobalAveragePooling）的实现

根据 **pooling.py** 中 **GlobalAveragePooling** 类的定义，以及提示的 forward 函数的输入输出维度，补全 **forward** 函数。

pooling.py

```
25
26  class GlobalAveragePooling(torch.nn.Module):
27      def __init__(self):
28          super(GlobalAveragePooling, self).__init__()
29          self.pooling = nn.AdaptiveAvgPool1d(1)
30
31      # ==========> code <==========
32      def forward(self, x):
33          '''
34          input: (64, 1500, 86)
35          output: (64, 1500)
36          '''
37          pass
38      # ==========> code <==========
```

2.2 x-vector 中深度神经网络的实现

在 **xvector.py** 文件中，已经基本完成了 Xvector 各层的定义，但仍然缺少池化层以及最终的线性层。请参考各层的定义代码，为模型添加池化层和最终的线性层，补全__init__函数。

xvector.py

```
59                      activation(),
60                      nn.BatchNorm1d(out_channels),
61                  ]
62              )
63              in_channels = tdnn_channels[block_index]
64
65          # global average pooling
66          # ==========> code <==========
67
68          # Final linear transformation
69
70          # ==========> code <==========
```

调用 Xvectorc 定义的各层，补全 Xvector 类的 **forward** 函数。

xvector.py

```
78      def forward(self, x, lens=None):
79          """Returns the x-vectors.
80          Arguments
81          ---------
82          x : torch.Tensor
83          """
84
85          # ==========> code <==========
86          pass
87          # ==========> code <==========
```

2.3 Softmax 损失函数的实现

损失函数的计算主要分为两步：1）线性层将嵌入映射到说话人标签空间上；2）采用交叉熵计算损失。请以此为依据补全 loss.py 文件中 Softmax 的 forward 函数。

loss.py 2

loss.py > Softmax > forward

```
5
6   class Softmax(nn.Module):
7       def __init__(self, embedding_size, num_classes):
8           super(Softmax, self).__init__()
9           self.embedding_size = embedding_size
10          self.num_classes = num_classes
11          self.fc = nn.Linear(embedding_size, num_classes)
12
13      def forward(self, embeddings, labels):
14          # ==========> code <===========
15
16
17          # ==========> code <===========
18          acc = self.accuracy(logits, labels)
19          return loss, acc
```

2.4 基于 x-vector 的声纹识别系统的训练

在 train.py 中完成模型、损失函数、学习器的定义。

train.py 3

train.py > main

```
62      # 定义数据的 dataloader
63      trainset = SpkTrainDataset(opts)
64      train_loader = DataLoader(trainset, batch_size = args.batchsize, shuffle = True, collate_fn = trainset.collate_fn)
65
66      # ==================> code <======================
67      # 定义模型
68
69
70      # 定义损失函数
71
72
73      # 定义学习器
74
75
76      # ==================> code <======================
```

在 train.py 中完成嵌入的提取、损失的计算、正确率的计算以及反向传递。

train.py 5

train.py > train

```
93
94              # ==================> code <======================
95
96
97
98
99
100             # ==================> code <======================
101
102             losses.update(loss.item(), feats.size(0))
103             accuracy.update(acc, feats.size(0))
```

2.5 请给出模型在训练过程中的输出截图或视频。

3、声纹识别系统的验证（1 学时）：

3.1 基于 x-vector 的声纹识别系统的验证

在 **EER.py** 中，定义并导入模型(xvector_gap_softmax_checkpoint.pth)；为测试数据生成深度嵌入。

```
42  def main():
43      testset = SpkTestDataset(opts)
44      test_loader =DataLoader(testset, batch_size = 100, shuffle = True, collate_fn = testset.collate_fn)
45
46      print('Load checkpoint')
47
48      # Load model from checkpoint
49      # ==================> code <======================
50      # ============== 定义并导入模型 ==================
51
52
53
54
55      # ==================> code <======================
56
57      model.eval()
58
59      dict_embedding = {}
60      # Enroll and test
61      for t, (data) in enumerate(test_loader):
62          feats, labels = data
63
64          feats = feats.to(device)
65          # ==================> code <======================
66          # ============ 为测试数据生成深度嵌入 ==============
67
68          # ==================> code <======================
```

在 **verification** 函数中计算注册数据嵌入和测试数据嵌入之间的余弦相似度。

EER.py 3 ×

EER.py > verification

```
88              with torch.no_grad():
89                  enroll_embedding = dict_embedding[enroll_filename].unsqueeze(0)
90                  test_embedding = dict_embedding[test_filename].unsqueeze(0)
91
92                  # ==================> code <======================
93                  # ================= 计算相似度 ====================
94
95                  # ==================> code <======================
96                  score = score.data.cpu().numpy()[0]
```

通过完成 **roc** 函数来绘制 ROC 曲线。

EER.py 3 ×

EER.py > roc

```
121     def roc(score_list, label_list):
122         # ==================> code <======================
123         # ================ 绘制roc曲线 ====================
124
125
126
127         pass
128         # ==================> code <======================
```

等错误率（Equal Error Rate EER）是声纹识别中常用的评价指标。请根据 EER.py 的相关代码 **get_eer** 函数，简述等错误率（EER）的计算流程。并在 ROC 曲线上画出 EER 的对应点。

EER.py 3 ×

EER.py > roc

```
110  def get_eer(score_list, label_list):
111      fpr, tpr, threshold = roc_curve(label_list, score_list, pos_label=1)
112      fnr = 1 - tpr
113      eer_threshold = threshold[np.nanargmin(np.absolute((fnr - fpr)))]
114      eer = fpr[np.nanargmin(np.absolute((fnr - fpr)))]
115      intersection = abs(1 - tpr - fpr)
116      print("Epoch=%d  EER= %.2f  Thres= %0.5f" % (
117      args.cp_num, 100 * fpr[np.argmin(intersection)], eer_threshold))
118
119      return eer, eer_threshold
```

3.2 请给出模型验证结果的截图以及绘制的 ROC 曲线。

4、声纹识别优化方法的研究*（1 学时）：

目前，在 x-vector 的基础上，涌现了许多有效的优化方法。其中，Am-softmax 以及 ECAPA-TDNN 分别在损失函数及神经网络架构上进行了改进。项目文件中已经进行了实现，请尝试选择一至两种方法进行调用，替换训练代码中的对应部分，并简要说明方法的优势。（注：https://arxiv.org/pdf/1801.05599.pdf；https://arxiv.org/pdf/2005.07143.pdf）

4.1 Am-softmax 的实现

Am-softmax 的相关代码保存在 **loss.py** 文件中，请在 **train.py** 文件中对其进行调用并运行，给出训练过程中的输出截图或视频。

loss.py 2 ×

loss.py > Softmax > forward

```
28  class Amsoftmax(nn.Module):
29      def __init__(self, embedding_size, num_classes, s, margin):
30          super(Amsoftmax, self).__init__()
31          self.embedding_size = embedding_size
32          self.num_classes = num_classes
33          self.s = s
34          self.margin = margin
35          self.weights = nn.Parameter(torch.Tensor(num_classes, embedding_size))
36          nn.init.kaiming_normal_(self.weights)
37
38      def forward(self, embeddings, labels):
39          logits = F.linear(F.normalize(embeddings), F.normalize(self.weights))
40          margin = torch.zeros_like(logits)
41          margin.scatter_(1, labels.view(-1,1), self.margin)
42          m_logits = self.s * (logits - margin)
43          loss = F.cross_entropy(m_logits, labels)
44
45          acc = self.accuracy(logits, labels)
46          return loss, acc
```

4.2 ECAPA-TDNN 的实现

ECAPA-TDNN 的相关代码保存在 **ecapa.py** 文件中，请在 **train.py** 文件中对其进行调用并运行，给出训练过程中的输出截图或视频。（提示：在使用 ECAPA-TDNN 计算语音的深度嵌入时，需要转置特征的第 1 维和第 2 维）

四、 参考资料

[1] SpeechBrain 官方文档及代码

https://speechbrain.readthedocs.io/en/latest/index.html

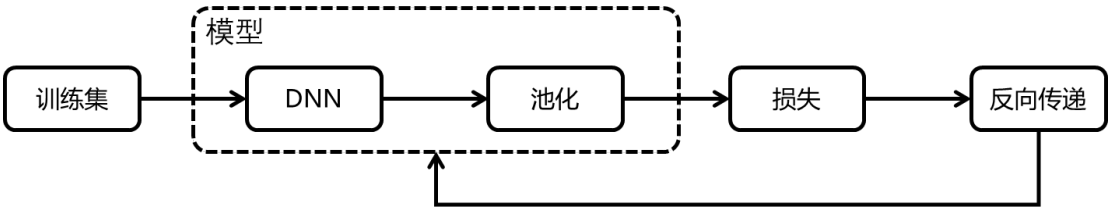
https://github.com/speechbrain/speechbrain;

[2] 声纹识别系统搭建

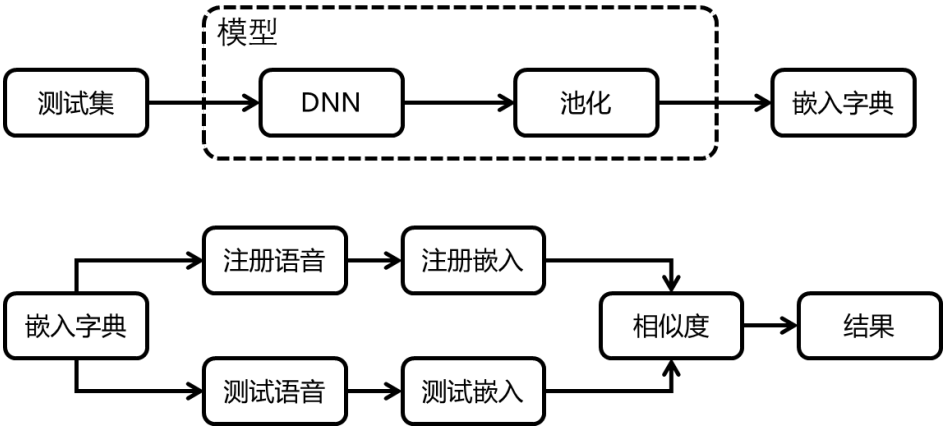
https://github.com/zengchang233/asv_beginner/tree/master

五、 声纹识别及模型搭建参考

1. 声纹识别系统训练流程



2. 声纹识别系统应用流程



3. Pytorch 模型搭建参考

https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py

Define the network

Let's define this network:

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # 5*5 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
print(net)
```

Loss Function

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

There are several different loss functions under the nn package . A simple loss is: `nn.MSELoss` which computes the mean-squared error between the input and the target.

For example:

```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

Backprop

To backpropagate the error all we have to do is to `loss.backward()` . You need to clear the existing gradients though, else gradients will be accumulated to existing gradients.

Now we shall call `loss.backward()` , and have a look at conv1's bias gradients before and after the backward.

```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

We can implement this using simple Python code:

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

However, as you use neural networks, you want to use various different update rules such as SGD, Nesterov-SGD, Adam, RMSProp, etc. To enable this, we built a small package: `torch.optim` that implements all these methods. Using it is very simple:

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```