浙江大学
Zhejiang University

夏令营
智能语音计算实验报告

姓名： 杨千

所在营： 软件服务分营

本科院校： 武汉大学

2021 年 7 月 19 日

目录

一、任务定义

此次夏令营的任务是进行音频的真伪鉴别，区分是真实语音还是合成语音。这个领域工作主要有两个方向：语音伪造和虚假语音鉴别。

1.1 语音伪造：以深度语音伪造为例

语音深度伪造是指利用人工智能技术（如机器学习算法、神经网络等）对人的声音进行“学习—模仿”，换句话说就是一种对声音进行重现（reenactment）、替换（replace）、编辑（editting）和合成（synthesis）的新型技术。这项技术的出现意味着“声音不再属于自己”，任何人的声音都是可以被伪造和替换的。

语音深度伪造技术已愈发成熟。由于语音深度伪造技术在医疗康复（如失声患者“重建”声音）、娱乐（如搞笑视频）等领域拥有巨大的发展潜力和应用价值，世界各国对此项技术的开发和推广投入了大量精力，相关技术也愈发成熟。

相比于人脸深度伪造（face deepfake），语音深度伪造出现时间较晚，主要兴起于 2019 年。语音深度伪造技术，在本质上是一种“文本—语音”转换系统（text-to-speech system，以下简称“TTS 系统”）。早期 TTS 系统通过语音合成技术，可以将录入的文本信息转化为对应的语音信号。然而，正如 E.Helander 和 J.Nurminen 所指出的早期 TTS 系统合成的语音信号在自然度、可懂度和连续性等方面的效果并不理想，也就是常说的听起来像机器声（robotic voice）。近年来，随着语音合成技术的不断进步，TTS 系统生成的语音信号质量在这些方面已经有了极大的提高。

T. Chen 指出，语音的深度伪造技术，就是将高质量的 TTS 系统和声音转化（voice conversion）相结合的语音合成技术。首先，计算机通过机器学习算法（如高斯混合模型（GMM）、卷积神经网络等）对说话人的语音样本进行特征识别，并建立相应的 TTS 系统。然后，把通过文本输入、语音转写等手段获取的文本信息转化为语音信号（包括实时和延时转换）。

目前，深度伪造语音不仅在拟人度、真实性和自然度等方面有了极大提升，而且面向不同语言（如汉语、英语、越南语等）的语音深度伪造软件已面向公众开放、且使用门槛和难度逐渐降低。

1.2 虚假语音鉴别

虚假语音鉴别主要有以下几个方面：

第一，从宏观言语特征角度开展鉴伪研究。与微观的声谱特征不同，言语特征（如口头禅、赘语、方言口音、发音习惯等）从宏观角度反映了说话人在语用层面的特点。E. Sapir 认为其形成与说话人的语言习得、性别、社会背景、工作等因素都有着密不可分的关系。基于声谱特征的语音深度伪造技术很难实现对说话人宏观言语特征的模拟，这就为伪造语音的专家检验提供了充分的可能性。今

后研究可以从言语特征的角度对语音进行分析，寻找能够体现说话人个体言语特点的有效载体和显著特征。相关结果对于语音的真伪鉴别具有较高的参考价值。

第二，探索真伪语音在声谱上的差异。尽管深度伪造的语音在声谱特征上与原声存在较高的相似度，但是前人利用专业软件仍能发现二者在声谱上的细微差异。Nios 公司的技术专家利用 Spectrum3d 软件对深度伪造的语音和原声的声谱特征进行对比分析，发现尽管二者在听觉上非常相似，但是伪造语音的声谱分布连续性较差，且在高频区域反复出现波峰。造成这个现象的原因可能是深度伪造软件为了提高和原声的相似度，将多个声道的语音叠加所致。随后，在对语音信号进行增幅后，可以检测到原声存在微弱的背景噪音，而伪造的语音未发现任何噪音痕迹。由此可见，真伪语音在频域分布、背景噪音等方面存在一定差异。在未来研究和实践中，应充分利用专业分析软件，发掘真伪语音的声谱差异，总结出规律性的知识。

第三，尝试改进算法和视角，进一步提升计算机自动鉴伪的效果。目前，已有学者如 M. Alzantot、B. Chettri 等，通过改进机器学习算法（如 2-D 卷积神经网络），将语音真伪判别的正确率提升到 75%左右。此外，还有学者如 T. Mittal 等，则提出一种从情感识别的角度进行鉴定的思路，利用深度学习网络（deep learning network）对视频中人脸和语音的真伪进行鉴定。他们首先通过感知实验让被试者分别对人脸和语音所表达的情感进行判断，然后基于感知结果对不同情感的特征进行提取和学习。最终，以情感特征为判断依据获得了较高的真伪鉴别正确率（84.4%以上）。可见，算法和视角的改进，对于提升计算机鉴伪效果具有一定作用，值得未来开展更广泛、更深入的研究。

二、代码改写

2.1 算法来源

算法框架和主体源码来自于 ASV SPOOF 2019 上面的一篇开源代码：https://github.com/yzyouzhang/AIR-ASVspoof

在此开源代码的基础上，我针对我们数据集的特点做相应的修改，进行训练并最终测试。

2.2 特征提取

语音识别的第一步就是提取相关的特征，本实验中特征提取是用的 ASV SPOOF 2019 的官方 MATLAB 代码，而我在此官方代码上做了许多修改符合我们数据集的规范。

本次特征提取的是线性频率倒谱系数（LFCC）特征，这个特征在这里我不做过多赘述。

整个 MATLAB 代码的流程比较简单，但也有一些细节需要注意，运行官方库

还是有许多的繁琐之处。

首先需要声明的是，此 MATLAB 代码所用的主要特征提取方法是官方已经写好的代码：用到了两个库，这两个库是我在 ASV SPOOF 官方得到的，花费了一定的功夫：

```
% add required libraries to the path
addpath('baseline/LFCC/');
addpath('baseline/CQCC_v1.0');
```

需要遍历阅读 txt 文件以获得所有文件名（这种情况对应于 train 和 dev 这两种有对应 txt 文件的情况）：（以下代码以 train 为例）

```
% read train protocol
trainfileID = fopen(trainProtocolFile);
trainprotocol = textscan(trainfileID, '%s%s%s');
fclose(trainfileID);
trainfilelist = trainprotocol{1};
```

接下来直接进行特征提取，先通过 audioread 读音频文件，紧接着通过 extract_lfcc 提取 LFCC 特征，再将其保存即可。

```
% extract features for training data and store them
disp('Extracting features for training data...');
trainFeatureCell = cell(length(trainfilelist), 3);
for i=1:length(trainfilelist)
    filePath = fullfile(pathToDatabase,'train/flac',[trainfilelist{i} '.flac']);
    [x,fs] = audioread(filePath);
    [stat,delta,double_delta] = extract_lfcc(x,fs,20,512,20);
    LFCC = [stat delta double_delta]';
    filename_LFCC = fullfile(pathToFeatures, 'train', horzcat('LFCC_', trainfilelist{i}, '.mat'))
    parsave(filename_LFCC, LFCC)
    LFCC = [];
end
disp('Done!');
```

最终的提取结果形成 60 维的 LFCC 特征。

2.3 数据预处理

有了提取到的特征，再对文件数据进行一定的预处理，作为训练前的最后准备。整个预处理包括两个部分，第一个部分是针对没有结果文件的 eval.txt，需要直接按照文件名遍历，它不同于之前的 train 和 dev：

```
for path,dir_list,file_list in g:
    for file_name in file_list:
        file_name =  file_name.replace(".flac", "")
        f.write("%s spoof\n"%file_name)
```

第二个部分是 reload_data 部分，这部分的主要工作是将 MATLAB 生成的特征文件.mat 转换为 Python 易处理的.pkl 文件：

```
def reload_data(path_to_features, part):
    matfiles = find_files(path_to_mat + part + '/', ext='mat')
    for i in range(len(matfiles)):
        #matfiles[i] = matfiles[i].split('\\')[7]
        if matfiles[i][len(path_to_mat)+len(part)+1:].startswith('LFCC'):
            #print(1)
            key = matfiles[i][len(path_to_mat) + len(part) + 6:-4]
            lfcc = sio.loadmat(matfiles[i], verify_compressed_data_integrity=False)['x']
            with open(path_to_features + part +'/'+ key + 'LFCC.pkl', 'wb') as handle2:
                pickle.dump(lfcc, handle2, protocol=pickle.HIGHEST_PROTOCOL)
```

2.4 网络架构

整体的网络架构源自深度残差网络（deep residual network）：ResNet-18。总的网络架构如下，我用 init 部分做以展示：

```
def __init__(self, num_nodes, enc_dim, resnet_type='18', nclasses=2):
        self.in_planes = 16
        super(ResNet, self).__init__()

        layers, block = RESNET_CONFIGS[resnet_type]
        self._norm_layer = nn.BatchNorm2d
        self.conv1 = nn.Conv2d(1, 16, kernel_size=(9, 3), stride=(3, 1), padding=(1, 1), bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.activation = nn.ReLU()
        self.layer1 = self._make_layer(block, 64, layers[0], stride=1)
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.conv5 = nn.Conv2d(512 * block.expansion, 256, kernel_size=(num_nodes, 3), stride=(1, 1), paddi
ng=(0, 1),
                               bias=False)
        self.bn5 = nn.BatchNorm2d(256)
        self.fc = nn.Linear(256 * 2, enc_dim)
        self.fc_mu = nn.Linear(enc_dim, nclasses) if nclasses >= 2 else nn.Linear(enc_dim, 1)
        self.initialize_params()
        self.attention = SelfAttention(256)
```

可以看到，全局的池化层被时空注意力层（attentive temporal pooling）所取代。该结构以提取的 LFCC 特征作为输入，输出置信度来表示分类结果，此层网络使用最后一个全连通层前的中间输出作为语音的嵌入层。

2.5 损失定义

本实验定义损失函数 oc-softmax 用来替代传统的二元损失函数，其定义如下：

```
class OCSoftmax(nn.Module):
    def __init__(self, feat_dim=2, r_real=0.9, r_fake=0.5, alpha=20.0):
        super(OCSoftmax, self).__init__()
        self.feat_dim = feat_dim
        self.r_real = r_real
        self.r_fake = r_fake
        self.alpha = alpha
        self.center = nn.Parameter(torch.randn(1, self.feat_dim))
        nn.init.kaiming_uniform_(self.center, 0.25)
        self.softplus = nn.Softplus()

    def forward(self, x, labels):
        """
        Args:
            x: feature matrix with shape (batch_size, feat_dim).
            labels: ground truth labels with shape (batch_size).
        """
        w = F.normalize(self.center, p=2, dim=1)
        x = F.normalize(x, p=2, dim=1)
        scores = x @ w.transpose(0,1)
        #print("1,scores:",scores)
        output_scores = scores.clone()
        #print("2,labels:",labels)
        scores[labels == 0] = self.r_real - scores[labels == 0]
        scores[labels == 1] = scores[labels == 1] - self.r_fake
        loss = self.softplus(self.alpha * scores).mean()
        return loss, -output_scores.squeeze(1)
```

此损失函数使用一个权重向量，此权重向量指向的是目标类别嵌入层的优化方向。同时为真实语音和虚假语音引入两个边界值，再以类交叉熵的方式定义最终的二元损失。

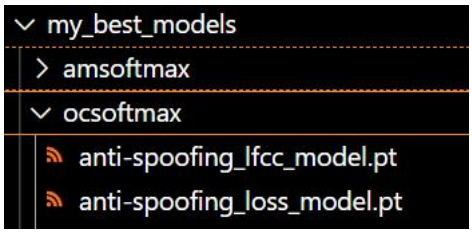
三、实验调试

整个实验在 train 之前，需要在源代码上进行许多改写，包括重写 dataset 类和 dataloader。由于是标准的 pytorch 训练范式，此处我不对源码中对 pytorch 训练范式的修改做过多阐述。

为达到最优的效果，结合我们的较为小量的数据集，需要不断对 train 所需参数做调整。整个 train 涉及到的参数已被标准化写入命令行参数中以便调试方便：

```
parser.add_argument('--num_epochs', type=int, default=100, help="Number of epochs for training")
parser.add_argument('--batch_size', type=int, default=64, help="Mini batch size for training")
parser.add_argument('--lr', type=float, default=0.0003, help="learning rate")
parser.add_argument('--lr_decay', type=float, default=0.5, help="decay learning rate")
parser.add_argument('--seed', type=int, help="random number seed", default=598)
parser.add_argument('--r_real', type=float, default=0.9, help="r_real for ocsoftmax")
parser.add_argument('--r_fake', type=float, default=0.2, help="r_fake for ocsoftmax")
```

其中，我对 r_real 和 r_fake 只做了简单的变化尝试，但试了一下还是之后都选择源码中的默认值，其余的参数都做了变化调整选出最优的模型如下：



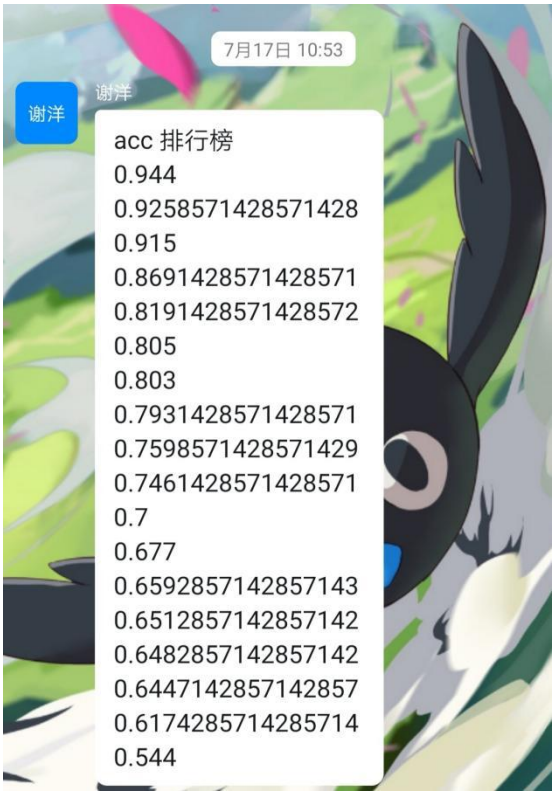
整个模型的训练相当耗时，我使用 Google 的 colab 带双核 GPU 训练了三天。

四、测试结果

使用最优的模型，在验证集 dev 上的结果的指标已经达到了非常高：

```
accuarcy:0.998800
recall:0.988506
precision:1.000000
```

同时在已经公布的 eval1 的评估结果中（7 月 17 日），我对 eval1 测试集的准确率达到了 0.944，高居**第一名**：


7月17日 10:53
谢洋
谢洋
acc 排行榜
0.944
0.9258571428571428
0.915
0.8691428571428571
0.8191428571428572
0.805
0.803
0.7931428571428571
0.7598571428571429
0.7461428571428571
0.7
0.677
0.6592857142857143
0.6512857142857142
0.6482857142857142
0.6447142857142857
0.6174285714285714
0.544