Project 1：语音端点检测

520030910334 张涵雪

1. 基于线性分类器和语音短时能量的简单语音端点检测算法

1.1. 数据预处理及特征提取

在此任务中，我最初是提取了语音的短时能量，过零率作为特征，且通过设置自适应的短时能量下限的操作实现消除直流分量的预处理。同时，考虑到到信噪比较低时能量为依据判断可能造成的错误，我选择在某些特定情况下使用基频作为特征，尽管在本任务中因为数据很好而并没有显著效果，但在我自己生成的低信噪比音频片段中该处理表现优异，是对传统双门限算法的一个小改进。

- 短时能量：较短时间内的语音能量，通常指一帧。由于浊音短时能量高，因此可通过该指标判断是否有浊音从而判断是否为发声段，公式如下：

$$E_n = \sum_{n=0}^{N-1} x(n)^2 \tag{1}$$

其中 N 为一帧所包含的采样点数（下同）。

- 短时过零率：每帧内信号通过零值的次数。轻音部分的短时能量和静音部分相当，但清音的短时过零率比静音部分高出很多，因此该指标可用于判断是否为清音而扩大语音段，公式如下：

$$Z_n = \sum_{n=0}^{N-1} |sgn(n) - sgn(n-1)| \tag{2}$$

- 基频：与音高类似，尽管二者并不相同。基频可看作发声的频率分段，一般对于人类的说话声音，基频观测值大约在 60Hz 到 300Hz。在信噪比很低的情况下，能量已经不能用于区分背景噪声和浊音。然而，即使在强噪声场景，谐波这一特征也是存在的。针对这种强噪声的情况，应该舍弃能量判断，使用自相关的方法找到基频，再通过基频判断是否为发声段。

- 消除直流预处理：在我们的算法中，对于能量部分不仅要求存在上限，同样要求存在下限，且该下限由数据自身决定。这其实就是一步去除直流的预处理操作。这种操作成功地避免了因为某些环境噪声而导致的误判，提高了各项指标。

1.2. 算法描述

改进的双门限算法如下：

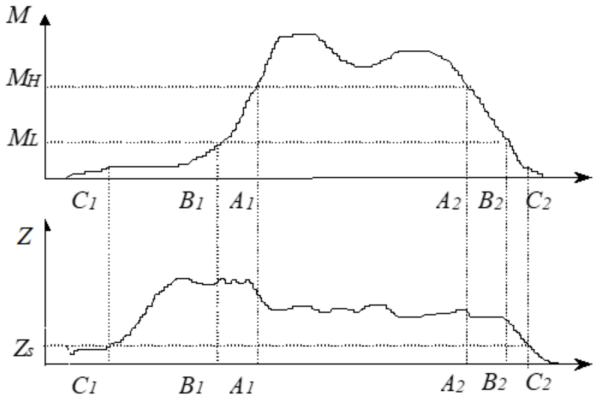

图 1: 传统的双门限算法示意

- 计算出音频数据的短时能量和短时过零率。
- 取一个较高的短时能量作为阈值 MAX，分出语音中的浊音部分。(经调试，该指标取所有帧的短时能量的平均数的一半最为合适。)
- 取一个较低的能量阈值，从上一步中判断出的分段向两端扩张，将能量高于该阈值的语音部分加入到语音段。将前五帧看作静音段。

经调试，该指标设置为将静音段能量均值和 MAX 的平均数的一半最为合适。

- 判断此时识别结果是否为从开头到结尾。如果是，说明基于能量的预测失效了，因此我们应采用基频作为特征，剔除频率不在范围内的部分。

- 记短时过零率的阈值为 Zs。以此为指标对语音段继续向两端进行搜索。经调试，将短时过零率大于 3 倍 Zs 的部分认为是语音的清音部分，并将该部分加入语音段。

- 输出结果。

1.3. 实验结果

我们的结果展示见表。

表 1: 给定数据集上的结果

实验内容	(AUC, EER)	Acc
标准双门限法	(0.94, 0.07)	0.93
添加基频的双门限法	(0.94, 0.07)	0.93

表 2: 低信噪比情况

实验内容	(AUC, EER)	Acc
未使用基频	(0.21, 0.67)	0.48
使用基频	(0.91, 0.08)	0.87

在实现的过程中我发现了两个问题：一是注意到我的算法中静音段取自前五帧，而数据集中有部分音频在开始就不是静音，导致这类数据在处理的过程中将阈值设置过高，从而使整段音频被识别为静音；二是前五帧中存在清音导致我的过零率阈值判断错误，从而造成对清音的识别失败。我对初始非静默的情况进行了一系列处理，尽管效果有了略微的提升，但仍然不尽人意。由于时间关系，我并没有找到除了调整参数外非常有效的解决方案，希望在未来有时间进行进一步探究，发现更有趣的结论。

2. 基于统计模型分类器和语音频域特征的语音端点检测算法

2.1. 数据预处理及特征提取

在任务二中，我对语音进行了归一化并加汉明窗后提取 MFCC 和 PLP 特征的工作，以 DNN 作为模型，分别以上述两个特征进行训练，最终选取效果更好的 MFCC。下面具体描述所用特征与操作。

- MFCC：这是基于人耳听觉特性提出来的频率，由于人耳对低频信号更敏感，对高频信号更不敏感，因此基于梅尔频谱与 Hz 频率的非线性对应关系，计算得到 Hz 频谱特征，示意图如下：

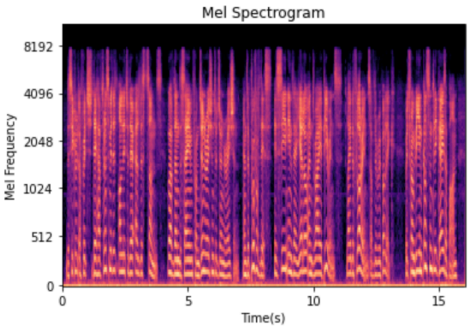

图 2: *mfcc* 特征示意图

- PLP：一种基于听觉模型的特征参数，采取线性预测方法实现语音信号的解卷处理，得到对应的声学特征参数。在实际的应用中，我计算 PLP 特征的计算量过大，由于资源和时间的限制最终应用效果并不好。

- 加窗预处理：在信号处理时需要截得信号片段进行各种处理，理想的情况是方窗，然而矩形窗在边缘处将信号突然截断，窗外时域信息全部消失，导致频谱泄漏，产生误差。在我的处理中我选择了汉明窗来截取信号，使得过度尽可能平滑，减少泄漏误差。

- 归一化预处理：归一化可以使后续处理更加方便，还可以使其对应某种概率分布，使数据更加形象。

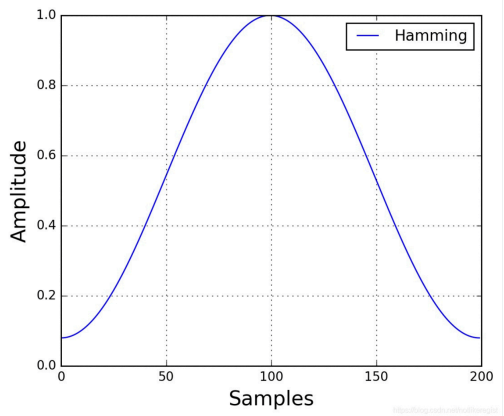

图 3: *Hamming window*

2.2. 算法描述

我的朴素 DNN 模型参数如下：

```
use_cuda = True # use gpu or cpu
val_ratio = 10 # Percentage of validation set
start = 1 # Start epoch
n_epochs = 15 # How many epochs?
end = start + n_epochs # Last epoch

lr = 1e-2 # Initial learning rate
wd = 1e-4 # Weight decay (L2 penalty)
optimizer_type = 'sgd' # ex) sgd, adam, adagrad

batch_size = 128 # Batch size for training
valid_batch_size = 32 # Batch size for validation
use_shuffle = True # Shuffle for training or not
input_size = 12 #dim of mfcc
model = DNN(input_size=input_size,  num_classes
    =1)
criterion = nn.BCELoss()
optimizer = create_optimizer(optimizer_type,
    model, lr, wd)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, 'min', patience=1, min_lr=1e-4,
    verbose=1)
```

朴素 DNN 结构如下：

```
class DNN(nn.Module):
    # define model elements
    def __init__(self, input_size):
        super(DNN, self).__init__()
        # input to first hidden layer
        self.hidden1 = Linear(input_size,8)
        kaiming_uniform_(self.hidden1.weight,
    nonlinearity='relu')
        self.act1 = ReLU()
       # second hidden layer
        self.hidden2 = Linear(8, 4)
        kaiming_uniform_(self.hidden2.weight,
    nonlinearity='relu')
        self.act2 = ReLU()
        # third hidden layer and output
        self.hidden3 = Linear(4, 1)
        xavier_uniform_(self.hidden3.weight)
        self.act3 =Sigmoid()

    # forward propagate input
    def forward(self, X):
        # input to first hidden layer
        X = self.hidden1(X)
        X = self.act1(X)
        # second hidden layer
        X = self.hidden2(X)
        X = self.act2(X)
        # third hidden layer and output
        X = self.hidden3(X)
        X = self.act3(X)
        return X
```

我的最终算法如下：

- 提取数据的 MFCC 特征，并以一帧为单个训练样本制作数据集。
- 对模型进行预训练，将得到的参数作为模型的初始值。
- 将训练数据放入 DNN 模型进行训练一定 epoch。
- 在模型中添加 dropout 并缩小学习率训练一定 epoch，实现调优并防止过拟合。
- 在开发集上输出结果。

其中用到的主要一些 trick 如下：

★ dropout：在初步训练后增加 dropout 继续训练，使一部分神经元失效，达到了调优且防止过拟合的效果。

★ 训练轮数：在模型收敛后停止训练，避免不必要的算力浪费。

★ 模型预训练：用小规模数据集对模型进行预训练，再以此时的参数作为模型的初始化参数，达到更好的训练结果。

★ 模型规模：注意到我初始模型规模较小，通过适当增大模型的规模，可以提高最终的指标，但也要注意精度和泛化能力的 trade-off。

★ 学习率衰减：观察到模型训练在前几个 epoch 迭代时 loss 收敛，而后开始震荡，考虑我们使用的是 SGD 而非 Adam 优化器，不能自动迭代学习率，因此人为在每个 epoch 为其乘上一个衰减系数，以达到更好的效果。

2.3. 实验结果

我的结果如表所示，其中包含了朴素 DNN 模型，在训练了一些 epoch 后通过 Dropout 及学习率衰减细致优化后的 DNN 模型以及进一步扩大 HiddenSize 为 128*128 的 DNN 模型。结果符合预期。

表 3: 开发集性能表现

Method	**(AUC, EER)**	**Acc**
DNN-Naive	(0.94, 0.09)	0.92
DNN-Drop-lr	(0.96, 0.08)	0.93
DNN-LargeScale	(0.97, 0.07)	0.94

除此以外，还有一些由于时间限制仍未实现的想法：

- 在生成数据集的时候可以不仅仅以单个标签作为单位（如 [[1], [0], [0], ...]），还可以将标签制作成为窗的形式（如 [[1, 0, 0, 1], [1, 1, 1, 1], ...]）。这样从直观上感觉神经网络可以学到一些时域上的特性，尤其是在有声和无声交界的地方，并且或许可以添加前序信息作为特征。
- 如果能知道各种声音的特征，可以将 VAD 推广应用到其他声音事件的检测，例如：是否出现男声、是否出现鸟叫等等。

希望在假期可以对其进行进一步探究。

3. 参考资料

https://blog.csdn.net/rocketeerLi/article/details/83307435