目录

数字信号处理课程报告

赵海凯

学号：16307110258

专业：计算机科学与技术

摘要

本项目基于课堂上讲授的的信号与系统以及语音特征提取的知识，通过提取语音信号的梅尔频谱图以及梅尔倒谱系数分别作为语音信号的特征，并结合深度学习训练方法，对两种特征提取方法进行多次实验与比较。最后笔者实现了一个识别准确率到达约 97%，并且能够在网页浏览器运行的孤立词语音识别系统，

而本实验报告主要分为两大部分：一是对基于上课所学的关于语音识别的概念知识 (如加窗的原理，过零率，傅里叶变换等)，使用项目的数据集进行实验并进行可视化，以便加深笔者对语音识别知识的理解。第二部分是介绍算法原理以及数据处理，特征提取和模型搭建的详细步骤。

关键字: 语音识别, 卷积神经网络, 数字信号处理, 梅尔倒谱系数, 深度学习, 傅里叶变换

第一章　背景

语音识别其实是一门交叉学科，其涉及的领域包括但不限于：模式识别，发声原理与机制，数字信号处理，概率论，信息论等。

而语音识别的方法也有很多，从 70 年代的码本生成算法 (LBG) 与线性预测编码 (LPC), 到 80 年代随着隐马尔科夫模型与高斯混合模型的理论与应用日趋完善，GMM-HMM 模型成为了语音识别的主流框架。这时候语音识别的研究也从孤立词语音识别慢慢转移到对非特定人连续语音识别当中。

一直到 06 年深度置信网络的提出，促使了深度神经网络的发展，越来越多的研究开始集中在把深度神经网络应用于声学建模当中。而 2012 年在 Imagenet 图像识别竞赛当中大放异彩的 AlexNet 更是把深度学习推向了顶峰。从此 DNN-HMM 框架取代了 GMM-HMM 成为了语音识别的主流框架，目前主流的对语音识别的研究都是基于深度学习的建模上。一种思路主要是把语音信号转化为频谱图，那么使用卷积神经网络就相当于把语音识别任务转化为对语音的频谱图进行识别的图像识别任务。另一种思路主要是利用递归神经网络结合隐马尔科夫模型，把语音信号看成一段自然语言的序列并对其进行建模，把语音识别任务转化为类似于文本分类的任务。

而且语音识别现在已经有非常成熟的落地产品如苹果的 siri，微软的 Cortana，百度语音等，能够较好地完成非特定人的连续语音识别任务。

所以本项目要求实现的孤立词语音识别其实难度并不算非常大，毕竟这是 80 年代就已经开始研究的课题，想必使用最新的建模与学习方法必然能达到不错的效果。

第二章　实验部分

由于本课程介绍了非常多而且容易令人混淆的概念，因此拿到数据的时候，我没有先急着开始做项目，而是先针对老师介绍的各种语音识别所用到的特征以及信号与系统部分介绍的内容进行复习，以及使用数据集进行相关原理代码的复现和可视化。

第 1 节　傅里叶变换

加一点数学公式

我们知道，任何连续测量的时序或者信号，都可以表示为不同频率的正弦波信号的无限叠加。而傅里叶变换，就是利用直接测量到的原始信号，以累加方式计算信号中不同正弦波信号的频率，振幅与相位。本质上是把难以处理的时域信号转化为容易分析的频域信号，也就是频谱。

我们拿一对发音比较相近的词“北京”与“背景”进行实验, 分别绘制其在时域和频域上的图形。

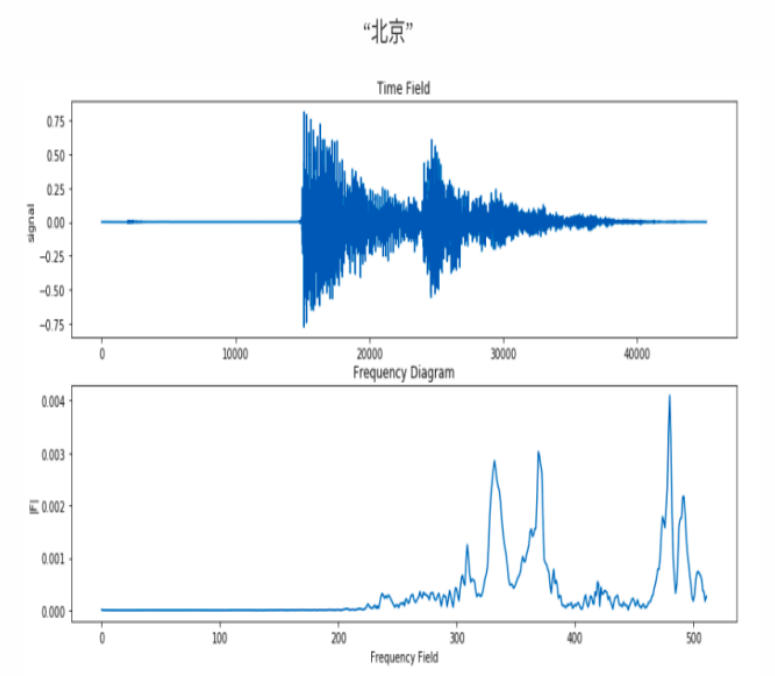

图 2.1:

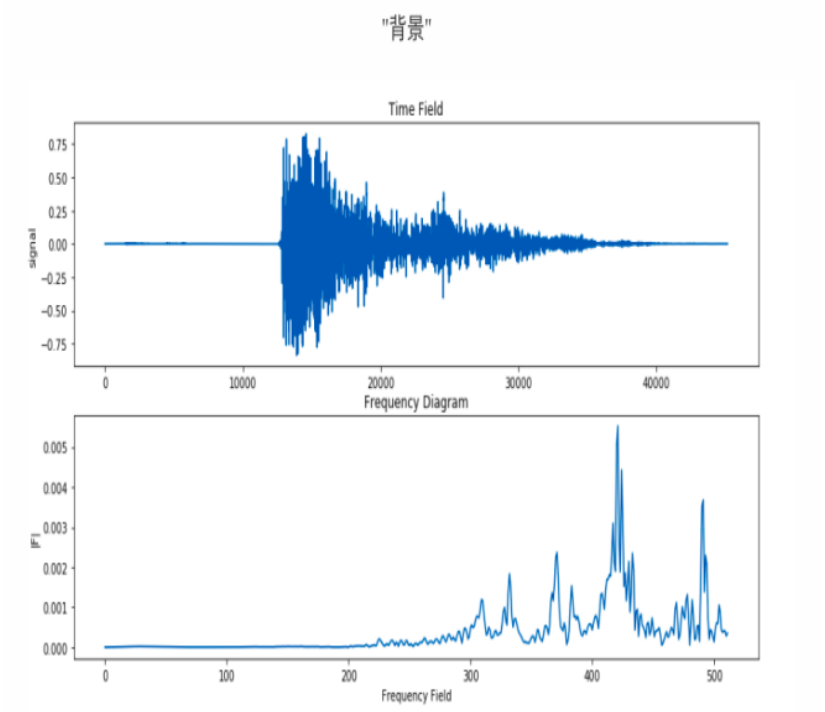

图 2.2:

上面两幅图，上方的子图代表该信号时域上的分布，可以看见从时域比较难分析出其特征以及区别。而傅里叶变换的强大之处就在于，它能把信号的表示从时域转化为频域，下方的子图就是两个词分别在频域上的分布。这样可以很直观的看出两个语音数据的频域分布的差异性。如北京的频率峰值大概在 470-480 左右，而背景的频率峰值大概在 420 左右。

第 2 节　短时能量分析与加窗

对于语音信号短时分析，信号流是要经过分帧处理，而分帧是通过可以移动的有限长窗口加权实现的。加窗时候如果每次移动的距离与窗宽度相同，则各帧信号会相互衔接。但是通常移动距离要比窗宽小，就是帧与帧之间要有重叠部分，这是因为帧之间连接的信号会因为加窗而弱化，如果没有重叠部分，这部分的信息会丢失。

下面以较大篇幅来探讨为什么要加窗：

FFT 分析信号的频率成分时，分析的是有限的数据集合。换言之，FFT 认为波形是一组有限数据的集合，即一个连续的波形是由若干段小波形组成的。

如果测量信号是周期信号，而且采集时间内刚好有整数个周期。例如下图余弦波 $f = cos(20 * 2\pi t)$, 频率为 20Hz，周期为 0.05s，那么这里采样 1s 获得了 20 个余弦波的完整数据进行 fft 分析，结果很好，能够很明显地在频谱图看出其频率在 20Hz。

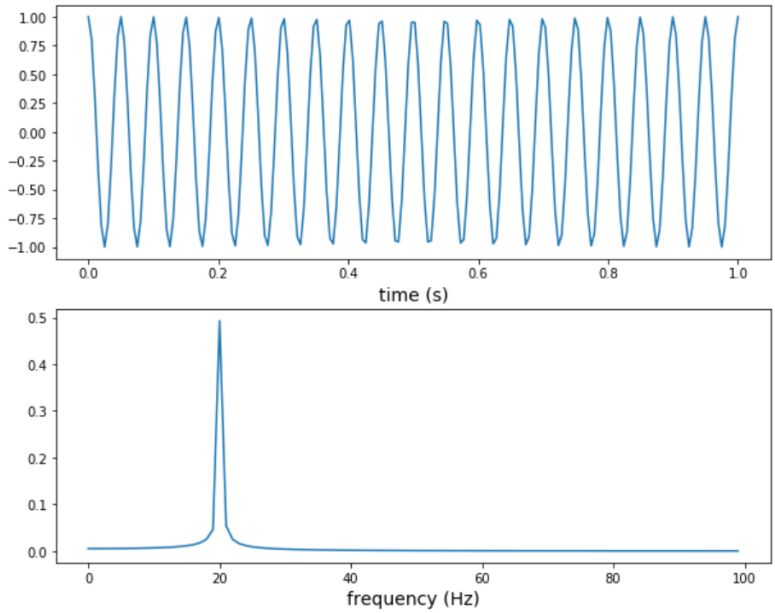

图 2.3:

那如果获得的是半个，1 个，两个余弦波呢？
下面是采样 0.025s 的情况，由于余弦波周期为 0.05s，因此这里只能采集到半个波。

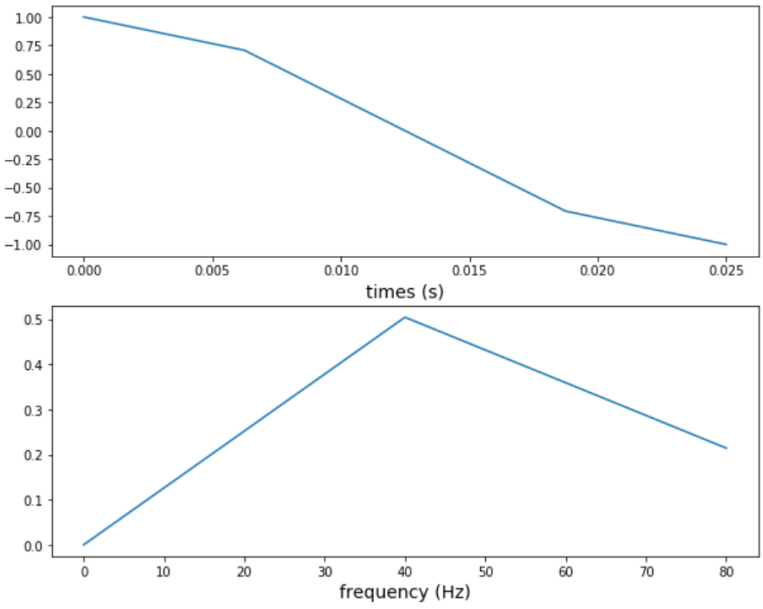

图 2.4:

可以看到 fft 分析的结果很糟糕，基本看不出其频率为 20Hz。
下面是采样 0.05s 的情况，只能得到一个波，fft 分析效果不是特别好，但能勉强看

到频率高峰在 20Hz 处。

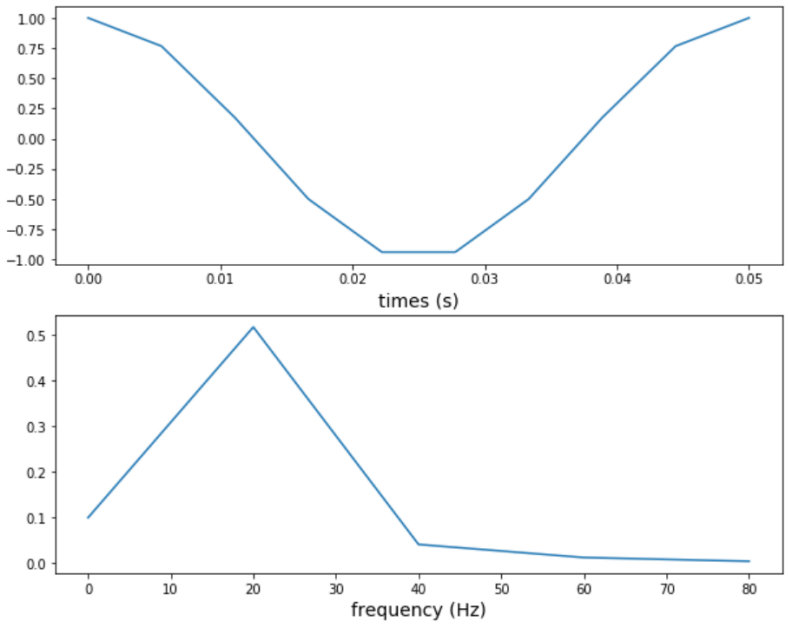

图 2.5:

下面是采样 0.2s 的情况，只能得到 4 个余弦波，fft 分析效果明显好转，可以看出频域高峰大概在 20Hz 处.

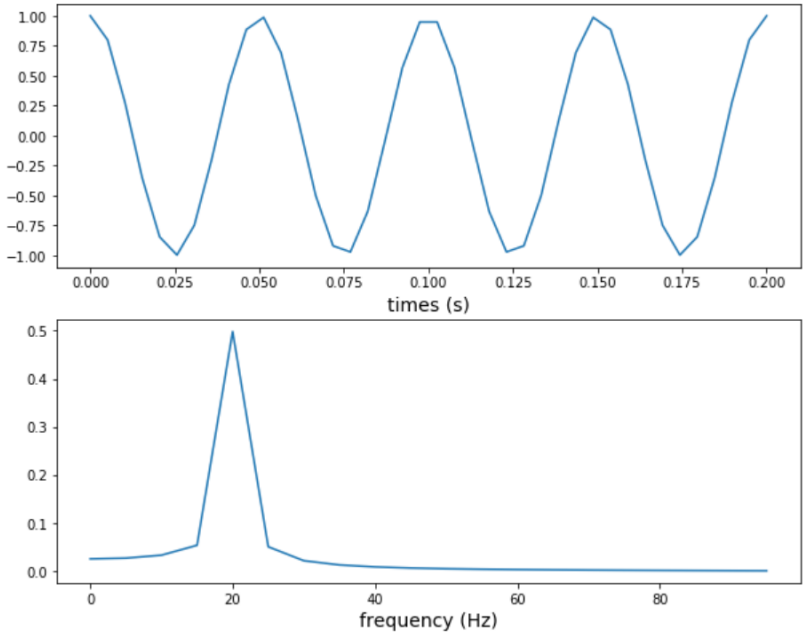

图 2.6:

当采样时间为 0.4s 的时候，频率谱更尖，所以我们发现当采集得到的波数目越多，fft 效果越来越好。

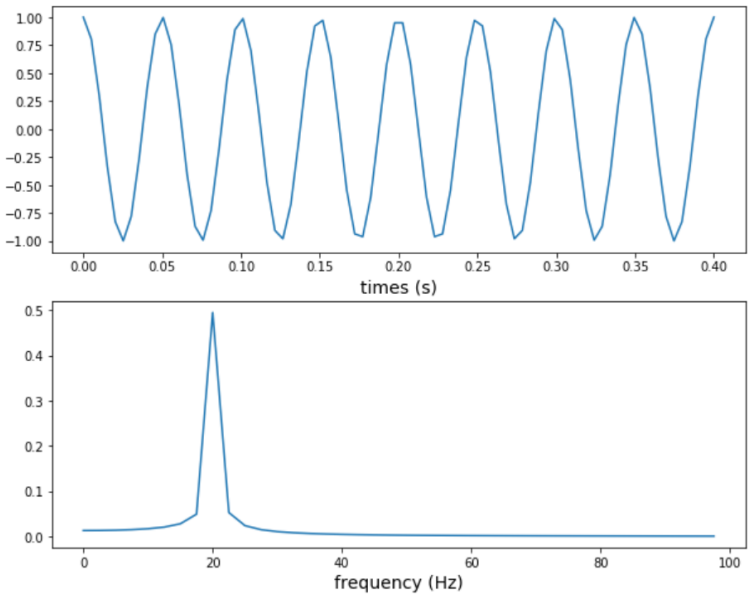

图 2.7:

分析

有限的时间段内对信号进行测量，无法知道测量范围之外的信号情况，而傅里叶变换对测量范围之外的假设是范围之外的信号都是测量信号的重复。

但并不是所有情况下都能测量到整数个周期。数据是从信号周期中间截断获得的，与时间连续的原信号显示出不同的特征。有限数据采样会使得测量信号产生不连续性，表现为这些不连续片段在 FFT 中显示为其他的频率成分。实际上，相当于低频的能量泄漏到其他频率，这种现象叫做**频谱泄漏**。

那如果数据采样得不到完整的余弦波，如只记录 60 条数据，即只记录了一个半余弦波，我们看看那会发生什么？

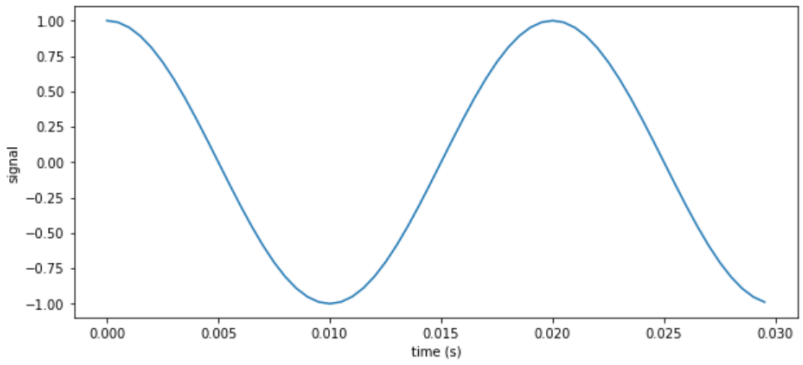

图 2.8:

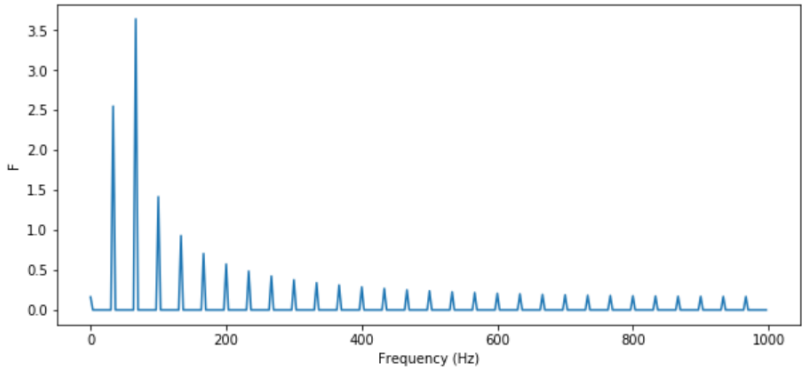

图 2.9:

上图 2.9 就表示发生了频谱泄漏，这样 FFT 的分析效果很差，因此我们就要引入窗函数。

2.1. 窗函数

引入窗函数的目的就是为了尽可能减少在非整数个周期上进行 FFT 所产生的误差，具体来讲，是减少这些边界不连续部分的的幅值，窗的幅度会在靠近边缘处接近于 0，结果是尽可能呈现出一个连续的波形，以减少剧烈的变化。

常用窗函数有汉宁窗和汉明窗，以下是代码实现：

```
def get_window(name='Hamming', N=1024):
window = None
if name == 'Hanning':
window = np.array([0.54 - 0.46 * np.cos(2 * np.pi * n / (N - 1)) for n in range(N)]
                                        )
elif name == 'Hamming':
window = np.array([0.5 - 0.5 * np.cos(2 * np.pi * n / (N - 1)) for n in range(N)])
```

```
#plt.plot(window)
return window
```

那我们不仿试试对上图提取 60 个采样点的情况进行加窗，看看结果如何？加入汉明窗前后的比较:

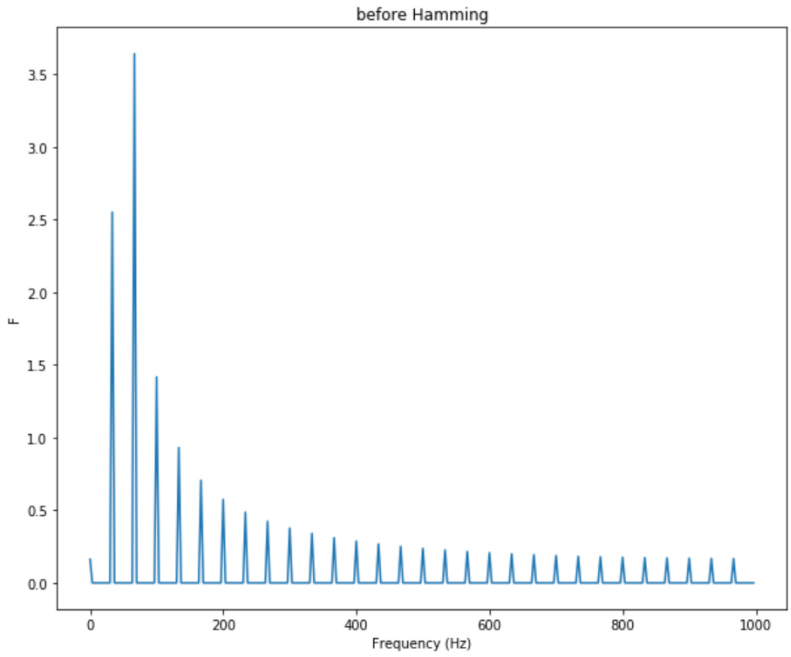

图 2.10:

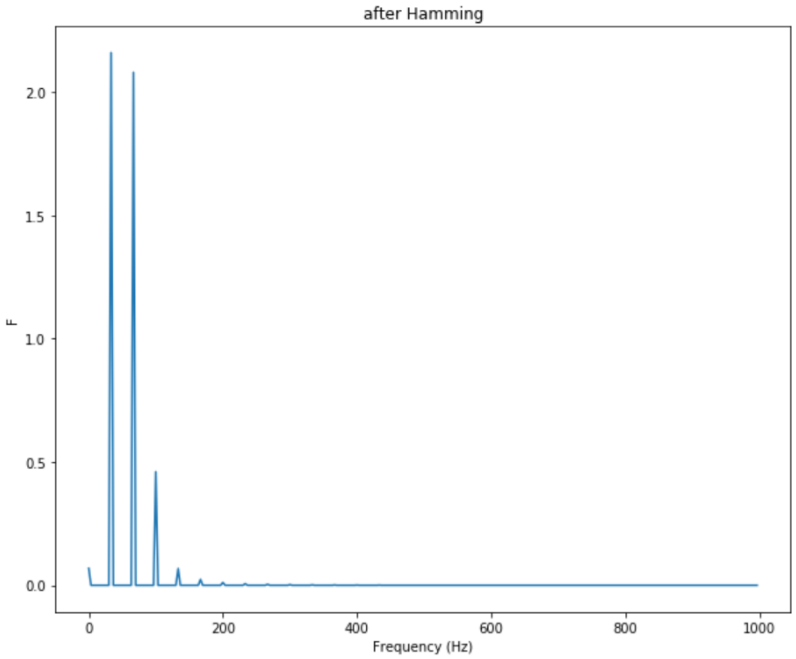

图 2.11:

加入汉宁窗前后的比较:

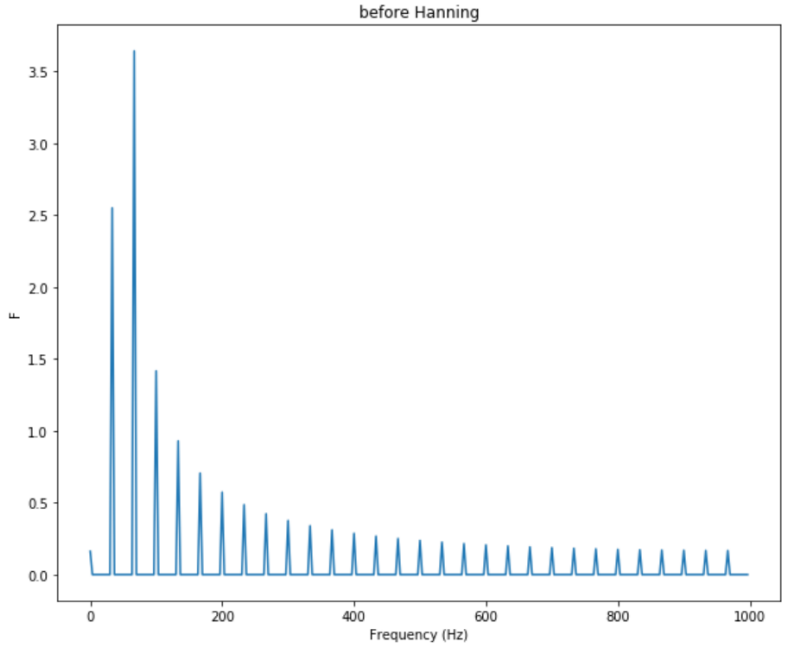

图 2.12:

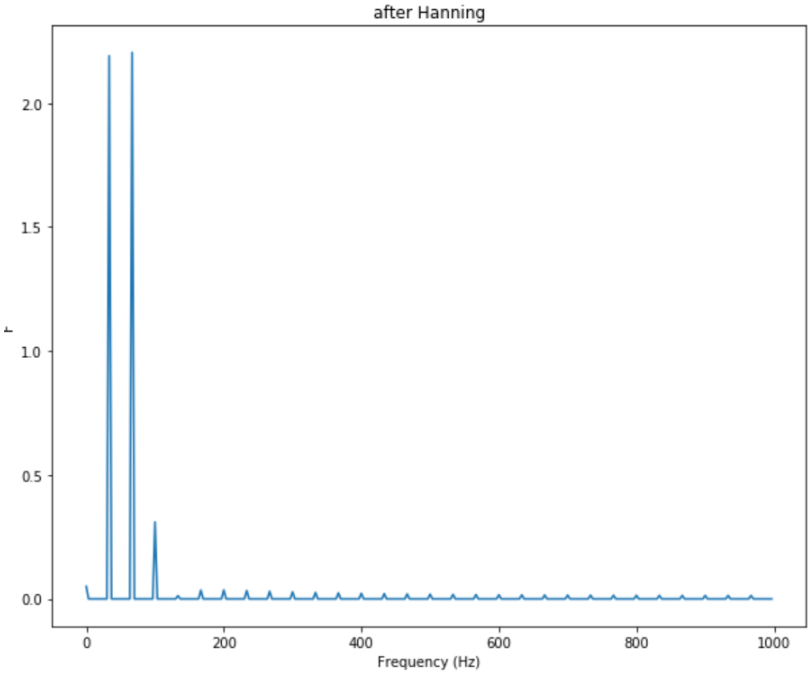

图 2.13:

可以看出，引入窗函数之后，fft 分析的频谱泄漏现象得到有效的缓解，而且频域会集中在正确的频率 50hz 左右处，而且貌似汉明窗的效果比汉宁窗更好，因此后续提取特征的时候默认选择使用汉明窗。

2.2. 短时平均能量

现在我们来探讨短时平均能量这个概念

计算公式：

$$E_n = \sum_{m=-\infty}^{\infty} [x(m)w(n-m)]^2$$

用途：

1. 可以作为区分清音和浊音的特征参数，浊音的短时能量会比清音大很多。

2. 在信噪比比较高的情况下，短时能量能够作为区分是否有声音的依据。

3. 可以作为辅助的特征参数用于语音识别当中。而在本次课程项目中，我尝试最后把提取出来的 mfcc 倒谱系数以及短时平均能量做特征融合，并与没有做特征融合的结果进行比较。

下面是分别对背景，北京两词的短时平均能量做可视化。

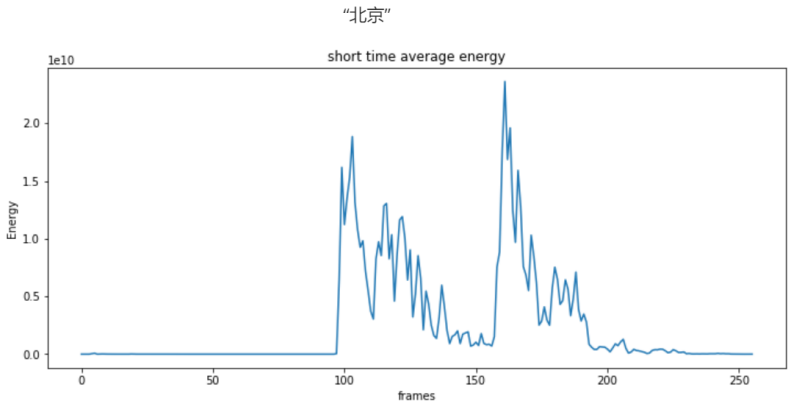

图 2.14:

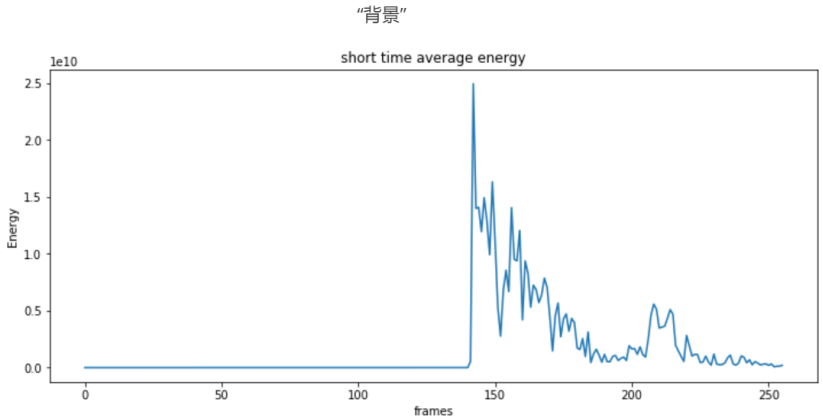

图 2.15:

第 3 节　短时平均过零率

物理意义为一帧语音中语音信号波形穿过横轴的次数，也可以理解为样本数值改变符号的次数。计算公式为：

$$Z_n = \sum_{m=-\infty}^{\infty} |[sgnx(m)] - sgn[x(m-1)]|w(n-m)$$

其中 w(n-m) 是窗口序列，sgn 是符号函数

在语音分析中，可以用来粗略描述频谱特性，因此可以用来区分清音与浊音，清音的时候能量集中在高频，浊音相反。利用这个性质可以再背景噪声中寻找出语音信号，用来判断无语音以及有语音的时候起点与终点位置 (即端点分析)

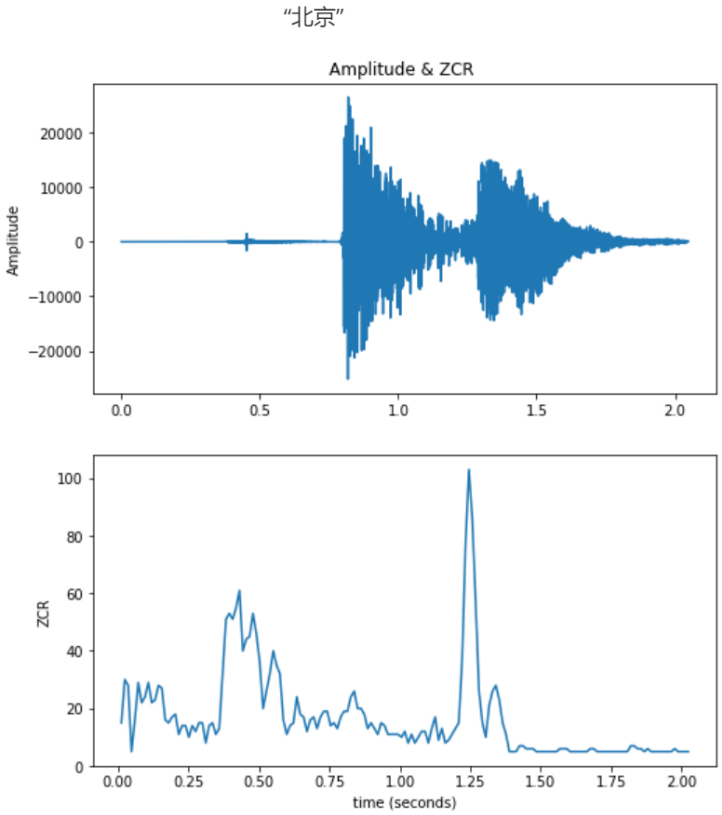
“北京”
Amplitude & ZCR
Amplitude
20000
10000
0
−10000
−20000
0.0
0.5
1.0
1.5
2.0
ZCR
100
80
60
40
20
0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
time (seconds)

图 2.16:

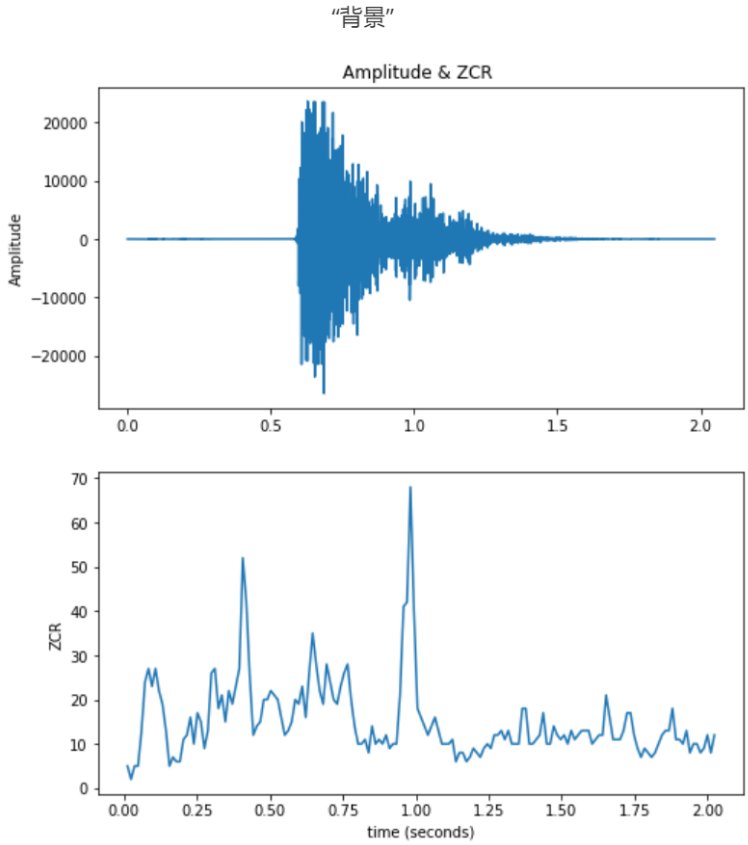

图 2.17:

第 4 节　频谱分析与倒谱分析

上面的短时能量和短时平均过零率都是属于语音信号的时域分析，下面我们来探讨频域分析。

“北京”的倒谱图以及梅尔标度频谱图

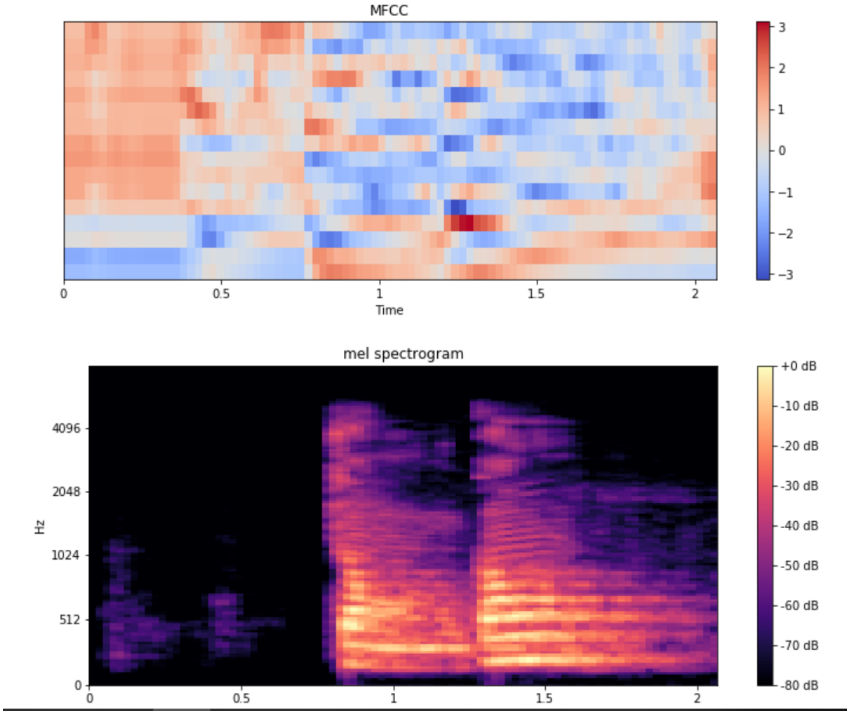

图 2.18:

“背景”的倒谱图以及梅尔标度频谱图

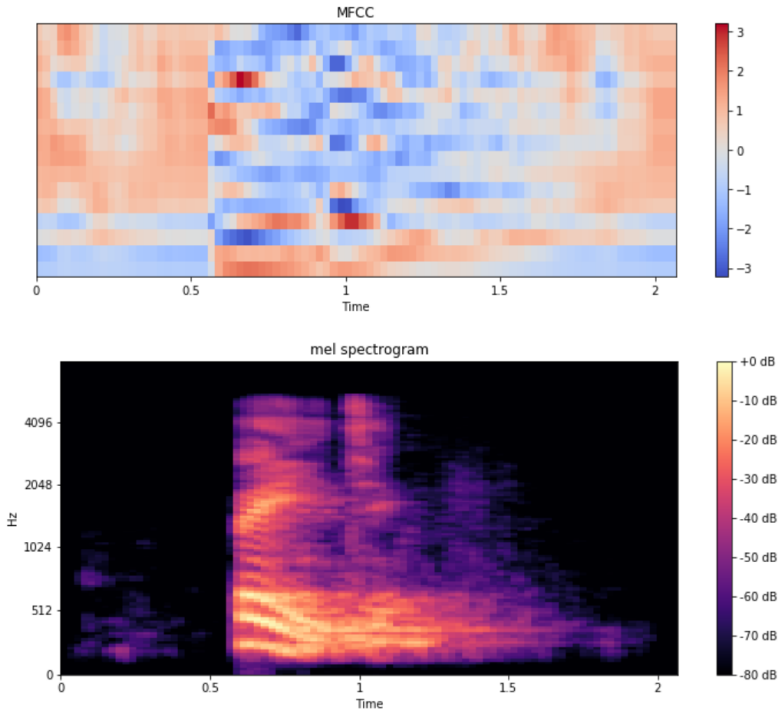

图 2.19:

语谱图的横坐标是时间，纵坐标是频率，坐标点值为语音数据的能量。由于是采用二维平面表达三维信息，所以能量值的大小是通过颜色来表示的，颜色深，表示该点的语音能量越强。

但由于本项目特征提取我是用倒谱系数 mfcc 以及梅尔标度的频谱来作为分类的特征的，因此这部分详细的过程以及原理放在报告介绍项目特征提取部分当中，这里仅给出“背景”与“北京”两词的频谱可视化。

第 5 节　频谱质心

频谱能量的集中点，一般来说，此值越小，说明越多的能量集中在低频范围。这是描述音色属性的重要物理参数，是频率成分的重心，是在一定频率范围内通过能量加权平均的频率，查阅资料发现该参数常用于对乐器声色的分析研究当中。对本项目而言该特征参数用处不大，我试过用第三方库提取该特征，使用 VGG13 作为 baseline 模型，测试准确率只有 53.6% 左右，效果比较差，所以就没有再做进一步的实验。

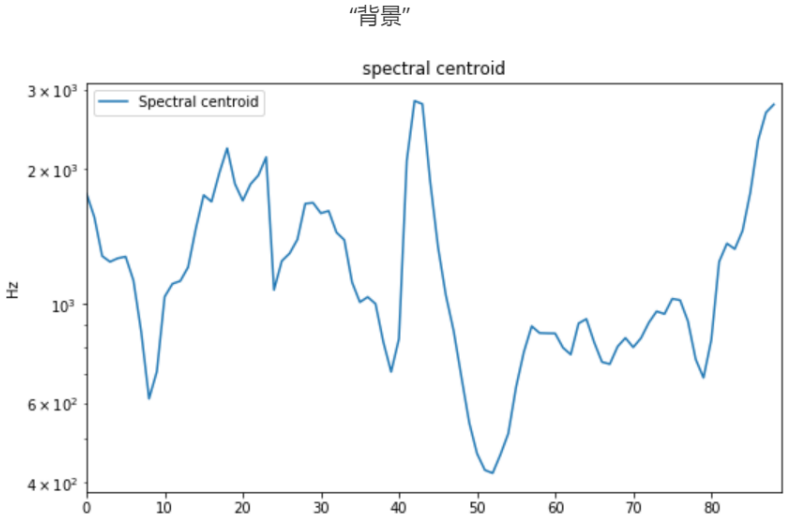

图 2.20:

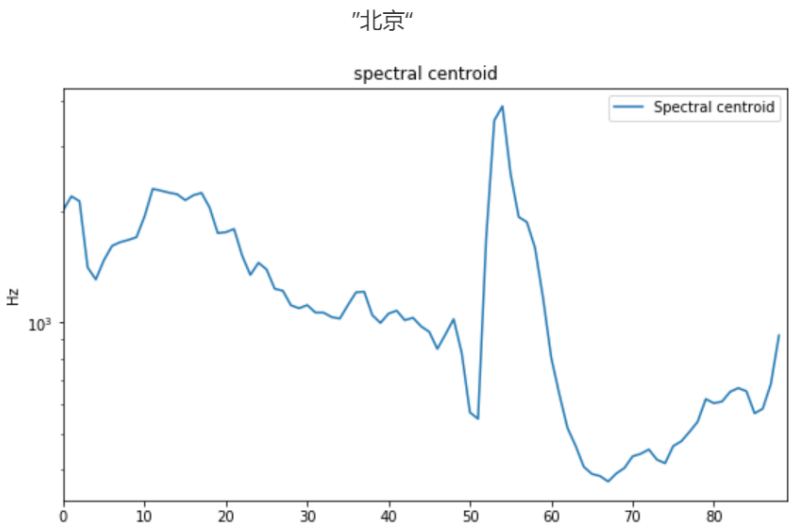

图 2.21:

实验部分做到这里，我对许多信号处理以及语音特征概念有了更深层次的了解，然后就可以正式开始做项目了。

第三章　项目部分

第 1 节　数据集介绍

给定 20 个词，每位同学朗读 20 遍，每名同学有 400 条语音数据，总共有 34 名同学的数据，语音数据总数为 34*400 = 13600 条，每个词分别有 680 条语音数据。

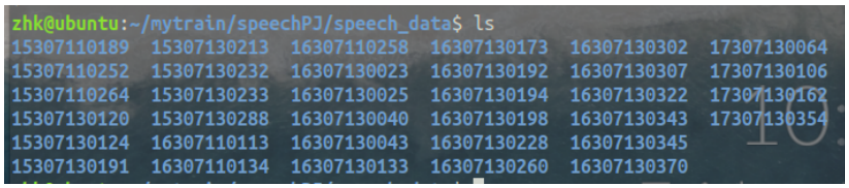

图 3.1:

我其中选择了学号为

'16307110258','16307130345','16307130370','16307130260','16307130133'

的五名同学共 2000 条数据集作为测试集，不参与训练。这里我把自己的录音数据拿出来当测试集，目的是我自己测试的时候能够更精确反映模型的泛化能力。而其他的数据作为训练集，并且在训练过程中采用 10 fold cross-validation(十折交叉) 来划分 training data 与 validation data。

第 2 节　技术栈

开发环境	Ubuntu 18.04
GPU	四块 Tesla K20m
语音数据处理部分	Python 3.7+Jupyter Notebook
前端+录音插件	Javascript
深度学习框架	Pytorch 1.0
网页服务端	Python Flask + Php
python 第三方库	numpy,librosa,scipy

图 3.2:

注：语音处理的核心代码部分都是利用 python+numpy 自己编写，除了数据载入使用 librosa 之外，并没有调用 librosa 这个语音第三方库的相关特征处理代码。

第 3 节　项目文件架构

核心代码部分：

/utils
 /Featurize.py 语音数据处理的核心代码
 /dataProcess.py 封装数据类
/models
 /BasicModule.py 模型的基类定义
 /VGG.py
 /ResNet.py
/training.py 训练模型所用代码
/app.py 开启 flask 服务端代码
/index.html 网页端代码
/static
 app.js 控制网页端的逻辑
 upload.php 用于把录音数据保存到本地

第 4 节　特征提取

在这部分，我将会详细介绍本项目中我最后使用的两种特征:**mel 标度频谱**以及其做对数以及离散余弦变换得到的 **mfcc 倒谱系数**。

首先我们知道，声音信号是一维的时域信号，直观上很难看到频率的变化规律。在实验部分我们提到，可以通过傅里叶变换把它变换到频域上，这样子虽然可以看到信号的频率分布，但是会丢失掉时域信息，无法看出频率分布随着时间的变化。而**短时傅里叶变换**是最经典的时域频域分析方法。

短时傅里叶变换是对短时间的语音信号做傅里叶变换，而短时信号是通过对长时的信号分帧得到的。那么我们把一段长信号通过分帧，加窗再对每一帧做傅里叶变换，最后把每一帧的结果按时间维度堆叠起来，就可以得到一副二维的图像，也就是**声谱图**。这也启示着我们，可以利用在图像识别领域大放异彩的卷积神经网络作为该项目的分类器。

而为什么这里选择使用 mel 标度频谱呢？什么又是 mel 标度？

在课堂 < 语音产生模型 > 中提到这么一种现象：声音成分中高频成分会影响对低频成分的感受，使其变得不易察觉，这种现象称为掩蔽效应。查阅资料发现，人耳能听到的频率范围是 20-20000Hz, 但人耳对 Hz 这种标度单位并不是线性感知关系。于是就引入了 mel 标度，把线性频率转化为 mel 标度，方便来描述人耳频率的非线性特性。映射关系如下：这是一个经验公式

$$Mel(f) = 2595 * log(1 + \frac{f}{700})$$

线性频率标度与梅尔频率标度关系，可以观察到频率越高，mel 频率增长速度越慢，本质上是模拟了**人耳在高频时迟钝的特性**。

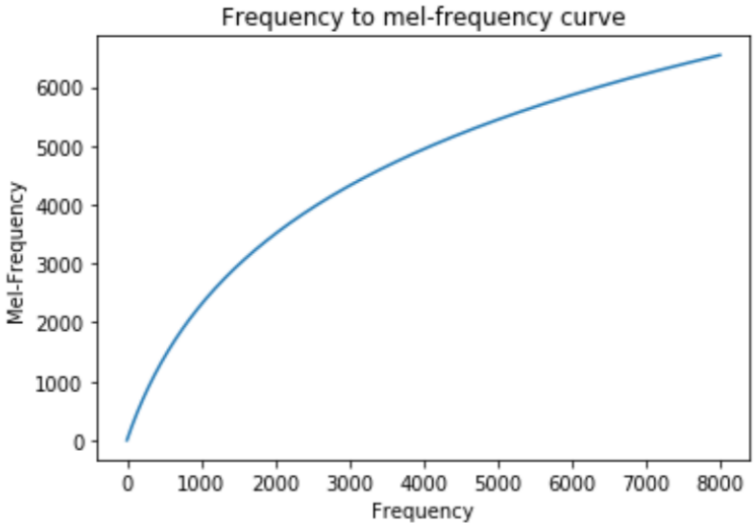

图 3.3:

于是可以从低频到高频这一段频带中按**临界带宽**的大小安排一组带通滤波器 (梅尔标度滤波器组 **mel-scale filter banks**，并将每个带通滤波器输出的**能量**作为信号的基本特征，这就得到了**梅尔标度频谱**。

那么什么又是 mfcc 呢？

我们把做梅尔滤波之后得到的结果取对数，之所以这样做可以理解为放大低能量处的能量，也可以理解为模仿人耳对对数式感知声强的特点。

然后做离散余弦变换，离散余弦变换是傅里叶变换的一个变种，好处是结果只有实部没有虚部，作离散余弦变换可以理解为把梅尔滤波提取出来的能量特征进行去相关并降维，转换到倒频谱域。MFCC 就是这个倒频谱的幅度。倒谱的低频分量是频谱的包络，高频分量是频谱的细节。查阅资料发现，通常使用前 13 个系数作为特征，这里提取低位的倒谱系数可以理解为模拟一个低通滤波器。

这 13 维系数又称作静态系数，是由 1 log Energy 系数 + 12 dct 系数组成的。其中 log energy 系数可以用来区分语音帧和非语音帧。而有时候也常用 39 维的 MFCC 作为特征，这里 39 维分别是 13 维的静态系数 +13 维的一阶差分系数 +13 维二阶差分系数。这里的差分系数用来描述动态特征，可理解为声学特征在相邻帧间的变化情况。

在做实验的时候，我尝试了分别提取 16 维以及 40 维的 mfcc 作为特征，并对模型训练后的精确度进行比较。

原理介绍完毕，下面介绍特征提取的基本流程。

4.1. 预加重

预加重，把语音信号通过一个高通滤波器

$$s[n] = s[n] - \mu * s[n-1]$$

μ 通常取 0.97，是为了提高高频成分，使得信号的频谱变得平坦，保持在低频到高频的整个频带中，能用同样的信噪比求频谱。同时也为了突出高频中的共振峰，补偿语音信号受到发音系统锁抑制的高频成分。

```
def preEmphasis(audioSignal,preEmphCof):
"""
:param audioSignal:
:param preEmphCof: 0.97
:return:
"""
firstSignal = audioSignal[0]
firstAfterSignal = audioSignal[1:]
exceptFirstSignal = audioSignal[:-1]

result = np.append(firstSignal,firstAfterSignal-preEmphCof*exceptFirstSignal)
return result
```

4.2. 分帧

分帧是因为语音信号是快速变化的，傅里叶变换适用于分析平稳的信号。一个帧会把 N 个采样点集合起来。N 取值有讲究，要保证一帧内既有足够多的周期，又不会变化剧烈。而且相邻两帧之间会有一段重叠区域，重叠区域包含了 M 个取样点。之所以会重叠，是因为语音信号是时变的，在短时范围内特征变化较小，可以当做稳态来处理。超出这短时范围的话语音信号就会有变化。所以为了使特征参数变化平滑，要让相邻两帧之间有一段重叠区域。我做实验的时候对不同帧长进行比较，发现帧长取 1024，即一帧包含 1024 个点，相邻帧之间重叠一半的时候效果最好。

```
def frame(signal,frame_length=1024,hop_length=512):

from numpy.lib.stride_tricks import as_strided
"""

:param signal:
:param frame_length: 帧长度
:param hop_length: 不同帧之间的重叠的样本个数
:return:
"""
#compute number of frames
num_frames = 1 + int((len(signal)-frame_length)/hop_length)
y_frames = as_strided(signal,shape=(frame_length,num_frames),strides=(signal.
                                        itemsize,hop_length*signal.itemsize))

return y_frames
```

4.3. 加窗与补零

由于每段语音信号长度不一样，因此要对语音信号补 0。而且每次分帧的时候，为了满足 fft 操作需要，数据要变为 2^n 个点，因此编写了一个 padding 函数用于进行补 0 或对齐操作。

```
def padding(data,size,axis=-1,**kwargs):
"""
补0操作

:param data: np.ndarray
:param size:
:param axis:
:param kwargs:
:return:

"""
kwargs.setdefault('mode','constant')

n = data.shape[axis]

padding_len = int((size-n)//2)

lengths = [(0,0)] * data.ndim
lengths[axis] = (padding_len,int(size-n-padding_len))
if padding_len < 0:
```

```
raise Exception(('目标size (:d}) 最少为 ({:d})').format(size, n))
return np.pad(data,lengths,**kwargs)
```

至于为什么要加窗，以及选择汉明窗的理由，已经在前文实验部分详细讲述过了，这里不赘述。简单来说就是要使得帧两端平滑地衰减到 0，以取得更高质量的频谱。**这也解释了为什么相邻帧之间要有 overlap** 那是因为帧与帧之间连接处的信号会因为加窗而弱化，如果没有 overlap，这部分信息会丢失。

4.4. 快速傅里叶变换

信号在时域上的变换难以看出信号的特性，因此要转换为频域上的能量分布来观察。用能量分布来代表不同语音的特性。加窗之后要经过快速傅里叶变换以得到频谱上的能量分布，这样就得到了各帧的频谱，取平方后就得到功率谱。

$$X_a(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi k/N} \quad 0 \le k \le N$$

```
def stft(signal,n_fft=1024,hop_length=None,win_length=1024,center=True,
window='Hamming',pad_mode= 'reflect'):
"""

短时快速傅里叶变换

:param signal: [shape=(n,)] time series of input signal
:param n_fft: FFT WINDOW SIZE
:param hop_length: 即不同窗口之间样本的重叠个数 number audio of frames between STFT
                                        columns deafult: win_length/1
:param win_length:  = nfft window will be of length 'win_length' and then padded
                                        with zeros to match n_fft
:param center: signal padded in the center
:param window:
:param pad_mode:
:return: np.ndarray[shape=(1+n_fft/2,t)]
"""

#预加重
signal = preEmphasis(signal,0.97)

if hop_length == None:
hop_length = int(win_length//2)

#加窗
#fft_window = scipy.signal.get_window(window,win_length,fftbins=True)
fft_window = get_window(window,win_length)
```

```
#窗口补0
fft_window = padding(fft_window,n_fft)

#reshape he window so window can broadcast
fft_window = fft_window.reshape((-1,1))

#信号补零，左右两边补
signal = np.pad(signal,int(n_fft//2),mode=pad_mode)

#分帧
signal_frames = frame(signal,frame_length=n_fft,hop_length = hop_length)

#预分配内存 the STFT matrix
stft_matrix = np.empty((int(1+n_fft//2),signal_frames.shape[1]),dtype=np.complex64,
                                            order='F')

#h
MAX_MEM_BLOCK = 2 ** 8 * 2 ** 10 #256MB

n_columns = int(MAX_MEM_BLOCK / (stft_matrix.shape[0] * stft_matrix.itemsize))

for bl_s in range(0,stft_matrix.shape[1],n_columns):
#加窗并且FFT
bl_t = min(bl_s+n_columns,stft_matrix.shape[1])
stft_matrix[:,bl_s:bl_t] = fft.fft(fft_window*signal_frames[:,bl_s:bl_t],axis=0)[:
                                            stft_matrix.shape[0]]

return stft_matrix
```

4.5. 滤波操作

计算三角滤波的时候公式如下

$$H_m(k) = \begin{cases} 0 & (k < f(m-1)) \\ \frac{k-f(m-1)}{f(m)-f(m-1)} & (f(m-1) \le k \le f(m)) \\ \frac{f(m+1)-k}{f(m+1)-f(m)} & (f(m) < k \le f(m+1)) \\ 0 & (k > f(m+1)) \end{cases}$$

这个公式和代码是对应的。

在预实验阶段我尝试使用 32,64,128 个梅尔滤波器，发现滤波器数量越多，模型准确率越高，于是这里默认使用 128 个梅尔滤波器。

```
def mel(sigRate,n_fft,n_mels=128,fmin = 0.0,fmax = None):
```

```
"""
梅尔滤波

:param sigRate:
:param n_fft:
:param n_mels: 滤波器个数
:param fmin:
:param fmax:
:return:
"""

#initialize
n_mels = int(n_mels)
weights = np.zeros((n_mels,int(1+n_fft//2)))
#最高频率为采样频率一半
fmax = float(sigRate) / 2

#计算每个FFT bin的中心频率
fft_frequencies = np.linspace(0,float(sigRate)/2,int(1+n_fft//2),endpoint=True)

#计算梅尔频率
mel_f = mel_frequencies(n_mels+2,fmin= fmin,fmax=fmax)

# 实际上是返回这么一个vector
# array([(mel_f[2]-mel_f[1])....(mel_f[i+1]-mel_f[i])])
fdiff = np.diff(mel_f)

# 梅尔标度中心频率与实际上的fft_frequency相减
ramps = np.subtract.outer(mel_f,fft_frequencies)

#三角滤波
for i in range(n_mels):

#这里实际上是ppt 的 (k-f(m-1)) / (f(m) - f(m-1))
lower = - ramps[i] / fdiff[i]

# (f(m+1)-k) / (f(m+1) - f(m))
upper = ramps[i+2] / fdiff[i+1]

weights[i] = np.maximum(0,np.minimum(lower,upper))

# slaney-style 归一化
enorm  = 2.0 /(mel_f[2:n_mels+2] - mel_f[:n_mels])
weights*=enorm[:,np.newaxis]

return weights
```

4.6. 计算梅尔频谱

到这一步我们就完成了对梅尔频谱的计算，我们可以用这里提取的特征作为模型的输入了。当然，把这里的特征做离散余弦变换，就可以而得到 mfcc 倒谱系数。

```
def melspectrogram(sigRate = 16000,Spec = None):

"""
计算梅尔频谱
:param sigRate:
:param Spec:
:return:
"""
n_fft = 2*(Spec.shape[0]-1)

mel_basis = mel(sigRate,n_fft)

#梅尔滤波
feat = np.dot(mel_basis,Spec)

feat = np.asarray(feat)

# scaling
# 10 * log10 (10*feat / ref)
magnitude = np.abs(feat)
ref_value = np.max(magnitude)

amin = 1e-10
top_db = 80 #db: decibel (dB) units
#取对数 要模拟人耳对声音响度的敏感特性
log_spec = 10.0 * np.log10(np.maximum(amin,magnitude))
log_spec -= 10.0 * np.log10(np.maximum(amin,ref_value))
log_spec = np.maximum(log_spec,log_spec.max()-top_db)

return log_spec
```

4.7. 计算 MFCC

这里曾经尝试过手动实现离散余弦变换

$$C(n) = \sum_{m=0}^{N-1} s(m) cos(\frac{\pi n(m-0.5)}{M}) n = 1, 2, 3..L$$

但算出来结果与调库相比相差较大，而且查看 scipy 自带的离散余弦变换函数代码发现，其是实现了某种归一化，这种归一化比较复杂，因此没有使用手写的 dct。这里

提取前 40 维系数作为特征，我在实验部分尝试过使用 16 维系数作为特征，发现准确率比不上 40 维的，因此这里默认选择 40 维。

```
def mfcc(Spec,n_mfcc=40):

"""
梅尔频谱取对数之后做离散余弦变换便是mfcc
有考虑过手写dct,但算出来结果与调库不太一样,因此作罢


:param Spec:
:param n_mfcc:
:return:
"""
return scipy.fftpack.dct(Spec,axis=0,type=2,norm='ortho')[:n_mfcc]
```

总结提取流程：

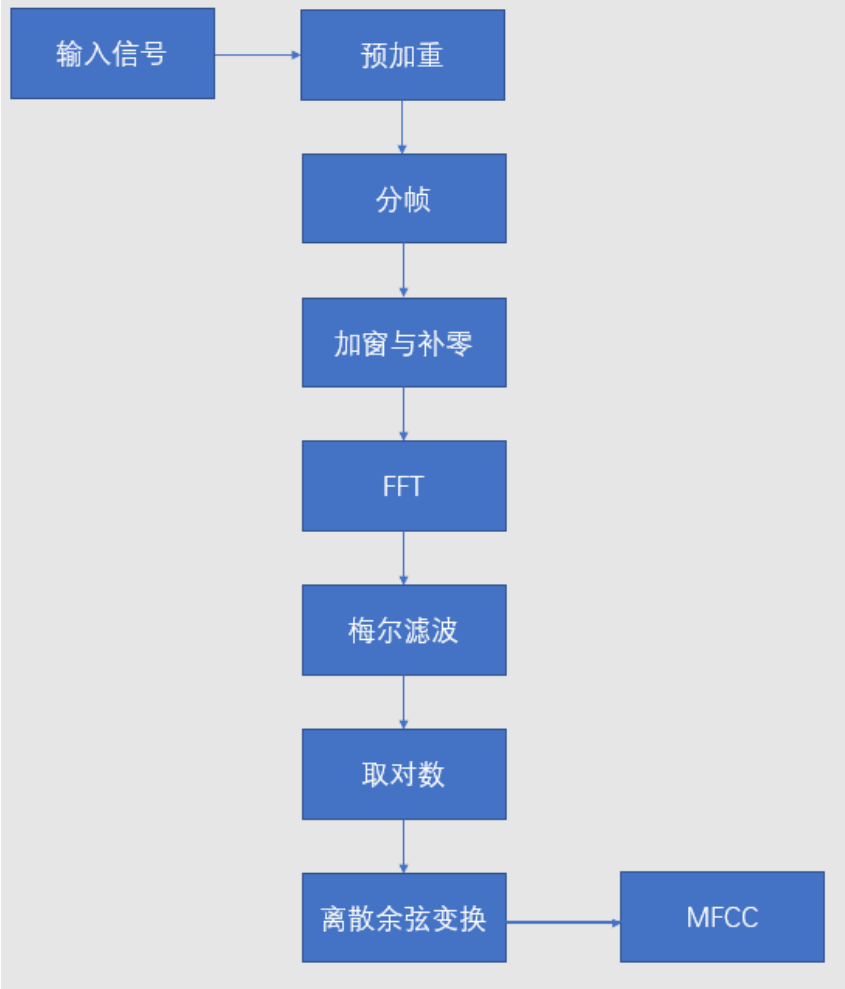

图 3.4:

第 5 节　分类模型

前面我们提到，我们提取到的特征是频谱图或者倒谱图，而我们又是在做多分类任务，所以可以考虑利用在图像识别领域大放异彩的卷积神经网络，把语音识别任务转化为图像识别任务。

而这里我主要使用 VGG 以及 ResNet 这两个比较经典的卷积神经网络架构，这里简单回顾一下这两个网络的架构。而在代码实现部分，一开始我是用 pytorch 官方实现的源码跑的，但想到组里做的是视觉方向，所以我参照官方源码，手动把两个网络的基本架构手写了一遍，添加了很多注释，以加深理解。

5.1. VGG

下图是从原 VGG 的论文中提取出来的结构介绍图。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 3.5:

以 VGG13 为例 (B)，包含了 16 个隐含层（13 个卷积层以及 3 个全连接层）

在结构上主要的创新是采用连续的 3x3 卷积核来做级联，替代如 AlexNet 中较大的卷积核 (11x11,7x7,5x5)，这样可以在保证具有相同感知野的条件下，提升网络的深

度，以学习到更复杂的特征，并且有效减少隐含层参数数量。（注意 VGG 的参数主要在全连接层，而且它有三层全连接层，使得参数数量十分庞大，消耗大量的计算资源。我在使用梅尔频谱作为特征的时候，需要 4 张显卡火力全开才能勉强跑得动）

5.2. ResNet

Residual Network 的提出是为了解决随着网络层数增加，梯度反向传播过程中出现的梯度弥散现象以及层数增加所带来的训练误差增加问题 (退化问题)。

下图是从 ResNet 源论文中提取出来的结构介绍图。

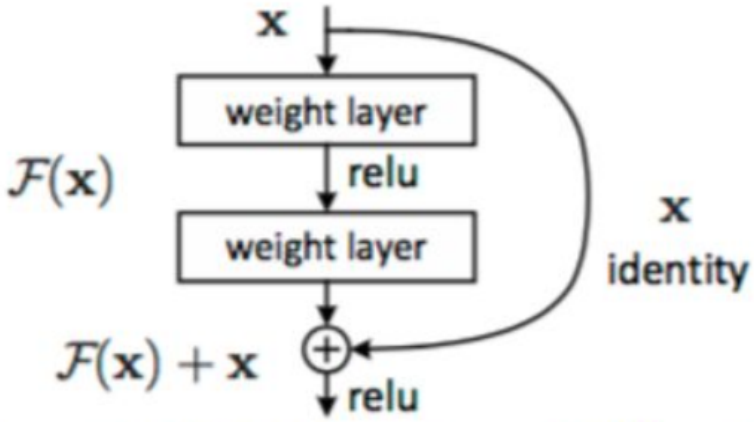

图 3.6:

ResNet 架构的创新体现在残差块中，引入了 shortcut connection，即上图右侧箭头。

优化目标由原来输出 H(x) 变成 H(x)-x ，即输出和输入的差，H（x）为原始的输出。模型学习的是残差函数 F(x) = H(x)-x。

这个结构有两方面好处：

1. 如果没有 shortcut connection，函数最优输出是 x，那么模型需要优化 H(x) 为 x，但如果有了 shortcut connection，只需要把 F(x) = H(x)-x 优化为 0，即优化 F(x) = 0，这显然比优化 H(x) = x 要简单。这解释了其可以解决深层网络训练困难的问题。

2. 反向传播的时候，对 $F(x) + x$ 求导，结果即 $F(x)' + 1$, 我们知道，由于反向传播过程中链式法则的连乘，会导致梯度下降到非常小，以致于忽略不计的情况，这就是梯度弥散。而这里下降到非常小的梯度加上了 1，使得梯度不至于消失，使得梯度信号能够通过网络反向传播到前面层。这解释了其解决了反向传播中梯度弥散的问题，使得深层网络的构建成为可能。

第 6 节　训练过程与结果

我使用了 Pytorch 框架来进行神经网络的搭建，在 Jupyter Notebook 进行调参与实验，并使用 Tensorboard 对训练结果进行可视化。对于提取的两类特征梅尔频谱和 mfcc，分别使用 VGG 和 ResNet 进行了多次实验。实验参数和结果如下表格所示。

6.1. 使用梅尔频谱图作为输入特征

输入特征	模型	epoch	窗口参数	val 准确率	test准确率
梅尔频谱(128维)	VGG11	11	win_length=2048,overlap=512	98.06	93.12
梅尔频谱(128维)	VGG13	9	win_length=2048,overlap=512	98.87	95.5
梅尔频谱(128维)	VGG16	9	win_length=2048,overlap=512	96.3	89.5
梅尔频谱(128维)	VGG13	6	win_length=1024,overlap=256	98.0	93.25
梅尔频谱(128维)	VGG16	9	win_length=1024,overlap=256	97.24	93.4
梅尔频谱(128维)	VGG13	9	win_length=1024,overlap=512	97.8	**97.4**

图 3.7:

注：窗口参数 $win_length = 2048$ 表示一个窗口中采样 2048 个点，overlap=512 表示相邻窗口之间重叠 512 个点，即重叠 1/4

结果分析：

1. 模型越复杂，准确率不一定越高。如分析第一二三行的数据发现，VGG13 的效果比 VGG11 好，但 VGG16 的效果又会比 VGG13 的要差，可以解释为层数越多，模型训练更加困难，出现退化现象。

2. 一开始窗口设置得过大，而且窗口之间的重叠只有 1/4，猜测可能会导致窗口重叠部分一些信息的丢失，因此尝使使用不同大小的窗口，以及加大窗口间的重叠程度，最终在 VGG13 模型上使用 $win_length = 1024$，overlap=512 的参数训练得到测试准确率达 97.4% 的模型。

Tensorboard 可视化

输入特征为梅尔频谱（128 维), 模型为 VGG13 的，参数为 $win_length = 1024, overlap = 512$ 结果可视化

epoch1 与 epoch2 的损失函数以及 Top1，Top5 准确率的变化情况

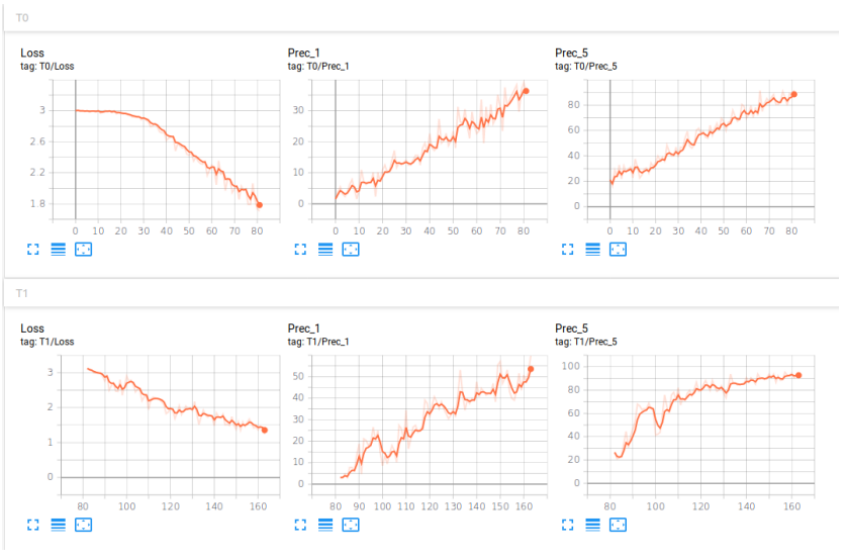

图 3.8:

epoch9 和 epoch10 的损失函数以及 Top1，Top5 准确率的变化情况，可以看出来模型其实早就开始收敛了。

图 3.9:

6.2. 使用 MFCC 作为输入特征

输入特征	模型	epoch	窗口参数	val 准确率	test准确率
mfcc 16维	VGG11	15	win_length=2048,overlap=512	87.08	83.34
mfcc 16维	VGG19	30	win_length=2048,overlap=512	93.83	86.5
mfcc 40维	VGG11	15	win_length=2048,overlap=512	96.583	90.8
mfcc 40维	VGG13	15	win_length=2048,overlap=512	95.3	89.34
mfcc 40维	VGG13	15	win_length=1024,overlap=256	96.8	92.5
mfcc 40维	VGG13	15	win_length=1024,overlap=512	98.1	**93.5**
mfcc+ 短时平均能量 41维	VGG13	15	win_length=1024,overlap=512	98.1	**93.4**
mfcc+ 短时平均能量 41维	ResNet34	30	win_length=1024,overlap=512	98.7	88.1
mfcc+ 短时平均能量 41维	ResNet50	30	win_length=1024,overlap=512	98.18	86.85

图 3.10:

结果分析

1. 一开始只提取了 16 维的 mfcc，可以看见效果并不是特别好，test 准确率上不了 90%, 因此改为提取前 40 维，发现准确率有明显提高。

2. 然后尝试改变窗口长度, 在 $win_length = 1024, overlap = 512$，VGG13 模型下训练得到测试准确率达 93.5% 的模型

3. 接着尝试特征融合，把短时平均能量与 mfcc 结合起来作为输入特征，但发现准确率并没有改变，仍然在 93.5% 左右

4. 最后尝试选用 Residual Network，发现收敛需要的 epoch 数不但增加，而且还出现了过拟合现象：val 准确率达 98%，但测试准确率却达不到 90%,ResNet 模型对于 mfcc 来说可能太复杂，因此后续就没有继续调参做实验了。

Tensorboard 可视化

输入特征为 mfcc+ 短时平均能量（41 维), 模型为 VGG13 的结果可视化

epoch1 与 epoch2 的损失函数以及 Top1，Top5 准确率的变化情况

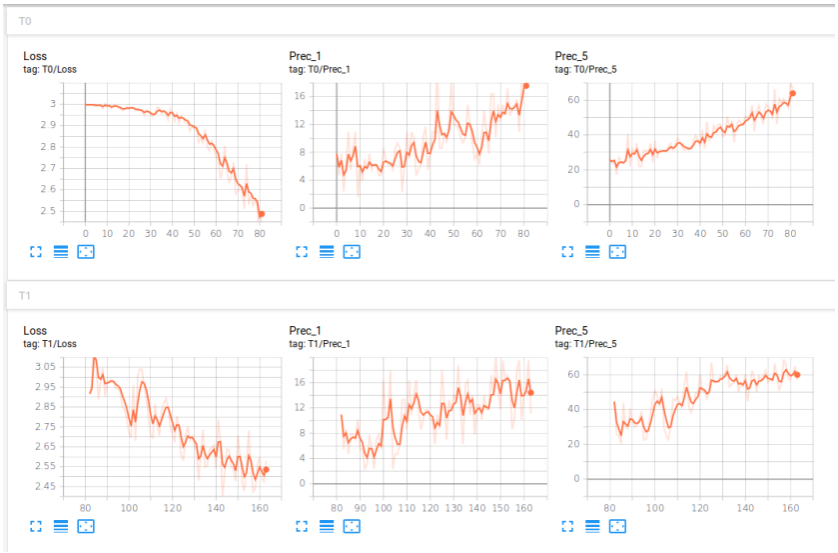

图 3.11:

epoch14 和 epoch15 的损失函数以及 Top1，Top5 准确率的变化情况

图 3.12:

第四章　结论

在本项目中，我分别使用 python 手动实现了提取语音数据的 mfcc 和 mel 频谱图的方法，配合 pytorch 深度学习框架，手动实现了 VGG 和 ResNet 网络，经过多次调参，最终分别实现了一个最高识别精度达 97%(mel 频谱图作为特征) 与 93%(mfcc 作为特征) 的孤立词语音识别系统。

总结一下我学到的知识：

1. 最重要的是对语音识别的整个框架流程有了清晰的认识，同时在手动实现提取 mfcc 和频谱图的过程中，加深了对语音信号特征 (如短时平均能量, 梅尔频谱图,MFCC 等) 原理的理解，以及使用 numpy 的熟练程度。

2. 我重新学习了一遍 VGG 和 ResNet 的大体网络架构，并且加强了我对 Pytorch 搭建模型的熟练程度，也让我对卷积的有关知识有了更深一步的了解。到这一步其实就和做一般的图像识别没什么区别了。

3. 学会如何利用 Python Flask+JavaScript 搭建一个简单的浏览器前端界面，并且了解了 php 如何作为服务端进行工作的原理。这一部分主要加强了我的 javascript 的工程能力。

4. 在比较 VGG 与 ResNet 性能的时候，让我了解到神经网络的层数并不是越深越好，有时候在解决较为简单的问题时，使用深层的网络可能会导致过拟合以及模型难以收敛的现象。我曾统计过使用 ResNet50 作为 baseline 模型的时候，训练时间是使用 VGG13 的两倍以上，而且 VGG 往往在 10 轮以内就已经完全收敛，而 ResNet 通常需要 20-30 个 epoch 才开始收敛。更不用提 ResNet 所占用的内存远比 VGG 要大得多。因此这给我的启发是在解决深度学习任务的时候，要根据给定的任务“因地制宜”地选择合适的模型而不是盲目地去追求层次更深，更复杂的网络。

5. 我在写录音网页端的时候，设置了录音时间为 2s，大概与读一个词的时间相符。但其实还有待改进的部分，即可以在提取特征之前对语音信号做端点检测，并对真正有声音的部分做截取。同时在测试的时候发现系统对环境噪声十分敏感，抗噪声能力比较差。因此可以把语音信号通过 **LMS 滤波器**，从而进行降噪或语音增强。但由于深度学习的强大的参数学习能力，这些更有技术含量的过程即使被省去，也不会对性能结果造成太大的影响。我认为这也是深度学习既迷人又让人感觉不安的一点：一个黑盒般，难以解释的工具，虽然性能非常的好，但从理论研究的角度来看真的有值得研究之处吗？背后有强有力的数理统计理论来支撑吗？因此我对深度学习一直保有谨慎迟疑的态度，同时也认为人工智能的发展不应该仅仅着眼于深度学习这一块。好吧有点扯远了。

参考文献

[1] K.Simon,A.Zisserman,Very Deep Convolutional Networks for Large-Scale Image Recognition,2014.

[2] K.He etal.,Deep Residual Learning for Image Recognition,2015.

[3] suan2014[EB/OL].https://blog.csdn.net/suan2014/article/details/82021324

[4] CS@CMU[EB/OL].$http://www.speech.cs.cmu.edu/15-492/slides/03_mfcc.pdf$,2015

[5] ZytheMoon[EB/OL].https://blog.csdn.net/ZytheMoon/article/details/78751159,2017.

[6] A.Downey,Python 数字信号处理应用, 人民邮电出版社,2018.

[7] Librosa[EB/OL]https://librosa.github.io/librosa/,2018

[8] $Pelhans[EB/OLhttp://pelhans.com/2018/07/04/speeh_process_note2/$, 2018

[9] 胡航, 现代语音信号处理, 电子工业出版社，2014.