同济大学
TONGJI UNIVERSITY

《数字语音处理》

结课论文

# 基于梅尔频率倒谱系数特征提取和卷积神经网络的语音识别

课程名称： 现代通信与信息技术(10226702)

论文题目： 基于梅尔频率倒谱系数特征提取和卷积神经网络的语音识别

学院(系)： 电子与信息工程学院　　任课教师： 赵晓群


姓　　名： 王赢珩

学　　号： 1652448


专　　业： 电子信息工程

上课时间： 星期四　3、4 节　　完成时间： 2019 年 12 月 23 日

2019—2020 学年 第 1 学期

【摘　要】本文基于梅尔频率倒谱系数特征提取的相关方法以及卷积神经网络的深度学习架构实现了一种语音识别模型，该模型主要基于 python 语言和 tensorflow 1.0 的神经网络体系，训练集由作者本人的声音录制而成，测试集既包含其他人的相同内容的语音文件也包含作者本人以不同语气语调录入的相同内容的语音文件，最终预测结果都达到了极高的准确率。

【关键词】梅尔频率倒谱系数 卷积神经网络 语音识别

本文的主体分为四个部分，第一部分介绍了人工智能、深度学习以及将以上框架与语音处理相结合的应用及发展趋势。在第二部分中，本文对模型所涉及的算法原理以及模型概念进行了详细的阐释。而第三部分通过语言叙述和模型可视化对本文模型进行了直观剖析。最后，本文给出了项目完成的结果和相关代码。

## 一、背景分析

### 1.人工智能简介

人工智能是当前的一个热点话题，从当前 Google 旗下的 AlphaGO 到智能汽车，人工智能已经步入我们生活的方方面面。

机器学习是一种实现人工智能的方法，这种方法是用算法来分析数据，然后从中学习，最后对现实做出预测和决策。而深度学习，则是机器学习的一种技术。从上个世纪七八十年代 BP 算法的出现及其在神经网络中的应用，很大推进了机器学习的发展。这种算法基于梯度下降法基础之上，并且适合于多层神经网络之中。这个阶段只包含一层隐藏层节点，因此此阶段被称为浅层学习。到 2006 年以后，随着研究的继续深入，模型包含层次越来越多，深度学习在工程方面的应用得到巨大发展。

### 2.深度学习简介

深度学习，与浅层学习相比，顾名思义，其包含隐藏节点的层数往往在 5 层以上，并且其是通过提取每一层特征，将样本在原来空间的特征变换到一个新的特征空间来表示原来的数据。

深度学习主要分为以下几类：

（1）监督学习。就是用标签的数据调整所有层的权值和阈值，然后对网络进行微调。

（2）非监督学习。与监督学习相反，其是用无标签数据进行每一层预训练，然后将其训练结果作为高一层的输入。

（3）半监督学习。顾名思义，就是将监督学习与非监督学习相结合，部分层采用监督学习，部分层采用非监督学习。此种类型在实际中应用最为广泛。

目前常用的深度学习模型主要有：

（1）卷积神经网络（CNNs）。这是一种前馈神经网络，即各神经元分层排列，每个神经元只与前一层的神经元相连，接受前一层的输出，并输出给下一层。它包括卷积层和池层。目前其主要用来识别位移、缩放和其他形式的二维图形。

（2）递归神经网络（RNNs）。其分为两类，一为时间递归神经网络，其神经元间连接构成有向图；二为结构递归神经网络，利用相似的神经网络结构递归构造更为复杂的深度网络。递归神经网络中，不仅包含前馈连接，还有单元之间的自连接或者到前面层的连接，可以当做短期记忆，使网络记得过去的事情。

（3）限制玻尔兹曼机（RBM）。限制玻尔兹曼机是一种无监督学习模型，子模块有两层，

每层中各节点之间是没有连接的，第一层为可视层，第二层为隐藏层，其关系如图 2.1 所示。一个 RBM 中包含权值、可视层偏置、隐藏层偏置这三个模型参数。

（4）自动编码器（AE）。其同样是一种无监督学习模型，是由自动关联器演变而来的。自动关联器是一种 MLP 结构，其中输出、输入维度一样，并定义输出等于输入。为了能够在输出层重新产生输入，MLP 得找出输入在隐藏层的最佳表示。一旦训练完成，从输入到隐藏层的第一层充当编码器，而隐层单元的值形成编码表示。从隐藏单元到输出单元的第二层充当解码器，由原信号的编码表示重构原信号。

### 3.深度学习在语音处理中的应用

随着人工智能的发展，人与计算机之间的自由交互也变得越来越重要，语音处理则是其中的重要一环。现阶段，语音处理主要包括语音识别、语音合成等技术。

语音识别是一种将人类所表述语言转换成文字的技术，目前国内外许多著名的科技企业，如谷歌、微软、讯飞等都在此领域有深入研究，在生活中，例如苹果 Siri、微软 Cortana 等也被得到广泛应用，极大的方便了人们的生活。

语音识别的过程如下：首先是对输入的训练语音信号进行预处理和提取特征，并训练声学模型；而语言模型则是通过从训练语料学习词或句之间的相互关系，来估计假设词序列的可能性；解码搜索是对测试语音也经过预处理和特征提取后的特征向量序列与若干假设词序列计算声学模型分数与语言模型分数，最后将总体输出分数最高的词序列当做识别结果。

语音合成是通过机械的、电子的方法产生人造语音的技术。百度于 2017 年 3 月推出了实时语音合成神经网络系统（Real-Time Neural Text-to-Speech for Production），定名为 Deep Voice，它由 5 个部分组成：用于定位音素边界的分割模型；用于字素转音素的转换模型；判断音素能持续多长时间的预测模型；基频预测模型；音频合成模型。在同样的 CPU 与 GPU 上，系统比起谷歌 DeepMind 的 WaveNet 要快 400 倍。

过程如下：第一步是将字素转换为音素，利用一个简单的音素字典，把每个句子直接转换为对应的音素；第二步是持续时间的预测，因为音素应该基于上下文来决定它们或长或短的持续时间，另外，还需要做基本频率预测，即图中的 F0。最后一步，就是合并音素、持续时间和频率，得出输出声音。

## 二、算法原理

### 1.梅尔频率倒谱系数特称提取原理

正如前文所述，任何自动语音识别系统的第一步是提取特征，即识别音频信号的组成部分，这些组成部分有助于识别语言内容，并丢弃所有其他携带的诸如背景噪声、情绪等信息的东西。

理解语音的要点是人类产生的声音被声道的形状过滤，包括舌头，牙齿等。这种形状决定了声音是什么样的。 如果我们能够准确地知晓该形状，我们就能准确地表示其产生的音素（phoneme）。声道的形状以短时功率谱的包络的形式表现出来，而 MFCC 的作用就是是准确表示这个包络。

梅尔频率倒谱系数（MFCC）是一种广泛用于自动语音和说话人识别的特征。它们是 Davis 和 Mermelstein 在 20 世纪 80 年代引入的，迄今为止一直都是最先进的。在引入 MFCC 之前，线性预测系数（LPC）和线性预测倒谱系数（LPCC）曾是自动语音识别（Automatic Speech Recognition,ASR）的主要特征类型，特别是对于 HMM 分类器而言。

梅尔频率倒谱系数特征提取的步骤如下：

① 将信号帧化为短帧。

② 对于每一帧，计算功率谱的周期图估计。
③ 将梅尔滤波器组应用于功率谱，将每个滤波器中的能量相加。
④ 取所有滤波器组能量的对数。
⑤ 对对数滤波器组能量求 DCT。
⑥ 保持 DCT 系数 2-13，丢弃其余部分。

为什么要进行上述步骤呢？因为，音频信号通常是不断变化的，为了简化，假设在短时尺度上音频信号没有太大变化（当说它没有变化时意思是统计上平稳，而实际上样本甚至短时间尺度也是不断变化的）。这就是将信号帧化为 20-40ms 帧的原因。如果帧再短的话，将没有足够的样本来获得可靠的频谱估计，如果帧越长，信号在整个帧中变化太大。

下一步是计算每帧的功率谱。这是由人耳蜗（human cochlea)（耳朵中的器官）启发的，它根据进入的声音的频率在不同的点处振动。根据耳蜗中振动的位置，不同的神经会向大脑发出消息，告知大脑存在某些频率。周期图估计为我们执行类似的工作，识别帧中存在哪些频率。

然而，周期图频谱估计仍然包含自动语音识别（ASR）不需要的许多信息。特别地，耳蜗不能辨别两个紧密间隔的频率之间的差异。随着频率的增加，这种效果变得更加明显。出于这个原因，采集了一组周期图 bins 并总结它们以了解不同频率区域中存在多少能量。这是由梅尔滤波器组执行的：第一个滤波器非常窄，并指示在 0 赫兹附近存在多少能量。随着频率越来越高，滤波器越来越宽，因为我们越来越不关心变化。我们只关心每个点大概产生多少能量。 梅尔刻度告诉我们如何间隔滤波器组以及设置它们的宽度为多少。

一旦有了滤波器组能量，就取它们的对数。这也是受人类听觉的启发：我们没有听到线性音阶的响度。通常，为了使声音的感知体积增加一倍，我要将 8 倍的能量投入其中。这意味着如果刚开始时声音很大，实际听起来却没有那么大的区别。这种压缩操作使特征与人类实际听到的功能更加接近。那么问题来了，为什么是对数而不是立方根？对数允许我们使用倒频谱平均减法，这是一种信道归一化技术。

最后一步是计算对数滤波器组能量的 DCT。这么做主要有两个原因。一来是因为滤波器组都是重叠的，所以滤波器组能量彼此非常相关。而利用 DCT 可以实现对能量去相关处理，这意味着对角线协方差矩阵可用于对例如 HMM 分类器的特征进行建模。此外，26 个 DCT 系数中只有 12 个被保留。这是因为较高的 DCT 系数代表滤波器组能量的快速变化，并且事实证明这些快速变化实际上是降低了 ASR 性能的，因此我们通过丢弃它们获得了小的改进。

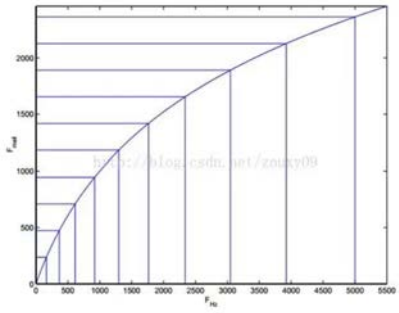
这里又涉及到另外一个问题，什么是梅尔尺度？梅尔音阶将纯音的感知频率或音调，与其实际测量频率相关联。人们在识别低频时音高的微小变化方面比在高频时更好。结合这种尺度，该特征与人类听到的特征更加接近。频率与梅尔频率的转换公式如公式（1）所示：

$$M(f) = 1125\ln(1 + f/700) \qquad (1)$$

反过来如公式（2）所示：

$$M^{-1}(m) = 700(\exp(m/1125) - 1) \qquad (2)$$

由下图可以看到，它可以将不统一的频率转化为统一的频率，也就是统一的滤波器组。




在梅尔频域内，人对音调的感知度为线性关系。举例来说，如果两段语音的梅尔频率相差两倍，则人耳听起来两者的音调也相差两倍。

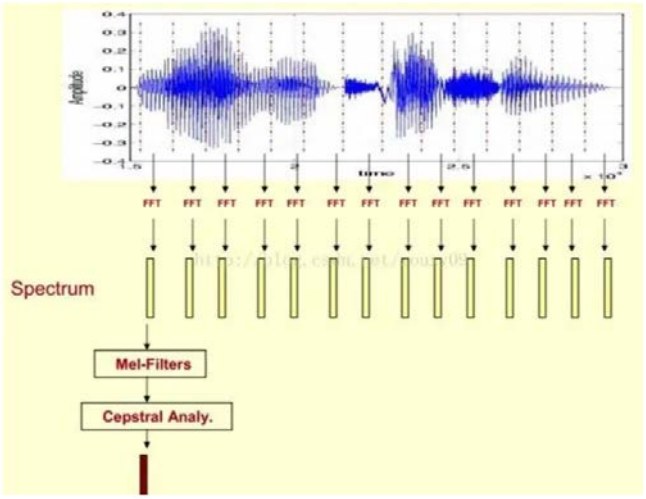
下面介绍梅尔频率倒谱系数的获取方法：

我们将频谱通过一组梅尔滤波器就得到梅尔频谱。公式表述就是：log X[k]=log (Mel-Spectrum)。这时候我们在 log X[k]上进行倒谱分析：

1）取对数：log X[k] = log H[k] + log E[k]。

2）进行逆变换：x[k] = h[k] + e[k]。

在梅尔频谱上面获得的倒谱系数 h[k]就称为梅尔频率倒谱系数，简称 MFCC。




具体过程如下：

1）先对语音进行预加重、分帧和加窗：

分帧：为了方便对语音分析，可以将语音分成一个个小段，称之为：帧。先将 N 个采样点集合成一个观测单位，称为帧。通常情况下 N 的值为 256 或 512，涵盖的时间约为 20~30ms 左右。为了避免相邻两帧的变化过大，因此会让两相邻帧之间有一段重叠区域，此重叠区域包含了 M 个取样点，通常 M 的值约为 N 的 1/2 或 1/3。通常语音识别所采用语音信号的采样频率为 8KHz 或 16KHz，以 8KHz 来说，若帧长度为 256 个采样点，则对应的时间长度是 256/8000×1000=32ms。

加窗：语音在长范围内是不停变动的，没有固定的特性无法做处理，所以将每一帧代入窗函数，窗外的值设定为 0，其目的是消除各个帧两端可能会造成的信号不连续性。常用的窗函数有方窗、汉明窗和汉宁窗等，根据窗函数的频域特性，常采用汉明窗。

2）对每一个短时分析窗，通过 FFT 得到对应的频谱；（获得分布在时间轴上不同时间窗内的频谱）

3）将上面的频谱通过梅尔滤波器组得到梅尔频谱；（通过梅尔频谱，将线形的自然频谱转换为体现人类听觉特性的梅尔频谱）

4）在梅尔频谱上面进行倒谱分析（取对数，做逆变换，实际逆变换一般是通过 DCT 离散余弦变换来实现，取 DCT 后的第 2 个到第 13 个系数作为 MFCC 系数），获得梅尔频率倒谱系数 MFCC，这个 MFCC 就是这帧语音的特征；（倒谱分析，获得 MFCC 作为语音特征）

这时候，语音就可以通过一系列的倒谱向量来描述了，每个向量就是每帧的 MFCC 特征向量。

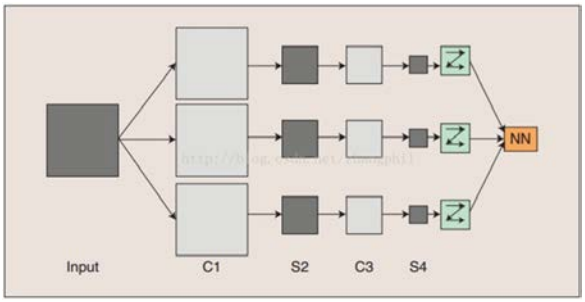
## 2. 卷积神经网络原理

卷积神经网络是人工神经网络的一种，卷积神经网络是机器深度学习中的一种“前馈神经网络”，前馈是信号量的输入获得，到输出过程是往前方传导的，简言之信号是往前方传递的，所以称之为前馈。前馈用以神经网络的计算输出，不对神经网络调整，每一层中每一个神经元算出该层的输出并向下一层传递到输出层，进而计算出网络的输出结果。Back Propagation 神经网络，即 BP 神经网络。反向传播训练神经网络权值和阈值的调整。网络前

向传递计算输出结果时与正确结果存在误差，因此需要 Back Propagation 调整神经网络的前向计算过程。

卷积神经网络本质上是一种输入到输出的映射网络，它能够学习大量的输入与输出之间的映射关系，而不需要任何输入和输出之间的精确的数学表达式，只要用已知的模式对卷积神经网络加以训练，神经网络就具有输入输出之间的映射能力。卷积神经网络执行的是有监督训练，所以其样本集是由形如：（输入向量，理想输出向量）的向量对构成。这些向量对，可以是从实际运行系统中采集来。在开始训练前，所有的权重都应该用一些不同的小随机数进行初始化。小随机数用来保证神经网络不会因权值过大而进入饱和状态，从而导致训练失败，权值不同用来保证神经网络可以正常地学习。事实上，如果用相同的权值去初始化矩阵，则神经网络无能力学习。

卷积神经网络是一个多层的神经网络，每层由多个二维平面组成，而每个平面由多个独立神经元组成。卷积神经网络的神经元感知周围神经单元。



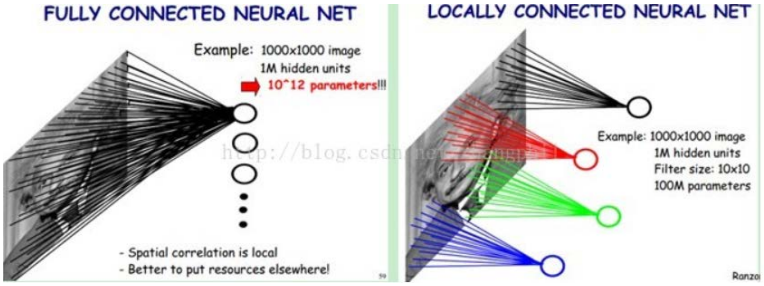

输入数据通过和三个可训练的滤波器和可加偏置进行卷积，滤波过程如图，卷积后在 C1 层产生三个特征映射图，然后特征映射图中每组的四个像素再进行求和，加权值，加偏置，通过一个 Sigmoid 函数得到三个 S2 层的特征映射图。这些映射图再进入滤波器得到 C3 层。这个层级结构再和 S2 一样产生 S4。最终，这些像素值被光栅化，并连接成一个向量输入到传统的神经网络，得到输出。

C 层为特征提取层，每个神经元的输入与前一层的局部感受器（探测器）相连，并提取该局部的特征，一旦该局部特征被提取后，它与其他特征间的位置关系也随之确定下来；S 层是特征映射层，卷积神经网络的每个计算层由多个特征映射组成，每个特征映射为一个平面，平面上所有神经元的权值相等。卷积运算可以使原信号特征增强，并且降低噪音，这种特征提取结构在识别时对输入样本有较高的畸变容忍能力。

卷积神经网络包含卷积层和池化层。卷积神经网络特征检测层通过数据训练进行学习，避免显示的特征提取，隐式地从训练数据中进行深度学习，同一特征映射面上的神经元权值相同，意味着可以并行训练学习。流行的分类方式多是基于统计特征，这意味着在进行分类前必须提取某些特征。然而，显式的特征提取并不容易，在一些实际问题中也并非总是可靠的。卷积神经网络，避免了显式的特征取样，隐式地从训练数据中进行学习。这使得卷积神经网络明显有别于其他基于神经网络的分类器，通过结构重组和减少权值将特征提取功能融合进多层感知器。

卷积神经网络降低参数数量，目的是降低计算负荷压力。其中一个策略是局部感知，以图像识别为例，如果把一个 1000 X 1000 的图像作为输入参数，每个隐层神经元都连接感知图像的每一个像素点，如果全连接，那数据就太多，1000x1000x1000000=10^12，计算压力太大，所以要考虑一种有效的方式。局部感知可以解决这个问题，局部感知基于以下基本假设：人类的大脑在认知周围世界时候，是从局部到全局的。任一图像中，某一特定像素点和局部周围的像素关系紧密，而和较远距离的全局像素关系不是非常密切，即平面图像的空间联系是局部的。每一个神经元，没有必要对全局的像素点都要认知，一不必要，二是资源浪费。每一个神经元都无须对全局图像做感受，每个神经元只感受局部的图像，然后在更高层，这些感受到不同局部的神经元综合起来就可以得到全局信息。

神经网络中的参数经过局部感知后，有些时候数据集可能仍然非常大。此时可以考虑第二种方案，权值共享。假设图像中有 1000X1000=1000000（一百万）个像素点，经过处理，每个神经元 100（卷积操作）个特征相同，那么此时要处理的 1000000 数据集问题就缩减为处理 100 的问题，数据量大大降低。局部感受野是 10X10，隐层每个感受野只需要和这 10X10 的局部图像相连接，所以 1 百万个隐层神经元就只有一亿个连接，即 10^8 个参数。比原来降低四个 0（数量级）。

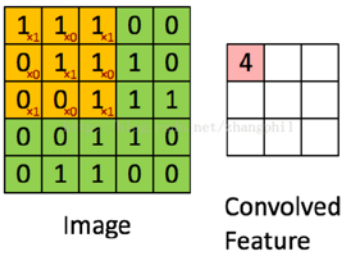



隐层的每一个神经元都连接 10X10 个像素点，即每一个神经元存在 10 X 10=100 个连接权值参数。如果每个神经元这 100 个参数是相同的呢？也就是说每个神经元用的是同一个卷积核去卷积图像。这样我们就只有 100 个参数，不管隐含层的神经元个数有多少，两层间的连接我只有 100 个参数，这就是卷积神经网络权值共享。

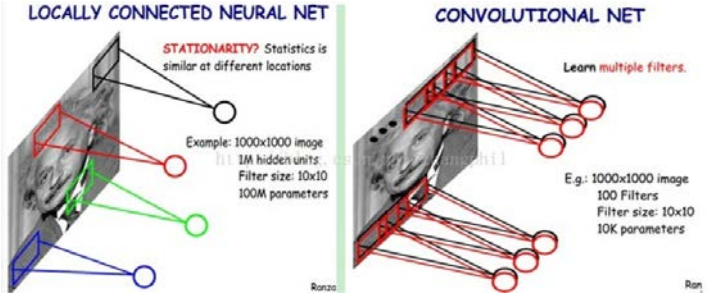
权值共享降低了网络的复杂度，避免了特征提取和分类过程中模型重建开销。权值共享的方式：假设针对图像，设定 100 个提取（学习）特征，这 100 个提取特征与图像中的不同区域及位置没有关系，是打算要深度学习的特征。图像的一块（局部感受野）作为层级结构中最低层的输入，信号再依次传输到不同的层，每层通过一个数字滤波器（或探测器）去获取感知数据的最显著的特征。

举例，假设从一个图像中取出 8 X 8 的像素样本中首先训练机器学习到的特征作为探测器，然后再回归到图像中的任一区域。举例，假设从图像中选取一小块 8 X 8 的区域，通过对 8 X 8 的小区域学习到的特征与 1000 X 1000 的图形做卷积，从而就得到该完整图像中任一位置的有价值的输出值，激活值。

下图展示了 3 X 3 的卷积核在 5 X 5 的图像上做卷积的过程。每一次卷积均以固定的算式算出激活值，假设激活值越大越符合要求的条件，那么通过这种筛选方式就可以筛出需要的图像：



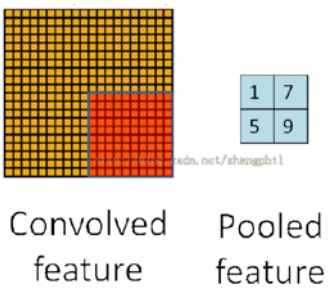

实际的运用中，往往要多层卷积，多层卷积越往层级高的地方，特征越具有全局意义，单层卷积学习到的特征更多的是局部性特征。如果用 100 个参数时，只有一个 10 X 10 的卷积核，这样的特征提取是不充分的。这样只提取一种特征，现实中需要提取多种特征。假如一种滤波器，也就是一种卷积核，去提取图像的一种特征，比如某个方向边缘。那需要提取不同的特征，就需要加多几种滤波器（探测器）。所以当加到 100 种滤波器，每种滤波器的参数不一样，表示它提出输入图像的不同特征，例如不同的边缘。这样每种滤波器去卷积完整图像就得到对图像不同特征的映射 Feature Map。因此 100 种卷积核就有 100 个 Feature Map。这 100 个 Feature Map 就组成了一层神经元。100 种卷积核乘以每种卷积核共享 100 个参数=100x100=10K，每一层也就是 1 万个参数。




再假设针对图像设定更多的卷积核，比如 32 个，这也就意味着可以学习 32 个特征。多个卷积核时候，32 个卷积核，生成 32 幅图像。这 32 幅图可以认为是完整图像的不同通道。

通过卷积算出特征后，开始利用特征进行分类，理论上可以用所有提取到的特征训练分拣器，但这里计算机要处理的数据太多太多。比如一个 96 X 96 像素图像，假设已经学习得到 400 个定义的 8 X 8 输入的特征，每一个特征和图像卷积都会得到一个 (96- 8+1)×(96 - 8+1)=7921 维的卷积特征，由于有 400 个特征，所以每个样例就会得到一个 7921 × 400 = 3,168,400 维的卷积特征向量。学习一个拥有超过三百多万特征输入的分类器任务量太大，并且容易出现过拟合 (over-fitting)。

卷积后的特征值，在完整图像的不同区域极有可能仍然适用，因此为了描述一个大图像的特征，可以针对不同位置的特征进行聚合统计。比如可以计算图像中某一块区域的平均值（或者最大值），这些概略性的特征统计，第一可以大大降低需要计算的维度，第二还可以从一定程度上改善拟合结果。这种聚合的计算过程就叫做池化 (Pooling)，或称之为平均池化或者最大池化 (依据计算池化的方式)。



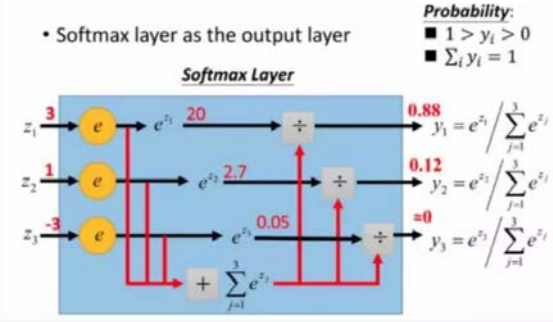

### 3. softmax 分类器原理

Softmax 是用于分类过程，用来实现多分类的，简单来说，它把一些输出的神经元映射到（0-1）之间的实数，并且归一化保证和为 1，从而使得多分类的概率之和也刚好为 1。这是一种较为通俗的解释，当然我们也可以直接从这个名字入手去解释，Softmax 可以分为 soft 和 max，max 也就是最大值，假设有两个变量 a,b。如果 a>b，则 max 为 a，反之为 b。那么在分类问题里面，如果只有 max，输出的分类结果只有 a 或者 b,是个非黑即白的结果。但是在现实情况下，我们希望输出的是取到某个分类的概率，或者说，我们希望分值大的那一项被经常取到，而分值较小的那一项也有一定的概率偶尔被取到，所以我们就应用到了 soft 的概念，即最后的输出是每个分类被取到的概率。

函数定义如下：

$$S_i = \frac{e^{V_i}}{\sum_i^C e^{V_i}}$$

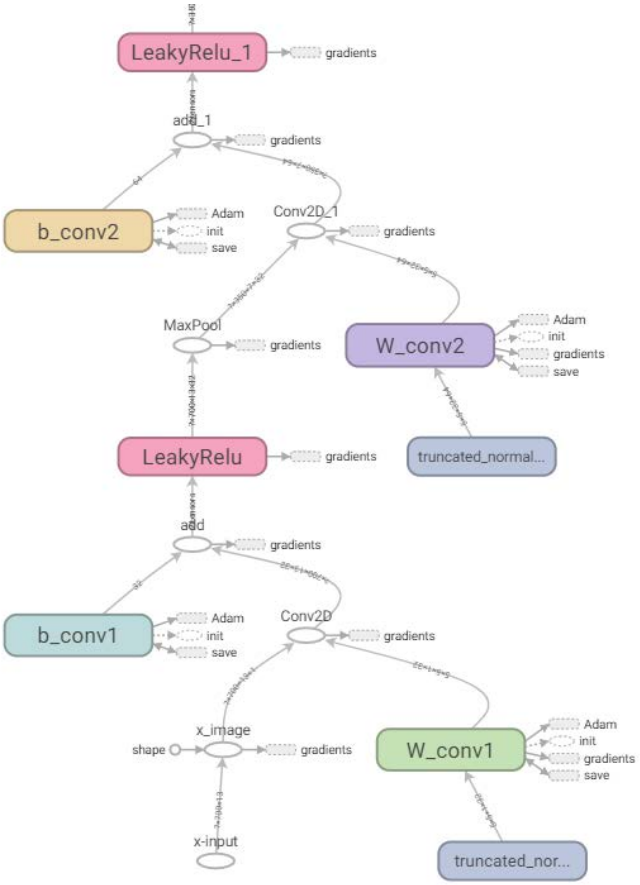
其中，Vi 是分类器前级输出单元的输出。i 表示类别索引，总的类别个数为 C。Si 表示的是当前元素的指数与所有元素指数和的比值。通过这个 Softmax 函数，就可以将多分类的输出数值转化为相对概率。



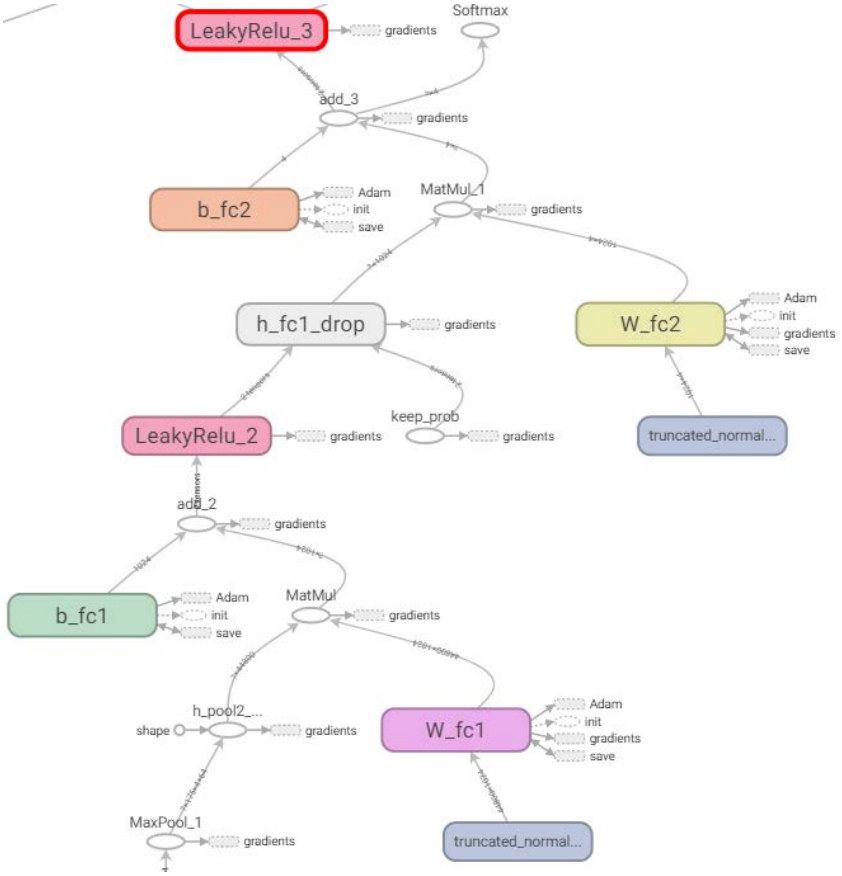

## 三、模型概述

模型主要分为三部分：先导入音频文件，对其进行梅尔频率倒谱系数特征提取的处理，得到预处理过的音频特征作为卷积神经网络的输入；之后构建一个七层的卷积神经网络（包含两个卷积层，两个池化层，两个全连接层和一个 dropout 层），得到输出后将其输入至 softmax 分类器中，通过最小化交叉熵损失函数进行反向梯度传导从而优化更新神经网络内部的参数，利用 tensorboard 对模型进行可视化如下：

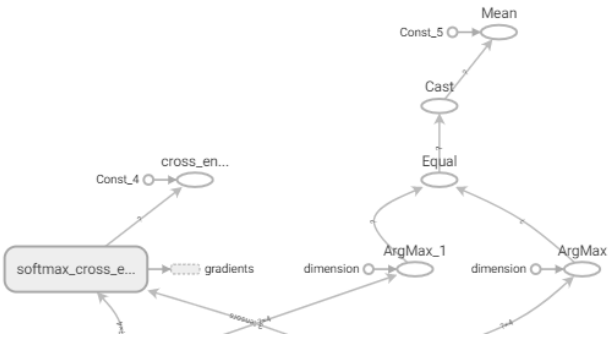
1） 两个卷积层和两个池化层：




2） 两个全连接层和一个 dropout 层：




3） softmax 层以及准确度计算：




## 四、预测结果

### 1. 先利用训练集将模型训练至 100%准确度

```
训练第 15 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 16 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 17 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 18 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 19 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 20 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 21 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 22 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 23 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 24 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 25 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 26 次, 训练集准确率= 0.6 , 测试集准确率= 0.6666667
训练第 27 次, 训练集准确率= 0.7 , 测试集准确率= 0.6666667
训练第 28 次, 训练集准确率= 0.7 , 测试集准确率= 0.6666667
训练第 29 次, 训练集准确率= 0.8 , 测试集准确率= 1.0
训练第 30 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 31 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 32 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 33 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 34 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 35 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 36 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 37 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 38 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 39 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 40 次, 训练集准确率= 0.9 , 测试集准确率= 1.0
训练第 41 次, 训练集准确率= 1.0 , 测试集准确率= 1.0
训练集和测试集准确率均达到100%
```

### 2. 再对新的测试集进行预测，观察其准确率及置信度

```
取值置信度[[0.03801883 0.37819284 0.22740094 0.3563874 ]]
实际 :1 ,预测: 1 ,预测可靠度: 0.37819284
取值置信度[[0.00568614 0.22568057 0.0934616  0.6751717 ]]
实际 :3 ,预测: 3 ,预测可靠度: 0.6751717
```

由上可以看出，本模型的预测准确率在作者所使用的数据集中可达到 100%，由于作者使用的数据集囊括了多种音源及内容的音频数据，因此，可以说明该模型应该具有良好的泛化性能。

## 五、参考文献及代码

### 1. 参考文献


- 《机器学习》
- http://www.speech.cs.cmu.edu/15-492/slides/03_mfcc.pdf


### 2. 代码

```python
import tensorflow as tf
import scipy.io.wavfile as wav
from python_speech_features import mfcc,delta
import os
import numpy as np
import sklearn.preprocessing

path_film = os.path.abspath('.')
path = path_film + "/data/xunlian/"
test_path = path_film + "/data/test_data/"
isnot_test_path = path_film + "/data/isnot_test_path/"

# 使用 one-hot 编码，将离散特征的取值扩展到了欧式空间
# 全局 one-hot 编码空间
```

```
label_binarizer = ""
def def_one_hot(x):
    if label_binarizer == "":
        binarizer = sklearn.preprocessing.LabelBinarizer()
    else:
        binarizer = label_binarizer
    binarizer.fit(range(max(x)+1))
    y= binarizer.transform(x)
    # 标签二值化：以 ont-hot 向量表示多类标签
    return y

def read_wav_path(path):
    # 前面是路径，后面是文件名
    map_path, map_relative = [str(path) + str(x) for x in os.listdir(path) if os.path.isfile(str(
path) + str(x))], [y for y in os.listdir(path)]
    return map_path, map_relative

def def_wav_read_mfcc(file_name):
    # 对 wav 文件进行 mfcc 操作
    fs, audio = wav.read(file_name)
    processed_audio = mfcc(audio, samplerate=fs, nfft=2000)
    return processed_audio

def matrix_make_up(audio):
    # 构建 mfcc 矩阵
    new_audio = []
    for aa in audio:
        zeros_matrix = np.zeros([700, 13],np.int8)
        a, b = np.array(aa).shape
        for i in range(a):
            for j in range(b):
                zeros_matrix[i, j] = zeros_matrix[i, j] + aa[i, j]
        new_audio.append(zeros_matrix)
    return new_audio, 700, 13

def read_wav_matrix(path):
    # 对 wav 文件进行 mfcc 操作并获取其 labels 和 shape
    map_path, map_relative = read_wav_path(path)
    audio=[]
    labels=[]
    for idx, folder in enumerate(map_path):
        processed_audio_delta = def_wav_read_mfcc(folder)
        audio.append(processed_audio_delta)
        labels.append(int(map_relative[idx].split(".")[0].split("_")[0]))
    x_data, h, l = matrix_make_up(audio)
    x_data = np.array(x_data)
    x_label = np.array(def_one_hot(labels))
    return x_data, x_label, h, l

# 初始化权值
def weight_variable(shape,name):
    initial = tf.truncated_normal(shape,stddev=0.01)  # 生成一个截断的正态分布
    return tf.Variable(initial,name=name)

# 初始化偏置
def bias_variable(shape,name):
    initial = tf.constant(0.01,shape=shape)
    return tf.Variable(initial,name=name)

# 卷积层
```

```
def conv2d(x,W):
    # x input tensor of shape `[batch, in_height, in_width, in_channels]`
    # W filter / kernel tensor of shape [filter_height, filter_width, in_channels, out_channels]
    # `strides[0] = strides[3] = 1`. strides[1]代表x方向的步长，strides[2]代表y方向的步长
    # padding: A `string` from: `"SAME", "VALID"`
    return tf.nn.conv2d(x,W,strides=[1,1,1,1],padding='SAME')

# 池化层
def max_pool_2x2(x):
    # ksize [1,x,y,1]
    return tf.nn.max_pool(x,ksize=[1,2,2,1],strides=[1,2,2,1],padding='SAME')

def train_main(path,test_path):
    x_train, y_train, h, l = read_wav_matrix(path)
    x_test, y_test, h, l = read_wav_matrix(test_path)

    print(x_train.shape)

    m,n = y_train.shape
    # 命名空间
    # 定义两个placeholder
    x = tf.placeholder(tf.float32, [None, h, l], name='x-input')
    y = tf.placeholder(tf.float32, [None, n], name='y-input')
    # 改变x的格式转为4D的向量[batch, in_height, in_width, in_channels]`
    x_image = tf.reshape(x, [-1, h, l, 1], name='x_image')

    # 初始化第一个卷积层的权值和偏置
    W_conv1 = weight_variable([5, 5, 1, 32], name='W_conv1')  # 5*5的采样窗口，32个卷积核从3个平面抽取特征
    b_conv1 = bias_variable([32], name='b_conv1')  # 每一个卷积核一个偏置值

    # 把x_image和权值向量进行卷积，再加上偏置值，然后应用于relu激活函数
    conv2d_1 = conv2d(x_image, W_conv1) + b_conv1
    h_conv1 = tf.nn.leaky_relu(conv2d_1)
    h_pool1 = max_pool_2x2(h_conv1)  # 进行max-pooling

    # 初始化第二个卷积层的权值和偏置
    W_conv2 = weight_variable([5, 5, 32, 64], name='W_conv2')  # 5*5的采样窗口，64个卷积核从32个平面抽取特征
    b_conv2 = bias_variable([64], name='b_conv2')  # 每一个卷积核一个偏置值

    # 把h_pool1和权值向量进行卷积，再加上偏置值，然后应用于relu激活函数

    conv2d_2 = conv2d(h_pool1, W_conv2) + b_conv2
    h_conv2 = tf.nn.leaky_relu(conv2d_2)
    h_pool2 = max_pool_2x2(h_conv2)  # 进行max-pooling

    # 300*300的图片第一次卷积后还是300*300，第一次池化后变为150*150
    # 第二次卷积后为150*150，第二次池化后变为了75*75
    # 进过上面操作后得到64张75*75的平面

    # 初始化第一个全连接层的权值
    W_fc1 = weight_variable([175 * 4 * 64, 1024], name='W_fc1')  # 上一场有75*75*64个神经元，全连接层有1024个神经元
    b_fc1 = bias_variable([1024], name='b_fc1')  # 1024个节点

    # 把池化层2的输出扁平化为1维
    h_pool2_flat = tf.reshape(h_pool2, [-1, 175 * 4 * 64], name='h_pool2_flat')

    # 求第一个全连接层的输出
```

```
    wx_plus_b1 = tf.matmul(h_pool2_flat, W_fc1) + b_fc1
    h_fc1 = tf.nn.leaky_relu(wx_plus_b1)

    # keep_prob 用来表示神经元的输出概率
    keep_prob = tf.placeholder(tf.float32, name='keep_prob')
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob, name='h_fc1_drop')

    # 初始化第二个全连接层
    W_fc2 = weight_variable([1024, n], name='W_fc2')
    b_fc2 = bias_variable([n], name='b_fc2')
    wx_plus_b2 = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

    # 计算输出
    prediction = tf.nn.leaky_relu(wx_plus_b2)

    tf.add_to_collection('predictions', prediction)

    p = tf.nn.softmax(wx_plus_b2)

    tf.add_to_collection('p', p)

    # 交叉熵代价函数
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=predi
ction),
                                   name='cross_entropy')

    # 使用 AdamOptimizer 进行优化
    train_step = tf.train.AdamOptimizer(1e-5).minimize(cross_entropy)
    # train_step = tf.train.GradientDescentOptimizer(5).minimize(cross_entropy)

    # 求准确率
    # 结果存放在一个布尔列表中
    correct_prediction = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))  # argmax 返回一维张量
中最大的值所在的位置
    # 求准确率
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    # 保存模型使用环境
    saver = tf.train.Saver()

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        # 创建一个协调器，管理线程
        coord = tf.train.Coordinator()
        # 启动 QueueRunner，此时文件名队列已经进队
        threads = tf.train.start_queue_runners(sess=sess, coord=coord)

        for i in range(100001):
            # 训练模型
            sess.run(train_step, feed_dict={x: x_train, y: y_train, keep_prob: 1.0})
            # 存储 graph 至 tensorboard
            summary_writer = tf.summary.FileWriter('./log/', sess.graph)

            test_acc = sess.run(accuracy, feed_dict={x: x_test, y: y_test, keep_prob: 1.0})
            train_acc = sess.run(accuracy, feed_dict={x: x_train, y: y_train, keep_prob: 1.0})
            print("训练第 " + str(i) + " 次，训练集准确率= " + str(train_acc) + " ，测试集准确率
= " + str(test_acc))

            if test_acc == 1 and train_acc >= 0.95:
```

```
                print("训练集和测试集准确率均达到100%")
                # 保存模型
                saver.save(sess, 'nn/my_net.ckpt')
                break

        # 通知其他线程关闭
        coord.request_stop()
        # 其他所有线程关闭之后，这一函数才能返回
        coord.join(threads)


def test_main(isnot_test_path):
    # 本地情况下生成数据
    x_test, y_test, h, l = read_wav_matrix(isnot_test_path)
    m,n = y_test.shape

    # 迭代网络
    with tf.Session() as sess:
        # 保存模型使用环境
        saver = tf.train.import_meta_graph("nn/my_net.ckpt.meta")
        saver.restore(sess, 'nn/my_net.ckpt')

        predictions = tf.get_collection('predictions')[0]
        p = tf.get_collection('p')[0]

        graph = tf.get_default_graph()

        input_x = graph.get_operation_by_name('x-input').outputs[0]
        keep_prob = graph.get_operation_by_name('keep_prob').outputs[0]

        for i in range(m):
            result = sess.run(predictions, feed_dict={input_x: np.array([x_test[i]]),keep_prob:1.
0})
            pre_prob = sess.run(p, feed_dict={input_x: np.array([x_test[i]]), keep_prob: 1.0})
            print("取值置信度"+str(pre_prob))
            print("实际 :"+str(np.argmax(y_test[i]))+" ,预测: "+str(np.argmax(result))+" ,预测可靠
度: "+str(np.max(pre_prob)))

if __name__ == '__main__':

    train_main(path,test_path)
    # test_main(isnot_test_path)
```