语音识别学习笔记

学习大法好

作者：杜沪

组织：BIT, SpeechOcean

时间：September 4, 2019

版本：0.1

ElegantLaTeX Program

Victory won't come to us unless we go to it. — M. Moore

目　　录

插图目录

表格目录

第 1 章 HMM 相关知识点总结

1.1 basic infomation

设 Q 是所有可能状态的集合，V 是所有可能观测的集合。其中 N 是可能的状态数，M 是可能的观测数。

$$Q = \{q_1, q_2, ..., q_N\}, V = \{v_1, v_2, ..., v_M\}$$

I 是长度为 T 的状态序列，O 是对应的观测序列。

$$I = \{i_1, i_2, ..., i_T\}, O = \{o_1, o_2, ..., o_T\}$$

A 为状态转移矩阵，如公式1.1，其中 $a_{ij} = P(i_{t+1} = q_j | i_t = q_i)$，$i = 1, 2, ..., N; j = 1, 2, ..., N$，是在时刻 t 处于状态 q_i 的条件下在时刻 $t + 1$ 转移到状态 q_j 的概率。

$$A = [a_{ij}]_{N \times N} \tag{1.1}$$

B 是观测概率矩阵，如公式1.2，其中 $b_j(k) = P(o_t = v_k | i_t = q_j)$，$k = 1, 2, ..., M; j = 1, 2, ..., N$ 是 t 时刻处于状态 q_j 的条件下生成观测 v_k 的概率。

$$B = [b_j(k)]_{N \times M} \tag{1.2}$$

π 是初始状态概率向量，如公式1.3，其中 $\pi_i = P(i_1 = q_i)$，$i = 1, 2, ..., N$。

$$\pi = (\pi_i) \tag{1.3}$$

HMM 有三个基本问题：

（1）概率计算问题。给定模型 $\lambda = (A, B, \pi)$ 和观测序列 $O = (o_1, o_2, ..., o_T)$，计算在模型 λ 的条件下观测序列 O 出现的概率 $P(O|\lambda)$。

（2）学习问题。已知观测序列 $O = (o_1, o_2, ..., o_T)$，估计模型 $\lambda = (A, B, \pi)$ 参数，使得在该模型下观测序列 $P(O|\lambda)$ 最大，即用最大似然估计的方法估计参数。

（3）预测问题，也称为解码问题。已知模型 $\lambda = (A, B, \pi)$ 和观测序列 $O = (o_1, o_2, ..., o_T)$，求对给定观测序列条件概率 $P(I|O)$ 最大的状态序列 $I = \{i_1, i_2, ..., i_T\}$，即给定观测序列，求最有可能的对应的状态序列。

1.2 概率计算问题

1.2.1 前向算法

1.2.2 后向算法

1.3 学习问题

1.4 解码问题

第 2 章 微软 Edx 语音识别笔记

本章笔记主要是对微软 Edx 的课程Speech Recognition System的记录，首版主要是翻译，再加上自己翻阅其他资料综合起来的一些思考和总结。代码见Speech-Recognition

2.1 Background and Fundamentals

2.1.1 Phonetics

Phonetics（语音学）是 Linguistics（语言学）的一个分支，其研究的是人类语音发出的声音（sound）。语音学围绕着声音的产生（通过人类的发音器官）、声音的声学特性和感知。语音学有三个基本的分支，这三个分支都与 ASR 有关系。

1. Articulatory Phonetics（发音语音学）：通过发音器官、不同说话人而产生的声音；
2. Acoustic Phonetics（声学语音学）：声音从说话人到听者的传输；
3. Auditory Phonetics（听觉语音学）：听者对于声音的接收和感知。

声音的最小单元我们成为Phoneme，即音素。序列中的词（Words）是由一个或多个音素组成的。一个音素的声学实现称为Phone。图2.1展示了美式英语的音素和一般实现办法。

Phonemes	Word Examples	Description
ih	*fill, hit, lid*	front close unrounded (lax)
ae	*at, carry, gas*	front open unrounded (tense)
aa	*father, ah, car*	back open unrounded
ah	*cut, bud, up*	open-mid back unrounded
ao	*dog, lawn, caught*	open-mid back round
ay	*tie, ice, bite*	diphthong with quality: aa + ih
ax	*ago, comply*	central close mid (schwa)
ey	*ate, day, tape*	front close-mid unrounded (tense)
eh	*pet, berry, ten*	front open-mid unrounded
er	*turn, fur, meter*	central open-mid unrounded rhoti-
ow	*go, own, tone*	back close-mid rounded
aw	*foul, how, our*	diphthong with quality: aa + uh
oy	*toy, coin, oil*	diphthong with quality: ao + ih
uh	*book, pull, good*	back close-mid unrounded (lax)
uw	*tool, crew, moo*	back close round
b	*big, able, tab*	voiced bilabial plosive
p	*put, open, tap*	voiceless bilabial plosive
d	*dig, idea, wad*	voiced alveolar plosive
t	*talk, sat*	voiceless alveolar plosive &
t	*meter*	alveolar flap
g	*gut, angle, tag*	voiced velar plosive
k	*cut, ken, take*	voiceless velar plosive
f	*fork, after, if*	voiceless labiodental fricative
v	*vat, over, have*	voiced labiodental fricative
s	*sit, cast, toss*	voiceless alveolar fricative
z	*zap, lazy, haze*	voiced alveolar fricative
th	*thin, nothing, truth*	voiceless dental fricative
dh	*Then, father, scythe*	voiced dental fricative
sh	*she, cushion, wash*	voiceless postalveolar fricative
zh	*genre, azure*	voiced postalveolar fricative
l	*lid*	alveolar lateral approximant
l	*elbow, sail*	velar lateral approximant
r	*red, part, far*	retroflex approximant
y	*yacht, yard*	palatal sonorant glide
w	*with, away*	labiovelar sonorant glide
hh	*help, ahead, hotel*	voiceless glottal fricative
m	*mat, amid, aim*	bilabial nasal
n	*no, end, pan*	alveolar nasal
ng	*sing, anger*	velar nasal
ch	*chin, archer, march*	voiceless alveolar affricate: t + sh
jh	*joy, agile, edge*	voiced alveolar affricate: d + zh

图 2.1: 美式英语的音素和一般实现办法

一般我们将音素分为两类：元音（Vowel）和辅音（Consonants）。

1. Vowels: 元音有两个特点，一是元音都是发声的声音（voiced sound），这意味着从声带（vocal

chords）到口腔（mouth cavity）的气流是由声带的某种基频的震动（或者音高）产生的。二是舌头在生产过程中不会以任何方式形成气流收缩。每个元音发声的时候舌头、嘴唇和下巴的造型都不一样。这些不同的方式形成了不同的共振态，我们称之为共振峰。这些共振峰的共振态频率形成了不同的元音。

2. Consonants: 辅音是通过在口腔中或者空气中很明显的气流收缩形成的。有些辅音和元音一样是发声的，有些是不发声的。不发声的音素不会激活声带，因此也不存在基频或者音高。一些辅音音调成对出现，只有在有声或无声的情况下才有所不同，但在其他方面是相同的。比如说/b/和/p/这两个音素的发音方式是相同的，因为你的嘴唇、下巴还有舌头的姿势是一样的。但是/b/是发声的，/p/是不发声的。

音素的另外一个重要特性是其根据不同的上下文音素发音是会改变的。我们称之为Phonetic Context。之所以会这样，是因为协同发音（coarticulation）。这些声音连续起来发音会改变其原有的特征。由协同发音产生的音素我们称为音素变体（allophones）。

所有当前的这些语音识别系统都使用了音素的语境相关的特性来建立处于不同 phonetic context 的音素模型。

2.1.2 Words and Syntax

Syllable 是一串声音，是个序列，由一个核心的音素，可能有初始音素和终止音素，这个核心音素一般是个元音或者一个音节辅音（syllablic consonant），是能够唱出来或者吼出来的声音。

举个例子，英文单词"bottle" 包含两个 syllable 。第一个 syllable 有三个 phone ，在 Arpabet 音素描述代码里，是"b aa t" 。这个"aa" 就是核心音素，"b" 是发声的初始音素，"t" 是不发音的终止音素；第二个 syllable 是只包含一个 syllablic cosonant "l"。

一个 syllable 也可以组成一个词，其本身就是一个单独的音素，比如说，"Eye"，"uh"，或者"eau"（医: 水）。

语音识别里面，syllable 很少会考虑作为声学模型的建模单元，而词一般是变成音素来建模。

Syntax（句法规则）描述了给定词和定义了语法的规则下，句子的形成。而 Semantics（语义学）一般指代的是句子中的词或者短语是如何形成句意的。Syntax 和 Semantics 是 NLP 的重要组成部分，但是在语音识别里面，不起主要作用。

2.1.3 Measuring Performance

在语音识别系统的搭建和实验中，如何来衡量一个系统的好坏呢？由于语音识别是一个序列任务，跟图像当中的分类不一样，因此我们在衡量系统的性能时需要考虑到整个序列。

语音识别准确率衡量最常用的一个指标是词错误率（word error rate，WER）。一般识别出来的结果可能会产生三种错误：替换（substitution）、删除（delete）和插入（insert）。替换指的是一个词被识别成了另外一个词；删除指的是原本有词，但是没有识别出来；插入指的是原本没

有词，多识别出来了词。WER 的计算方式如公式2.1。

$$WER = \frac{N_{sub} + N_{ins} + N_{del}}{N_{ref}} \tag{2.1}$$

其中 N_{sub}、N_{ins} 和 N_{del} 分别是替换、插入和删除的数量，而 N_{ref} 是参考文本描述中词的个数。

WER 的计算用的是通过计算实际输出描述和参考文本描述之间的字符串编辑距离得到的。编辑距离的实现通过动态规划算法。因为长文本的编辑距离可能不可靠，所以我们通过逐句的计算累积的错误，这些错误最终整合到一起来计算测试集的 WER。

表2.1呈现了实际输出和参考文献之间的不同，以及对应的三种错误。

```
Ref: however a little later we had a comfortable chat
Hyp: how never a little later he had comfortable chat
```

表 2.1: WER 计算公式中的三种错误实例演示

Reference	Hypothesis	Error
however	how	Substitution
	never	Insertion
a	a	
little	little	
later	later	
we	he	Substitution
had	had	
a		Deletion
comfortable	comfortable	
chat	chat	

在某些情况中，这三种错误的成本不对等，那么计算编辑距离的时候可以作相应的调整。

句错误率（Sentence Error Rate，SER）是另外一种衡量系统的标准，其计算方式是整句没有出现任何错误。SER 仅仅作为一个指标，来看下错误的句子占全部句子的比例。

2.1.4 Significance Testing

统计显著性检验（statistical significance testing）涉及测量两个实验（或算法）之间的差异在多大程度上归因于两个算法中的实际差异，或者仅仅是数据，实验设置或其他因素中的结果固有变异性。统计显著性是所有分类任务的基石，只是统计显著性检验的方法取决于任务的特性。大多数方法的核心是假设检验的概念中存在一个无效假设。问题在于你有多大的 confidence 能够说无效假设会被拒绝。

对于语音识别来说，比较两个实验或者算法最常用的方法是 Matched Pairs Sentence-Segment Word Error(MAPSSWE) 检验，简称为 Matched Pairs Test[(3)]。

在这个方法中，测试集被分为几份，假设这些子测试集中任意一个的错误都与其他子测试集统计独立。这个假设和语音识别的实验很贴合，因为测试数据都是一句一句的经由识别器输出结果。给定了每一个句子的 WER，就很容易构建一个 matched pairs[11]。

2.1.5 Other Consideration

除去准确率，对识别系统性能的影响还可能包括计算需求，处理速度或者延迟什么的。解码速度一般用实时因子（real-time factor，RTF）来衡量。RTF 为 1.0 指的是系统处理 10s 的数据需要花 10s 的时间。

RTF 高于 1.0 意味着系统要花更多的时间来解码，对于某些应用，也许是可以接受的，比如希望获得一个会议或者讲座的转写，相对于快速的得到转写结果，准确率可能更重要一些，因此多花一些时间也是可以接受的。

当 RTF 低于 1.0，系统会在当前数据达到前就处理好了之前的数据。当不止一个系统在同一个机器上运行的时候，这个就比较有用了。在这种情况下，我们可以用多线程来并行处理多路音频流。此外 RTF 低于 1.0 意味着系统能够满足在线实时解码的音频流应用。比如说，当我们在处理一个手机上远程音频需求的时候，网络阻塞可能会使得服务器接收音频产生间隙和延迟。如果语音识别器能够以比实时更快的速度来处理，那么它就可以在数据达到之后迅速跟进，追上最新的音频进度，以速度来掩盖网络的延迟。

一般来说，语音识别系统能够在准确率和速度之间调整，但是这种调整也是有限的，不可能无限好或者无限快。对于一个给定的模型和测试集，speed-accuracy 图有一条不可被突破的渐近线（asymptote），即便给予无限的算力。所以准确率是有个极限的，这个时候的错误率可以认为就是模型带来的错误。一旦根据模型搜索找到了最好的结果，进一步的处理也不会带来准确率上的提升。

2.1.6 The Fundamental Equation

语音识别可以看作一个优化任务。特别地，给定一个观测序列 $O = \{O_1, ..., O_N\}$，我们找寻的是最有可能的词序列 $W = \{W_1, ..., W_M\}$，也就是说我们要找到最大化后验概率 $P(W|O)$ 的词序列，如公式2.2。

$$\hat{W} = \arg\max_{W} P(W|O) \tag{2.2}$$

利用贝叶斯规则，我们得到公式2.3。

$$P(W|O) = \frac{P(W)P(O|W)}{P(O)} \tag{2.3}$$

因为词序列并不依赖于观测序列的边缘概率分布 $P(O)$，我们可以忽略这个部分，综合上述两个公式，我们得到公式2.4。

$$\hat{W} = \arg\max_{W} P(W)P(O|W) \tag{2.4}$$

这就是语音识别的基本公式。语音识别问题就可以看作是在这个联合模型上的搜索。

公式中的 $P(O|W)$ 叫做声学模型（acoustic model）。这个模型描述了在给定词序列 W 的条件下，声学观测 O 的分布。声学模型表征的是词序列是如何转换成声学实现的，进而转换成 ASR 系统的声学观测的。

公式中的 $P(W)$ 叫做语言模型（language model），其只取决于词序列 W。语言模型给每一个可能的词序列一个概率值。它是由日常使用的一些词序列训练成的。一个训练好的英语语言模型会给"I like turtles" 高的概率值，给"Turtles sing table" 低的概率值。语言模型促使着词序列的搜索沿着训练数据中的模式开展。语言模型也可以在一些纯文本的应用中见到，比如浏览器的自动补全等。

由于诸多原因，构建一个语音识别系统比这个简单地公式所能呈现的要复杂的多得多。

2.1.7 Lab 1: Create a speech recognition scoring program

本模块有一个实验作业，标题为"Create a speech recognition scoring program"。

Required files:

- wer.py
- M1_score.py

Instructions:

In this lab, you will write a program in Python to compute the word error rate (WER) and sentence error rate (SER) for a test corpus. A set of hypothesized transcriptions from a speech recognition system and a set of reference transcriptions with the correct word sequences will be provided for you.

This lab assumes the transcriptions are in a format called the "trn" format, created by NIST. The format is as follows. The transcription is output on a single line followed by a single space and then the root name of the file, without any extension, in parentheses. For example, the audio file "tongue_twister.wav" would have a transcription

sally sells seashells by the seashore (tongue_twister)

Notice that the transcription does not have any punctuation or capitalization, nor any other formatting (e.g. converting "doctor" to "dr.", or "eight" to "8"). This formatting is called "Inverse Text Normalization" and is not part of this course.

The python code M1_Score.py and wer.py contain the scaffolding for the first lab. A main function

parses the command line arguments and string_edit_distance() computes the string edit distance between two strings.

Add code to read the trn files for the hypothesis and reference transcriptions, to compute the edit distance on each, and to aggregate the error counts. Your code should report:

- Total number of reference sentences in the test set
- Number of sentences with an error
- Sentence error rate as a percentage
- Total number of reference words
- Total number of word errors
- Total number of word substitutions, insertions, and deletions
- The percentage of total errors (WER) and percentage of substitutions, insertions, and deletions

The specific format for outputting this information is up to you. Note that you should not assume that the order of sentences in the reference and hypothesis trn files is consistent. You should use the utterance name as the key between the two transcriptions.

When you believe your code is working, use it to process hyp.trn and ref.trn in the misc directory, and compare your answers to the solution.

2.2 Speech Signal Processing

2.2.1 Introduction

通过空气传播的音频波形通过麦克风的捕捉，将这些压力波转换成可捕捉的电信号活动。对这些电信号活动进行采样，这样就得到了一系列采样波形，我们用这些波形来描述信号。音乐信号一般采样率为 44100 Hz，即一秒钟会有 44100 个采样点。根据奈奎斯特定理（Nyquist theorem），只有频率低于 22050 Hz 的音频才可以被捕捉到。如果信号的高频部分比较少，最高频率为 8000Hz 的话，采样率一般就是 16000Hz。传统的电话和大部分的手机带限是 3400 Hz，所以 8000 Hz 的采样率就足够了。所以电话语音的采样率一般就是 8000Hz。

一个典型的音频波形图如2.2（左），其说的句子是"speech recognition is cool stuff"。

回忆上一节中讨论的发声音素和不发声音素，我们来瞅一下最后一个单词"stuff" 的波形图，如图2.2（右）。从图上我们看到，这个单词的发音有三个不同的部分，初始不发声语音"st"，中间发声语音"uh"，终止不发声语音"f"。不发声的语音部分看上去像噪音，具有随机性。而发声部分的语音由于声带的振动而具有周期性。

在拉近一点，我们看看发声的这个元音"/uh/"，如图2.2（下）可以更明显的看到这个元音的周期性。

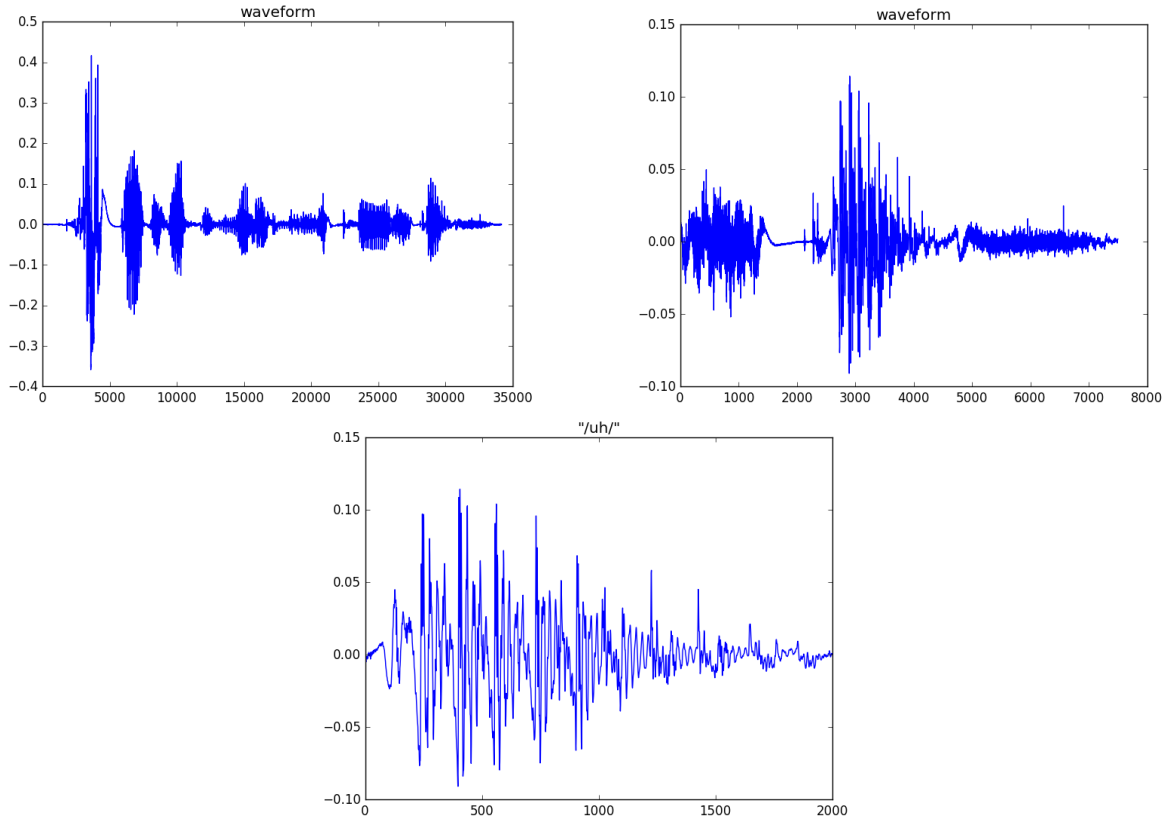

图 2.2: 完整音频波形图与部分波形图

从这些波形图中，我们可以看到波形的特征形成有两方面的因素：（1）声带的应激反应（excitation）趋势着空气从声道和嘴巴流出；（2）发出某种特定声音时，声道本身的形状。

举例来说，图2.2（右）中"st" 和"f" 看上去都像噪声，但是他们的形状却不相同，就是因为他们是不一样的声音。而"uh" 的声音更具周期性，因为发声的应激反应，其由于发声的时候声道的作用有独特的形状。所以对于同一个说话人来说，不同的元音可能会有着相近的周期，但是波形的整体形状是不一样的，就因为这些波形都是由相同的声带产生的，但是发不同的声音声道是不一样的。

在信号处理中，一般用源滤波器模型（source-filter model）来对这个语音产生的过程进行建模。声源是由通过声道的声带产生的激励信号，我们将其建模成为时变线性滤波器。源滤波器在语音识别中有很多应用，比如语义分析和编码。而且有很多种办法来评估源信号和滤波器的参数，比如很有名的线性预测编码（Linear Predictive Coding，LPC）。

对于语音识别来说，音素分类在很大程度上取决于声道的形状，也就是说取决于源滤波器模型的滤波器部分。激励信号或者原信号大都被忽略或者舍弃了。所以语音识别的特征提取过程一般设计成捕捉话语过程的时变滤波器形状。

2.2.2 Feature Extraction

从波形图中，很明显语音是非平稳信号（non-stationary signal），这就意味着语音信号的统计特性会随着时间变化而变化。所以为了分析语音信号，我们需要将信号分成一个又一个 chunk（也成为窗或者帧），这些 chunk 短到可以认为它们是平稳信号。这样我们就可以去分析一系列短时的有重叠的语音帧。在语音识别中，我们一般选窗长为 25ms，窗移位 10ms，也就是说一秒钟会被分成 100 帧。

因为我们是从一个长音频提取的 chunk，所以对于每一个 chunk 的边缘，我们要进行一些处理。一般对每一帧数据加个窗函数，常用的是汉明窗（Hamming Windows），当然也有用其他窗函数的。定义 m 为某一帧的索引，n 为采样点的索引，L 是这一帧采样点的个数，N 是采样中的偏移量。那么从原始信号中提取出来的每一帧计算公式见2.5。

$$x_m[n] = w[n]x[mN + n], n = 0, 1, \ldots, L - 1 \tag{2.5}$$

其中 $w[n]$ 是窗函数。

然后我们利用离散傅里叶变换将每一帧的数据转换到频域，如公式2.6。所有现代软件中都可以有效的计算快速傅里叶变换。

$$X_m[k] = \sum_{n=0}^{N-1} x_m[n]e^{-j2\pi knN} \tag{2.6}$$

傅里叶表征 $X_M[k]$ 是一个很复杂的数，因为它包含了每一帧和每一个频率的频谱幅值（绝对幅值）和相位信息。为了提取特征，我们去掉了相位信息，只考虑幅值 $|X_m[k]|$。

频谱图描述了对语音信号进行 FFT 操作得到的 log 幅值（或者 log-power），如图2.4（右）。横轴是帧索引（单位为 10ms），纵轴是频率，其范围是 0Hz 到采样率的一般，也就是对应的 Nyquist 频率。图中呈现的是"speech recognition is cool stuff"。在频谱图中，黄色和红色区域表示该区域能量高。

2.2.3 Mel Filtering

从频谱图中可以看出高频的高能量区域大致对应着不发声的辅音，低频的高能量区域大致对应着发声的元音。频谱图中，发声区域的水平线（horizonal lines）呈现的是语音的谐波结构（harmonic structure）。

由于发声区域的谐波结构和不发声区域的随机噪声，频谱中存在着变数（variability）。为了移除这些变数，我们对幅度谱（magnitude spectrum）进行频谱光滑操作。受听觉系统处理语音信号的启发，我们对频谱图进行滤波器组操作（filterbank），该滤波器组对频率轴进行了 approximately logarithmic scale。也就是说随着频率的升高，滤波器也会变得更宽间隔更大。最常用于特征提

取的 filterbank 是 **mel filterbank**。一个 mel filterbank 包含 40 个滤波器，如图2.3。每一个滤波器会对不同频率区间的能量谱求平均。

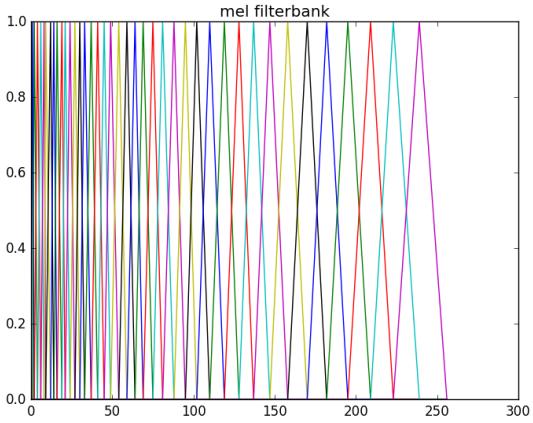

图 2.3: mel filterbank

mel filterbank 图在左边的滤波器很密集，在右边的滤波器间隔就比较远了。这是符合人耳听觉系统的，因为人耳对低频的信号更加敏感，对高频的信号不太敏感，所以低频的信息是更重要的，那么就需要多一些滤波器，从而提取更多有效特征。

P 维的 mel filterbank 的系数计算公式如2.7。一个 mel filterbank 一般会计算出 40 个系数，虽然现代系统有的会多一些或者少一些。平滑过多，系数就少一些，反之则反之。

$$X_{\text{mel}}[p] = \sum_{k} M[p,k]\,|X_m[k]|, \quad p = 0,1,\ldots,P-1 \tag{2.7}$$

2.2.4 Log Compression

特征提取的最后一步是对经过滤波器组得到的系数进行对数压缩。这个操作有助于压缩信号的动态范围，还能模拟听觉系统对声音的非线性压缩效果。我们把对数压缩后的输出称为"filterbank" 系数。

将提取的特征以频谱图式的方式呈现出来之后，如图2.4（左），与原始信号频谱图（右）进行比较，我们可以看出沿着频率轴的 Fbank 系数要平滑得多，这是因为高频噪声和 pitch/谐波结构都被移除了。

除了上面的这些操作，在提取特征的过程中可能还会有一些其他的操作。其中有：

1. Dithering（抖动）：在原始音频信号中加入一个很小的噪声，为了防止在提取特征的时候出现数学问题，尤其是出现 $\log 0$；
2. DC-removal（直流常数移除）：在提取特征之前，去除音频中的常数偏置；
3. Pre-emphasis（预加重）：在提取特征之前用一个高通滤波器处理信号，因为发声的语音部分低频能量比不发声的语音部分高频能量要大得多，用一个高通滤波器来抵消下这个问

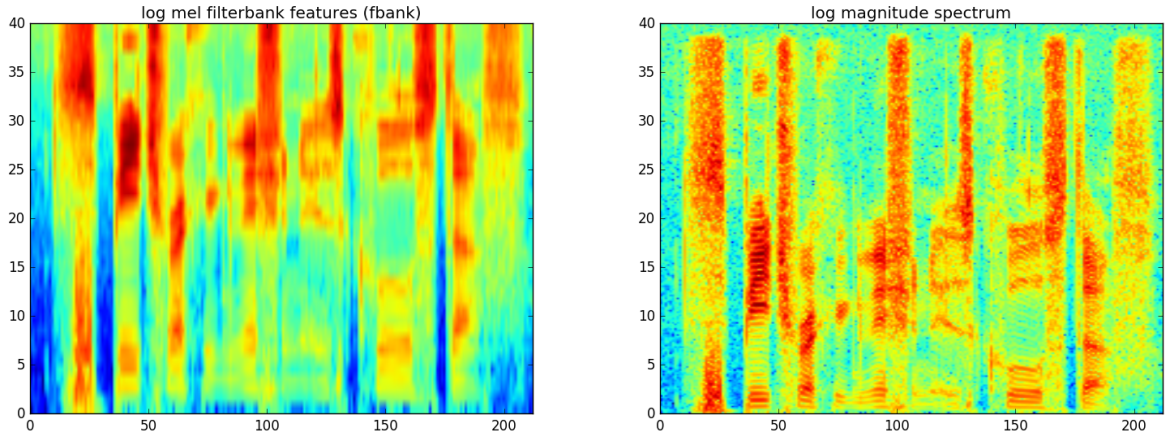

图 2.4: 提取 Fbank 特征（左）与原始信号的频谱图（右）

题。实际操作的时候就是用了一个简单地线性滤波器，见公式2.8，其中 $\alpha = 0.97$。

$$y[n] = x[n] - \alpha x[n-1] \tag{2.8}$$

4. DCT（离散余弦变换）：Fbank 系数是强相关的，为了减弱这种相关性，将上述步骤提取出来的 Fbank 特征值再经过 DCT。经过 DCT 之后的特征，一般取前 13 个，余下的由于所含信息不足舍弃了。DCT 公式如2.9。

$$d_t = \frac{\sum_{n=1}^{N} n(c_{t+n} - c_{t-n})}{2\sum_{n=1}^{N} n^2} \tag{2.9}$$

2.2.5 Feature Normalization

通信信道可能会对捕获的语音信号引入一些偏差（恒定滤波）。比如说，麦克风的频率响应不平稳，此外，即使相同语音的基础信号，其信号增益的变化也可能导致计算的滤波器组系数的差异。这些信道的影响可以用时域上的卷积进行建模，等价于频域表征的信号进行点乘（elementwise multiplication）。

因此信道的影响可以用恒定滤波来建模（constant filter），如公式2.10。

$$X_{t,\text{obs}}[k] = H[k]X_t[k] \tag{2.10}$$

其观测幅度为：

$$\left|X_{t,\text{obs}}[k]\right| = |H[k]|\,|X_t[k]| \tag{2.11}$$

如果我们对公式2.11两边取对数，并计算句子中所有帧的均值，则我们有公式2.12。

$$\begin{aligned}\mu_{\text{obs}} &= \frac{1}{T}\sum_{t}\log\left(\left|X_{t,\text{obs}}[k]\right|\right)\\ &= \frac{1}{T}\sum_{t}\log\left(|H[k]|\,|X_t[k]|\right)\\ &= \frac{1}{T}\sum_{t}\log(|H[k]|) + \frac{1}{T}\sum_{t}\log\left(|X_t[k]|\right)\end{aligned} \tag{2.12}$$

假设滤波器在时间轴上是常数，且语音信号的对数幅值均值为 0，那么公式2.12可以简化为2.13。

$$\mu_{ttobs} = \log(|H[k]|) \tag{2.13}$$

以上，如果我们计算出句子的对数幅值的均值，并且对句子中的每一帧都减去这个均值，这样我们就可以除去信号中所有的恒定信道效应。

为了简便，我们直接对取了 log 之后 fbank 特征进行归一化（normalization）。为了对比一系列操作的结构，图2.5展现了原始信号的频谱图（左）、fbank 特征的频谱图（右）和对 fbank 特征归一化后的频谱图（下）。

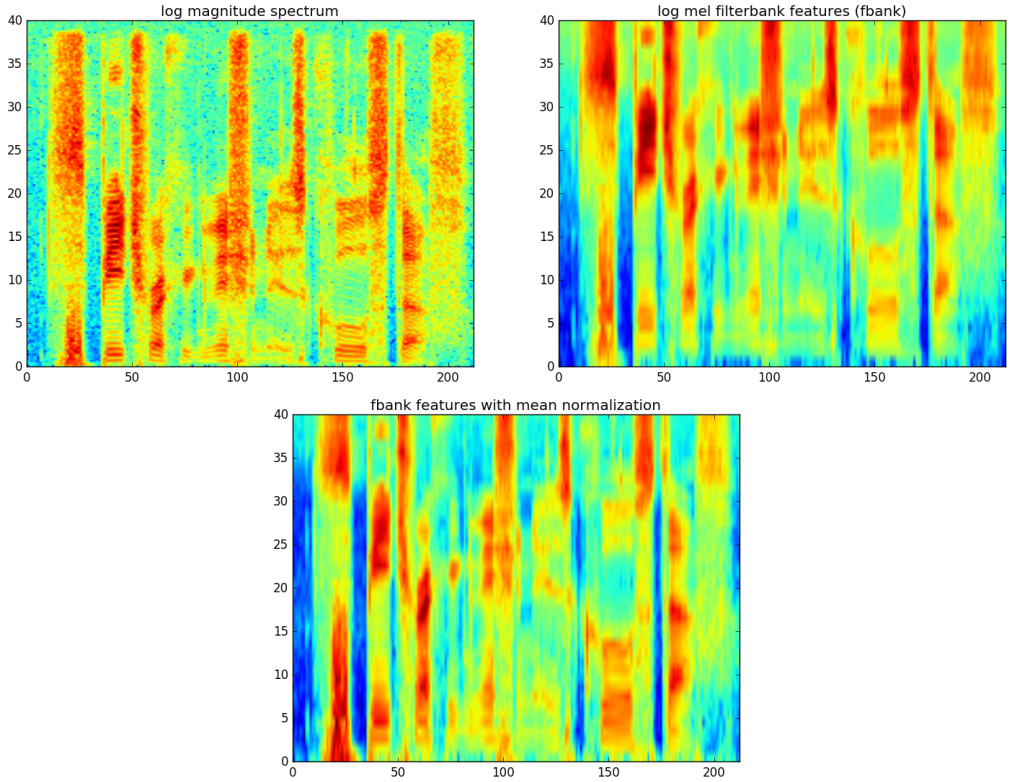

图 2.5: 原始信号的频谱图（左）、提取 Fbank 特征（右）和归一化后的 Fbank 特征（下）

2.2.6 Summary

为了从语音信号中提取语音识别所需的特征，我们希望提取到与声道形状有关的时变频谱信息，其由源滤波器模型中的一个滤波器建模，计算语句中的特征步骤如下：

1. 预处理信号，包括预加重和 dithering；
2. 将信号切割成有重叠部分的帧，一般帧长为 25ms，帧移为 10ms；
3. 对于每一帧：
   - 用汉明窗处理信号；
   - 使用 FFT 进行傅里叶变换；
   - 计算频谱的幅值；
   - 应用 mel filterbank；
   - 进行对数操作；
4. 如果需要进行信道补偿，则对每一帧的 fbank 系数进行均值归一化。

2.2.7 Lab 2: Feature extraction for speech recognition

本模块有一个实验作业，标题为"Feature extraction for speech recognition"。

Required files:

- M2_Wav2Feat_Single.py
- M2_Wav2Feat_Batch.py
- speech_sigproc.py
- htk_featio.py

Instructions:

In this lab, you will write the core functions necessary to perform feature extraction on audio waveforms. Your program will convert an audio file to a sequence of log mel frequency filterbank ("FBANK") coefficients.

The basic steps in features extraction are

1. Pre-emphasis of the waveform
2. Dividing the signal into overlapping segments or frames
3. For each frame of audio:
   - Windowing the frame
   - Computing the magnitude spectrum of the frame
   - Applying the mel filterbank to the spectrum to create mel filterbank coefficients
   - Applying a logarithm operation to the mel filterbank coefficient

In the lab, you will be supplied with python file called **speech_sigproc.py**. This file contains a partially completed python class called **FrontEnd** that performs feature extraction, using methods that perform

the steps listed above. The methods for dividing the signal into frames (step 2) will be provided for you, as will the code for generating the coefficients of the mel filterbank that is used in step 3c. You are responsible for filling in the code in all the remaining methods.

There are two top-level python scripts that call this class. The first is called **M2_Wav2Feat_Single.py**. This function reads a single pre-specified audio file, computes the features, and writes them to a feature file in HTK format.

In the first part of this lab, you are to complete the missing code in the **FrontEnd** class and then modify **M2_Wav2Feat_Single.py** to plot the following items:

1. Waveform
2. Mel frequency filterbank
3. Log mel filterbank coefficients

You can compare the figures to the figures below. Once the code is verified to be working, the feature extraction program should be used to create feature vector files for the training, development, and test sets. This will be done using **M2_Wav2Feat_Batch.py**. This program takes a command line argument **-set** (or **-s**) which takes as an argument either **train**, **dev**, or **test**. For example

$ python M2_Wav2Feat_Batch.py -set train

This program will use the code you write in the **FrontEnd** class to compute feature extraction for all the files in the LibriSpeech corpus. You need to call this program 3 times, once each for train, dev, and test sets.

When the training set features are computed (**-set train**) the code will also generate the global mean and precision (inverse standard deviation) of the features in the training set. These quantities will be stored in two ASCII files in the **am** direction for use by CNTK during acoustic model training in the next module.

Here are the outputs you should get from plotting:

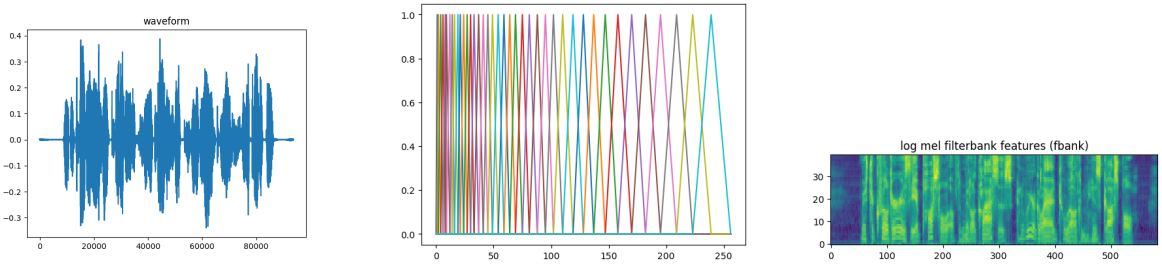

图 2.6: Lab 2 的期望输出

2.3 Acoustic Modeling

2.3.1 Introduction

本节讨论的是语音识别器中的声学模型。声学模型是一个混合模型，通过 DNN 得到逐帧的预测标签，再通过 HMM 将这些预测的音素转换成序列预测。HMM 常用于对离散时间序列事件的建模。HMM 的基本概念可以追溯到数十年前，而且 HMM 有很多应用。

2.3.2 Markov Chains

了解一点马尔科夫链（Markov Chains）对学习 HMM 大有帮助。马尔科夫链是一种对随机过程进行建模的方法。在马尔科夫链中，用一系列状态（states）来对离散时间进行建模。状态之间的运动由随机过程控制。

举例说明，在一个天气预测的应用中，状态为"**S**unny"、"**P**artly Cloud"、"**C**loudy"、和"**R**aining"。我们考虑某个连续五天的特定天气概率，比如 $P(p,p,c,r,s)$，我们可以使用贝叶斯规则将这个联合概率分布打散成一系列条件概率的乘积，如公式2.14。

$$p(X_1,X_2,X_3,X_4,X_5) = p(X_5|X_4,X_3,X_2,X_1)p(X_4|X_3,X_2,X_1)p(X_3|X_2,X_1)p(X_2|X_1)p(X_1) \tag{2.14}$$

假设天气模型满足一阶马尔科夫假设，即满足公式2.15。

$$p(X_i|X_1,...,X_{i-1}) = p(X_i|X_{i-1}) \tag{2.15}$$

那么连续五天天气的联合概率分布可以简化为公式2.16。

$$\begin{aligned} p(X_1,X_2,X_3,X_4,X_5) &= p(X_5|X_4)p(X_4|X_3)p(X_3|X_2)p(X_2|X_1)p(X_1) \\ &= p(X_1)\prod_{i=2}^{5} p(x_i|x_{i-1}) \end{aligned} \tag{2.16}$$

一个马尔科夫链的核心元素有状态的定义（此处为天气预测）和转移概率 $p(X_i|X_{i-1})$，转移概率描述的是从一个状态移动到另一个状态的概率值（也包括转移到自身状态）。

比如说，天气预报的一个完整的（大体完整的）马尔科夫链可以用图2.7来表示。

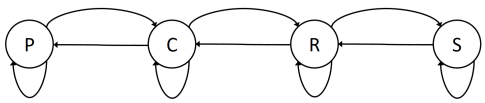

图 2.7: 天气预报模型的马尔科夫链

需要注意的是除了上面说的转移概率 $p(X_i|X_{i-1})$，我们还需要知道这个序列的第一个元素，即第一天的某种天气的概率值 $p(X_1)$。

所以除了状态清单和状态转移概率，我们还需要知道从马尔科夫链每一个状态开始的初始概率值。假设先验概率（每个状态的初始概率值）如公式2.17。

$$\begin{aligned} p(p) &= \pi(p) \\ p(c) &= \pi(c) \\ p(s) &= \pi(s) \\ p(r) &= \pi(r) \end{aligned} \tag{2.17}$$

现在我们回到这个例子，公式2.18说明了如何求 $P(p,p,c,r,s)$。

$$\begin{aligned} p(p,p,c,r,s) &= p(s|p,p,c,r)p(r|p,p,c)p(c|p,p)p(p|p)p(p) \\ &= p(s|r)p(r|c)p(c|p)p(p|p)p(p) \end{aligned} \tag{2.18}$$

第 3 章 Kaldi 学习笔记

3.1 kaldi 中的数据扰动

kaldi 程序中对原始数据进行扰动以达到数据增强的效果，一般是在单音素对齐，三音素对齐之后，在生成 ivector 的时候进行扰动处理，扰动有两种方式：速度扰动和音量扰动。如果存在 segment 文件的话，那么对应的起始和终止时间点会存在 segment 文件中，提取特征时会根据这个 segment 中存储的时间节点进行操作；如果不存在的话，相当于整段 wav 音频都是有效的，那么起始时间点和 wav 文件相同。扰动也会根据 segment 文件的有无进行对应的操作。

3.1.1 速度扰动

速度扰动一般是对音频进行加速和减速，根据 Povey 大佬的论文《Audio Augmentation for Speech Recognition》中的第二部分 Audio Perturbation，对 mel 频谱进行一个偏移就能得到类似加速和减速的效果。首先定义一个扰动因子 α，假定 segment 中某一段音频的起始时间和终止时间为 t_1 和 t_2，那么新的音频起始时间和终止时间计算方式如公式3.1。

$$\begin{aligned} t_1' &= \frac{t_1}{\alpha} \\ t_2' &= \frac{t_2}{\alpha} \end{aligned} \tag{3.1}$$

kaldi 一般取 α 为 0.9 和 1.1 以达到加速和减速的目的。得到了 segment 文件之后，在 wav.scp 文件中存储原始音频的位置，加速后音频 sox 指令和减速后音频 sox 指令。其详细的脚本指令见代码 utils/data/perturb_data_dir_speed.sh 第 74 行。最终重新提取特征存于代码根目录下 mfcc_pertubed 文件夹中。由于速度扰动对音频时间轴有改动，因此此时需要对音频进行重新对齐的操作。

3.1.2 音量扰动

音量扰动一般是对音频进行增大音量和减小音量。音量增加或者减小的幅度默认取 $[0.125, 2]$ 之间的正态分布值。使用 sox 工具中的"sox - -vol volume" 来进行实际操作。其详细的脚本指令见代码 utils/data/perturb_data_dir_volume.sh 第 71 行，其 sox 操作见代码 utils/data/internal/perturb_volume.py。其重新提取的特征位于代码根目录下 mfcc_hires 文件夹中。此时由于仅仅对音频进行音量大小的扰动，并没有对时间维度进行操作，因此无需再进行一遍对齐操作，其标签对齐直接采用上一步即速度扰动后生成的对齐结果。此时重新提取特征时，MFCC 特征的维度是 40d。其原因是 40d 的 MFCC 和 40d 的 Fbank 维度相同，保存的信息量相似，同时 MFCC 由于

其相关性较弱（DCT 去相关），所以能更好的压缩特征，因此 Kaldi 一般都是采用 40d 的 MFCC 作为神经网络的输入特征（见 kaldi 的各个 egs 里 conf 下 mfcc_hires.conf）。

> "Config for high-resolution MFCC features, intended for neural network training. Note: we keep all cepstra, so it has the same info as filterbank features, but MFCC is more easily compressible (because less correlated) which is why we prefer this method. "

3.2 kaldi 中的 UBM

通用背景模型 UBM(Universal Background Model)

3.3 kaldi 通过 lattice 输出语音对齐音素和词

我们通过解码之后得到一堆的 lat.*.gz 文件，这些文件是 lattice 对齐后生成的对齐文件，其为压缩格式，所以首先通过 gunzip 指令对齐解压，我们以 lat.1.gz 为例来讲这一节的知识点。

```
gunzip exp/chain/tdnn7q_sp_online/decode_data_tgsmall/lat.1.gz
```

这样会在 exp/chain/tdnn7q_sp_online/decode_data_tgsmall/ 下生成一个 lat.1 文件，原先的 lat.1.gz 消失不见了……lat.1 文件是二进制的格式，其由指令 online2-wav-nnet3-latgen-fatser 生成。

3.4 kaldi 中的数据准备

准备词典 首先 lexicon.txt 的格式是"<word> pronunciation"。

第 4 章 FFmpeg 和 sox

4.1 FFmpeg

4.1.1 安装 FFmpeg

在 Centos 中利用 yum 安装 FFmpeg步骤如下：

1. 升级系统

```
sudo yum install epel-release -y
sudo yum update -y
sudo shutdown -r now
```

2. 安装 Nux Dextop Yum 源

   由于 CentOS 没有官方 FFmpeg rpm 软件包。但是，我们可以使用第三方 YUM 源（Nux Dextop）完成此工作。

   - CentOs 7

```
sudo rpm --import http://li.nux.ro/download/nux/RPM-GPG-KEY-nux.ro
sudo rpm -Uvh http://li.nux.ro/download/nux/dextop/el7/x86_64/nux-dextop-release-0-5.el7.nux.noarch.rpm
```

   - CentOs 6

```
sudo rpm --import http://li.nux.ro/download/nux/RPM-GPG-KEY-nux.ro
sudo rpm -Uvh http://li.nux.ro/download/nux/dextop/el6/x86_64/nux-dextop-release-0-2.el6.nux.noarch.rpm
```

3. 安装 FFmpeg 和 FFmpeg 开发包

```
sudo yum install ffmpeg ffmpeg-devel -y
```

4. 测试是否安装成功

```
ffmpeg
```

备注：

（1）查看机器的 centos 版本：cat /etc/redhat-release

（2）在执行安装 FFmpeg 和 FFmpeg 包的时候，可能会出现一些错误，因为有些依赖包没有安装，看下未安装的包有哪些，然后去pkgs官网上去搜索下载对应的版本即可。

（3）FFmpeg的使用参考资料

- reach296的博客；
- feixiao 的 GitHub 库；
- ffprobe,ffplay ffmpeg 常用的命令行命令

4.2 sox

sox 的参考资料：

- SoX —音频处理工具里的瑞士军刀

第 5 章 Linux 相关笔记

5.1 linux 备忘录

以下零零散散的一些笔记用于记录不太熟的 Linux 指令，不断扩充中……

1. 永久修改 ip 地址。首先 ifconfig 找到对应的网卡，其次编辑 vi /etc/sysconfig/network-scripts/ifcfg-lo，此处假设网卡是 lo。
2. 当我们输入 history 的时候，会出现历史命令。这些命令从 1 开始到 history 这个指令的索引数，可以用!+index 的方式来调用。比如 history 中显示的第五条命令是 ls，那么当我们在终端输入!5 的时候会自动调用 ls 指令。而且! 也可以接指令的部分内容，其会自动找寻最近的一条与该内容相关的指令并执行。比如说历史中有两条 ls 指令。一条是 ls / ；一条是 ls /home，假设第二条是最近的指令，那么我们执行！ls，那么系统就会执行 ls /home 这条指令。
3. alias short_cmd='real_cmd'，使用这个指令可以将 real_cmd 用 short_cmd 代替，减少常用命令的输出时间。使用 unalias short_cmd 可以取消对应的映射。alias 单独运行可以查看当前有哪些 short_cmd。而且我们可以将这个指令直接添加到 ~/.bashrc 中，这样就万事大吉，省心了，重启的时候也不会消失。
4. > » 2> 和 2» 还有 > bash run.sh 1»result 2>1。正确错误的都会传到 result 里面。
5. free -m 以 M 为单位显示磁盘存储空间
6. 修改 Ubuntu 下载源可以修改 /etc/apt/sources.list 文件
7. 查看 Ubuntu 版本可以使用 cat /etc/issue
8. Ubuntu 下解决 ifconfig command not found 的办法：sudo apt-get install net-tools
9. 当拥有多个用户，想把某个文件或者文件夹的权限下放到某个用户或者限制某个用户对某个文件或者文件夹的访问的时候，可以使用 setfacl 这个指令。具体的下放权限指令操作为：setfacl -m u:user1:rw test.txt，清空对于某个文件或者文件夹设置的权限时指令操作为：setfacl -b test.txt。查看某个文件或者文件夹更细化的权限信息可以使用 getfacl 指令。对目录以及子目录设置 acl 权限时，指令为：setfacl -m u:user1:rw -R /mnt。如果后续当前目录有生成新的子目录或者文件，想要继承现有权限的时候，需要在执行上述指令之后，再执行一次 setfacl -m d:u:user1:rw -R /mnt
10. 设置用户对于某个命令的执行权限，使用指令 visudo（需要在 root 下执行），举例：首先执行 visudo，会打开一个文件，要添加某个用户对于某个指令的权限时，需要给出对应命令的绝对路径，假设给予 user4 添加新用户的权限，则在 visudo 打开的文件添加一句 user4 localhost=/usr/sbin/useradd，之后使用 sudo /usr/sbin/useradd user5 之后再输入 user4 的密码，

就可以创建 user5。多个命令使用“,”隔开。

11. 分配无密码的 sudo 命令：同样使用 visudo，然后在文本中添加一句：user4 ALL=NOPASSWD: /usr/sbin/useradd, /usr/sbin/userdel，之后再调用 sudo /usr/sbin/useradd user5 的时候就不需要 user4 的密码了。也可以使用 user4 localhost=NOPASSWD: /usr/sbin/useradd, /usr/sbin/userdel，这就意味着只有在 user4 下的时候才可以不用输入密码，而 ALL 表示在所有用户下使用 sudo 都不用输入密码。
12. 任务计划 crontab -e 创建一个新的任务计划。30 17 * * 5 task，其中前面的表示周五的 17:30，在这个时间点执行任务 task。可以用 crontab -l 查看任务计划的内容。
13. ls -R 递归式的显示当前目录所有的东西，包括文件，子目录，子目录中的所有东西
14. 创建一个 ftp 服务站代码如下，这个时候在/var 目录下会生成一个 ftp 的目录，ftp 下还有个 pub 目录，这个目录，使得我们可以在其他操作系统访问。在 Windows 下，打开文件夹，在文件栏输入 ftp://your_ip 即可访问 ftp 的 pub 目录，ftp 启动指令为 service vsftpd start。

```
yum -y install vsftpd*
```

15. 查看 linux 某个端口是否被占用指令：netstat -anp|grep $port
16. 当我们不需要显示输出结果，但是 Linux 指令默认自动输出指令的时候，我们可以在指令后面加上 &>/dev/null，这样所有输出就重定向到一个黑洞里，全部都消失了。
17. 解决 Linux 中文乱码问题：以 docker 安装的 Ubuntu 为例。

```
>>locale -a
C
C.UTF-8
POSIX
>>export LC_ALL='C.UTF-8' #这是临时修改的
>>export LC_ALL='C.UTF-8' >> /etc/bash.bashrc | source /etc/bash.
   bashrc #这是永久修改。
>> docker restart <container-ip> #退出container，重启container即可
```

18. docker 启动命令：systemctl start docker
19. bad interpreter: No such file or directory：出现这个错误的原因是我用 winscp 连接服务器，然后用 Windows 系统下的 VScode 写代码。因为 Windows 下文本默认的格式是 DOS，而 Linux 是 unix，所以在服务器上运行的时候会出现这个错误。修改的办法有两个：第一个是在 Linux 里面用 vim 打开这个脚本，输入 set ff 的时候会出现个 DOS，所以我们要设置下属性，如下所示。另外我们还可以在 VScode 中直接修改，写好脚本之后，点击右下角的 CRLF，上边栏会出现一个选项，选择 LF 即可，如图5.1。

```
:set ff=unix
```

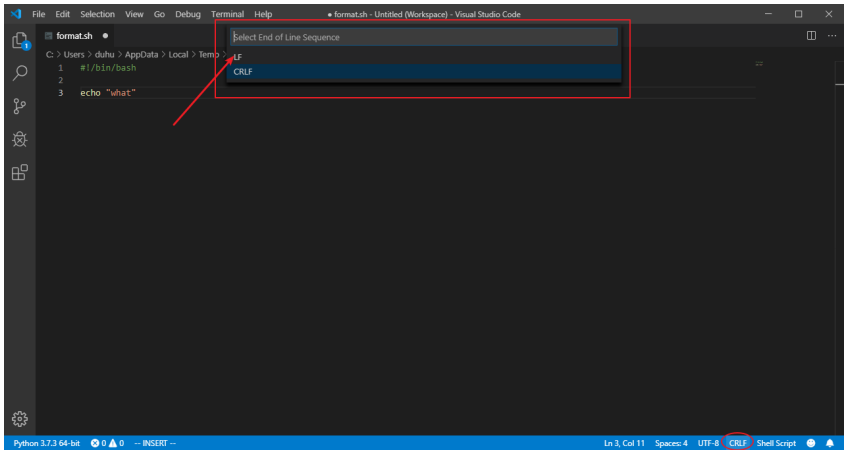

图 5.1: VScode 设置脚本格式为 Unix

20. 我们在 Linux 上安装软件时，经常需要选择机器所对应的版本，即机器是多少位的。查看指令有：
    - uname -a
    - file /bin/ls
    - cat /proc/vision
    - getconf LONG_BIT
21. 删除文本中的空行：grep -v '' file
22. 如何对文本指定列进行排序:

```
sort –n –k2 file #numerical sort;k2表示的是某一列，从小到大排序
sort –g –k2 file #general–numerical sort
sort –nr –k2 file #从大到小排序
```

23. 出现错误："[: too many arguments"：这个是在字符串比较中出现的，表示的是比较的对象数量不一致，也就是说可能出现多个字符串，这个时候给这个变量添加上双引号就可以了。

5.2 Shell 指令笔记

5.2.1 简单的 shell 规则

此处记载一些简单的 shell 指令和编程规则。不断补充……

1. 将键盘输入的内容赋值给变量：read [-p "some message"] 变量名。

2. 双引号（""）允许通过 $ 符号引用其他变量值；单引号（’’）禁止引用其他变量值，$ 视为普通字符；反撇号（``）将命令执行的结构输出给变量。
3. 关于 shell 中外部传输变量的一些操作：
   - $#：命令行中位置参数的个数；
   - $*：所有位置参数的内容，当格式为"$*" 时，参数作为一个整体传递给程序；
   - $?：上一条命令执行后返回的状态，当返回状态值为 0 时，表示执行正常；非 0 则表示执行异常；
   - $0：当前执行的进程/程序名；
   - $n：$n \in \{0,1,2,3,4,5,6,7,8,9\}$，表示外部传入的第 n 个参数。
   - $@：传递所有参数的内容，当格式为"$@" 的时候，表示将参数分开传递给程序；
4. shell 中的数学运算指令：expr。注意乘法是 *
5. 解析字符串中的转义字符，可以使用 echo -e，比如 echo -e "test\ntest"，echo -e 还可以修改输出的颜色和背景色，指令是 \[033[前景颜色; 背景颜色 m。\[033[0m 表示恢复到系统默认颜色。其前景颜色的数值为：默认 =0，黑色 =30，红色 =31，绿色 =32，黄色 =33，蓝色 =34，紫色 =35，天蓝色 =36，白色 =3；背景颜色的数值为：默认 =0，黑色 =30，红色 =31，绿色 =32，黄色 =33，蓝色 =34，紫色 =35，天蓝色 =36，白色 =3。
6. shell 默认在输出之后换行的，如果想要不换行，可以选择参数 -n。例如 echo -n "Enter your name: "，这样在终端显示的时候就不会换行。只有一个 echo 的时候，会输出一个换行。
7. cat 的另外一种用法，比如可以用于制作菜单，如下程序所示，这个程序在执行的时候会保留原样输出两个 x 之间的内容，包括换行和空格等。

```
cat<<x
  please input your name:
     1) user1
     2) user2
     3) user3
x
```

8. nl 指令：给输出标上行，比如 cat test.txt | nl，这条指令会给 test.txt 中的每一行进行顺序标号；
9. tee 指令：在程序执行输出的时候额外保留一份输出到文件中，比如./test.sh | tee test.txt。能够起到一个备份的作用。
10. shift 指令：用于迁移位置变量，将 $1 到 $9 依次向左传递。
11. 两个文件的交集、并集和去重：

```
cat file1 file2 | sort | uniq > result #求两个文件的并集，如果有重复的行
```

```
只保留一行。
cat file1 file2 | sort | uniq -d > result #求两个文件的交集，即两个文件中都有的行。
cat file1 file2 | sort | uniq -u > result #求两个文件的差集，即只有一个文件中有的行。
cat file1 file2 > result #追击的方式，如果file1有n行，file2有m行，result为n+m行
paste file1 file2 > result #一个文件的内容在左边，一个文件命令在右边。
sort file |uniq #将重复的多行变为一行
sort file |uniq -u #sort file |uniq -u
```

5.2.2 shell 中的条件测试操作

1. test 命令：
   (a). 用途：测试特定的表达式是否成立，成立返回 0，不成立则返回非 0；
   (b). 格式：test 条件表达式 [条件表达式]
2. 常见的测试类型
   (a). 测试文件状态：
   - 格式：[操作符文件或者目录]
   - 常用的文件操作符，如表5.1。

表 5.1: 常用的文件操作符及其意义

文件操作符	作用
-d	是否为目录（Directory）
-f	是否为文件（File）
-e	是否存在文件或者目录（Exist）
-r	当前用户是否可读（Read）
-w	当前用户是否可写（Write）
-x	当前用户是否可操作（Excute）
-L	文件是否为符号链接文件（Link）

   (b). 字符串比较：
   - 格式：[字符串 1 = 字符串 2]，[字符串 1 ！ = 字符串 2]，[-z 字符串]
   - 常用的字符串操作符，如表5.2。

   (c). 整数值比较：
   - 格式：[整数 1 操作符整数 2]
   - 常用的整数值比较操作符，如表5.3。

表 5.2: 常用的字符串操作符及其意义

字符串操作符	作用
=	字符串内容相同
! =	字符串内容不相同
-z	字符串内容为空

表 5.3: 常用的整数操作符及其意义

整数操作符	作用
-eq	等于（Equal）
-ne	不等于（Not Equal）
-gt	大于（Greater Than）
-lt	小于（Less Than）
-ge	大于等于（Greater or Equal）
-le	小于等于（Less or Equal）

(d). 逻辑测试：此处需要注意的是与操作和或操作有的时候可以起到一个开关的作用，比如 A&&B，假设 B 是一条指令的话，那么只有 A 为真的情况下才会进行下一步操作；类似的，如果是或操作，只有前面为假才会执行后面的操作。

- 格式：[表达式 1] 操作符 [表达式 2] ...
- 常用的逻辑操作符操作，如表5.4。

表 5.4: 常用的逻辑操作符及其意义

逻辑操作符	作用
-a/&&	逻辑与
-o/||	逻辑或
-!	逻辑否

5.2.3 find

基本使用 find <directory> -args contents，比如说 find . -name fun。find 的指令和例子如下代码5.1所示。

Listing 5.1: find 指令详解

```
find . -name "[a-z]*" #寻找所有a-z开头的文件或者文件夹
find . -name "[0-9]*" #寻找所有0-9开头的文件或者文件夹
find . -perm 775 #根据权限寻找文件
find . -user root #根据文件创建者来寻找文件
find . -mtime -5 #寻找更改时间为5天以内的文件
```

```
find . -mtime +3 #寻找更改时间为3天以前的文件
find . -type d[f;I] #寻找类型为文件夹[文件；链接]的文件
find . -size +1000000c #寻找文件大小大于1M的文件
find . -perm 700 | xargs chmod 777 #找到权限为700的改为777
find . -type f | xagrs ls -l #找到所有文件并展示出来
find . \( -name *.gt -o -name *.pre\) #同时找到不同类型的多个文件，每添加一个类型，都需要加上 -o -name
```

5.2.4 grep

本小节分为两块一个是 grep 指令的一般性操作，如代码5.2。

Listing 5.2: grep 指令的一般性操作

```
grep "<contents>" #查找内容
grep -c "<contents>" file #文件中有多少行匹配到查找的内容
grep -n "<contents>" file #文件中有多少行匹配到找到的文件，并显示行号
grep -i "<contents>" file #查找内容，并忽略大小写
grep -v "<contents>" file #过滤掉内容
```

另外一部分就是配合正则表达式进行文本的查找，先说下一些正则表达式的表示，如下所示：

1. ’l̂inux’：以 linux 开头的行；
2. ’linux$’：以 linux 结尾的行；
3. ’.’：匹配任意单字符；
4. ’.+’：匹配任意多个字符；
5. ’.*’：匹配 0 个或多个字符；
6. ’[0-9a-z]’: 匹配 [] 内任意一个字符；
7. ’(linux)+’：出现多次 linux 单词；
8. ’(linux){2}’：出现两次 linux；
9. ’\’：转义字符；
10. ’$’：空行。
11. ’[l̂inux]’：查找不以 linux 开头的行，[] 内的’ˆ表示取反。

5.2.5 awk

awk 是非常强的一个文本编辑的命令，主要按照列来对文本进行分割处理。一般指令为：awk -F: '{print $1}'，这段指令的意思是以':' 为分隔符，输出第一列的数据。awk 不太好总结，因为用到的还不是很多，因此，以逐条总结的方式，将一些不太常用，但是关键时刻很管用的记录下来[15]，等以后更熟悉了再做全面的总结。代码如5.3。

Listing 5.3: awk 指令的一般性操作

```
>>cat log.txt
2 this is a test
3 Are you like awk
This's a test
10 There are orange,apple,mongo

>>awk –v #设置变量
>>awk –va=1 '{print $1,$1+a}' log.txt #此时a就是1，所以如果$1是数字，那么数字加一，如果不是，那么输出就是1
>>awk –vb=s '{print $1,$1+a}' log.txt #b为s，所以在第一列的所有元素后面加个s
>>awk '$1>2' log.txt #输出第一列中大于2的行
>>awk '$1==2 {print $1,$3}' log.txt #输出第一列中等于2的行的第一列和第三列
>>awk '$1>2 && $2="Are" {print $1, $2, $3}' log.txt #输出第一列大于2的行的第一、二和三列
```

awk 用于求和、求最大值、求最小值和求平均值，见代码5.4。

Listing 5.4: awk 求和、求平均、求最大最小值

```
cat data|awk '{sum+=$1} END {print "Sum = ", sum}' #求和
cat data|awk '{sum+=$1} END {print "Average = ", sum/NR}' #求平均值
cat data|awk 'BEGIN {max = 0} {if ($1>max) max=$1 fi} END {print "Max=", max}' #求最大值
awk 'BEGIN {min = 1999999} {if ($1<min) min=$1 fi} END {print "Min=", min}' #求最小值
```

5.2.6 sed

sed 与 awk 相辅相成，其主要对行进行操作，主要细节参考菜鸟教程[16]。

第 6 章 Windows 相关

本章用于记录一些 Windows 使用的一些问题，方便再遇到的时候查询。

6.1 win10 .net framework 3.5 安装报错 0x800F0954 问题

本问题解决参考tOneDay的博客，其他的一些博客都没有解决问题。以此为准：

1. 打开注册表：cmd+r 输入 regedit，确定；
2. 找到路径 HKEY_LOCAL_MACHINE-SOFTWARE-Policies-Microsoft-Windows-WindowsUpdate-AU，其中 UseWUServer 默认值为 1，改成 0；
3. 打开服务列表，重启 Windows Update service；
4. 此时可以正常安装.net framework 3.5；
5. 将第二步的修改还原，并重启 Windows Update service。

第 7 章 Python 笔记

7.1 一些小技巧

此处记录一些常用到小技巧，省时省力省心还漂亮……不断补充中……

1. 按照 Value 对字典进行排序

```
xs = {'a':4, 'b':3, 'c':2, 'd':1}
sorted(xs.items(), key=lambda x:x[1])
import operator
sorted(xs.items(), key=operator.itemgetter(1))
```

2. 假设 B 是列表 A 的子集，想要求 B 的补集，代码如下：

```
x = [1,2,3,4,5,6,7]
y = [1,2,3]
z = list(set(x+y))
```

3. 假设列表 B 和列表 A 有重复元素，目的去除重复元素，去除的代码为函数 remove_same(A, B)。列表长度的判断是为了减少循环次数，为了测试时间写了一个简单的脚本，结果显示循环短列表大概块 1 秒钟，这个取决于短列表有多短，只是聊胜于无的减少了些代码运行时间。

```
from time import time
def remove_same(A, B):
  a, b = A.copy(), B.copy()
  for i in A:
    if i in B:
      a.remove(i)
      b.remove(i)
  return a, b
A = []
B = []
for i in range(100000):
    if i%2 == 0:
        A.append(i)
    if i%33 == 0:
```

```
        B.append(i)
print("lenA:□{},□\t□lenB:□{}".format(len(A), len(B)))
st = time()
b, a = remove_same(B, A)
mt = time()
a, b = remove_same(A, B)
et = time()
if len(A) > len(B):
  b, a = remove_same(B, A)
else:
  a, b = remove_same(A, B)
et2 = time()
print("Cycle□B:", mt-st)
print("Cycle□A:", et-mt)
print("Cycle□Smaller□One:", et2-et)
```

4. TextGrid 格式的文本处理参考python 的 textgrid 库调研小结

7.2 python 中的线程、进程、协程与并行、并发

我们先介绍下设计到这几个东西的概念吧。

1. GIL：在 Cpython 解释器中，同一个进程下的多个线程，同一时刻只能有一个线程执行，无法利用多核优势。GIL 本质就是一把互斥锁, 即会将并发运行变成串行, 以此来控制同一时间内共享数据只能被一个任务进行修改, 从而保证数据的安全性保护不同的数据时, 应该加不同的锁，GIL 是解释器级别的锁, 又叫做全局解释器锁 CPython 加入 GIL 主要的原因是为了降低程序的开发复杂度, 让你不需要关心内存回收的问题, 你可以理解为 Python 解释器里有一个独立的线程, 每过一段时间它起 wake up 做一次全局轮询看看哪些内存数据是可以被清空的, 此时你自己的程序里的线程和 Python 解释器自己的线程是并发运行的, 假设你的线程删除了一个变量, py 解释器的垃圾回收线程在清空这个变量的过程中的 clearing 时刻, 可能一个其它线程正好又重新给这个还没来及得清空的内存空间赋值了, 结果就有可能新赋值的数据被删除了, 为了解决类似的问题, Python 解释器简单粗暴的加了锁, 即当一个线程运行时, 其它人都不能动, 这样就解决了上述的问题, 这可以说是 Python 早期版本的遗留问题. 毕竟 Python 出来的时候, 多核处理还没出来呢, 所以并没有考虑多核问题，以上就可以说明, Python 多线程不适合 CPU 密集型应用, 但适用于 IO 密集型应用

   ”’

> In CPython, the global interpreter lock, or GIL, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because CPython’s memory management is not thread-safe. (However, since the GIL exists, other features have grown to depend on the guarantees that it enforces.)
> ”””

2. 进程
3. 线程
4. 协程
5. 并行
6. 并发

7.2.1 进程和线程

进程和线程操作系统中的概念，这也是操作系统中的核心概念。

7.2.1.1 进程

进程是对正在运行程序的一个抽象，即一个进程就是一个正在执行程序的实例。从概念上说每个进程拥有它自己的虚拟 CPU，当然，实际上是真正的 CPU 在各个进程之间来回切换，这种快速切换就是多道程序设计，但是某一瞬间，一个 CPU 只能运行一个进程，但是在 1 秒钟期间，它可能运行多个进程，就是 CPU 在进行快速的切换，有时人们所说的伪并行就是指这种情况。

1. 创建进程

   操作系统中有四种事件会导致进程的创建：

   - 系统初始化，启动操作系统时，通常会创建若干个进程，分为前台进程和后台进程；
   - 执行了正在运行的进程所调用的进程创建系统调用；
   - 用户请求创建一个新的进程；
   - 一个批处理作业的初始化。

   从技术上来看，在所有这些情况中，新进程都是由一个已经存在的进程执行了一个用于创建进程的系统调用而创建的。这个进程可以是一个运行的用户过程，一个由键盘或者鼠标启动的系统进程或者一个批处理管理进程。这个进程所做的工作是执行一个用来创建新进程的系统调用。在 Linux/Unix 系统中提供了一个fork()，用来创建进程的子进程，在 python 的 os 模块中封装了常见的系统调用。代码如下：

```
1 import os
2 # os.getpid()获取父进程的ID
```

```
print("Process␣%s␣start..." % os.getpid())
# fock()调用一次会返回两次
pid = os.fork()
# 子进程返回0
if pid == 0:
    print("I␣am␣child␣process␣%s␣and␣my␣parent␣is␣%s"%(os.getpid(), os.
        getppid()))
# 父进程返回子进程的ID
else:
    print("I␣%s␣just␣created␣a␣child␣process␣%s"%(os.getpid(), pid))
```

7.3 客户端向服务端传送一个音频文件及信息

假定我们有一个客户端程序，一个服务端程序，要求从客户端发送一个音频到服务端，包括音频的名字、时长、采样率,采样宽度和通道数。应该怎么去做?这里面主要有四个库:socketserver，用于服务端部署，socket 用于客户端发送数据，struct 用于将音频的额外信息打包，wave 用于读取客户端音频及其信息，生成二进制采样点，等音频的这些数据发送到服务端的时候，再通过传送过来的音频数据和音频信息生成完全相同的音频。

之所以要这么一个奇怪的需求，是因为我们可以在服务端部署一个在线语音识别引擎，这样服务端的引擎一直在运行，当接收到客户端传送过来的数据的时候就开始在客户端也生成一个同样的音频，再调用识别模块对音频进行解析，最终生成文本再传送回客户端。这样我们就可以完成在线语音识别的任务了。那么首先遇到的一个问题就是……

……怎么传文件……

首先我们挨个库介绍下吧……

7.3.1 wave

wave是 python 中专门用于读取音频信息的一个库，可读可写，都是二进制的操作。音频有一些重要的信息包括采样率，采样宽度，时长和通道数，以及音频各个采样点的值都可以通过 wave 获得。通过下面的代码，我们就可以完成一个 wave 读取一个音频，再重新创建一个音频文件，写入读取的音频信息生成一个完全相同的音频的过程。有点类似于复制的感觉……不解释太多，一切尽在代码7.1中。

Listing 7.1: wave 库读取和写入音频

```
import sys
```

```
import wave
def read_wav(audio_path)
  f = wave.open(audio_path, 'rb')
  nchannels, samplewidth, framerate, nframes = f.getparams()[:4]
  audio_contents = f.readframes(nframes) #f.readframes(n)里面的n是帧数，
                                         #通过这个参数我们可以对音频进行裁剪
  f.close()
  return audio_contents, nchannels, samplewidth, framerate, nframes
def write_wav(audio_contents, nchannels, samplewidth, framerate,
    audio_path)
  f = wave.open(audio_path, 'wb')
  f.setnchannels(nchannels)
  f.setsampwidth(samplewidth)
  f.setframerate(framerate)
  f.writeframes(audio_contents)
  f.close()
audio_path, copyed_audio = sys.argv[1:] #外部给原始地址和存储地址
audio_contents, nchannels, samplewidth, framerate, nframes = read_wav(
    audio_path)
write_wav(audio_contents, nchannels, samplewidth, framerate, copyed_audio)
```

7.3.2 struct

struct是用来处理二进制数据的。有三个函数是比较重要的：pack、unpack 和 calsize。当我们调用 pack 这个函数的时候，格式是 struck.pack("<fmt>", data)，其中 <fmt> 指的是打包数据的格式，因为不同的格式在计算机中占据的存储空间不同，因此需要指定下，同样，当我们在调用 unpack 这个函数的时候，格式是 struct.unpack("<fmt>", data)，其格式跟 pack 函数差不多，因为解包的时候也一样得知道这个压缩的数据包中都是什么样的数据，这样可以根据这个 <fmt> 得到原始的数据，其返回的是一个 tuple。而 calsize 这个函数就是用来计算如果以格式"<fmt>" 打包或者解包所需要的存储空间，其格式是 struct.calsize("<fmt>") 。而不同类型的数据占据的字节数不同，表7.1列出了一般的数据类型、其表示和占据字节数。

这里面还有一些补充的信息如下：

- 每个格式前可以有一个数字，表示个数，比如"5i" 表示有五个整型数据，则占用 20 个字节；

表 7.1: struct 中常用的数据类型、C 语言的对应类型和占用字节数

Format	C Type	Python	字节数
x	pad byte	no value	1
c	char	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	*Bool*	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer or long	4
l	long	integer	4
L	unsigned long	long	4
q	long long	long	8
Q	unsigned long long	long	8
f	float	float	4
d	double	float	8
s	char[]	string	1
p	char[]	string	1
P	void *	long	unclear

- s格式表示一定长度的字符串，"4s" 表示长度为 4 的字符串，而 p 表示的是 pascal 字符串；
- q 和 Q 只在机器 64 位操作时有意义；
- P 用来转换一个指针，其长度和机器字长相关；

为了同 C 中的结构体交换数据，还要考虑有的 c 或者 C++ 编译器使用了字节对齐，通常是以 4 个字节为单位的 32 位系统，故而 struct 根据本地机器字节顺序转换，可以用格式中的第一个字符来改变对齐方式。这个字符的类型和定义如表7.2。

表 7.2: struct 中的字节对齐操作符号及其含义

Character	Bytes Order	Size and alignment
@	native	native 凑够 4 个字节
=	native	standard 按原字节数
<	little-endian	standard 按原字节数
>	big-endian	standard 按原字节数
!	network(=big-endian)	standard 按原字节数

在讲到 struct 在我们这个需求中应用之前，我们先看一些小例子，将不同类型的数据打包起来，再按照原始格式给解包。需要注意的是字符串必须先转换成二进制，先编码再转换；解包之后也需要进行解码才可以得到原始的字符串。一切尽在代码7.2中……

Listing 7.2: struct 打包和解包不同类型的数据

```
import struct
a,b,c,d,e = 1,2,3,'this', 'good'
d,e = bytes(d.encode('utf-8')), bytes(e.encode('utf-8'))
y = struct.pack("3i4s4s", a,b,c,d,e)
p,q,r,s,t = struct.unpack("3i4s4s", y)
s,t = s.decode('utf-8'), t.decode('utf-8')
```

如果我们需要通过 socket 传输一个音频，并且希望可以在服务器端可以完全重建音频，那么我们需要打包的音频信息有哪些呢？

- 音频长度：因为 socket 传输的时候，是根据服务端来确定接收多少个字节的，一个音频比较长，需要分成很多段去接收，那么什么时候算接收完了呢？就需要这个音频长度去确定了。
- 音频的基本信息：采样率，采样宽度，通道数。在服务器端重建时，wave 这个库需要用到这些信息；
- 音频的名字：我们需要在服务器端生成一个完全一样的同名 wav 文件，那么文件名是必须传输的。

那么我们在客户端打包的时候格式很好确定，音频长度、采样率、采样宽度和通道数共四个整数；那我们需要确定文件名的长度啊，不然服务端怎么解包，所以再加一个整数：文件名的长度；最后还有文件名。那么如果想要在服务器端直接利用这些信息的格式来解包，那么我们就需要将这个格式传输过去，因为这个格式还是字符串，所以我们还需要传输一个格式的长度，这个是整数。因此这个包就分为了两块：第一块是存储格式的长度和对应的格式；第二块是存储上面说的音频信息。所以结合上面7.1，我们可以这样处理，代码7.3中仅写了打包发送音频信息的部分，其他部分见后面 socket。

Listing 7.3: struct 打包音频信息和数据

```
import os
import struct
def encode(str):
  return bytes(str.encode('utf-8'))
cons, nchan, sampwid, frate, nfr = read_wav(audio_path)
filename = os.path.basename(audio_path)
#----------------------client-----------------------
fmt = ">4i%is"%(len(filename))
info = struct.pack(">i%is4i%is"%(len(fmt), len(filename)), \
                   len(fmt), encode(fmt), \
```

```
                len(cons), nchan, sampwid, \
                frate, encode(filename))
#--------------------server---------------------

fmt_len = struct.unpack(">i", info[:4])
fmt = struct.unpack("%is"%fmt_len, info[4:fmt_len])
fmt_size = struct.calsize(fmt)
info = info[(4+fmt_len):]
conlen, nchan, sampwid, frate, fname = \
      struct,unpack(fmt, info[:fmt_size])
```

7.3.3 socket

socket是个用来进行网络传输的库，爬虫的时候经常能看见这玩意，咱们介绍下 socket 常用的一些操作，并举一些例子，需要注意的是本需求中，我们只在客户端用 socket 库。

socket 常用的函数如下：

1. socket(socket.AF_INET, socket.SOCK_STREAM)：创建了一个 socket 连接的实例，括号内的参数可变，一般常用的就是这两个，其他的不甚了解，以后再补上；
2. bind((host, port)): 绑定 ip 和端口，host 是要连接的服务器端的 ip 地址，port 是服务器端的端口；
3. connect((host, port))：客户端连接服务端的 ip 和端口；
4. listen(n)：服务器端开始监听，其中 n 表示监听的队列的个数；
5. accept()：接收客户端传来的数据，返回两个值：(conn, address)，conn 是新的套接字对象，用来接收数据和返回数据，address 是客户端的地址和 ip，类型是 tuple，conn 是和 address 绑定的；
6. recv(byte)：接收数据，其中的 byte 表示一次接收多少个字节；
7. send(string)：发送二进制字符串，并返回发送的字节大小，就发送一次。这个字节长度可能是小于实际要发送的字节的长度的，假设要发送的字节数是 1025，服务端一次接收 1024 个字节，那么最后那个字节就丢了……所以如果用这个函数的话，可能就需要写一个多次发送的循环；
8. sendall(string)：发送二进制字符串，发送成功返回 None，失败则出错。这个就是整个数据都发送，发完为止；

基本的函数说完了，咱们就来分别写一个服务端和客户端的小 demo。见代码7.4和7.5，这里面用 struct 打包了要发送的数据的长度，为避免数据传输不完整。

Listing 7.4: socket 服务端代码

```
import socket
import struct
def server(host, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind((host, port))
    sock.listen(5)
    while True:
        conn, addr = sock.accept()
        print("get msg from ", addr)
        data = conn.recv(1024)
        len_msg = struct.unpack(">i", data[:4])[0]
        print(len_msg)
        msg = data[4:]
        if data:
            while len(msg) < len_msg:
                data = conn.recv(1024)
                msg += data
            print(msg)
            conn.sendall(msg)
        else:
            continue
if __name__ == "__main__":
    host = 'localhost'
    port = 8888
    server(host, port)
```

Listing 7.5: socket 客户端代码

```
import socket
import struct
def client(host, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
```

```
    msg = b"hello"
    sock.send(struct.pack('>i', len(msg))+msg)
    data = sock.recv(1024)
    print(data)
if __name__ == "__main__":
    host = 'localhost'
    port = 8888
    client(host, port)
```

7.3.4 socketserver

socketserver是一个搭建网络服务器的库，基于 socket 扩展的。将其用于服务端的搭建更省心一些。其有一些基类，再对这些类中的一些方法进行复写。其定义的类有不少，介绍四个，以后再补充。如下所示：

1. socketserver.TCPServer：负责处理 TCP 协议的类，网络传输数据的时候我们用这个比较多，因为比较稳定，这个是有链接的，客户端和客户端的交流只能是通过服务端来搞定；
2. socketserver.UDPServer：负责处理 UDP 协议的类，这个是没有中心链接的，客户端之间可以直接交流，可以用来写聊天器（存疑 ing）；
3. socketserver.BaseRequestHandler：开发者自定义的处理 request 的类，用来接收客户端的连接和数据传输等；
4. socketserver.ThreadingMixIn：用来处理多线程连接的类。

同样我们是根据一些小 demo 来理解这个库，我们就改写下 socket 中的服务器端的代码，见代码7.6。

Listing 7.6: socketserver 构建服务器端代码

```
import socketserver
import struct
class myTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    def __init__(self, address, handler):
        socketserver.TCPServer.__init__(self, address, handler)

class myTCPRequestHandler(socketserver.BaseRequestHandler):
    def handle(self):
        data = self.request.recv(1024)
        len_msg = struct.unpack(">i", data[:4])[0]
```

```
        msg = data[4:]
        while len(msg) < len_msg:
            data = self.request.recv(1024)
            msg += data
        print(msg)
        self.request.sendall(msg)
if __name__ == "__main__":
    host = 'localhost'
    port = 8888
    server = myTCPServer((host,port), myTCPRequestHandler)
    server.serve_forever()
```

7.3.5 网络传输音频并保存

okokok，累死我了。讲到这儿，我觉得把上面讲的这些串起来，写一个客户端发送音频，服务端接收音频并原样保存的代码并不困难了，那么直接把代码放上来，不做太多解释了，见代码7.7和7.8。

Listing 7.7: 音频传输的 client 端代码

```
import os
import sys
import wave
import struct
import socket
def callback():
    if sys.argv[1] is not None:
        audio_path = sys.argv[1]
        filename = os.path.basename(audio_path)
        cons, nchannels, samplewidth, framerate, _ = read_wav(audio_path)
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((HOST, PORT))
        # sock.sendall(struct.pack('>i', len(cons))+cons)
        fmt = ">4i%is"%len(filename)
        info = struct.pack('>i%is4i%is'%(len(fmt), len(filename)),\
                           len(fmt), strencode(fmt), \
```

```
                            len(cons), nchannels, \
                            samplewidth, framerate, \
                            strencode(filename))
        sock.sendall(info+cons)
        recived = sock.recv(1024)
        print("FILENAME:", recived)
def read_wav(path):
    f = wave.open(path)
    nchannels, samplewidth, framerate, nframes = f.getparams()[:4]
    cons = f.readframes(nframes)
    return cons,nchannels, samplewidth, framerate, nframes
def strencode(_str):
    return bytes(_str.encode('utf-8'))
if __name__ == "__main__":
    HOST = "localhost"
    PORT = 8888
    callback()
```

Listing 7.8: 音频传输的 server 端代码

```
import sys
import wave
import struct
import socketserver
class myTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    def __init__(self, address, handlerclass):
        socketserver.TCPServer.__init__(self, address, handlerclass)
class myTCPRequestHandler(socketserver.BaseRequestHandler):
    def handle(self):
        chunk = self.request.recv(1024) #一次只能接受1024个字节的数据，其他的会
            继续传过来
        len_fmt = struct.unpack('>i', chunk[:4])[0]
        fmt = struct.unpack('>%is'%len_fmt, chunk[4:4+len_fmt])[0].decode('
            utf-8')
```

```
        size_fmt = struct.calcsize(fmt)
        chunk = chunk[(4+len_fmt):]
        target_length, nchannels, samplewidth, framerate, filename = \
            struct.unpack(fmt, chunk[:size_fmt])
        filename = filename.decode('utf-8')
        cons = chunk[size_fmt:]
        while len(cons) < target_length:
            chunk = self.request.recv(1024)
            cons += chunk
        filename = self._write_to_file(cons, str(filename), nchannels,
            samplewidth, framerate)
        self.request.sendall(filename.encode('utf-8'))
    def _write_to_file(self, data, filename, nchannels, samplewidth,
        framerate):
        filename = filename.split('.')[0] + '_' + self.client_address[0] + '
            .wav'
        file = wave.open(filename, 'wb')
        file.setnchannels(nchannels)
        file.setframerate(framerate)
        file.setsampwidth(samplewidth)
        file.writeframes(data)
        file.close()
        return filename
if __name__ == "__main__":
    HOST = "localhost"
    PORT = 8888
    server = myTCPServer((HOST, PORT), myTCPRequestHandler)
    print('----------------------------')
    print("Server Start")
    print('----------------------------')
    server.serve_forever()
```

7.4 Python 中的正则表达式

第 8 章 C++ 学习笔记

```cpp
#include <iostream>
#include <string.h>
#include <stdlib.h>
using namespace std;

int main()
{
  printf("Hello World");
  return 0;
}
```

第 9 章 Docker

9.1 安装 docker 和 nvidia-docker

安装 docker：

```
wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/Centos-7.repo
yum install epel-release
yum install epel-release
yum install docker-ce
```

安装 nvidia-docker

```

distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.repo | sudo tee /etc/yum.repos.d/nvidia-docker.repo

sudo yum install -y nvidia-container-toolkit
sudo systemctl restart docker
# Install nvidia-docker and nvidia-docker-plugin
wget -P /tmp https://github.com/NVIDIA/nvidia-docker/releases/download/v1.0.1/nvidia-docker-1.0.1-1.x86_64.rpm
sudo rpm -i /tmp/nvidia-docker*.rpm && rm /tmp/nvidia-docker*.rpm
sudo systemctl start nvidia-docker

# Test nvidia-smi
nvidia-docker run --rm nvidia/cuda nvidia-smi
```

9.2 常用操作

镜像（image）和容器（container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除和暂停等。

容器的实质是进程，但与直接在宿主机执行的进行不同，容器进程运行于属于自己的独立的命名空间。

```
#开启镜像
sudo nvidia-docker run -it -v $(pwd)/DeepSpeech:/DeepSpeech -v /data1/asr_data:/mnt/data -v //data/kaldi/2019_0521_kaldi/kaldi-master:/mnt/kaldi paddlepaddle/deep_speech:latest-gpu /bin/bash
# 挂起镜像
Ctrl+P+Q
#运行已挂起的镜像
docker attach $CONTAINER_ID
```

9.3 实践要求

docker 如果想要挂载上本地的 IP 地址，可以再运行 docker 的时候加上指令 –net=host。如果要挂载本地物理磁盘，加上指令 -v。如果要将宿主机的端口与容器的端口绑定，可以使用 -p < 宿主机端口 >:< 容器端口 >。

```
nvidia-docker run -it --net=host -p 50001:22 -v $(pwd)/DeepSpeech:/DeepSpeech -v /data1/asr_data:/mnt/data -v /data/kaldi/2019_0521_kaldi/kaldi-master:/mnt/kaldi duhu/ds-server /bin/bash
```

1. 容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用数据卷（Volume）、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主（或网络存储）发生读写，其性能和稳定性更高。
2. 数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此，使用数据卷后，容器删除或者重新运行之后，数据却不会丢失。

9.4 docker 安装 TensorFlow

为了避免影响到主机上诸多配置，因此选用 docker 安装 TensorFlow，想怎么造就怎么造。安装 TensorFlow 时，使用以下指令就会启动该镜像安装。

```
docker pull tensorflow/tensorflow
```

第 10 章 数学知识总结

10.1 各类矩阵定义

定义 10.1. 转置矩阵

把矩阵 A 的行换乘同序数的列得到一个新矩阵，就叫做 A 的转置矩阵，记作 A^T。例如矩阵

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & -1 & 1 \end{bmatrix} \tag{10.1}$$

的转置矩阵为

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & -1 \\ 0 & 1 \end{bmatrix} \tag{10.2}$$

♣

定义 10.2. 对称矩阵

设 A 为 n 阶方阵，如果满足 $A^T = A$，即：

$$a_{ij} = a_{ji}(i, j = 1, 2, ..., n) \tag{10.3}$$

那么 A 称为对称矩阵，简称为对称阵。对称阵的特点是：它的元素以对角线为对称轴对应相等。

♣

定义 10.3. 复共轭矩阵

设 $A \in C^{m \times n}$，用 $\bar{A}$ 表示以 A 的元素的共轭复数为元素组成的矩阵，命：

$$A^H = (\bar{A})^T \tag{10.4}$$

则称 A^H 为 A 的复共轭转置矩阵。

♣

定义 10.4. Hermitian 矩阵

设 $A \in R^{n \times n}$，若 $A^H = A$，则称 A 为 Hermitian 矩阵。若 $A^H = -A$，则称 A 为反 Hermitian 矩阵。

♣

定义 10.5. 正交矩阵

如果 n 阶矩阵 A 满足

$$A^T A = E \tag{10.5}$$

即：

$$A^T = A^{-1} \tag{10.6}$$

则称 A 为正交矩阵，简称正交阵。 ♣

定义 10.6. 酉矩阵

如果 n 阶复矩阵 A 满足

$$A^H A = AA^H = E \tag{10.7}$$

则称 A 为酉矩阵，记作 $A \in U^{n\times n}$。 ♣

定义 10.7. 奇异矩阵

当 $|A| = 0$ 时，A 称为奇异矩阵，否则称为非奇异矩阵。A 是可逆矩阵的充分必要条件是 $|A| \neq 0$，即可逆矩阵就是非奇异矩阵。 ♣

定义 10.8. 正规矩阵

设 $A \in C^{n\times n}$，若：

$$A^H A = AA^H \tag{10.8}$$

则称 A 为正规矩阵，$A \in R^{n\times n}$，显然有 $A^H = A^T$，上式就变成了：

$$A^T A = AA^T \tag{10.9}$$

则称 A 为实正规矩阵。 ♣

定义 10.9. 幂等矩阵

设 $A \in C^{n\times n}$，若：

$$A^2 = A \tag{10.10}$$

则称 A 是幂等矩阵。 ♣

定义 10.10. 正定矩阵

设 $A \in C^{n\times n}$，若 A 的所有特征值均为正数，则称 A 为正定矩阵；若 A 的特征值均为非负数，则称 A 为版正定矩阵。

判断一个矩阵为正定矩阵的充要条件有：

1. A 的所有特征值 λ_i 均为正数；
2. $x^T Ax \geq 0$ 对所有非零向量 x 都成立；
3. 存在秩满矩阵 R，使得 $A = R^T R$。 ♣

定义 10.11. Jacobi 矩阵

假设某函数从 $f : R^n \to R^m$，从 $x \in R^n$ 映射到向量 $f(x) \in R^m$，其 Jacobi 矩阵的维度是 $m \times n$，如下所示：

$$H = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{10.11}$$

♣

定义 10.12. Hessian 矩阵

若实值函数 $f(x_1, x_2, ..., x_n)$ 的所有二阶偏导都存在并在定义域内连续，那么函数 f 的 Hessian 矩阵为：

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{10.12}$$

根据：

$$\frac{\partial^2 f}{\partial x_1 x_2} = \frac{\partial^2 f}{\partial x_2 x_1} \tag{10.13}$$

可知 Hessian 矩阵为对称阵。 ♣

10.2 瑞利商

对于一个Hermitian 矩阵M 及非零向量 x，瑞利商(Rayleigh quotient) 的定义如公式10.14，其中 x^H 为 x 的共轭转置向量。

$$R(M,x) = \frac{x^H M x}{x^H x} \tag{10.14}$$

若 M 和 x 中元素均为实数，瑞利商可以写成公式10.15。

$$R(M,x) = \frac{x^T M x}{x^T x} \tag{10.15}$$

设 M 的特征值与特征向量分别为 $\lambda_1,...,\lambda_n$ 和 $v_1,..,v_n$，且满足 $\lambda_{\min} = \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n = \lambda_{\max}$，那么在 M 已知的情况下有：

$$\begin{aligned} \max_x R(M,x) &= \lambda_n \\ \min_x R(M,x) &= \lambda_1 \end{aligned} \tag{10.16}$$

以下为证明公式10.16的过程：

由于 M 是 Hermitian 矩阵，存在一个酉矩阵 U，满足公式10.17。

$$M = UAU^T \tag{10.17}$$

其中 $A = diag\{\lambda_1,...,\lambda_n\}$。

因此公式10.15可以转换如下：

$$\begin{aligned} R(M,x) &= \frac{x^T UAU^T x}{x^T x} \\ &= \frac{(U^T x)^T A(U^T x)}{x^T x} \end{aligned} \tag{10.18}$$

设 $P = U^T x$，则：

$$\begin{aligned} R(M,x) &= \frac{P^T AP}{x^T x} \\ &= \frac{\sum_{i=1}^n \lambda_i |P_i|^2}{\sum_{i=1}^n |x_i|^2} \end{aligned} \tag{10.19}$$

根据特征值的大小关系，我们可以得到不等式10.20。

$$\lambda_1 \sum_{i=1}^{n} |P_i|^2 \le \sum_{i=1}^{n} \lambda_i |P_i|^2 \le \lambda_n \sum_{i=1}^{n} |P_i|^2 \tag{10.20}$$

所以公式10.19的范围如下：

$$\lambda_1 \frac{\sum_{i=1}^{n} |P_i|^2}{\sum_{i=1}^{n} |x_i|^2} \le R(M,x) \le \lambda_n \frac{\sum_{i=1}^{n} |P_i|^2}{\sum_{i=1}^{n} |x_i|^2} \tag{10.21}$$

设 U 第 i 行第 j 列的元素为 u_{ij}，则 U^T 第 i 行第 j 列的元素为 u_{ji}，由 $P = U^T x$ 和 $P^T = x^T U$ 可得：

$$\begin{aligned} p_i &= \sum_{i=1}^{n} u_{ji} x_j \\ p_i^T &= \sum_{j=1}^{n} x_i u_{ij} \end{aligned} \tag{10.22}$$

则：

$$|p_i|^2 = p_i^T p_i = \sum_{j=1}^{n} \sum_{k=1}^{n} x_j u_{ij} u_{ki} x_k \tag{10.23}$$

于是：

$$\begin{aligned} \sum_{i=1}^{n} |p_i|^2 &= \sum_{i=1}^{n} p_i^T p_i \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} x_j u_{ij} u_{ki} x_k \\ &= \sum_{j=1}^{n} \sum_{k=1}^{n} (\sum_{i=1}^{n} u_{ki} u_{ij}) x_j x_k \end{aligned} \tag{10.24}$$

因为 U 是酉矩阵，满足 $U^T U = I$，所以：

$$I_{jk} = \sum_{i=1}^{n} u_{ji} u_{ik} \tag{10.25}$$

其满足如下等式：

$$I_{jk} = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases} \tag{10.26}$$

结合10.24和10.26可得如下等式：

$$\sum_{i=1}^{n} |p_i|^2 = \sum_{i=1}^{n} |x_i|^2 \tag{10.27}$$

代入公式10.21可得：

$$\lambda_1 \leq R(M,x) \leq \lambda_n \tag{10.28}$$

且：

$$R(M,x) = \begin{cases} \lambda_1 & x = v_1 \\ \lambda_n & x = v_n \end{cases} \tag{10.29}$$

如果用 $x^{'} = cx$ 代入公式10.15有：

$$\begin{aligned} R(M,x^{'}) &= \frac{x^{'T} M x^{'}}{x^{'T} x^{'}} \\ &= \frac{c^2 x^T M x}{c^2 x^T x} \\ &= \frac{x^T M x}{x^T x} \end{aligned} \tag{10.30}$$

由此可以看出来对 x 进行缩放不影响瑞利商的值，即：

$$R(M,cx) = R(M,x) \tag{10.31}$$

因此我们可以限定 $x^T x = 1$，那么公式10.15可以简化为：

$$R(M,x) = x^T M x \tag{10.32}$$

那么 $R(M,x)$ 的极值就可以转换成约束条件下的拉格朗日乘法，如公式10.33。

$$L(x,\lambda) = x^T M x - \lambda(x^T x - 1) \tag{10.33}$$

对 x 求导并置为 0 可得：

$$\nabla L(x,\lambda) = Mx - \lambda x = 0 \tag{10.34}$$

即 M 的特征值能使得瑞利商取极值，且有：

$$R(M,x)=\lambda \tag{10.35}$$

瑞利商可以推广至广义瑞利商 (Generalized Rayleigh Quotient)，其形式如公式10.37。

$$R(A,B,x)=\frac{x^HAx}{x^HBx} \tag{10.36}$$

其中 A, B 均为 $n\times n$ 的 Hermitian 矩阵，且 B 为正定矩阵。

令 $x=B^{-\frac{1}{2}}x'$，广义瑞利商可以改写成：

$$\begin{aligned}R(A,B,x)&=\frac{(B^{-\frac{1}{2}}x')^HA(B^{-\frac{1}{2}}x'))}{(B^{-\frac{1}{2}}x')^HB(B^{-\frac{1}{2}}x')}\\&=\frac{x'^H(B^{-\frac{1}{2}})^HAB^{-\frac{1}{2}}x'}{x'^H(B^{-\frac{1}{2}})^HBB^{-\frac{1}{2}}x'}\\&=\frac{x'^H(B^{-\frac{1}{2}})^HAB^{-\frac{1}{2}}x'}{x'^Hx'}\end{aligned} \tag{10.37}$$

此时 $R(A,B,x)$ 的最大特征值和最小特征值即为 $(B^{-\frac{1}{2}})^HAB^{-\frac{1}{2}}$ 的最大和最小特征值。其实等价于当 $M=(B^{-\frac{1}{2}})^HAB^{-\frac{1}{2}}$ 时的 $R(M,x')$，$x'=B^{\frac{1}{2}}x$。

为简单起见，我们可以令 $P=B^{-\frac{1}{2}}$，公式10.37可以写作：

$$\begin{aligned}R(A,B,x)&=\frac{x'^H(B^{-\frac{1}{2}})^HAB^{-\frac{1}{2}}x'}{x'^Hx'}\\&=\frac{x'^HP^HAPx'}{x'^Hx'}\\&=\frac{(Px')^HAPx'}{x'^Hx'}\end{aligned} \tag{10.38}$$

类比上面提到的拉格朗日乘法，我们可以得到如下等式：

$$\nabla L(x,\lambda)=P^HAPx'-\lambda x'=0 \tag{10.39}$$

代入 $x'=P^{-1}x$ 有：

$$\nabla L(x,\lambda)=P^HAPP^{-1}x-\lambda P^{-1}x \tag{10.40}$$

解得：

$$PP^H Ax = \lambda x \tag{10.41}$$

又因为 $B^{-1} = PP^H$，所以最终求解特征值和特征向量可以依据：

$$B^{-1} Ax = \lambda x \tag{10.42}$$

10.3 EM 算法

本节来自于 Andrew Ng 的课堂讲义[10] 以及李航老师的《统计学习方法》[14]。

10.3.1 Jensen's Inequality

首先介绍一下Jensen 不等式。

定义 f 为实域的函数，若 $f''(x) \geq 0$，$x \in R$，则 f 为凸函数。若自变量 x 为向量，则当 f 关于 x 的 Hessian 矩阵为半正定矩阵，即 $H \geq 0$ 时，f 为凸函数；

如果对于所有的 $x \in R$，有 $f''(x) > 0$ 或对于所有的向量 x，有 $H > 0$，f 为严格意义上的凸函数。那么 Jensen 不等式定义定理10.1：

定理 10.1. Jsensen's Inequality

假定 f 是一个凸函数，X 是一个随机变量，那么：

$$E[f(X)] \geq f(EX) \tag{10.43}$$

且，如果 f 是严格凸函数，则当且仅当 $X = E[X]$ 时定理10.1取等号。一般地，用 Y 表示观测随机变量的数据，Z 表示隐随机变量的数据，Y 和 Z 连在一起称为完全数据。观测数据 Y 又称为不完全数据。假设给定观测数据 Y，其概率分布是 $P(Y|\theta)$，其中 θ 是需要估计的参数，那么不完全数据 Y 的似然函数是 $P(Y|\theta)$，对数似然函数是 $L(\theta) = \log P(Y|\theta)$；假设 Y 和 Z 的联合概率分布是 $P(Y, Z|\theta)$，那么完全数据的对数似然函数是 $L(\theta) = \log P(Y, Z|\theta)$。

EM 算法通过迭代求 $L(\theta) = \log P(Y|\theta)$ 的极大似然估计，每次迭代分为两步：E 步，求期望；M 步，求极大化。

输入：观测变量数据 Y，隐变量数据 Z，联合分布 $P(Y, Z|\theta)$，条件分布 $P(Z|Y, \theta)$

10.4 混合高斯分布

定义 10.13. 方差

方差用于描述数据的离散或波动程度。假定变量为 X，均值为 $\bar{X}$，N 为总体样本数，方差计算公式如下：

$$var(X) = \frac{\sum_{i=1}^{N}(X_i - \bar{X})^2}{N-1} \tag{10.44}$$

定义 10.14. 协方差

协方差表示了变量线性相关的方向，取值范围是 $[-\infty, \infty]$，一般来说协方差为正值，说明一个变量变大另一个变量也变大；取负值说明一个变量变大另一个变量变小，取 0 说明两个变量没有相关关系.

$$cov(X) = \frac{\sum_{i=1}^{N}(X_i - \bar{X})^2(Y_i - \bar{Y})^2}{N-1} \tag{10.45}$$

定义 10.15. 相关系数

协方差可反映两个变量之间的相互关系及相关方向，但无法表达其相关的程度，皮尔逊相关系数不仅表示线性相关的方向，还表示线性相关的程度，取值 $[-1, 1]$，也就是说，相关系数为正值，说明一个变量变大另一个变量也变大；取负值说明一个变量变大另一个变量变小，取 0 说明两个变量没有相关关系，同时，相关系数的绝对值越接近 1，线性关系越显著。

$$\rho_{XY} = \frac{cov(X,Y)}{\sqrt{DX}\sqrt{DY}} \tag{10.46}$$

定义 10.16. 协方差矩阵

当 $X \in R^n$ 为高维数据时，协方差矩阵可以很好的反映数据的性质，在协方差矩阵中，对角线元素反映了数据在各个维度上的离散程度，协方差矩阵为对角阵，非对角线元素反映了数据各个维度的相关性，其形式如下：

$$\Sigma = \begin{bmatrix} cov(x_1,x_1) & cov(x_1,x_2) & \cdots & cov(x_1,x_n) \\ cov(x_2,x_1) & cov(x_2,x_2) & \cdots & cov(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ cov(x_n,x_1) & cov(x_n,x_2) & \cdots & cov(x_n,x_n) \end{bmatrix} \tag{10.47}$$

单变量高斯分布公式如10.48，其中 μ 和 σ^2 分别为均值和方差。

$$N(x;\mu,\sigma)=\frac{1}{\sqrt{2\pi}\sigma}\exp\{-\frac{(x-\mu)^2}{2\sigma^2}\} \tag{10.48}$$

多变量高斯分布公式如10.49，其中 μ 和 Σ 分别为均值和协方差矩阵。

$$N(x;\mu,\Sigma)=\frac{1}{(2\pi)^{-\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}\exp\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\} \tag{10.49}$$

混合高斯模型 (Gaussian Mixture Model) 表示的是多个高斯分布叠加在一起的分布，其公式如10.50，其中 K 为高斯分量的个数，π_k 为各个分量的权重，其满足 $0\le\pi_k\le 1$，且 $\sum_{k=1}^{K}\pi_k=1$。$p(x)$ 表示的是多个高斯分量加权后的分布。

$$p(x)=\sum_{k=1}^{K}\pi_k N(x;\mu_k,\Sigma_k) \tag{10.50}$$

10.5 线性判别分析

线性判别分析（Linear Discriminative Analysis, LDA）是一种有监督的降维学习方法，其不仅仅可以达到降维的目的，还可以对原始数据进行聚类，使得类间距变大，类内距变小。有监督意味着 LDA 中所有的数据都是有标签的，这也是和 PCA 的一个重要区别，PCA 无需样本类别，是一种无监督的降维方法。

LDA 概况起来就是“投影后类内方差最小，类间方差最大。”LDA 是对数据进行投影，将其投影到低维空间，投影后相同类别的样本距离更近，不同类别的类别中心更远。本节首先对二类 LDA 进行分析，再推广至多类 LDA。

二类 LDA 的形象表述如图10.1右。左边的图不同类之间有交叉，决策边界有重合，而右图既使得相同类更集中，也使得不同类的分类边界更清晰，这就是 LDA 达到的效果。

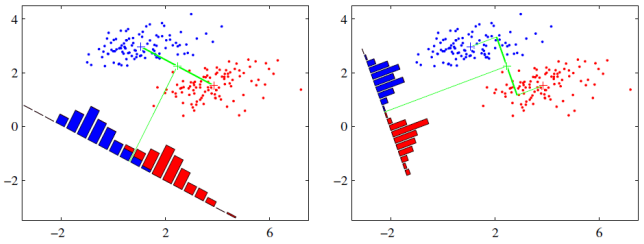

图 10.1: 二类 LDA 转换效果图

假定数据集 $D=\{(x_1,y_1),(x_2,y_2),...,(x_m,y_m)\}$，其中 $x_i\in R^n$，$y_i\in\{0,1\}$。我们定义 $N_j(j=0,1)$

为第 j 类样本的个数，$X_j(j=0,1)$ 为第 j 类样本的合集，$\mu_j(i=0,1)$ 为第 j 类样本的均值向量，$\Sigma_j(j=0,1)$ 为第 j 类样本缺少分母部分的协方差矩阵。那么 μ_j 和 Σ_j 的表达式分别如公式10.51和公式10.52所示。

$$\mu_j = \frac{1}{N_j}\sum_{x\in j} x \tag{10.51}$$

$$\Sigma_j = \sum_{x\in j}(x-\mu_j)(x-\mu_j)^T \tag{10.52}$$

由于只有两类数据，所以只需要将这些数据投影到一条直线上就可以，假设投影向量为 w，则对任意一个样本，其在直线上的投影为 $w^T x$，类别中心的投影分别为 $w^T\mu_0$ 和 $w^T\mu_1$，LDA 要求不同类别之间的类别中心尽可能的远，所以需要最大化 $\| w^T\mu_0 - w^T\mu_1 \|_2^2$，同时我们还希望同一类别尽可能接近，也就是样本投影之后的协方差尽可能的小，投影后的协方差如公式10.53。

$$\begin{aligned}\Sigma_j^{'} &= \sum_{x\in j}(w^T x - w^T\mu_j)(w^T x - w^T\mu_j)^T \\ &= \sum_{x\in j} w^T(x-\mu_j)(x-\mu_j)^T w \\ &= w^T\Sigma_j w\end{aligned} \tag{10.53}$$

所以我们希望最小化 $w^T\Sigma_0 w + w^T\Sigma_1 w$，由此我们可以得到需要优化的目标函数，如公式10.54。

$$\begin{aligned}\arg\max_w J(w) &= \arg\max_w \frac{\| w^T\mu_0 - w^T\mu_1 \|_2^2}{w^T\Sigma_0 w + w^T\Sigma_1 w} \\ &= \arg\max_w \frac{w^T(\mu_0-\mu_1)(\mu_0-\mu_1)^T w}{w^T(\Sigma_0+\Sigma_1)w}\end{aligned} \tag{10.54}$$

类内散度矩阵 S_w 和类间散度矩阵 S_b 分别定义为公式10.55和公式10.56。

$$S_w = \Sigma_0 + \Sigma_1 \tag{10.55}$$

$$S_b = (\mu_0-\mu_1)(\mu_0-\mu_1)^T \tag{10.56}$$

所以目标函数就变成了：

$$\arg\max_w J(w) = \arg\max_w \frac{w^T S_b w}{w^T S_w w} \tag{10.57}$$

也就是求解出 w 使得 $J(w)$ 最大。根据10.2中的介绍，我们可以通过计算矩阵 $S_w^{-1}S_b$ 的特征值和特征向量得到对应的 w，即求解公式10.58。$J(w)$ 的最大值为 $S_w^{-1}S_b$ 的最大特征值，最小值为 $S_w^{-1}S_b$ 的最小特征值，而 S_w 和 S_b 均可由原始数据求解得出，因此很容易就可以求解出 $J(w)$

的最大值。

$$S_w^{-1} S_b w = \lambda w \tag{10.58}$$

接下来我们分析下多类 LDA 的原理。

假定数据集 $D = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$，其中 $x_i \in R^n$，$y_i \in \{C_1, C_2, ..., C_k\}$。我们定义 $N_j (j = 0, 1, ..., k)$ 为第 j 类样本的个数，$X_j (j = 0, 1, ..., k)$ 为第 j 类样本的合集，$\mu_j (i = 0, 1, ..., k)$ 为第 j 类样本的均值向量，$\Sigma_j (j = 0, 1, ..., k)$ 为第 j 类样本缺少分母部分的协方差矩阵。此时是多类分类，因此投影后的空间不再是一条直线，而是一个超平面。假设投影后的低维空间维度为 d，对应的基向量为 $(w_1, w_2, ..., w_d)$，基向量组成的矩阵为 $W \in R^{n*d}$。

此时类内的散度矩阵 S_W 仍旧存在，如公式10.59。

$$S_W = \sum_{j=1}^{k} \Sigma_j \tag{10.59}$$

但是类间的散度矩阵就有所不同了。此时用每个类别的均值到全局均值的距离来衡量类间距如图10.2。

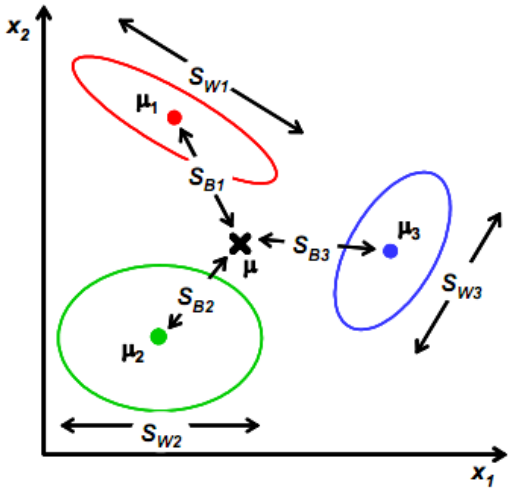

图 10.2: 多类 LDA 的类间散度矩阵示意图

其定义为公式10.60。

$$S_B = \sum_{j=1}^{k} N_j (\mu_j - \mu)(\mu_j - \mu)^T \tag{10.60}$$

其中：

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{10.61}$$

$$\mu_j = \frac{1}{N_j}\sum_{x_j \in C_j} x_j \tag{10.62}$$

同样此时的优化目标为公式10.63。

$$\arg\max_W J(W) = \arg\max_W \frac{W^T S_B W}{W^T S_W W} \tag{10.63}$$

此时目标函数求解转换成了公式10.64：

$$S_W^{-1} S_B W = \lambda W \tag{10.64}$$

以上，可以总结出多类 LDA 的求解步骤：

1. 计算每个类别的均值向量和方差，以及全局均值向量；
2. 根据均值向量和方差，计算 S_w 和 S_B；
3. 对 $S_W^{-1} S_B W = \lambda W$ 进行求解，求出 $S_W^{-1} S_B$ 的特征值和特征向量；
4. 对特征向量进行排序，设定低维空间的维度 d，选取前 d 个特征值和特征向量，特征向量组合成投影矩阵 W；
5. 通过投影矩阵计算出投影后的输入数据 $x_i^{'} = W^T x_i$；
6. 得到输出的新数据集：$\{(x_1^{'}, y_1), (x_2^{'}, y_2), ..., (x_m^{'}, y_m)\}$。

10.6 最大似然线性变换

最大似然线性变换（Maximum Likelihood Linear Transform）

在 HMM 系统中，协方差矩阵的选择可以是对角阵，分块对角阵或者全矩阵。相对于对角阵来说，全矩阵的优势在于对特征向量元素之间关系的建模，劣势在于参数量巨大。

10.7 Beta 分布

10.8 MLE 和 MAP

MLE vs MAP

第 11 章 端到端语音识别汇总

11.1 CTC

CTC 说实话，就是比较麻烦……我已经看过很多遍了，不过看的原理居多，代码这块，前后向的实现以及梯度的求取这些都没看过……先总结下 CTC 的基本原理吧还是。

列出本节参考的一些文献和博客：

1. CTC 的开山之作[2]《Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks》，不建议看这篇……因为这篇在讲到求输出序列的时候使用的前后向算法中前向和后向算法的时候对当前时刻 t 的输出概率算了两遍，后面还得再去掉，感觉没啥意思。其次在介绍前后向算法的时候不够简洁，比他的博士论文复杂不少；
2. Graves 大佬的博士毕业论文[1]《Supervised Sequence Labelling with Recurrent Neural Networks》里面的推导过程写的就相对来说更清楚一些，过程非常简洁，虽然还有一点点小错误，但是不影响整体的理解，所以推荐看这个来搞清楚 CTC 的前世今生；
3. 关于 CTC 的原理图形化描述请参考sequence modeling with ctc[7]，用很多小例子来展现 CTC 的原理和解码等，有助于去理解结果；
4. 关于 CTC 求导部分[13] 的得到最后一步结果可以参考教程：Connectionist Temporal Classification 详解补充；
5. 关于 CTC 前后向 python 代码实现请参考CTC 原理及实现。

以上资料看完，我觉得就完全可以理解 CTC 的原理了，当然手推公式是少不了的，接下来进入正题。

11.1.1 白话 CTC

首先明确：语音识别是一个序列分类的任务，那么其输入是逐帧的，如果不知道每一帧对应的标签，那么我们想要用 connectionist network 去干这件事，就得有目标函数。整个神经网络学习的准则就来自于目标函数，所以什么都可以没有，不能没有目标函数；话又说回来了，语音识别在训练的时候，逐帧输入，则必然对应着逐帧的输出，如果不想要对齐，想要端到端，那么就必须想办法把这些逐帧的输出映射成序列输出。真正难的地方就在这儿，怎么去映射，映射了之后怎么定义目标函数；

CTC 就是来干这个事情的。

我们先回忆一下使用传统的 HMM-DNN 模型来搭建语音识别框架，其中包含很多个子任务。首先我们需要使用 EM 算法训练 HMM-GMM 模型以拿到对齐的数据，即一帧对应一个音

素。其次以这些数据来训练 DNN 模型，训练好了 DNN 模型之后，通过 HMM 和 vocabulary 映射到更高的建模单元，再结合词典、语言模型来用 WFST 进行解码。

好的，这个过程很复杂，而且每一块信息量都很足，即要花很多的精力去学习和设计；

不用担心，现在端到端已经越来越火热，效果看上去也是越来越好了。那么咱们说一说 CTC 是怎么干的。

CTC 呢，有一些很重要的设定：（1）每一帧的输出是相互独立的；（2）只要最终映射出来的序列是对的，在任何时刻输出某个标签都可以。有个图很好玩，来自 PilgrimHui 的博客[12]，如图11.1。我觉得这个图很好的说明了 CTC 的内涵和特点。

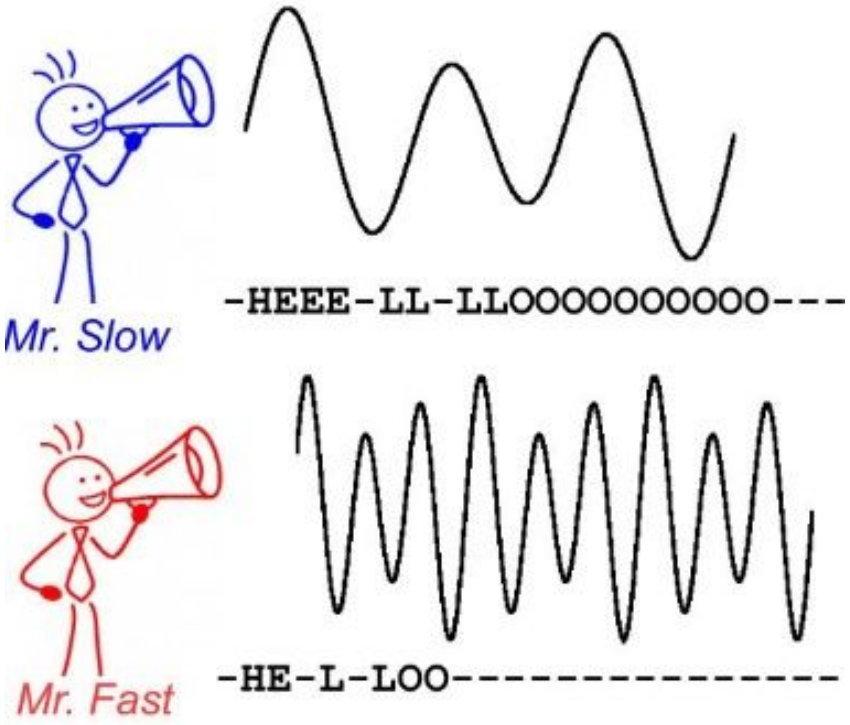

图 11.1: 同一单词不一样的输出音频的图解

第一个设定是为了后面使用最大似然估计去定义 loss 函数的，第二个设定呢就是彻底扔掉对齐数据的限制，按照这个设定，是没有固定的对齐结果的。在讲怎么从连续帧的输出映射到一整个序列之前，我们还要再讲一讲一些规则。为了避免相邻输出相同和更好的对建模单元的边界进行建模，CTC 引入了一个非常重要的输出标签：blank。你可以认为这个东西就是没有标签的意思，因为它的出现对句中无意义的片段比如静音段提供了建模单元。

假设字母表为 A，则 $A' = A \cup \{blank\}$，激活函数的输出 y_k^t 为 t 时刻输出 A' 中元素 k 的概率值，给定输入长度为 T 的音频序列 x；基于输出标签集合 A'，定义 A'^T 为长度为 T 的序列集合。

CTC 有如下假设：每一个时间步输出的标签概率值与其他时间步之间相互独立，或者说在

x 的条件下，相互独立。

那么对于 $\pi \in A'^{T}$，其条件概率如公式11.1。

$$P(\pi|x) = \prod_{t=1}^{T} y_{\pi_t}^{t} \tag{11.1}$$

为了和域 A 内的标签序列 l 区分开来，我们称 π 为域 A' 内的一条路径。由此定义一个 many-to-one 的函数：$\mathcal{F}: A'^{T} \mapsto A^{\leq T}$，因为最终系统要输出的是整个正常的序列，而不是穿插着一堆 blank 还有一堆重复 label 的奇奇怪怪的东西，我们得到的 π 是沿着时间线严格输出的标签值，而 l 是实际上某句话的内容，所以需要 $\mathcal{F}$ 来映射成我们需要的结果，这样才能叫做端到端呀。举个例子，比如某一段音频有 15 帧，这 15 帧说的是英文单词"beef"，这个就是 l，那实际上有 15 帧就会输出 15 帧的输出结果，可能是这样的"_ _ b b _ e _ _ e e _ f f _ _"，这个就是 π，$\mathcal{F}$ 干的事就是把 π 映射成 l，其规则是：

1. 合并重复的标签："_ _ b b _ e _ _ e e _ f f _ _" ⟶ "_ b _ e _ e _ f _";
2. 移除 blank："_ b _ e _ e _ f _" ⟶ "beef"。

这样的路径有很多很多条，而每一条路径都是排他的，因此我们就可以通过公式11.2计算出域 A 内某个标签序列的概率值了，即从一段音频中，输出一句话的概率值，CTC 牛逼。

$$P(l|x) = \sum_{\pi \in \mathcal{F}^{-1}(l)} P(\pi|x) \tag{11.2}$$

这种整合的意义在于我不在乎你在什么时间点输出什么，我只在乎最后映射的结果是我想要的。只要最终结果是一样的，中间有多少个 blank，同样的标签重复了多少次，我不在乎。这就使得 CTC 可以使用没有对齐的数据来进行语音识别。

好的，现在还剩下一些问题，公式11.2这玩意怎么计算，穷举吗？计算机表示：无能为力。序列越长，标签越多，尤其是碰到汉语这种用字建模的，基本上想都不要想。回想起 HMM 中讲到的前后向算法，CTC 也可以这么干啊，毕竟都是路径概率求解问题。

11.1.2 CTC 中的前后向算法

对于公式11.2，假设标签序列的长度为 U，音频的长度为 T，共有 $2^{T-U^2+U(T-3)}3^{(U-1)(T-U)-2}$ 条路径。

……

说实话，我不知道这个是怎么算出来的，但是还是挺恐怖的，还是按照上面举的例子，$U = 4$，$T = 15$，算一下就是 $2^{47}3^{31}$。

……

再见！

前后向算法的核心是标签序列 l 对应的所有路径概率和可以转换成标签前缀的迭代求和。

为了能让 blank 出现在输出路径中，我们将 l 修改下，在 l 的前后都加上 blank，同时在每个标签之间也都加上 blank，记作 $l^{'}$ 即"b e e f"⟶"_ b _ e _ e _ f _"。若 l 的长度为 U，则 $l^{'}$ 的长度为 $U^{'} = 2U + 1$。为计算 $l^{'}$ 的前缀概率，我们允许 blank 和 non-blank、非重复的 non-blank 之间直接转换，多解释下这句话的意思："beef" 这个词，"b" 在下一个时刻直接跳转到"b"、"e" 或者"_" 都是可以的，但是第一个"e" 只能跳转到自己本身"e" 和"_"，不可以跳转到下一个"e"，不然的话，在进行合并操作的时候，第二个"e" 就没了，这个需要好好理解，因为后面的前后向算法里面有用到这个特点。

对于一个标签序列 l，前向变量 $\alpha(t,u)$ 表示所有长度为 t 的路径的概率和，这些路径通过 $\mathcal{F}$ 映射到 l 中长度为 $u/2$ 的前缀，$u/2$ 向下取整。对于序列 s，$s_{p:q}$ 为 s 的子序列 $s_p, s_{p+1}, ..., s_{q-1}, s_q$，定义集合 $V(t,u) = \{\pi \in A^{''} : \mathcal{F}(\pi) = l_{1:u/2}, \pi_t = l^{'}_u\}$，则 $\alpha(t,u)$ 的计算如公式11.3。

$$\alpha(t,u) = \sum_{\pi \in V(t,u)} \prod_{i=1}^{t} y^i_{\pi_i} \tag{11.3}$$

我们可以通过 $t-1$ 时刻的前缀来计算 $\alpha(t,u)$。

给定上面的公式，我们就可以求得 l 的概率，其等于 T 时刻标签为 blank 和 non-blank 的前向概率和，如公式11.4。

$$p(l|x) = \alpha(T, U^{'}) + \alpha(T, U^{'} - 1) \tag{11.4}$$

所有正确的路径一定是以 blank 或者 l 中的第一个元素开头的，因此我们可以得到 $\alpha(t,u)$ 的初始值，如公式11.5。

$$\begin{aligned} \alpha(1,1) &= y^1_b \\ \alpha(1,2) &= y^1_{l_1} \\ \alpha(1,u) &= 0, \forall u > 2 \end{aligned} \tag{11.5}$$

初始化的值得到之后，我们可以通过对前缀进行迭代求和的方式得到当前时刻标签索引为 u 的路径概率值，如公式11.6。

$$\alpha(t,u) = y^t_{l^{'}_u} \sum_{i=f(u)}^{u} \alpha(t-1, i) \tag{11.6}$$

其中 $f(u)$ 如公式11.7。

$$f(u) = \begin{cases} u-1 & l^{'}_u = l^{'}_{u-2} \\ u-2 & otherwise \end{cases} \tag{11.7}$$

我觉得有必要解释下公式11.7。

公式11.6左边表示的是在 t 时刻标签索引为 u 的输出概率，右边是在已经确定了 t 时刻输出索引为 u 的情况下，$t-1$ 时刻的前向概率。因为已经确定了 t 时刻的输出，因此我们需要知道在 $t-1$ 时刻，哪些标签能够在下一个时刻转移到标签 u。对这些可能的标签前向概率求和，再乘以当前时刻的概率，就得到了我们想要的 $\alpha(t,u)$。

那么 $t-1$ 时刻究竟可能是哪些标签呢？

我们还是拿"beef" 举例子。

l ="b e e f"，l' ="_ b _ e _ e _ f _"，为了说的更清楚，我们来给 l' 里面的字母都来标个号，l' ="_(1) b(2) _(3) e(4) _(5) e(6) _(7) f(8) _(9)" 先来看第一种情况，如果 t 时刻的标签 u 有 $l'_u = l'_{u-2}$，那么 l'_u ="e(6)" 或者 l'_u ="_(i)",i={3,5,7,9}。

1. l'_u ="e(6)"：那么 $t-1$ 时刻的标签等于"e(6)" 是完全有可能的；等于 _(5) 也是完全可以的；那么能等于 e(4) 可以吗？假设可以，连起来看 $t-1$ 和 t 时刻的子序列，就是"e(4)e(6)"，乍一看好像没问题啊，但是要知道我们在训练的时候"e" 是标签，是没有"e(4)" 和"e(6)" 这种形式的标签的，他们都是"e"，根据上面说的 CTC 的合并规则，这俩就会合并成一个，无法表征两个相同的连续元素。也就是说，CTC 是不允许两个相同的连续元素直接跳转过去的，"e(4)" 是不可以直接跳转到"e(6)"，所以啊，$f(u) = u-1$ 表示前一个时刻的标签索引只可能和当前时刻的索引相同或者是当前时刻的索引的前一个；
2. l'_u ="_(i)",i={3,5,7,9}：这个解释起来和上面是一样一样的。同样我们的输出标签里面就只有"_" 这个东西，所以按照合并规则，但凡重复了的"_" 都看作是同一个"_"，因此是没有办法表征序列 l' 中的上一个"_"，因此 $f(u) = u-1$。

当 $l'_u \neq l'_{u-2}$，这个就比较无所谓了，咱们举个例子，假设 l'_u ="f(8)"，同样 $t-1$ 时刻的输出为"f(8)" 没得问题，"_(7)" 也没得问题，"e(6)" 当然也没得问题，反正他们不会乱合并。所以有 $f(u) = u-2$。

对应 t 时刻的 u 不能太小，太大无所谓大，大不了剩下的 $T-t$ 帧都重复输出某一个标签或者 blank 就可以了。如果太小，剩下的 $T-t$ 帧的输出都不够剩下还没有输出的标签了，所以考虑最极端的一种情况就是剩下的每一帧都输出一个 l 中的标签，且这些标签不会自身重复，也就是说每一帧输出的标签都只会输出一次，这样刚好够完整的输出 l。这就是临界条件，因此我们可以得到 u 的取值范围：

$$T-t \geq \frac{U'-u-1}{2} \tag{11.8}$$

解得：

$$u \geq U' - 2(T-t) - 1 \tag{11.9}$$

反过来说，也就是意味着：

$$\alpha(t,u) = 0, \forall u < U^{'} - 2(T-t) - 1 \tag{11.10}$$

前向算法讲完了，后向算法就很类似了。同样我们定义一个后向变量 $\beta(t,u)$，如果说想要得到输出序列 l，假定 t 时刻的输出是 $l^{'}_{u}$，$\beta(t,u)$ 可以看成是这一些路径的后部分，我们定义 $\beta(t,u)$ 为 $t+1$ 时刻配合上 $\alpha(t,u)$ 恰好构成完整的 l 的所有的后缀路径的概率总和。比照着 $V(t,u)$，定义集合 $W(t,u) = \{\pi \in A^{'T-t} : \mathcal{F}(\hat{\pi} + \pi) = l, \forall \hat{\pi} \in V(t,u)\}$。那么直接计算 $\beta(t,u)$ 的公式如11.11。

$$\beta(t,u) = \sum_{\pi \in W(t,u)} \prod_{i=1}^{T-t} y_{\pi_i}^{t+i} \tag{11.11}$$

后向算法的初始化如公式11.12。

$$\begin{aligned} \beta(T, U^{'}) &= 1 \\ \beta(T, U^{'} - 1) &= 1 \\ \beta(T, u) &= 0, \forall u < U^{'} - 1 \\ \beta(T, U^{'} + 1) &= 0 \end{aligned} \tag{11.12}$$

公式11.13描述了 $\beta(t,u)$ 的计算方式。

$$\beta(t,u) = \sum_{i=u}^{g(u)} \beta(t+1,i) y_{l^{'}_{i}}^{t+1} \tag{11.13}$$

其中：

$$g(u) = \begin{cases} u+1 & l^{'}_{u} = l^{'}_{u+2} \\ u+2 & otherwise \end{cases} \tag{11.14}$$

公式11.14表明了对 $t+1$ 时刻能达到的标签的限制，这部分和前向很类似，在此就不再赘述了。

同样后向算法中的 u 不能太大，因为 u 过大的意思就是前 t 个时刻得输出 u 个标签，如果说 t 稍微小了点，那就不够输出这么多标签了。同样，考虑临界条件：前 t 个时刻恰好能够无重复的输出 non-blank 标签。我们就可以得到如下不等式：

$$t \geq \frac{u}{2} \tag{11.15}$$

解得：

$$u \leq 2t \tag{11.16}$$

即：

$$\beta(t,u) = 0, \forall u > 2t \tag{11.17}$$

上述迭代过程，中间会涉及到概率的求和，如果直接就在计算机中按照上述方式实现的话，很容易就会导致 underflows，所以需要将概率变成 log 概率值，对于两个概率的和，我们一般使用公式11.18来实现，等到算完之后，再对其进行指数操作，就可以还原回来了。此外大部分的编程语言都会有一个稳定的函数去计算 $\log(1+x)$，当 x 接近于 0 的时候。

$$\log(e^a + e^b) = \max\{a,b\} + \log(1 + e^{-|a-b|}) \tag{11.18}$$

到这儿，CTC 中的前后向算法就讲完了，下一步是定义 loss 函数和求梯度。

11.1.3 CTC 中的 loss 函数和梯度

对于数据集 $\mathcal{S}$，CTC 的损失函数 $\mathcal{L}(\mathcal{S})$ 定义为数据集中所有输出序列正确的负 log 概率，如公式11.19。

$$\begin{aligned} \mathcal{L}(\mathcal{S}) &= -\ln \prod_{(x,z)\in S} p(z|x) \\ &= -\sum_{(x,z)\in S} \ln p(z|x) \end{aligned} \tag{11.19}$$

因为损失函数是可微分的，所以我们可以通过 BPTT（以后这块也得写个总结）去求权重对的梯度。所以任何一个基于梯度的非线性优化函数都可用来训练 CTC 网络。我们先看下单个样本的 loss，如公式11.20。

$$\mathcal{L}(x,z) = -\ln p(z|x) \tag{11.20}$$

则有：

$$\mathcal{L}(\mathcal{S}) = \sum_{(x,z)\in S} \mathcal{L}(x,z) \tag{11.21}$$

从而有：

$$\frac{\partial \mathcal{L}(S)}{\partial w} = \sum_{(x,z)\in S} \frac{\partial \mathcal{L}(x,z)}{\partial w} \tag{11.22}$$

设 $l = z$，定义集合 $X(t,u) = \{\pi \in A'^{T} : \mathcal{F}(\pi) = z, \pi_t = z'_u\}$，这个 $X(t,u)$ 表示的是所有最终整合之后的序列为 z，且在 t 时刻输出索引为 u 的路径集合。根据公式11.3和公式11.11，我们可以得到在集合 $X(t,u)$ 中的所有路径元素的概率和，如公式11.23。

$$\alpha(t,u)\beta(t,u) = \sum_{\pi\in X(t,u)} \prod_{t=1}^{T} y^t_{\pi_t} \tag{11.23}$$

公式11.23中说明了当 t 时刻，输出的标签索引为 u 的所有路径的可能性，那么我们希望获得的是输出 z 的概率值，那么我们只需要知道 t 时刻共有多少个可能的标签，然后对这些标签按照公式11.23去求解，最后将这些可能性都加起来，不就得到了输出序列 z 的概率值了吗？

所以我们得到了输出序列 z 的概率值如公式11.24。

$$p(z|x) = \sum_{u=1}^{|z'|} \alpha(t,u)\beta(t,u) \tag{11.24}$$

那么单个样本的损失函数我们就可以算出来了，如公式11.25。

$$\mathcal{L}(x,z) = -\ln \sum_{u=1}^{|z'|} \alpha(t,u)\beta(t,u) \tag{11.25}$$

ok，接下来，我们就求一下 t 时刻单个样本的损失函数对 softmax 层神经元输出的梯度，只要这个求出来了，其他的就比较简单了。其梯度如公式11.26。

$$\begin{aligned}
\frac{\partial \mathcal{L}(x,z)}{\partial y^t_k} &= -\frac{\partial \ln p(z|x)}{\partial y^t_k} \\
&= -\frac{1}{p(z|x)} \frac{\partial p(z|x)}{\partial y^t_k} \\
&= -\frac{1}{p(z|x)} \frac{\partial}{\partial y^t_k} \sum_{u=1}^{|z'|} \alpha(t,u)\beta(t,u) \\
&= -\frac{1}{p(z|x)} \sum_{u=1}^{|z'|} \frac{\partial \alpha(t,u)\beta(t,u)}{\partial y^t_k}
\end{aligned} \tag{11.26}$$

因为 $p(z|x)$ 是已知的，我们把重点放在求解 $\frac{\partial p(z|x)}{\partial y^t_k}$ 上。由于我们是对标签 k 所对应的神经

元输出求导，所以我们只需要去理会那些在 t 时刻输出为 k 的路径即可，那么我们定义个集合 $B(z,k)=\{u:z'_u=k\}$，结合公式11.23我们可以得到结论11.27。

$$\frac{\partial\alpha(t,u)\beta(t,u)}{\partial y_k^t}=\begin{cases}\frac{\alpha(t,u)\beta(t,u)}{y_k^t} & if\quad k\quad occurs\quad in\quad z'\\ 0 & otherwise\end{cases} \tag{11.27}$$

这里我又得嘴碎多说一句，为啥这个导数是这个样子的，因为从公式11.23中，我们可以看出来那些求和单元每一个路径中都包含了一个 y_k^t，这求导以后就把这个给弄没了，那么为了后续的运算，我们分子分母同时乘以 y_k^t，这样就可以得到这一块。

首先我们回忆下 softmax 的输出 $y_{k'}^t$ 对 a_k^t 的导数，见公式11.28，这块的求导细节不赘述了。

$$\frac{\partial y_{k'}^t}{\partial a_k^t}=y_{k'}^t(\delta_{k'k}-y_k^t) \tag{11.28}$$

接下来我们结合上面的这些信息来对 softmax 层前的那一层神经元的输出 a_k^t 求导，如公式11.29。

$$\begin{aligned}\frac{\partial\mathcal{L}(x,z)}{\partial a_k^t}&=\sum_{k'}\frac{\partial\mathcal{L}(x,z)}{\partial y_{k'}^t}\frac{\partial y_{k'}^t}{\partial a_k^t}\\&=-\frac{1}{p(z|x)y_{k'}^t}\sum_{k'}\sum_{u\in B(z,k)}\alpha(t,u)\beta(t,u)y_{k'}^t(\delta_{k'k}-y_k^t)\\&=-\frac{1}{p(z|x)}\sum_{k'}\sum_{u\in B(z,k)}\alpha(t,u)\beta(t,u)(\delta_{k'k}-y_k^t)\\&=-\frac{1}{p(z|x)}\sum_{u\in B(z,k)}\alpha(t,u)\beta(t,u)+\frac{1}{p(z|x)}\sum_{k'}\sum_{u\in B(z,k)}\alpha(t,u)\beta(t,u)y_k^t\\&=-\frac{1}{p(z|x)}\sum_{u\in B(z,k)}\alpha(t,u)\beta(t,u)+y_k^t\frac{1}{p(z|x)}\sum_{k'}\sum_{u\in B(z,k)}\alpha(t,u)\beta(t,u)\\&=-\frac{1}{p(z|x)}\sum_{u\in B(z,k)}\alpha(t,u)\beta(t,u)+y_k^t\end{aligned} \tag{11.29}$$

解释下倒数第二步是怎么转换到倒数第一步的。$\sum_{u\in B(z,k)}\alpha(t,u)\beta(t,u)$ 表示的是 t 时刻经过标签索引为 u 的所有的路径之和，加上限定 $u\in B(z,k)$ 之后，这个式子指的就是当第 u 个标签为 k 的所有路径的概率和。$\sum_{k'}$ 中的 k' 表示的就是标签，这一求和就意味着算了一下在 t 时刻

所有可能的标签的所有路径的概率和，咱们翻过头看看公式11.24，我们可以得到公式11.30。

$$\begin{aligned} p(z|x) &= \sum_{u=1}^{|z'|} \alpha(t,u)\beta(t,u) \\ &= \sum_{k'} \sum_{u \in B(z,k)} \alpha(t,u)\beta(t,u) \end{aligned} \tag{11.30}$$

所以……以上就是梯度的求导过程。另外原论文中在链式求导的那块有个公式错误如图11.2，多了个负号。

$$\frac{\partial \mathcal{L}(\mathbf{x},\mathbf{z})}{\partial y_k^t} = -\frac{1}{p(\mathbf{z}|\mathbf{x}) y_k^t} \sum_{u \in B(\mathbf{z},k)} \alpha(t,u)\beta(t,u). \tag{7.31}$$

Finally, to backpropagate the gradient through the output layer, we need the loss function derivatives with respect to the outputs a_k^t *before* the activation function is applied:

$$\frac{\partial \mathcal{L}(\mathbf{x},\mathbf{z})}{\partial a_k^t} = \ominus \sum_{k'} \frac{\partial \mathcal{L}(\mathbf{x},\mathbf{z})}{\partial y_{k'}^t} \frac{\partial y_{k'}^t}{\partial a_k^t} \tag{7.32}$$

where k' ranges over all the output units. Recalling that for softmax outputs

$$y_k^t = \frac{e^{a_k^t}}{\sum_{k'} e^{a_{k'}^t}}$$

图 11.2: 原 Alex 博士论文中链式求导部分的错误

11.1.4 CTC 的解码

CTC 的解码常用的有两种方式，greedy search 和 prefix beam search。greedy 解码速度很快，但是很容易出错，但是 prefix beam search 解码速度慢，准确率较高。接下来挨个介绍两种解码方式的算法和流程，以及对应的代码解释。

11.1.4.1 greedy search

greedy search，也就是 best path search，就是说找个每个时间点输出的最大概率值对应的标签，然后去重，再去掉 blank。这种方式很快，因为不需要考虑整个路径输出，只考虑当前时刻的输出即可。。但是这种方式效果不太好，输出可能比较奇怪。代码如下，来自[4]，这个库中还包含了一些关于 CTC 解码的论文。

```
def ctcBestPath(mat, classes):
  "implements best path decoding as shown by Graves (Dissertation, p63)"

  # dim0=t, dim1=c
  maxT, maxC = mat.shape #blank的索引是最后一个
```

```
  label = '' #初始化输出标签序列
  blankIdx = len(classes)
  lastMaxIdx = maxC # init with invalid label

  for t in range(maxT):
    maxIdx = np.argmax(mat[t, :]) #找到当前帧最大标签的索引

    if maxIdx != lastMaxIdx and maxIdx != blankIdx: #去重去blank
      label += classes[maxIdx]

    lastMaxIdx = maxIdx #重置上一个标签的索引

  return label
```

11.1.4.2 prefix beam search

First-Pass Large Vocabulary Continuous Speech Recognition using Bi-Directional Recurrent DNNs[8] 中针对 CTC 提出了一种 prefix beam search 的算法，这种算法能够避免全局搜索的复杂度过高无法实现的问题，同时比 greedy search 的结果要好很多。

首先介绍下一些公式。公式11.31是最终的解码公式，配合语言模型和网络输出（声学模型），得到最有可能的 W 词序列。

$$W = \arg\max_{W} p_{net}(W;X)p_{lm}^{\alpha}(W)|W|^{\beta} \tag{11.31}$$

其中 p_{net} 是网络的输出，p_{lm} 是语言模型的输出，α 和 β 分别为语言模型的权重和补偿系数。一般情况下，我们会降低语言模型对整体输出的影响，所以 α 一般取 0.2 0.7。

接下来我们讲一下 prefix beam search 的 demo[9]。

总体是有三个循环，第一个是时间维度上的，时间 t 从 1 到 T，也就是逐帧解码。第二个是对应候选输出序列的，这个是 beam search，那么一定得设置一个 beam size，那么会考虑所有的候选序列跟当前输出结合起来的概率值，那当前输出的话被剪枝之后还有很多个标签，再挨个的把每一个候选和每一个当前帧的标签进行结合计算。然后再利用这个结合的概率值进行重新排序得到新的候选。

```
pruned_alphabet = [alphabet[i] for i in np.where(ctc[t] > prune)[0]]
```

这一步是为了减少计算，也就是说先设定一个阈值，当前 t 时刻的时候，会做一个判断，只有当前时刻各个标签概率值大于 prune 的时候才会去做后续的操作，这就意味着当前时刻所有概率低于 prune 的标签都会被抛弃，不再参与到当前时刻的解码过程中。因为这些标签概率值太小，不太可能是正确的输出标签，留着只会增加计算量，还不如删掉，省时省心省力！

```
if len(l) > 0 and l[-1] == '>':
  Pb[t][l] = Pb[t - 1][l]
  Pnb[t][l] = Pnb[t - 1][l]
  continue
```

这一步就是判断下是不是到结尾了，结尾的表示符号是">"。如果到了结尾呢，说明这个时候输出的标签序列和 $t-1$ 时刻是一毛一样滴。t 时刻输出序列 l 以 blank 结尾的概率跟 $t-1$ 时刻以 blank 结尾的概率是一样的，t 时刻输出序列 l 以 non-blank 结尾的概率跟 $t-1$ 时刻以 non-blank 结尾的概率是一样的。然后就跳出当前时刻，因为当前时刻表示这个序列已经到了结尾了，没必要再折腾下去了。

```
if c == '%':
  Pb[t][l] += ctc[t][-1] * (Pb[t - 1][l] + Pnb[t - 1][l])
```

我们假设 % 代表 blank 这个标签。剪枝之后，会对还剩下的那些标签走一遍遍历，每一个标签都会尝试着和之前存起来的候选序列进行结合，算出来一个概率值。那么既然是遍历，当然会轮到牛逼的 blank。所以首先看看当前的这个标签是不是 blank。如果是 blank 的话，我们就没必要对当前的这个候选序列做啥子改动了，也就是当前时刻的输出标签序列还是 l，因为最终输出的时候，blank 也不会出现。既然当前这个标签是 blank，那么 t 时刻的标签序列 l 的概率需要和当前时刻输出为 blank 的概率结合一下，变成当前时刻的 $Pb[t][l]$。其计算公式从代码里就可以看到。

```
l_plus = l + c
if len(l) > 0 and c == l[-1]:
  Pnb[t][l_plus] += ctc[t][c_ix] * Pb[t - 1][l]
  Pnb[t][l] += ctc[t][c_ix] * Pnb[t - 1][l]
```

如果说当前时刻的标签不是 blank，而是上个时刻的这个候选序列的最后一个输出标签，也就是说当前时刻的输出和上一个时刻的候选序列尾部产生了重复，这个时候其实是有两种情况的，第一种情况是上一个时刻的输出标签正好是 blank，因为从上面那一步代码中我们可以看出

来，候选序列中是不会出现 blank 的，那如果是这种的情况，说明实际的输出序列就是有两个一样的字母，输出就是 *l_plus*，其尾部有两个一样的字母，这个时候候选序列的概率就等于当前时刻的标签概率乘以上一个时刻输出为 blank 的序列概率，也就是 $Pb[t-1][l]$；第二种情况是上一个时刻的输出确实也是这个标签，那么说明这个时候的候选序列不需要做啥变动，但是概率值需要变一下，当前时刻标签概率乘以上一个时刻输出是 non-blank 的序列概率值。

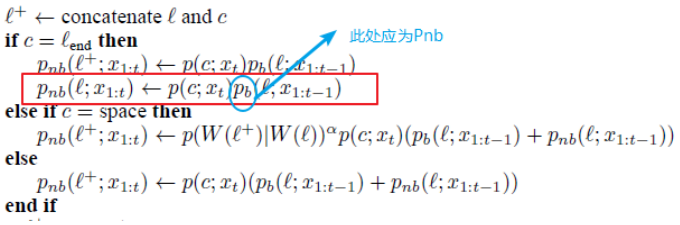

图 11.3: Prefix Beam Search 原论文中算法错误地方

另外原论文中关于这一块的计算步骤写错了，也就是算法中的这一步，如图11.3。简直坑死个人。

```
elif len(l.replace(' ', '')) > 0 and c in (' ', '>'):
  lm_prob = lm(l_plus.strip(' >')) ** alpha
  Pnb[t][l_plus] += lm_prob * ctc[t][c_ix] * (Pb[t - 1][l] + Pnb[t - 1][l
      ])
```

那还有可能当前的输出是’ ’(space)，也就是说输出是空格或者是结尾，这个时候说明一个完整的词出现了，我们就可以利用语言模型（LM）来对输出进行修正和约束，避免出现毫无意义的结果。那么这个词代入到语言模型中会出现一个概率值，当前候选序列的概率值就通过公式11.31来计算，从代码中也可以看出来。

```
Pnb[t][l_plus] += ctc[t][c_ix] * (Pb[t - 1][l] + Pnb[t - 1][l])
```

还有最后一种情况，就是既不是 blank，又不是 space，当前输出标签和候选标签序列的最后一个字母也不一样，这个时候，就直接算出候选标签序列和当前标签的概率乘积就行，候选标签序列也有两种情况：上一个时刻以 blank 或者以 non-blank 结尾。综上集中基本的情况都已经讲完了。

```
if l_plus not in A_prev:
  Pb[t][l_plus] += ctc[t][-1] * (Pb[t - 1][l_plus] + Pnb[t - 1][l_plus])
  Pnb[t][l_plus] += ctc[t][c_ix] * Pnb[t - 1][l_plus]
```

按照原始论文中，还有上面这个公式。也就是说算出来的 *l_plus* 不在候选标签序列里面，就会去之前时刻的候选序列里面去找，再利用之前的后续序列概率计算当前的概率值。百度的 Deep Speech2 代码里面说：这个部分不知道在干啥，还没啥用，所以就给去掉了。

我觉得……也不太好理解……

讲到这儿核心的代码部分已经讲完了，剩下的就是把当前时刻的标签，不管是以 blank 结尾的还是非 blank 结尾的综合起来，然后进行排序，根据 beam size 的大小得到新的候选序列，如此循环往复，直到这个序列输出到了尽头，就可以得到最终的结果啦~~

完整代码如下：

```
from collections import defaultdict, Counter
from string import ascii_lowercase
import re
import numpy as np

def prefix_beam_search(ctc, lm=None, k=25, alpha=0.30, beta=5, prune
    =0.001):
  """
  Performs prefix beam search on the output of a CTC network.

  Args:
    ctc (np.ndarray): The CTC output. Should be a 2D array (timesteps x
        alphabet_size)
    lm (func): Language model function. Should take as input a string and
        output a probability.
    k (int): The beam width. Will keep the 'k' most likely candidates at
        each timestep.
    alpha (float): The language model weight. Should usually be between 0
        and 1.
    beta (float): The language model compensation term. The higher the '
        alpha', the higher the 'beta'.
    prune (float): Only extend prefixes with chars with an emission
        probability higher than 'prune'.

  Retruns:
```

```
  string: The decoded CTC output.
"""

lm = (lambda l: 1) if lm is None else lm # if no LM is provided, just set to function returning 1
W = lambda l: re.findall(r'\w+[\s|>]', l)
alphabet = list(ascii_lowercase) + ['␣', '>', '%']
F = ctc.shape[1]
ctc = np.vstack((np.zeros(F), ctc)) # just add an imaginative zero'th step (will make indexing more intuitive)
T = ctc.shape[0]

# STEP 1: Initiliazation
O = ''
Pb, Pnb = defaultdict(Counter), defaultdict(Counter)
Pb[0][O] = 1
Pnb[0][O] = 0
A_prev = [O]
# END: STEP 1

# STEP 2: Iterations and pruning
for t in range(1, T):
  pruned_alphabet = [alphabet[i] for i in np.where(ctc[t] > prune)[0]]
  for l in A_prev:

    if len(l) > 0 and l[-1] == '>':
      Pb[t][l] = Pb[t - 1][l]
      Pnb[t][l] = Pnb[t - 1][l]
      continue

    for c in pruned_alphabet:
      c_ix = alphabet.index(c)
      # END: STEP 2

```

```
      # STEP 3: “Extending” with a blank
      if c == '%':
        Pb[t][l] += ctc[t][-1] * (Pb[t - 1][l] + Pnb[t - 1][l])
      # END: STEP 3

      # STEP 4: Extending with the end character
      else:
        l_plus = l + c
        if len(l) > 0 and c == l[-1]:
          Pnb[t][l_plus] += ctc[t][c_ix] * Pb[t - 1][l]
          Pnb[t][l] += ctc[t][c_ix] * Pnb[t - 1][l]
      # END: STEP 4

        # STEP 5: Extending with any other non-blank character and LM
            constraints
        elif len(l.replace('□', '')) > 0 and c in ('□', '>'):
          lm_prob = lm(l_plus.strip('□>')) ** alpha
          Pnb[t][l_plus] += lm_prob * ctc[t][c_ix] * (Pb[t - 1][l] + Pnb[t
              - 1][l])
        else:
          Pnb[t][l_plus] += ctc[t][c_ix] * (Pb[t - 1][l] + Pnb[t - 1][l])
        # END: STEP 5

        # STEP 6: Make use of discarded prefixes
        if l_plus not in A_prev:
          Pb[t][l_plus] += ctc[t][-1] * (Pb[t - 1][l_plus] + Pnb[t - 1][
              l_plus])
          Pnb[t][l_plus] += ctc[t][c_ix] * Pnb[t - 1][l_plus]
        # END: STEP 6

    # STEP 7: Select most probable prefixes
    A_next = Pb[t] + Pnb[t]
    sorter = lambda l: A_next[l] * (len(W(l)) + 1) ** beta
    A_prev = sorted(A_next, key=sorter, reverse=True)[:k]
```

```
    # END: STEP 7

return A_prev[0].strip('>')
```

11.2 RNN-Tranducer

RNN-Tranducer，简称 RNN-T，同样是 Alex Graves 大神的作品[6; 5]。

11.3 Attention

11.4 Transformer

11.5 CNNs

11.6 Mixed Models

11.6.1 Self-Attention Transducers for End-to-End Speech Recognition

这篇论文的作者是田正坤，来自中国科学院自动化所。本论文的主要贡献有：

1. 用 self-attention 模块替代了原来 RNN-T 中的 RNN 部分，可以用于并行计算；
2. 利用 path-aware regularization 帮助 SA-T 学习对齐；
3. 使用了 chunk-flow 机制来进行解码。

11.6.1.1 SA-T 的基本结构

11.6.1.2 path-aware regularization

11.6.1.3 chunk flow mechanism

11.7 如何计算 WER？

第 12 章 论文阅读笔记

12.1 Light Gated Recurrent Units for Speech Recogntion

Li-GRU是 Mirco Ravanelli 于 2018 年发表的论文，他也是pytorch-kaldi的作者。这篇论文主要针对的是对GRU(Gated Recurrent Units) 的改进，而且 Li-GRU 是专门为语音识别去设计的。本论文主要的工作有两方面：

（1）去掉了重置门 (reset gate)，去掉重置门对于模型的效果没什么影响，而且原始的 GRU 的重置门和更新门之间有冗余，因此去掉重置门的模型结构更合理；

（2）将原始 GRU 中的激活函数 Tanh 换成了 Relu，由于 Relu 本身函数具备的特性，其效果比 Relu 要好多。之所以以前的 RNN（包括 GRU 和 LSTM）不用 Relu，是因为 Relu 的值可以任意大，RNN 不停的迭代中，Relu 的值无法控制，容易导致数值不稳定（numerical instability）。本文作者采用了批量正则（Batch Normalization）的方式来避免数值不稳定的情况。这么做既可以避免梯度消失，又可以加速网络收敛，减少网络的时间。

本节的论文笔记分为三个部分：（1）GRU 的介绍；（2）Li-GRU 的学习；（3）实验配置和结果；（4）个人心得体会。

12.1.1 GRU 的介绍

语音识别是一个序列任务，那么上下文的信息对当前时刻信息的影响很大，RNN 的结构表明其可以动态的决定对于当前时间步使用多少上下文的信息。但是 RNN 存在的梯度消失和爆炸问题使得其学习长期依赖变得困难。所以一般我们都会使用一种门控 RNN(Gated RNN) 来解决这个问题。门控 RNN 的核心思想是引入一种门机制来控制不同时间步之间的信息流动。

常用的门控 RNN 有两种：LSTM 和 GRU。LSTM 的结构复杂且运算效率比较低，LSTM 有三个门，而 GRU 只有两个，所以运算起来快很多。而本论文也是基于 GRU 做的改进，所以不讨论 LSTM。

GRU 的计算公式如下：

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{12.1}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{12.2}$$

$$\tilde{h}_t = tanh(W_h x_t + U_h(h_{t-1} \odot r_t) + b_h) \tag{12.3}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t \tag{12.4}$$

其中

- x_t：t 时刻输入特征向量
- r_t：t 时刻重置门向量
- u_t：t 时刻更新门向量
- h_t：t 时刻状态向量
- h_t：t 时刻候选状态向量
- W, U, b：参数矩阵和向量

由公式12.4可知，当前状态向量 h_t 是前一刻状态向量 h_{t-1} 和当前时刻的状态候选向量 $\tilde{h}_t$ 之间的一个线性插值。两者之间的权重由更新门 z_t 决定，权重的值表示了更新信息的多少。这个线性插值就是 GRU 学习长期依赖的核心。如果 z_t 接近于 1，那么先前状态的信息就得以保留，以此学习到间隔长的时间步之间的信息关联。如果 z_t 接近于 0，那么网络更倾向于候选状态 $\tilde{h}_t$，而候选状态更依赖于当前输入和临近时间步的状态。同时候选状态还依赖于重置门 r_t，其使得模型通过忘记之前计算的状态来清除过去的记忆。

GRU 的模型结构图如12.1所示。

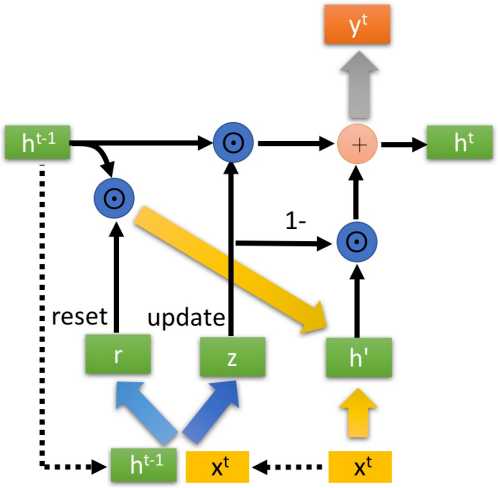

图 12.1: GRU 模型结构图

12.1.2 Li-GRU 的学习

本论文对 GRU 的改进主要涉及到三个部分：重置门、ReLU 激活函数和 BN。

1. 移除重置门：

对于序列中可能出现的 significant discontinuity，重置门可以起到一个清除过去信息的作用。比如说语言建模，当输入从一个句子跳转到另一个语义无关的句子的时候，重置门就可以起到很好的作用：避免过去无关信息对当前状态的干扰。但是对于语音识别来说，重置门的作用可能就不明显了，语音识别中的输入变化都比较小（一般的偏移量才 10ms），这表明过去的信息

还是挺有用的。即便是元音（Vowel）和擦音（Fricative）间的边界有很强的不连续现象，完全去掉过去信息也可能是有害的。另外基于一些音素的转移更相似，存住 phonotactic features 还是很有用的。

与此同时，重置门和更新门之间存在着某种冗余。也就是说 z_t 和 r_t 的变化比较同步，当前输入信息比较重要的时候，z_t 和 r_t 都比较小；过去时刻信息比较重要的时候，z_t 和 r_t 都比较大。拿 TIMIT 中的一段音频来看，更新门和重置门的平均激活值有着时域上的关联性，如图12.2。

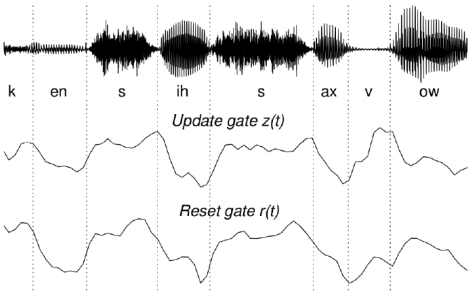

Fig. 1: Average activations of the update and reset gates for a GRU trained on TIMIT in a chunk of the utterance "*sx403*" of speaker "*faks0*".

图 12.2: TIMIT 中更新门与重置门音频上的时域关联

两个门之间的冗余程度可以量化描述，其公式 cross-correlation $C(z,r)$ 如下：

$$C(z,r) = \bar{z}_t \star \bar{r}_t \tag{12.5}$$

其中：

- $\bar{z}_t$：更新门神经元的平均激活值
- $\bar{r}_t$：重置门神经元的平均激活值
- $\star$：*cross-correlation* 的算子

图12.3显示了重置门和更新门的 cross-correlation。门激活值计算的是所有的输入帧，所有的隐藏神经元的激活值计算了个平均。从图中我们可以看到重置门和更新门之间相似度还挺高，因此冗余程度也很高。

因此我们决定去掉重置门，那么公式12.3就变成了公式12.6。这么处理之后，运算效率就提高了，就剩下一个门，所以 GRU 的结构变得更紧凑了。

$$\tilde{h}_t = tanh(W_h x_t + U_h h_{t-1} + b_h) \tag{12.6}$$

2. ReLU 激活函数

我们将 tanh 替换成 ReLU。tanh 其实在前馈神经网络中很少使用，因为当神经网络的层数变深时，它就容易陷入左右的边界值 -1 和 1，此时的梯度接近于 0，网络参数更新缓慢，收敛

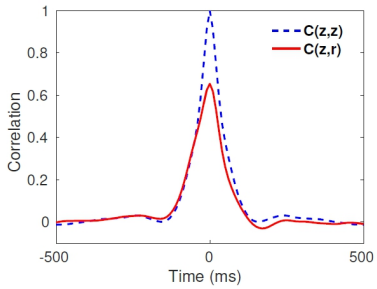

Fig. 2: Auto-correlation $C(z,z)$ and cross-correlation $C(z,r)$ between the average activations of the update (z) and reset (r) gates. Correlations are normalized by the maximum of $C(z,z)$ for graphical convenience.

图 12.3: Auto-correlation $C(z,z)$ 和 cross-correlation $C(z,r)$

的也就很慢。ReLU 就不存在这样的问题，但是 ReLu 的值域没有边界，因此容易出现一些数值问题，为了解决这个问题，我们将 ReLU 和 BN 一起使用，这样就很不存在数值问题了。将 tanh 改为 ReLU 之后，GRU 的公式如下：

$$\tilde{h}_t = ReLU(W_h x_t + U_h h_{t-1} + b_h) \tag{12.7}$$

3.BN

神经网络训练时，计算每一个 mini-batch 每层激活前输出的均值和方差，再利用均值和方差对激活前输出进行归一化能够解决所谓的 internal covariate shift 问题，这个就是传说中的 Batch Normalization。BN 既可以加快网络训练速度，又可以提高模型的效果。本文中只对前馈部分进行 BN 操作，因为其只对前馈神经网络进行操作，完全可以实现并行计算。其公式如下：

$$z_t = \sigma(BN(W_z x_t) + U_z h_{t-1}) \tag{12.8}$$

$$\tilde{h}_t = ReLU(BN(W_h x_t) + U_h h_{t-1}) \tag{12.9}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t \tag{12.10}$$

其中 $BN(\cdot)$ 如公式12.11所示，μ_b 和 σ_b 分别为当前 mini-batch 的均值和方差。

$$BN(a) = \gamma \odot \frac{a - \mu_b}{\sqrt{(\sigma_b)^2 + \epsilon}} + \beta \tag{12.11}$$

因为 BN 中已经包含了 β，因此之前公式中的偏置 b_z 和 b_h 就不再需要了。Li-GRU 将 ReLU 和 BN 结合起来，既利用了 ReLU 和 BN 两者的优点，同时还避免了 ReLU 的数值不稳定问题。

12.1.3 个人心得体会

综上所言，Li-GRU 通过减少了一个重置门来达到轻量级的效果，与此同时根据语音识别任务的特殊性，其认为语音序列任务中，对过往记忆清零对模型效果是有伤害的，而且重置门和更新门之间关联比较深，两个门显得冗余，因此其去掉了重置门。为了让网络更新更快，效果更好，以 Relu 函数代替了 tanh 作为候选状态的激活函数，同时以 BN 来解决 Relu 无边界的数值问题。BN 还可以帮助快速训练和提升效果。

附上一些音素的分类，如表12.1

表 12.1: 音素分类及示例

Phonetic Cat.	音素类别	Phone Lists
Vowels	元音	{iy, ih, eh, ae, ..., oy, aw, ow, er}
Liquids	流音	{l, r, y, w, el}
Nasals	鼻音	{en, m, n, ng}
Fricatives	擦音	{ch, jh, dh, z, v, f, th, s, sh, hh, ,zh}
Stops	塞音	{b, d, g, p, t, k, dx, cl, vcl, epi}

参考文献

[1] Graves. Alex. Supervised sequence labelling with recurrent neural networks. *Studies in Computational Intelligence*, 385:52–81, 2008.

[2] Graves. Alex, Santiago Fernández, Faustino J. Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *International Conference on Machine Learning*, 2006.

[3] L. Gillick and S. J. Cox. Some statistical issues in the comparison of speech recognition algorithms. In *in Proceedings ICASSP*, pages 532–535, 1989.

[4] githubharald. demo for greedy-search. https://github.com/githubharald/CTCDecoder.

[5] Alex Graves. Sequence transduction with recurrent neural networks. *CoRR*, abs/1211.3711, 2012.

[6] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[7] Awni Hannun. Sequence modeling with ctc. *Distill*, 2017. https://distill.pub/2017/ctc.

[8] Andrew L. Maas, Awni Y. Hannun, Daniel Jurafsky, and Andrew Y. Ng. First-pass large vocabulary continuous speech recognition using bi-directional recurrent dnns. *CoRR*, abs/1408.2873, 2014.

[9] Timothy I Murphy. demo for prefix-beam-search. https://github.com/corticph/prefix-beam-search.

[10] Andrew Ng. 统计学习方法. http://cs229.stanford.edu/notes/cs229-notes8.pdf.

[11] D. S. Pallet, W. M. Fisher, and J. G. Fiscus. Tools for the analysis of benchmark speech recognition tests. In *International Conference on Acoustics*, 1990.

[12] PilgrimHui. Ctc 自由输出的图解. https://www.cnblogs.com/liaohuiqiang/p/9953978.html.

[13] 农民小飞侠. 教程：connectionist temporal classification 详解补充. https://blog.csdn.net/w5688414/article/details/77867786.

[14] 李航. 统计学习方法. 2012. http://www.dgt-factory.com/uploads/2018/07/0725/%E7%BB%9F%E8%AE%A1%E5%AD%A6%E4%B9%A0%E6%96%B9%E6%B3%95.pdf.

[15] 菜鸟. awk 命令详解. https://www.runoob.com/linux/linux-comm-awk.html.

[16] 菜鸟. sed 命令详解. https://www.runoob.com/linux/linux-comm-sed.html.