语音识别学习笔记

学习大法好

作者：杜沪

组织：BIT, SpeechOcean

时间：August 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

2.1 基础知识

2.2 信号处理

2.3 声学模型

2.4 语言模型

2.5 解码器

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)

第 4 章 Linux 相关笔记

4.1 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。

```
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
```

4.2 Shell 指令笔记

4.2.1 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；
   - $?：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。
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 直接的内容，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。

4.2.2 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4.1。

表 4.1: 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4.2。

表 4.2: 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

(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4.3。

表 4.3: 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4.4。

表 4.4: 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

4.2.3 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4.1所示。

Listing 4.1: find 指令详解 commentstyle

```
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
```

4.2.4 grep

本小节分为两块一个是 grep 指令的一般性操作，如代码4.2。

Listing 4.2: grep 指令的一般性操作 commentstyle

```
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. ’^linux’：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. ’[^linux]’：查找不以 linux 开头的行，[] 内的’^’表示取反。

4.2.5 awk

4.2.6 sed

第 5 章 Python 笔记

5.1 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:{},\tlenB:{}".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)
```

5.2 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

5.2.1 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。

5.2.1.1 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
3 print("Process%sstart..." % 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))
```

5.3 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……

5.3.1 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5.1中。

Listing 5.1: 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)
```

5.3.2 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5.1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；
- 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；

表 5.1: 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

- 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5.2。

表 5.2: 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5.2中……

Listing 5.2: struct 打包和解包不同类型的数据

```
1 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5.1，我们可以这样处理，代码5.3中仅写了打包发送音频信息的部分，其他部分见后面 socket。

Listing 5.3: 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])
```

5.3.3 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5.4和5.5，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 5.4: 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 5.5: 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)
```

5.3.4 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5.6。

Listing 5.6: 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()
```

5.3.5 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5.7和5.8。

Listing 5.7: 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 5.8: 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()
```

第 6 章 Docker

6.1 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
```

6.2 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。

6.3 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。

```
1 docker pull tensorflow/tensorflow
```

第 7 章 数学知识总结

7.1 各类矩阵定义

定义 7.1. 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

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & -1 & 1 \end{bmatrix} \tag{7.1}$$

的转置矩阵为

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & -1 \\ 0 & 1 \end{bmatrix} \tag{7.2}$$

♣

定义 7.2. 对称矩阵

设 A 为 n 阶方阵，如果满足 $A^T = A$，即：

$$a_{ij} = a_{ji}(i, j = 1, 2, ..., n) \tag{7.3}$$

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 7.3. 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：

$$A^H = (\bar{A})^T \tag{7.4}$$

则称 A^H 为 A 的复共轭转置矩阵。

♣

定义 7.4. 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 7.5. 正交矩阵

如果 n 阶矩阵 A 满足

$$A^T A = E \tag{7.5}$$

即：

$$A^T = A^{-1} \tag{7.6}$$

则称 A 为正交矩阵，简称正交阵。 ♣

定义 7.6. 酉矩阵

如果 n 阶复矩阵 A 满足

$$A^H A = AA^H = E \tag{7.7}$$

则称 A 为酉矩阵，记作 $A \in U^{n\times n}$。 ♣

定义 7.7. 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 7.8. 正规矩阵

设 $A \in C^{n\times n}$，若：

$$A^H A = AA^H \tag{7.8}$$

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：

$$A^T A = AA^T \tag{7.9}$$

则称 A 为实正规矩阵。 ♣

定义 7.9. 幂等矩阵

设 $A \in C^{n\times n}$，若：

$$A^2 = A \tag{7.10}$$

则称 A 是幂等矩阵。 ♣

定义 7.10. 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 $A = R^T R$。 ♣

7.2 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7.11，其中 x^H 为 x 的共轭转置向量。

$$R(M,x) = \frac{x^H Mx}{x^H x} \tag{7.11}$$

若 M 和 x 中元素均为实数，瑞利商可以写成公式7.12。

$$R(M,x) = \frac{x^T Mx}{x^T x} \tag{7.12}$$

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：

$$\begin{aligned} \max_x R(M,x) &= \lambda_n \\ \min_x R(M,x) &= \lambda_1 \end{aligned} \tag{7.13}$$

以下为证明公式7.13的过程：

由于 M 是 Hermitian 矩阵，存在一个酉矩阵 U，满足公式7.14。

$$M = UAU^T \tag{7.14}$$

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

因此公式7.12可以转换如下：

$$\begin{aligned} R(M,x) &= \frac{x^T U A U^T x}{x^T x} \\ &= \frac{(U^T x)^T A (U^T x)}{x^T x} \end{aligned} \tag{7.15}$$

设 $P = U^T x$，则：

$$\begin{aligned} R(M,x) &= \frac{P^T A P}{x^T x} \\ &= \frac{\sum_{i=1}^{n} \lambda_i |P_i|^2}{\sum_{i=1}^{n} |x_i|^2} \end{aligned} \tag{7.16}$$

根据特征值的大小关系，我们可以得到不等式7.17。

$$\lambda_1 \sum_{i=1}^{n} |P_i|^2 \le \sum_{i=1}^{n} \lambda_i |P_i|^2 \le \lambda_n \sum_{i=1}^{n} |P_i|^2 \tag{7.17}$$

所以公式7.16的范围如下：

$$\lambda_1 \frac{\sum_{i=1}^{n} |P_i|^2}{\sum_{i=1}^{n} |x_i|^2} \le R(M,x) \le \lambda_n \frac{\sum_{i=1}^{n} |P_i|^2}{\sum_{i=1}^{n} |x_i|^2} \tag{7.18}$$

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：

$$\begin{aligned} p_i &= \sum_{i=1}^{n} u_{ji} x_j \\ p_i^T &= \sum_{j=1}^{n} x_i u_{ij} \end{aligned} \tag{7.19}$$

则：

$$|p_i|^2 = p_i^T p_i = \sum_{j=1}^{n} \sum_{k=1}^{n} x_j u_{ij} u_{ki} x_k \tag{7.20}$$

于是：

$$\begin{aligned}\sum_{i=1}^{n}|p_i|^2 &= \sum_{i=1}^{n}p_i^T p_i \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}x_j u_{ij}u_{ki}x_k \\ &= \sum_{j=1}^{n}\sum_{k=1}^{n}(\sum_{i=1}^{n}u_{ki}u_{ij})x_j x_k\end{aligned} \tag{7.21}$$

因为 U 是酉矩阵，满足 $U^T U = I$，所以：

$$I_{jk} = \sum_{i=1}^{n}u_{ji}u_{ik} \tag{7.22}$$

其满足如下等式：

$$I_{jk} = \begin{cases}1 & j = k \\ 0 & j \neq k\end{cases} \tag{7.23}$$

结合7.21和7.23可得如下等式：

$$\sum_{i=1}^{n}|p_i|^2 = \sum_{i=1}^{n}|x_i|^2 \tag{7.24}$$

代入公式7.18可得：

$$\lambda_1 \leq R(M,x) \leq \lambda_n \tag{7.25}$$

且：

$$R(M,x) = \begin{cases}\lambda_1 & x = v_1 \\ \lambda_n & x = v_n\end{cases} \tag{7.26}$$

如果用 $x' = cx$ 代入公式7.12有：

$$\begin{aligned}R(M,x') &= \frac{x'^T M x'}{x'^T x'} \\ &= \frac{c^2 x^T M x}{c^2 x^T x} \\ &= \frac{x^T M x}{x^T x}\end{aligned} \tag{7.27}$$

由此可以看出来对 x 进行缩放不影响瑞利商的值，即：

$$R(M, cx) = R(M, x) \tag{7.28}$$

因此我们可以限定 $x^T x = 1$，那么公式7.12可以简化为：

$$R(M, x) = x^T M x \tag{7.29}$$

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7.30。

$$L(x, \lambda) = x^T M x - \lambda(x^T x - 1) \tag{7.30}$$

对 x 求导并置为 0 可得：

$$\nabla L(x, \lambda) = Mx - \lambda x = 0 \tag{7.31}$$

即 M 的特征值能使得瑞利商取极值，且有：

$$R(M, x) = \lambda \tag{7.32}$$

瑞利商可以推广至广义瑞利商 (Generalized Rayleigh Quotient)，其形式如公式7.34。

$$R(A, B, x) = \frac{x^H A x}{x^H B x} \tag{7.33}$$

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：

$$\begin{aligned} R(A, B, x) &= \frac{(B^{-\frac{1}{2}} x')^H A (B^{-\frac{1}{2}} x'))}{(B^{-\frac{1}{2}} x')^H B (B^{-\frac{1}{2}} x')} \\ &= \frac{x'^H (B^{-\frac{1}{2}})^H A B^{-\frac{1}{2}} x'}{x'^H (B^{-\frac{1}{2}})^H B B^{-\frac{1}{2}} x'} \\ &= \frac{x'^H (B^{-\frac{1}{2}})^H A B^{-\frac{1}{2}} x'}{x'^H x'} \end{aligned} \tag{7.34}$$

此时 $R(A, B, x)$ 的最大特征值和最小特征值即为 $(B^{-\frac{1}{2}})^H A B^{-\frac{1}{2}}$ 的最大和最小特征值。其实等价于当 $M = (B^{-\frac{1}{2}})^H A B^{-\frac{1}{2}}$ 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7.34可以写作：

$$\begin{aligned} R(A,B,x) &= \frac{x'^H (B^{-\frac{1}{2}})^H A B^{-\frac{1}{2}} x'}{x'^H x'} \\ &= \frac{x'^H P^H A P x'}{x'^H x'} \\ &= \frac{(Px')^H A P x'}{x'^H x'} \end{aligned} \tag{7.35}$$

类比上面提到的拉格朗日乘法，我们可以得到如下等式：

$$\nabla L(x,\lambda) = P^H A P x' - \lambda x' = 0 \tag{7.36}$$

代入 $x' = P^{-1}x$ 有：

$$\nabla L(x,\lambda) = P^H A P P^{-1} x - \lambda P^{-1} x \tag{7.37}$$

解得：

$$P P^H A x = \lambda x \tag{7.38}$$

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

$$B^{-1} A x = \lambda x \tag{7.39}$$

7.3 混合高斯分布

定义 7.11. 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：

$$var(X) = \frac{\sum_{i=1}^{N}(X_i - \bar{X})^2}{N-1} \tag{7.40}$$

定义 7.12. 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.

$$cov(X) = \frac{\sum_{i=1}^{N}(X_i - \bar{X})^2(Y_i - \bar{Y})^2}{N-1} \tag{7.41}$$

定义 7.13. 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。

$$\rho_{XY} = \frac{cov(X,Y)}{\sqrt{DX}\sqrt{DY}} \tag{7.42}$$

定义 7.14. 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:

$$\Sigma = \begin{bmatrix} cov(x_1,x_1) & cov(x_1,x_2) & \cdots & cov(x_1,x_n) \\ cov(x_2,x_1) & cov(x_2,x_2) & \cdots & cov(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ cov(x_n,x_1) & cov(x_n,x_2) & \cdots & cov(x_n,x_n) \end{bmatrix} \tag{7.43}$$

单变量高斯分布公式如7.44，其中 μ 和 σ^2 分别为均值和方差。

$$\mathcal{N}(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\{-\frac{(x-\mu)^2}{2\sigma^2}\} \tag{7.44}$$

多变量高斯分布公式如7.45，其中 μ 和 Σ 分别为均值和协方差矩阵。

$$\mathcal{N}(x;\mu,\Sigma) = \frac{1}{(2\pi)^{-\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}\exp\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\} \tag{7.45}$$

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7.46，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。

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k) \tag{7.46}$$

7.4 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7.1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。

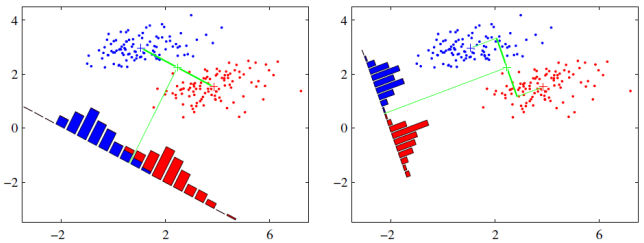

图 7.1: 二类 LDA 转换效果图

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7.47和公式7.48所示。

$$\mu_j = \frac{1}{N_j} \sum_{x \in j} x \tag{7.47}$$

$$\Sigma_j = \sum_{x \in j} (x - \mu_j)(x - \mu_j)^T \tag{7.48}$$

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7.49。

$$\begin{aligned}\Sigma_j' &= \sum_{x\in j}(w^T x - w^T\mu_j)(w^T x - w^T\mu_j)^T \\ &= \sum_{x\in j} w^T(x-\mu_j)(x-\mu_j)^T w \\ &= w^T\Sigma_j w\end{aligned} \tag{7.49}$$

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7.50。

$$\begin{aligned}\arg\max_w J(w) &= \arg\max_w \frac{\| w^T\mu_0 - w^T\mu_1 \|_2^2}{w^T\Sigma_0 w + w^T\Sigma_1 w} \\ &= \arg\max_w \frac{w^T(\mu_0-\mu_1)(\mu_0-\mu_1)^T w}{w^T(\Sigma_0+\Sigma_1)w}\end{aligned} \tag{7.50}$$

类内散度矩阵 S_w 和类间散度矩阵 S_b 分别定义为公式7.51和公式7.52。

$$S_w = \Sigma_0 + \Sigma_1 \tag{7.51}$$

$$S_b = (\mu_0-\mu_1)(\mu_0-\mu_1)^T \tag{7.52}$$

所以目标函数就变成了：

$$\arg\max_w J(w) = \arg\max_w \frac{w^T S_b w}{w^T S_w w} \tag{7.53}$$

也就是求解出 w 使得 $J(w)$ 最大。根据7.2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7.54。$J(w)$ 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。

$$S_w^{-1}S_b w = \lambda w \tag{7.54}$$

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7.55。

$$S_W = \sum_{j=1}^{k} \Sigma_j \tag{7.55}$$

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7.2。

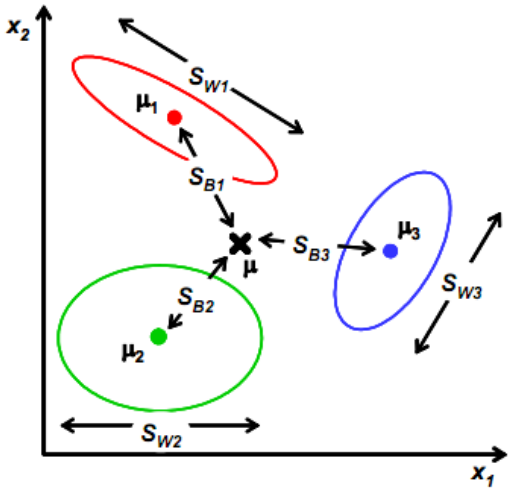

图 7.2: 多类 LDA 的类间散度矩阵示意图

其定义为公式7.56。

$$S_B = \sum_{j=1}^{k} N_j(\mu_j - \mu)(\mu_j - \mu)^T \tag{7.56}$$

其中：

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{7.57}$$

$$\mu_j = \frac{1}{N_j}\sum_{x_j \in C_j} x_j \tag{7.58}$$

同样此时的优化目标为公式7.59。

$$\arg\max_W J(W) = \arg\max_W \frac{W^T S_B W}{W^T S_W W} \tag{7.59}$$

此时目标函数求解转换成了公式7.60：

$$S_W^{-1} S_B W = \lambda W \tag{7.60}$$

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：$\{(x_1^{'}, y_1), (x_2^{'}, y_2), ..., (x_m^{'}, y_m)\}$。

7.5 EM 算法

本节来自于 Andrew Ng 的课堂讲义[3] 以及李航老师的《统计学习方法》[4]。

7.5.1 Jensen's Inequality

首先介绍一下Jensen 不等式。

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 $P(Z|Y, \theta)$

7.6 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。

7.7 Beta 分布

7.8 MLE 和 MAP

MLE vs MAP

第 8 章 Connectionist Temporal Classification

8.1 CTC 的原理

8.2 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。

8.2.1 greedy search

8.2.2 prefix beam search

First-Pass Large Vocabulary Continuous Speech Recognition using Bi-Directional Recurrent DNNs[1] 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8.1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。

$$W = \arg\max_{W} p_{net}(W;X)p_{lm}^{\alpha}(W)|W|^{\beta} \tag{8.1}$$

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[2]。

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。

$\ell^+ \leftarrow$ concatenate ℓ and c
if $c = \ell_{end}$ **then**
  $p_{nb}(\ell^+; x_{1:t}) \leftarrow p(c; x_t) p_b(\ell; x_{1:t-1})$
  $p_{nb}(\ell; x_{1:t}) \leftarrow p(c; x_t) p_b(\ell; x_{1:t-1})$
else if $c =$ space **then**
  $p_{nb}(\ell^+; x_{1:t}) \leftarrow p(W(\ell^+)|W(\ell))^\alpha p(c; x_t)(p_b(\ell; x_{1:t-1}) + p_{nb}(\ell; x_{1:t-1}))$
else
  $p_{nb}(\ell^+; x_{1:t}) \leftarrow p(c; x_t)(p_b(\ell; x_{1:t-1}) + p_{nb}(\ell; x_{1:t-1}))$
end if

此处应为Pnb

图 8.1: 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8.1。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8.1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]
```

```
82      # END: STEP 7
83
84    return A_prev[0].strip('>')
```

第 9 章 论文阅读笔记

9.1 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。

9.1.1 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：

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{9.1}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{9.2}$$

$$\tilde{h}_t = tanh(W_h x_t + U_h(h_{t-1} \odot r_t) + b_h) \tag{9.3}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t \tag{9.4}$$

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9.4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9.1所示。

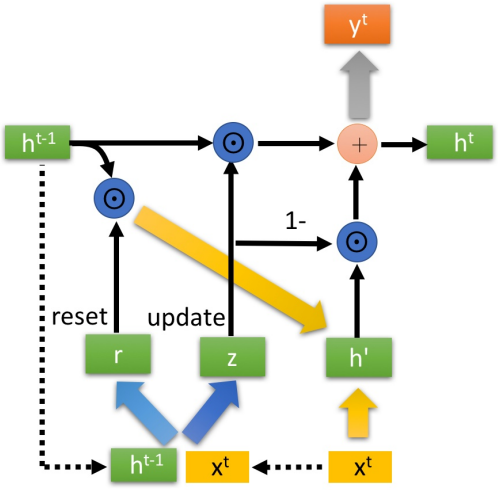

图 9.1: GRU 模型结构图

9.1.2 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9.2。

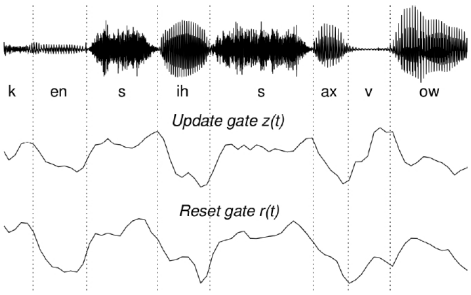

Fig. 1: Average activations of the update and reset gates for a GRU trained on TIMIT in a chunk of the utterance "*sx403*" of speaker "*faks0*".

图 9.2: 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：

$$C(z,r) = \bar{z}_t \star \bar{r}_t \tag{9.5}$$

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9.3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9.3就变成了公式9.6。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。

$$\tilde{h}_t = tanh(W_h x_t + U_h h_{t-1} + b_h) \tag{9.6}$$

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

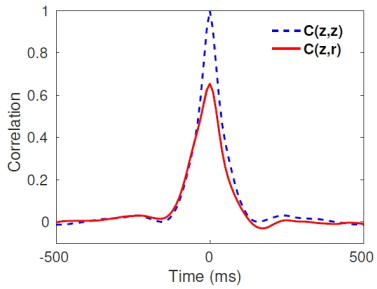

Fig. 2: Auto-correlation $C(z,z)$ and cross-correlation $C(z,r)$ between the average activations of the update (z) and reset (r) gates. Correlations are normalized by the maximum of $C(z,z)$ for graphical convenience.

图 9.3: 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：

$$\tilde{h}_t = ReLU(W_h x_t + U_h h_{t-1} + b_h) \tag{9.7}$$

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：

$$z_t = \sigma(BN(W_z x_t) + U_z h_{t-1}) \tag{9.8}$$

$$\tilde{h}_t = ReLU(BN(W_h x_t) + U_h h_{t-1}) \tag{9.9}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t \tag{9.10}$$

其中 $BN(\cdot)$ 如公式9.11所示，μ_b 和 σ_b 分别为当前 mini-batch 的均值和方差。

$$BN(a) = \gamma \odot \frac{a - \mu_b}{\sqrt{(\sigma_b)^2 + \epsilon}} + \beta \tag{9.11}$$

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。

9.1.3 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9.1

表 9.1: 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}

9.2 Deep Speech2: End-to-End Speech Recognition in English and Mandarin

DS2

9.3 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。

9.3.1 SA-T 的基本结构

9.3.2 path-aware regularization

9.3.3 chunk flow mechanism

参考文献

[1] 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.

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

[3] Andrew Ng. The em algorithm. http://cs229.stanford.edu/notes/cs229-notes8.pdf.

[4] 李航. 统计学习方法. 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.