语音识别学习笔记

学习大法好

作者：杜沪

组织：BIT, SpeechOcean

时间：October 8, 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 工作笔记

1.1 转写数据训练

1.1.1 准备数据

1.1.2 准备词典和词表

1.1.3 开始训练

1.1.4 第一轮训练结果记录与分析

经过第一轮训练之后，对提取的测试集进行解码，为了看不同语言模型下模型的性能，我们采用了四种语言模型去进行解码，这四种 LM 的特点如下：

- 整个数据集包含的文本训练的 LM，记作 all-lm；
- 大语言模型测试的小语言模型，记作 small-lm；
- 大语言模型进行 rescore，记作 big-rescore-lm。

经过这些不同的语言模型解码之后得到的 CER 结果如表1.1。

表 1.1: 不同语言模型解码的测试集 CER

语言模型	大小	CE(%)	INS	DEL	SUB	Total
all-lm	83M	22.97	7420	22157	29619	257710
small-lm	50M	28.45	7376	23291	42650	257710
big-rescore-lm	3.7G	27.58	7169	23825	40092	257710

转写数据实际上由 12 个数据库组成，对每一个数据库单独统计 CER，得到表1.2。

表 1.2: 不同语言模型解码的测试集 CER

Project	ID	WRDS	SUB	INS	DEL	CER
TR2017097	5	15480	2416	361	552	0.215052
TR2018011	11	5511	716	87	161	0.174923
TR2018035	13	39569	7209	1038	1439	0.244788
TR2018046	16	20610	3725	518	12202	0.797914
TR2018063	18	5821	1021	119	225	0.234496
TR2018067	20	6959	1231	419	362	0.289122
TR2018077	22	11382	1078	272	389	0.152785
TR2018086	25	2677	648	91	202	0.351513
TR2018092	26	28489	6890	1127	2285	0.361613
TR2018109	30	37833	6641	744	2447	0.259879
TR2019068	44	17187	3089	344	748	0.243265

1.1.5 无效数据剔除

第 2 章 数学知识总结

2.1 各类矩阵定义

定义 2.1. 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

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & -1 & 1 \end{bmatrix} \tag{2.1}$$

的转置矩阵为

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & -1 \\ 0 & 1 \end{bmatrix} \tag{2.2}$$

♣

定义 2.2. 对称矩阵

设 A 为 n 阶方阵，如果满足 $A^T = A$，即：

$$a_{ij} = a_{ji}(i, j = 1, 2, ..., n) \tag{2.3}$$

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 2.3. 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：

$$A^H = (\bar{A})^T \tag{2.4}$$

则称 A^H 为 A 的复共轭转置矩阵。

♣

定义 2.4. 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 2.5. 正交矩阵

如果 n 阶矩阵 A 满足

$$A^T A = E \tag{2.5}$$

即：

$$A^T = A^{-1} \tag{2.6}$$

则称 A 为正交矩阵，简称正交阵。 ♣

定义 2.6. 酉矩阵

如果 n 阶复矩阵 A 满足

$$A^H A = AA^H = E \tag{2.7}$$

则称 A 为酉矩阵，记作 $A \in U^{n\times n}$。 ♣

定义 2.7. 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 2.8. 正规矩阵

设 $A \in C^{n\times n}$，若：

$$A^H A = AA^H \tag{2.8}$$

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：

$$A^T A = AA^T \tag{2.9}$$

则称 A 为实正规矩阵。 ♣

定义 2.9. 幂等矩阵

设 $A \in C^{n\times n}$，若：

$$A^2 = A \tag{2.10}$$

则称 A 是幂等矩阵。 ♣

定义 2.10. 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 2.11. 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：

$$H = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{2.11}$$

♣

定义 2.12. 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：

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{2.12}$$

根据：

$$\frac{\partial^2 f}{\partial x_1 x_2} = \frac{\partial^2 f}{\partial x_2 x_1} \tag{2.13}$$

可知 Hessian 矩阵为对称阵。 ♣

2.2 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2.14，其中 x^H 为 x 的共轭转置向量。

$$R(M,x)=\frac{x^HMx}{x^Hx} \tag{2.14}$$

若 M 和 x 中元素均为实数，瑞利商可以写成公式2.15。

$$R(M,x)=\frac{x^TMx}{x^Tx} \tag{2.15}$$

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 $\lambda_{\min}=\lambda_1\le\lambda_2\le...\le\lambda_n=\lambda_{\max}$，那么在 M 已知的情况下有：

$$\begin{aligned} cR(M,x)&=\lambda_n\\ \min_x R(M,x)&=\lambda_1 \end{aligned} \tag{2.16}$$

以下为证明公式2.16的过程：

由于 M 是 Hermitian 矩阵，存在一个酉矩阵 U，满足公式2.17。

$$M=UAU^T \tag{2.17}$$

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

因此公式2.15可以转换如下：

$$\begin{aligned} R(M,x)&=\frac{x^TUAU^Tx}{x^Tx}\\ &=\frac{(U^Tx)^TA(U^Tx)}{x^Tx} \end{aligned} \tag{2.18}$$

设 $P=U^Tx$，则：

$$\begin{aligned} R(M,x)&=\frac{P^TAP}{x^Tx}\\ &=\frac{\sum_{i=1}^n\lambda_i|P_i|^2}{\sum_{i=1}^n|x_i|^2} \end{aligned} \tag{2.19}$$

根据特征值的大小关系，我们可以得到不等式2.20。

$$\lambda_1 \sum_{i=1}^{n} |P_i|^2 \leq \sum_{i=1}^{n} \lambda_i |P_i|^2 \leq \lambda_n \sum_{i=1}^{n} |P_i|^2 \tag{2.20}$$

所以公式2.19的范围如下：

$$\lambda_1 \frac{\sum_{i=1}^{n} |P_i|^2}{\sum_{i=1}^{n} |x_i|^2} \leq R(M,x) \leq \lambda_n \frac{\sum_{i=1}^{n} |P_i|^2}{\sum_{i=1}^{n} |x_i|^2} \tag{2.21}$$

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：

$$\begin{aligned} p_i &= \sum_{i=1}^{n} u_{ji} x_j \\ p_i^T &= \sum_{j=1}^{n} x_i u_{ij} \end{aligned} \tag{2.22}$$

则：

$$|p_i|^2 = p_i^T p_i = \sum_{j=1}^{n} \sum_{k=1}^{n} x_j u_{ij} u_{ki} x_k \tag{2.23}$$

于是：

$$\begin{aligned} \sum_{i=1}^{n} |p_i|^2 &= \sum_{i=1}^{n} p_i^T p_i \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} x_j u_{ij} u_{ki} x_k \\ &= \sum_{j=1}^{n} \sum_{k=1}^{n} (\sum_{i=1}^{n} u_{ki} u_{ij}) x_j x_k \end{aligned} \tag{2.24}$$

因为 U 是酉矩阵，满足 $U^T U = I$，所以：

$$I_{jk} = \sum_{i=1}^{n} u_{ji} u_{ik} \tag{2.25}$$

其满足如下等式：

$$I_{jk} = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases} \tag{2.26}$$

结合2.24和2.26可得如下等式：

$$\sum_{i=1}^{n} |p_i|^2 = \sum_{i=1}^{n} |x_i|^2 \tag{2.27}$$

代入公式2.21可得：

$$\lambda_1 \leq R(M, x) \leq \lambda_n \tag{2.28}$$

且：

$$R(M, x) = \begin{cases} \lambda_1 & x = v_1 \\ \lambda_n & x = v_n \end{cases} \tag{2.29}$$

如果用 $x^{'} = cx$ 代入公式2.15有：

$$\begin{aligned} R(M, x^{'}) &= \frac{x^{'T} M x^{'}}{x^{'T} x^{'}} \\ &= \frac{c^2 x^T M x}{c^2 x^T x} \\ &= \frac{x^T M x}{x^T x} \end{aligned} \tag{2.30}$$

由此可以看出来对 x 进行缩放不影响瑞利商的值，即：

$$R(M, cx) = R(M, x) \tag{2.31}$$

因此我们可以限定 $x^T x = 1$，那么公式2.15可以简化为：

$$R(M, x) = x^T M x \tag{2.32}$$

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2.33。

$$L(x, \lambda) = x^T M x - \lambda(x^T x - 1) \tag{2.33}$$

对 x 求导并置为 0 可得：

$$\nabla L(x, \lambda) = Mx - \lambda x = 0 \tag{2.34}$$

即 M 的特征值能使得瑞利商取极值，且有：

$$R(M,x)=\lambda \tag{2.35}$$

瑞利商可以推广至广义瑞利商 (Generalized Rayleigh Quotient)，其形式如公式2.37。

$$R(A,B,x)=\frac{x^HAx}{x^HBx} \tag{2.36}$$

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：

$$\begin{aligned}R(A,B,x)&=\frac{(B^{-\frac{1}{2}}x')^HA(B^{-\frac{1}{2}}x'))}{(B^{-\frac{1}{2}}x')^HB(B^{-\frac{1}{2}}x')}\\&=\frac{x'^H(B^{-\frac{1}{2}})^HAB^{-\frac{1}{2}}x'}{x'^H(B^{-\frac{1}{2}})^HBB^{-\frac{1}{2}}x'}\\&=\frac{x'^H(B^{-\frac{1}{2}})^HAB^{-\frac{1}{2}}x'}{x'^Hx'}\end{aligned} \tag{2.37}$$

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2.37可以写作：

$$\begin{aligned}R(A,B,x)&=\frac{x'^H(B^{-\frac{1}{2}})^HAB^{-\frac{1}{2}}x'}{x'^Hx'}\\&=\frac{x'^HP^HAPx'}{x'^Hx'}\\&=\frac{(Px')^HAPx'}{x'^Hx'}\end{aligned} \tag{2.38}$$

类比上面提到的拉格朗日乘法，我们可以得到如下等式：

$$\nabla L(x,\lambda)=P^HAPx'-\lambda x'=0 \tag{2.39}$$

代入 $x'=P^{-1}x$ 有：

$$\nabla L(x,\lambda)=P^HAPP^{-1}x-\lambda P^{-1}x \tag{2.40}$$

解得：

$$PP^H Ax = \lambda x \tag{2.41}$$

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：

$$B^{-1} Ax = \lambda x \tag{2.42}$$

2.3 EM 算法

本节来自于 Andrew Ng 的课堂讲义[16] 以及李航老师的《统计学习方法》[25]。

2.3.1 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2.1：

定理 2.1. Jsensen's Inequality

假定 f 是一个凸函数，X 是一个随机变量，那么：

$$E[f(X)] \geq f(EX) \tag{2.43}$$

且，如果 f 是严格凸函数，则当且仅当 $X = E[X]$ 时定理2.1取等号。图2.1提供了一个理解 Jensen 不等式的例子。

图中实线部分 f 是一个凸函数，随机变量 X 分别以 0.5 的概率取值 a 和 b，因此随机变量 X 的期望为 a 和 b 的中间值，即：

$$EX = \frac{a+b}{2} \tag{2.44}$$

期望对应的函数值为 $f(EX)$，如图所示。而 $E[f(X)]$ 等于：

$$E[f(X)] = \frac{f(a)+f(b)}{2} \tag{2.45}$$

从图上我们可以很明显的看出来二者的大小关系，又因为 X 是随机变量，当我们取 $X = EX$

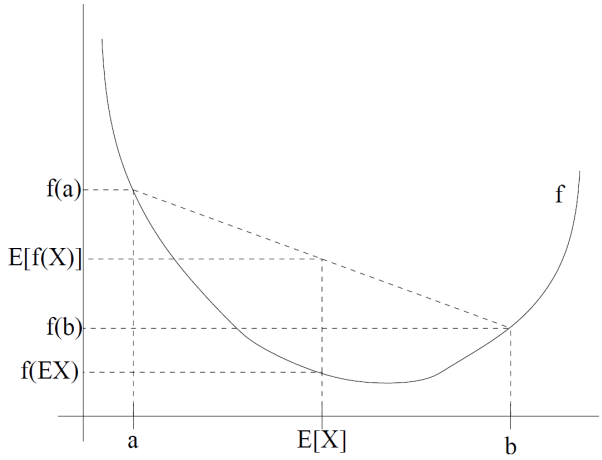

图 2.1: Jensen's Inequality 示例

的时候，函数的期望 $E[f(x)]$ 和期望的函数 $f(EX)$ 是相等的，所以有定理2.1。

2.3.2 EM 算法推导

一般地，用 X 表示观测随机变量的数据，Z 表示隐随机变量的数据，X 和 Z 连在一起称为完全数据。观测数据 X 又称为不完全数据。假设给定观测数据 X，其概率分布是 $P(X|\theta)$，其中 θ 是需要估计的参数，那么不完全数据 X 的似然函数是 $P(X|\theta)$，对数似然函数是 $L(\theta) = \log P(X|\theta)$；假设 X 和 Z 的联合概率分布是 $P(X, Z|\theta)$，那么完全数据的对数似然函数是 $L(\theta) = \log P(X, Z|\theta)$。

假设有 m 个相互独立的训练样本 $\{x^{(1)}, ..., x^{(m)}\}$，我们需要估计这 m 个样本概率分布的参数。若隐变量为 z，想要找到拟合模型 $p(x, z)$ 的参数，其对数似然函数如公式2.46。

2005/06/2

由于隐变量 z 的存在，导致很难使用最大似然估计来得到参数 θ，在这种情况下，EM 算法做的事情是给似然函数找个下界（E 步），然后最大化这个下界（M 步），如此反复迭代多次，最终收敛为止。

对于每一个样本 $x^{(i)}$, 我们定义 Q_i 为隐随机变量 z 的分布，其满足：

$$\sum_z Q_i(z) = 1,\ Q_i(z) \geq 0 \tag{2.47}$$

下面，我们来推导一下 EM 算法的精髓过程，见公式2.48。

$$\begin{aligned}\sum_{i=1}^{m}(x_{(i)};\theta) &= \sum_{i=1}^{m}\log\sum_{z^{(i)}}p(x^{(i)},z;\theta)\\ &= \sum_{i=1}^{m}\log\sum_{z^{(i)}}Q_i(z^{(i)})\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}\\ &\geq \sum_{i=1}^{m}\sum_{z^{(i)}}Q_i(z^{(i)})\log\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}\end{aligned} \tag{2.48}$$

公式2.48的最后一步是通过 Jensen's Inequality 得到的。我们分析下这个推导。

首先我们知道对数函数为凹函数，因为对数函数的二阶导 $f''(x) = -1/x^2 < 0$，其中 $x \in R^+$。而

$$\sum_{z^{(i)}}Q_i(z^{(i)})\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}$$

是

$$\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}$$

关于服从分布 Q_i 的随机变量 z_i 期望。

由 Jensen's Inequality 可以得到公式2.49：

$$f\left(E_{z^{(i)}\sim Q_i}\left[\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}\right]\right) \geq E_{z^{(i)}\sim Q_i}\left[f\left(\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}\right)\right] \tag{2.49}$$

其中下角标 $z^{(i)} \sim Q_i$ 表示这个期望是针对服从分布 Q_i 的随机变量 $z^{(i)}$，根据公式2.49就可以推出2.48的最后一步。

ok，关于公式2.49多说两句，这里面其实用到的公式就是 Jensen 不等式，但是这个式子和前面我们提到的定理2.1中的区别在于 $E[f(X)] \geq f(EX)$ 中的 X 同样是个函数，其等于：

$$\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}$$

以上公式2.48给 $l\theta$ 提供了一个下界，我们不断地最大化这个下界，就可以不断地增加 $l(\theta)$，说明每一次迭代模型都能找到拟合的更好的参数。

好的，下一个问题是，我们如何最大化这个下界呢？假设 t 轮迭代后的参数为 $\theta^{(t)}$，我们希望通过这个 $\theta^{(t)}$ 来得到 $Q_i(z^{(i)})$，即隐变量的分布。而 t 轮迭代的时候一定是满足下界最大化的，

而最大化很明显就是变不等号为等号的临界部分，即定理2.1中等号成立的条件。所以第 t 轮得到的 $\theta^{(t)}$ 必然满足公式2.50。

$$E_{z^{(i)}\sim Q_i}\left[\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}\right]=\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})} \tag{2.50}$$

展开之后有：

$$\begin{aligned}\sum_z Q_i(z^{(i)}\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}&=\sum_z p(x^{(i)},z;\theta)\\&=\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})}\end{aligned} \tag{2.51}$$

所以我们有：

$$\begin{aligned}Q_i(z^{(i)})&=\frac{p(x^{(i)},z;\theta)}{\sum_z p(x^{(i)},z;\theta)}\\&=\frac{p(x^{(i)},z;\theta)}{p(x^{(i)};\theta)}\\&=p(z|x^{(i)};\theta)\end{aligned} \tag{2.52}$$

上面的推导过程其实就是 EM 算法中的 E 步。

总结下 EM 算法步骤：

1. E 步：对于每一个样本，作：

$$Q_i:=p(z|x^{(i)};\theta) \tag{2.53}$$

2. M 步：

$$\theta:=\arg\max_\theta\sum_{i=1}^m\sum_{z^{(i)}}Q_i(z^{(i)})\log\frac{p(x^{(i)},z;\theta)}{Q_i(z^{(i)})} \tag{2.54}$$

在用 EM 算法求模型参数的时候，首先定义模型的初始化参数，得到初始化参数之后，我们就可以得到隐随机变量 z 的分布，得到这个分布之后，我们再通过 M 步去算最大似然概率，得到新的参数，如此反复迭代，最终到 EM 算法收敛为止。

2.3.3 EM 算法收敛性证明

如何证明 EM 算法最终一定会收敛呢？假设 $\theta^{(t)}$ 和 $\theta^{(t+1)}$ 为连续的两次迭代后的参数，我们只需要证明最大似然随着迭代次数是单调递增即可，即证明不等式2.55。

$$\theta^{(t+1)} \geq \theta^{(t)} \tag{2.55}$$

根据 t 轮迭代后的参数 $\theta^{(t)}$，我们可以得到

$$Q_i^{(t)} = p(z|x^{(i)};\theta^{(t)})$$

那么 t 步的似然度为（此时公式2.48的最后一步取等号）：

$$l(\theta^{(t)}) = \sum_{i=1}^{m}\sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z;\theta^{(t)})}{Q_i^{(t)}(z^{(i)})}$$

参数 $\theta^{(t+1)}$ 则是通过最大化上述公式得到的。因此我们有：

$$\begin{aligned} l(\theta^{(t+1)}) &\geq \sum_{i=1}^{m}\sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z;\theta^{(t+1)})}{Q_i^{(t)}(z^{(i)})} && (2.56)\\ &\geq \sum_{i=1}^{m}\sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z;\theta^{(t)})}{Q_i^{(t)}(z^{(i)})} && (2.57)\\ &= l(\theta^{(t)}) && (2.58) \end{aligned}$$

不等式2.56是由公式2.48推导出来的，其对于任意的 Q_i 和 θ 都是成立的，自然对于 $Q_i = Q_i^{(t)}$ 和 $\theta = \theta^{(t)}$ 也成立。而不等式2.57是通过 EM 算法中的 M 步得到的，因为：

$$\theta^{(t+1)} = \arg\max_{\theta} \sum_{i=1}^{m}\sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z;\theta)}{Q_i^{(t)}(z^{(i)})} \tag{2.59}$$

公式2.58即为 Jensen 不等式处于临界条件时的结果。

由此可以得知 $l(\theta)$ 是随着迭代次数的增加单调递增的，即似然度是单调收敛的。因此 EM 算法的终止条件是似然度收敛，即似然度不会再增加了为止。但是实际情况中一般是看两个相邻的迭代过程似然度的差是否低于一个预设的值，如果低于该值，则停止迭代。

定义公式2.60：

$$J(Q,\theta) = \sum_{i=1}^{m}\sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z;\theta)}{Q_i(z^{(i)})} \tag{2.60}$$

我们知道 $l(\theta) \geq J(Q,\theta)$，我们也可以把 EM 算法看成一个坐标上升的过程，在 E 步的时候，相当于固定 θ 求 Q；在 M 步的时候，相当于固定了 Q 求 θ。

题外话：这个固定一个变量，最大化另外一个变量，通过另外一个变量求这个变量这个思想感觉跟 GAN 很类似啊……GAN 里面也是这样的，先固定生成器，优化判别器；然后固定判别器，优化生成器。有时间把这两个联系到一起来想想，还挺有意思的……

2.4 混合高斯分布

定义 2.13. 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：

$$var(X) = \frac{\sum_{i=1}^{N}(X_i - \bar{X})^2}{N-1} \tag{2.61}$$

定义 2.14. 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.

$$cov(X) = \frac{\sum_{i=1}^{N}(X_i - \bar{X})^2(Y_i - \bar{Y})^2}{N-1} \tag{2.62}$$

定义 2.15. 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。

$$\rho_{XY} = \frac{cov(X,Y)}{\sqrt{DX}\sqrt{DY}} \tag{2.63}$$

定义 2.16. 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：

$$\Sigma = \begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) & \cdots & cov(x_1, x_n) \\ cov(x_2, x_1) & cov(x_2, x_2) & \cdots & cov(x_2, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ cov(x_n, x_1) & cov(x_n, x_2) & \cdots & cov(x_n, x_n) \end{bmatrix} \tag{2.64}$$

♣

单变量高斯分布公式如2.65，其中 μ 和 σ^2 分别为均值和方差。

$$\mathcal{N}(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\{-\frac{(x-\mu)^2}{2\sigma^2}\} \tag{2.65}$$

多变量高斯分布公式如2.66，其中 μ 和 Σ 分别为均值和协方差矩阵。

$$\mathcal{N}(x; \mu, \Sigma) = \frac{1}{(2\pi)^{-\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp\{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\} \tag{2.66}$$

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2.67，其中 K 为高斯分量的个数，π_k 为各个分量的权重，其满足 $0 \leq \pi_k \leq 1$，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。

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k) \tag{2.67}$$

那么给定一堆训练数据，我们如何根据这些数据来得到 GMM 的参数呢？

假设训练集 $\{x^{(1)}, ..., x^{(m)}\}$，由于估计 GMM 的算法为无监督学习算法，因此这些数据都是没有标签的。我们用一个联合分布模型来对这些数据进行建模，即：

$$p(x^{(i)}, z^{(i)}) = p(x^{(i)}|z^{(i)}) p(z^{(i)}) \tag{2.68}$$

其中，$z^i \sim Multinomial(\phi)$，$s_j \geq 0$，且 $\sum_{j=1}^{k} \phi_j = 1$，$\phi_j$ 表示的是 $p(z^{(i)} = j)$，此外 $p(x^{(i)}|z^{(i)} = j) \sim \mathcal{N}(\mu_j, \Sigma_j)$。$k$ 表示的是 $z^{(i)}$ 所能取的值的个数。

我们选取的模型假设每个 $x^{(i)}$ 都由随机选取的 $z^{(i)}$ 生成，$z^{(i)} \in \{1, ..., k\}$。我们可以理解成 GMM 模型有 k 个高斯分量，z 就是表征每个分量权重的随机变量，因此 z 就是隐变量。$p(z^{(i)}) = \phi_j$ 就是 $x^{(i)}$ 这个数据来源于分量 j 的概率。

综上，GMM 模型的参数为 μ, Σ, ϕ。现在的问题就是如何求解这些参数。我们可以得到似然

函数如公式2.69。

$$\begin{aligned} l(\mu, \Sigma, \phi) &= \sum_{i=1}^{m} \log p(x^{(i)}; \mu, \Sigma, \phi) \\ &= \sum_{i=1}^{m} \log \sum_{z_{(i)}=1}^{k} p(x^{(i)}|z^{(i)}; \mu, \Sigma) p(z^{(i)}; \phi) \end{aligned} \tag{2.69}$$

如果我们直接对这个似然函数进行求导，由于隐变量的存在，我们无法得到关于参数的确切解。那么既然问题出在隐变量上，如果我们已经知道了每一个 $x^{(i)}$ 所属的分量，公式2.69就可以写成公式2.70，需注意此时隐变量的分布我们还是不知道，我们只知道训练集中的样本分别属于哪一个高斯分量。

$$\begin{aligned} l(\mu, \Sigma, \phi) &= \sum_{i=1}^{m} \log p(x^{(i)}|z^{(i)}; \mu, \Sigma) p(z^{(i)}; \phi) \\ &= \sum_{i=1}^{m} \Big[\log p(x^{(i)}|z^{(i)}; \mu, \Sigma) + \log p(z^{(i)}; \phi) \Big] \end{aligned} \tag{2.70}$$

此时用最大似然概率求解的办法就可以得到这些参数，我们写细一点，求解一下看看。

首先，求解 ϕ_j，在此之前，我们可以看出 l 中与 ϕ 相关的项为：

$$\sum_{i=1}^{m} \log p(z^{(i)}; \phi)$$

我们知道 ϕ 是服从多项式分布的，ϕ 受限于 $\sum_{j=1}^{k} \phi_j = 1$，因此关于 ϕ 的求解我们需要构建一个拉格朗日函数如下：

$$\mathcal{L}(\phi) = \sum_{i=1}^{m} \log p(z^{(i)}; \phi) + \beta(\sum_{j=1}^{k} \phi_j - 1)$$

因此关于求解 ϕ 的步骤如公式2.71：

$$\begin{aligned} \frac{\partial \mathcal{L}(\phi)}{\partial \phi_j} &= \frac{\partial \sum_{i=1}^{m} \log p(z^{(i)}; \phi) + \beta(\sum_{j=1}^{k} \phi_j - 1)}{\partial \phi_j} \\ &= \sum_{i=1}^{m} \frac{\partial \log p(z^{(i)}; \phi)}{\partial \phi_j} \end{aligned} \tag{2.71}$$

因为不一定每个 $x^{(i)}$ 都有 j 分量产生，我们定义 $1\{z^{(i)} = j\}$ 如下，也就是说第 i 个样本由第

j 个样本产生的时候，我们取为 1。

$$1\{z^{(i)}=j\}=\begin{cases}1 & if\ z^{(i)}=j\\ 0 & otherwise\end{cases} \tag{2.72}$$

公式2.71可以写成公式2.73：

$$\begin{aligned}\frac{\partial \mathcal{L}(\phi)}{\partial \phi_j}&=\frac{\partial \sum_{i=1}^{m}\log p(z^{(i)};\phi)+\beta(\sum_{j=1}^{k}\phi_j-1)}{\partial \phi_j}\\&=\sum_{i=1}^{m}1\{z^{(i)}=j\}\frac{\partial \log \phi_j+\beta\phi_j}{\partial \phi_j}\\&=\sum_{i=1}^{m}1\{z^{(i)}=j\}\left[\frac{1}{\phi_j}+\beta\right]\end{aligned} \tag{2.73}$$

令公式2.73为 0，解得：

$$\phi_j=\frac{\sum_{i=1}^{m}1\{z^{(i)}=j\}}{-\beta} \tag{2.74}$$

由此可知：$\phi_j\propto\sum_{i=1}^{m}1\{z^{(i)}=j\}$，又因为 $\sum_{j=1}^{k}\phi_j=1$，将 ϕ_j 代入约束条件有：

$$\sum_{j=1}^{k}\phi_j=\frac{\sum_{j=1}^{k}\sum_{i=1}^{m}1\{z^{(i)}=j\}}{-\beta}=1 \tag{2.75}$$

即

$$-\beta=\sum_{j=1}^{k}\sum_{i=1}^{m}1\{z^{(i)}=j\} \tag{2.76}$$

也就是说 $-\beta$ 等于属于各个高斯分量的训练样本的和，也就是 m。至此，我们可以得到如下公式：

$$\phi_j=\frac{1}{m}\sum_{i=1}^{m}1\{z^{(i)}=j\} \tag{2.77}$$

接着我们来求 μ，如公式2.78。

$$\begin{aligned}
\frac{\partial l(\mu,\Sigma,\phi)}{\partial \mu_j} &= \frac{\partial \sum_{i=1}^{m}\left[\log p(x^{(i)}|z^{(i)};\mu,\Sigma)+\log p(z^{(i)};\phi)\right]}{\partial \mu_j} \\
&= \frac{\partial \sum_{i=1}^{m}\log p(x^{(i)}|z^{(i)};\mu,\Sigma)}{\partial \mu_j} \\
&= \frac{\partial \sum_{i=1}^{m}1\{z^{(i)}=j\}\log p(x^{(i)}|z^{(i)};\mu,\Sigma)}{\partial \mu_j} \\
&= \frac{\partial \sum_{i=1}^{m}1\{z^{(i)}=j\}\log \mathcal{N}(x^{(i)};\mu_j,\Sigma_j)}{\partial \mu_j} \\
&= \frac{\partial \sum_{i=1}^{m}1\{z^{(i)}=j\}\log \frac{1}{(2\pi)^{-\frac{n}{2}}|\Sigma_j|^{\frac{1}{2}}}\exp\{-\frac{1}{2}(x^{(i)}-\mu_j)^T\Sigma_j^{-1}(x^{(i)}-\mu_j)\}}{\partial \mu_j} \\
&= \frac{\partial \sum_{i=1}^{m}1\{z^{(i)}=j\}[-\frac{1}{2}(x^{(i)}-\mu_j)^T\Sigma_j^{-1}(x^{(i)}-\mu_j)]}{\partial \mu_j} \\
&= \frac{\partial \sum_{i=1}^{m}1\{z^{(i)}=j\}[-\frac{1}{2}(x^{(i)T}\Sigma_j^{-1}x^{(i)}-\mu_j^T\Sigma_j^{-1}x^{(i)}-x^{(i)T}\Sigma_j^{-1}\mu_j+\mu_j^T\mu_j)]}{\partial \mu_j} \\
&= \frac{\partial \sum_{i=1}^{m}1\{z^{(i)}=j\}[-\frac{1}{2}(-\mu_j^T\Sigma_j^{-1}x^{(i)}-x^{(i)T}\Sigma_j^{-1}\mu_j+\mu_j^T\mu_j)]}{\partial \mu_j} \\
&= \sum_{i=1}^{m}1\{z^{(i)}=j\}\Sigma_j^{-1}(x^{(i)}-\mu_j)
\end{aligned} \tag{2.78}$$

令公式2.78为零，则有：

$$\sum_{i=1}^{m}1\{z^{(i)}=j\}\Sigma_j^{-1}(x^{(i)}-\mu_j)=0 \tag{2.79}$$

解得：

$$\mu_j=\frac{\sum_{i=1}^{m}1\{z^{(i)}=j\}x^{(i)}}{\sum_{i=1}^{m}1\{z^{(i)}=j\}} \tag{2.80}$$

最后，根据方差的定义，我们可以计算出方差的值，如公式2.81

$$\Sigma_j=\frac{\sum_{i=1}^{m}1\{z^{(i)}=j\}(x^{(i)}-\mu_j)(x^{(i)}-\mu_j)^T}{\sum_{i=1}^{m}1\{z^{(i)}=j\}} \tag{2.81}$$

上面我们算了一下在已知每个样本所属的高斯分量的情况下参数 ϕ,μ,Σ 的估计值。回到最初那个问题，训练数据是没有标签的，那么怎么估计参数？

只要知道了每一个训练样本所属的高斯分量，我们就可以通过最大似然估计求出参数。既

然如此，我们可以先随机的将训练样本进行分配，初始化时，随机指定每一个训练样本所属的高斯分量，结合2.3中讲的 EM 算法，我们可以算出 ϕ, μ, Σ，再用这个参数来更新每一个样本所属的高斯分量，因为在得到参数后，我们将训练样本 $x^{(i)}$ 代入到 GMM 中，可以得到该样本分别属于每一个高斯分量的概率值，即求取 $p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma), j \in \{1, ...K\}$。得到这些概率之后，我们认为每一个高斯分量以权重 $p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma)$ 生成了样本 $x^{(i)}$，知道这些权重之后就可以继续更新参数，这样往往复复，不停的迭代，就会不停的朝着正确分类的方向走去，最终收敛即可。

下面介绍下 GMM 参数估计的 EM 算法：

1. E 步：对于 i, j，作：

$$w_j^{(i)} := p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma) \tag{2.82}$$

2. M 步：

$$\phi_j := \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)} \tag{2.83}$$

$$\mu_j := \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}} \tag{2.84}$$

$$\Sigma := \frac{\sum_{i=1}^{m} w_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} w_j^{(i)}} \tag{2.85}$$

2.5 HMM 相关知识点总结

本节笔记来自于 CS229 课程中 Daniel Ramage 的 Hidden Markov Models Fundamentals[19]。

2.5.1 Markov Models

给定状态集合 $S = \{s_1, s_2, ..., s_{|S|}\}$，我们可以观测到沿着时间线的序列 $\vec{z} \in S^T$。比方说一个天气系统 $S = \{sun, cloud, rain\}$，其 $|S| = 3$。假设连续五天观察到的天气序列为 $\{z_1 = s_{sun}, z_2 = s_{cloud}, z_3 = s_{cloud}, z_4 = s_{rain}, z_5 = s_{cloud}\}$。

上述天气系统中观测到的状态序列是时间线上的随机过程，如果不作进一步假设的话，t 时刻的状态 s_j 可能是任意变量的函数，不仅仅包括从时刻 1 到时刻 $t-1$ 的状态，甚至还包括一些没有建模的变量，所以我们作出两个 **Markov Assumptions** 使模型更可行：

1. **Limited Horizon Assumption**：假设 t 时刻的处于某个状态的概率只取决于 $t-1$ 时刻的状态。本假设的直观感受是：$t-1$ 时刻的状态囊括了对过去时刻状态的总结，因而可以用于

预测 t 时刻的输出。即：

$$P(z_t|z_{t-1}, z_{t-2}, ..., z_1) = P(z_t|z_{t-1}) \tag{2.86}$$

2. **Stationary Process Assumption**：给定当前状态，下一个时刻的状态分布不随着时间变化而变化。即：

$$P(z_t|z_{t-1}) = P(z_2|z_1) \tag{2.87}$$

其中 $z_{t-1} = z_1$。

按照惯例，我们假设存在初始状态和初始观测 $z_0 \equiv s_0$，s_0 表示的是在 $t = 0$ 处的初始概率分布。这么表达初始状态方便我们计算第一个真实状态的先验概率为 $P(z_1|z_0)$，且可以得到 $P(z_t|z_{t-1}, ..., z_1) = P(z_t|z_{t-1}, ..., z_1, z_0)$。此外，也有表达方式将这些先验置信度表示为 $\pi \in R^{|S|}$。

我们定义一个状态转移矩阵 $A \in R^{(|S|+1)\times(|S|+1)}$ 来表示各个状态之间的转移，其中 A_{ij} 表示的是任意时刻从状态 i 转移到状态 j 的概率。对于上述提到的天气系统，其转移矩阵可能如下：

$$A = \begin{array}{c|cccc} & s_0 & s_{sun} & s_{cloud} & s_{rain} \\ s_0 & 0 & 0.33 & 0.33 & 0.33 \\ s_{sun} & 0 & 0.8 & 0.1 & 0.1 \\ s_{cloud} & 0 & 0.2 & 0.6 & 0.2 \\ s_{rain} & 0 & 0.1 & 0.2 & 0.7 \end{array} \tag{2.88}$$

上面这个编出来的矩阵说明了天气是自我关联的，如果今天是晴天，明天还是晴天的概率会比较大；如果今天多云，明天还是多云的概率就比较大。这个规律在很多 Markov Model 里都有，从转移矩阵的对角线上也可以看出来。在这个天气系统中，初始化天气是均匀分布的。

2.5.2 Markov Model 的两个基本问题

将上节中讲到的 **Markov Assumption** 和状态转移矩阵 A 结合起来，我们就可以回答 Markov Chain 中关于状态序列的两个基本问题了。

- 给定一个状态序列 $\vec{z}$，如何求这个序列的概率？
- 如何通过最大化一个观测序列 $\vec{z}$ 的似然概率来估计模型的参数？

2.5.2.1 状态序列的概率

给定一个状态序列 $\vec{z}$，其长度为 t。我们通过概率的链式法则（Chain Rule）来求这个状态序列出现的概率，如公式2.89。

$$\begin{aligned} P(\vec{z}) &= P(z_t, z_{t-1}, ..., z_1; A) \\ &= P(z_t, z_{t-1}, ..., z_1, z_0; A) \\ &= P(z_t|z_{t-1}, ..., z_1, z_0; A)P(z_{t-1}|z_{t-2}, ..., z_1, z_0; A)...P(z_1|z_0; A) \\ &= P(z_t|z_{t-1}; A)P(z_{t-1}|z_{t-2}; A)...P(z_1|z_0; A) \\ &= \prod_{t'=1}^{t} P(z_{t'}|z_{t'-1}; A) \\ &= \prod_{t'=1}^{t} A_{z_{t'-1}z_{t'}} \end{aligned} \tag{2.89}$$

根据公式2.89，我们可以求一下上节中提到的天气序列的概率：

$$\begin{aligned} &P(z_1 = s_{sun}, z_2 = s_{cloud}, z_3 = s_{cloud}, z_4 = s_{rain}, z_5 = s_{cloud}) \\ &= P(s_{sun}|s_0)P(s_{cloud}|s_{sun})P(s_{cloud}|s_{cloud})P(s_{rain}|s_{cloud})P(s_{cloud}|s_{rain}) \\ &= 0.33 \times 0.1 \times 0.2 \times 0.7 \times 0.2 \\ &= 0.000924 \end{aligned} \tag{2.90}$$

2.5.2.2 Markov Model 的最大似然估计

出于学习的目的，我们希望找到参数 A 能够最大化长度为 T 的观测序列 $\vec{z}$ 的最大对数似然。比方说求出天气系统中的从 sunny 转移到 cloudy 或者从 sunny 转移到 sunny 的概率等等，以使得观测序列最有可能出现。定义对数似然函数如公式2.91。

$$\begin{aligned} l(A) &= \log P(\vec{z}; A) \\ &= \log \prod_{t=1}^{T} A_{z_{t-1}z_t} \\ &= \sum_{t=1}^{T} \log A_{z_{t-1}z_t} \end{aligned} \tag{2.91}$$

我们要估计的是 A，所以我们需要将 A 的每一个元素 A_{ij} 都呈现在对数似然函数中，可是就从公式2.91中，我们没法呈现 A_{ij}。当然 $\vec{z}$ 是已知的，即每个时刻的状态是确定的，但是我们没办法直接去指定 $\vec{z}$ 中的每一个元素去估计 A，因为我们希望得到的是一般意义的最大似然函

数，即不论 $\vec{z}$ 中都有哪些状态，公式2.91都可以表示针对样本序列 $\vec{z}$ 的最大似然函数。因此我们可以定义一个 **indicator function** 如公式2.92（这个函数在 GMM 那一讲中也出现过）。

$$1\{z_t = j \wedge z_{t-1} = i\} = \begin{cases} 1 & if\ z_t = j\ and\ z_{t-1} = i \\ 0 & otherwise \end{cases} \tag{2.92}$$

这个函数的意义是当 $t-1$ 时刻状态为 i，并且 t 时刻状态为 j，其取值为 1。这样就和 A_{ij} 对应起来了，而且无需担心观测序列 $\vec{z}$ 中是否有前一时刻为 i、当前时刻为 j 的存在。那么我们可以继续推导公式2.91，即枚举所有可能出现的序列，而对应的状态序列是否存在则由 $\vec{z}$ 来决定，如果某个状态转移存在，那么其状态转移的概率就会被保留下来用于计算最大似然函数，如果不存在，那么其状态转移概率就会被扔掉。

$$\begin{aligned} l(A) &= \sum_{t=1}^{T} \log A_{z_{t-1} z_t} \\ &= \sum_{i=1}^{|S|} \sum_{j=1}^{|S|} \sum_{t=1}^{T} 1\{z_t = j \wedge z_{t-1} = i\} \log A_{ij} \end{aligned} \tag{2.93}$$

由于 $1\{z_t = j \wedge z_{t-1} = i\}$ 的限制，公式2.93一定会迫使着最大似然函数是对应着观测序列 $\vec{z}$ 的。

为了解决这个优化问题，我们需要保证最终求出来的转移矩阵 A 是有效的。那么针对每一个状态，从该状态跳转到下一个时刻状态的概率和应当等于 1，当然，A 中的每一个元素都必须是非负的。因此，我们用拉格朗日算子来解决这个问题，如公式2.94。

$$\begin{aligned} A &= \arg\max_A l(A) \\ s.t &\sum_{j=1}^{|S|} A_{ij} = 1, i \in \{1, 2, ..., |S|\} \\ & A_{ij} \geq 0, i, j \in \{1, 2, ..., |S|\} \end{aligned} \tag{2.94}$$

我们可以构造一个拉格朗日函数来解决这个受约束的最优化问题，如公式2.95。

$$\begin{aligned} \mathcal{L}(A) &= l(A) + \sum_{i=1}^{|S|} \alpha_i (1 - \sum_{j=1}^{|S|} A_{ij}) \\ &= \sum_{i=1}^{|S|} \sum_{j=1}^{|S|} \sum_{t=1}^{T} 1\{z_t = j \wedge z_{t-1} = i\} \log A_{ij} + \sum_{i=1}^{|S|} \alpha_i (1 - \sum_{j=1}^{|S|} A_{ij}) \end{aligned} \tag{2.95}$$

求偏微分并置 0，有：

$$\begin{aligned}\frac{\partial \mathcal{L}(A)}{\partial A_{ij}} &= \frac{\partial \sum_{i=1}^{|S|}\sum_{j=1}^{|S|}\sum_{t=1}^{T} 1\{z_t = j \wedge z_{t-1} = i\} \log A_{ij} + \sum_{i=1}^{|S|} \alpha_i (1 - \sum_{j=1}^{|S|} A_{ij})}{\partial A_{ij}} \\ &= \frac{1}{A_{ij}} \sum_{t=1}^{T} 1\{z_t = j \wedge z_{t-1} = i\} + \alpha_i \\ &= 0\end{aligned} \tag{2.96}$$

解得：

$$A_{ij} = \frac{1}{\alpha_i} \sum_{t=1}^{T} 1\{z_t = j \wedge z_{t-1} = i\} \tag{2.97}$$

又因为：

$$\sum_{j=1}^{|S|} A_{ij} = 1, i \in \{1, 2, ..., |S|\} \tag{2.98}$$

(2.99)

代入公式2.97求得的 A_{ij} 有：

$$\sum_{j=1}^{|S|} \frac{1}{\alpha_i} \sum_{t=1}^{T} 1\{z_t = j \wedge z_{t-1} = i\} = 1 \tag{2.100}$$

解得：

$$\begin{aligned}\alpha_i &= \sum_{j=1}^{|S|}\sum_{t=1}^{T} 1\{z_t = j \wedge z_{t-1} = i\} \\ &= \sum_{t=1}^{T} 1\{z_{t-1} = i\}\end{aligned} \tag{2.101}$$

所以：

$$A_{ij} = \frac{\sum_{t=1}^{T} 1\{z_t = j \wedge z_{t-1} = i\}}{\sum_{t=1}^{T} 1\{z_{t-1} = i\}} \tag{2.102}$$

以上为Markov Model在已知观测序列的条件下估计状态转移矩阵 A 的推导。从公式2.102中，我们可以直观的看到 A_{ij} 就等于观测序列中，状态 i 跳转到状态 j 的次数除以位于状态 i 的个数。

2.5.3 Hidden Markov Model

Markov Model 是处理时序数据的一大利器，但是有些时候我们是无法观测状态序列的，在这种情况下，Markov Model 就无法使用了。在状态序列无法观测的情况下，我们知道这些状态的概率函数，那么如何利用这些状态序列的概率函数来对模型建模呢？比如说语音识别中的，我们能观测到的是根据词发出的声音，但是其对应的词我们是不知道的，如何根据这些声音的声学特征来估计对应的词呢？

为了介绍 HMM，我们借用下 Jason Eisner 论文中提出的“Ice Cream Climatology”[6]，此处附上原文：

> The situation: You are a climatologist in the year 2799, studying the history of global warming. You can't find any records of Baltimore weather, but you do find my (Jason Eisner's) diary, in which I assiduously recorded how much ice cream I ate each day. What can you figure out from this about the weather that summer?

我们可以用 HMM 来解决这个问题。这个问题中模型的状态的序列（每一天的天气）是未知的，我们只能够观测到状态的输出序列（每天吃多少冰淇淋）。

正式地，HMM 是一种 Markov Model，其有一系列的观测输出 $x = \{x_1, x_2, ..., x_T\}, x_t \in V, t = 1, ..., T$，其中 $V = \{v_1, v_2, ..., v_{|V|}\}$ 为观测输出集合。同样有状态序列 $z = z_1, z_2, .., z_T, z_t \in S$，其中 $S = \{s_1, s_2, ..., s_{|S|}\}$ 为状态的集合。在 HMM 中，状态是不可观测的。状态之间的转移概率仍然由状态转移矩阵 A 表示。

同时生成观测输出的概率为隐状态的函数。为了满足这个这个条件，我们假设 HMM 中所有的输出是相互独立的，且定义：

$$P(x_t = v_k | z_t = s_j) = P(x_t = v_k | x_1, ..., x_T, z_1, ..., z_T) \tag{2.103}$$

$$= B_{jk} \tag{2.104}$$

矩阵 B 表示的是在某个时刻处于状态 s_j 生成观测 v_k 的概率值。

回归到上面讲的天气例子，假设有连续四天的冰淇淋消耗量的日志 $\vec{x} = \{x_1 = v_3, x_2 = v_2, x_3 = v_1, x_4 = v_2\}$，对应的 $V = \{v_1 = 1icecream, v_2 = 2icecreams, v_3 = 3icecreams\}$。那么我们要解决 HMM 的哪些问题呢？

2.5.4 HMM 的三个基本问题

HMM 有三个基本问题，

- 某个特定观测序列的概率值为多少（我们有多大的可能性看到连续四天消耗冰淇淋的量为 3,2,1,2）？

- 生成某个特定观测序列的最有可能的状态序列是什么（那四天的冰淇淋消耗量对应的状态序列是多少）?
- 给定一些观测序列，怎么去估计 HMM 的参数 A 和 B?

2.5.4.1 概率计算问题

概率计算问题指的是已知观测序列，求的是 HMM 模型输出该观测序列的概率值，此时，状态转移矩阵 A 和发射矩阵 B 都是已知的。

在 HMM 中，数据生成的过程如下：

假定状态序列 $\vec{z}$ 由一个 Markov Model 生成，其状态转移矩阵为 A。在每一个时间步 t，我们选定输出 x_t，x_t 是关于 z_t 的函数。为了得到观测序列的概率，给定每一个可能的观测序列，我们需要加上数据 $\vec{x}$ 的似然度，见公式2.105

$$\begin{aligned} P(\vec{x}; A, B) &= \sum_{\vec{z}} P(\vec{x}, \vec{z}; A, B) \\ &= \sum_{\vec{z}} P(\vec{x}|\vec{z}; A, B) P(\vec{z}; A; B) \end{aligned} \tag{2.105}$$

公式2.105对于任意概率分布都成立；然而 HMM 的假设简化这个表达式，如公式2.106。

$$\begin{aligned} P(\vec{x}; A, B) &= \sum_{\vec{z}} P(\vec{x}|\vec{z}; A, B) P(\vec{z}; A; B) \\ &= \sum_{\vec{z}} (\prod_{t=1}^{T} P(x_t|z_t; B))(\prod_{t=1}^{T} P(z_t|z_{t-1}; A)) \\ &= \sum_{\vec{z}} (\prod_{t=1}^{T} B_{z_t x_t})(\prod_{t=1}^{T} A_{z_{t-1} z_t}) \end{aligned} \tag{2.106}$$

在参数已知的情况下，公式2.106是一个比较简单的表达式。HMM 有三大假设：

- output independent assumption;
- markov assumption;
- stationary process assumption.

但是在计算公式2.106的过程中，由于求和是针对每一个可能的 $\vec{z}$ 的，而任意时刻的 z_t 可以取 $|S|$ 个值，计算上述的求和需要 $O(|S|^T)$ 次操作。一般我们使用一种动态规划算法：前后向算法来计算 $P(\vec{x}; A, B)$。

前向算法定义 $\alpha_i(t) = P(x_1, x_2, ..., x_t, z_t = s_i; A, B)$，$\alpha_t$ 表示的是直到 t 时刻的观测序列的概率

值，且此时的状态为 s_i。那么观测序列的概率值为：

$$\begin{aligned} P(\vec{x}; A, B) &= P(x_1, x_2, ..., x_T; A, B) \\ &= \sum_{i}^{|S|} P(x_1, x_2, ..., x_T, z_T = s_i; A, B) \\ &= \sum_{i}^{|S|} \alpha_i(T) \end{aligned} \tag{2.107}$$

算法2.1描述了前向算法的迭代过程，其提供了一个效率更高的方式来计算 $\alpha_i(t)$。由算法可知，在每个时间步只需要进行 $O(|S|)$ 次运算，最终计算出 $P(\vec{x}; A, B)$ 需要 $O(|S|\dot{T})$ 次运算。

Algorithm 2.1 HMM 中的前向算法

1. Base Case: $\alpha_i(0) = A_{0i}, i = 1, 2, ..., |S|$;
2. Recursion: $\alpha_j(t) = \sum_{i=1}^{|S|} \alpha_i(t-1)A_{ij}B_{jx_t}, j = 1, ..., |S|, t = 1, ..., T$

类似的算法是后向算法，其定义后向概率 $\beta_i(t) = P(x_T, x_{T-1}, ..., x_{t+1}, z_t = s_i; A, B)$。我们同样可以得到：

$$\begin{aligned} P(\vec{x}; A, B) &= P(x_1, x_2, ..., x_T; A, B) \\ &= \sum_{i}^{|S|} P(x_T, x_{T-1}, ..., x_1, z_0 = s_i; A, B) \\ &= \sum_{i}^{|S|} \beta_i(0) \end{aligned} \tag{2.108}$$

其复杂度与前向算法相同，如算法2.2。

Algorithm 2.2 HMM 中的后向算法

1. Base Case: $\beta_i(T) = 1, i = 1, 2, ..., |S|$;
2. Recursion: $\beta_j(t) = \sum_{i=1}^{|S|} A_{ji}B_{jx_t}\beta_i(t+1), j = 1, ..., |S|, t = 1, ..., T$

2.5.4.2 预测问题

HMM 中最常见的一个问题是：给定观测序列 $\vec{x} \in V^T$，最有可能产生 $\vec{x}$ 的状态序列 $\vec{x}$ 是什么？正式地：

$$\arg\max_{\vec{z}} P(\vec{z}|\vec{x};A,B) = \arg\max_{\vec{z}} \frac{P(\vec{z},\vec{x};A,B)}{P(\vec{x};A,B)} \tag{2.109}$$

$$= \arg\max_{\vec{z}} \frac{P(\vec{x},\vec{z};A,B)}{\sum_{\vec{z}} P(\vec{x},\vec{z};A,B)} \tag{2.110}$$

$$= \arg\max_{\vec{z}} P(\vec{x},\vec{z};A,B) \tag{2.111}$$

公式2.109和公式2.110利用的是贝叶斯定理和全概率公式，由公式2.109可知，其分母与 $\vec{z}$ 无关，因此式2.111成立。我们可以尝试所有可能的 $\vec{z}$，然后得到每一个 $\vec{z}$ 的后验概率，选取其中最大值作为上述公式的解，然而这种解法同样需要 $|S|^T$ 次操作。参考前面提到的前向算法，我们同样可以利用动态规划算法来求解 $\vec{z}$。由于我们需要求解的是每一次迭代的最大值，而不是对每一个时间步求和，因此将 $\sum_{\vec{z}}$ 替换成 max 即可。我们称之为 Viterbi 算法，如算法2.3。得到每一个时间步的最大概率状态序列，一直迭代到 T 为止，我们就得到了最有可能输出给定观测序列的状态序列。

Algorithm 2.3 HMM 中的解码算法

1. Base Case: $\alpha_i(0) = \max(A_{0i}), i = 1, 2, ..., |S|$;
2. Recursion: $\alpha_j(t) = \max(\alpha_i(t-1)A_{ij}B_{jx_t}), for\ i\ in\ \{1, 2, ..., |S|\}, j = 1, ..., |S|, t = 1, ..., T$

2.5.4.3 学习问题

关于 HMM，在给定一系列观测序列的条件下，如何去估计最有可能产生这些观测序列的状态转移矩阵 A 和发射概率矩阵 B？比如说语音识别中，已知语音训练集，我们可以通过这些语音数据去训练 HMM 模型，估计出最适合的参数，从而可以利用这些参数去算未知语音的状态序列的最大似然。换大白话说一下这个例子：我们有一堆语音数据，这些数据的音频特征是已知的，这就是观测值，而对应的建模单元（音素什么的）是关于这些音频特征的函数，也就是状态值，我们希望通过这个训练数据来训练 HMM 模型，得到 HMM 的参数值，知道参数值之后，就可以用于识别其他音频了，这时候就变成了 HMM 的预测问题了。

所以如何通过训练集估计出模型的参数就变成了一个很重要的问题。本小节我们介绍一种 EM 算法的变体用于估计 HMM 的参数。算法2.4。从算法中，我们可以看到 M-Step 有一些约束条件，这些约束条件保证了 HMM 的两个矩阵 A 和 B 的有效性。正如前面我们求解 Markov Model 的参数一样，对于这个这个优化问题需构建一个拉格朗日算子。可是不论是 E-Step 还是 M-Step，都是对 S^T 内所有可能的状态序列 $\vec{z}$ 进行枚举，E、M 两步对于所有可能的标签序列（状

态序列）$\vec{z}$ 都有 $|S|^T$ 的复杂度。为了计算效率，同样使用前面提到的前后向算法来计算 E、M 两步的结果。

Algorithm 2.4 HMM 中的 EM 算法 1

Repeat until convergence { (E-Step) For every possible labeling $\vec{z} \in S^T$, set

$$Q(\vec{z}) := p(\vec{z}|\vec{x}; A, B)$$

(M-Step) Set

$$A, B := \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log \frac{P(\vec{z}, \vec{x}; A, B)}{Q(\vec{z})}$$

$$s.t. \sum_{j=1}^{|S|} A_{ij} = 1, i = 1, 2, ..., |S|; A_{ij} \geq 0, i, j = 1, 2, ..., |S|$$

$$\sum_{k=1}^{|V|} B_{ik} = 1, i = 1, 2, ..., |S|; B_{ik} \geq 0, i = 1, ..., |S|, k = 1, ..., |V|$$

}

首先让我们利用 Markov Assumption 重写估计 HMM 参数的目标函数，目标函数推导第二行是因为对数函数中的分母与所求的值无关，舍去了；第三行利用了 Markov Assumption，而最后一行使用了 **indicator functions**，这么表示之后，我们可以直接利用矩阵的索引来表示 A 和 B。

$$\begin{aligned}
A, B &= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log \frac{P(\vec{z}, \vec{x}; A, B)}{Q(\vec{z})} \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log P(\vec{z}, \vec{x}; A, B) \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log \left[\prod_{t=1}^{T} p(\vec{z}_t|\vec{z}_{t-1}; A) p(\vec{x}_t|\vec{z}_t; B) \right] \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log \left[(p(\vec{z}_t|\vec{z}_{t-1}; A))(\prod_{t=1}^{T} p(\vec{x}_t|\vec{z}_t; B)) \right] \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} \left[\log A_{z_{t-1} z_t} + \log B_{z_t x_t} \right] \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \sum_{i=1}^{|S|} \sum_{j=1}^{|S|} \sum_{k=1}^{|V|} \sum_{t=1}^{T} \left[1\{z_{t-1} = s_i \wedge z_t = s_j\} \log A_{ij} + 1\{z_t = s_j \wedge x_t = v_k\} \log B_{jk} \right]
\end{aligned} \tag{2.112}$$

与前面求解 Markov Model 的最大似然估计一样，根据 HMM 的约束条件，我们构造一个拉

格朗日函数，如公式2.113，将约束条件包含到拉格朗日函数之后，这样我们就不需要担心这些不等式约束了，因为求解出来的值都是非负数。

$$\begin{aligned}\mathcal{L}(A,B,\delta,\epsilon) = &\sum_{\vec{z}} Q(\vec{z}) \sum_{i=1}^{|S|}\sum_{j=1}^{|S|}\sum_{k=1}^{|V|}\sum_{t=1}^{T}\left[1\{z_{t-1}=s_i \wedge z_t = s_j\}\log A_{ij} + 1\{z_t = s_j \wedge x_t = v_k\}\log B_{jk}\right] \\ &+ \sum_{j=1}^{|S|}|S|\epsilon_j(1-\sum_{k=1}^{|V|}B_{jk}) + \sum_{i=1}^{|S|}|S|\delta_i(1-\sum_{j=1}^{|S|}A_{ij})\end{aligned} \tag{2.113}$$

公式2.113分别对 HMM 的参数矩阵中的各个元素求偏导数并置 0。

对 A_{ij} 有：

$$\frac{\partial \mathcal{L}(A,B,\delta,\epsilon)}{\partial A_{ij}} = \sum_{\vec{z}} Q(\vec{z}) \frac{1}{A_{ij}} \sum_{t=1}^{T} 1\{z_{t-1}=s_i \wedge z_t = s_j\} - \delta_i \tag{2.114}$$

置 0 解得：

$$A_{ij} = \frac{1}{\delta_i}\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1}=s_i \wedge z_t = s_j\} \tag{2.115}$$

对 B_{jk} 有：

$$\frac{\partial \mathcal{L}(A,B,\delta,\epsilon)}{\partial B_{jk}} = \sum_{\vec{z}} Q(\vec{z}) \frac{1}{B_{jk}} \sum_{t=1}^{T} 1\{z_t=s_i \wedge x_t = v_k\} - \epsilon_j \tag{2.116}$$

置 0 解得：

$$B_{jk} = \frac{1}{\epsilon_j}\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t=s_j \wedge x_t = v_k\} \tag{2.117}$$

再对拉格朗日算子中的参数求偏导，并代入上述求得的 A_{ij} 和 B_{jk}。

对 δ_i 有

$$\begin{aligned}\frac{\partial \mathcal{L}(A,B,\delta,\epsilon)}{\partial \delta_i} &= 1 - \sum_{j=1}^{|S|} A_{ij} \\ &= 1 - \sum_{j=1}^{|S|}\frac{1}{\delta_i}\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1}=s_i \wedge z_t = s_j\}\end{aligned} \tag{2.118}$$

置 0 解得：

$$\begin{aligned}\delta_i &= \sum_{j=1}^{|S|}\sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_{t-1}=s_i \wedge z_t = s_j\} \\ &= \sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_{t-1}=s_i\}\end{aligned} \tag{2.119}$$

对 ϵ_i 有

$$\begin{aligned}\frac{\partial \mathcal{L}(A,B,\delta,\epsilon)}{\partial \epsilon_i} &= 1-\sum_{k=1}^{|V|} B_{jk} \\ &= 1-\sum_{k=1}^{|V|}\frac{1}{\epsilon_j}\sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_t=s_j \wedge x_t = v_k\}\end{aligned} \tag{2.120}$$

置 0 解得：

$$\begin{aligned}\epsilon_i &= \sum_{k=1}^{|V|}\sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_t=s_j \wedge x_t = v_k\} \\ &= \sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_t=s_j\}\end{aligned} \tag{2.121}$$

再将 δ_i 和 ϵ_j 代回到公式2.115和公式2.117有：

$$\begin{aligned}A_{ij} &= \frac{\sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_{t-1}=s_i \wedge z_t = s_j\}}{\sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_{t-1}=s_i\}} \\ B_{jk} &= \frac{\sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_t=s_j \wedge x_t = v_k\}}{\sum_{\vec{z}} Q(\vec{z})\sum_{t=1}^{T} 1\{z_t=s_j\}}\end{aligned} \tag{2.122}$$

公式2.122中的 A_{ij} 和 B_{jk} 即为每一轮迭代中的 M-Step 所求，但是从公式上来看，都是对所有可能的标签序列 $\vec{z}\in S^T$ 的枚举求和。前面提到的 E-step 中的 $Q_{\vec{z}}$ 的定义是 $P(\vec{z}|\vec{x};A,B)$。我们

考虑下如何用前后向概率 $\alpha_i(t)$ 和 $\beta_j(t)$ 来表征参数 A_{ij} 中的分子：

$$\begin{aligned}
&\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \\
&= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} Q(\vec{z}) \\
&= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} P(\vec{z}|\vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} P(\vec{z}, \vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1)
\end{aligned} \tag{2.123}$$

解释下上面这个牛逼的公式。第二行和第三行是重新整理了下公式并且代入了 $Q(\vec{z})$ 的定义。第四步是根据贝叶斯定理。第五步就是我个人认为非常吊的一步了。我们仔细看下 $1\{z_{t-1} = s_i \wedge z_t = s_j\}$，这个式子的意思是 t 时刻处于状态 s_i，$t+1$ 时刻处于 s_j，好了，其他时刻，从 1 到 $t-1$，从 $t+2$ 到 T，这些个时候处于什么状态，我们不在乎。是啥都可以。那么我们先来考虑下从 1 到 t 这个时间段，再配合上 $\sum \vec{z} z_{t-1} = s_i$、$z_t = s_j$，一个从到 s_i，一个是从 s_j 出发，前面就是终止状态为 s_i 的前向概率，后面就是初始状态为 s_j 的后向概率。

所以这个转换太厉害了。内心鼓掌……

……

同样分母可以转换成：

$$\begin{aligned}
&\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i\} \\
&= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i\} Q(\vec{z}) \\
&= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i\} P(\vec{z}|\vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i\} P(\vec{z}, \vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{t=1}^{T} \alpha_i(t) A_{ij}
\end{aligned} \tag{2.124}$$

需要时刻谨记的是上述分子分母中的 A_{ij} 和 B_{jx_t} 都是上一轮迭代出来的参数，用于更新本

轮参数，为了避免产生混淆，我们将本轮更新后的参数分别记作 $\hat{A}_{ij}$ 和 $\hat{B}_{jx_t}$。因此：

$$\hat{A}_{ij} = \frac{\sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1)}{\sum_{t=1}^{T} \alpha_i(t) A_{ij}} \tag{2.125}$$

另外，此处说明一下，分母的计算与参考资料[19] 中有些不同，其计算过程为：

$$\begin{aligned} &\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i\} \\ &= \sum_{j=1}^{|S|} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} Q(\vec{z}) \\ &= \frac{1}{P(\vec{x}; A, B)} \sum_{j=1}^{|S|} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1) \end{aligned} \tag{2.126}$$

最终求得的：

$$\hat{A}_{ij} = \frac{\sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1)}{\sum_{j=1}^{|S|} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1)} \tag{2.127}$$

同样地，$\hat{B}_{jk}$ 的分子可以转换如下：

$$\begin{aligned} &\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\} \\ &= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_t = s_j \wedge x_t = v_k\} Q(\vec{z}) \\ &= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_t = s_j \wedge x_t = v_k\} P(\vec{z}|\vec{x}; A, B) \\ &= \frac{1}{P(\vec{x}; A, B)} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_t = s_j \wedge x_t = v_k\} P(\vec{z}; \vec{x}; A, B) \\ &= \frac{1}{P(\vec{x}; A, B)} \sum_{i=1}^{|S|} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j \wedge x_t = v_k\} P(\vec{z}; \vec{x}; A, B) \\ &= \frac{1}{P(\vec{x}; A, B)} \sum_{i=1}^{|S|} \sum_{t=1}^{T} 1\{x_t = v_k\} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1) \end{aligned} \tag{2.128}$$

其分母可以转换为：

$$\begin{aligned}
&\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j\} \\
&= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_t = s_j\} Q(\vec{z}) \\
&= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_t = s_j\} P(\vec{z}|\vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{i=1}^{|S|} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} P(\vec{z}; \vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{i=1}^{|S|} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1)
\end{aligned} \tag{2.129}$$

故而：

$$\hat{B}_{jk} = \frac{\sum_{i=1}^{|S|} \sum_{t=1}^{T} 1\{x_t = v_k\} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1)}{\sum_{i=1}^{|S|} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1)} \tag{2.130}$$

算法2.5描述了利用前后向算法来进行参数学习的 Baum-Welch 算法：

Algorithm 2.5 HMM 中的 Baum-Welch 算法

Initialization: Set A and B as random valid probability matrices where $A_{i0} = 0$ and $B_{0k} = 0$, for $i = 1, ..., |S|$ and $k = 1, ..., |V|$.repeat until convergence{(E-Step) Run the forward-backward algorithms to compute α_i and β_j for $i, j = 1, ..., |S|$

$$\gamma_t(i, j) \coloneqq \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1) \tag{2.131}$$

(M-Step) Re-estimate the maximum likelihood parameters as:

$$\begin{aligned}
\hat{A}_{ij} &= \frac{\sum_{t=1}^{T} \gamma_t(i, j)}{\sum_{j=1}^{|S|} \sum_{t=1}^{T} \gamma_t(i, j)} \\
\hat{B}_{jk} &= \frac{\sum_{i=1}^{|S|} \sum_{t=1}^{T} 1 x_t = v_k \gamma_t(i, j)}{\sum_{i=1}^{|S|} \sum_{t=1}^{T} \gamma_t(i, j)}
\end{aligned} \tag{2.132}$$

}

仍然需要对算法2.5进行一些补充说明。与其计算所有可能的 $\vec{z}$ 对应的 $Q(\vec{z})$，我们计算更有效率的 $\gamma_t(i, j) = \alpha_i(t) A_{ij} B_{jx_t} \beta_j(t+1)$。

$\gamma_t(i, j)$ 表示的是所有的观测序列 $\vec{x}$，这些 $\vec{x}$ 对应的 t 和 $t+1$ 时刻状态分别为 s_i 和 s_j。再仔

细观察下 M-Step 的公式，我们可以更直观的感受到 A_{ij} 等于从状态 s_i 跳转到 s_j 的次数除以出现状态 s_i 的次数，B_{jk} 等于状态 s_j 生成观测 v_k 除以状态 s_j 的次数。

正如其他很多 EM 算法的应用场景一样，HMM 的参数学习是一个非凸优化问题，其包含很多个局部最优点。不同的初始化参数会收敛到不同的最大似然，所以最好多跑几次。另外对参数矩阵的概率平滑也是很有必要的，这样可以防止参数矩阵中出现转移概率或者发射概率为 0 的情况。

2.6 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2.2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。

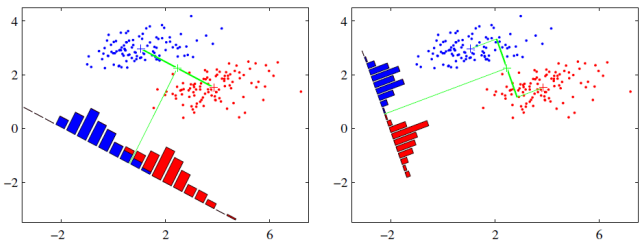

图 2.2: 二类 LDA 转换效果图

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2.133和公式2.134所示。

$$\mu_j = \frac{1}{N_j} \sum_{x \in j} x \tag{2.133}$$

$$\Sigma_j = \sum_{x \in j} (x - \mu_j)(x - \mu_j)^T \tag{2.134}$$

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2.135。

$$\begin{aligned} \Sigma_j' &= \sum_{x \in j} (w^T x - w^T \mu_j)(w^T x - w^T \mu_j)^T \\ &= \sum_{x \in j} w^T (x - \mu_j)(x - \mu_j)^T w \\ &= w^T \Sigma_j w \end{aligned} \tag{2.135}$$

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2.136。

$$\begin{aligned} \arg\max_w J(w) &= \arg\max_w \frac{\| w^T \mu_0 - w^T \mu_1 \|_2^2}{w^T \Sigma_0 w + w^T \Sigma_1 w} \\ &= \arg\max_w \frac{w^T (\mu_0 - \mu_1)(\mu_0 - \mu_1)^T w}{w^T (\Sigma_0 + \Sigma_1) w} \end{aligned} \tag{2.136}$$

类内散度矩阵 S_w 和类间散度矩阵 S_b 分别定义为公式2.137和公式2.138。

$$S_w = \Sigma_0 + \Sigma_1 \tag{2.137}$$

$$S_b = (\mu_0 - \mu_1)(\mu_0 - \mu_1)^T \tag{2.138}$$

所以目标函数就变成了：

$$\arg\max_w J(w) = \arg\max_w \frac{w^T S_b w}{w^T S_w w} \tag{2.139}$$

也就是求解出 w 使得 $J(w)$ 最大。根据2.2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2.140。$J(w)$ 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。

$$S_w^{-1} S_b w = \lambda w \tag{2.140}$$

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2.141。

$$S_W = \sum_{j=1}^{k} \Sigma_j \tag{2.141}$$

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2.3。

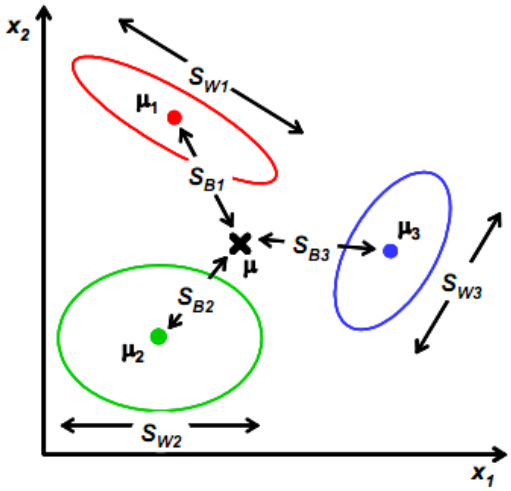

图 2.3: 多类 LDA 的类间散度矩阵示意图

其定义为公式2.142。

$$S_B = \sum_{j=1}^{k} N_j(\mu_j - \mu)(\mu_j - \mu)^T \tag{2.142}$$

其中：

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x_i \tag{2.143}$$

$$\mu_j = \frac{1}{N_j} \sum_{x_j \in C_j} x_j \tag{2.144}$$

同样此时的优化目标为公式2.145。

$$\arg\max_W J(W) = \arg\max_W \frac{W^T S_B W}{W^T S_W W} \tag{2.145}$$

此时目标函数求解转换成了公式2.146：

$$S_W^{-1} S_B W = \lambda W \tag{2.146}$$

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

2.7 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。

2.8 Beta 分布

2.9 MLE 和 MAP

MLE vs MAP

2.10 熵相关

第 3 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

3.1 Background and Fundamentals

3.1.1 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3.1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 3.1: 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。

3.1.2 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。

3.1.3 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3.1。

$$WER = \frac{N_{sub} + N_{ins} + N_{del}}{N_{ref}} \tag{3.1}$$

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。编辑距离的原理见补充知识点中的3.7.3节。

表3.1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 3.1: 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。

3.1.4 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[7]。

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[17]。

3.1.5 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。

3.1.6 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3.2。

$$\hat{W} = \arg\max_{W} P(W|O) \tag{3.2}$$

利用贝叶斯规则，我们得到公式3.3。

$$P(W|O) = \frac{P(W)P(O|W)}{P(O)} \tag{3.3}$$

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3.4。

$$\hat{W} = \arg\max_{W} P(W)P(O|W) \tag{3.4}$$

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。

3.1.7 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.

Solutions:

我们将课程提供的代码补充完整，加上对应的代码注释和结果展示。首先说下测试文本的格式。测试文本有两个

- **hyp.trn**：模型输出结果；
- **ref.trn**：参考文本。

hyp.trn 的内容如下：

```
apple coconut date eggplant elephant fig (0000-000000-0000)
one tiger three flamingo five six (0000-000000-0001)
delaware cat georgia dog mouse massachusetts (0000-00000-0002)
```

ref.trn 的内容如下：

```
apple banana coconut date eggplant fig (0000-000000-0000)
one two three four five six (0000-000000-0001)
delaware pennsylvania new_jersey georgia connecticut massachusetts
```

```
(0000-00000-0002)
```

可以很明显的看到，hyp 和 ref 是一一对应的，格式相同，都是 trn 文件，其内部都是"text (token)" 的形式。

本节的代码逻辑是构建一个函数 **string_edit_distance**，其接受 ref 和 hyp 两个变量，返回的是一个 tuple，依次为 token、edits、deletions、insertions 和 substitutions。该函数存储于 **wer.py** 中，以供代码 **M1_score.py** 调用。

我们先来看 **wer.py**，理解了函数 **string_edit_distance** 自然就会调用了。代码和注释如下：

```
import numpy as np


def string_edit_distance(ref=None, hyp=None):

    if ref is None or hyp is None:
        RuntimeError("ref␣and␣hyp␣are␣required,␣cannot␣be␣None")

    x = ref
    y = hyp
    tokens = len(x)
    if (len(hyp)==0):
        return (tokens, tokens, tokens, 0, 0)

    # p[ix,iy] consumed ix tokens from x, iy tokens from y
    p = np.PINF * np.ones((len(x) + 1, len(y) + 1)) # track total errors
    e = np.zeros((len(x)+1, len(y) + 1, 3), dtype=np.int) # track deletions
        , insertions, substitutions
    p[0] = 0
    for ix in range(len(x) + 1):
        for iy in range(len(y) + 1):
            cst = np.PINF*np.ones([3])
            s = 0
            if ix > 0:
                cst[0] = p[ix - 1, iy] + 1 # deletion cost
            if iy > 0:
                cst[1] = p[ix, iy - 1] + 1 # insertion cost
```

```
            if ix > 0 and iy > 0:
                s = (1 if x[ix - 1] != y[iy -1] else 0)
                cst[2] = p[ix - 1, iy - 1] + s # substitution cost
            if ix > 0 or iy > 0:
                idx = np.argmin(cst) # if tied, one that occurs first wins
                p[ix, iy] = cst[idx]

                if (idx==0): # deletion
                    e[ix, iy, :] = e[ix - 1, iy, :]
                    e[ix, iy, 0] += 1
                elif (idx==1): # insertion
                    e[ix, iy, :] = e[ix, iy - 1, :]
                    e[ix, iy, 1] += 1
                elif (idx==2): # substitution
                    e[ix, iy, :] = e[ix - 1, iy - 1, :]
                    e[ix, iy, 2] += s

    edits = int(p[-1,-1])
    deletions, insertions, substitutions = e[-1, -1, :]
    return (tokens, edits, deletions, insertions, substitutions)
```

3.2 Speech Signal Processing

3.2.1 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3.2（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3.2（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

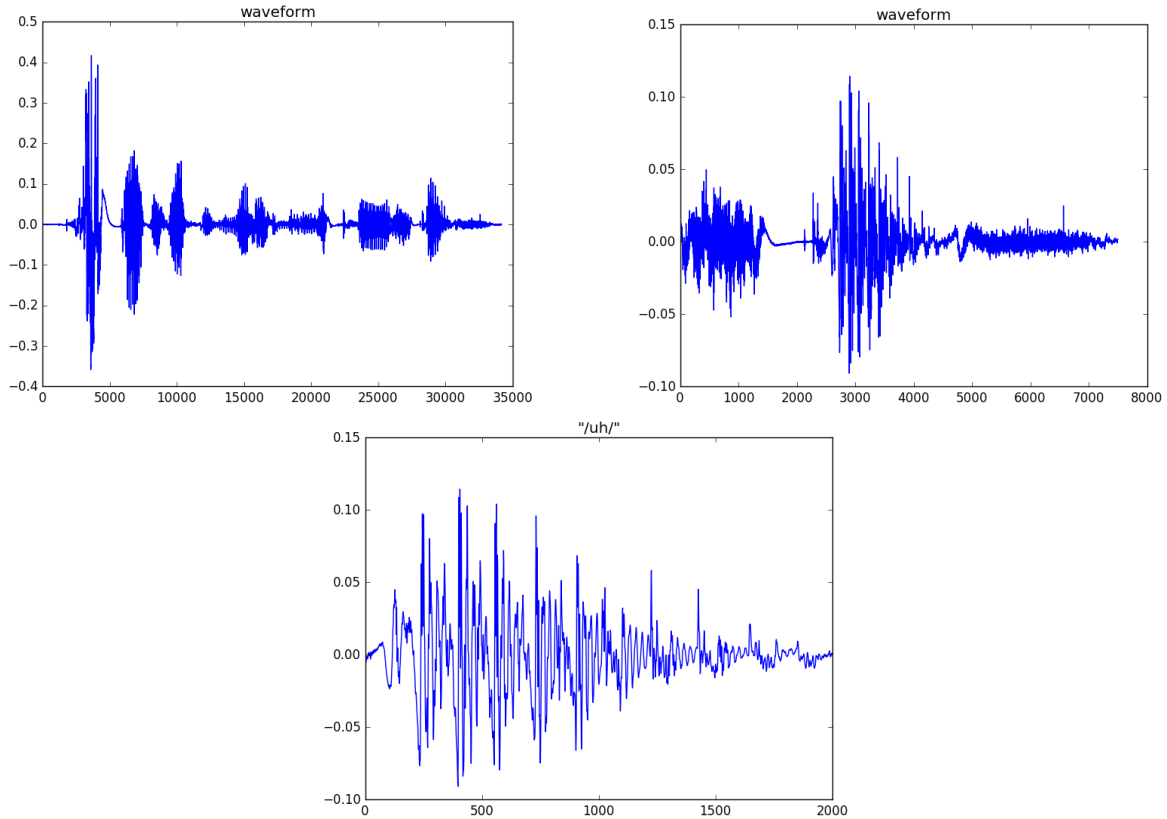

图 3.2: 完整音频波形图与部分波形图

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3.2（下）可以更明显的看到这个元音的周期性。

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3.2（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。

3.2.2 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3.5。

$$x_m[n] = w[n]x[mN + n], n = 0, 1, \ldots, L - 1 \tag{3.5}$$

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3.6。所有现代软件中都可以有效的计算快速傅里叶变换。

$$X_m[k] = \sum_{n=0}^{N-1} x_m[n]e^{-j2\pi knN} \tag{3.6}$$

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3.4（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。

3.2.3 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3.3。每一个滤波器会对不同频率区间的能量谱求平均。

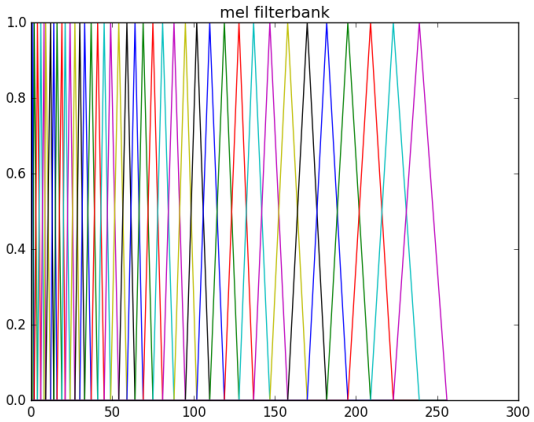

图 3.3: mel filterbank

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3.7。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。

$$X_{\text{mel}}[p] = \sum_{k} M[p,k]\,|X_m[k]|, \quad p = 0, 1, \ldots, P-1 \tag{3.7}$$

3.2.4 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3.4（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：

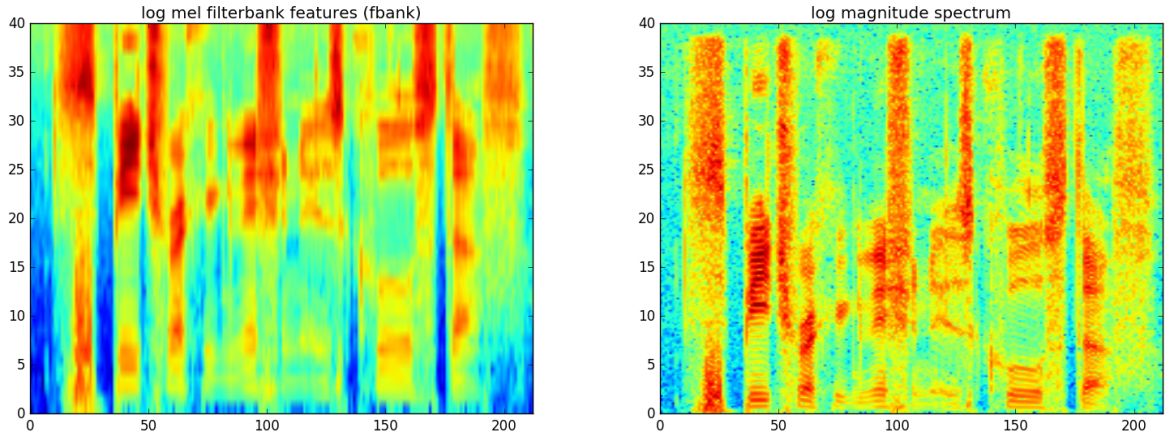

图 3.4: 提取 Fbank 特征（左）与原始信号的频谱图（右）

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3.8，其中 $\alpha = 0.97$。

$$y[n] = x[n] - \alpha x[n-1] \tag{3.8}$$

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3.9。

$$d_t = \frac{\sum_{n=1}^{N} n(c_{t+n} - c_{t-n})}{2\sum_{n=1}^{N} n^2} \tag{3.9}$$

3.2.5 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3.10。

$$X_{t,\text{obs}}[k] = H[k]X_t[k] \tag{3.10}$$

其观测幅度为：

$$\left|X_{t,\text{obs}}[k]\right| = |H[k]|\,|X_t[k]| \tag{3.11}$$

如果我们对公式3.11两边取对数，并计算句子中所有帧的均值，则我们有公式3.12。

$$\begin{aligned}\mu_{\text{obs}} &= \frac{1}{T}\sum_t \log\left(\left|X_{t,\text{obs}}[k]\right|\right)\\ &= \frac{1}{T}\sum_t \log\left(|H[k]|\,|X_t[k]|\right)\\ &= \frac{1}{T}\sum_t \log(|H[k]|) + \frac{1}{T}\sum_t \log\left(|X_t[k]|\right)\end{aligned} \tag{3.12}$$

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3.12可以简化为3.13。

$$\mu_{ttobs} = \log(|H[k]|) \tag{3.13}$$

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3.5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）。

3.2.6 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。

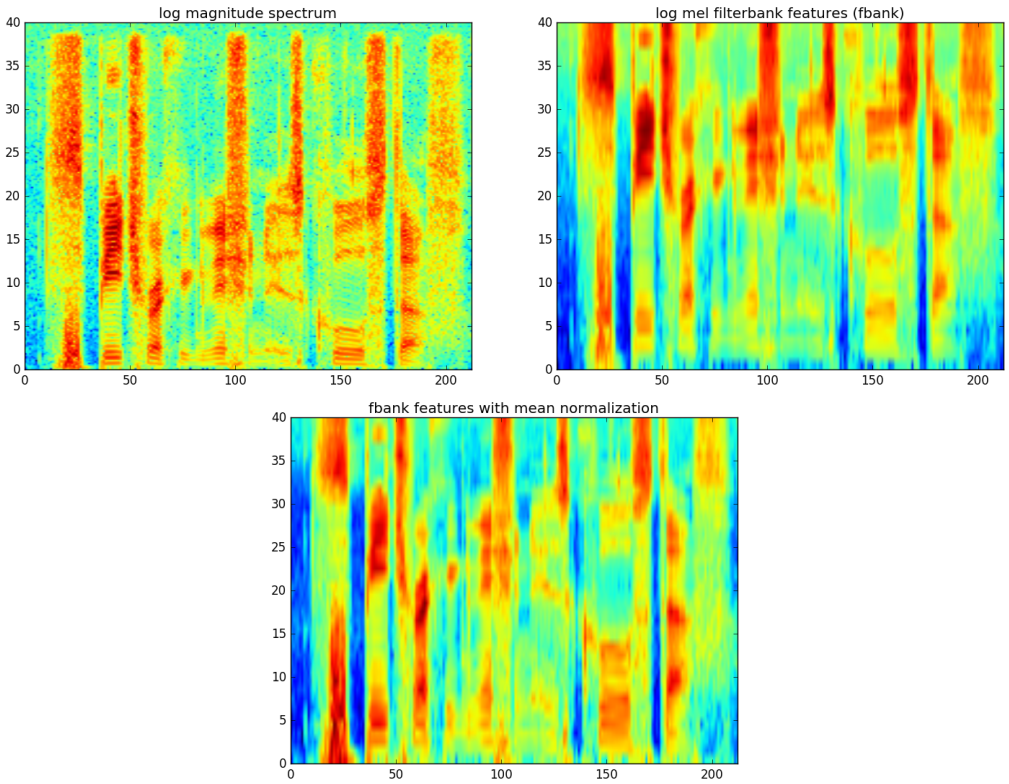

图 3.5: 原始信号的频谱图（左）、提取 Fbank 特征（右）和归一化后的 Fbank 特征（下）

3.2.7 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:

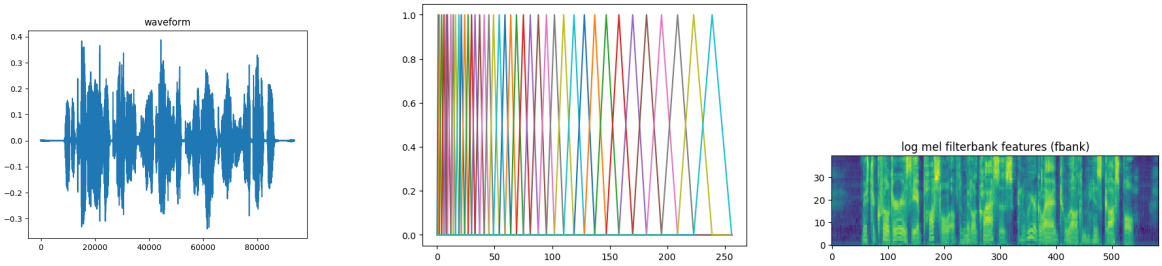

图 3.6: Lab 2 的期望输出

3.3 Acoustic Modeling

3.3.1 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。

3.3.2 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3.14。

$$p(X_1, X_2, X_3, X_4, X_5) = p(X_5|X_4, X_3, X_2, X_1)p(X_4|X_3, X_2, X_1)p(X_3|X_2, X_1)p(X_2|X_1)p(X_1) \tag{3.14}$$

假设天气模型满足一阶马尔科夫假设，即满足公式3.15。

$$p(X_i|X_1, ..., X_{i-1}) = p(X_i|X_{i-1}) \tag{3.15}$$

那么连续五天天气的联合概率分布可以简化为公式3.16。

$$\begin{aligned} p(X_1, X_2, X_3, X_4, X_5) &= p(X_5|X_4)p(X_4|X_3)p(X_3|X_2)p(X_2|X_1)p(X_1) \\ &= p(X_1)\prod_{i=2}^{5} p(x_i|x_{i-1}) \end{aligned} \tag{3.16}$$

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3.7来表示。

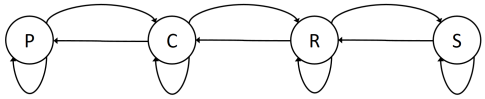

图 3.7: 天气预报模型的马尔科夫链

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3.17。

$$\begin{aligned} p(p) &= \pi_p \\ p(c) &= \pi_c \\ p(s) &= \pi_s \\ p(r) &= \pi_r \end{aligned} \tag{3.17}$$

现在我们回到这个例子，公式3.18说明了如何求 $P(p, p, c, r, s)$。

$$\begin{aligned} p(p, p, c, r, s) &= p(s|p, p, c, r)p(r|p, p, c)p(c|p, p)p(p|p)p(p) \\ &= p(s|r)p(r|c)p(c|p)p(p|p)p(p) \end{aligned} \tag{3.18}$$

上述 Markov Models 也可以称为可观测的马尔科夫模型（observable markov models）。因为当某个时刻所处状态已知，其输出是显性可见的，比如说今天会下雨。而 HMM 中的每个状态的输出并不是确定性事件或者确定性观察，而是事件或者观察的概率分布。这就使得 HMM 是双随机的（double stochastic）：状态之间的转移是随机的，状态的观测值也是随机的。

同样以天气举例子，我们可以将天气系统的 Markov model 转变成 HMM，将原先的状态"**S**unny"、"**P**artly Cloud"、"**C**loudy"、和"**R**aining" 替换成"Hot"、"Chilly"、"Cold" 和"Stormy"，这些状态以不同的概率值对应着不同的天气，如图3.8，当状态为"Hot" 的时候，观测值有 0.9 的概率为"Sunny"，有 0.1 的概率为"Partly Cloudy"。

因此一个包含了 N 个状态的 HMM 模型，其参数为：

- 转移矩阵 A，其元素为 a_{ij}，表征了从一个状态跳转到下一个状态的转移概率；
- 发射矩阵 B，其元素为 $b_i x$，表征了状态 i 的观测值为 x 的概率值，其中 $i = 1, 2, ..., N$；
- 初始化状态的先验概率，即初始状态为某个状态的概率值，$\pi = \{\pi_1, \pi 2, ..., \pi_N\}$。

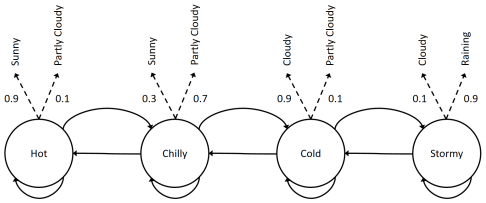

图 3.8: 天气预报模型的隐马尔科夫模型

我们将这些参数记成 $\Phi = \{A, B, \pi\}$。

HMM 有三个基本问题，每一个问题都有着很牛逼的解决办法，此处仅做一个简单介绍，详细的过程参考2.5.3节。

The Evaluation Problem

给定一个模型和观测序列，这些观测序列由这个模型生成的概率有多大？

这个问题可以通过将所有可能的状态序列的概率加起来求得，这些状态序列都可以产生观测序列，通过转移概率和发射概率就可以求得。但是直接这么干是不划算的，因为其时间复杂度为 $O(N^T)$，其中 T 为观测序列的长度，N 为所有可能的观测序列。

前向算法是一种动态规划算法，相比较而言要有效得多。前向算法是在每个时间步都处理下子序列的前向概率。其在每个时间步都会存储 N 个值，时间复杂度为 $O(N^2T)$。

The Decoding Problem

给定一个模型和观测序列，最有可能产生这个观测序列的状态序列是什么？

这个问题可以用 Viterbi 算法解决，Viterbi 算法被广泛的应用于语音识别的解码上，后续我们会介绍其是如何应用于语音识别以及如何将 Viterbi 算法并入训练准则之中。

The Training Problem

给定一个模型和观测序列，如何去估计模型的参数 Φ？

这个问题可以用 Baum-Welch 算法来解决，这个算法中就包含了前后向算法。本处的前向算法计算了 t 时刻处于状态 i 的所有可能的子序列的前向概率，后向算法计算了 t 时刻处于状态 i 为出发点直到 T 的所有可能的子序列的后向概率，此时时间步的初始值为 $t+1$，因此将前后向算法合并起来就是整个 T 时间步，所有在 t 时刻处于状态 i 的路径概率之和。

我们知道了每一个时刻各个状态的后验概率之后，Baum-Welch 算法将这些后验概率作为隐藏状态序列的直接观测值，并且更新模型的参数来优化目标函数。

3.3.3 Hidden Markov Models for Speech Recognition

在语音识别中 HMM 用于对声学观测（声学特征向量）建模，其建模单元为子词，比如说音素。

典型的做法是一个音素对应三个 HMM 的状态，用于对这个音素的开始、中间和结尾建模，每个状态有个自转移概率和转移到下一个状态的概率值，如图3.9。

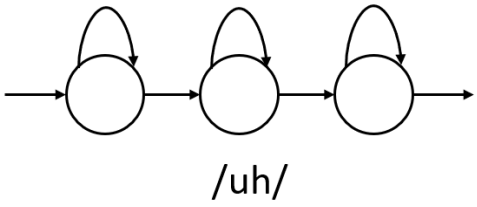

图 3.9: 音素"/uh/" 的 HMM

词的 HMM 可以通过将其连续的音素 HMM 结合起来形成，比如说单词"cup" 的 HMM 模型如图3.10。

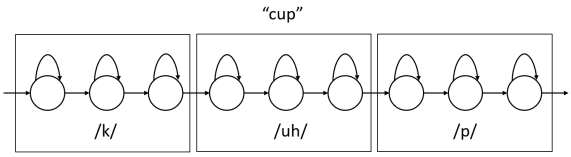

图 3.10: 单词"cup" 的 HMM

因此，**一个高质量的、包含每个单词音素表示的发音词典对于声学模型来说至关重要！**

在 HMM-GMM 模型中，HMM 的状态都有一个概率分布，由 GMM 建模，其定义如公式3.19。

$$p(x|s) = \sum_m w_m N(x; \mu_m, \Sigma_m) \tag{3.19}$$

其中 $N(x; \mu_m, \Sigma_m)$ 是高斯分布，w_m 是混合权重，其满足 $\sum_m w_m = 1$。因此模型每个状态都有自己的 GMM。Baum-Welch 算法会估计出所有的状态转移概率，以及 GMM 中的均值、方差和混合权重。

如今语音识别系统中不再使用 GMM 来对声学观测值建模了，DNN 取代了 GMM，DNN 的输出标签表示的是所有音素的所有状态。比如说有 40 个音素，每个音素是 3-state 的 HMM，那么 DNN 的输出就会有 $40 \times 3 = 120$ 个标签。

这样的声学模型被称作混合模型，即 DNN-HMM，其取代了 GMM 来对声学进行建模。但是 HMM 其他的部分，尤其是 HMM 的拓扑结构和转移概率仍然会被用到。

3.3.3.1 Choice of Subword Units

我们现在已经知道了如何通过发音词典，链式的绑定音素来形成词 HMM。这些音素通常记作"Context Independent" Phones，即 CI phones。而事实上单个音素的发音很大程度上依赖于其前面的音素和后面的音素。比如说对于音素"/ah/"，在"bat" 和"cap" 中的发音就是不一样的。

有鉴于此，如果使用"Context-Dependt"（CD）phones，识别的准确率会有很明显的提升。因此对"bat" 中，我们用"/b-ah+t/" 来表示"/ah/"，这样可以得到对应音素的上下文信息；在"cap" 中，我们用"/k-ah+p/" 来表示"/ah/"。比如说"cup" 在句子"a cup of coffee" 中，即可用图3.11来对单词"cup" 建模。

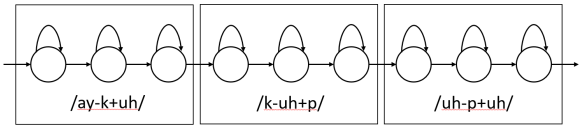

图 3.11: 单词"cup" 的三音素 HMM

因为 CD phones 由三个连续的音素来表示的，记作 **triphone**。有的系统用更长的上下文关系来建模，比如"quinphones" 就是 5 个连续的音素表示的，不过这种的不常见。

当 ASR 使用 CI phones 来作为声学模型的建模单元时，状态数并不多：N 个音素乘以每个音素的状态数 P。美式英语一般有 40 个音素，每个音素三个状态，那么就有 120 个 CI states。那 triphone 作为建模单元的话，会有多少个状态呢？triphone 有 N^3 个，其状态数就有 $40^3 * 3 = 192000$ 个，这会产生两个很严重的问题：

1. 用于训练每一个 triphone 的数据太少；
2. 可能有些训练集中没有的 triphone，在测试集中出现了。

广为流传的一种办法是对多个 CD states 进行池化（pooling），这些 CD states 具备相似的性质，将它们绑定到一起以形成单个"tied" 或者"shared"HMM 状态。这些 tied states 叫做 **senone**。这些被绑定的原始状态都通过计算 senone 的声学分数来得到。

使用决策树（decision tree）将一个集合内的 CD triphones states 聚类到 senones 中。对每一个 CI phone 的每一个状态都会构建一个决策树。

决策树的聚类过程如下：

1. 将某个特定状态中，共有中间音素的所有 triphone 放到一起形成根节点，比如所有满足格式"/*-p+*/" 的三音素的第二个状态；
2. 通过问一系列关于这个三音素的左边 context 或者右边 context 的语言学问题使得决策树生长，这些问题都是二选一的问题。比如说："is the left context phone a back vowel?" 或者"is the right context phone voiced?"。在每一个节点，选择训练数据似然度增加最大的问题；
3. 继续生长，直到节点数达到预设值或者继续分割下去似然度的增加值小于某个阈值；
4. 决策树的叶子就定义这个 CD phone state 的 senones。

通过决策树聚类生成 senones 就可以解决上述两大问题，首先，映射成同一个 senones 的 triphone 之间可以共享数据，所以其数据的估计也会比较稳定；其次如果测试的时候有个 triphone 没有出现过，其对应的 senone 就是顺着决策树搜索，最终找到合适的输出。

几乎所有基于 phone 的语音识别系统都会使用 senone 作为包含上下文信息的建模单元。一

般工业级的语音识别器大概有 10000 个 senones。这虽然比 120 个 CI state 多不少，但是比 192000 少得多了。

关于决策树的生成，参考 RossYoung 的博文[20]，如图3.12。

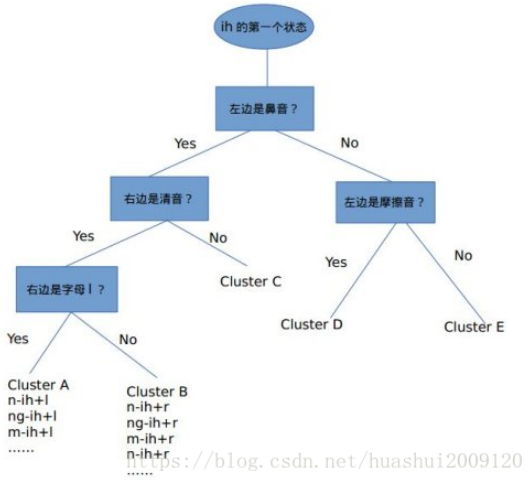

图 3.12: senones 的生成

此外，kaldi 中的决策树规则是由算法生成的，具体看4.5。

3.3.4 Deep Neural Network Acoustic Models

这几年语音识别领域出现的最重要的改进就是使用 DNN 声学模型。前面也提到过，混合 DNN 系统使用单个 DNN 替换了原先的 GMMs（每个 senone 都有一个 GMM），DNN 的输出标签对应着 senones。

训练分类神经网络模型最常用的目标函数是交叉熵（cross entropy），对于一个 M 类的多分类任务，比如说 senones 分类，单个样本的目标函数如公式3.20。

$$E = -\sum_{m=1}^{M} t_m \log y_m \tag{3.20}$$

其中 t_m 是标签值（如果这个样本是第 m 类，t_m 为 1，否则为 0），y_m 是网络的实际输出值，神经网络最后一层的输出经过 softmax 函数映射到概率空间。因此对于每一帧来说，有个 M 维的 one-hot 向量，其对应的 groundtruth，即本帧真真正正对应的 senone。

所以，每一帧我们都得给个标签。

为了给所有训练数据的每一帧加上对应 senones 的标签，我们需要对所有的数据进行强制对齐。强制对齐的操作基本上就是使用 HMM 模型对每一条语音数据进行解码，限制其解码出来的路径会产生正确的参考文本。强制对齐产生单条最有可能的路径，因此就得到了语句中每一帧对应的 senone 的标签。

强制对齐操作需要一个语音识别系统，可以是一个基于 GMM 的初始模型，也可以是一个训练好的 DNN 模型，但是前提是待对齐的和训练好的模型他们的 senones 是一样的。

强制对齐的输出文件包含了每一句的起始帧和终止帧，以及对应的 senone 标签的。不同的工具，这个文件的格式也不一样，以 HTK 为例，输出是一个 MLF 格式的文件，如图3.13，该文件每一列表示的分别是：

1. 开始时间，单位是 100ns；
2. 结束时间，单位是 100ns；
3. Senone 的 ID；
4. Senone 片段的声学模型得分；
5. CD triphone HMM 模型；
6. triphone HMM 模型的声学模型得分；
7. Senone 对应文本中的词

```
#!MLF!#
"test_utterance.lab"
0         17700000 sil[2]      0.031854    sil 719.849243 <s>
17700000  27100000 sil[3]      140.979706
27100000  34800000 sil[4]      152.653412
34800000  35500000 g_s2_6      8.032190    sil-g+uh 27.682287 good
35500000  35800000 g_s3_25     6.538751
35800000  36100000 g_s4_11     13.111345
36100000  36200000 uh_s2_7     3.356279    g-uh+d 11.706375
36200000  36300000 uh_s3_15    4.878324
36300000  36400000 uh_s4_15    3.471773
36400000  36500000 d_s2_51     1.766836    uh-d+m 16.887865
36500000  36800000 d_s3_44     11.397981
36800000  36900000 d_s4_26     3.723048
36900000  37200000 m_s2_7      13.519948   d-m+ao 38.862099 morning
37200000  37500000 m_s3_21     20.174339
37500000  37600000 m_s4_79     5.167810
37600000  37700000 ao_s2_51    3.014942    m-ao+r 11.643064
37700000  37900000 ao_s3_56    5.589801
37900000  38000000 ao_s4_46    3.038320
38000000  38100000 r_s2_178    5.339163    ao-r+n 26.518557
38100000  38300000 r_s3_100    14.487432
38300000  38400000 r_s4_61     6.691961
38400000  38600000 n_s2_28     17.327641   r-n+ih 35.669243
38600000  38700000 n_s3_145    9.536380
38700000  38800000 n_s4_112    8.805222
38800000  38900000 ih_s2_15    5.309596    n-ih+ng 25.129721
38900000  39000000 ih_s3_160   5.600624
39000000  39200000 ih_s4_123   14.219500
39200000  39300000 ng_s2_8     4.993470    ih-ng+m 7.568606
39300000  39400000 ng_s3_14    2.377748
39400000  39500000 ng_s4_4     0.197388
```

图 3.13: MLF 文件示例

有这样的文件或者类似的文件之后，我们就能得到训练 DNN 所需的标签了。

3.3.5 Training Feedforward Deep Neural Networks

构建声学模型的神经网络中，最简单也是最常见的就是前馈神经网络（feed-forward neural network）。前馈神经网络的相关知识网上到处都是，本处咱们只聊基于 DNN 的声学模型的关键信息。

尽管我们训练一个 DNN 是为了预测每一个输入帧的标签，如果我们提供一个 context window 内的帧作为输入的话，而不是使用单独的一帧作为输入，分类的效果会好很多。具体来讲呢，对于 t 时刻的某一帧，DNN 的输入是本帧的前 N 帧、本帧和本帧的后 N 帧，这个 context window 是对称的。因此假设 x_t 是 t 时刻的特征向量，那么对应该时刻 DNN 的输入为：

$$X_t = [x_{t-N}, x_{t-N+1}, ..., x_t, ..., x_{t+N-1}, x_{t+N}] \tag{3.21}$$

一般呢，N 的取值范围在 5 到 11 之间，这个取决于训练集的体量。越大的 context window 提供了越多的信息，但是同时会增加输入层的参数矩阵的大小，如果没有足够的数据的话，训练起来会比较困难。

通常还会做一些数据增强，通过加入这些特征的时间导数，记作 delta 特征。这些特征可以通过简单地差分或者复杂的回归公式计算得到，比如说：

$$\begin{aligned} \Delta x_t &= x_{t+2} - x_{t-2} \\ \Delta^2 x_t &= \Delta x_{t+2} - \Delta x_{t-2} \end{aligned} \tag{3.22}$$

这样的话，DNN 中每一帧的输入就是一个 context window 堆叠（stack）的特征，其包含原始的特征向量，delta features 和 delta-delta features，如下：

$$x_t, \Delta x_t, \Delta^2 x_t \tag{3.23}$$

上述的特征经过一系列全连接层，最终通过 softmax 函数映射成每个 senone 的概率值，以交叉熵为目标函数，通过 BP 算法来进行参数的迭代，直到训练结束。整个过程如图3.14。

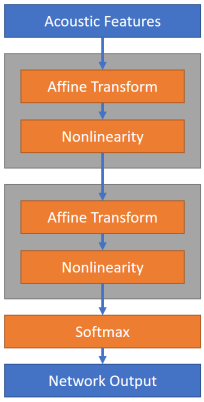

图 3.14: 基于 DNN 的声学模型训练过程

除了基于前馈神经网络的声学模型外，我们经常使用的还有 RNN。与 DNN 不同的是，RNN

处理的是序列数据，其权重具备时域依赖性。RNN 隐含层的输出如公式3.24。

$$h_t^i = f(W^i h_t^{i-1} + U^i h_{t-1}^i + c^i) \tag{3.24}$$

其中 $f(\cdot)$ 是一个非线性函数，比如 sigmoid 函数或者 relu 函数，i 是网络的层数，t 表示的是帧数或者时间索引，输入 h_t^{i-1} 等于第 $i-1$ 层的输出，其实就是上一层的输出，第一层的 RNN 的输入就等于 x_t。

与 DNN 相比，循环层的输出既依赖于当前输入，又依赖于上一个时刻的输出。如果你比较熟悉信号处理中的滤波操作，RNN 层就可以看作是一个非线性无限脉冲响应（nonlinear infinite impulse response,IIR）滤波器。

对于离线操作的应用来说，延迟并不是很重要的一个指标，所以可以利用双向 RNN（bi-RNN）来提升模型的性能。这种情况下，每一层都会有一些参数处理前向序列，同时还会有另一些参数处理后向序列。这样就会有两个输出，将它们拼接（concatenate）到一起作为下一层的输入，如公式3.25，其中下角标 f 和 b 分别指前向和后向。

$$\begin{aligned}
\overrightarrow{h_t^i} &= f(W_f^i h_t^{i-1} + U_f^i h_{t-1}^i + c_f^i) \\
\overleftarrow{h_t^i} &= f(W_b^i h_t^{i-1} + U_b^i h_{t+1}^i + c_b^i) \\
h_t^i &= \left[\overrightarrow{h_t^i}, \overleftarrow{h_t^i}\right]
\end{aligned} \tag{3.25}$$

由于 RNN 可以学习到特征向量序列之间的时域模式，因此很适合用于声学建模。为了训练 RNN，训练序列的序列特性就会保留下来。因此与其用 DNN 中基于单帧随机化的方式来训练模型，不如采用基于语句随机化的方式。这样的情况下，语句被随机化打乱了，但是语句中的序列特性还是会保留下来。

因为 RNN 本身的特性决定了模型会学习到数据沿着时间线上的关联关系，就不需要再用一个很宽的 context window 来捕获每一帧的上下文关系了。对于单向 RNN 来说，提供一些 future context 帧是有积极作用的，但是这个数量比 DNN 要少很多。在双向 RNN 中，提供 context window 就失去了意义，因为处理每一帧，网络都可以看到整句的信息，自然会吸收上下文的关联关系。

训练 RNN 同样使用交叉熵作为目标函数，只是在计算梯度的时候有些微不同。基于这个模型的时域特性，使用 BP 的变体 BPTT 算法来更新网络的参数。我们想想看单向 RNN 的特点，当前时刻的输出不仅依赖于当前输入，还依赖于之前所有时刻的输入。

正如标准的 BP 算法，BPTT 使用梯度下降来优化这个模型，目标函数关于模型参数的梯度是通过链式规则计算得到的，这样导数中就包含了很多梯度的乘积（取决于前面已经处理了多少个时间步）。这些梯度都不存在任何限制，很有可能就会出现梯度接近于 0（网络不再学习了）或者接近于无穷大（网络训练就不稳定，也无法收敛）。这个问题就是梯度消失或爆炸。

为了抑制梯度消失的问题，我们一般有两种办法：

1. 采用 RNN 的变体结构，比如 LSTM，这个我们一会再说；
2. 限制 BPTT 回溯的长度，就是说只允许 BPTT 利用限定长度的历史信息，这样就会限制连乘梯度的数量。

为了抑制梯度爆炸的问题，可以用 gradient clipping（梯度裁剪）的方法。设定一个梯度的上限，对于模型中所有参数的梯度，绝对值超过这个上限的，就让其梯度等于这个上限。

前面提到了 LSTM 可以抑制梯度消失或爆炸的问题以更好的学习训练集中的长时关系。LSTM 中有个概念叫 cell，这个东西就是用来存储状态信息的。cell 中的信息可以随着时间变化一直被保存着，也可以擦除过去的信息，存入当前的信息。这些操作都依赖于 gate 来实现。

Gate 接近于 0，信息就无法通过；Gate 接近于 1，信息就可以通过。输入门决定了是否将当前时间步的信息存储到 cell 中；遗忘门决定了是保存 cell 中信息还是擦掉重写入当前的信息；输出门决定了是否将 cell 中的信息传出去。LSTM 单个神经元的内部如图3.15。更多关于 LSTM 的资料参考[5]。

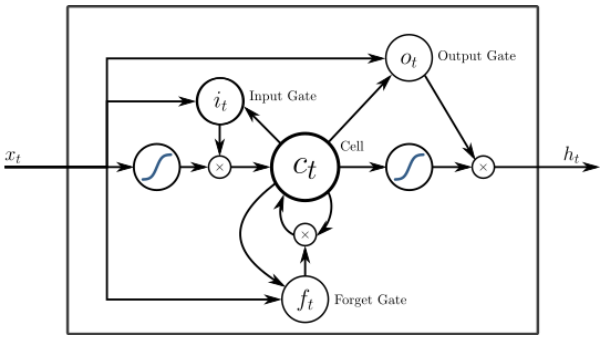

图 3.15: 基于 DNN 的声学模型训练过程

其他的 LSTM 的变体，比如说 GRU，就是 LSTM 的一个简化版，GRU 与 LSTM 的表现相当，但是参数更少，见12.1.1。

3.3.6 Using a Sequence based Objective Function

RNN 是一个序列声学模型，因为它们可以对声学特征向量作为一个时域上连续的序列进行建模。但是其目标函数仍然是帧独立的，就是说在计算目标函数的时候，每一帧的损失是独立计算的。然而因为语音识别是一个序列分类任务，如果我们能搞个基于序列的目标函数，模型的性能应该会有所提升。

GMM-HMM 中就有基于序列的目标函数，如今也可以应用于 DNN-HMM 中。基于帧的交叉熵和序列判别性目标函数的区别就在于基于序列的目标函数更好的对解码过程进行了建模。

具体来说，序列区分性训练中应用了语言模型，来找到那些与参考文本很相似的输出，这些和 groundtruth 很接近的序列正是网络需要区分开的。相对而言，基于帧的交叉熵作为目标函数，所有不正确的分类都会受到惩罚，即便这些不正确的分类在解码的时候根本不会出现，当

然解码的时候需要用的 HMM 拓扑结构或者语言模型。在序列区分性训练中，正确分类的这些 competitors 是由解码训练数据得到的。

最常见的一个序列区分性训练目标函数是最大互信息（maximum mutual information, MMI），如公式3.26。

$$F_{MMI} = \sum_u \frac{\log p(X_u|S_u)p(W_u)}{\sum_{w'} \log p(X_u|S_{W'})p(W')} \tag{3.26}$$

其中 u 是训练集中句子的索引，W_u 是语句 u 的参考词序列，S_u 是对应的状态序列，分母呢是所有可能的词序列的概率和，分母通常由 word lattice 近似得到，解码的时候 word lattice 能得到可能输出的词序列。

使用这个目标函数，让声学模型的训练复杂了很多，但是确实能提高识别性能。同样有很多 MMI 的变体，比如 Minimum Phone Error（MPE），和 state-level 的 Minimum Bayes Risk（SMBRa），见3.7.5。

3.3.7 Decoding with Neural Network Acoustic Models

DNN 声学模型计算的是在已知声学特征的条件下，某个 senone 的条件概率，即 $p(s|x_t)$，这些 state-level 的后验概率必须转换成状态的似然度 $p(x_t|s)$，只有这样我们才能够使用 HMM 来解码。如何得到 $p(x_t|s)$ 呢？

根据贝叶斯规则：

$$p(x_t|s) = \frac{p(s|x_t)p(x_t)}{p(s)} \propto \frac{p(s|x_t)}{p(s)} \tag{3.27}$$

公式3.27之所以成立是因为对于所有的 senones 观测鲜艳 $p(x_t)$ 是个常数，其只是说给似然度增加了个系数而已，因此舍掉它。因此似然度 $p(x_t|s)$ 等于 DNN 的输出除以 senone 的先验 $p(s)$。而 senone 的先验可以通过简单地统计训练集中对应 senone 出现的次数来估计。

上述似然度叫做 **scaled likelihood**，指代的是其是通过 senone 的先验 scale 其后验概率得到的。

3.3.8 Lab 3

Required files:

- M3_Train_AM.py
- M3_Plot_Training.py

Instructions:

In this lab, we will use the features generated in the previous lab along with the phoneme state

alignments provided in the course materials to train two different neural network acoustic models, a DNN and an RNN.

The inputs to the training program are:

- **lists/feat_train.rscp, lists/feat_dev.rscp** - List of training and dev feature files, stored in a format called RSCP. This standard for relative SCP file, where SCP is HTK-shorthand for script file. It is simply a list of files in the two sets. The dev set is used in training to monitor overfitting and perform early stopping. These files should have been generated as part of completing lab 2.
- **am/feat_mean.ascii, am/feat_invstddev.ascii** - The global mean and precision (inverse standard deviation) of the training features, also computed in lab 2
- **am/labels_all.cimlf**- The phoneme-state alignments that have been generated as a result of forced alignment of the data to an initial acoustic model. Generating this file requires the construction of a GMM-HMM acoustic model which is outside the scope of this course, so we are providing it to you. The labels for both the training and dev data are in this file.
- **am/labels.ciphones**- The list of phoneme state symbols which correspond to the output labels of the neural network acoustic model
- **am/ abels_ciprior.ascii** - The prior probabilities of the phoneme state symbols, obtained by counting the occurences of these labels in the training data.

The training, dev, and test RSCP files and the training set global mean and precision files were generated by the lab in Module 2. The remaining files have been provided for you and are in the **am** directory.

PART1: TRAINING A FEEDFORWARD DNN

We have provided a python program called **M3_Train_AM.py** which will train a feed-forward deep network acoustic model using the files described above. The program is currently configured to train a network with the following hyperparameters:

- 4 hidden layers of size 512 hidden units per layer.
- 120 output units corresponding to the phoneme states
- input context window of 23 frames, which means the input to the network for a given frame is the current frame plus 11 frames in the past and 11 frames in the future
- minibatch size of 256
- Learning is performed with Momentum SGD with a learning rate of $1e-04$ per sample with momentum as a time constant of 2500
- One epoch is defined as a complete pass of the training data and training will run for 100 epochs
- The development set will be evaluated every 5 epochs.

This can be executed by running:

```
python M3_Train_AM.py
```

On a GTX 965M GPU running on a laptop, the network trained as a rate of 63,000 samples/sec or about 20 seconds per epoch. Thus 100 epochs will run in 2000 seconds or about 30 minutes.

After 100 epochs, the result of training, obtained from the end of the log file, was

```
Finished Epoch[100 of 100]: [CE_Training] loss = 1.036854 * 1257104,
   metric = 32.74% * 1257104 17.146s (73317.6 samples/s);
Finished Evaluation [20]: Minibatch[1-11573]: metric = 44.26% * 370331;
```

Thus, the training set has a cross entropy of 1.04 per sample, and a 32.74% frame error rate, while the held-out dev set has a frame error rate of 44.3

After training is complete, you can visualize the training progress using **M3_Plot_Training.py**. It takes a CNTK log file as input and will plot epoch vs. cross-entropy of the training set on one figure and epoch vs. frame error rate of the training and development sets on another figure.

```
python M3_Plot_Training.py --log <logfile>
```

For this experiment, **<logfile>** would be **../am/dnn/log**

Here is an example of the figure produced by this script.

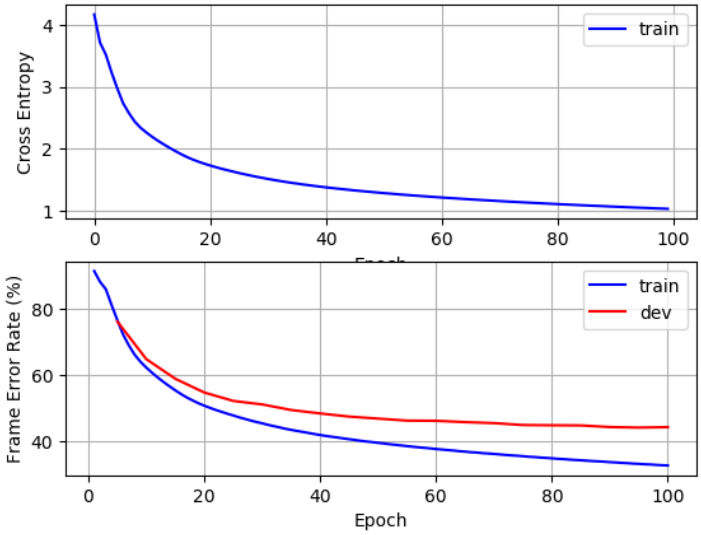

图 3.16: DNN 声学模型训练图

As you can see from the figure, overfitting has not yet occurred as the development set performance is still the best in the final epoch. It is possible that small additional improvements can be obtained with additional training iterations.

You can now experiment with this neural network training script. You can modify the various hyperparameters to see if the performance can be further improved. For example, you can vary the

- Number of layers
- Number of hidden units in each layer
- Learning rate
- Minibatch size
- Number of epochs
- Learning algorithm (see the CNTK documentation for details on using other learners, such as Adam or AdaGrad)

PART 2: TRAINING A RECURRENT NEURAL NETWORK

In the second part of this lab, you will modify the code to train a Bidirectional LSTM (BLSTM) network, a type of recurrent neural network.

To train an BLSTM, there are several changes in the code that you should be aware of.

1. In DNN training, all frames (samples) are processed independently, so the frames in the training data are randomized across all utterances. In RNN training, the network is trying to learn temporal patterns in the speech sequence, so the order of the utterances can be randomized but the utterances themselves must be kept intact. Thus, we set **frame_mode=False** in the **MinibatchSource** instantiated by **create_mb_source()**.
2. Change the network creation to create a BLSTM
   - In **create_network()** , we've created a function called **MyBLSTMLayer** as specified below. This function uses the Optimized_RNN Stack functionality in CNTK. A complete description and additional examples can be found in the CNTK documentation. One thing to be aware of is that with a BLSTM, the size of the hidden layer is actually applied to both directions. Thus, setting the number of hidden units to 512 means that both the forward and backward layers consist of 512 cells. The outputs of the forward and backward layer are then concatenated forming an output of 1024 units. This is then projected back to 512 using the weight matrix W.

```
1 def MyBLSTMLayer(hidden_size=128, num_layers=2):
2   W = C.Parameter((C.InferredDimension, hidden_size), init=C.
        he_normal(1.0), name='rnn_parameters')
3   def _func(operand):
4     return C.optimized_rnnstack(operand, weights=W, hidden_size=
          hidden_size, num_layers=num_layers, bidirectional=True,
          recurrent_op='lstm')
```

```
    return _func
```

- The code calls **MyBLSTMLayer** when the **model_type** is **BLSTM**. We've reduced the number of hidden layers to 2, since the BLSTM layers have more total parameters than the DNN layers.

3. For utterance based processing, entire utterance needs to be processed during training. Thus the minibatch size specifies the total number of frames to process but will pack multiple utterances together if possible. Setting the minibatch size to a larger number will allow for efficient processing with multiple utterances in each minibatch size. We have set the minibatch size to 4096.

The traing the BLSTM model, you can execute the following command.

```
python M3_Train_AM.py --type BLSTM
```

Because of the sequential nature of the BLSTM processing, they are inherently less parallelizable, and thus, train much slower than DNNs. On a GTX 965M GPU running on a laptop, the network trained as a rate of 440 seconds per epoch, or 20 times slower than the DNN. Thus, we will only train for 10 epochs to keep processing time reasonable.

Here too, you can use **M3_Plot_Training.py** to inspect the learning schedule in training. And again, if you are interested, you can vary the hyperparameters to try to find a better solution.

3.4 Language Modeling

3.4.1 Introduction

本模块介绍语言模型的基础知识。语言模型是语音识别的一个组成部分，其估计的是可能的话语的先验概率 $P(W)$。回忆一下前面说过的语音识别的基本公式，这个概率是和声学模型的似然度结合起来以获得最好的输出假设的，如公式3.28。

$$\hat{W} = \arg\max_{W} P(O|W)P(W) \tag{3.28}$$

因此 LM 表征的是识别器所具备的知识，这个知识表达了可能的词序列是什么，即使这个识别器还没有听到任何实际的音频，因为 LM 表达的是词序列的先验知识。LM 应当给可能的句子高的概率值，不太可能的句子低的概率值，而不是硬性的规定语法语义规则来允许一些句子的存在，不允许另一些句子的存在。LM 不应当定下一个绝对的规则是因为没有人知道说话人会说些什么。LM 赋予某个句子的概率值也不是通过语言学或者什么规则得到的，它和声学模型一样，来源于数据。因此我们让实际数据的统计学特性来决定在一种语言中，在一个场景中或者在一个应用中，什么词是比较可能出现的。

注意下关于术语的使用，在语言模型中，我们一般把整条音频对应的句子（sentence）称作词序列（word sequence），这里面没有任何暗示表明这个词序列就是一个服从传统语法的、正确的、完整的句子。事实上一个句子的 LM 针对的是某个场景下说话人可能说出的任何句子。

3.4.2 N gram Models

3.4.2.1 Vocabulary

我们需要给每一个可能的词序列赋予一个概率值：

$$W = w_1 w_2 ... w_n \tag{3.29}$$

其中 n 是词的个数，原则上来说这个数量是没有上限的。

首先，我们将问题简化下，用一个有限集合来限制词的可选择空间，即 LM 的 vocabulary。注意 LM 的 vocabulary 也就是识别器的 vocabulary，我们是没有办法识别一个不被 LM 包括在内的词的，其概率值无限接近于 0 的。

vocabulary 之外的词呢一般称为是 out-of-vocabulary 的词，简称 OOV。在输入数据中如果存在一个 OOV，那么识别器就至少会产生一个错误，所以选择的词典应当最小化 OOV 出现的可能性。常用的策略是选择出现在训练集中的最频繁的那些词，比如说我们选择训练集中前 N 个出现的最频繁的词或者所有出现次数大于 K 的词，这个 N 或者 K 要选择适当。一般呢最优的词典其大小能够在识别速度和准确率之间找到一个平衡，因为词典越大意味着解码的时候效率越低，而减小 OOVs 又能提高准确率。当然添加那种基本上不咋出现的词对准确率的影响可以忽略不计，甚至可能降低准确率，因为解码搜索的时候会产生一些错误也会造成声学的模糊性。

3.4.2.2 Markov factorization and N-grams

即便有了一个有限数量的词典，我们同样有无限个词序列，很显然，列举出所有可能的句子是不现实的。就算说我们概念上可以这么干，这样做也没办法去评估 LM 的参数，因为大部分的句子出现的可能性很小，可能性越小的概率值就得需要更多的数据来评估，这样得到的概率值才是可靠的。

为了解决这两个问题，我们使用一种技巧：使用链式规则将句子的概率值分解成一堆词的概率连乘，如公式3.30，然后应用马尔科夫假设限制状态和参数的数量。

$$P(W) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_1 w_2) \times ... \times P(w_n|w_1 ... w_{n-1}) \tag{3.30}$$

如果我们假设是 LM 是一阶马尔科夫模型，那么我们就可以改写 $P(W)$ 如公式3.31，这就是 bigram 模型，因为模型只利用了相邻的两个词的统计特性，这样每个词靠前面出现的那个词就

可以预测出来。类似的，二阶马尔科夫的 LM 就称为 trigram 模型，当前词是靠前面出现的两个词预测出来的。

$$P(W) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_2) \times ... \times P(w_n|w_{n-1}) \tag{3.31}$$

这种技巧泛化之后我们就称之为 N-gram 模型，即当前词依赖于前面出现的 $N-1$ 个词。事实表明 trigram 比 bigram 效果要好不少，但是随着 N 的增加，性能提升的就越少。因此实际中我们很少使用超过 4-gram 或者 5-gram 以上的 LM。实验室一般用的 trigram 居多，后续讲解我们都以 bigram 来说明，但是要切记，这些 bigram 的特点或者 bigram 的应用都是可以泛化到 N-gram 的。

3.4.2.3 Sentence start and end

为了让 N-gram 给所有可能的有限词序列赋予概率，现在有个小问题：模型怎么知道什么时候这个句子结束了呢？我们当然可以搞一个模型来表示句子的长度 n，但是这样并不方便。我们引入一个特殊的句尾标记 </s> 用来标记这个句子的结束位置。也就是说 LM 根据条件分布从左向右生成词，碰到了 </s> 就停止了。而且，</s> 的出现使得所有句子的概率和等于 1，详情见3.7.7。

类似地，我们引入句首标记 <s>，其位于第一个词 w_1 之前，表示了第一个词的 context，这一点至关重要，因为我们希望 LM 能够学习到第一个词是在句首出现的，比如有些词"I" 或者"well" 是经常出现在句首的，使用句首标记符号，我们就可以用 $P(w_1| < s >)$ 表示第一个词的 bigram 概率值。

bigram 模型的完整公式如3.32。

$$P(W) = P(w_1| < s >) \times P(w_2|w_1) \times P(w_3|w_2) \times ... \times P(w_n|w_{n-1}) \times P(< /s > |w_n) \tag{3.32}$$

3.4.2.4 N-gram probability estimation

N-gram 的条件概率可以通过简单地统计频率得到。令 $c(w_1, ..., w_k)$ 为 k-gram $w_1, .., w_k$ 出现的次数，比如说词"bites" 出现在词"dog" 之后的比例为：

$$P(bites|dog) = \frac{c(dog\ bites)}{c(dog)}$$

那包含相同 context 的 bigrams 概率之和是等于 1 的。假设 context 为 $< con >$，其在语料库中出现的次数为 $c(< con >) = n_0$，而以 $< con >$ 为 context 的 bigrams 共有 m 个，分别记作

$< con_1 >, ..., < con_m >$，其出现的次数分别为 $n_1, ..., n_m$，由此可知：

$$n_0 = \sum_{i=1}^{m} n_i$$

每一个 bigram 的概率值为：

$$p(< con_i > | < con >) = \frac{n_i}{n_0}$$

所以所有 bigram 的概率和为：

$$\begin{aligned} \sum_{i=1}^{m} p(< con_i > | < con >) &= \sum_{i=1}^{m} \frac{n_i}{n_0} \\ &= \frac{\sum_{i=1}^{m} n_i}{\sum_{i=1}^{m} n_i} \\ &= 1 \end{aligned}$$

一般地，k-gram 的概率估计如公式3.33。

$$P(w_k | w_1, ..., w_{k-1}) = \frac{c(w_1, ..., w_k)}{c(w_1, ..., w_{k-1})} \tag{3.33}$$

3.4.2.5 N-gram smoothing and discounting

使用相对频率的方式来估计概率值有个很严重的问题：任何在训练集中没有出现的 N-gram，其概率值都是 0。要知道训练集是有限的，我们不应该仅仅因为有限的语言样本不包含单词组合而排除这些组合，此外，语言并不是一个静态系统，说话人会持续的说出新的表达甚至新的词，要么因为他们有创意，要么语言在交流过程中出错了。

所以我们需要一种原则性的方法将非零概率估计值分配给从来没出现过的 N-grams。这个方法就是 LM smoothing，我们可以把从未出现过的 N-grams 看成是模型中的“洞”，我们现在要把洞填平。关于 LM 的 smoothing 已经是 LM 研究的一个分支了，其中的好多方法可以查阅 ML wiki 中的 Smoothing for Language Models[(14)]，部分 LM smoothing 算法在 SRILM 中都有。此处我们详细讨论一种方法：Witten-Bell smoothing。因为其一这个方法好解释也好实现，其二这个方法鲁棒性很好。

Witten-Bell smoothing 的思想是把从前未见到的词当成个 event，跟那些出现过的词一块计算。训练集中未出现过的词出现了多少次？对于每个独特的词，第一次碰到的时候，这个词就

被视作一个新词。对于 unigram（上下文无关的），平滑后的概率估计值为：

$$\hat{P}(w) = \frac{c(w)}{c(\cdot) + V} \tag{3.34}$$

其中 $c(w)$ 是 unigram 的次数，$c(\cdot)$ 是词数量的总和（即训练集文本的长度），V 是第一次碰见"event" 的次数，即词典的大小。分母多出来的 V 降低了所有 unigram 的概率值，因为之前的 unigram 是通过出现频率计算出来的，因为 unigram 概率值都下降了，所以 LM smoothing 也叫做 discounting，即 N-gram 的概率值低于其出现频率。这样就会匀出来部分概率值分给那些未见过的词。Witten-Bell smoothing 中未出现的词的概率值为：

$$P(unseenword) = \frac{V}{c(\cdot) + V} \tag{3.35}$$

我们可以将 unigram 的情况推广到长度为 k 的 N-gram，前 $k-1$ 个词是第 k 个词的 context，然后找到在这个 context 的条件下出现的独特词类型的数量，如公式**??**。

$$\hat{P}(w_k|w_1...w_{k-1}) = \frac{c(w_1...w_k)}{c(w_1...w_{k-1}) + V(w_1...w_{k-1}\cdot)} \tag{3.36}$$

其中 $V(w_1...w_{k-1}\cdot)$ 指的是在该 context 下观测到的词典的大小。同样释放出来的概率值会赋给之前在这个 context 下没有见过的词。如公式3.37。

$$P(unseenword|w_1...w_{k-1}) = \frac{V(w_1...w_{k-1}\cdot)}{c(w_1...w_{k-1}) + V(w_1...w_{k-1}\cdot)} \tag{3.37}$$

3.4.2.6 Back-off in N-gram models

在给定一个 context 条件下，释放出来的那些概率，也就是 discounted probability 应该如何分配呢？

一种可能的办法是整个词表上均匀分布。比如说 context 是"white dog"，在训练集中，基于这个 context 的 trigram 还有一个，"white dog barked"，这个 trigram 出现了两次，那么根据 Witten-Bell smoothing，我们可以得到：

$$\begin{aligned}\hat{P}(barked|white\ dog) &= \frac{c(white\ dog\ barked)}{c(white\ dog) + V(white\ dog\cdot)} \\ &= \frac{2}{2+1} \\ &= \frac{2}{3}\end{aligned}$$

所以释放出来 1/3 的概率给所有其他以"white dog" 为 context 的 trigram。这 1/3 的概率值如

果均匀分布给这些 unseen words 就忽略了有些词的出现的频率比其他的高的事实。那么我们可以按照这些词的 unigram 来给它们分配 trigram 的概率，但是问题是"white dog the" 分配到的概率值比"white dog barks" 大得多，因为"the" 在语料中出现的次数要比"barks" 多得多。更好的办法是使用 reduced Context，在这个例子中，我们就用"dog" 作为 reduced context 来算 unseen words 在原始 context 下的概率分配。这意味着我们可以统计所有"dog" 出现的情况来猜测接下来的词是什么。这个方法叫做回退（back-off），因为碰到没有在 full context（2-words）下出现过的词，我们会退回到了更短的 context（1-word）下，如公式3.38。

$$\hat{P}_{\text{bo}}(w_k|\, w_1 \ldots w_{k-1}) = \begin{cases} \hat{P}(w_k|\, w_1 \ldots w_{k-1}), & c(w_1 \ldots w_k) > 0 \\ \hat{P}_{\text{bo}}(w_k|\, w_2 \ldots w_{k-1})\, \alpha(w_2 \ldots w_{k-1}), & c(w_1 \ldots w_k) = 0 \end{cases} \tag{3.38}$$

$\hat{P}_{bo}$ 是对于所有 N-grams 的回退估计（back-off estimate），如果 N-gram 在训练语料中存在，也就是 $c(w_1 \ldots w_k) > 0$ 的时候，就直接使用前面提到的 Witten-Bell smoothing 计算的概率值 $\hat{P}$。如果在训练语料中没有出现过的 N-gram，那么就以递归的方式去估计更短的 context 下的回退概率值，再用系数 α 来量化，α 是 context 的函数，也就是词 w_k 在 reduced context $w_2 \ldots w_{k-1}$ 下的概率值，是依据 back-off 分布 $\hat{P}_{bo}(\cdot|w_2 \ldots w_{k-1})$ 除以 w_k 的概率和，所以所有的 w_k 的估计概率总和为 1.

α 被称为 back-off 权重，其不是模型的可变参数，一旦 N-gram 概率 $\hat{P}$ 确定了，α 就确定下来了。因为要这个权重的作用就是让 N-gram 的回退概率和为 1，α 的计算也叫做"re-normalizing the model"。

3.4.3 Language Model Evalution

给定两个语言模型 A 和 B，怎么知道哪个模型更好呢？

直觉上来讲，如果模型 A 赋予测试集（验证集）中存在的词的概率总是比模型 B 高，那么 A 就更好一些。因为模型 A“浪费”在不会出现的那些词上的概率更少。

对于一个测试集 $w_1, ..., w_n$，bigram 模型输出的概率为：

$$P(w_1 \ldots w_n) = P(w_1|\, < s >) \times P(w_2|\, w_1) \times P(w_3|\, w_2) \times \ldots \times P(< /s >\, |\, w_n) \tag{3.39}$$

一个测试集包含多个句子，所以再句子的边界处我们会将 context 重置为 $< s >$。由于连乘的关系，概率值会很小，所以我们换用成对数概率：

$$\log\, P(w_1 \ldots w_n) = \log P(w_1|\, < s >) + \log P(w_2|\, w_1) + \ldots + \log P(< /s >\, |\, w_n) \tag{3.40}$$

模型的好坏决定了测试集对数概率的大小，因此我们也把公式3.40称为对应测试数据的模型对数似然。对数似然只可能是负值，因为概率是小于 1 的，我们取其相反数，再取个平均值

有：

$$-\frac{1}{n}\log P(w_1...w_n) \tag{3.41}$$

我们把这个指标称为熵，其是词流的信息率的一种度量。使用 LM 计算出来的概率，熵可以计算出对词流进行编码所需的平均比特数，越有可能的词所需的比特数越少，所以需要最小化所需的比特率。

另一种评估模型质量的指标是单词概率的平均倒数（the average reciprocal of the word probability），即 perplexity（PPL）。如果词的平均概率为 1/100，那么 PPL 就是 100。作为所讨论的实际模型，PPL 是同等可能性单词的词汇量的大小，这些同等可能性单词对于接下来的内容产生相同的不确定性。

怎么去计算 PPL 呢？

因为模型对于某个数据集的概率是连乘的，所以 PPL 为词概率连乘的几何平均（geometric aevrage），如公式3.42。

$$\sqrt[n]{\frac{1}{P(w_1\dots w_n)}} = P(w_1\dots w_n)^{-\frac{1}{n}} \tag{3.42}$$

PPL 就是个熵的对数反函数（指数）。所以这俩指标是等价的。它们的关系为：

HIGH likelihood ↔ LOW entroy ↔ LOW perplexity ↔ GOOD model

LOW likelihood ↔ HIGH entroy ↔ HIGH perplexity ↔ BAD model

需要注意的是使用测试集来评估模型质量应当与训练集是相互独立的，这样才能得到无偏估计。

3.4.4 Operations on Language Models

3.4.4.1 N gram Pruning

N-gram LM 有效地记录了训练集中 N-grams，因为每个这样的 N-gram 产生的概率估计都是 LM 的参数。这么做有个问题，随着训练数据训练的增加，模型的大小随之线性增加。所以有必要去掉那些冗余的参数，比如说 back-off 机制能在删除高阶 N-gram 的参数之后还能给出基本一致的结果。实际应用中，我们会通过移除不那么重要的参数来将模型缩小到一定的大小，这样就可以节省存储空间。

我们可以原则性的使用熵或者 PPL 的概念来进行 LM 的剪枝。对于模型中的每一个 N-gram 的概率，我们计算熵或者 PPL 的变化，如果变化低于某些阈值，就删除掉，或者剪掉这个参数。剪枝之后，模型需要重新归一化，即重新计算 back-off 权重。

为了让剪枝更贴合实际应用，我们不需要也不想另搞一套测试集来估计这个熵。我们可以使用模型分布所包含的熵，只使用模型包含的信息就使得剪枝标准[22] 变得简单高效。

3.4.4.2 Interporlating Probabilities

假定有两个已经训练好的 LM，其概率估计分别为 $\hat{P}_1$ 和 $\hat{P}_2$。我们怎么样结合这两个模型才能获得一个比较好的 N-gram 概率呢？

如果可以的话，我们可以重新收集这两个模型的训练数据，扔到一起，然后用这个混合数据训练一个新的模型。但是这样干不方便，还会产生一些别的问题。如果其中一个模型的训练数据比另外一个模型的训练数据多得多，那咋整？比如说哈，其中大的模型是用新闻语料训练的，小的模型是从一个新应用上收集的小数据集训练出来的，如果用混合数据训练一个新的模型，那么大量的、不匹配的新闻数据集就会完全淹没了 N-gram 的统计特性，这个模型会和预料的应用产生不匹配。

一个更好的办法是以概率来结合这两个模型，通过插入（interpolating）他们的估计。Interpolating 就是说我们计算下这两个模型估计的一个非均匀平均，即每个模型的估计值都有一个参数，如公式3.43。

$$\hat{P}(w_k|w_1 \dots w_{k-1}) = \lambda \hat{P}_1(w_k|w_1 \dots w_{k-1}) + (1-\lambda)\hat{P}_2(w_k|w_1 \dots w_{k-1}) \tag{3.43}$$

参数 λ 决定了这两个模型的相对影响力，趋近于 1 则第一个模型为主要模型，趋近于 0 第二个模型就有更高的权重。λ 的最优值是使用 held-out 数据计算出的，这些 held-out 数据与这两个模型的训练数据是分开的。λ 就是能让 held-out 数据 PPL 最低的那个值。

Model Interpolation 很容易就可以推广到多个模型的情况：M 个模型的组合就有 M 个权重参数，$\lambda_1, \lambda_2, \dots, \lambda_M$，这些参数满足 $\lambda_1 + \lambda_2, + \dots + \lambda_M = 1$。和为 1 的约束条件是为了让 Interpolation 操作后得到的模型在所有词序列上是归一化的概率值。

3.4.4.3 Merging Models

虽然说不重新训练个模型，model interpolation 也能很高效的降低 PPL，但是还是有个实际问题存在：我们得始终在磁盘或者内存里保留这些模型，在估计测试集的 interpolated 模型的时候，对每一个模型进行估计，最终结合这些模型的概率值。

基于 back-off 的 N-gram LM，我们可以构建一个综合 N-gram 模型，这个模型可以很好地去近似 interpolated 模型。其算法如3.1。

我们在计算 $\hat{P}$ 的时候，其 $\hat{P}_1$ 或者 $\hat{P}_1$ 可以通过 back-off 机制获得。

Algorithm 3.1 Merged N-gram 算法

```
for all for k=1,...,N: do
    for all  for all ngrams w_1 ... w_k do
        Insert w_1 ... w_k into the new model
        Assign probability P̂(w_k|w_1 ... w_{k-1} according to equation 3.43
    end for
    Compute backoff weights for all ngram contexts of length k1 in the new model
end for
```

3.4.5 Advanced LM Topics

本节我们来了解下一些比较新的 LM 建模的算法。LM 建模的方法一直都有在研究，但是根据评估[21]，最先进的 LM 的 PPL 仍然比人类要差两到三倍，所以任重而道远呀！

3.4.5.1 Class-based LM

N-gram 的一个缺点是所有的词都被看作不同的，所以模型需要在训练集中看到足够多次的词才能学到比较好的 N-gram。那比如说"Tuesday" 和"Wednesday" 是语法和意思上都有很多共同点的，那么看到其中一个词 N-gram 就让我们觉得另一个词的 N-gram 也存在（看见"store open Tuesday" 就会让我们联想到"store open Wednesday" 也是可能的 N-gram），所以我们应该利用词的相似性来提高 LM 的泛化能力。

Class-based LM 将一些词组起来形成 word class，然后以这些类的标签而不是词来统计 N-gram 的统计特性。比如说我们定义了一个叫"WEEKDAY" 的类，其成员为"Monday"，"Tuesday"，...，"Friday"，N-gram "store open Tuesday" 就会被当成"store open WEEKDAY" 的一个实例。这么干的话，一个纯字符串的概率就等于两个部分乘积：

（1）包含类标签的字符串的概率（以通常的方式计算，类标签是 N-gram 词汇表的一部分），也就是计算这个实例的 N-gram；

（2）类成员的概率，比如说 $P(\text{Tuesday} \cup \text{WEEKDAY})$。类成员的概率可以通过数据估计得到，相对于所有的 weekday，"Tuesday" 出现了多少次，或者直接设置成一个均匀分布，比如说都等于 15。

有两种比较基本的办法来得到好的词类。

其中一种使用先验知识，尤其是特定应用领域。比如说要搭建一个旅游应用的语言模型，目的地（"Tahiti", "Oslo"）、航线以及周几这些实例都应该被覆盖到，甭管说训练数据是不是有这些实体，哪怕训练集中就没有这些实体。我们可以利用类来定义 N-gram 模型，比如说"AIRPORT"、"AIRLINE" 和"WEEKDAY"，用这个类的 N-gram 来保证这些实例有被覆盖到。这也意味着我们可以通过领域统计特性来设置类成员的的概率，比如说某些旅游景点的受欢迎程度。同时还表示着将词类成员的概率泛化到词组上，比如说"Los Angeles" 之于"AIRPORT"。类 N-gram 的模

型框架可以用来推荐词组[12]。

另外一种方式是用纯数据驱动的方式来定义词类，这里面不掺杂任何的人类或者领域知识。我们可以搜索所有可能的单词/类映射的空间，并选择一个最小化训练数据上所得到的基于类的模型的 PPL（或最大化似然度）。如果想要在合理的时间内搜索到合适的词类，实现细节就很重呀了，详情可以参考一些论文[4]。

3.4.5.2 Neural Network LMs

基于神经网络的机器学习算法已经占领了很多领域，比如说语音识别的声学建模等等。同样人工神经网络在 LM 中也可以用，只要给足够的数据，效果要比 N-gram 好的多！

DNN 之所以能在 LM 界吃香的喝辣的，因为它搞定了 N-gram 模型的两大限制。

第一个限制是跨词的泛化不好，前面提到的构建词类就是为了解决这个问题的，但是词类又引入了一个新的问题：多少个词类合适呢？第一个提出的神经网络 LM[3] 结构叫做前馈 LM，其通过一个 word embedding 层解决了词的泛化问题，word embedding 层的作用是将离散的词标签映射到一个密集向量空间。

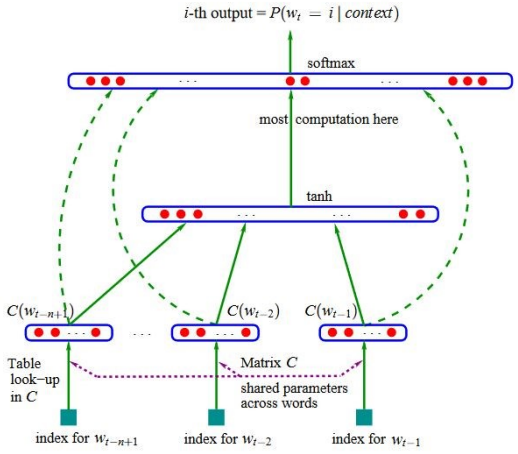

图 3.17: 前馈 LM 结构图

如图3.17所示，网络的输入是 one-hot 向量，输入将形成 N-gram context 的 $N-1$ 个词进行编码，输出是预测随后的词的概率值，输入和输出向量的长度都是词典的大小。所以网络模型就是旧的基于 N-gram 的 LM 的直接替代品。关键点是输入的词通过一个共享的矩阵重新编码成了新的向量，这些向量就不再是 one-hot 形式了，每个输入都会被映射到高维空间。该映射对于所有上下文单词位置是共享的，并且至关重要的是，与下一单词预测器同时进行训练。这种方式的牛逼之处就在于训练好的词嵌入能够用于表征词的相似度，从而来预测词。换句话说，类似地影响下一个单词的上下文单词将被编码为空间中的附近点，然后当遇到新颖组合中的单词时，网络可以利用这种相似性。这是因为所有的网络层都是平滑映射，即相近的输入将会产生

类似的输出，事实证明类似于"Tuesday" 和"Wednesday" 的词他们的嵌入向量有着相似性。

N-gram 的另外一个限制时其截断上下文，它总是总是限制在前面的 $N-1$ 个单词。问题在于，语言允许在相关的单词之间嵌入子句、任意长的形容词列表以及任意距离的其他结构，而这些被分隔开的词在预测下一个词的时候是有用的。任何合理的 N 都没办法在一个 context 中捕捉所有的可预测的单词。这种限制由 RNN 解决，RNN 将某个隐含层在 $t-1$ 时刻的激活值作为额外的输出送给下一个时刻 t，如图3.18。

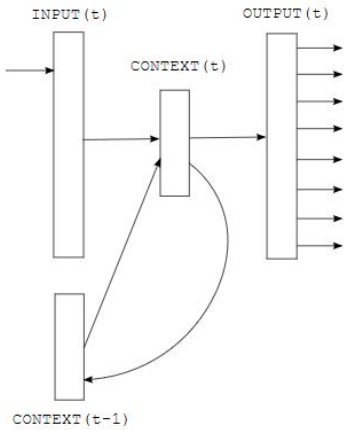

图 3.18: RNNLM 结构图

RNNLM 允许重复性的将某个词位置的信息传递给下一个词，不存在关于词之间间隔距离的硬性限制，所以很远位置的词信息也可以用于预测当前时刻的此预测。当然 RNN 本身结构存在的一些缺陷，梯度消失或者梯度爆炸都可以用门信息流的 RNN 变体来解决，常见的用 LSTM 和 GRU 等。

前馈 LM 和 RNNLM 都随着神经网络的发展而获得更好的效果，比如说更深层的网络和更好的训练方式。基于神经网络的 LM 还有另外一个趋势是对字而不是词单元进行建模。很明显，人工神经网络在高级信息流方面为模型架构进行试验提供了很大的灵活性，我们不必担心编码和概率分布的详细设计，其已经大大推进了该领域的发展，未来必将有更多的发展。

3.4.6 Lab 4: Language Modeling

INTRODUCTION AND SETUP

In this lab, we will practice the main techniques for building N-gram based language models for our speech recognizer. In subsequent modules, the resulting LM will be used in the speech recognition decoder, together with the acoustic model from the preceding lab.

This lab is carried out in a Linux command shell environment. The get started, make sure you know how to invoke the Linux bash shell, either on a native Linux system, using Cygwin on a Windows system, or in the Windows subsystem for Linux. Bash is the default shell on most systems.

Inside bash, change into the M4_Language_Modeling directory:

- cd M4_Language_Modeling

We will be using pre-built executables from the SRI Language Modeling toolkit (SRILM). Start by adding the SRILM binary directories to your search path. If you are in a Cygwin,

- PATH=$PWD/srilm/bin/cygwin64:$PWD/srilm/bin:$PATH

In Linux or Windows Subsystem for Linux, use

- PATH=$PWD/srilm/bin/i686-m64:$PWD/srilm/bin:$PATH

You can put this command in the .bashrc file in your home directory, so it is run automatically next time you invoke the shell. Also, make sure you have the gawk utility installed on your system. As a check that all is set up, run

- ngram-count -write-vocab –
- compute-oov-rate < /dev/null

which should each output a few lines of text without error messages. It will be helpful to also install the optional wget command.

Since language modeling involves a fair amount of text processing it will be useful to have some familiarity with Linux text utilities such as sort, head, wc, sed, gawk or perl, and others, as well as Linux mechanisms for redirecting command standard input/output, and pipelining several commands. We will show you commands that you can copy into the shell to follow along, using this symbol

- command argument1 argument2 ⋯

but we encourage you try your own solutions to achieve the stated goals of each exercise, and to explore variants.

PREPARING THE DATA

We will be using the transcripts of the acoustic development and test data as our dev and test sets for language modeling.

TASK: Locate files in the 'data' subdirectory and count the number of lines and words in them.

SOLUTION:

- ls data
- wc -wl data/dev.txt data/test.txt

TASK: View the contents of these files, using your favorite pager, editor, or other tool. What do you notice about the format of these files? How do they differ from text you are used to?

SOLUTION:

- head data/*.txt

You will notice that the data is in all-lowercase, without any punctuation. This is because we will model sequences of words only, devoid of textual layout, similar to how one your read or speak them.

The spelling has to match the way words are represented in the acoustic model. The process of mapping text to the standard form adopted for modeling purposes is called text normalization (or TN for short), and typically involves stripping punctuation, mapping case, fixing typos, and standardizing spellings of words (like MR. versus MISTER). This step can consume considerable time and often relies on powerful text processing tools like sed or perl.

Because it is so dependent on the source of the data, domain conventions, and tool knowledge, we will not elaborate on it here. Instead, we will download an LM training corpus that has already been normalized,

- wget http://www.openslr.org/resources/11/librispeech-lm-norm.txt.gz

If your system doesn't have the wget command you can download this file in a browser and move it into the LM lab directory.

TASK: Inspect the file and count lines and word tokens. How does the text normalization of this file differ from our test data?

SOLUTION: The file is compressed in the gzip (.gz) so we must use the gunzip tool

- gunzip -c librispeech-lm-norm.txt.gz | head
- gunzip -c librispeech-lm-norm.txt.gz | wc -wl

The second command can take a while as the file is large. You will notice that this file is text normalized but uses all-uppercase instead of all-lowercase.

Language model training data, and language models themselves, are often quite large but compress well since they contain text. Therefore, we like to keep them in compressed form. The SRILM tools know how to read/write .gz files, and it is easy to combine gzip/gunzip with Linux text processing tools.

OPTIONAL TASK: Download the raw training data at http://www.openslr.org/12/librispeech-lm-corpus.tgz, and compare it to the normalized text. How would you perform TN for this data?

SOLUTION: left to the readers!

DEFINING A VOCABULARY

The first step in building a LM is to define the set of words that it should model. We want to cover the largest possible share of the word tokens with the smallest set of words, so as to keep model size to a minimum. That suggests picking the words that are most frequent based on the training data.

One of the functions of the ngram-count tool is to count word and ngram occurrences in a text file.

- ngram-count -text TEXT -order 1 -write COUNTS -tolower

Will count 1-grams (i.e., words) and write the counts to a file. The final option above maps all text to lowercase, thus dealing with the mismatch we have between our training and test data.

TASK: Extract the list of the 10,000 most frequent word types in the training data. What kinds of

words do you expect to be at the top of the list? Check your intuition.

HINT: Check out the Linux sort, head, and cut commands.

SOLUTION:

- ngram-count -text librispeech-lm-norm.txt.gz -order 1 -write librispeech.1grams -tolower
- sort -k 2,2 -n -r librispeech.1grams | head -10000 > librispeech.top10k.1grams
- cut -f 1 librispeech.top10k.1grams | sort > librispeech.top10k.vocab

The intermediate file librispeech.top10k.1grams contains the words and their counts sorted most frequent first. As you might expect, common function words liked "the", "and", "of" appear at the top of the list. Near the top we also find two special tags, $< s >$ and $< /s >$. These are added by ngram-count to mark the start and end, respectively, of each sentence. Their count equals the number of non-empty lines in the training data, since it is assumed that each line contains one sentence (empty lines are ignored).

We now want to find out how well out 10k vocabulary covers the test data. We could again use Linux tools for that, but SRILM contains a handy script compute-oov-rate that takes two arguments: the unigram count file and the list of vocabulary words.

TASK: What is the rate of out-of-vocabulary (OOV) words on the training, dev and test sets?

HINT: Use the same method as before to generate the unigrams for dev and test data.

SOLUTION:

- compute-oov-rate librispeech.top10k.vocab
- ngram-count -text data/dev.txt -order 1 -write dev.1grams
- compute-oov-rate librispeech.top10k.vocab dev.1grams
- ngram-count -text data/test.txt -order 1 -write test.1grams
- compute-oov-rate librispeech.top10k.vocab test.1grams

Usually we expect the OOV rate to be lowest on the training set because we used it to select the words (the vocabulary is biased toward the training set), but in this case the test sets have been chosen to be "cleaner" and have lower OOV rates. (The training data actually contains some languages other than English, though most of those will not make it into the vocabulary.)

Note that compute-oov-rate also reports about "OOV types". OOV types are the number of unique words that are missing from the vocabulary, regardless of how many times they occur.

The OOV rate of around 5% is quite high –remember that we will never be able to recognize those OOV words since the LM does not include them (they effectively have probability zero). However, we chose the relatively small vocabulary size of 10k to speed up experiments with the decoder later.

OPTIONAL TASK: Repeat the steps above for different vocabulary sizes (5k, 20k, 50k, 100k, 200k). Plot the OOV rate as a function of vocabulary size. What shape do you see?

TRAINING A MODEL

We are now ready to build a language model from the training data and the chosen vocabulary. This is also done using the ngram-count command. For instructional purpose we will do this in two steps: compute the N-gram statistics (counts), and then estimate the model parameters. (ngram-count can do both in one step, but that's not helpful to understand what happens under the hood.)

TASK: Generate a file containing counts of all trigrams from the training data. Inspect the resulting file

HINT: Consult the ngram-count man page and look up the options -order, -text, and -write. Remember the case mismatch issue.

SOLUTION: The first command uses about 10GB of memory and takes 15 minutes on a 2.4GHz Intel Xeon E5 CPU, so be sure to procure a sufficiently equipped machine and some patience.

- ngram-count -text librispeech-lm-norm.txt.gz -tolower -order 3 -write librispeech.3grams.gz
- gunzip -c librispeech.3grams.gz | less

Note that we want to compress the output file since it is large. The -order option in this case is strictly speaking optional since order 3 is the default setting. Note that the output is grouped by common prefixes of N-grams, but that the words themselves are not alphabetically sorted. You can use the -sort option to achieve the latter.

Now we can build the LM itself. (Modify the output file names from previous steps according to your own choices.)

TASK: Estimate a backoff trigram LM from librispeech.3grams.gz, using the Witten-Bell smoothing method.

HINT: Consult the ngram-count man page for options -read, -lm, -vocab, and -wbdiscount .

SOLUTION:

- ngram-count -debug 1 -order 3 -vocab librispeech.top10k.vocab -read librispeech.3grams.gz -wbdiscount -lm librispeech.3bo.gz

We added the -debug 1 option to output a bit of information about the estimation and resulting LM, in particular the number of N-grams output.

We will now try to understand the way LM parameters are stored in the model file. Peruse the file using

- gunzip -c librispeech.3bo.gz | less

or, if you prefer, gunzip the entire file using

- gunzip librispeech.3bo.gz

and open librispeech.3bo in an editor. Note: the editor better be able to handle very large files –the

LM file has a size of 1.3 GB.

Model evaluation

Consult the description of the backoff LM file format ngram-format(5), and compare to what you see in our model file, to be used in the next task.

TASK: Given the sentence “a model was born”, what is the conditional probability of “born”?

HINT: Consult the ngram-count man page for options -read, -lm, -vocab, and -wbdiscount .

SOLUTION: The model is a trigram, so the longest N-gram that would yield a probability to predict “born” would be “model was born”. So let’s check the model for that trigram. (One way to locate information in the model file is the zgrep command, which searches a compressed file for text strings. Each search string below starts with a TAB character to avoid spurious matches against other words that contain the string as a suffix. You can use your favorite tools to perform these searches.)

- zgrep " model was born" librispeech.3bo.gz

This outputs nothing, meaning that trigram is not found in the model, and we have to use the back-off mechanism. We look for the line that contains the context bigram “model was” following a whitespace character:

- zgrep -E “was” librispeech.3bo.gz | head -1

  -2.001953 model was 0.02913048

The first number is the log probability $P(was|model)$, which is of no use to use here. The number at the end is the backoff weight associated with the context “model was”. It, too, is encoded as a base-10 logarithm. Next, we need to find the bigram probability we’re backing off to, i.e., $P(born|was)$:

- zgrep -E “born” librispeech.3bo.gz | head -1

  -2.597636 was born -0.4911189

The first number is the bigram probability $P(born|was)$. We can now compute the log probability for $P(born|modelwas)$ as the sum of the backoff weight and the bigram probability:

$0.02913048 + -2.597636 = -2.568506$ or as a linear probability $10^{-2.568506} = 0.002700813$.

TASK: Compute the total sentence probability of “a model was born” using the ngram -ppl function. Verify that the conditional probability for “born” is as computed above.

SOLUTION: We feed the input sentence to the ngram command in a line of standard input, i.e., using “-“as the filename argument to -ppl. Use the option -debug 2 to get a detailed breakdown of the sentence-level probability:

- echo “a model was born” | ngram -debug 2 -lm librispeech.3bo.gz -ppl -

```
a model was born

p( a | <s> ) =
```

```
2gram
0.01653415
-1.781618

p( model | a ...) =
3gram
0.0001548981
-3.809954

p( was | model ...) =
3gram
0.002774693
-2.556785

p( born | was ...) =
2gram
0.002700813
-2.568506

p( </s> | born ...) =
3gram
0.1352684
-0.8688038

1 sentences, 4 words, 0 OOVs

0 zeroprobs, logprob= -11.58567 ppl= 207.555 ppl1= 787.8011
```

Notice how ngram adds the sentence start and end tags, <s> and </s>. The final line gives both the log probability and the perplexity of the entire sentence. The line starting "$p(born|was)$" has the conditional word probability that we computed previously. The label "2gram" indicates that a backoff to bigram was used. The final "logprob" value -11.58567 is just the sum of the log probabilities printed for each word token. Let's verify the perplexity value based on it's definition: we divide the logprob by the number of word tokens (including the end-of-sentence), convert to a probability and take the reciprocal (by negating the exponent): $10^{\frac{11.58567}{5}} = 207.555$. Of course this is not a good estimate of

perplexity as it is based on only 5 data points.

TASK: Compute the perplexity of the model over the entire dev set.

SOLUTION: The exact same invocation of ngram can be used, except we use the file containing the dev set as ppl input. We also omit the -debug option to avoid voluminous output. Note: these commands take a few seconds to run, only because loading the large LM file into memory takes some time –the model evaluation itself is virtually instantaneous.

- ngram -lm librispeech.3bo.gz -ppl data/dev.txt
  file dev.txt: 466 sentences, 10841 words, 625 OOVs
  0 zeroprobs, logprob= -21939 ppl= 113.1955 ppl1= 140.4475

We thus have a perplexity of about 113. The first line of summary statistics also gives the number of out-of-vocabulary words (which don't count toward the perplexity, since they get probability zero). In this case the OOV rate is 625/10841 = 5.8

Running the same command on the test set (data/test.txt) yields a perplexity of 101 and an OOV rate of 4.9%. Both statistics indicate that the test portion of the data is a slightly better match to our model than the development set.

TASK: Vary the size of the training data and observe the effect this has on model size and quality (perplexity).

HINT: The original librispeech-lm-norm.txt.gz has about 40 million lines. Use gunzip and the head command to prepare training data that is $\frac{1}{2}$, $\frac{1}{4}$, $\cdots$, of the full size. (This is very easy, but can you think of better ways to pare down the data?)

SOLUTION: Rebuild the model (using the original vocabulary), and evaluate perplexity for different amounts of data. Plot model size (number of ngrams in the head of the model file) and perplexity as a function of training data size. Details left to the student, using the steps discussed earlier.

Model adaptation

We will now work through the steps involved in adapting an existing LM to a new application domain. In this scenario we typically have a small amount of training data for the new, target domain, but a large amount, albeit mismatched data from other sources. For this exercise we target the AMI domain of multi-person meetings as our target domain. The language in this are spontaneous utterances from face-to-face interactions, whereas the "librispeech" data we used so far consisted of read books, a dramatic mismatch in speaking styles and topics.

We will use the "librispeech" corpus as our out-of-domain data, and adapt the model we just created from that corpus to the AMI domain, using a small amount of target-domain data corpus. Corpus subsets for training and test are in the data directory:

- wc -wl data/ami-*.txt

6473 data/ami-dev.txt

2096 20613 data/ami-test.txt

86685 924896 data/ami-train.txt

Also provided is a target domain vocabulary consisting of all words occurring at least 3 times in the training data, consisting of 6171 words:

- wc -l data/ami-train.min3.vocab

  6271 data/ami-train.min3.vocab

TASK: Build the same kind of Witten-Bell-smoothed trigram model as before, using the provide AMI training data and vocabulary. Evaluate its perplexity on the AMI dev data.

SOLUTION:

- ngram-count -text data/ami-train.txt -tolower -order 3 -write ami.3grams.gz
- ngram-count -debug 1 -order 3 -vocab data/ami-train.min3.vocab -read ami.3grams.gz -wbdiscount -lm ami.3bo.gz
- ngram -lm ami.3bo.gz -ppl data/ami-dev.txt

  file data/ami-dev.txt: 2314 sentences, 26473 words, 1264 OOVs

  0 zeroprobs, logprob= -55254.39 ppl= 101.7587 ppl1= 155.5435

TASK: Evaluate the previously built librispeech model on the AMI dev set.

SOLUTION: Again, this takes a few seconds due to the loading time of the large model.

- ngram -lm librispeech.3bo.gz -ppl data/ami-dev.txt

  file data/ami-dev.txt: 2314 sentences, 26473 words, 3790 OOVs

  0 zeroprobs, logprob= -56364.05 ppl= 179.8177 ppl1= 305.3926

Note how both the perplexity and the OOV count are substantially higher for this large model than for the much small, but well-matched AMI language model. If we modified the vocabulary of the old model to match the new domain its perplexity would increase further. (Can you explain why?)

We will now adapt the old model by interpolating it with the small AMI LM. As explained in the course materials, model interpolation means that all N-gram probabilities are replaced by weighted averages of the two input models. So we need to specify the relative weights of the two existing models, which must sum to 1. A good rule of thumb is to give the majority of weight (0.8 or 0.9) to the in-domain model, leaving a small residual weight (0.2 or 0.1) to the out-of-domain model.

TASK: Construct an interpolated model based on the existing librispeech and AMI models, giving weight 0.8 to the AMI model, and evaluate it on the dev set.

HINT: Make use of the ngram options -mix-lm, -lambda, and -write-lm.

- ngram -debug 1 -order 3 -lm ami.3bo.gz -lambda 0.8 -mix-lm librispeech.3bo.gz -write-lm ami+librispeech.bo.gz

- ngram -lm ami+librispeech.3bo.gz -ppl data/ami-dev.txt

  file ami-dev.txt: 2314 sentences, 26473 words, 783 OOVs

  0 zeroprobs, logprob= -56313.77 ppl= 102.546 ppl1= 155.6145

At first sight, this result is disappointing. Note how the perplexity is now 102, slightly up from the value that the AMI-only model produced. But also note how the number of OOVs was almost halved (from 1264 to 783), due to the addition of words covered by the out-of-domain model that were not in the AMI model. The model now has more words to choose from when making its predictions. An important lesson from this exercise is that we can only compare perplexity values when the underlying vocabularies are the same. Otherwise, the enlarged vocabulary is a good thing, as it reduces OOVs. The interpolation step has effectively adapted not only the model probabilities, but the vocabulary as well.

Still, it would be nice to do an apples-to-apples comparison to see the effect of just the probability interpolation on model perplexity. We can do this by telling the ngram tool to only use words from the AMI vocabulary in the interpolated model:

- ngram -debug 1 -order 3 -lm ami.3bo.gz -lambda 0.8 -mix-lm librispeech.3bo.gz -write-lm ami+librispeech.bo.gz -vocab data/ami-train.min3.vocab -limit-vocab

This is the same command as before, but with the -limit-vocab option added, telling ngram to only use the vocabulary specified by the -vocab option argument. We can now evaluate perplexity again:

file ami-dev.txt: 2314 sentences, 26473 words, 1264 OOVs

0 zeroprobs, logprob= -53856.04 ppl= 90.52426 ppl1= 136.8931

The number of OOVs is now back to the same as with ami.3bo.gz, but perplexity is reduced from 102 to 90.

TASK (optional): Repeat this process for different interpolation weights, and see if you can reduce perplexity further. Check results on both AMI dev and test sets.

This step is best carried out using the enlarged vocabulary, since that is what we want to use in our final model. But notice how we are now effectively using the dev set to train another model parameter, the interpolation weight. The result will thus be tuned to the dev set. This is why we better have another test set held out (data/ami-test.txt in this case) to verify that the result of this tuning also improves the model (lowers the perplexity) on independent data.

The tuning of interpolation weights would be rather tedious if carried out by trial and error. Fortunately, there is an efficient algorithm that finds optimal weights based on expectation maximation, implemented in the command compute-best-mix, described under ppl-scripts.

TASK (optional): Use compute-best-mix to find the best -lambda value for interpolation for the two models we built.

HINT: As input to the command, generate detailed perplexity output for both models, using ngram

-debug 2 -ppl data/ami-dev.txt .

Model pruning

We saw earlier that model size (and perplexity) varies with the amount of training data. However, if a model gets too big for deployment as the data size increases it would be a shame to have to not use it just for that reason. A better approach is to train a model on all available data, and then eliminate parameters that are redundant or have little effect on model performance. This is what model pruning does.

A widely used algorithm for model pruning based on entropy is implemented in the ngram tool. The option -prune takes a small value, such as 10^{-8} or 10^{-9}, and remove all ngrams from the model that (by themselves) raise the perplexity of the model less than that value in relative terms.

TASK: Shrink the large librispeech model trained earlier, using pruning values between 10^{-5} and 10^{-10} (stepping by powers of ten). Observe/plot the resulting model sizes and perplexities, and compare to the original model.

SOLUTION: Starting with $1e-5$ (= 10^{-5} in floating point notation), create the pruned model:

- ngram -debug 1 -lm librispeech.3bo.gz -prune 1e-5 -write-lm librispeech-pruned.3bo.gz

Then evaluate the librispeech-pruned.3bo.gz model on the entire dev set, as before. The -debug option lets the tool output the number of ngrams pruned and written out. Add the resulting number of bigrams and trigrams to characterize the pruned model size (roughly, the number of model parameters, since the number of unigrams is fixed to the vocabulary, and the backoff weights are determined by the probability parameters).

3.5 Speech Decoding

3.5.1 Overview

声学模型的目的是给对应的标签序列打分，对于输入数据的每一帧，AM 会给每一个标签打个分数。

语言模型的目的是给词序列打分，对于词表中的词构成的每个有效序列，LM 都会给这个序列打个分数。

粘合这两个模型的就是解码图（decoding graph）。解码图是结合语言模型打分，将有效声学标签序列映射成对应词序列的函数。

3.5.2 Weighted Finite State Transducers

3.5.2.1 Weighted Finite State Transducers

本课程中，我们将会使用到一些加权有限状态转移器（WFST）的结构。

有限状态自动机（finite state automata，FSA）是一个紧凑的图结构（a compact graph structures），FSA 对字符串集进行了编码，用于高效的搜索。

有限状态转移机（finite state transducer）是类似的结构，只不过 FST 将一个字符集到另一个字符集的映射进行了编码。

FSA 和 FST 都可以赋予权重，这样的，FSA 中的每一个元素，或者 FST 中的每一个映射对都有一个数值化的权重。

我们将在课程中去探索语音识别的任务是如何被编码成 WFST 的。

3.5.2.2 The Grammar, a Finite State Acceptor

举个例子，我们要搭建一个语音识别系统，所有有效的短语集都是由以下的五元素集合组成的，这样的集合可以用有限状态接收器（Finite State Acceptor，FSA）来表达。

```
any thinking
some thinking
anything king
something king
thinking
```

这个系统的词汇表包含六个单词：

```
any
anything
king
some
something
thinking
```

描述这些短语的 FSA 如图3.19，FSA 将短语编码成 graph 中的路径，始于初始状态，终于终止状态，词典中的词是 graph 中的弧。FAS 的这个示例图遵循着这些图的视觉惯例：

- 所有的路径都是从初始状态，也就是标签 0 开始的，结束于任意的状态。初始状态是双层圆；
- 状态之间的弧是有向弧，并且弧被标注为词典中的某个词；
- 接收器只有一个 symbol，transducer 有两个 symbol，由冒号隔开；

- 弧的权重默认为 0，除非特意制定。

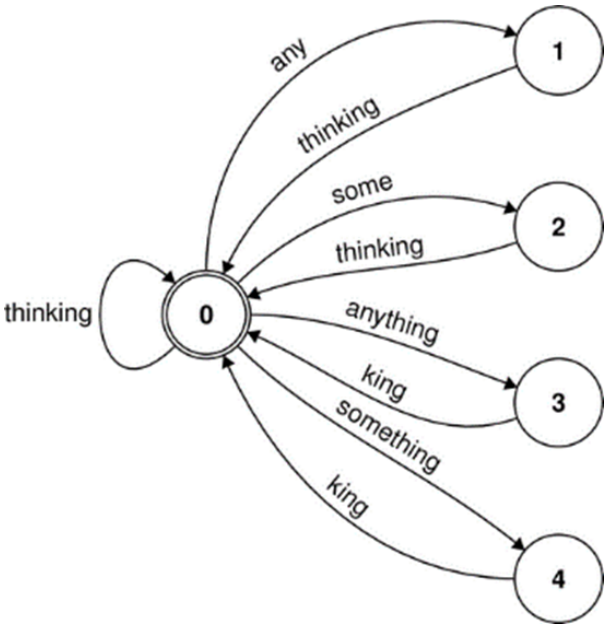

图 3.19: FSA 图例

为了使用 OpenFST 工具来编译这个解码图，我们需要一个 symbol 表，存入 **vocabulary.sym** 中，一个解码图的文本说明，存入 **Grammar.tfst** 中。symbol 表呢是对词典进行了映射，将人可读的文本映射到机器喜欢的整数。所以词典如下面这个表所示，其中 <eps> 这个 symbol 有点特殊，一会再说。

```
<eps> 0
any 1
anything 2
king 3
some 4
someting 5
thinking 6
```

解码图中的文本表征有一系列的弧和终止状态组成，弧的定义由初始状态、终止状态、输入标签、输出标签和一个可选的权重组成。终止状态的定义由一个状态标签和一个可选的权重组成，我们的 grammar 如下，比如说"0 1 any any" 指的是从状态 0 出发，结束于状态 1，输入标签是 any，输出标签也是 any，后面没有其他东西了，所以此处没有权重。

```
0 1 any any
1 0 thinking thinking
0 2 some some
```

```
2 0 thinking thinking
0 3 anything
3 0 king
0 4 something
4 0 king thinking
0 0 thinking
0 0
```

要注意哈，第一个弧定义的初始状态应当是整个图的初始状态，我们也将整个图的初始状态定义为文件最后一行的终止状态。那一般来讲呢，FSA 的输入和输出是一样的。

这样就可以用 **fstcompile** 指令来编译解码图：

```
fstcompile --isymbols=vocabulary.sym --osymbol=vocabulary.sym --
    keep_isymbols --keep_osymbols Grammar.tfst Grammar.fst
```

3.5.2.3 The pronounciation Lexicon, a Finite State Transducer

语音识别系统的一些组成部分里，我们需要将某种 symbol 序列与另外一种 symbol 序列联系起来，比如说词序列隐含着音素序列，音素序列又隐含着声学模型标签序列。

上小节中的六个单词，其音素发音如表3.2所示。这种从词映射到音素的映射叫做发音词典（pronounciation lexicon）。描述发音词典的 FST 接收有效的音素序列为输入，其输出为词序列。

表 3.2: 发音词典举例

Pronunciation	Word
EH N IY	any
EH N IY TH IH NG	anything
K IH NG	king
S AH M	some
S AH M TH IH NG	something
TH IH NG K IH NG	thinking

构建这样的一个 FST，一种可能的办法是给发音词典中的每个词都搭一条路径，FST 的第一部分看起来如下：

```
0 1 EH any
1 2 N <eps>
2 0 IY <eps>
0 3 EH anything
```

```
3 4 N <eps>
4 5 IY <eps>
5 6 TH <eps>
6 7 IH <eps>
7 0 NG <eps>
0 8 K king
8 9 IH <eps>
9 0 NG <eps>
...
```

此处我们又遇到了 <eps>，在构建解码图中的某条路径上的输入或者输出字符串的时候，epsilon symbol 是被忽略掉的。所以穿过状态 0,1,2,0 的路径的输出是 any, <eps>, <eps>，那忽略掉 <eps>，其实这条路径的输出就只有 any。

那仔细观察下上面的发音词典 FST，就会发现解码图中有些冗余。比如说，输入音素序列的起始部分是有效音素"EH"，但是编码"EH" 的弧却有两条，通往两个不同的状态。我们可以消除这些状态，即接受"EH" 为输入的时候，将它的输出设为 <eps>，然后将"any" 和"anything" 延迟输出，如下所示：

```
0 1 EH ~~any~~ <eps>
1 2 N ~~<eps>~~any
2 0 IY <eps>
~~0 3 EH anything~~
~~31~~ 4 N ~~<eps>~~anything
4 5 IY <eps>
5 6 TH <eps>
6 7 IH <eps>
7 0 NG <eps>
0 8 K king
8 9 IH <eps>
9 0 NG <eps>
...
```

"any" 这个词的编码路径就是 0, 1, 2, 0。"anything" 的编码路径是 0,1,4,5,6,7,0，状态 3 被去掉了，其所对应的那条弧也被取消了，而这个时候状态 1 就有两条弧了，这俩弧共享一个输入"N"，而且这个过程是可重复的。通过将从同一个状态出发的并且共享输入的弧合并来压缩解码图，这种操作叫做 **determinization**，它保证了每一个独有的输入字符串前缀对应着图中独有的状态。determinization 的互补操作叫做 **FST minimization**，以同样的合并方式来对字符串的后缀进行处理。在不影响 FSA 中描述中字符串集合的情况下，determinization 和 minimization 都可

以压缩图结构。如果没有这两张操作, 构建一个合理的解码图也很困难。

将用于描述发音词典的 FST 经过压缩后如图3.20，注意图中的状态和弧与上面未经压缩的 FST 的状态和弧并不一样，因为经过两种操作之后，有些状态和弧就不再需要了，为了使得状态是数字连续的，所以与上面提到的弧和状态不太一样。这个比图3.19中描述的 grammar FST 要复杂的多。图3.20中一条可能的路径为 0,4,8,11,3,7,0，其对应着输入字符串"EH N IY TH IH NG"，输出字符串为"<eps> <eps> <eps> anything <eps> <eps>"，"<eps>" symbol 是可以忽略掉的，所以很明显这条路径就对应着上面发音词典中的某个条目，即对应着词"EH N IY TH IH NG anything"。

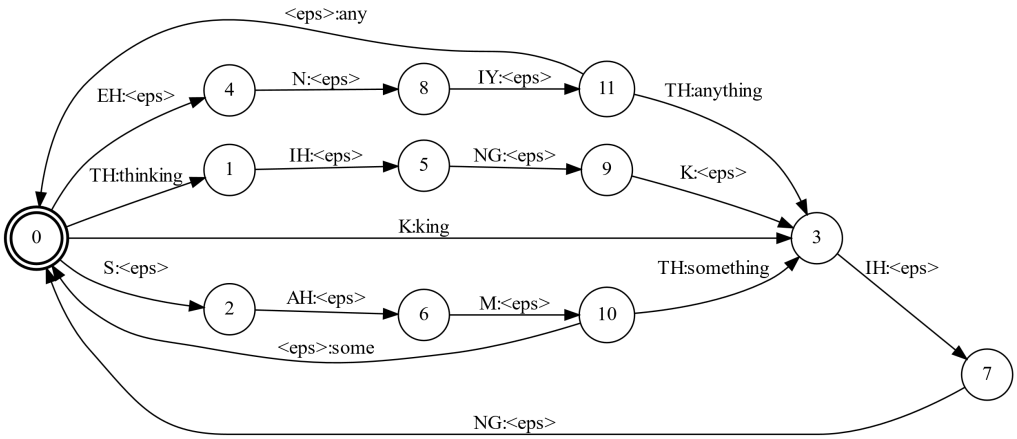

图 3.20: determinization 和 minimization 压缩后的 lexicon FST

类似于图3.20的 transducer，可能不止一条路径会接收相同的输入字符串，如果这些路径对应着不同的输出，这个 transducer 就是 non-functional。我们可以把这样的路径看成某个表达式的代数等价，这个代数表达式单个输入对应着两个不同的输出。

在图3.20中，穿过状态 0,4,8,11,0,1,5,9,3,7,0 的路径将"EH N IY TH IH NG K IH NG" 映射成"any thinking"。另外一条经过状态 0,4,8,11,3,7,0,3,7,0 的路径将一样的输入，即将"EH N IY TH IH NG K IH NG" 映射成"anything king"。因为函数应当是每一个输入只有一个输出，而该解码图没有这个特性，所以它是 non-functional。

尽管 non-functionalFST 是有效地结构，但是有些图算法是没有在这样的 FST 上应用的。比如说，标准的 determinization 算法就没法用于 non-functionalFST 中。（

在语音识别中，会出现 non-functionaltransducer 的情况有：

- 发音词典中的同音异形词（homophones）；
- 序列同音异形词（sequence homophones），比如上面的例子："anything king" 和"any thinking"；
- 声学模型的状态到音素序列的映射不是唯一的；

前两种情况可以用个技巧搞定，就是混入"disambiguation symbols"。第三种情况呢，超出了本课程的内容。

disambiguation symbols 就是修改词典中的每个发音，在这些发音后的最后加上一个人造的音素，加入这些 symbol 之后，我们的词典就变成了表3.3。

表 3.3: 发音词典举例

Pronunciation	Word
EH N IY #0	any
EH N IY TH IH NG #0	anything
K IH NG #0	king
S AH M #0	some
S AH M TH IH NG #0	something
TH IH NG K IH NG #0	thinking

这么搞就可以解决掉序列同音异形词序列的问题，"any thinking" 对应的输入为"EH N IY 0 TH IH NG K IH NG"；"anything king" 对应的输入为"EH N IY TH IH NG 0 K IH NG"。在发音词典中如果包含同音异形词，就是发音一样，词不一样，就给其发音后面加入一个独一无二的 symbol，从"#0" 开始，然后另外的发音依次在后面加入"#1"，"#2" 等等。恰当的使用 disambiguation symbols，任意的发音词典 FST 都是有效的。

另外一种是将一个 non-functional FST 变成一个唯一的（determinizable）的 FSA，即对 FST 进行编码。为了将 FST 编码成 FSA，解码图每个弧的输出和输出 symbol 都会被融合成一个单独的 symbol。这样输入和输出的概念就没了，编码而来的 FSA 描述了一个输入/输出对（input/output pair）的序列。因为是 FSA，自然就可以唯一化。在唯一化编码后的 FSA 之后，再逆转这个编码过程即可。

3.5.2.4 The HMM State Tranducer, a Finite State Tranducer

HMM 的状态序列 transducer H 是将声学模型的状态序列映射成音素标签序列。结合发音词典，该映射如表3.4。

表 3.4: 发音词典举例

Phone Sequence	Word
AH_s2 AH_s3 AH_s4	AH
EH_s2 EH_s3 EH_s4	EH
IH_s2 IH_s3 IH_s4	IH
IY_s2 IY_s3 IY_s4	IY
K_s2 K_s3 K_s4	K
N_s2 N_s3 N_s4	N
NG_s2 NG_s3 NG_s4	NG
S_s2 S_s3 S_s4	S
TH_s2 TH_s3 TH_s4	TH
#0	0

我们的模型结构是每一个音素都对应着三个声学标签序列，这些声学标签表征了一个音素声学实现的开始、中间和结束。大的声学模型一般对应着更多的声学标签，这些声学标签用来捕捉每一个音素声学实现的方式，这些声学实现依赖于序列中它所邻近的声音。这些 context dependent 模型不在本课程范围内。

值得注意的是表3.4应当包含发音词典中提到过的 disambiguation symbol，也就是表格的最后一行。对于 H transducer，任何嵌入到音素序列中的 disambiguation symbol 都应当映射到声学标签序列 symbols 中去。意思就是音素中的 disambiguation symbol 应当在 HMM 的状态中有所体现，这样才能连贯统一起来。

在本节的例子中，共有 10 个音素，每个音素对应三个 HMM 中的状态（也就是声学标签），其对应的 HMM transducer 如图3.21。每一个 phone 的模型都构成一个闭环，闭环中的输入是该音素的声学标签，比如说"AH_s2"，输出就是该音素，即"AH"。

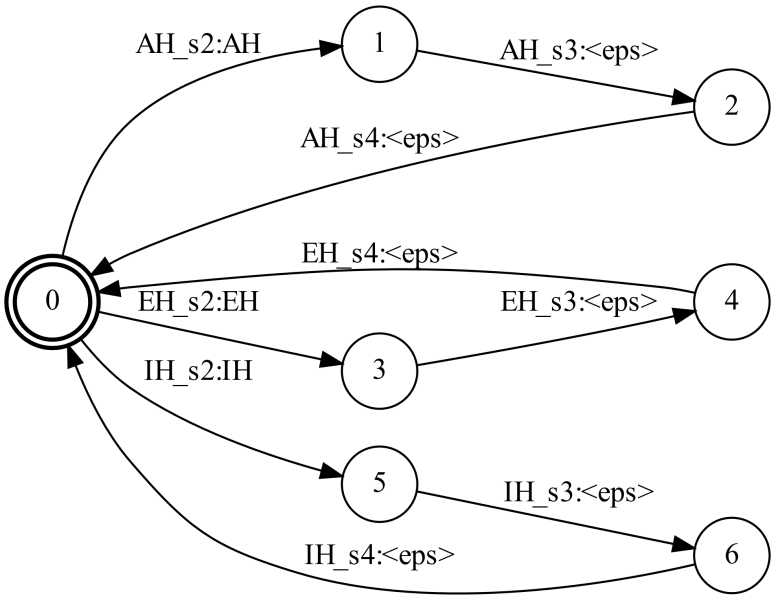

图 3.21: HMM 的 WFST

3.5.3 WFSTs and Acceptors

Weighted Finite State Automaton(WFSA) 同样还是 FSA，只不过 WFSA 给每个接收的序列分配个分数。每个 automaton 接收到的字符串都会映射成一个分数。发音词典 FST 可能会以权重的方式来增加每个单词的发音变体的相对编码概率。

一个 N-gram LM 就可以用 WFSA 来近似表达，其接收词串，其中包含着一些由 non-word token 表示的词，这些 token 意味着 backoff 的出现。如下所示，这是 transducer 中的一小部分，其编码了一些 ngrams。

```
-0.03195206 half an hour
-1.020315 half an -1.11897
-1.172884 an hour
-1.265547 an old
-1.642946 an instant
-1.698012 an end
```

这些 ngrams 的 WFST 如图3.22所示。在这解码图中，开始的三个 token 是起始 token，跟随着"half" 和"an"，然后这个图计算下一个词的权重。因为在英语中"half an hour" 是一个常见用法，所以它直接有一条路径从状态 3 到状态 4，这个路径编码了"hour" 这个词在 bigram "half an" 之后出现的概率。如果"hour" 后面还有其他的词，那么这个词所对的 context 就是"an hour"。

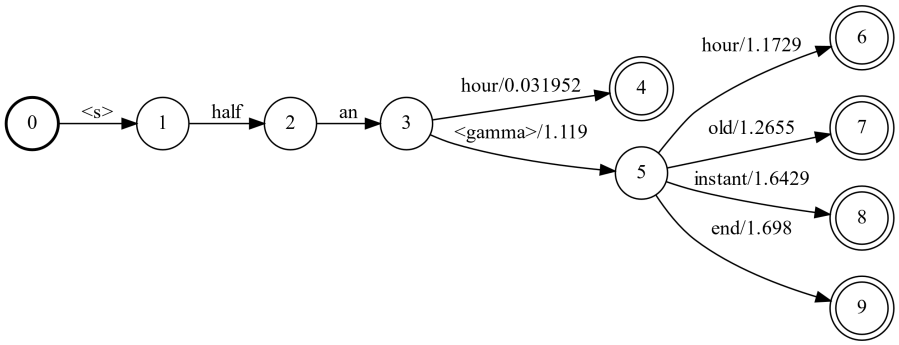

图 3.22: ngrams 的 WFST

对于剩下的那些也跟在"half an" 之后的词，即使在 LM 的定义中，没有它们的 trigram，解码图中依然有对应的路径。状态 3 代表的 context 是"half an"，而状态 5 只代表了一个 context，就是"an"。由于 LM 中没有这些路径的 trigram，那么使用 backoff 来收缩 context，从而得到这些路径所对应的权重。经过这种 penalty 操作之后，模型使得另外四个词在"an" 这个 context 下出现。这四个词里面还是有"hour"，这是因为"an hour" 在英语里也经常出现。

3.5.4 Graph Composition

我们比较感兴趣的最后一个算法是 FST compostion。正如代数函数的 composition 一样，通过将第一个 FST 的输出作为第二 FST 的输入，FST 的 compostion 就可以合并两个 FST。如果第一个 WFST 以权重 x 将 A 串映射到 B 串，第二个 WFST 以权重 y 将 B 串映射到 C 串，那么合并的 FST 就是以权重 $x + y$ 将 A 串映射到 C 串。（为什么权重是 $x + y$ 而不是 xy，因为所有的权重都是对数概率吗？）

3.5.4.1 The Decoding Graph

我们希望能有个解码图能将声学模型的状态序列和词序列统一起来。

在3.5.3节和3.5.2节中，我们看到了语音识别问题的结构是如何被编码成 WFST 的：

1. 语法 **G** 是个 FSA，其编码了短语集合，或者是个 WFSA，跟语言模型似的，给每个短语分配了个权重；
2. 发音词典 **L** 是个 WFST，其将带有 disambiguation symbols 的音素序列映射成词序列；
3. HMM transducer **H** 将 HMM 状态（senone labels）序列映射成 HMM 模型的名字（phones, triphones）；

虽然说本课程没有提到，但是在很多语音识别系统中，有四个 WFST，有一个咱们这里没讲的是将 triphone 序列映射成音素序列，一般叫做 context transducer **C**。

为了将声学模型状态序列和词序列联系起来，transducers H, C, L，G 应当是接连着被应用的。HCLG 可以在解码器运行的是完成。如果 compostion 是离线做的，预处理好的，解码器就简单多了。

将上述四个 WFST 合并起来，如图3.23。

实际应用中，一般是从左向右 compose 的，在每一次 composition 之后都要执行 determinize 和 minimize 操作。在3.5.5节的实验中，我们准备好已经 compose 的 H，C 和 L，所以剩下的就是创建 G 图，然后将 G 图也 compose 进去，最终生成完整的 HCLG WFST。

G 图在其输入端是包含一些 backoff symbol 的，HCL 在这些 backoff symbols 出现的时候将它们也传递过来，因此 HCLG 的输入中就会有这些 symbols。

同样，L 图中包含一些 disambiguation symbols，HCL 也会有这些 disambiguation symbols 作为输入，因此完整的 HCLG 也会有这些 symbols 作为输入。

因为解码器不需要这个信息，所以这些 symbol 一般都是由"<eps>" symbol 代替，这意味着遍历这些弧对输入串没有任何影响。（整个关于处理 backoff 和 disambiguation 的地方没有看懂，回头补上说明。）

3.5.4.2 The Search

语音识别的解码过程是找到能够最大化 LM 分数和 AM 分数联合分布的词序列。声学状态的得分由声学模型赋予，LM 模型的分数是通过解码图中该词序列的路径赋予的。

语音识别的解码算法就是通过解码图中的路径搜索算法，其中某一条路径的分数是解码图赋予它的分数和声学模型赋予它的分数之和。由模型的特性，我们可以用简答的动态规划算法来找到最短路径。如果最好的路径于 T 时刻经过状态 S，那么这个路径就一定包含最好的前缀路径，这个前缀路径在 T 时刻和状态 S 停止。

一个典型的帧同步的 beam search 算法有三个阶段，对于每一个时间 t：

1. 以跨过具有 non-epsilon 输入 symbol 的弧的方式，将图形中的每个部分假设向前推进；
   (a). 这么搞的话，所有的新的部分假设在其路径中都会有 t 个输入 symbols；
   (b). 如果两个部分假设碰到了同一个状态，那么只保留分高的那个假设

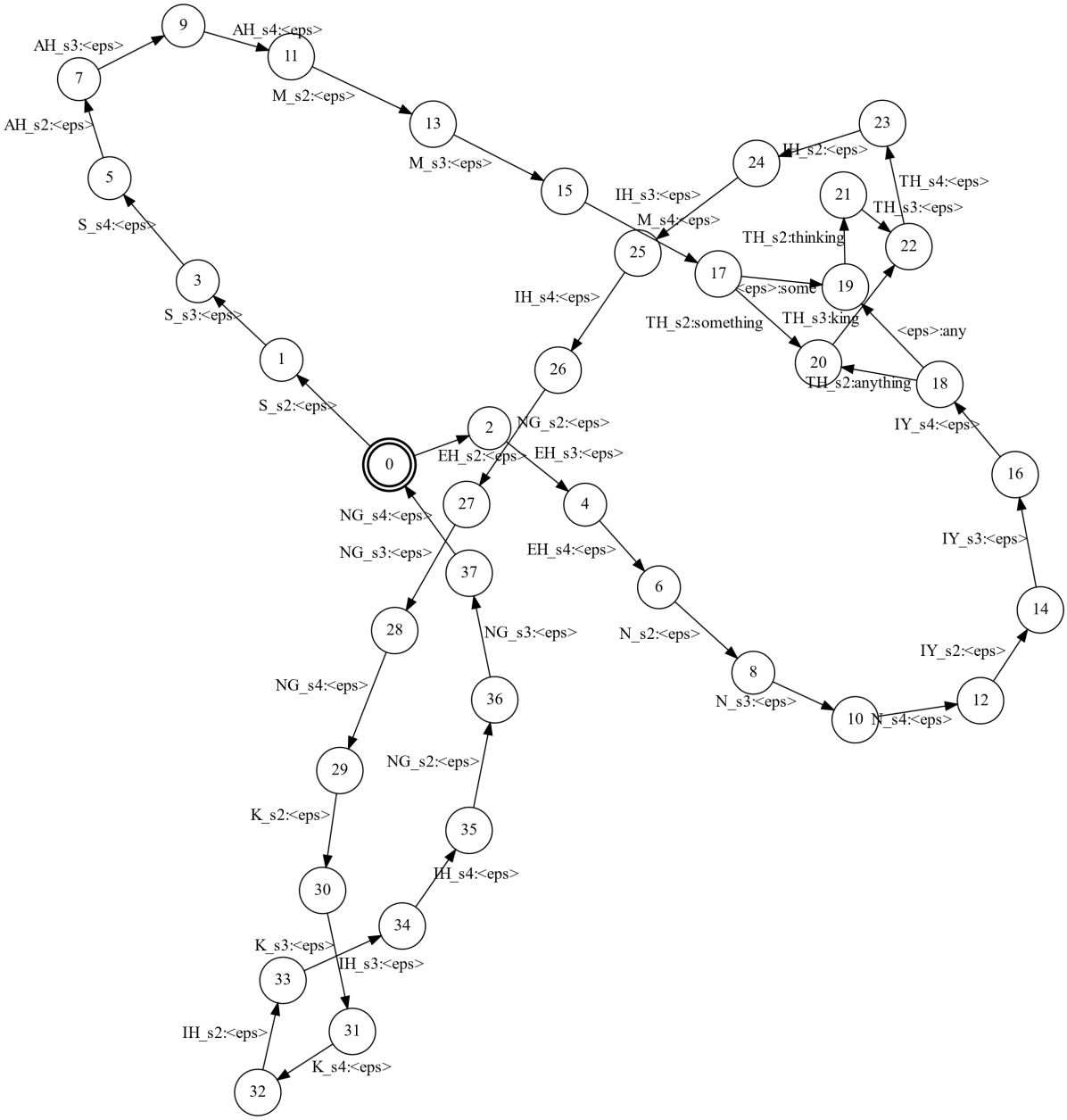

图 3.23: HCLG 解码器图例

2. 消除所有"out of beam" 的部分假设，也就是说要么保留 K 个最高的部分假设（K 是最多的 token 数），要么将所有比最好的部分假设分数低 B 分的假设都去掉（B 是 beam width）；
3. 跨过以"<eps>" 为输入 symbol 的弧来推进剩下的 tokens，同样如果两个部分假设碰到了同一个状态，保留高分假设。

beam width B 和最多 token count K 是剪枝搜索中重要的参数。beam width 保证了远离当前最好假设的部分假设会被丢掉；最多 token count 限制了每一帧需要处理的总工作量。

对于某个时刻 t，最好的路径可能分数很低，会被剪枝操作给遗弃掉。在这种情况下，beam-seach 就只能得到一个局部优化的路径。这种情况发生的时候，我们就说算法产生了一个搜索错误（search error）。通过增大 beam width 和增加 maximum token count 可以讲这些错误减小到零。

3.5.5 Lab 5: Decoding

REQUIRED FILES:

1. HCL.fst

   This is a finite state transducer that maps sequences of context independent phone states (acoustic model labels) to sequences of words. The lexicon has 200,000 words.

   HCL.fst has not been determinized. Instead, it has been prepared to make composition with a language model WFST as efficient as possible.

   This FST has disambiguation symbols on its input side. They ensure that every unique input sequence has a unique output sequences, regardless of homonyms. This preserves the functional nature of the transducer, which makes determinization possible.

   It is expected that the language model contains "<gamma>" symbols on its input side, which represent the backoff transitions of the language model. This HCL.fst contains arcs with "<gamma>" labels on the input and output so these transitions will also be present on the input side of the fully composed graph. If it did not, then the composed graph would not be functional, and determinization would be impossible.
2. DecodingGraph.fst

   This is the result of compiling HCL.fst with a trigram language model, and applying a series of transformations to remove both disambiguation and language model backoff symbols, as well as to compact the structure into fewer arcs.
3. H.FST.isym and L.fst.osym

   These are the input and output symbol tables that should cover the input and output of HCL.fst, DecodingGraph.fst, and any other decoding graph you build in this lab.
4. StaticDecoder.py

   This is a simple Python-based beam decoder. It relies on loading a CNTK acoustic model, a

WFST decoding graph, and pre-processed acoustic features.

INSTRUCTIONS:

1. Run the StaticDecoder.py to decode the test data using the provided DecodingGraph.fst and the Experiments/lists/feat_test.rscp generated in lab 2.
   (a). The provided DecodingGraph.fst is in OpenFst format, but the decoder expects it to be in text format. Create DecodingGraph.fst.txt using the OpenFST fstprint tool.
   (b). Run the provided decoder with default parameters on the test data, using any acoustic model built in Section 3 of this class.
   (c). Measure the word error rate of the decoder's output with respect to the given reference text, using the word error rate module from Section 1 of this class.
2. Create a new decoding graph using a language model you have trained in Module 4 of this class.
   (a). Convert the ARPA format language model to its FST approximation. The arpa2fst.py tool is provided for this purpose.
   (b). Compose your new G.fst with the given HCL.fst.
   (c). Process the graph using a mixture of label pushing, encoding, decoding, minimization, and determinization. As part of this process, all disambiguation symbols and language model backoff symbols should be manually converted into "<eps>".
   (d). Use the resulting HCLG.fst in place of DecodingGraph.fst to repeat Assignment 2 above.
3. Measure the time-accuracy tradeoff of the decoder.
   (a). Run the decoder with the provided DecodingGraph.fst two more times: once with the beam width decreased by a factor of ten, and once with the beam width increased by a factor of ten.
   (b). What do you observe about the relationship between decoding speed and word error rate? What do you expect if the beam width was increased to the point that no pruning occurred?

SUPPLEMENTARY MATERIAL:

https://cs.nyu.edu/~mohri/pub/hbka.pdf

This chapter from the Springer Handbook on Speech Processing and Speech Communication has more information than you will need to complete the labs, but may be interesting for the motivated student. Section 3 details some standard algorithms, and Section 4 describes how the WFST framework is typically applied for the speech recognition task.

3.6 Advanced Acoustic Modeling

3.6.1 Imporved Objective Functions

我们在3.3.4节中曾经提到过训练分类神经网络最常用的目标函数是基于帧的交叉熵，将每一个输入帧的 one-hot 标签 $z[t]$ 和声学模型的 softmax 输出进行比较。

我们定义：

$$z[i,t] = \begin{cases} 1 & z[t] = i, \\ 0 & \text{otherwise} \end{cases} \tag{3.44}$$

那么 softmax 输出 $y[i,t]$ 的交叉熵为：

$$L = -\sum_{t=1}^{T}\sum_{i=1}^{M} z\,[i,t]\log\left(y\,[i,t]\right) \tag{3.45}$$

对于声学建模任务来说，使用基于帧的交叉熵作为目标函数有三个不正确的事：

1. 声学数据的每一帧都有一个正确的标签；
2. 预测出来的正确的标签必须和其他帧是独立不相关的；
3. 训练数据中的每一帧是同等重要的。

那本小节咱们就来聊聊一些其他策略来解决这些不足。

3.6.2 Sequential Objective Function

从本质上来说，声学建模是一个序列任务（sequential task）。给定一个声学特征向量序列，输出一个词序列。如果有个模型能做到给特征序列输出词序列，那么从特征向量到声学标签的准确对齐就不重要了。在不考虑声学信号与标签序列的相对对齐的情况下，sequential objective function（SOF）要训练的是直接输出正确标签序列的模型。需要注意的是本处的 SOF 和原来咱们提过的序列区分性训练是不一样的。

SOF 允许训练标签随着时间变动。当模型收敛后，它会找到解释标签的一个 segmentation，而且模型遵守 ground truth 标签序列未经改变的限制。

基于帧的 CE 目标函数需要标签 $z[t]$ 的序列，序列的长度和声学特征向量序列的长度一样。SOF 对于每一个句子有个 symbol 序列，$S = \{s_0, s_1, ..., s_{K-1}\}$，$t$ 帧声学特征到 K 个 symbols 的对齐记作 $\pi[t]$，t 时刻在 S 中所对应的标签为 $\pi[t]$。

目标函数从基于帧变成基于 segment，由此来提升效果。只不过基于帧的 CE 给每一帧都赋予了标签，基于序列的 CE 有标签序列，可是赋予给哪些帧呢？本质上来讲，标签是可以随着时间变化的。训练模式时，SOF 会找到 segmentation，这样的 segmentation 好建模，还可以解释

数据，同时还遵守 ground truth 标签序列未曾改变的限制。与对齐中的 $z[i,t]$ 相比，我们定义一个伪对齐 $\gamma[i,t]$，其和 ground truth 有着一样的顺序，$\gamma[i,t]$ 是当前网络输出的函数。其可以是一个软对齐，也可以是个硬对齐。通过将对齐序列转换成个 HMM，再使用 Viterbi（硬对齐）或者 forward-backward（软对齐）就可以算出来这个 $\gamma[i,t]$。硬标签如图3.24。

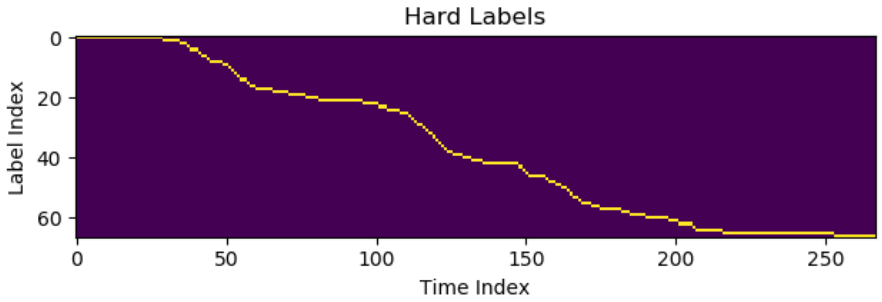

图 3.24: hard label 图

SOF 的公式如3.46。

$$\begin{aligned} L = & P(S|\pi) P(\pi) \\ = & P(\pi) \prod_t y[\pi(t), t] \\ = & -\sum_i \gamma[i,t] \log(y[i,t]) \end{aligned} \tag{3.46}$$

令 $z[k]$ 表示标签序列中的 symbol，此时重复项已经被移除。定义一个 HMM，表示依次移动这些标签。HMM 始终从状态 0 开始，总是结束于状态 $K-1$，并且会在状态 k 释放 symbol $z[k]$。软对齐由前向变量 α 和后向变量 β 计算得到，即：

$$\gamma[z[k], t] = \alpha[k,t] \beta[k,t] \tag{3.47}$$

前向递归计算了给定直到 t 时刻（包括 t 时刻）声学 evidence 下状态 k 的分数，其初始状态为序列中第一个标签的分数的模型预测值，如公式3.48。

$$\alpha[k,0] = \begin{cases} y[z[k], 0] & k = 0, \\ 0 & \text{otherwise} \end{cases} \tag{3.48}$$

递归沿着时间线向前计算，首先乘以带有元素 t_{ij} 的转移矩阵 T，再乘以对应的模型分数，

如公式3.49。

$$\alpha[k,t] = y[\bar{z}[k],t]\sum_{j} t_{kj}\alpha[j,\ t-1] \tag{3.49}$$

转移矩阵 T 是为了限制模型的拓扑结构的，使得 HMM 只能依照着标签序列从左向右前进，如公式3.50。

$$t_{\mathrm{ij}} = \begin{cases} 1 & i = j, \\ 1 & i = j+1, \\ 0 & \text{otherwise} \end{cases} \tag{3.50}$$

一段长度为 2.6s 的音频，其前向变量如图3.25。这段音频包含 66 个标签，黄色表示前向变量较大，紫色表示前向变量比较小。结构会离开主分支，沿着时间线搜索可能的路径。因为某个特定时刻的前向计算是不包括未来信息的，所以其使用的是当前信息来搜索所有可行的路径。

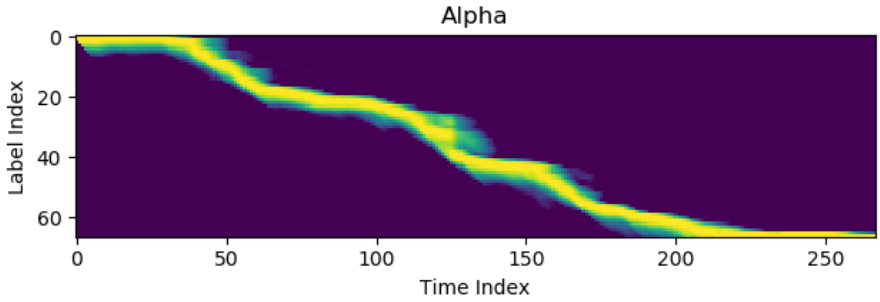

图 3.25: 前向变量 α 例图

后向递归计算的是给定从 segment 终点处到当前时刻 t（不包括当前时刻）的声学 evidence 的条件下状态 k 的分数。后向递归的初始化状态 $T-1$ 不包含任何声学 evidence，就是模型的终止状态，如公式3.51。

$$\beta[k,T-1] = 1 \tag{3.51}$$

后向递归首先利用模型给出的声学分数，然后再乘以转移矩阵 T 的转置，如公式3.52。

$$\beta_k[t] = \sum_{j} t_{jk}\beta[j,t+1]y[\bar{z}[j],t+1] \tag{3.52}$$

上文提到的 2.6s 音频的后向变量如图3.26，同样主分支周围有很多的可能路径。

将前向变量和后向变量结合起来，形成 γ 变量，每一个时间片段都包含整个句子的信息，分支的结构就消失了，标签索引和时间索引之间的对齐变得平滑了许多，如图3.27。

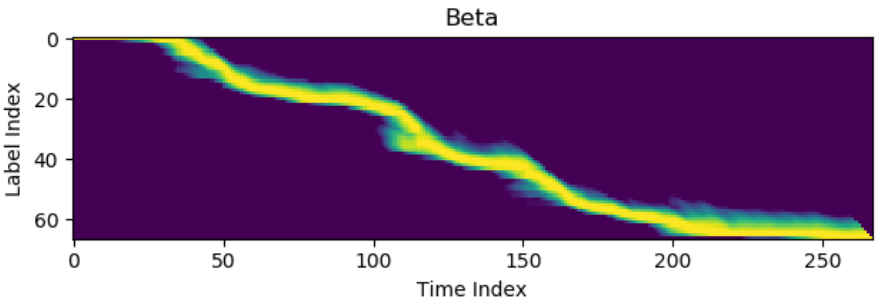

图 3.26: 后向变量 β 例图

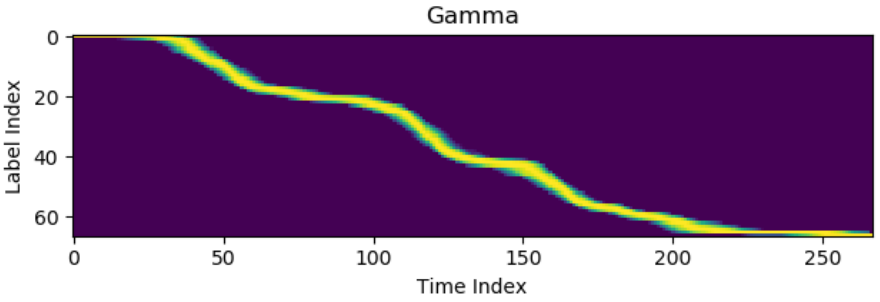

图 3.27: 前后向变量 γ 例图

那需要注意的是SOF仍然使用交叉熵作为损失函数，只不过这个是的ground truth和softmax输出针对的是segment。

3.6.3 Connectionist Temporal Classsification

CTC是一种SOF的特例，它减轻了交叉熵建模中的一些负担。交叉熵族的损失函数有一个显而易见的缺陷，它强制模型用标签去解释每一帧输入数据。CTC修改了标签集合，在字母表中加入了一个"don't care" 或者说"blank" symbol。正确标签路径的分数只取决于non-blank symbol。如果数据的某一帧关于音频的标签没有提供任何信息，CE还是强制性的要求这一帧做个选择。CTC系统就可以通过输出"blank" 的方式来表示这一帧没有足够的信息来区分这些标签。其损失函数如公式3.53。

$$L = \sum_{\pi} P(S|\pi) P(\pi) = \sum_{\pi} P(\pi) \prod_{t} y[\pi(t), t] \tag{3.53}$$

关于CTC的原理、推导和解码请去11.1节。

3.6.4 Sequence Discriminative Objective Functions

我们在3.3.6中提到过一种完全不同的目标函数，有的时候也叫 SOF，但是还是有很大的不同的。

在3.6.2中，"sequential objective function" 表示忽略声学数据和标签之前的对齐，只考虑沿着某条路径的序列标签，在3.3.6中，"sequence based objective function" 的意思是某条路径的后验概率不是通过所有标签序列来归一化的，而是在给定当前模型参数和解码限制下比较相似的这些序列。

比如说，MMI 损失函数如下：

$$F_{\text{MMI}} = \sum_{u} \log \frac{p\left(X_u|S_u\right) p\left(W_u\right)}{\sum_{W'} p\left(X_u|S_{W'}\right) p(W')} \tag{3.54}$$

最大化分子和最小化分母将会增加正确词序列的似然度。如果分母没有对词序列是否有效进行限制，这个目标函数就是基于帧的交叉熵损失函数。

为了不造成迷糊，我们偏向于把在训练中产生硬对齐或者软对齐的目标函数叫做_sequence training_，而那些限制 competitors 的目标函数叫做_discriminative training_。如果二者在目标函数中都有体现，那么我们就叫做_sequence discriminative training_

3.6.5 Grapheme or Word Labels

在新的任务上设定一个好的 senone 标签不仅耗时而且需要语言信息。我们需要了解语言的音素清单，协同发音效果的模型，发音词典，并且还得有带标签的数据才能这么干。因此尽管 senone 标签和语音的声学表征相匹配，我们还是不希望让它成为声学模型的目标输出。

Graphemes（字素）就可以用来替代 senones，它要简单地多。senones 与语言的声学实现相关，grapheme 对应的就是语言的书写格式。表3.5解释了六个常见单词的可能的 grapheme 表示和 phoneme 表示。

表 3.5: Words, Graphemes, Phonemes

Word	Phonemic Representation	Graphemic Representation
any	EH N IY	A N Y
anything	EH N IY TH IH NG	A N Y T H I N G
king	K IH NG	K I N G
some	S AH M	S O M E
omething	S AH M TH IH NG	S O M E T H I N G
thinking	TH IH NG K IH NG	T H I N K I N G

本例中，grapheme 来自于英语字母表的 26 个字母，这种表示方式的好处在于它不需要任何关于英语中这些字母应当如何发声的知识。那缺点呢就是这些发声的规则必须由声学模型从数

据中学习得到。一般来说的话，同样的训练数据，以 senone 为建模单元的效果要比以 grapheme 为建模单元的效果好。

为了提高基于 grapheme 的语音识别系统的性能，我们可以选择更为简化的 symbol 集合。我们可以利用一些简单的语言学只是来达到这个目的，比如说，在英语中：

- 字母对，比如说"T H" 和"N G" 经常发一个音，那么我们就可以用"TH" 和"NG" 来替代他们作为新的 symbol；
- 字母"Q" 经常后面跟着"U"，我们就可以引入"QU" symbol；
- 撇号没有发音，但是我们可以用 symbol 来表示缩写和复数结尾，比如说'T 和'S；
- 一些字母序列比较罕见，只在个别词中有，比如说两个"i" 为结尾的"hawaii"。用一个 symbol 来表示这些序列能消除因数据稀少带来的建模压力。

一种极端的例子是完全去掉 grapheme，就直接输出整词 symbol。这种模型一般用循环声学模型结合 CTC 来训练。尽管这样的系统比较简单，但是很难训练，而且会出现严重的词表问题。一个训练好的系统只能识别一个闭集内的单词，如果要增加词表的大小就得重新训练。怎么解决这样的限制是语音识别现在比较热门的方向。

grapheme 的解码器，类比于基于音素的识别系统的解码器，经常能提升识别效果。其解码网络是将字母序列映射成词序列，搜索的时候路径分数由 LM 分数和 grapheme 模型分数得到。

3.6.5.1 Encoder-Decoder Networks

尽管大多数语音识别系统采用单独的解码过程将标签分配给给定的语音帧，但是 encoder-decoder 网络使用神经网络来递归地生成其输出。

encoder-decoder 网络在机器翻译系统中很常见，用于将源语言转换成同义的目标语言。这类人物不需要包含词序的信息，而且通常源语言和目标语言的词也没有一对一的关系。

当这种思想应用到语音识别上，源语言就是音频的声学实现，目标语言是音频的文本描述。

跟机器翻译不同，语音识别任务是单调且一对一的，每一个说出来的词都对应着一个书面形式的词。所以需要对 encoder-decoder 网络进行一些修改。

encoder-decoder 的基本形式是：encoder 部分将整个 segment 编码成一个向量，传递给 decoder，受输入向量的刺激，decoder 应当递归式的输出正确的输出。因为这种长期记忆和总结能力受今天的 RNN 的限制，所以这种结构一般会用 attention mechanism 来代替。注意机制是解码器每个循环步骤的辅助输入，其中 decoder 实际上可以根据其内部状态查询 encoder 网络的某些状态。

decoder 网络经过训练可以递归地发射 symbol 并更新其状态，就像 RNN 语言模型一样。通常使用 beam search 算法针对可能的 decoder 网络状态树找到给定 encoder 网络状态的最可能输出。

3.7 补充知识点

3.7.1 傅里叶变换

3.7.2 Nyquist 定理

3.7.3 编辑距离

本节分析参考的是明无梦的 blog[24]。**Description:**

给定 2 个字符串 a,b，编辑距离是将 a 转换为 b 的最少操作次数，其操作只允许以下三种：

1. 插入一个字符（Insert）：例如 $fj \rightarrow fxj$；
2. 删除一个字符（Delete）：例如 $fxj \rightarrow fj$；
3. 替换一个字符（Substitue）：例如 $jxj \rightarrow fxj$；

Solutions:

采用分而治之的思想来解决这个问题，将复杂的问题分解成相似的子问题。

假设字符串 a，共有 m 位，从 $a[1]$ 到 $a[m]$；字符串 b，共有 n 位，从 $b[1]$ 到 $b[n]$。

设 $d[i][j]$ 表示字符串 $a[1]$ 到 $a[i]$ 转换为 $b[1]$ 到 $b[j]$ 的编辑距离。那么有如下递归规律，其中 $a[i]$ 和 $b[j]$ 分别为

3.7.4 nonlinear infinite impulse response (IIR)

3.7.5 序列区分性训练中的目标函数详解

3.7.6 概率与似然度的是是非非

3.7.7 <s> 和 </s> 概率上的意义

第 4 章 Kaldi 学习笔记

4.1 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。

4.1.1 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4.1。

$$\begin{aligned} t_1' &= \frac{t_1}{\alpha} \\ t_2' &= \frac{t_2}{\alpha} \end{aligned} \tag{4.1}$$

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。

4.1.2 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. "

4.2 kaldi 中的 UBM

通用背景模型 UBM(Universal Background Model)

4.3 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。

4.4 kaldi 中的数据准备

4.4.1 音频、文本、说话人准备

准备词典 词典的作用是将词映射成音素，不论是基于 HMM-GMM 的对齐，还是 HMM-DNN 的声学模型训练，其建模单元为音素，但是在实际交流中，我们看到的都是词或者词组成的句子，因此我们需要这么一个文件，在对齐或者训练声学的时候，将词映射成发音；在解码的时候我们看到的是词而不是音素，因此需要训练语言模型将音素转换成最有可能的词。因此，词典是非常非常重要的。

首先 lexicon.txt 的格式是**"<word> <pronunciation>"**。对应的"<pronounciation>" 就是词的音素。表4.1分别展示了中英文的 lexicon.txt 的示例。

在准备词典的时候，其对应的路径为 **data/local/lm**，其文本内容类似于图4.1。词和发音之间以 Tab 隔开，音素之间以空格隔开。

表 4.1: 中英文 word 和对应 pronounciation 示例

Word	Pronouciation
七叶树	qi iH iH yi ehH ehL shu uH uL
episode	EH1 P IH0 S OW2 D

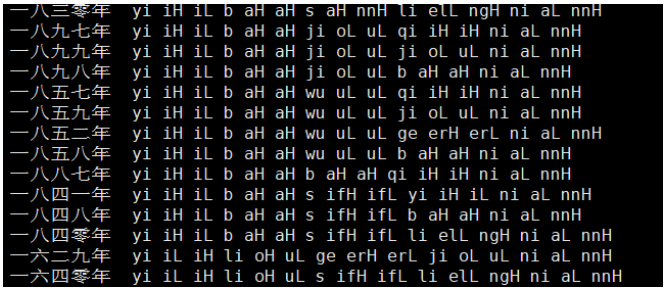

图 4.1: lexicon.txt 中存储格式

另外一个很重要的文件是 vocab.txt，其路径为 **data/local/lm/vocab.txt**。其存储了待建模的数据库中包含的所有的词。如果是中文的话，就需要对中文语句进行分词，如果对应的数据库已经分好了词，那么

4.5 kaldi 中的决策树

4.6 kaldi 错误及解决办法集锦

1. ERROR: FstHeader::Read: Bad FST header: -：出现的原因是 Memory 不够，语言模型太大了。构图的时候，在终端输入 top 指令，再摁个 E，就可以很清晰的看到随着构图的进行，Memory 逐渐被吃掉……log 如下：

```
fstminimizeencoded
fstdeterminizestar --use-log=true
fsttablecompose /data2/src/tmp/big-lm/lang_pp_test_tgsmall/L_disambig.
    fst /data2/src/tmp/big-lm/lang_pp_test_tgsmall/G.fst
fstpushspecial
ERROR: FstHeader::Read: Bad FST header: -
ERROR (fstdeterminizestar[5.5]:ReadFstKaldi():kaldi-fst-io.cc:35)
    Reading FST: error reading FST header from standard input

[ Stack-Trace: ]
kaldi::MessageLogger::LogMessage() const
```

```
kaldi::MessageLogger::LogAndThrow::operator=(kaldi::MessageLogger
    const&)
fst::ReadFstKaldi(std::string)
main
__libc_start_main
fstdeterminizestar() [0x425209]

kaldi::KaldiFatalErrorERROR: FstHeader::Read: Bad FST header: –
ERROR (fstminimizeencoded[5.5]:ReadFstKaldi():kaldi–fst–io.cc:35)
    Reading FST: error reading FST header from standard input

[ Stack–Trace: ]
kaldi::MessageLogger::LogMessage() const
kaldi::MessageLogger::LogAndThrow::operator=(kaldi::MessageLogger
    const&)
fst::ReadFstKaldi(std::string)
main
__libc_start_main
fstminimizeencoded() [0x413559]

kaldi::KaldiFatalErrorERROR: FstHeader::Read: Bad FST header: –
ERROR (fstpushspecial[5.5]:ReadFstKaldi():kaldi–fst–io.cc:35) Reading
    FST: error reading FST header from standard input

[ Stack–Trace: ]
kaldi::MessageLogger::LogMessage() const
kaldi::MessageLogger::LogAndThrow::operator=(kaldi::MessageLogger
    const&)
fst::ReadFstKaldi(std::string)
main
__libc_start_main
fstpushspecial() [0x401b99]
```

那么我们怎么来使用大的语言模型呢？可以先用小的语言模型参与解码，然后生成对应的 lattice，再使用 rescoring 的方法利用大的语言模型来进行重新估计，解决步骤如下：

(a). 使用 utils/format_lm.sh 生成小语言模型的 G.fst；

(b). 使用 utils/format_lm.sh 生成大语言模型的 G.fst；

(c). 使用 utils/mkgraph.sh 来逐步构图，生成 HCLG.fst；

(d). 使用 steps/online/nnet3/prepare_online_decoding.sh来准备在线解码；

(e). 使用 steps/online/nnet3/decode.sh来进行在线解码生成 lattice；

(f). 使用 steps/lmrescore.sh 来进行 rescoring。切记**如果是 chain model，在运行这条指令时要加上"–self-loop-scale 1.0"**。

2. ……

4.7 OpenFst

第 5 章 FFmpeg 和 sox

5.1 FFmpeg

5.1.1 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

5.2 sox

sox 的参考资料：

- SoX —音频处理工具里的瑞士军刀

第 6 章 Linux 相关笔记

6.1 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6.1。

```
:set ff=unix
```

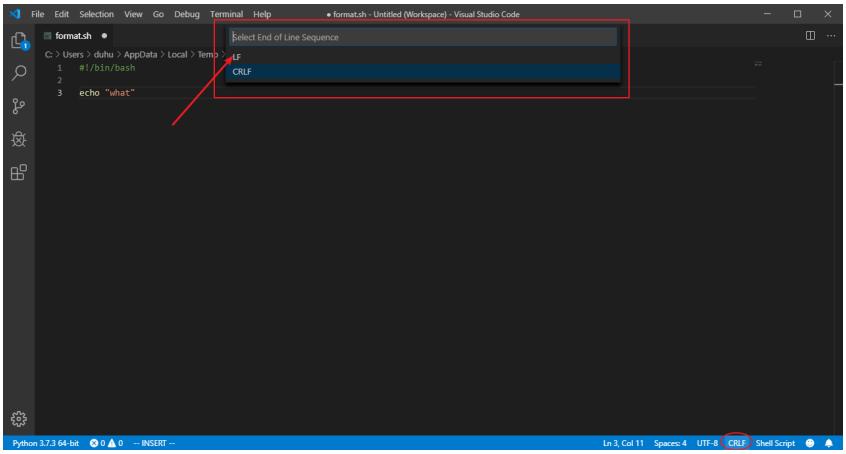

图 6.1: VScode 设置脚本格式为 Unix

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。

6.2 Shell 指令笔记

6.2.1 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
```

6.2.2 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6.1。

表 6.1: 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6.2。

   (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6.3。

表 6.2: 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 6.3: 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6.4。

表 6.4: 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

6.2.3 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6.1所示。

Listing 6.1: 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
```

6.2.4 grep

本小节分为两块一个是 grep 指令的一般性操作，如代码6.2。

Listing 6.2: 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。

6.2.5 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[26]，等以后更熟悉了再做全面的总结。代码如6.3。

Listing 6.3: 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6.4。

Listing 6.4: 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
```

6.2.6 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[27]。

第 7 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。

7.1 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 8 章 Python 笔记

8.1 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
5. 进度条效果：

```
sys.stdout.write('.')
sys.stdout.flush()
```

8.2 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

8.2.1 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。

8.2.1.1 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：

```
import os
# 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))
```

8.3 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……

8.3.1 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8.1中。

Listing 8.1: 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)
```

8.3.2 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8.1列出了一般的数据类型、其表示和占据字节数。

表 8.1: 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

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；
- 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8.2。

表 8.2: 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8.2中……

Listing 8.2: 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8.1，我们可以这样处理，代码8.3中仅写了打包发送音频信息的部分，其他部分见后面 socket。

Listing 8.3: 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])
```

8.3.3 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8.4和8.5，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 8.4: 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 8.5: 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)
```

8.3.4 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8.6。

Listing 8.6: 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()
```

8.3.5 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8.7和8.8。

Listing 8.7: 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 8.8: 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()
```

8.4 Python 中的正则表达式

8.5 替代 os 的 pathlib

8.6 Python 中的单元测试框架 unittest

第 9 章 C++ 学习笔记

9.1 C 语言碎记

1. system("pause") 可以让程序输出的 cmd 窗口卡住，方便看程序输出结果，在 VS 中没啥用，VC 中有点用，其来自于 stdlib.h；

9.2 C++ 库

9.2.1 标准库

<cassert>：
此头文件原作为 <assert.h> 存在于 C 标准库。此头文件是错误处理库的一部分
<utility>：

9.3 C++ 中的预定义宏，__FILE__ 等

9.4 #pragma once 和 #ifndef 作用与区别

9.5 C++ 指针

1. 指针和字符串：先看下程序。

```
#include <iostream>
#include <cstring>
int main()
{
  using namespace std;
  int x[3] = { 1,2,3 };
  int* p1 = x;
  char animal[20] = "bear";
  const char* bird = "wren";
  char* p = animal;
  cout <<"x:␣" << x << endl;
  cout << "pl:␣" << p1 << endl;
```

```
    cout << "*pl:␣" << *p1 << endl;
    cout << "animal:␣" << animal << endl;
    cout << "(int␣*)␣animal" << (int*) animal << endl;
    cout << "p:␣" << p << endl;
    cout << "(int␣*)␣p" << (int*) p << endl;
    cout << "*p:␣" << *p << endl;
    cout << "bird:␣" << bird << endl;
    cout << "*bird:␣" << *bird << endl;
    return 0;
}
```

根据输出我们可以看到指针对于字符串和数组的处理方式是不一样的，对于字符串，指针指向这个字符串，那么指向的是这个字符串的首地址，但是直接输出指针的话，输出的是字符串的内容，如果我们对这个指针取"*"，即解除引用，输出的就是字符串的第一个字符，如何能得到字符串首字母的地址呢？使用 (int *) pointer 或者 (int *) name_str 就可以了。而对于数组，指针指向的是数组的首地址，那么输出指针，输出的就是这个首地址。程序的输出如下：

```
x: 0x7ffca993eb00
pl: 0x7ffca993eb00
*pl: 1
animal: bear
(int*) animal: 0x7ffca993eae0
p: bear
(int*) p: 0x7ffca993eae0
*p: b
bird: wren
*bird: w
```

2. 混合类型指针

```
#include <iostream>
#include <cstring>
using namespace std;
struct antarctica_years_end
{
  int year;
```

```
};
int main()
{
  antarctica_years_end s1, s2, s3;
  s1.year = 1998;
  antarctica_years_end* pa = &s2;
  pa->year = 1999;
  antarctica_years_end trio[3];
  trio[0].year = 2003;
  cout << trio->year << endl;
  const antarctica_years_end* arp[3] = { &s1, &s2, &s3 };
  cout << arp[0]->year << endl;
  const antarctica_years_end** ppa = arp;
  cout << (*ppa)[0].year << endl;
  cout << (*ppa)->year << endl;
  cout << (*(ppa + 1))->year << endl;
  return 0;
}
```

9.6 swig 对 C++ 库进行 python 包装

第 10 章 Docker

10.1 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
```

10.2 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
```

10.3 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。

10.4 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 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[11]，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[23] 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[18]，如图11.1。我觉得这个图很好的说明了 CTC 的内涵和特点。

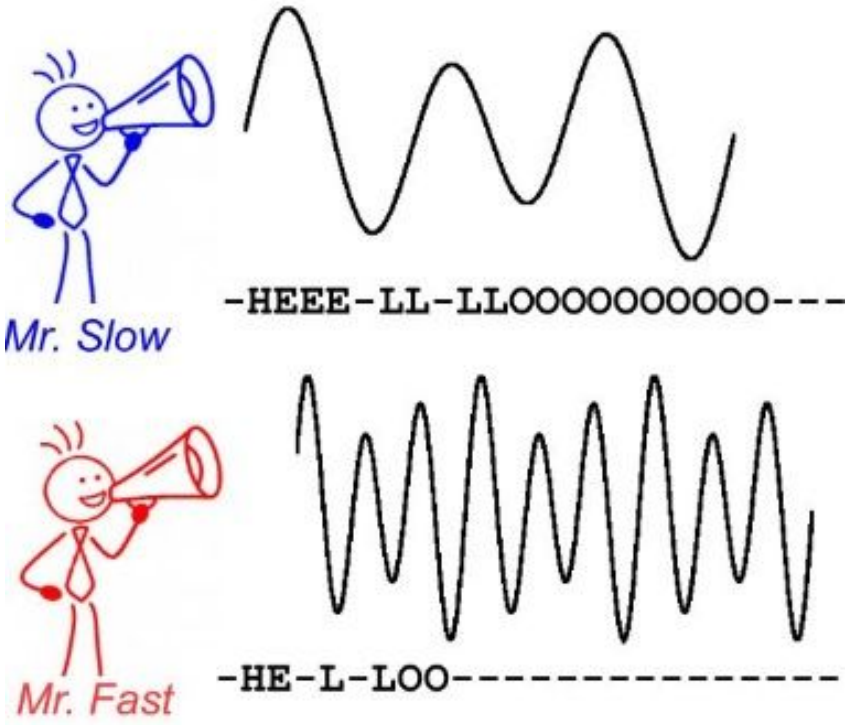

图 11.1: 同一单词不一样的输出音频的图解

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 $V(t,u) = \{\pi \in A^{'t} : \mathcal{F}(\pi) = l_{1:u/2}, \pi_t = l^{'}_u\}$，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[8]，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[13] 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[15]。

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。

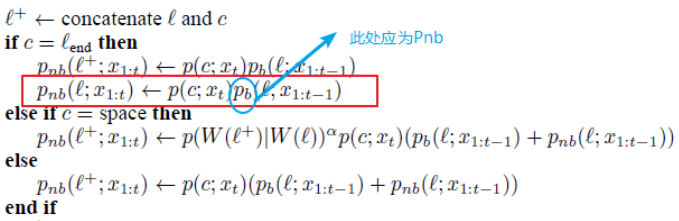

图 11.3: Prefix Beam Search 原论文中算法错误地方

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[10; 9]。

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。

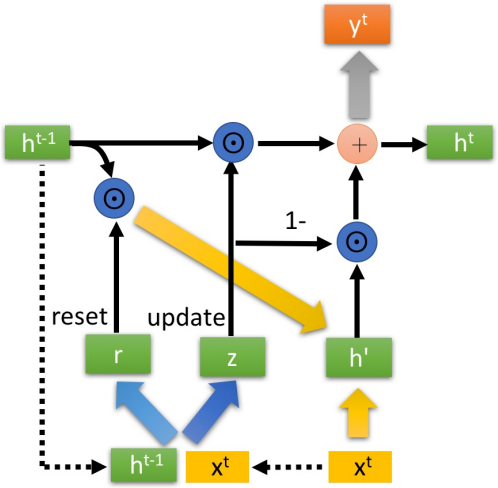

图 12.1: GRU 模型结构图

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。

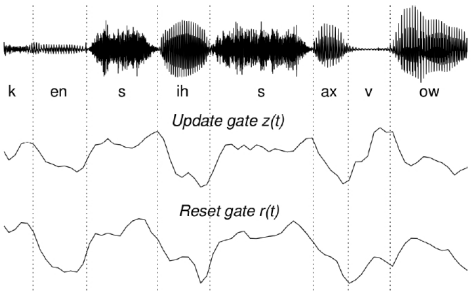

Fig. 1: Average activations of the update and reset gates for a GRU trained on TIMIT in a chunk of the utterance "*sx403*" of speaker "*faks0*".

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

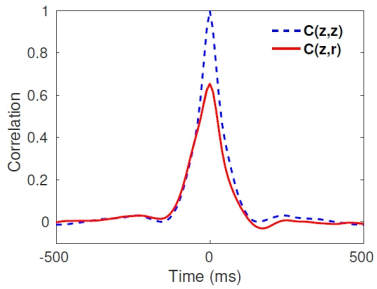

Fig. 2: Auto-correlation $C(z,z)$ and cross-correlation $C(z,r)$ between the average activations of the update (z) and reset (r) gates. Correlations are normalized by the maximum of $C(z,z)$ for graphical convenience.

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}

第 13 章 深度学习框架笔记

13.1 paddlepaddle

本处记录一些用 paddlepaddle 跑百度的 DeepSpeech2 的笔记和问题：

1. 基于一些原因，服务器 A 上的 GPU 暂时不可用，因此在服务器 B 上重新部署了百度基于 paddlepaddle 的DS2。A 和 B 有共享目录，因此这个 repository 是放在一个共享目录下的，但是在运行./run_train.sh 之后，模型初始化没有任何问题，log 显示已经开始在训练了，但是在存模型那块死活不动。因为之前在服务器 A 上训练过一段时间，所以对应存储 checkpoint 的路径是服务器 A 的用户创建的，如果没有给够权限的话，服务器 B 是没有办法存储 checkpoint 到原来服务器 A 的那个文件夹下的，所以两种办法：给够路径足够的权限或者以服务器 B 的用户重新创建一个路径；
2. 在利用训练和测试数据生成 vocab.txt 的时候，因为是从对应的文本中提取的 vocab.txt，不同的数据库文本不一样，所以最终生成的 vocab.txt 也是不一样的。千万不要用其他数据库的 vocab.txt 来对当前库进行训练或者测试，因为结果会惨不忍睹；
3. DS2 提取的音频特征是 80 维的 fbank。

13.2 pytorch

13.3 tensorflow

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] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[4] Peter F. Brown, Vincent J. Della Pietra, Peter V. deSouza, Jenifer C. Lai, and Robert L. Mercer. Class-based n-gram models of natural language. *Computational Linguistics*, 18:18–4, 1990.

[5] colah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[6] Jason Eisner. An interactive spreadsheet for teaching the forward-backward algorithm. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 10–18, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

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

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

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

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

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

[12] Michael Levit, Sarangarajan Parthasarathy, Shawn Chang, Andreas Stolcke, and Benoit Dumoulin. Word-phrase-entity language models: Getting more mileage out of n-grams. In *Proc. Interspeech*, pages 666–670. ISCA - International Speech Communication Association, September 2014.

[13] 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.

[14] mlwiki. Understanding lstm networks. http://mlwiki.org/index.php/Smoothing_for_Language_Models.

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

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

[17] 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.

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

[19] Daniel Ramage. Hidden markov models fundamentals. http://cs229.stanford.edu/section/cs229-hmm.pdf.

[20] RossYoung. 语音识别中的决策苏. https://blog.csdn.net/huashui2009120/article/details/79545164.

[21] Xiaoyu Shen, Youssef Oualil, Clayton Greenberg, Mittul Singh, and Dietrich Klakow. Estimation of gap between current language models and human performance. In *Proc. Interspeech 2017*, pages 553–557, 2017.

[22] A. Stolcke. Entropy-based pruning of backoff language models. In *Proceedings DARPA Broadcast News Transcription and Understanding Workshop*, pages 270–274, Lansdowne, VA, 2002.

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

[24] 明无梦. 编辑距离（edit distance）. https://www.dreamxu.com/books/dsa/dp/edit-distance.html.

[25] 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.

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

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