Model on Speech Recognition

Todo List

1. Chiu, Chung-Cheng, et al. "State-of-the-art speech recognition with sequence-to-sequence models." *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018.
2. Zou, Wei, et al. "Comparable study of modeling units for end-to-end mandarin speech recognition." *2018 11th International Symposium on Chinese Spoken Language Processing (ISCSLP)*. IEEE, 2018.
3. Park, Daniel S., et al. "Specaugment: A simple data augmentation method for automatic speech recognition." *arXiv preprint arXiv:1904.08779* (2019).
4. Hannun, Awni, et al. "Deep speech: Scaling up end-to-end speech recognition." *arXiv preprint arXiv:1412.5567* (2014).
5. Amodei, Dario, et al. "Deep speech 2: End-to-end speech recognition in english and mandarin." *International conference on machine learning*. 2016.
6. Battenberg, Eric, et al. "Exploring neural transducers for end-to-end speech recognition." *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, 2017.
7. T. N. Sainath and C. Parada. Convolutional neural networks for small-footprint keyword spotting. In Sixteenth Annual Conference of the International Speech Communication Association, 2015
8. tensorflow优化器源码

AIC 和 BIC 准则

Notes

模型选择问题在 **模型复杂度** 与 **模型对数据集描述能力** 之间寻求最佳平衡；

AIC 准则

赤池信息准则（Akaike Information Criterion，AIC），AIC 定义为：

$$AIC = 2k - 2ln(L)$$

其中 k 为参数个数，L 为似然函数。从一组可供选择的模型中选择最佳模型时，**通常选择 AIC 最小的模型**：

- 当两个模型之间存在较大差异时，差异主要体现在似然函数项，当似然函数差异不显著时，上式第一项，即模型复杂度则起作用，从而**参数个数少**的模型是较好的选择；
- 一般而言，当模型复杂度提高（k 增大）时，似然函数 L 也会增大，从而使 AIC 变小，但是 k 过大时，似然函数增速减缓，导致 AIC 增大，模型过于复杂容易造成过拟合现象；
- 目标是选取AIC最小的模型，AIC不仅要提高模型拟合度（极大似然），而且引入了惩罚项，使模型参数尽可能少，有助于降低过拟合的可能性；

BIC 准则

贝叶斯信息准则（Bayesian Information Criterion，BIC），BIC 定义为：

$$BIC = kln(n) - 2ln(L)$$

其中，k 为模型参数个数，n 为样本数量，L 为似然函数。从一组可供选择的模型中选择最佳模型时，**通常选择 BIC 最小的模型**；

比较

AIC 和BIC 的公式中后半部分是一样的；当 $n \geq 10^2$ 时，$kln(n) \geq 2k$，所以，BIC 相比AIC 在**大数据量时对模型参数惩罚得更多**，导致 BIC 更倾向于选择参数少的简单模型。所以还是**考虑使用 BIC 准则**；

Links

- 参考链接：AIC和BIC准则详解

交叉熵、相对熵、JS 散度

熵

熵（信息熵）指的是信息量的期望；

$$H(X) = -\sum_{i=1}^{n} p(x_i)\log(p(x_i))$$

相对熵（KL 散度）

相对熵（KL 散度）用来衡量两个分布的差异；

$$D_{KL}(p||q) = \sum_{i=1}^{n} p(x_i)\ log\left(\frac{p(x_i)}{q(x_i)}\right)$$

相对熵是非对称的，使用时 $p(x)$ 用来表示样本的真实分布，而 $q(x)$ 用来表示模型所预测的分布；

交叉熵

交叉熵可以通过相对熵变化而来，在机器学习中通常直接用交叉熵作为损失函数；

$$H(p,q) = -\sum_{i=1}^{n} p(x_i)\log(q(x_i))$$

JS 散度

JS 散度用来衡量两个分布的相似度，是基于相对熵的变体，解决了相对熵的非对称问题；

$$JS(P_1||P_2) = \frac{1}{2}KL(P_1||\frac{P_1+P_2}{2}) + \frac{1}{2}KL(P_2||\frac{P_1+P_2}{2})$$

Wasserstein 距离

Wasserstein 距离用来度量两个概率分布之间的距离，解决了当两个分布 P, Q 相差很远时，KL 散度和 JS 散度梯度消失的问题；

$$W(P_1, P_2) = \inf_{\gamma \sim \Pi(P_1, P_2)} \mathbb{E}_{(x,y)\sim\gamma}[||x-y||]$$

Optimizer in Deep Learning

梯度下降法（Gradient Descent）

梯度下降法的计算过程就是沿梯度下降的方向求解极小值，也可以沿梯度上升方向求解最大值。使用梯度下降法更新参数：

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J(\theta)$$

批量梯度下降法（BGD）

在整个训练集上计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \nabla_\theta J_i(\theta, x^i, y^i)$$

因为要计算整个数据集，收敛速度慢，但其优点在于更趋近于全局最优解；

随机梯度下降法（SGD）

每次只随机选择一个样本计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J_i(\theta, x^i, y^i)$$

训练速度快，但是容易陷入局部最优点，导致梯度下降的波动非常大；

小批量梯度下降法（Mini-batch Gradient Descent）

每次随机选择 n 个样本计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{m} \cdot \sum_{i=x}^{i=x+m-1} \cdot \nabla_\theta J_i(\theta, x^i, y^i)$$

这种方法是 BGD 和 SGD 的折衷；

动量优化法

Momentum

参数更新时在一定程度上保留之前更新的方向，同时又利用当前 batch 的梯度微调最终的更新方向。在 SGD 的基础上增加动量，则参数更新公式如下：

$$
\begin{aligned}
m_{t+1} &= \mu \cdot m_t + \alpha \cdot \nabla_\theta J(\theta) \\
\theta_{t+1} &= \theta_t - m_{t+1}
\end{aligned}
$$

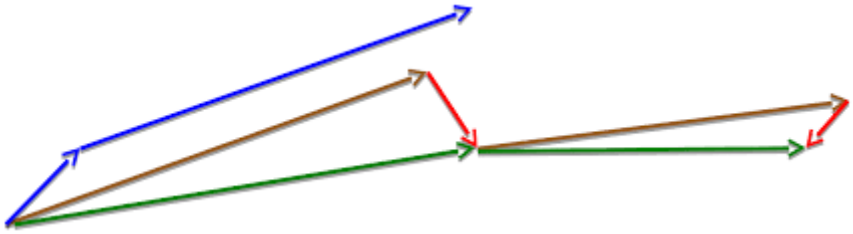
在梯度方向发生改变时，Momentum 能够降低参数更新速度，从而减少震荡；在梯度方向相同时，Momentum 可以加速参数更新，从而加速收敛。

NAG（Nesterov Accelerated Gradient）

与 Momentum 不同的是，NAG 是在更新梯度是做一个矫正，即提前往前探一步，并对梯度作修改。参数更新公式如下：

$$
\begin{aligned}
m_{t+1} &= \mu \cdot m_t + \alpha \cdot \nabla_\theta J(\theta - \mu \cdot m_t) \\
\theta_{t+1} &= \theta_t - m_{t+1}
\end{aligned}
$$

两者的对比：蓝色为 Momentum， 剩下的是 NAG；



自适应学习率

AdaGrad

AdaGrad 算法期望在模型训练时有一个较大的学习率，而随着训练的不断增多，学习率也跟着下降。参数更新公式如下：

$$
\begin{aligned}
g &\leftarrow \nabla_\theta J(\theta) \\
r &\leftarrow r + g^2 \\
\Delta\theta &\leftarrow \frac{\delta}{\sqrt{r+\epsilon}} \cdot g \\
\theta &\leftarrow \theta - \Delta\theta
\end{aligned}
$$

学习率随着 **梯度的平方和**（r）的增加而减少. 缺点：

- 需要手动设置学习率 δ ，如果 δ 过大，会使得正则化项 $\frac{\delta}{\sqrt{r+\epsilon}}$ 对梯度的调节过大；
- 中后期，参数的更新量会趋近于 0，使得模型无法学习；

Adadelta

Adadelta 算法将 梯度的平方和 改为 **梯度平方的加权平均值。**参数更新公式如下：

$$
\begin{aligned}
g_t &\leftarrow \nabla_\theta J(\theta) \\
E[g^2]_t &\leftarrow \gamma \cdot E[g^2]_{t-1} + (1-\gamma) \cdot g_t^2 \\
\Delta\theta_t &\leftarrow -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \\
\theta_{t+1} &\leftarrow \theta_t + \Delta\theta_t
\end{aligned}
$$

上式中仍存在一个需要自己设置的全局学习率 η，可以通过下式**消除全局学习率的影响**：

$$
\begin{aligned}
E[\Delta\theta^2]_t &\leftarrow \gamma \cdot E[\Delta\theta^2]_{t-1} + (1-\gamma)\cdot \Delta\theta_t^2 \\
\Delta\theta_t &\leftarrow -\frac{\sqrt{E[\Delta\theta^2]_{t-1}+\epsilon}}{\sqrt{E[g^2]_t+\epsilon}}\cdot g_t \\
\theta_{t+1} &\leftarrow \theta_t + \Delta\theta_t
\end{aligned}
$$

RMSProp

RMSProp 算法是 AdaGrad 算法的改进，修改 梯度平方和 为 **梯度平方的指数加权移动平均**，解决了学习率急剧下降的问题。参数更新公式如下：

$$
\begin{aligned}
g_t &\leftarrow \nabla_\theta J(\theta) \\
E[g^2]_t &\leftarrow \rho \cdot E[g^2]_{t-1} + (1-\rho)\cdot g_t^2 \\
\Delta\theta &\leftarrow \frac{\delta}{\sqrt{E[g^2]_t+\epsilon}}\cdot g \\
\theta &\leftarrow \theta - \Delta\theta
\end{aligned}
$$

Adam（Adaptive Moment Estimation）

Adam 算法在动量的基础上，结合了偏置修正。参数更新公式如下：

$$
\begin{aligned}
g_t &\leftarrow \Delta_\theta J(\theta) \\
m_t &\leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot g_t \\
v_t &\leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot g_t^2 \\
\hat{m}_t &\leftarrow \frac{m_t}{1-(\beta_1)^t} \\
\hat{v}_t &\leftarrow \frac{v_t}{1-(\beta_2)^t} \\
\theta_{t+1} &= \theta_t - \frac{\delta}{\sqrt{\hat{v}_t}+\epsilon}\cdot \hat{m}_t
\end{aligned}
$$

论文伪代码：

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1-\beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1-\beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t}+\epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Links

- 参考链接：优化算法 Optimizer 比较和总结
- Adam 论文链接：Kingma D P, Ba J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

高斯混合模型 GMM

Notes

高斯分布

例如，对城市人口的身高水平进行抽样调查，调查的结果就符合高斯分布，如下图所示：

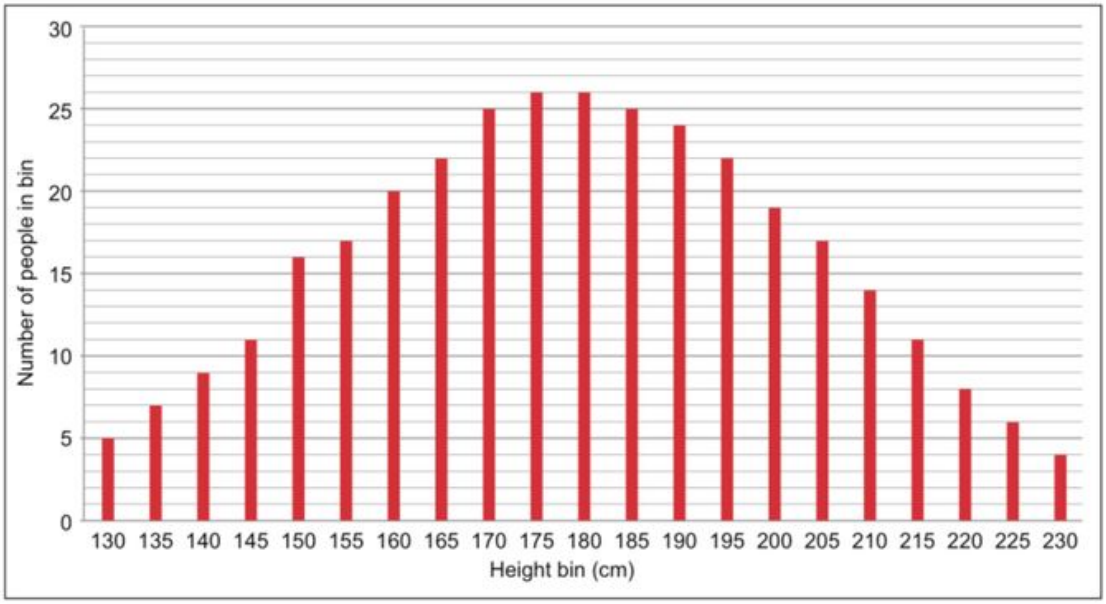


Figure 2.9 Histogram of a normal distribution, in this case the height of 334 fictitious people. The modal (most frequently occurring) bin is centered at 180 cm.

一元高斯分布的概率密度函数公式如下所示：

$$f_s(x|\mu,\delta^2)=\frac{1}{\sqrt{2\delta^2\pi}}e^{-\frac{(x-\mu)^2}{2\delta^2}}$$

多元高斯分布的概率密度函数公式如下所示：（n 为数据的维度）

$$f_m(x|\mu,\delta^2)=\frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}\cdot e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$$

高斯混合分布

例如，对城市中的男女身高水平分别建模，这时候男生的身高符合一个高斯分布，而女生的身高则符合另外一个高斯分布，即形成了一个高斯混合分布，如下图所示：

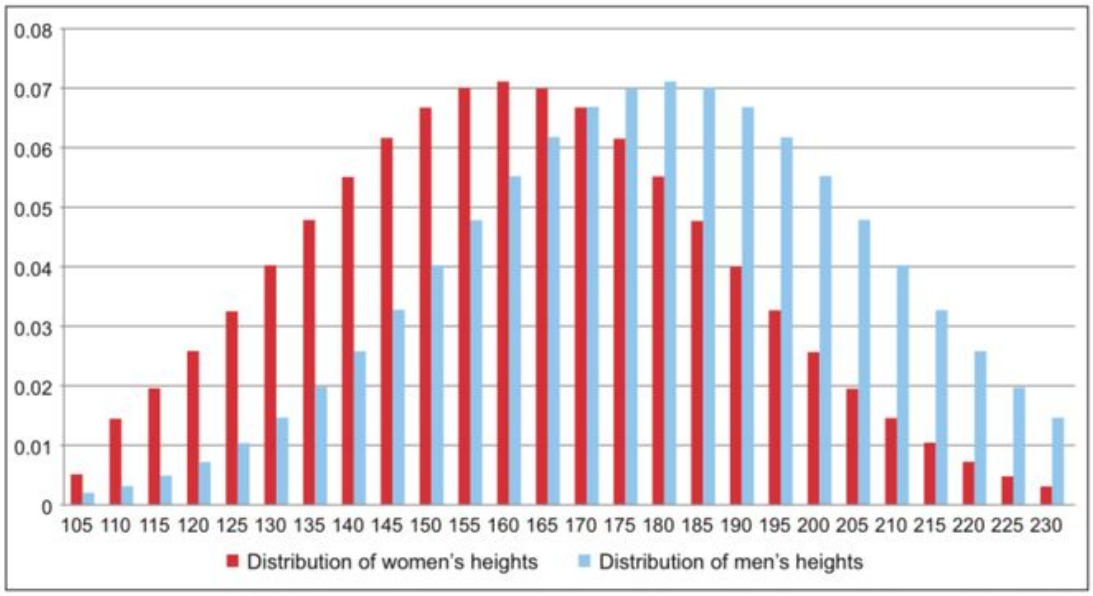


Figure 2.11 Probability distributions of height for men and women. Note that these probabilities are conditioned on the fact that gender is known: so, for example, given that we know a particular person is a woman, her probability of having a height in a particular bucket can be read off the y-axis.

一元高斯混合分布的概率密度函数公式如下所示：

$$p(x|\phi,\mu,\delta)=\sum_{i=1}^{K}\phi_i\cdot\frac{1}{\sqrt{2\delta_i^2\pi}}e^{-\frac{(x-\mu_i)^2}{2\delta_i^2}}=\sum_{i=1}^{K}\phi_i\cdot f_s(x|\mu_i,\delta_i)$$

多元高斯混合分布的概率密度函数公式如下所示：

$$p(x|\phi,\mu,\delta)=\sum_{i=1}^{K}\phi_i\cdot\frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma_i|^{\frac{1}{2}}}\cdot e^{-\frac{1}{2}(x-\mu_i)^T\Sigma_i^{-1}(x-\mu_i)}=\sum_{i=1}^{K}\phi_i\cdot f_m(x|\mu_i,\delta_i)$$

多元高斯分布的最大似然估计

假设我们对 m 个样本采样，得到**似然估计函数**：

$$L(\mu,\Sigma)=(2\pi)^{-\frac{mn}{2}}\cdot|\Sigma|^{-\frac{m}{2}}\exp\left(-\frac{1}{2}\sum_{i=1}^{m}(x^{(i)}-\mu)^T\Sigma^{-1}(x^{(i)}-\mu)\right)$$

计算**对数似然估计函数**：

$$\ln(L(\mu,\Sigma))=-\frac{mn}{2}\cdot\ln 2\pi-\frac{m}{2}\cdot\ln|\Sigma|-\frac{1}{2}\cdot\sum_{i=1}^{m}(x^{(i)}-\mu)^T\Sigma^{-1}(x^{(i)}-\mu)$$

参数优化的目的是，采样这 m 个样本时，尽可能地保证它们出现的概率最大化，即最大化对数似然估计。下面通过求极值的方式（偏导为 0）来计算参数值：

$$\begin{cases}\frac{\partial\ln(L(\mu,\Sigma))}{\partial\mu}=0\\ \frac{\partial\ln(L(\mu,\Sigma))}{\partial\Sigma}=0\end{cases}$$

最终得到：（详细过程见 参考链接2）

$$\begin{cases}\hat{\mu}=\frac{1}{m}\cdot\sum_{i=1}^{m}x^{(i)}\\ \hat{\Sigma}=\frac{1}{m}\cdot\sum_{i=1}^{m}(x^{(i)}-\mu)(x^{(i)}-\mu)^T\end{cases}$$

多元高斯混合分布的 EM 算法流程

EM 算法是用于**含有隐变量**的概率模型参数的最大似然估计方法，其首先估计样本属于某一个概率模型的可能性，然后在这个可能性的基础上最大化似然函数，不断迭代更新参数直至收敛。多元高斯混合分布参数的具体更新流程如下：

1. 初始化模型参数，这里初始化方法有两种：**随机初始化** 或 **K-means初始化**；
2. **E-Step**：根据当前参数 $(\phi^{(t)}, \mu^{(t)}, \delta^{(t)})$，计算**每个数据 x_j 属于第 k 个多元高斯分布的可能性**（t 表示迭代的轮数）；

$$\gamma_{j,k}^{(t+1)} = \frac{\phi_k^{(t)} \cdot f_m(x_j|\mu_k^{(t)}, \delta_k^{(t)})}{\sum_{i=1}^{K} \phi_i^{(t)} \cdot f_m(x_j|\mu_i^{(t)}, \delta_i^{(t)})}$$

3. **M-Step**：更新模型参数（m 为样本个数）；

$$\mu_k^{(t+1)} = \frac{\sum_{j=1}^{m} \gamma_{j,k}^{(t+1)} \cdot x_j}{\sum_{j=1}^{m} \gamma_{j,k}^{(t+1)}}$$

$$\Sigma_k^{(t+1)} = \frac{\sum_{j=1}^{m} \gamma_{j,k}^{(t+1)} \cdot (x_j - \mu_k^{(t+1)})(x_j - \mu_k^{(t+1)})^T}{\sum_{j=1}^{m} \gamma_{j,k}^{(t+1)}}$$

$$\phi_k^{(t+1)} = \frac{\sum_{j=1}^{m} \gamma_{j,k}^{(t+1)}}{m}$$

4. 重复迭代 $2, 3$ 步骤，直至收敛，收敛的条件是**样本集似然概率**的增长小于某个阈值；

$$\text{Log-Likelihood}(\mathcal{X}, \phi_k^{(t)}, \mu_k^{(t)}, \delta_k^{(t)}) = \sum_{j=1}^{m} \log\left(\sum_{i=1}^{K} \phi_i^{(t)} \cdot f_m(x_j|\mu_i^{(t)}, \delta_i^{(t)})\right)$$

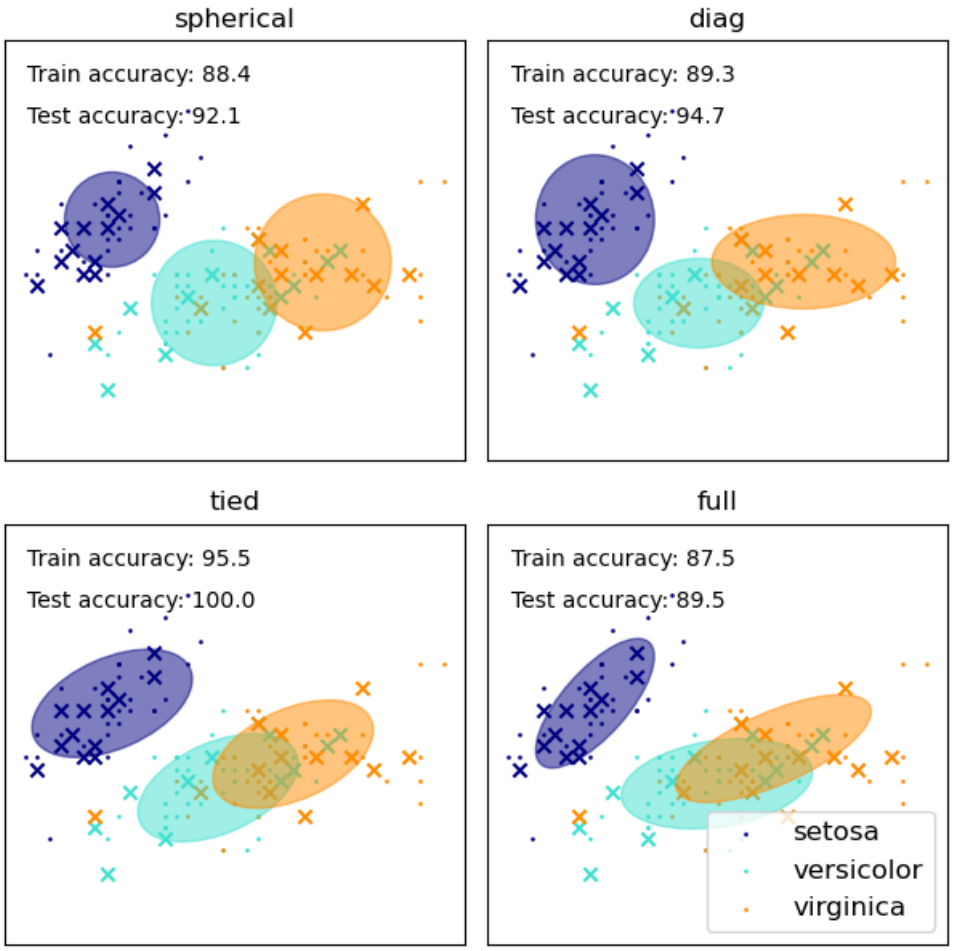
$$\text{If } \left(\text{Log-Likelihood}(\mathcal{X}, \phi_k^{(t+1)}, \mu_k^{(t+1)}, \delta_k^{(t+1)}) - \text{Log-Likelihood}(\mathcal{X}, \phi_k^{(t)}, \mu_k^{(t)}, \delta_k^{(t)})\right) \leq \text{threshold}, \text{ Then stop E/M steps.}$$

Codes

实验一

参考代码：https://scikit-learn.org/stable/auto_examples/mixture/plot_gmm_covariances.html#sphx-glr-auto-examples-mixture-plot-gmm-covariances-py

- 代码结果：



- 均值的初始化：和实验二中的初始化方法（随机初始化）不同的是，这里在随机初始化之后，还**额外自定义初始化了每个高斯分布的均值**，导致这个不同的关键点在于**实验一的训练数据是知道隐状态的**，**额外知道隐状态十分有利于提升模型的精度和拟合能力**，下面列出代码中的不同

```
# 关键差异：知道训练数据的隐状态
X_train = iris.data[train_index]
y_train = iris.target[train_index]
......
# 初始化每个高斯分布的均值
estimator.means_init = np.array([X_train[y_train == i].mean(axis=0)
                                 for i in range(n_classes)])
```

- 对 `covariance_type` 参数的理解：

  (1) `covariance_type='full', each component has its own general covariance matrix`，每个高斯分布的轴的方向不同，且可以任意旋转，并且轴长不相同；

  (2) `covariance_type='tied', all components share the same general covariance matrix`，每个高斯分布的轴（轴长，轴的方向）是相同的，但轴的方向可以旋转；

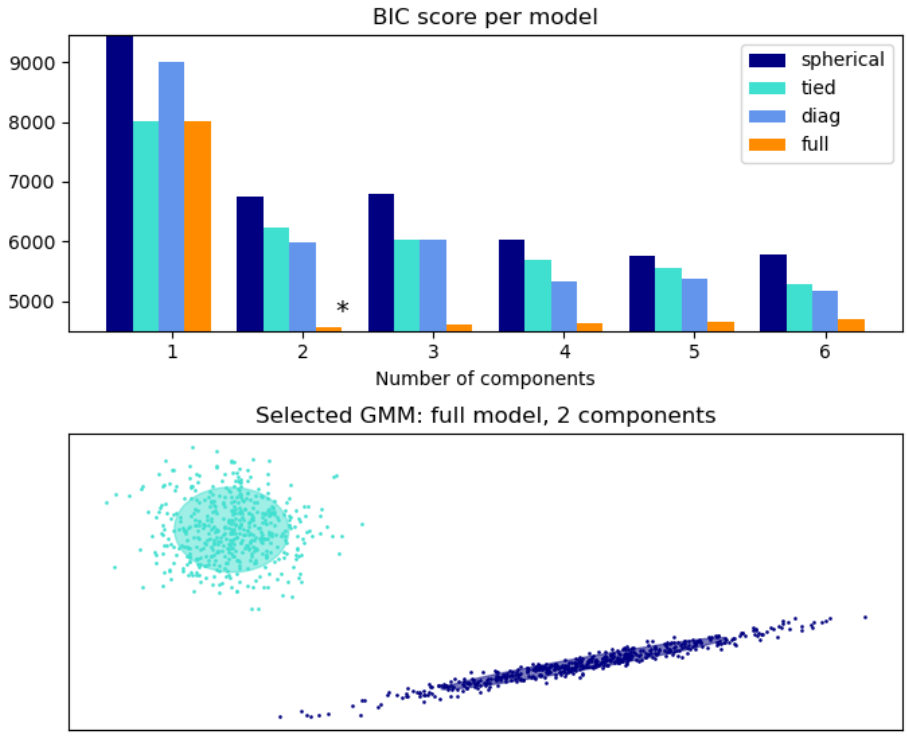
  (3) `covariance_type='diag', each component has its own diagonal covariance matrix`，每个高斯分布的轴的方向是固定的（平行于坐标轴方向），但是轴长不同；

  (4) `covariance_type='spherical', each component has its own single variance`，每个高斯分布的轴的方向是固定的（平行于坐标轴方向），轴长不同，形状是一个圆；

实验二

参考代码：https://scikit-learn.org/stable/auto_examples/mixture/plot_gmm_selection.html#sphx-glr-auto-examples-mixture-plot-gmm-selection-py

- 代码结果：



- 代码中使用 **BIC 准则**来挑选模型；

源码分析

由于 `covariance_type` 存在四种选择，理解起来比较复杂，这里只对 `covariance_type='full'` 进行分析；另外需要**注意**的一个点是，计算过程都是在 `ln` 域中进行的；

- 协方差矩阵的优化计算：计算过程中，需要求协方差矩阵的逆，直接求的复杂度很高，所以优化计算；（参考一下 链接）

    1. 求出协方差矩阵的 `cholesky` 分解；
    2. 对分解得到的下三角矩阵求逆；
    3. 保存求逆的结果 `precisions_cholesky_`；

```
"""
precisions_cholesky_ : array-like
    The cholesky decomposition of the precision matrices of each mixture
    component. A precision matrix is the inverse of a covariance matrix.
    A covariance matrix is symmetric positive definite so the mixture of
    Gaussian can be equivalently parameterized by the precision matrices.
    Storing the precision matrices instead of the covariance matrices makes
    it more efficient to compute the log-likelihood of new samples at test
    time. The shape depends on `covariance_type`
"""
```

- E-Step：具体的实现过程涉及到 `_estimate_log_gaussian_prob`，`_estimate_log_weights`，`_estimate_log_prob_resp` 等函数。
    - 计算 $(x-\mu)^T\Sigma^{-1}(x-\mu)$：这里就用到了上面的 `precisions_cholesky_`；

```
log_prob = np.empty((n_samples, n_components))
for k, (mu, prec_chol) in enumerate(zip(means, precisions_chol)):
    y = np.dot(X, prec_chol) - np.dot(mu, prec_chol)
    log_prob[:, k] = np.sum(np.square(y), axis=1)
```

    - 计算 $-\frac{1}{2}\cdot\ln|\Sigma|$：这里我们要对这个公式转换一下 $\frac{1}{2}\cdot\ln\frac{1}{|\Sigma|}$，根据 性质 1得到 $\frac{1}{2}\cdot\ln|\Sigma^{-1}|$，根据 性质 9 得到 $\ln|L|$，这里的 L 指的是 `precisions_cholesky_`（是一个三角矩阵），三角矩阵的行列式等于对角线的乘积，所以就是计算对角线元素 `ln` 值的累和，代码中也是这样来实现的；

```
# det(precision_chol) is half of det(precision)
log_det = _compute_log_det_cholesky(
    precisions_chol, covariance_type, n_features)
# 在函数_compute_log_det_cholesky中这样计算
n_components, _, _ = matrix_chol.shape
log_det_chol = (np.sum(np.log(
    matrix_chol.reshape(
        n_components, -1)[:, ::n_features + 1]), 1))
```

    - 计算 $\ln f_m(x_j|\mu_k^{(t)},\delta_k^{(t)})$：把前面两项和 一个常数项 累和即可；

```
-.5 * (n_features * np.log(2 * np.pi) + log_prob) + log_det
```

    - 计算 $\ln\gamma_{j,k}^{(t+1)}$：因为前面计算都是在 $\ln$ 域的，所以要先求 $\exp$，累和后再计算 $\ln$；

```
weighted_log_prob = self._estimate_weighted_log_prob(X)
log_prob_norm = logsumexp(weighted_log_prob, axis=1)
log_resp = weighted_log_prob - log_prob_norm[:, np.newaxis]
```

至此 E-Step 的工作结束。

- M-Step：具体过程涉及到 `_estimate_gaussian_parameters`，`_estimate_gaussian_covariances_full` 等函数。
    - 计算 $\phi_k^{(t+1)}$：这里需要注意的是，前面 E-Step 的计算都是 $\ln$ 域的，所以在 M-Step 中直接 $\exp$ 计算；

```
# 函数调用的时候直接exp
self.weights_, self.means_, self.covariances_ =
_estimate_gaussian_parameters(X, np.exp(log_resp), self.reg_covar,
self.covariance_type)
# 计算每类高斯分布的样本数量，函数调用外层会除以样本的总数
nk = resp.sum(axis=0) + 10 * np.finfo(resp.dtype).eps
```

    - 计算 $\mu_k^{(t+1)}$ 和 $\Sigma_k^{(t+1)}$：均值、方差的计算很清晰，其中 `reg_covar=1e-6` 是用来保证协方差矩阵恒大于 0 的；

```
# 计算均值
means = np.dot(resp.T, X) / nk[:, np.newaxis]
# 计算方差
def _estimate_gaussian_covariances_full(resp, X, nk, means,
reg_covar):
    """注释详见源码"""
    n_components, n_features = means.shape
    covariances = np.empty((n_components, n_features, n_features))
    for k in range(n_components):
        diff = X - means[k]
        covariances[k] = np.dot(resp[:, k] * diff.T, diff) / nk[k]
        covariances[k].flat[::n_features + 1] += reg_covar
    return covariances
# 因为 covariance_type 有四种类型，所以他的调用写的还是比较有意思的
covariances = {"full": _estimate_gaussian_covariances_full,
               "tied": _estimate_gaussian_covariances_tied,
               "diag": _estimate_gaussian_covariances_diag,
               "spherical": _estimate_gaussian_covariances_spherical
               }[covariance_type](resp, X, nk, means, reg_covar)
```

    - 计算 `precisions_cholesky_`：更新的时候不能把它忘了；

```
self.precisions_cholesky_ =
_compute_precision_cholesky(self.covariances_, self.covariance_type)
```

  至此 M-Step 的工作结束。

- 最优模型的获取：参数 `n_init` 可以指定代码拟合多个随机初始化的模型，挑选一个最优的模型；

```
for init in range(n_init): # 拟合多个模型
    if do_init:
        self._initialize_parameters(X, random_state)  # 初始化参数
    lower_bound = (-np.infty if do_init else self.lower_bound_)
    for n_iter in range(1, self.max_iter + 1):
        prev_lower_bound = lower_bound
        log_prob_norm, log_resp = self._e_step(X)  # e-step
        self._m_step(X, log_resp)  # m-step
        lower_bound = self._compute_lower_bound(
```

```
            log_resp, log_prob_norm)
        change = lower_bound - prev_lower_bound
        if abs(change) < self.tol:  # 判断是否收敛
            self.converged_ = True
            break
    self._print_verbose_msg_init_end(lower_bound)
    if lower_bound > max_lower_bound:  # 判断是否为最优模型
        max_lower_bound = lower_bound
        best_params = self._get_parameters()
        best_n_iter = n_iter
```

- 模型参数初始化：参数初始化有 **随机数** 和 `k-means` 两种方法；

```
def _initialize_parameters(self, X, random_state):
    n_samples, _ = X.shape
    if self.init_params == 'kmeans':
        resp = np.zeros((n_samples, self.n_components))
        label = cluster.KMeans(n_clusters=self.n_components,
n_init=1,

random_state=random_state).fit(X).labels_
        resp[np.arange(n_samples), label] = 1
    elif self.init_params == 'random':
        resp = random_state.rand(n_samples, self.n_components)
        resp /= resp.sum(axis=1)[:, np.newaxis]
    else:
        raise ValueError("Unimplemented initialization method '%s'"
                         % self.init_params)

    self._initialize(X, resp)
```

- 模型是否收敛：前面计算过程中，已经得到 `log_prob_norm` 这个参数（指的就是 ln 域的概率），所以计算 Log-Likelihood （在程序中用 `lower_bound` 参数来指代）的时候只需要求和即可，这里我们直接求均值

```
def _compute_lower_bound(self, _, log_prob_norm):
    return log_prob_norm
```

在函数 `fit_predict` 函数中判断是否收敛

```
for n_iter in range(1, self.max_iter + 1):
    prev_lower_bound = lower_bound
    log_prob_norm, log_resp = self._e_step(X)  # e-step
    self._m_step(X, log_resp)  # m-step
    lower_bound = self._compute_lower_bound(  # 计算对数似然概率
        log_resp, log_prob_norm)
    change = lower_bound - prev_lower_bound
    self._print_verbose_msg_iter_end(n_iter, change)
    if abs(change) < self.tol:  # 判断收敛
        self.converged_ = True
        break
```

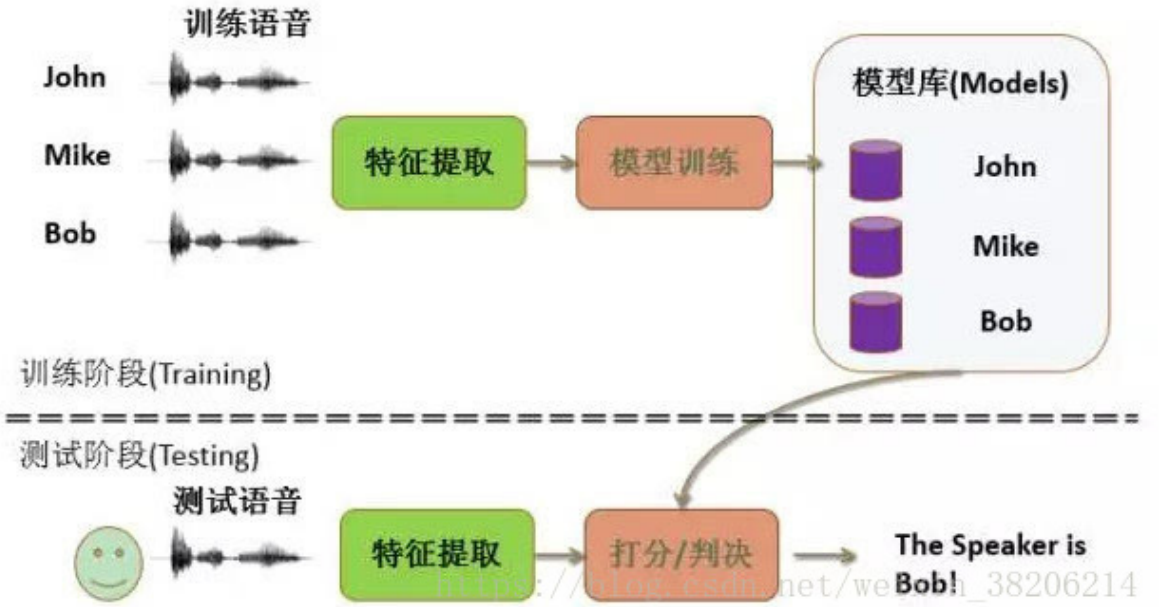
Links

- 参考链接 1：一文详解高斯混合模型原理
- 参考链接 2：概率笔记12——多维正态分布的最大似然估计
- 参考链接 3：高斯混合模型（GMM）
- 参考链接 4：GMM covariances
- 参考链接 5：矩阵分解——三角分解（Cholesky 分解）
- 参考链接 6：线性代数之——行列式及其性质
- 参考链接 7：三角矩阵

GMM-UBM 模型

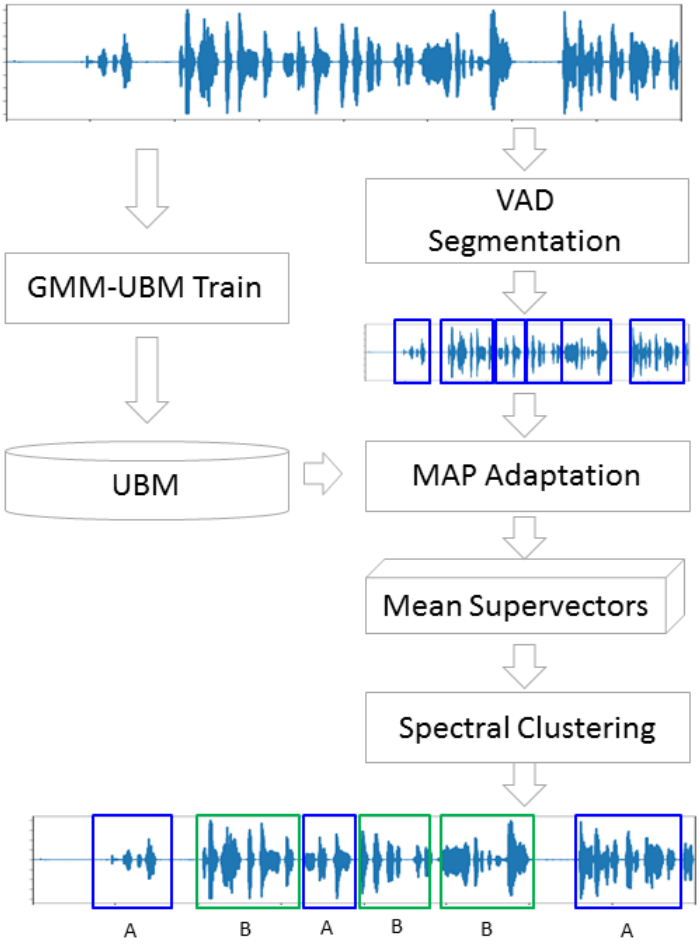
Notes

算法背景



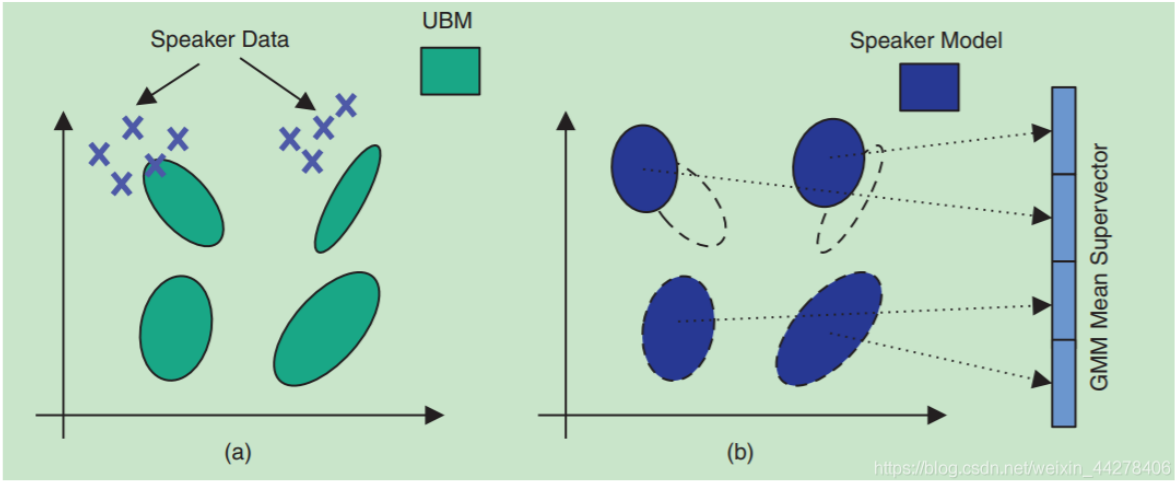
GMM 模型将空间分布的概率密度用**多个高斯概率密度函数的加权**来拟合，可以平滑地逼近任意形状地概率密度函数，并且是一个易于处理地参数模型，具备对实际数据极强额表征力。当 GMM 的规模越来越庞大，表征力越强，为了训练模型的参数，**需要的数据量也就越多**，但是**目标说话人的数据恰恰是不容易获得的**；

算法整体流程



GMM-UBM 模型的目标就是解决这个问题。首先，使用大量的背景数据（**非目标说话人的语料**）训练一个通用背景模型（Universal Background Model，是一个 GMM 模型）；接着使用少量的背景数据（**目标说话人的语料**）和最大后验估计 MAP 算法，调整 UBM 各个高斯分布的均值，生成目标用户的 GMM 模型；

最大后验估计 MAP 算法



（算法的原理在这里不再探究，我们直接来看这个算法是如何更新 UBM 的参数的）现在我们假设有少量的目标说话人的训练语料 $\mathcal{X} = \{x_1, x_2, x_3, \ldots, x_T\}$，和一个已经通过大量的非目标说话人语料训练得到的 UBM 模型：

$$\mathrm{UBM}(\phi, \mu, \delta) = \sum_{i=1}^{K} \phi_i \cdot f_m(x|\mu_i, \delta_i)$$

- **Step 1**：计算每段训练语料属于第 k 个高斯混合分布的概率；

$$\Pr(k|x_t)^{(t+1)} = \frac{\phi_k^{(t)} \cdot f_m(x|\mu_k^{(t)}, \delta_k^{(t)})}{\sum_{i=1}^{K} \phi_i^{(t)} \cdot f_m(x|\mu_i^{(t)}, \delta_i^{(t)})}$$

- **Step 2**：估计每个高斯分布的加权数量值、均值和方差；（前两步的做法是和 EM 算法的 E-Step 是一样的）

$$\begin{aligned}
\mathrm{N}_i^{(t+1)} &= \sum_{t=1}^{T} \Pr(k|x_t)^{(t+1)} \\
\mathrm{E}_i(\mathcal{X})^{(t+1)} &= \frac{1}{n} \cdot \left(\sum_{t=1}^{T} \Pr(k|x_t)^{(t+1)} \cdot x_t \right) \\
\mathrm{E}_i(\mathcal{X}^2)^{(t+1)} &= \frac{1}{n} \cdot \left(\sum_{t=1}^{T} \Pr(k|x_t)^{(t+1)} \cdot x_t^2 \right)
\end{aligned}$$

- **Step 3**：更新参数；

$$\begin{aligned}
\phi_i^{(t+1)} &= \gamma \cdot \left[\alpha_i^w \cdot \frac{\mathrm{N}_i^{(t+1)}}{T} + (1 - \alpha_i^w) \cdot \phi_i^{(t)} \right] \\
\mu_i^{(t+1)} &= \alpha_i^m \cdot \mathrm{E}_i(\mathcal{X})^{(t+1)} + (1 - \alpha_i^m) \cdot \mu_i^{(t)} \\
\delta_i^{(t+1)} &= \alpha_i^v \cdot \mathrm{E}_i(\mathcal{X}^2)^{(t+1)} + (1 - \alpha_i^v) \cdot \left((\delta_i^{(t)} + (\mu_i^{(t)})^2) - (\mu_i^{(t+1)})^2 \right)
\end{aligned}$$

其中，γ 为权重项的归一化因子；α_i^ρ，$(\rho \in \{w, m, v\})$ 是自适应系数，用来控制新 / 老估计量之间的平衡，其公式定义为：

$$\alpha_i^\rho = \frac{\mathrm{N}_i}{\mathrm{N}_i + r^\rho}$$

r^ρ 是一个固定的相关因子。在 GMM-UBM 系统中，通常会使用相同的 α_i^ρ 来更新参数。实验表明，r^ρ 的取值范围为 $[8, 20]$ 最有效，且**自适应过程只更新均值效果最佳**，即 $\alpha_i^w = \alpha_i^v = 0$；

- **Step 4**：重复前面的步骤，直至模型收敛；
- **说话人识别**：识别说话人的方法有以下两种
    - 可以直接遍历不同说话人的 GMM 模型，计算当前语音片段在各个 GMM 下的 $\ln$ 域的似然估计，哪个似然估计值高，则当前语音片段属于哪个 GMM，即属于某个说话人；
    - 另外，可以使用对数似然比的方法，计算 $\Lambda(x)$ 大于某阈值，则认为 x 属于说话人 spk；

$$\Lambda(x) = \log p(x|\mathrm{GMM}_{spk}) - \log p(x|\mathrm{UBM})$$

Codes

因为 `MAP Adaptation` 算法和 GMM 的训练过程基本相似，我们就来简单看看参考代码的整体结构：

- 计算 MFCC：脚本 `extract_mfcc_conefficient.py` 将语音转换成 13 维的 MFCC 特征；
- 训练 GMM-UBM 模型：脚本 `UBM.py` 用来训练 UBM 模型，并调用 MAP 算法（原先我一直以为说话人的识别是以一段语音去拟合其分布的，但是看了这块代码才发现是以**每一帧的 MFCC 去拟合高斯混合分布**的）；

- MAP 算法：脚本 `MAP_adapt.py` 主要用来实现 MAP 算法；
- 说话人识别：脚本 `testing_model.py` 主要用来测试音频为目标说话人的概率；

代码里面我有几个疑问：

- 算法里面没有使用 `VAD` 算法，这个可能导致误差；
- 最后给定一段语音，可以得到每帧的概率，那最终如何得到这个语音是否属于目标说话人呢；

另外，代码里面有一处错误，这个错误导致了数组的越界问题，我已经在 `issue` 中给出了解决方案；

Links

- 参考链接：声纹识别之GMM-UBM系统框架简介
- 参考代码：speaker_recognition_GMM_UBM
- 代码 `Bug`：I totally got MFCC 3380 lines, but it errors, Wouly you help me,thank you very much?

基于 i-vector 和 PLDA 的说话人识别技术

Notes

Links

- 参考：《Kaldi 语音识别实践》基于 i-vector 和 PLDA 的说话人识别技术

Deep speech: Scaling up end-to-end speech recognition

Contribution

Notes

代码理解

Links

- 论文链接：Hannun A, Case C, Casper J, et al. Deep speech: Scaling up end-to-end speech recognition[J]. arXiv preprint arXiv:1412.5567, 2014.
-

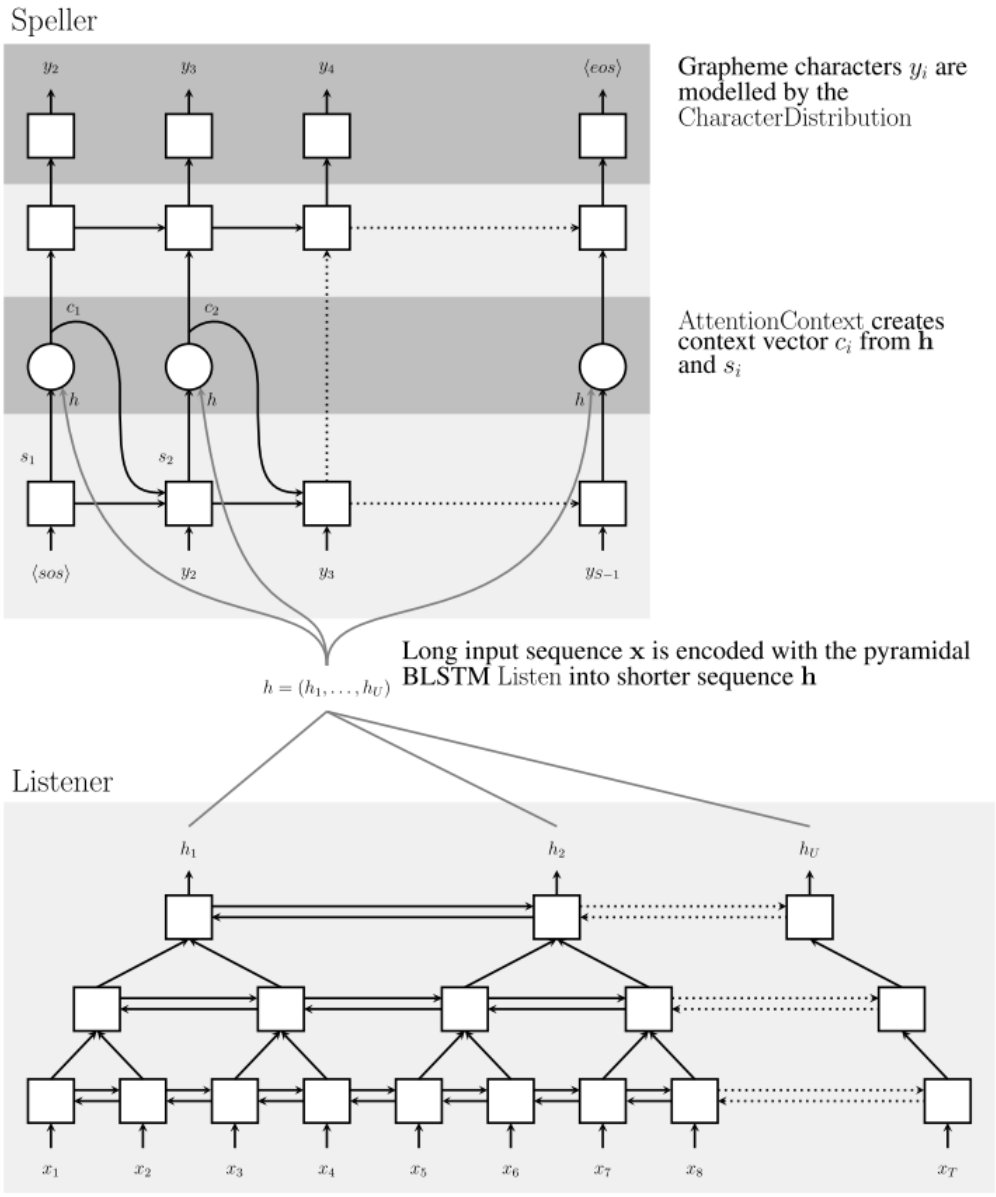
Listen, Attend and Spell

Contribution

1. 提出了一种**新的端到端**的语音识别模型 LAS；

Notes

1. **模型架构**：分成两个模块，一个是 Listen Encoder 模块，从语音时序序列中提取出高维特征，采用 pBLSTM (pyramid BLSTM) 的架构；另一个是 Attend and Spell 模块，从语音高维特征中输出单词，采用 Attention + LSTM 架构。架构图如下：（直接从这个图看的话，感觉模型是比较简洁的，但是真的从代码层面再来画一个细粒度的架构图，其实复杂得多。）



2. **Listen Encoder 模块**，使用 pBLSTM 的架构，每层在时间维度上减少一倍，带来的优点有两个：

   (1) 减少模型的复杂性（一定程度上是比较合理的，因为语音的前后帧之间有非常多的冗余信息）；

   (2) 加快模型的拟合速度（作者发现直接用 BLSTM 的话，用一个月的时间训练都没有办法得到好的结果）；

形式化的公式为：（和代码结合来看：关注公式 $[h_{2i}^{j-1}, h_{2i+1}^{j-1}]$ 部分，在程序的实现中，首先通过 设置 LSTM的特征维度 将**特征逐层降维**，然后通过合并前后帧的特征 对**时间降维**（而特征维度则升高））

$$h_i^j = \text{pBLSTM}(h_{i-1}^j, \left[h_{2i}^{j-1}, h_{2i+1}^{j-1}\right])$$

3. **Attend and Spell 模块**，该模块采用 2 层 LSTM 单元来记忆、更新模型的状态 s (**模型的状态包括 LSTM 的状态和 Attention 上下文状态**)：

$$\begin{aligned} c_i &= \text{AttentionContext}(s_i, \mathbf{h}) \\ s_i &= \text{RNN}(s_{i-1}, y_{i-1}, c_{i-1}) \\ P(y_i|\mathbf{x}, y_{<i}) &= \text{CharacterDistribution}(s_i, c_i) \end{aligned}$$

(1) Attention 单元：根据当前的状态 s_i （**在代码中，s_i 指的是第二层 LSTM 单元的输出**）从语音特征 h 中分离出“当前模型关心的”上下文信息 c_i；

(2) LSTM 单元：根据前一时刻的状态 s_{i-1} （**在代码中，这个 s_{i-1} 指的是第二层 LSTM 单元的状态**）、前一时刻输出的字符 y_{i-1} 和前一时刻的上下文信息 c_{i-1} 来更新产生当前时刻的状态 s_i；

(3) MLP 单元：根据当前状态 s_i 和上下文信息 c_i 计算得到最可能的字符 y_i；

另外， **Attention 单元在模型中的具体实现**：将模型状态 s_i 和语音特征 h 分别经过两个不同的 MLP 模型，计算出一个标量能量 (点积) e ，经过 softmax 层归一化后作为权重向量，和原来的特征 h 加权生成上下文信息 c_i。形式化的公式如下：

$$\begin{aligned} e_{i,u} &= \langle \phi(s_i), \psi(h_u) \rangle \\ \alpha_{i,u} &= \frac{\exp(e_{i,u})}{\sum_u \exp(e_{i,u})} \\ c_i &= \sum_u \alpha_{i,u} h_u \end{aligned}$$

4. **Learning**. 模型的目标是，在给定 **全部** 语音信号和 **上文** 解码结果的情况下，模型输出正确字符的概率最大。形式化的公式如下：

$$\max_\theta \sum_i \log P(y_i|\mathbf{x}, y_{<i}^*; \theta)$$

在训练的时候，我们给的 y 都是 ground truth，但是解码的时候，模型不一定每个时间片都会产生正确的标签。虽然模型对于这种错误是具有宽容度，单训练的时候可以增加 **trick**：以 **10%** 的概率从前一个解码结果中挑选 (根据前一次的概率分布) 一个标签作为 ground truth 进行训练。形式化公式如下：

$$\begin{aligned} \tilde{y}_i &\sim \text{CharacterDistribution}(s_i, c_i) \\ \max_\theta \sum_i \log P(y_i|\mathbf{x}, \tilde{y}_{<i}; \theta) \end{aligned}$$

另外，作者发现预训练 (主要是预训练 Listen Encoder 部分) 对 LAS 模型没有作用。

5. **Decoding & Rescoring**. 解码的时候使用 Beam-Search 算法，目标是希望得到概率最大的字符串。形式化公式如下：

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log P(\mathbf{y}|\mathbf{x})$$

可以用语言模型对最后一轮 Beam-Search 的结果进行重打分，形式化公式如下：

$$s(\mathbf{y}|\mathbf{x}) = \frac{\log P(\mathbf{y}|\mathbf{x})}{|\mathbf{y}|_c} + \lambda \log P_{\text{LM}}(\mathbf{y})$$

增加解码结果的长度项 |y| 来**平衡产生长句、短句的权重**，另外语言模型的权重 lambda 可以通过验证集数据来确定。

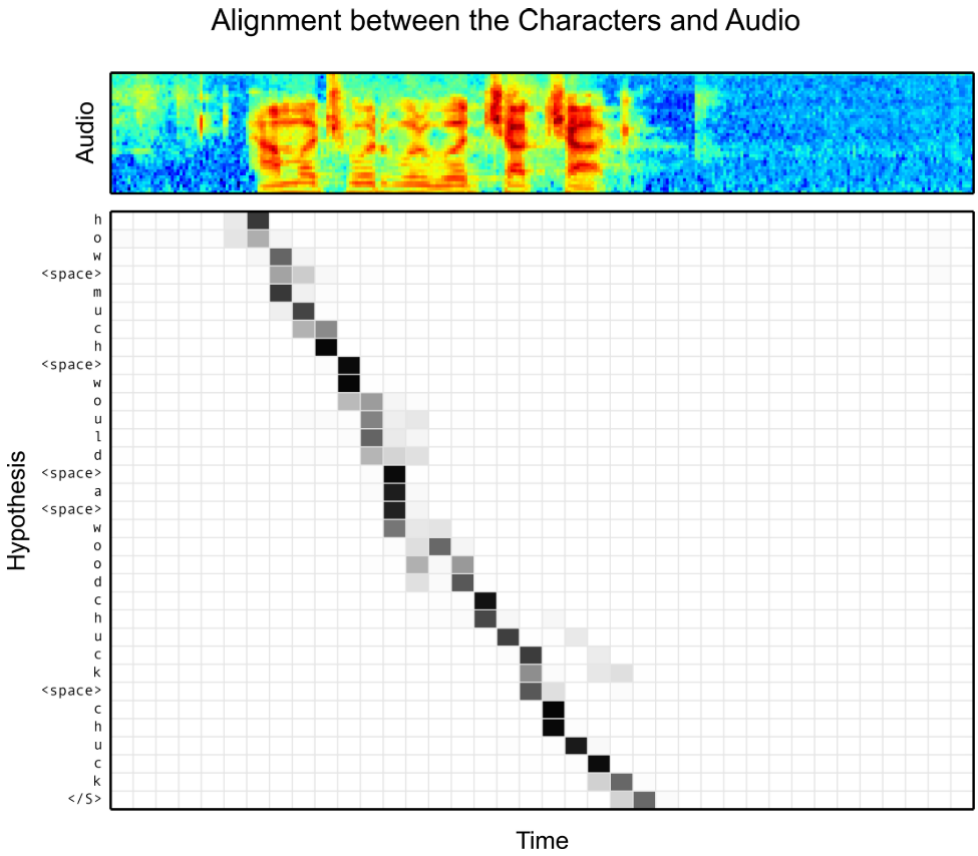
6. 实验结果：

(1) 使用 log-mel filter bank 特征

(2) 整体对比，LAS 刚出来的时候并打不过传统的 DNN-HMM 模型；

Model	Clean WER	Noisy WER
CLDNN-HMM [20]	8.0	8.9
LAS	16.2	19.0
LAS + LM Rescoring	12.6	14.7
LAS + Sampling	14.1	16.5
LAS + Sampling + LM Rescoring	10.3	12.0

(3) Attention 模块确实更加关注对应时间片段的特征；



(4) 模型对于较短的语句或者较长的语句效果都不是很好；

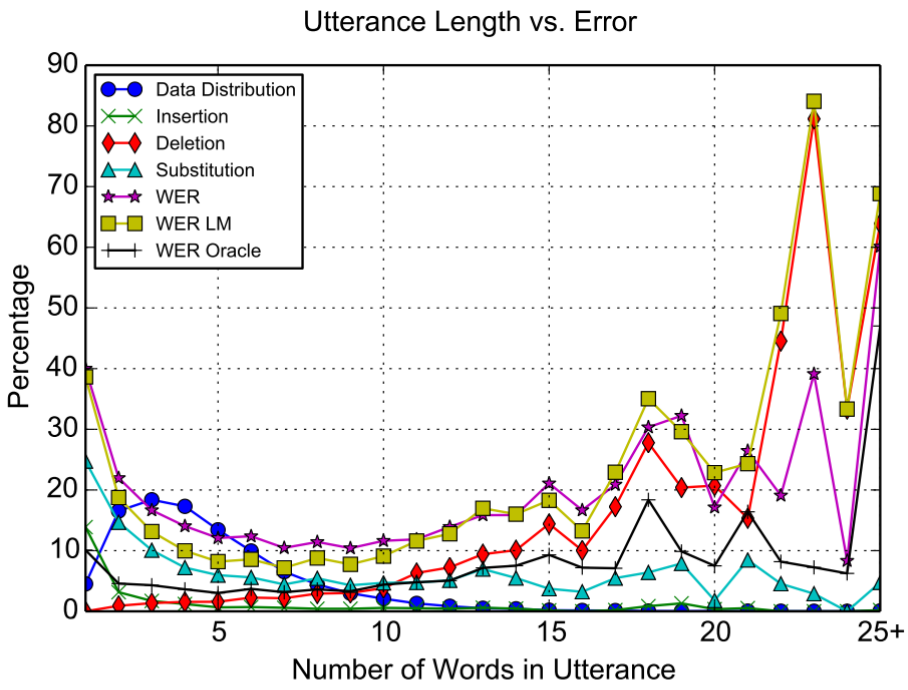


Figure 4: The correlation between error rates (insertion, deletion, substitution and WER) and the number of words in an utterance. The WER is reported without a dictionary or language model, with language model rescoring and the oracle WER for the clean Google voice search task. The data distribution with respect to the number of words in an utterance is overlaid in the figure. LAS performs poorly with short utterances despite an abundance of data. LAS also fails to generalize well on longer utterances when trained on a distribution of shorter utterances. Insertions and substitutions are the main sources of errors for short utterances, while deletions dominate the error for long utterances.

Shortcoming

1. 必须要得到整个语音后才能解码，限制了模型的流式处理能力；
2. Attention 机制需要消耗大量的计算量；
3. 输入长度对于模型的影响较大；

代码理解

Tensorflow 2 （Keras）实现

> 这个库只实现了 LAS 模型部分，没有完整的预处理等过程，故先通过这个库来简单学习下 LAS 模型原理以及 Tensorflow 的使用，期待一下库作者的更新；

(1) 整体框架：

```
def LAS(dim, f_1, no_tokens): # dim-神经网络内特征维度，f_1-输入特征维度，no_tokens-分类维度
  input_1 = tf.keras.Input(shape=(None, f_1))  # shape: (..., None, f_1)
  input_2 = tf.keras.Input(shape=(None, no_tokens))  # shape: (..., None, no_tokens)

  #Listen; Lower resoultion by 8x
  x = pBLSTM( dim//2 )(input_1) # (..., audio_len//2, dim*2)
  x = pBLSTM( dim//2 )(x) # (..., audio_len//4, dim*2)
  x = pBLSTM( dim//4 )(x) # (..., audio_len//8, dim)

  #Attend
  x = tf.keras.layers.RNN(att_rnn(dim), return_sequences=True)(input_2, constants=x) # (..., seq_len, dim*2)

  #Spell
  x = tf.keras.layers.Dense(dim, activation="relu")(x) # (..., seq_len, dim)
```

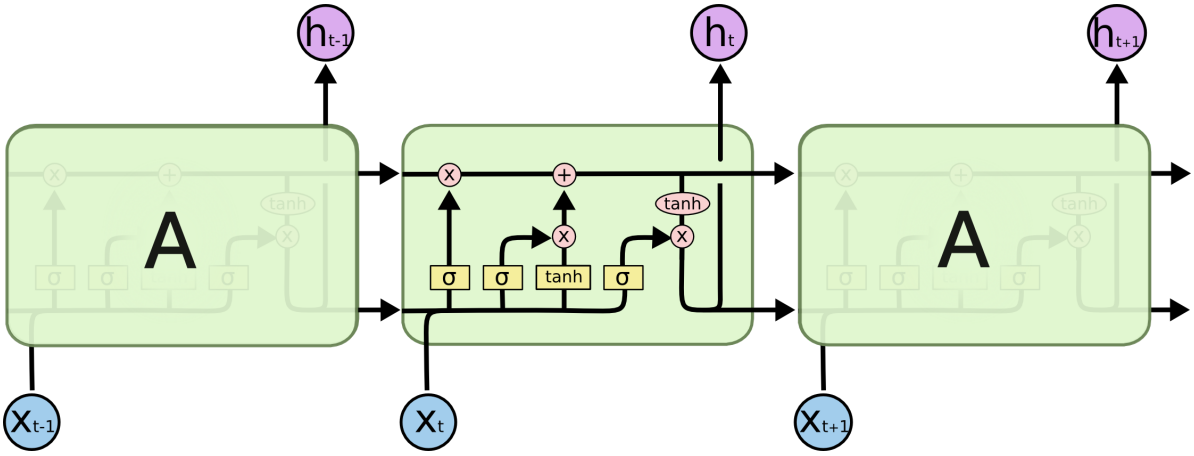
```
  x = tf.keras.layers.Dense(no_tokens, activation="softmax")(x) # (..., seq_len, no_tokens)

  model = tf.keras.Model(inputs=[input_1, input_2], outputs=x)
  return model
```

(2) Listen 模块：使用 3 层 `pBLSTM` 实现，其中需要注意的是 `tf.keras.layers.Bidirectional` 的使用（我一开始判断错了输出的维度）

```
class pBLSTM(tf.keras.layers.Layer):
  def __init__(self, dim):
    super(pBLSTM, self).__init__()

    self.dim        = dim
    self.LSTM       = tf.keras.layers.LSTM(self.dim, return_sequences=True)
    self.bidi_LSTM  = tf.keras.layers.Bidirectional(self.LSTM)

  @tf.function
  def call(self, inputs):
    y = self.bidi_LSTM(inputs) # (..., seq_len, dim*2)

    if tf.shape(inputs)[1] % 2 == 1:
      y = tf.keras.layers.ZeroPadding1D(padding=(0, 1))(y)

    y = tf.keras.layers.Reshape(target_shape=(-1, int(self.dim*4)))(y) # (..., seq_len//2, dim*4)
    return y
```

(3) Attend 模块：

- 如果对 LSTM 不太熟悉的话，结合 LSTM 的结构图一起来看代码会轻松一点：



- 双层 LSTM 代码如下：使用 2 层 LSTM 模型来存储模型的状态；

```
class att_rnn( tf.keras.layers.Layer):
  def __init__(self, units,):
    super(att_rnn, self).__init__()
    self.units      = units
    self.state_size = [self.units, self.units]

    self.attention_context  = attention(self.units)
    self.rnn                = tf.keras.layers.LSTMCell(self.units)  # LSTM 1，用来记忆模型的状态
```

```
    self.rnn2                = tf.keras.layers.LSTMCell(self.units)  # LSTM 2，用来记忆模型的状态

  def call(self, inputs, states, constants):
    h       = tf.squeeze(constants, axis=0)  # 删除为1的维度，shape: (..., seq_len, F)

    s       = self.rnn(inputs=inputs, states=states) # [(..., F), [(..., F), (..., F)]]
    s       = self.rnn2(inputs=s[0], states=s[1])[1] # [(..., F), (..., F)]

    c       = self.attention_context([s[0], h]) # (..., F)
    out     = tf.keras.layers.concatenate([s[0], c], axis=-1) # (..., F*2)

    return out, [c, s[1]]
```

- Attention 代码如下：全连接层（变换维度） -> 向量点积（计算权重） -> softmax（权重归一化） -> 得到重要的上下文信息；

```
class attention(tf.keras.layers.Layer):  # Attention 类，用来计算上下文的权重
  def __init__(self, dim):
    super(attention, self).__init__()

    self.dim      = dim
    self.dense_s  = tf.keras.layers.Dense(self.dim)
    self.dense_h  = tf.keras.layers.Dense(self.dim)

  def call(self, inputs):
    # Split inputs into attentions vectors and inputs from the LSTM output
    s     = inputs[0] # (..., depth_s)
    h     = inputs[1] # (..., seq_len, depth_h)

    # Linear FC
    s_fi   = self.dense_s(s) # (..., F)
    h_psi  = self.dense_h(h) # (..., seq_len, F)

    # Linear blendning < ϕ(s_i), ψ(h_u) >
    # Forced seq_len of 1 since s should always be a single vector per batch
    e = tf.matmul(s_fi, h_psi, transpose_b=True) # (..., 1, seq_len)

    # Softmax vector
    alpha = tf.nn.softmax(e) # (..., 1, seq_len)

    # Context vector
    c = tf.matmul(alpha, h) # (..., 1, depth_h)
    c = tf.squeeze(c, 1) # (..., depth_h)

    return c
```

(4) Spell 模块：两个全连接层，输出最后的概率；

```
x = tf.keras.layers.Dense(dim, activation="relu")(x) # (..., seq_len, dim)
x = tf.keras.layers.Dense(no_tokens, activation="softmax")(x) # (..., seq_len, no_tokens)
```

Pytorch 实现

Links

- 论文链接：Listen, Attend and Spell
- LAS 模型缺点参考链接：LAS 语音识别框架发展简述
- Tensorflow 2 （Keras）实现：Listen, attend and spell
- Pytorch 实现：End-to-end-ASR-Pytorch (**暂未阅读代码**)
- LSTM 详解：Understanding LSTM Networks

Lingvo: a modular and scalable framework for sequence-to-sequence modeling

谷歌开源的基于tensorflow的序列模型框架。

Notes

Links

- 论文链接：Lingvo: a modular and scalable framework for sequence-to-sequence modeling
- Github：Lingvo