Lecture 16: A tutorial on hidden Markov models and selected applications in speech recognition, part 3

Mark Hasegawa-Johnson

ECE 537, Fall 2022

The Rest of the Article

Today, we'll talk about the rest of Rabiner's article.

- Section IV: Types of HMMs
- Section V: Implementation Issues for HMMs
- Section VI: Implementation of Speech Recognizers Using HMMs
- Section VII: Connected Word Recognition Using HMMs

Outline

Continuous Observation Density HMMs

- Vector-quantization of $\vec{o}_t$ loses information. If we use a continuous pdf, then we don't lose that information.
- The mixture Gaussian pdf is the most general pdf for which the Baum-Welch formula has been solved in closed form:

$$b_j(\vec{o}_t) = \sum_{m=1}^{M} c_{jm} \mathcal{N}(\vec{o}_t, \mu_{jk}, U_{jk})$$

$$c_{jm} \geq 0, \quad \sum_{m=1}^{M} c_{jm} = 1$$

- As $M \to \infty$, the GMM can represent **any** pdf with error $\to 0$.
- A neural net can also represent any pdf with error $\to 0$, but the Baum-Welch algorithm can't be solved in closed form; it must be solved using gradient descent.

Baum's Auxiliary for the CDHMM

The Baum-Welch algorithm iteratively improves $p(O|\lambda)$ by analytically solving for $\bar{\lambda}$:

$$\text{Variant 1:} \quad Q(\lambda, \bar{\lambda}) = \sum_{Q} P(Q|O, \lambda) \log P(O, Q|\bar{\lambda})$$

In reference [35], Rabiner, Juang, Levinson & Sondhi introduce the **mixture component indices** as a new variable, $M = [m_1, \ldots, m_T]$, and propose maximizing:

$$\text{Variant 2:} \quad Q(\lambda, \bar{\lambda}) = \sum_{Q,M} P(Q, M|O, \lambda) \log P(O, Q, M|\bar{\lambda})$$

In the homework, feel free to use either variant (published solutions will use variant 1, but variant 2 is more useful).

Baum-Welch Algorithm for CDHMM

Solution:

$$\bar{c}_{jm} = \frac{\sum_t \gamma_t(j,m)}{\sum_t \sum_k \gamma_t(j,k)} = \frac{\sum_t \gamma_t(j,m)}{\sum_t \gamma_t(j)}, \quad \bar{\mu}_{jm} = \frac{\sum_t \vec{o}_t \gamma_t(j,m)}{\sum_t \gamma_t(j,m)},$$

$$\bar{U}_{jm} = \frac{\sum_t \gamma_t(j,m)(\vec{o}_t - \mu_{jm})(\vec{o}_t - \mu_{jm})^T}{\sum_t \gamma_t(j,m)},$$

Variant 1:
$$\gamma_t(j,m) = P(q_t = j|O,\lambda)P(m_t = m|q_t = j,O,\bar{\lambda})$$
$$= \gamma_t(j)\left[\frac{\bar{c}_{jm}\mathcal{N}(\vec{o}_t,\bar{\mu}_{jm},\bar{U}_{jm})}{\sum_{k=1}^{M}\bar{c}_{jk}\mathcal{N}(\vec{o}_t,\bar{\mu}_{jk},\bar{U}_{jk})}\right]$$

Variant 2:
$$\gamma_t(j,m) = P(q_t = j, m_t = m|O,\lambda)$$
$$= \gamma_t(j)\left[\frac{c_{jm}\mathcal{N}(\vec{o}_t,\mu_{jm},U_{jm})}{\sum_{k=1}^{M}c_{jk}\mathcal{N}(\vec{o}_t,\mu_{jk},U_{jk})}\right]$$

Re-estimating the covariance

$$\bar{U}_{jm} = \frac{\sum_{t=1}^{T} \gamma_t(j,m)(\vec{o}_t - \vec{\mu}_{jm})(\vec{o}_t - \vec{\mu}_{jm})^T}{\sum_{t=1}^{T} \gamma_t(j,m)}$$

Here's a bad thing that can happen:

- $\gamma_t(j,m)$ is nonzero for a number of frames that is smaller than the dimension of $\vec{o}$.
- Therefore, the formula above results in a singular-valued $\bar{U}_{jm}$. Thus $|\bar{U}_{jm}| = 0$, and $b_j(\vec{o}_t) = \infty$.

Writing Baum-Welch as a Matrix Equation

Let's re-write the M-step as a matrix equation. Define two new matrices, O and W:

$$O = [(\vec{o}_1 - \vec{\mu}_{jm}), (\vec{o}_2 - \vec{\mu}_{jm}), \cdots, (\vec{o}_T - \vec{\mu}_{jm})]$$

$$W = \begin{bmatrix} \frac{\gamma_1(j,m)}{\sum_{t=1}^{T} \gamma_t(j,m)} & 0 & \cdots & 0 \\ 0 & \frac{\gamma_2(j,m)}{\sum_{t=1}^{T} \gamma_t(j,m)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 00 & \cdots & \frac{\gamma_T(j,m)}{\sum_{t=1}^{T} \gamma_t(j,m)} & \end{bmatrix}$$

Writing Baum-Welch as a Matrix Equation

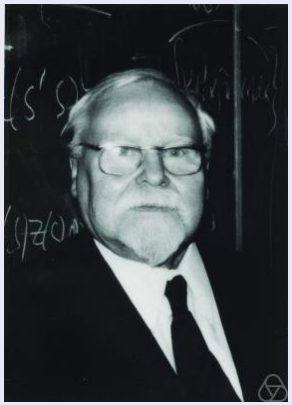
In terms of those two matrices, the Baum-Welch re-estimation formula is:

$$\bar{U}_{jm} = OWO^T$$

. . . and the problem we have is that OWO^T is singular, so that $(OWO^T)^{-1}$ is infinite.

Tikhonov Regularization

Andrey Tikhonov



Andrey Tikhonov studied ill-posed problems (problems in which we try to estimate more parameters than the number of data points, e.g., covariance matrix has more dimensions than the number of training tokens).

Tikhonov regularization

Tikhonov proposed a very simple solution that guarantees $\bar{U}_{jm}$ to be nonsingular:

$$\bar{U}_{jm} = OWO^T + \alpha I$$

. . . where I is the identity matrix, and α is a tunable hyperparameter called the "regularizer."

Outline

Autoregressive HMMs

- Suppose that the observation vector is composed of speech samples, $\vec{o}_t = [o_{t,1}, \ldots, o_{t,K}]^T$. Then we can use a mixture-LPC model,

$$b_j(\vec{o}_t) = \sum_{m=1}^{M} c_{jm} b_{jm}(\vec{o}_t)$$

$$b_{jm}(\vec{o}_t) = \frac{1}{\sqrt{(2\pi/K)^K}} e^{-\frac{K}{2}\left(\phi(0,0) - \vec{a}_{jm}^T \Phi \vec{a}_{jm}\right)},$$

- $\phi(0,0)$ and Φ are the energy and the autocovariance matrix, respectively, of the speech frame $\vec{o}_t$.
- Model parameters are now c_{jm} and $\vec{a}_{jm}$.

Outline

Tied States

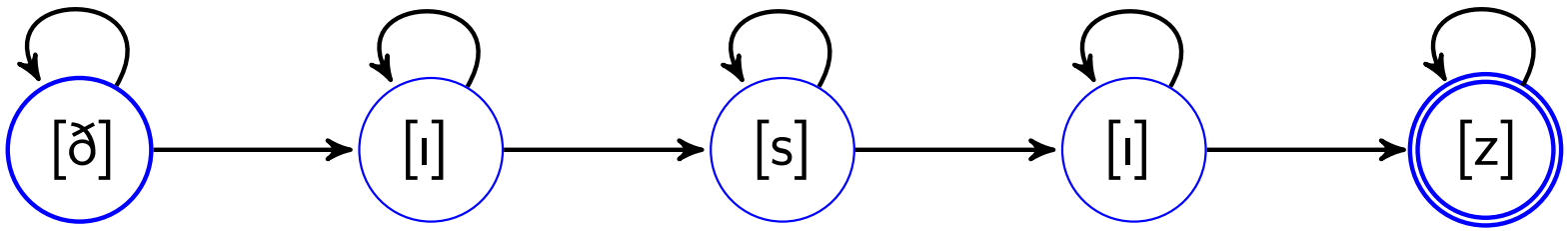
Different words share phonemes. For example, the words ⟨this⟩ and ⟨The⟩ share the same initial phoneme, which is [ð] ("eth") in the International Phonetic Alphabet (IPA). We can learn a better model of [ð] if we share relevant training data, pooled across all of the words that contain [ð]:

$$\bar{\mu}_{[ð],m} = \frac{\sum_{w:[ð]\in w} \sum_{j\sim[ð]} \gamma_{w,t}(j,m)\vec{o}_t}{\sum_{w:[ð]\in w} \sum_{j\sim[ð]} \gamma_{w,t}(j,m)},$$

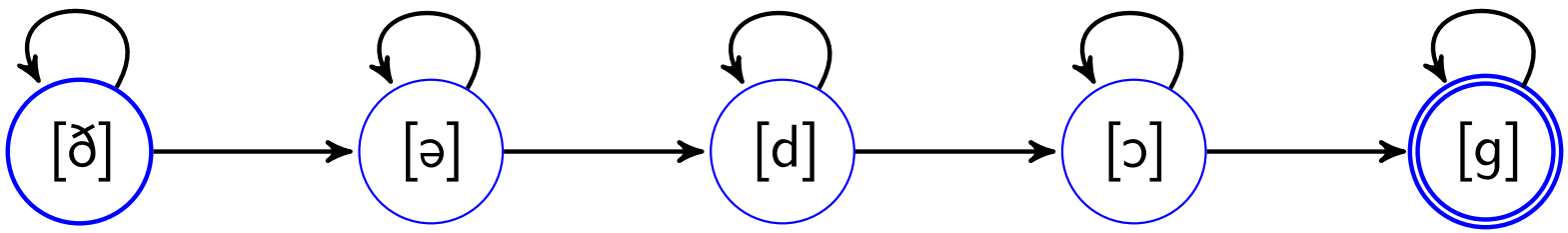
- $\gamma_{w,t}(j,m)$ is the probability of being in mixture component m of state j, given observation $\vec{o}_t$ in training waveform w.
- $w : [ð] \in w$ means "w is a waveform that contains the phoneme [ð]."
- $j \sim [ð]$ means "j is a state that represents the phoneme [ð] in waveform w."

In order to train a model with tied states, we first take the transcription of each training waveform, and convert it into an HMM, unique to that training waveform.

- For example, here is the training-time HMM for the utterance ⟨`this is`⟩. The two states marked [ɪ] are tied:



- . . . and here is the training-time HMM for the utterance ⟨`the dog`⟩. Its [ð] state is tied to the [ð] of ⟨`this is`⟩.



Null Transitions

During test time, we can combine many different words into a single graph if we introduce three new types of states:

- A **word-final** state is a state that specifies its phoneme, and also specifies which word it belongs to. For example, the last phoneme in the word $\langle \texttt{this} \rangle$ is

$$q_t = [\text{s}] : \langle \texttt{this} \rangle$$

- A **word-nonfinal** state is a state that specifies its phoneme, but does not know which word it belongs to. For example, the first phoneme in the word "this" is shared with the first phoneme in the word "the;" they are both

$$q_t = [\text{ð}] : \phi$$

- A **null transition** (or **non-emitting state**) is a state that does not generate any output. We denote it as

$$q = \phi : \phi$$

Null Transitions, a.k.a. Non-Emitting States

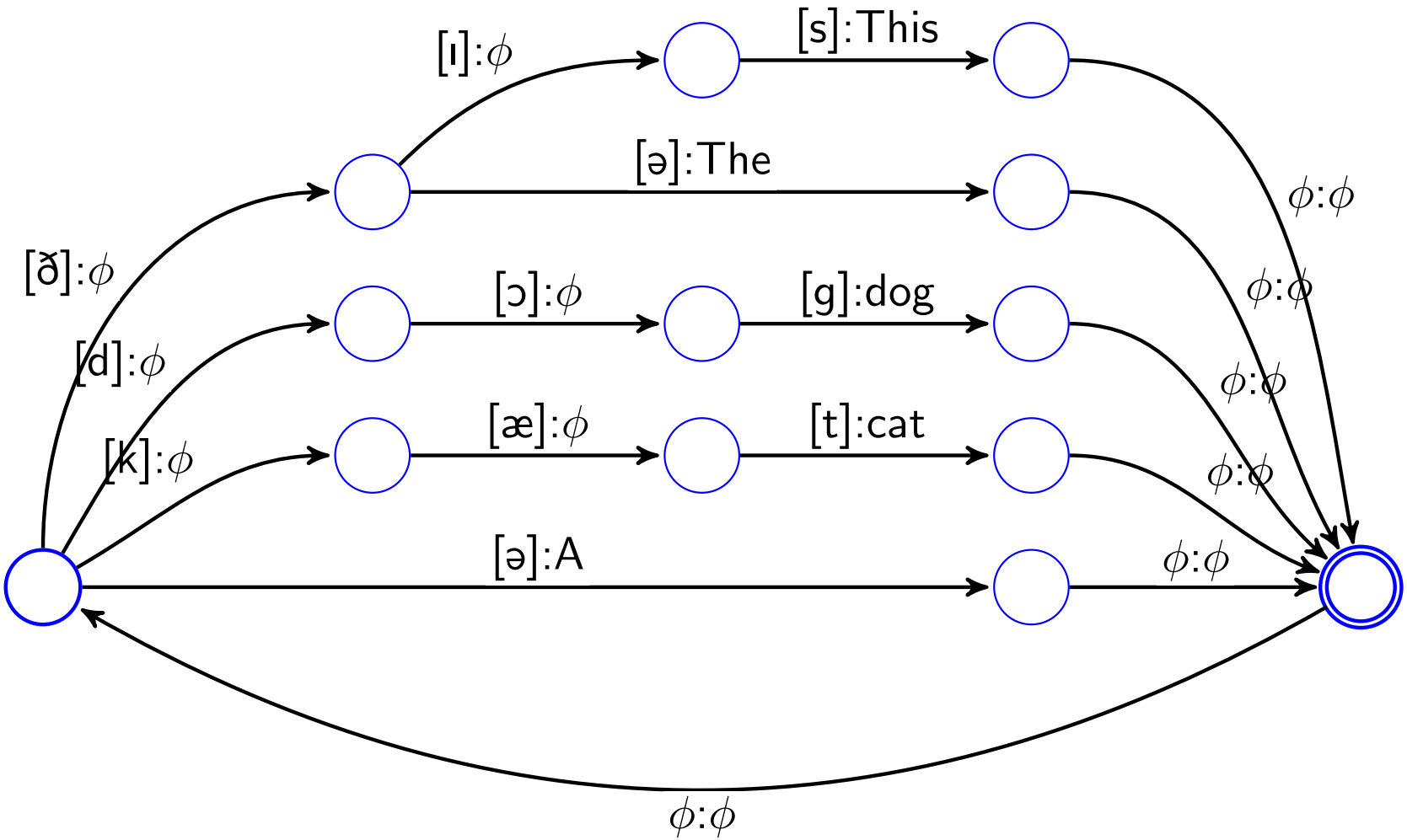
In the probability calculation, a non-emitting state does not emit any observation vector. For example, if states i and j are normal emitting states, but state ϕ is non-emitting, then

$$P\left(q_1 = i, \vec{o}_1, q_{1+} = \phi, q_2 = j, \vec{o}_2\right)$$

$$=$$

$$\pi_i b_i(\vec{o}_1) a_{i,\phi} a_{\phi,j} b_j(\vec{o}_2)$$

Example ASR Search Space



Outline

HMMs with Explicit Duration Probabilities

In a regular HMM, the probability of staying in state i for d frames is:

$$\text{Regular HMM:} \quad p_i(d) = (a_{ii})^{d-1} \sum_{j \neq i} a_{ij}$$

That's called a **geometric pmf**. A geometric pmf is a pretty bad model of real phoneme durations. You can learn a more realistic model if we redefine α to

Explicit Duration HMM:

$\alpha_t(i) = P(\vec{o}_1, \ldots, \vec{o}_t, \text{ and frame } t \text{ is the last frame of state } i)$,

which leads to the following re-estimation formua:

$$\alpha_t(j) = \sum_{i=1}^{N} \sum_{d=1}^{D} \alpha_{t-d}(i) a_{ij} p_j(d) \prod_{s=t-d+1}^{t} b_j(\vec{o}_s)$$

Outline

Regular HMM, Forward Algorithm

Here's the forward algorithm for the regular HMM. Definition: $\alpha_t(i) \equiv p(\vec{o}_1, \ldots, \vec{o}_t, q_t = i|\lambda)$. Computation:

1. **Initialize:**

$$\alpha_1(i) = \pi_i b_i(\vec{o}_1), \quad 1 \leq i \leq N$$

2. **Iterate:**

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(\vec{o}_t), \quad 1 \leq j \leq N, \ 2 \leq t \leq T$$

3. **Terminate:**

$$p(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

Numerical Issues

The forward algorithm is susceptible to massive floating-point underflow problems. Consider this equation:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(\vec{o}_t)$$

$$= \sum_{q_1=1}^{N} \cdots \sum_{q_{t-1}=1}^{N} \pi_{q_1} b_{q_1}(\vec{o}_1) \cdots a_{q_{t-1}q_t} b_{q_t}(\vec{o}_t)$$

First, suppose that $b_q(x)$ is discrete, with $k \in \{1, \ldots, K\}$. Suppose $K \approx 1000$ and $T \approx 100$, in that case, each $\alpha_t(j)$ is:

- The sum of N^T different terms, each of which is
- the product of T factors, each of which is
- the product of two probabilities: $a_{ij} \sim \frac{1}{N}$ times $b_j(x) \sim \frac{1}{K}$, so

$$\alpha_T(j) \approx N^T \left(\frac{1}{NK}\right)^T \approx \frac{1}{K^T} \approx 10^{-300}$$

The Solution: Scaling

The solution is to just re-scale $\alpha_t(j)$ at each time step, so it never gets really small. For example, we can normalize it so that $\sum_i \hat{\alpha}_t(i) = 1$. That gives us:

$$\hat{\alpha}_t(j) = \frac{\sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij} b_j(\vec{o}_t)}{\sum_{\ell=1}^{N} \sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{i\ell} b_\ell(\vec{o}_t)}$$

Now the problem is. . . if $\alpha_t(j)$ has been re-scaled, how do we perform recognition? Remember we used to have $p(O|\lambda) = \sum_i \alpha_t(i)$. How can we get $p(O|\lambda)$ now?

What exactly is alpha-hat?

Let's look at this in more detail. $\alpha_t(j)$ is defined to be $p(\vec{o}_1, \ldots, \vec{o}_t, q_t = j|\lambda)$. Let's define a "scaling term," c_t, equal to the denominator in the scaled forward algorithm. So, for example, at time $t = 1$ we have:

$$c_1 = \sum_{\ell=1}^{N} \alpha_1(\ell) = \sum_{\ell=1}^{N} p(\vec{o}_1, q_1 = \ell|\lambda) = p(\vec{o}_1|\lambda)$$

and therefore

$$\hat{\alpha}_1(i) = \frac{\alpha_1(i)}{c_1} = \frac{p(\vec{o}_1, q_1 = i|\lambda)}{p(\vec{o}_1|\lambda)} = p(q_1 = i|\vec{o}_1, \lambda)$$

What exactly is alpha-hat?

At time t, we need a new intermediate variable. Let's call it $\tilde{\alpha}_t(j)$:

$$\begin{aligned}
\tilde{\alpha}_t(j) &= \sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij} b_j(\vec{o}_t) \\
&= \sum_{i=1}^{N} p(q_{t-1} = i | \vec{o}_1, \ldots, \vec{o}_{t-1}, \lambda) p(q_t = j | q_{t-1} = i) p(\vec{o}_t | q_t = j) \\
&= p(q_t = j, \vec{o}_t | \vec{o}_1, \ldots, \vec{o}_{t-1}, \lambda) \\
c_t &= \sum_{\ell=1}^{N} \tilde{\alpha}_t(\ell) = p(\vec{o}_t | \vec{o}_1, \ldots, \vec{o}_{t-1}, \lambda)
\end{aligned}$$

$$\hat{\alpha}_t(j) = \frac{\tilde{\alpha}_t(j)}{c_t} = \frac{p(\vec{o}_t, q_t = j | \vec{o}_1, \ldots, \vec{o}_{t-1}, \lambda)}{p(\vec{o}_t | \vec{o}_1, \ldots, \vec{o}_{t-1}, \lambda)} = p(q_t = j | \vec{o}_1, \ldots, \vec{o}_t, \lambda)$$

Scaled Forward Algorithm: The Variables

So we have not just one, but three new variables:

1. The intermediate forward probability:

$$\tilde{\alpha}_t(j) = p(q_t = j, \vec{o}_t | \vec{o}_1, \ldots, \vec{o}_{t-1}, \lambda)$$

2. The scaling factor:

$$c_t = p(\vec{o}_t | \vec{o}_1, \ldots, \vec{o}_{t-1}, \lambda)$$

3. The scaled forward probability:

$$\hat{\alpha}_t(j) = p(q_t = j | \vec{o}_1, \ldots, \vec{o}_t, \lambda)$$

The Solution

The second of those variables is interesting because we want $p(O|\lambda)$, which we can now get from the c_ts—we no longer actually need the αs for this!

$$p(O|\lambda) = p(\vec{o}_1|\lambda)p(\vec{o}_2|\vec{o}_1,\lambda)p(\vec{o}_3|\vec{o}_1,\vec{o}_2,\lambda)\cdots = \prod_{t=1}^{T} c_t$$

But that's still not useful, because if each $c_t \sim 10^{-19}$, then multiplying them all together will result in floating point underflow. So instead, it is better to compute

$$\ln p(O|\lambda) = \sum_{t=1}^{T} \ln c_t$$

The Scaled Forward Algorithm

1. **Initialize:**

$$\hat{\alpha}_1(i) = \frac{1}{c_1}\pi_i b_i(\vec{o}_1)$$

2. **Iterate:**

$$\tilde{\alpha}_t(j) = \sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij} b_j(\vec{o}_t)$$

$$c_t = \sum_{j=1}^{N} \tilde{\alpha}_t(j)$$

$$\hat{\alpha}_t(j) = \frac{1}{c_t}\tilde{\alpha}_t(j)$$

3. **Terminate:**

$$\ln p(O|\lambda) = \sum_{t=1}^{T} \ln c_t$$

The Scaled Backward Algorithm

This can also be done for the backward algorithm:

1. **Initialize:**

$$\hat{\beta}_T(i) = 1, \quad 1 \leq i \leq N$$

2. **Iterate:**

$$\tilde{\beta}_t(i) = \sum_{j=1}^{N} a_{ij} b_j(\vec{o}_{t+1}) \hat{\beta}_{t+1}(j)$$

$$\hat{\beta}_t(i) = \frac{1}{c_t} \tilde{\beta}_t(i)$$

Outline

Multiple Observation Sequences: Training from a Corpus

Let $\alpha_t^k(i)$ and $\beta_t^k(j)$ be the forward and backward probabilities at frame t in training waveform k, and let $P_k = P(O_k|\lambda)$ be the total probability of training waveform k. The paper argues that

$$\bar{a}_{ij} = \frac{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \alpha_t^k(i) a_{ij} b_j(\vec{o}_t^{(k)}) \beta_{t+1}^k(j)}{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \alpha_t^k(i) \beta_{t+1}^k(i)}$$

But notice, that's the same thing as

$$\begin{aligned}\bar{a}_{ij} &= \frac{\sum_{k=1}^{K} \sum_{t=1}^{T_k-1} \xi_t^k(i,j)}{\sum_{k=1}^{K} \sum_{t=1}^{T_k-1} \gamma_t^k(i)} \\ &= \frac{E\left[\#\text{ frames in which } q_{t-1} = i, q_t = j\right]}{E\left[\#\text{ frames in which } q_{t-1} = i\right]}\end{aligned}$$

So really, there is nothing new that you need to learn. You already know how to do multi-file training corpora.

Outline

Other Sections in Part V

- Part V.C: Initial estimates of HMM parameters. There is a method called "segmental K-means" which basically uses the Viterbi algorithm to estimate "hard" state and mixture alignments, Q and M, then calculates μ_{jm} using the hard mixture alignments.
- Part V.D: Effects of insufficient training data. The method of "deleted interpolation" simultaneously trains a high-parameter-count model, λ, and a low-parameter-count model, λ', so that the high-parameter-count model is only used for test utterances on which it is confident.
- Part V.E: Choice of model. ASR uses left-to-right models, but other models work for other signals.

Outline

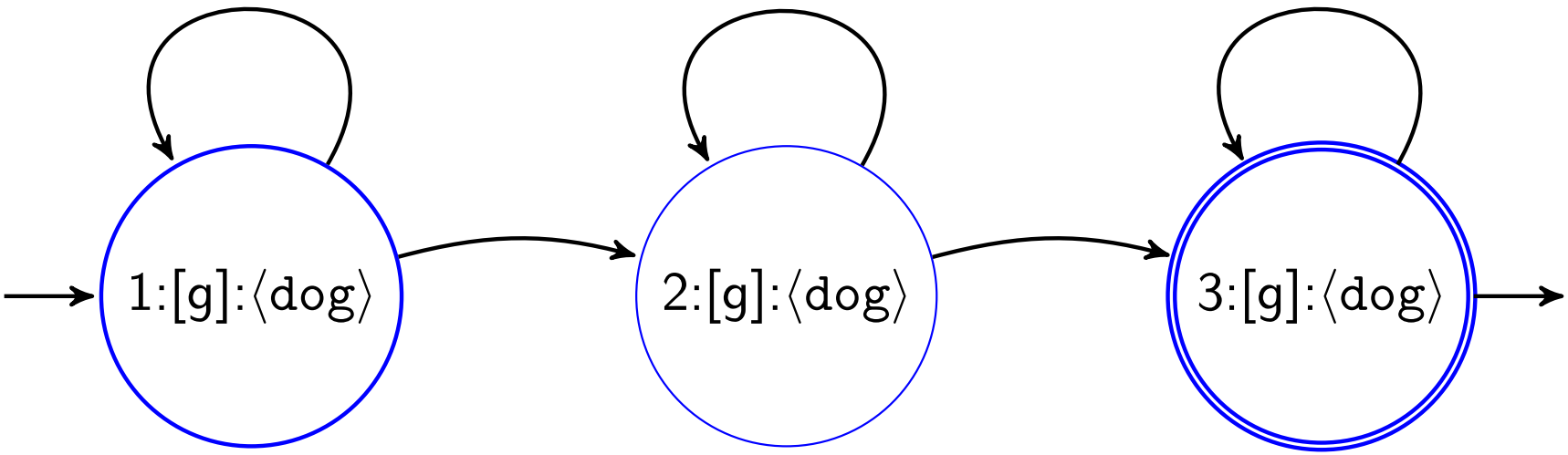
Overall Recognition System

The overall recognition system has basically three levels:

- The **language model** calculates the probability of a sequence of words, $P(w_n|w_{n-1}, w_{n-2}, w_{n-3})$.
- The **pronunciation model**, a.k.a. lexicon, lists the **triphones** (context-dependent phoneme labels) that make up each word.
- Each **triphone** is composed of three **senones**.

Overall Recognition System: Senones

So far, I've been treating a phoneme as if it were a single HMM state, but usually we model each triphone as being composed of three states in a row. For example, the phoneme [g] in the word ⟨dog⟩ might actually be composed of the following three states:



The sequence of those three states would replace the q_t =[g]:⟨dog⟩ edge on the ASR search graph I showed you earlier.

Outline

Connected Word Recognition Using HMMs

- The article describes a two-stage algorithm called "level-building," in which we first test every word for every block of frames, then find the most probable word segmentation.
- Level-building was still necessary in 1989 because it wasn't possible to store, in RAM, a complete search graph including every senone of every triphone of every word in English.
- It's still difficult: such a search graph is typically 500GB. But it's possible, and modern server-side ASR does so.

Outline

The Scaled Forward Algorithm

1. **Initialize:**

$$\hat{\alpha}_1(i) = \frac{1}{c_1}\pi_i b_i(\vec{o}_1)$$

2. **Iterate:**

$$\tilde{\alpha}_t(j) = \sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij} b_j(\vec{o}_t)$$

$$c_t = \sum_{j=1}^{N} \tilde{\alpha}_t(j)$$

$$\hat{\alpha}_t(j) = \frac{1}{c_t}\tilde{\alpha}_t(j)$$

3. **Terminate:**

$$\ln p(O|\lambda) = \sum_{t=1}^{T} \ln c_t$$

The Scaled Backward Algorithm

1. **Initialize:**

$$\hat{\beta}_T(i) = 1, \quad 1 \leq i \leq N$$

2. **Iterate:**

$$\tilde{\beta}_t(i) = \sum_{j=1}^{N} a_{ij} b_j(\vec{o}_{t+1}) \hat{\beta}_{t+1}(j)$$

$$\hat{\beta}_t(i) = \frac{1}{c_t} \tilde{\beta}_t(i)$$