Lecture Notes in Speech Production, Speech Coding, and Speech Recognition

Mark Hasegawa-Johnson
University of Illinois at Urbana-Champaign

February 17, 2000

Chapter 8

Speech Recognition

8.1 Introduction to Recognition

8.1.1 Applications of Speech Recognition

Speech recognition applications vary depending on the vocabulary size, the quality of the microphone, the number of users, and the tolerance for error of the users. A few examples include:

- Hands-free control of machinery.
  - Small vocabulary.
  - Speaker-independent.
  - High-quality microphone (often a headset microphone).
  - Very low error tolerance (error tolerance can be increased with verbal feedback).
- Automatic telephone dialing.
  - Small vocabulary.
  - Mixture of speaker-independent (digits) and speaker-dependent (names).
  - Low-quality microphone (telephone handset).
  - Moderate error tolerance (if the sysems asks for confirmation before dialing!)
- Telephone access to databases.
  - Moderate vocabulary.
  - Speaker-independent.
  - Low-quality microphone.
  - High error tolerance.
- Word processing.
  - Large vocabulary.
  - Speaker-dependent.
  - High-quality microphone.
  - High error tolerance.

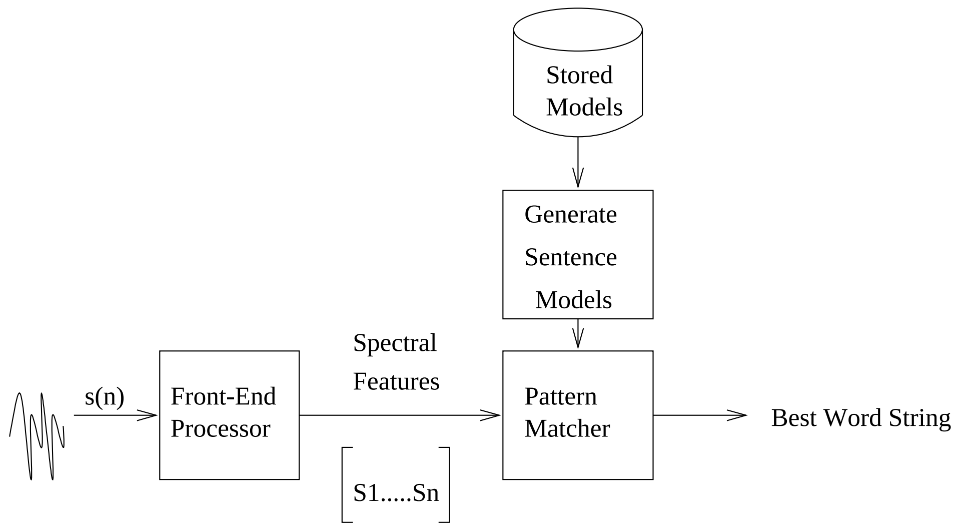

Figure 8.1: Most speech recognizers include a front-end processor, which converts the signal into some sort of spectral vectors, and a pattern-matching unit, which tries to match the spectral vectors to a set of pre-defined models.

8.1.2 Front-End Processor and Pattern-Matcher

All speech recognizers have at least two stages:

1. A front-end processor. The job of this block is reduce the amount of information in the speech signal without throwing out anything that might help identify the speech sound. The output is a feature vector or matrix which is passed to the pattern matching unit.
2. A pattern-matching unit compares the feature matrix to all of the possible word or sentence models, and chooses the word or sentence model which matches best.

8.1.3 Types of Front-End Processing

Modern systems use one of two key technologies. These technologies differ primarily in the way that they handle timing information:

1. **Frame-Based:**
   - **Acoustic feature set** consists of spectra (or LPC coefficients, or cepstra) measured once every "frame" (typically once per 10ms or so).
   - **Front-end processor** measures one spectrum per frame, then reduces the information in the spectrum using a transform similar to factor analysis.
   - **Pattern matcher** is responsible for aligning the input spectra with a template or model of each candidate word, and then deciding which template or model best fits the data.
   - **Examples:** hidden Markov model (HMM), dynamic time warping (DTW).
2. **Segment-Based or Landmark-Based System:**
   - **Front-end processor** finds candidate segment boundaries or landmarks, and measures spectral features at specific offsets relative to each landmark.
   - **Acoustic feature set** consists of candidate landmark times, and, associated with each landmark, a series of up to 6-10 spectra measured at fixed times relative to the landmark.

- **Pattern matcher** is responsible for deciding whether each landmark is a segment-boundary or segment-internal, aligning the input landmarks with a template or model of each candidate word, and then deciding which template or model best fits the data.
- **Examples:** MIT, Boston University systems.

8.1.4 Statistical Speech Recognition: A Type of Pattern Matcher

In the recognizers discussed above, the pattern matcher could be one of many different types of things. For example, it might use a neural network classifier. It might just use a "minimum-distance classifier," which compares the input feature vectors to a set of stored examples representing each of the words of interest, and picks the word which is most similar to the input.

A "statistical speech recognizer" is a recognizer which picks the phoneme or word that has the highest "probability" of matching the unknown utterance. In other words, if o_t is the observed spectrum at some time t, and if the possible phoneme hypotheses are $\lambda_1 =$/i/ and $\lambda_2 =$/a/, then a statistical speech recognizer chooses a "best hypothesis" $\hat{\lambda}$ according to the following rule:

$$\hat{\lambda} = \arg\max_{\lambda_i} p(\lambda_i|o_t) \tag{8.1}$$

There is no easy way to estimate $p(\lambda_i|o_t)$ directly, but using the definition of conditional probability, we can express $p(\lambda_i|o_t)$ in terms of things that *can* be estimated:

$$p(\lambda_i|o_t) = \frac{p(o_t|\lambda_i)p(\lambda_i)}{p(o_t)} \tag{8.2}$$

Since the denominator $p(o_t)$ doesn't depend on λ, it drops out of the classification rule:

$$\hat{\lambda} = \arg\max_{\lambda_i} p(\lambda_i|o_t) = \arg\max_{\lambda_i} p(o_t|\lambda_i)p(\lambda_i) \tag{8.3}$$

The probability $p(\lambda_i|o_t)$ is called the *a posteriori* probability of λ_i, and equation 8.1 is called the *Maximum a Posteriori* (or MAP) rule of classification. Notice that the MAP rule requires us to know two probabilities: the *a priori* probability $p(\lambda_i)$, and the conditional probability $p(o_t|\lambda_i)$.

$p(\lambda_i)$ is a measure of how probable I think it is that the next thing you say will be an λ_i, before I actually hear you say it. In speech recognition, $p(\lambda_i)$ is called the "language model," because it represents our knowledge of the sequences of words or phonemes which are likely in a particular language.

$p(o_t|\lambda_i)$ is the probability that a particular word or sequence of words (λ_i) will be represented by a particular sequence of spectra (o_t). Most of the technical machinery of speech recognition is aimed at estimating this probability in a computationally efficient manner.

Suppose we believe that all speech sounds have equal *a priori* probabilities (an absurd hypothesis, but sometimes this can be a useful simplification). In this case, equation 8.3 simplifies to the equation

$$\hat{\lambda} = \arg\max_{\lambda_i} p(o_t|\lambda_i) \tag{8.4}$$

Equation 8.4 is the rule which says that we should choose whichever class λ_i makes the observed data most "likely," so it is sometimes called "maximum likelihood" classification rule (ML).

8.2 Classification of a Single Spectrum

Suppose we are given a single spectral vector o_t measured at some time t, and we would like to use maximum likelihood classification to choose the most likely phoneme class λ based on this single spectral measurement. For maximum likelihood classification, we need a model of the probability $p(o_t|\lambda)$.

Notice that it is impossible to store this probability exactly: there are an infinite number of possible o_t, so we would need to store an infinite number of probabilities. Instead, we assume that the probability is a function of a small number of parameters, and we estimate these parameters using training data.

8.2.1 Gaussian Probability Models

In order to use either ML or MAP classification rules, we need to create a model of the probability $p(o|\lambda)$ for each of the different possible classes λ. In statistical classification, the way we do this is by gathering 10-1000 spectral vectors, $o_n = [o_n(1), \ldots]$, which we know for sure to be examples of λ, computing statistics, and using the statistics as parameters in some probability model.

For example, $p(o|\lambda)$ can be modeled using a Gaussian distribution. Given N training tokens in class λ, we can create a Gaussian model by just finding the sample mean μ_i, and the sample covariance matrix U_i:

$$\mu_i = \frac{1}{N}\sum_{n=1}^{N} o_n \tag{8.5}$$

$$U_i = \frac{1}{N-1}\sum_{n=1}^{N}(o_n-\mu_i)'(o_n-\mu_i) \tag{8.6}$$

Then, if we want to know the probability that some new spectral vector o belongs to class λ, we calculate $p(o|\lambda)$ using the standard Gaussian formula:

$$p(o|\lambda) = \mathcal{N}(o;\mu_i,U_i) \tag{8.7}$$

where $\mathcal{N}(o;\mu,U)$ is notation for a Gaussian distribution with mean μ and covariance U:

$$\mathcal{N}(o;\mu,U) = \frac{1}{\sqrt{(2\pi)^p|U|}}\exp\left(-\frac{1}{2}(o-\mu)U^{-1}(o-\mu)'\right) \tag{8.8}$$

8.2.2 Contour Plots of a Gaussian Distribution

If o has only two dimensions ($c(1)$ and $c(2)$), it is possible to visualize the probability distribution $p(o|\lambda)$ by defining several "altitudes" θ_k, and drawing the contour lines

$$p(o|\lambda) = \theta_k \tag{8.9}$$

When $p(o|\lambda)$ is a Gaussian probability distribution, the resulting contour plot is always an ellipse in two dimensions, as shown in the upper left plot of figure 8.2.

A special case of particular interest is shown in the upper right plot of figure 2. If $c(1)$ and $c(2)$ are independent of each other — that is, if $u_i(1,2) = 0$ – then the major and minor axes of the ellipse are always parallel to the x_1 and x_2 axes. In the figure, for example, the major axis is x_1, and the minor axis is x_2.

8.2.3 Mixture Gaussian Models

A Gaussian distribution is only a good model if the data is really distributed in an ellipse. A better model for most distributions is a "mixture Gaussian model." The mixture Gaussian model can be thought of as a random choice between M different "Gaussian sub-classes." The probability of choosing sub-class G_{jk} is a constant, c_{jk}:

$$p(G_{jk}|\lambda_j) = c_{jk} \tag{8.10}$$

Then, once we have chosen a particular sub-class, the probability of the output o is calculated using the appropriate Gaussian model:

$$p(o|G_{jk}) = \mathcal{N}(o;\mu_{jk},U_{jk}) \tag{8.11}$$

Putting these together, we get

$$p(o|\lambda_j) = \sum_{k=1}^{M} p(o|G_{jk})p(G_{jk}|\lambda_j) = \sum_{k=1}^{M} c_{jk}\mathcal{N}(o;\mu_{jk},U_{jk}) \tag{8.12}$$

An example of a series of Gaussian probability densities which might be added together to give a mixture Gaussian is shown in the bottom panel of figure 8.2. Notice that a mixture Gaussian can represent a non-elliptical probability density; in fact, if M is large enough, a mixture Gaussian can represent any probability density with arbitrarily good precision.

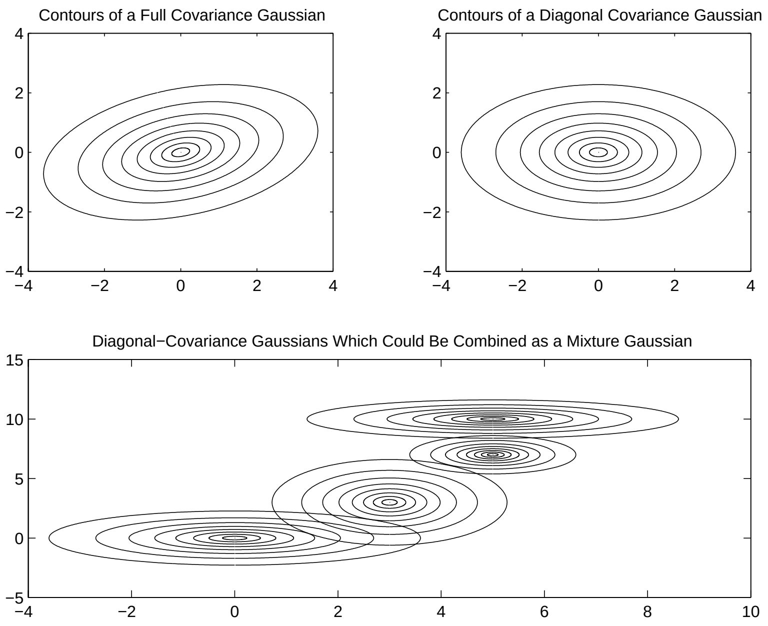

Figure 8.2: Contour plots of Gaussian and mixture-Gaussian probability densities.

8.3 Classification of a Sequence of Spectra

Now suppose that, instead of a single spectrum, you are given a sequence of spectra, of the form

$$O = [o_1, \ldots, o_t, \ldots, o_T] \tag{8.13}$$

For simplicity, let's start by assuming that O is the spectrogram of a single word. Remember that, in order to do maximum likelihood speech recognition, we need "models" λ_i of every possible word. Each of these "models" must consist of a functional specification and a list of trainable parameters which will allow us to compute $p(O|\lambda_i)$. The functional specification must allow us to classify sequences of unknown length, since we don't know in advance how long T may be.

The model which allows us to calculate $p(O|\lambda_i)$ for an O of unknown length is called the hidden Markov model.

8.3.1 Symbol-Timed Markov Process

Imagine a process in which some person or computer is writing the words "ONE" and "TWO," in random order on a strip of paper. What is the probability of observing the sequence of symbols "ONE ONE TWO ONE TWO TWO ONE"?

The person or computer writing these symbols can be modeled as a simple finite state machine, as shown in figure 8.3.1.

Note the following facts:

- The model satisfies the Markov assumption. The Markov assumption states that the probability that the state at time $t+1$ is $q_{t+1} = j$ is a function only of q_t. This probability is called the "transition probability" a_{ij}:

$$P(q_{t+1} = j | q_t = i, q_{t-1} = h, \ldots) = P(q_{t+1} = j | q_t = i) \equiv a_{ij} \tag{8.14}$$

- Notice that we must make a distinction between the state q_t and the observed symbol o_t, because states 1 and 7 output the same symbol ("O"), but their transition probabilities are quite different.

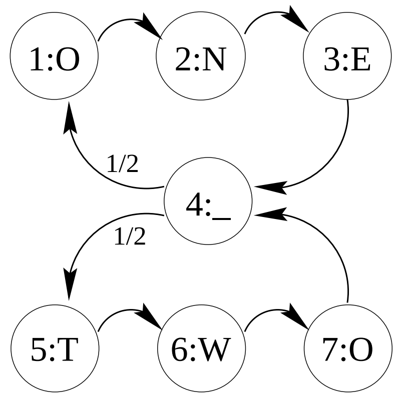

Figure 8.3: A model which generates a random sequence of ones and twos.

- The state $q_{t+1} = j$ can be reached only if the model executes a transition from i to j, where i is whatever the current state happens to be. The probability $P(q_{t+1} = j)$ can be calculated by summing over i:

$$P(q_{t+1} = j) = \sum_{i=1}^{N} P(q_t = i) a_{ij} \qquad \text{or, in matrix notation,} \qquad P_{t+1} = P_t A \tag{8.15}$$

$$P_t = [P(q_t = 1), \ldots, \; P(q_t = N)], \qquad A = \begin{bmatrix} a_{11} & \ldots & a_{1N} \\ \vdots & \vdots & \vdots \\ a_{N1} & \ldots & a_{NN} \end{bmatrix} \tag{8.16}$$

- Given the transition probabilities a_{ij} and the initial state probabilities $\pi_i = P(q_1 = i)$, the probability of any particular sequence of states is

$$P(q_1 = i, q_2 = i, q_3 = k, \ldots) = \pi_i a_{ij} a_{jk} \ldots, \qquad \pi_i \equiv P(q_1 = i) \tag{8.17}$$

The probability of being in state m at time t is the sum over all intermediate states,

$$P(q_t = m) = \sum_{i=1}^{N} \sum_{j=1}^{N} \ldots \sum_{l=1}^{N} \pi_i a_{ij} \ldots a_{lm}, \qquad \text{or, in matrix notation,} \qquad P_t = A^{t-1} \Pi \tag{8.18}$$

$$\Pi = [\pi_1, \ldots, \pi_N], \qquad A = \begin{bmatrix} a_{11} & \ldots & a_{1N} \\ \vdots & \vdots & \vdots \\ a_{N1} & \ldots & a_{NN} \end{bmatrix} \tag{8.19}$$

- For example, using the process shown, the sequence of observations

$$O = [o_1, \; o_2, \ldots, \; o_T] = \text{“ ONE TWO”} \tag{8.20}$$

corresponds to the sequence of states

$$Q = [q_1, \; q_2, \ldots, \; q_T] = [4, \; 1, \; 2, \; 3, \; 4, \; 5, \; 6, \; 7] \tag{8.21}$$

which occurs with a probability of

$$P(Q|q_1 = 4) = a_{41} a_{12} a_{23} a_{34} a_{45} a_{56} a_{67} = (1/2)(1)(1)(1)(1/2)(1)(1) = 1/4 \tag{8.22}$$

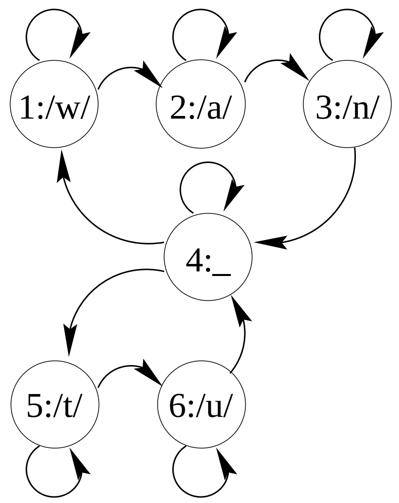

Figure 8.4: A model of a process which speaks the words “one” and “two” in random order.

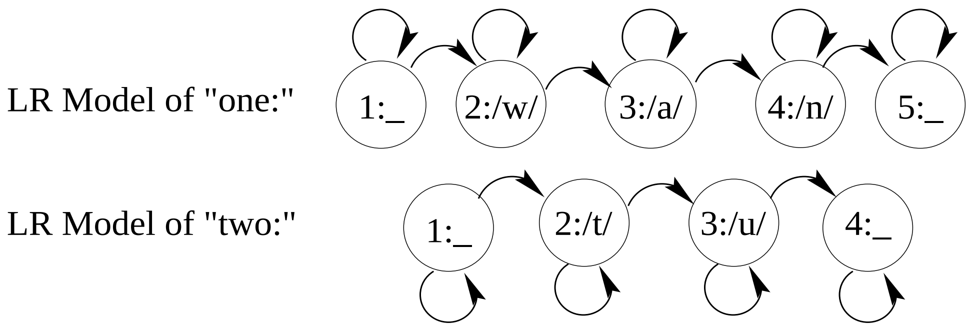

Figure 8.5: A left-to-right Markov process.

8.3.2 Clock-Timed Markov Process: Self-Loops

The example above is “timed” such that there is one clock tick per symbol. In speech, there is no master clock to tell us when a new symbol has been spoken; instead, we need to analyze the speech signal in fixed time increments of, say, 10ms each. If each state now represents one phoneme, then it becomes necessary to introduce self-loop transition probabilities a_{ii} in order to model the variable duration of a phoneme. For example, if the words “one” and “two” are transcribed /w/a/n/ and /t/u/, the model above becomes something like the model shown in figure 8.3.2. Transition probabilities are not shown, but if the frame size is only 10ms, then $a_{ii} \gg a_{ij}$ for any $i \neq j$.

8.3.3 Left-to-Right Models

- An “ergodic” process is a process in which any state can be reached (eventually) from any other state, as shown above.
- A process which generates one word and then quits can be modeled as a “left-to-right” process, as shown below. A “left-to-right” process is a process in which the transition matrix A is upper triangular, so that all transitions have to happen from left to right.

8.3.4 Hidden Markov Models

In the models above, we assumed that the state q_t is directly observable. In speech, the “states” are not observed — instead, all that we observe are the “outputs,” which are vectors v_k containing the spectral,

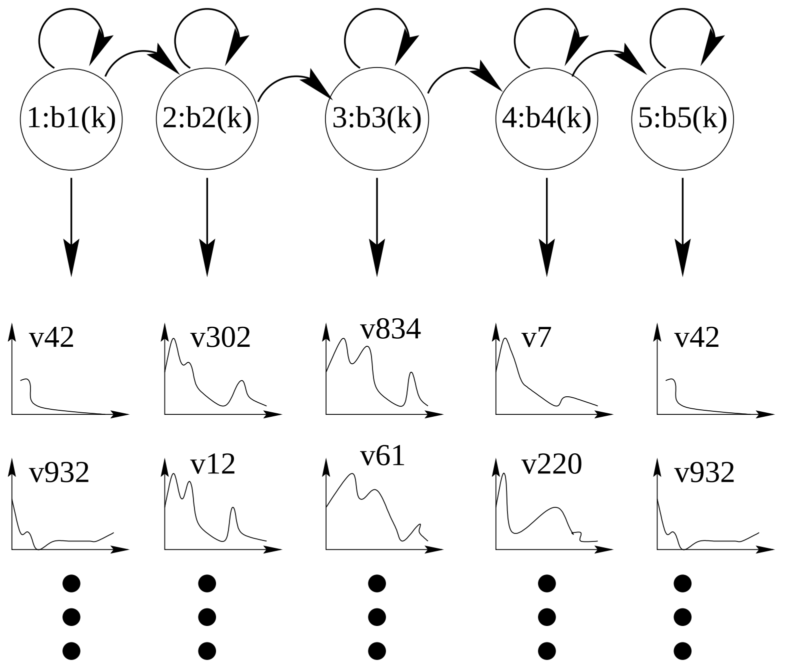

Figure 8.6: A hidden Markov model generates spectral vectors based on some internal state; the internal state of the model can never be known with certainty.

cepstral, or LPC information at time t.

Suppose that that there are only M possible spectral vectors, numbered from v_1 to v_M. Then a hidden Markov model is defined by the initial probabilities $\Pi = [\pi_i]$, the transition probabilities $A = [a_{ij}]$, and the discrete observation probabilities $B = [b_j(k)]$:

$$\begin{aligned} \lambda &= (\pi_i, a_{ij}, b_j(k)), \qquad 1 \le i \le N,\ 1 \le j \le N,\ 1 \le k \le M & (8.23)\\ \pi_i &= p(q_1 = i) & (8.24)\\ a_{ij} &= p(q_t = j | q_{t-1} = i) & (8.25)\\ b_j(k) &= p(o_t = v_k | q_t = j) & (8.26) \end{aligned}$$

8.3.5 Example: Hidden Coin Toss

For example, consider an experiment in which your friend is tossing two coins behind a curtain, and yelling out the result of each coin toss. Your friend is switching back and forth between two coins, but he is not going to tell you when he switches. All you know is that the probability of a coin change on any given toss is always 25%:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 0.75 & 0.25 \\ 0.25 & 0.75 \end{bmatrix} \tag{8.27}$$

Furthermore, you do not know exactly *which* two coins your friend is using. You know that one of the coins is fair, but the other coin might be the "head-weighted" coin (which produces heads 75% of the time) or the "tail-weighted" coin (which produces tails 75% of the time). The two models you have available are:

$$B_1 = \begin{bmatrix} b_1(H) & b_1(T) \\ b_2(H) & b_2(T) \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 0.75 & 0.25 \end{bmatrix}, \qquad B_2 = \begin{bmatrix} b_1(H) & b_1(T) \\ b_2(H) & b_2(T) \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 0.25 & 0.75 \end{bmatrix} \tag{8.28}$$

Finally, you are told that your friend always starts with the unfair coin, regardless of *which* unfair coin he is using:

$$\Pi_1 = [P(q_1 = 1), \quad P(q_1 = 2)] = [0,\ 1] \tag{8.29}$$

If your friend yells out the following sequence, is he using the head-weighted coin or the tail-weighted coin?

$$O = [o_1, \ldots,\ o_T] = [H, T] \tag{8.30}$$

- Case 1: Head-weighted coin. The probability of the sequence "HT" is

$$P(O|\lambda_1) = \sum_{j=1}^{2} \sum_{i=1}^{2} b_j(T) a_{ij} b_i(H) \pi_i = \sum_{j=1}^{2} b_j(T) a_{2j} b_2(H) = (0.5 \times 0.25 \times 0.75) + (0.25 \times 0.75 \times 0.75) = \frac{15}{64} \tag{8.31}$$

- Case 2: Tail-weighted coin. The probability of the sequence "HT" is

$$P(O|\lambda_2) = \sum_{j=1}^{2} b_j(T) a_{2j} b_2(H) = (0.5 \times 0.25 \times 0.25) + (0.75 \times 0.75 \times 0.25) = \frac{11}{64} \tag{8.32}$$

So the sequence "HT" is more likely to be produced if your friend starts with the head-weighted coin — model 1.

8.3.6 Continuous-Distribution HMMs

The observations in a hidden Markov model may be continuous random variables, distributed according to a mixture Gaussian distribution:

$$b_j(o) = p(o_t = o | q_t = j) = \sum_{k=1}^{M} c_{jk} \mathcal{N}(o; \mu_{jk}, U_{jk}), \qquad 1 \le j \le N \tag{8.33}$$

In this case, rather than specifying a set of discrete probabilities $b_j(k)$, a model is specified by finding the mixture weights, the means, and the covariance matrices:

$$\lambda = (\pi_i, a_{ij}, c_{jk}, \mu_{jk}, U_{jk}), \qquad 1 \le i \le N,\ 1 \le j \le N,\ 1 \le k \le M \tag{8.34}$$

8.3.7 Example: Automatic Language Identification

Problem Statement

As a simple application of the HMM, consider a system which records a person saying "yes" in either Japanese or Swedish, and then identifies the language.

Being the polyglot that you are, you know that yes in Japanese is "hai," and yes in Swedish is "ja." In order to keep complexity down, let's assume that "hai" is approximately an /a/ sound followed by an /i/ sound, while "ja" is approximately an /i/ sound followed by an /a/ sound, as shown in figure 8.7. Suppose further that both models always start in the first state shown, i.e. "hai" always starts in /a/, and "ja" always starts in /i/.

The only spectral measurement available is the second formant frequency, F2. Means and standard deviations of F2 for /i/ and /a/ can be approximated by combining the data of Peterson and Barney (1952) for adult male and female speakers, yielding the following observation probability densities:

$$b_{/i/}(o) = \mathcal{N}(o; 2540\text{Hz}, (350\text{Hz})^2) \tag{8.35}$$
$$b_{/a/}(o) = \mathcal{N}(o; 1160\text{Hz}, (150\text{Hz})^2) \tag{8.36}$$

Now suppose that we are given the following (very short!) unknown utterance:

$$O = [\ 1800,\quad 1500\ ] \tag{8.37}$$

Which language is this person speaking?

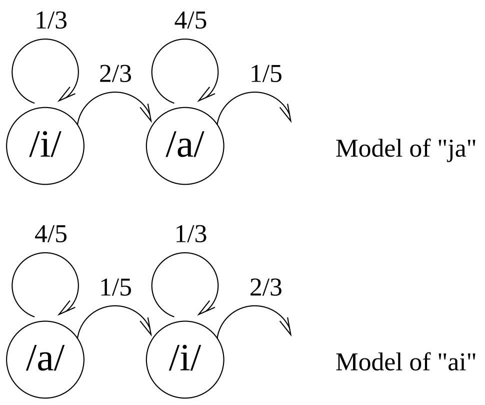

Figure 8.7: Simple Markov models of the words “hai” (/ai/, if we ignore the /h/) and “ja” (/ia/, if we pretend that /j/ and /i/ are the same). Transition probabilities are designed so that the /i/ states last an average of 1.5 frames, and the /a/ states last an average of 5 frames.

Frame	o (kHz)	$\mathcal{N}(o; 2.54, 0.35^2)$	$\mathcal{N}(o; 1.15, 0.15^2)$
1	1.8	0.12	0.014
3	1.5	0.00022	0.17

Table 8.1: Column 3 is an estimate of the probability that the F2 values in column 2 are produced as part of an /i/ vowel. Column 3 is an estimate of the probability that the F2 values are produced as part of an /a/ vowel. Both columns 3 and 4 show probability density per kilohertz, assuming Gaussian distributions.

Solution

The only information we have about the distribution of F2 is its mean and standard deviation, so let’s use Gaussian observation probability densities. The observation probability densities of the /i/ model in “ja” and the /i/ model in “hai” happen to be exactly the same (in speech recognition, we say that these two states have “tied” observation densities), as are the distributions of /a/ in both “ja” and “hai.” The observation densities $b_{/i/}(o)$ and $b_{/a/}(o)$ are given in table 8.1.

If the person is speaking Japanese, the model must have started in /a/; it might have stayed in /a/ for the second frame, or it might have transitioned to /i/. Adding up the likelihoods of both possibilities, we get:

$$\begin{aligned} p(O|\text{“hai”}) &= b_{/a/}(1800)a_{/a/a/}b_{/a/}(1500) + b_{/a/}(1800)a_{/a/i/}b_{/i/}(1500) && (8.38)\\ &= 0.00026 && (8.39)\end{aligned}$$

Likewise, the likelihood that the person is speaking Swedish is

$$\begin{aligned} p(O|\text{“ja”}) &= b_{/i/}(1800)a_{/i/i/}b_{/i/}(1500) + b_{/i/}(1800)a_{/i/a/}b_{/a/}(1500) && (8.40)\\ &= 0.0101 && (8.41)\end{aligned}$$

The observed falling F2 pattern seems much more likely to have come from the word “ja” than from the word “hai.” If the two words have equal *a priori* probabilities, then we can be pretty confident in declaring that the word was “ja.”

8.4 Recognition Using a Hidden Markov Model

Suppose that you have hidden Markov models of two words. The word “one” is represented by the model $\lambda_1 = (A_1, B_1, \pi_1)$. The word “two” is represented by the model $\lambda_2 = (A_2, B_2, \pi_2)$. Suppose, finally, that you observe a sequence of spectral vectors of the form:

$$O = [o_1, \ldots,\ o_t, \ldots,\ o_T] \qquad (8.42)$$

Speech recognition (given that you have already somehow trained the models λ_1 and λ_2) boils down to the following problem: Given two models, λ_1 and λ_2, which model is most likely to have produced the observation sequence O? That is, which model maximizes the likelihood $p(O|\lambda)$?

Suppose we know for a fact that the model went through the following state sequence:

$$Q = [q_1,\ q_2, \ldots,\ q_T] \tag{8.43}$$

The probability of O given Q and λ is

$$p(O|Q,\lambda) = b_{q_T}(o_T)b_{q_{T-1}}(o_{T-1})\ldots b_{q_1}(o_1) \tag{8.44}$$

The probability of Q is

$$p(Q|\lambda) = a_{q_{T-1}q_T}\ldots a_{q_1q_2}\pi_{q_1} \tag{8.45}$$

Combining these two equations, we get the following:

$$p(O,Q|\lambda) = b_{q_T}(o_T)a_{q_{T-1}q_T}b_{q_{T-1}}(o_{T-1})\ldots a_{q_1q_2}b_{q_1}(o_1)\pi_{q_1} \tag{8.46}$$

8.4.1 Maximum Likelihood Recognition: The Forward Algorithm

In order to do ML classification correctly, we need the probability $p(O|\lambda)$. $p(O|\lambda)$ can be obtained from $p(O,Q|\lambda)$ by summing over all possible state sequences Q:

$$P(O|\lambda) = \sum_Q P(O,Q|\lambda) \tag{8.47}$$

$$= \sum_{q_T}\cdots\sum_{q_2}\sum_{q_1} b_{q_T}(o_T)a_{q_{T-1}q_T}b_{q_{T-1}}(o_{T-1})\ldots b_{q_2}(o_2)a_{q_1q_2}b_{q_1}(o_1)\pi_{q_1} \tag{8.48}$$

Suppose that we decide to add up all of the information from time $t=1$ as soon as we have it, then add up all of the information from time $t=2$, and so on. At time $t=1$, we define

$$\alpha_1(j) = p(o_1, q_1 = j|\lambda) = b_j(o_1)\pi_j \tag{8.49}$$

At time $t=2$, we have

$$\alpha_2(j) = p(o_1, o_2, q_2 = j|\lambda) = b_j(o_2)\sum_{i=1}^{N} a_{ij}\alpha_1(i) \tag{8.50}$$

Likewise, at every new time until $t=T$, we have

$$\alpha_t(j) = p(o_1,\ldots,o_t, q_t = j|\lambda) = b_j(o_t)\sum_{i=1}^{N} a_{ij}\alpha_{t-1}(i) \tag{8.51}$$

Finally, at time T, we see that

$$p(O|\lambda) = p(o_1,\ldots,o_T|\lambda) = \sum_{i=1}^{N}\alpha_T(i) \tag{8.52}$$

Equations 8.49 through 8.52 are called the "forward algorithm," because the iteration moves forward in time; the same thing could also be done backward in time.

The Backward Algorithm

Equation **??** can also be broken down into a recursion which moves *backward* in time:

$$P(O|\lambda) = \sum_{q_1}\pi_{q_1}b_{q_1}(o_1)\sum_{q_2}a_{q_1q_2}b_{q_2}(o_2)\ldots\sum_{q_T}a_{q_{T-1}q_T}b_{q_T}(o_T) \tag{8.53}$$

This recursion is written most simply if we define the backward variable $\beta_t(i)$:

$$\beta_t(i) \equiv P(o_{t+1},\ o_{t+2},\ldots,\ o_T|q_T = i,\ \lambda) \tag{8.54}$$

Then equation 8.53 is calculated using:

1. Initialization
$$\beta_T(i) = 1, \quad 1 \leq i \leq N \tag{8.55}$$
2. Induction
$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \tag{8.56}$$
3. Termination
$$P(O|\lambda) = \sum_{i=1}^{N} \pi_i b_i(o_1) \beta_1(i) \tag{8.57}$$

The calculation of this procedure is about the same as the forward algorithm, but it is not used as often in recognition, because the recursion works backward in time. However, this algorithm is often used in segmentation and training.

8.4.2 Approximate Recognition: The Viterbi Algorithm

Suppose that, for whatever reason, we don't want to do the addition in equation 8.51 at every time step for every possible combination of states. If this is the case, we can do a sort of approximate maximum likelihood classification. Instead of finding $p(O|\lambda)$, we find $p(O, Q|\lambda)$ for the best possible state sequence, which we can call $Q^*(O, \lambda)$:

$$\begin{aligned} Q^*(O,\lambda) &= \arg\max_Q p(O,Q|\lambda) && (8.58) \\ P^*(O,\lambda) &= \max_Q p(O,Q|\lambda) && (8.59) \\ &= \max_{q_T} \ldots \max_{q_2} \max_{q_1} b_{q_T}(o_T) a_{q_{T-1}q_T} b_{q_{T-1}}(o_{T-1}) \ldots b_{q_2}(o_2) a_{q_1 q_2} b_{q_1}(o_1) \pi_{q_1} && (8.60) \end{aligned}$$

Then, in order to decide which sequence of words is the correct sequence, we just look for the model which gives us the largest P^*.

Suppose we decide to do the $\max_{q_1}$ operation as soon as we have all of the information from time $t = 1$, and then do the $\max_{q_2}$ operation as soon as we have all of the information from time $t = 2$, and so on. At time $t = 1$, we can define
$$\delta_1(i) = \pi_i b_i(o_1) \tag{8.61}$$
Then, at every new time t until $t = T$, we find the best path which ends up in state j. $\delta_t(j)$ keeps track of the maximum probability, and $\psi_t(j)$ is a "back-pointer" which points backward from state j to the best previous state:

$$\begin{aligned} \delta_t(j) &= b_j(o_t) \max_{1\leq i\leq N} \delta_{t-1}(i) a_{ij} && (8.62) \\ \psi_t(j) &= \arg\max_{1\leq i\leq N} \delta_{t-1}(i) a_{ij} && (8.63) \end{aligned}$$

Finally, at time T, we see that the best final probability is

$$P^*(O|\lambda) = \max_{1\leq i\leq N} \delta_T(i) \tag{8.64}$$

and the best state in which to end up is:

$$q_T^* = \arg\max_{1\leq i\leq N} \delta_T(i) \tag{8.65}$$

We can find out what state sequence yielded P^* by working our way backward in time, from time $t = T$ to time $t = 1$, following the "back-pointers" given by the $\psi_t(i)$ variables:

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \quad 1 \leq t \leq T-1 \tag{8.66}$$

$$Q^*(O|\lambda) = [\ q_1^*,\ \ q_2^*, \ldots q_T^*\ ] \tag{8.67}$$

Local Recognition: The Forward-Backward Algorithm

Suppose we are only interested in finding the state at time t which maximizes:

$$\gamma_t(i) = P(q_t = i|O, \lambda) = \frac{P(O, q_t = i|\lambda)}{P(O|\lambda)} \tag{8.68}$$

This can be calculated as follows:

$$\begin{aligned} P(O, q_t = i|\lambda) &= P(o_1,\ o_2, \ldots,\ o_t,\ q_t = i,\ o_{t+1},\ o_{t+2},\ \ldots,\ o_T|\lambda) && (8.69)\\ &= P(o_1,\ o_2, \ldots,\ o_t,\ q_t = i|\lambda) P(o_{t+1},\ o_{t+2},\ \ldots,\ o_T|q_T = i,\ \lambda) && (8.70)\\ &= \alpha_t(i)\beta_t(i) && (8.71)\end{aligned}$$

therefore,

$$\gamma_t(i) = \frac{P(O, q_t = i|\lambda)}{\sum_{i=1}^N P(O, q_t = i|\lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)} \tag{8.72}$$

The most likely state q_t at time t is therefore the state which maximizes $\gamma_t(i)$.

Now suppose we are only interested in finding the most likely two-frame sequence of states. In other words, we would like to find i and j to maximize

$$\xi_t(i,j) = P(q_t = i, q_{t+1} = j|O, \lambda) = \frac{P(q_t = i, q_{t+1} = j, O|\lambda)}{P(O|\lambda)} \tag{8.73}$$

By calculations similar to the calculations for $\gamma_t(j)$, we can show that this probability is

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1})\beta_{t+1}(j)}{P(O|\lambda)} \tag{8.74}$$

So the most likely two-state sequence at times t and $t+1$ is the sequence which maximizes $\xi_t(i,j)$.

Notice that the likely two-frame sequence $q_t = i, q_{t+1} = j$ may not be part of a likely state sequence spanning the entire utterance. If you want to find a state sequence which spans the entire utterance, you have to use the Viterbi algorithm.

8.5 Training a Hidden Markov Model

The goal of training a hidden Markov model is that the parameter π_i, for example, should be proportional to the number of times that the model started in state i out of all of the observed training tokens. In other words, we would like to have model parameters which look something like this:

$$\pi_i = \frac{\text{number of times in state } i \text{ at time } (t=1)}{\text{total number of training utterances}} \tag{8.75}$$

$$a_{ij} = \frac{\text{number of transitions from state } i \text{ to state } j}{\text{total number of transitions out of state } i} \tag{8.76}$$

$$c_{jk} = \frac{\text{number of times we choose Gaussian sub-class } k \text{ while in state } j}{\text{total number of times in state } j} \tag{8.77}$$

$$\mu_{jk} = \frac{\text{sum of } o_t \text{ for all frames spent in class } j\text{, sub-class } k}{\text{total number of times spent in class } j\text{, sub-class } k} \tag{8.78}$$

$$U_{jk} = \frac{\text{sum of } (o_t - \mu_{jk})'(o_t - \mu_{jk}) \text{ for all frames spent in class } j\text{, sub-class } k}{\text{total number of times spent in class } j\text{, sub-class } k} \tag{8.79}$$

The big problem with equations 8.75 through 8.79 is that we don't *know* how often the model is in state j — remember, the state transitions are "hidden"! There are (at least) two different ways to solve this problem: the segmental K-means algorithm (which is typically used to initialize the parameters of an HMM), and the Baum-Welch re-estimation procedure (which is typically used to refine a previously-estimated set of parameters).

8.5.1 Initializing the Observation Densities: Segmental K-Means

The segmental K-means algorithm is based on equations 8.75 through 8.79. In segmental K-means, we use the Viterbi algorithm to figure out which state the model is in at any given time, in a sort of boot-strapping procedure which goes like this:

1. Divide each of the training utterances into N segments, where N is the number of states in the model. Any arbitrary segmentation will work, although a phonetically motivated segmentation often leads to faster convergence.

2. In order to initialize state number j, gather observation vectors from the jth segment in each segmented training utterance. Call these training vectors y_{nj},

$$y_{nj} = o_t \text{ iff } t \text{ is in segment number } j \text{ of some training utterance} \tag{8.80}$$

3. For each state j, cluster the training vectors y_{nj} into M regions V_{jk} using a bottom-up clustering algorithm. Let N_{jk} be the number of vectors in V_{jk}, and let x_{jk} be the centroid of V_{jk}; then the new observation density parameters are

$$c_{jk} = \frac{N_{jk}}{N_j} \tag{8.81}$$

$$\mu_{jk} = x_{jk} \tag{8.82}$$

$$U_{jk} = \frac{1}{N_{jk}} \sum_{y_{n,j} \in V_{jk}} (y_{nj} - x_{jk})'(y_{nj} - x_{jk}) \tag{8.83}$$

4. Given the new parameter estimates, use the Viterbi algorithm to resegment each of the training utterances. If the new segmentation is different from the previous segmentation, go to step number 2.

8.5.2 Refining the Model: Baum-Welch Algorithm

There are several algorithms available for training hidden Markov model parameters, depending on the criterion which you want to optimize. The most common algorithm is the Baum-Welch algorithm, also called the Expectation-Maximization (EM) method. The algorithm works like this: suppose we define the function

$$f(O, Q, \lambda_2) = \log P(O, Q|\lambda_2) \tag{8.84}$$

A reasonable goal of parameter re-estimation would be to maximize the expected value of f over all possible Q, given the model λ_2:

$$E[f(O, Q, \lambda_2)|\lambda_2] = \sum_Q P(O, Q|\lambda_2) \log P(O, Q|\lambda_2) \tag{8.85}$$

Unfortunately, given the structure of an HMM, equation 8.85 can not be maximized in just one step. Instead, the Baum-Welch algorithm tries to maximize equation 8.85 iteratively: first we guess a model λ_1, then we find a new model λ_2 which maximizes

$$E[f(O, Q, \lambda_2)|\lambda_1] = \sum_Q P(O, Q|\lambda_1) \log P(O, Q|\lambda_2) \tag{8.86}$$

Iterating equation 8.86 several times moves the model λ_2 toward a local maximum of equation 8.85.

How To Do It

It can be shown that, given λ_1, the function in equation 8.86 is maximized if the parameters of λ_2 are as follows:

$$\bar{\pi}_i = \text{expected number of times in state } i \text{ at time } (t = 1) \tag{8.87}$$

$$\bar{a}_{ij} = \frac{\text{expected number of transitions from state } i \text{ to state } j}{\text{expected number of transitions out of state } i} \tag{8.88}$$

$$\bar{b}_j(k) = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} \tag{8.89}$$

These expectations can be calculated:

$$E[\#\text{ times in state } i \text{ at time } t=1] = P(q_1 = i|O,\lambda) = \gamma_1(i) \tag{8.90}$$

$$E[\#\text{ times in state } i] = \sum_{t=1}^{T} P(q_t = i|O,\lambda) = \sum_{t=1}^{T} \gamma_t(i) \tag{8.91}$$

$$E[\#\text{ times in state } i \text{ and observing } o_t = v_k] = \sum_{t=1}^{T} P(q_t = i, o_t = v_k|O,\lambda) = \sum_{t \text{ s.t. } o_t=v_k} \gamma_t(i) \tag{8.92}$$

$$E[\#\text{ transitions from } i \text{ to } j] = \sum_{t=1}^{T} P(q_t = i, q_{t+1} = j|O,\lambda) = \sum_{t=1}^{T} \xi_t(i,j) \tag{8.93}$$

$$\tag{8.94}$$

Usually, we train an HMM in two steps. In the first step, the parameters of the mixture-Gaussian observation densities are initialized using the segmental K-means" algorithm — this is the algorithm used in the HTK program HInit, for example. The reason we don't stop there is that the segmental K-means algorithm assumes that the Viterbi algorithm will pick out all of the spectra which correspond to a particular state. In fact, the Viterbi algorithm only picks out the spectra which are *most likely* to have come from a particular state — the less likely spectra are usually assigned to the neighboring state. This means that the segmental K-means algorithm tends to underestimate the amount of variability which is really present in the training data.

In the second step, therefore, the Baum-Welch re-estimation procedure is used to refine the parameter estimates in order to find a "local maximum" of the conditional probability of the training data, $E[\log p(O|\lambda)]$. The words "local maximum" mean that, once you've run the Baum-Welch algorithm, if you then make a *small* change in the parameters, the performance of the recognizer will always get worse. It's entirely possible, however, that if you make a *big* change in the parameters, the performance might get better – in fact, this often happens! This odd and problematic behavior is the reason we initialize parameters first using the segmental K-means algorithm.

8.5.3 Multiple Observation Sequences

Suppose we want to train a model using data from K different waveform files. Each waveform file is a sequence of data vectors,

$$O^{(k)} = [o_1^{(k)}, \ldots,\ o_T^{(k)}], \qquad 1 \le k \le K \tag{8.95}$$

and the total dataset consists of the "sequence of sequences,"

$$\mathbf{O} = [O^{(1)}, \ldots, O^{(K)}] \tag{8.96}$$

The expected values needed for Baum-Welch re-estimation are:

$$E[\#\text{ times in state } i \text{ at time } t=1] = \sum_{k=1}^{K} P(q_1 = i|O^{(k)},\lambda) = \sum_{k=1}^{K} \gamma_1^{(k)}(i) \tag{8.97}$$

$$E[\#\text{ times in state } i] = \sum_{k=1}^{K}\sum_{t=1}^{T} P(q_t = i|O^{(k)},\lambda) = \sum_{k=1}^{K}\sum_{t=1}^{T} \gamma_t^{(k)}(i) \tag{8.98}$$

$$E[\#\text{ times in state } i \text{ and observing } o_t = v_k] = \sum_{k=1}^{K}\sum_{t=1}^{T} P(q_t = i, o_t = v_k|O^{(k)},\lambda) = \sum_{k=1}^{K} \left(\sum_{t \text{ s.t. } o_t=v_k} \gamma_t^{(k)}(i) \right) \tag{8.99}$$

$$E[\#\text{ transitions from } i \text{ to } j] = \sum_{k=1}^{K}\sum_{t=1}^{T} P(q_t = i, q_{t+1} = j|O^{(k)}, \lambda) = \sum_{k=1}^{K}\sum_{t=1}^{T} \xi_t^{(k)}(i,j) \tag{8.100}$$

$$\tag{8.101}$$

where

$$\gamma_t^{(k)}(i) = P(q_t = i|O^{(k)}, \lambda) = \frac{\alpha_t^{(k)}(i)\beta_t^{(k)}(i)}{P(O^{(k)}|\lambda)} \tag{8.102}$$

$$\xi_t^{(k)}(i,j) = P(q_t = i, q_{t+1} = j|O^{(k)}, \lambda) = \frac{\alpha_t^{(k)}(i)a_{ij}b_j(o_{t+1}^{(k)})\beta_{t+1}^{(k)}(j)}{P(O^{(k)}|\lambda)} \tag{8.103}$$

Essentially, the algorithm is exactly the same as it would be with one file, except that if you are only using one file, the formulas for $\bar{a}_{ij}$, $\bar{b}_j(o_k)$, and $\bar{\pi}_i$ simplify in ways which are not possible if you are using multiple files (compare equations 6.40 and 6.110-112 in the text).

8.6 Explicit State Duration Models

8.6.1 Duration Probabilities and Transition Probabilities

In the original HMM model, the probability of remaining in state i for d_i time steps is a geometric PMF:

$$p_i(d) = (1 - a_{ii})a_{ii}^{d-1} \quad \text{if } a_{ii} \text{ independent of } d \tag{8.104}$$

The geometric PMF is a bad model of the distribution of real phonemes or phone-like units. For this reason, it is sometimes useful to train and use an explicit model of the duration PMF. Given an explicit model of $p_i(d)$, it is possible to calculate duration-dependent transition probabilities $a_{ij}(d)$ as follows:

$$a_{ij}(d) = \begin{cases} P(d_i > d|d_i \geq d) & j = i \\ P(d_i = d,\ q_{t+1} = j|d_i \geq d) & j \neq i \end{cases} \tag{8.105}$$

If we assume that the model still has no long-term memory, except that $a_{ii}(d)$ is a function of duration, then the following formulas result:

$$a_{ij}(d_i) = \begin{cases} \frac{1-\sum_{d=1}^{d_i} p_i(d)}{1-\sum_{d=1}^{d_i-1} p_i(d)} & j = i \\ \breve{a}_{ij}(1 - a_{ii}(d_i)) & j \neq i \end{cases} \tag{8.106}$$

where the parameters $\breve{a}_{ij}$ are transition probabilities conditioned on a change in state:

$$\breve{a}_{ij} \equiv P(q_{t+1} = j|q_t = i, q_{t+1} \neq i) \tag{8.107}$$

8.6.2 Recognition Using Explicit Probability Densities

Suppose that, in T time steps, an HMM sequentially visits S states:

$$q_t = r_s, \qquad t_s < t \leq t_s + d_s \tag{8.108}$$

Suppose that the model remains in state r_s for d_s time steps before transitioning to some other state. Then the event Q is the intersection of two events: a "transitions" event R, and a "durations" event D:

$$Q = R \cap D, \qquad R = [r_1, \ldots,\ r_S], \quad D = [d_1, \ldots,\ d_S] \tag{8.109}$$

Assuming that the various state durations and transitions are independent, the probabilities of these two events are

$$P(R|\lambda) = \prod_{s=1}^{S} \breve{a}_{r_{s-1}r_s}, \qquad \breve{a}_{r_0 r_1} \equiv \pi_{r_1} \tag{8.110}$$

$$P(D|\lambda, R) = \prod_{s=1}^{S} p_{r_s}(d_s) \tag{8.111}$$

The probability of any particular state sequence Q can be calculated in terms of the probabilities $p_i(d)$ and $\breve{a}_{ij}$:

$$\begin{aligned} P(O, Q|\lambda) &= \prod_{s=1}^{S} \left(\breve{a}_{r_{s-1}r_s} p_{r_s}(d_s) \prod_{\tau=1}^{d_s} b_{r_s}(o_{t_s+\tau}) \right) && (8.112) \\ &= P(D|\lambda, R) \prod_{t=1}^{T} \tilde{a}_{q_{t-1}q_t} b_{q_t}(o_t) && (8.113) \\ &= P(D|\lambda, R) P(O, Q|\tilde{\lambda}) && (8.114) \end{aligned}$$

where the pseudo-model $\tilde{\lambda}$ is

$$\tilde{\lambda} \equiv [\pi_i,\ \tilde{a}_{ij},\ b_j(o)], \qquad \tilde{a}_{ij} \equiv \begin{cases} 1 & i = j \\ \breve{a}_{ij} & i \neq j \end{cases} \tag{8.115}$$

The recognition probability $P(O|\lambda)$ is computed by adding up $P(O, Q|\lambda)$ over all possible Q:

$$P(O|\lambda) = \sum_{\text{all } Q} P(D|\lambda, R) P(O, Q|\tilde{\lambda}) \tag{8.116}$$

8.6.3 Approximate Duration Modeling using Viterbi and Forward Algorithms

Equation 8.116 can be expressed as a recursion, similar to the forward algorithm, but the computational complexity of the resulting algorithm is so high that it is rarely used (essentially, the HMM is augmented to include ND states, where D is the maximum possible state duration). Instead, many recognizers use the Viterbi algorithm in parallel with the forward algorithm in order to compute an approximate recognition probability.

In the parallel approximation, $P(O|\lambda)$ is computed as follows:

$$P(O|\lambda) \approx P(D^*|\lambda, R^*) \sum_{\text{all } Q} P(O, Q|\tilde{\lambda}) = P(D^*|\lambda, R^*) P(O|\tilde{\lambda}) \tag{8.117}$$

The quantity $P(O|\tilde{\lambda})$ can be computed using the forward algorithm. $P(D^*|\lambda, R^*)$ is the probability of the state durations associated with the single maximum-likelihood state sequence Q^*, as returned by a Viterbi search:

$$Q^* = \arg\max_{Q} P(O, Q|\lambda), \qquad Q^* = D^* \cap R^* \tag{8.118}$$

8.7 Continuous Observation Probability Densities

8.7.1 Multivariate Gaussian Densities

If X is a Gaussian random column vector of dimension p, the probability density of X is

$$p_X(X) = \mathcal{N}(X; \mu, \Sigma) = \frac{1}{(2\pi)^{p/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2} (X - \mu)' \Sigma^{-1} (X - \mu) \right) \tag{8.119}$$

where μ is the mean of X, and Σ is the covariance matrix.

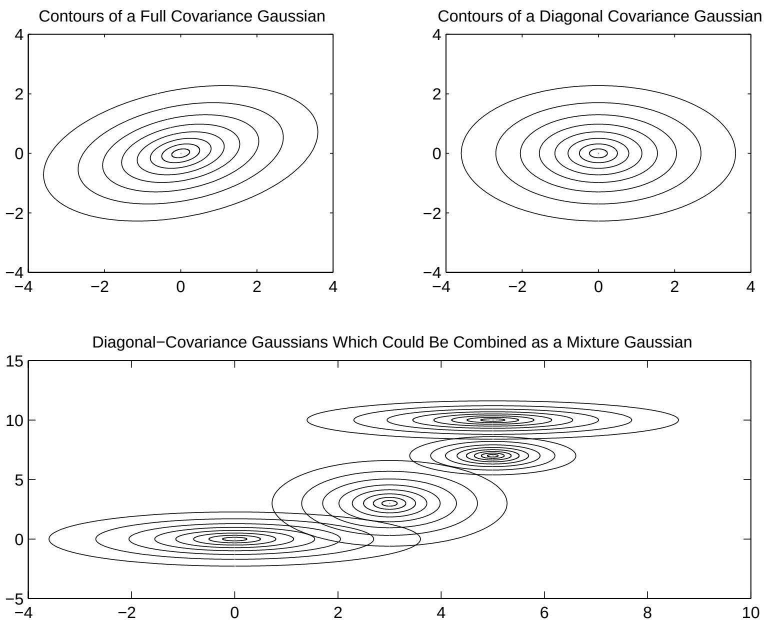

Figure 8.8: Contour plots of Gaussian and mixture-Gaussian probability densities.

Contour Plots of a Gaussian Distribution

Suppose that X is a two-dimensional vector:

$$X = [x_1,\ x_2]', \quad \mu = [\mu_1,\ \mu_2]', \quad P = \Sigma^{-1} = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix} \tag{8.120}$$

It is possible to visualize the probability density $p_X(X)$ by defining several "altitudes" c_i, and drawing, in X space, the curves

$$p_X(X) = c_i \tag{8.121}$$

By setting equation 8.119 equal to c_i, we derive

$$\begin{aligned} -2\log\left(c_i 2\pi\sqrt{|\Sigma|}\right) &= (X-\mu)'P(X-\mu) &(8.122)\\ &= (x_1-\mu_1)^2 p_{11} + (x_1-\mu_1)(x_2-\mu_2)(p_{12}+p_{21}) + (x_2-\mu_2)^2 p_{22} &(8.123)\end{aligned}$$

For a constant c_i, equation 8.122 is the equation for an ellipse, so the contour plots of $p_X(X)$ are shaped like ellipses, as shown in the upper left plot of figure 1.

A special case of particular interest is shown in the upper right plot of figure 2. If Σ, the covariance matrix, is diagonal, then $P = \Sigma^{-1}$ is also diagonal, and equation 8.122 becomes

$$(X-\mu)'P(X-\mu) = (x_1-\mu_1)^2 p_{11} + (x_2-\mu_2)^2 p_{22} \tag{8.124}$$

This is still an ellipse, but the major and minor axes of the ellipse are always parallel to the x_1 and x_2 axes. In the figure, for example, the major axis is x_1, and the minor axis is x_2.

Gaussian Densities in a Hidden Markov Model

The observations in a hidden Markov model may be continuous random variables, distributed according to a Gaussian distribution:

$$b_j(o) = p(o|q_t = j) = \mathcal{N}(o; \mu_j, U_j), \qquad 1 \le j \le N \tag{8.125}$$

If we are given a model λ which includes initial estimates of μ_j and U_j for each state, it is possible to calculate the forward and backward variables $\alpha_t(j)$ and $\beta_t(j)$, and to multiply them to find

$$\gamma_t(j) = P(q_t = j|\, O, \lambda) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\, \lambda)} \qquad 1 \le j \le N,\ 1 \le t \le T \tag{8.126}$$

Using the statistic $\gamma_t(j)$, the expected number of times that the model visits state j in T time steps is

$$E[N_j|\, O, \lambda] = \sum_{t=1}^{T} 1 \times P(q_t = j|\, O, \lambda) + 0 \times P(q_t \ne j|\, O, \lambda) = \sum_{t=1}^{T} \gamma_t(j) \qquad 1 \le j \le N \tag{8.127}$$

Suppose we wish to re-estimate the value μ_j, the mean of o in state j. A good estimate would be

$$\bar{\mu}_j \text{ is something like } \left(\frac{1}{N_j}\right) \sum_{t \text{ s.t. } q_t=j} o_t \qquad 1 \le j \le N \tag{8.128}$$

Unfortunately, both the numerator and the denominator are random variables, so we need to take expected values:

$$\bar{\mu}_j = \frac{E\left[\sum_{t \text{ s.t. } q_t=j} o_t | O, \lambda\right]}{E[N_j|\, O, \lambda]} = \frac{\sum_{t=1}^{T} o_t \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \qquad 1 \le j \le N \tag{8.129}$$

This turns out to be the maximum-likelihood re-estimation value $\bar{\mu}_j$. The maximum-likelihood updated estimate of the covariance, $\bar{U}_j$, is

$$\bar{U}_j = \frac{E[\sum_{t \text{ s.t. } q_t=j} (o_t - \mu_j)(o_t - \mu_j)' | O, \lambda]}{E[N_j | O, \lambda]} = \frac{\sum_{t=1}^{T} (o_t - \mu_j)(o_t - \mu_j)' \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \qquad 1 \le j \le N \tag{8.130}$$

8.7.2 Mixture Gaussian Models

A Gaussian distribution is only a good model if the true distribution $b_j(o)$ is shaped like an ellipse (or a p-dimensional ellipsoid, if $p > 2$). A better model for most distributions is a mixture Gaussian model. An example of a mixture Gaussian distribution is shown in the bottom panel of Fig. 1; the formula is as follows:

$$b_j(o) = \sum_{k=1}^{M} c_{jk} \mathcal{N}(o; \mu_{jk}, U_{jk}) \qquad 1 \le j \le N \tag{8.131}$$

The mixture Gaussian model can be thought of as a random choice between M different Gaussian probability densities. We roll the dice, and with probability c_{jk}, choose output distribution number $G_t = k$:

$$b_j(o) = \sum_{k=1}^{M} c_{jk} b_{jk}(o), \qquad b_{jk}(o) = \mathcal{N}(o; \mu_{jk}, U_{jk}) \tag{8.132}$$

Since the c_{jk}'s are probabilities, they must be normalized:

$$c_{jk} = P(G_t = k | q_t = j), \qquad \sum_{k=1}^{M} c_{jk} = 1 \tag{8.133}$$

We can think of the probabilities c_{jk} as transition probabilities, from a state $q_t = j$ which takes up time but produces no output, to a state $G_t = k$ which produces an output but takes up no time. In one time step, the Markov process travels from $q_t = j$ to $G_t = k$, produces an output, and then continues on to state q_{t+1} (see figure 6.9 in the text).

Re-Estimation Equations

Re-estimation for a mixture Gaussian model depends on the training statistic:

$$\gamma_t(j,k) = P(q_t = j, G_t = k|O,\lambda) = \gamma_t(j)P(G_t = k|q_t = j, O, \lambda) = \gamma_t(j)\left[\frac{c_{jk}\mathcal{N}(o_t;\mu_{jk},U_{jk})}{\sum_{k=1}^{M} c_{jk}\mathcal{N}(o_t;\mu_{jk},U_{jk})}\right] \tag{8.134}$$

Notice that

$$\gamma_t(j) = \sum_{k=1}^{M} \gamma_t(j,k) \tag{8.135}$$

Consider the number N_{jk}, the number of times that the model moves from state $q_t = j$ to Gaussian $G_t = k$. The re-estimation probabilities for the mixture Gaussian parameters are then

$$\bar{c}_{jk} = \frac{E[N_{jk}|O,\lambda]}{E[N_j|O,\lambda]} = \frac{\sum_{t=1}^{T}\gamma_t(j,k)}{\sum_{t=1}^{T}\gamma_t(j)} \qquad 1 \le j \le N,\ 1 \le k \le M \tag{8.136}$$

$$\bar{\mu}_j = \frac{E\left[\sum_{t \text{ s.t. } q_t=j,G_t=k} o_t|O,\lambda\right]}{E[N_{jk}|O,\lambda]} = \frac{\sum_{t=1}^{T} o_t\gamma_t(j,k)}{\sum_{t=1}^{T}\gamma_t(j,k)} \qquad 1 \le j \le N,\ 1 \le k \le M \tag{8.137}$$

$$\bar{U}_j = \frac{E[\sum_{t \text{ s.t. } q_t=j,G_t=k}(o_t-\mu_j)(o_t-\mu_j)'|O,\lambda]}{E[N_{jk}|O,\lambda]} = \frac{\sum_{t=1}^{T}(o_t-\mu_j)(o_t-\mu_j)'\gamma_t(j,k)}{\sum_{t=1}^{T}\gamma_t(j,k)} \qquad 1 \le j \le N,\ 1 \le k \le M \tag{8.138}$$

8.7.3 Feedforward Neural Networks

Classification Using a Neural Network

Figure 8.9 shows the ith level-one neuron in a feed-forward neural network. The output of the neuron model, $y_i(x_1,x_2)$ is the result of linearly combining x_1 and x_2, shifting by some constant a_0, and passing the result through a sigmoidal (S-shaped) nonlinearity $f()$:

$$y_i(x_1,x_2) = f(a_0 + a_1x_1 + a_2x_2), \qquad f(x) \approx \left\{ \begin{array}{ll} 0 & x < 0 \\ 1 & x > 1 \end{array} \right. \tag{8.139}$$

$y_i(x_1,x_2)$ is a classification function, which is one over almost half of the (x_1,x_2) plane, zero over almost half of the plane, and between one and zero in some transition region (shaded in Fig. 2):

$$y_i(x_1,x_2) = \left\{ \begin{array}{ll} 0 & a_0 + a_1x_1 + a_2x_2 < 0 \\ 0 < y_i < 1 & \text{transition region} \\ 1 & a_0 + a_1x_1 + a_2x_2 > 1 \end{array} \right. \tag{8.140}$$

Often, it is useful to create a classification function $z_i(x_1,x_2)$ which is zero for all but a convex set C_i of points. $z_i(x_1,x_2)$ is created by combining the y's, and passing the result through another sigmoidal nonlinearity:

$$z_i(x_1,x_2) = f(b_0 + \sum_{j=1}^{N_y} b_jy_j) \tag{8.141}$$

The resulting system is shown in figure 8.10.

Suppose that points which are in the region C_i tend to belong to some underlying class of points Φ_i. Points in the center of the region always belong to Φ_i, but points on the periphery (in the transition region) belong to Φ_i with some probability. If the neural network is trained to classify points as either belonging or not belonging to Φ_i, then the variable $z_i(x_1,x_2)$ can be considered a model of the classification probability:

$$P(x_1,x_2 \in \Phi_i) \approx z_i(x_1,x_2) \ \text{ if } \Phi_i \text{ convex, and model order and training are sufficient} \tag{8.142}$$

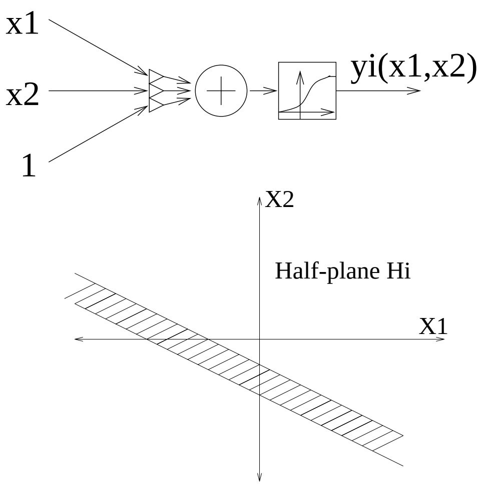

Figure 8.9: Flow-chart and classification space of a single-neuron neural network

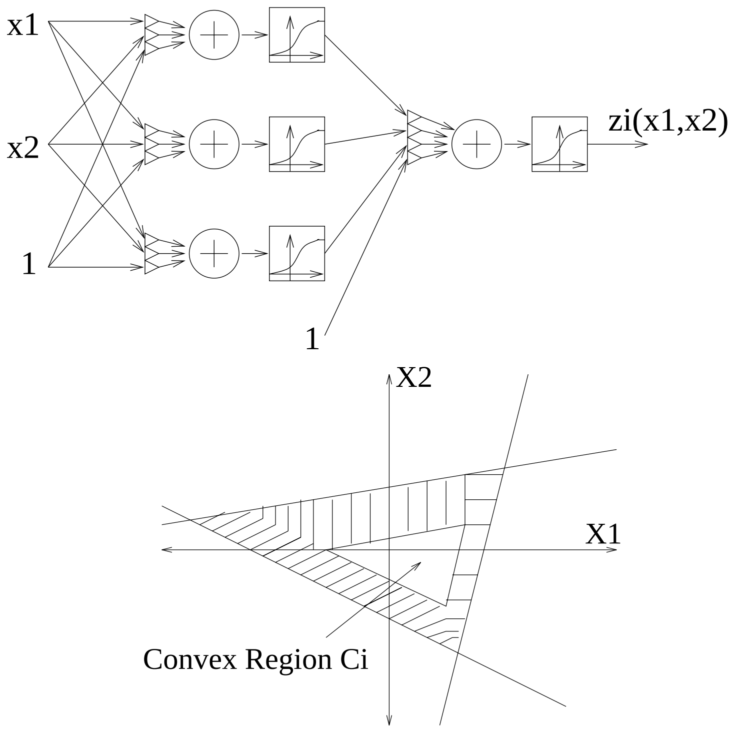

Figure 8.10: Flow-chart, and classification space, of a two-level neural network.

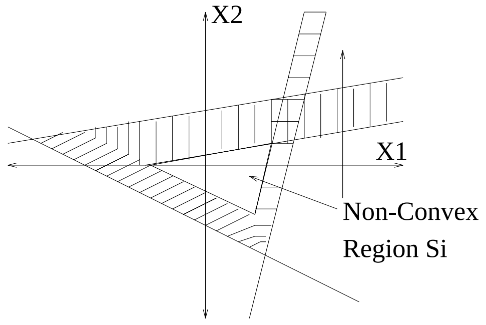

Figure 8.11: Classification space showing a non-convex region S_i.

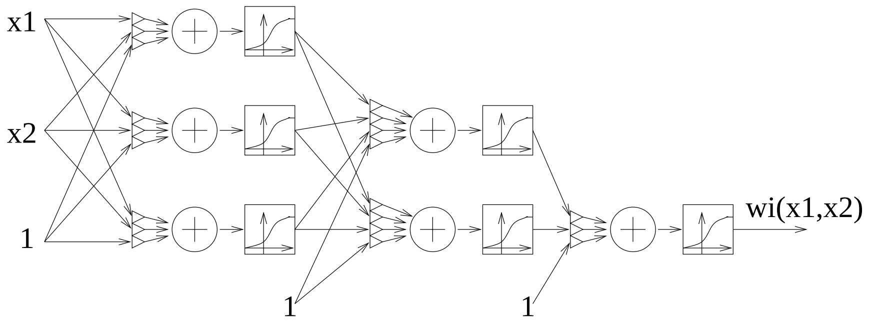

Figure 8.12: Flow-chart of a three-level neural network.

Sometimes, as shown in figure 8.11, the points in a class may not lie in a convex region. If Φ_i is non-convex, the probability $P(x_1, x_2 \in \Phi_i)$ can be calculated by a three-stage neural network, as shown in figure 8.12. The non-convex region is formed by adding together one or more convex regions:

$$P(x_1, x_2 \in \Phi_i) \approx w_i(x_1, x_2) = f(c_0 + \sum_{j=1}^{N} c_j z_j(x_1, x_2)) \tag{8.143}$$

Neural Networks in Hidden Markov Models

Neural networks are very good at classifying static spectra. Given training spectra o of several phonemes Φ_i of interest, it is possible to train neural networks to estimate, with very good accuracy, the quantity

$$P(\Phi_i|o) = w_i(o) \tag{8.144}$$

Unfortunately, neural nets are not very good at representing patterns which expand and contract in the time domain. However, the superior spectral classification accuracy of a neural network can be combined with the temporal flexibility of an HMM if we recognize that

$$b_i(o_t) = p(o_t|q_t = \Phi_i) = \frac{P(\Phi_i|o_t)p(o_t)}{\sum_{i=1}^{N} P(\Phi_i|o_t)p(o_t)} = \frac{P(\Phi_i|o_t)}{\sum_{i=1}^{N} P(\Phi_i|o_t)} \tag{8.145}$$

so the HMM observation probabilities may be calculated from the outputs of an appropriately trained neural network using the formula

$$b_i(o_t) = \frac{w_i(o_t)}{\sum_{i=1}^{N} w_i(o_t)} \tag{8.146}$$

8.7.4 Initializing the Observation Densities: Segmental K-Means

Given a list of training vectors O, Baum-Welch re-estimation is guaranteed to converge to a model λ which is locally optimum, in the sense that it is a local maximum of

$$Q(\lambda, \lambda) = E[\log P(O, Q|\lambda)|\lambda] \tag{8.147}$$

There is no guarantee that the re-estimation procedure will find a global minimum of Q. In practice, Baum-Welch usually finds good values of the transition probabilities a_{ij}, but it may find truly terrible values of the observation parameters c_{jk}, μ_{jk}, and U_{jk} unless the parameters are first initialized to something useful. The *segmental K-means algorithm* is a way to initialize the estimates of c_{jk}, μ_{jk}, and U_{jk}:

1. Divide each of the training utterances into N segments, where N is the number of states in the model. Any arbitrary segmentation will work, although a phonetically motivated segmentation often leads to faster convergence.

2. In order to initialize state number j, gather observation vectors from the jth segment in each segmented training utterance. Call these training vectors y_{nj},

$$y_{nj} = o_t \text{ iff } t \text{ is in segment number } j \text{ of some training utterance} \tag{8.148}$$

3. For each state j, cluster the training vectors y_{nj} into M Voronoi regions V_{jk} using the K-means algorithm. Let N_{jk} be the number of vectors in V_{jk}, and let x_{jk} be the centroid of V_{jk}; then the new observation density parameters are

$$c_{jk} = \frac{N_{jk}}{N_j} \tag{8.149}$$

$$\mu_{jk} = x_{jk} \tag{8.150}$$

$$U_{jk} = \frac{1}{N_{jk}} \sum_{y_{n,j} \in V_{jk}} (y_{nj} - x_{jk})(y_{nj} - x_{jk})' \tag{8.151}$$

   Note that this algorithm always underestimates U_{jk}, so it may be useful to multiply the entire matrix U_{jk} by an expansion factor of perhaps 1.5.

4. Given the new parameter estimates, use the Viterbi algorithm to resegment each of the training utterances. If the new segmentation is different from the previous segmentation, go to step number 2.

5. After the segmental K-means algorithm described above converges to some reasonable initial estimates of c_{jk}, μ_{jk}, and U_{jk}, you should use the full Baum-Welch re-estimation procedure to find the maximum-likelihood values of these parameters.

There are several different versions of this algorithm. For some recognition strategies (e.g. if observation densities are modeled using neural networks), it may be desirable to train the observation densities entirely using a segmental K-means algorith, without trying to incorporate observation density training into the Baum-Welch algorithm.

8.7.5 Tied Mixtures, Continuous Density Codebook

The ML estimation technique adjusts the parameters of an HMM in order to maximize the likelihood $P(O|\lambda)$. If the observation probabilities are Gaussian or mixture Gaussian, then on average,

$$P(O|\lambda) \sim \sum_k \frac{1}{\sqrt{|U_{jk}|}} \tag{8.152}$$

The ML re-estimation algorithm will therefore always try to maximize $P(O|\lambda)$ by choosing the smallest possible U_{jk} which covers the training data. Unfortunately, a small training set can be covered by a U_{jk} which is much too small to represent the variability of new data.

One way to increase the training data for each U_{jk} is by tying together the means and covariances of several states. Instead of an independent mean and covariance for each state, the mean and covariance are common for all states within some class $C(j)$:

$$b_j(o) = \sum_{k=1}^{M} c_{jk}\mathcal{N}(o; \mu_{C(j)k}, U_{C(j)k}) \tag{8.153}$$

The mean and covariance are then calculated by adding across all states in the specified class:

$$\mu_{C(j)k} = \frac{\sum_{t=1}^{T}\sum_{j\in C(j)} o_t\gamma_t(j,k)}{\sum_{t=1}^{T}\sum_{j\in C(j)} \gamma_t(j,k)} \tag{8.154}$$

There are several different methods for determining the composition of the classes $C(j)$:

1. Classes can be formed based on phonetic knowledge; for example, states representing similar vowels in two different word models might be tied together.
2. States can be clustered in a bottom-up fashion based on acoustic similarity. At each stage of the clustering process, a pair of states are tied such that the total likelihood of the training data $P(O|\lambda)$ is decreased as little as possible.
3. In the "mixture codebook" method, all states in all of the models draw from the same "codebook" of M =256-1024 means and covariances, and only the weights c_{jk} depend on the state:

$$b_j(o) = \sum_{k=1}^{M} c_{jk}\mathcal{N}(o; \mu_k, U_k) \tag{8.155}$$

8.8 Spectral Dynamics

According to the recognition model we've developed so far, speech is produced by selecting spectra randomly from the observation PDF $b_i(o)$ for a certain period of time, and then transitioning instantaneously to the new PDF $b_j(o)$ when the HMM changes state.

In real speech, the rate of spectral change may tell you as much about the phoneme as the spectrum itself does. For example, one of the best ways to distinguish syllable-initial /b/ and /w/ is by noting that the formants change more slowly in /w/.

8.8.1 Spectral and Cepstral Derivatives

Remember that the log-power-STFT is a function of both time and frequency. Its inverse transform, the power cepstrum, is therefore a function of both signal time and cepstral lag:

$$c_t(m) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log S_t(\omega)e^{j\omega m}d\omega, \qquad S_t(\omega) \equiv |X_t(\omega)|^2 \tag{8.156}$$

Suppose we are interested in the time-derivative of the log power spectrum. This can be computed by taking the time-derivative of the cepstrum:

$$\frac{\partial c_t(m)}{\partial t} = \frac{1}{2\pi}\int_{-\pi}^{\pi} \frac{\partial \log S_t(\omega)}{\partial t}e^{j\omega m}d\omega \tag{8.157}$$

Suppose that the rate of change of the log power spectrum is governed by a Gaussian distribution:

$$\frac{\partial \log S_t(\omega)}{\partial t} \sim \mathcal{N}(S, \mu_S, U_S) \tag{8.158}$$

then the cepstral derivative is a weighted sum of Gaussians, and is therefore itself a Gaussian random variable:

$$\frac{\partial c_t(m)}{\partial t} \sim \mathcal{N}(o, \mu, U) \tag{8.159}$$

8.8.2 Cepstral Differences

One of the most straightforward ways to model the rate of spectral change using an HMM is by including cepstral differences in your observation vector. Many systems include both a short-term cepstral difference and a long-term cepstral difference, for example

$$o_t = [\ldots,\ c_t(m), \ldots,\ \frac{c_{t+\delta}(m) - c_{t-\delta}(m)}{2}, \ldots,\ \frac{c_{t+\Delta}(m) - c_{t-\Delta}(m)}{2}, \ldots] \tag{8.160}$$

where the offsets δ and Δ are adjusted to model short-term and long-term changes, respectively, e.g.

$$\delta \approx 10 - 20\text{ms}, \qquad \Delta \approx 40 - 80\text{ms} \tag{8.161}$$

It is possible to use different windowing functions at each delay:

$$o_t = [\ldots,\ c_t(m)w_1(m), \ldots,\ \left(\frac{c_{t+\delta}(m) - c_{t-\delta}(m)}{2}\right) w_2(m), \ldots,\ \left(\frac{c_{t+\Delta}(m) - c_{t-\Delta}(m)}{2}\right) w_3(m), \ldots] \tag{8.162}$$

In particular, the spectral energy, $c_t(0)$, is a function of the recording level, so that taken by itself, it tells you nothing at all about the phoneme. However, the rate of change of spectral energy can tell you a great deal about the phoneme, so most recognizers include $c_{t+\delta}(0) - c_{t-\delta}(0)$ and $c_{t+\Delta}(0) - c_{t-\Delta}(0)$ in the observation vector, even if they don't include any other dynamic information.

Likewise, $c_t(1)$ contains information about the spectral slope, which depends on such extraneous factors as the type of microphone and the speaker identity. The rate of change of the spectral slope, however, contains a great deal of phonetic information, so it is useful to include $c_{t+\delta}(1) - c_{t-\delta}(1)$ in your observation vector.

8.8.3 Cepstral Derivative Estimates

The simple first cepstral difference discussed above is a noisy estimate of the cepstral derivative. Instead of using a short-term and long-term cepstral difference, Rabiner & Juang suggest using parametric estimates of the first and second cepstral derivatives. The first and second cepstral derivatives are calculated by fitting a quadratic curve to the cepstral trajectory, in order to minimize the error

$$E = \sum_{t=-M}^{M} [c_t(m) - (h_1(m) + h_2(m)t - h_3(m)t^2)]^2 \tag{8.163}$$

where the window size M is comparable to the long-term cepstral difference window Δ.

The book gives formulas for the optimum h_1, h_2, and h_3 in terms of the cepstral coefficients. Once these coefficients have been computed, the cepstral derivative estimates are

$$\frac{\partial c_t(m)}{\partial t} \approx h_2(m), \qquad \frac{\partial^2 c_t(m)}{\partial t^2} \approx 2h_3(m) \tag{8.164}$$

8.8.4 RASTA

A spectral derivative estimate can be viewed as a high-pass filter of the log-power spectrum or cepstrum. For example, a short-term cepstral difference can be written as

$$d_t(m) = c_{t+1}(m) - c_{t-1}(m), \qquad D_z(m) = H(z)C_z(m), \quad H(z) = z(1 - z^{-2}) \tag{8.165}$$

- One of the biggest advantages of high-pass filtering $D_z(m)$ is that it removes the relatively constant effects of microphone and room tone. Thus, for example, $d_t(0)$ and $d_t(1)$ are more useful than $c_t(0)$ and $c_t(1)$, because constant offsets in the spectral energy and spectral slope caused by variations in recording conditions have been factored out.

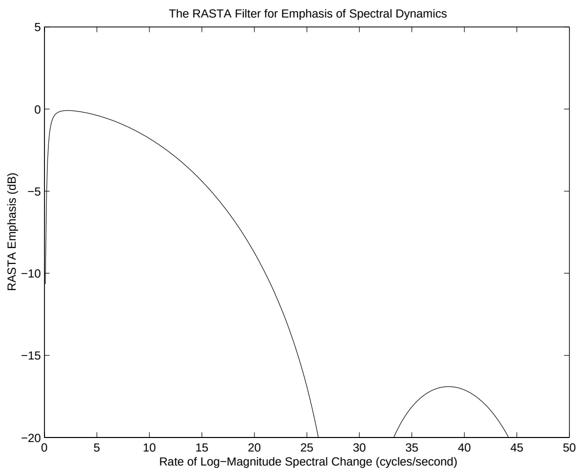

Figure 8.13: In the RASTA method, frame-to-frame variations in a spectral estimate are smoothed using a filter like the one shown here.

- The biggest disadvantage of the high-pass filter is that it emphasizes rapid spectral changes which may not be very important perceptually. In fact, psychophysical research suggests that humans are most sensitive to spectral changes which occur at a rate of about 4-6 cycles per second (about the rate of syllable production in normal speech), and that sensitivity to spectral change drops off at higher frequencies. The high-pass filter in equation 8.165 rises at 6dB per octave all the way up to half the frame rate, e.g. if the frame length is 10ms, $H(z)$ gives the most emphasis to changes at a rate of 50 cycles per second. This does not reflect human hearing very well, which is part of the reason why the observation vector o_t must contain samples of $c_t(m)$ as well as samples of $d_t(m)$.

The RASTA (RelAtive SpecTrAl) method replaces the high-pass filter in equation 8.165 with a band-pass filter. The band-pass filter has the following characteristics:

- A very sharp zero at zero frequency, to remove the effect of recording conditions.
- A relatively flat pass-band from 2 to 6 Hertz, which allows RASTA-filtered coefficients $r_t(m)$ to be used in place of the original coefficients $c_t(m)$.
- A slow roll-off above about 6 Hz, which de-emphasizes rapid spectral changes which are mostly inaudible to human listeners.

The original RASTA filter is as follows, but any filter with the characteristics above could be used just as well:

$$H(z) = \frac{2 + z^{-1} - z^{-3} - 2z^{-4}}{10z^{-2}(1 - 0.98z^{-1})} \tag{8.166}$$

The RASTA technique fulfills one of the original purposes of the delta-cepstrum (removing the influence of recording conditions), but the other condition is not fulfilled. Since $r_t(m)$ has a flat pass-band, it tends to model relatively steady-state spectra, not spectral change; for example, the difference in spectral rate of change between a /b/ and a /w/ is not captured by $r_t(m)$! Therefore, it seems reasonable to use an observation vector which contains RASTA-cepstra and delta-RASTA cepstra, e.g.

$$o_t = [\ldots,\ r_t(m), \ldots,\ \frac{r_{t+\delta}(m) - r_{t-\delta}(m)}{2}, \ldots] \tag{8.167}$$

Unfortunately, the RASTA technique is relatively new, and it is not yet clear whether including delta-RASTA in the observation vector reduces recognition error or not.

8.9 Probability Scaling in the Forward-Backward Algorithm

8.9.1 What's the Problem?

Remember that the induction steps for the forward-backward algorithm are

$$\alpha_t(i) = b_i(o_t)\sum_{j=1}^{N}\alpha_{t-1}(j)a_{ji}, \qquad 1 \le i \le N \tag{8.168}$$

$$\beta_t(i) = \sum_{j=1}^{N}a_{ij}b_j(o_{t+1})\beta_{t+1}(j), \qquad 1 \le i \le N \tag{8.169}$$

$$\tag{8.170}$$

If, for example, $b_j(o_t)$ is calculated using a Gaussian density with covariance matrix U_j, then

$$b_j(o_t) < \frac{1}{|U_j|^{1/2}(2\pi)^{p/2}} = b_{max} \tag{8.171}$$

and therefore

$$\alpha_t(i) < b_{max}^t, \qquad \beta_t(i) < b_{max}^{T-t} \tag{8.172}$$

If $|U_j| > 1$ (as is usually the case), then $\alpha_t(i)$ and $\beta_t(i)$ approach zero very quickly; within 5-10 time steps, they can easily be smaller than the floating point resolution of the computer.

8.9.2 The Scaled Forward Algorithm

The solution involves computing scaled forward and backward variables, $\hat{\alpha}_t(i)$ and $\hat{\beta}_t(i)$. The scaled forward algorithm essentially re-normalizes the αs at every time step so that

$$\sum_{i=1}^{N}\hat{\alpha}_t(i) = 1 \tag{8.173}$$

We can get this normalization by calculating a scaling constant c_t at each time step, as follows:

1. Initialization

$$\hat{\alpha}_1(i) = c_1\alpha_1(i), \qquad c_1 = \frac{1}{\sum_{i=1}^{N}\alpha_1(i)} \tag{8.174}$$

2. Induction

$$\hat{\hat{\alpha}}_t(i) = b_i(o_t)\sum_{j=1}^{N}\hat{\alpha}_{t-1}(j)a_{ji} \tag{8.175}$$

$$\hat{\alpha}_t(i) = c_t\hat{\hat{\alpha}}_t(i), \qquad c_t = \frac{1}{\sum_{i=1}^{N}\hat{\hat{\alpha}}_t(i)} \tag{8.176}$$

8.9.3 Recognition Using the Scaled Forward Algorithm

In the scaled forward algorithm, the scaling factors accumulate over time, so that

$$\hat{\alpha}_t(i) = \alpha_t(i)\prod_{\tau=1}^{T}c_\tau \tag{8.177}$$

The termination step in the normal forward algorithm is

$$P(O|\lambda) = \sum_{i=1}^{N}\alpha_T(i) = \frac{\sum_{i=1}^{N}\hat{\alpha}_T(i)}{\prod_{t=1}^{T}c_t} \tag{8.178}$$

However, remember that the constants c_T are chosen so that

$$\sum_{i=1}^{N} \hat{\alpha}_T(i) = 1 \tag{8.179}$$

Therefore

$$P(O|\lambda) = \frac{1}{\prod_{t=1}^{T} c_t} \tag{8.180}$$

8.9.4 The Scaled Backward Algorithm

It turns out that training works best if the backward algorithm uses the same scaling factors c_t as the forward algorithm, as follows:

1. Initialization

$$\hat{\beta}_T(i) = c_T, \qquad c_1 = \frac{1}{\sum_{i=1}^{N} \alpha_1(i)} \tag{8.181}$$

2. Induction

$$\hat{\beta}_t(i) = c_t \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \hat{\beta}_{t+1}(j), \qquad c_t = \frac{1}{\sum_{i=1}^{N} \hat{\alpha}_t(i)} \tag{8.182}$$

8.9.5 Re-Estimation Using Scaled Parameters

Remember that the original re-estimation algorithm for a_{ij} was

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{8.183}$$

where the training statistics ξ_t and γ_t are defined to be

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{P(O|\lambda)} \tag{8.184}$$

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} = \sum_{j=1}^{N} \xi_t(i,j) \tag{8.185}$$

$$\tag{8.186}$$

The scaled and unscaled forward and backward parameters are related by products of c_t, as follows:

$$\hat{\alpha}_t(i) = \alpha_t(i) \prod_{\tau=1}^{t} c_\tau, \qquad \hat{\beta}_{t+1}(j) = \beta_{t+1}(j) \prod_{\tau=t+1}^{T} c_\tau, \qquad P(O|\lambda) = \frac{1}{\prod_{t=1}^{T} c_t} \tag{8.187}$$

By combining equations 8.184 and 8.187, it is possible to write $\xi_t(i,j)$ in terms of $\hat{\alpha}$ and $\hat{\beta}$:

$$\xi_t(i,j) = \alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \prod_{t=1}^{T} c_t = \hat{\alpha}_t(i) a_{ij} b_j(o_{t+1}) \hat{\beta}_{t+1}(j) \tag{8.188}$$

Calculating $\gamma_t(j)$ is a little trickier. γ is still the sum of ξ, but it is not simply the product of $\hat{\alpha}$ and $\hat{\beta}$:

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j) = \hat{\alpha}_t(i) \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \hat{\beta}_{t+1}(j) = \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{c_t} \tag{8.189}$$

8.10 Phone Models, Lexicon, Phonological Rules

In very large vocabulary recognizers, it is hard to find enough training data to train models of every word. Often, a better approach is to train models of phone-like units (PLUs), and then create each word model by stringing together appropriate PLU models.

8.10.1 Recognition

Suppose you want to test the hypothesis that the sequence of words in a test utterance is

$$W = [w_1, w_2, \ldots, w_q, \ldots, w_Q] \tag{8.190}$$

In a PLU-based recognizer, the first step in recognition is to find each of the words w_q in a look-up table called a "lexicon." The "lexicon" gives all of the known possible pronunciations of the word, as sequences of phone-like units $P_n(w_q) = [p_1, \ldots, p_r, \ldots, p_R]$. Since there are several possible pronunciations of each word, the lexicon also gives a probability for each of the possible pronunciations:

$$w_q \rightarrow P_n(w_q) \ \text{ with probability } \ p(P_n|w_q) \tag{8.191}$$

Normal pronunciations of a word are usually listed in the lexicon. Once a sentence is constructed, however, unusual phonemes in word w_{q-1} or w_{q+1} may cause unusual changes to the pronunciation of word w_q. For example, "did you" is often pronounced as D-IH-JH-AX ("didja"). Most such changes can be accounted for using a small set of phonological rules, which are applied to a sentence after all of the words have been converted into PLU transcriptions using an appropriate table lookup.

Finally, each candidate sequence of phones, $P = [P(w_1), \ldots, P(w_Q)]$, is converted into a sequence of HMM states by looking up the phones in another lookup table. This results in a single giant HMM network of states, with three types of state transitions: transitions which remain inside a phone, transitions which cross from one phone to the next inside a word, and transitions from one word to the next. Transitions which remain inside a phone occur with a probability a_{ij} which is specified by the model

$$p(q_t = j|q_{t-1} = i) = a_{ij} \qquad \text{if } i, j \text{ both states in the model for phone } p_r \tag{8.192}$$

Transitions from one phone to the next phone occur with a probability specified by the lexical probability:

$$\begin{aligned} p(q_t = j|q_{t-1} = i) &= p(q_t = j|p_r)p(p_r|p_{r-1}, w_q) \qquad \text{if } i \text{ in } p_{r-1},\ j \text{ in } p_r,\ \text{both in } w_q & (8.193)\\ &= \frac{p([\ldots, p_{r-1}, p_r, \ldots]|w_q)}{p([\ldots, p_{r-1}, \ldots]|w_q)} \times \pi_{j|p_r} & (8.194)\end{aligned}$$

Finally, transitions from one word to the next word occur with a probability specified by the language model.

8.10.2 Training

Few training databases are transcribed with the beginning and end times of individual phones, so PLU models are trained using an "embedded re-estimation" procedure. In "embedded re-estimation," the word sequence in any training token is first parsed to generate a sequence of phones, then the sequence of phones is parsed to generate a sequence of Markov states, and then finally, the Markov states are matched to the utterance, and the state parameters are updated using the Baum-Welch algorithm.

Similarly, initialization of a PLU model uses an "embedded" version of the segmental K-means algorithm:

1. Look up the PLU transcription of each word in a training utterance.
2. If there are N PLUs in the phonemic transcription of a sentence, divide the training utterance into N equal-length segments.
3. Estimate model parameters using the K-means algorithm, or using the Baum-Welch re-estimation procedure within the specified segment.
4. Re-segment using the Viterbi algorithm.
5. If the segmentation has changed since the previous iteration, go back to step 3.

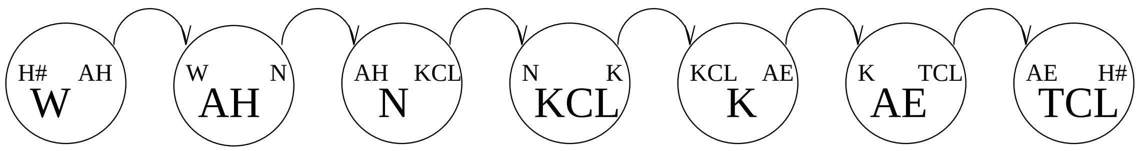

Figure 8.14: A network of triphone models representing the phrase "one cat." Phones are written in the ARPABET transcription system.

8.11 Context-Dependent Phone Models

In fluent speech, the articulators move smoothly from one phoneme target to another, stopping only briefly (if at all) at each phoneme target. As a result, most of the acoustic signal in continuous speech is composed of phoneme transitions.

Transitions can be modeled explicitly by creating diphone or triphone models instead of phone models. A triphone model $p_L - p + p_R$ is a model of the center phone, p, in the context of a particular phone on the left, p_L, and a particular phone on the right, p_R. For example, the phrase "one cat" might be expanded into the network of phone models shown in figure 8.14 (where the phonemes are written in the ARPABET transcription system, and the symbol /H#/ means silence):

The problem with diphone and triphone models is that there are often not enough data to robustly train a separate HMM for every phone in every context. If there are insufficient training data (at least one new speaker per ten HMM models, according to one study), then recognition accuracy may be very high on the training data, and very low on any test data independent of the training set.

If there is insufficient data to train all of the triphones, triphone models may be combined in one of several ways:

1. Triphones which are not well represented in the training data may be replaced by appropriate left-context or right context diphone models, either $p_L - p$ or $p + p_R$. If there is insufficient data to train a diphone model, then use an isolated phone model p.

2. Triphones may be clustered based on acoustic similarity. Initially, all triphones in the database are modeled separately; then triphones are combined, one at a time, in the order which causes the smallest decrease in $P(O|\lambda)$ measured on the training database.

8.12 Deleted Interpolation

Often, in an HMM, it is desirable to combine the model parameters from a more-specific model (which better represents the training data) and a less-specific model (which may be more robust to differences between the training and test data). For example, it may be desirable to combine information from triphone, diphone, and isolated phone models.

Suppose that four different observation densities have been trained to represent state q_j. These densities, numbered $b_j^1(o)$ through $b_j^4(o)$, might represent a triphone model, an isolated phone model, and left and right context diphone models, as shown in figure 8.12.

These four densities can be combined using a hidden transition, with transition weights e_1 through e_4:

$$b_j(o) = \sum_{i=1}^{4} e_i b_j^i(o), \qquad \sum_{i=1}^{4} e_i = 1 \tag{8.195}$$

In order to keep the number of Gaussian mixtures in $b_j(o)$ under control, it is sensible to train the distributions $b_j^i(o)$ with tied mixtures:

$$b_j^i(o) = \sum_{k=1}^{M} c_{jk}^i \mathcal{N}(o, \mu_{jk}, U_{jk}) \tag{8.196}$$

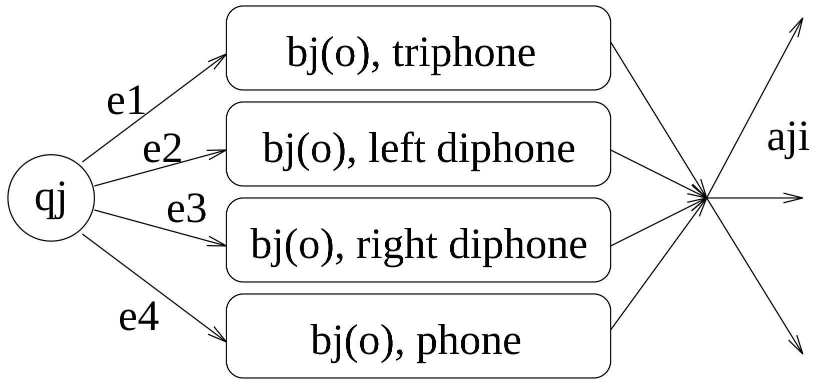

Figure 8.15: Deleted interpolation linearly combines the trained model parameters of monophone, diphone, and triphone models.

If the $b_j^i(o)$ have been properly trained, the most specific distribution (e.g. the triphone model) should always represent the training data better than any of the others. Therefore, it makes no sense to train the e_i parameters on the same data that was used to train the $b_j^i(o)$; we would always get $e_1 = 1$.

The e_i parameters are therefore trained on a dataset called the "development test" set, which is independent of the original training data.

8.13 Connected Word Recognition

In connected word recognition, we seek to find a single word sequence $W(T)$ which maximizes the probability of the observation vectors O_T:

$$O_T = [o_1,\ o_2, \ldots,\ o_T], \quad T = \text{number of frames} \tag{8.197}$$

$$W(T) = W_Q = [w_1,\ w_2, \ldots,\ w_Q], \quad Q = \text{number of words in time } T \tag{8.198}$$

$$P_A^* = \max_{Q,W_Q} (P(O_T|W_Q)) \tag{8.199}$$

$$W^* = \arg\max_{Q,W_Q} (P(O_T|W_Q)) \tag{8.200}$$

In order to calculate P_A^* and W^* in a practical system, it is useful to define two intermediate probabilities called $\alpha_t(i,v)$ and $P_A(t,v)$:

$$\alpha_t(i|v) = \max_{W(t-1)} P(O_t,\ q_t = i \mid W(t-1),\ w(t) = v) \tag{8.201}$$

$$P_A(t|v) = \max_{W(t-1)} P(O_t \mid W(t-1),\ w(t) = v) = \sum_{i=1}^{N_v} \alpha_t(i|v) \tag{8.202}$$

$$\tag{8.203}$$

If we assume that there are V different word models, then the optimum word sequence probability P_A^* can be written as

$$P_A^* = \max_{1\le v\le V} P_A(T|v) = \max_{1\le v\le V} \sum_{i=1}^{N_v} \alpha_T(i|v) \tag{8.204}$$

8.13.1 The One-Pass Algorithm

$\alpha_t(i|\ v)$ can be calculated using the following recursion, which is sort of a combination of the Viterbi and the forward algorithms. Many variations of this algorithm exist, and go by names such as the "one-pass algorithm" and the "frame-synchronous level-building algorithm (FSLB)."

1. **Initialization**

$$\alpha_1(i|v) = \pi_{iv} b_{iv}(o_1) \tag{8.205}$$

$$P_A(1|v) = \sum_{i=1}^{N_v} \alpha_1(i|v) \tag{8.206}$$

where N_v is the number of states in word model λ_v, π_{iv} is the initial-state probability given $w_1 = v$, and $b_{iv}(o_1)$ is the probability of observing o_1 given $q_t = i$ and $w_t = v$.

2. **Recursion**

   At each time step, the model can either remain in the same word, in which case $w(t) = w_q$, or change to a different word, in which case $w(t) = w_{q+1}$. The accumulated word probability is the maximum of these two choices:

$$P_A(t|v) = \max\left(P_A(t|v = w(t-1)),\ P_A(t|v \neq w(t-1))\right) \tag{8.207}$$

   If the model remains in the same word, then the normal forward algorithm is used:

$$\alpha_t(i|v = w(t-1)) = b_{iv}(o_t) \sum_{j=1}^{N_v} \alpha_{t-1}(j|v) a_{ji} \tag{8.208}$$

   If the model changes words, then we require, by convention, that the model must make a transition from the last state of one word to the first state of the next word. In this case,

$$\alpha_t(i|v \neq w(t-1)) = \begin{cases} b_{1v}(o_t) \max_{1 \leq w_q \leq V} \alpha_{t-1}(N_{w_q}, w_q) P(v|W_q) & i = 1 \\ 0 & i \neq 1 \end{cases} \tag{8.209}$$

   where $P(v|W_q)$ is the word transition probability, also known as the "language model:"

$$P(v|W_q) \equiv P(w_{q+1} = v|w_1, \ldots,\ w_q) \tag{8.210}$$

   In either case, we have that

$$P_A(t|v) = \sum_{i=1}^{N_v} \alpha_t(i|v) \tag{8.211}$$

   Combining equations 8.207 to 8.211, we obtain the following recursion:

$$\begin{aligned} P_A(t|v) &= \max\left(\sum_{i=1}^{N} b_{iv}(o_t) \sum_{j=1}^{N_v} a_{ji}\alpha_{t-1}(j|v),\ \ b_{1v}(o_t) \max_{1 \leq w_q \leq V} \alpha_{t-1}(N_{w_q}|w_q) P(v|W_q)\right) && (8.212) \\ \psi_t(v) &= \arg\max\left(\sum_{i=1}^{N} b_{iv}(o_t) \sum_{j=1}^{N_v} a_{ji}\alpha_{t-1}(j,v),\ \ b_{1v}(o_t) \max_{1 \leq w_q \leq V} \alpha_{t-1}(N_{w_q}, w_q) P(v|W_q)\right) && (8.213) \end{aligned}$$

3. **Termination**

$$\begin{aligned} P_A^* &= \max_{1 \leq v \leq V} P_A(T|v) && (8.214) \\ w_t^* &= \arg\max_{1 \leq v \leq V} P_A(T|v) && (8.215) \end{aligned}$$

4. **Backtracking**

$$w_t^* = \psi_{t+1}(w_{t+1}^*) \tag{8.216}$$

8.14 Language Modeling

The word transition probabilities $P(w_{q+1}|W_q)$ express, in a statistical manner, all of the different ways in which the word sequence is constrained by syntax, word meaning, and knowledge of the task being performed. This set of probabilities is called a "statistical language model."

8.14.1 Maximum A Posteriori Recognition

Many applications lend themselves readily to two very different types of language model:

- A low-complexity language model, such as an N-gram or class N-gram model. Using this model, the probability of any word sequence $P(W)$ can be computed very quickly.
- A high-complexity language model. The high-complexity model may take the form of a state transition diagram tailored to the task, and it may even include information about the state of the man-machine dialog.

It is generally not possible to use the high-complexity model to evaluate the likelihood of every word sequence considered by the HMM. Instead, a two-stage recognition process is used:

1. The low-complexity language model is used to evaluate $P(O|W)$ for all possible word sequences, and to identify the best N word sequences.
2. The probability $P(W)$ of each of the N best word sequences is computed using the high-complexity model. The optimum word sequence W^* is then the sequence which maximizes the *a posteriori* probability $P(W|O)$:

$$P(W|O) = \frac{P(O|W)P(W)}{P(O)} \tag{8.217}$$

$$W^* = \arg\max_W P(W|O) = \arg\max_W P(O|W)P(W) \tag{8.218}$$

8.14.2 N-Gram Language Models

The most common language model is a model in which the probability of each word depends only on the $N-1$ preceding words:

$$P(w_q|W_{q-1}) = P(w_q|w_{q-N+1},\ldots,w_{q-1}) \tag{8.219}$$

The N-gram probabilities are typically stored in a lookup table. If N is too large, the lookup table becomes impossible to use in a practical situation. Most coders therefore use either a bigram ($N=2$) or trigram ($N=3$) language model.

The N-gram model may be trained from either speech data or, if you have a text database which reflects the kinds of utterances you expect people to say, it may be trained from text. Training involves counting $N(w_1,w_2,w_3)$, the number of times that word w_3 follows words w_1 and w_2:

$$\bar{P}(w_3|w_1,w_2) = \frac{N(w_1,w_2,w_3)}{N(w_1,w_2)} \tag{8.220}$$

If the number of possible words is large, many valid trigram combinations will be rare or nonexistant in the training data. If the language model is trained using 8.220, trigrams which do not exist in the training data will be marked as impossible, and will never be recognized correctly if they occur in the test data. In order to avoid this problem, the trigram probabilities may be estimated by interpolating the relevant trigram, bigram, and unigram frequencies, as follows:

$$\bar{P}(w_3|w_1,w_2) = p_1\frac{N(w_1,w_2,w_3)}{N(w_1,w_2)} + p_2\frac{N(w_2,w_3)}{N(w_2)} + p_3\frac{N(w_3)}{\sum_{w_3} N(w_3)} \tag{8.221}$$

The interpolation probabilities may vary depending on the word frequencies $N(w_1,w_2,w_3)$, but they should always be normalized so that

$$p_1+p_2+p_3 = 1 \tag{8.222}$$

8.14.3 Perplexity

A speech recognizer with a vocabulary of V words does not need to consider V different possibilities for every new word w_q. Intuitively, the recognizer only needs to consider words for which the language model probability $P(w_q|W_{q-1})$ is large. This intuition can be formalized by defining the entropy of the language model:

$$H_Q = -E[\log_2 P(w_Q|W_{Q-1})] = -\sum_{w_Q=1}^{V} P(w_Q|W_{Q-1}) \log_2 P(w_Q|W_{Q-1}) \tag{8.223}$$

where Q is chosen so that, for the particular language model in question,

$$H_Q = \lim_{q\to\infty} H_q \tag{8.224}$$

The entropy H_Q can be interpreted as a measure of the difficulty encountered by the recognizer in trying to identify each new word. In particular, a recognition task with entropy H_Q can be said to be as difficult as a task in which words are chosen randomly from a vocabulary of B words, where

$$B = 2^{H_Q} \tag{8.225}$$

B is called the "perplexity" or "branching factor" of the language model. It is possible to build quite accurate speech recognizers with very large vocabularies if the task is constrained in such a way that the branching factor B is low.

8.14.4 Class N-Gram

The N-gram model is the most direct language model possible, but unless given infinite training data, it will often miss important language constraints and possibilities. For example, if the training data contains the sentences "cats drink milk" and "dogs swim in water," one might hope that the language model will assign a non-zero probability to the word string "dogs drink water," even if that sentence is not found in the training data.

This kind of generalization is possible if every word w_j is assigned to a word class $C(w_j)$ before training. The language model probabilities are then

$$\bar{P}(w_3|C(w_1), C(w_2)) = \frac{N(C(w_1), C(w_2), C(w_3))}{N(C(w_1), C(w_2))} \tag{8.226}$$

The classes might be syntactic ("noun" and "verb"), or, for a more precise model, they can combine syntactic and semantic information ("animal," "drinkable liquid"). If the classes are designed to fit the application, language models built in this way can be extremely accurate.

8.14.5 Hierarchical Language Models

Phrase Structure

It is possible to create an intuitively pleasing language model by grouping words together into phrases. For example, the sentence

"I would like to fly to San Francisco tomorrow morning." (8.227)

Might be parsed as shown in figure 8.14.5.

In figure 8.14.5, the words have been parsed in several levels:

1. **Word Class**

   Each word w_q can be parsed as belonging to a particular word class c_q (e.g. PRO=pronoun, AUX=auxiliary verb, DAY, TIME, etc.), with the associated probability

$$P(w_q|c_q) \tag{8.228}$$

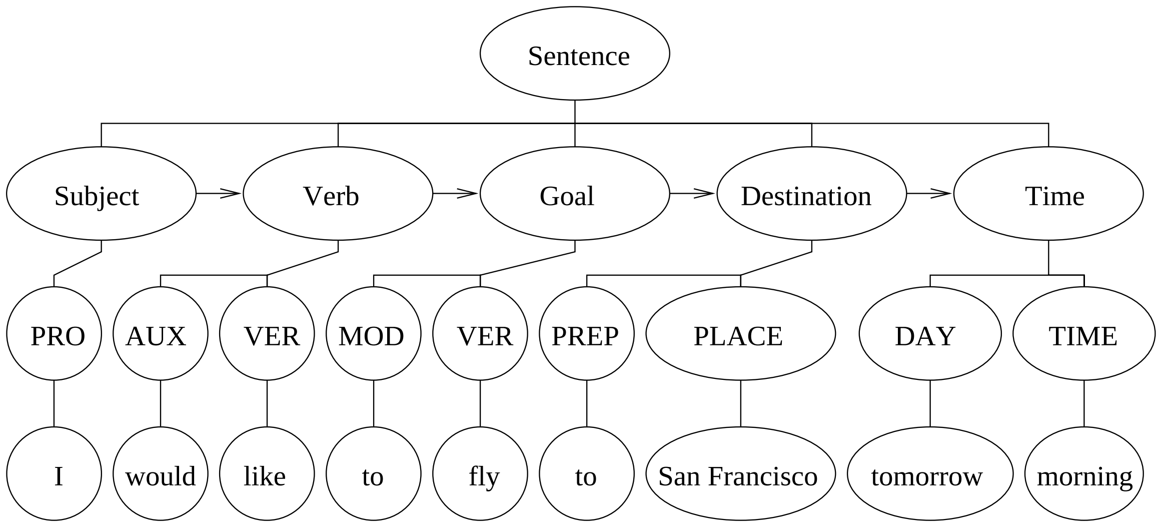

Figure 8.16: A detailed model of word transition probabilities can be created by parsing words into phrases, and phrases into complete sentences.

2. **Phrase Composition**

   Each sequence of word class tokens, $C_n = [\ c_{n,1}, \ldots,\ c_{n,M}]$, can be parsed as being part of a particular phrase, ϕ_n. For example, in the figure above, the phrases are

   $$\Phi = [\phi_1, \ldots,\ \phi_N] = [\ \text{Subject},\ \text{Verb Phrase},\ \text{Goal},\ \text{Place Phrase},\ \text{Time Phrase}\ ] \tag{8.229}$$

   Each of these phrases is mapped to a word-class sequence C with the associated probability

   $$P(C_n = [c_{n,1}, \ldots,\ c_{n,M}] \mid \phi_n) \tag{8.230}$$

   Since the phrase structure of human languages is often recursive (phrases may contain phrases), the computation of $P(C|\phi)$ may also be designed in a recursive fashion; recursion complicates the Viterbi algorithm, but the system designer may decide that the added generality of the language model justifies the added complication. For example, if $C_n = [c_{n,1},\ \phi_{sub}]$, the probability $P(C_n|\phi_n)$ might be computed as:

   $$P(C_n \mid \phi_n) = P([c_{n,1},\ \phi_{sub}] \mid \phi_n) P(C_{sub} \mid \phi_{sub}) \tag{8.231}$$

3. **Sentence Composition**

   The entire phrase sequence composes a "sentence" $\Phi = [\phi_1,\ \phi_2, \ldots,\ \phi_N]$ (since system users do not always speak in grammatical sentences, Φ may or may not be a complete sentence). If the probability of a particular phrase sequence is $P(\Phi)$, then the probability of the word sequence W is

   $$P(W) = P(\Phi) \prod_{n=1}^{N} P(C_n|\phi_n) \prod_{m=1}^{M} P(w_{n,m}|c_{n,m}) \tag{8.232}$$

Training

The language model includes three probabilities: $P(\Phi)$, $P(C|\Phi)$, and $P(w_q|c_q)$. All three probabilities can trained from data using a version of the segmental K-means algorithm:

1. The system designer must make initial estimates of the probabilities $P(\Phi)$, $P(C|\phi)$, and $P(w_q|c_q)$:

   - Since there is a potentially infinite set of phrase sequences Φ and word-class sequences C, the designer must specify the possible sequences at each of these two levels, and assign approximate probabilities for each possible sequence.

- The word-class probabilities $P(w_q|c_q)$ are difficult to control during training, so these probabilities should be set conservatively. For any given word w_q, $P(w_q|c_q)$ should be non-zero for the minimum possible number of word classes (often only one).

2. Based on the initial probabilities, the Viterbi algorithm is used to find the most likely phrase-level and class-level transcriptions, Φ^* and C^*, of each sentence.
3. The model probabilities are re-estimated based on frequency of occurrence.
4. If there has been a change in any of the model probabilities, repeat from step 2.
5. The final phrase-level transcriptions should be checked by hand, to identify mistakes. Mistakes in the final transcription often result from the speaker's use of a grammatical structure which the designer didn't consider possible. Such problems are usually easy to spot, because they will usually prevent the Viterbi algorithm from finding a path with non-zero probability.

8.15 User Interface Design

8.15.1 Dialog Modeling

Hierarchical language models are intuitively appealing, but the improved accuracy of $P(W)$ is often not worth the computational cost of the model. However, hierarchical modeling has other benefits in a system designed for a task such as database access, in which it is important for the system to keep track of the state of a man-machine dialog. Consider the following exchange:

- *(User): I would like to fly to San Francisco tomorrow morning.*
- `(System): From which airport will you be departing?`
- *(User): From Los Angeles.*
- `(System): There are 45 flights from Los Angeles to San Francisco tomorrow morning before noon. On which airline would you like to fly?`

Since the first sentence contains phrases specifying destination, date, and approximate time, the system is able to fill in appropriate blanks in a "dialog frame" which might look like this:

Quality	General	Specific
Day	This Week	Tomorrow
Time	Before 12:00	
Departure Airport		
Arrival Airport	Bay Area	San Francisco
Airline		

By parsing the above table, the system realizes that it needs to know the departure airport before it can access the database, so it asks the user for this piece of information. When the resulting database query provides too many options, a heuristic rule suggests that the system should narrow the query by asking for the name of an airline.

8.15.2 Guidelines for User Interface Design

The objective of good user interface design is to maximize the user's probability of successfully completing his or her task. The user is most likely to be successful if the user knows what the system can do, and if the user only says things which the system can understand.

In order to test algorithms for user interaction, spoken language system designers often employ Wizard of Oz simulations. In a Wizard of Oz simulation, the user believes that he or she is interacting with a completed spoken language system, when in fact the system prompts are generated by a hidden experimenter sitting at a keyboard in another room.

Based on Wizard of Oz simulations, Bernsen, Dybkjaer, and Dybkjaer (Computer, December 1997) formulated rules for spoken-language interaction in seven broad categories, including the following four:

1. **Background Knowledge**

   (a) Ask if the user is familiar with the system. Expert users confronted by repetition of the same old instructions become bored and uncooperative; novice users without sufficient instruction can not use the system.

   (b) Tell novice users clearly what the system can and cannot do, and provide clear and sufficient instructions on how to interact with the system. Goal-directed users will cooperate with the system, if you tell them how to do so.

   (c) Take into account possible (and possibly erroneous) user inferences from related task domains. If a misunderstanding is possible, state clearly the correct interpretation. If you don't know whether the user has some piece of background knowledge, ask (e.g. "Do you know that there is a discount if you stay over Saturday night?")

   (d) Consider, and try to meet, legitimate expectations about the system's knowledge. For example, a system claiming to know about domestic flights should know about all domestic airports.

2. **Informativeness**

   (a) Be explicit in communicating to users the commitments they have made. For example, before a user purchases a ticket, she usually expects a complete review of the details of the ticket.

   (b) Speech recognizers sometimes make mistakes. The easiest way to correct recognition mistakes is by briefly repeating back to the user each new piece of information relevant to the task.

   (c) Allow users to exploit the system's task domain knowledge, e.g. by listing relevant flights so that the user may choose one.

   (d) Avoid interactions that are superfluous or redundant. If the system is too talkative, users will become annoyed and stop using it.

3. **Manner**

   (a) Speak in a manner which is brief, orderly, and unambiguous.

   (b) Provide the same formulation of the same question (or address) everywhere in the system's dialog exchanges. If the system uses different wording to ask for the same piece of information, the user may mistakenly believe that the system is asking for different information.

4. **Repair and Clarification**

   Humans easily shift back and forth between a task-directed communication mode, in which all communication is about the task at hand, and *metacommunication* (communication about communication), used to repair and clarify problems of communication ("What did you say?" "Do you mean Springfield, Massachusetts, or Springfield, Oregon?"). Modern speech recognizers are likely to break down if the user initiates unexpected metacommunication, because the *a priori* probability $P(W)$ of any particular metacommunication is so much lower at all times than the probability of a normal task-oriented communication. Since user-initiated metacommunication is likely to be misunderstood, it is usually better if the system identifies the communication breakdown first, and initiates the required metacommunication.

   (a) Initiate repair metacommunication if system understanding has failed.

   (b) Initiate clarification metacommunication in case of inconsistent user input.

   (c) Initiate clarification metacommunication in case of ambiguous user input.

8.16 Exercises

1. **Recognition Probabilities**

   Days are either Good (G) or So-so (S). The probability that today is a good day depends only on whether or not yesterday was a good day:

   $$P(q_t = G|\ q_{t-1} = G) = 3/4, \quad P(q_t = G|\ q_{t-1} = S) = 1/4 \tag{8.233}$$

   Unfortunately, you have no way of directly measuring whether a given day is Good or So-so. You have noticed, however, that on Good days, the Bomb Shelter is more likely to serve your favorite lunch (filet mignon with fresh asparagus, truffles, and slivered almonds):

   $$P(o_t = \text{filet} \mid q_t = G) = 3/4, \quad P(o_t = \text{filet} \mid q_t = S) = 1/4 \tag{8.234}$$

   You have also noticed that the first day of a new quarter is always a good day:

   $$P(q_1 = G) = 1 \tag{8.235}$$

   Given this model, what is the probability that the Bomb Shelter will serve your favorite lunch for the first two days of a quarter?

8.17 Final Project

The goal of this project is to build, in matlab, an isolated-word recognizer which can recognize the eleven digits ("one" through "nine" plus "zero" and "oh"). Since this is a pretty big project, you will work in teams of four.

8.17.1 Recognizer Specifications

Spectral Processing

You should use a perceptually-based recognizer front end. This may be an MFCC or PLP spectral representation, or an auditory model.

You are strongly encouraged, but not required, to use spectral dynamic information. Spectral dynamics can be incorporated using first and second order cepstral differences or cepstral derivative estimates, as discussed in the text in section 4.6. The RASTA method may also be used as an indirect, perceptually-based method of calculating spectral dynamics.

Observation Densities

Your hidden Markov models should calculate observation probabilities assuming that the observations are continuous random variables.

The type of observation probability model you construct will depend somewhat on your spectral representation. Mixture Gaussian and ANN probability estimators tend to work well with cepstral coefficients, so if your spectral representation is a cepstrum, you can use a mixture Gaussian model (ANN models are hard to implement unless you have previous ANN experience). LPC coefficients can be converted into LPC cepstra and tested using a mixture Gaussian model, or they can be tested directly using the autoregressive output probability estimators discussed in section 6.7 of your text.

In addition to calculating the observation probability density $b_j(o)$, the observation-densities expert should write a function which re-estimates the model parameters for a given state given appropriate training statistics. For example, if you are using a mixture Gaussian model, you should write a function which estimates the parameters c_{jk}, μ_{jk}, and U_{jk} given the observations o_t and the training statistics $\gamma_t(j,k)$ and $\xi_t(i,j)$. You will probably want to use the technique in 6.12.4, or one of the techniques in section 6.13, in order to avoid creating a covariance matrix U_{jk} which is too small.

State Transitions and Recognition Probabilities

The model likelihood, $P(O|\lambda)$, should be calculated using the forward algorithm. In order to avoid floating point underflow, you will almost certainly need to implement probability scaling, as discussed in section 6.12.1 of your text.

You are encouraged but not required to implement some form of explicit state duration probability densities. You may use either the multi-level forward-backward algorithm of section 6.9, or the parallel Viterbi search suggested in section 6.15.3; the parallel Viterbi search requires a lot less time to compute. Either method requires a noticeable amount of additional programming; if your team decides to include a duration model, you may wish to shift some of the other responsibilities from the transitions expert to the training expert, or to some other person.

In addition to calculating the forward and backward parameters $\alpha_t(i)$ and $\beta_t(i)$, the transition-probabilities expert should write a function which re-estimates the transition probabilities π_i and a_{ij} (and the duration probability density $p_i(d)$, if desired) given appropriate training statistics.

You should decide, in advance, which transitions will be allowed. For example, a left-to-right model is probably appropriate for isolated word recognition. Should the model be allowed to skip states? How many?

Model Training

You may train your model using the standard ML approach, or using the MDI or MMI approaches discussed in section 6.10 of your text.

Training the model requires several steps. For example, ML training requires at least the following steps:

1. Load the training data, one utterance at a time, and run the forward and backward algorithms on each utterance.
2. Multiply together the parameters $\alpha_t(i)$ and $\beta_t(i)$ in order to generate training statistics for each sentence, including the functions $\gamma_t(j,k)$ and $\xi_t(i,j)$.
3. Once training statistics have been computed for every training utterance, the statistics should be handed off to functions written by the observations expert and the transitions expert, in order to re-estimate the model parameters.

The exact boundary between the work of the various experts is up to you to define. For example, what training statistics are required for re-estimation of the observation and transition probabilities? How and when should the training statistics for different sentences be combined? How should the model parameters be stored and accessed? All of these questions should be answered in your project outline.

Training Data

Training and test data is all stored in .au files, and can be read using the matlab function `[y, fs, bits]=auread(filename)`. The files are stored using 16-bit words, at a 16kHz sampling rate, so your computer may not be able to play the file without matlab intervention.

The training files are stored in the directories ~ee214a/tidigits/(gender)/(initials)/(digit)a.au, where "gender" is the speaker's gender (m or f), "initials" is a two-character identifier unique to each speaker, and "digit" is the digit (1 through 9, z for "zero," and o for "oh").

There are 112 speakers, and you have been given one token of each digit as spoken by each speaker (a total of 1232 training tokens). That should be plenty for good quality training; in fact, you may decide to reduce your training time by using only part of the training data.

Since part of your grade will depend on your recognizer's ability to generalize to speakers who are not in the training data, you may want to check this ability during the training process. The best way to test generalization is by training on, say, 100 of the training speakers, and then testing your recognizer on the remaining 12.

8.17.2 The Design Process

You are expected to work as a team to design the recognizer. You may write code independently, if you prefer, or you may write all of your code in committee fashion, with four people sitting around the workstation commenting while one person types. In either case, you should think early and often about how to integrate the various bits of code into a working recognizer.

Project Outline

The first stage in the recognizer design is a project outline, which should be written jointly by all members of the team. The outline should specify:

1. What functions will you write?
   (a) What will be the inputs and outputs of each function?
   (b) How will all of the functions be combined in order to recognize a new word? Which functions are called by which other functions, and when?
   (c) How will all of the functions be combined in order to train the models?
2. What are the model parameters?
   (a) What kind of spectral representation will you use? What will be your frame length? Will you use any dynamic spectral information? What is the total dimension of the observation vector o_t?
   (b) What kind of observation densities will you use? What are the required model parameters?
   (c) How many states are there in each word model? Which transitions are allowed?

(d) Will you use explicit state durations? How will they be computed?

(e) How will the model parameters be stored and accessed?

3. What is each team member's area of expertise? It need not be the case that each function is the responsibility of just one team member; if a function includes code from two or more areas of expertise, then all of those team members will be considered "responsible" for the function. Also, remember that everybody is responsible for the integration of functions into a working recognizer.

Mathematical Summaries

When you hand in your project outline, you should also hand in mathematical summaries of the algorithms in each area of expertise. These may be formatted as subsections of the project outline.

Each team member (with the help of the other team members) should explain, via text and equations, how his or her inputs will be converted into appropriate outputs:

- The spectral representations expert should show how input speech signals will be converted into spectral and dynamic spectral vectors.
- The observations expert should show how $b_j(o)$ is calculated, and how to re-estimate the observation model parameters given appropriate training statistics.
- The transitions expert should show how to calculate $\alpha_t(i)$ and $\beta_t(i)$, how to scale them to avoid underflow, how to re-estimate π_i and a_{ij}, and how to calculate and re-estimate duration probabilities (if desired).
- The training expert should show how to calculate the training statistics needed in order to re-estimate the model parameters.

The break-down above assumes that you will divide the problem of recognition and training exactly as specified in section 8.17.1, but this is not required. If you want to divide the problem differently, just decide among yourselves how to do it, and then write mathematical summaries which show your own idea of the best division of labor.

Oral Presentations

The first hour of class on 6/11/98 will consist of oral presentations by each team, describing the algorithms used to perform training and recognition.

Presentations should be planned by the team as a group, so that the entire recognition process is described. Each team member should present for no more than five minutes, leaving ten minutes at the end for questions. Your presentation does not have to be about your area of expertise, but you should be ready to answer questions about your area of expertise.

Recognizer Competition

For the last half hour or so of class on 6/11/98, we will move upstairs to the speech lab, and test your recognizers on eleven randomly chosen training words and eleven new words not present in the training data.

Please have a master recognizer routine ready, of the form

```
digit = recognize(filename)                    (8.236)
```

where "digit" is a digit between 1 and 9, or one of the letters "o" or "z," and filename is the name of a 16kHz audio file (.au format).

Since HMMs are not always fast (especially in matlab), the competition may run over the end of class time. You are welcome to leave at the end of class, but please be sure that all of your code is in one place, ready to run with or without your help.

Matlab Code

All matlab code for the recognizer should be turned in together, in one package for each team. Comments at the beginning of each function should describe:

- A brief summary of what the function intends to do.
- A brief description of each of the inputs and outputs.
- A description of how the function fits into your recognition and training hierarchies, i.e. which other functions call this one, and when.
- A brief description (in words) of the algorithm used.
- A list of authors.

Write-Ups

The entire team should work together on a set of four sub-reports, written in conference-paper style (i.e. 4-5 pages per paper, preferably in two-column format). All members of the team should be listed as authors on each paper.

Each member of the team should be listed as first author on the paper which describes algorithms in his or her area of expertise.

Papers should include words and equations to describe the algorithms you wrote, and figures and tables as appropriate to describe the results. Papers should not include any matlab code.