PRELIMINARY PROOFS.
Unpublished Work ©2008 by Pearson Education, Inc. To be published by Pearson Prentice Hall, Pearson Education, Inc., Upper Saddle River, New Jersey. All rights reserved. Permission to use this unpublished Work is granted to individuals registering through Melinda_Haggerty@prenhall.com for the instructional purposes not exceeding one academic term or semester.

Chapter 9
Automatic Speech Recognition

When Frederic was a little lad he proved so brave and daring,
His father thought he'd 'prentice him to some career seafaring.
I was, alas! his nurs'rymaid, and so it fell to my lot
To take and bind the promising boy apprentice to a **pilot** —
A life not bad for a hardy lad, though surely not a high lot,
Though I'm a nurse, you might do worse than make your boy a pilot.
I was a stupid nurs'rymaid, on breakers always steering,
And I did not catch the word aright, through being hard of hearing;
Mistaking my instructions, which within my brain did gyrate,
I took and bound this promising boy apprentice to a **pirate.**

The Pirates of Penzance, Gilbert and Sullivan, 1877

Alas, this mistake by nurserymaid Ruth led to Frederic's long indenture as a pirate and, due to a slight complication involving 21st birthdays and leap years, nearly led to 63 extra years of apprenticeship. The mistake was quite natural, in a Gilbert-and-Sullivan sort of way; as Ruth later noted, "The two words were so much alike!" True, true; spoken language understanding is a difficult task, and it is remarkable that humans do as well at it as we do. The goal of **automatic speech recognition** (**ASR**) research is to address this problem computationally by building systems that map from an acoustic signal to a string of words. **Automatic speech understanding** (**ASU**) extends this goal to producing some sort of understanding of the sentence, rather than just the words.

ASR

The general problem of automatic transcription of speech by any speaker in any environment is still far from solved. But recent years have seen ASR technology mature to the point where it is viable in certain limited domains. One major application area is in human-computer interaction. While many tasks are better solved with visual or pointing interfaces, speech has the potential to be a better interface than the keyboard for tasks where full natural language communication is useful, or for which keyboards are not appropriate. This includes hands-busy or eyes-busy applications, such as where the user has objects to manipulate or equipment to control. Another important application area is telephony, where speech recognition is already used for example in spoken dialogue systems for entering digits, recognizing "yes" to accept collect calls, finding out airplane or train information, and call-routing ("Accounting, please", "Prof. Regier, please"). In some applications, a multimodal interface combining speech and pointing can be more efficient than a graphical user interface without speech (Cohen et al., 1998). Finally, ASR is applied to dictation, that is, transcription of extended monologue by a single specific speaker. Dictation is common in fields such as law and is also important as part of augmentative communication (interaction between computers and humans with some disability resulting in the inability to type, or the inability to speak). The blind Milton famously dictated *Paradise Lost* to his daughters, and Henry James dictated his later novels after a repetitive stress injury.

Before turning to architectural details, let's discuss some of the parameters of the speech recognition task. One dimension of variation in speech recognition tasks is

DRAFT

the vocabulary size. Speech recognition is easier if the number of distinct words we need to recognize is smaller. So tasks with a two word vocabulary, like *yes* versus *no* detection, or an eleven word vocabulary, like recognizing sequences of digits, in what is called the **digits task** task, are relatively easy. On the other end, tasks with large vocabularies, like transcribing human-human telephone conversations, or transcribing broadcast news, tasks with vocabularies of 64,000 words or more, are much harder.

Digit recognition

A second dimension of variation is how fluent, natural, or conversational the speech is. **Isolated word** recognition, in which each word is surrounded by some sort of pause, is much easier than recognizing **continuous speech**, in which words run into each other and have to be segmented. Continuous speech tasks themselves vary greatly in difficulty. For example, human-to-machine speech turns out to be far easier to recognize than human-to-human speech. That is, recognizing speech of humans talking to machines, either reading out loud in **read speech** (which simulates the dictation task), or conversing with speech dialogue systems, is relatively easy. Recognizing the speech of two humans talking to each other, in **conversational speech** recognition, for example for transcribing a business meeting or a telephone conversation, is much harder. It seems that when humans talk to machines, they simplify their speech quite a bit, talking more slowly and more clearly.

Isolated word

Continuous speech

Read speech

Conversational speech

A third dimension of variation is channel and noise. The **dictation** task (and much laboratory research in speech recognition) is done with high quality, head mounted microphones. Head mounted microphones eliminate the distortion that occurs in a table microphone as the speaker's head moves around. Noise of any kind also makes recognition harder. Thus recognizing a speaker dictating in a quiet office is much easier than recognizing a speaker in a noisy car on the highway with the window open.

A final dimension of variation is accent or speaker-class characteristics. Speech is easier to recognize if the speaker is speaking a standard dialect, or in general one that matches the data the system was trained on. Recognition is thus harder on foreign-accented speech, or speech of children (unless the system was specifically trained on exactly these kinds of speech).

Table 9.1 shows the rough percentage of incorrect words (the **word error rate**, or WER, defined on page 330) from state-of-the-art systems on different ASR tasks.

Task	**Vocabulary**	**Error Rate %**
TI Digits	11 (zero-nine, oh)	.5
Wall Street Journal read speech	5,000	3
Wall Street Journal read speech	20,000	3
Broadcast News	64,000+	10
Conversational Telephone Speech (CTS)	64,000+	20

Figure 9.1 Rough word error rates (% of words misrecognized) reported around 2006 for ASR on various tasks; the error rates for Broadcast News and CTS are based on particular training and test scenarios and should be taken as ballpark numbers; error rates for differently defined tasks may range up to a factor of two.

Variation due to noise and accent increases the error rates quite a bit. The word error rate on strongly Japanese-accented or Spanish accented English has been reported to be about 3 to 4 times higher than for native speakers on the same task (Tomokiyo, 2001).

DRAFT

And adding automobile noise with a 10dB SNR (signal-to-noise ratio) can cause error rates to go up by 2 to 4 times.

In general, these error rates go down every year, as speech recognition performance has improved quite steadily. One estimate is that performance has improved roughly 10 percent a year over the last decade (Deng and Huang, 2004), due to a combination of algorithmic improvements and Moore's law.

LVCSR

While the algorithms we describe in this chapter are applicable across a wide variety of these speech tasks, we chose to focus this chapter on the fundamentals of one crucial area: **Large-Vocabulary Continuous Speech Recognition** (**LVCSR**). Large-vocabulary generally means that the systems have a vocabulary of roughly 20,000 to 60,000 words. We saw above that **continuous** means that the words are run together naturally. Furthermore, the algorithms we will discuss are generally **speaker-independent**; that is, they are able to recognize speech from people whose speech the system has never been exposed to before.

Speaker independent

The dominant paradigm for LVCSR is the HMM, and we will focus on this approach in this chapter. Previous chapters have introduced most of the core algorithms used in HMM-based speech recognition. Ch. 7 introduced the key phonetic and phonological notions of **phone**, **syllable**, and intonation. Ch. 5 and Ch. 6 introduced the use of the **Bayes rule**, the **Hidden Markov Model** (**HMM**), the **Viterbi** algorithm, and the Baum-Welch training algorithm. Ch. 4 introduced the N**-gram** language model and the **perplexity** metric. In this chapter we begin with an overview of the architecture for HMM speech recognition, offer an all-too-brief overview of signal processing for feature extraction and the extraction of the important MFCC features, and then introduce Gaussian acoustic models. We then continue with how Viterbi decoding works in the ASR context, and give a complete summary of the training procedure for ASR, called **embedded training**. Finally, we introduce word error rate, the standard evaluation metric. The next chapter will continue with some advanced ASR topics.

9.1 Speech Recognition Architecture

Noisy channel

The task of speech recognition is to take as input an acoustic waveform and produce as output a string of words. HMM-based speech recognition systems view this task using the metaphor of the noisy channel. The intuition of the **noisy channel** model (see Fig. 9.2) is to treat the acoustic waveform as an "noisy" version of the string of words, i.e.. a version that has been passed through a noisy communications channel. This channel introduces "noise" which makes it hard to recognize the "true" string of words. Our goal is then to build a model of the channel so that we can figure out how it modified this "true" sentence and hence recover it.

The insight of the noisy channel model is that if we know how the channel distorts the source, we could find the correct source sentence for a waveform by taking every possible sentence in the language, running each sentence through our noisy channel model, and seeing if it matches the output. We then select the best matching source sentence as our desired source sentence.

Implementing the noisy-channel model as we have expressed it in Fig. 9.2 requires

DRAFT

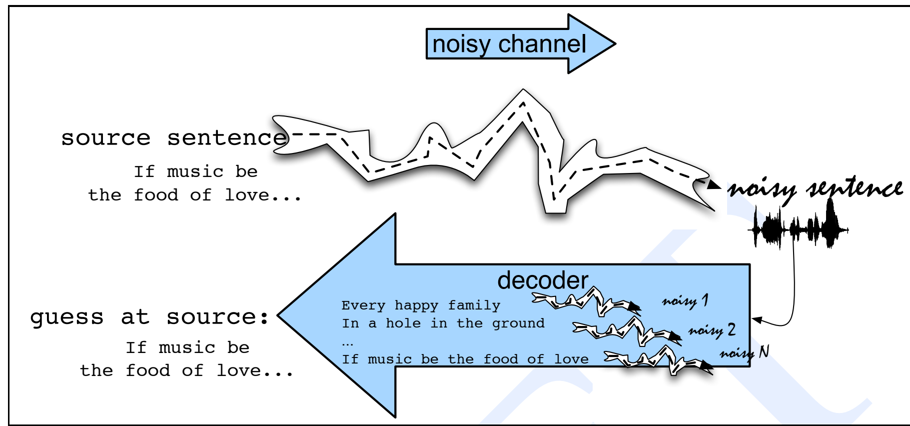

Figure 9.2 The noisy channel model. We search through a huge space of potential "source" sentences and choose the one which has the highest probability of generating the "noisy" sentence. We need models of the prior probability of a source sentence (N-grams), the probability of words being realized as certain strings of phones (HMM lexicons), and the probability of phones being realized as acoustic or spectral features (Gaussian Mixture Models).

solutions to two problems. First, in order to pick the sentence that best matches the noisy input we will need a complete metric for a "best match". Because speech is so variable, an acoustic input sentence will never exactly match any model we have for this sentence. As we have suggested in previous chapters, we will use probability as our metric. This makes the speech recognition problem a special case of **Bayesian inference**, a method known since the work of Bayes (1763). Bayesian inference or Bayesian classification was applied successfully by the 1950s to language problems like optical character recognition (Bledsoe and Browning, 1959) and to authorship attribution tasks like the seminal work of Mosteller and Wallace (1964) on determining the authorship of the Federalist papers. Our goal will be to combine various probabilistic models to get a complete estimate for the probability of a noisy acoustic observation-sequence given a candidate source sentence. We can then search through the space of all sentences, and choose the source sentence with the highest probability.

Bayesian

Second, since the set of all English sentences is huge, we need an efficient algorithm that will not search through all possible sentences, but only ones that have a good chance of matching the input. This is the **decoding** or **search** problem, which we have already explored with the Viterbi decoding algorithm for HMMs in Ch. 5 and Ch. 6. Since the search space is so large in speech recognition, efficient search is an important part of the task, and we will focus on a number of areas in search.

In the rest of this introduction we will review the probabilistic or Bayesian model for speech recognition that we introduced for part-of-speech tagging in Ch. 5. We then introduce the various components of a modern HMM-based ASR system.

Recall that the goal of the probabilistic noisy channel architecture for speech recognition can be summarized as follows:

> *"What is the most likely sentence out of all sentences in the language $\mathscr{L}$ given some acoustic input O?"*

DRAFT

We can treat the acoustic input O as a sequence of individual "symbols" or "observations" (for example by slicing up the input every 10 milliseconds, and representing each slice by floating-point values of the energy or frequencies of that slice). Each index then represents some time interval, and successive o_i indicate temporally consecutive slices of the input (note that capital letters will stand for sequences of symbols and lower-case letters for individual symbols):

$$O = o_1, o_2, o_3, \ldots, o_t \tag{9.1}$$

Similarly, we treat a sentence as if it were composed of a string of words:

$$W = w_1, w_2, w_3, \ldots, w_n \tag{9.2}$$

Both of these are simplifying assumptions; for example dividing sentences into words is sometimes too fine a division (we'd like to model facts about groups of words rather than individual words) and sometimes too gross a division (we need to deal with morphology). Usually in speech recognition a word is defined by orthography (after mapping every word to lower-case): *oak* is treated as a different word than *oaks*, but the auxiliary *can* ("can you tell me...") is treated as the same word as the noun *can* ("i need a can of...").

The probabilistic implementation of our intuition above, then, can be expressed as:

$$\hat{W} = \underset{W \in \mathscr{L}}{\operatorname{argmax}} P(W|O) \tag{9.3}$$

Recall that the function $\operatorname{argmax}_x f(x)$ means "the x such that f(x) is largest". Eq. 9.3 is guaranteed to give us the optimal sentence W; we now need to make the equation operational. That is, for a given sentence W and acoustic sequence O we need to compute $P(W|O)$. Recall that given any probability $P(x|y)$, we can use Bayes' rule to break it down as follows:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{9.4}$$

We saw in Ch. 5 that we can substitute (9.4) into (9.3) as follows:

$$\hat{W} = \underset{W \in \mathscr{L}}{\operatorname{argmax}} \frac{P(O|W)P(W)}{P(O)} \tag{9.5}$$

The probabilities on the right-hand side of (9.5) are for the most part easier to compute than $P(W|O)$. For example, $P(W)$, the prior probability of the word string itself is what is estimated by the N-gram language models of Ch. 4. And we will see below that $P(O|W)$ turns out to be easy to estimate as well. But $P(O)$, the probability of the acoustic observation sequence, is harder to estimate. Luckily, we can ignore $P(O)$ just as we saw in Ch. 5. Why? Since we are maximizing over all possible sentences, we will be computing $\frac{P(O|W)P(W)}{P(O)}$ for each sentence in the language. But $P(O)$ doesn't change for each sentence! For each potential sentence we are still examining the same observations O, which must have the same probability $P(O)$. Thus:

$$\hat{W} = \underset{W \in \mathscr{L}}{\operatorname{argmax}} \frac{P(O|W)P(W)}{P(O)} = \underset{W \in \mathscr{L}}{\operatorname{argmax}} P(O|W)P(W) \tag{9.6}$$

DRAFT

To summarize, the most probable sentence W given some observation sequence O can be computed by taking the product of two probabilities for each sentence, and choosing the sentence for which this product is greatest. The general components of the speech recognizer which compute these two terms have names; $P(W)$, the **prior probability**, is computed by the **language model**. while $P(O|W)$, the **observation likelihood**, is computed by the **acoustic model**.

Language model

Acoustic model

$$\hat{W} = \underset{W \in \mathscr{L}}{\operatorname{argmax}} \overbrace{P(O|W)}^{\text{likelihood}} \; \overbrace{P(W)}^{\text{prior}} \tag{9.7}$$

The language model (LM) prior $P(W)$ expresses how likely a given string of words is to be a source sentence of English. We have already seen in Ch. 4 how to compute such a language model prior $P(W)$ by using N-gram grammars. Recall that an N-gram grammar lets us assign a probability to a sentence by computing:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k | w_{k-N+1}^{k-1}) \tag{9.8}$$

This chapter will show how the HMM we covered in Ch. 6 can be used to build an Acoustic Model (AM) which computes the likelihood $P(O|W)$. Given the AM and LM probabilities, the probabilistic model can be operationalized in a search algorithm so as to compute the maximum probability word string for a given acoustic waveform. Fig. 9.3 shows the components of an HMM speech recognizer as it processes a single utterance, indicating the computation of the prior and likelihood. The figure shows the recognition process in three stages. In the **feature extraction** or **signal processing** stage, the acoustic waveform is sampled into **frames** (usually of 10, 15, or 20 milliseconds) which are transformed into **spectral features**. Each time window is thus represented by a vector of around 39 features representing this spectral information as well as information about energy and spectral change. Sec. 9.3 gives an (unfortunately brief) overview of the feature extraction process.

In the **acoustic modeling** or **phone recognition** stage, we compute the likelihood of the observed spectral feature vectors given linguistic units (words, phones, subparts of phones). For example, we use Gaussian Mixture Model (GMM) classifiers to compute for each HMM state q, corresponding to a phone or subphone, the likelihood of a given feature vector given this phone $p(o|q)$. A (simplified) way of thinking of the output of this stage is as a sequence of probability vectors, one for each time frame, each vector at each time frame containing the likelihoods that each phone or subphone unit generated the acoustic feature vector observation at that time.

Finally, in the **decoding** phase, we take the acoustic model (AM), which consists of this sequence of acoustic likelihoods, plus an HMM dictionary of word pronunciations, combined with the language model (LM) (generally an N-gram grammar), and output the most likely sequence of words. An HMM dictionary, as we will see in Sec. 9.2, is a list of word pronunciations, each pronunciation represented by a string of phones. Each word can then be thought of as an HMM, where the phones (or sometimes subphones) are states in the HMM, and the Gaussian likelihood estimators supply the HMM output likelihood function for each state. Most ASR systems use the Viterbi algorithm for

decoding, speeding up the decoding with wide variety of sophisticated augmentations such as pruning, fast-match, and tree-structured lexicons.

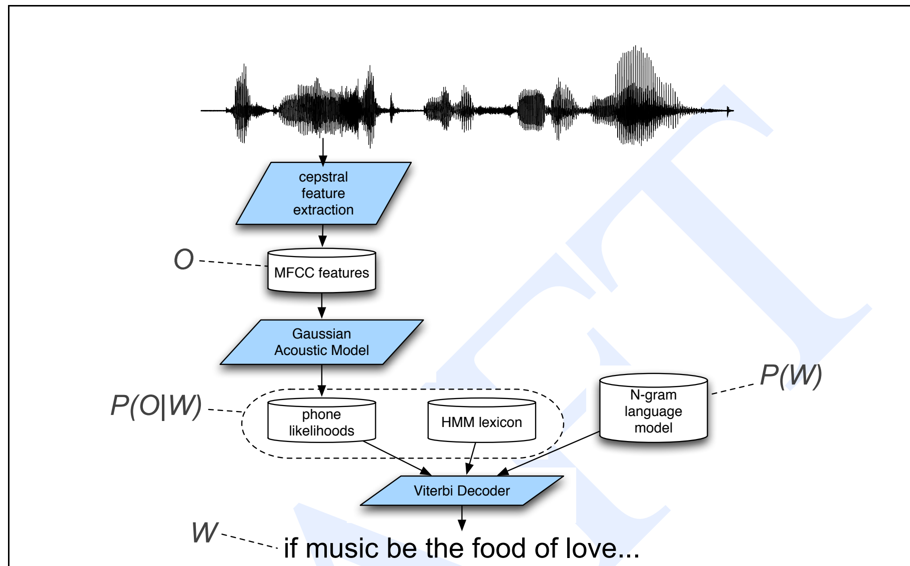

Figure 9.3 Schematic architecture for a (simplified) speech recognizer decoding a single sentence. A real recognizer is more complex since various kinds of pruning and fast matches are needed for efficiency. This architecture is only for decoding; we also need a separate architecture for training parameters.

9.2 Applying the Hidden Markov Model to Speech

Let's turn now to how the HMM model is applied to speech recognition. We saw in Ch. 6 that a Hidden Markov Model is characterized by the following components:

$Q = q_1 q_2 \ldots q_N$	a set of **states**
$A = a_{01} a_{02} \ldots a_{n1} \ldots a_{nn}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$
$O = o_1 o_2 \ldots o_N$	a set of **observations**, each one drawn from a vocabulary $V = v_1, v_2, ..., v_V$.
$B = b_i(o_t)$	A set of **observation likelihoods:**, also called **emission probabilities**, each expressing the probability of an observation o_t being generated from a state i.
q_0, q_{end}	a special **start and end state** which are not associated with observations.

Furthermore, the chapter introduced the **Viterbi** algorithm for decoding HMMs, and the **Baum-Welch** or **Forward-Backward** algorithm for training HMMs.

All of these facets of the HMM paradigm play a crucial role in ASR. We begin here by discussing how the states, transitions, and observations map into the speech recognition task. We will return to the ASR applications of Viterbi decoding in Sec. 9.6. The extensions to the Baum-Welch algorithms needed to deal with spoken language are covered in Sec. 9.4 and Sec. 9.7.

Recall the examples of HMMs we saw earlier in the book. In Ch. 5, the hidden states of the HMM were parts-of-speech, the observations were words, and the HMM decoding task mapped a sequence of words to a sequence of parts-of-speech. In Ch. 6, the hidden states of the HMM were weather, the observations were 'ice-cream consumptions', and the decoding task was to determine the weather sequence from a sequence of ice-cream consumption. For speech, the hidden states are phones, parts of phones, or words, each observation is information about the spectrum and energy of the waveform at a point in time, and the decoding process maps this sequence of acoustic information to phones and words.

The observation sequence for speech recognition is a sequence of **acoustic feature vectors**. Each acoustic feature vector represents information such as the amount of energy in different frequency bands at a particular point in time. We will return in Sec. 9.3 to the nature of these observations, but for now we'll simply note that each observation consists of a vector of 39 real-valued features indicating spectral information. Observations are generally drawn every 10 milliseconds, so 1 second of speech requires 100 spectral feature vectors, each vector of length 39.

The hidden states of Hidden Markov Models can be used to model speech in a number of different ways. For small tasks, like **digit recognition**, (the recognition of the 10 digit words *zero* through *nine*), or for **yes-no** recognition (recognition of the two words **yes** and **no**), we could build an HMM whose states correspond to entire words. For most larger tasks, however, the hidden states of the HMM correspond to phone-like units, and words are sequences of these phone-like units.

Let's begin by describing an HMM model in which each state of an HMM corresponds to a single phone (if you've forgotten what a phone is, go back and look again at the definition in Ch. 7). In such a model, a word HMM thus consists of a sequence of HMM states concatenated together. Fig. 9.4 shows a schematic of the structure of a basic phone-state HMM for the word *six*.

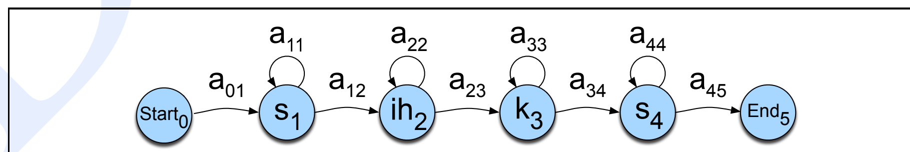

Figure 9.4 An HMM for the word *six*, consisting of four emitting states, two non-emitting states, and the transition probabilities A. The the observation probabilities *B* are not shown.

Note that only certain connections between phones exist in Fig. 9.4. In the HMMs described in Ch. 6, there were arbitrary transitions between states; any state could transition to any other. This was also in principle true of the HMMs for part-of-speech tagging in Ch. 5; although the probability of some tag transitions was low, any tag

DRAFT

could in principle follow any other tag. Unlike other HMM applications, HMM models for speech recognition do not allow arbitrary transitions. Instead, they place strong constraints on transitions based on the sequential nature of speech. Except in unusual cases, HMMs for speech don't allow transitions from states to go to earlier states in the word; in other words, states can transition to themselves or to successive states. As we saw in Ch. 6, this kind of **left-to-right** HMM structure is called a **Bakis network**.

Bakis network

The most common model used for speech, illustrated in a simplified form in Fig. 9.4 is even more constrained, allowing a state to transition only to itself (self-loop) or to a single succeeding state. The use of self-loops allows a single phone to repeat so as to cover a variable amount of the acoustic input. Phone durations vary hugely, dependent on the phone identify, the speaker's rate of speech, the phonetic context, and the level of prosodic prominence of the word. Looking at the Switchboard corpus, the phone [aa] varies in length from 7 to 387 milliseconds (1 to 40 frames), while the phone [z] varies in duration from 7 milliseconds to more than 1.3 seconds (130 frames) in some utterances! Self-loops thus allow a single state to be repeated many times.

For very simple speech tasks (recognizing small numbers of words such as the 10 digits), using an HMM state to represent a phone is sufficient. In general LVCSR tasks, however, a more fine-grained representation is necessary. This is because phones can last over 1 second, i.e., over 100 frames, but the 100 frames are not acoustically identical. The spectral characteristics of a phone, and the amount of energy, vary dramatically across a phone. For example, recall from Ch. 7 that stop consonants have a closure portion, which has very little acoustic energy, followed by a release burst. Similarly, diphthongs are vowels whose F1 and F2 change significantly. Fig. 9.5 shows these large changes in spectral characteristics over time for each of the two phones in the word "Ike", ARPAbet [ay k].

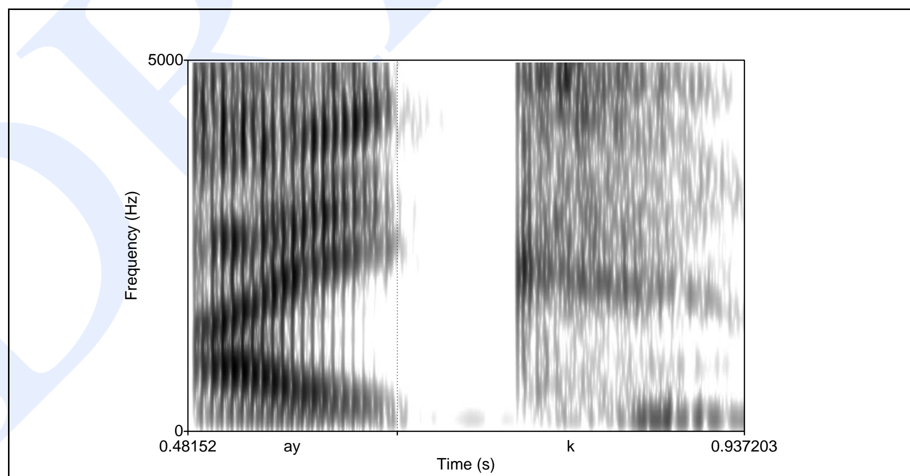

Figure 9.5 The two phones of the word "Ike", pronounced [ay k]. Note the continuous changes in the [ay] vowel on the left, as F2 rises and F1 falls, and the sharp differences between the silence and release parts of the [k] stop.

To capture this fact about the non-homogeneous nature of phones over time, in

LVCSR we generally model a phone with more than one HMM state. The most common configuration is to use three HMM states, a beginning, middle, and end state. Each phone thus consists of 3 emitting HMM states instead of one (plus two non-emitting states at either end), as shown in Fig. 9.6. It is common to reserve the word **model** or **phone model** to refer to the entire 5-state phone HMM, and use the word **HMM state** (or just **state** for short) to refer to each of the 3 individual subphone HMM states.

Phone model

HMM state

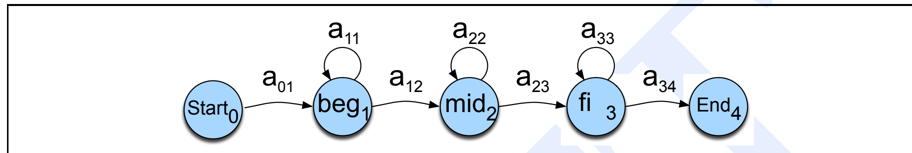

Figure 9.6 A standard 5-state HMM model for a phone, consisting of three emitting states (corresponding to the transition-in, steady state, and transition-out regions of the phone) and two non-emitting states.

To build a HMM for an entire word using these more complex phone models, we can simply replace each phone of the word model in Fig. 9.4 with a 3-state phone HMM. We replace the non-emitting start and end states for each phone model with transitions directly to the emitting state of the preceding and following phone, leaving only two non-emitting states for the entire word. Fig. 9.7 shows the expanded word.

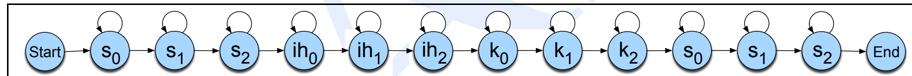

Figure 9.7 A composite word model for "six", [s ih k s], formed by concatenating four phone models, each with three emitting states.

In summary, an HMM model of speech recognition is parameterized by:

$Q = q_1 q_2 \ldots q_N$	a set of **states** corresponding to **subphones**
$A = a_{01}a_{02}\ldots a_{n1}\ldots a_{nn}$	a **transition probability matrix** A, each a_{ij} representing the probability for each subphone of taking a **self-loop** or going to the next subphone.
$B = b_i(o_t)$	A set of **observation likelihoods:**, also called **emission probabilities**, each expressing the probability of a cepstral feature vector (observation o_t) being generated from subphone state i.

Another way of looking at the A probabilities and the states Q is that together they represent a **lexicon**: a set of pronunciations for words, each pronunciation consisting of a set of subphones, with the order of the subphones specified by the transition probabilities A.

We have now covered the basic structure of HMM states for representing phones and words in speech recognition. Later in this chapter we will see further augmentations of the HMM word model shown in Fig. 9.7, such as the use of triphone models which make use of phone context, and the use of special phones to model silence. First,

though, we need to turn to the next component of HMMs for speech recognition: the observation likelihoods. And in order to discuss observation likelihoods, we first need to introduce the actual acoustic observations: feature vectors. After discussing these in Sec. 9.3, we turn in Sec. 9.4 the acoustic model and details of observation likelihood computation. We then re-introduce Viterbi decoding and show how the acoustic model and language model are combined to choose the best sentence.

9.3 Feature Extraction: MFCC vectors

Feature vector

MFCC

Cepstrum

Our goal in this section is to describe how we transform the input waveform into a sequence of acoustic **feature vectors**, each vector representing the information in a small time window of the signal. While there are many possible such feature representations, by far the most common in speech recognition is the **MFCC**, the **mel frequency cepstral coefficients**. These are based on the important idea of the **cepstrum**. We will give a relatively high-level description of the process of extraction of MFCCs from a waveform; we strongly encourage students interested in more detail to follow up with a speech signal processing course.

Sampling

Sampling rate

Nyquist frequency

Telephone-bandwidth

Wideband

We begin by repeating from Sec. 7.4.2 the process of digitizing and quantizing an analog speech waveform. Recall that the first step in processing speech is to convert the analog representations (first air pressure, and then analog electric signals in a microphone), into a digital signal. This process of **analog-to-digital conversion** has two steps: **sampling** and **quantization**. A signal is sampled by measuring its amplitude at a particular time; the **sampling rate** is the number of samples taken per second. In order to accurately measure a wave, it is necessary to have at least two samples in each cycle: one measuring the positive part of the wave and one measuring the negative part. More than two samples per cycle increases the amplitude accuracy, but less than two samples will cause the frequency of the wave to be completely missed. Thus the maximum frequency wave that can be measured is one whose frequency is half the sample rate (since every cycle needs two samples). This maximum frequency for a given sampling rate is called the **Nyquist frequency**. Most information in human speech is in frequencies below 10,000 Hz; thus a 20,000 Hz sampling rate would be necessary for complete accuracy. But telephone speech is filtered by the switching network, and only frequencies less than 4,000 Hz are transmitted by telephones. Thus an 8,000 Hz sampling rate is sufficient for **telephone-bandwidth** speech like the Switchboard corpus. A 16,000 Hz sampling rate (sometimes called **wideband**) is often used for microphone speech.

Quantization

Even an 8,000 Hz sampling rate requires 8000 amplitude measurements for each second of speech, and so it is important to store the amplitude measurement efficiently. They are usually stored as integers, either 8-bit (values from -128–127) or 16 bit (values from -32768–32767). This process of representing real-valued numbers as integers is called **quantization** because there is a minimum granularity (the quantum size) and all values which are closer together than this quantum size are represented identically.

We refer to each sample in the digitized quantized waveform as $x[n]$, where n is an index over time. Now that we have a digitized, quantized representation of the

DRAFT

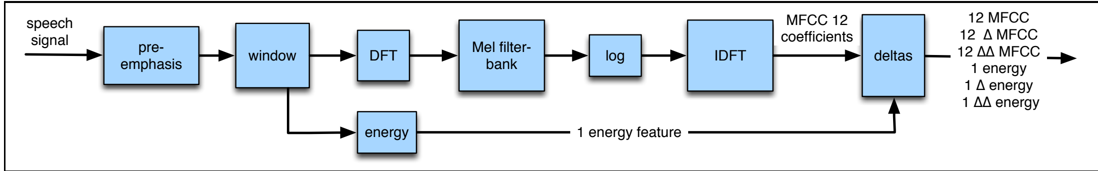

Figure 9.8 Extracting a sequence of 39-dimensional MFCC feature vectors from a quantized digitized waveform

waveform, we are ready to extract MFCC features. The seven steps of this process are shown in Fig. 9.8 and individually described in each of the following sections.

9.3.1 Preemphasis

The first stage in MFCC feature extraction is to boost the amount of energy in the high frequencies. It turns out that if we look at the spectrum for voiced segments like vowels, there is more energy at the lower frequencies than the higher frequencies. This drop in energy across frequencies (which is called **spectral tilt**) is caused by the nature of the glottal pulse. Boosting the high frequency energy makes information from these higher formants more available to the acoustic model and improves phone detection accuracy.

Spectral tilt

This preemphasis is done by using a filter[1] Fig. 9.9 shows an example of a spectral slice from the first author's pronunciation of the single vowel [aa] before and after preemphasis.

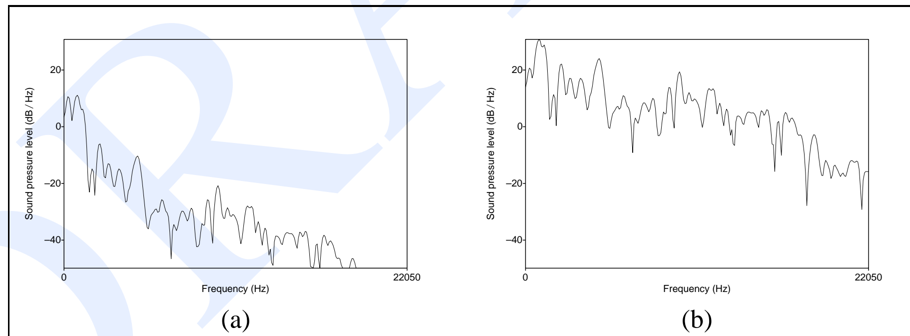

Figure 9.9 A spectral slice from the vowel [aa] before (a) and after (b) preemphasis.

9.3.2 Windowing

Recall that the goal of feature extraction is to provide spectral features that can help us build phone or subphone classifiers. We therefore don't want to extract our spectral features from an entire utterance or conversation, because the spectrum changes very quickly. Technically, we say that speech is a **non-stationary** signal, meaning that its statistical properties are not constant across time. Instead, we want to extract spectral

Non-stationary

[1] For students who have had signal processing: this preemphasis filter is a first-order high-pass filter. In the time domain, with input $x[n]$ and $0.9 \leq \alpha \leq 1.0$, the filter equation is $y[n] = x[n] - \alpha x[n-1]$.

features from a small **window** of speech that characterizes a particular subphone and for which we can make the (rough) assumption that the signal is **stationary** (i.e. its statistical properties are constant within this region).

Stationary

We'll do this by using a window which is non-zero inside some region and zero elsewhere, running this window across the speech signal, and extracting the waveform inside this window.

We can characterize such a windowing process by three parameters: how **wide** is the window (in milliseconds), what is the **offset** between successive windows, and what is the **shape** of the window. We call the speech extracted from each window a **frame**, and we call the number of milliseconds in the frame the **frame size** and the number of milliseconds between the left edges of successive windows the **frame shift**.

Frame
Frame size
Frame shift

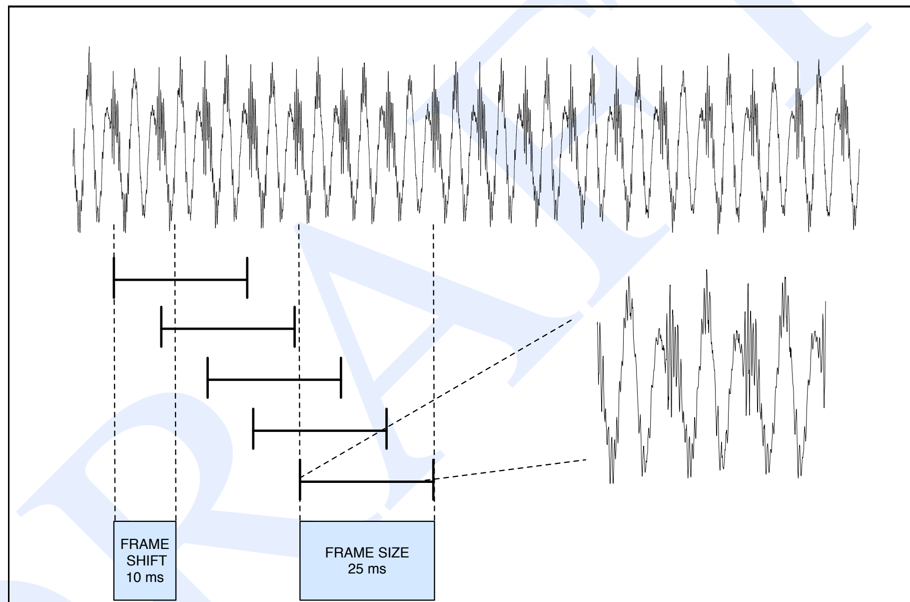

Figure 9.10 The windowing process, showing the frame shift and frame size, assuming a frame shift of 10ms, a frame size of 25 ms, and a rectangular window. After a figure by Bryan Pellom.

The extraction of the signal takes place by multiplying the value of the signal at time n, $s[n]$, with the value of the window at time n, $w[n]$:

$$y[n] = w[n]s[n] \tag{9.9}$$

Fig. 9.10 suggests that these window shapes are rectangular, since the extracted windowed signal looks just like the original signal. Indeed the simplest window is the **rectangular** window. The rectangular window can cause problems, however, because it abruptly cuts of the signal at its boundaries. These discontinuities create problems when we do Fourier analysis. For this reason, a more common window used in MFCC extraction is the **Hamming** window, which shrinks the values of the signal toward zero at the window boundaries, avoiding discontinuities. Fig. 9.11 shows both of these windows; the equations are as follows (assuming a window that is L frames long):

Rectangular
Hamming

DRAFT

$$\text{(9.10)} \quad rectangular \qquad w[n] = \begin{cases} 1 & 0 \le n \le L-1 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{(9.11)} \quad hamming \qquad w[n] = \begin{cases} 0.54 - 0.46\cos(\frac{2\pi n}{L}) & 0 \le n \le L-1 \\ 0 & \text{otherwise} \end{cases}$$

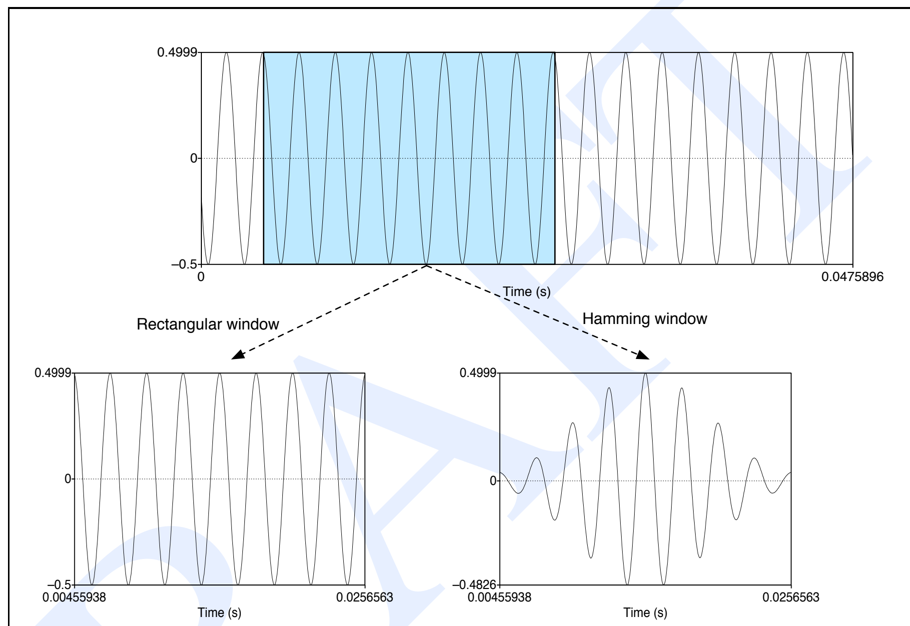

Figure 9.11 Windowing a portion of a pure sine wave with the rectangular and Hamming windows.

9.3.3 Discrete Fourier Transform

The next step is to extract spectral information for our windowed signal; we need to know how much energy the signal contains at different frequency bands. The tool for extracting spectral information for discrete frequency bands for a discrete-time (sampled) signal is the **Discrete Fourier Transform** or **DFT**.

Discrete Fourier Transform
DFT

The input to the DFT is a windowed signal $x[n] \ldots x[m]$, and the output, for each of N discrete frequency bands, is a complex number $X[k]$ representing the magnitude and phase of that frequency component in the original signal. If we plot the magnitude against the frequency, we can visualize the **spectrum** that we introduced in Ch. 7. For example, Fig. 9.12 shows a 25 ms Hamming-windowed portion of a signal and its spectrum as computed by a DFT (with some additional smoothing).

We will not introduce the mathematical details of the DFT here, except to note that Fourier analysis in general relies on **Euler's formula**:

Euler's formula

$$e^{j\theta} = \cos\theta + j\sin\theta \tag{9.12}$$

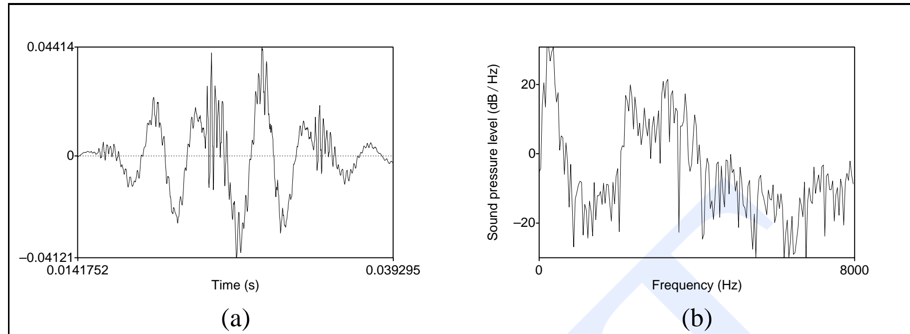

Figure 9.12 (a) A 25 ms Hamming-windowed portion of a signal from the vowel [iy] and (b) its spectrum computed by a DFT.

As a brief reminder for those students who have already had signal processing, the DFT is defined as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\frac{\pi}{N}kn} \tag{9.13}$$

Fast Fourier Transform
FFT

A commonly used algorithm for computing the DFT is the **Fast Fourier Transform** or **FFT**. This implementation of the DFT is very efficient, but only works for values of N which are powers of two.

9.3.4 Mel filter bank and log

Mel

The results of the FFT will be information about the amount of energy at each frequency band. Human hearing, however, is not equally sensitive at all frequency bands. It is less sensitive at higher frequencies, roughly above 1000 Hertz. It turns out that modeling this property of human hearing during feature extraction improves speech recognition performance. The form of the model used in MFCCs is to warp the frequencies output by the DFT onto the **mel** scale mentioned in Ch. 7. A **mel** (Stevens et al., 1937; Stevens and Volkmann, 1940) is a unit of pitch defined so that pairs of sounds which are perceptually equidistant in pitch are separated by an equal number of mels. The mapping between frequency in Hertz and the mel scale is linear below 1000 Hz and the logarithmic above 1000 Hz. The mel frequency m can be computed from the raw acoustic frequency as follows:

$$mel(f) = 1127\ln(1+\frac{f}{700}) \tag{9.14}$$

During MFCC computation, this intuition is implemented by creating a bank of filters which collect energy from each frequency band, with 10 filters spaced linearly below 1000 Hz, and the remaining filters spread logarithmically above 1000 Hz. Fig. 9.13 shows the bank of triangular filters that implement this idea.

Finally, we take the log of each of the mel spectrum values. In general the human response to signal level is logarithmic; humans are less sensitive to slight differences in amplitude at high amplitudes than at low amplitudes. In addition, using a log makes

DRAFT

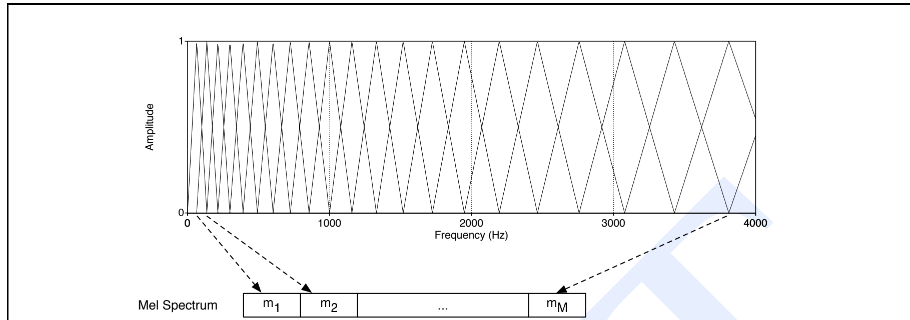

Figure 9.13 The Mel filter bank, after Davis and Mermelstein (1980). Each triangular filter collects energy from a given frequency range. Filters are spaced linearly below 1000 Hz, and logarithmically above 1000 Hz.

the feature estimates less sensitive to variations in input (for example power variations due to the speaker's mouth moving closer or further from the microphone).

9.3.5 The Cepstrum: Inverse Discrete Fourier Transform

While it would be possible to use the mel spectrum by itself as a feature representation for phone detection, the spectrum also has some problems, as we will see. For this reason, the next step in MFCC feature extraction is the computation of the **cepstrum**. The cepstrum has a number of useful processing advantages and also significantly improves phone recognition performance.

Cepstrum

One way to think about the cepstrum is as a useful way of separating the **source** and **filter**. Recall from Sec. 7.4.6 that the speech waveform is created when a glottal source waveform of a particular fundamental frequency is passed through the vocal tract, which because of its shape has a particular filtering characteristic. But many characteristics of the glottal **source** (its fundamental frequency, the details of the glottal pulse, etc) are not important for distinguishing different phones. Instead, the most useful information for phone detection is the **filter**, i.e. the exact position of the vocal tract. If we knew the shape of the vocal tract, we would know which phone was being produced. This suggests that useful features for phone detection would find a way to deconvolve (separate) the source and filter and show us only the vocal tract filter. It turns out that the cepstrum is one way to do this.

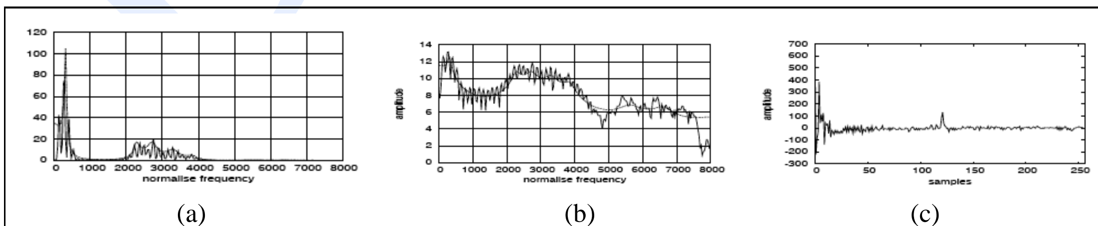

Figure 9.14 PLACEHOLDER FIGURE. The magnitude spectrum (a), the log magnitude spectrum (b), and the cepstrum (c). From Taylor (2008). The two spectra have a smoothed spectral enveloped laid on top of them to help visualize the spectrum.

DRAFT

For simplicity, let's ignore the pre-emphasis and mel-warping that are part of the definition of MFCCs, and look just at the basic definition of the cepstrum. The cepstrum can be thought of as the *spectrum of the log of the spectrum*. This may sound confusing. But let's begin with the easy part: the *log of the spectrum*. That is, the cepstrum begins with a standard magnitude spectrum, such as the one for a vowel shown in Fig. 9.14(a) from Taylor (2008). We then take the log, i.e. replace each amplitude value in the magnitude spectrum with its log, as shown in Fig. 9.14(b).

The next step is to visualize the log spectrum *as if itself were a waveform*. In other words, consider the log spectrum in Fig. 9.14(b). Let's imagine removing the axis labels that tell us that this is a spectrum (frequency on the x-axis) and imagine that we are dealing with just a normal speech signal with time on the x-axis. Now what can we say about the spectrum of this 'pseudo-signal'? Notice that there is a high-frequency repetitive component in this wave: small waves that repeat about 8 times in each 1000 along the x-axis, for a frequency of about 120 Hz. This high-frequency component is caused by the fundamental frequency of the signal, and represents the little peaks in the spectrum at each harmonic of the signal. In addition, there are some lower frequency components in this 'pseudo-signal'; for example the envelope or formant structure has about four large peaks in the window, for a much lower frequency.

Fig. 9.14(c) shows the **cepstrum**: the spectrum that we have been describing of the log spectrum. This cepstrum (the word **cepstrum** is formed by reversing the first letters of **spectrum**) is shown with **samples** along the x-axis. This is because by taking the spectrum of the log spectrum, we have left the frequency domain of the spectrum, and gone back to the time domain. It turns out that the correct unit of a cepstrum is the sample.

Examining this cepstrum, we see that there is indeed a large peak around 120, corresponding to the F0 and representing the glottal pulse. There are other various components at lower values on the x-axis. These represent the vocal tract filter (the position of the tongue and the other articulators). Thus if we are interested in detecting phones, we can make use of just the lower cepstral values. If we are interested in detecting pitch, we can use the higher cepstral values.

For the purposes of MFCC extraction, we generally just take the first 12 cepstral values. These 12 coefficients will represent information solely about the vocal tract filter, cleanly separated from information about the glottal source.

It turns out that cepstral coefficients have the extremely useful property that the variance of the different coefficients tends to be uncorrelated. This is not true for the spectrum, where spectral coefficients at different frequency bands are correlated. The fact that cepstral features are uncorrelated means, as we will see in the next section, that the Gaussian acoustic model (the Gaussian Mixture Model, or GMM) doesn't have to represent the covariance between all the MFCC features, which hugely reduces the number of parameters.

For those who have had signal processing, the cepstrum is more formally defined as the **inverse DFT of the log magnitude of the DFT of a signal**, hence for a windowed frame of speech $x[n]$:

(9.15)
$$c[n] = \sum_{n=0}^{N-1} log\left(\left|\sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}\right|\right)e^{j\frac{2\pi}{N}kn}$$

9.3.6 Deltas and Energy

The extraction of the cepstrum via the Inverse DFT from the previous section results in 12 cepstral coefficients for each frame. We next add a thirteenth feature: the energy from the frame. Energy correlates with phone identity and so is a useful cue for phone detection (vowels and sibilants have more energy than stops, etc). The **energy** in a frame is the sum over time of the power of the samples in the frame; thus for a signal x in a window from time sample t_1 to time sample t_2, the energy is:

Energy

(9.16)
$$Energy = \sum_{t=t_1}^{t_2} x^2[t]$$

Another important fact about the speech signal is that it is not constant from frame to frame. This change, such as the slope of a formant at its transitions, or the nature of the change from a stop closure to stop burst, can provide a useful cue for phone identity. For this reason we also add features related to the change in cepstral features over time.

We do this by adding for each of the 13 features (12 cepstral features plus energy) a **delta** or **velocity** feature, and a **double delta** or **acceleration** feature. Each of the 13 delta features represents the change between frames in the corresponding cepstral/energy feature, while each of the 13 double delta features represents the change between frames in the corresponding delta features.

Delta feature

Double delta

A simple way to compute deltas would be just to compute the difference between frames; thus the delta value $d(t)$ for a particular cepstral value $c(t)$ at time t can be estimated as:

(9.17)
$$d(t) = \frac{c(t+1) - c(t-1)}{2}$$

Instead of this simple estimate, however, it is more common to make more sophisticated estimates of the slope, using a wider context of frames.

9.3.7 Summary: MFCC

After adding energy, and then delta and double-delta features to the 12 cepstral features, we end up with 39 MFCC features:

12	cepstral coefficients
12	delta cepstral coefficients
12	double delta cepstral coefficients
1	energy coefficient
1	delta energy coefficient
1	double delta energy coefficient
39	MFCC features

DRAFT

Again, one of the most useful facts about MFCC features is that the cepstral coefficients tend to be uncorrelated, which will turn out to make our acoustic model much simpler.

9.4 Computing Acoustic Likelihoods

The last section showed how we can extract MFCC features representing spectral information from a wavefile, and produce a 39-dimensional vector every 10 milliseconds. We are now ready to see how to compute the likelihood of these feature vectors given an HMM state. Recall from Ch. 6 that this output likelihood is computed by the B probability function of the HMM. Given an individual state q_i and an observation o_t, the observation likelihoods in B matrix gave us $p(o_t|q_i)$, which we called $b_t(i)$.

For part-of-speech tagging in Ch. 5, each observation o_t is a discrete symbol (a word) and we can compute the likelihood of an observation given a part-of-speech tag just by counting the number of times a given tag generates a given observation in the training set. But for speech recognition, MFCC vectors are real-valued numbers; we can't compute the likelihood of a given state (phone) generating an MFCC vector by counting the number of times each such vector occurs (since each one is likely to be unique).

In both decoding and training, we need an observation likelihood function that can compute $p(o_t|q_i)$ on real-valued observations. In decoding, we are given an observation o_t and we need to produce the probability $p(o_t|q_i)$ for each possible HMM state, so we can choose the most likely sequence of states. Once we have this observation likelihood B function, we need to figure out how to modify the Baum-Welch algorithm of Ch. 6 to train it as part of training HMMs.

9.4.1 Vector Quantization

One way to make MFCC vectors look like symbols that we could count is to build a mapping function that maps each input vector into one of a small number of symbols. Then we could just compute probabilities on these symbols by counting, just as we did for words in part-of-speech tagging. This idea of mapping input vectors to discrete quantized symbols is called **vector quantization** or **VQ** (Gray, 1984). Although vector quantization is too simple to act as the acoustic model in modern LVCSR systems, it is a useful pedagogical step, and plays an important role in various areas of ASR, so we use it to begin our discussion of acoustic modeling.

Vector quantization
VQ

In vector quantization, we create the small symbol set by mapping each training feature vector into a small number of classes, and then we represent each class by a discrete symbol. More formally, a vector quantization system is characterized by a **codebook**, a **clustering algorithm**, and a **distance metric**.

Codebook
Prototype vector
Codeword

A **codebook** is a list of possible classes, a set of symbols constituting a vocabulary $V = \{v_1, v_2, ..., v_n\}$. For each symbol v_k in the codebook we list a **prototype vector**, also known as a **codeword**, which is a specific feature vector. For example if we choose to use 256 codewords we could represent each vector by a value from 0 to 255; (this

is referred to as 8-bit VQ, since we can represent each vector by a single 8-bit value). Each of these 256 values would be associated with a prototype feature vector.

Clustering

The codebook is created by using a **clustering** algorithm to cluster all the feature vectors in the training set into the 256 classes. Then we chose a representative feature vector from the cluster, and make it the prototype vector or codeword for that cluster. **K-means clustering** is often used, but we won't define clustering here; see Huang et al. (2001) or Duda et al. (2000) for detailed descriptions.

K-means clustering

Once we've built the codebook, for each incoming feature vector, we compare it to each of the 256 prototype vectors, select the one which is closest (by some **distance metric**), and replace the input vector by the index of this prototype vector. A schematic of this process is shown in Fig. 9.15.

The advantage of VQ is that since there are a finite number of classes, for each class v_k, we can compute the probability that it is generated by a given HMM state/sub-phone by simply counting the number of times it occurs in some training set when labeled by that state, and normalizing.

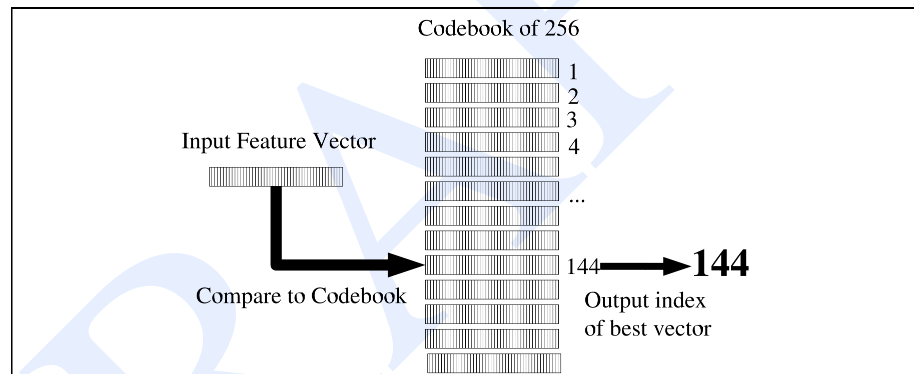

Figure 9.15 Schematic architecture of the (trained) vector quantization (VQ) process for choosing a symbol v_q for each input feature vector. The vector is compared to each codeword in the codebook, the closest entry (by some distance metric) is selected, and the index of the closest codeword is output.

Distance metric

Both the clustering process and the decoding process require a **distance metric** or **distortion** metric, that specifies how similar two acoustic feature vectors are. The distance metric is used to build clusters, to find a prototype vector for each cluster, and to compare incoming vectors to the prototypes.

Euclidean distance

The simplest distance metric for acoustic feature vectors is **Euclidean distance**. Euclidean distance is the distance in N-dimensional space between the two points defined by the two vectors. In practice we use the phrase 'Euclidean distance' even though we actually often use the square of the Euclidean distance. Thus given a vector x and a vector y of length D, the (square of the) Euclidean distance between them is defined as:

$$d_{\text{euclidean}}(x,y) = \sum_{i=1}^{D}(x_i - y_i)^2 \tag{9.18}$$

The (squared) Euclidean distance described in (9.18) (and shown for two dimensions in Fig. 9.16) is also referred to as the sum-squared error, and can also be expressed using the vector transpose operator as:

$$d_{\text{euclidean}}(x,y) = (x-y)^T(x-y) \tag{9.19}$$

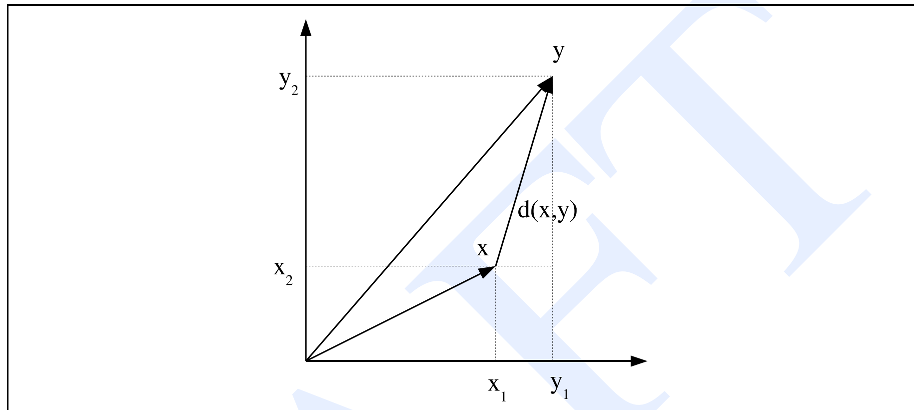

Figure 9.16 Euclidean distance in two dimensions; by the Pythagorean theorem, the distance between two points in a plane $x = (x1, y1)$ and $y = (x2, y2)$ $d(x,y) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

The Euclidean distance metric assumes that each of the dimensions of a feature vector are equally important. But actually each of the dimensions have very different variances. If a dimension tends to have a lot of variance, then we'd like it to count less in the distance metric; a large difference in a dimension with low variance should count more than a large difference in a dimension with high variance. A slightly more complex distance metric, the **Mahalanobis distance**, takes into account the different variances of each of the dimensions.

Mahalanobis distance

If we assume that each dimension i of the acoustic feature vectors has a variance σ_i^2, then the Mahalanobis distance is:

$$d_{\text{mahalanobis}}(x,y) = \sum_{i=1}^{D} \frac{(x_i - y_i)^2}{\sigma_i^2} \tag{9.20}$$

For those readers with more background in linear algebra here's the general form of Mahalanobis distance, which includes a full covariance matrix (covariance matrices will be defined below):

$$d_{\text{mahalanobis}}(x,y) = (x-y)^T \Sigma^{-1} (x-y) \tag{9.21}$$

In summary, when decoding a speech signal, to compute an acoustic likelihood of a feature vector o_t given an HMM state q_j using VQ, we compute the Euclidean or Mahalanobis distance between the feature vector and each of the N codewords, choose the closest codeword, getting the codeword index v_k. We then look up the likelihood of the codeword index v_k given the HMM state j in the pre-computed B likelihood matrix defined by the HMM:

DRAFT

$$\hat{b}_j(o_t) = b_j(v_k) \text{ s.t. } v_k \text{ is codeword of closest vector to } o_t \quad (9.22)$$

Since VQ is so rarely used, we don't use up space here giving the equations for modifying the EM algorithm to deal with VQ data; instead, we defer discussion of EM training of continuous input parameters to the next section, when we introduce Gaussians.

9.4.2 Gaussian PDFs

Vector quantization has the advantage of being extremely easy to compute and requires very little storage. Despite these advantages, vector quantization turns out not to be a good model of speech. A small number of codewords is insufficient to capture the wide variability in the speech signal. Speech is simply not a categorical, symbolic process.

Modern speech recognition algorithms therefore do not use vector quantization to compute acoustic likelihoods. Instead, they are based on computing observation probabilities directly on the real-valued, continuous input feature vector. These acoustic models are based on computing a **probability density function** or **pdf** over a continuous space. By far the most common method for computing acoustic likelihoods is the **Gaussian Mixture Model** (**GMM**) pdfs, although neural networks, support vector machines (SVMs) and conditional random fields (CRFs) are also used.

Probability density function

Gaussian Mixture Model

GMM

Let's begin with the simplest use of Gaussian probability estimators, slowly building up the more sophisticated models that are used.

Univariate Gaussians

The **Gaussian** distribution, also known as the **normal distribution**, is the bell-curve function familiar from basic statistics. A Gaussian distribution is a function parameterized by a **mean**, or average value, and a **variance**, which characterizes the average spread or dispersal from the mean. We will use μ to indicate the mean, and σ^2 to indicate the variance, giving the following formula for a Gaussian function:

Gaussian

Normal distribution

Mean

Variance

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(x-\mu)^2}{2\sigma^2}) \quad (9.23)$$

Recall from basic statistics that the mean of a random variable X is the expected value of X. For a discrete variable X, this is the weighted sum over the values of X (for a continuous variable, it is the integral):

$$\mu = E(X) = \sum_{i=1}^{N} p(X_i)X_i \quad (9.24)$$

The variance of a random variable X is the weigthed squared average deviation from the mean:

$$\sigma^2 = E(X_i - E(X))^2 = \sum_{i=1}^{N} p(X_i)(X_i - E(X))^2 \quad (9.25)$$

When a Gaussian function is used as a probability density function, the area under the curve is constrained to be equal to one. Then the probability that a random variable

DRAFT

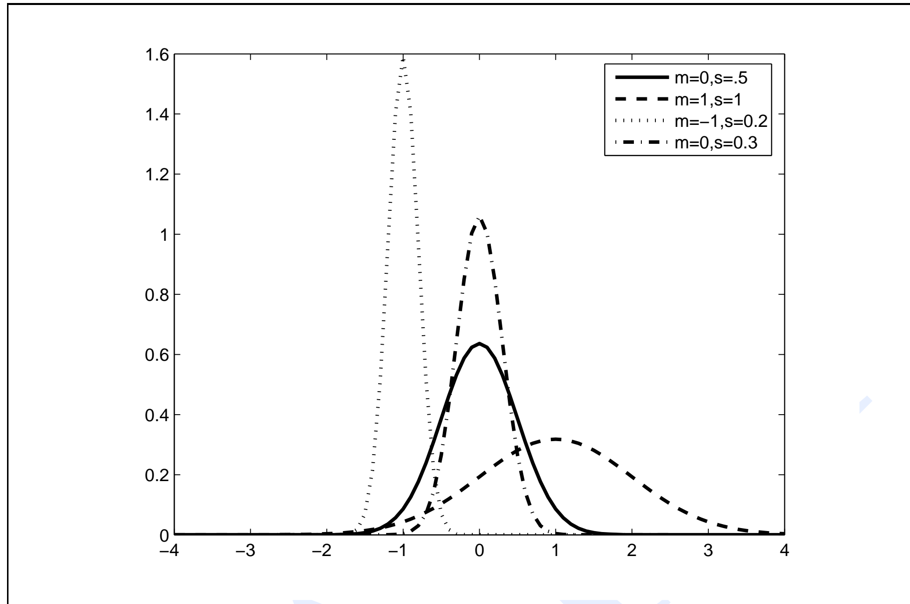

Figure 9.17 Gaussian functions with different means and variances.

takes on any particular range of values can be computed by summing the area under the curve for that range of values. Fig. 9.18 shows the probability expressed by the area under an interval of a Gaussian.

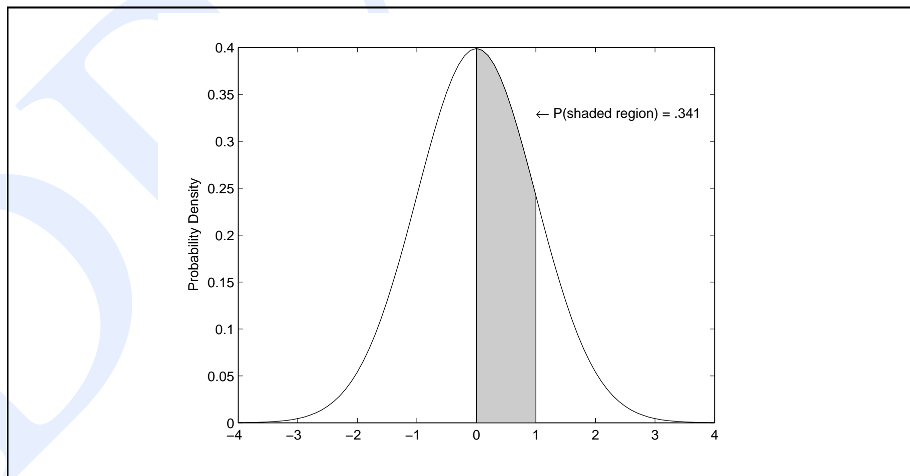

Figure 9.18 A Gaussian probability density function, showing a region from 0 to 1 with a total probability of .341. Thus for this sample Gaussian, the probability that a value on the X axis lies between 0 and 1 is .341.

We can use a univariate Gaussian pdf to estimate the probability that a particular HMM state j generates the value of a single dimension of a feature vector by assuming that the possible values of (this one dimension of the) observation feature vector o_t are

normally distributed. In other words we represent the observation likelihood function $b_j(o_t)$ for one dimension of the acoustic vector as a Gaussian. Taking, for the moment, our observation as a single real valued number (a single cepstral feature), and assuming that each HMM state j has associated with it a mean value μ_j and variance σ_j^2, we compute the likelihood $b_j(o_t)$ via the equation for a Gaussian pdf:

$$b_j(o_t) = \frac{1}{\sqrt{2\pi\sigma_j^2}} exp\left(-\frac{(o_t-\mu_j)^2}{2\sigma_j^2}\right) \tag{9.26}$$

Eq. 9.26 shows us how to compute $b_j(o_t)$, the likelihood of an individual acoustic observation given a single univariate Gaussian from state j with its mean and variance. We can now use this probability in HMM decoding.

But first we need to solve the training problem; how do we compute this mean and variance of the Gaussian for each HMM state q_i? Let's start by imagining the simpler situation of a completely labeled training set, in which each acoustic observation was labeled with the HMM state that produced it. In such a training set, we could compute the mean of each state by just taking the average of the values for each o_t that corresponded to state i, as show in (9.27). The variance could just be computed from the sum-squared error between each observation and the mean, as shown in (9.28).

$$\hat{\mu}_i = \frac{1}{T}\sum_{t=1}^{T} o_t \text{ s.t. } q_t \text{ is state } i \tag{9.27}$$

$$\hat{\sigma}_j^2 = \frac{1}{T}\sum_{t=1}^{T} (o_t-\mu_i)^2 \text{ s.t. } q_t \text{ is state } i \tag{9.28}$$

But since states are hidden in an HMM, we don't know exactly which observation vector o_t was produced by which state. What we would like to do is assign each observation vector o_t to every possible state i, prorated by the probability that the HMM was in state i at time t. Luckily, we already know how to do this prorating; the probability of being in state i at time t was defined in Ch. 6 as $\xi_t(i)$, and we saw how to compute $\xi_t(i)$ as part of the Baum-Welch algorithm using the forward and backward probabilities. Baum-Welch is an iterative algorithm, and we will need to do the probability computation of $\xi_t(i)$ iteratively since getting a better observation probability b will also help us be more sure of the probability ξ of being in a state at a certain time. Thus we give equations for computing an updated mean and variance $\hat{\mu}$ and $\hat{\sigma}^2$:

$$\hat{\mu}_i = \frac{\sum_{t=1}^{T}\xi_t(i)o_t}{\sum_{t=1}^{T}\xi_t(i)} \tag{9.29}$$

$$\hat{\sigma}_i^2 = \frac{\sum_{t=1}^{T}\xi_t(i)(o_t-\mu_i)^2}{\sum_{t=1}^{T}\xi_t(i)} \tag{9.30}$$

Eq. 9.29 and Eq. 9.30 are then used in the forward-backward (Baum-Welch)training of the HMM. As we will see, the values of μ_i and σ_i are first set to some initial estimate, which is then re-estimated until the numbers converge.

DRAFT

Multivariate Gaussians

Eq. 9.26 shows how to use a Gaussian to compute an acoustic likelihood for a single cepstral feature. Since an acoustic observation is a vector of 39 features, we'll need to use a multivariate Gaussian, which allows us to assign a probability to a 39-valued vector. Where a univariate Gaussian is defined by a mean μ and a variance σ^2, a multivariate Gaussian is defined by a mean vector $\vec{\mu}$ of dimensionality D and a covariance matrix Σ, defined below. As we discussed in the previous section, for a typical cepstral feature vector in LVCSR, D is 39:

$$f(\vec{x}|\vec{\mu},\Sigma) = \frac{1}{(2\pi)^{\frac{D}{2}}|\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^{\mathrm{T}}\Sigma^{-1}(x-\mu)\right) \tag{9.31}$$

The covariance matrix Σ captures the variance of each dimension as well as the covariance between any two dimensions.

Recall again from basic statistics that the covariance of two random variables X and Y is the expected value of the product of their average deviations from the mean:

$$\Sigma = E[(X-E(X))(Y-E(Y))] = \sum_{i=1}^{N} p(X_iY_i)(X_i-E(X))(Y_i-E(Y)) \tag{9.32}$$

Thus for a given HMM state with mean vector μ_j and covariance matrix Σ_j, and a given observation vector o_t, the multivariate Gaussian probability estimate is:

$$b_j(o_t) = \frac{1}{(2\pi)^{\frac{D}{2}}|\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(o_t-\mu_j)^T\Sigma_j^{-1}(o_t-\mu_j)\right) \tag{9.33}$$

The covariance matrix Σ_j expresses the variance between each pair of feature dimensions. Suppose we made the simplifying assumption that features in different dimensions did not covary, i.e., that there was no correlation between the variances of different dimensions of the feature vector. In this case, we could simply keep a distinct variance for each feature dimension. It turns out that keeping a separate variance for each dimension is equivalent to having a covariance matrix that is **diagonal**, i.e. non-zero elements only appear along the main diagonal of the matrix. The main diagonal of such a diagonal covariance matrix contains the variances of each dimension, $\sigma_1^2, \sigma_2^2, \ldots \sigma_D^2$;

Diagonal

Let's look at some illustrations of multivariate Gaussians, focusing on the role of the full versus diagonal covariance matrix. We'll explore a simple multivariate Gaussian with only 2 dimensions, rather than the 39 that are typical in ASR. Fig. 9.19 shows three different multivariate Gaussians in two dimensions. The leftmost figure shows a Gaussian with a diagonal covariance matrix, in which the variances of the two dimensions are equal. Fig. 9.20 shows 3 contour plots corresponding to the Gaussians in Fig. 9.19; each is a slice through the Gaussian. The leftmost graph in Fig. 9.20 shows a slice through the diagonal equal-variance Gaussian. The slice is circular, since the variances are equal in both the X and Y directions.

The middle figure in Fig. 9.19 shows a Gaussian with a diagonal covariance matrix, but where the variances are not equal. It is clear from this figure, and especially from

DRAFT

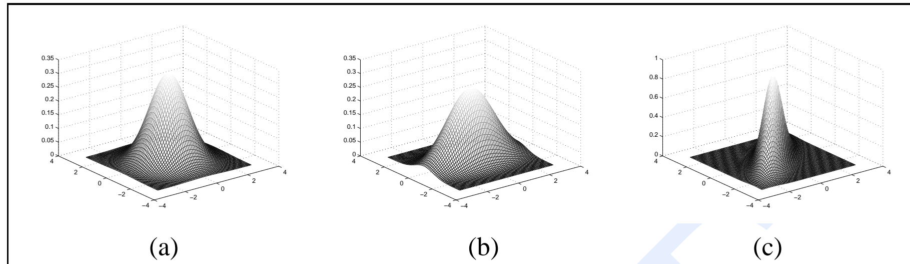

Figure 9.19 Three different multivariate Gaussians in two dimensions. The first two have diagonal covariance matrices, one with equal variance in the two dimensions $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, the second with different variances in the two dimensions, $\begin{bmatrix} .6 & 0 \\ 0 & 2 \end{bmatrix}$, and the third with non-zero elements in the off-diagonal of the covariance matrix: $\begin{bmatrix} 1 & .8 \\ .8 & 1 \end{bmatrix}$.

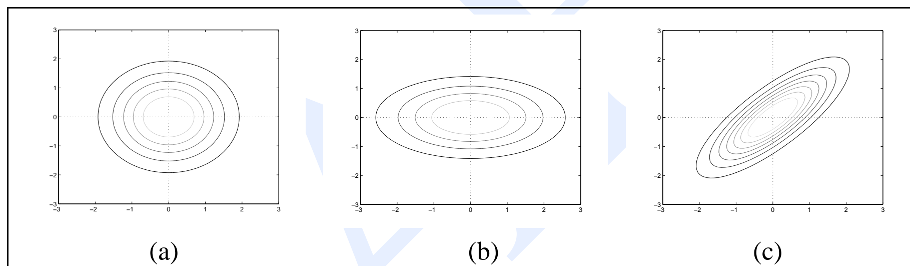

Figure 9.20 The same three multivariate Gaussians as in the previous figure. From left to right, a diagonal covariance matrix with equal variance, diagonal with unequal variance, and nondiagonal covariance. With non-diagonal covariance, knowing the value on dimension X tells you something about the value on dimension Y.

the contour slice show in Fig. 9.20, that the variance is more than 3 times greater in one dimension than the other.

The rightmost graph in Fig. 9.19 and Fig. 9.20 shows a Gaussian with a non-diagonal covariance matrix. Notice in the contour plot in Fig. 9.20 that the contour is not lined up with the two axes, as it is in the other two plots. Because of this, knowing the value in one dimension can help in predicting the value in the other dimension. Thus having a non-diagonal covariance matrix allows us to model correlations between the values of the features in multiple dimensions.

A Gaussian with a full covariance matrix is thus a more powerful model of acoustic likelihood than one with a diagonal covariance matrix. And indeed, speech recognition performance is better using full-covariance Gaussians than diagonal-covariance Gaussians. But there are two problems with full-covariance Gaussians that makes them difficult to use in practice. First, they are slow to compute. A full covariance matrix has D^2 parameters, where a diagonal covariance matrix has only D. This turns out to make a large difference in speed in real ASR systems. Second, a full covariance matrix has many more parameters and hence requires much more data to train than a diagonal covariance matrix. Using a diagonal covariance model means we can save room for

DRAFT

using our parameters for other things like triphones (context-dependent phones) to be introduced in Sec. 10.3.

For this reason, in practice most ASR systems use diagonal covariance. We will assume diagonal covariance for the remainder of this section.

Eq. 9.33 can thus be simplified to the version in (9.34) in which instead of a covariance matrix, we simply keep a mean and variance for each dimension. Eq. 9.34 thus describes how to estimate the likelihood $b_j(o_t)$ of a D-dimensional feature vector o_t given HMM state j, using a diagonal-covariance multivariate Gaussian.

$$b_j(o_t) = \prod_{d=1}^{D} \frac{1}{\sqrt{2\pi\sigma_{jd}^2}} exp\left(-\frac{1}{2}[\frac{(o_{td}-\mu_{jd})^2}{\sigma_{jd}{}^2}]\right) \tag{9.34}$$

Training a diagonal-covariance multivariate Gaussian is a simple generalization of training univariate Gaussians. We'll do the same Baum-Welch training, where we use the value of $\xi_t(i)$ to tell us the likelihood of being in state i at time t. Indeed, we'll use exactly the same equation as in (9.30), except that now we are dealing with vectors instead of scalars; the observation o_t is a vector of cepstral features, the mean vector $\vec{\mu}$ is a vector of cepstral means, and the variance vector $\vec{\sigma_i^2}$ is a vector of cepstral variances.

$$\hat{\mu}_i = \frac{\sum_{t=1}^{T} \xi_t(i) o_t}{\sum_{t=1}^{T} \xi_t(i)} \tag{9.35}$$

$$\hat{\sigma}_i^2 = \frac{\sum_{t=1}^{T} \xi_t(i)(o_t-\mu_i)(o_t-\mu_i)^T}{\sum_{t=1}^{T} \xi_t(i)} \tag{9.36}$$

Gaussian Mixture Models

The previous subsection showed that we can use a multivariate Gaussian model to assign a likelihood score to an acoustic feature vector observation. This models each dimension of the feature vector as a normal distribution. But a particular cepstral feature might have a very non-normal distribution; the assumption of a normal distribution may be too strong an assumption. For this reason, we often model the observation likelihood not with a single multivariate Gaussian, but with a weighted mixture of multivariate Gaussians. Such a model is called a **Gaussian Mixture Model** or **GMM**. Eq. 9.37 shows the equation for the GMM function; the resulting function is the sum of M Gaussians. Fig. 9.21 shows an intuition of how a mixture of Gaussians can model arbitrary functions.

Gaussian Mixture Model
GMM

$$f(x|\mu,\Sigma) = \sum_{k=1}^{M} c_k \frac{1}{\sqrt{2\pi|\Sigma_k|}} exp[(x-\mu_k)^T \Sigma^{-1}(x-\mu_k)] \tag{9.37}$$

Eq. 9.38 shows the definition of the output likelihood function $b_j(o_t)$

$$b_j(o_t) = \sum_{m=1}^{M} c_{jm} \frac{1}{\sqrt{2\pi|\Sigma_{jm}|}} exp[(x-\mu_{jm})^T \Sigma_{jm}^{-1}(o_t-\mu_{jm})] \tag{9.38}$$

DRAFT

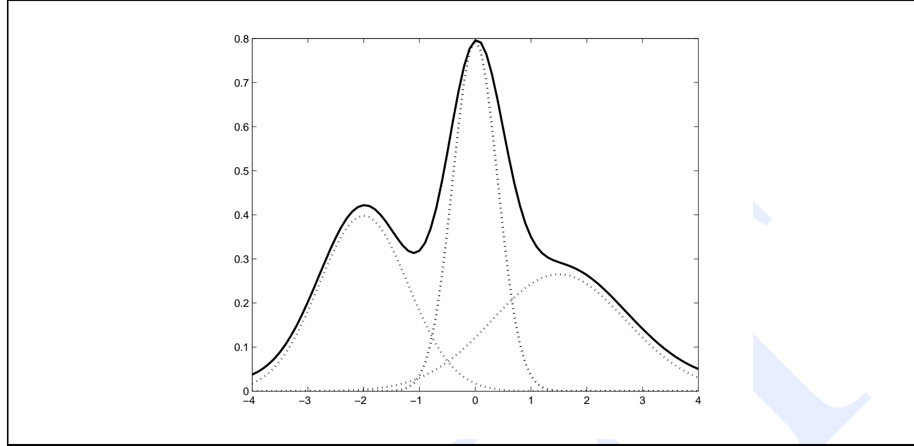

Figure 9.21 An arbitrary function approximated by a mixture of 3 gaussians.

Let's turn to training the GMM likelihood function. This may seem hard to do; how can we train a GMM model if we don't know in advance which mixture is supposed to account for which part of each distribution? Recall that a single multivariate Gaussian could be trained even if we didn't know which state accounted for each output, simply by using the Baum-Welch algorithm to tell us the likelihood of being in each state j at time t. It turns out the same trick will work for GMMs; we can use Baum-Welch to tell us the probability of a certain mixture accounting for the observation, and iteratively update this probability.

We used the ξ function above to help us compute the state probability. By analogy with this function, let's define $\xi_{tm}(j)$ to mean the probability of being in state j at time t with the mth mixture component accounting for the output observation o_t. We can compute $\xi_{tm}(j)$ as follows:

$$\xi_{tm}(j) = \frac{\sum_{i=1} N\alpha_{t-1}(j)a_{ij}c_{jm}b_{jm}(o_t)\beta_t(j)}{\alpha_T(F)} \tag{9.39}$$

Now if we had the values of ξ from a previous iteration of Baum-Welch, we can use $\xi_{tm}(j)$ to recompute the mean, mixture weight, and covariance using the following equations:

$$\hat{\mu}_{im} = \frac{\sum_{t=1}^{T} \xi_{tm}(i) o_t}{\sum_{t=1}^{T} \sum_{m=1}^{M} \xi_{tm}(i)} \tag{9.40}$$

$$\hat{c}_{im} = \frac{\sum_{t=1}^{T} \xi_{tm}(i)}{\sum_{t=1}^{T} \sum_{k=1}^{M} \xi_{tk}(i)} \tag{9.41}$$

$$\hat{\Sigma}_{im} = \frac{\sum_{t=1}^{T} \xi_t(i)(o_t - \mu_{im})(o_t - \mu_{im})^T}{\sum_{t=1}^{T} \sum_{k=1}^{M} \xi_{tm}(i)} \tag{9.42}$$

9.4.3 Probabilities, log probabilities and distance functions

Logprob

Up to now, all the equations we have given for acoustic modeling have used probabilities. It turns out, however, that a **log probability** (or **logprob**) is much easier to work with than a probability. Thus in practice throughout speech recognition (and related fields) we compute log-probabilities rather than probabilities.

One major reason that we can't use probabilities is numeric underflow. To compute a likelihood for a whole sentence, say, we are multiplying many small probability values, one for each 10ms frame. Multiplying many probabilities results in smaller and smaller numbers, leading to underflow. The log of a small number like $.00000001 = 10^{-8}$, on the other hand, is a nice easy-to-work-with-number like -8. A second reason to use log probabilities is computational speed. Instead of multiplying probabilities, we add log-probabilities, and adding is faster than multiplying. Log-probabilities are particularly efficient when we are using Gaussian models, since we can avoid exponentiating.

Thus for example for a single multivariate diagonal-covariance Gaussian model, instead of computing:

$$b_j(o_t) = \prod_{d=1}^{D} \frac{1}{\sqrt{2\pi\sigma_{jd}^2}} exp\left(-\frac{1}{2}\frac{(o_{td}-\mu_{jd})^2}{\sigma_{jd}^2}\right) \tag{9.43}$$

we would compute

$$\log b_j(o_t) = -\frac{1}{2}\sum_{d=1}^{D}\left[log(2\pi)+\sigma_{jd}^2+\frac{(o_{td}-\mu_{jd})^2}{\sigma_{jd}^2}\right] \tag{9.44}$$

With some rearrangement of terms, we can rewrite this equation to pull out a constant C:

$$\log b_j(o_t) = C - \frac{1}{2}\sum_{d=1}^{D}\frac{(o_{td}-\mu_{jd})^2}{\sigma_{jd}^2} \tag{9.45}$$

where C can be precomputed:

$$C = -\frac{1}{2}\sum_{d=1}^{D}\left(\log(2\pi)+\sigma_{jd}^2\right) \tag{9.46}$$

In summary, computing acoustic models in log domain means a much simpler computation, much of which can be precomputed for speed.

The perceptive reader may have noticed that equation (9.45) looks very much like the equation for Mahalanobis distance (9.20). Indeed, one way to think about Gaussian logprobs is as just a weighted distance metric.

A further point about Gaussian pdfs, for those readers with calculus. Although the equations for observation likelihood such as (9.26) are motivated by the use of Gaussian probability density functions, the values they return for the observation likelihood, $b_j(o_t)$, are not technically probabilities; they may in fact be greater than one. This is because we are computing the value of $b_j(o_t)$ at a single point, rather than integrating over a region. While the total area under the Gaussian PDF curve is constrained to one,

DRAFT

the actual value at any point could be greater than one. (Imagine a very tall skinny Gaussian; the value could be greater than one at the center, although the area under the curve is still 1.0). If we were integrating over a region, we would be multiplying each point by its width dx, which would bring the value down below one. The fact that the Gaussian estimate is not a true probability doesn't matter for choosing the most likely HMM state, since we are comparing different Gaussians, each of which is missing this dx factor.

In summary, the last few subsections introduced Gaussian models for acoustic training in speech recognition. Beginning with simple univariate Gaussian, we extended first to multivariate Gaussians to deal with the multidimensionality acoustic feature vectors. We then introduced the diagonal covariance simplification of Gaussians, and then introduced Gaussians mixtures (GMMs).

9.5 The Lexicon and Language Model

Since previous chapters had extensive discussions of the N-gram language model (Ch. 4) and the pronunciation lexicon (Ch. 7), in this section we just briefly recall them to the reader.

Language models for LVCSR tend to be trigrams or even fourgrams; good toolkits are available to build and manipulate them (Stolcke, 2002; Young et al., 2005). Bigrams and unigram grammars are rarely used for large-vocabulary applications. Since trigrams require huge amounts of space, however, language models for memory-constrained applications like cell phones tend to use smaller contexts (or use compression techniques). As we will discuss in Ch. 24, some simple dialogue applications take advantage of their limited domain to use very simple finite-state or weighted finite-state grammars.

Lexicons are simply lists of words, with a pronunciation for each word expressed as a phone sequence. Publicly available lexicons like the CMU dictionary (CMU, 1993) can be used to extract the 64,000 word vocabularies commonly used for LVCSR. Most words have a single pronunciation, although some words such as homonyms and frequent function words may have more; the average number of pronunciations per word in most LVCSR systems seems to range from 1 to 2.5. Sec. 10.5.3 in Ch. 10 discusses the issue of pronunciation modeling.

9.6 Search and Decoding

We are now very close to having described all the parts of a complete speech recognizer. We have shown how to extract cepstral features for a frame, and how to compute the acoustic likelihood $b_j(o_t)$ for that frame. We also know how to represent lexical knowledge, that each word HMM is composed of a sequence of phone models, and each phone model of a set of subphone states. Finally, in Ch. 4 we showed how to use N-grams to build a model of word predictability.

In this section we show how to combine all of this knowledge to solve the problem of **decoding**: combining all these probability estimators to produce the most probable string of words. We can phrase the decoding question as: 'Given a string of acoustic observations, how should we choose the string of words which has the highest posterior probability?'

Decoding

Recall from the beginning of the chapter the noisy channel model for speech recognition. In this model, we use Bayes rule, with the result that the best sequence of words is the one that maximizes the product of two factors, a language model prior and an acoustic likelihood:

$$\hat{W} = \underset{W \in \mathscr{L}}{\operatorname{argmax}} \overbrace{P(O|W)}^{\text{likelihood}} \overbrace{P(W)}^{\text{prior}} \tag{9.47}$$

Now that we have defined both the acoustic model and language model we are ready to see how to find this maximum probability sequence of words. First, though, it turns out that we'll need to make a modification to Eq. 9.47, because it relies on some incorrect independence assumptions. Recall that we trained a multivariate Gaussian mixture classifier to compute the likelihood of a particular acoustic observation (a frame) given a particular state (subphone). By computing separate classifiers for each acoustic frame and multiplying these probabilities to get the probability of the whole word, we are severely underestimating the probability of each subphone. This is because there is a lot of continuity across frames; if we were to take into account the acoustic context, we would have a greater expectation for a given frame and hence could assign it a higher probability. We must therefore reweight the two probabilities. We do this by adding in a **language model scaling factor** or **LMSF**, also called the **language weight**. This factor is an exponent on the language model probability $P(W)$. Because $P(W)$ is less than one and the LMSF is greater than one (between 5 and 15, in many systems), this has the effect of decreasing the value of the LM probability:

LMSF

$$\hat{W} = \underset{W \in \mathscr{L}}{\operatorname{argmax}} P(O|W)P(W)^{LMSF} \tag{9.48}$$

Reweighting the language model probability $P(W)$ in this way requires us to make one more change. This is because $P(W)$ has a side-effect as a penalty for inserting words. It's simplest to see this in the case of a uniform language model, where every word in a vocabulary of size $|V|$ has an equal probability $\frac{1}{|V|}$. In this case, a sentence with N words will have a language model probability of $\frac{1}{|V|}$ for each of the N words, for a total penalty of $\frac{N}{|V|}$. The larger N is (the more words in the sentence), the more times this $\frac{1}{V}$ penalty multiplier is taken, and the less probable the sentence will be. Thus if (on average) the language model probability decreases (causing a larger penalty), the decoder will prefer fewer, longer words. If the language model probability increases (larger penalty), the decoder will prefer more shorter words. Thus our use of a LMSF to balance the acoustic model has the side-effect of decreasing the word insertion penalty. To offset this, we need to add back in a separate **word insertion penalty**:

Word insertion penalty

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} P(O|W)P(W)^{LMSF} WIP^N \tag{9.49}$$

Since in practice we use logprobs, the goal of our decoder is:

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} \log P(O|W) + LMSF \times \log P(W) + N \times \log WIP \tag{9.50}$$

Now that we have an equation to maximize, let's look at how to decode. It's the job of a decoder to simultaneously segment the utterance into words and identify each of these words. This task is made difficult by variation, both in terms of how words are pronounced in terms of phones, and how phones are articulated in acoustic features. Just to give an intuition of the difficulty of the problem imagine a massively simplified version of the speech recognition task, in which the decoder is given a series of discrete phones. In such a case, we would know what each phone was with perfect accuracy, and yet decoding is still difficult. For example, try to decode the following sentence from the (hand-labeled) sequence of phones from the Switchboard corpus (don't peek ahead!):

[ay d ih s hh er d s ah m th ih ng ax b aw m uh v ih ng r ih s en l ih]

The answer is in the footnote.[2] The task is hard partly because of coarticulation and fast speech (e.g., [d] for the first phone of *just*!). But it's also hard because speech, unlike English writing, has no spaces indicating word boundaries. The true decoding task, in which we have to identify the phones at the same time as we identify and segment the words, is of course much harder.

For decoding, we will start with the Viterbi algorithm that we introduced in Ch. 6, in the domain of **digit recognition**, a simple task with a vocabulary size of 11 (the numbers *one* through *nine* plus *zero* and *oh*).

Recall the basic components of an HMM model for speech recognition:

$Q = q_1 q_2 \ldots q_N$	a set of **states** corresponding to **subphones**
$A = a_{01} a_{02} \ldots a_{n1} \ldots a_{nn}$	a **transition probability matrix** A, each a_{ij} representing the probability for each subphone of taking a **self-loop** or going to the next subphone. Together, Q and A implement a **pronunciation lexicon**, an HMM state graph structure for each word that the system is capable of recognizing.
$B = b_i(o_t)$	A set of **observation likelihoods:**, also called **emission probabilities**, each expressing the probability of a cepstral feature vector (observation o_t) being generated from subphone state i.

The HMM structure for each word comes from a lexicon of word pronunciations. Generally we use an off-the-shelf pronunciation dictionary such as the free CMUdict dictionary described in Ch. 7. Recall from page 295 that the HMM structure for words

[2] I just heard something about moving recently.

in speech recognition is a simple concatenation of phone HMMs, each phone consisting of 3 subphone states, where every state has exactly two transitions: a self-loop and a loop to the next phones. Thus the HMM structure for each digit word in our digit recognizer is computed simply by taking the phone string from the dictionary, expanding each phone into 3 subphones, and concatenating together. In addition, we generally add an optional silence phone at the end of each word, allowing the possibility of pausing between words. We usually define the set of states Q from some version of the ARPAbet, augmented with silence phones, and expanded to create three subphones for each phone.

The A and B matrices for the HMM are trained by the Baum-Welch algorithm in the **embedded training** procedure that we will describe in Sec. 9.7. For now we'll assume that these probabilities have been trained.

Fig. 9.22 shows the resulting HMM for digit recognition. Note that we've added non-emitting start and end states, with transitions from the end of each word to the end state, and a transition from the end state back to the start state to allow for sequences of digits. Note also the optional silence phones at the end of each word.

Digit recognizers often don't use word probabilities, since in many digit situations (phone numbers or credit card numbers) each digit may have an equal probability of appearing. But we've included transition probabilities into each word in Fig. 9.22, mainly to show where such probabilities would be for other kinds of recognition tasks. As it happens, there are cases where digit probabilities do matter, such as in addresses (which are often likely to end in 0 or 00) or in cultures where some numbers are lucky and hence more frequent, such as the lucky number '8' in Chinese.

Now that we have an HMM, we can use the same forward and Viterbi algorithms that we introduced in Ch. 6. Let's see how to use the forward algorithm to generate $P(O|W)$, the likelihood of an observation sequence O given a sequence of words W; we'll use the single word "five". In order to compute this likelihood, we need to sum over all possible sequences of states; assuming *five* has the states [f], [ay], and [v], a 10-observation sequence includes many sequences such as the following:

```
f ay ay ay ay v  v  v  v  v
f f  ay ay ay ay v  v  v  v
f f  f  f  ay ay ay ay v  v
f f  ay ay ay ay ay ay v  v
f f  ay ay ay ay ay ay ay v
f f  ay ay ay ay ay v  v  v
...
```

The forward algorithm efficiently sums over this large number of sequences in $O(N^2T)$ time.

Let's quickly review the forward algorithm. It is a dynamic programming algorithm, i.e. an algorithm that uses a table to store intermediate values as it builds up the probability of the observation sequence. The forward algorithm computes the observation probability by summing over the probabilities of all possible paths that could generate the observation sequence.

Each cell of the forward algorithm trellis $\alpha_t(j)$ or *forward*$[t, j]$ represents the probability of being in state j after seeing the first t observations, given the automaton λ. The

DRAFT

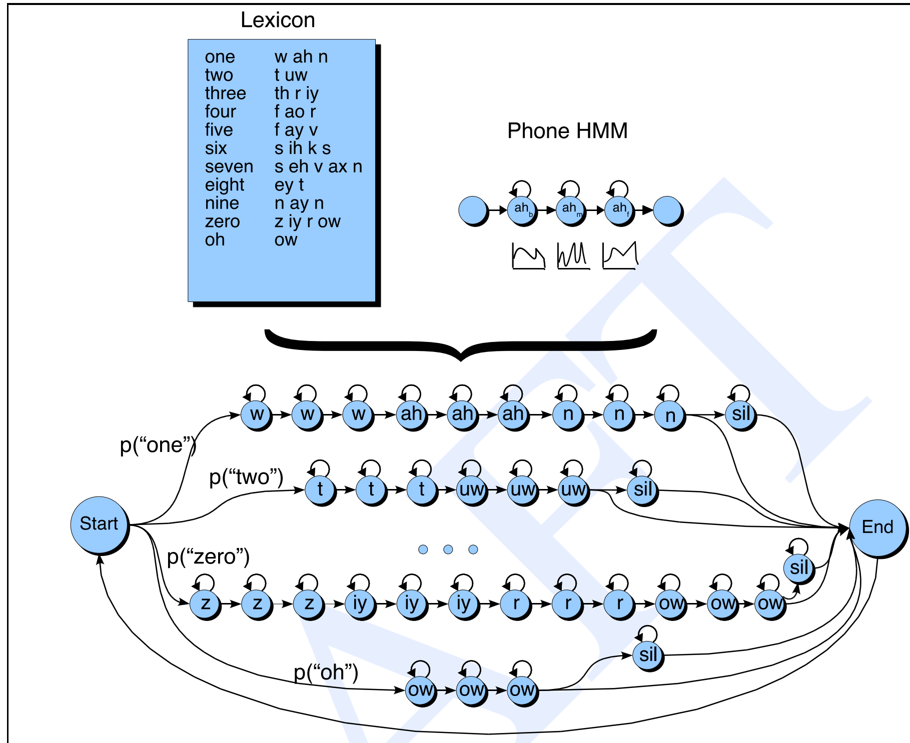

Figure 9.22 An HMM for the digit recognition task. A lexicon specifies the phone sequence, and each phone HMM is composed of three subphones each with a Gaussian emission likelihood model. Combining these and adding an optional silence at the end of each word, results in a single HMM for the whole task. Note the transition from the End state to the Start state to allow digit sequences of arbitrary length.

value of each cell $\alpha_t(j)$ is computed by summing over the probabilities of every path that could lead us to this cell. Formally, each cell expresses the following probability:

$$\alpha_t(j) = P(o_1, o_2 \dots o_t, q_t = j|\lambda) \tag{9.51}$$

Here $q_t = j$ means "the probability that the tth state in the sequence of states is state j". We compute this probability by summing over the extensions of all the paths that lead to the current cell. For a given state q_j at time t, the value $\alpha_t(j)$ is computed as:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t) \tag{9.52}$$

The three factors that are multiplied in Eq˙ 9.52 in extending the previous paths to compute the forward probability at time t are:

DRAFT

$\alpha_{t-1}(i)$	the **previous forward path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

The algorithm is described in Fig. 9.23.

```
function FORWARD(observations of len T, state-graph of len N) returns forward-prob

  create a probability matrix forward[N+2,T]
  for each state s from 1 to N do                    ;initialization step
      forward[s,1] ← a_{0,s} * b_s(o_1)
  for each time step t from 2 to T do                ;recursion step
    for each state s from 1 to N do
                   forward[s,t] ← Σ_{s'=1}^{N} forward[s',t-1] * a_{s',s} * b_s(o_t)
  forward[q_F,T] ← Σ_{s=1}^{N} forward[s,T] * a_{s,q_F}          ; termination step
  return forward[q_F, T]
```

Figure 9.23 The forward algorithm for computing likelihood of observation sequence given a word model. $a[s,s']$ is the transition probability from current state s to next state s', and $b[s',o_t]$ is the observation likelihood of *s'* given o_t. The observation likelihood $b[s',o_t]$ is computed by the **acoustic model**.

Let's see a trace of the forward algorithm running on a simplified HMM for the single word *five* given 10 observations; assuming a frame shift of 10ms, this comes to 100ms. The HMM structure is shown vertically along the left of Fig. 9.24, followed by the first 3 time-steps of the forward trellis. The complete trellis is shown in Fig. 9.6, together with B values giving a vector of observation likelihoods for each frame. These likelihoods could be computed by any acoustic model (GMMs or other); in this example we've hand-created simple values for pedagogical purposes.

Let's now turn to the question of decoding. Recall the Viterbi decoding algorithm from our description of HMMs in Ch. 6. The Viterbi algorithm returns the most likely state sequence (which is not the same as the most likely word sequence, but is often a good enough approximation) in time $O(N^2T)$.

Each cell of the Viterbi trellis, $v_t(j)$ represents the probability that the HMM is in state j after seeing the first t observations and passing through the most likely state sequence $q_1...q_{t-1}$, given the automaton λ. The value of each cell $v_t(j)$ is computed by recursively taking the most probable path that could lead us to this cell. Formally, each cell expresses the following probability:

$$v_t(j) = P(q_0, q_1...q_{t-1}, o_1, o_2 \ldots o_t, q_t = j|\lambda) \quad (9.53)$$

Like other dynamic programming algorithms, Viterbi fills each cell recursively. Given that we had already computed the probability of being in every state at time $t-1$, We compute the Viterbi probability by taking the most probable of the extensions

DRAFT

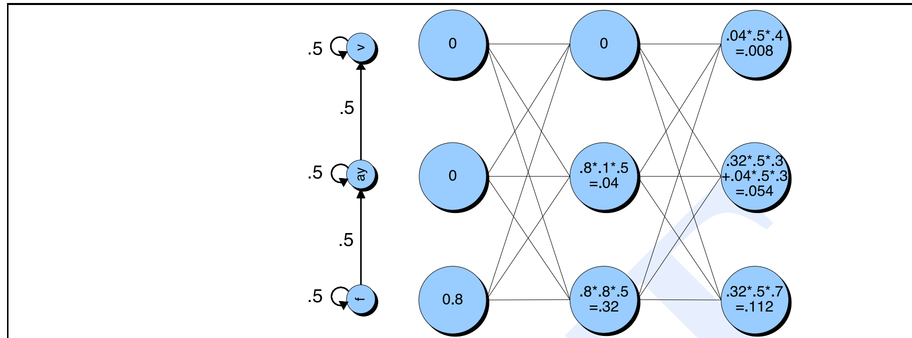

Figure 9.24 The first 3 time-steps of the forward trellis computation for the word *five*. The A transition probabilities are shown along the left edge; the B observation likelihoods are shown in Fig. 9.6.

V	0	0	0.008	0.0093	0.0114	0.00703	0.00345	0.00306	0.00206	0.00117
AY	0	0.04	0.054	0.0664	0.0355	0.016	0.00676	0.00208	0.000532	0.000109
F	0.8	0.32	0.112	0.0224	0.00448	0.000896	0.000179	4.48e-05	1.12e-05	2.8e-06
Time	1	2	3	4	5	6	7	8	9	10
	f 0.8	*f 0.8*	*f 0.7*	*f 0.4*	*f 0.4*	*f 0.4*	*f 0.4*	*f 0.5*	*f 0.5*	*f 0.5*
	ay 0.1	*ay 0.1*	*ay 0.3*	*ay 0.8*	*ay 0.8*	*ay 0.8*	*ay 0.8*	*ay 0.6*	*ay 0.5*	*ay 0.4*
B	*v 0.6*	*v 0.6*	*v 0.4*	*v 0.3*	*v 0.3*	*v 0.3*	*v 0.3*	*v 0.6*	*v 0.8*	*v 0.9*
	p 0.4	*p 0.4*	*p 0.2*	*p 0.1*	*p 0.1*	*p 0.1*	*p 0.1*	*p 0.1*	*p 0.3*	*p 0.3*
	iy 0.1	*iy 0.1*	*iy 0.3*	*iy 0.6*	*iy 0.6*	*iy 0.6*	*iy 0.6*	*iy 0.5*	*iy 0.5*	*iy 0.4*

Figure 9.25 The forward trellis for 10 frames of the word *five*, consisting of 3 emitting states (*f*, *ay*, *v*), plus non-emitting start and end states (not shown). The bottom half of the table gives part of the B observation likelihood vector for the observation o at each frame, $p(o|q)$ for each phone q. B values are created by hand for pedagogical purposes. This table assumes the HMM structure for *five* shown in Fig. 9.24, each emitting state having a .5 loopback probability.

of the paths that lead to the current cell. For a given state q_j at time t, the value $v_t(j)$ is computed as:

$$v_t(j) \;=\; \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t) \tag{9.54}$$

The three factors that are multiplied in Eq. 9.54 for extending the previous paths to compute the Viterbi probability at time t are:

$v_{t-1}(i)$	the **previous Viterbi path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

Fig. 9.26 shows the Viterbi algorithm, repeated from Ch. 6.

Recall that the goal of the Viterbi algorithm is to find the best state sequence $q = (q_1q_2q_3 \ldots q_T)$ given the set of observations $o = (o_1o_2o_3 \ldots o_T)$. It needs to also find

```
function VITERBI(observations of len T,state-graph of len N) returns best-path

  create a path probability matrix viterbi[N+2,T]
  for each state s from 1 to N do                          ;initialization step
      viterbi[s,1] ← a_{0,s} * b_s(o_1)
      backpointer[s,1] ← 0
  for each time step t from 2 to T do                       ;recursion step
    for each state s from 1 to N do
      viterbi[s,t] ← max_{s'=1}^{N} viterbi[s',t−1] * a_{s',s} * b_s(o_t)
      backpointer[s,t] ← argmax_{s'=1}^{N} viterbi[s',t−1] * a_{s',s}
  viterbi[q_F,T] ← max_{s=1}^{N} viterbi[s,T] * a_{s,q_F}            ; termination step
  backpointer[q_F,T] ← argmax_{s=1}^{N} viterbi[s,T] * a_{s,q_F}     ; termination step
    return the backtrace path by following backpointers to states back in time from
backpointer[q_F,T]
```

Figure 9.26 Viterbi algorithm for finding optimal sequence of hidden states. Given an observation sequence of words and an HMM (as defined by the A and B matrices), the algorithm returns the state-path through the HMM which assigns maximum likelihood to the observation sequence. $a[s',s]$ is the transition probability from previous state s' to current state s, and $b_s(o_t)$ is the observation likelihood of s given o_t. Note that states 0 and F are non-emitting start and end states.

the probability of this state sequence. Note that the Viterbi algorithm is identical to the forward algorithm except that it takes the MAX over the previous path probabilities where forward takes the SUM.

Fig. 9.27 shows the computation of the first three time-steps in the Viterbi trellis corresponding to the forward trellis in Fig. 9.24. We have again used the made-up probabilities for the cepstral observations; here we also follow common convention in not showing the zero cells in the upper left corner. Note that only the middle cell in the third column differs from Viterbi to forward. Fig. 9.6 shows the complete trellis.

Note the difference between the final values from the Viterbi and forward algorithms for this (made-up) example. The forward algorithm gives the probability of the observation sequence as .00128, which we get by summing the final column. The Viterbi algorithm gives the probability of the observation sequence given the best path, which we get from the Viterbi matrix as .000493. The Viterbi probability is much smaller than the forward probability, as we should expect since Viterbi comes from a single path, where the forward probability is the sum over all paths.

The real usefulness of the Viterbi decoder, of course, lies in its ability to decode a string of words. In order to do cross-word decoding, we need to augment the A matrix, which only has intra-word state transitions, with the inter-word probability of transitioning from the end of one word to the beginning of another word. The digit HMM model in Fig. 9.22 showed that we could just treat each word as independent, and use only the unigram probability. Higher-order N-grams are much more common. Fig. 9.29, for example, shows an augmentation of the digit HMM with bigram proba-

DRAFT

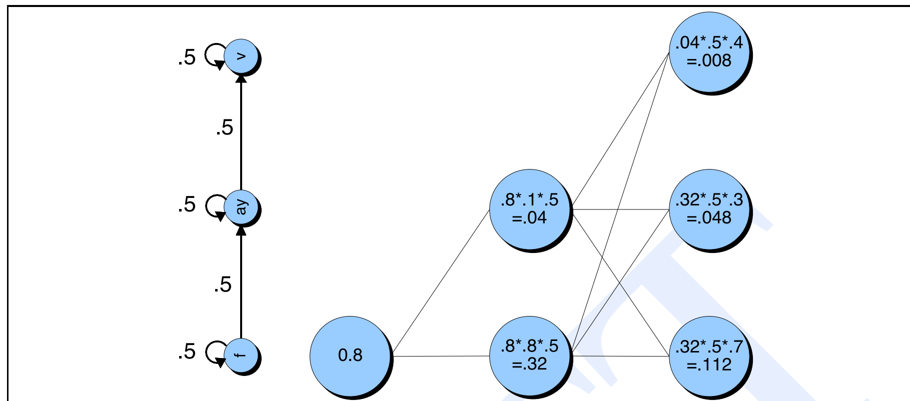

Figure 9.27 The first 3 time-steps of the viterbi trellis computation for the word *five*. The *A* transition probabilities are shown along the left edge; the *B* observation likelihoods are shown in Fig. 9.6. In this computation we make the simplifying assumption that the probability of starting in state 1 (phone [f]) is 1.0

V	0	0	0.008	0.0072	0.00672	0.00403	0.00188	0.00161	0.000667	0.000493
AY	0	0.04	0.048	0.0448	0.0269	0.0125	0.00538	0.00167	0.000428	8.78e-05
F	0.8	0.32	0.112	0.0224	0.00448	0.000896	0.000179	4.48e-05	1.12e-05	2.8e-06
Time	1	2	3	4	5	6	7	8	9	10
	f 0.8	*f 0.8*	*f 0.7*	*f 0.4*	*f 0.4*	*f 0.4*	*f 0.4*	*f 0.5*	*f 0.5*	*f 0.5*
	ay 0.1	*ay 0.1*	*ay 0.3*	*ay 0.8*	*ay 0.8*	*ay 0.8*	*ay 0.8*	*ay 0.6*	*ay 0.5*	*ay 0.4*
B	*v 0.6*	*v 0.6*	*v 0.4*	*v 0.3*	*v 0.3*	*v 0.3*	*v 0.3*	*v 0.6*	*v 0.8*	*v 0.9*
	p 0.4	*p 0.4*	*p 0.2*	*p 0.1*	*p 0.1*	*p 0.1*	*p 0.1*	*p 0.1*	*p 0.3*	*p 0.3*
	iy 0.1	*iy 0.1*	*iy 0.3*	*iy 0.6*	*iy 0.6*	*iy 0.6*	*iy 0.6*	*iy 0.5*	*iy 0.5*	*iy 0.4*

Figure 9.28 The Viterbi trellis for 10 frames of the word *five*, consisting of 3 emitting states (*f*, *ay*, *v*), plus non-emitting start and end states (not shown). The bottom half of the table gives part of the *B* observation likelihood vector for the observation o at each frame, $p(o|q)$ for each phone q. *B* values are created by hand for pedagogical purposes. This table assumes the HMM structure for *five* shown in Fig. 9.24, each emitting state having a .5 loopback probability.

bilities.

A schematic of the HMM trellis for such a multi-word decoding task is shown in Fig. 9.30. The intraword transitions are exactly as shown in Fig. 9.27. But now between words we've added a transition. The transition probability on this arc, rather than coming from the A matrix inside each word, comes from the language model $P(W)$.

Once the entire Viterbi trellis has been computed for the utterance, we can start from the most-probable state at the final time step and follow the backtrace pointers backwards to get the most probable string of states, and hence the most probable string of words. Fig. 9.31 shows the backtrace pointers being followed back from the best state, which happens to be at w_2, eventually through w_N and w_1, resulting in the final word string $w_1 w_N \cdots w_2$.

The Viterbi algorithm is much more efficient than exponentially running the forward algorithm for each possible word string. Nonetheless, it is still slow, and much modern research in speech recognition has focused on speeding up the decoding pro-

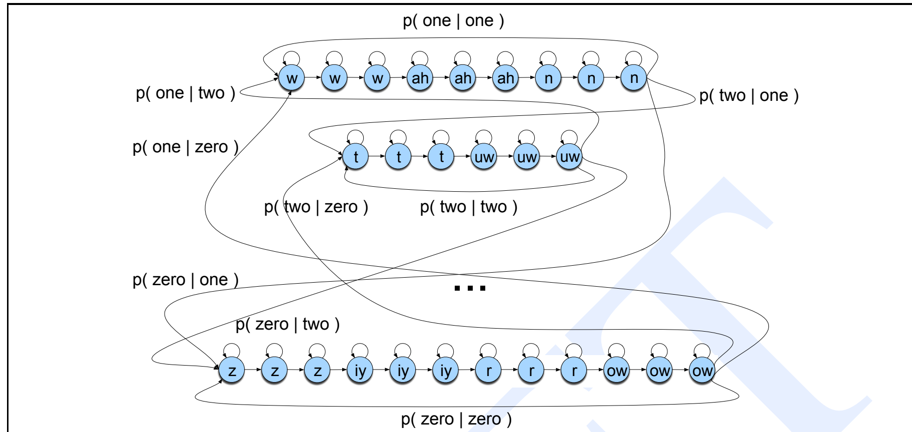

Figure 9.29 A bigram grammar network for the digit recognition task. The bigrams give the probability of transitioning from the end of one word to the beginning of the next.

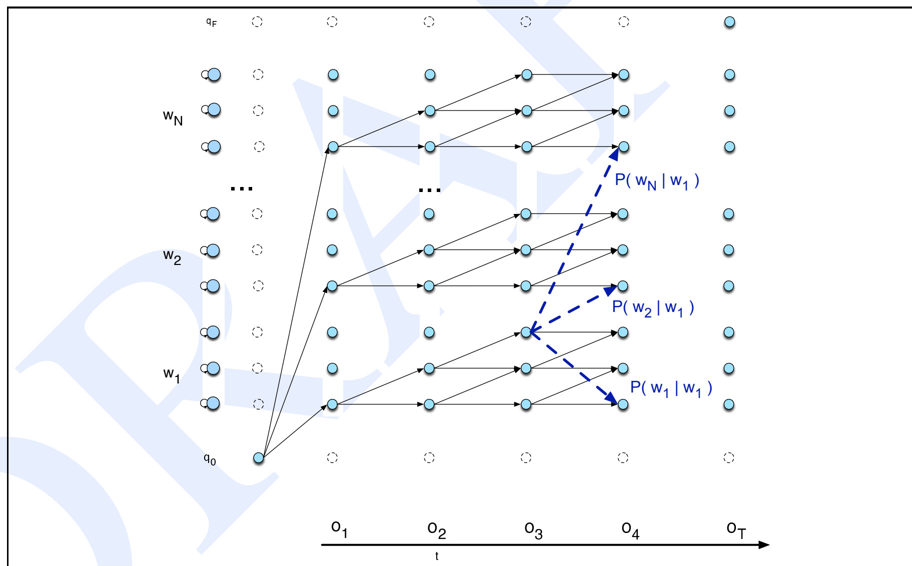

Figure 9.30 A schematic of the HMM Viterbi trellis for a bigram language model. The intraword transitions are the same as in Fig. 9.27. Between words, a potential transition is added (shown just from w_1 as a dark dashed line) from the end state of each word to the beginning state of every word, labeled with the bigram probability of the word pair.

cess. For example in practice in large-vocabulary recognition we do not consider all possible words when the algorithm is extending paths from one state-column to the next. Instead, low-probability paths are **pruned** at each time step and not extended to the next state column.

Pruning

Beam search

This pruning is usually implemented via **beam search** (Lowerre, 1968). In beam search, at each time t, we first compute the probability of the best (most-probable) state/path D. We then prune away any state which is worse than D by some fixed

DRAFT

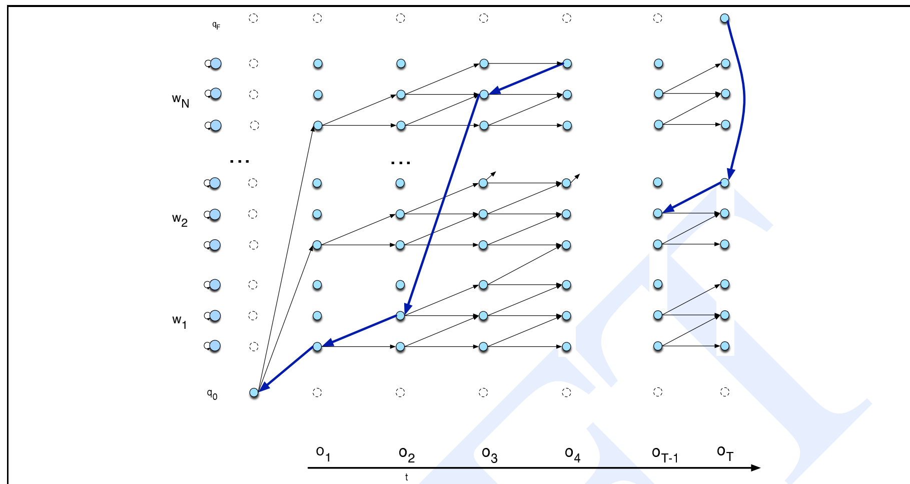

Figure 9.31 Viterbi backtrace in the HMM trellis. The backtrace starts in the final state, and results in a best phone string from which a word string is derived.

Beam width

threshold (**beam width**) θ. We can talk about beam-search in both the probability and negative log probability domain. In the probability domain any path/state whose probability is less than $\theta * D$ is pruned away; in the negative log domain, any path whose cost is greater then $\theta + D$ is pruned. Beam search is implemented by keeping for each time step an **active list** of states. Only transitions from these words are extended when moving to the next time step.

Active list

Making this beam search approximation allows a significant speed-up at the cost of a degradation to the decoding performance. Huang et al. (2001) suggest that empirically a beam size of 5-10% of the search space is sufficient; 90-95% of the states are thus not considered. Because in practice most implementations of Viterbi use beam search, some of the literature uses the term **beam search** or **time-synchronous beam search** instead of Viterbi.

9.7 Embedded Training

We turn now to see how an HMM-based speech recognition system is trained. We've already seen some aspects of training. In Ch. 4 we showed how to train a language model. In Sec. 9.4, we saw how GMM acoustic models are trained by augmenting the EM algorithm to deal with training the means, variances, and weights. We also saw how posterior AM classifiers like SVMs or neural nets could be trained, although for neural nets we haven't yet seen how we get training data in which each frame is labeled with a phone identity.

In this section we complete the picture of HMM training by showing how this augmented EM training algorithm fits into the whole process of training acoustic models. For review, here are the three components of the **acoustic model**:

$Q = q_1 q_2 \ldots q_N$	the **subphones** represented as a set of **states**
$A = a_{01} a_{02} \ldots a_{n1} \ldots a_{nn}$	a **subphone transition probability matrix** A, each a_{ij} representing the probability for each subphone of taking a **self-loop** or going to the next subphone. Together, Q and A implement a **pronunciation lexicon**, an HMM state graph structure for each word that the system is capable of recognizing.
$B = b_i(o_t)$	A set of **observation likelihoods:**, also called **emission probabilities**, each expressing the probability of a cepstral feature vector (observation o_t) being generated from subphone state i.

We will assume that the pronunciation lexicon, and thus the basic HMM state graph structure for each word, is pre-specified as the simple linear HMM structures with loopbacks on each state that we saw in Fig. 9.7 and Fig. 9.22. In general, speech recognition systems do not attempt to learn the structure of the individual word HMMs. Thus we only need to train the B matrix, and we need to train the probabilities of the non-zero (self-loop and next-subphone) transitions in the A matrix. All the other probabilities in the A matrix are set to zero and never change.

The simplest possible training method, is **hand-labeled isolated word** training, in which we train separate the B and A matrices for the HMMs for each word based on hand-aligned training data. We are given a training corpus of digits, where each instance of a spoken digit is stored in a wavefile, and with the start and end of each word and phone hand-segmented. Given such a hand-labeled database, we can compute the B Gaussians observation likelihoods and the A transition probabilities by merely counting in the training data! The A transition probability are specific to each word, but the B Gaussians would be shared across words if the same phone occurred in multiple words.

Unfortunately, hand-segmented training data is rarely used in training systems for continuous speech. One reason is that it is very expensive to use humans to hand-label phonetic boundaries; it can take up to 400 times real time (i.e. 400 labeling hours to label each 1 hour of speech). Another reason is that humans don't do phonetic labeling very well for units smaller than the phone; people are bad at consistently finding the boundaries of subphones. ASR systems aren't better than humans at finding boundaries, but their errors are at least consistent between the training and test sets.

For this reason, speech recognition systems train each phone HMM embedded in an entire sentence, and the segmentation and phone alignment are done automatically as part of the training procedure. This entire acoustic model training process is therefore called **embedded training**. Hand phone segmentation do still play some role, however, for example for bootstrapping initial systems for discriminative (SVM; non-Gaussian) likelihood estimators, or for tasks like phone recognition.

Embedded training

In order to train a simple digits system, we'll need a training corpus of spoken digit sequences. For simplicity assume that the training corpus is separated into separate wavefiles, each containing a sequence of spoken digits. For each wavefile, we'll need to know the correct sequence of digit words. We'll thus associate with each wavefile a

transcription (a string of words). We'll also need a pronunciation lexicon and a phoneset, defining a set of (untrained) phone HMMs. From the transcription, lexicon, and phone HMMs, we can build a "whole sentence" HMM for each sentence, as shown in Fig. 9.32.

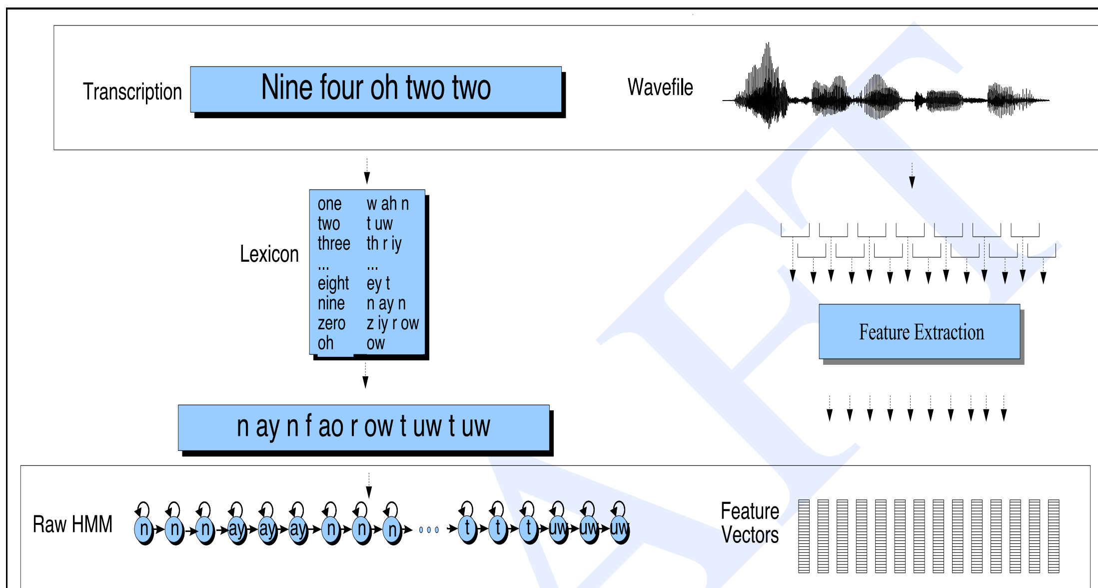

Figure 9.32 The input to the embedded training algorithm; a wavefile of spoken digits with a corresponding transcription. The transcription is converted into a raw HMM, ready to be aligned and trained against the cepstral features extracted from the wavefile.

We are now ready to train the transition matrix A and output likelihood estimator B for the HMMs. The beauty of the Baum-Welch-based paradigm for embedded training of HMMs is that this is all the training data we need. In particular, we don't need phonetically transcribed data. We don't even need to know where each word starts and ends. The Baum-Welch algorithm will sum over all possible segmentations of words and phones, using $\xi_j(t)$, the probability of being in state j at time t and generating the observation sequence O.

Flat start

We will, however, need an initial estimate for the transition and observation probabilities a_{ij} and $b_j(o_t)$. The simplest way to do this is with a **flat start**. In flat start, we first set to zero any HMM transitions that we want to be 'structurally zero', such as transitions from later phones back to earlier phones. The γ probability computation in Baum-Welch includes the previous value of a_{ij}, so those zero values will never change. Then we make all the rest of the (non-zero) HMM transitions equiprobable. Thus the two transitions out of each state (the self-loop and the transition to the following subphone) each would have a probability of 0.5. For the Gaussians, a flat start initializes the mean and variance for each Gaussian identically, to the global mean and variance for the entire training data.

Now we have initial estimates for the A and B probabilities. For a standard Gaussian HMM system, we now run multiple iterations of the Baum-Welch algorithm on

the entire training set. Each iteration modifies the HMM parameters, and we stop when the system converges. During each iteration, as discussed in Ch. 6, we compute the forward and backward probabilities for each sentence given the initial A and B probabilities, and use them to re-estimate the A and B probabilities. We also apply the various modifications to EM discussed in the previous section to correctly update the Gaussian means and variances for multivariate Gaussians. We will discuss in Sec. 10.3 in Ch. 10 how to modify the embedded training algorithm to handle mixture Gaussians.

In summary, the basic **embedded training procedure** is as follows:

Given: phoneset, pronunciation lexicon, and the transcribed wavefiles

1. Build a "whole sentence" HMM for each sentence, as shown in Fig. 9.32.
2. Initialize A probabilities to 0.5 (for loop-backs or for the correct next subphone) or to zero (for all other transitions).
3. Initialize B probabilities by setting the mean and variance for each Gaussian to the global mean and variance for the entire training set.
4. Run multiple iterations of the Baum-Welch algorithm.

The Baum-Welch algorithm is used repeatedly as a component of the embedded training process. Baum-Welch computes $\xi_t(i)$, the probability of being in state i at time t, by using forward-backward to sum over all possible paths that were in state i emitting symbol o_t at time t. This lets us accumulate counts for re-estimating the emission probability $b_j(o_t)$ from all the paths that pass through state j at time t. But Baum-Welch itself can be time-consuming.

Viterbi training

There is an efficient approximation to Baum-Welch training that makes use of the Viterbi algorithm. In **Viterbi training**, instead of accumulating counts by a sum over all paths that pass through a state j at time t, we approximate this by only choosing the Viterbi (most-probable) path. Thus instead of running EM at every step of the embedded training, we repeatedly run Viterbi.

Forced alignment

Running the Viterbi algorithm over the training data in this way is called **forced Viterbi alignment** or just **forced alignment**. In Viterbi training (unlike in Viterbi decoding on the test set) we know which word string to assign to each observation sequence, So we can 'force' the Viterbi algorithm to pass through certain words, by setting the a_{ij}s appropriately. A forced Viterbi is thus a simplification of the regular Viterbi decoding algorithm, since it only has to figure out the correct state (subphone) sequence, but doesn't have to discover the word sequence. The result is a **forced alignment**: the single best state path corresponding to the training observation sequence. We can now use this alignment of HMM states to observations to accumulate counts for re-estimating the HMM parameters. We saw earlier that forcd alignment can also be used in other speech applications like text-to-speech, whenver we have a word transcript and a wavefile in which we want to find boundaries.

The equations for retraining a (non-mixture) Gaussian from a Viterbi alignment are as follows:

$$\hat{\mu}_i = \frac{1}{T}\sum_{t=1}^{T} o_t \text{ s.t. } q_t \text{ is state } i \tag{9.55}$$

DRAFT

$$\hat{\sigma}_j^2 = \frac{1}{T}\sum_{t=1}^{T}(o_t - \mu_i)^2 \text{ s.t. } q_t \text{ is state } i \tag{9.56}$$

We saw these equations already, as (9.27) and (9.28) on page 310, when we were 'imagining the simpler situation of a completely labeled training set'.

It turns out that this forced Viterbi algorithm is also used in the embedded training of hybrid models like HMM/MLP or HMM/SVM systems. We begin with an untrained MLP, and using its noisy outputs as the B values for the HMM, perform a forced Viterbi alignment of the training data. This alignment will be quite errorful, since the MLP was random. Now this (quite errorful) Viterbi alignment give us a labeling of feature vectors with phone labels. We use this labeling to retrain the MLP. The counts of the transitions which are taken in the forced alignments can be used to estimate the HMM transition probabilities. We continue this hill-climbing process of neural-net training and Viterbi alignment until the HMM parameters begin to converge.

9.8 Evaluation: Word Error Rate

Word error

The standard evaluation metric for speech recognition systems is the **word error** rate. The word error rate is based on how much the word string returned by the recognizer (often called the **hypothesized** word string) differs from a correct or **reference** transcription. Given such a correct transcription, the first step in computing word error is to compute the **minimum edit distance** in words between the hypothesized and correct strings, as described in Ch. 3. The result of this computation will be the minimum number of word **substitutions**, word **insertions**, and word **deletions** necessary to map between the correct and hypothesized strings. The word error rate (WER) is then defined as follows (note that because the equation includes insertions, the error rate can be greater than 100%):

$$\text{Word Error Rate} = 100 \times \frac{\text{Insertions} + \text{Substitutions} + \text{Deletions}}{\text{Total Words in Correct Transcript}}$$

We sometimes also talk about the SER (Sentence Error Rate), which tells us how many sentences had at least one error:

$$\text{Sentence Error Rate} = 100 \times \frac{\text{\# of sentences with at least one word error}}{\text{total \# of sentences}}$$

Alignment

Here is an example of the **alignments** between a reference and a hypothesized utterance from the CALLHOME corpus, showing the counts used to compute the word error rate:

REF:	i	***	**	UM	the	PHONE	IS	i	LEFT	THE	portable	****	PHONE	UPSTAIRS	last	night
HYP:	i	GOT	IT	TO	the	*****	FULLEST	i	LOVE	TO	portable	FORM	OF	STORES	last	night
Eval:		I	I	S		D	S		S	S		I	S	S		

This utterance has six substitutions, three insertions, and one deletion:

$$\text{Word Error Rate} = 100\frac{6+3+1}{13} = 76.9\%$$

The standard method for implementing minimum edit distance and computing word error rates is a free script called `sclite`, available from the National Institute of Standards and Technologies (NIST) (NIST, 2005). `sclite` is given a series of reference (hand-transcribed, gold-standard) sentences and a matching set of hypothesis sentences. Besides performing alignments, and computing word error rate, sclite performs a number of other useful tasks. For example, it gives useful information for **error analysis**, such as confusion matrices showing which words are often misrecognized for others, and gives summary statistics of words which are often inserted or deleted. `sclite` also gives error rates by speaker (if sentences are labeled for speaker id), as well as useful statistics like the **sentence error rate**, the percentage of sentences with at least one word error.

Sentence error rate

Finally, `sclite` can be used to compute significance tests. Suppose we make some changes to our ASR system and find that our word error rate has decreased by 1%. In order to know if our changes really improved things, we need a statistical test to make sure that the 1% difference is not just due to chance. The standard statistical test for determining if two word error rates are different is the Matched-Pair Sentence Segment Word Error (MAPSSWE) test, which is also available in `sclite` (although the **McNemar test** is sometimes used as well).

McNemar test

The MAPSSWE test is a parametric test that looks at the difference between the number of word errors the two systems produce, averaged across a number of segments. The segments may be quite short or as long as an entire utterance; in general we want to have the largest number of (short) segments in order to justify the normality assumption and for maximum power. The test requires that the errors in one segment be statistically independent of the errors in another segment. Since ASR systems tend to use trigram LMs, this can be approximated by defining a segment as a region bounded on both sides by words that both recognizers get correct (or turn/utterance boundaries).

Here's an example from NIST (2007b) with four regions:

```
           I            II             III             IV
REF:   |it was|the best|of|times it|was the worst|of times|  |it was
       |      |        |  |        |             |        |  |
SYS A: |ITS   |the best|of|times it|IS the worst |of times|OR|it was
       |      |        |  |        |             |        |  |
SYS B: |it was|the best|  |times it|WON the TEST |of times|  |it was
```

In region I, system A has 2 errors (a deletion and an insertion) and system B has 0; in region III system A has 1 (substitution) error and system B has 2. Let's define a sequence of variables Z representing the difference between the errors in the two systems as follows:

N_A^i	the number of errors made on segment i by system A
N_B^i	the number of errors made on segment i by system B
Z	$N_A^i - N_B^i, i = 1, 2, \cdots, n$ where n is the number of segments

For example in the example above the sequence of Z values is $\{2, -1, -1, 1\}$. Intuitively, if the two systems are identical, we would expect the average difference, i.e. the average of the Z values, to be zero. If we call the true average of the differences mu_z, we would thus like to know whether $mu_z = 0$. Following closely the original pro-

DRAFT

posal and notation of Gillick and Cox (1989), we can estimate the true average from our limited sample as $\hat{\mu}_z = \sum_{i=1}^{n} Z_i/n$.

The estimate of the variance of the Z_i's is:

$$\sigma_z^2 = \frac{1}{n-1}\sum_{i=1}^{n}(Z_i - \mu_z)^2 \tag{9.57}$$

Let

$$W = \frac{\hat{\mu}_z}{\sigma_z/\sqrt{n}} \tag{9.58}$$

For a large enough n (> 50) W will approximately have a normal distribution with unit variance. The null hypothesis is $H_0 : \mu_z = 0$, and it can thus be rejected if $2 * P(Z \geq |w|) \leq 0.05$ (two-tailed) or $P(Z \geq |w|) \leq 0.05$ (one-tailed). where Z is standard normal and w is the realized value W; these probabilities can be looked up in the standard tables of the normal distribution.

Could we improve on word error rate as a metric? It would be nice, for example, to have something which didn't give equal weight to every word, perhaps valuing content words like *Tuesday* more than function words like *a* or *of*. While researchers generally agree that this would be a good idea, it has proved difficult to agree on a metric that works in every application of ASR. For dialogue systems, however, where the desired semantic output is more clear, a metric called *concept error rate* has proved extremely useful, and will be discussed in Ch. 24 on page 851.

9.9 Summary

Together with Ch. 4 and Ch. 6, this chapter introduced the fundamental algorithms for addressing the problem of **Large Vocabulary Continuous Speech Recognition**.

- The input to a speech recognizer is a series of acoustic waves. The **waveform**, **spectrogram** and **spectrum** are among the visualization tools used to understand the information in the signal.
- In the first step in speech recognition, sound waves are **sampled**, **quantized**, and converted to some sort of **spectral representation**; A commonly used spectral representation is the **mel cepstrum** or **MFCC** which provides a vector of features for each frame of the input.
- GMM acoustic models are used to estimate the **phonetic likelihoods** (also called **observation likelihoods**) of these **feature vectors** for each frame.
- **Decoding** or **search** or **inference** is the process of finding the optimal sequence of model states which matches a sequence of input observations. (The fact that there are three terms for this process is a hint that speech recognition is inherently inter-disciplinary, and draws its metaphors from more than one field; **decoding** comes from information theory, and **search** and **inference** from artificial intelligence).

DRAFT

- We introduced two decoding algorithms: time-synchronous **Viterbi** decoding (which is usually implemented with pruning and can then be called **beam search**) and **stack** or $\mathbf{A}^*$ decoding. Both algorithms take as input a sequence of cepstral feature vectors, a GMM acoustic model, and an N-gram language model, and produce a string of words.
- The **embedded training** paradigm is the normal method for training speech recognizers. Given an initial lexicon with hand-built pronunciation structures, it will train the HMM transition probabilities and the HMM observation probabilities.

Bibliographical and Historical Notes

The first machine which recognized speech was probably a commercial toy named "Radio Rex" which was sold in the 1920s. Rex was a celluloid dog that moved (via a spring) when the spring was released by 500 Hz acoustic energy. Since 500 Hz is roughly the first formant of the vowel [eh] in "Rex", the dog seemed to come when he was called (David and Selfridge, 1962).

By the late 1940s and early 1950s, a number of machine speech recognition systems had been built. An early Bell Labs system could recognize any of the 10 digits from a single speaker (Davis et al., 1952). This system had 10 speaker-dependent stored patterns, one for each digit, each of which roughly represented the first two vowel formants in the digit. They achieved 97–99% accuracy by choosing the pattern which had the highest relative correlation coefficient with the input. Fry (1959) and Denes (1959) built a phoneme recognizer at University College, London, which recognized four vowels and nine consonants based on a similar pattern-recognition principle. Fry and Denes's system was the first to use phoneme transition probabilities to constrain the recognizer.

The late 1960s and early 1970s produced a number of important paradigm shifts. First were a number of feature-extraction algorithms, include the efficient Fast Fourier Transform (FFT) (Cooley and Tukey, 1965), the application of cepstral processing to speech (Oppenheim et al., 1968), and the development of LPC for speech coding (Atal and Hanauer, 1971). Second were a number of ways of handling **warping**; stretching or shrinking the input signal to handle differences in speaking rate and segment length when matching against stored patterns. The natural algorithm for solving this problem was dynamic programming, and, as we saw in Ch. 6, the algorithm was reinvented multiple times to address this problem. The first application to speech processing was by Vintsyuk (1968), although his result was not picked up by other researchers, and was reinvented by Velichko and Zagoruyko (1970) and Sakoe and Chiba (1971) (and (1984)). Soon afterward, Itakura (1975) combined this dynamic programming idea with the LPC coefficients that had previously been used only for speech coding. The resulting system extracted LPC features for incoming words and used dynamic programming to match them against stored LPC templates. The non-probabistic use of dynamic programming to match a template against incoming speech is called **dynamic time warping**.

Warping

Dynamic time warping

DRAFT

The third innovation of this period was the rise of the HMM. Hidden Markov Models seem to have been applied to speech independently at two laboratories around 1972. One application arose from the work of statisticians, in particular Baum and colleagues at the Institute for Defense Analyses in Princeton on HMMs and their application to various prediction problems (Baum and Petrie, 1966; Baum and Eagon, 1967). James Baker learned of this work and applied the algorithm to speech processing (Baker, 1975) during his graduate work at CMU. Independently, Frederick Jelinek, Robert Mercer, and Lalit Bahl (drawing from their research in information-theoretical models influenced by the work of Shannon (1948)) applied HMMs to speech at the IBM Thomas J. Watson Research Center (Jelinek et al., 1975). IBM's and Baker's systems were very similar, particularly in their use of the Bayesian framework described in this chapter. One early difference was the decoding algorithm; Baker's DRAGON system used Viterbi (dynamic programming) decoding, while the IBM system applied Jelinek's stack decoding algorithm (Jelinek, 1969). Baker then joined the IBM group for a brief time before founding the speech-recognition company Dragon Systems. The HMM approach to speech recognition would turn out to completely dominate the field by the end of the century; indeed the IBM lab was the driving force in extending statistical models to natural language processing as well, including the development of class-based *N*-grams, HMM-based part-of-speech tagging, statistical machine translation, and the use of entropy/perplexity as an evaluation metric.

The use of the HMM slowly spread through the speech community. One cause was a number of research and development programs sponsored by the Advanced Research Projects Agency of the U.S. Department of Defense (ARPA). The first five-year program starting in 1971, and is reviewed in Klatt (1977). The goal of this first program was to build speech understanding systems based on a few speakers, a constrained grammar and lexicon (1000 words), and less than 10% semantic error rate. Four systems were funded and compared against each other: the System Development Corporation (SDC) system, Bolt, Beranek & Newman (BBN)'s HWIM system, Carnegie-Mellon University's Hearsay-II system, and Carnegie-Mellon's Harpy system (Lowerre, 1968). The Harpy system used a simplified version of Baker's HMM-based DRAGON system and was the best of the tested systems, and according to Klatt the only one to meet the original goals of the ARPA project (with a semantic accuracy rate of 94% on a simple task).

Beginning in the mid-1980s, ARPA funded a number of new speech research programs. The first was the "Resource Management" (RM) task (Price et al., 1988), which like the earlier ARPA task involved transcription (recognition) of read-speech (speakers reading sentences constructed from a 1000-word vocabulary) but which now included a component that involved speaker-independent recognition. Later tasks included recognition of sentences read from the Wall Street Journal (WSJ) beginning with limited systems of 5,000 words, and finally with systems of unlimited vocabulary (in practice most systems use approximately 60,000 words). Later speech-recognition tasks moved away from read-speech to more natural domains; the Broadcast News domain (LDC, 1998; Graff, 1997) (transcription of actual news broadcasts, including quite difficult passages such as on-the-street interviews) and the Switchboard, CALLHOME, CALLFRIEND, and Fisher domains (Godfrey et al., 1992; Cieri et al., 2004) (natural telephone conversations between friends or strangers) . The Air Traffic Information

DRAFT

System (ATIS) task (Hemphill et al., 1990) was an earlier speech understanding task whose goal was to simulate helping a user book a flight, by answering questions about potential airlines, times, dates, and so forth.

Bake-off

Each of the ARPA tasks involved an approximately annual **bake-off** at which all ARPA-funded systems, and many other 'volunteer' systems from North American and Europe, were evaluated against each other in terms of word error rate or semantic error rate. In the early evaluations, for-profit corporations did not generally compete, but eventually many (especially IBM and ATT) competed regularly. The ARPA competitions resulted in widescale borrowing of techniques among labs, since it was easy to see which ideas had provided an error-reduction the previous year, and were probably an important factor in the eventual spread of the HMM paradigm to virtual every major speech recognition lab. The ARPA program also resulted in a number of useful databases, originally designed for training and testing systems for each evaluation (TIMIT, RM, WSJ, ATIS, BN, CALLHOME, Switchboard, Fisher) but then made available for general research use.

Speaker identification
Speaker verification
Language idenntification

Speech research includes a number of areas besides speech recognition; we already saw computational phonology in Ch. 7, speech synthesis in Ch. 8, and we will discuss spoken dialogue systems in Ch. 24. Another important area is **speaker identification** and **speaker verification**, in which we identify a speaker (for example for security when accessing personal information over the telephone) (Reynolds and Rose, 1995; Shriberg et al., 2005; Doddington, 2001). This task is related to **language identification**, in which we are given a wavefile and have to identify which language is being spoken; this is useful for automatically directing callers to human operators that speak appropriate languages.

There are a number of textbooks and reference books on speech recognition that are good choices for readers who seek a more in-depth understanding of the material in this chapter: Huang et al. (2001) is by far the most comprehensive and up-to-date reference volume and is highly recommended. Jelinek (1997), Gold and Morgan (1999), and Rabiner and Juang (1993) are good comprehensive textbooks. The last two textbooks also have discussions of the history of the field, and together with the survey paper of Levinson (1995) have influenced our short history discussion in this chapter. Our description of the forward-backward algorithm was modeled after Rabiner (1989), and we were also influenced by another useful tutorial paper, Knill and Young (1997). Research in the speech recognition field often appears in the proceedings of the annual INTERSPEECH conference, (which is called ICSLP and EUROSPEECH in alternate years) as well as the annual IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP). Journals include *Speech Communication*, *Computer Speech and Language*, the *IEEE Transactions on Audio, Speech, and Language Processing*, and the *ACM Transactions on Speech and Language Processing*.

DRAFT

Exercises

9.1 Analyze each of the errors in the incorrectly recognized transcription of "um the phone is I left the..." on page 330. For each one, give your best guess as to whether you think it is caused by a problem in signal processing, pronunciation modeling, lexicon size, language model, or pruning in the decoding search.

Logprob

9.2 In practice, speech recognizers do all their probability computation using the **log probability** (or **logprob**) rather than actual probabilities. This helps avoid underflow for very small probabilities, but also makes the Viterbi algorithm very efficient, since all probability multiplications can be implemented by adding log probabilities. Rewrite the pseudocode for the Viterbi algorithm in Fig. 9.26 on page 323 to make use of logprobs instead of probabilities.

9.3 Now modify the Viterbi algorithm in Fig. 9.26 on page 323 to implement the beam search described on page 325. Hint: You will probably need to add in code to check whether a given state is at the end of a word or not.

9.4 Finally, modify the Viterbi algorithm in Fig. 9.26 on page 323 with more detailed pseudocode implementing the array of backtrace pointers.

9.5 Using the tutorials available as part of a publicly available recognizer like HTK or Sonic, build a digit recognizer.

9.6 Take the digit recognizer above and dump the phone likelihoods for a sentence. Now take your implementation of the Viterbi algorithm and show that you can successfully decode these likelihoods.

DRAFT