7 HMMS AND 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 twenty-first 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 which 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.

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 po-

tential 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 for entering digits, recognizing "yes" to accept collect calls, or call-routing ("Accounting, please", "Prof. Landauer, please"). Finally, ASR is being applied to dictation, i.e. 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.

Different applications of speech technology necessarily place different constraints on the problem and lead to different algorithms. We chose to focus this chapter on the fundamentals of one crucial area: **Large-Vocabulary Continuous Speech Recognition** (**LVCSR**), with a small section on acoustic issues in speech synthesis. Large-vocabulary generally means that the systems have a vocabulary of roughly 5,000 to 60,000 words. The term **continuous** means that the words are run together naturally; it contrasts with **isolated-word** speech recognition, in which each word must be preceded and followed by a pause. 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.

LVCSR

CONTINUOUS

ISOLATED-WORD

SPEAKER-INDEPENDENT

The chapter begins with an overview of speech recognition architecture, and then proceeds to introduce the HMM, the use of the Viterbi and A* algorithms for decoding, speech acoustics and features, and the use of Gaussians and MLPs to compute acoustic probabilities. Even relying on the previous three chapters, summarizing this much of the field in this chapter requires us to omit many crucial areas; the reader is encouraged to see the suggested readings at the end of the chapter for useful textbooks and articles. This chapter also includes a short section on the acoustic component of the speech synthesis algorithms discussed in Chapter 4.

7.1 SPEECH RECOGNITION ARCHITECTURE

Previous chapters have introduced many of the core algorithms used in speech recognition. Chapter 4 introduced the notions of **phone** and **syllable**. Chapter 5 introduced the **noisy channel model**, the use of the **Bayes rule**, and the **probabilistic automaton**. Chapter 6 introduced the ***N*-gram** language model and the **perplexity** metric. In this chapter we introduce the remaining components of a modern speech recognizer: the **Hidden Markov Model** (**HMM**), the idea of **spectral features**, the **forward-backward** algorithm for HMM training, and the **Viterbi** and **stack decoding** (also called **A* decoding** algorithms for solving the **decoding** problem: mapping from strings of phone probability vectors to strings of words.

A* DECODING

Let's begin by revisiting the noisy channel model that we saw in Chapter 5. Speech recognition systems treat the acoustic input as if it were a 'noisy' version of the source sentence. In order to 'decode' this noisy sentence, we consider all possible sentences, and for each one we compute the probability of it generating the noisy sentence. We then chose the sentence with the maximum probability. Figure 7.1 shows this noisy-channel metaphor.

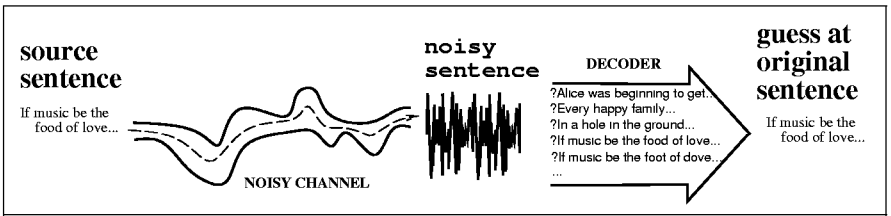

Figure 7.1 The noisy channel model applied to entire sentences (Figure 5.1 showed its application to individual words). Modern speech recognizers work by searching through a huge space of potential 'source' sentences and choosing the one which has the highest probability of generating the 'noisy' sentence. To do this they must have models that express the probability of sentences being realized as certain strings of words (*N*-grams), models that express the probability of words being realized as certain strings of phones (HMMs) and models that express the probability of phones being realized as acoustic or spectral features (Gaussians/MLPs).

Implementing the noisy-channel model as we have expressed it in Figure 7.1 requires 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, and will show how to combine the various probabilistic estimators to get a complete estimate for the probability of a noisy observation-sequence given a candidate sentence. 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, and we will summarize two approaches: the **Viterbi** or **dynamic programming** decoder, and the **stack** or **A*** decoder.

In the rest of this introduction we will introduce the probabilistic or Bayesian model for speech recognition (or more accurately re-introduce it, since we first used the model in our discussions of spelling and pronunciation in Chapter 5); we leave discussion of decoding/search for pages 242–249.

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 $\mathcal{L}$ given some acoustic input O?"*

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):

$$O = o_1, o_2, o_3, \ldots, o_t \tag{7.1}$$

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

$$W = w_1, w_2, w_3, \ldots, w_n \tag{7.2}$$

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'd like to talk about 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..."). Recent ASR research has begun to focus on building more so-

phisticated models of ASR words incorporating the morphological insights of Chapter 3 and the part-of-speech information that we will study in Chapter 8.

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

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} P(W|O) \tag{7.3}$$

Recall that the function $\operatorname{argmax}_x f(x)$ means 'the x such that f(x) is largest'. Equation (7.3) 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:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{7.4}$$

We saw in Chapter 5 that we can substitute (7.4) into (7.3) as follows:

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} \frac{P(O|W)P(W)}{P(O)} \tag{7.5}$$

The probabilities on the right hand of (7.5) 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 exactly what is estimated by the n-gram language models of Chapter 6. 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, turns out to be harder to estimate. Luckily, we can ignore $P(O)$ just as we saw in Chapter 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:

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} \frac{P(O|W)P(W)}{P(O)} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} P(O|W)\,P(W) \tag{7.6}$$

To summarize, the most probable sentence W given some observation sequence O can be computing by taking the product of two probabilities for each sentence, and choosing the sentence for which this product is greatest. These two terms have names; $P(W)$, the **prior probability**, is called the **language model**. $P(O|W)$, the **observation likelihood**, is called the **acoustic model**.

LANGUAGE MODEL

ACOUSTIC MODEL

Key Concept #5.
$$\hat{W} = \operatorname*{argmax}_{W \in \mathcal{L}} \overbrace{P(O|W)}^{\text{likelihood}} \overbrace{P(W)}^{\text{prior}} \tag{7.7}$$

We have already seen in Chapter 6 how to compute the language model prior $P(W)$ by using N-gram grammars. The rest of this chapter will show how to compute the acoustic model $P(O|W)$, in two steps. First we will make the simplifying assumption that the input sequence is a sequence of phones F rather than a sequence of acoustic observations. Recall that we introduced the **forward** algorithm in Chapter 5, which was given 'observations' that were strings of phones, and produced the probability of these phone observations given a single word. We will show that these probabilistic phone automata are really a special case of the **Hidden Markov Model**, and we will show how to extend these models to give the probability of a phone sequence given an entire sentence.

One problem with the forward algorithm as we presented it was that in order to know which word was the most-likely word (the 'decoding problem'), we had to run the forward algorithm again for each word. This is clearly intractable for sentences; we can't possibly run the forward algorithm separately for each possible sentence of English. We will thus introduce two different algorithms which *simultaneously* compute the likelihood of an observation sequence given each sentence, *and* give us the most-likely sentence. These are the **Viterbi** and the **A*** decoding algorithms.

Once we have solved the likelihood-computation and decoding problems for a simplified input consisting of strings of phones, we will show how the same algorithms can be applied to true acoustic input rather than pre-defined phones. This will involve a quick introduction to acoustic input and **feature extraction**, the process of deriving meaningful features from the input soundwave. Then we will introduce the two standard models for computing phone-probabilities from these features: **Gaussian** models, and **neural net (multi-layer perceptrons)** models.

Finally, we will introduce the standard algorithm for training the Hidden Markov Models and the phone-probability estimators, the **forward-backward** or **Baum-Welch** algorithm) (Baum, 1972), a special case of the the **Expectation-Maximization** or **EM** algorithm (Dempster *et al.*, 1977).

As a preview of the chapter, Figure 7.2 shows an outline of the components of a speech recognition system. The figure shows a speech recognition system broken down into three stages. In the **signal processing** or **feature extraction** stage, the acoustic waveform is sliced up into **frames** (usually of 10, 15, or 20 milliseconds) which are transformed into **spectral features**

which give information about how much energy in the signal is at different frequencies. In the **subword** or **phone recognition** stage, we use statistical techniques like neural networks or Gaussian models to tentatively recognize individual speech sounds like *p* or *b*. For a neural network, the output of this stage is a vector of probabilities over phones for each frame (i.e. 'for this frame the probability of [p] is .8, the probability of [b] is .1, the probability of [f] is .02, etc'); for a Gaussian model the probabilities are slightly different. Finally, in the **decoding** stage, we take a dictionary of word pronunciations and a language model (probabilistic grammar) and use a Viterbi or A* **decoder** to find the sequence of words which has the highest probability given the acoustic events. DECODER

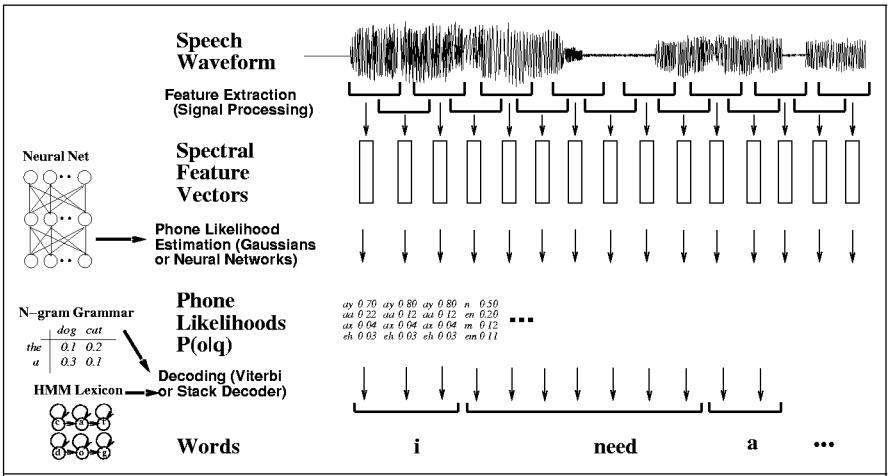

Figure 7.2 Schematic architecture for a (simplified) speech recognizer

7.2 Overview of Hidden Markov Models

In Chapter 5 we used **weighted finite-state automata** or **Markov chains** to model the pronunciation of words. The automata consisted of a sequence of states $q = (q_0 q_1 q_2 \ldots q_n)$, each corresponding to a phone, and a set of transition probabilities between states, a_{01}, a_{12}, a_{13}, encoding the probability of one phone following another. We represented the states as nodes, and the transition probabilities as edges between nodes; an edge existed between two nodes if there was a non-zero transition probability between the two nodes. We also saw that we could use the **forward** algorithm to compute the

likelihood of a sequence of observed phones $o = (o_1 o_2 o_3 \ldots o_t)$. Figure 7.3 shows an automaton for the word *need* with sample observation sequence of the kind we saw in Chapter 5.

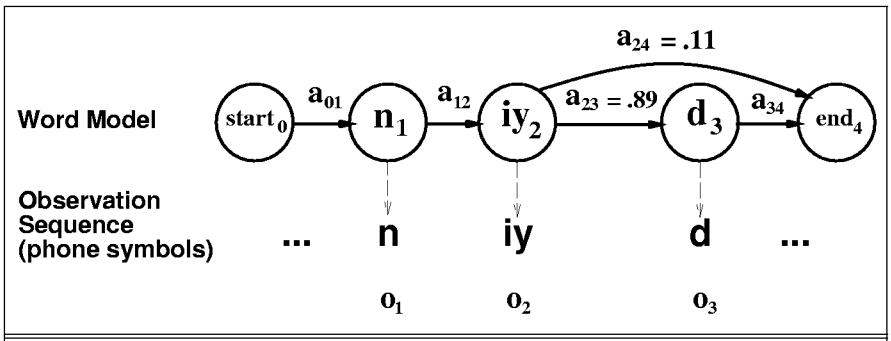

Figure 7.3 A simple weighted automaton or Markov chain pronunciation network for the word *need*, showing the transition probabilities, and a sample observation sequence. The transition probabilities a_{xy} between two states x and y are 1.0 unless otherwise specified.

While we will see that these models figure importantly in speech recognition, they simplify the problem in two ways. First, they assume that the input consists of a sequence of symbols! Obviously this is not true in the real world, where speech input consists essentially of small movements of air particles. In speech recognition, the input is an ambiguous, real-valued representation of the sliced-up input signal, called **features** or **spectral features**. We will study the details of some of these features beginning on page 258; acoustic features represent such information as how much energy there is at different frequencies. The second simplifying assumption of the weighted automata of Chapter 5 was that the input symbols correspond exactly to the states of the machine. Thus when seeing an input symbol [b], we knew that we could move into a state labeled [b]. In a **Hidden Markov Model**, by contrast, we can't look at the input symbols and know which state to move to. The input symbols don't uniquely determine the next state.[1]

HIDDEN MARKOV MODEL

Recall that a weighted automaton or simple Markov model is specified by the set of **states** Q, the set of **transition probabilities** A, a defined **start state** and **end state(s)**, and a set of **observation likelihoods** B. For weighted

[1] Actually, as we mentioned in passing, by this second criterion some of the automata we saw in Chapter 5 were technically HMMs as well. This is because the first symbol in the input string [n iy] was compatible with the [n] states in the words *need* or *an*. Seeing the symbols [n], we didn't know which underlying state it was generated by, *need-n* or *an-n*.

automata, we defined the probabilities $b_i(o_t)$ as 1.0 if the state i matched the observation o_t and 0 if they didn't match. An HMM formally differs from a Markov model by adding two more requirements. First, it has a separate set of *observation symbols* O, which is not drawn from the same alphabet as the state set Q. Second, the observation likelihood function B is not limited to the values 1.0 and 0; in an HMM the probability $b_i(o_t)$ can take on any value from 0 to 1.0.

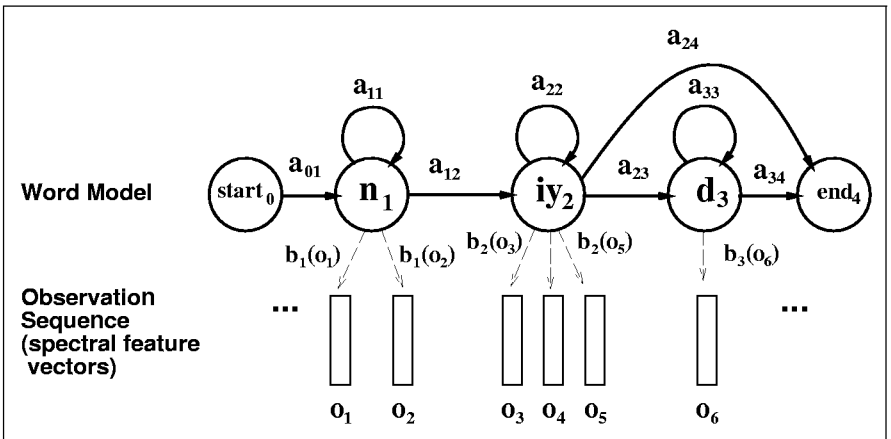

Figure 7.4 An HMM pronunciation network for the word *need*, showing the transition probabilities, and a sample observation sequence. Note the addition of the output probabilities B. HMMs used in speech recognition usually use self-loops on the states to model variable phone durations.

Figure 7.4 shows an HMM for the word *need* and a sample observation sequence. Note the differences from Figure 7.3. First, the observation sequences are now vectors of spectral features representing the speech signal. Next, note that we've also allowed one state to generate multiple copies of the same observation, by having a loop on the state. This loops allows HMMs to model the variable duration of phones; longer phones require more loops through the HMM.

In summary, here are the parameters we need to define an HMM:

- **states:** A set of states $Q = q_1 q_2 \ldots q_N$.
- **transition probabilities:** A set of probabilities $A = a_{01} a_{02} \ldots a_{n1} \ldots a_{nn}$. Each a_{ij} represents the probability of transitioning from state i to state j. The set of these is the **transition probability matrix**¿
- **observation likelihoods:** A set of observation likelihoods $B = b_i(o_t)$,

each expressing the probability of an observation o_t being generated from a state i.

In our examples so far we have used two 'special' states (**non-emitting states**) as the start and end state; as we saw in Chapter 5 it is also possible to avoid the use of these states by specifying two more things:

- **initial distribution:** An initial probability distribution over states, π, such that π_i is the probability that the HMM will start in state i. Of course some states j may have $\pi_j = 0$, meaning that they cannot be initial states.
- **accepting states:** A set of legal accepting states.

As was true for the weighted automata, the sequences of symbols that are input to the model (if we are thinking of it as recognizer) or which are produced by the model (if we are thinking of it as a generator) are generally called the **observation sequence**, referred to as $O = (o_1 o_2 o_3 \ldots o_T)$.

7.3 The Viterbi Algorithm Revisited

Chapter 5 showed how the forward algorithm could be used to compute the probability of an observation sequence given an automaton, and how the Viterbi algorithm can be used to find the most-likely path through the automaton, as well as the probability of the observation sequence given this most-likely path. In Chapter 5 the observation sequences consisted of a single word. But in continuous speech, the input consists of sequences of words, and we are not given the location of the word boundaries. Knowing where the word boundaries are massively simplifies the problem of pronunciation; in Chapter 5 since we were sure that the pronunciation [ni] came from one word, we only had 7 candidates to compare. But in actual speech we don't know where the word boundaries are. For example, try to decode the following sentence from Switchboard (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 mainly it's the lack of spaces indicating word boundaries that make the task difficult. The task of finding word boundaries in connected speech is called **segmentation** and we will solve it by using the Viterbi algorithm just as we did for

[2] I just heard something about moving recently.

Chinese word-segmentation in Chapter 5; Recall that the algorithm for Chinese word-segmentation relied on choosing the segmentation that resulted in the sequence of words with the highest frequency. For speech segmentation we use the more sophisticated N-gram language models introduced in Chapter 6. In the rest of this section we show how the Viterbi algorithm can be applied to the task of decoding and segmentation of a simple string of observations phones, using an n-gram language model. We will show how the algorithm is used to segment a very simple string of words. Here's the input and output we will work with:

Input	**Output**
[aa n iy dh ax]	*I need the*

Figure 7.5 shows word models for *I*, *need*, *the*, and also, just to make things difficult, the word *on*.

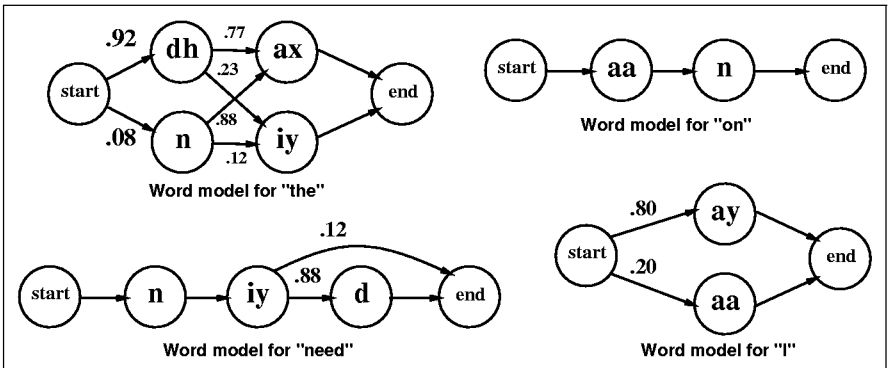

Figure 7.5 Pronunciation networks for the words *I*, *on*, *need*, and *the*. All networks (especially *the*) are significantly simplified.

Recall that the goal of the Viterbi algorithm is to find the best state sequence $q = (q_1 q_2 q_3 \ldots q_t)$ given the set of observed phones $o = (o_1 o_2 o_3 \ldots o_t)$. A graphic illustration of the output of the dynamic programming algorithm is shown in Figure 7.6. Along the y-axis are all the words in the lexicon; inside each word are its states. The x-axis is ordered by time, with one observed phone per time unit.[3] Each cell in the matrix will contain the probability of

[3] This x-axis component of the model is simplified in two major ways that we will show how to fix in the next section. First, the observations will not be phones but extracted spectral features, and second, each phone consists of not time unit observation but many observations (since phones can last for more than one phone). The y-axis is also simplified in this example, since as we will see most ASR system use multiple 'subphone' units for each phone.

the most-likely sequence ending at that state. We can find the most-likely state sequence for the entire observation string by looking at the cell in the right-most column that has the highest-probability, and tracing back the sequence that produced it.

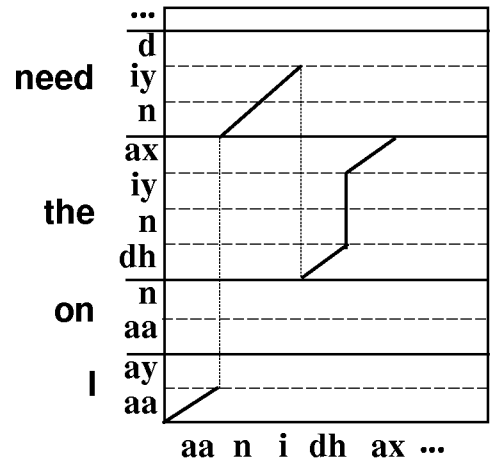

Figure 7.6 An illustration of the results of the Viterbi algorithm used to find the most-likely phone sequence (and hence estimate the most-likely word sequence).

More formally, we are searching for the best state sequence $q^* = (q_1 q_2 \dots q_T$ given an observation sequence $o = (o_1 o_2 \dots o_T)$ and a model (a weighted automaton or 'state graph') λ. Each cell *viterbi[i, t]*of the matrix contains the probability of the best path which accounts for the first t observations and ends in state i of the HMM. This is the most-probable path out of all possible sequences of states of length $t-1$:

$$viterbi[t,i] = \max_{q_1,q_2,\dots,q_{t-1}} P(q_1 q_2 \dots q_{t-1}, q_t = i, o_1, o_2 \dots o_t | \lambda) \qquad (7.8)$$

DYNAMIC PROGRAMMING INVARIANT

In order to compute *viterbi[t,i]*, the Viterbi algorithm assumes the **dynamic programming invariant**. This is the simplifying (but incorrect) assumption that if the ultimate best path for the entire observation sequence happens to go through a state q_i, that this best path must include the best path up to and including state q_i. This doesn't mean that the best path at any time t is the best path for the whole sequence. A path can look bad at the beginning but turn out to be the best path. As we will see later, the Viterbi assumption breaks down for certain kinds of grammars (including trigram

grammars) and so some recognizers have moved to another kind of decoder, the **stack** or **A*** decoder; more on that later. As we saw in our discussion of the minimum-edit-distance algorithm in Chapter 5, the reason for making the Viterbi assumption is that it allows us to break down the computation of the optimal path probability in a simple way; each of the best paths at time t is the best extension of each of the paths ending at time $t-1$. In other words, the recurrence relation for the best path at time t ending in state j, *viterbi[t,j]*, is the maximum of the possible extensions of every possible previous path from time $t-1$ to time t:

$$viterbi[t,j] = \max_i (viterbi[t-1,i]\, a_{ij})\, b_j(o_t) \tag{7.9}$$

The algorithm as we describe it in Figure 7.9 takes a sequence of observations, and a single probabilistic automaton, and returns the optimal path through the automaton. Since the algorithm requires a single automaton, we will need to combine the different probabilistic phone networks for *the*, *I*, *need*, and *a* into one automaton. In order to build this new automaton we will need to add arcs with probabilities between any two words: bigram probabilities. Figure 7.7 shows simple bigram probabilities computed from the combined Brown and Switchboard corpus.

I need	0.0016	need need	0.000047	# Need	0.000018
I the	0.00018	need the	0.012	# The	0.016
I on	0.000047	need on	0.000047	# On	0.00077
I I	0.039	need I	0.000016	# I	0.079
the need	0.00051	on need	0.000055		
the the	0.0099	on the	0.094		
the on	0.00022	on on	0.0031		
the I	0.00051	on I	0.00085		

Figure 7.7 Bigram probabilities for the words *the*, *on*, *need*, and *I* following each other, and starting a sentence (i.e. following #). Computed from the combined Brown and Switchboard corpora with add-0.5 smoothing.

Figure 7.8 shows the combined pronunciation networks for the 4 words together with a few of the new arcs with the bigram probabilities. For readability of the diagram, most of the arcs aren't shown; the reader should imagine that each probability in Figure 7.7 is inserted as an arc between every two words.

The algorithm is given in Figure 5.19 in Chapter 5, and is repeated here for convenience as Figure 7.9. We see in Figure 7.9 that the Viterbi

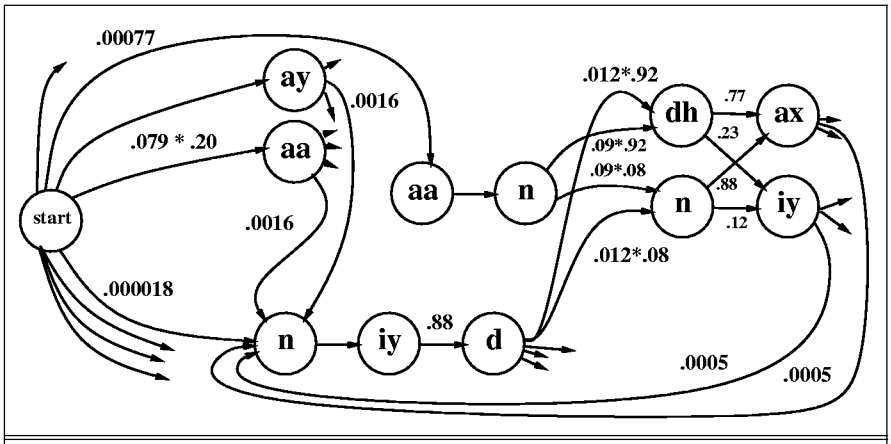

Figure 7.8 Single automaton made from the words *I*, *need*, *on*, and *the*. The arcs between words have probabilities computed from Figure 7.7. For lack of space the figure only shows a few of the between-word arcs.

algorithm sets up a probability matrix, with one column for each time index t and one row for each state in the state graph. The algorithm first creates $T+2$ columns; Figure 7.9 shows the first 6 columns. The first column is an initial pseudo-observation, the next corresponds to the first observation phone [aa], and so on. We begin in the first column by setting the probability of the *start* state to 1.0, and the other probabilities to 0; the reader should find this in Figure 7.10. Cells with probability 0 are simply left blank for readability. For each column of the matrix, i.e. for each time index t, each cell *viterbi[t,j]*, will contain the probability of the most likely path to end in that cell. We will calculate this probability recursively, by maximizing over the probability of coming from all possible preceding states. Then we move to the next state; for each of the i states *viterbi[0,i]* in column 0, we compute the probability of moving into each of the j states *viterbi[1,j]* in column 1, according to the recurrence relation in (7.9). In the column for the input *aa*, only two cells have non-zero entries, since $b_1(aa)$ is zero for every other state except the two states labeled *aa*. The value of *viterbi(1,aa)* of the word *I* is the product of the transition probability from # to *I* and the probability of *I* being pronounced with the vowel *aa*.

Notice that if we look at the column for the observation *n*, that the word *on* is currently the 'most-probable' word. But since there is no word or set of words in this lexicon which is pronounced *i dh ax*, the path starting with *on* is a dead end, i.e. this hypothesis can never be extended to cover the whole

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

num-states ← NUM-OF-STATES(*state-graph*)
Create a path probability matrix *viterbi[num-states+2,T+2]*
viterbi[0,0] ← 1.0
for each time step *t* **from** 0 **to** *T* **do**
  for each state *s* **from** 0 **to** *num-states* **do**
    for each transition s' from *s* specified by *state-graph*
      new-score ← *viterbi*[*s*, *t*] * $a[s,s']$ * $b_{s'}(o_t)$
      if ((*viterbi*[s',*t+1*] = 0) || (*new-score* > *viterbi*[s', *t+1*]))
        then
          viterbi[s', *t+1*] ← *new-score*
          back-pointer[s', *t+1*] ← *s*
Backtrace from highest probability state in the final column of *viterbi[]* and return path

Figure 7.9 Viterbi algorithm for finding optimal sequence of states in continuous speech recognition, simplified by using phones as inputs (duplicate of Figure 5.19). Given an observation sequence of phones and a weighted automaton (state graph), the algorithm returns the path through the automaton which has minimum probability and accepts the observation sequence. $a[s,s']$ 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.

utterance.

By the time we see the observation *iy*, there are two competing paths: *I need* and *I the*; *I need* is currently more likely. When we get to the observation *dh*, we could have arrived from either the *iy* of *need* or the *iy* of *the*. The probability of the *max* of these two paths, in this case the path through *I need*, will go into the cell for *dh*.

Finally, the probability for the best path will appear in the final *ax* column. In this example, only one cell is non-zero in this column; the *ax* state of the word *the* (a real example wouldn't be this simple; many other cells would be non-zero).

If the sentence had actually ended here, we would now need to backtrace to find the path that gave us this probability. We can't just pick the highest probability state for each state column. Why not? Because the most likely path early on is not necessarily the most likely path for the whole sentence. Recall that the most likely path after seeing *n* was the word *on*. But the most likely path for the whole sentence is *I need the*. Thus we had to

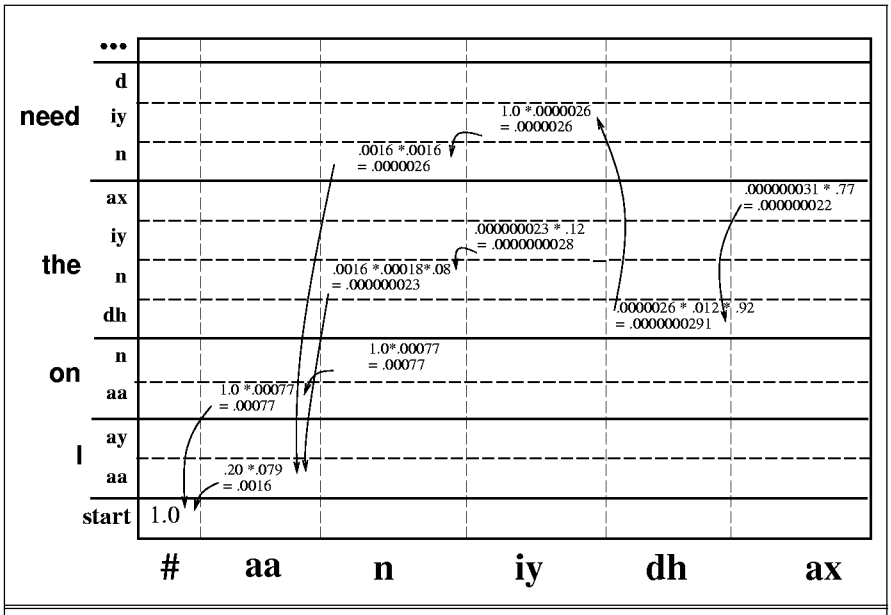

Figure 7.10 The entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. Backtracing from the successful last word (*the*), we can reconstruct the word sequence *I need the*.

rely in Figure 7.10 on the 'Hansel and Gretel' method (or the 'Jason and the Minotaur' method if you like your metaphors more classical): whenever we moved into a cell, we kept pointers back to the cell we came from. The reader should convince themselves that the Viterbi algorithm has simultaneously solved the segmentation and decoding problems.

The presentation of the Viterbi algorithm in this section has been simplified; actual implementations of Viterbi decoding are more complex in three key ways that we have mentioned already. First, in an actual HMM for speech recognition, the input would not be phones. Instead, the input is a **feature vector** of spectral and acoustic features. Thus the **observation likelihood probabilities** $b_i(t)$ of an observation o_t given a state i will not simply take on the values 0 or 1, but will be more fine-grained probability estimates, computed via mixtures of Gaussian probability estimators or neural nets. The next section will show how these probabilities are computed.

Second, the HMM states in most speech recognition systems are not simple phones but rather **subphones**. In these systems each phone is divided into 3 states: the beginning, middle and final portions of the phone. Dividing

up a phone in this way captures the intuition that the significant changes in the acoustic input happen at a finer granularity than the phone; for example the closure and release of a stop consonant. Furthermore, many systems use a separate instance of each of these subphones for each **triphone** context (Schwartz *et al.*, 1985; Deng *et al.*, 1990). Thus instead of around 60 phone units, there could be as many as 60^3 context-dependent triphones. In practice, many possible sequences of phones never occur or are very rare, so systems create a much smaller number of triphones models by **clustering** the possible triphones (Young and Woodland, 1994). Figure 7.11 shows an example of the complete phone model for the triphone b(ax,aw).

TRIPHONE

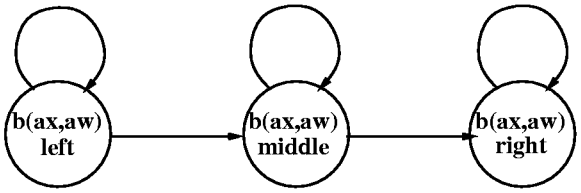

Figure 7.11 An example of the context-dependent triphone b(ax,aw) (the phone [b] preceded by a [ax] and followed by a [aw], as in the beginning of *about*, showing its left, middle, and right subphones.

Finally, in practice in large-vocabulary recognition it is too expensive to 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. This is usually implemented via **beam search**: for each state column (time step), the algorithm maintains a short list of high-probability words whose path probabilities are within some percentage (**beam width**) of the most probable word path. Only transitions from these words are extended when moving to the next time step. Since the words are ranked by the probability of the path so far, which words are within the beam (active) will change from time step to time step. Making this beam search approximation allows a significant speed-up at the cost of a degradation to the decoding performance. This beam search strategy was first implemented by Lowerre (1968). 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.

BEAM SEARCH

BEAM WIDTH

7.4 ADVANCED METHODS FOR DECODING

There are two main limitations of the Viterbi decoder. First, the Viterbi decoder does not actually compute the sequence of words which is most probable given the input acoustics. Instead, it computes an approximation to this: the sequence of *states* (i.e. *phones* or *subphones*) which is most probable given the input. This difference may not always be important; the most probable sequence of phones may very well correspond exactly to the most probable sequence of words. But sometimes the most probable sequence of phones does not correspond to the most probable word sequence. For example consider a speech recognition system whose lexicon has multiple pronunciations for each word. Suppose the correct word sequence includes a word with very many pronunciations. Since the probabilities leaving the start arc of each word must sum to 1.0, each of these pronunciation-paths through this multiple-pronunciation HMM word model will have a smaller probability than the path through a word with only a single pronunciation path. Thus because the Viterbi decoder can only follow one of these pronunciation paths, it may ignore this word in favor of an incorrect word with only one pronunciation path.

A second problem with the Viterbi decoder is that it cannot be used with all possible language models. In fact, the Viterbi algorithm as we have defined it cannot take complete advantage of any language model more complex than a bigram grammar. This is because of the fact mentioned early that a trigram grammar, for example, violates the **dynamic programming invariant** that makes dynamic programming algorithms possible. Recall that this invariant is the simplifying (but incorrect) assumption that if the ultimate best path for the entire observation sequence happens to go through a state q_i, that this best path must include the best path up to and including state q_i. Since a trigram grammar allows the probability of a word to be based on the two previous words, it is possible that the best trigram-probability path for the sentence may go through a word but not include the best path to that word. Such a situation could occur if a particular word w_x has a high trigram probability given w_y, w_z, but that conversely the best path to w_y didn't include w_z (i.e. $P(w_y|w_q, w_z)$ was low for all q).

There are two classes of solutions to these problems with Viterbi decoding. One class involves modifying the Viterbi decoder to return multiple potential utterances and then using other high-level language model or pronunciation-modeling algorithms to re-rank these multiple outputs. In

general this kind of **multiple-pass decoding** allows a computationally efficient, but perhaps unsophisticated, language model like a bigram to perform a rough first decoding pass, allowing more sophisticated but slower decoding algorithms to run on a reduced search space.

For example, Schwartz and Chow (1990) give a Viterbi-like algorithm which returns the N**-best** sentences (word sequences) for a given speech input. Suppose for example a bigram grammar is used with this N-best-Viterbi to return the 10,000 most highly-probable sentences, each with their likelihood score. A trigram-grammar can then be used to assign a new language-model prior probability to each of these sentences. These priors can be combined with the acoustic likelihood of each sentence to generate a posterior probability for each sentence. Sentences can then be **rescored** using this more sophisticated probability.Figure 7.12 shows an intuition for this algorithm.

N-BEST

RESCORED

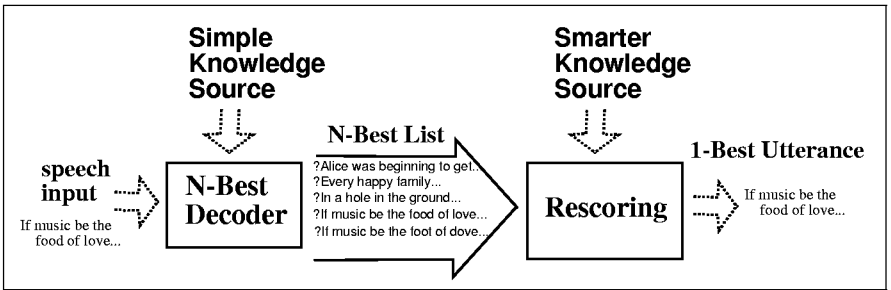

Figure 7.12 The use of N-best decoding as part of a two-stage decoding model. Efficient but unsophisticated knowledge sources are used to return the N-best utterances. This significantly reduces the search space for the second pass models, which are thus free to be very sophisticated but slow.

An augmentation of N-best, still part of this first class of extensions to Viterbi, is to return, not a list of sentences, but a **word lattice**. A word lattice is a directed graph of words and links between them which can compactly encode a large number of possible sentences. Each word in the lattice is augmented with its observation likelihood, so that any particular path through the lattice can then be combined with the prior probability derived from a more sophisticated language model. For example Murveit *et al.* (1993) describe an algorithm used in the SRI recognizer Decipher which uses a bigram grammar in a rough first pass, producing a word lattice which is then refined by a more sophisticated language model.

WORD LATTICE

The second solution to the problems with Viterbi decoding is to employ

STACK DECODER
A*
A* SEARCH

a completely different decoding algorithm. The most common alternative algorithm is the **stack decoder**, also called the **A*** decoder (Jelinek, 1969; Jelinek *et al.*, 1975). We will describe the algorithm in terms of the **A* search** used in the artificial intelligence literature, although the development of stack decoding actually came from the communications theory literature and the link with AI best-first search was noticed only later (Jelinek, 1976).

A* Decoding

To see how the A* decoding method works, we need to revisit the Viterbi algorithm. Recall that the Viterbi algorithm computed an approximation of the forward algorithm. Viterbi computes the observation likelihood of the single best (MAX) path through the HMM, while the forward algorithm computes the observation likelihood of the total (SUM) of all the paths through the HMM. But we accepted this approximation because Viterbi computed this likelihood *and* searched for the optimal path simultaneously. The A* decoding algorithm, on the other hand, will rely on the complete forward algorithm rather than an approximation. This will ensure that we compute the correct observation likelihood. Furthermore, the A* decoding algorithm allows us to use any arbitrary language model.

The A* decoding algorithm is a kind of best-first search of the lattice or tree which implicitly defines the sequence of allowable words in a language. Consider the tree in Figure 7.13, rooted in the START node on the left. Each leaf of this tree defines one sentence of the language; the one formed by concatenating all the words along the path from START to the leaf. We don't represent this tree explicitly, but the stack decoding algorithm uses the tree implicitly as a way to structure the decoding search.

The algorithm performs a search from the root of the tree toward the leaves, looking for the highest probability path, and hence the highest probability sentence. As we proceed from root toward the leaves, each branch leaving a given word node represent a word which may follow the current word. Each of these branches has a probability, which expresses the conditional probability of this next word given the part of the sentence we've seen so far. In addition, we will use the forward algorithm to assign each word a likelihood of producing some part of the observed acoustic data. The A* decoder must thus find the path (word sequence) from the root to a leaf which has the highest probability, where a path probability is defined as the product of its language model probability (prior) and its acoustic match to the data (likelihood). It does this by keeping a **priority queue** of partial paths

PRIORITY QUEUE

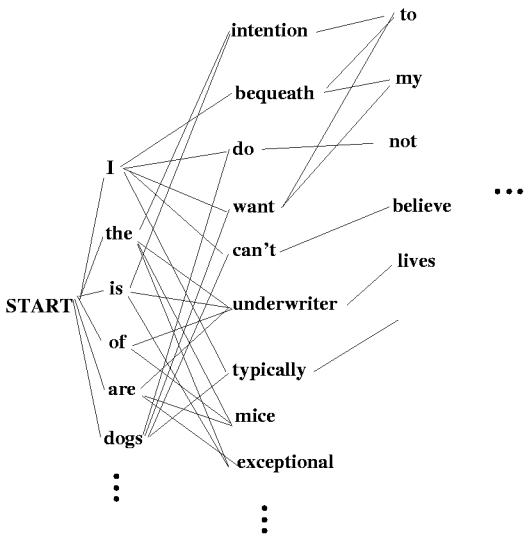

Figure 7.13 A visual representation of the implicit lattice of allowable word sequences which defines a language. The set of sentences of a language is far too large to represent explicitly, but the lattice gives a metaphor for exploring substrings of these sentences.

(i.e. prefixes of sentences, each annotated with a score). In a priority queue each element has a score, and the *pop* operation returns the element with the highest score. The A* decoding algorithm iteratively chooses the best prefix-so-far, computes all the possible next words for that prefix, and adds these extended sentences to the queue. The Figure 7.14 shows the complete algorithm.

Let's consider a stylized example of a A* decoder working on a waveform for which the correct transcription is *If music be the food of love.* Figure 7.15 shows the search space after the decoder has examined paths of length one from the root. A **fast match** is used to select the likely next words. A fast match is one of a class of heuristics designed to efficiently winnow down the number of possible following words, often by computing some approximation to the forward probability (see below for further discussion of fast matching). FAST MATCH

At this point in our example, we've done the fast match, selected a subset of the possible next words, and assigned each of them a score. The word *Alice* has the highest score. We haven't yet said exactly how the scoring works, although it will involve as a component the probability of the hypoth-

function STACK-DECODING() **returns** *min-distance*

Initialize the priority queue with a null sentence.
Pop the best (highest score) sentence s off the queue.
If (s is marked end-of-sentence (EOS)) output s and terminate.
Get list of candidate next words by doing fast matches.
For each candidate next word w:
    Create a new candidate sentence $s + w$.
    Use forward algorithm to compute acoustic likelihood L of $s + w$
    Compute language model probability P of extended sentence $s + w$
    Compute 'score' for $s + w$ (a function of L, P, and ???)
    if (end-of-sentence) set EOS flag for $s + w$.
    Insert $s + w$ into the queue together with its score and EOS flag

Figure 7.14 The A* decoding algorithm (modified from Paul (1991) and Jelinek (1997)). The evaluation function that is used to compute the score for a sentence is not completely defined here; possibly evaluation functions are discussed below.

esized sentence given the acoustic input $P(W|A)$, which itself is composed of the language model probability $P(W)$ and the acoustic likelihood $P(A|W)$.

Figure 7.16 show the next stage in the search. We have expanded the *Alice* node. This means that the *Alice* node is no longer on the queue, but its children are. Note that now the node labeled *if* actually has a higher score than any of the children of *Alice*.

Figure 7.17 shows the state of the search after expanding the *if* node, removing it, and adding *if music*, *if muscle*, and *if messy* on to the queue.

We've implied that the scoring criterion for a hypothesis is related to its probability. Indeed it might seem that the score for a string of words w_1^i given an acoustic string y_1^j should be the product of the prior and the likelihood:

$$P(y_1^j|w_1^i)P(w_1^i)$$

Alas, the score cannot be this probability because the probability will be much smaller for a longer path than a shorter one. This is due to a simple fact about probabilities and substrings; any prefix of a string must have a higher probability than the string itself (e.g. P(START the...) will be greater than P(START the book)). Thus if we used probability as the score, the A* decoding algorithm would get stuck on the single-word hypotheses.

Instead, we use what is called the A* evaluation function (Nilsson, 1980; Pearl, 1984) called $f^*(p)$, given a partial path p:

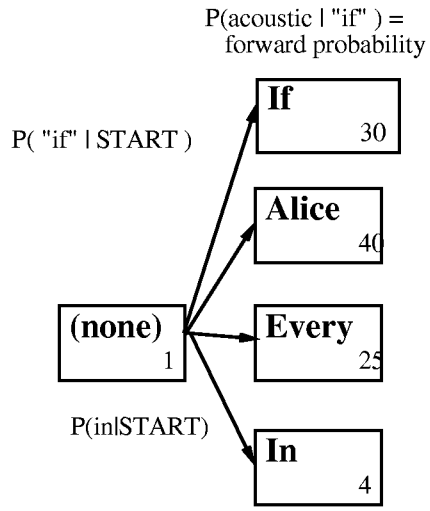

Figure 7.15 The beginning of the search for the sentence *If music be the food of love.* At this early stage *Alice* is the most likely hypothesis (it has a higher score than the other hypotheses).

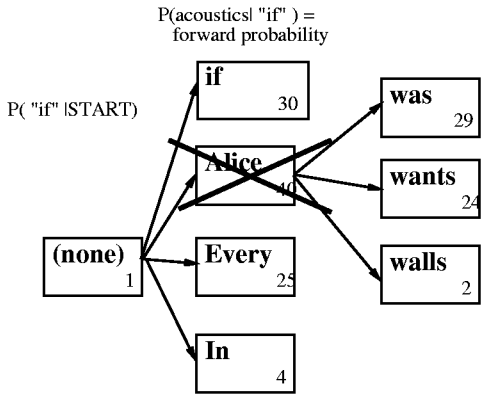

Figure 7.16 The next step of the search for the sentence *If music be the food of love.* We've now expanded the *Alice* node, and added three extensions which have a relatively high score (*was*, *wants*, and *walls*). Note that now the node with the highest score is *START if*, which is not along the *START Alice* path at all!

$$f^*(p) = g(p) + h^*(p)$$

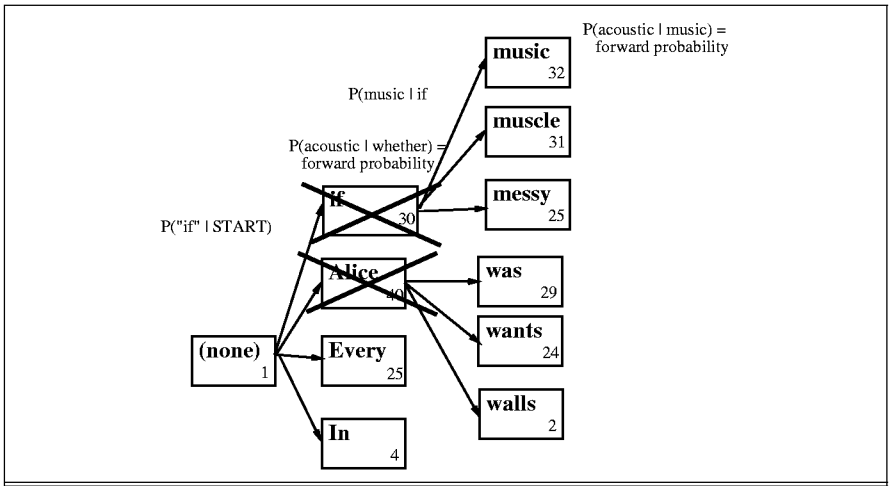

Figure 7.17 We've now expanded the *if* node. The hypothesis *START if music* currently has the highest score.

$f^*(p)$ is the *estimated* score of the best complete path (complete sentence) which starts with the partial path p. In other words, it is an estimate of how well this path would do if we let it continue through the sentence. The A* algorithm builds this estimate from two components:

- $g(p)$ is the score from the beginning of utterance to the end of the partial path p. This g function can be nicely estimated by the probability of p given the acoustics so far (i.e. as $P(A|W)P(W)$ for the word string W constituting p).
- $h^*(p)$ is an estimate of the best scoring extension of the partial path to the end of the utterance.

Coming up with a good estimate of h^* is an unsolved and interesting problem. One approach is to choose as h^* an estimate which correlates with the number of words remaining in the sentence (Paul, 1991); see Jelinek (1997) for further discussion.

We mentioned above that both the A* and various other two-stage decoding algorithms require the use of a **fast match** for quickly finding which words in the lexicon are likely candidates for matching some portion of the acoustic input. Many fast match algorithms are based on the use of a **tree-structured lexicon**, which stores the pronunciations of all the words in such a way that the computation of the forward probability can be shared for words which start with the same sequence of phones. The tree-structured

TREE-STRUCTURED LEXICON

lexicon was first suggested by Klovstad and Mondshein (1975); fast match algorithms which make use of it include Gupta *et al.* (1988), Bahl *et al.* (1992) in the context of A* decoding, and Ney *et al.* (1992) and Nguyen and Schwartz (1999) in the context of Viterbi decoding. Figure 7.18 shows an example of a tree-structured lexicon from the Sphinx-II recognizer (Ravishankar, 1996). Each tree root represents the first phone of all words beginning with that context dependent phone (phone context may or may not be preserved across word boundaries), and each leaf is associated with a word.

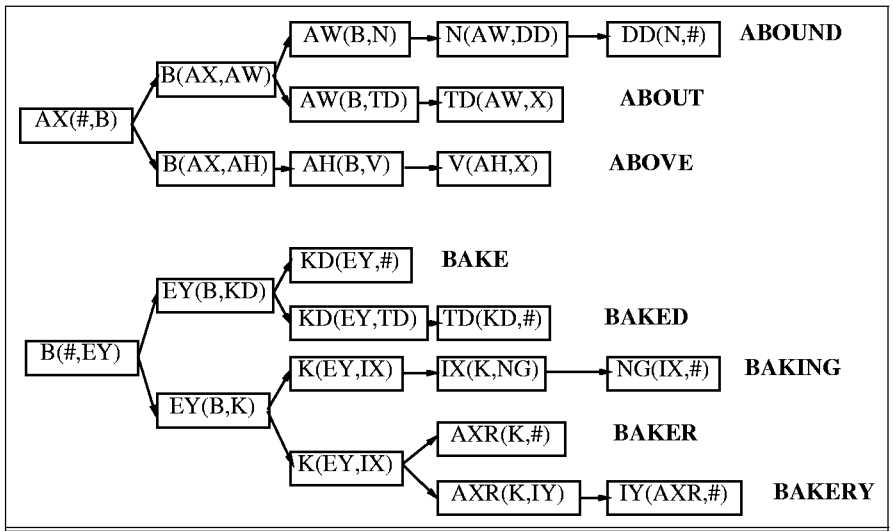

Figure 7.18 A tree-structured lexicon from the Sphinx-II recognizer (after Ravishankar (1996)). Each node corresponds to a particular triphone in a slightly modified version of the ARPAbet; thus EY(B,KD) means the phone EY preceded by a B and followed by the closure of a K.

There are many other kinds of multiple-stage search, such as the **forward-backward** search algorithm (not to be confused with the **forward-backward** algorithm) (Austin *et al.*, 1991) which performs a simple forward search followed by a detailed backward (i.e. time-reversed) search.

FORWARD-BACKWARD

7.5 ACOUSTIC PROCESSING OF SPEECH

FEATURE EXTRACTION
SIGNAL ANALYSIS
LPC
PLP
SPECTRAL FEATURES

This section presents a very brief overview of the kind of acoustic processing commonly called **feature extraction** or **signal analysis** in the speech recognition literature. The term **features** refers to the vector of numbers which represent one time-slice of a speech signal. A number of kinds of features are commonly used, such as **LPC** features and **PLP** features. All of these are **spectral features**, which means that they represent the waveform in terms of the distribution of different **frequencies** which make up the waveform; such a distribution of frequencies is called a **spectrum**. We will begin with a brief introduction to the acoustic waveform and how it is digitized, summarize the idea of frequency analysis and spectra, and then sketch out different kinds of extracted features. This will be an extremely brief overview; the interested reader should refer to other books on the linguistics aspects of acoustic phonetics (Johnson, 1997; Ladefoged, 1996) or on the engineering aspects of digital signal processing of speech (Rabiner and Juang, 1993).

Sound Waves

The input to a speech recognizer, like the input to the human ear, is a complex series of changes in air pressure. These changes in air pressure obviously originate with the speaker, and are caused by the specific way that air passes through the glottis and out the oral or nasal cavities. We represent sound waves by plotting the change in air pressure over time. One metaphor which sometimes helps in understanding these graphs is to imagine a vertical plate which is blocking the air pressure waves (perhaps in a microphone in front of a speaker's mouth, or the eardrum in a hearer's ear). The graph measures the amount of **compression** or **rarefaction** (uncompression) of the air molecules at this plate. Figure 7.19 shows the waveform taken from the Switchboard corpus of telephone speech of someone saying "she just had a baby".

FREQUENCY
AMPLITUDE
CYCLES PER SECOND
HERTZ

Two important characteristics of a wave are its **frequency** and **amplitude**. The frequency is the number of times a second that a wave repeats itself, or **cycles**. Note in Figure 7.19 that there are 28 repetitions of the wave in the .11 seconds we have captured. Thus the frequency of this segment of the wave is 28/.11 or 255 **cycles per second**. Cycles per second are usually called **Hertz** (shortened to **Hz**), so the frequency in Figure 7.19 would be described as 255 Hz.

AMPLITUDE

The vertical axis in Figure 7.19 measures the amount of air pressure variation. A high value on the vertical axis (a high **amplitude**) indicates

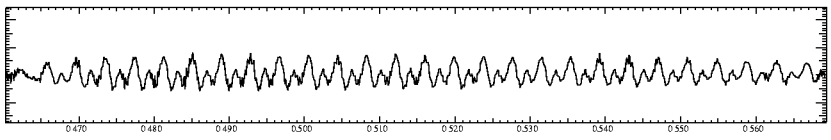

Figure 7.19 A waveform of the vowel [iy] from the utterance shown in Figure 7.20. The y-axis shows the changes in air pressure above and below normal atmospheric pressure. The x-axis shows time. Notice that the wave repeats regularly.

that there is more air pressure at that point in time, a zero value means there is normal (atmospheric) air pressure, while a negative value means there is lower than normal air pressure (rarefaction).

Two important perceptual properties are related to frequency and amplitude. The **pitch** of a sound is the perceptual correlate of frequency; in general if a sound has a higher-frequency we perceive it as having a higher pitch, although the relationship is not linear, since human hearing has different acuities for different frequencies. Similarly, the **loudness** of a sound is the perceptual correlate of the **power**, which is related to the square of the amplitude. So sounds with higher amplitudes are perceived as louder, but again the relationship is not linear.

PITCH

How to Interpret a Waveform

Since humans (and to some extent machines) can transcribe and understand speech just given the sound wave, the waveform must contain enough information to make the task possible. In most cases this information is hard to unlock just by looking at the waveform, but such visual inspection is still sufficient to learn some things. For example, the difference between vowels and most consonants is relatively clear on a waveform. Recall that vowels are voiced, tend to be long, and are relatively loud. Length in time manifests itself directly as length in space on a waveform plot. Loudness manifests itself as high amplitude. How do we recognize voicing? Recall that voicing is caused by regular openings and closing of the vocal folds. When the vocal folds are vibrating, we can see regular peaks in amplitude of the kind we saw in Figure 7.19. During a stop consonant, for example the closure of a [p], [t], or [k], we should expect no peaks at all; in fact we expect silence.

Notice in Figure 7.20 the places where there are regular amplitude peaks indicating voicing; from second .46 to .58 (the vowel [iy]), from sec-

ond .65 to .74 (the vowel [ax]) and so on. The places where there is no amplitude indicate the silence of a stop closure; for example from second 1.06 to second 1.08 (the closure for the first [b], or from second 1.26 to 1.28 (the closure for the second [b]).

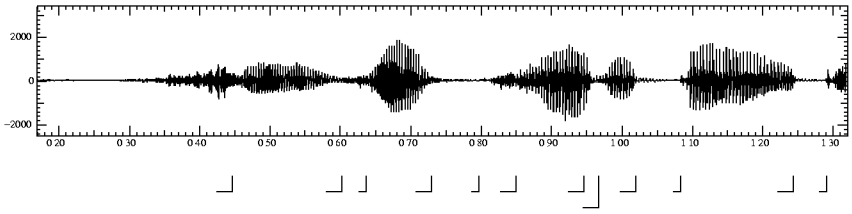

Figure 7.20 A waveform of the sentence "She just had a baby" from the Switchboard corpus (conversation 4325). The speaker is female, was 20 years old in 1991 which is approximately when the recording was made, and speaks the South Midlands dialect of American English. The phone labels show where each phone ends.

Fricatives like [sh] can also be recognized in a waveform; they produce an intense irregular pattern; the [sh] from second .33 to .46 is a good example of a fricative.

Spectra

While some broad phonetic features (presence of voicing, stop closures, fricatives) can be interpreted from a waveform, more detailed classification (which vowel? which fricative?) requires a different representation of the input in terms of **spectral** features. Spectral features are based on the insight of Fourier that every complex wave can be represented as a sum of many simple waves of different frequencies. A musical analogy for this is the chord; just as a chord is composed of multiple notes, any waveform is composed of the waves corresponding to its individual "notes".

SPECTRAL

Consider Figure 7.21, which shows part of the waveform for the vowel [æ] of the word *had* at second 0.9 of the sentence. Note that there is a complex wave which repeats about nine times in the figure; but there is also a smaller repeated wave which repeats four times for every larger pattern (notice the four small peaks inside each repeated wave). The complex wave has a frequency of about 250 Hz (we can figure this out since it repeats roughly 9 times in .036 seconds, and 9 cycles/.036 seconds = 250 Hz). The smaller

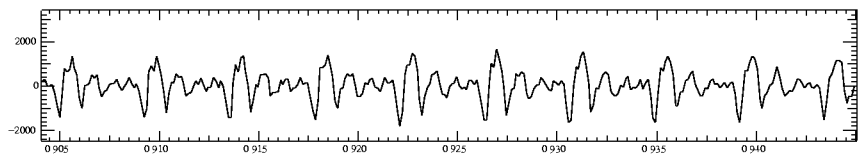

Figure 7.21 The waveform of part of the vowel [æ] from the word *had* cut out from the waveform shown in Figure 7.20.

wave then should have a frequency of roughly 4 times the frequency of the larger wave, or roughly 1000 Hz. Then if you look carefully you can see two little waves on the peak of many of the 1000 Hz waves. The frequency of this tiniest wave must be roughly twice that of the 1000 Hz wave, hence 2000 Hz.

SPECTRUM

FOURIER TRANSFORM

A **spectrum** is a representation of these different frequency components of a wave. It can be computed by a **Fourier transform**, a mathematical procedure which separates out each of the frequency components of a wave. Rather than using the Fourier transform spectrum directly, most speech applications use a smoothed version of the spectrum called the **LPC** spectrum (Atal and Hanauer, 1971; Itakura, 1975).

LPC

Figure 7.22 shows an LPC spectrum for the waveform in Figure 7.21. LPC (**Linear Predictive Coding**) is a way of coding the spectrum which makes it easier to see where the **spectral peaks** are.

SPECTRAL PEAKS

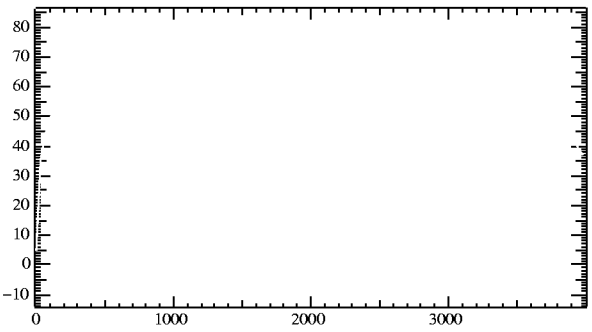

Figure 7.22 An LPC spectrum for the vowel [æ] waveform of *She just had a baby* at the point in time shown in Figure 7.21. LPC makes it easy to see **formants**.

The x-axis of a spectrum shows frequency while the y-axis shows some measure of the magnitude of each frequency component (in decibels (dB), a logarithmic measure of amplitude). Thus Figure 7.22 shows that there are important frequency components at 930 Hz, 1860 Hz, and 3020 Hz, along with many other lower-magnitude frequency components. These important components at roughly 1000 Hz and 2000 Hz are just what we predicted by looking at the wave in Figure 7.21!

Why is a spectrum useful? It turns out that these spectral peaks that are easily visible in a spectrum are very characteristic of different sounds; phones have characteristic spectral 'signatures'. For example different chemical elements give off different wavelengths of light when they burn, allowing us to detect elements in stars light-years away by looking at the spectrum of the light. Similarly, by looking at the spectrum of a waveform, we can detect the characteristic signature of the different phones that are present. This use of spectral information is essential to both human and machine speech recognition. In human audition, the function of the **cochlea** or **inner ear** is to compute a spectrum of the incoming waveform. Similarly, the features used as input to the HMMs in speech recognition are all representations of spectra, usually variants of LPC spectra, as we will see.

COCHLEA
INNER EAR

While a spectrum shows the frequency components of a wave at one point in time, a **spectrogram** is a way of envisioning how the different frequencies which make up a waveform change over time. The x-axis shows time, as it did for the waveform, but the y-axis now shows frequencies in Hz. The darkness of a point on a spectrogram corresponding to the amplitude of the frequency component. For example, look in Figure 7.23 around second 0.9, and notice the dark bar at around 1000 Hz. This means that the [iy] of the word *she* has an important component around 1000 Hz (1000 Hz is just between the notes B and C). The dark horizontal bars on a spectrogram, representing spectral peaks, usually of vowels, are called **formants**.

SPECTROGRAM
FORMANTS

What specific clues can spectral representations give for phone identification? First, different vowels have their formants at characteristic places. We've seen that [æ] in the sample waveform had formants at 930 Hz, 1860 Hz, and 3020 Hz. Consider the vowel [iy], at the beginning of the utterance in Figure 7.20. The spectrum for this vowel is shown in Figure 7.24. The first formant of [iy] is 540 Hz; much lower than the first formant for [æ], while the second formant (2581 Hz) is much higher than the second formant for [æ]. If you look carefully you can see these formants as dark bars in Figure 7.23 just around 0.5 seconds.

The location of the first two formants (called F1 and F2) plays a large

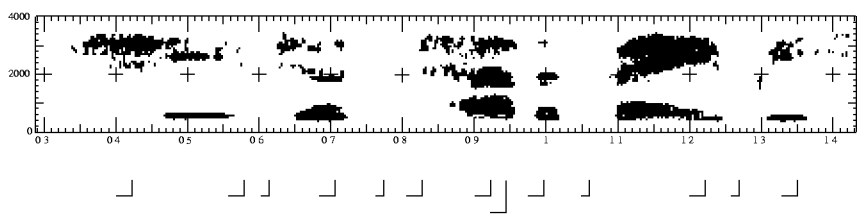

Figure 7.23 A spectrogram of the sentence "She just had a baby" whose waveform was shown in Figure 7.20. One way to think of a spectrogram is as a collection of spectra (time-slices) like Figure 7.22 placed end to end.

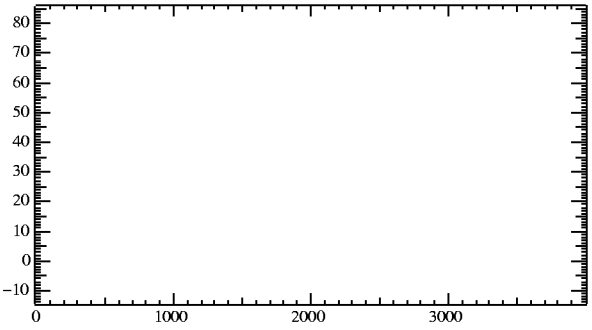

Figure 7.24 A smoothed (LPC) spectrum for the vowel [iy] at the start of *She just had a baby*. Note that the first formant (540 Hz) is much lower than the first formant for [æ] shown in Figure 7.22, while the second formant (2581 Hz) is much higher than the second formant for [æ].

role in determining vowel identity, although the formants still differ from speaker to speaker. Formants also can be used to identify the nasal phones [n], [m], and [ŋ], the lateral phone [l], and [r]. Why do different vowels have different spectral signatures? The formants are caused by the resonant cavities of the mouth. The oral cavity can be thought of as a filter which selectively passes through some of the harmonics of the vocal cord vibrations. Moving the tongue creates spaces of different size inside the mouth which selectively amplify waves of the appropriate wavelength, hence amplifying different frequency bands.

Feature Extraction

Our survey of the features of waveforms and spectra was necessarily brief, but the reader should have the basic idea of the importance of spectral features and their relation to the original waveform. Let's now summarize the process of extraction of spectral features, beginning with the sound wave itself and ending with a **feature vector**.[4] An input soundwave is first **digitized**. 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. Common sampling rates are 8,000 Hz and 16,000 Hz. 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 2 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.

SAMPLING
QUANTIZATION
SAMPLING RATE
NYQUIST FREQUENCY

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 a real-valued number as a integer 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.

QUANTIZATION

Once a waveform has been digitized, it is converted to some set of spectral features. An LPC spectrum is represented by a vector of features; each formant is represented by two features, plus two additional features to represent spectral tilt. Thus 5 formants can be represented by 12 (5x2+2) features. It is possible to use LPC features directly as the observation sym-

[4] The reader might want to bear in mind Picone's (1993) reminder that the use of the word **extraction** should not be thought of as encouraging the metaphor of features as something 'in the signal' waiting to be extracted.

bols of an HMM. However, further processing is often done to the features. One popular feature set is **cepstral coefficients**, which are computed from the LPC coefficients by taking the Fourier transform of the spectrum. Another feature set, **PLP** (**Perceptual Linear Predictive** analysis (Hermansky, 1990)), takes the LPC features and modifies them in ways consistent with human hearing. For example, the spectral resolution of human hearing is worse at high frequencies, and the perceived loudness of a sound is related to the cube rate of its intensity. So PLP applies various filters to the LPC spectrum and takes the cube root of the features.

CEPSTRAL COEFFICIENTS

PLP

7.6 Computing Acoustic Probabilities

The last section showed how the speech input can be passed through signal processing transformations and turned into a series of vectors of features, each vector representing one time-slice of the input signal. How are these feature vectors turned into probabilities?

One way to compute probabilities on feature vectors is to first **cluster** them into discrete symbols that we can count; we can then compute the probability of a given cluster just by counting the number of times it occurs in some training set. This method is usually called **vector quantization**. Vector quantization was quite common in early speech recognition algorithms but has mainly been replaced by a more direct but compute-intensive approach: computing observation probabilities on a real-valued ('continuous') input vector. This method thus computes a **probability density function** or **pdf** over a continuous space.

CLUSTER

VECTOR QUANTIZATION

PROBABILITY DENSITY FUNCTION

There are two popular versions of the continuous approach. The most widespread of the two is the use of **Gaussian** pdfs, in the simplest version of which each state has a single Gaussian function which maps the observation vector o_t to a probability. An alternative approach is the use of **neural networks** or **multi-layer perceptrons** which can also be trained to assign a probability to a real-valued feature vector. HMMs with Gaussian observation-probability-estimators are trained by a simple extension to the forward-backward algorithm (discussed in Appendix D). HMMs with neural-net observation-probability-estimators are trained by a completely different algorithm known as **error back-propagation**.

GAUSSIAN

NEURAL NETWORKS MULTI-LAYER PERCEPTRONS

ERROR BACK-PROPAGATION

In the simplest use of Gaussians, we assume that the possible values of the observation feature vector o_t are normally distributed, and so we represent the observation probability function $b_j(o_t)$ as a Gaussian curve with

mean vector μ_j and covariance matrix $\sum_j$; (prime denotes vector transpose). We present the equation here for completeness, although we will not cover the details of the mathematics:

$$b_j(o_t) = \frac{1}{\sqrt{(2\pi)|\sum j|}} e^{[(o_t-\mu_j)'\Sigma_j^{-1}(o_t-\mu_j)]} \tag{7.10}$$

Usually we make the simplifying assumption that the covariance matrix Σ_j is diagonal, i.e. that it contains the simple variance of cepstral feature 1, the simple variance of cepstral feature 2, and so on, without worrying about the effect of cepstral feature 1 on the variance of cepstral feature 2. This means that in practice we are keeping only a single separate mean and variance for each feature in the feature vector.

Most recognizers do something even more complicated; they keep multiple Gaussians for each state, so that the probability of each feature of the observation vector is computed by adding together a variety of Gaussian curves. This technique is called **Gaussian mixtures**. In addition, many ASR systems share Gaussians between states in a technique known as **parameter tying** (or **tied mixtures**) (Huang and Jack, 1989). For example acoustically similar phone states might share (i.e. use the same) Gaussians for some features.

GAUSSIAN MIXTURES

TIED MIXTURES

How are the mean and covariance of the Gaussians estimated? It is helpful again to consider the simpler case of a non-hidden Markov Model, with only one state i. The vector of feature means μ and the vector of covariances Σ could then be estimated by averaging:

$$\hat{\mu}_i = \frac{1}{T}\sum_{t=1}^{T} o_t \tag{7.11}$$

$$\hat{\Sigma}_i = \frac{1}{T}\sum_{t=1}^{T}[(o_t-\mu_j)'(o_t-\mu_j)] \tag{7.12}$$

But since there are multiple hidden states, we don't know which observation vector o_t was produced by which state. Appendix D will show how the forward-backward algorithm can be modified to 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.

An alternative way to model continuous-valued features is the use of a **neural network**, **multilayer perceptron** (**MLP**) or **Artificial Neural Networks** (**ANN**s). Neural networks are far too complex for us to introduce in a page or two here; thus we will just give the intuition of how they are used

NEURAL NETWORK

MULTILAYER PERCEPTRON

MLP

in probability estimation as an alternative to Gaussian estimators. The interested reader should consult basic neural network textbooks (Anderson, 1995; Hertz *et al.*, 1991) as well as references specifically focusing on neural-network speech recognition (Bourlard and Morgan, 1994).

A neural network is a set of small computation units connected by weighted links. The network is given a vector of input values and computes a vector of output values. The computation proceeds by each computational unit computing some non-linear function of its input units and passing the resulting value on to its output units.

The use of neural networks we will describe here is often called a **hybrid** HMM-MLP approach, since it uses some elements of the HMM (such as the state-graph representation of the pronunciation of a word) but the observation-probability computation is done by an MLP instead of a mixture of Gaussians. The input to these MLPs is a representation of the signal at a time t and some surrounding window; for example this might mean a vector of spectral features for a time t and 8 additional vectors for times $t + 10ms$, $t + 20ms$, $t + 30ms$, $t + 40ms$, $t - 10ms$, etc. Thus the input to the network is a set of nine vectors, each vector having the complete set of real-valued spectral features for one time slice. The network has one output unit for each phone; by constraining the values of all the output units to sum to 1, the net can be used to compute the probability of a state j given an observation vector o_t, or $P(j|o_t)$. Figure 7.25 shows a sample of such a net.

HYBRID

This MLP computes the probability of the HMM state j given an observation o_t, or $P(q_j|o_t)$. But the observation likelihood we need for the HMM, $b_j(o_t)$, is $P(o_t|q_j)$. The Bayes rule can help us see how to compute one from the other. The net is computing:

$$p(q_j|o_t) = \frac{P(o_t|q_j)p(q_j)}{p(o_t)} \tag{7.13}$$

We can rearrange the terms as follows:

$$\frac{p(o_t|q_j)}{p(o_t)} = \frac{P(q_j|o_t)}{p(q_j)} \tag{7.14}$$

The two terms on the right-hand side of (7.14) can be directly computed from the MLP; the numerator is the output of the MLP, and the denominator is the total probability of a given state, summing over all observations (i.e. the sum over all t of $\sigma_j(t)$). Thus although we cannot directly compute $P(o_t|q_j)$, we *can* use (7.14) to compute $\frac{p(o_t|q_j)}{p(o_t)}$, which is known as a **scaled likelihood** (the likelihood divided by the probability of the observation) . In fact, the scaled likelihood is just as good as the regular likelihood, since

SCALED LIKELIHOOD

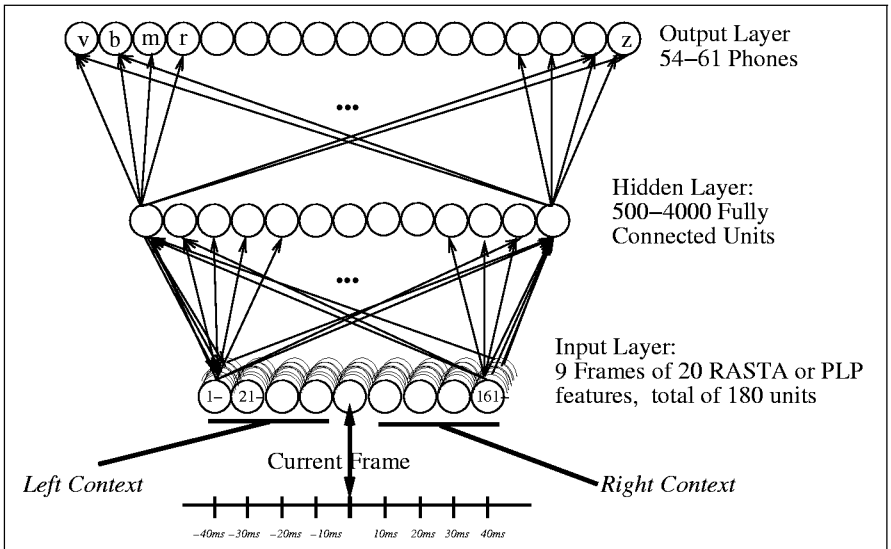

Figure 7.25 A neural net used to estimate phone state probabilities. Such a net can be used in an HMM model as an alternative to the Gaussian models. This particular net is from the MLP systems described in Bourlard and Morgan (1994); it is given a vector of features for a frame and for the four frames on either side, and estimates $p(q_j|o_t)$. This probability is then converted to an estimate of the observation likelihood $b = p(o_t|q_j)$ using the Bayes rule. These nets are trained using the error-back-propagation algorithm as part of the same **embedded training** algorithm that is used for Gaussians.

the probability of the observation $p(o_t)$ is a constant during recognition and doesn't hurt us to have in the equation.

The error-back-propagation algorithm for training an MLP requires that we know the correct phone label q_j for each observation o_t. Given a large training set of observations and correct labels, the algorithm iteratively adjusts the weights in the MLP to minimize the error with this training set. In the next section we will see where this labeled training set comes from, and how this training fits in with the **embedded training** algorithm used for HMMs. Neural nets seem to achieve roughly the same performance as a Gaussian model but have the advantage of using less parameters and the disadvantage of taking somewhat longer to train.

METHODOLOGY BOX: WORD ERROR RATE

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. 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 is then defined as follows (note that because the equation includes insertions, the error rate can be great than 100%):

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

Here is an example of **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
HYP:	i	GOT	IT	TO	the	*****	FULLEST	i	LOVE	TO	portable
Eval:		I	I	S		D	S		S	S	

REF:	****	PHONE	UPSTAIRS	last	night	so	the	battery	ran	out	
HYP:	FORM	OF	STORES	last	night	so	the	battery	ran	out	
Eval:	I	S	S								

This utterance has 6 substitutions, 3 insertions, and 1 deletion:

$$\text{Word Error Rate} = 100\frac{6+3+1}{18} = 56\%$$

As of the time of this writing, state-of-the-art speech recognition systems were achieving around 20% word error rate on natural-speech tasks like the National Institute of Standards and Technology (NIST)'s Hub4 test set from the Broadcast News corpus (Chen *et al.*, 1999), and around 40% word error rate on NIST's Hub5 test set from the combined Switchboard, Switchboard-II, and CALLHOME corpora (Hain *et al.*, 1999).

7.7 TRAINING A SPEECH RECOGNIZER

We have now introduced all the algorithms which make up the standard speech recognition system that was sketched in Figure 7.2 on page 239. We've seen how to build a Viterbi decoder, and how it takes 3 inputs (the observation likelihoods (via Gaussian or MLP estimation from the spectral features), the HMM lexicon, and the N-gram language model) and produces the most probable string of words. But we have not seen how all the probabilistic models that make up a recognizer get trained.

EMBEDDED TRAINING

In this section we give a brief sketch of the **embedded training** procedure that is used by most ASR systems, whether based on Gaussians, MLPs, or even vector quantization. Some of the details of the algorithm (like the forward-backward algorithm for training HMM probabilities) have been removed to Appendix D.

Let's begin by summarizing the four probabilistic models we need to train in a basic speech recognition system:

- **language model probabilities:** $P(w_i|w_{i-1}w_{i-2})$
- **observation likelihoods:** $b_j(o_t)$
- **transition probabilities:** a_{ij}
- **pronunciation lexicon:** HMM state graph structure

In order to train these components we usually have

- a training corpus of speech wavefiles, together with a word-transcription.
- a much larger corpus of text for training the language model, including the word-transcriptions from the speech corpus together with many other similar texts.
- often a smaller training corpus of speech which is phonetically labeled (i.e. frames of the acoustic signal are hand-annotated with phonemes).

Let's begin with the N-gram language model. This is trained in the way we described in Chapter 6; by counting N-gram occurrences in a large corpus, then smoothing and normalizing the counts. The corpus used for training the language model is usually much larger than the corpus used to train the HMM a and b parameters. This is because the larger the training corpus the more accurate the models. Since N-gram models are much faster to train than HMM observation probabilities, and since text just takes less space than speech, it turns out to be feasible to train language models on huge corpora of as much as half a billion words of text. Generally the corpus used for training the HMM parameters is included as part of the language

model training data; it is important that the acoustic and language model training be consistent.

The HMM lexicon structure is built by hand, by taking an off-the-shelf pronunciation dictionary such as the PRONLEX dictionary (LDC, 1995) or the CMUdict dictionary, both described in Chapter 4. In some systems, each phone in the dictionary maps into a state in the HMM. So the word *cat* would have 3 states corresponding to [k], [ae], and [t]. Many systems, however, use the more complex **subphone** structure described on page 249, in which each phone is divided into 3 states: the beginning, middle and final portions of the phone, and in which furthermore there are separate instances of each of these subphones for each **triphone** context.

The details of the embedded training of the HMM parameters varies; we'll present a simplified version. First, we need some initial estimate of the transition and observation probabilities a_{ij} and $b_j(o_t)$. For the transition probabilities, we start by assuming that for any state all the possible following states are all equiprobable. The observation probabilities can be bootstrapped from a small hand-labeled training corpus. For example, the TIMIT or Switchboard corpora contain approximately 4 hours each of phonetically labeled speech. They supply a 'correct' phone state label q for each frame of speech. These can be fed to an MLP or averaged to give initial Gaussian means and variances. For MLPs this initial estimate is important, and so a hand-labeled bootstrap is the norm. For Gaussian models the initial value of the parameters seems to be less important and so the initial mean and variances for Gaussians often are just set identically for all states by using the mean and variances of the entire training set.

Now we have initial estimates for the a and b probabilities. The next stage of the algorithm differs for Gaussian and MLP systems. For MLP systems we apply what is called a **forced Viterbi** alignment. A forced Viterbi alignment takes as input the correct words in an utterance, along with the spectral feature vectors. It produces the best sequence of HMM states, with each state aligned with the feature vectors. A forced Viterbi is thus a simplification of the regular Viterbi decoding algorithm, since it only has to figure out the correct phone sequence, but doesn't have to discover the word sequence. It is called **forced** because we constrain the algorithm by requiring the best path to go through a particular sequence of words. It still requires the Viterbi algorithm since words have multiple pronunciations, and since the duration of each phone is not fixed. The result of the forced Viterbi is a set of features vectors with 'correct' phone labels, which can then be used to retrain the neural network. The counts of the transitions which are taken in

FORCED VITERBI

the forced alignments can be used to estimate the HMM transition probabilities.

For the Gaussian HMMs, instead of using forced Viterbi, we use the forward-backward algorithm described in Appendix D. 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. Just as for the MLP situation, the forward-backward algorithm needs to be constrained by our knowledge of the correct words. The forward-backward algorithm computes its probabilities given a model λ. We use the 'known' words sequence in a transcribed sentence to tell us which word models to string together to get the model λ that we use to compute the forward and backward probabilities for each sentence.

7.8 Waveform Generation for Speech Synthesis

Now that we have covered acoustic processing we can return to the acoustic component of a text-to-speech (TTS) system. Recall from Chapter 4 that the output of the linguistic processing component of a TTS system is a sequence of phones, each with a duration, and a F0 contour which specifies the pitch. This specification is often called the **target**, as it is this that we want the synthesizer to produce.

TARGET

WAVEFORM CONCATENATION

The most commonly used type of algorithm works by **waveform concatenation**. Such **concatenative synthesis** is based on a database of speech that has been recorded by a single speaker. This database is then segmented into a number of short units, which can be phones, diphones, syllables, words or other units. The simplest sort of synthesizer would have phone units and the database would have a single unit for each phone in the phone inventory. By selecting units appropriately, we can generate a series of units which match the phone sequence in the input. By using signal processing to smooth joins at the unit edges, we can simply concatenate the waveforms for each of these units to form a single synthetic speech waveform.

Experience has shown that single phone concatenative systems don't produce good quality speech. Just as in speech recognition, the context of the phone plays an important role in its acoustic pattern and hence a /t/ before a /a/ sounds very different from a /t/ before an /s/.

The triphone models described in Figure 7.11 on page 249 are a popular choice of unit in speech recognition, because they cover both the left and right contexts of a phone. Unfortunately, a language typically has a

very large number of triphones (tens of thousands) and it is currently prohibitive to collect so many units for speech synthesis. Hence **diphones** are often used in speech synthesis as they provide a reasonable balance between context-dependency and size (typically 1000–2000 in a language). In speech synthesis, diphone units normally start half-way through the first phone and end half-way through the second. This is because it is known that phones are more stable in the middle than at the edges, so that the middles of most /a/ phones in a diphone are reasonably similar, even if the acoustic patterns start to differ substantially after that. If diphones are concatenated in the middles of phones, the discontinuities between adjacent units are often negligible.

DIPHONES

Pitch and Duration Modification

The diphone synthesizer as just described will produce a reasonable quality speech waveform corresponding to the requested phone sequence. But the pitch and duration (i.e. the prosody) of each phone in the concatenated waveform will be the same as when the diphones were recorded and will not correspond to the pitch and durations requested in the input. The next stage of the synthesis process therefore is to use signal processing techniques to change the prosody of the concatenated waveform.

The linear prediction (LPC) model described earlier can be used for prosody modification as it explicitly separates the pitch of a signal from its spectral envelope If the concatenated waveform is represented by a sequence of linear prediction coefficients, a set of pulses can be generated corresponding to the desired pitch and used to re-excite the coefficients to produce a speech waveform again. By contracting and expanding frames of coefficients, the duration can be changed. While linear prediction produces the correct F0 and durations it produces a somewhat "buzzy" speech signal.

Another technique for achieving the same goal is the time-domain pitch-synchronous overlap and add (**TD-PSOLA**) technique. TD-PSOLA works **pitch-synchronously** in that each frame is centered around a **pitchmark** in the speech, rather than at regular intervals as in normal speech signal processing. The concatenated waveform is split into a number of frames, each centered around a pitchmark and extending a pitch period either side. Prosody is changed by recombining these frames at a new set of pitchmarks determined by the requested pitch and duration of the input. The synthetic waveform is created by simply overlapping and adding the frames. Pitch is increased by making the new pitchmarks closer together (shorter pitch periods implies higher frequency pitch), and decreased by making them further

TD-PSOLA

apart. Speech is made longer by duplication frames and shorter by leaving frames out. The operation of TD-PSOLA can be compared to that of a tape recorder with variable speed – if you play back a tape faster than it was recorded, the pitch periods will come closer together and hence the pitch will increase. But speeding up a tape recording effectively increases the frequency of *all* the components of the speech (including the formants which characterize the vowels) and will give the impression of a "squeaky", unnatural voice. TD-PSOLA differs because it separates each frame first and then decreases the distance between the frames. Because the internals of each frame aren't changed, the frequency of the non-pitch components is hardly altered, and the resultant speech sounds the same as the original except with a different pitch.

Unit Selection

While signal processing and diphone concatenation can produce reasonable quality speech, the result is not ideal. There are a number of reasons for this, but they all boil down to the fact that having a single example of each diphone is not enough. First of all, signal processing inevitably incurs distortion, and the quality of the speech gets worse when the signal processing has to stretch the pitch and duration by large amounts. Furthermore, there are many other subtle effects which are outside the scope of most signal processing algorithms. For instance, the amount of vocal effort decreases over time as the utterance is spoken, producing weaker speech at the end of the utterance. If diphones are taken from near the start of an utterance, they will sound unnatural in phrase-final positions.

Unit-selection synthesis is an attempt to address this problem by collecting several examples of each unit at different pitches and durations and linguistic situations, so that the unit is close to the target in the first place and hence the signal processing needs to do less work. One technique for unit-selection (Hunt and Black, 1996) works as follows:

The input to the algorithm is the same as other concatenative synthesizers, with the addition that the F0 contour is now specified as three F0 values per phone, rather than as a contour. The technique uses phones as its units, indexing phones in a large database of naturally occurring speech Each phone in the database is also marked with a duration and three pitch values. The algorithm works in two stages. First, for each phone in the target word, a set of candidate units which match closely in terms of phone identity, duration and F0 is selected from the database. These candidates are ranked

using a **target cost** function, which specifies just how close each unit actually is to the target. The second part of the algorithm works by measuring how well each candidate for each unit joins with its neighbor's candidates. Various locations for the joins are assessed, which allows the potential for units to be joined in the middle, as with diphones. These potential joins are ranked using a **concatenation cost** function. The final step is to pick the best set of units which minimize the overall target and concatenation cost for the whole sentence. This step is performed using the Viterbi algorithm in a similar way to HMM speech recognition: here the target cost is the observation probability and the concatenation cost is the transition probability.

By using a much larger database which contains many examples of each unit, unit-selection synthesis often produces more natural speech than straight diphone synthesis. Some systems then use signal processing to make sure the prosody matches the target, while others simply concatenate the units following the idea that a utterance which only roughly matches the target is better than one that exactly matches it but also has some signal processing distortion.

7.9 HUMAN SPEECH RECOGNITION

Speech recognition in humans shares some features with the automatic speech recognition models we have presented. We mentioned above that signal processing algorithms like PLP analysis (Hermansky, 1990) were in fact inspired by properties of the human auditory system. In addition, four properties of human **lexical access** (the process of retrieving a word from the mental lexicon) are also true of ASR models: **frequency**, **parallelism**, **neighborhood effects**, and **cue-based processing**. For example, as in ASR with its N-gram language models, human lexical access is sensitive to word **frequency**, High-frequency spoken words are accessed faster or with less information than low-frequency words. They are successfully recognized in noisier environments than low frequency words, or when only parts of the words are presented (Howes, 1957; Grosjean, 1980; Tyler, 1984, inter alia). Like ASR models, human lexical access is **parallel**: multiple words are active at the same time (Marslen-Wilson and Welsh, 1978; Salasoo and Pisoni, 1985, inter alia). Human lexical access exhibits **neighborhood effects** (the neighborhood of a word is the set of words which closely resemble it). Words with large frequency-weighted neighborhoods are accessed slower than words with less neighbors (Luce *et al.*, 1990). Jurafsky (1996) shows

LEXICAL ACCESS

that the effect of neighborhood on access can be explained by the Bayesian models used in ASR.

Finally, human speech perception is **cue-based**: speech input is interpreted by integrating cues at many different levels. For example, there is evidence that human perception of individual phones is based on the integration of multiple cues, including acoustic cues, such as formant structure or the exact timing of voicing, (Oden and Massaro, 1978; Miller, 1994), visual cues, such as lip movement (Massaro and Cohen, 1983; Massaro, 1998), and lexical cues such as the identity of the word in which the phone is placed (Warren, 1970; Samuel, 1981; Connine and Clifton, 1987; Connine, 1990). For example, in what is often called the **phoneme restoration effect**, Warren (1970) took a speech sample and replaced one phone (e.g. the [s] in *legislature*) with a cough. Warren found that subjects listening to the resulting tape typically heard the entire word *legislature* including the [s], and perceived the cough as background. Other cues in human speech perception include semantic **word association** (words are accessed more quickly if a semantically related word has been heard recently) and **repetition priming** (words are accessed more quickly if they themselves have just been heard). The intuitions of both of these results are incorporated into recent language models discussed in Chapter 6, such as the cache model of Kuhn and de Mori (1990), which models repetition priming, or the trigger model of Rosenfeld (1996) and the LSA models of Coccaro and Jurafsky (1998) and Bellegarda (1999), which model word association. In a fascinating reminder that good ideas are never discovered only once, Cole and Rudnicky (1983) point out that many of these insights about context effects on word and phone processing were actually discovered by William Bagley (Bagley, 1901). Bagley achieved his results, including an early version of the phoneme restoration effect, by recording speech on Edison phonograph cylinders, modifying it, and presenting it to subjects. Bagley's results were forgotten and only rediscovered much later.

WORD ASSOCIATION
REPETITION PRIMING

One difference between current ASR models and human speech recognition is the time-course of the model. It is important for the performance of the ASR algorithm that the the decoding search optimizes over the entire utterance. This means that the best sentence hypothesis returned by a decoder at the end of the sentence may be very different than the current-best hypothesis, half way into the sentence. By contrast, there is extensive evidence that human processing is **on-line**: people incrementally segment and utterance into words and assign it an interpretation as they hear it. For example, Marslen-Wilson (1973) studied **close shadowers**: people who are able to

ON-LINE

shadow (repeat back) a passage as they hear it with lags as short as 250 ms. Marslen-Wilson found that when these shadowers made errors, they were syntactically and semantically appropriate with the context, indicating that word segmentation, parsing, and interpretation took place within these 250 ms. Cole (1973) and Cole and Jakimik (1980) found similar effects in their work on the detection of mispronunciations. These results have led psychological models of human speech perception (such as the Cohort model (Marslen-Wilson and Welsh, 1978) and the computational TRACE model (McClelland and Elman, 1986)) to focus on the time-course of word selection and segmentation. The TRACE model, for example, is a **connectionist** or **neural network** interactive-activation model, based on independent computational units organized into three levels: feature, phoneme, and word. Each unit represents a hypothesis about its presence in the input. Units are activated in parallel by the input, and activation flows between units; connections between units on different levels are excitatory, while connections between units on single level are inhibitatory. Thus the activation of a word slightly inhibits all other words.

CONNECTIONIST
NEURAL NETWORK

We have focused on the similarities between human and machine speech recognition; there are also many differences. In particular, many other cues have been been shown to play a role in human speech recognition but have yet to be successfully integrated into ASR. The most important class of these missing cues is prosody. To give only one example, Cutler and Norris (1988), Cutler and Carter (1987) note that most multisyllabic English word tokens have stress on the initial syllable, suggesting in their metrical segmentation strategy (MSS) that stress should be used as a cue for word segmentation.

7.10 SUMMARY

Together with chapters 4, 5, and 6, this chapter introduced the fundamental algorithms for addressing the problem of **Large Vocabulary Continuous Speech Recognition** and **Text-To-Speech synthesis**.

- 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, wound waves are **sampled**, **quantized**, and converted to some sort of **spectral representation**; A commonly used spectral representation is the **LPC cepstrum**, which

provides a vector of features for each time-slice of the input.

- These **feature vectors** are used to estimate the **phonetic likelihoods** (also called **observation likelihoods**) either by a mixture of **Gaussian** estimators or by a **neural net**.
- **Decoding** or **search** is the process of finding the optimal sequence of model states which matches a sequence of input observations. (The fact that are two 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** from artificial intelligence).
- 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 **A*** decoding. Both algorithms take as input a series of feature vectors, and 2 ancillary algorithms: one for assigning likelihoods (e.g. Gaussians or MLP) and one for assigning priors (e.g. an N-gram language model). Both give as output 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. This HMM observation probability estimation can be done via a Gaussian or an MLP.
- One way to implement the acoustic component of a TTS system is with **concatenative synthesis**, in which an utterance is built by concatenating and then smoothing diphones taken from a large database of speech recorded by a single speaker.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The first machine which recognized speech was probably a commercial toy named "Radio Rex" which was sold in the 1920's. Rex was a celluloid dog which 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 in "Rex", the dog seemed to come when he was called (David and Selfridge, 1962).

By the late 1940's and early 1950's, 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 1970's 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 Chapter 5, 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 afterwards, 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.

WARPING

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 error rate of 94% on a simple task).

Beginning in the mid-80's, 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 (also called Hub-4) domain (LDC, 1998; Graff, 1997) (transcription of actual news broadcasts, including quite difficult passages such as on-the-street interviews) and the CALLHOME and CALLFRIEND domain (LDC, 1999) (natural telephone conversations between friends), part of what was also called Hub-5. The Air Traffic Information System (ATIS) task (Hemphill *et al.*, 1990) was a speech understanding task whose goal

was to simulate helping a user book a flight, by answering questions about potential airlines, times, dates, etc.

Each of the ARPA tasks involved an approximately annual **bake-off** at BAKE-OFF
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) but then made available for general research use.

There are a number of textbooks on speech recognition that are good choices for readers who seek a more in-depth understanding of the material in this chapter: Jelinek (1997), Gold and Morgan (1999), and Rabiner and Juang (1993) are the most comprehensive. The last two textbooks also have comprehensive 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). Another useful tutorial paper is Knill and Young (1997). Research in the speech recognition field often appears in the proceedings of the biennial EUROSPEECH Conference and the International Conference on Spoken Language Processing (ICSLP), held in alternating years, as well as the annual IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP). Journals include Speech Communication, Computer Speech and Language, IEEE Transactions on Pattern Analysis and Machine Intelligence, and IEEE Transactions on Acoustics, Speech, and Signal Processing.

EXERCISES

7.1 Analyze each of the errors in the incorrectly recognized transcription of "um the phone is I left the..." on page 269. 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

7.2 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 Figure 7.9 on page 247 to make use of logprobs instead of probabilities.

7.3 Now modify the Viterbi algorithm in Figure 7.9 on page 247 to implement the beam search described on page 249. Hint: You will probably need to add in code to check whether a given state is at the end of a word or not.

7.4 Finally, modify the Viterbi algorithm in Figure 7.9 on page 247 with more detailed pseudocode implementing the array of backtrace pointers.

7.5 Implement the Stack decoding algorithm of Figure 7.14 on 254. Pick a very simple h^* function like an estimate of the number of words remaining in the sentence.

7.6 Modify the forward algorithm of Figure 5.16 to use the tree-structured lexicon of Figure 7.18 on page 257.