SPEECH RECOGNITION

Steve Renals

February 1998

Chapter 1

TEMPLATE MATCHING

1.1 Introduction

A well-studied approach to automatic speech recognition is based on the storage of one or more acoustic patterns (**templates**) for each word in the recognition vocabulary. If the system is **speaker-dependent**, then these words will have been spoken by the intended user. The recognition process then consists of **matching** the incoming speech with stored templates.

This process has been most used for isolated word recognition, In that case the input speech consists of an isolated whole word (which has been **endpoint detected**, typically using an energy-based measure) and this word is matched against each of the stored templates. The template with the lowest **distance** from the input is the recognised word. In these notes, we'll mainly be considering isolated word recognition.

There are two key issues that need to be dealt with to construct an isolated word recognition system based on template matching:

Features In what form should the speech data be represented? (i.e. what are the feature vectors?)

Distances How do we compute the distances between two speech patterns?

The answer to the second question is crucial. It can be broken down into two parts:

1. How do we compute the **local distance** between two feature vectors?
2. How do we compute the **global distance** between two speech patterns (words) from the **local distances**, given that input word and the template may have different durations/time scales?

1.2 Feature Vectors

The way data is represented is crucial to speech pattern recognition. As an example consider fig. 1.1. This shows an utterance "She had your dark suit in greasy wash water all year" spoken twice by the same speaker, represented as a time-amplitude waveform and as a spectrogram.

All the information in the spectrogram is in the waveform—further processing will never create new information—but the spectrogram seems to contain most of the essential information in a "clearer" way. The feature extraction process might then be regarded as an information reduction process in which "irrelevant" information is thrown away.

Of course there are many possible feature extraction processes e.g. wideband and narrowband FFTs, auditory representations (e.g. based on gammatone filterbank).

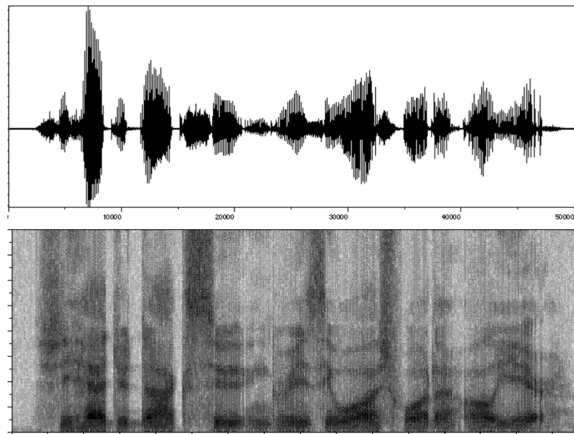

Figure 1.1: Waveform and spectrogram representations of “She had your dark suit in greasy wash water all year” spoken by a male speaker

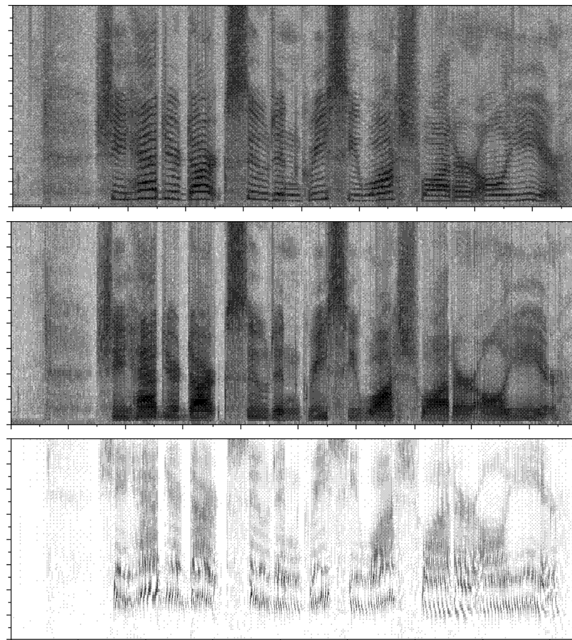

Figure 1.2: Feature extraction processes: narrowband FFT (top), wideband FFT (middle) and auditory filterbank excitation pattern (bottom).

The question of optimal feature extraction is not trivial. If we are interested in recognition, then an ideal feature extractor might be one that produces a string of words (making the rest of the recognizer redundant)! On the other hand, separating the feature extraction process from the pattern recognition process is a sensible thing to do, since it enables us to encapsulate the pattern recognition process.

We shall concern ourselves with frame based feature extraction. That is, the feature analysis consists of computing a feature vector at regular intervals. For example if we perform a filterbank analysis, then our feature vector may consist of the energies in each band averaged over (say) 20ms. For a linear predictive analysis the feature vector consists of the prediction coefficients (or transformations of them). A common feature vector type used in speech recognition are Mel Frequency Cepstral Coefficients (MFCCs), covered in COM325.

1.3 Local Distances

An important operation is the comparison of two input vectors—how similar are vectors $\mathbf{x}$ and $\mathbf{y}$, or what is the **distance** between them? In speech recognition we often refer the distance between two frames of data as a **local distance**, with the overall distance between an input word and a template referred to as a **global distance**.

We shall use the notation $d(\mathbf{x},\mathbf{y})$ for the local distance between two feature vectors. Two possible choices for a local distance metric are:

- The **Euclidean** distance metric:

$$d_2(\mathbf{x},\mathbf{y}) = \sqrt{(\mathbf{x}-\mathbf{y})^T(\mathbf{x}-\mathbf{y})} = \sqrt{\sum_i (x_i - y_i)^2} \tag{1.1}$$

  where x_i is the ith element of $\mathbf{x}$ and x^T signifies the transpose of x.

- The **City Block** or **Manhattan** distance metric:

$$d_1(\mathbf{x},\mathbf{y}) = |(\mathbf{x}-\mathbf{y})| = \sum_i |(x_i - y_i)| \tag{1.2}$$

The principal benefit of the City Block metric is that it is computationally cheaper than the Euclidean metric. The Euclidean distance gives more weight to large differences in a single feature (relative to smaller differences in all features) compared with the City Block distance. The Euclidean distance is much more frequently used, particularly as it has certain theoretical properties that arise when it is used in statistically-based speech recognition.

Both these metrics make the implicit assumption that individual elements of a feature vector are not correlated with each other. This is clearly not the case for a simple filter-bank representation – what goes on in one channel is clearly related to what is going on in the neighbouring channel. However cepstral feature vectors do have uncorrelated elements; furthermore, it can be shown that using the Euclidean distance to compare cepstra has several "nice" theoretical properties.

Finally, there are some speech specific distance measures. One of the best known is the **Itakura** distance which is used with an LPC analysis. This distance is similar in concept to Residual Excited LPC. Essentially the distance measure is the residual that results from using the LPC filter derived from one signal to inverse filter the other.

1.4 Global Distances

Speech is a time-dependent process. Several utterances of the same word are likely to have different durations, and utterances of the same word with the same duration will differ in

the middle, due to different parts of the words being spoken at different rates. To obtain a global distance between two speech patterns (represented as a sequence of vectors) a **time alignment** must be performed.

This problem is illustrated in figure 1.4, in which a "time-time" matrix is used to visualize the alignment. As with all the time alignment examples the reference pattern (template) goes up the side and the input pattern goes along the bottom. In this illustration the input "SsPEEhH" is a 'noisy' version of the template "SPEECH". The idea is that 'h' is a closer match to 'H' compared with anything else in the template. The input "SsPEEhH" will be matched against all templates in the system's repository. The best matching template is the one for which there is the lowest distance path aligning the input pattern to the template. A simple global distance score for a path is simply the sum of local distances that go to make up the path.

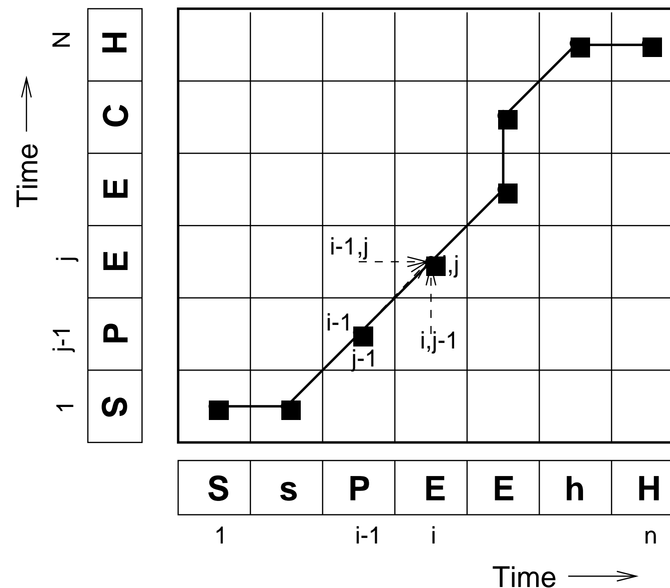

Figure 1.3: Illustration of a time alignment path between a template pattern "SPEECH" and a noisy input "SsPEEhH".

How do we find the best-matching (= lowest global distance) path between an input and a template? We could evaluate all possible paths – but this is extremely inefficient as the number of possible paths is exponential in the length of the input. Instead, we will consider what constraints there are on the matching process (or that we can impose on that process) and use those constraints to come up with an efficient algorithm. The constraints we shall impose are straightforward and not very restrictive:

1. Matching paths cannot go backwards in time;
2. Every frame in the input must be used in a matching path;
3. Local distance scores are combined by adding to give a global distance.

For now we will also extend (2) to say that every frame in the template and the input must be used in a matching path. This means that if we take a point (i, j) in the time-time matrix (where i indexes the input pattern frame, j the template frame), then previous point must have been $(i-1, j-1)$, $(i-1, j)$ or $(i, j-1)$ (see figure 1.4). The key idea in dynamic programming is that at point (i, j) we just continue with the lowest distance path from $(i-1, j-1)$, $(i-1, j)$ or $(i, j-1)$.

This algorithm is known as **Dynamic Programming** (DP). When applied to template-based speech recognition, it is often referred to as **Dynamic Time Warping** (DTW). DP is guaranteed to find the lowest distance path through the matrix, while minimizing the amount of computation. The DP algorithm operates in a **time-synchronous** manner: each column of the time-time matrix is considered in succession (equivalent to processing the input frame-by-frame) so that, for a template of length N, the maximum number of paths being considered at any time is N.

Exercise: Make sure you can convince yourself of this.

If we denote the global distance score up to (i,j) as $D(i,j)$ and the local distance score at that point as $d(i,j)$, then we can express dynamic programming using the following relation:

$$D(i,j) = \min[D(i-1,j-1), D(i-1,j), D(i,j-1)] + d(i,j) \tag{1.3}$$

Given that $D(1,1) = d(1,1)$ (this is the initial condition), we have the basis for an efficient recursive algorithm for computing $D(i,j)$. The final global distance $D(n,N)$ gives us the overall matching score of the template with the input. The input word is then recognized as the word corresponding to the template with the lowest matching score. (Note that N will be different for each template.)

For basic speech recognition DP has a small memory requirement, the only storage required by the search (as distinct from the templates) is an array that holds a single column of the time-time matrix.

Exercise: Show this is so.

If it is required to trace back along the best-matching path (rather than just knowing the score at the end of it) then a **backtrace array** (or backpointer array) must be kept with entries in the array pointing to the preceding point in that path.

1.5 Extensions to Basic DP

Although the basic DP algorithm (1.3) has the benefit of symmetry (i.e. all frames in both input and reference must be used) this has the side effect of penalising horizontal and vertical transitions relative to diagonal ones.

EXERCISE: Convince yourself that this is so by considering the different ways of going from $(i-1,j-1)$ to (i,j).

One way to avoid this effect is to double the contribution of $d(i,j)$ when a diagonal step is taken. This has the effect of charging no penalty for moving horizontally or vertically rather than diagonally. This is also not desirable (*why?*), so independent penalties d_h and d_v can be applied to horizontal or vertical moves. In this case (1.3) becomes:

$$D(i,j) = \min[D(i-1,j-1) + 2d(i,j), D(i-1,j) + d(i,j) + d_h, D(i,j-1) + d(i,j) + d_v] \tag{1.4}$$

Suitable values for d_h and d_v may be found experimentally.

This approach will favour shorter templates over longer templates, so a further refinement is to normalize the final distance score by template length to redress the balance.

If we restrict the allowable path transitions to be:

- $(i-1,j-2) \rightarrow (i,j)$ (skips a template frame – diagonal slope 2)
- $(i-1,j-1) \rightarrow (i,j)$ (usual diagonal – slope 1)
- $(i-1,j) \rightarrow (i,j)$ (duplicates a template frame – slope 0)

then we are assume that each frame of the input pattern is used once and only once. This means that we can dispense with template-length normalization and it is not required to add

the local distance in twice for diagonal (slope 1) path transitions. This approach is referred to as **asymmetric dynamic programming** (in contrast to the version in figure 1.4 which is symmetric).

Dynamic programming may be regarded as an efficient algorithm to perform an exhaustive search. This means that it will always find the path with the lowest global distance when matching an input with a template, and will be a lot more efficient than exhaustive search. However, there can still be a lot of computation, particularly when there are many templates to compare an input pattern against. A significant saving in computation can be made if **pruning** is employed – this is sometimes called **beam search**. The basic idea of pruning is very simple: poorly scoring paths (i.e., those paths whose global distance is further away from the lowest distance path at that time) are pruned from the search. This is illustrated in figure 1.4. Pruning is a heuristic, which means it does not guarantee that the algorithm will find the minimum distance path. An example of this situation is given in figure 1.5 in which the input pattern is very poorly matched to the template and if pruning had been employed, the best scoring path would not have been found.

Figure 1.4: An example of DP search using the asymmetric approach together with path pruning.

Figure 1.5: An example of DP search comparing an input (”three”) with a dissimilar template (”eight”). This is an example of a case where the final lowest distance path would have been pruned mid-utterance. The path marked by crosses was the lowest distance until the final frame.

1.6 Connected Word Recognition

So far we have considered dynamic programming as an algorithm for isolated word recognition. However it is quite possible to generalize the algorithm to the case of connected word recognition. In this case a point in the search space is represented by three indices (i, j, k), representing frame i of the input, frame j of template k — unlike isolated word recognition where one template is processed at a time, in connected word recognition the word template being processed is a variable to keep track of (figure 1.6). The best path through the word templates k can be found in the same way as for isolated word recognition. However, the start and end points of the templates are not known so for every input frame when a valid path reaches the end of a template k_1, a new template k_2 might begin.

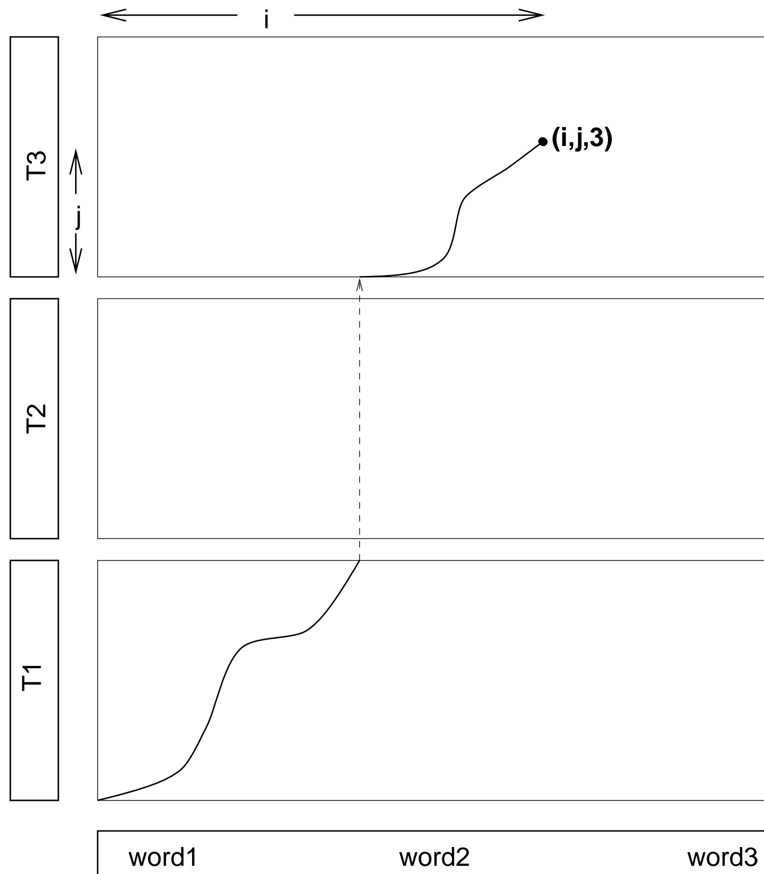

Figure 1.6: Dynamic programming for connected word recognition.

The global distance of a path is kept in the usual way; it is the distance summed over all templates the path has passed through, so the distance for individual templates is not recorded. This means that template duration normalization is not possible, so asymmetric dynamic programming should be used.

If paths through two (or more) templates reach the template end at the same frame, then only one of those paths can be extended by starting a new template (figure 1.7).

Exercise: Why is this so?

Which path should be chosen? The answer is provided by the dynamic programming principal: choose the past with the lowest global distance. We can represent this in the

following equation:

$$D(i,1,\ell) = \min_k D(i-1,\text{len}(k),k) \tag{1.5}$$

where $\text{len}(k)$ is the length (index of the final frame) of template k.

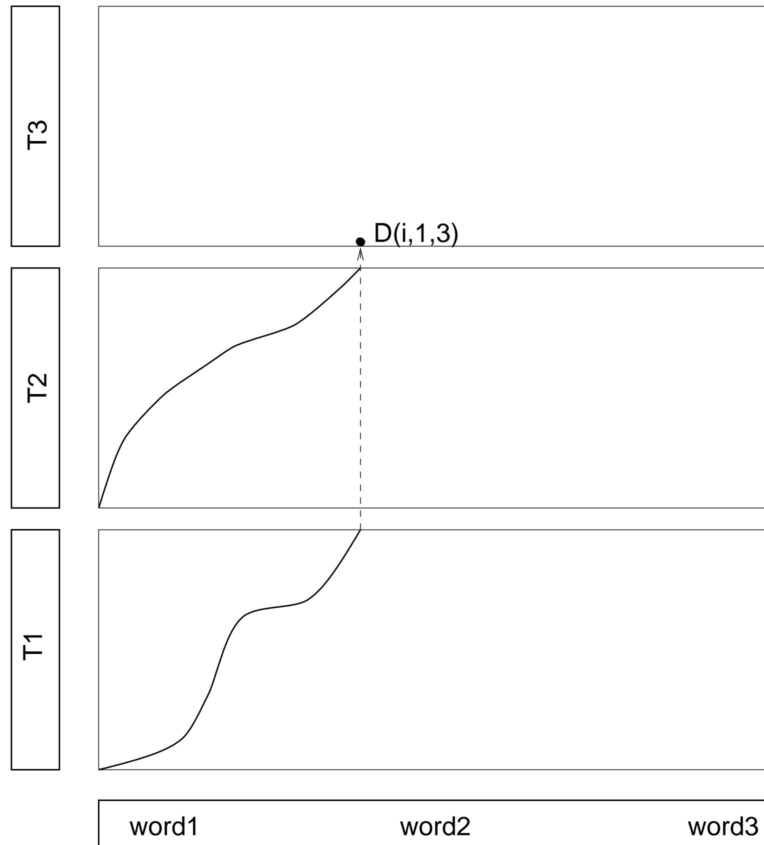

Figure 1.7: DP for connected word recognition when valid paths have reached the end-point of templates T1 and T2 at the same frame. In this case, $D(i,1,3) = \min\ [\ D(i-1, len(1), 1),\ D(i-1, len(2), 2)\ ] + d(i,1,3)$.

As usual we choose the recognised string as the one corresponding to the lowest distance path at the end of the input utterance. But at this point we only know the identity of the final template: we need to know the template sequence which made up this path. This is obtained by keeping track of the template sequence using **backpointers**. When a path enters a new template, the previous template index is stored along with the current path (recall that we choose the previous template whose path had the smallest distance). Upon exiting a template we make an entry in the **backpointer array** that points to the previous template. A typical entry in this array is of the form:

```
backpointer[k][i] = (k-prev, i-prev)
```

That is each array point to a previous template and the exit time of that path from that template. Using this information the sequence of templates that made up a path can be found by following the sequence of backpointers.

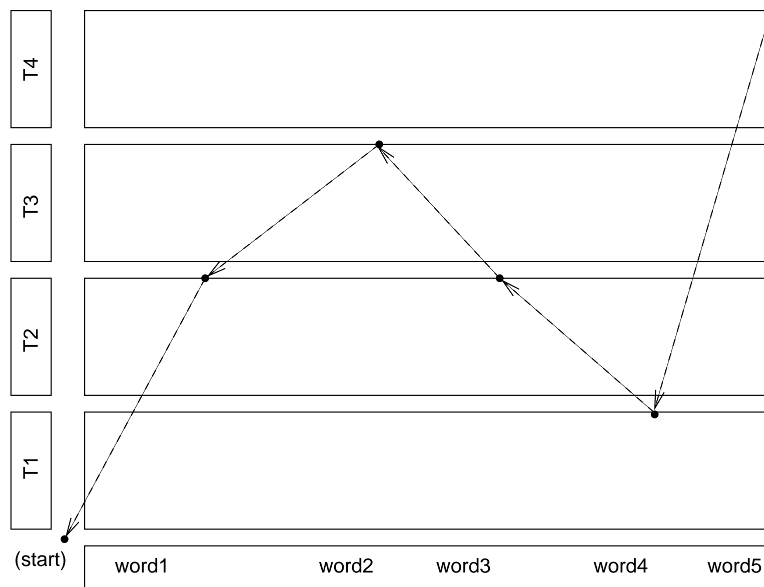

Figure 1.8: Illustration of backpointers in DP connected word recognition. In this case, the recognised sequence of templates was T2, T3, T2, T1, T4.

Chapter 2

FINITE STATE MODELS

2.1 Introduction

As covered in the lectures, there are major drawbacks to using template-based systems for large vocabularies, multiple speakers/speaker-independence and continuous speech. Rather than storing many examples and matching input speech against all possible templates and template sequences, the **statistical modelling** approach represents each word (or other unit of speech) by a mathematical model.

There are two basic classes of model that can be used for speech recognition (and other pattern processing tasks): **recognition models** (recognizers) and **generative models** (generators). (NB: don't confuse the model description "recognizer" with the general process of speech recognition.)

Recognizers Recognition models are used in pattern recognition by constructing the boundaries between classes. They are *discriminative* in that the aim of the model is to discriminate between patterns (or pattern sequences) of different classes.

Generators A generative model is used to generate patterns (or pattern sequences). Generative models can be used for recognition tasks by assigning an input pattern to the class whose model was most likely to have generated the pattern (or the class whose model generates patterns that are most similar to the input).

Recognition models and generative models are conceptually different. A recognition model concentrates on the class boundaries, aiming to find the key features that discriminates between different classes. A generative model aims to model a class as accurately as possible, irrespective of what the other models are doing.

Both generative and recognition models have been used for speech recognition. Most state-of-the-art speech recognition systems are based on generative models, although some labs have started to develop state-of-the-art systems based on recognition model.

2.2 Finite State Machines

Most modern speech recognition systems are based on finite state machine models. As we'll see, finite state machines can be used for either recognition or generative models. From previous courses, you should be pretty familiar with finite state machines being used to model and recognize sequences of symbols.

Recall that a nondeterministic finite state machine may be defined as a 5-tuple (X,Y,Q,f,g), where X is the alphabet of input symbols, Y is the alphabet of output symbols, Q is the set of states, $f(Q \times X) \to 2^Q$ is the next-state function and $g(Q) \to Y$ is the output function. We're interested into specializations of this finite state model:

- Finite state recognizer (X, Q, f) which has no output alphabet or output function;
- Finite state generator (Y, Q, f, g) which has no input alphabet, and whose next-state function is defined $f(Q) \rightarrow 2^Q$.

A finite state recognizer will recognize a certain class of symbol sequences (specified by the corresponding regular grammar). A finite state generator will generate a regular set of symbol sequences. The basic approach to using either of these models for speech recognition is to construct one finite state machine model per unit of speech.

To use such finite state machines for speech recognition we have to do one of the following:

1. Make the speech data look like a string of symbols; or
2. Adapt the finite state machine model to deal with a stream of feature vectors.

Both of these are possible. For the moment we'll see how **vector quantization** can be used to achieve the first of these.

2.3 Vector Quantization

The idea behind vector quantization (VQ) is that each feature vector is represented by one of K symbols, where K is typically 128–1024. This is achieved by automatically separating the data vectors into K groups or *clusters*. Clustering algorithms are usually unsupervised — there is no oracle that specifies which class a particular feature vector comes from. Instead, the algorithm aims to cluster similar vectors together.

The most widely used clustering algorithm is known as the K-means algorithm. This algorithm is outlined below:

1. Initialize with an arbitrary set of K code vectors ($\mathbf{x}_k$)
2. For each feature vector $\mathbf{x}$ in the training set "quantize" $\mathbf{x}$ into the closest code vector $\mathbf{x}_{k*}$:

   $$k* = \operatorname*{argmin}_k d(\mathbf{x}, \mathbf{x}_k)$$

   Where $d(\cdot, \cdot)$ is the distance measure in feature space (e.g. Euclidean distance)
3. Compute the total distortion that occurs using this quantization:

   $$D = \sum_{\mathbf{x}} d(\mathbf{x}, Q(\mathbf{x})$$

   where $Q(\mathbf{x})$ is the code to which $\mathbf{x}$ is assigned.
4. If D is sufficiently small STOP.
5. Foreach k recompute the centroid of all vectors such that $\mathbf{x}_k = Q(\mathbf{x})$. Update the codebook with this new set of centroids and goto 2.

After the K-means algorithm has completed, new vectors can be assigned to the cluster whose centroid they are closest to. These vectors are then quantized into the label (or "codeword") of that cluster. The VQ process may be summarized as:

1. Training: Use a clustering algorithm to compute the VQ codebook
2. Run Time: For each feature vector find the closest codebook entry (i.e. assign it to a cluster) and represent that vector by the label of the closest codebook entry.

The VQ process does two things for us:

- It reduces the computation since each *n*-dimensional feature vector are reduced to a symbol;
- It enables us to use modelling approaches that work on sequences of symbols.

2.4 Finite State Models of Speech

Now if speech was well modelled by a regular grammar, then it would (of course) be straightforward to use finite state models. For example considered an idealized word that is represented by the regular expression $a^*c^*b^*$, where a, b, c are VQ codewords. The finite state generator on the left of figure 2.1 will produce this regular set, examples of which include *aaacbbb*, *aaaaaaccccbb*, etc. This is all very well, but if we get a codeword sequence *aaaaccdcbb*, it may be that none of our finite state models will generate such a sequence. Intuitively it seems as though this sequence is close to the $a^*c^*b^*$ model, but since it is not a member of that regular set, it will be not be generated. We could extend our finite to state model (on the right of figure 2.1) and this will certainly generate the sequence, but will also generate many others such as *adddddddb*.

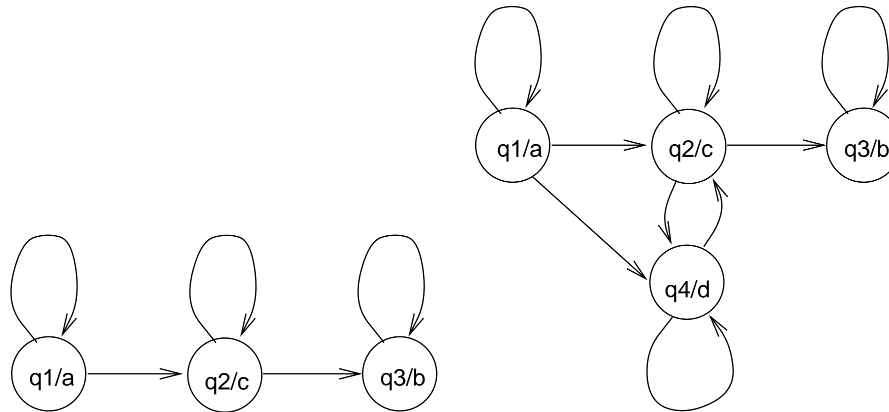

Figure 2.1: Finite state generators for $a^*c^*b^*$ (left) and $a^*(c \vee d)^*b^*$ (right)

The generators in figure 2.1 are **Moore machines**: that is the outputs are function of the state. A related finite state model is known as the **Mealy machine** in which outputs are attached to transitions (i.e. $g(Q \times Q) \rightarrow Y$). A Mealy machine generator is shown in figure 2.2.

We will mostly consider Moore finite state generators in this course.

2.5 Stochastic Finite State Models of Speech

We would be doing better if we could weight our finite state models, so that sequences such as *aaaaccccbbb* had a stronger weighting relative to *aaaaaccccdcccbb*, which in turn was more strongly weighted compared with *aaaacddcdcddddbb*. One way to do this is to make the next-state function f and the output function g probabilistic. We will also extend the model so that the output function on a state is also non-deterministic (there are several possible symbols that may be output by a state). In the case of a finite state generator we

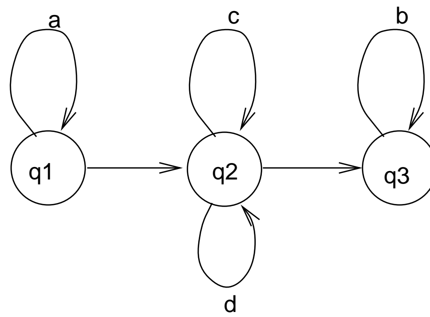

Figure 2.2: Mealy finite state generator for $a^*(c \vee d)^* b^*$

are interested in the probabilities:

$$f(Q) : P(q(t+1)|q(t))$$
$$g(Q) : P(y(t)|q(t))$$

Where $q(t)$ is the state at time t and $y(t)$ is the output at time t. For a finite state recognizer there is no output function:

$$f(Q,X) : P(q(t+1)|q(t), q(t))$$

A finite state machine with probabilistic next-state and/or output functions is referred to as a **stochastic** finite state machine. Figure 2.3 illustrates a stochastic finite state generator that models the noisy $a^* c^* b^*$ situation better than a non-stochastic model could. Figure 2.4 illustrates a stochastic finite state recognizer for the same task.

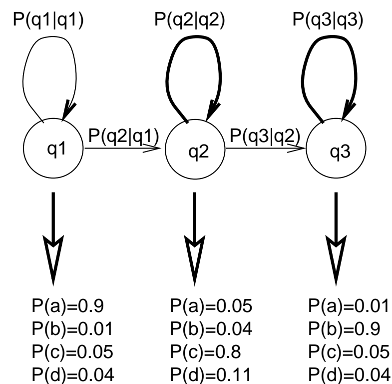

Figure 2.3: Stochastic finite state generator

We've intuitively motivated the use of stochastic finite state machines as a speech recognition model. Let's be explicit about what assumptions we're making in the case of a stochastic finite state generators:

1. **The next-state depends only on the current state (and not any previous states or outputs).** This assumption tells us that the current state of the finite state generator, encapsulates everything we need to know about the system — the system's memory does not extend back to previous states. This is sometimes called the *first-order Markov* property.

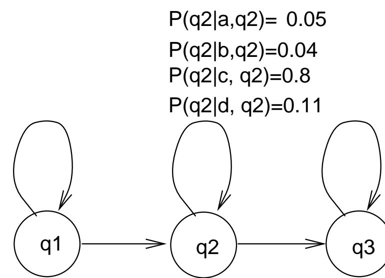

Figure 2.4: Stochastic finite state recognizer (probabilities only shown for 1 transition)

2. **The generated output symbol does not depend on any previous outputs**. This assumption (which is sometimes called *observation independence* or *output independence*) tells us that the current output is not dependent on any previous outputs, but only the current state.

3. **The output symbol depends only on the current state (and not on previous states).** This is assumption tells us that we are working with a Moore machine.

Exercise: Comment on the validity of each of these assumptions.

Chapter 3

HIDDEN MARKOV MODELS

3.1 Introduction

In the speech recognition literature stochastic finite state generators are nearly always referred to as **hidden Markov models** (HMMs). They are referred to as *Markov* models owing to the Markov assumption introduced in part 2. They are called *hidden* because the underlying state sequence is not known, only the output symbols that are generated by the machine.

How can these models be used for speech recognition?

We'll start of by considering the simple case of isolated word recognition. Each word in the vocabulary is represented by a stochastic finite state generator (or HMM). What do we need to specify these HMMs?

- Model topology — how many states and how are they connected
- The **state transition probabilities** — these are the probabilities $P(q(t+1) = m|q(t) = l)$ that specify the next-state function;
- The **output probabilities** — these may be regarded as a histogram for each state. If we have R output symbols ($Y = \{y_1, \ldots, y_R\}$) then each state $q(t) = l$ will have a histogram of R probabilities, $P(y(t) = i|q(t) = l)$

The two sets of probabilities (transition probabilities and output probabilities) are the model parameters, and we will represent the entire parameter set by the symbol Θ. The model itself is represented by the symbol M. (In the simple isolated word case, we will have several models, one for each word.) We will use the notation $Y_1^N = (y(1), y(2), \ldots, y(N)$ to represent a sequence of output symbols, and the notation $Q_1^N = (q(1), q(2), \ldots, q(N))$ to represent a state sequence.

The model topology must be specified in advance. Once it has been specified, there are three key problems that must be addressed if we are to use HMMs for speech recognition:

The Evaluation Problem How do we compute the probability $P(Y_1^N|\Theta, M)$ of HMM (stochastic finite state generator) M with parameters Θ generating an output sequence Y_1^N?

The solution to this problem enables us to evaluate the probability of different HMMs generating the same observation sequence. This computation can be used to compare the probability of different models generating the same data. If we have a different HMM for each word, then the recognized word is the one whose model has the largest probability of generating the data.

The Decoding Problem Given an output sequence Y_1^N, and a model M, with parameters Θ, how do we compute the most probable state sequence Q_1^N?

$$Q_1^N = \operatorname*{argmax}_R P(R_1^N|Y_1^N, \Theta, M)$$

This problem is concerned with uncovering the hidden state sequence from knowledge of the symbols output by the machine. The solution to this problem is useful to get an understanding of the meaning of the states of the machine. It will also turn out to be important in most modern applications of HMMs when elementary models are concatenated together.

The Training Problem How do we adjust the model parameters Θ to maximize the likelihood of the model M producing the output sequence Y_1^N?

The solution to this problem will provided an automatic way to estimate the parameters (output and transition probabilities) of each HMM, using a set of training data.

3.2 Evaluation

The task is to calculate the probability that a model M with parameters Θ generates a particular sequence of output symbols, $P(Y_1^N|\Theta,M)$. If the hidden state sequence ($Q_1^N = q(1),q(2),\ldots,q(N)$) that had resulted in this output was known, then the solution is simple:

$$\begin{aligned} P(Y_1^N|Q_1^N,\Theta,M) &= P(y(1),y(2),\ldots,y(N)|q(1),q(2),\ldots,q(N),\Theta,M) \\ &= P(y(1)|q(1),\Theta,M)P(y(2)|q(2),\Theta,M)\ldots P(y(N)|q(N),\Theta,M) \end{aligned} \tag{3.1}$$

We are using two of the assumptions here:

1. Output independence: $y(t)$ is independent of $y(t-1)$
2. The output $y(t)$ is only dependent on the state $q(t)$

But the state sequence is hidden. $P(Y_1^N|\Theta,M)$ is the probability of the model M generating the data regardless of the state sequence. We can obtain this from (3.1) by considering every possible state sequence.We don't want to weight these equally, so we must include the probability of each state sequence:

$$P(Y_1^N|\Theta,M) = \sum_{Q_1^N} P(Y_1^N|Q_1^N,\Theta,M)P(Q_1^N|\Theta,M) \tag{3.2}$$

We can obtain $P(Q_1^N|\Theta,M)$ from the state transition probabilities:

$$P(Q_1^N|\Theta,M) = P(q(1))P(q(2)|q(1))P(q(3)|q(2))\ldots P(q(n)|q(n-1)) \tag{3.3}$$

We are using the finite state generator assumption (or Markov assumption) that the current state depends only on the previous state. $P(q(1))$ is the *prior probability* for state $q(1)$.

Using equations (3.1), (3.2) and (3.3) we can express $P(Y_1^N|\Theta,M)$ in terms of the basic parameters ($P(y|q)$ and $P(q(t+1)|q(t))$):

$$\begin{aligned} P(Y_1^N|\Theta,M) &= \sum_{Q} (P(y(1)|q(1))P(y(2)|q(2))\ldots P(y(N)|q(N))) \\ &\quad (P(q(1))P(q(2)|q(1))\ldots P(q(n)|q(n-1))) \\ &= \sum_{q(1),q(2),\ldots,q(N)} \left(P(q(1))P(y(1)|q(1)) \prod_{i=2}^{i=N-1} P(q(i)|q(i-1))P(y(i)|q(i)) \right) \end{aligned} \tag{3.4}$$

(For clarity the dependences on Θ and M have been left out.)

Each term inside the sum is the probability of a particular state sequence generating the output sequence — and this is obtained by multiplying the output probabilities and transition probabilities together. But the sum is over all possible state sequences — and if there are S states then there are $O(S^N)$ state sequences!

Fortunately we can compute this sum over state sequences efficiently, using a recurrence relation. The key quantities we use are the so-called "forward probabilities", $\alpha_t(l)$:

$$\alpha_t(l) = P(y(1), y(2), \ldots, y(t), q(t) = l|\Theta, M) \tag{3.5}$$

The forward probability $\alpha_t(l)$ is the probability of the HMM outputting the sequence of outputs $y(1), y(2), \ldots, y(t)$ until time t, and being in state $q(t) = l$ at time t. Note that we can now express the probability of a model M generating a sequence of outputs Y_1^N as a sum (overall final states) of the forward probabilities at time N:

$$\begin{aligned} P(Y_1^N|\Theta, M) &= \sum_{s=1}^{S} P(y(1), y(2), \ldots, y(N), q(N) = s|\Theta, M) \\ &= \sum_{s=1}^{S} \alpha_N(s) \end{aligned} \tag{3.6}$$

This is a big improvement on (3.4): instead of summing over S^N state sequences, we only have to sum over S final states. But we need to be able to calculate $\alpha_t(l)$ efficiently. Fortunately we can do this using the following recurrence relation (which again arises from the HMM assumptions):

$$\begin{aligned} \alpha_1(s) &= P(q(1) = s)P(y(1)|q(1) = s) \\ a_{t+1}(s) &= \left(\sum_{j=1}^{S} \alpha_t(j)P(q(t+1) = s|q(t) = j)\right) P(y(t+1)|q(t+1) = s) \end{aligned} \tag{3.7}$$

To update the forward probabilities requires $O(S^2)$ calculations at each step — much better than the $O(S^N)$ required by the naive computation.

Thus a model is evaluated using the following procedure:

1. Compute the forward probabilities for each time and state using (3.7)
2. Compute the probability of the model generating a particular output sequence using (3.6)

The forward recursion is one of the critical computations in HMMs and will be used again in training.

3.3 Decoding

The forward algorithm recursion is very similar to the dynamic programming recursion used in template matching. The main differences are:

1. Probabilities ($P(y|q)$) are used as the local distances;
2. The allowed transitions are between states (rather than frames of the template) and are also specified as probabilities ($P(q(t+1)|q(t))$);
3. The probabilities of paths are summed in (3.7) — in dynamic programming the maximum path is kept.

The application of the dynamic programming algorithm in a probabilistic context is usually referred to as the *Viterbi algorithm*. In this case we approximate the probability of the model generating the output data, as the probability of the most likely path (state sequence) through the model generating the output data. That is, (3.4) becomes:

$$\tilde{P}(Y_1^N|\Theta, M) = \max_{q(1),q(2),\ldots,q(N)} \left(P(q(1))P(y(1)|q(1)) \prod_{i=2}^{i=N-1} P(q(i)|q(i-1))P(y(i)|q(i)) \right) \tag{3.8}$$

This quantity can be computed efficiently by the Viterbi algorithm — basically the summations in the forward probability recursion (3.7) are replaced by maximizations.

Exercise: Can you see how dynamic programming can accomplish this efficiently?

If an array of backpointers is stored (as in connected word template recognition) then the most probable state sequence that was used to compute $\tilde{P}(Y_1^N|\Theta, M)$ may be determined by backtracing though this array.

3.4 Training

The training problem for HMMs consists of using the training data to estimate the two sets of parameters:

- The output probabilities $P(y(t) = i|q(t) = l)$
- The transition probabilities $P(q(t+1) = m|q(t) = l)$

If we knew the state sequence that had produced the training data then estimating these sets of probabilities is straightforward. If n_{il}^y is the number of times that symbol i has been output by state l, then the re-estimation formula for $P(y = i|q = l)$ is:

$$P(y = i|q = l) = \frac{n_{il}^y}{\sum_j n_{jl}^y} \tag{3.9}$$

And the estimate for the transition probability is:

$$P(q(t+1) = l|q(t) = k) = \frac{n_{kl}^q}{\sum_m n_{km}^q} \tag{3.10}$$

where n_{kl}^q is the number of transitions from state $q = k$ to state $q = l$. It is possible to train a set of HMMs in this manner, using the Viterbi algorithm (section 3.3) to compute the most likely state sequence, and then assume that the training data was generated by this state sequence. This is known as *Viterbi Training*.

However, there is a more general way to train the parameters of an HMM, which does not rely on assuming that the data was produced by the most probable state sequence. Instead, as in the evaluation problem, we sum over all possible state sequences, weighted by the probability of each state sequence. In this way, at each time step rather than knowing precisely which state generated that frame of training data, we compute the probability of each state having generated the training data, and weight the counts by this probability. The algorithm to do this is known as the *Forward-Backward* algorithm, and to do the computations efficiently uses the forward probabilities (equations 3.5 and 3.7) along with another set of probabilities that can be computed recursively, the *backward probabilities*.

The backward probabilities, $\beta_t(q)$ give the probability of outputting the remainder of the output data, given that the state is $q(t)$ at time t:

$$\beta_t(l) = P(y(t+1), y(t+2), \ldots, y(N)|q(t) = l, \Theta, M) \tag{3.11}$$

The recurrence relation to compute these probabilities runs backwards in time:

$$\beta_N(l) = 1 \tag{3.12}$$

$$\beta_t(l) = \sum_{m=1}^{S} P(q(t+1) = m|q(t) = l) P(y(t+1)|q(t+1) = m) \beta_{t+1}(m)$$

Initially the backward probabilities are 1 since there is no remaining sequence to account for. The backward recursion then updates the probabilities by observing that in order to account for the rest of the sequence a transition from every preceding state $q(t)$ had to be

made to state $q(t+1) = m$, with observation symbol $y(t+1)$ being accounted for in that state, and then the rest of the observation sequence accounted for by the previous backward probability ($\beta_{t+1}(m)$).

Together the forward and backward probabilities enable us to calculate some very useful quantities:

1. The probability that the model emits output sequence Y_1^N, given that it is in state $q(t) = l$ at time t:

$$\begin{aligned} P(q(t) = l, Y_1^N|\Theta, M) &= P(y(1), y(2), \ldots, y(t), q(t) = l|\Theta, M) \\ & \quad P(y(t+1), y(t+2), \ldots, y(N)|q(t) = l, \Theta, M) \\ &= \alpha_t(l)\beta_t(l) \end{aligned} \tag{3.13}$$

2. The state occupation probability, $\gamma_t(l)$, the probability of being in state l at time t.

$$\begin{aligned} \gamma_t(l) = P(q(t) = l|Y_1^N, \Theta, M) &= \frac{P(q(t) = l, Y_1^N|\Theta, M)}{P(Y_1^N l|\Theta, M)} \\ &= \frac{\alpha_t(l)\beta_t(l)}{P(Y_1^N|\Theta, M)} \\ &= \frac{\alpha_t(l)\beta_t(l)}{\sum_m \alpha_t(m)\beta_t(m)} \end{aligned} \tag{3.14}$$

3. The probability of being in state $q(t) = l$ at time t and making a transition to state $q(t+1) = m$, $\xi_t(l,m)$:

$$\begin{aligned} \xi_t(l,m) &= P(q(t) = l, q(t+1) = m|Y_1^N, M, \Theta) \\ &= \frac{P(q(t) = l, q(t+1) = m, Y_1^N|M, \Theta)}{P(Y_1^N|M, \Theta)} \\ &= \frac{\alpha_t(l)P(q(t+1) = m|q(t) = l)P(y(t+1)|q(t+1) = m)\beta_{t+1}(m)}{\sum_{j,k}\alpha_t(j)P(q(t+1) = k|q(t) = j)P(y(t+1)|q(t+1) = k)\beta_{t+1}(k)} \end{aligned} \tag{3.15}$$

   In this equation $\alpha_t(l)$ accounts for the first t output symbols, $P(q(t+1)|q(t))$ accounts for the state transition from time t to time $t+1$, $P(y(t+1)|q(t+1)$ accounts for the output symbol at time $t+1$ and $\beta_{t+1}(m)$ accounts for the rest of the output.

(In all these cases the denominator is there for normalization to ensure that things add up to 1.)

We can now see how the output and transition probabilities are reestimated:

Output Probabilities The output probability $P(y = i|q = l)$ is estimated using the ratio of the probability of symbol i being output from state l to the probability of being in state i and outputting any symbol:

$$\begin{aligned} P'(y = i|q = l) &= \frac{P(y = i, q = l|Y_1^N, \Theta, M)}{P(q = l|Y_1^N, \Theta, M)} \\ &= \frac{\sum_{t=1}^N \delta_t(i)\gamma_t(l)}{\sum_{t=1}^N \gamma_t(l)} \end{aligned} \tag{3.16}$$

where $\delta_t(i)$ is a variable which indicates when symbol $y(t) = i$:

$$\begin{aligned} \delta_t(i) &= 1 \quad \text{if } y(t) = i \\ &= 0 \quad \text{if } y(t) \neq i \end{aligned}$$

Transition Probabilities The transition probability $P(q(t+1)=m|q(t)=l)$ is estimated using the ratio of the probability of being in state $q(t)=l$ at time t and making a transition to state $q(t+1)=m$, to the probability of being in state $q(t)$ and making a transition to any state:

$$P'(q(t+1)=m|q(t)=l) = \frac{\sum_{n=1}^{N-1} P(q(n)=l, q(n+1)=k|Y_1^N, M, \Theta)}{\sum_{n=1}^{N-1} P(q(n)=l|Y_1^N, M, \Theta)} \tag{3.17}$$
$$= \frac{\sum_{n=1}^{N-1} \xi_n(l,m)}{\sum_{n=1}^{N-1} \gamma_n(l)}$$

(Note that the summation is only up to $N-1$ since the final transition is from time $N-1$ to N.

In both cases the form of the reestimation equations is the same:

- The denominator is the probability of being in the state at that time, given the data;
- The numerator is the probability of being in that state and outputting the relevant symbol or making the specified transition.

These equations are reestimations; after they are used to reestimate the parameters, the models can be re-evaluated and the parameter estimation process iterated. It can be proved that this training process will eventually converge to an optimal set of HMM parameters that maximize the likelihood of the model producing the data, $P(Y_1^N|M)$.

Summary of HMM Training

The following steps make up one iteration of the training process for an HMM M with using training data Y_1^N (where M may correspond to a model for a word and Y_1^M the acoustic data for that word):

1. Use equations (3.7) and (3.12) to compute the set of forward and backward probabilities for each time and state;
2. Use the forward and backward probabilities in equations (3.14) and (3.15) to compute the probabilities γ and ξ for each time and state;
3. Use the γ and ξ probabilities to reestimate the transition and output probabilities.

This process is iterated until the transition and output probabilities have converged (i.e., do not change from one iteration to another).

The final issue is how the training process is initialized. In theory and in practice it is adequate just to initialize all transition probabilities to $1/S$ and all output probabilities to $1/R$ if there are R output symbols and S states.

Chapter 4

CONTINUOUS SPEECH RECOGNITION

4.1 The Statistical Framework

We can formulate the problem of continuous speech recognition using a statistical framework. The problem is to find the most probable string of words W corresponding to an acoustic signal Y. We can write this as an equation:

$$W = \underset{W'}{\operatorname{argmax}} P(W'|Y) \tag{4.1}$$

What this means is that we need to compute the probability of every word sequence W' given the acoustics Y, and choose the most probable as the recognized output.. This is all very well, but it means that we need to estimate $P(W|Y)$ for every possible word string! We'll look at how we can do this.

The first thing we can do is apply Bayes' Rule to split the probability $P(W|Y)$ into two parts:

$$P(W|Y) = \frac{P(Y|W)P(W)}{P(Y)} \tag{4.2}$$

$$= \left(\frac{P(Y|W)}{P(Y)}\right)(P(W)) \tag{4.3}$$

Now $P(Y)$, which is the probability of the data independent of the word string, does not depend on W so we can simplify this further:

$$P(W|Y) \propto \underbrace{P(Y|W)}_{\text{acoustic model}} \underbrace{P(W)}_{\text{language model}} \tag{4.4}$$

Acoustic Model $P(Y|W)$ is the probability of a model of the word string (the *utterance model*) generating the acoustics Y. The utterance model will usually be an HMM (or some other kind of stochastic finite state machine). This model is estimated from a speech corpus.

Language Model $P(W)$ is the prior probability of a sequence of words. Since this probability does not depend on the acoustic data Y it can be estimated from a large text corpus.

4.2 Hierarchical Modelling

If the utterance model W is represented by an HMM, then the acoustic model probability $P(Y|W)$ is exactly that obtained when the evaluation problem is solved for HMMs. However in large vocabulary continuous speech recognition we may have a vocabulary of 60,000 words, and utterances may be any length. It is clearly impossible to build a separate, independent model for each utterance.

The solution is to model particular, well-defined units of speech. A separate HMM is constructed for each unit, an utterance model can then be defined by concatenating the basic HMMs (figure 4.1). The same basic HMM can be used in any number of different utterance models. During training we adjust the parameters of each utterance model in the training data: but since each utterance model is built from simpler models, this means that the parameters of the basic models are adjusted. For example, the word could be chosen as the basic unit of speech. In this case an utterance model of a string of words would consist of the concatenation of the component word models (if a word is repeated then that component HMM just gets inserted into the utterance model twice.

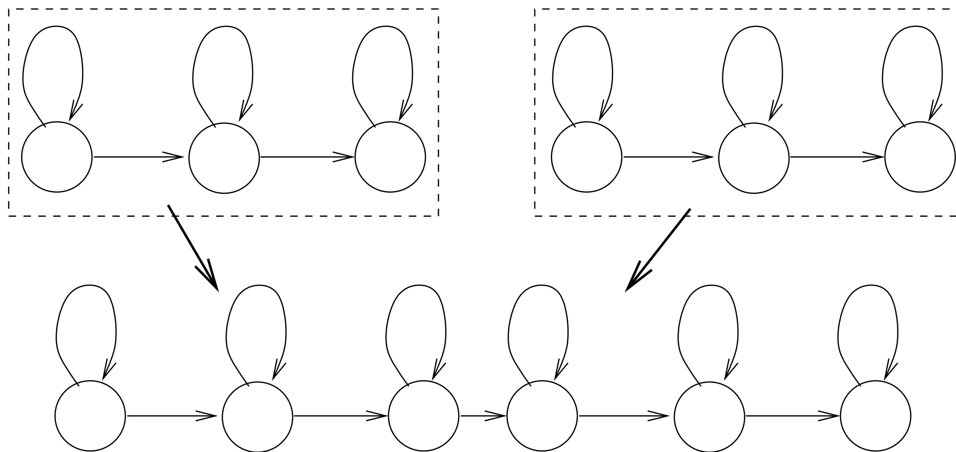

Figure 4.1: Concatenation of two HMMs

If we are interested in building a recognizer with a very limited vocabulary (say a recognizer for the digits 'zero' to 'nine') then choosing the word as the basic unit of speech might be a reasonable approach. However, this approach is not feasible with larger vocabularies, in which some words may occur only rarely (or not at all) in the training data. In this case, there would not be sufficient training data to estimate the parameters for individual HMMs for each word.

A solution to this problem is to define a basic set of *subword units*. Subword units may be put together to form words. The pros and cons of various units of speech were discussed in the context of waveform synthesis and most of the same arguments hold. or diphones as the basic subword unit, most systems are *phone* based. In British English there are about 50 phones which may be used to build pronunciations of any word in the language. However, the acoustic realization of phones can vary tremendously, depending on their phonetic context: to cope with this variation, *context-dependent* phone models are often employed.

If we use a system based on subword units:

- Word models are built from subword models, with a *pronunciation dictionary* defining how to concatenate the subword models;
- Utterance models are built by concatenating these word models.

4.3 Training

If our continuous speech recognition system is based on subword HMMs, then the parameters of the models (transition probabilities and output probabilities) are estimated from the training data using the forward-backward algorithm (also known as the Baum-Welch algorithm) previously described (part 3). To train these models we need the following:

- A set of basic subword HMMs (typically a three-state left-to-right topology is used for each model (as in figures 3 and 4 in part 2 Finite State Models).
- A pronunciation dictionary giving a pronunciation for each word in the vocabulary in terms of the subword units. Phone-based systems for British English often use a the British English Example Pronunciation (BEEP) dictionary, based on the Oxford Advanced Learners Dictionary.
- Training data: the acoustics for each utterance (processed using an appropriate front-end, e.g., mel scaled cepstral coefficients) and vector quantized, together with the *orthographic transcription*. The transcription is simply the word sequence for that utterance: it does not need to be time-aligned and does not need to be phonetically labelled.

The following procedure can be used to train the subword models from continuous speech:

- Initialize the subword HMMs (see below)
- For each training utterance create an utterance model: each word model is a sequence of subword HMMs, concatenated using the pronunciation dictionary. The utterance model is the concatenated sequence of word models.
- When building the utterance models optional silence models can be placed in between words.
- The forward-backward procedure is then applied to each large concatenated utterance HMM. The forward and backward probabilities are computed using equations (7) and (12), the output probabilities are re-estimated using (14) and (16), and the transition probabilities re-estimated using (14), (15) and (17). (All equation numbers refer to part 3 notes (HMMs).)
- The forward-backward procedure is then iterated several times

Note:

- Word boundaries are not estimated — the procedure has complete freedom to align the utterance model against the acoustics
- Training each utterance model results in adjusting the parameters of the baseform HMMs and so will affect other utterance models. We can think of the utterance model being a concatenated sequence of pointers to baseform HMMs
- We make assumptions about how each word is pronounced when training (supplied by the dictionary), which may not correspond to the actual acoustics. But since these inaccurate assumptions are the same at both training and testing, we are being consistent.
- This is a re-estimation procedure — our existing models are taken to be the best current speech model, the forward-backward procedure then improves these models.

There are two ways in which the baseform HMMs

1. Use a phonetically labelled and time-aligned database (e.g. TIMIT) to initialize the parameters for each subword model, using the forward-backward algorithm. This is a straightforward procedure, since the speech is segmented into phones (or other relevant subword units), so each basic subword HMM has a clearly defined set of training data.

2. "Flat-Start" training. All models start identically (or with random parameters, ensuring sum-to-one constraints are not violated). This is clearly not a good initial model, but after one or two iterations of the forward-backward algorithm over the training data, the models will have "self-organized".

With flat start training a continuous speech recognition system can be trained without using any time-aligned or phonetic labelling whatsoever — all that is required is the sequence of words corresponding to the acoustic signal.

4.4 Evaluation

It is important to be able evaluate speech recognizers—both to compare one speech recognizer with another and to be able to quantitatively assess the effects of any changes made to the system. Evaluation is performed on an independent *test set* of speech data. There are three classes of speech data set used in training and evaluating a speech recognizer:

- **Training** set — the data set used to train the parameters of the system (HMM transition, probabilities, VQ codebook, etc.)

- **Development** set — the data set used for incremental development, etc. of the recognizer (e.g. comparing different front ends).

- **Test** set — test data used to evaluate the the recognizer: this data should never have been used with the recognizer before.

The difference between development and test data is very important - if every potential improvement is evaluated on the same test set that is used for the final evaluation, then you run the risk of 'tuning' the recognizer to that data.

There are three types of error that a continuous speech recognizer can make: substitution, insertion and deletion. Consider recognizing the utterance "computer science department"

- Substitution: e.g. "computer SIGHS department"

- Insertion: e.g. "computer science THE department"

- Deletion: e.g "computer department"

The number of substitutions (S), deletions (D) and insertions (I) is obtained by aligning the recognizers output with the actual utterance.

Note that the % correct doesn't tell you everything:

- "computer science department" - 100% correct

- "the computer seen science the department to" - 100% correct

The %ge of words recognized correctly doesn't include insertions. In fact:

$$\% \text{ correct} = 100 \times \left(1 - \frac{(S+D)}{N}\right)$$

Where N is the number of words in the correct transcription. The measure that is usually employed is the *word error rate* (%) (or equivalently the **recognition rate**):

$$\text{word-error-rate} = 100 \times \left(\frac{(S + D + I)}{N} \right)$$

$$\text{recognition-rate} = 100 - \text{word-error-rate}$$

It is quite possible to have a recognition rate of $< 0\%$ (or a word error rate of over 100%), due to insertions. “the computer seen science the department to” may be 100% correct, but it has a word error rate of $100 \times (0 + 0 + 4)/3 = 133.3\%$, giving a recognition rate of -33.3%!

The word error rate is a measure of the transcription accuracy of the recognizer and gives a basic measure of the performance of the recognizer (relative to a test set). However if the recognizer is being used for a specific task, then the relevant performance measure is one that measures how well it performs. For example, a recognizer that is being used as a user interface to an air travel information system should be evaluated on the % of correct data base queries generated from the speech input, not on the word error rate — a user doesn’t care about the internal transcription they just want a sensible answer to their questions.

Chapter 5

ACOUSTIC MODELLING

5.1 Modelling Context

To build a high-performance continuous speech recognition system using a set of subword units, it is important that the subword units are able to accurately represent their acoustic classes. To do this we want to reduce the variability of each acoustic class, thus increasing the potential accuracy of the subword models. Much of the variability in speech is due to coarticulation and other contextual effects. If the subword units explicitly model these contextual effects then the overall accuracy of the system should be increased.

There are various causes of contextual variation. Some are due to the environment (is it noisy? are there multiple speakers?), the channel (telephone? wide-band microphone?), the speaker (male or female? age?) or the speaking style (rate? dictated or spontaneous?). These effects are extremely important and modelling them is an active research issue, but here we shall be more concerned with *local* contextual effects: principally *coarticulation*, but also stress and emphasis.

5.1.1 Coarticulation

Since speech is produced by relatively slowly moving articulators, the articulatory movements leading to the production of a particular phone is strongly influenced by both the preceding and following phones. If we consider each phone to have an "articulatory target" then in continuous speech the articulators do not make discrete jumps from one target to another, but are in motion moving from one target to the next. Indeed, especially during fluent speech, the articulatory targets will not be achieved, and the slowly moving articulators will always be in a state of transition. This context dependence will also be reflected in the acoustic realization of the phone. We call this phenomenon coarticulation. Coarticulation means that the acoustics of a phone may be reasonably consistent when the surrounding phonetic context is taken into account, compared with when a single model is used for the phone for all contexts.

Humans deal with coarticulation effortlessly. However statistical speech recognizers need to be told about it!

5.1.2 Context-Dependent Phone Models

The most usual way in which coarticulation effects are modelled is through the use of *context-dependent* phone models. Put simply, this involves using several basic subword units for each phone, rather than just one. This technique assumes that the preceding and succeeding articulatory targets may be predicted by the surrounding phones. (Of course these predictions are not explicit—we just assume that by doing this the acoustic variability in each subword unit is reduced.) The amount of context used can vary:

Monophone context This is the simple case in which surrounding phonetic context is not used. In British English (using the BEEP dictionary) this means that a total of 45 phones (plus silence) are required. The word "SPEECH" would be represented by the model sequence:

SPEECH = `s p iy ch`

Allophone context This is similar to monophone context (i.e., no context), but there are multiple possible units for each phone. Using the same example:

SPEECH = `s(1) p(3) iy(2) ch(1)`

Biphone context In this case each phone model is represented with a particular left (or right) context. Potentially there are $45 \times 46 = 2,070$ left (or right) biphone context phone models (plus silence).

SPEECH = `$-s s-p p-iy iy-ch` (left biphones) or
SPEECH = `s+p p+iy iy+ch ch+$` (right biphones)

(Note that "$" is the symbol for a word boundary context, and that "-" signifies a left context, and "+" a right context.)

Triphone context Each phone model is represented with a particular left and right context. There are a possible $45 \times 46 \times 46 = 95,220$ triphone context dependent models (plus silence).

SPEECH = `$-s+p s-p+iy p-iy+ch iy-ch+$`

Word Context In this case each phone model depends on the context of the surrounding word.

SPEECH = `s(speech) p(speech) iy(speech) ch(speech)`

The number of possible context-dependent units increases exponentially with the amount of context. However not all contexts occur in natural speech and it is not necessary in practice to construct a model for each possible context. The most usual context dependent subword unit employed in modern continuous speech recognition systems is the triphone context dependent phone model.

Since we are dealing with continuous speech we need to consider what to do at word boundaries. There are two basic possibilities:

- **Word internal** context dependence: phonetic context across word boundaries is not used:

  SPEECH PROCESSING = `$-s+p s-p+iy p-iy+ch iy-ch+$ $-p+r p-r+o o-s+eh eh-s+ix s-ix+ng ix-ng+$`

- **Cross word** context dependence: phonetic context is considered across word boundaries:

  SPEECH PROCESSING = `$-s+p s-p+iy p-iy+ch iy-ch+p ch-p+r p-r+o o-s+eh eh-s+ix s-ix+ng ix-ng+$`

In continuous speech recognition word boundaries are not delimited by silences, so cross-word coarticulation is important. However there are two reasons why it simpler to implement a system using word-internal context dependency:

Size There are many more contexts that can occur in the cross-word case compared with the word-internal case (e.g. `ch-p+r` above). In the cross-word case that means that every word in the dictionary needs start (and end) context-dependent phone models for all possible preceding (succeeding) words. For example in the case of a 20,000 word dictionary for a business news task, there were around 14,000 possible word-internal triphone context models, compared with 54,000 possible cross-word triphone context models.

Complexity Using word-internal context dependent models means that each word can be considered in isolation; cross-word models means that the acoustic model for a word cannot be constructed in advance by concatenating the relevant subword models, since the word model is dependent on the preceding and succeeding words. This makes the search (decoding) task more complex for cross-word context dependent systems.

In the business news task mentioned above although there are over 54,000 possible cross-word triphone context dependent phones, only around 23,000 cross-word triphone context dependent phones appear in a standard training data set, consisting of around 10 hours of speech. There is no way that it is going to be possible to estimate the parameters for 54,000 context dependent models, when over half of them do not appear in the training data. Indeed, it is generally reckoned that the minimum number of examples needed to estimate the parameters of a particular context-dependent subword models is between 10 and 1,000 (depending on the exact nature of the subword HMM). However, it is not acceptable to not model contexts that do not occur in the training data, since we must be prepared for all possible contexts occurring in the training data.

Two contrasting solutions to this problem are:

Backing-off In this method, if not enough data exists to model a triphone context dependent model, then a less specific (left or right) biphone context dependent model is used instead. If there is not enough data for either biphone context dependent model, then we back-off further to the context independent monophone model. This approach guarantees that well-trained models will always be used. However, it can mean that relatively few models will using full triphone context when data is relatively sparse (typically the case for cross-word models), so some of the potential benefits of triphone context dependence will be lost (e.g. relatively few cross-words will be modelled by triphone context dependent models).

Sharing Rather than backing-off to less specific models, try to ensure that all (or most) phone models are fully context dependent by sharing models between different contexts. This is sensible since although acoustic realisations of phones do vary with context, every context does not result in a completely different acoustic realisation to every other context. This approach means that context-dependency is maintained, while each model has “enough” training data. The possible drawback is that inappropriate shareing can lead to models with high variability (less specific models).

5.1.3 Shared Context-Dependent Models

We shall look at three basic approaches to shared context-dependent phone models. The first uses “linguistic knowledge”, the other two are data-driven.

Broad Phonetic Classes

In this case we use linguistic knowledge to group possible contexts into “broad” phonetic classes, e.g.:

1. Stops (e.g. `b d`)

2. Nasals (e.g. `em ng`)
3. Fricatives (e.g. `ch z`)
4. Liquids (e.g. `l w`)
5. Front Vowels (e.g. `ae ix`)
6. Central Vowels (e.g. `ah er`)
7. Back Vowels (e.g. `ow uh`)

In this case there will be $45 \times 7 \times 7 = 2205$ broad class triphone context dependent phone models (plus silence). This is a simple procedure, but it assumes that these broad phonetic classes do specify the articulatory context well. In practice it turns out that this assumption does not hold very well, resulting in subword units of less specificity attempting to model a good deal of acoustic variability.

Generalised Triphones

This is a "bottom up" data driven approach. This approach begins by assigning a triphone context dependent model to all phones in all observed contexts. Of course, not all of these models will have enough training data. The algorithm proceeds by merging the "closest" models (according to some distance measure). This pairwise merging process continues until all the resultant models have "enough" training data (accoriding to some criterion). These are known as *generalised triphones*. If acoustic realisations of phones in different contexts are similar, then it wil result in a reduced set of accurate models.

The principal drawback of this approach is that training examples are required for all initial models — this is common to all bottom up merging procedures. This makes it impossible to model contexts unseen in the training data. When unseen contexts are encountgered at recognition time a backing-off approach must be used.

Decision Trees

This is a "top down" data driven approach. In this case we start with a set of context-independent models and start a spitting procedure. The key point is that the data is used to specify which models should be split. An approach to doing this uses a *decision tree*. The root of the tree is a context independent model. Each node of the tree contains a question about the context (e.g. "is the left context a nasal" "is the right context `l`"), used to split the current model. The leaves of the tree correspond to the resultant context-dependent models. The advantage of this approach is that all possible contexts will be modelled by a context-dependent model, with a specificity determined by the training data.

The key issue is how the tree is constructed — which questions are asked at each node? An automatic algorithm is used which cycles through all possible questions ate each node and chooses the best (if any) to split the model at the node. Without going into the details of algorithms that are used for phonetic decision tree construction the key constraints are:

- Each leaf (context-dependent model) must have a minimum number of training examples
- A finite set of questions must be chosen from to split each node
- The resultant leaves must be able to be well modelled by HMMs

5.2 Continuous Density HMMs

Recall the following two facts:

1. After signal processing, we represent a speech signal as a sequence of real-valued vectors (e.g. a vector of mel frequency cepstral coefficients).
2. Our speech modelling machinary, based on stochastic finite state machines, operate using discrete symbols (e.g. the output functions of a finite state generator produce discrete symbols).

So far we have used vector quantization to resolve this conflict — a stream of continuous valued speech feature vectors is transformed into a string of VQ codewords. However, this process must lose some information in the speech signal (do you see why?) and some of the lost information might be helpful for speech recognition. Rather than messing with the data to fit the model, can we mess with the model to fit the data? Explicitly, can we adapt the stochastic finite state generator model to have an output function that produces continuous valued vectors rather than discrete symbols? We can, by defining *continuous density* HMMs.

5.2.1 Gaussian Output Functions

The output function for a state of a discrete symbol HMMs can be regarded as a histogram — each symbol has a probability of emitted by that state. If we are to use HMMs as a generative model of continuous vectors, then we require an output function that can generate continuous vectors with a particular probability. Unlike discrete symbols we can't assign a particular probability to each possible continuous vector. The solution is to use a *probability density function*, for example a Gaussian (or Normal) Distribution. A Gaussian is defined by a mean vector, and a variance, which measures the average (squared) deviation from the mean (or the width of the Gaussian). In one dimension it is the familiar bell-shaped curve (figure 5.1), and it may be generalised to multiple dimensions:

$$p(\mathbf{y}(t)|q_t) = K \exp\left(\sum_k \frac{(y_k(t) - \mu_k(q_t))^2}{2\sigma_k}\right) \tag{5.1}$$

$$= N(\mathbf{y}(t); \mu, \sigma) \tag{5.2}$$

The area under a Gaussian (and, indeed, any probability density function) will sum to one. A particular point on this curve is the *likelihood* of the corresponding vector being generated.

We can use this idea to construct continuous output stochastic finite state generators. Instead of having a histogram over the K possible VQ codewords, we use a d-dimensional Gaussian, where d is the dimension of the feature vectors. Instead of reading off the probability of a VQ codeword from the histogram, in the continuous case we read off the likelihood of generating the feature vector from the Gaussian probability density function.

This means we have different parameters to estimate. Instead of the K symbol probabilities $P(y(t) = i|q)$ defining the output function, the continuous valued output function is defined by the parameters of the Gaussian: the mean μ and the variance Σ. Fortunately it turns out that we can still use the forward-backward algorithm to re-estimate the means and the variances. Basically, the means are reestimated as the sample means of the vectors assigned to a particular Gaussian (state) (weighted by the probability of assignment).

5.2.2 Gaussian Mixture Models

The problem with using a Gaussian output distribution is that it assumes that the data assigned to that state must fit a Gaussian. In particular it assumes the output distribution is

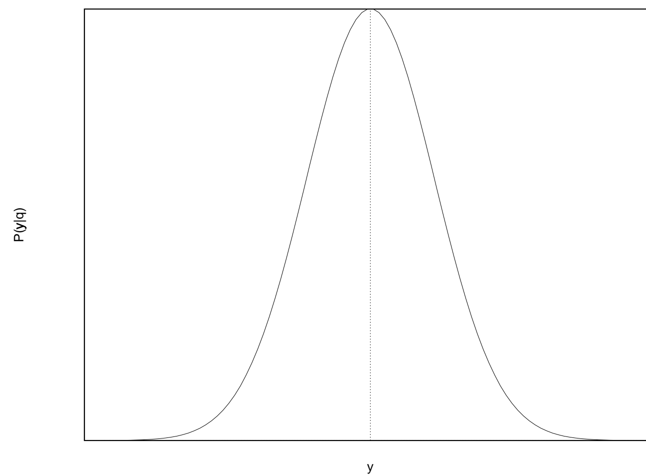

Figure 5.1: A one-dimensional Gaussian (Normal) Density Function

unimodal (i.e. there is a single peak). This is often a very bad assumption — for exampe, if the function has two humps, then it may be very poorly modelled by a Gaussian (see figure 5.2).

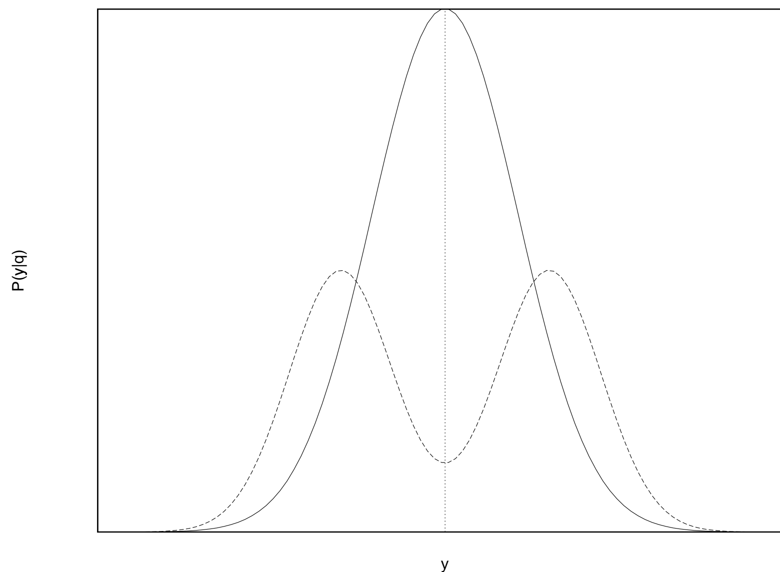

Figure 5.2: A probability density made up of two Gaussians. Fitting a Gaussian to this data results in a distribution whose mean is at a point where there is very little data.

What should we do? We want to use a continuous output function, and Gaussians are very convenient since we can estimate their means and variances using the forward-backward algorithm. A solution is to use a combination or *mixture* of Gaussians for the output function. A mixture of Gaussians can have as many component Gaussians as desired — some modern continuous speech recognition systems use mixtures of up to 32 Gaussians

per state. A Gaussian mixture model is given by:

$$p(\mathbf{x}|q) = \sum_i a_i N(\mathbf{x}; \mu_i, \sigma_i) \tag{5.3}$$

where the a_i are called the mixture coefficients, and sum to one. Fortunately we can use the forward-backward algorithm to train the means and variances of all the Gaussians, the Gaussian mixture coefficients and the HMM transition probabilities simultaneously. In "regular" HMM training we estimate the probability that a particular state generated a particular feature vector at a particular time; with a Gaussian mixture output function we further estimate the probability that a particular component of the Gaussian mixture was responsible for generating the feature vector.

Most state of the art continuous speech recognition systems are based on Gaussian Mixture output functions.

5.3 Parameter Tying

There is a continual tension between the search for more accurate models — which can mean using more parameters per model (e.g, more states per model, more components per mixture) or a greater number of basic subword models — and the need to estimate the model parameters from limited training data. One way to increase model accuracy without increasing the intrinsic model complexity is through models sharing parameters. This is sometimes described as parameters being tied between models. Parameter tying can happen at many levels. In a Gaussian mixture based systems parameter tying may occur at the level of:

- means
- covariances
- mixture coefficients
- mixture components
- distributions
- states
- transition matrices

Examples of parameter tying in current systems include:

Grand Covariance Matrix Covariance parameters can be the most difficult to estimate directly. And since a covariance matrix has $O(d^2)$ independent elements, there is a potentially huge number of parameters in a complex system. One way around this is to constrain all the Gaussians to use the same covariance matrix, typically computed over all the training data — the so-called Grand Covariance matrix.

Tied Mixtures In the tied mixture model, all the output functions share the same basic set of component Gaussians, with each output function having it's own set of mixture coefficients. This is like a "soft" vector quantization (with the set of Gaussians being analogous to the codebook) and is sometimes known as the Semi-Continuous HMM.

State Clustering In a state-clustered system different subword models can share the same states. For example, consider the phone `a` in a triphone context dependent HMM. Let `k-a+t` be the model for `a` in left context `k` and right context `t`, and `r-a+t` be the model for `a` in left context `r` and right context `t`. These models may be quite different (the left context is very different), but may have a similar final state. In this

case, the state clustering (also known as tied state) approach will cluster (tie) the final states, i.e., constrain them to be equal. The training data is used to determine which states should be tied.

Chapter 6

LANGUAGE MODELLING AND SEARCH

6.1 Language Modelling

Language modelling is essential to reduce the complexity of the recognition task. Speech recognition is dealing with a natural language, and in any particular context, some words are more likely to appear than others. In the statistical framework for speech recognition, the language component arises naturally when we decompose the problem of evaluating the probability $P(W|X)$ of a word sequence given the acoustics into two parts, the acoustic model and the language model:

$$P(W|Y) \propto \underbrace{P(Y|W)}_{\text{acoustic model}} \underbrace{P(W)}_{\text{language model}} . \tag{6.1}$$

The *language model* $P(W)$ is the prior probability of a sequence of words. Since this probability does not depend on the acoustic data Y it can be estimated from a large text corpus, and does not require (acoustic) speech data.

So what form does our knowledge about the language take?

We could follow the AI/linguistics NLP route and construct elaborate context-free grammars (or similar) designed to cover spoken language. This approach is problematic:

- It is very difficult to automatically acquire a grammar from data, and it is extremely labour-intensive to develop one by hand;
- Non-probabilistic grammars mark word stings as "grammatical" or "not grammatical": therefore some word strings cannot be recognized.

The current state-of-the-art approach to language modelling is, in some senses, much cruder. The model of language that is used is the n-gram. An n-gram is simply a sequence of n words, $w_1, w_2, \ldots, w_n$. In n-gram modelling of language the building blocks of our model are probabilities of the form $p(w_n|w_{n-1}, \ldots, w_1)$: the probability of a word given the previous $n-1$ words. These arise due to the fact that we can decompose (without assumptions) the probability of a word string as:

$$P(w_1, w_2, w_3 \ldots, w_{\ell-1}, w_\ell) = P(w_1)P(w_2|w_1)P(w_3|w_2, w_1) \ldots P(w_\ell|w_{\ell-1}, w_{\ell-2}, \ldots, w_2, w_1) \tag{6.2}$$

In the n-gram model we limit the context used in evaluating $P(w_1, w_2, \ldots, w_n)$ to $n-1$ words, i.e. for a 3-gram (*trigram*):

$$P_{\text{trigram}}(w_1, w_2, w_3 \ldots, w_{\ell-1}, w_\ell) = P(w_1)P(w_2|w_1)P(w_3|w_2, w_1)P(w_4|w_3, w_2) \ldots P(w_\ell|w_{\ell-1}, w_{\ell-2}) . \tag{6.3}$$

If we have a vocabulary of N words, then a table of N^n n-gram probabilities completely specifies the language model and may be used to compute the language model probability of any string of words (in the vocabulary). n typically has the values 2 (bigrams) or 3 (trigrams).

A powerful aspect of n-gram modelling is the fact that n-gram probabilities can be directly and straightforwardly estimated from data. (Since language models do not depend on the acoustics, a text corpus may be used to estimate language model probabilities.) The estimation process is simple. For trigrams it is:

1. Count all word triples appearing in the text: $c(w_i, w_j, w_k)$

2. Estimate trigram probabilities as follows:

$$p(w_k|w_i, w_j) = \frac{P(w_i, w_j, w_k)}{P(w_i, w_j)} \tag{6.4}$$

$$\approx \frac{c(w_i, w_j, w_k)}{\sum_{w_\ell} c(w_i, w_j, w_\ell)} \tag{6.5}$$

This naive approach has the problem that n-grams that do not occur in the training set will be given a zero probability. This means that word triplets that do not occur in the training set cannot be recognized. We don't want this to happen — no sequence of words should be given a zero probability: if the acoustics are strong enough evidence for a word sequence, then the language model should not disallow it.

One solution to the zero probability is known as the "back-off". In this approach, the when estimating trigram probabilities based on (6.5) a certain amount of probability is with held for "unseen n-grams" not in the training data. If a trigram is not seen in the training data it's probability estimate is based on this reserved probability mass for unseen trigrams, which is then split up using bigram probabilities. The same process happens when estimating bigrams, with the reserved probability mass being split up according to unigram probabilities.

There are several problems with using n-gram language models, for example:

1. They assume that all the contextual information is encoded in the previous n words;

2. It is difficult to automatically adapt them to a particular topic or task

3. They don't use any notion of word classes or categories

However, they have proven to be extremely effective, since they can be estimated from very large databases. Modern systems use language models with over 10 million probabilities, estimated from several hundred million words of text.

6.2 Search

The search problem in large vocabulary continuous speech recognition (LVCSR) can be simply stated: find the most probable sequence of words given a sequence of acoustic observations (and given an acoustic model and a language model). This is a demanding problem since word boundary information is not available in continuous speech: thus each of the words in the dictionary may be hypothesized to start at each frame of acoustic data. The problem is further complicated by the vocabulary size (typically 20,000 words or larger) and the structure on the search space imposed by the language model.

Using the statistical framework, we can express the goal of the speech recognizer as finding the word sequence $\hat{W}$ with the maximum probability given the acoustic observations

Y:

$$\hat{W} = \underset{W}{\operatorname{argmax}} P(W|Y) \tag{6.6}$$

$$= \underset{W}{\operatorname{argmax}} \left(\frac{p(Y|W)P(W)}{p(Y)} \right) \tag{6.7}$$

$$\propto \underset{W}{\operatorname{argmax}} P(Y|W)P(W) \tag{6.8}$$

The task for the search algorithm is to evaluate (6.6) — i.e., determine $\hat{W}$ given the various models and the acoustic data. Direct evaluation of all the possible word sequences is impossible (given the large vocabulary) and an efficient search algorithm will consider only a very small subset of all possible utterance models. This is a difficult problem, since word boundaries are not known at recognition time—so if there is a 60,000 word vocabulary, then a potential 60,000 words can start each frame.

6.2.1 Decoding Strategies

Two basic decoding (search) strategies are used for continuous speech recognition:

Viterbi decoding is based on the dynamic programming algorithm and is similar to the DP algorithm for connected word recognition covered in part 1 of these notes. This is a *time-synchronous* algorithm — this means that the search progresses frame by frame, forward in time.

Stack decoding algorithms are based on the ideas of heuristic search. They are *time asynchronous*: the best scoring path or hypothesis, irrespective of time, is chosen for extension and this process is continued until a complete hypothesis is determined.

Without pruning Viterbi decoding is an exhaustive search—the most probable word sequence will be found. However an unpruned search for large vocabulary continuous speech recognition is computationally infeasible. Stack decoding is a heuristic search — the key is choosing the most probable hypothesis: the function used to choose this not only consider the probability of the partial hypothesis or path, but also must estimate the probability of the hypothesis explaining the rest of the acoustic sequence.

Although the set of basic units in a LVCSR system will be some kind of subword unit, the basic unit of the search is the word model (which is constructed as a sequence of subword models). This is because:

- The pronunciation dictionary (which is used to define the word models) defines which sequences of subword units are allowable. (This is in contrast to the language model which assigns probabilities to word sequences, but all word sequences are allowable with a certain probability.)
- Because of the language model effect the same subword units at the same time will have different path probabilities.

6.2.2 Viterbi Decoding

The Viterbi decoding algorithm is based on the solution to the Evaluation Problem for HMMs (discussed in part 3). This specifies the probability of the most probable state sequence; by storing an array of backpointers (as used in the DTW algorithm for connected word recognition) the corresponding state sequence and/or word sequence can be found by backtracing.

Figure 6.1 shows how the word models are constructed.

To incorporate bigram language model probabilities is straightforward: see figure 6.2.

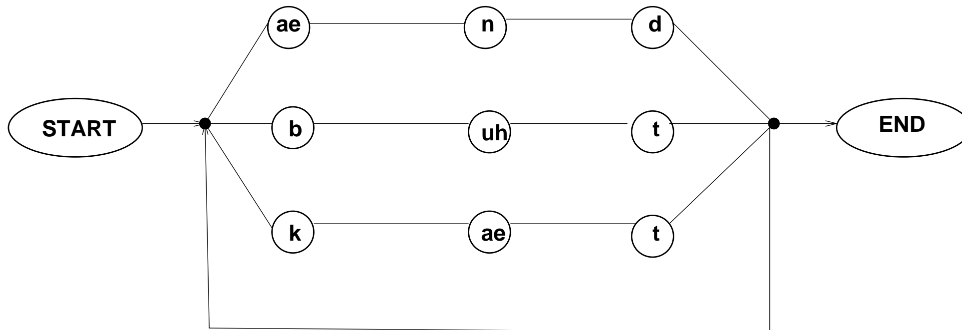

Figure 6.1: Basic Model for Viterbi Decoding for Continuous Speech Recognition

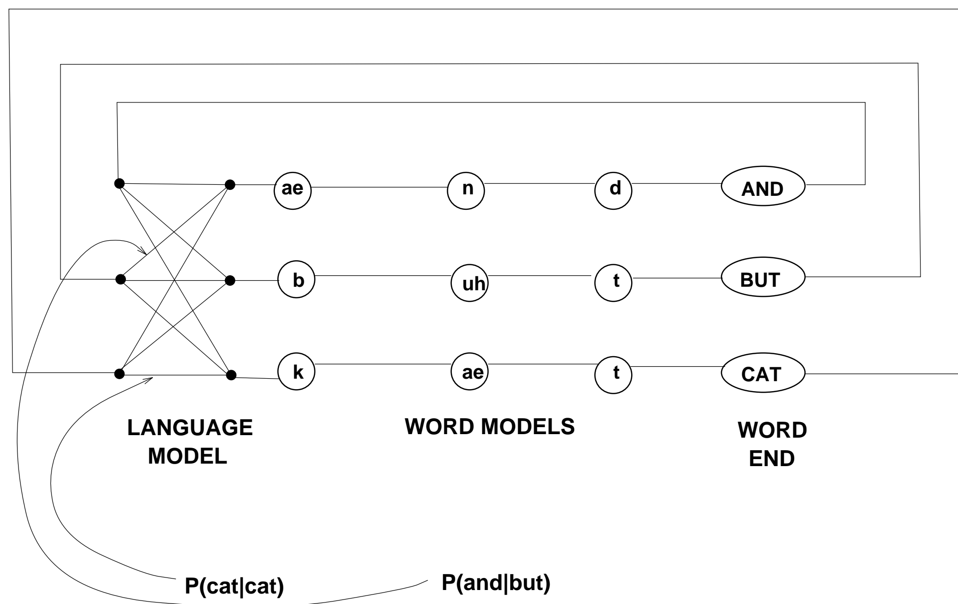

Figure 6.2: Decoding Scheme for a Bigram Language Model

However, using a trigram language model is more complex and involves changes to the dynamic programming algorithm. In dynamic programming, computation is saved when paths converge on the same state — the principal of optimality says that we need only consider the path with the highest probability (the *dominant path*. However with a trigram a language model we cannot do this — the probability of a path depends on the previous two words, not just the previous word. This means that a path is only dominant over paths with the same language model context. In effect, using a trigram language model will result in a search space with many more states.

6.2.3 Pruning

Effective pruning techniques are essential for an efficient search. These are based on beam search:

- Paths with a probability that is more than a factor δ less than the best path at time t are pruned;
- A limit is put on the maximum number of any paths at any time.

6.2.4 Tree-structured Dictionary

A simple search will then consist of a set of individual word models, each constructed from the constituent (shared) subword units. However, some computation can be shared by noting some words share beginnings (see figure 6.3).

6.2.5 Stack Decoding

Stack decoding algorithms are time asynchronous — the best scoring path or hypothesis, irrespective of time, is chosen for extension and this process is continued until a complete hypothesis is determined. The crucial function for these algorithms, $f_h(t)$, is the estimated score of hypothesis h at time t:

$$f_h(t) = a_h(t) + b_h^*(t) \ , \tag{6.9}$$

where $a_h(t)$ is the score of the partial hypothesis using information to time t and $b_h^*(t)$ estimates the best possible score (minimum cost) in extending the partial hypothesis to a valid complete hypothesis. $a_h(t)$ is the score (log probability) of h at t, so to estimate $f_h(t)$ we need to estimate $b_h^*(t)$. It may be shown that as long as $b_h^*(t)$ is an upper bound on the actual score (i.e., it never underestimates the probability) then the search algorithm will be *admissible*: no errors will be introduced that would not occur if an exhaustive search was performed.

Stack decoding is a best-first algorithm. The most probable (partial) hypothesis (i.e., the hypothesis for which $f_h(t)$ is greatest) from the "stack" of hypotheses under consideration is popped and extended by a word, with newly created extended hypotheses being added to the stack. Provided the estimate of $f_h(t)$ is admissible, the first complete hypothesis to be popped from the stack will corresponds to the most probable utterance model.

The data structure used for the stack is a *priority queue*, typically implemented as a heap. This is an efficient data structure which is appropriate for accessing and inserting data, while maintaining a sorted order.

The key aspect of a stack decoder is how $b_h^*(t)$ is estimated. This is the probability of any word sequence completing the path.

Various approximations have been tried. Most require some kind of lookahead, usually using simpler acoustic and/or language models (often known as a *fast match*). An alternative approach, which is discussed here, avoids looking ahead ordering hypotheses primarily by reference time t_h and secondarily by the probability. Hypotheses with earlier reference

time are extended first. This is equivalent to using multiple stacks, one for each reference time, and processing the stacks in order of increasing reference time. As extended hypotheses are generated, with an increased reference time, they are inserted into the appropriate stack. This ordering ensures that hypotheses are compared against other hypotheses with the same reference time, thus reducing the need for lookahead.

The pruning strategies used for stack decoding are basically the same as for Viterbi decoding. Also tree-structuring the dictionary can be done when using a stack decoder.

Stack decoding has several potential advantages over Viterbi decoding:

- The language model is decoupled from the acoustic model and is not used to generate new recognition hypotheses;
- It is easy to incorporate non-Markovian knowledge sources (e.g., long-span LMs) without massively expanding the state space;
- It is a heuristic search rather than an exhaustive search (unlike the Viterbi algorithm, without pruning), and thus need not explore the full state space;
- The Viterbi assumption is not embedded in the search and thus a full maximum likelihood search criterion may be used with little or no computational overhead.

6.2.6 Assessing Search Algorithms

Search algorithms may be assessed using four basic criteria:

Efficiency This is a measure of the amount of computation (and memory use) required by the search algorithm — and can be highly implementation dependent. Platform-independent measures of efficiency include average number of HMM states evaluated per frame, and number of language model lookups.

Accuracy As we make approximations (e.g. in pruning) to ensure an efficient decoding, there is no longer a guarantee that the most probable hypothesis will be found. Thus there is a tradeoff between accuracy and efficiency. The accuracy of the search algorithm (as opposed to the acoustic and language modelling accuracy) is measured by the *search error* — this is a measure of the extra errors caused by the approximations made in the search algorithm.

Scalability This is a measure of how the decoder performance scales with increasing vocabulary size. Although accuracy will not rise linearly with vocabulary size, as the extra words will be relatively uncommon; for the same reason a search algorithm that scales well will have a computation that scales slower than linear.

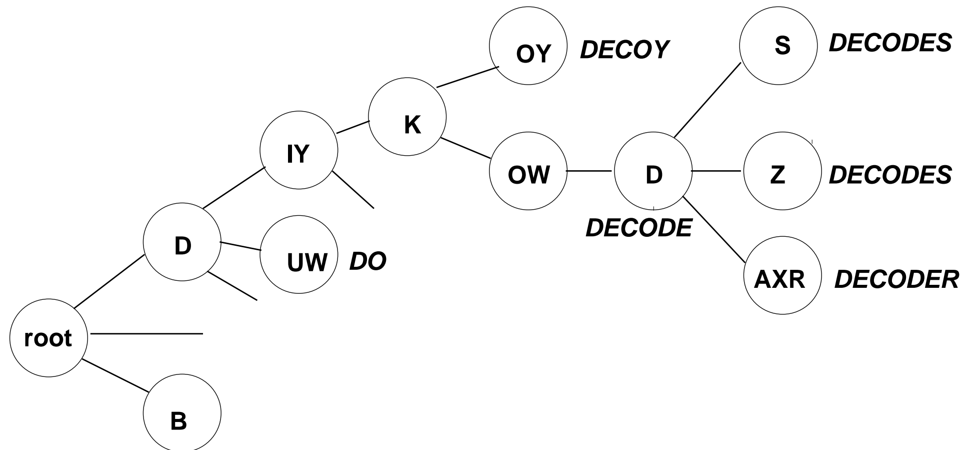

Figure 6.3: A pronunciation prefix tree — words whose pronunciations have similar beginnings can share computation..

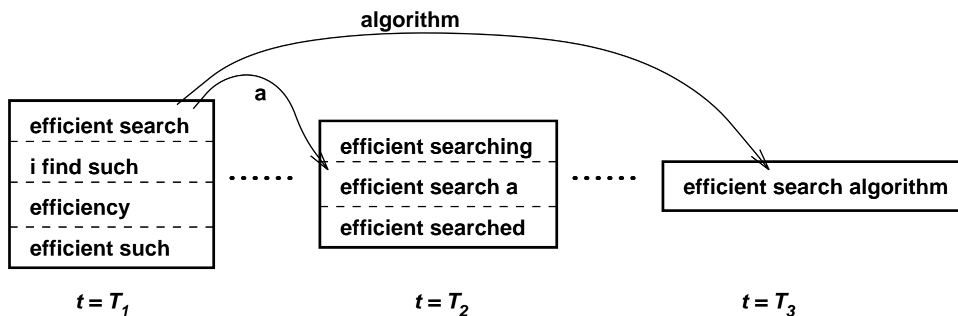

Figure 6.4: Illustration of the start synchronous search strategy. The stack at time $t = T_1$ is being processed. The most likely hypothesis at this time ("efficient search") is extended by the most probable one word extensions ("a" and "algorithm" are illustrated). The resultant extended hypotheses are inserted into the stack at their reference time — in this case "algorithm" has duration $T_3 - T_1$. In practice all hypotheses with identical reference times are extended in parallel.