The Beginner’s Guide to the ICSI Speech Software

Dan Jurafsky and the ICSI Speech, Dining, Dancing and Salsa Society

July 12, 1995

1 Introduction

So you want to be an ICSI speech engineer? Does your mother know about this? Well, if you’re really set on it, and nothing will dissuade you, this document is intended to give you a quick introduction to our fundamental assumptions and to our speech recognition software packages. This manual is very much a work-in-progress, please mail relevant suggestions, corrections, and amusing anecdotes to Dan.

2 Understanding the Pieces

2.1 Tasks and Databases

OK, now that that’s out of the way, let’s begin by exploring the task. At ICSI we do both speech recognition and speech understanding. We use the term speech understanding to refer to the problem of mapping acoustic signals to some representation of meaning. The term speech recognition refers to a sub-part of this problem: mapping from acoustic signals to strings of words. The Berkeley Restaurant Project task, in which people ask spoken-language questions about Berkeley Restaurants and the system answers them, is a speech understanding system. All the rest of our tasks are speech recognition tasks.

In an orthogonal classification, we classify the task according to the nature of the input. In isolated word recognition, the input comes to us already broken up into words. The simplest isolated word task that we use at ICSI is the Bellcore Isolated Digits database, which has 13 words in the vocabulary: 1, 2, 3, 4, 5, 6, 7, 8, 9, zero, oh, yes, no. They are spoken by 200 speakers over the telephone network, producing a total of 2600 utterances. Here the training and test data consists of sentences with one word each. It is simple not only because each word is pronounced separately but also because each sentence only has a single word.

In continuous recognition, the input sentences have many words often with no pauses separating them. The Numbers’93 corpus, for example, is a continuous-speech database collected by the CSLU at the Oregon Graduate Institute. It consists of numbers spoken spontaneously over telephone lines on the public-switched network. These numbers are extracted from the addresses spoken by the callers of CSLU’s Spelled and Spoken Names Corpus (?). The Numbers’93 database consists of 2167 speech files of spoken numbers
produced by 1132 callers. There are 36 words in the vocabulary, namely zero, oh, 1, 2, 3,...,20, 30, 40, 50,...,100, 1000, a, and, dash, hyphen, and double. This task, including a much larger collection of spontaneously spoken numbers, is being made publically available from CSLU. The Numbers'93 task has inputs like "twelve thirty" or "nineteen seventy four". Here the problem is more difficult both because there are multiple words per sentence, meaning that we need to consider the question of which words can follow each other, and also because the words often run together.

Other continuous recognition tasks we sometimes use are the Resource Management (RM) and Wall Street Journal (WSJ) tasks. RM is a database of ARPA sentences involving ship movements. It has about a 1000-word vocabulary and a simplified grammar. The WSJ task has a database of sentences which are read out loud from the Wall Street Journal. There is a 5000-word, a 20000-word, and a Really Many Word version of this task. Both of these tasks are microphone-quality (not telephone) speech.

We also use the smaller TI/NIST Connected-Digits Recognition Task (“TI-Digits”) (?). Ever since this corpus was made available in 1984, it has became a quasi standard for benchmarking small vocabulary speaker-independent recognition systems. The database was collected in an acoustically treated sound room and digitized at 20 kHz. It has 11 words in the vocabulary: 1, 2, 3, 4, 5, 6, 7, 8, 9, zero, oh. Only the adult speakers are used for training and testing. There are 225 speakers, each providing 77 utterances:

### 2.1.1 TIMIT

Finally, we must mention the TIMIT and NTIMIT database, which we use for much of our bootstrapping (we’ll get to this below). The TIMIT database is a large database of speech that was collected at Texas Instruments and labeled by MIT (thus TI-MIT). The database was collected for the purpose of training speaker-independent phonetic recognition systems.

TIMIT contains recordings from 630 American English speakers. Each speaker is classified as belonging to one of eight dialect regions (see Table 1). 70% of the speakers are male. Table ?? shows the distribution of the speakers according to ethnic background.

<table>
<thead>
<tr>
<th>Dialect Region</th>
<th>Number of Speakers</th>
<th>Percent of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>New England</td>
<td>49</td>
<td>7.7</td>
</tr>
<tr>
<td>Northern</td>
<td>102</td>
<td>16.2</td>
</tr>
<tr>
<td>North Midland</td>
<td>102</td>
<td>16.2</td>
</tr>
<tr>
<td>South Midland</td>
<td>100</td>
<td>15.9</td>
</tr>
<tr>
<td>Southern</td>
<td>98</td>
<td>15.6</td>
</tr>
<tr>
<td>New York City</td>
<td>46</td>
<td>7.3</td>
</tr>
<tr>
<td>Western</td>
<td>100</td>
<td>15.9</td>
</tr>
<tr>
<td>Army Brat (moved around)</td>
<td>33</td>
<td>5.2</td>
</tr>
</tbody>
</table>

Table 1: Geographical Distribution of Speakers in TIMIT

Each speaker in the database recorded 10 sentences. There are three types of sentences:

- 2 “sa” sentences. These two sentences were designed by SRI to elicit dialectal variations through the use of phonetic contexts in which such variations are known to occur. The
<table>
<thead>
<tr>
<th>Ethnicity</th>
<th>Number of Speakers</th>
<th>Percent of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native American</td>
<td>2</td>
<td>.03</td>
</tr>
<tr>
<td>Hispanic American</td>
<td>3</td>
<td>.05</td>
</tr>
<tr>
<td>Asian American</td>
<td>3</td>
<td>.05</td>
</tr>
<tr>
<td>Unknown</td>
<td>17</td>
<td>2.7</td>
</tr>
<tr>
<td>African American</td>
<td>26</td>
<td>4.1</td>
</tr>
<tr>
<td>Euro-American</td>
<td>578</td>
<td>91.7</td>
</tr>
</tbody>
</table>

Table 2: Distribution of Speakers in TIMIT according to ethnic background

two sentences are “She had your dark suit in greasy wash water all year” and “Don’t ask me to carry an oily rag like that.” These sentences were spoken by all of the speakers in the database.

- 5 “sx” sentences. These “phonetically compact” sentences were designed by MIT to give a complete coverage of as many phonetic pairs as possible. MIT designed a total of 450 of these “phonetically compact” sentences and each speaker recorded five of the 450. Each sentence was thus recorded by seven different speakers.

- 3 “si” sentences. Since the small set of phonetically-compact sentences could not cover all possible phonetic contexts, a set of 1,890 sentences was selected by TI from the Brown corpus¹ (?). Each speaker recorded three of these sentences; thus, all 1,890 sentences were spoken once.

The speech was recorded digitally at a sampling rate of 20 kHz and then downsampled to 16 kHz. A Sennheiser close-talking microphone was used for all of the recordings. The speech was initially labeled using an automatic procedure (?) and then hand-corrected by linguists. The speech was labeled at both the phonetic and the word levels.

### 2.2 Algorithms

Let’s now go into more detail on our recognition and training procedures.

Figure 1 summarizes the information flow in a single recognition pass through the icsi software. We will address the training issue later.

Here’s a quick summary of the recognition and training procedure.

Let’s assume we begin with an acoustic signal. This can come from one of 3 sources.

1. Someone else gives it to us. This is the case with Wall-Street Journal, Switchboard, etc., data.

2. the signal is sampled (16bit,16 Khz) from a headset microphone via a SCSI device (the Gradient A-D converter) connected to a Sparc station. The Gradient does automatic silence detection, automatically clipping the silence off of both ends of the speech

¹The Brown corpus is a one-million-word corpus assembled at Brown University in 1963-64. It contains samples of text from a wide variety of genres.
speech signal before passing it to the host. We only have 2 of these Gradient machines, and we are trying to avoid using them. So in the future most of our data will come from:

3. the signal is sampled (16bit, 16Khz) using the Sparc audio device driver on a sparv 5, 10, or 20. We only just now have a working silence detector on this sparv audio. (WHERE IS IT, WHERE’S THE MAN PAGE?).

Now, whether we are using this signal for training some system or for testing, the first thing that happens is that spectral features are extracted from it either every 10 msec (for non-telephone data) or every 12.5 msec (for telephone data) (so 10msec or 12.5 msec = 1 frame). (There’s nothing magic about these numbers, we just use them to be consistent with all our past experiments).

We use a number of different possible feature extraction models:

- PLP
- Rasta-PLP (?)
- J-Rasta-PLP

The next step is to use these features to estimate the probability of a phone for each frame. The pfie or stream of feature vectors is input to a Multi Layer Perceptron (MLP), which computes a vector of phone probabilities. That is, for each frame, the MLP will compute the probability of each phone accounting for the frame. These phone probabilities are stored in an inafile.

The inafile or stream of probability vectors (between 40 and 61 values each frame) is now passed to the decoder. A decoder is a program which finds the most likely sequence
of words given a sequence of phone (or subphone) probability vectors. We use two different
decoders: Y0 and Noway. The decoder takes the following input:

- phone probability vectors (lnfile).
- lexicon of word pronunciations in terms of phones.
- prior probabilities of phones
- language model (e.g., bigram or other grammar)
- various other parameters

The output of the decoder is a single word string which accounts for the sentence.

In parts of our software we do further processing than just word strings; for example in
BeRP, we need to do semantic analysis of the sentence, and so there is further processing
(parsing, database queries, etc) that will be discussed when we discuss BeRP.

How about training the system? At the very minimum, this involves setting the weights
on the multi-layer perceptron (MLP) used to estimate phonetic likelihoods, and estimating
the durations for each phone in the HMM pronunciation lexicon. Unless we are doing isolated
word recognition, we also need to train the bigram grammar used for recognition decoding.

In order to train the MLP, we need a training corpus and the set of correct labels for each
frame in the training corpus. In the best case, as is true with the TIMIT database, we have
relatively high-quality hand-derived labels. So in order to train a TIMIT-net, we take these
labels and use a Error-Back-Propagation-based training algorithm to learn net weights.

For most databases, however, we do not have labels for each frame. Instead, we have
transcriptions at the word level; i.e., an (ascii) string of words for each sentence in the
database. In this case our training algorithm differs in only one very significant regard:
instead of getting the correct answers from the hand-transcribed phone labels, we use the
recognizer itself to convert word transcriptions into phone transcriptions. The process by
which we do this is called forced-Viterbi. The basic idea is to start with some bootstrapping
MLP (which itself is training on a hand-labeled corpus like TIMIT). This bootstrapping
MLP is used to get initial phone probabilities vectors for the training input. Then we run
the standard Viterbi decoding algorithm (see below) except that we tell it what the strings
of words are, rather than making the program figure them out. The result is an optimal
string of phones. Now we have labels that we can use to train up a new net. More details
below.

2.3 Feature Extraction and Pfile Creation

OK, now the same thing again in more detail. Again, let’s assume we begin with an acoustic
signal. At the risk of boring you, this can come from one of 3 sources.

1. Someone else gives it to us. This is the case with Wall-Street Journal, Switchboard,
etc., data.
2. the signal is sampled (16bit, 16 Khz) from a headset microphone via a SCSI device (the Gradient A-D converter) connected to a Sparc station. The Gradient does automatic silence detection, automatically clipping the silence off of both ends of the speech waveform before passing it to the host. We only have 2 of these Gradient machines, and we are trying to avoid using them. So in the future most of our data will come from:

3. the signal is sampled (16bit, 16Khz) using the Sparc audio device driver on a sparc 5, 10, or 20. We only just now have a working silence detector on this sparc audio. (WHERE IS IT, WHERE’S THE MAN PAGE?).

Now, whether we are using this signal for training some system or for testing, the first thing that happens is that spectral features are extracted from it either every 10 msec (for non-telephone data) or every 12.5 msec (for telephone data) (so 10msec or 12.5 msec = 1 frame). (There’s nothing magic about these numbers, we just use them to be consistent with all our past experiments).

We use a number of different possible feature extraction models:

- PLP
- Rasta-PLP (?)
- J-Rasta-PLP

Perceptual linear predictive analysis (PLP) is an extension of linear predictive analysis that takes into account some aspects of human sound perception (?). Log-RASTA is based on PLP but also aims at reducing the effect of linear spectral distortion. J-RASTA tries to handle both linear spectral distortion and additive noise simultaneously. (?)

Each of these methods produces a vector of real numbers (usually 8 or 12) for each acoustic frame. In addition, for each frame we also compute the first and second derivative of the features as well. Once they are computed, features are usually stored in a pfile (see pfile(1)), a binary database file.

So the first thing you need to do for any task is to build a pfile for the training corpus and a pfile for the test corpus. We usually build these once, and leave them around for the whole time-span of an suite of experiments (i.e. months or years).

The command to build a pfile is called make_pfile (1). It takes as input a list of files with digitized acoustic data (a list of wave files) and produces a pfile with the extracted features you have requested for all of these wavefile.

Here is what make_pfile (1) does:

1. Call rasta (1) with the parameters you request (PLP, Rasta-PLP, whether to remove DC offsets, etc etc) and extract features for each wavefile and store the features of each wavefile in a single (ASCII) ‘feature file’. (In a slightly confusing bit of software packaging, the program that does our feature extraction is called rasta even if you are using plp and not rasta encoding). Read the man page.

2. Use appendplp (1) to concatenate all the single feature files to make one big (ASCII) "proto" file.
3. (if desired) Call `calc_deltas (1)` to calculate the delta (first derivative) and double delta (2nd derivative) features.

4. Use `pfle (1)` to create a binary pfile from the above proto file plus the deltas.

So for example, suppose you want to build a pfile with plp features, using 12 plp coefficients, delta features, and double-deltas from a bunch of wavefiles you had lying around. You would specify the of wavefiles and all this information to `make_pfile (1)`, along with other parameters like whether you want DC offset removed, what the window size is, etc. Then it would produce a binary file which has 36 real-valued features for each frame, right? (12 plp feature plus 12 delta-plp plus 12 delta-delta-plp)? Wrong. I forgot to mention that `rasta (1)` also assumes you want an energy features as well as the coefficients. So if you ask for 12 plp coefficients you get 13 features. That means if you ask for delta and double-deltas you get 39 features.

What does this pfile look like? A pfile consists of a fixed-length ascii header section (32768 bytes), followed by a binary data section. The parameters in the header section are as follows:

- `num_sentences`: number of sentences in the data
- `num_frames`: number of frames of data
- `num_features`: number of features per frame encoded in the data
- `num_labels`: number of labels per frame encoded in the data
- `first_feature_column`: the column number (counting from 0) of the first feature
- `first_label_column`: the column number (counting from 0) of the first label
- `format`: a string of characters, one for each column, which determines
  - the type of each column:
    - `x`: hexadecimal number
    - `d`: decimal number
    - `f`: floating-point number (single precision IEEE)

The data section is laid out in the following column format:

```
ID | feature | label
columns | columns | columns
^ ^ ^
| | | _contains label numbers for frame
| | | _contains feature data for frames
| | _contains identification for frames (e.g. sentence number, frame number)
```

The number and format of the columns are determined in the `first_feature_column`, `first_label_column` and `format` parameters in the header.

Usually our pfiles will have two ID columns (a sentence number and a frame number), and only one label column. The label column will be used to put in the correct phone label when we use the pfile for training the MLP.
So for our example above (PLP12 + delta-PLP12 + delta-delta-PLP12) that’s 2 IDS plus 39 features plus 1 label = 42 total columns.

`pfile (1)` is used both to create and to examine pfiles (since they are binary, you can’t just `cat` them). For example, here’s the parameters of a sample plp12 + delta + delta-delta BeRP training pfile:

```plaintext
prompt% pfile amer.trn.plp12+dd.pfile

num_sentences = 3051
num_frames = 894581
num_features = 39
num_labels = 1
first_feature_column = 2
first_label_column = 41
format = 2d39fd
pfile_header = version 0 size 32768
data = size 37572402 offset 0 ndim 2 nrow 894581 ncol 42
prompt%
```

Notice that `2d39fd` means 2 (decimal) ID fields followed by 39 (float) feature fields plus 1 (decimal) label field.

To examine one parameter, just add the parameter name after the file name:

```plaintext
prompt% pfile amer.trn.plp12+dd.pfile num_sentences
num_sentences = 3051
prompt%
```

Read the man page for more details. Note, by the way, that the pfile name `amer.trn.plp12+dd.pfile` is long but very descriptive; it tells us that this is training data from american speakers encoded with plp12 + deltas and double deltas. Try to use long descriptive names and avoid names like `mynewpfile` or `pfile.new.37c.fixed`.

Creating a pfile works the same way whether you are creating it for training or for testing; usually the pfiles for testing will be smaller.

### 2.4 Phonetic Likelihood Estimation

#### 2.4.1 The Basic Idea

Recall that we model speech recognition as a maximization problem; choosing the string of words that was most probable given the data, or:

\[
\text{argmax}_{w \in \mathcal{L}} P(w) P(y|w)
\]  

In order to talk about the likelihood \(P(y|w)\), we need to look at the problem in slightly more fine-grained detail. First, we generally model the acoustic input \(y\) as a string of individual acoustic observations \(y_i\). That is, we sample the input sentence in units 20 ms
wide every 10 ms (or 25 ms every 12.5 ms for telephone speech) and call each chunk an individual observation \( y_i \). Second, we model each word as made up of a string of phones. So we can think of the smallest part of the likelihood computation as computing the likelihood of just one sub-part of the acoustic data (a single time slice or small set of time slices), given just one subpart of the word string (a single phone from a single word). We call this little piece the phonetic likelihood estimation problem. The problem is to find a function which gives us the likelihood that any given single input frame is produced by any given single phone. This likelihood is then used as the emission probability for the HMMs that model the pronunciations of words. The two most popular ways to solve this problem are the use of Gaussian estimators and the use of MultiLayer Perceptrons (MLPs).

Our software uses a discriminatively-trained Multi-Layer Perceptron (MLP) in an iterative Viterbi procedure to estimate emission probabilities. (7) and (8) show that with a few assumptions, an MLP may be viewed as estimating the probability \( P(q|x) \) where \( q \) is a subword model (or a state of a subword model) (i.e. for our purposes here, a phone) and \( x \) is the input acoustic speech data. We can then compute the likelihood \( P(x|q) \) needed by the Viterbi algorithm by dividing by the prior \( P(q) \), according to Bayes’ rule; we ignore \( P(x) \) since it is constant in time-synchronous Viterbi:

\[
P(x | q) = \frac{P(q | x)P(x)}{P(q)}
\]  

(2)

The estimator consists of a simple three-layer feed forward MLP trained with the back-propagation algorithm (see Figure 2).

Our MLPs have a layered feedforward architecture with an input layer (consisting of the input variables), zero or more hidden (intermediate) layers, and an output layer, as shown in Figure 4.

The input layer consists of 9 frames of input speech data. Each frame, typically representing 10 msec of speech, is encoded by whatever vector of features you choose; such as 39 features (RASTA-12, delta-RASTA-12, delta-delta-RASTA-12, energy, delta-energy, delta-delta-energy, although we often skip the energy). Typically, we use 200-1000 hidden units. The output layer has between 40 and 61 units, one for each of the context-independent phonetic classes used in our various lexicons.

Each layer computes a set of linear discriminant functions (7) (via a weight matrix) followed by a nonlinear function, which is often a sigmoid function

\[
f(x) = \frac{1}{1 + \exp(-x)}
\]  

(3)

As discussed in (7), this nonlinear function performs a different role for the hidden and the output units. On the hidden units, it serves to generate high order moments of the input; this can be done effectively by many nonlinear functions, not only by sigmoids. On the output units, the nonlinearity can be viewed as a differentiable approximation to the decision threshold of a threshold logic unit or perceptron (7), i.e., essentially to count errors. For this purpose, the output nonlinearity should be a sigmoid or sigmoid-like function. Alternatively, a function called the softmax can be used, as it approximates a statistical sigmoid function.
For an output layer of \( K \) units, this function would be defined as

\[
f(x_i) = \frac{\exp(x_i)}{\sum_{n=1}^{K} \exp(x_n)}
\]

(4)

![Diagram of the Phonetic Likelihood Estimator](image)

Figure 2: Phonetic Likelihood Estimator

We’ll talk about how to train the net below.

### 2.4.2 The Forward Pass: How to Do It

OK, so now you have the general idea, let’s talk about how to actually generate a set of phone likelihoods from a pfile, given a net. We call this the *forward pass* (since we are running the network forward from inputs to outputs; running backwards from outputs to inputs is what we do when training).

The program we use is called **recog_ffp** (1) (**ffp** stands for Feed-Forward Pass). The input to **recog_ffp** (1) consists of

- the pfile (i.e. the extracted features from the acoustics we want to recognize)
- the MLP (specified by a weights file)

The output will be an **lnafile** (5). The lnafile contains the phone likelihoods for each phone for each frame, in a compressed format.
Like most of our scripts, `recog_ffp (1)` is a `pmake (1)` script, and so you specify all your input parameters in a file called `files.mk` in the directory in which you execute the command. The `files.mk` file (or equivalent command-line specifications) should specify the following variables; I have given sample values for expository purposes only.

```
RAPHOST = unraps
weights = ./train.4014.july22/net/iter0.wts
pfile = /u/jurafsky/berp/runs/data/berp.g+a-tst.r8+d.pfile
norms = /u/jurafsky/berp/runs/data/berp.g+a-tst.r8+d.norm
lnafile = /n/unrap/db/rap/jurafsky/tst.g+a.july29/iter1.lna
mlp_window = 9
mlp_num_hidden = 512
mlp_num_output = 61
mlp_num_input = 162
mlp_opt_params = "-some_bob_option -some_other_bob_option"
```

Let’s look at what each of these arguments specify. Recall that in order to perform the phonetic likelihood estimation quickly, we usually offload the computation to special-purpose hardware, the Ring Array Processor (RAP) (?). It’s recently become possible to run the estimation efficiently on a workstation, so I’ll update this manual soon. Meanwhile, we usually use one of the three machines `rap`, `unrap`, or `rap2`. So:

**RAPHOST** : the name of the rap to run on.

**weights** : the mlp weights file.

**norms** : the mlp norms file. This is something like the means and variance for each input feature, so the net can adjust its dynamic range per input feature. If this description is wrong, which I suspect, please send a bug report asap. `recog_ffp` will automatically compute the norms for the net and put them in norms. If the file norms already exists and is newer than weights, it will be used instead of recomput- ing the norms. For a test run, we are usually required to use the norms from the training set, not the norms from the test set, so don’t recompute them on the test set.

**pfile** : the test pfile.

**lnafile** : the name of the lnafile to be output

The rest of the parameters in the sample above are passed to `bob (1)`, the program which manages our net forward and training passes. There are currently four required parameters:

**mlp_window** : the width of the input window in frames

**mlp_num_hidden** : the number of hidden units

**mlp_num_input** : the number of input units

**mlp_num_output** : the number of output units

If you want to specify any other parameters, put them as the value of `mlp_opt_params`. 
2.5 Decoding: The idea

The task of the decoder is to take as input a string of probability vectors over phones, and find the most likely string of words which fits these vectors. The most effective way we know to solve this problem is to cast it as a maximization problem: “What is the most likely string of words \( w \) given some acoustic input \( y \)”, which we can formalize as follows:

\[
\arg\max_{w \in \mathcal{L}} P(w|y)
\]

That is, find the string of words \( w \) out of all possible strings of words in the language which maximizes the posterior probability \( P(w|y) \). We can conveniently expand this via the Bayes rule as follows:

\[
\arg\max_{w \in \mathcal{L}} \frac{P(w)P(y|w)}{P(y)}
\]

Here we’re looking for the string of words \( w \) whose prior probability \( P(w) \) multiplied by the likelihood of the acoustic input given these words \( P(y|w) \) and divided by the prior probability of the acoustic input \( y \) is maximal. But since we are maximizing over strings of words which account for the same input, we can ignore the prior \( P(y) \), since that will be the same for all word strings, and won’t effect our result. So we can further simplify the problem of speech recognition as

\[
\arg\max_{w \in \mathcal{L}} P(w)P(y|w)
\]

Intuitively, we can view this maximization problem as taking every possible string of words in our language and checking how well this string matches the acoustic input, as well as how likely a string it is a priori.

\[
\Rightarrow
\]

- Alice was beginning to get
- Every happy family
- In a hole in the ground
- Whether tis nobler

Figure 3: Searching the word string most likely to match the input

In order to solve this problem we need four things:

**pronunciation dictionary**: For each word in our language, we’ll need to know how to pronounce it. This will come down to knowing what phones it is made up of, and what order they come in (and if a word can be pronounced more than one way, we’ll want the probabilities for each pronunciation).
word likelihood algorithm: An algorithm for taking such a pronunciation model for a word and telling us the likelihood of part of the acoustic input given that word.

language model: Since we’re maximizing over all strings of the language, we need to know what it means to be a string of the language. This is what a language model (LM) or grammar tells us; in operational terms, it tells us which words are allowed to follow which other words, and with what probability.

search algorithm: If we are going to need to search through all possible strings of words in the language, we had better have an efficient algorithm to do it with.

2.5.1 The Language Model

In the context of speech recognition the grammar is called a Language Model (LM). It computes $P(w)$, the prior probability of the string of words. The task of the LM is to compute the probability of a string of words $w^n$ (we’ll use this notation to mean a consecutive sequence of words from numbered 1 through $n$). By definition, this probability can be expressed as follows:

$$P(w^n) = P(w_1)P(w_2|w_1)P(w_3|w_2)\ldots P(w_n|w_{n-1})$$

$$= \prod_{k=1}^{n} P(w_k|w_{k-1}) \quad (8)$$

The idea of the bigram or n-gram grammar is to give an approximation to this product without having to compute each of the terms, which can be quite complex. We make the simplifying assumption that it’s the last word (in the bigram) or the last $N - 1$ words (in the b-gram) that are the most important, and so we can approximate the probability of a word given a whole string with its probability given the last word(s) of the string:

$$P(w_n|w_{n-1}) \approx P(w_n|w_{n-N+1}) \quad (9)$$

N-gram grammars have many advantages; they can be trained very easily on large corpora simply by counting the number of times words follow other words. Formally, the maximum likelihood estimate of the parameters for an n-gram model is:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1})} \quad (10)$$

The simplest kind of LM we use is the bigram grammar, which is a function specifying: for each word $w_i$, for each word $w_j$, the probability that $w_j$ follows $w_i$. For the special case of the bigram, the equation simplifies to

$$P(w_2|w_1) = \frac{C(w_1w_2)}{\sum_w C(w_1)} \quad (11)$$

For example, here’s part of a BeRP grammar for the words following the word eat:
2.5.2 The Lexicon: The Hidden Markov Model (HMM)

In order to compute the likelihood of a series of acoustic observations given a string of words, we obviously need to know the likelihood of a series of acoustic observations given a single word. In order to know this, we need to know how a word is pronounced, i.e., what phones it is composed of, and in what order, with what probabilities. The most popular model of lexical pronunciation is the **Hidden Markov Model (HMM)**. Our HMMs represent each phone of a word as a state and the different pronunciations of words as different paths through different states in a network. The arcs between states are augmented with probabilities which indicate the conditional probability of transitioning to the following state given the preceding one. For a discrete, first-order Markov process, we can represent this system as a single N-by-N probability matrix of state transitions, where entry $a_{ij}$ indicates the probability of transitioning from state $i$ to state $j$. In order to model the fact that a phone/state may stretch over more than one input frame, each state may have self-loops.

LOTS MORE HERE WITH PICTURE AND ASSUMPTIONS. Discuss multiple pronunciations, etc, etc.

Of course the input to the decoder is not phones, but acoustic features. So the HMM needs a way to map features to phones. Each state is an HMM is associated with an output probability density function (pdf). For each state $s_i$, the pdf assigns to each output symbol (where an output symbol is composed of a vector of real-valued acoustic features) the likelihood that this symbol was produced by this state, i.e.: $P(O_i|S_i)$

2.5.3 Decoding: How to Implement It

The decoding architecture:

```
cat lnafile | lna2y0new ( | addgarbage) | y0
```

For example, an actual decoding stream for 10 berp sentences chosen from a larger lna file using the timit61 phone set:

```
cat test_iter1_0.lna |
lna2y0new -b -start 0 -end 9 -ninp 61 |
addgarbage datatype=binary priorfile=net/iter0.priors numphones=30
verbose=true |
y0 binary=true wordtranspenalty=1.0e-04 lexicon=../train.4014.july22/lex/iter1.lex
lmscale=7.0 amscale=3.0 bigram=../bg.out maxactivewords=10 startpruningframe=0
transcut=1.00 beamwidth=150.0 phoneprune=100.0 verbose=true
```
lna2y0new  lna2y0new converts the posteriori phone probabilities stored in Tony Robinson’s LNA probability format to a format usable by y0 and noway. There is a man page (lna2y0new (1)). lna2y0

lna2y0 allows lna2y0new can produce ascii or binary format that can be read by y0. Furthermore it allows to select sentences from .lna files. It’s typically used in a pipe like: cat `lna-file` - | lna2y0new -b -start `first sent` -end `last sent` - | y0 -b ... or to examine the contents of a lna file: cat `lna-file` - | lna2y0new -a -start `first sent` -end `last sent` - | less

addgarbage

y0  y0 program assumes that the network outputs are estimates of the posterior probability of each class. Y0 divides the output of the network the priors in order to get the local distances (likelihood of each class given the data) for the dynamic programming.

2.6 Scoring

2.7 Training the MLP

2.7.1 The Basic Idea

To reiterate, in order to train the MLP, we need a training corpus and the set of correct labels for each frame in the training corpus. In the simplest case, as is true with the TIMIT database, we have relatively high-quality hand-derived labels. So in order to train a TIMIT-net, we take these labels and use a Error-Back-Propagation-based training algorithm to learn net weights.

For most databases, however, we do not have labels for each frame. Instead, we have transcriptions at the word level; i.e., an (ascii) string of words for each sentence in the database. In this case our training algorithm differs in only one very significant regard: instead of getting the correct answers from the hand-transcribed phone labels, we use the recognizer itself to convert word transcriptions into phone transcriptions. The process by which we do this is called *forced-Viterbi*. The basic idea is to start with some bootstrapping MLP (which itself is training on a hand-labeled corpus like TIMIT). This bootstrapping MLP is used to get initial phone probabilities vectors for the training input. Then we run the standard Viterbi decoding algorithm (see below) except that we tell it what the strings of words are, rather than making the program figure them out. The result is an optimal string of phones. Now we have labels that we can use to train up a new net. More details below.

2.7.2 The Algorithm

Let’s start with a summary of the training architecture quoted from Morgan and Bourlard (1995).

MLPs with enough hidden units can (in principle) provide arbitrary mappings $g(x)$ between input and output. MLP parameters (the elements of the weight matrices) are trained to associate a “desired” output vector with an input vector. This is achieved via the Error
**Back-Propagation (EBP)** algorithm (see (?), (?), (?), and (?) for multilayer networks; (?), (?), and (?) for single layer networks; (?) for a control theory version) that uses a steepest descent procedure to iteratively minimize a cost function.

Popular cost functions are, among others, the *Mean Square Error* (MSE) criterion:

$$E = \sum_{n=1}^{N} \left\| g(x_n) - d(x_n) \right\|^2$$  \hspace{1cm} (12)

or the relative entropy criterion:

$$E_e = \sum_{n=1}^{N} \sum_{k=1}^{K} \left[ d_k(x_n) \ln \frac{d_k(x_n)}{g_k(x_n)} + (1 - d_k(x_n)) \ln \left( \frac{1 - d_k(x_n)}{1 - g_k(x_n)} \right) \right]$$ \hspace{1cm} (13)

where \(x_n\) is the pattern to be classified, \(d(x_n) = (d_1(x_n), \ldots, d_k(x_n), \ldots, d_K(x_n))^t\) represents the desired output vector (for classes \(q_k\), \(k = 1, \ldots, K\)), \(g(x_n) = (g_1(x_n), \ldots, g_k(x_n), \ldots, g_K(x_n))^t\) the observed output vector, \(K\) the total number of classes, and \(N\) the total number of training patterns.

We and others have used on-line training instead of off-line (true gradient) backpropagation. In this approach, the weights are adjusted in the direction of the error gradient with respect to the weight vector, as estimated from a single pattern. With an accurate estimate of the error gradient, one could proceed in the direction of the local training minimum. However, the per-pattern gradient estimate can be viewed as a noisy estimate of the gradient over the entire training set. The size of the learning step can be viewed as the magnitude of the noise; in the limit, very large learning steps move over the error surface randomly, while very small steps closely correspond to the true gradient. In fact, it can be beneficial to have more noise (larger steps) initially, in order to escape from potentially poor local solutions. Additionally, given realistic training data, which is typically quite redundant, each full pass through the data represents many passes through similar subsets, and thus can be relatively efficient.

In practice, using on-line gradient search and a relative entropy error criterion, only a small number of passes through the data are required to phonetically train the network (typically 1 to 5).

In addition to the use of on-line training, other aspects of the training method include:

- **Cross-validation** – It is necessary to use a stopping criterion based on an independent portion of the data, i.e., utterances that are not used for training. While this is a good general rule in training pattern classifiers, most of the early published suggestions for neural network stopping criteria were measures based on the training set, e.g., gradient magnitude or slope. The networks that were ultimately successful for continuous speech recognition are quite large, often using hundreds of thousands to millions of parameters. These nets are susceptible to overfitting the training data, resulting in bad probability estimation and very poor generalization performance on the test set.

In addition to merely halting the training based on performance for an independent validation set, a training procedure can be used in which the learning rate is also adjusted to improve generalization (?), (?). Specifically, the learning rate is reduced...
(typically by a factor of 2) when cross-validation indicates that a given rate is no longer useful. Additionally, we have empirically noted that after the first reduction, only a single epoch at each rate is useful. The heuristic of only permitting a single pass for any learning rate after the initial one cuts down the number of epochs by almost a factor of two, and has little effect on final performance.

- Training criterion – Using relative entropy (6) instead of the MSE criterion speeds convergence. The correction resulting from this criterion is always linear and does not saturate when the outputs values are at the extremes (tails) of the sigmoid (where the correction for the MSE criterion is negligible).

- Initialization of output biases – Histograms of the output biases of phonetically trained MLPs showed a narrow distribution around a strongly negative value (typically around -4). This is no coincidence, since the input to the sigmoid nonlinearity for and output unit produces the log odds, or \( \log \frac{p(j|x)}{q_j(x)} \) when the output produces \( p(q|x) \). When the evidence from the data is equivocal, this is roughly equal to \( \log \frac{p(x)}{1-p(x)} \), and since these each \( p(q) \) is much less than 1, the sigmoid input is roughly equal to \( \log p(q) \). Under the assumption that the data is uninformative, the weighted sum due to the input from the previous layer can be ignored, and the bias should be roughly the log prior for the associated class. This is a rough argument, and for specific distributions (such as a Gaussian) it can be shown to be inaccurate. Nonetheless, the empirical observation (from histograms) is that it is roughly true, at least in the sense that the average output bias of the converged network is close to the average log prior probability. Additionally, it has been confirmed that initializing the biases to the rough range that they will ultimately approach speeds convergence, and slightly improves the results.

- Random pattern presentation – In earlier forms of our analysis we presented the data sequentially according to the speech signal. Sequential presentation of the acoustic vectors to the net (i.e., in the order that they were spoken) can cause slow convergence, requiring a very low learning rate in the case of on-line training. In the current method, the speech vectors are presented at random (preserving the relative frequencies of the classes), which speeds up MLP training, and also slightly improves the results. In a variant on this approach for practical training using speech databases whose size exceeds the physical memory, blocks of sequential sentences (which can be randomized at the sentence level) are read from disk into physical memory, and frames can be presented randomly from within the block. In both schemes, it does not appear to matter whether random sampling is done with or without replacement (i.e., it does not matter whether each random frame choice is constrained to be a different one than had already been chosen).

2.7.3 The Software

The script for training the MLP is called train (1) [no snide remarks about lack of imagination please], it takes a pfile file (i.e. you remember, that familiar feature file) and a set of labels as input, and uses BOB on a rap (or a sparc if you don’t have any pressing deadlines
before, say, the end of the millenium) to train up a net. Again, like most of our other scripts
\texttt{train (1)} is a \texttt{p}make file, and takes its input in a ”files.mk” file in the directory in which you
execute the command. The files.mk file (or equivalent command-line specifications) should
specify the following variables; I have given sample values for expository purposes only.

\begin{verbatim}
RAPHOST = unrap
job=march15_a
init_weights = $(datadir)/timit.weights
pfile = /n/icsib13/xa/jurafsky/berp.dc012_all.r8+d.pfile
norms = /n/unrap/db/rap/jurafsky/pfiles/berp.dc012_all.r8+d.norm
weights = ./net/iter0.wts
labels = ./out/iter0.labels
mlp_window = 9
mlp_num_hidden = 512
mlp_num_output = 61
mlp_num_input = 162
mlp_opt_training_params = "-bob_other_param learning_rates='0.008 0.004'"
BOB = my_personal_bob
\end{verbatim}

\textbf{RAPHOST} : the name of the rap to run on; rap, unrap, or rap2.
\textbf{job} : the job-name you want to give to this run. It will be appended (or prepended) to
various output files.
\textbf{init_weights} : is the mlp weights file to initialize off of; usually a timit net.
\textbf{train} : will automatically compute the norms for the input features and put them in norms.
If the file norms already exists and is newer than weights, it will be used instead of
recomputing the norms.
\textbf{pfile} : the training pfile.
\textbf{labels} : the labels file, with one label per frame of the pfile. It will be patched into the
‘labels’ column of the pfile before training
\textbf{mlp_window} : the width of the input window in frames
\textbf{mlp_num_hidden} : the number of hidden units
\textbf{mlp_num_input} : the number of input units
\textbf{mlp_num_output} : the number of output units

If you want to specify any other parameters, specify them as the value of \texttt{mlp\_opt\_training\_params}.
If you don’t set the parameter \texttt{learning\_rates} in \texttt{mlp\_opt\_training\_params}, then the
learning rate will automatically default to the following BoB parameters; you don’t need to
specify them:

\begin{verbatim}
   initial\_learning\_rate 0.008
   -min\_improvement\_ramp 0.5
   -min\_improvement\_stop 0.5
   -max\_num\_epoch 15
\end{verbatim}
This means that the learning rate will start at .008 and will stay at the same rate every epoch until ... then cut in half... for a max of ... FINISH THIS.
You can set BOB if you don't want to use the standard BOB.

2.8 Embedded Training of Net and Lexicon

It is possible to use what is called ‘embedded’ Viterbi training to iteratively optimize both the phone segmentation (and hence the pronunciation of words) and the MLP parameters. In this procedure, illustrated in Figure 4, each MLP training is done using labels from the previous Viterbi alignment. In turn, an MLP is used to estimate training set state probabilities, and dynamic programming given the training set models is used to determine the new labels for the next MLP training.

Of course, as for standard HMM Viterbi training, one must start this procedure somewhere, and also have a consistent criterion for stopping. Many initializations can be used, including initializing the training set segmentation linearly or in proportion to average phoneme durations. More recently we have achieved better results initializing the procedure by training an MLP on a standard hand-segmented corpus (TIMIT for the case of American English), and using this MLP to align the training set for any new unlabeled corpus.

MUCH MORE HERE AND THEN DESCRIBE embed (1).

2.9 Training LMs
ERIC COULD YOU WRITE THIS PLEASE.

3 FILES AND FILE FORMATS

3.1 LNA

The output of the RAP (a posteriori probabilities) are stored in a compressed form to avoid huge files. The compression uses only 8 bit per probability resulting in slightly quantized data. Files containing compressed data have the extension .lna. To get the uncompressed data use lna2y0new (see LNA2Y0).

The lna file format is discussed in lna (5).

3.2 PFILE

The data used to train a neural net is encoded in binary files called ”pfiles”. A pfile consists of a fixed-length ascii header section (32768 bytes), followed by a binary data section.

The parameters in the header section are as follows:

num_sentences : number of sentences in the data
num_frames : number of frames of data
num_features : number of features per frame encoded in the data
num_labels : number of labels per frame encoded in the data + first_feature_column: the column number (counting from 0) of the first feature + first_label_column: the column number (counting from 0) of the first label + format: a string of characters, one for each column, which determines + the type of each column: + x: hexadecimal number + d: decimal number + f: floating-point number (single precision IEEE)

+ The data section is laid out in the following column format:

<table>
<thead>
<tr>
<th>ID</th>
<th>feature</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>columns</td>
<td>columns</td>
</tr>
</tbody>
</table>

| | | | contains label numbers for frame |
| | | | contains feature data for frames |
| | | | contains identification for frames (e.g. sentence number, frame number) |

+ The number and format of the columns are determined in the + parameters first_feature_column, first_label_column and format + in the header.

### 3.3 Lexicon

A multiple pronunciation lexicon file always begins with a line containing “MPF” (Multiple Pronunciation File). On the next line is an integer specifying the number of words in the file. Following this are the entries for each of the words in the file. Each entry has the format shown in Figure 5. Where <word num> is just the index of the word and is ignored. <number of states> is the number of states in the word. NOTE: this is not the same as the number of phonemes in a word (unless you have only 1 state per phoneme.) <spelling> is the spelling of the word.

The second line of each entry gives the indexes of the states of the word. The first two indexes are always -1 and -2. These are the indexes of the initial null state and the final null state of the word. Following these are the indexes of the rest of the states of the word. The number of integers on this line (including the -1 and -2) should be equal to <number of states>.

The next line of each entry lists all of the transitions that can leave from state 0 (the initial null state). The first integer on the line, <state>, is the index of this state in the list of states for the word. So for the initial null state, the <state> number is 0. NOTE: this is different from the indexes listed on line 2 of the entry. This index is an index into line 2. The second integer on the line, <#out-trans>, is the number of out transitions that occur for this state. Following this are pairs of <tostate> and <prob> numbers. There should be <#out-trans> pairs of these numbers. The <tostate> is the index of the state that we are transitioning to from this state. (Again note that this index is an index into line 2 of this entry.) <prob> is a floating point number indicating the probability of making that transition.

The next line of the entry lists all of the transitions that can leave from state 1 (the final null state). Since nothing leaves from the final null state of the word, this line will always look like “1 0”.

20
The rest of the lines of the entry just list the transition that can leave from each state of the word along with their transition probabilities.

Here is an example of a multiple pronunciation file containing only 1 word, the word “a”:

MPF
1
1 4 a
-1 -2 0 0
0 1 2 1.0
1 0
2 2 0.5 3 0.5
3 2 1 0.5 3 0.5

In this example, the word “a” has four states. They are -1 (initial null), -2 (final null) and two “real” states. Both of these “real” states have the same index (index 0) into the output layer of the MLP.

The initial null state (state 0) has one output transition to state 2 (which has index 0 into the output layer of the MLP.) Since there is only one output transition, it’s probability is 1.0. The final null state (state 1) doesn’t have any transitions out of it. State 2 has two transitions, a self-loop with probability of 0.5 and a transition to state 3 with probability of 0.5. State 3 has two output transitions, one self-loop with probability of 0.5 and one to state 1 (the final null state) with probability of 0.5.

3.4 Backoff Bigram

The default language model for Y0 is a back-off bigram. For each word in the grammar, there is a unigram probability, back-off weight and list of word/probability pairs of the specified bigrams. All values in this file should be in the log domain (base e). The format of the file is:

NUM-OF-WORD
>wordx wordx-unigram wordx-backoff-weight
word-following-wordx bigram-probability
word-following-wordx bigram-probability
word-following-wordx bigram-probability

>wordy wordy-unigram wordy-backoff-weight
word-following-wordy bigram-probability
word-following-wordy bigram-probability
word-following-wordy bigram-probability

Note that there is a
before wordx and wordy. The spelling of (almost) all of the words in this file must match the spelling found in the lexicon file.

Note that the words <s> and </s> should appear in the bigram file and not in the lexicon. The <s> represents the (null-) model for sentence start and </s> the (null-) model for sentence end. These are used to specify the language model for a word starting or ending a sentence.

The back-off bigram works in the following way. If the wordx to wordz transition is specified in the file, the specified bigram probability is used. If the wordx to wordz is not specified, then the bigram is computed by wordx-backoff-weight times wordz-unigram.

3.5 Priors

3.6 Phonesets

A phoneset file is used to define each set of phones or events used in the ICSI speech software. Currently we have only three standard phonesets: timit61, icsi56, and Sulin’s in-progress event phoneset. timit61 is the standard ARPAbet used for TIMIT. icsi56 is a more useful subset of the timit set where phones which are in practice hard to distinguish are merged together - for example the many silence phones of timit61 into one silence phone in icsi56. Sulin also as

Phonesets are currently stored in /u/drspeech/data/phonesets. Look in /u/drspeech/data/phonesets/ph for a chart written by Gary Tajchman comparing our phonesets with other standards that we sometimes need to deal with.

The phoneset file has the following format:

```
<#-of-phones-in-set>
  <Phone0> <Phone0-Index>
  <Phone1> <Phone1-Index>
  <Phone2> <Phone2-Index>
  ...
```

For example, the timit61 phoneset file starts off:

```
61
b 0
d 1
g 2
p 3
t 4
```

3.7 Weights

The weight file format is:
weigvec <\# of in->hidden weights>
<weights in floating point, one per line>

weigvec <\# of hidden->output weights>
<weights..>

biasvec <\# of hidden units>
<bias for hidden unit>

biasvec <\# of output units>
<bias for output unit>

The weights are in order by inputs for each output. For example, input unit 0 to output unit 0 weight, input unit 1 to output unit 0 weight, input unit <\# input - 1> to output unit 0 weight, input unit 0 to output unit 1 weight, input unit 1 to output unit 1 weight, input unit <\# input - 1> to output unit 1 weight, input unit <\# input - 1> to output unit <\# output - 1> weight,

SOMEONE FIX THIS: 1) Assuming that the right most column of the weight matrices are set up to be the biases, what should the corresponding fixed input unit be set to, -1 or +1.?????
I believe that it probably is -1, but I don’t know.

3.8  Context-Dependent Durations
Phone duration files are stored in the following formate

DURF
<num phones>
TRIPHONE <num triphones>
<triphone section>
RBIPHONE <num rbiphones>
<right biphone section>
LBIPHONE <num lbiphones>
<left biphone section>
MONOPHONE <num monophones>
<monophone section>
FUNCTION_WORDS <num function_words>
(function words section)

Each section is formated as follows:

TRIPHONE  Each line is of the following format:
            <phone> <left-context> <right-context> <avg duration in frames> <num samples>

RBIPHONE  Each line is of the following format:
            <phone> <right-context> <avg duration in frames> <num samples>
LBIPHONE Each line is of the following format:
<phone> <left-context> <avg duration in frames> <num samples>

MONOPHONE Each line is of the following format:
<phone> <avg duration in frames> <num samples>

FUNCTION_WORDS Each subsection has the following format:
<word> <max phone number>

4 Dave Johnson’s Guide to Successfully Writing ICSI Software

5 Where is all the software anyhow?

Everything is in /u/drspeech. src

5.1 Setting up your environment

Before doing anything else, ensure that the proper environment variables are set. Two environment variables need to be set in users’ initialization scripts (e.g. .cshrc) in order to use the speech software. SPEECH_DIR indicates the root speech directory, typically /u/drspeech. SPEECH_ARCH specifies the "architecture" of a users workstation. Typical values are "sun4", "sun4-sos5" (i.e. running Solaris) and "iris". (NOTE: as sun4 executables also run under Solaris, choosing "sun4" as an architecture is sensible even under Solaris).

The user also needs to set her path to include both the architecture specific and shared speech binary directories, and to include the speech manual pages in your MANPATH environment variable.

A suitable section of .cshrc might be:

if (! $?SPEECH_ARCH) then
  switch('/bin/uname')
  case SunOS:
    setenv SPEECH_ARCH sun4
    breaksw
  case IRIX:
    setenv SPEECH_ARCH iris
    breaksw
  default:
    setenv SPEECH_ARCH unknown
    breaksw
  endsw
endif
if (-d /u/drspeech) then
  setenv SPEECH_DIR /u/drspeech

    24
setenv MANPATH "$MANPATH":$SPEECH_DIR/man
endif
if (-d $SPEECH_DIR/$SPEECH_ARCH/bin) then
    set path = ($SPEECH_DIR/$SPEECH_ARCH/bin $SPEECH_DIR/share/bin $path)
endif

6 Writing Software

The languages of preference for writing speech shell scripts are the Bourne shell (sh) and PERL (perl), although the C shell (csh) and TCL/Tk (wish) are also acceptable if necessary.

Shell scripts should explicitly set their own paths, not rely on a users path. Shell scripts should also use environment variables for the names of all programs and scripts, with defaults provided. This allows a user to override the installed version of any program without having to edit the script.

Code for parsing the ICSI style arguments (i.e. arg=value) has been written for common scripting languages - check out /u/drspeech/share/lib/icsiargs.sh,csh,pl. These scripts also handle the setting up of a suitable PATH. Note that if these scripts are included (e.g. with the sh "source" command) MUST be accessed using the SPEECH_DIR environment variable.

A well written shell script should:

- allow all program and path names to be overridden by the environment (standard Unix utilities excepted)
- store all temporary files in /tmp and erase them when finished or on interrupt
- return 0 on success, 1 on error

Scripts should be installed into /u/drspeech/share/bin/ from a directory under /u/drspeech/src/. Typically several shell scripts and/or executables will be developed together in a source directory (along with a suitable test environment!) and installed to their final destination using a makefile. In the source directory, scripts should have a ".sh" suffix (to allow easy grepping), but should be installed in the bin directory without a suffix (someday the functionality may be replaced by an executable - we want to be able to do this without changing everything that references this script).
Figure 4: The ICSI Embedded Training Algorithm
<word num> <number of states> <spelling>
-1 -2 <1st-state> <2nd-state> ...
<state> <#out-trans> <tostate> <prob> <tostate> <prob> ...
<state> <#out-trans> <tostate> <prob> <tostate> <prob> ...
<state> <#out-trans> <tostate> <prob> <tostate> <prob> ...
...

Figure 5: Format for Multiple Pronunciation Lexicon Files