Lecture 8: Speech Recognition Using Finite State Transducers

Lecturer: Mark Hasegawa-Johnson (jhasegaw@uiuc.edu)
TA: Sarah Borys (sborys@uiuc.edu)
Web Page: http://www.ifp.uiuc.edu/speech/courses/minicourse/

June 27, 2005

This lecture introduces two different ways to use finite state transducers to implement speech recognition testing, and one method that uses FSTs to implement speech recognizer training.

1 Using the ATT Decoder

In order to use HTK-trained speech recognition models with the AT&T speech recognition search engine, three types of conversion are necessary. First, you must convert the HTK-format hidden Markov models into ATT format acoustic models. Second, you'll need to write finite state transducers for the language model, dictionary, and context dependency transducer. Third, acoustic feature files need to be converted from HTK format to raw floating point. After implementing these conversions, it's necessary to set up configuration files and then run `drecog`.

1.1 HMM Conversion

Both AT&T and HTK implement the idea of a "macro" — a piece of an HMM that may be shared between different acoustic models. For example, consider the MMF shown in Fig. 1. This MMF defines just one HMM ("an_hmm"), with two states, each of which has two diagonal-covariance Gaussian mixtures. The first state is tied to the macro "statemacro1." The second state has two mixtures, the first of which is tied to macro "mixmacro1," and the second of which is defined by the macros "meanvec1" and "varvec1." The HMM's transition probability matrix is defined by the macro "transition_macro." Many of these same elements are re-used several times. Even though the HMM has two states with two mixtures per state, tying has reduced the total number of trainable parameters, so that there are only 3 distinct mean vectors defined in this file (in macros "meanvec1," "mixmacro1," and "statemacro1"), and only 2 distinct variance vectors.

The following files implement the same tying scheme for the AT&T acoustic models. First, there is a params file that tells `amcompile` where it should look for the various acoustic model components:

```
#parameter value
hmms         example.hmms
states         example.states
pdfs         example.pdfs
means         example.means
variances example.vars
state_durations example.durs
```

The HMMs file `example.hmms` defines just one HMM, with two states:

```
#hmm phn/grp state1 state2
0 0 0 1
```

This file specifies that phoneme 0 is implemented using HMM 0. HMM 0 has two states: state 0, and state 1. One can create a vocabulary file, `example.voc`, that specifies the mapping between HTK named macros and ATT numbered macros:

```
~u "meanvec1"
<MEAN> 2
0.3 0.4
~v "varvec1"
<Variance> 2
2.0 1.5
~m "mixmacro1"
<Mean> 2
-0.1 -0.2
~v "varvec1"
~s "statemacro1"
<NumMixes> 2
<Mixture> 1 0.42
~m "mixmacro1"
<Mixture> 2 0.58
<Mean> 2
0.0 0.0
<Variance> 2
1.0 1.0
~t "transition_macro"
<TRANSP> 4
 0.000000e+00 1.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 9.513021e-01 4.869799e-02 0.000000e+00
 0.000000e+00 0.000000e+00 6.631493e-01 3.368506e-01
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
~h "an_hmm"
<BEGINHMM>
<NUMSTATES> 4
<STATE> 2
~s "statemacro1"
<State> 3
<NumMixes> 2
<Mixture> 1 0.23
~m "mixmacro1"
<Mixture> 2 0.77
~u "meanvec1"
~v "varvec1"
~t "transition_macro"
<EndHMM>
```

Figure 1: File "example.mmf:" an example of an MMF with lots of tying.

```
an_hmm 0
statemacro1 0
mixmacro1 0
meanvec1 0
varvec1 0
```

Thus the HMM "an_hmm" is HMM number 0, and the state macro "statemacro1" is state number 0. State number 1 has no name listed in the vocabulary file, because it has no name specified in the HTK MMF.

The States file `example.states` defines two states:

```
#state count pdf1 weight1         pdf2    weight2
0 -1 0 4.200000e-01 1 5.800000e-01
1 -1 0 2.300000e-01 2 7.700000e-01
```

An AT&T parlance "PDF" is the same as an HTK "mixture" – in other words, in AT&T recognizers, a "PDF" is always a Gaussian. The file `example.states` specifies that state 0 uses PDFS 0 and 1 (with mixture weights 0.42 and 0.58), while state 1 uses PDFs 0 and 2. Notice that PDF 0 ("mixturemacro1") has been re-used. The `count` (second column) is the number of times that this state occurred in the training data: since the MMF doesn't give us that information, this column is set to -1.

The PDFs file `example.pdfs` defines three Gaussian PDFs:

```
#pdf mean var basis
0 1 0 0
1 2 1 0
2 0 0 0
```

All three PDFs use different mean vectors, but PDFs 0 and 2 share the same variance vector. All three PDFs use the same identity transformation basis, basis 0. In order to implement non-diagonal covariance matrices, we would define PDF-dependent non-identity transformation basis matrices.

The files `example.means` and `example.vars` define the mean and variance vectors.

```
#mean count coord1          coord2
0 -1 3.000000e-01 4.000000e-01
1 -1 -1.000000e-01 -2.000000e-01
2 -1 0.000000e+00 0.000000e+00
```

`example.vars` looks like this:

```
#var count coord1         coord2
0 -1 2.000000e+00 1.500000e+00
1 -1 1.000000e+00 1.000000e+00
```

The AT&T acoustic models have no representation for transition probabilities. There are at least three ways in which the transition probabilities can be converted.

In order to bring HTK transition probabilities across to the AT&T model exactly, it is necessary to do two things: (1) create a finite state machine representation of the HMM states, (2) run `drecog` with the `model_level` parameter set to `state`; in this case `drecog` will completely ignore `example.hmms`, and will instead use the FSM file to explicitly represent transitions among the specified HMM states. For example, to represent the transition probabilities given above, you could create the file `example_state_transitions.fst.txt`. The file would look like this one, except that the `-log()` functions need to be evaluated to create real numbers:

```
0 0  0 -       -log(9.513021e-01)
0 1  0 -       -log(4.869799e-02)
1 1  1 -       -log(6.631493e-01)
1 2  1 an_hmm  -log(3.368506e-01)
2
```

The input string in this transducer is the state number. The output string is the HMM label — `drecog` actually doesn't care what the output strings are, but you will probably find it useful to set the output string equal to the HMM label, so that this state_transitions FSM can be composed with a context_dependency FSM (and thence with a dictionary, and thence with a grammar). Also, if this method is used (i.e., if all self-loops are explicitly coded in the state_transitions FST), then the state_durations line in `example.params` needs to be excluded: `drecog` should be forced to assume that each transition in the file `state_transitions.fst.txt` lasts exactly one frame.

The second method relies on the internal state-duration models used by `drecog`. Instead of explicitly modeling self-loops, `drecog` explicitly models the duration probability density of each state (or of each HMM). In order to use this algorithm, compute the mean and variance of the duration of each state, and write them to a file `example.durs` specified by the `state_durations` option in `example.params`. If the self-loop probability of a state in HTK is a_{ii}, then the probability that the HMM stays in that state for d_i frames is a geometric PMF:

$$p_i(d_i) = \begin{cases} (1 - a_{ii}) a_{ii}^{d_i - 1} & d_i \geq 1 \\ 0 & d_i \leq 0 \end{cases} \quad (1)$$

The geometric PMF has a mean value and standard deviation that are both equal to $\sigma_{d_i} = \bar{d}_i = 1/(1 - a_{ii})$. Thus we could write the durations file `example.durs` as

```
#state count durmean durvar
0 -1 2.96868 8.81306
1 -1 20.5347 421.674
```

In reality, though, it is almost certainly true that $\sigma_{d_i} < \bar{d}_i$; the duration histograms of most phonetic events are much more compact than Eq. 1. The best thing to do would be to re-estimate `durmean` and `durvar` separately. Assuming that we don't have the facilities to do that, a reasonable kluge/heuristic would be to just cut `durvar` in half:

```
#state count durmean durvar
0 -1 2.96868 4.40653
1 -1 20.5347 210.839
```

The left-to-right, no-skip assumption is true of all standard HMMs created in all example HTK systems, with the sole exception of the special `sp` word-boundary symbol. If `sp` is your only HMM with a skip, you can represent this skip by allowing your speech recognition search graph to skip over the `sp` model as described in the next section.

The third method for representing state transitions in the ATT toolkit is a hybrid of the first two. Explicit state duration probabilities are represented using `example.durs`. In addition, inter-state transitions are explicitly represented using an external `state_transitions.fst.txt`. file. The file `state_transitions.fst.txt` should now be modified to NOT include self-loops — instead, each state in the FST file is allowed to persist for more than one frame. For a strict left-to-right HMM, the resulting `state_transitions.fst.txt` file is very simple:

```
0 1  0 -
1 2  1 an_hmm
2
```

The advantage of this method is that it includes the realistic explicit state durations, but it also allows for non-trivial state transition matrices; e.g., skip transitions and non-trivial loop transitions are possible.

All of the components described above can be created automatically using the perl script `htk2att.pl` distributed with this lecture. `htk2att.pl` writes out an explicit durations file, rather than a state_transitions FSM. It does *not* cut the duration variances in half; if you want to do that, please do it yourself.

After running `htk2att.pl` to create the component files (or after modifying them yourself), call the AT&T `amcompile` program to compile them:

```
amcompile example.params > example.amm
```

1.2 Language Models and Dictionaries

Code has not yet been written for the purpose of generating the context_dependency, dictionary, and language model transducers. This section sketches what needs to be done. The basic structure of these models was discussed in lecture 7.

The context_dependency transducer needs to map from HMM ID numbers (at the input) to an output label set equal to the phoneme units used in your dictionary. Usually we create the transducer in the opposite order (phones in, triphones out), then invert it using `fsminvert`. When you're mapping from an HTK dictionary to HTK triphones, usually the context-dependency transducer should do the following:

- Phone `aa` at the input, preceded by phone `bb` and followed by phone `cc`, is replaced by triphone `bb-aa+cc`. Usually this replacement has a delay of one symbol, as described in lecture 7, and as implemented in the following line from `context_dependency.fst.txt`:

  ```
  bb_aa aa_cc cc bb-aa+cc
  ```

- The special word boundary symbol `sp` is never included in the context; e.g. if `aa` is preceded by `sp` and followed by `cc`, the output symbol should be `aa+cc`, NOT `sp-aa+cc`:

  ```
  bb_aa aa_sp sp bb-aa
  sp_aa aa_cc cc aa+cc
  ```

- When the input symbol is `sp`, the output should also be `sp` (possibly with a one symbol delay). As noted in the last section, you may want to allow your recognizer to skip over the `sp` symbol. This can be done by creating redundant output transitions, one with `sp` at the output, and one with no symbol in the output. For example, in the following FSM snippet, the sequence `bb sp cc` at the input is replaced by either `sp` or `-` in the output:

  ```
  bb_sp sp_cc cc sp
  bb_sp sp_cc cc -
  ```

The dictionary transducer needs to map from phones to words; usually this is created by inverting a words-to-phones transducer. The dictionary transducer is created by taking your input HTK-format dictionary, and running it through a script of your own devising that creates separate transducers for every word in the dictionary. For example, if you intend to create a left-to-right tree-shaped dictionary, the transducer for the word "corn" might be `CORN.fst.txt`:

```
0 1 k -
1 2 ow -
2 3 r -
3 4 n -
4 5 - CORN
5
```

After creating transducers for all words, you should `fsmunion` them, then `fsmrmepsilon`, `fsmdeterminize`, and `fsmminimize`.

The language model transducer is a backed-off N-gram. The HTK program `HLStats`, for example, will compute bigram statistics, and output a backed-off bigram to a text file, in the following format:

```
\data\
ngram 1=3042
ngram 2=8504

\1-grams:
-1.6394 a -0.4632
-4.0973 a.m. -0.2945
```

```
-4.0973 abandoning -0.2999
...

\2-grams:
-1.6450 a year
-0.3010 a.m. and
-0.3010 abandoning its
...
```

This file starts by specifying the number of distinct 1-grams and 2-grams found in the input. Then it lists all of the unique 1-grams and 2-grams. The number in front of each entry is the logarithm of its 1-gram or 2-gram probability. The unigrams are also followed by a number, which is the backoff weight given that word as context. These numbers can be inverted, then copied directly to `bigram.fst.txt`. If `_BACKOFF` is the special backoff state, then `bigram.fst.txt` might look like this:

```
_BACKOFF a   a  1.6394
_BACKOFF a.m.   a.m.   4.0973
_BACKOFF abandoning abandoning   4.0973
a _BACKOFF  -   0.4632
a year     year   1.6450
a.m. _BACKOFF -   0.2945
a.m. and   and  0.3010
abandoning _BACKOFF -   0.2999
abandoning its its   0.3010
```

1.3 Acoustic Features

In order to use `drecog`, acoustic features must be converted from HTK format to raw floating-point format.

First, disable feature compression (set `USE_COMPRESSION` to `FALSE` in your params file; make sure that the parameter type does not include the `_C` option).

Without compression, an HTK file is a series of 4-byte big-endian floating-point numbers, preceded by a 12-byte header. Verify this: calculate the number of features in a file (number of frames times number of elements per frame), and verify that the file size in bytes (`ls -l`) is four times the matrix size, plus 12.

Use the unix utility `dd` to strip off the header. Type `man dd` to see its man page. You can specify that it should read the input in 12-byte blocks, and skip the first block:

```
dd if=features.htk of=features.bigendian bs=12 skip=1
```

Since most of our machines are linux (little-endian), you will also need to byte-swap the feature files (HTK files are always big-endian, no matter what machine you're on; raw files are assumed to be in the format native to the machine you're using). `dd` will byte-swap two-byte numbers (short integers), but not 4-byte numbers (floats). You can byte-swap short integers using matlab, or C, or perl; the goal is simply to read four bytes, then write them out in reverse order. For example, the following command-line perl script should do it:

```
perl -e 'while(!feof(STDIN)){
    fread(STDIN,$a,1);fread(STDIN,$b,1);
    fread(STDIN,$c,1);fread(STDIN,$d,1);
    print STDOUT $d,$c,$b,$a;
}' < features.bigendian > features.raw
```

2 Speech Recognition Using fsmcompose

Most speech recognition tools are written in order to be as fast as possible, because without these speedups, recognition of a short utterance can take days (before optimization, one of our systems at WS04 was running at 50-100 times real time, meaning that 3.5 hours of data would take 6 weeks to recognize; optimization

reduced this number to about 10 times real time). The problem with fast code is that it is pedagogically useless; none of the programs we've talked about so far would help you to learn the nuts and bolts of modern ASR algorithms.

FSM tools seem to be an exception. Speech recognition, in FSM-speak, is the process of (1) computing observation probabilities, and then (2) composing observation probabilities with the search graph.

2.1 Compute Observation Probabilities

Observation probabilities are computed in every frame, for every state in the recognizer. Thus, if state q is composed of the mixtures k, for $1 \leq k \leq K$, then for each time t, you want to compute:

$$-\log p(\vec{x}_t|q) = -\log\left(\sum_{k=1}^{K} c_k \left(\frac{1}{\sqrt{(2\pi)^d|\Sigma_{qk}|}} e^{-0.5(\vec{x}_t-\mu_{qk})^T\Sigma_{qk}^{-1}(\vec{x}_t-\mu_{qk})}\right)\right) \tag{2}$$

These probabilities define a very simple sort of finite state machine. Each observation frame is a set of FSM transitions: one transition per HMM state. Thus the total number of edges is equal to the number of states in the HMM times the number of frames in the observation. This number of edges might be pruned, if you want.

Here is an example. This utterance has 120 frames (1.2 seconds long), so the observations are numbered x1 through x120, and the FSM states are numbered frame1 through frame120, followed by the special state utterance_end. The probabilities $-\log p(x_t|q)$ should be substituted by the numbers that you actually computed, using Eq. 2 (I assume that you can write your own code to compute Eq. 2). The states are numbered aa-aa+aa1 (first state of HMM "aa-aa+aa") through zh3 (third state of HMM "zh;" the monophone state "zh3" sorts lexicographically after all of its associated triphone states, because "3" sorts after "-"):

```
frame1 frame2 aa-aa+aa1 -logp(x1|aa1)
frame1 frame2 aa-aa+aa2 -logp(x1|aa2)
...
frame1 frame2 zh3 -logp(x1|zh3)
frame2 frame3 aa-aa+aa1 -logp(x2|aa1)
frame2 frame3 aa-aa+aa2 -logp(x2|aa2)
...
frame120 utterance_end zh2 -logp(x120|zh2)
frame120 utterance_end zh3 -logp(x120|zh3)
utterance_end
```

2.2 Compose with the Search Graph

The methods of Sec. 1.2 resulted in a finite state transducer that maps from HMM states (input symbols of the FST) to words (output symbols of the FST). This complete FST is sometimes called the "speech recognition search graph," and it might look something like this (where the costs shown here are just made-up numbers, but should include the results of any log probabilities derived from the language model and the dictionary and the HMM transition probabilities):

```
0 1 aa1:- 0.5938
0 2 ah1:- 0.2849
0 3 ah+b1:-  0.3459
...
1 1 aa1:- 0.12873
2 2 ah1:- 0.918723
3 3 ah+b1:-  0.9732
...
1 65 aa2:- 0.9837
2 66 ah2:- 0.8712
3 67 ah+b2:-  0.189
```

```
...
65 65 aa2:- 0.2387
66 66 ah2:- 0.29037
67 67 ah+b2:-  0.92873
...
67 152 aa3:AH 0.09387
68 153 ah3:UH 0.09873
69 154 ah+b3:-  0.7682
...
```

Notice that the example shown above includes explicit self-loops for every HMM state. Thus it's possible for "aa2" to be the string observed for two frames in a row; it's also possible to transition from "aa2" to "aa3."

The search graph summarizes all of your prior knowledge about the English language. In the terms often used in speech recognition, this graph summarizes the *a priori* probability $p(Q, W)$ of state sequence $Q = [q(t = 1), \ldots, q(t = T)]$ matching word sequence $W = [w_1, \ldots, w_M]$. The transcription graph computed in the previous section is a summary of the likelihoods $p(X|Q)$. By multiplying these two probabilities (adding their log probabilities), we get the joint probability of the observation, state sequence, and word sequence, $p(X, Q, W)$. The *a posteriori* probability is related to the joint probability by a term that does not depend on W:

$$-\log p(W|X) = -\log p(X, Q, W) + \log p(X, Q) \tag{3}$$

Therefore, if the goal is to find the word sequence that maximizes $p(W|X)$, it is sufficient to maximize $p(X, Q, W)$.

$p(X, Q, W)$ is computed, for every path $Q : W$, by multiplying the two path probabilities recorded in the transcription graph and in the search graph. These two probabilities may be multiplied by composing the two graphs:

```
fsmcompose transcription.fst search.fst > jointprob.fst
```

The speech recognition output is then the best path (the path with minimum cost = maximum probability) through `jointprob.fst`:

```
fsmbestpath jointprob.fst | fsmprint -i states.voc -o words.voc -s frames.voc
```

3 Speech Recognizer Training Using fsmcompose

Speech recognizer training uses either the Baum-Welch algorithm [3, 1] or likelihood backpropagation [2] or some combination of these things. In both of these algorithms, the time-consuming step is the computation of the posterior probability or "gamma probability" at each time:

$$\gamma_t(q) = p(q_t = q|x_1, \ldots, x_T) \tag{4}$$

This is exactly the same probability you computed in Sec. 2. Thus speech recognizer training has two steps: (1) compute $\gamma_t(q)$ using the methods of Sec. 2, (2) update all of the speech recognizer parameters using the Baum or likelihood-backprop equations, given in other textbooks.

This lecture will not dwell further on step 2, because it would take us far afield, and because other good textbooks exist (for the Baum algorithm, at any rate). Step 1, however, involves a few extra wrinkles worth considering briefly.

First, the search graph used in Sec. 2 didn't assume any prior knowledge about the particular waveform being recognized (thus it would be appropriate for unsupervised training of an automatic speech recognizer). In most large ASR training corpora, you don't know where individual words start and end, much less phonemes; you do know, however, which words were uttered in each waveform file (courtroom reporters and similar transcription professionals can create transcripts of this kind very efficiently). When knowledge is available, you should always use it. In this case, you can use it by replacing the general-purpose language model with an utterance-specific language model, also known as a "transcript:"

```
0 1 HI
1 2 MY
2 3 NAME
3 4 IS
4 5 MARK
5
```

...or the like. Notice that there are no probabilities listed; all transitions have probability of exactly 1.0 (for this particular utterance). This transcription is composed with the dictionary, context-dependency transducer, and HMM-to-state transducer in order to create a file-specific search graph, which is then composed with the observation graph.

Second, ASR training algorithms usually require not just the posterior probability of the state; they usually require the posterior probability of the state and its mixture, $p(q_t = q, k_t = k|X)$. This posterior probability can be computed with two additional steps. First, instead of computing the observation graph with state likelihoods, compute it using mixture likelihoods:

$$-\log p(x_t|q_t = q, k_t = k) = \frac{1}{2}\left(d\log(2\pi) + \log(|\Sigma_{qk}|) + (\vec{x}_t - \mu_{qk})^T\Sigma_{qk}^{-1}(\vec{x}_t - \mu_{qk})\right) \tag{5}$$

Second, compose the search graph with the inverse of a states-to-mixtures graph. This is a simple graph that says simply that a state can always be instantiated with any one of its mixtures, with a probability given by the mixture weight c_{qk}. For example, here is the FSM for a model with 13 mixtures per state:

```
0 0 aa-aa+aa1:aa-aa+aa1_mix1  -log(c1(aa-aa+aa1))
0 0 aa-aa+aa1:aa-aa+aa1_mix2  -log(c2(aa-aa+aa1))
...
0 0 aa-aa+aa1:aa-aa+aa1_mix13  -log(c13(aa-aa+aa1))
0 0 aa-aa+aa2:aa-aa+aa2_mix1  -log(c1(aa-aa+aa2))
...
0 0 zh3:zh3_mix13 -log(c13(zh3))
```

The new search graph maps from the known word sequence (specified by the courtroom reporter) down to the various mixtures that might spell it out, frame by frame. Composing this with the new likelihood FST will result in a `jointpdf.fst` that specifies the posterior probability of every mixture in every frame.

References

[1] Leonard E. Baum and J. A. Eagon. An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bull. Am. Math. Soc.*, 73:360–363, 1967.

[2] Y. Bengio, R. De Mori, G. Flammia, and R. Kompe. Global optimization of a neural network - hidden markov model hybrid. *IEEE Trans. Neural Networks*, 3(2):252–259, 1992.

[3] Stephen E. Levinson. *Mathematical Models of Language*. John Wiley and Sons, 2005.