Speech Recognition

- a practical guide
Lecture 4

Decoding
"Decoding" means: given a model and an utterance, give me the most likely word-sequence.

Difficult to do efficiently.
Viterbi algorithm

Given a HMM and a sequence of output symbols (or vectors, in our case)...

Give me the state-sequence through the HMM that contributes the most to the likelihood.

Algorithm takes time $O(N T)$, where $N$ is #arcs in HMM, $T$ is #observations.

Simple dynamic programming: for each (state, time $t$), keep record of best state at $t-1$
Viterbi algorithm-- application

We’re actually searching a very large “HMM” that contains the information in:

- The lexicon (pronouncing dictionary)
- The language model
- The phonetic decision tree
- The topologies of the phone HMMs

We’ll use the Weighted Finite State Transducer (WFST) framework to combine them...
The graph we need to search is too big—would be very slow with standard Viterbi.

Beam-pruning means we access the frames (feature vectors) one by one, and on each frame, prune away states with score worse than (best-score minus beam).

For reasonable beam values, we won’t lose too much accuracy by doing this.

This is the most basic component of a speech-recognition decoder.
This script calls scripts/mkgraph.sh to make the “decoding graph”

Then it decodes the different test sets that Resource Management comes with.
"Prior" probability of sentences are often given by N-gram language model (for large vocabulary tasks).

Read “A Bit of Progress in Language Modeling” (Joshua Goodman) for intro.

Can represent this in a graph structure using the Weighted Finite State Transducer (WFST) framework.

Gives a very large graph!
Language models

- A “backoff language model” is the standard type of LM.

- Typically represented in the “ARPA” or equivalently “arpabo” format (look it up).

- In the bigram case (one word of history):
  - If last word was “b”, we’ll have a number of successor-words with explicit probabilities given (e.g. b->c, b->d)...
  - We’ll also “back off” with some weight, to the unigram distribution-- so all successor-words have a nonzero probability.
Language models: ARPA

- $ cd ~/kaldi-trunk/egs/wsj/s3
- $ less data/local/lm_bg.arpa.gz

\data\
 ngram 1=19982
 ngram 2=1460564

\1-grams:
-1.7282 <UNK> -0.7021
-1.3780 </s> 0.0000
-99.999 <s> -1.5255
-5.2162 'EM -0.3535
-5.0255 'N -1.1501
-1.6566 A -1.8164
-5.5854 A'S -0.1161
-2.8439 A -0.9302

<snip>

\2-grams:
-1.3457 <UNK> <UNK>
-1.1291 <UNK> </s>
-4.5784 <UNK> 'EM
-4.2774 <UNK> 'N

- Has log-probabilities in log-base 10 (backoff weights at end of line, in log-base-10).

- This example from WSJ (no ARPA in RM)
Language models, graph
Language models, graph

- Graph for LM has separate state for each seen history-state

  - For trigram LM, a history state would correspond to a pair of words

- Also has states for less-specific history states (corresponding to single words), and unigram-backoff state (the "empty history")

- Each state (except unigram state) has "backoff arc" to less specific history state

  - Backoff-arc has epsilon on it.
This is a more “grammar-like” example (not a backoff LM)

Taken from Mohri’s “Springer Handbook” article (search for hbka.pdf)
RM has a simple word-pair grammar instead of a "proper" language model (it comes with this; it’s conventional to use this in testing).
Gives for each word, a set of alternative pronunciations (possibly with weights)

This example is also from Mohri’s article

In Kaldi, the lexicon also encodes optional silences between words.
This is the original file from which we create the lexicon (it’s created by a human).
We require each pronunciation to be unique.

This relates to issues of WFSTs (determinizability).

Add “fake phones” #1, #2, ... to pronunciations.
This script takes the lexicon in text form and outputs as a WFST (in text format)

Handles adding optional silences (a bit like alternative pronunciations)
We use a transducer (C) to handle phonetic context dependency.

It expands phones into context-dependent phones (i.e. triphones, in the normal case)

In our case the phones are on the right (output symbols), context-dep. on left (input)
Context-dependency

A few extra details to be aware of:

The symbol $ is the “end-of-sentence symbol” -- needed to handle issues relating to determinizability.

We need self-loops on all states in C, with the disambiguation symbols #0, #1, ... on.

Want it deterministic on phones for fast composition... as a result, input and output are not “synced up”: there is a delay of 1 (for triphone case)
Overall recipe

- Standard WFST recipe is “HCLG”
- Composition of:
  - H = HMM structure
  - C = phonetic context dependency
  - L = lexicon
  - G = grammar (or LM).
- Optimizations such as determinization, minimization, weight-pushing done to improve beam-pruning performance.
First set up some variables relating to probability scales

Relates to scaling the transition probabilities.

We treat self-loops differently.
Making the graph: LG

Compose L and G, then determinize and minimize.

Then check result is stochastic (probabilities out of each state sum to one)

actually it won’t be, quite.

These programs are similar to OpenFst programs, but are all Kaldi programs.
Purpose of each stage

- "compose" (of L and G) expands each word into phones.
- "determinize" is like making a tree-structured lexicon (in each language-model state)
- "minimize" is like suffix-sharing
Kaldi FST tools

- We use a combination of OpenFst’s native command-line tools, and Kaldi versions of them
- fsttablecompose like fstcompose but faster (uses heuristics that OpenFst authors would find ugly)
- fstminimizeencoded is a convenience mechanism (would require several native OpenFst commands)
- fstdeterminizestar is like fstdeterminize but does epsilon-removal as part of determinization.

Both for mathematical and practical reasons.
Compose C with LG to make CLG

C is created “on the fly” in memory (we only expand the parts we need).

Also outputs file “ilabels_3_1”, which maps the (integer) input symbols of CLG to triphones.

```
$ less scripts/mkgraph.sh
<snip>
grep '#' $lang/phones_disambig.txt | awk '{print $2}' > $lang/tmp/disambig_phones.list
clg=$lang/tmp/CLG_${N}_${P}.fst

fstcomposecontext --context-size=${N} --central-position=${P} \
   --read-disambig-syms=$lang/tmp/disambig_phones.list \
   --write-disambig-syms=$lang/tmp/disambig_ilabels_${N}_${P}.list \
   $lang/tmp/ilabels_${N}_${P} $lang/tmp/LG.fst >$clg

fstisstochastic $clg || echo "warning: CLG not stochastic."
```
H is the “HMM-structure” transducer (info from HMM topologies, transition-probs, decision-tree)

Ha.fst is as H.fst, but no self-loops (more compact)

Input symbols are “transition-ids” which encode the “pdf-id” (id of the mixture of Gaussians) and also a bit more information (e.g. the phone)

Only includes triphones listed in ilabels_3_1
Making the graph: HCLG

Compose Ha.fst with CLG.fst

Remove epsilons and determinize [fstdeterminizestar]...

Do this in log semiring, which keeps the weights suitably "sum-to-one".

Remove disambiguation symbols #0, #1, ...

Remove any resulting epsilons and minimize.
Weight-pushing

- This was part of the originally published recipe (search for “hbka.pdf”)

- Gives better pruning behavior... make graph like properly normalized HMM.

- Can fail (crash, or give weird results) for ARPA-type LMs.

- We ensure this property “stochasticity” differently in our recipe...

- Idea is to ensure no stage will destroy stochasticity. Robust to not-quite-stochastic LMs.
Adding self-loops

This stage adds the self-loops to the final graph.

Corresponds to the self-loops of the HMM states.

Doing this as a separate stage allows us to build larger graphs.

Note: the --reorder option puts the self-loop after the transition out of the state.

Leads to faster-to-search graph.
Running the decoding

Run the decoding in the background.

Note: it uses about 6 cpus, for about 2 minutes

the 6 test sets are decoded in parallel
Look at the decoding command in a log file

Try running it from the command line (copy and paste)

You’ll have to source “path.sh” (type “. path.sh”)

Change the name of the “lat.gz” output or you’ll interfere with the decoding that’s running.
We actually don’t do Viterbi, we generate “lattices” (a graph-based record of the most likely utterances)

These are later rescored at various acoustic weights, and we pick the best.

The option for the acoustic scale during lattice generation only affects pruning behavior.

```bash
$ head exp/tri1/decode/feb89/decode.log
gmm-latgen-simple --beam=20.0 --acoustic-scale=0.1 --word-symbol-table=data/lang/words.txt exp/tri1/final.mdl exp/tri1/graph/HCLG.fst
'ark:compute-cmvn-stats --spk2utt=ark:data/test_feb89/spk2utt scp:data/test_feb89/feats.scp ark:- | apply-cmvn --norm-vars=false --utt2spk=ark:data/test_feb89/utt2spk ark:- scp:data/test_feb89/feats.scp ark:- | add-deltas ark:- ark:- | ' 'ark:|gzip -c > exp/tri1/decode/feb89/lat.gz'
```
Changing the beam

Change the beam width in the command from 20 to 10 and run again.

Notice the change in speed.

When the decoding finishes, change the beam width in the script to 10 and rerun (change exp/tri/decode to exp/tri/decode_10).

Check the effect on speed and accuracy.

```
$ head exp/tri1/decode/feb89/decode.log
gmm-latgen-simple --beam=10.0 --acoustic-scale=0.1 --word-symbol-table=data/lang/words.txt exp/tri1/final.mdl exp/tri1/graph/HCLG.fst
'ark:compute-cmvn-stats --spk2utt=ark:data/test_feb89/spk2utt scp:data/test_feb89/feats.scp ark:- | apply-cmvn --norm-vars=false --utt2spk=ark:data/test_feb89/utt2spk ark:- scp:data/test_feb89/feats.scp ark:- | add-deltas ark:- ark:- |' 'ark:|gzip -c > exp/tri1/decode/feb89/lat.gz'
```
# The decoder code

```cpp
$ cd ~/kaldi-trunk/src
$ less gmmbin/gmm-decode-simple.cc

DecodableAmDiagGmmScaled gmm_decodable(am_gmm, trans_model,
features, acoustic_scale);

decoder.Decode(&gmm_decodable);
```

- `gmm-decode-simple` just does beam-pruned Viterbi
- The “`gmm_decodable`” object is a simple interface to the acoustic model and features.
- It has a function that returns the log-likelihood for a given (frame, pdf-id)
Decoder code (internal)

$ cd ~/kaldi-trunk/src
$ less decoder/simple-decoder.h
<snip>
class SimpleDecoder {
<snip>
  bool Decode(DecodableInterface *decodable) {
    // clean up from last time:
    ClearToks(cur_toks_);
    ClearToks(prev_toks_);
    StateId start_state = fst_.Start();
    KALDI_ASSERT(start_state != fst::kNoStateId);
    Arc dummy_arc(0, 0, Weight::One(), start_state);
    cur_toks_[start_state] = new Token(dummy_arc, NULL);
    ProcessNonemitting();
    for (int32 frame = 0; !decodable->IsLastFrame(frame-1); frame++) {
      ClearToks(prev_toks_);
      cur_toks_.swap(prev_toks_);
      ProcessEmitting(decodable, frame);
      ProcessNonemitting();
      PruneToks(beam_, &cur_toks_);
    }
  }
}
Other decoders

Apart from simple-decoder.h (“internal” decoder code), there are:

- faster-decoder.h [a bit more optimized]
- lattice-faster-decoder.h [that also generates lattices]

A couple more that are less important.

Each model-type has a command-line wrapper for each decoder (e.g. gmm-latgen-faster)
End of this lecture