Preparing the initial symbol tables

We need to prepare the OpenFst symbol tables **words.txt** and **phones.txt**. These assign integer id's to all the words and phones in our system. **Note that OpenFst reserves symbol zero for epsilon**. An example of how the symbol tables look for the WSJ task is:

```
## head words.txt
<eps> 0
!SIL 1
<s> 2
</s> 3
<SPOKEN_NOISE> 4
<UNK> 5
<NOISE> 6
!EXCLAMATION-POINT 7
"CLOSE-QUOTE 8

## tail -2 words.txt
}RIGHT-BRACE 123683
#0 123684

## head data/phones.txt
<eps> 0
SIL 1
SPN 2
NSN 3
AA 4
AA_B 5
```

The words.txt file contains the single disambiguation symbol "#0" (used for epsilon on the input of G.fst). This is the last-numbered word in our recipe. Be careful with this if your lexicon contains a word "#0". The phones.txt file does not contain disambiguation symbols but after creating L.fst we will create a file phones_disambig.txt that has the disambiguation symbols in (this is just useful for debugging).

Preparing the lexicon L

First we create a lexicon in text format, initially without disambiguation symbols. Our C++ tools will never interact with this, it will just be used by a script that creates lexicon FST. A small part of our WSJ lexicon is:

```
## head data/lexicon.txt
!SIL SIL
<s>
</s>
<SPOKEN_NOISE> SPN
<UNK> SPN
<NOISE> NSN
!EXCLAMATION-POINT EH2_B K S K L AH0 M EY1 SH AH0 N P OY2 N T_E
"CLOSE-QUOTE K_B L OW1 Z K W OW1 T_E
```

The beginning, ending and stress重音 markers on the phones (e.g. T_E, or AH0) are specific to our WSJ recipe and as far as our toolkit is concerned, they are treated as distinct phones (however, we do handle the tree-building specially for this setup; read about the roots file in **The tree building process**).

Notice that we allow words with empty phonetic representations. **This lexicon will be used to create the L.fst used in training** (**without disambiguation symbols**). We also create a lexicon with disambiguation symbols, used in decoding-graph creation. An extract of this file is here:

```
# [from data/lexicon_disambig.txt]
!SIL    SIL
<s> #1
</s>    #2
```

```
<SPOKEN_NOISE>  SPN #3
<UNK>   SPN #4
<NOISE> NSN
...
{BRACE  B_B R EY1 S_E #4
{LEFT-BRACE L_B EH1 F T B R EY1 S_E #4
```

This file is created by a script; the script outputs the number of disambiguation symbols it had to add, and this is used to create the symbol table phones_disambig.txt. This is the same as phones.txt but it also contains the integer id's for the disambiguation symbols #0, #1, #2 and so on (#0 is a special disambiguation symbol which comes from G.fst but will be "passed through" L.fst via self-loops). A section of the middle of the file phones_disambig.txt is:

```
ZH_E 338
ZH_S 339
#0  340
#1  341
#2  342
#3  343
```

The numbers are so high because in this (WSJ) recipe we added stress and position information to the phones. Note that the disambiguation symbols used for the empty words (i.e. <s> and </s>) have to be distinct from those used for the normal words, so the "normal" disambiguation symbols in this example start from #3.

The command to convert the lexicon without disambiguation symbols into an FST is:

```
scripts/make_lexicon_fst.pl data/lexicon.txt 0.5 SIL | \
  fstcompile --isymbols=data/phones.txt --osymbols=data/words.txt \
  --keep_isymbols=false --keep_osymbols=false | \
   fstarcsort --sort_type=olabel > data/L.fst
```

不加**disambig_symbol**，就不用添加**fstaddselfloops**

Here, the script **make_lexicon_fst.pl** creates the text representation of the FST. The **0.5 is the silence probability** (i.e. at the beginning of sentence and after each word, we output silence with probability 0.5; the probability mass assigned to having no silence is 1.0 - 0.5 = 0.5. The rest of the commands in this example relate to converting the FST into compiled form; **fstarcsort is necessary because we are going to compose later.**

The structure of the lexicon is roughly as one might expect. There is one state (the "loop state") which is final. There is a start state that has two transitions to the loop state: one with silence and one without. From the loop state there is a transition corresponding to each word, and that word is the output symbol on the transition; the input symbol is the first phone of that word. It is important both for the efficiency of composition and the effectiveness of minimization that the output symbol should be as early as possible (i.e. at the beginning not the end of the word). At the end of each word, to handle optional silence, **the transition corresponding to the last phone is in two forms: one to the loop state and one to the "silence state" which has a transition to the loop state.** We don't bother putting optional silence after silence words, which we define as words that have just one phone that is the silence phone.

Creating the lexicon with disambiguation symbols is just slightly more complicated. The issue is that we have to add the self-loops to the lexicon so that the disambiguation symbol #0 from G.fst can be passed through the lexicon. We do this with the program **fstaddselfloops** (**c.f. Adding and removing disambiguation symbols**), although we could easily have done it "manually" in the script make_lexicon_fst.pl.

```
phone_disambig_symbol=`grep \#0 data/phones_disambig.txt | awk '{print $2}'`
word_disambig_symbol=`grep \#0 data/words.txt | awk '{print $2}'`

scripts/make_lexicon_fst.pl data/lexicon_disambig.txt 0.5 SIL  | \
   fstcompile --isymbols=data/phones_disambig.txt --osymbols=data/words.txt \
   --keep_isymbols=false --keep_osymbols=false |   \
   fstaddselfloops  "echo $phone_disambig_symbol |" "echo $ewq |" | \
   fstarcsort --sort_type=olabel > data/L_disambig.fst
```

The program fstaddselfloops is not one of the original OpenFst command-line tools, it is one of our own (we have a number of such programs).

Preparing the grammar G

The grammar G is for the most part an acceptor (i.e. input and output symbols are identical on each arc) with words as its symbols. The exception is the disambiguation symbol #0 which only appears on the input side. Assuming the input is

an Arpa file, we use the Kaldi program **arpa2fst** to convert it to an FST. The program arpa2fst outputs an FST with embedded symbols. In Kaldi we generally use FSTs without embedded symbols (i.e. we use separate symbol tables).

The steps we have to do aside from just running arpa2fst are as follows:

- **We have to remove the embedded symbols from the FST (and rely on the symbol tables on disk).**
- **We have to make sure there are no out-of-vocabulary words in the language model**
- **We have to remove "illegal" sequences of the start and end-of-sentence symbols, e.g. <s> followed by </s>, because these cause L o G to be non-determinizable.**
- **We have to replace epsilons on the input side with the special disambiguation symbol #0.**

A slightly simplified version of the actual script that does this is as follows:

```
  gunzip -c data_prep/lm.arpa.gz | \
     scripts/find_arpa_oovs.pl data/words.txt  > data/oovs.txt

  gunzip -c data_prep/lm.arpa.gz | \
    grep -v '<s> <s>' | \
    grep -v '<s> </s>' | \
    grep -v '</s> </s>' | \
     ../src/lm/arpa2fst - | fstprint | \
    scripts/remove_oovs.pl data/oovs.txt | \
    scripts/eps2disambig.pl | \
     fstcompile --isymbols=data/words.txt --osymbols=data/words.txt \
     --keep_isymbols=false --keep_osymbols=false > data/G.fst
  fstisstochastic data/G.fst
```

The last command (fstisstochastic) is a diagnostic step (see **Preserving stochasticity and testing it**). In one typical example, it prints out the numbers:

```
9.14233e-05 -0.259833
```

The first number is small, so it confirms that there is no state that has the probability mass of its arcs plus final-state significantly less than one. The second number is significant, and this means that there are states that have "too much" probability mass (the numeric values of the weights in the FSTs can generally be interpreted as negated log probabilities). Having some states with "too much" probability mass is normal for the FST representations of language models with backoff. During later graph creation steps we will be verifying that this non-stochasticity has not become worse than it was at the start.

The resulting FST G.fst is of course only used in test time. In training time we use linear FSTs generated from the training word-sequences, but this is done inside Kaldi processes, not at the script level.

Preparing LG

When composing L with G, we adhere in outline to a fairly standard recipe, i.e. we compute min(det(L o G)). The command line is as follows:

```
    fsttablecompose data/L_disambig.fst data/G.fst | \
        fstdeterminizestar --use-log=true | \
        fstminimizeencoded  > somedir/LG.fst
```

There are some small differences from OpenFst's algorithms. We use a more efficient composition algorithm (see **Composition**) implemented by our command-line tool "**fsttablecompose**". Our determinization is an algorithm that also removes epsilons, implemented by the command-line program **fstdeterminizestar**. The option **–use-log=true** asks the program to first cast the FST to the log semiring; this preserves stochasticity (in the log semiring); see **Preserving stochasticity and testing it**.

We do minimization with the program "**fstminimizeencoded**". This is mostly the same as the version of OpenFst's minimization algorithm that applies to weighted acceptors; the only change relevant here is that it avoids pushing weights, hence preserving stochasticity (see **Minimization** for details).

Preparing CLG

To get a transducer whose inputs are context-dependent phones, we need to prepare an FST called CLG, which is equivalent to C o L o G, where L and G are the lexicon and grammar and C represents the phonetic context. For a triphone system, the input symbols of C would be of the form a/b/c (i.e. triples of phones), and the output symbols

would be single phones (e.g. a or b or c). See **Phonetic context windows** for more context on the phonetic context windows, and how we generalize to different context sizes. Firstly, we describe how we would create the context FST C if we were to make it by itself and compose normally (our scripts do not actually work this way, for efficiency and scalability reasons).

Making the context transducer

In this section we explain how we can obtain C as a standalone FST.

The basic structure of C is that it has states for all possible phone windows of size N-1 (c.f. **Phonetic context windows**; N=3 in the triphone case N=3为三因素的状态，N表示因素的个数). The first state, meaning begin-of-utterance, would just correspond to N-1 epsilons. Each state has a transition for each of the phones (let's forget about self-loops for now). As a generic example, state a/b has a transition with c on the output and a/b/c on the input, going to state b/c. There are special cases at the begin and end of utterance.

At the beginning of utterance, suppose the state is <eps>/<eps> and the output symbol is a. Normally, the input symbol would be <eps>/<eps>/a. But this doesn't represent a phone since (assuming P = 1 表示中心因素), the central element is <eps> which is not a phone. In this case we let the input symbol of the arc be #-1 which is a special symbol we introduce for this purpose (we don't use epsilon here as the standard recipe does, as it can lead to nondeterminizability when there are empty words).

The end-of-utterance case is a little complicated. The context FST has, on the right (its output side), a special symbol $ that occurs at the end of utterances. Consider the triphone case. At the end of utterance, after seeing all symbols we need to flush out the last triphone (e.g. a/b/<eps>, where <eps> represents undefined context). The natural way to do this would be to have a transition with a/b/<eps> on its input and <eps> on its output, from the state a/b to a final state (e.g. b/<eps> or a special final state). But this is inefficient for composition, because if it was not the end of the utterance we would have to explore such transitions before finding them pruned away. Instead we use $ as the end-of-utterance symbol, and make sure it appears once at the end of each path in LG. Then we replace <eps> with $ on the output of C. **In general the number of repetitions of $ is equal to N-P-1.** In order to avoid避免 the hassle麻烦 having to work out how many subsequential后续 symbols to add to LG, we just allow允许 it to accept any number of such symbols at the end of utterance. This is acheived实现 by the function **AddSubsequentialLoop()** and the command-line program fstaddsubsequentialloop.

If we wanted C on its own, we would first need a list of disambiguation symbols; and we would also need to work out an unused symbol id we could use for the subsequential symbol, as follows:

```
 grep '#' data/phones_disambig.txt | awk '{print $2}' >
$dir/disambig_phones.list
 subseq_sym=`tail -1 data/phones_disambig.txt | awk '{print $2+1;}'`
```

We could then create C with the following command (however, see below regarding fstcomposecontext; we don't do this in practice as it is inefficient).

```
 fstmakecontextfst --read-disambig-syms=$dir/disambig_phones.list \
 --write-disambig-syms=$dir/disambig_ilabels.list data/phones.txt $subseq_sym \
   $dir/ilabels | fstarcsort --sort_type=olabel > $dir/C.fst
```

The program **fstmakecontextfst needs the list of phones, a list of disambiguation symbols and the identity of the subsequential symbol.** In addition to除了 C.fst, it writes out the file "ilabels" that interprets the symbols on the left of C.fst (see **The ilabel_info object**). The composition with LG can be done as follows:

```
fstaddsubsequentialloop $subseq_sym $dir/LG.fst | \
 fsttablecompose $dir/C.fst - > $dir/CLG.fst
```

For printing out C.fst and anything using the same symbols that index "ilabels", we can make a suitable symbol table using the following command:

```
 fstmakecontextsyms data/phones.txt $dir/ilabels > $dir/context_syms.txt
```

Create input symbols for CLG 上述命为创建CLG产生input symbols

This command knows about the "ilabels" format (**The ilabel_info object**). An example random path through the CLG fst (for Resource Management), printed out with this symbol table, is as follows:

```
## fstrandgen --select=log_prob $dir/CLG.fst | \
   fstprint --isymbols=$dir/context_syms.txt --osymbols=data/words.txt -
```

```
0   1   #-1 <eps>
1   2   <eps>/s/ax  SUPPLIES
2   3   s/ax/p  <eps>
3   4   ax/p/l  <eps>
4   5   p/l/ay  <eps>
5   6   l/ay/z  <eps>
6   7   ay/z/sil    <eps>
7   8   z/sil/<eps> <eps>
8
```

/*我认为创建C有很多种方法，但是用到的文件就那么几个，

disambig_phones.list

disambig_ilabels.list

ilabels

phones.txt

*/

Composing with C dynamically

In the normal graph creation recipe, we use a program **fstcomposecontext** which dynamically creates the needed states and arcs of C without wastefully creating it all. The command line is:

```
fstcomposecontext  --read-disambig-syms=$dir/disambig_phones.list \
                   --write-disambig-syms=$dir/disambig_ilabels.list \
                   $dir/ilabels < $dir/LG.fst >$dir/CLG.fst
```

If we had different context parameters N and P than the defaults (3 and 1), we would supply extra options to this program. This program writes the file "ilabels" (see **The ilabel_info object**) which interprets the input symbols of CLG.fst. The first few lines of an ilabels file from the Resource Management recipe are as follows:

```
65028 [ ]
[ 0 ]
[ -49 ]
[ -50 ]
[ -51 ]
[ 0 1 0 ]
[ 0 1 1 ]
[ 0 1 2 ]
...
```

The number 65028 is the number of elements in the file. Lines like [-49] are for disambiguation symbols; lines like [0 1 2] represent acoustic contexts windows; the first two entries are [ ] which is for epsilon (never used), and [0] which is for the special disambiguation symbol with printed form #-1 that we use at the beginning of C in place of epsilon, to ensure determinizability.

Reducing the number of context-dependent input symbols

After creating CLG.fst, there is an optional graph creation stage that can reduce its size. We use the program **make-ilabel-transducer**, which works out from the decision tree and the HMM topology information, which subsets of context-dependent phones would correspond to the same compiled graph and can therefore be merged (we pick an arbitrary element of each subset and convert all context windows to that context window). This is a similar concept to HTK's logical-to-physical mapping. The command is:

```
 make-ilabel-transducer --write-disambig-
syms=$dir/disambig_ilabels_remapped.list \
  $dir/ilabels $tree $model $dir/ilabels.remapped > $dir/ilabel_map.fst
```

This program requires the tree and the model; it outputs a new ilabel_info object called "ilabels.remapped"; this is in the same format as the original "ilabels" file, but has fewer lines. The FST "ilabel_map.fst" is composed with CLG.fst and remaps the labels. After doing this we determinize and minimize so we can immediately realize any size reductions:

```
fstcompose $dir/ilabel_map.fst $dir/CLG.fst  | \
  fstdeterminizestar --use-log=true | \
  fstminimizeencoded > $dir/CLG2.fst
```

For typical setups this stage does not actually reduce the graph size by very much (5% to 20% reduction is typical), and in any case it is only the size of intermediate graph-creation stages that we are reducing by this mechanism. But the savings could become significant for systems with wider context.

Making the H transducer

In the conventional FST recipe, the H transducer is the transducer that has, on its output, context dependent phones, and on its input, symbols representing acoustic states. In our case, the symbol on the input of H (or HCLG) is not the acoustic state (in our terminology, the pdf-id) but instead something we call the transition-id (see **Integer identifiers used by TransitionModel**). The transition-id encodes the pdf-id plus some other information including the phone. Each transition-id can be mapped to a pdf-id. The H transducer that we create does not encode the self-loops. These are added later by a separate program. The H transducer has a state that is both initial and final, and from this state there is a transition for every entry but the zeroth one in the ilabel_info object (the ilabels file, see above). The transitions for the context dependent phones go to structures for the corresponding HMMs (lacking self-loops), and then back to the start state. For the normal topology, these structures for the HMMs would just be linear sequences of three arcs. H also has self-loops on the initial state for each of the disambiguation symbols (#-1, #0, #1, #2, #3 and so on).

The section of script that makes the H transducer (we call it Ha because it lacks self-loops at this point), is:

```
make-h-transducer --disambig-syms-out=$dir/disambig_tstate.list \
   --transition-scale=1.0  $dir/ilabels.remapped \
   $tree $model  > $dir/Ha.fst
```

There is an optional argument to set the transition scale; in our current training scripts, this scale is 1.0. This scale only affects the parts of the transitions that do not relate to self-loop probabilities, and in the normal topology (Bakis model) it has no effect at all; see **Scaling of transition and acoustic probabilities** for more explanation. In addition to the FST, the program also writes a list of disambiguation symbols which must be removed later.

Making HCLG

The first step in making the final graph HCLG is to make the HCLG that lacks self-loops. The command in our current script is as follows:

```
  fsttablecompose $dir/Ha.fst $dir/CLG2.fst | \
   fstdeterminizestar --use-log=true | \
   fstrmsymbols $dir/disambig_tstate.list | \
   fstrmepslocal  | fstminimizeencoded > $dir/HCLGa.fst
```

Here, CLG2.fst is the version of CLG with a reduced symbol set ("logical" triphones, in HTK terminology). We remove the disambiguation symbols and any easy-to-remove epsilons (see Removing epsilons), before minimizing; our minimization algorithm is one that avoids pushing symbols and weights (hence preserving stochasticity), and accepts nondeterministic input (see Minimization).

Adding self-loops to HCLG

Adding self-loops to HCLG is done by the following command:

```
  add-self-loops --self-loop-scale=0.1 \
    --reorder=true $model < $dir/HCLGa.fst > $dir/HCLG.fst
```

See **Scaling of transition and acoustic probabilities** for an explanation of how the self-loop-scale of 0.1 is applied (note that it also affects the non-self-loop probabilities). For an explanation of the "reorder" option, see **Reordering transitions**; **the "reorder" option increases decoding speed** but is not compatible with the kaldi decoder. The add-self-loops program does not just add self-loops; it may also have to duplicate states and add epsilon transitions in order to ensure that the self-loops can be added in a consistent way. This issue is mentioned in slightly more detail in **Reordering transitions**. This is the only stage of graph creation that does not preserve stochasticity; it does not preserve it because the self-loop-scale is not 1. So the program fstisstochastic should give the same output for all of G.fst, LG.fst, CLG.fst and HCLGa.fst, but not for HCLG.fst. We do not determinize again after the add-self-loops stage; this would fail because we have already removed the disambiguation symbols. Anyway, this would be slow and we believe that there is nothing further to be gained from determinizing and minimizing at this point.