Mini-LibriSpeech with GMM-HMM Based on Wan's Blog

Jianhua Liu

Fall 2023

Contents

Kaldi Tutorial for a GMM/HMM System with Mini LibriSpeech Based on Qianhui Wan's Blog

This tutorial is based on Wan's blog published at https://medium.com/[@qianhwan/understanding-kaldi-recipes-with-mini-librispeech-example-part-1-hmm-models-472a7f4a0488].

We did minor changes to make sure everything works.

Mini-LibriSpeech is a small subset of LibriSpeech which consists of audio book reading speech. We will go through each step in `kaldi/egs/mini_librispeech/s5/run.sh`.

As we have made some changes to suit our needs, we will have the recipe and other files saved in the `egs/salai_mini_librispeech` folder. We will call it the new proj root folder from now on. The original one is called the old proj root folder.

Before we start, we have the following folders and files in the new proj root folder, copied form the old one:

```
conf
    decode.config
    mfcc.conf
    mfcc_hires.conf
    online_cmvn.conf
local
    chain
    chain2
    grammar
    kws
    lookahead
    nnet3
    data_prep.sh -> ../../../librispeech/s5/local/data_prep.sh
    download_and_untar.sh
    download_lm.sh
    format_lms.sh -> ../../../librispeech/s5/local/format_lms.sh
    prepare_dict.sh -> ../../../librispeech/s5/local/prepare_dict.sh
    score.sh
    subset_dataset.sh
steps -> ../../wsj/s5/steps/
utils -> ../../wsj/s5/utils/
cmd.sh
path.sh
run.sh
```

Parameters and environment setup

```
#!/usr/bin/env bash

# Change this location to somewhere where we want to put the data.
data=/data/mini_librispeech/

# Specify the URL for downloading the audio data.
data_url=www.openslr.org/resources/31
# Specify the URL for downloading vocabulary, lexicon, and pre-trained
# language model trained on LibriSpeech.
```

```
lm_url=www.openslr.org/resources/11

# Set up the runner. We need to use run.pl for local applications.
. ./cmd.sh
# Run path.sh which adds all kaldi executable dependencies to the
# environment path. This is required every time we start a new terminal.
# It can avoided by adding all paths in .bashrc, which is discouraged.
. ./path.sh

# Set which stage this script is on.
# We can set it to the stage number that has already been executed
# to avoid running the executed commands again.
stage=-1

# Enable argument parsing to kaldi scripts (e.g. ./run.sh --stage 2
# sets variable stage to 2). Every variable defined before the following
# line will be able to be passed using the "--key value" pair.
. utils/parse_options.sh

# Make the scripts exit safely when encountering an error.
set -euo pipefail

# Create the data folder if it doesn't exist already.
mkdir -p $data
```

Note that for the `cmd.sh` file, we have changed it to:

```
export train_cmd="run.pl"
export decode_cmd="run.pl"
export mkgraph_cmd="run.pl"
# export cuda_cmd="run.pl --gpu 1"
```

Running different stages

Each Kaldi recipe consists of multiple stages, which can be spotted with the following syntax:

```
if [ $stage -le x ]; then
  # do something
fi
```

This simply means that if `stage` is less than or equal to `x`, a given number, we will run the commands in this block . Note that we can change `-le` to `-eq`, which means equal, so that we can run the recipe step-by-step.

`stage` is set to 0 by default, which means the recipe will run all blocks. If we encounter an error, we can check which stages have successfully passed and re-run the recipe by `./run.sh --stage x`, starting again from the failed stage.

Note that this option is parsed by `parse_options.sh`. If we want to pass an option using the `--key value` format, the variable should be defined before the calling of `parse_options.sh`.

Stage -1: Downloading and unzipping the dataset

We put this in Stage -1.

Here, we download **dev-clean-2** (dev set) and **train-clean-5** (train set) from the URL specified before to the specified *data* folder and unzip them:

```
if [ $stage -le -1 ]; then
  for part in dev-clean-2 train-clean-5; do
    local/download_and_untar.sh $data $data_url $part
  done
fi
```

We can check the files in the `$data` folder after running the above stage.

Stage 0: Downloading the language model

Now, we download the pre-trained language model to `$data` and make a soft link to `data/local/lm` by running the following code:

```
local/download_lm.sh $lm_url $data data/local/lm
```

The files that are downloaded include:

- `3-gram.arpa.gz`, trigram ARPA LM.
- `3-gram.pruned.1e-7.arpa.gz`, pruned (with threshold 1e-7) trigram ARPA LM.
- `3-gram.pruned.3e-7.arpa.gz`, pruned (with threshold 3e-7) trigram ARPA LM.
- `librispeech-vocab.txt`, 200K word vocabulary for the LM.
- `librispeech-lexicon.txt`, pronunciations, some of which are automatically generated using the G2P (grapheme to phoneme) tool, for all words in the vocabulary.

Stage 1: Preparing data, dictionary, and language files

Step 1a. Creating data files

All the created data files should be in subfolders in the `s5/data` folder. We use the following script to create these files.

```
for part in dev-clean-2 train-clean-5; do
  # use underscore-separated names in data directories.
  local/data_prep.sh $data/LibriSpeech/$part \
      data/$(echo $part | sed s/-/_/g)
done
```

We see that these data are saved in the `train_clean_5` and `dev_train_2` subfolders. There are used for training and testing, respectively. See here for more details on data preparation.

Normally each Kaldi recipe comes with a specific data preparation script. All these scripts create the same files from different dataset. To train a model with our own dataset, we need to write our own data preparation script that produces the right Kaldi-style data files.

If we check `data/train_clean_5` after finishing the above commands, we will see the following text files:

- `wav.scp`: maps wav files to their paths (can be audio processing commands with pipe).
- `utt2spk`: maps utterances to their speaker, when speaker information is unknown, we treat each utterance as a different speaker.
- `spk2utt`: maps speakers to the utterances spoken by them.
- `text`: maps recordings to their transcribed text.

- `spk2gender`: maps speakers to their genders.
- `utt2dur`: maps utterances to their durations.
- `utt2num_frames`: maps utterances to their number of frames.

Each data set (train, dev, test) should have their own set of files. Among these files, `wav.scp`, `utt2spk`, `spk2utt`, and `text` are essential for building any Kaldi models.

Step 1b. Preparing dictionary files

The dictionary files are mainly about the lexicon file. We usually download the CMU dictionary for the in-vocabulary words and use the G2P tool for the OOV words.

For this specific recipe, we skip this normal procedure and use the prepared dictionary files linked in to the `data/local/lm` folder to save time.

The dictionary files can be put in the `dict` subfolder of `s5/data`. Here, the authors of the original recipe choose to put them in the `s5/local/dict_nosp` folder, which is perfectly fine. We use the following script to create these files.

```
local/prepare_dict.sh --stage 3 --nj 30 --cmd "$train_cmd" \
    data/local/lm data/local/lm data/local/dict_nosp
```

The usage of `prepare_dict.sh` is given below:

```
prepare_dict.sh [options] <lm-dir> <g2p-model-dir> <dst-dir>
```

- The `--stage 3` option says that we just need to run this script from stage 3. The first two stages are the checking and preparation of the CMU dictionary and the G2P tool.
- The `--nj 30` option says that we will run 30 different jobs. We can change this number to match the number of CPU cores of the computer.
- The directories are self-explanatory, and we know that the directory results are saved in the `data/local/data_nosp` folder.

Here, `nosp` refers to the dictionary without silence probabilities and pronunciation.

This step generates silence phones, non-silence phones, and optional silence phones in the `data/local/dict_nosp` directory. The generated files are as follows:

- `extra_questions.txt`, list of extra questions which will be included in addition to the automatically generated questions for decision trees.
- `lexicon.txt`, sorted lexicon with some additional silence phones.
- `lexiconp.txt`, lexicon with pronunciation probabilities.
- `lexicon_raw_nosil.txt`, the same lexicon.
- `nonsilence_phones.txt`, list of non-silence phones.
- `optional_silence.txt`, list of optional silence phones.
- `silence_phones.txt`, list of silence phones. More detailed explanation can be found here.

Step 1c. Creating language files

The language files are those produced according to the vocabulary of the dataset and the dictionary files prepared in the previous step. The final goal is to create the FST files for the lexicon used in the training dataset. The input will be the phoneme, and the output will be words.

The created language files can be in the `lang` subfolder of `s5/data`. Here, the authors of the original recipe choose to put them in `s5/data/lang_nosp`. We use the following script to create these files.

```
utils/prepare_lang.sh data/local/dict_nosp \
  "<UNK>" data/local/lang_tmp_nosp data/lang_nosp
```

The usage of `prepare_lang.sh` is given below:

```
prepare_lang.sh <dict-src-dir> <oov-dict-entry> <tmp-dir> <lang-dir>
```

The `data/lang_nosp` directory has the following files:

- `L.fst`, FST form of lexicon.
- `L_disambig.fst`, L.fst but including the disambiguation symbols.
- `oov.int`, mapped integer of out-of-vocabulary words.
- `oov.txt`, out-of-vocabulary words.
- `phones.txt`, maps phones with integers.
- `topo`, the topology of the HMMs we use.
- `words.txt`, maps words to integers.
- `phones/`, specifies various things about the phone set.

Step 1d. Formatting language models

The language model (LM) is different than the language files we created in the previous step. They are the n-gram models. We use the ARPA format LM so that we can use the existing tools to train our own LM if existing ones do not suit our need.

The formatted LM is usually saved in the `data/local/lm` folder. We use the following script to create these LMs.

```
local/format_lms.sh --src-dir data/lang_nosp data/local/lm
```

The usage of `format_lms.sh` is given below:

```
format_lms.sh [option] <lm-dir>
```

The only option is the source directory, which has the default as `data/lang`.

This script uses `data/lang_nosp/word.txt` to format two pruned ARPA LMs saved in `data/local/lm` to `G.fst` files (G for grammar) saved in `data/lang_nosp_test_tgmed` and `data/lang_nosp_test_tgsmall`, respectively.

Step 1e. Building constant ARPA LM model

```
utils/build_const_arpa_lm.sh data/local/lm/lm_tglarge.arpa.gz \
  data/lang_nosp data/lang_nosp_test_tglarge
```

It creates ConstArpaLm format language model, `G.carpa` in the `data/lang_nosp_test_tglarge` directory from the full 3-gram APAR LM.

This model is used later for performance comparison.

Stage 2: Extracting MFCC features

`mfccdir=mfcc` is used to specify where to store the extracted MFCCs (Mel-frequency cepstral coefficients) under `s5`. With this setup, the MFCC features will be saved in the `s5/mfcc/` folder.

A block of code is used for JHU computers and is not appropriate for the other users.

Step 2a. Extracting MFCC and computing CMVN

In addition to MFCC, CMVN (cepstral mean and variance normalization) data will be computed. The following block of code is used for this purpose:

```
for part in dev_clean_2 train_clean_5; do
    steps/make_mfcc.sh --cmd "$train_cmd" --nj 10 data/$part \
        exp/make_mfcc/$part $mfccdir
    steps/compute_cmvn_stats.sh data/$part
        exp/make_mfcc/$part $mfccdir
done
```

This script extracts MFCCs and computes CMVN stats according to the data links, the `wav.scp` files saved `data/dev_clean_2` and `data/train_clean_5`. The results are saved to the `s5/mfcc` folder using 10 parallel jobs. Logs can be found in `exp/make_mfcc`, which can be checked if something goes wrong.

Step 2b. Extracting features for shortest utterances

The following script creates a data subset of the shortest 500 utterances.

```
# Get the shortest 500 utterances first because those are more likely
# to have accurate alignments.
utils/subset_data_dir.sh --shortest data/train_clean_5 500 \
    data/train_500short
```

The most common usage of `subset_data_dir.sh` is shown below.

```
subset_data_dir.sh [--speakers|--shortest|--first|--last|--per-spk] <srcdir> <num-utt>
 ↪  <destdir>"
```

Note that we do not create or copy any MFCC here; looking into `data/train_500short`, we can find `feat.scp` and `cmvn.scp` files that map the utterances to where their MFCCs are stored.

Stage 3: Training the monophone model

Step 3a. Training with the 500 shortest utterances

The following trains a monophone system using the shortest 500 utterances and the LM created before.

```
steps/train_mono.sh --boost-silence 1.25 --nj 5 --cmd "$train_cmd" \
    data/train_500short data/lang_nosp exp/mono
```

The common usage of `train_mono.sh` is given below:

```
steps/train_mono.sh [options] <data-dir> <lang-dir> <exp-dir>
```

Here we have:

- `--boost-silence 1.25` sets the factor by which to boost silence likelihoods in alignment to 1.25.
- `-nj 5` sets the number of parallel jobs to 5.
- The trained model and logs are saved in `exp/mono`.

Step 3b. Computing the alignment

The following script computes the training alignments using the monophone model.

```
steps/align_si.sh --boost-silence 1.25 --nj 5 --cmd "$train_cmd" \
    data/train_clean_5 data/lang_nosp exp/mono exp/mono_ali_train_clean_5
```

The common usage of `aligh_si.sh` is given below:

```
steps/align_si.sh <data-dir> <lang-dir> <src-dir> <align-dir>
```

Here the `src-dir` is the directory used to save the trained model and `align-dir` the output of the alignment.

Stage 4: Training the delta + delta-delta triphone model

The following script trains a triphone model with MFCC + delta + delta-delta features using the training alignments generated in Stage 3. The dimension of the original MFCC is 13. With the addition of delta and delta-delta, the total dimension of the feature is 39.

```
steps/train_deltas.sh --boost-silence 1.25 --cmd "$train_cmd" \
    2000 10000 data/train_clean_5 data/lang_nosp \
    exp/mono_ali_train_clean_5 exp/tri1
```

The common usage of `train_deltas.sh` is shown below.

```
steps/train_deltas.sh <num-leaves> <tot-gauss> <data-dir> <lang-dir> <alignment-dir>
 ↪  <exp-dir>
```

Here, we will build the decision trees; `num-leaves` is the maximum allowed number of leave nodes, and `tot-gauss` is the maximum allowed number of pdfs of Gaussians.

Note that the model data is saved in `exp/tri1`, meaning triphone round 1.

With the new model, we can move ahead to compute the new alignment using the script below:

```
steps/align_si.sh --nj 5 --cmd "$train_cmd" \
    data/train_clean_5 data/lang_nosp exp/tri1 exp/tri1_ali_train_clean_5
```

Stage 5: Training the LDA + MLLT triphone model

Using the training alignments Obtained in Stage 4, we can move forward to train a triphone model with LDA and MLLT feature transforms. Here LDA stands for Linear Discriminant Analyses, and MLLT Maximum Likelihood Linear Transform.

We use the following scripts to do so.

```
steps/train_lda_mllt.sh --cmd "$train_cmd" \
    --splice-opts "--left-context=3 --right-context=3" 2500 15000 \
    data/train_clean_5 data/lang_nosp exp/tri1_ali_train_clean_5 exp/tri2b
```

The options and parameters are similar to those we have seen before. Note that we have increased the numbers for the model as the results are getting better, and expressing weaker (supported by less data) leave nodes and pdfs is more appropriate.

Note that this is triphone model training round 2, and the model data is thus saved in `exp/tri2b`.

The corresponding alignment is done below:

```
steps/align_si.sh  --nj 5 --cmd "$train_cmd" --use-graphs true \
    data/train_clean_5 data/lang_nosp exp/tri2b exp/tri2b_ali_train_clean_5
```

Stage 6: Training the LDA + MLLT + SAT triphone model

Now, we the above alignment, we can start to train a triphone model with SAT (Speaker Adaptation Training).

```
steps/train_sat.sh --cmd "$train_cmd" 2500 15000 \
    data/train_clean_5 data/lang_nosp exp/tri2b_ali_train_clean_5 exp/tri3b
```

Stage 7: Re-creating language model and computing the alignments from SAT model

Step 7a. Computing the number of pronunciations

This is done using the script below.

```
steps/get_prons.sh --cmd "$train_cmd" \
    data/train_clean_5 data/lang_nosp exp/tri3b
```

There are several things happening here in this command:

- Linear lattices (single path) are generated for each utterance in `train_clean_5` using the latest alignment and LM.
- A bunch of `pron.x.gz`, x = 1, 2, …, 5, are created in the `exp/tri3b/` folder with the format of `<utterance-id> <begin-frame> <num-frames> <word> <phone1> … <phoneN>`
- Word pronunciation count, which contains the counts of pronunciations (generated by aligning training data, not from the original text), is saved in `pron_counts_nowb.txt`. This is used in the next step.

Step 7b. Creating a dictionary with pronunciation probabilities

This is done by taking the pronunciation counts.

```
utils/dict_dir_add_pronprobs.sh --max-normalize true \
    data/local/dict_nosp \
    exp/tri3b/pron_counts_nowb.txt exp/tri3b/sil_counts_nowb.txt \
    exp/tri3b/pron_bigram_counts_nowb.txt data/local/dict
```

The usage of `dict_dir_add_pronprobs.sh` is given below:

```
dict_dir_add_pronprobs.sh [options] <input-dict-dir> <input-pron-counts> \
    [input-sil-counts] [input-bigram-counts] <output-dict-dir>
```

The modified dictionary directory with pronunciation probabilities is `data/local/dict`.

Step 7c. Building a new ConstArpa LM

With the new dictionary, we can move on to build a new ConstArpa LM, as shown below.

```
utils/prepare_lang.sh data/local/dict \
    "<UNK>" data/local/lang_tmp data/lang

local/format_lms.sh --src-dir data/lang data/local/lm
```

```
utils/build_const_arpa_lm.sh \
    data/local/lm/lm_tglarge.arpa.gz data/lang data/lang_test_tglarge
```

All the commands above have been explained before, and we will not discuss it further.

Step 7d. Aligning using the SAT model

The last step in Stage 7 is to compute the training alignments using the SAT model and new `L.fst`, as shown below. (==Generating a new L.fst or using the new L.fst?==)

```
steps/align_fmllr.sh --nj 5 --cmd "$train_cmd" \
    data/train_clean_5 data/lang exp/tri3b exp/tri3b_ali_train_clean_5
```

The common usage of `align_fmllr.sh` is shown below.

```
steps/align_fmllr.sh <data-dir> <lang-dir> <src-dir> <align-dir>
```

Stage 8: Creating the final graph and decoding using the graph

Step 8a. Creating the final graph

Finally, it is time to create the final `HCLG.fst` graph. We can use the small trigram LM.

```
utils/mkgraph.sh data/lang_test_tgsmall \
    exp/tri3b exp/tri3b/graph_tgsmall
```

The common usage of `mkgraph.sh` is shown below.

```
utils/mkgraph.sh [options] <lang-dir> <model-dir> <graphdir>
```

Step 8a. Decoding

The following code for decoding will be run in a `for` loop so that we can decode multiple datasets.

Here, `test` is the `dev_clean_2` dataset.

Decoding with the small trigram LM The following code decodes the `$test` dataset using the SAT model and the small trigram LM.

```
steps/decode_fmllr.sh --nj 10 --cmd "$decode_cmd" \
  exp/tri3b/graph_tgsmall data/$test \
  exp/tri3b/decode_tgsmall_$test
```

The common usage of `decode_fmllr.sh` is shown below:

```
steps/decode_fmllr.sh [options] <graph-dir> <data-dir> <decode-dir>
```

The lattices and WERs can be found at `exp/tri3b/decode_tgsmall_dev_clean_2`.

Rescoring the decoded lattice using the medium trigram LM To see the decoding performance difference using different trigram LM, we re-score decoded lattice in `exp/tri3b/decode_tgsmall_dev_clean_2` with the medium trigram LM, as shown below.

```
steps/lmrescore.sh --cmd "$decode_cmd" \
  data/lang_test_{tgsmall,tgmed} \
  data/$test exp/tri3b/decode_{tgsmall,tgmed}_$test
```

The common usage of `lmrescore.sh` is shown below:

```
lmrescore.sh [options] <old-lang-dir> <new-lang-dir> <data-dir> <input-decode-dir>
 ↪  <output-decode-dir>
```

Note that this scripts need to use both the old and new LM.

The lattices and WERs after re-scoring can be found at `exp/tri3b/decode_tgmed_dev_clean_2`.

Rescoring the decoded lattice using the ConstArpa LM Now, let's re-score the decoded lattice in `exp/tri3b/decode_tgsmall_dev_clean_2` with the large ConstArpa LM, as shown below.

```
steps/lmrescore_const_arpa.sh --cmd "$decode_cmd" \
  data/lang_test_{tgsmall,tglarge} \
  data/$test exp/tri3b/decode_{tgsmall,tglarge}_$test
```

The parameters of `lmrescore_const_arpa.sh` are the same as that for `lmrescore.sh`.

The lattices and WERs after re-scoring can be found at `exp/tri3b/decode_tglarge_dev_clean_2`.

A note on decoding

To see the WER performance, we can use the following decoding

```
(
  utils/mkgraph.sh data/lang_nosp_test_tgsmall \
      exp/mono exp/mono/graph_nosp_tgsmall
  for test in dev_clean_2; do
      steps/decode.sh --nj 10 --cmd "$decode_cmd" \
          exp/mono/graph_nosp_tgsmall \
          data/$test exp/mono/decode_nosp_tgsmall_$test
  done
)&
```

This creates the final graph, `HCLG.fst model`, and decodes `data/dev_clean_2` using the graph. We can find WERs in `exp/mono/decode_nosp_tgsmall_dev_clean_2`.

In the `mini_librispeech` recipe each training stage (monophone, triphone, DNN etc.) can be attached with a decoding step. It is a good practice to see improvements when the model gets more complicated.

As we can see in `exp/mono/decode_nosp_tgsmall_dev_clean_2`, there are more than one WER file. This is because `steps/decode.sh` calls `local/score.sh` where we play with some scoring parameters to get the best WER. These parameters corresponds to the hyperparameters of machine learning.

In the example of `wer_10_0.5`, 10 is the LM-weight for lattice rescoring, 0.5 is the word insertion penalty factor.

We should be able to see the WER improvements from `exp/mono/decode_nosp_tgsmall_dev_clean_2` to `exp/tri3b/decode_tglarge_dev_clean_2`.