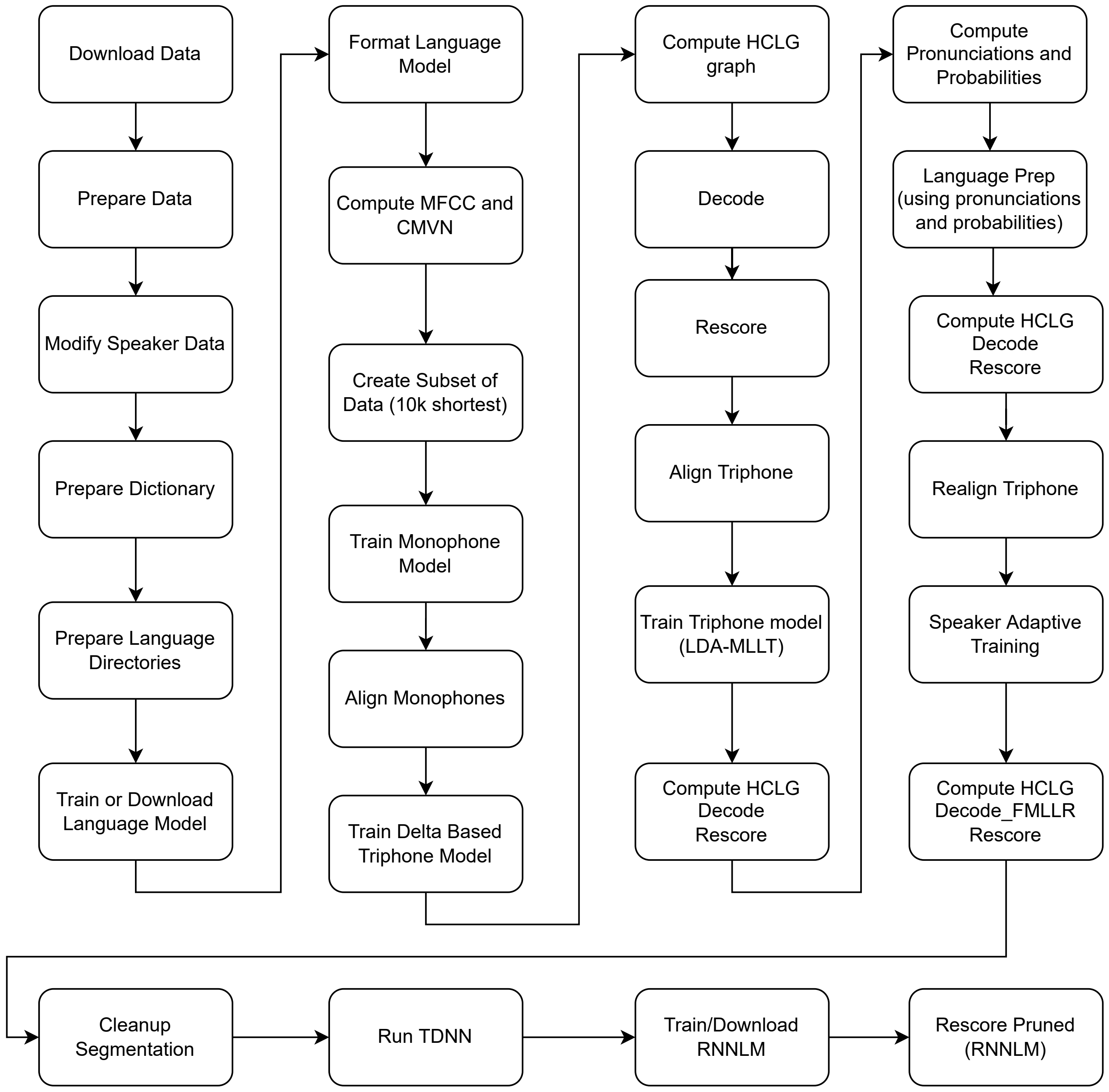
Kaldi Tedlium s5_r3 Pipeline Writeup

Sourya Kakarla (sk5057)

1. Introduction


Download Data
Prepare Data
Modify Speaker Data
Prepare Dictionary
Prepare Language Directories
Train or Download Language Model
Format Language Model
Compute MFCC and CMVN
Create Subset of Data (10k shortest)
Train Monophone Model
Align Monophones
Train Delta Based Triphone Model
Compute HCLG graph
Decode
Rescore
Align Triphone
Train Triphone model (LDA-MLLT)
Compute HCLG Decode Rescore
Compute Pronunciations and Probabilities
Language Prep (using pronunciations and probabilities)
Compute HCLG Decode Rescore
Realign Triphone
Speaker Adaptive Training
Compute HCLG Decode_FMLLR Rescore
Cleanup Segmentation
Run TDNN
Train/Download RNNLM
Rescore Pruned (RNNLM)

2. Stages

2.1. Stage 0 - Download TEDLIUM release-3 data

In this stage, the `local/download_data.sh` script is run to download the Tedlium dataset from OpenSLR using `wget`. The downloaded `TEDLIUM_release-3.tgz` is saved in the `db/` folder and extracted there. After downloading the data, `download_data.sh` finally validates the number of `.sph` files in the `TEDLIUM_release-3/data/` directory. The expected count is 2351 if everything goes well.

2.2. Stage 1 - Prepare Data

In this stage, processing on the data downloaded in Stage 0 is done by running the script `prepare_data.sh` and files of different kinds are generated for the dev, test, and train partitions which will be used in the later stages as part of language model training and so on. They are as follows:

- `text`: Contains mappings between utterances (transciptions) and utterance ids which will be used by Kaldi
- `segments`: Contains mappings between `utterance-id` and segment of a particular recording. It contains the start and end times for each utterance in an audio file. It is of the format `utt_id file_id start_time end_time` [1].
- `utt2spk` : One-to-one mapping between utterance ids and the corresponding speaker identifiers.
- `spk2utt`: Mapping between the speaker identifiers and all the utterance identifiers associated with the speaker recording.
- `wav.scp`: A list of Utterances to Filenames with each line of format `<recording_id><extended_filename>`
- `reco2file_and_channel`: Used when scoring (measuring error rates) with NIST's *sclite* tool.
- `glm`: An empty glm file is generated.

As part of the processing, silence is padded to the segments, especially at the beginning. Further, the speakers are split into 3-minutes chunks by running the `utils/data/modify_speaker_info.sh` The generated files are validated to check if they meet Kaldi's format specifications.

2.3. Stage 2 - Prepare Dictionary

In this stage, `prepare_dict.sh` script is run to create the files at `data/local/dict_nosp` required downstream for the language model. The main steps in this stage are as follows:

- Processing is done on the `db/TEDLIUM_release-3/TEDLIUM.152k.dic` dictionary by ignoring instances of `<unk>`, entries with numbers, and sorting alphabetically to create the `data/local/dict_nosp/lexicon_words.txt`.

- Files are created for non-silent phones (`data/local/dict_nosp/nonsilence_phones.txt`) and silent phones respectively (`data/local/dict_nosp/silence_phones.txt`). The silent phones are `SIL` (silence) and `NSN` (non-spoken noise).
- `optional_silence.txt` is generated which simply contains an `SIL` model.
- `data/local/dict_nosp/lexicon.txt` is created by taking unique entries from `lexicon_words.txt` and adding the silence (`SIL`) and noise (`NSN`) phones to them. The lines in the `lexicon.txt` are in the format of `word phone1 phone2...phoneN`

2.4. Stage 3 - Pepare Language Files

In this stage, the `utils/prepare_lang.sh` script is run with following parameters as input

- `data/local/dict_nosp` (dictionary source directory)
- `<unk>` (lexical entry for uknown)
- `data/local/lang_nosp` (temporary directory for processing)
- `data/lang_nosp` (target language directory)

It prepares the `data/lang_nosp/` directory in the Kaldi standard format which contains the following files and folders. `nosp` refers to the model before computation of silence probabilities and pronunciation.

- `G.fst`: Finite State Transducer form of the language model.
- `L.fst`: Finite State Transducer form of the lexicon .
- `L_disambig.fst`: The lexicon, as above but including the disambiguation symbols #1, #2, and so on, as well as the self-loop with #0 on it to ”pass through” the disambiguation symbol from the grammar.
- `oov.txt`: Contains a single line of the format `<unk>` which is the word to which all out-of-vocabulary words map to during training.
- `oov.int`: Integer form of the above (`38`).
- `phones/`: Directory containing various files regarding the phone set. Most of the files exist in 3 separate versions - `txt`, `int`, and `csl`.
- `phones.txt`: Symbol-table file, in the OpenFst format, where each line is in the text form and integer form. Example: `T_E 132`. Used by Kaldi to map back and forth between the integer and text forms of these symbols.
- `words.txt`: Similar to `phones.txt` but for words. Example: `arsenal 7054`.
- `topo`: Specifies the topology of the HMMS used.

All the generated files are validated using `utils/validate_lang.pl` script.

2.5. Stage 4 - Train or Download Language Model

Depending upon the `train_lm` variable in `run.sh`, this stage trains a language model by running the `ted_train_lm.sh` script which trains a language model using the TEDLIUM 6 data sources if `train_lm` is set to `true` or runs `ted_download_lm.sh` to download ARPA-style 4-gram language models (arpa.gz) from `kaldi-asr.org` if otherwise. By default, `train_lm` is set to `fasle`.

In the `ted_train_lm.sh` script, the training is does on Cantab-Tedlium text data and tedlium acoustic training data which from the earlier stages. It is based on the example scripts that come with PocoLM [3] and creates models in the ARPA format [2].

2.6. Stage 5 - Format Language Model

In this stage, the `format_lms.sh` script is run to generate certain language models in suitable formats as follows:

- The script checks for the existence of `data/lang_nosp/G.fst` and if not available, creates it by converting the ARPA-format language model `4gram_small.arpa.gz` into an OpenFST format using `arpa2fst` [4] which is a Kaldi program that turns the ARPA-format language model into a Weight Finite State Transducer. Further, it verifies if `G.fst` is stochastic enough using `fstistochastic`.

- It checks for the `data/lang_nosp/G.carpa` and if not available, creates it by converting the ARPA-format language model `4gram_big.arpa.gz` into a ConstArpaLm [6] format model by running the script `utils/build_const_arpa_lm.sh`. This model is used downstream for rescoring.

It also performs a series of checks on the encoding of all the files in `data/lang_nosp/` directory using the `utils/validate_lang.pl` script.

2.7. Stage 6 - Feature Extraction

In this stage, MFCC features are extracted for all the 3 data partitions (test, dev, and train) using the `steps/make_mfcc.sh` script on the data placed in the `data/<$set>` ($\$set = test, dev, train$) directories. It uses the `compute-mfcc-feats` command-line tool to produce the required MFCC features. A broad overview [14] of the MFCC feature extractions is as follows:

- Work out the number of frames in the file (typically 25 ms frames shifted by 10ms each time).

- For each frame:
  - Extract the data, do optional dithering, preemphasis and dc offset removal, and multiply it by a windowing function.
  - Calculate the energy (negative log), do FFT and compute the power spectrum.
  - Compute log of the energy in each mel bin and take cosine transform keeping the required number of coefficients.

Further, cepstral mean and variance statistics (CMVN) [10] per speaker are computed for all the 3 data partitions using the `steps/compute_cmvn_stats.sh` script which helps in compensation of noise and reduces mismatch between different data partitions.

2.8. Stage 7 - Select 10k Shortest Utterances

In this stage, 10k shortest utterances are selected using the `utils/subset_data_dir.sh` script from the `data/train` directory and put in `data/train_10kshort`. Further, excessive repetitions of utterances are eliminated if they occur more than a particular number of times (10) and output is placed in `data/train_10kshort_nodup` directory by running the `remove_dup_utts.sh` with the maximum allowed number of repetitions as 10. The number of utterances drop to 9135 from 10000. This subset creation is useful for monophone training downstream.

2.9. Stage 8 - Monohpone Model Training

In this stage, the monophone models are trained using the `train_mono.sh` script on the `data/train_10kshort_nodup.sh` (generated in the previous stage) and `data/lang_nosp` and are saved in the `exp/mono/` directory.

This is the primary step of the actual training process in the pipeline. The smaller subset of 10k utterances is used to achieve efficiency. Even with little data, satisfactory monophone models can be trained and are used to bootstrap training for later models in the upcoming stages.

The monophone model is a purely acoustic model and does not include any contextual information about the adjacent phones. It is the building block for the models which have context embedded into them (such as the triphone in the next stage).

2.10. Stage 9 - Align Monophones and Train delta-based triphones

By running the script `align_si.sh`, data in `data/train` is aligned using model (through deltas) from `exp/mono/` and the alignments are put in `exp/mono_ali`. Further, delta-based triphones are trained using the language model `data/lang_nosp` and monophone alignments `mono_ali` and are placed in the `exp/tri1/` directory.

Even though the parameters of the acoustic model are estimated in the monophone model training in the previous stage, the aim is to improve it in this stage using contextual information, thus the triphones. Viterbi training is used to achieve this. By aligning the audio to the transript from the training data with the most current acoustic model, training algorithms use this to improve the parameters of the model.

As only a small subset of all triphone possibilites are actually relevant for the model, phonetic decision tree groups are used to classify the phones into a minimal set of acoustically distinct units, thus reducing the number of parameters.

Consequently, each training step is followed by an alignment step so that the audio and the text can be realigned.

2.11. Stage 10 - Decoding and Rescoring

A fully expanded decoding graph (HCLG.fst model) is computed at `exp/tri2/graph_nosp` by running `utils/mkgraph.sh` that represents the language-model, lexicon, context-dependency, and HMM structure in the model. HCLG = H o C o L o G [7].

- G is an acceptor (i.e. its input and output symbols are the same) that encodes the grammar or language model.
- L is the lexicon; its output symbols are words and its input symbols are phones.
- C represents the context-dependency: its output symbols are phones and its input symbols represent context-dependent phones, i.e. windows of N phones.
- H contains the HMM definitions; its output symbols represent context-dependent phones and its input symbols are transition-ids, which encode the pdf-id and other information.

After the HCLG graph is calculated, decoding the utterances using the graph is done by running `steps/decode.sh` on the graph at `exp/tri2/graph_nosp` and WERs are calculated using `lmrescore_const_arpa.sh` which rescores the lattices with ConstArpaLm format language model.

2.12. Stage 11 - Align delta-based triphones and Train (LDA-MLLT)

Similar to the first part of Stage 9, triphones are aligned by running the `align_si.sh` , the difference being `exp/tri1/` being the input model and output alignments placed in `exp/tri1_ali`.

A triphone model with LDA (Linear Discriminant Analysis) [12] and MLLT (Maximum Likelihood Linear Transform) [13] feature transforms is trained using the training alignments obtained from above (`exp/tri1_ali`) by running the `steps/train_lda_mllt.sh` script and the new model is placed in `exp/tri2` directory.

2.13. Stage 12 - Decoding and Rescoring Updated Triphone Model

This stage is identical to Stage 10 except for that the decoding and rescoring is done using the latest model (`exp/tri2/`) from Stage 11.

2.14. Stage 13 - Compute Pronunciations and Probabilities

Linear lattices are computed for each utterance in the training data `data/train/` by using the latest alignment `exp/tri2` and language model `data/lang_nosp/` by running the script `steps/get_prons.sh`.

A lattice is a representation of the alternative word-sequences that are ”sufficiently likely” for a particular utterance [11]. Files of the form `prons.*.gz` are generated and the various counts of pronunciations are computed which are used in the next step in this stage.

Next, by running the `utils/dict_dir_add_pronprobs.sh` on the counts obtained from above, a modified dictionary directory with pronunciation probabilities is computed in `data/local/dict/`, for example data/local/dict/lexiconp.txt and data/local/dict/lexiconp_silprob.txt which includes the silence probabilities as well.

2.15. Stage 14 - Language Prep, Decoding, and Rescoring

The first step in this stage is similar to Stage 3 except that data/local/dict/ is used to generate the files at data/lang/ (as opposed to data/lang_nosp/) as the dictionary source directory as we now have the probabilities.

The decoding graph is now recomputed in exp/tri2/graph/ using the new language files at data/lang/ and decoding is done as before. Rescoring is done to evaluate the performance of the upgraded language model.

2.16. Stage 15 - Realign Trihpones and Speaker Adaptive Training (SAT)

The tirphone alignments are recomputed using the latest language files (with pronunciation probabilities) from the previous stage.

Next, Speaker Adaptive Training is done by running steps/train_sat.sh using the latest language model and alignments and a new triphone model is generated at exp/tri3/. SAT performs speaker and noise normalization by adapting to each speaker with a particular data transform resulting in more homogeneous data. This allows the model to be phoneme based than speaker and/or environment based.

In this stage, the decoding step has a significant alteration. The script steps/decode_fmllr.sh is run for the decoding after the graph creation as opposed to steps/decode.sh in the previous stages. This script uses feature-space MLLR (fMLLR)[8] transform for the first time in this recipe for speaker adaptive decoding.

Finally rescoring is done using the lastest model and outputs from the fMLLR decoding.

2.17. Stage 16 - Cleanup Segmentation

The segmentation done earlier is improved by running the script run_cleanup_segmentation.sh which re-segments the training data by selecting only ”good” audio that matches the transcripts.

In running the script, steps/cleanup/clean_and_segment_data.sh script is used to clean the training data based on the acoustic and language models trained so far.

Next, updated alignments are computed using the steps/align_fmllr.sh script. Further, Speaker Adaptive Training is redone with latest segmentations. Finally decoding (fMLLR) and rescoring is like in the previous stage.

2.18. Stage 17 - Run Time Delayed Neural Network

In this stage, the tuning/run_tdnn_1d.sh script is run to train a TDNN model to learn acoustic features on the latest data.

First, i-Vector extraction is done by running the run_ivector_common.sh script. An iVector is a vector of dimension several hundred (one or two hundred, in this particular context) which represents the speaker properties [9]. As part of the extraction, data augmentation is done, and speed-perturbed data is prepared from that [15]. Next, MFCC and CMVN of the speed-perturbed data is computed. Finally training alignments of the speed-perturbed data are obtained. Volume perturbation is then done on speed-perturbed train dataset. High-resoultion MFCCs and CMVNs are extracted from the volume and speed perturbed data.

The `steps/nnet3/chain/gen_topo.py` script is run to populate the `data/lang_chain/` directory with a chain topology [5]. This topology is used for Kaldi nnet3 DNN-HMM models.

Next alignments and lattices are obtained from low resoultion MFCCs by running `steps/align_fmllr_lats.sh`.

Next a new decision tree is built using lowe-resolution MFCCs, the new topology and alignments by running `steps/nnet3/chain/build_tree.sh`

A config file for the DNN sturcture is built using `steps/nnet3/xconfig_to_configs.py`

The configured DNN is trained using the high-resolution MFCCs, new decisions tree and i-vectors extracted in the previous steps by running `steps/nnet3/chain/train.py`.

Finally graph is recomputed (using `utils/mkgraph.sh`), decoding and rescoring are done similar to the previous stages.

2.19. Stage 18 - Download or Run RNNLM

Based on the variable `train_rnnlm`, a language model is downloaded (if `False`) using `local/ted_download_rnnlm.sh` or a Recurrent Neural Network Language Model is trained locally by running the `local/rnnlm/tuning/run_lstm_tdnn_a.sh` and `local/rnnlm/average_rnnlm.sh` scripts. A RNN language model is better performing than an n-gram model as it deals well with the sparseness.

After necessary preparation and configuration steps, it is by running `rnnlm/train_rnnlm.sh` script the model is first trained. Later, `average_rnnlm.sh` script takes the default `rnnlm_dir` of the recipe and averages the best model and the 10 previous and following ones (if they exist).

2.20. Stage 19 - Rescore Lattices (Pruned Algorithm)

This is the last stage of the pipeline. The script `rnnlm/lmrescore_pruned.sh` is run to rescore lattices with Kaldi RNNLM using a pruned algorithm [16]. This algorithm improves the n-gram approximation method. The pruned algorithm further limits the search space and uses heuristic search to pick better histories when expanding the lattice. It achieves better ASR accuracies while running much faster than the standard algorithm.

References

[1] E. Chodroff. Kaldi tutorial, 2015. URL https://eleanorchodroff.com/tutorial/kaldi/index.html.

[2] CMUsphinx. Arpa language models, 2022. URL https://cmusphinx.github.io/wiki/arpaformat/.

[3] danpovey. Pocolm github, 2022. URL https://github.com/danpovey/pocolm.

[4] Kaldi-ASR. arpa2fst.cc, 2018. URL https://github.com/kaldi-asr/kaldi/blob/master/src/lmbin/arpa2fst.cc.

[5] Kaldi-ASR. The chain model in kaldi, 2022. URL https://kaldi-asr.org/doc/chain.html#chain_model.

[6] Kaldi-ASR. Constarpalm class reference, 2022. URL https://kaldi-asr.org/doc/classkaldi_1_1ConstArpaLm.html.

[7] Kaldi-ASR. Decoding graph construction in kaldi, 2022. URL https://kaldi-asr.org/doc/graph.html.

[8] Kaldi-ASR. Global cmllr/fmllr transforms in kaldi, 2022. URL https://kaldi-asr.org/doc/transform.html#transform_cmllr_global.

[9] Kaldi-ASR. Neural net based online decoding with ivectors in kaldi, 2022. URL https://kaldi-asr.org/doc/online_decoding.html#online_decoding_nnet2.

[10] Kaldi-ASR. Cepstral mean and variance normalization in kaldi, 2022. URL https://kaldi-asr.org/doc/transform.html#transform_cmvn.

[11] Kaldi-ASR. Lattices in kaldi, 2022. URL https://kaldi-asr.org/doc/lattices.html.

[12] Kaldi-ASR. Linear discriminant analysis (lda) transforms in kaldi, 2022. URL https://kaldi-asr.org/doc/transform.html#transform_lda.

[13] Kaldi-ASR. Maximum likelihood linear transform (mllt) estimation in kaldi, 2022. URL https://kaldi-asr.org/doc/transform.html#transform_mllt.

[14] Kaldi-ASR. Computing mfcc features, 2022. URL https://kaldi-asr.org/doc/feat.html.

[15] qianhwan. Understanding kaldi recipes with mini-librispeech example (part 2— dnn models), 2019. URL https://medium.com/@qianhwan/understanding-kaldi-recipes-with-mini-librispeech-example-part-2-dnn-models-d1b85

[16] H. Xu, T. Chen, D. Gao, Y. Wang, K. Li, N. Goel, Y. Carmiel, D. Povey, and S. Khudanpur. A pruned rnnlm lattice-rescoring algorithm for automatic speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5929–5933, 2018. doi: 10.1109/ICASSP.2018.8461974.