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

Jianhua Liu

Fall 2023

Contents

Kaldi Tutorial for a DNN/HMM System with Mini LibriSpeech Based on Qianhui Wan's Blog **1**
Setting up parameters . . . 8
Extracting the i-vector . . . 9
Step 1: Checking required files . . . 9
Step 2: Augmenting data and computing features . . . 9
Step 3: Aligning with the low-resolution data . . . 10
Step 4: Aligning with the low-resolution data . . . 10
Checking required files in `run_tdnn.sh` . . . 10
Stage 10: Creating chain-type topology . . . 11
Stage 11: Generating lattices from low-resolution MFCCs . . . 11
Stage 12: Building a new tree . . . 12
Stage 13: Creating the config file for the DNN . . . 12
Stage 14: Training the DNN . . . 12
Stage 15: Compiling the final graph . . . 13

Kaldi Tutorial for a DNN/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-2-dnn-models-d1b851a56c49].

A DNN can be used to replace the GMM part of the model so that we can build a DNN/HMM based ASR system. Here, we discuss the script for such a system step by step. Specifically, we will go through `local/chain/run_tdnn.sh` in the `mini_librispeech/s5` folder, which is listed below, with **modifications** for formatting purposes.

`run_tdnn.sh`:

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

# Comparison of different ASR systems:
# local/chain/compare_wer.sh --online \
#   exp/chain/tdnn1j_sp exp/chain_online_cmn/tdnn1k_sp
# System                     tdnn1j_sp tdnn1k_sp
# WER dev_clean_2 (tgsmall)      10.97     10.64
```

```
#              [online:]         10.97     10.62
# WER dev_clean_2 (tglarge)       7.57      7.17
#              [online:]          7.65      7.16
# Final train prob              -0.0623   -0.0618
# Final valid prob              -0.0793   -0.0793
# Final train prob (xent)       -1.4448   -1.4376
# Final valid prob (xent)       -1.5605   -1.5461
# Num-params                    5210944   5210944

# 1k is like 1j; additionally, it introduces 'apply-cmvn-online' that does
# cmn normalization for both i-extractor and TDNN input.

# steps/info/chain_dir_info.pl exp/chain/tdnn1j_sp
# exp/chain/tdnn1j_sp: num-iters=34 nj=2..5 \
#   num-params=5.2M dim=40+100->2336 \
#   combine=-0.068->-0.064 (over 4) \
#   xent:train/valid[21,33,final]=(-1.65,-1.48,-1.44/-1.77,-1.58,-1.56) \
#   logprob:train/valid[21,33,final]=(-0.076,-0.068,-0.062/-0.091,-0.084,-0.079)

# steps/info/chain_dir_info.pl exp/chain_online_cmn/tdnn1k_sp
# exp/chain_online_cmn/tdnn1k_sp: num-iters=34 nj=2..5 \
#   num-params=5.2M dim=40+100->2336 combine=-0.067->-0.062 (over 5) \
#   xent:train/valid[21,33,final]=(-1.63,-1.47,-1.44/-1.73,-1.57,-1.55) \
#   logprob:train/valid[21,33,final]=(-0.074,-0.067,-0.062/-0.093,-0.085,-0.079)

# Set -e here so that we catch if any executable fails immediately
set -euo pipefail

# First the options that are passed through to run_ivector_common.sh
# (some of which are also used in this script directly).
stage=0
decode_nj=10
train_set=train_clean_5
test_sets=dev_clean_2
gmm=tri3b
nnet3_affix=_online_cmn

# Setting 'online_cmvn' to true replaces 'apply-cmvn' by
# 'apply-cmvn-online' both for i-vector extraction and TDNN input.
# The i-vector extractor uses the config 'conf/online_cmvn.conf' for
# both the UBM and the i-extractor. The TDNN input is configured via
# '--feat.cmvn-opts' that is set to the same config, so we use the
# same cmvn for i-extractor and the TDNN input.
online_cmvn=true

# The rest are configs specific to this script.  Most of the parameters
# are just hardcoded at this level, in the commands below.
affix=1k   # affix for the TDNN directory name
tree_affix=
train_stage=-10
get_egs_stage=-10
decode_iter=

# training options
```

```
# training chunk-options
chunk_width=140,100,160
common_egs_dir=
xent_regularize=0.1

# training options
srand=0
remove_egs=true
reporting_email=

#decode options
test_online_decoding=true  # if true, it will run the last decoding stage.


# End configuration section.
echo "$0 $@"  # Print the command line for logging

. ./cmd.sh
. ./path.sh
. ./utils/parse_options.sh

if ! cuda-compiled; then
  cat <<EOF && exit 1
This script is intended to be used with GPUs but you have not compiled Kaldi
with CUDA. If you want to use GPUs (and have them), go to src/, and configure
and make on a machine where "nvcc" is installed.
EOF
fi

# The iVector-extraction and feature-dumping parts are the same as the standard
# nnet3 setup, and you can skip them by setting "--stage 11" if you have
# already run those things.
local/nnet3/run_ivector_common.sh --stage $stage \
                                  --train-set $train_set \
                                  --gmm $gmm \
                                  --online-cmvn-iextractor $online_cmvn \
                                  --nnet3-affix "$nnet3_affix" || exit 1;

# Problem: We have removed the "train_" prefix of our training set in
# the alignment directory names! Bad!
gmm_dir=exp/$gmm
ali_dir=exp/${gmm}_ali_${train_set}_sp
tree_dir=exp/chain${nnet3_affix}/tree_sp${tree_affix:+_$tree_affix}
lang=data/lang_chain
lat_dir=exp/chain${nnet3_affix}/${gmm}_${train_set}_sp_lats
dir=exp/chain${nnet3_affix}/tdnn${affix}_sp
train_data_dir=data/${train_set}_sp_hires
lores_train_data_dir=data/${train_set}_sp
train_ivector_dir=exp/nnet3${nnet3_affix}/ivectors_${train_set}_sp_hires

for f in $gmm_dir/final.mdl $train_data_dir/feats.scp \
    $train_ivector_dir/ivector_online.scp \
    $lores_train_data_dir/feats.scp $ali_dir/ali.1.gz; do
  [ ! -f $f ] && echo "$0: expected file $f to exist" && exit 1
```

```
done

if [ $stage -le 10 ]; then
  echo "$0: creating lang directory $lang with chain-type topology"
  # Create a version of the lang/ directory that has one state per phone in the
  # topo file. [note, it really has two states.. the first one is only repeated
  # once, the second one has zero or more repeats.]
  if [ -d $lang ]; then
    if [ $lang/L.fst -nt data/lang/L.fst ]; then
      echo "$0: $lang already exists, not overwriting it; continuing"
    else
      echo "$0: $lang already exists and seems to be older than data/lang..."
      echo " ... not sure what to do.  Exiting."
      exit 1;
    fi
  else
    cp -r data/lang $lang
    silphonelist=$(cat $lang/phones/silence.csl) || exit 1;
    nonsilphonelist=$(cat $lang/phones/nonsilence.csl) || exit 1;
    # Use our special topology... note that later on may have to tune this
    # topology.
    steps/nnet3/chain/gen_topo.py $nonsilphonelist $silphonelist >$lang/topo
  fi
fi

if [ $stage -le 11 ]; then
  # Get the alignments as lattices (gives the chain training more freedom).
  # use the same num-jobs as the alignments
  steps/align_fmllr_lats.sh --nj 75 --cmd "$train_cmd" \
    ${lores_train_data_dir} \
    data/lang $gmm_dir $lat_dir
  rm $lat_dir/fsts.*.gz # save space
fi

if [ $stage -le 12 ]; then
  # Build a tree using our new topology.  We know we have alignments for the
  # speed-perturbed data (local/nnet3/run_ivector_common.sh made them), so use
  # those.  The num-leaves is always somewhat less than the num-leaves from
  # the GMM baseline.
  if [ -f $tree_dir/final.mdl ]; then
    echo "$0: $tree_dir/final.mdl already exists, refusing to overwrite it."
    exit 1;
  fi
  steps/nnet3/chain/build_tree.sh \
    --frame-subsampling-factor 3 \
    --context-opts "--context-width=2 --central-position=1" \
    --cmd "$train_cmd" \
    3500 ${lores_train_data_dir} \
    $lang $ali_dir $tree_dir
fi

if [ $stage -le 13 ]; then
  mkdir -p $dir
  echo "$0: creating neural net configs using the xconfig parser";
```

```
num_targets=$(tree-info $tree_dir/tree |grep num-pdfs|awk '{print $2}')
# JHL: The following line can be changed to Bash expr
learning_rate_factor=$(echo "print (0.5/$xent_regularize)" | python)

tdnn_opts="l2-regularize=0.03"
tdnnf_opts="l2-regularize=0.03 bypass-scale=0.66"
linear_opts="l2-regularize=0.03 orthonormal-constraint=-1.0"
prefinal_opts="l2-regularize=0.03"
output_opts="l2-regularize=0.015"

mkdir -p $dir/configs
cat <<EOF > $dir/configs/network.xconfig
input dim=100 name=ivector
input dim=40 name=input

# this takes the MFCCs and generates filterbank coefficients.  The MFCCs
# are more compressible so we prefer to dump the MFCCs to disk rather
# than filterbanks.
idct-layer name=idct input=input dim=40 cepstral-lifter=22
↪  affine-transform-file=$dir/configs/idct.mat
batchnorm-component name=batchnorm0 input=idct
spec-augment-layer name=spec-augment freq-max-proportion=0.5 time-zeroed-proportion=0.2
↪  time-mask-max-frames=20

delta-layer name=delta input=spec-augment
no-op-component name=input2 input=Append(delta, Scale(0.4, ReplaceIndex(ivector, t,
↪  0)))

# the first splicing is moved before the lda layer, so no splicing here
relu-batchnorm-layer name=tdnn1 $tdnn_opts dim=768 input=input2
tdnnf-layer name=tdnnf2 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=1
tdnnf-layer name=tdnnf3 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=1
tdnnf-layer name=tdnnf4 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=1
tdnnf-layer name=tdnnf5 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=0
tdnnf-layer name=tdnnf6 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=3
tdnnf-layer name=tdnnf7 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=3
tdnnf-layer name=tdnnf8 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=3
tdnnf-layer name=tdnnf9 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=3
tdnnf-layer name=tdnnf10 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=3
tdnnf-layer name=tdnnf11 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=3
tdnnf-layer name=tdnnf12 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=3
tdnnf-layer name=tdnnf13 $tdnnf_opts dim=768 bottleneck-dim=96 time-stride=3
linear-component name=prefinal-l dim=192 $linear_opts

## adding the layers for chain branch
prefinal-layer name=prefinal-chain input=prefinal-l $prefinal_opts small-dim=192
↪  big-dim=768
output-layer name=output include-log-softmax=false dim=$num_targets $output_opts

# adding the layers for xent branch
prefinal-layer name=prefinal-xent input=prefinal-l $prefinal_opts small-dim=192
↪  big-dim=768
```

```
  output-layer name=output-xent dim=$num_targets
↪  learning-rate-factor=$learning_rate_factor $output_opts
EOF
  steps/nnet3/xconfig_to_configs.py --xconfig-file $dir/configs/network.xconfig
   ↪  --config-dir $dir/configs/
fi

if [ $stage -le 14 ]; then
  if [[ $(hostname -f) == *.clsp.jhu.edu ]] && [ ! -d $dir/egs/storage ]; then
    utils/create_split_dir.pl \
     /export/b0{3,4,5,6}/$USER/kaldi-data/egs/mini_librispeech-$(date
↪  +'%m_%d_%H_%M')/s5/$dir/egs/storage $dir/egs/storage
  fi

  # JHL: we may have to change the num-jobs-xxx to 1.
  steps/nnet3/chain/train.py --stage=$train_stage \
    --cmd="$decode_cmd" \
    --feat.online-ivector-dir=$train_ivector_dir \
    --feat.cmvn-opts="--config=conf/online_cmvn.conf" \
    --chain.xent-regularize $xent_regularize \
    --chain.leaky-hmm-coefficient=0.1 \
    --chain.l2-regularize=0.0 \
    --chain.apply-deriv-weights=false \
    --chain.lm-opts="--num-extra-lm-states=2000" \
    --trainer.add-option="--optimization.memory-compression-level=2" \
    --trainer.srand=$srand \
    --trainer.max-param-change=2.0 \
    --trainer.num-epochs=20 \
    --trainer.frames-per-iter=3000000 \
    --trainer.optimization.num-jobs-initial=2 \
    --trainer.optimization.num-jobs-final=5 \
    --trainer.optimization.initial-effective-lrate=0.002 \
    --trainer.optimization.final-effective-lrate=0.0002 \
    --trainer.num-chunk-per-minibatch=128,64 \
    --egs.chunk-width=$chunk_width \
    --egs.dir="$common_egs_dir" \
    --egs.opts="--frames-overlap-per-eg 0 --online-cmvn $online_cmvn" \
    --cleanup.remove-egs=$remove_egs \
    --use-gpu=true \
    --reporting.email="$reporting_email" \
    --feat-dir=$train_data_dir \
    --tree-dir=$tree_dir \
    --lat-dir=$lat_dir \
    --dir=$dir  || exit 1;
fi

if [ $stage -le 15 ]; then
  # Note: it's not important to give mkgraph.sh the lang directory with the
  # matched topology (since it gets the topology file from the model).
  utils/mkgraph.sh \
    --self-loop-scale 1.0 data/lang_test_tgsmall \
    $tree_dir $tree_dir/graph_tgsmall || exit 1;
fi
```

```
if [ $stage -le 16 ]; then
  frames_per_chunk=$(echo $chunk_width | cut -d, -f1)
  rm $dir/.error 2>/dev/null || true

  for data in $test_sets; do
    (
      nspk=$(wc -l <data/${data}_hires/spk2utt)
      steps/nnet3/decode.sh \
          --acwt 1.0 --post-decode-acwt 10.0 \
          --frames-per-chunk $frames_per_chunk \
          --nj $nspk --cmd "$decode_cmd"  --num-threads 4 \
          --online-ivector-dir exp/nnet3${nnet3_affix}/ivectors_${data}_hires \
          $tree_dir/graph_tgsmall \
          data/${data}_hires ${dir}/decode_tgsmall_${data} || exit 1
      steps/lmrescore_const_arpa.sh --cmd "$decode_cmd" \
        data/lang_test_{tgsmall,tglarge} \
       data/${data}_hires ${dir}/decode_{tgsmall,tglarge}_${data} || exit 1
    ) || touch $dir/.error &
  done
  wait
  [ -f $dir/.error ] && echo "$0: there was a problem while decoding" && exit 1
fi

# Not testing the 'looped' decoding separately, because for
# TDNN systems it would give exactly the same results as the
# normal decoding.

if $test_online_decoding && [ $stage -le 17 ]; then
  # note: if the features change (e.g. you add pitch features), you will have to
  # change the options of the following command line.
  steps/online/nnet3/prepare_online_decoding.sh \
    --mfcc-config conf/mfcc_hires.conf \
    --online-cmvn-config conf/online_cmvn.conf \
    $lang exp/nnet3${nnet3_affix}/extractor ${dir} ${dir}_online

  rm $dir/.error 2>/dev/null || true

  for data in $test_sets; do
    (
      nspk=$(wc -l <data/${data}_hires/spk2utt)
      # note: we just give it "data/${data}" as it only uses the wav.scp, the
      # feature type does not matter.
      steps/online/nnet3/decode.sh \
        --acwt 1.0 --post-decode-acwt 10.0 \
        --nj $nspk --cmd "$decode_cmd" \
        $tree_dir/graph_tgsmall data/${data} \
        ${dir}_online/decode_tgsmall_${data} || exit 1
      steps/lmrescore_const_arpa.sh --cmd "$decode_cmd" \
        data/lang_test_{tgsmall,tglarge} \
        data/${data}_hires \
        ${dir}_online/decode_{tgsmall,tglarge}_${data} || exit 1
    ) || touch $dir/.error &
  done
  wait
```

```
  [ -f $dir/.error ] && echo "$0: there was a problem while decoding" && exit 1
fi

exit 0;
```

Setting up parameters

This is the first part of the script, starting after `set -euo pipefail`.

```
# First the options that are passed through to run_ivector_common.sh
# (some of which are also used in this script directly).
stage=0  # set stage for i-vector extraction
decode_nj=10  # number of decoding parallel jobs
train_set=train_clean_5  # train set
test_sets=dev_clean_2  # test set
gmm=tri3b  # folder to find the final hmm model (exp/tri3b/final.mdl)
nnet3_affix=  # affix for nnet3 (DNN) training related files and outputs

# The rest are configs specific to this script.  Most of the parameters
# are just hardcoded at this level, in the commands below.
affix=1j  # affix for the TDNN directory name
tree_affix=
train_stage=-10
get_egs_stage=-10
decode_iter=

# training options
# training chunk-options
chunk_width=140,100,160
common_egs_dir=
xent_regularize=0.1

# training options
srand=0
remove_egs=true
reporting_email=

#decode options
test_online_decoding=true  # if true, it will run the last decoding stage.

. ./cmd.sh
. ./path.sh
. ./utils/parse_options.sh
```

Note that at the end of the above script, we have set up command, Kaldi paths, and run the parsing tool, `parse_options.sh`. We can use `utils/parse_options.sh` to pass parameters using the `<--key value>` format.

We then check if Kaldi is compiled with CUDA since this scripts requires a GPU to run.

```
if ! cuda-compiled; then
  cat <<EOF && exit 1
This script is intended to be used with GPUs but you have not compiled Kaldi
with CUDA. If you want to use GPUs (and have them), go to src/, and configure
and make on a machine where "nvcc" is installed.
```

```
EOF
fi
```

Extracting the i-vector

```
# The iVector-extraction and feature-dumping parts are the same as the standard
# nnet3 setup, and you can skip them by setting "--stage 11" if you have already
# run those things.
local/nnet3/run_ivector_common.sh --stage $stage \
                                  --train-set $train_set \
                                  --gmm $gmm \
                                  --nnet3-affix "$nnet3_affix" || exit 1;
```

This command calls `local/nnet3/run_ivector_common.sh`.

Let's take a closer look at what is happening inside this script. To this end, let's see the code snippets from `local/nnet3/run_ivector_common.sh`.

Step 1: Checking required files

Check if required files `data/train_clean_5/feats.scp` and `exp/tri3b/final/mdl` exist.

```
# Skip the same parameter setups which are described before.

# Directory for the final hmm model
gmm_dir=exp/${gmm}
# Directory for training alignment (does not exist yet) for speed-perturbed data
ali_dir=exp/${gmm}_ali_${train_set}_sp

for f in data/${train_set}/feats.scp ${gmm_dir}/final.mdl; do
  if [ ! -f $f ]; then
    echo "$0: expected file $f to exist"
    exit 1
  fi
done
```

Step 2: Augmenting data and computing features

Here, we will first augment data; we prepare speed-perturbed data of `train_clean_5` and store them at `data/train_clean_5_sp`. Here, `sp` stands for speed perturbed.

We will then compute MFCC and CMVN of the speed-perturbed data.

```
if [ $stage -le 1 ]; then
  # Although the nnet will be trained by high resolution data, we still
  # have to perturb the normal data to get the alignment.
  # _sp stands for speed-perturbed
  echo "$0: preparing directory for low-resolution speed-perturbed data (for alignment)"
  utils/data/perturb_data_dir_speed_3way.sh \
    data/${train_set} data/${train_set}_sp
  echo "$0: making MFCC features for low-resolution speed-perturbed data"
  steps/make_mfcc.sh --cmd "$train_cmd" --nj 10 \
    data/${train_set}_sp || exit 1;
  steps/compute_cmvn_stats.sh data/${train_set}_sp || exit 1;
```

```
  utils/fix_data_dir.sh data/${train_set}_sp
fi
```

Step 3: Aligning with the low-resolution data

Here, we align the speed-perturbed data using `tri3b/final.mdl` and store them at `exp/${gmm}_ali_${train_set}_sp` which we set before.

```
if [ $stage -le 2 ]; then
  echo "$0: aligning with the perturbed low-resolution data"
  steps/align_fmllr.sh --nj 20 --cmd "$train_cmd" \
    data/${train_set}_sp data/lang $gmm_dir $ali_dir || exit 1
fi
```

Step 4: Aligning with the low-resolution data

First we do volume-perturbation (another data augmentation strategy) on speed-perturbed train set and test set (no speed-perturbation).

Then we extract high-resolution MFCCs and CMVNs on the volume- and speed-perturbed train set and original test set.

```
if [ $stage -le 3 ]; then
  # Create high-resolution MFCC features (with 40 cepstra instead of 13).
  echo "$0: creating high-resolution MFCC features"
  mfccdir=data/${train_set}_sp_hires/data

  for datadir in ${train_set}_sp ${test_sets}; do
    utils/copy_data_dir.sh data/$datadir data/${datadir}_hires
  done

  # Do volume-perturbation on the training data prior to extracting hires
  # features; this helps make trained nnets more invariant to test data volume.
  utils/data/perturb_data_dir_volume.sh \
    data/${train_set}_sp_hires || exit 1;

  for datadir in ${train_set}_sp ${test_sets}; do
    steps/make_mfcc.sh --nj 10 --mfcc-config conf/mfcc_hires.conf \
      --cmd "$train_cmd" data/${datadir}_hires || exit 1;
    steps/compute_cmvn_stats.sh data/${datadir}_hires || exit 1;
    utils/fix_data_dir.sh data/${datadir}_hires || exit 1;
  done
fi
```

The remaining part of `run_ivector_common.sh` is self-explanatory by its own comments and will not be discussed here.

Checking required files in `run_tdnn.sh`

Now, we are back to `run_tdnn.sh`. Let's check if required files exist.

```
# directory contains the final hmm model
gmm_dir=exp/$gmm
# directory contains the training alignments of speed perturbed data
ali_dir=exp/${gmm}_ali_${train_set}_sp
```

```
# directory to put the new decision tree, does not exist yet.
tree_dir=exp/chain${nnet3_affix}/tree_sp${tree_affix:+_$tree_affix}
# new lang directory with the new topology, does not exist yet.
lang=data/lang_chain
# directory to put lattices, does not exist yet.
lat_dir=exp/chain${nnet3_affix}/${gmm}_${train_set}_sp_lats
# directory to put other files, does not exist yet.
dir=exp/chain${nnet3_affix}/tdnn${affix}_sp
# directory of training set, which are the high-resolution MFCCs
train_data_dir=data/${train_set}_sp_hires
# directory of training set, which are the low-resolution MFCCs
lores_train_data_dir=data/${train_set}_sp
# directory contains i-vectors
train_ivector_dir=exp/nnet3${nnet3_affix}/ivectors_${train_set}_sp_hires

for f in $gmm_dir/final.mdl $train_data_dir/feats.scp \
    $train_ivector_dir/ivector_online.scp \
    $lores_train_data_dir/feats.scp $ali_dir/ali.1.gz; do
  [ ! -f $f ] && echo "$0: expected file $f to exist" && exit 1
done
```

Stage 10: Creating chain-type topology

Now, make a `$lang` directory and create a chain-type topology. Think this as an topology that is used for the Kaldi nnet3 DNN-HMM models. See Kaldi: 'Chain' models (kaldi-asr.org) for detailed explanations.

```
if [ $stage -le 10 ]; then
  echo "$0: creating lang directory $lang with chain-type topology"
  # Create a version of the lang/ directory that has one state per phone in the
  # topo file. [note, it really has two states.. the first one is only repeated
  # once, the second one has zero or more repeats.]
  if [ -d $lang ]; then
    if [ $lang/L.fst -nt data/lang/L.fst ]; then
      echo "$0: $lang already exists, not overwriting it; continuing"
    else
      echo "$0: $lang already exists and seems to be older than data/lang..."
      echo " ... not sure what to do.  Exiting."
      exit 1;
    fi
  else
    cp -r data/lang $lang
    silphonelist=$(cat $lang/phones/silence.csl) || exit 1;
    nonsilphonelist=$(cat $lang/phones/nonsilence.csl) || exit 1;
    # Use our special topology... note that later on may have to tune this
    # topology.
    steps/nnet3/chain/gen_topo.py $nonsilphonelist $silphonelist >$lang/topo
  fi
fi
```

Stage 11: Generating lattices from low-resolution MFCCs

```
if [ $stage -le 11 ]; then
  # Get the alignments as lattices (gives the chain training more freedom).
```

```
  # use the same num-jobs as the alignments
  steps/align_fmllr_lats.sh --nj 75 --cmd "$train_cmd"
    ${lores_train_data_dir} \
    data/lang $gmm_dir $lat_dir
  rm $lat_dir/fsts.*.gz # save space
fi
```

Stage 12: Building a new tree

Here, we build a new decision-tree using low-resolution MFCCs, the new topology, and the training alignments of speed-perturbed data.

```
if [ $stage -le 12 ]; then
  # Build a tree using our new topology.  We know we have alignments for the
  # speed-perturbed data (local/nnet3/run_ivector_common.sh made them), so use
  # those.  The num-leaves is always somewhat less than the num-leaves from
  # the GMM baseline.
  if [ -f $tree_dir/final.mdl ]; then
    echo "$0: $tree_dir/final.mdl already exists, refusing to overwrite it."
    exit 1;
  fi
  steps/nnet3/chain/build_tree.sh \
    --frame-subsampling-factor 3 \
    --context-opts "--context-width=2 --central-position=1" \
    --cmd "$train_cmd" 3500 ${lores_train_data_dir} \
    $lang $ali_dir $tree_dir
fi
```

Stage 13: Creating the config file for the DNN

See the original code.

Need to know what are the following two layers:

- `idct-layer`
- `spec-augment-layer`

Stage 14: Training the DNN

Finally, it is time to train the DNN use the high-resolution MFCCs, new decision tree, and i-vectors extracted before.

```
if [ $stage -le 14 ]; then
  steps/nnet3/chain/train.py --stage=$train_stage \
    --cmd="$decode_cmd" \
    --feat.online-ivector-dir=$train_ivector_dir \
    --feat.cmvn-opts="--norm-means=false --norm-vars=false" \
    --chain.xent-regularize $xent_regularize \
    --chain.leaky-hmm-coefficient=0.1 \
    --chain.l2-regularize=0.0 \
    --chain.apply-deriv-weights=false \
    --chain.lm-opts="--num-extra-lm-states=2000" \
    --trainer.add-option="--optimization.memory-compression-level=2" \
    --trainer.srand=$srand \
    --trainer.max-param-change=2.0 \
```

```
    --trainer.num-epochs=20 \
    --trainer.frames-per-iter=3000000 \
    --trainer.optimization.num-jobs-initial=2 \
    --trainer.optimization.num-jobs-final=5 \
    --trainer.optimization.initial-effective-lrate=0.002 \
    --trainer.optimization.final-effective-lrate=0.0002 \
    --trainer.num-chunk-per-minibatch=128,64 \
    --egs.chunk-width=$chunk_width \
    --egs.dir="$common_egs_dir" \
    --egs.opts="--frames-overlap-per-eg 0" \
    --cleanup.remove-egs=$remove_egs \
    --use-gpu=true \
    --reporting.email="$reporting_email" \
    --feat-dir=$train_data_dir \
    --tree-dir=$tree_dir \
    --lat-dir=$lat_dir \
    --dir=$dir  || exit 1;
fi
```

If the above code fails with an error of GPU out of memory, we need to change to use following parameter values:

```
    --trainer.optimization.num-jobs-initial=1 \
    --trainer.optimization.num-jobs-final=1 \
    --use-gpu=wait \
```

Stage 15: Compiling the final graph

```
if [ $stage -le 15 ]; then
  # Note: it's not important to give mkgraph.sh the lang directory with the
  # matched topology (since it gets the topology file from the model).
  utils/mkgraph.sh \
    --self-loop-scale 1.0 data/lang_test_tgsmall \
    $tree_dir $tree_dir/graph_tgsmall || exit 1;
fi
```