# Lab 1: Data Preparation and Feature Extraction

University of Edinburgh

January 23, 2018

The main goal of this lab is to get acquainted with Kaldi. We will begin by creating and exploring a data directory for the TIMIT dataset. Then we will extract features for TIMIT upon which we can train a complete speech recognition system in the coming labs. An underlying goal of this lab is to get you acquainted with Kaldi. Notes on UNIX commands are included in boxes; feel free to skip them if you're already familiar. *Most importantly, don't be afraid to ask questions when you get stuck.*

The work in this and future labs will use TIMIT. This corpus is interesting because it is *phonetically* labelled with 60 phone labels (and one end of sentence silence marker). TIMIT is often used as a benchmark for performance, although results on TIMIT are not always transferable to other corpora. Some historical results are shown in Figure 1.

## 1 Kaldi setup

First, lets set up a local directory where we can run experiments. Open a terminal window on DICE. Change directory to a directory we've set up for you, inserting your UUN in place of <UUN>:

```
cd /afs/inf.ed.ac.uk/group/teaching/asr/Work/<UUN>
```

> `cd dir` changes the directory to `dir`, and `ls` lists its contents. `cd ..` moves up one directory, `cd -` moves to the previous directory, and `cd ~` moves to your home directory.

This is your work directory. It is tedious to remember that long path, so lets go back to your home directory, create a soft link called `asrworkdir`, and `cd` back into it:

```
cd ~
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/Work/<UUN> asrworkdir
cd asrworkdir
```

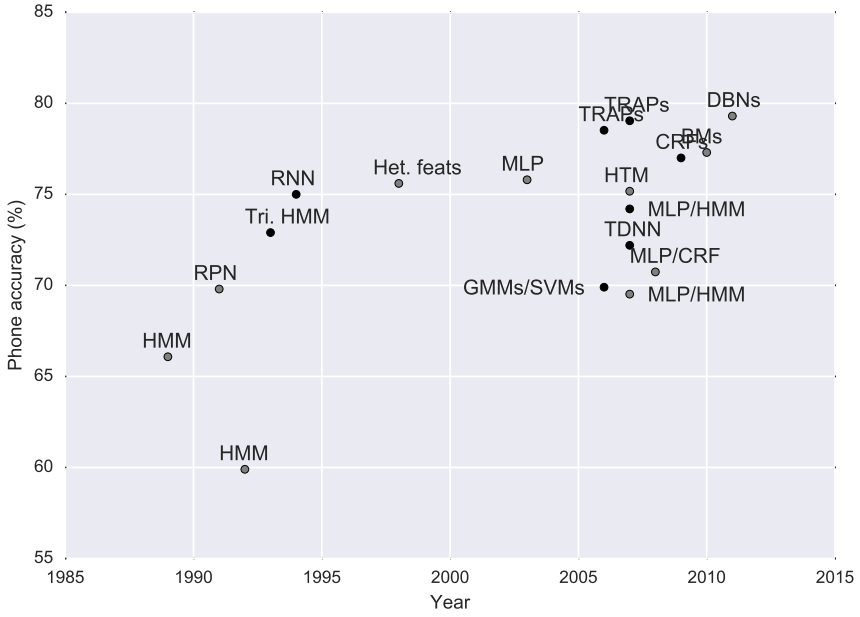



Figure 1: Results on TIMIT with various techniques. Most numbers from [1].

We'll now set up some files we need to run experiments in Kaldi. Run the following commands to create softlinks to your directory and to create a few empty directories. The dot means to create the links or copies in the current directory.

```
# First, we will create softlinks of some directories
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/steps .
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/utils .
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/local .
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/path.sh .

# Second, we will copy/make some directories so we can modify them
cp -r /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/conf .
mkdir data
mkdir exp
```

> `ln -s f1 f2` creates a soft link from `f1` to `f2`, so that any changes made to one will affect the other. When you want to copy instead, use `cp`: `cp -r dir1 dir2` copies the directory `dir1` to `dir2`, the `-r` (recursive) flag is required for directories.

Your work dir now has a typical directory structure for Kaldi. Type the following command to list its contents

```
ls
```

You should see the following files and folders:

**steps** contains scripts for creating an ASR system

**utils** contains scripts to modify Kaldi files in certain ways, for example to subset data directories into smaller pieces

**local** this directory typically contains files that relate only to the corpus we're working on (e.g. TIMIT). In this case it also may contain files we have provided for you.

**data** will contain any data directories, such as a `train` and `test` directory for TIMIT. We will create these below.

**exp** contains the actual experiments and models, as well as logs.

**conf** contains configurations for certain scripts that may read them. More on this later.

**path.sh** contains the path to the Kaldi source directory

Kaldi binaries are stored some place else than the experiment directory. To access them from anywhere, we set - inside the file `path.sh` - an environment variable `KALDI_ROOT` to point to the Kaldi installation. To set this variable type

```
source path.sh
```

or equivalently

```
. ./path.sh
```

To see whether it is set and where it points run

```
echo $KALDI_ROOT
```

> The command `echo` prints any string to the terminal along with any variables (e.g. `$KALDI_ROOT`). To omit newlines use the flag `-n`.

Similarly, the `steps` and `utils` directories are symbolic links to directories in the base install. `steps` includes essential scripts to train and decode ASR systems, while `utils` contains a number of scripts to for example manipulate the data directory. Local files pertaining to the current experiment go in `local`. This is a good place to put any utility scripts you write.

The `path.sh` file is called at the beginning of all Kaldi scripts, e.g. look at the first 18 lines of the script that computes MFCCs:

```
head -18 steps/make_mfcc.sh
```

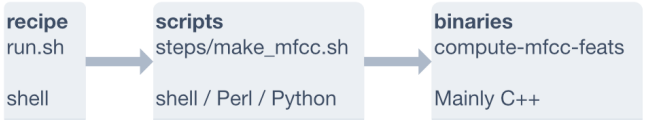
> `head -l file` and `tail -l file` prints the first and last `l` lines of `file`. A useful variation is `tail -n +l` which prints from line `l` to the end.

Now that the environment variables are set, try to run a typical Kaldi binary *without any arguments*, e.g.

```
feat-to-dim
```

It should provide an explanation of its purpose and usage instructions. This is common to all Kaldi binaries and scripts.

> Forgetting to run `source path.sh` if one of the most common mistakes. If you are getting errors like:
>
> `bash: feat-to-dim: command not found`
>
> Try to source path.sh and rerun the previous command.
>
> Another common mistake is running commands from other directories than ∼`/asrworkdir`, if you get an error, make sure that you are in ∼`/asrworkdir` with the `pwd` command.
>
> Other common errors can be found in Appendix 2.3.

> Kaldi comes with recipes for various corpora. These are typically embodied in a `run.sh` script in the main directory, with supporting files in `local`. This script will call high level scripts in `steps` and `utils`, which in turn call binaries which perform the actual computation.
>
> 
> 

# 2 TIMIT

We will create a data directory for TIMIT and extract features. We will write the commands one-by-one into the shell.

If you're new to Bash scripting or need a refresher, here's a few resources you may find useful. The first three expect no previous knowledge of Bash, the last is good to get to know many useful commands.

- BASH Programming - Introduction How-To: `http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`
- Advanced Bash-Scripting Guide: `http://www.tldp.org/LDP/abs/html/index.html`
- Bash Guide for Beginners: `http://www.tldp.org/LDP/Bash-Beginners-Guide/html/index.html`
- UNIX for Poets: `http://www.cs.upc.edu/~padro/Unixforpoets.pdf`

## 2.1 Data preparation

In the data preparation step we will create directories in `data` which will store any training and test sets, features and eventually a language model.

The first line sets the environment variables, *if path.sh exists*. It's a good idea to run this at the beginning of any Kaldi scripts:

```
[ -f ./path.sh ] && . ./path.sh
```

> `&&` will execute the next command if the previous succeeded (a typical Kaldi convention is using the opposite, `||` in its scripts, ending lines with `command || exit 1`, which means to exit the script with status 1 (error) if the preceding commands did *not* succeed).

Next, create we create data directories for TIMIT by running the following two lines. Don't worry about warnings of nonzero return status.

```
timit=/group/corporapublic/timit/original
local/timit_create_data.sh $timit
```

The data we just created is in the `data` directory. To appreciate better what this script does, navigate to the original TIMIT corpus training data directory and list its contents:

```
cd /group/corporapublic/timit/original/train
ls
```

It's split into multiple folders. Dive into the first and look at it

```
cd dr1
ls
```

Each of these directories represent a speaker. Move into the first speaker's directory and list the contents

```
cd fcjf0
ls
```

For each utterance there are four files: `.phn`, `.txt`, `.wav` and `.wrd`.

Look at each file in turn to figure out what they represent using the command `less`. You can use the up and down arrows to navigate. Hit `q` to exit.

```
less sa1.phn
less sa1.txt
less sa1.wav
less sa1.wrd
```

> `less` is useful when you only want to view a file and not edit it. It is also smart in that it doesn't read the entire file into memory at once, so files like `sa1.wav` which are not normally something you would look at, are handled neatly.

It's probably more interesting to listen to `sa1.wav`. On DICE you can write

```
play sa1.wav
```

Let's go back to our Kaldi work directory and see what we created with the command above:

```
cd ~/asrworkdir
```

Navigate to one of the created subdirs and look at the contents:

```
cd data/train
ls
```

The following files should be present. Have a look at each:

```
less text
less spk2utt
less wav.scp
less spk2gender
```

The script has combined all the information from the TIMIT directory we just looked at into files that neatly contain the information in a way that Kaldi can work with it.

The files are closely related by *utterance* and *speaker* ids, abbreviated to *utt_id* and *spk_id* in Figure 2. If each utterance is from a different speaker, or if we have no information about speakers, then the utterance and speaker ids match. Otherwise the speaker information is used to pool statistics across utterances for speaker adaptation and for speaker specific scoring. In the absence of a `segments` file, which sets out what portions of each audio file should be used for an utterance id, the recording ids are equivalent to the utterance ids. In this case we use the entire length of each audio file set out in `wav.scp`.

Change directory back to the main workdir:

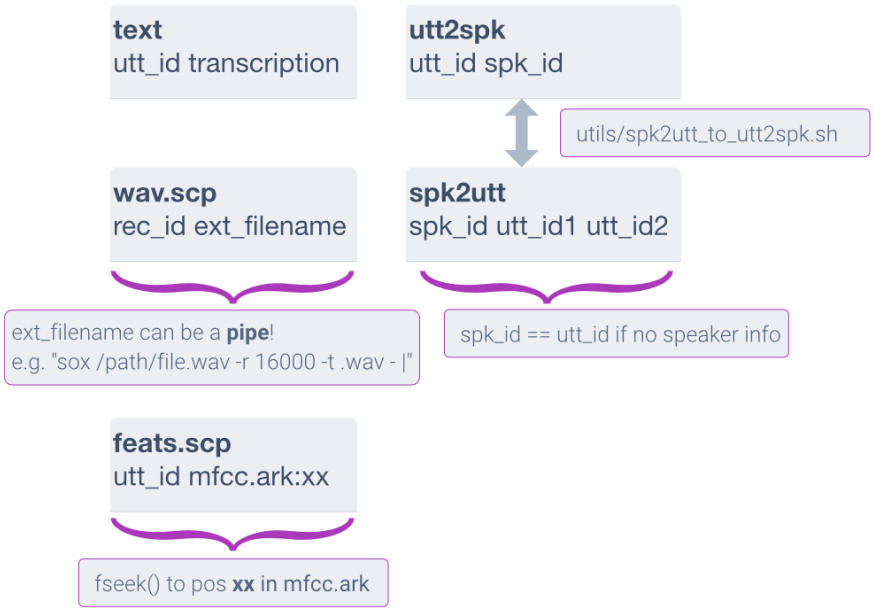



Figure 2: Illustration of a Kaldi data directory structure.

```
cd ~/asrworkdir
```

To check that the data directories conforms to Kaldi specifications, validate them by running the following two lines:

```
utils/validate_data_dir.sh data/train
utils/validate_data_dir.sh data/test
```

Uh oh. We're missing `utt2spk`, but we have `spk2utt`. These two files contain the same information, just with the mapping reversed. So we can easily convert one into the other. In the `utils` directory there is a file called `spk2utt_to_utt2spk.pl`. This is a Perl script which reads from `stdin` and writes to `stdout`. To pipe into the script we use `<`, to pipe out and into a file (instead of `stdout`) we use `>`. Run the following commands:

```
utils/spk2utt_to_utt2spk.pl < data/train/spk2utt > data/train/utt2spk
utils/spk2utt_to_utt2spk.pl < data/test/spk2utt > data/test/utt2spk
```

Run the validation scripts again. There should only be a `feats.scp` file missing, which we'll create next.

Have a look at the file you just created. How does it relate to `spk2utt`?

```
less data/train/utt2spk
less data/train/spk2utt
```

To see how many utterances there are in the training directory, we can use the command `wc`:

```
wc -l < data/train/utt2spk
```

- How many *speakers* are there in the training data?

## 2.2 Features

We'll now generate the features and the corresponding `feats.scp` script file, that will map utterance ids to positions in an archive, e.g. `feats.ark`.

For GMM-HMM systems we typically use MFCC or PLP features, and then apply cepstral mean and variance normalisation.

> For the next step it can be handy to use a `for` loop, to loop over directory names. In Bash the syntax is:
>
> ```
> for var in item1 item2 item3; do
>   echo $var;
> done
> ```
>
> This will print:
>
> ```
> item1
> item2
> item3
> ```

We will create MFCCs for our data. Run the following lines, which loops over the data directories and extracts features for each.

```
for dir in train test; do
  steps/make_mfcc.sh data/$dir exp/make_mfcc/$dir mfcc
done
```

This will have created `feats.scp` with corresponding archives in a folder called `mfcc` and written log files to `exp/make_mfcc`.

- You will now compute cepstral mean and variance normalisation statistics for the data. Find the appropriate script in the `steps` folder - perhaps using `ls steps/*cmvn*`. The run the script as above:

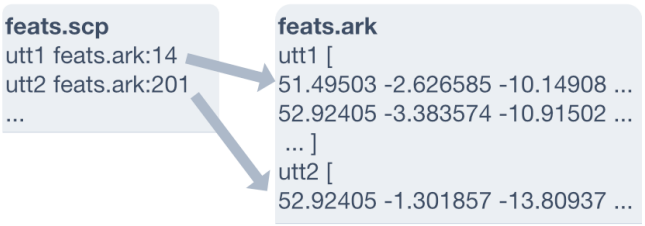
  ```
  for dir in train test; do
    steps/<insert-script-here> data/$dir
  done
  ```

  This will create `cmvn.scp` in each data directory.

- Validate the data directory again.

### 2.2.1 Script and archives (*.scp, *.ark)

`scp` files map utterance ids to positions in `ark` files. The latter contain the actual data. Kaldi binaries generally read and write script and archives interchangeably, as long as the filename is prepended with the type of file you wish to read or write, e.g. `scp:feats.scp` or `ark:mfcc.ark` or `ark:-` to write to stdout. Archives will be written in binary, unless you append the `,t` modifier: `ark,t:mfcc.ark`.




For more see the documentation on Kaldi I/O mechanisms, see: `http://kaldi-asr.org/doc/io.html#io_sec_tables`

Kaldi binaries typically read and/or write script and archive files. When this is the case, the usage message will show `rspecifier` or `wspecifier`. Scripts and archives represent the same data, so passing either to a program yields the same results.

Let's try using the programme `feat-to-dim` to find the dimensions of the features we just created:

```
feat-to-dim scp:data/train/feats.scp -
feat-to-dim ark:mfcc/raw_mfcc_train.1.ark -
```

Are they the same?

Let's have a look at the actual features too. The archives are by default written in binary, but we can make a suitable copy using the program `copy-feats` and a suitable write specifier (see box above). We pipe it into `head` to avoid overflowing the terminal window:

```
copy-feats scp:data/train/feats.scp ark,t:- | head
```

Do the features match what you got from `feat-to-dim`?

Read specifiers can take bash commands ending with a pipe (`|`) as arguments. This can be handy if you only want to look at the features for a particular utterance.

- Try replacing the read specifier `scp:data/train/feats.scp` in your previous solution with

  ```
  'scp:grep fdfb0_sx58 data/train/feats.scp |'
  ```

What does this do?

> `grep` will search for a string in a file and output that entire line by default: `grep string filename`. The string could be a regex query and there are a lot of options. See `man grep` for more.

- Try the same trick as above, but find how many frames that utterance has using the program `feat-to-len`.

While *write specifiers* can write to `stdout` (e.g. `ark:-`), *read specifiers* can read from `stdin`. What does the following command do? This syntax is crucial to piping Kaldi programmes together.

```
head -10 data/train/feats.scp | tail -1 | copy-feats scp:- ark,t:- | head
```

`steps/make_mfcc.sh`, which you ran above, will use the program `compute-mfcc-feats` to extract features. This program looks for `conf/mfcc.conf` in the `conf` folder for any non-default parameters. These are passed to the corresponding binaries.

Look at that file

```
less conf/mfcc.conf
```

and compare it to the options for the program by running it without arguments:

```
compute-mfcc-feats
```

If you have time, let's combine what we've learned and create filterbank features.

- Create copies of your data directories and generate filterbank and pitch features for each (look in the `steps` folder for a suitable script). However, first create a `conf/fbank.conf` file (using some text editor or see box below). Include an argument to set the dimension of the filterbank features to 40. (Hint: Look at `compute-fbank-feats` for arguments). Finally, check the feature dimension and make sure it is 43 (there are three pitch features).

> To open or create a file in `nano`, type
>
> `nano conf/fbank.conf`
>
> Inside `nano`, use the arrow keys to move around the text file. To exit, hit `ctrl+X` and hit `Y` or `N` to the question of whether to save any changes or not. Other commands are listed at the bottom of the window.

We're done! Next time we'll build a GMM-HMM system.

### 2.3 Appendix: Common errors

- Forgot to source `path.sh`, check current path with `echo $PATH`
- No space left on disk: check `df -h`
- No memory left: check `top` or `htop`
- Lost permissions reading or writing from/to AFS: run `kinit && aklog`. To avoid this, run long jobs with the `longjob` command.
- Syntax error: check syntax of a Bash script without running it using `bash -n scriptname`
- Avoid spaces after \when splitting Bash commands over multiple lines
- Optional params:
- command line utilities: `--param=value`
- shell scripts: `--param value`
- Most file paths are absolute: make sure to update the paths if moving data directories
- Search the forums: `http://kaldi-asr.org/forums.html`
- Search the old forums: `https://sourceforge.net/p/kaldi/discussion`

### 2.4 Appendix: UNIX

- `cd dir` - change directory to `dir`, or the enclosing directory by `..`
- `cd -` - change to previous directory
- `ls -l` - see directory contents
- `less script.sh` - view the contents of `script.sh`
- `head -l` and `tail -l` - show first or last `l` lines of a file
- `grep text file` - search for `text` in `file`
- `wc -l file` - compute number of lines in `file`

## References

[1] Carla Lopes and Fernando Perdigao. Phone recognition on the timit database. *Speech Technologies/Book*, 1:285–302, 2011.

# Lab 2: Training monophone models

University of Edinburgh

January 29, 2018

Last time we begun to get familiar with some of Kaldi's tools and set up a data directory for TIMIT. This time we will train Hidden Markov Model-Gaussian Mixture Model (HMM-GMM) systems on top of those features. To be clear about what exact commands need to be run or written, commands that you should run are now in a box with a red border, and notes in a box with a blue border:

```
Code to execute will appear in red boxes.
```

Notes will appear in blue boxes.

## 1 Building an acoustic model

Let's begin by opening a terminal window and `cd` to your workdir:

```
cd ~/asrworkdir
source path.sh
```

Before we start building the HMM-GMM models run the following script to check that you have the necessary files from the previous lab:

```
./local/lab1_check.sh
```

If that says everything is fine then continue to the next step.

### 1.1 Monophone models

Let's start training a monophone system. Kaldi has a script called `steps/train_mono.sh`. If we run it without any arguments we get the following usage message:

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

Kaldi needs a data directory and a language directory, and will store the model in the experiment directory. Plugging in for those, run the following command:

```
steps/train_mono.sh  --nj 4 data/train data/lang exp/mono
```

The option `--nj 4` instructs Kaldi to split computation into four parallel jobs. This can significantly reduce computation time, but only as long as there are at least equal number of processor cores which are not in use.

Building the monophone system may take a while, so open another terminal window and `cd` again to the workdir:

```
cd ~/asrworkdir
source path.sh
```

A folder we didn't look at much last time was the language directory, `data/lang`. Let's begin to have a look at the phones that are defined in our model:

```
less data/lang/phones.txt
```

How many phones have we defined in our model? (hint: use `wc`)
Similarly, the following file lists the words in our language model.

```
less data/lang/words.txt
```

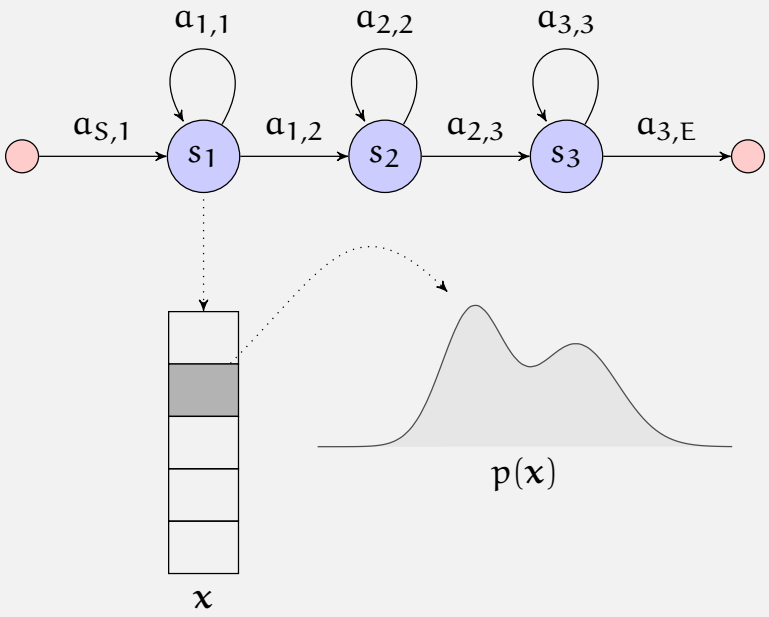
Are they similar? Why? Think about what we're trying to achieve with TIMIT.

Next, have a look at the topology by typing

```
less data/lang/topo
```

This sets out normal three-state HMMs per phone. The transition probabilities are defined after the <`Transition`> tags.

Recall from the lectures how we use HMMs and GMMs to model the data. Relate this figure to the topology and the acoustic model generated in Kaldi.




Compare the topology to the phones we just looked at. Which phone has a different topology? How is it different? Can you relate topology to the figure above?

By now the initial monophone model should have been written. Inspect it by running the following command:

```
gmm-copy --binary=false exp/mono/0.mdl - | less
```

The top should look familiar. Underneath the topology there will be a <`Triples`> tag. This maps each phone and one of its three states to a unique number. That is, there are $num\_phones \times num\_states = 144$ triples. Scroll down a few hundred lines more until you see tags such as <`DiagGMM`>. Can you work out what kind of information is stored here? How does it relate to the `Triples` in the topology? (How many PDFs are there?)

Last time we mentioned that binaries that take read or write *specifiers*, such as `ark:`, write in binary by default unless you append the `,t` flag (e.g. `ark,t:`). For binaries that read normal files, you instead provide the flag `--binary=false`, as above.

To get a summary of the information in the initial acoustic model, write

```
gmm-info exp/mono/0.mdl
```

When the monophone models are finished training, have a look at the final trained model:

```
gmm-info exp/mono/final.mdl
```

What has changed?
We can have a look at some of the mixture Gaussians by running:

```
gmm-copy --binary=false exp/mono/final.mdl - |\
  python2.7 local/plot_gmm.py -
```

Note the backslash at the end of the line: \ . This allows you to write bash expressions over multiple lines and can help readability in scripts.

If you get errors that you are missing a Python library, you can install these libraries by running:

```
pip install --user numpy scipy matplotlib
```

You should see something similar to Figure 1. These are the Gaussian Mixture Model densities corresponding to the first twelve cepstral coefficients for one of the states in the model.

This is a good time to dig a little bit deeper into the scripts. Have a look at the monophone training script:

```
less steps/train_mono.sh
```

At the top there are listed some default parameters. About halfway through there's a parameter called `totgauss`. This sets the upper limit of the number of Gaussians to include in total in the model. Search for the string `gmm-est` by typing (within `less`):

```
/gmm-est
```

This is the command that actually estimates the GMMs. Notice that the script is passing in a mix-up parameter, which is based upon the `totgauss` variable above. Leave `less` by pressing `q`.

When you're using `less` to look at a file, you can search for text in that file by typing `/` followed immediately by the string you're interested in and hitting enter. Press `n` repeatedly to loop through all occurrences of the search string.

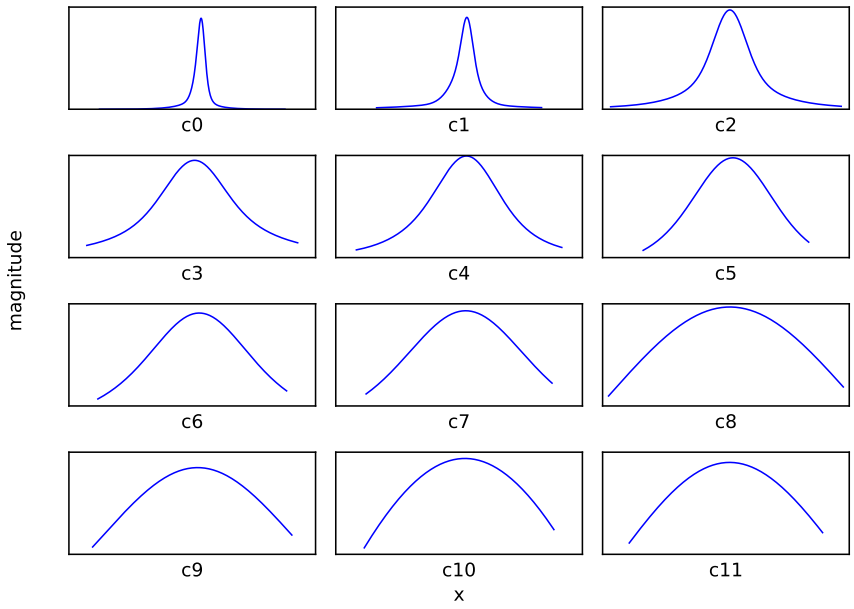



Figure 1: GMMs for the first 12 cepstral coefficients for a random density.

Lets start decoding our monophone model. Run the following commands:

```
utils/mkgraph.sh --mono data/lang_test_bg exp/mono exp/mono/graph
steps/decode.sh --nj 4 \
 exp/mono/graph data/test exp/mono/decode_test
```

While we're decoding, in the other window, have a look at the alignments from our monophone model. The alignments are stored compressed using `gzip`. Note that in the below command we unzip them and pipe them directly in as the read specifier. We then apply the `,t` tag to get readable output from the write specifier and finally we output only the first line.

```
convert-ali --reorder=false exp/mono/0.mdl \
  exp/mono/0.mdl exp/mono/tree \
  ark:"gunzip -c exp/mono/ali.1.gz |" ark,t:- | head -1
```

These numbers are called transition-ids. To see how they relate to our model, type

```
show-transitions data/lang/phones.txt exp/mono/0.mdl
```

Can you see how this relates to the topology we looked at above? We can combine this information and display it in a more friendly way. Type the

following command:

```
show-alignments data/lang/phones.txt exp/mono/0.mdl \
  "ark:gunzip -c exp/mono/ali.1.gz |" | head -n 2
```

The first line now contains the same transition ids, but grouped in a particular way (how?). The second line shows the phones.

Above we used the command `convert-ali` to view the alignments stored in `ali.1.gz`. However, if we just wanted to see exactly what was in the file, we could have just used:

```
copy-int-vector "ark:gunzip -c exp/mono/ali.1.gz|" \
  ark,t:- | head -n 1
```

Note that the output is different:

```
2 4 3 3 3 3 3 3 6 5 5 5 5
```

compared to

```
2 3 3 3 3 3 3 4 5 5 5 5 6
```

Here, id 4 indicates a transition from state 1 to 2, and id 3 indicates a self-loop in state 1. So in the first example we transition to state 2, and then perform a self-loop in state 1. It turns out Kaldi *reorders* transition probabilities, effectively placing the current state's self-loops at the end of the forward transition to the next state. This is done to optimise decoding later on, but doesn't affect the actual probabilities or results.

---

If you haven't signed up for Piazza, it is strongly recommended. It's a great place to post any questions and discuss the material in the course.

## 1.2 Decoding

Above we set off some jobs creating a graph and decoding the model. For this we're using a slightly different language directory which contains a bigram model of the phones, stored in `data/lang_test_bg/G.fst`. The first command above (`utils/mkgraph.sh`) combines the HMM structure in the trained model, any Context dependency (next lab), the Lexicon and the Grammar - collectively termed HCLG - and creates a *decoding graph* in the form of a Finite State Transducer (FST)[1]. The second script generates lattices of word (phone) sequences for the data given the model, which we then score.

```
grep Sum exp/mono/decode_test/score_*/*.sys | utils/best_wer.sh
```

[1] [1] is a good, related paper on FSTs in speech recognition.

What Phone Error Rate (PER) does the monophone model achieve on TIMIT? Note that Kaldi typically reports Word Error Rates, which is why the output says WER.

We're done! Next time we'll look at triphone and hybrid neural network models.

## 1.3 Appendix: Common errors

- Forgot to source `path.sh`, check current path with `echo $PATH`
- No space left on disk: check `df -h`
- No memory left: check `top` or `htop`
- Lost permissions reading or writing from/to AFS: run `kinit && aklog`. To avoid this, run long jobs with the `longjob` command.
- Syntax error: check syntax of a Bash script without running it using `bash -n scriptname`
- Avoid spaces after \when splitting Bash commands over multiple lines
- Optional params:
- command line utilities: `--param=value`
- shell scripts: `--param value`
- Most file paths are absolute: make sure to update the paths if moving data directories
- Search the forums: `http://kaldi-asr.org/forums.html`
- Search the old forums: `https://sourceforge.net/p/kaldi/discussion`

## 1.4 Appendix: UNIX

- `cd dir` - change directory to `dir`, or the enclosing directory by `..`
- `cd -` - change to previous directory
- `ls -l` - see directory contents
- `less script.sh` - view the contents of `script.sh`
- `head -l` and `tail -l` - show first or last `l` lines of a file
- `grep text file` - search for `text` in `file`
- `wc -l file` - compute number of lines in `file`

# References

[1] Mehryar Mohri, Fernando Pereira, and Michael Riley. Speech recognition with weighted finite-state transducers. In *Springer Handbook of Speech Processing*, pages 559–584. Springer, 2008.

# Lab 3: Word recognition and triphone models

University of Edinburgh

February 5, 2018

So far we have been working with phone recognition on TIMIT. In this lab we will move instead to word recognition. This requires a word-level language model, for which we will be using the Wall Street Journal (WSJ) language model.

**Path errors**

If you get errors such as `command not found`, try sourcing the path again:

```
source path.sh
```

In general, every time you open a new terminal window and `cd` to the work directory, you would need to source the path file in order to use any of the Kaldi binaries.

---

There is an appendix at the end of every lab with the most typical mistakes.

## 1 Word recognition

Let's begin by opening a terminal window, `cd` to your workdir and sourcing the path:

```
cd ~/asrworkdir
```

```
source path.sh
```

Before we start building new models run the following script to gather some new files we need to do word recognition:

```
./local/lab3_setup.sh
```

The message "`lab 3 preparation succeeded`" should appear.

## 1.1 Word-level language model

When you ran the above command it will have generated a new language model directory: `data/lang_wsj`, new training and test directories, and a monophone experiment directory that we have trained for you already.

There's a crucial difference to the new language model directory from the previous one. Compare the `words.txt` file in each:

```
less data/lang/words.txt
```

```
less data/lang_wsj/words.txt
```

What is different between the two? Let's also have a look at the new phone-set:

```
less data/lang_wsj/phones.txt
```

Our phones are now dependent upon where they occur in the words. For an explanation of the suffixes, see

```
less data/lang_wsj/phones/word_boundary.txt
```

To see how this relates to the lexicon, open the following file:

```
less data/lang_wsj/phones/align_lexicon.txt
```

Now search (remember `/`) for "`^ SPEECH`", which will show the first occurrence of a word that starts with "`SPEECH`". How does the mapping relate to the position-dependent phones we saw above?

Finally, have a look at the new training directory which now maps from utterance to words instead of phones:

```
less data/train_words/text
```

## 1.2 Monophone models

We have provided monophone models and alignments, as well as a decode on the test set. These are placed in `exp/words/mono` and `exp/words/mono_ali`.

Have a look at the score for the decode on the test set:

```
more exp/word/mono/decode_test/scoring_kaldi/best_wer
```

**more**

The command `more` is very similar to `less`, apart from that it prints the output to the console directly.

Ok! There's lots of room for improvement...

## 1.3 Triphone models

Let's start training a triphone model with delta and delta-delta features.

**Backslashes**

Backslashes in Bash (\) which is present in the next box, are simply a way of splitting commands over multiple lines. That is, the following two commands are identical:

```
some_script.sh --some-option somefile.txt
```

and

```
some_script.sh --some-option \
  somefile.txt
```

You may remove the backslash and type these commands on a single line. But sometimes when writing scripts, avoiding too long commands on a single line can help readability.

---

Be careful about spaces after \. Bash will expect a newline immediately, and a space here before the newline will make a script crash.

Run the following command to train a triphone system. This might take a few minutes...

```
steps/train_deltas.sh 2500 15000 data/train_words \
  data/lang_wsj exp/word/mono_ali exp/word/tri1
```

- While that is running, open another terminal window, change directory to the work directory and source the path.

## 1.4 Delta features

Above we started running a script called `train_deltas.sh`. This trains triphone models on top of `MFCC+delta+deltadelta` features. To avoid having to store features with `delta+deltadelta` applied, Kaldi adds this in an online fashion, just as another pipe, using the programme `add-deltas`. Remember how we checked the feature dimension of the features in the first lab? Run the following command.

```
feat-to-dim scp:data/train/feats.scp -
```

What do you expect the dimension to be be after applying `add-deltas`? Run the following command.

```
add-deltas scp:data/test/feats.scp ark:- | feat-to-dim ark:- -
```

## 1.5 Logs

Kaldi creates detailed logs during training. These can be very helpful when things go wrong. By now we should have some of the first ones created for the triphone training:

```
less exp/word/tri1/log/acc.1.1.log
```

Notice that on the top, the entire command which Kaldi ran (as set out by the script) is displayed. For this example it runs a command called `gmm-acc-stats-ali`, and then if you look closely there is a feature pipeline using the programmes `apply-cmvn` and `add-deltas` which applies these transforms and additions to the features in an online fashion.

## 1.6 Triphones

When we ran the triphone modelling script above we also passed two numbers, 2500 and 15000. These are respectively the number of leaves in the decision tree and the total number of Gaussians across all states in our model.

**Triphone clustering**

As you may recall from the lectures, having a separate model for each triphone is generally not feasible. With typically 48 phones we would require $48 \times 48 \times 48$, i.e. more than 110000 models. We don't have enough data to see all those, so we cluster them using a decision tree. The *number of leaves* parameter then sets the maximum number of leaves in the decision tree, and the *number of gaussians* the maximum number of Gaussians distributed across the leaves. So on average our model will have an average of $\frac{numgauss}{numleaves}$ Gaussians per leaf.

To see how many states have been seen, run the following command:

```
sum-tree-stats --binary=False - \
  exp/word/tri1/1.treeacc | head -1
```

The first number indicates the number of states with statistics. Dividing that number by three will roughly give the number of seen triphones (why is this?).

Let's have a closer look at the clustering in Kaldi. It's also a good opportunity to look a bit deeper at the Kaldi scripts. Open the training script we just ran by typing

```
less steps/train_deltas.sh
```

At the top is a configuration section with several default parameters. These are the parameters that the script can take by passing `--param setting` to the script when running it. Most scripts in Kaldi are set up this way. The first one is a variable called `stage`, this can be really useful to start a script partway through. Scroll down till line 88 which says

```
if [ $stage -le 2 ]; then
```

This is saying that if the stage variable is less than or equal to two, run this section. This is the section that builds the decision tree. It first calls a binary called `cluster-phones`, this uses e.g. k-means clustering to cluster similar phones. These clusters will be the basis for the questions in the decision tree. Kaldi doesn't use predefined questions such as "is the left phone a fricative?", but rather estimates them from the data. It writes these using the numeric phone identities to a file called `exp/word/tri1/questions.int`. Let's look at it using a utility that maps the integer phone identities to their more readable names. Leave `less` by pressing `q` and run the next command:

```
utils/int2sym.pl data/lang_wsj/phones.txt \
  exp/word/tri1/questions.int | less
```

Each line is a cluster. Some of these clusters are really large, but hopefully some of them should make sense. We use these to build a clustering tree, stored in `exp/word/tri1/tree`. Exit `less` and run the following command:

```
copy-tree --binary=false exp/word/tri1/tree - | less
```

This file sets out the entire clustering tree. We'll only get a gist of what it represents by looking at a few lines. The further down this file, the deeper we move into the tree. Move down to line 50 by typing the following letters on the keyboard while in `less`:

```
50g
```

You should see a line that says

```
SE 0 [ 40 41 42 43 92 93 94 95 96 97 98 99 228 229 230 231 ]
```

The numbers in the brackets are phones. By now you should be able to figure out which phones these represent (hint: language directory). `SE` stands for "SplitEventMap", which is essentially a split in the tree. The following number is typically either 0, 1, or 2, and they stand for left, centre and right. That is, for this line we are asking whether the *left* (0) phone is one of 40, 41, 42, ..., 231. If that is true, we

- proceed to the next line, asking whether the left phone is one of `[40,41,42,43]`, if that is true we
- proceed to the next line, this time asking whether the *right* (2) context is one of `[1,2,3,4,5]` (which phone is this?)

Finally, we proceed to the next line where `CE` stands for a ConstantEventMap and indicates a leaf in tree. If our previous question was true then we choose pdf-id 1302, and if not, we choose 1091. The subtree we have been looking at is illustrated in Figure 1. Can you tell how we would get pdf-id 1039?

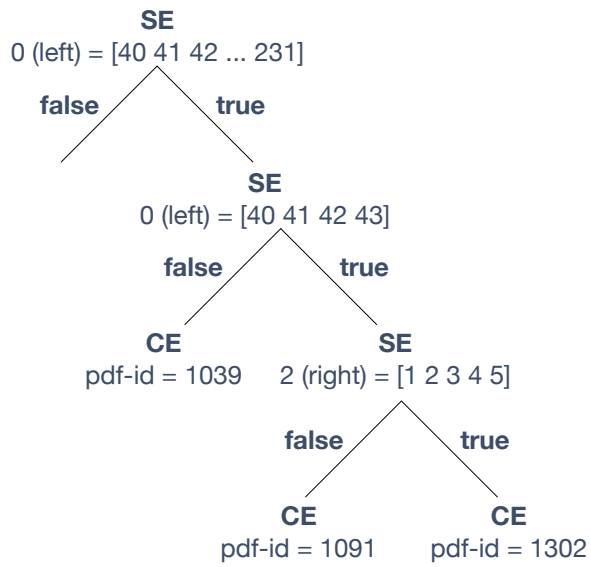



Figure 1: Tree clustering example

Ok, by now the triphone system should have finished training. Let's create a decoding graph and decode our triphone system, this may take a few minutes:

```
utils/mkgraph.sh data/lang_wsj_test_bg \
  exp/word/tri1 exp/word/tri1/graph
```

```
steps/decode.sh --nj 4 exp/word/tri1/graph \
  data/test_words exp/word/tri1/decode_test
```

When the decoding have finished, score the directory by running:

```
local/score_words.sh data/test_words exp/word/tri1/graph \
  exp/word/tri1/decode_test
```

We can then have a look at the WER by typing:

```
more exp/word/tri1/decode_test/scoring_kaldi/best_wer
```

That should be considerably better than the monophone system, but there's still lots of room for improvement. The typical progression in Kaldi would now be to train a system on top of decorrelated features and then train a system with speaker adaptive training. But that's for another lab...

We're done! Next time we'll look at training hybrid neural network models.

## 1.7 Appendix: Common errors

- Forgot to source `path.sh`, check current path with `echo $PATH`
- No space left on disk: check `df -h`
- No memory left: check `top` or `htop`
- Lost permissions reading or writing from/to AFS: run `kinit && aklog`. To avoid this, run long jobs with the `longjob` command.
- Syntax error: check syntax of a Bash script without running it using `bash -n scriptname`
- Avoid spaces after \when splitting Bash commands over multiple lines
- Optional params:
- command line utilities: `--param=value`
- shell scripts: `--param value`
- Most file paths are absolute: make sure to update the paths if moving data directories
- Search the forums: `http://kaldi-asr.org/forums.html`
- Search the old forums: `https://sourceforge.net/p/kaldi/discussion`

## 1.8 Appendix: UNIX

- `cd dir` - change directory to `dir`, or the enclosing directory by `..`
- `cd -` - change to previous directory
- `ls -l` - see directory contents
- `less script.sh` - view the contents of `script.sh`
- `head -l` and `tail -l` - show first or last `l` lines of a file
- `grep text file` - search for `text` in `file`
- `wc -l file` - compute number of lines in `file`

# Lab 4: Hybrid Acoustic Models

University of Edinburgh

March 19, 2018

This is the final lab, in which we will have a look at training hybrid neural network acoustic models using a frame-level cross-entropy loss. We will continue using the word level models from the last lab for alignments.

**Path errors**

If you get errors such as `command not found`, try sourcing the path again:

```
source path.sh
```

There is an appendix at the end of every lab with the most typical mistakes.

## 1 Setup

Let's begin by opening a terminal window, `cd` to your workdir and sourcing the path:

```
cd ~/asrworkdir
```

```
source path.sh
```

Before we start building new models run the following script to gather some new files we need to do word recognition:

```
./local/lab4_setup.sh
```

The message "`Lab 4 preparation succeeded`" should appear.

### 1.1 Alignments

Before we train the neural networks, lets try to get better alignments by making a stronger acoustic model.

Align the system from the last lab by running

```
steps/align_si.sh --nj 4 data/train_words \
  data/lang_wsj exp/word/tri1 exp/word/tri1_ali
```

### Alignments

Why do we have a separate step to align between model training? In Kaldi, we typically do a pass of alignment between each training phase. This is just to ensure that we have the absolute latest alignments for the latest model in each stage. Technically we could skip these parts, and most scripts that take alignment experiment directories will also accept a normal training experiment directory (e.g. `tri1` vs. `tri1_ali`).

Train a system on top of LDA+MLLT features, using the `tri1_ali` alignments:

```
steps/train_lda_mllt.sh \
  --splice-opts "--left-context=3 --right-context=3" \
  2500 15000 data/train_words data/lang_wsj \
  exp/word/tri1_ali exp/word/tri2
```

### LDA+MLLT

LDA and MLLT transformations have not been covered in the course. We provide a very brief introduction here. For more information see [1][a].

We first splice together 7 frames (`left` and `right-context=3` above) of the MFCC features. The dimensionality is then reduced to 40 using Linear Discriminant Analysis (LDA) using the acoustic states as classes. How much of a dimensionality reduction is this, given the 13-dimensional MFCCs and the context window?

Finally, during training we estimate a transform known as Maximum Likelihood Linear Transform (MLLT). Don't confuse this with Maximum Likelihood Linear Regression (MLLR), which we use for adaptation. Normally in HMM-GMM systems, we model the emission probabilities using diagonal covariance matrices for the GMMs. This is because there would otherwise be far too many parameters to estimate. However, modelling without covariances assumes that each element of the feature vectors (e.g. MFCCs) are independent. MLLT is a way to loosen this assumption somewhat, by sharing a few full covariance matrices across many distributions. This models some of the covariances, without the corresponding explosion in the number of parameters.

[a] `http://mi.eng.cam.ac.uk/~sjy/papers/gayo07.pdf`

Finally, align the system once more so that we have the latest possible alignments for the neural networks:

```
steps/align_si.sh --nj 4 data/train_words \
  data/lang_wsj exp/word/tri2 exp/word/tri2_ali
```

## 2 Neural networks

Kaldi comes with three neural network toolkits. We will use `nnet1`[1]. First, let's separate the data into training and validation data. The script `utils/subset_data_dir_tr_cv.sh` by default separates data into 90% for training and 10% for validation:

```
dir=data/train_words
utils/subset_data_dir_tr_cv.sh $dir ${dir}_tr90 ${dir}_cv10
```

This will have created these directories:

```
data/train_words_tr90
data/train_words_cv10
```

We will begin training a fairly small neural network model:

```
steps/nnet/train.sh --hid-layers 2 --hid-dim 256 --splice 5 \
  --learn-rate 0.008 \
  --skip-cuda-check true \
  data/train_words_tr90 data/train_words_cv10 data/lang_wsj \
  exp/word/tri2_ali exp/word/tri2_ali exp/word/nnet
```

**Important**

Since we are running this on a CPU, this will likely take longer than the lab time to complete (training on a GPU would be 10-20x faster). If it has not finished by the time the lab is over, just cancel it by hitting `ctrl+c`. Instead, we have provided a fully trained model for you for this lab, in `exp/word/tri3_nnet`.

Have a look at the training script. Open another terminal window and run:

```
cd ~/asrworkdir
less steps/nnet/train.sh
```

Scroll down to the section that begins with "`###### PREPARE ALIGNMENTS`". There are two important events occurring in this section, we first generate target

[1] `http://kaldi-asr.org/doc/dnn1.html`

labels using the alignments we created above. Then, we generate counts of the PDFs corresponding to the phones in the alignments. Convince yourself of where this is happening in the code.

Let's have a closer look at both cases.

## 2.1 Labels

Look at the labels by running the following command

```
ali-to-pdf exp/word/tri2_ali/final.mdl \
  "ark:gunzip -c exp/word/tri2_ali/ali.1.gz |" ark:- | \
   ali-to-post ark:- ark,t:- | less
```

Can you relate the first number in each bracket to a previous lab? This file sets out, for each frame, the phone state which will be on. Or equivalently, which output of the neural network will be set to 1[2]. This makes sense since we will only expect one phone state to be active at any given time. The second number is a weight, which for our cases will always be set to 1.0. Leave `less` by hitting `q`.

## 2.2 PDF counts

Next, run the following command.

```
ali-to-phones --per-frame=true exp/word/tri2_ali/final.mdl \
  ark:"gunzip -c exp/word/tri2_ali/ali.1.gz |" ark:- | \
  analyze-counts --verbose=1 ark:- -
```

What are these numbers? Recall from the lectures the theory on hybrid acoustic models. We still make use of Hidden Markov Models (HMMs), however, we now replace the GMMs used to estimate output pdfs by the outputs of neural networks. That is, we want to train the neural network to classify phones, and given the output probabilities, we want to compute the likelihood of the state $q$ given the features $\mathbf{x}$, $p(\mathbf{x}|q)$. We can interpret the output of the neural network as the probability of a phone class given the feature $p(q|\mathbf{x})$. Then, using Bayes rule we get *scaled* likelihoods:

$$\frac{p(q|\mathbf{x})}{p(q)} = \frac{p(\mathbf{x}|q)}{p(\mathbf{x})}, \tag{1}$$

where on the left hand side we have divided by the class priors. We don't get $p(\mathbf{x}|q)$ exactly, but this is fine since $p(\mathbf{x})$ does not depend on the class $q$. For more information, see lecture 8.

All this means that we will have to scale the outputs by the class priors, in order to use the neural network outputs in place of the GMMs. The

[2]The rest of the outputs are set to zero. Remember that we are doing frame cross-entropy training.

`train.sh` script computes these from the alignments and stores them in a file called `ali_train_pdf.counts`, using a command similar to the one above. Let's see how this plays out when decoding. Open the corresponding decoding script by running

```
less steps/nnet/decode.sh
```

Search for "counts" by typing a forward slash and then "counts":

```
/counts
```

and hit enter.

We see first that we can provide the counts if we'd like. Hit `n` a couple of times. There are now a couple of lines which look for particular files. Note that we fall back to `ali_train_pdf.counts` if we can't find anything else. This is the file created above. Hit `n` a few times more and you will get to a line that begins with `nnet-forward`. This is a forward pass that will be used to provide the class probabilities for decoding. Notice that the class frame counts are passed in as an argument.

Let's try to look at the output of a forward pass. First, as in the scripts, lets set up a feature stream:

```
# first set up the original features
feats="ark:copy-feats scp:data/train_words/feats.scp ark:- |"

# then splice with 5 context frames on either side
feats="$feats splice-feats --left-context=5 \
  --right-context=5 ark:- ark:- |"
```

Then run a forward pass and look at the output. What happens if you remove the prior counts?

```
nnet-forward \
 --class-frame-counts=exp/word/tri3_nnet/ali_train_pdf.counts \
 exp/word/tri3_nnet/final.nnet "$feats" ark,t:- | less
```

# 3 Architecture

There's one last, important piece of the puzzle. When we ran `steps/nnet/train.sh` above we specified some parameters for the network, such as the number of hidden layers, the layer width, etc. What this does is actually to create a prototype file, setting out the overall architecture. Look at it by typing

```
less exp/word/tri3_nnet/nnet.proto
```

Can you work out what each part means? Why is the input dimension 143? Recall that we ran the training script with `--splice 5`. For more interesting architectures, it can sometimes be worth modifying this file directly. It can then be passed to the training script using `--nnet-proto my.proto`. Let's go ahead and modify the prototype. First, copy the file:

```
cp exp/word/tri3_nnet/nnet.proto my.proto
```

Now open the file in a text editor, such as `nano`, `vim`, `emacs` or `gedit`, e.g.:

```
nano my.proto
```

Change it so that the input layer takes 13 units and that the first layer has 5 neurons. Finally, let's generate an initial model with all the required parameters:

```
nnet-initialize --binary=false my.proto my.init
```

Hopefully, that was successful. In which case, you can look at the model by typing

```
nnet-info my.init
```

Then, go ahead go ahead and open the file itself to see it in full detail:

```
less my.init
```

Can you relate the parameters you set to what you observe in the model file?

That's it! We've already decoded the model for you, using `steps/nnet/decode.sh`, which you can check by running

```
more exp/word/tri3_nnet/decode_test/scoring_kaldi/best_wer
```

Thanks for following along. We hope you enjoy the rest of the course.

## 3.1 Appendix: Common errors

- Forgot to source `path.sh`, check current path with `echo $PATH`
- No space left on disk: check `df -h`
- No memory left: check `top` or `htop`
- Lost permissions reading or writing from/to AFS: run `kinit && aklog`. To avoid this, run long jobs with the `longjob` command.

- Syntax error: check syntax of a Bash script without running it using `bash -n scriptname`
- Avoid spaces after \when splitting Bash commands over multiple lines
- Optional params:
- command line utilities: `--param=value`
- shell scripts: `--param value`
- Most file paths are absolute: make sure to update the paths if moving data directories
- Search the forums: `http://kaldi-asr.org/forums.html`
- Search the old forums: `https://sourceforge.net/p/kaldi/discussion`

### 3.2 Appendix: UNIX

- `cd dir` - change directory to `dir`, or the enclosing directory by `..`
- `cd -` - change to previous directory
- `ls -l` - see directory contents
- `less script.sh` - view the contents of `script.sh`
- `head -l` and `tail -l` - show first or last `l` lines of a file
- `grep text file` - search for `text` in `file`
- `wc -l file` - compute number of lines in `file`

## References

[1] Mark Gales and Steve Young. The application of hidden markov models in speech recognition. *Foundations and trends in signal processing*, 1(3):195–304, 2008.