MPhil Machine Learning and Machine Intelligence

Module MLMI2: Speech Recognition

TIMIT Speech Recognition

Report Due: 12 noon, Tuesday 18th January 2022; Online Submission

1 Introduction

This practical is concerned with phone-based continuous speech recognition using Hidden Markov Models (HMMs). The first part of the practical uses Gaussian and Gaussian mixture models (GMM-HMMs) to form the state output distributions, and the second part investigates the use of Artificial Neural Networks (ANNs) in a hybrid configuration. The experiment investigates both feed-forward deep neural networks (DNN-HMMs) and recurrent networks (RNN-HMMs). The aim is to develop speaker independent phone recognition systems for the TIMIT Acoustic-Phonetic speech database based on these types of models.

The practical involves training and testing HMMs for the TIMIT corpus and examining various configurations for front-end features including differential coefficients; the use of context independent and context dependent models; and allows the investigation of a number of extensions including the use of phone-level language models instead of a simple phone-loop grammar in recognition. Hence, the practical aims to allow you to experiment with some of the ideas presented in the lectures on speech recognition. You will find it helpful to refer to your notes when doing the practical.

All training and testing uses the HTK HMM toolkit. A set of scripts, written either in C-shell or Python, are provided to help run the experiments. These allow you to easily build and test certain configurations of models, and you are able to alter the setup as necessary if you wish to experiment further.

The remainder of this document is organised as follows. Section 2 outlines the theory of speech recognition using HMMs. Section 3 gives a brief introduction to the HTK Hidden Markov Model Toolkit and explains how to use the main HTK tools for training and testing GMM-HMMs and the use of ANN-HMMs in HTK is described in Section 3.1. Section 4 describes the supplied scripts and infrastructure to help use the HTK tools. Section 5 describes the detailed procedure for the first part of the practical on GMM-HMMs, section 6 describes the procedure for the part of the practical on both feed-forward DNN-HMMs, and RNN-HMMs. Finally, section 7 describes some possible extensions. The practical report should contain the experimental results and a discussion/conclusions for the various types of model investigated. Section 8 gives some information about what is required for the practical writeup.

2 Speech Recognition using HMMs

An HMM consists of a set of states connected by arcs (see Figure 1). Each arc from state i to state j has an associated transition probability a_{ij}. Each internal state j has an associated output probability density function $b_j(\mathbf{o})$ denoting the probability density of generating speech vector $\mathbf{o}$ in state j. The entry and exit states do not generate vectors and are called non-emitting (or confluent) states.

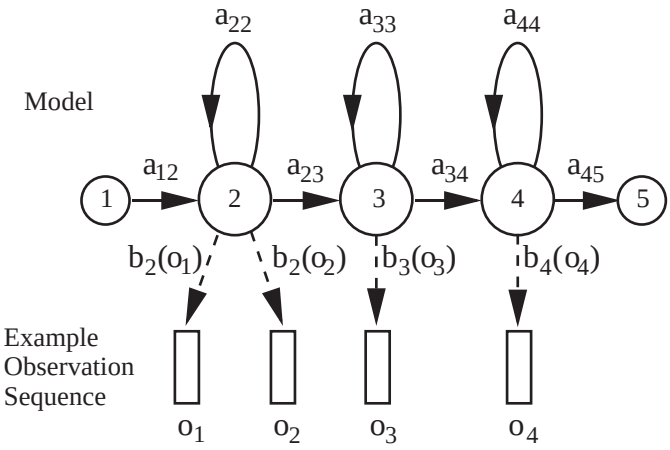

Figure 1: Basic HMM Structure

Each HMM is a simple statistical model of speech production. It is assumed that the HMM makes one state transition per time period and each time it enters a new internal state, it randomly generates a speech vector. Thus, in Figure 1, the HMM has been in states $S = \{1, 2, 2, 3, 4, 5\}$ and has generated vectors $\mathbf{O} = \{\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_3, \mathbf{o}_4\}$. The likelihood of generating this sequence is

$$p(\mathbf{O}, S) = a_{12}\ b_2(\mathbf{o}_1)\ a_{22}\ b_2(\mathbf{o}_2)\ a_{23}\ b_3(\mathbf{o}_3)\ a_{34}\ b_4(\mathbf{o}_4)\ a_{45} \tag{1}$$

The output probability density functions are represented by probability density functions. If, for example, a diagonal covariance multivariate Gaussian distribution is associated with each state, and if vector $\mathbf{o}$ contains components o_1 to o_D then

$$b_j(\mathbf{o}) = \prod_{d=1}^{D} \frac{1}{\sqrt{2\pi\sigma_{jd}^2}} \exp\left(-\frac{(o_d - \mu_{jd})^2}{2\sigma_{jd}^2}\right) \tag{2}$$

In this equation, μ_{jd} and σ_{jd}^2 are the mean and variance of the d'th component of the speech vectors generated by the j'th state. Thus, the vector of means associated with a given state can be regarded as the *typical* speech vector modelled by that state. Notice in (2) that the individual components of $\mathbf{o}$ are assumed to be statistically independent, so that a *diagonal covariance matrix* is used. In practice, the single Gaussian distribution given by equation 2 cannot accurately model the distribution of real speech. Hence, a mixture of Gaussians (to form a GMM-HMM) is often used where the probability density of a vector is computed as the weighted sum of single Gaussians, or another approach is to replace the Gaussian output distributions by a neural network.

Hence, the next parts of this practical is concerned with phone-based continuous speech recognition using Hidden Markov Models (HMMs) with state output distributions computed by Artificial Neural Networks (ANNs). Initially. the ANNs used here are feed-forward Multi-Layer Perceptrons (MLPs), with either a single hidden layer or with several hidden layers: a Deep Neural Network (DNN). Then the use of simple recurrent neural networks (RNN-HMMs) is investigated. The term ANN will be used to refer to either DNNs or RNNs. The direct use of ANN-derived state distributions[1] is known as a "hybrid" ANN-HMM configuration.

In a basic HMM-based speech recogniser, a single HMM is used to represent each linguistic unit (word, syllable, phoneme etc.). Recognition of a single isolated unit then depends on finding, for each HMM, the likelihood of that HMM generating the unknown sequence of speech vectors while following the most likely state sequence. The HMM with the highest such likelihood then identifies the unit. In principle, the required likelihood could be found by trying every possible state sequence and using a formula of the form of (1). However, this would be hopelessly inefficient and fortunately there is a better way.

Let $\Phi_j(t)$ represent the likelihood of the model generating some speech vectors $\mathbf{o}_1$ to $\mathbf{o}_t$ of the unknown utterance such that the state sequence used to obtain this likelihood starts at state 1 and ends in state j, while moving along the most likely state sequence to that point. If $\Phi_j(t-1)$ is known for all j, then $\Phi_j(t)$ can be found for $1 < j < N$ and $1 \leq t \leq T$ by the recursion

$$\Phi_j(t) = \max_{1 \leq i < N} \{\Phi_i(t-1) a_{ij}\} \, b_j(\mathbf{o}_t) \tag{3}$$

where $\Phi_1(0) = 1$, $\Phi_1(t) = 0$, $\Phi_i(0) = 0 \; \forall i \neq 1$ and T is the total number of frames in the unknown speech. After evaluating (3) for frames 1 through to T, the required maximum value can be computed by

$$\Phi_{max} = \max_{1 < i < N} \{\Phi_i(T) a_{iN}\} \tag{4}$$

The above procedure is known as the Viterbi algorithm. In practice the long sequence of products implied by (3) would quickly lead to arithmetic under-flow, hence log values are used so that all multiplications are replaced by additions. Let $S_j(t) = \log(\Phi_j(t))$ then (3) and (4) above may be rewritten as

$$S_j(t) = \max_{1 \leq i < N} \{S_i(t-1) + \log(a_{ij})\} + \log(b_j(\mathbf{o}_t)) \tag{5}$$

and

$$S_{max} = \max_{1 < i < N} \{S_i(T) + \log(a_{iN})\} \tag{6}$$

Equations (5) and (6) provide the basis of all HMM-based speech recognisers.
Given the structure of the basic HMM shown in Figure 1, it should be clear that recognition of continuous speech can be achieved by simply connecting all of the HMMs together into a loop (with appropriate inter-model transition probabilities) as shown in Figure 2. However, the direct extension of equations (5) and (6) to deal with this case becomes complicated due to the need to determine not just the maximum likelihood but also the sequence of models which gives that maximum. Note that the required sequence can be found by storing suitable information as paths are extended and by performing a backtrace to find the route taken by the best path at the end of the utterance.

[1] The ANN state posteriors are converted to scaled likelihoods before being used in the ANN-HMM.

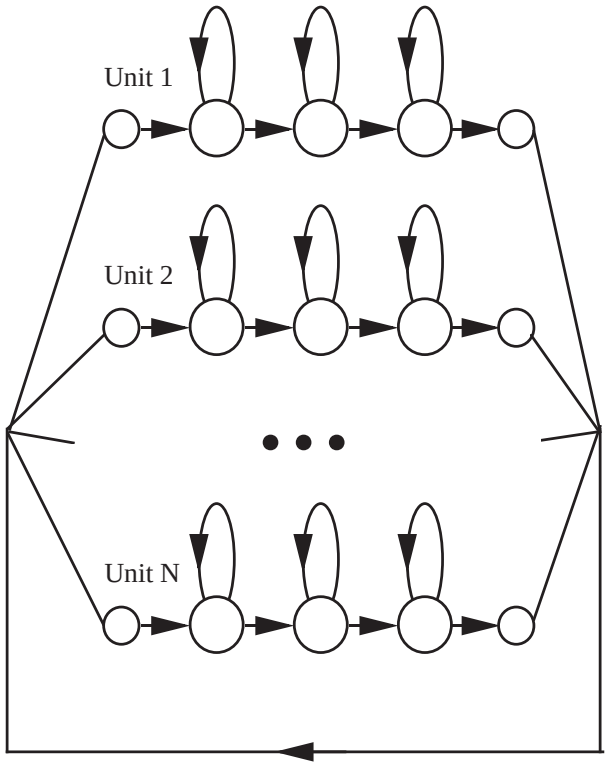

Figure 2: Combining HMMs for Connected Unit Recognition

A key advantage of using HMMs in speech recognition is that the parameters of HMMs (such as Gaussian means, covariance matrices and transition probabilities) are efficiently estimated from real speech data. The parameters are adjusted in an iterative fashion to maximise the likelihood that the models generated the observed speech data.[2] In this practical HMMs of various types will be trained using HTK tools for recognising phone units (rather than words) in the TIMIT database.

One issue with using phones as the basic modelling unit for speech recognition is the variability in acoustic realisation due to *co-articulation*. Co-articulation can be modelled by using Context-Dependent (CD) phone models in which the actual HMM used for a particular phone depends on e.g. the immediate phonetic context. If the CD phone HMM to be used depends on the immediately preceding phone and the immediately following phone, it is known as a *triphone*. In HTK notation, `l-c+r` denotes a phone `c` with a left context of `l` and a right context of `r`. It is also possible to have HMMs that are dependent on just the previous phone (a *left biphone*), or just on the following phone (a *right biphone*). Phone HMMs that don't depend on the surrounding context are termed Context Independent (CI) models or monophones.

The key issue with context dependent models is the number of models to be estimated, and techniques are required so that robust estimates can be formed for all model parameters. A widely used solution, described in lectures and in the HTK book, is to use *decision-tree state tying*. This clusters contexts to form equivalence classes between triphone contexts that are in different triphones of the same *base phone* for a particular phone state position. The decision trees make use of a set of pre-defined

[2]For ANN-HMMs, typically cross-entropy training is performed of the ANN models with supplied state-frame alignments.

phonetic questions that examine groups of contexts. The actual questions asked at any point tree are automatically chosen as the tree is grown to optimise an approximate training data likelihood criterion while also ensuring that there is sufficient data available to model the group of contexts at each tree leaf node.

A major advantage of the decision-tree approach is that after the tree structure is formed, an appropriate triphone model (state) can be selected for any context, even those that don't appear in the training data. The practical investigates the use of both context-independent and context dependent phone models using the decision tree state tying method for both GMM-HMMs and ANN-HMMs. A suitable set of phonetic questions are supplied for use for the practical.

3 The HTK Toolkit

The HTK Toolkit consists of two main parts: a library of useful routines for accessing speech files, label files, HMM definitions etc; and a set of tools (i.e programs) for building and testing HMM recognisers[3].

In this practical, the HTK tools are used to edit label files, train HMMs and analyse the performance of the speech recogniser output. Descriptions of the tools referred to in this section are given in the HTK Book. For registered users, the HTK Book can be accessed from the HTK website. However, there is also an on-line PDF version of the HTK Book for HTK 3.5[4] available on the Moodle pages and also at

```
/usr/local/teach/MLMI2/doc/htkbook35alpha.pdf
```

HTK Tools manipulate several types of files including label files, speech data files and HMM definition files. All tools are invoked from the command line by typing the name of the tool followed by options and arguments. Typing just the name of the tool returns brief usage information (for options and arguments) for all HTK tools. Additional control of HTK tools is provided by setting parameters in a configuration file. For example if the file `config` contained the following

```
SOURCEFORMAT = HTK
```

then typing the command

```
HTool -C config ...
```

would cause HTool to set the variable `SOURCEFORMAT` to the string `HTK`.

There are a number of other useful options to HTK tools (options that are used by many tools are denoted with capital letters) and these include:

- `-A` causes the tool to echo the command line which is useful in saving program output
- `-D` causes the tool to echo the contents of the configuration file at the start and end of execution (which shows which configuration variables have been used)
- `-S fname` A list of files to be processed is stored in `fname`

[3] General information about HTK can be found at the HTK website `http://htk.eng.cam.ac.uk`

[4] Note this version is in places incomplete and may contain errors. Bug reports for both the book and the source code are much appreciated!

- `-T number` sets the trace level to the `number` given. Tracing can also be set at various levels for individual library modules using configuration variables.

Raw waveform speech data is converted to parameterised versions. In this practical both mel filter-bank (`FBANK`) features and mel frequency cepstral coefficients (`MFCC`) have been pre-coded for use. The HTK tool `HCopy` is able to copy files and change the parameter encoding.

Label files are text-based and in their simplest form consist of a sequence of lines in the form

```
<start-time> <end-time> <label>
```

where `<start-time>` and `<end-time>` denote the start and end times of `<label>` in 100ns units. The HTK tool `HLEd` can be made to transform a label file by writing a series of simple edit commands.[5]

For example, suppose the file `edt` contains the following

```
RE C p k t d g b
RE V iy eh ae uh
```

then the command

```
HLEd -i bclabs.mlf -l '*' -n broad.lst edt a b c ...
```

would process each label file `a b c` ... and first sort all labels into time order, then replace all occurrences of p, k, t, d, g, and b by C and finally replace all occurrences of iy, eh, ae and uh by V. The new edited files are stored in a single composite label file called a *master label file* (MLF). The `-l` option makes the names of the label files embedded in `bclabs.mlf` independent of the absolute location of the speech data files that they refer to. In addition to the output MLF file, the file `broad.lst` is created containing a list of all labels present in the output MLF. Transformations of label files to reduce the number of phone classes present have already been applied to the TIMIT-derived labels that you will be using in this practical.

For reference, the full set of edit commands is given in the HTK Book. Note that when processing TIMIT label files, `HLEd` must be told that its input files are not in standard HTK format via the `-G` switch. Also, it is convenient to list all of the label file names to be edited and store them in a single so-called *script* file referenced by the `-S` option. Thus, in practice, `HLEd` would be invoked as

```
HLEd -G TIMIT -i bclabs.mlf -l '*' -n broad.lst -S labs.scp edt
```

where `labs.scp` is the name of script file, containing the list of files to process (a b c ... above), each on a new line. Note that filenames in a script file are not subject to filename expansion[6] and full path names are typically used.

HTK includes several different training tools for GMM-HMMs. Unusually for continuous speech databases, TIMIT includes manual phone segmentations (start/end times for each phone), and so the tools `HInit` and `HRest` (described below) can be used initially followed by `HERest` which only uses model sequence information (& not timings). It is also possible to just use `HERest` directly (a "flat-start") without first making models using `HInit` and `HRest`, and this method is frequently used for large data-sets without phone-level alignment, and will be evaluated as an option in this practical.

[5] The full list of HLEd commands is displayed by `HLEd -Q`, and a full explanation is in the HTK Book.
[6] Filename expansion is performed for command line arguments by the shell.

For TIMIT, single Gaussian HMMs can be created initially using `HInit` which takes as input a *prototype HMM definition* and a set of training data files and produces as output a new initialised HMM. For example, the following would create a new HMM called `C` in directory `hmm0` from prototype HMM `proto` by scanning all of the supplied data files and extracting all segments of speech labelled by `C`[7].

```
HInit -l C -M hmm0 -o C -I trainlabs.mlf -S train.scp proto
```

where `train.scp` holds a list of all the training files.

Once the initial single Gaussian HMMs have been built with `HInit`, the HTK tool `HRest` can be used to re-estimate their parameters using the Baum-Welch algorithm between the given model boundaries

```
HRest -l C -M hmm1 -I trainlabs.mlf -S train.scp hmm0/C
```

which would create a new version of `C` in directory `hmm1`. This improves the training set likelihood and sometimes improves recognition results. Both `HInit` and `HRest` use fixed phone boundaries, as specified by the label file, but whereas `HInit` uses Viterbi alignment during re-estimation, `HRest` uses Baum-Welch training for each phone model.

The next stage of HMM training is to avoid using the hand-produced label boundaries by a technique known as embedded training. This is done by `HERest` which simultaneously re-estimates the parameters of a full set of HMM definitions. For example, the command

```
HERest -d hmm1 -M hmm2 -I trainlabs.mlf -S train.scp hmm.lst
```

will re-estimate all the models listed in `hmm.lst` and stored in `hmm1` and output the new models to `hmm2`. Note that whereas `HRest` repeats the re-estimation procedure repeatedly until convergence is reached, `HERest` performs just one iteration of re-estimation. By default `HERest` merges all output HMMs into a single file called a master macro file (MMF). This can be prevented by setting the configuration variable `KEEPDISTINCT` to true[8].

Note that if a flat-start is to be performed, followed by repeated applications of `HERest` a suitable initial model with the same values for all Gaussian means and variance vectors[9] should be used and suitable initial models can be found using the tool `HCompV`.

HMMs can be further manipulated by using the HMM editor `HHEd`. This allows many functions, which include a iterative mixture splitting operation (`MU` command) , and decision tree state clustering via the `TB` command. HMMs are normally stored in MMFs and these contain both standard HMM definitions introduced by `~h` and various other *macro types* which facilitate sharing e.g. `~t` allows the definition of a transition matrix macro and `~s` allows the definition of a state level macro. A full description of the HTK HMM definition language is given in the HTK book, along with the commands available in `HHEd`.

The HMMs created by the above tools should be tested by using them to recognise the test sentences using the HTK tool `HVite`. `HVite` is a Viterbi decoder for word-based speech recognition that uses a supplied word network and dictionary and can support a number of HMM model types including context-dependent acoustic models up to triphones. HVite supports both GMM-HMMs and ANN-HMMs. It can also support generating word lattices of alternative recognition outputs.

[7] HTK does not automatically generate directories. If you want to directly run this command you will need to generate directory `hmm0`.
[8] Add the line `KEEPDISTINCT=T` to the configuration file `config` and include `-C config` on the command line.
[9] Diagonal covariance matrices are assumed in this practical

HVite is designed for recognising with medium vocabulary sizes and HTK tools can generate input word networks including a bigram language model[10]. HVite is implemented using a "token passing" approach which is described briefly in lectures and more fully in the HTK Book.

To enable HVite to be used in a phone recognition context, each phone symbol is defined as a "word" with the pronunciation of simply itself. The set of possible alternative "words" (i.e. phones) are then placed in a loop that enables the recognition of phone strings. A suitable phone-loop network has been generated for this practical by using the HTK tool HParse. To generate a suitable network for recognition with a bigram language model, the HBuild tool can be used. HVite will automatically expand the underlying model-level network to allow context-dependent models if such acoustic models are supplied.

A typical invocation of HVite might be

```
HVite -H hmm14/MMF -S test.scp -i recog.mlf -w wordnet dict hmmlist
```

where the HMMs to be used are stored in hmm14/MMF, the word-level network is in wordnet, the dictionary which gives the pronunciation of the words used in wordnet is in dict, a list of the available HMMs is in hmmlist, and the recognised label files are stored in the MLF recog.mlf.

Note that the output of HVite is a label file in standard HTK format as described earlier. When all of the test files have been processed, the tool HResults can be used to compare them with the hand-produced labels derived from the TIMIT .phn files. For continuous speech, a speech recogniser can produce substitution errors, deletions and insertion errors. HResults uses a dynamic programming based string alignment procedure to first find the minimum error string alignment and then accumulates statistics on substitutions, deletions and insertion errors.

Assuming that all of the label files output by the recogniser are stored in an MLF recog.mlf and the hand-produced reference labels are in the MLF trainlabs.mlf as before, then the command.

```
HResults -I trainlabs.mlf phonelist recog.mlf
```

will output the percentage number of correct labels in the recogniser output files and the percentage accuracy. Normally when optimising an HMM system the percentage accuracy is maximised (this includes the effect of insertions while % correct does not). Note that often results will be stated as an error rate which is 100% -%accuracy. HResults can also generate a confusion matrix by including the -p option and prints aligned transcriptions of each recognised file by including the -t option.

Finally, the contents of the speech data files (encoded or raw waveform) can be examined using the tool HList (see the HTK Book for details.) Note also that HMM definition files and label files can be examined using the standard UNIX commands cat and more.

3.1 ANNs in HTK 3.5

This section provides a brief introduction to the use of ANNs in HTK 3.5.

The ANN support in HTK 3.5 includes flexible input feature configurations and model architectures. It relies on a universal definition of ANN layer input features. It is designed to be compatible with as many existing functions in HTK as possible. This minimises the effort to reuse previous HTK

[10] In order to used with higher order language models with HVite, lattice rescoring is used

related source code and tools in the new framework and simplifies the transfer of many techniques, for instance, sequence training, and speaker adaptation, from the GMM to the ANN framework.

The HTK macro set for acoustic model definitions is extended to include ANN specific macros, and allows the ANN model definition to be a part of the HTK MMF model files. These can be used by many of the same HTK tools as before, for example, using `HHEd` for model editing and `HVite` for speech recognition and alignment. Detailed descriptions of the ANN related macros and the model format can be found in section 7.3 and 7.7 in HTK book 3.5 alpha.

Name	Function Descriptions/Updates
`HANNet`	ANN structures & core algorithms
`HCUDA`	CUDA based math kernel functions
`HFBLat`	Hybrid MMI/MPE/MWE computation
`HMath`	ANN related math kernel functions
`HModel`	ANN model reading/writing interface
`HNCache`	Data cache for data random access
`HParm`	New feature types for ANNs
`HDecode`	Tandem/hybrid system LVCSR decoder
`HDecode.mod`	Tandem/hybrid system model marking
`HHEd`	ANN model creation & editing
`HVite`	Tandem/hybrid decoder & alignment
`HNForward`	ANN evaluation & output generation
`HNTrainSGD`	SGD based ANN training

Table 1: *List of the ANN related modules and tools in HTK 3.5*

When native ANN support, was added in HTK 3.5, ANN related data structures and functions were developed as a number of new HTK modules (libraries) and tools, and other libraries and tools were extended to include functionality for ANNs. An overview of the HTK modules and tools that support ANNs is shown in Table 1. In particular, the ANN training tool, `HNTrainSGD`, and the model editing tool `HHEd` will be used frequently in this practical, and reference material on their use can be found in the HTK book in sections 17.16 and 17.8, respectively.

A set of math kernel functions required for ANN processing was added in HTK 3.5, and every new kernel has standard CPU, Intel MKL (Math Kernel Library), and GPU based implementations. Usually, using a GPU is faster than using a CPU implementation with the Intel MKL libraries, and both are much faster than standard CPU math kernels.

4 Supplied Infrastructure and Scripts

While HTK is very flexible, in order to allow rapid experimentation, a number of scripts are provided[11], which are written either in C-shell or in Python. The Python scripts make use of the PyHTK library which will be primarily used here in specify ANN training configurations. Additionally, various edit scripts (e.g. `*.hed` for `HHEd`, `*.led` for `HLEd`), a dictionary, word network for `HVite` etc are supplied. These scripts can be easily run to perform the initial stages of the practical. For the extensions you may need to use the HTK tools more directly.

Note that the original TIMIT labels are supplied with 61 phone labels as listed in Appendix B[12]

[11] The supplied scripts are similar to the types of scripts used in building HTK-based systems for research purposes in Cambridge.
[12] The full TIMIT corpus as distributed is available in `/usr/local/data/MLMI2/timit`.

These have been mapped for HMM training to a 48 phone set is often used for modelling purposes. For speech recognition scoring distinctions between only 39 phones are maintained. It may be worth considering if this final mapping is justifiable![13]

The directory `/usr/local/teach/MLMI2` is the top-level directory for the exercise associated with this practical. In turn it contains the following 4 subdirectories:

- `convert`: feature-type specific environment files and libraries (some of which are shared between feature types)
- `data`: the coded TIMIT files
- `doc`: the HTK book and practical handout sheets
- `tools`: both HTK source `htksrc`, binaries `htkbin`, and various scripts (`steps` and `scripts` sub-directories).

For a model build using a particular feature type, an `environment` file is used. There are two main feature types supplied which are sub-directories of the `convert` directory. These are

- `fbk25d` 24 channel `FBANK`
- `mfc13d` 12 MFCCs with normalised log energy (`MFCC_E`)

HTK can append delta coefficients to the input vector on-the-fly, but requires an appropriate prototype HMM definition, and also the value of `TARGETKIND` set appropriately in the HTK configuration file. These transformations of the basic acoustic feature vector are denoted by "qualifiers". In the `env` subdirectory (e.g. `convert/mfc13d/env`) there is an environment file for each expanded base feature type. All of these types include sentence-based mean removal (`_Z`), and may also include first differentials (e.g. `environment_E_D_Z`) and both 1st and 2nd differentials (`environment_E_D_A_Z`). These environment files define some key top-level characteristics that are used by other scripts.

The `lib` sub-directory (e.g. `convert/mfc13d/lib`) contains a number of files such as data-file lists (`flists`), HMM lists (`mlists`), various editing script files including decision tree questions (directory `edfiles`), and in `htefiles` are HTK Tool Environment files (HTE files) which define C-shell variables that are interpreted by the relevant tool-specific scripts etc.

The `tools/steps` are the high-level scripts which often call the basic scripts in `tools/scripts`. These scripts will all print basic usage information if run with no arguments.

Note that the specific files in `tools/steps` that relate to ANN-HMM creation, training and testing are:

- `steps/step-align`: performs state-level alignment from GMM-HMM or ANN-HMM models
- `steps/step-dnntrain`: Python script that performs SGD discriminative pre-training (if necessary) and fine-tuning, both are based on the cross entropy criterion. It is necessary to supply a state-level alignment. Many forms of model can be trained and specified in the appropriate `.ini` file which is used by the PyHTK library.

[13] Here we adopt the widely used practice of mapping phones in TIMIT recognition experiments following the paper "Speaker-Independent Phone Recognition Using Hidden Markov Models" by Kai-Fu Lee and Hsiao-Wuen Hon, 1988.

- `steps/step-decode`: runs recognition using a model set for GMM-HMMs, or ANN-HMMs

A description of the supplied files and infrastructure for this practical is listed in Appendix A. Examples of how it is used are given below in Sec 5 for the GMM-HMM part of the practical, and in Sec 6 for the ANN-HMM parts.

5 Practical Details for GMM-HMMs

5.1 Timescale

The first part of the practical on GMM-HMMs is scheduled to be done over approximately the first two weeks. Initially you should try and understand the infrastructure setup and tools and the HTK commands used as well as evaluating basic systems. Note that the extension areas are discussed later, one of which on language models can be applied to both GMM-HMMs and ANN-HMMs.

Note that you should make sure you understand the information required for the practical report as you go, so that all of the required experiments and analysis have been performed.

5.2 Initial Setup

You will work in a terminal window on one of the the `mlsalt-cpu` machines: `mlsalt-cpu1` and `mlsalt-cpu2` both have 24 physical CPU cores, and `mlsalt-cpu3` has 28 physical cores. While it is possible for one user to run multiple processes in parallel, please be thoughtful of your fellow students!

To start the practical first create a suitable working directory and form symbolic links to the `convert, data, tools` directories, e.g.

```
mkdir -p MLMI2/pracgmm
ln -s /usr/local/teach/MLMI2/{convert,data,doc,tools} MLMI2/pracgmm
cd MLMI2/pracgmm; mkdir exp; cd exp
```

The `exp` directory will be used to build and test various HMM systems. For example, the `step-mono` command can build a context-independent Gaussian mixture system with any of the parameterisations discussed above. It supports initialisation via `HInit` and `HRest` followed by `HERest` or a flat-start.

Assuming that you are in the `exp` directory created above you can use with the most basic feature vector `FBANK_Z` and HInit based initialisation:

```
../tools/steps/step-mono -NUMMIXES 8 ../convert/fbk25d/env/environment_Z \
                                        FBK_Z_Init/mono
```

This command performs a sequence of steps in the `FBK_Z_Init/mono` directory with the appropriate environment file (defines a `FBANK` parameterisation with no differentials) and first estimates models using `HInit` in the `hmm0` sub-directory, with `HRest` in the `hmm1` sub-directory. This initialised model is used as the basis for `HERest` training starting in the `hmm10` directory.

The naming convention usually used (after initialisation steps) is that the first digits after `hmm` denote the number of Gaussians and the final digit denotes the number of `HERest` iterations. After

4 iterations of HERest the script increases the number of Gaussian components to 2 and performs 4 more iterations of HERest to form the MMF in hmm24. This procedure is repeated until the target number of mixture components specified by -NUMMIXES is reached.

Note that each step of training contains various output information files that give the information that would be displayed by the HTK tools (often called LOG files). Furthermore a subdirectory of exp contains a CMDs subdirectory. Within CMDs, a sub-directory for all experimental runs is created and the step-* commands keep a record as to how they were invoked. This is useful to check how experiments were run.

Once this model has been trained (it will take a couple of minutes), it can be tested

```
../tools/steps/step-decode $PWD/FBK_Z_Init/mono hmm84 FBK_Z_Init/decode-mono-hmm84
```

The first argument needs to have an full absolute path , the second argument the is the HMM sub-directory to test and the final argument the directory in which to run the test. The performance of this model (i.e. the output of HResults can be found in FBK_Z_Init/decode-hmm84/test/LOG.

There are a number of options to step-decode. These include decoding just the TIMIT "core test-set" (with -CORETEST); testing on a sub-set of the training set using the -SUBTRAIN option; over-riding the setting of an insertion penalty in the HTE file (with -INSWORD); and over-riding the grammar scale factor in the HTE file with (-GRAMMARSCALE). For a simple word/phone word loop only an insertion penalty is used. For an N-gram, a grammar scale factor (and possibly an insertion penalty) is used. In HTK, a negative insertion penalty and a larger grammar scale factor are used to reduce insertions. It is useful to **roughly** set an appropriate insertion penalty (or grammar scale if a bigram is used). However this should only be changed between major set-up changes. This can be done approximately to maximise the test-set accuracy and can use the much smaller core test set. It is not expected that extensive tuning of these values will be performed.

The LOG file produced by step-decode includes the output of HResults. **It is important for the results to be comparable to ensure that all test set sentences have been decoded** (no output happens when HVite gives a no token survived warning). To ensure that the results are comparable, check that the correct number of sentences are listed in the HResults output (the N item in the SENT line). The full number of senetences in each test set are

(a) Subtrain set: 1,168

(b) Full test set: 1,344

(c) Core test set: 192

If any decoding run produces a number of sentences smaller than the numbers given above, normally the -BEAMWIDTH option in step-decode (this corresponds to the -t option to HVite) should be increased to ensure that all sentences are properly decoded.

Note that it is possible to test performance at any stage in HMM training, including the HInit and HRest stages.

Once you have been through this procedure once, you should ensure that you can understand the steps that the various scripts perform.

5.3 Flat Start

The `step-mono` script can also perform a flat-start:

```
../tools/steps/step-mono -FLATSTART -NUMMIXES 8 ../convert/fbk25d/env/environment_Z \
                                                    FBK_Z_FlatStart/mono
```

Test the performance of this setup and compare with that obtained from the `HInit` / `HRest` initialisation. Comment on the results and choose one initialisation method for future experimentation.

5.4 Other Front-End Parameterisations

You should also generate HMM systems for the various other front-end parameterisations and compare adding the `_D` and `_D_A` for the filter bank parameterisations and also build MFCC monophone models with the various delta parameters.

Compare the phone error rates of all of these models. What trends can be observed? How does the performance change with the addition of differential coefficients and as the number of Gaussians is increased?

5.5 Triphone Decision Tree State Tying

The `step-xwtri` script will build unclustered and then clustered "cross-word" (i.e. cross-phone!) triphones. This should be done on a well-performing front-end parameterisation. Assume that an appropriate monophone has been built in directory `MFC_E_D_A_Z_FlatStart` then

```
../tools/steps/step-xwtri -NUMMIXES 8 -ROVAL 200 -TBVAL 800 \
            $PWD/MFC_E_D_A_Z_FlatStart/mono hmm14 MFC_E_D_A_Z_FlatStart/xwtri
```

will take the initial monophone model specified and then create unclustered triphones, and then cluster them using the `-ROVAL` and `-TBVAL` specified and finally create a state clustered triphone system with the number of Gaussian mixture components `-NUMMIXES` specified. `-ROVAL` and `-TBVAL` set the thresholds for outlier states removal, `RO`, and minimum log-likelihood increase in tree growth, `TB`, which can be found in detail in the HTKBook. The `LOG` file from clustering is in `xwtri/hmm10/LOG`. It shows how many clustered states are present.

The models produced can again be tested using `step-decode`.

The clustering thresholds can be used to control the final number of clustered states. Consider how the number of clustered states should be set given a planned number of Gaussian mixture components per state.

6 Practical Details for ANN-HMMs

This part of the practical involves training and testing ANN-HMMs for TIMIT and examining various configurations of ANNs and features including different amounts of acoustic context; varying the number of hidden layers used; and comparing the use of context independent and context dependent output targets, as well as the use of a simple recurrent model. All ANN-HMM training uses the frame-based cross-entropy objective function. The practical aims to allow you to experiment with some of the ideas presented in the lectures on speech recognition and you may find it helpful to refer to your notes when doing the practical.

All training and testing uses the HTK HMM toolkit[14] A set of scripts are provided to help run the experiments. These allow you to easily build and test certain configurations of models, and you are able to alter the setup as necessary if you wish to experiment further.

6.1 Timescale

This part of the practical is expected to be done starting in the fifth week of Michaelmas Full Term.

Initially you should try and understand the infrastructure setup and new scripts supplied for ANN-HMM training and testing. Then you can perform the basic stages suggested in the practical procedure.

Note that you should make sure you understand the information required for the practical report as you go, so that all of the required experiments and analysis have been performed.

6.2 Initial Setup and Example Run

You will again work in a terminal window on one of the `mlsalt-cpu` machines. Machines `mlsalt-cpu1` and `mlsalt-cpu2` have a single NVIDIA K40 GPU card in addition to the 24 physical CPU cores in each machine[15]. Note that `mlsalt-cpu3` does not have a GPU card. While the GPU cards will support multiple users in parallel, since at present they are not accessed via a queueing system, the GPU can easily be overloaded and care will need to be taken to manage the loading of the GPUs. Therefore individual users should submit no more than one process at a time that uses the GPU. All practical steps can also be performed on the CPUs, but when unloaded the GPU is considerably faster.

To start the practical first create a suitable working directory and form symbolic links to the `convert, data, tools` directories for this part of the practical. Assuming the `MLMI2` directory already exists then e.g.

```
mkdir  MLMI2/pracann
ln -s /usr/local/teach/MLMI2/{convert,data,doc,tools} MLMI2/pracann
cd MLMI2/pracann; mkdir exp; cd exp
```

The `exp` directory will be used to build and test various ANN-HMM systems. It is also assumed that a directory `MLMI2/pracgmm` already exists and that the directory `MLMI2/pracgmm/exp` contains

[14] This version of HTK is a current development version which will lead to HTK 3.5.1: this should **NOT** be used for any purpose beyond this practical

[15] NVIDIA kindly donated these K40 cards for use on the MPhil course.

context-independent and context-dependent GMM-HMM model builds. If you have used a different naming convention for the GMM-HMM part of the practical then you will have to modify the steps below.

To train an ANN-HMM system it is necessary to supply a frame-level alignment of the training data with the target state-level labels. An appropriate alignment will be needed for both context independent and context dependent systems, but first we will start with a context-independent system. For instance, we might use:

```
cd ~/MLMI2/pracgmm/exp
../tools/steps/step-align $PWD/MFC_E_D_A_Z_FlatStart/mono hmm84 \
                                          MFC_E_D_A_Z_FlatStart/align-mono-hmm84
```

For ANN training, directories with a reduced length name for brevity will be used. As an example, the build process for an MFCC based hybrid DNN-HMM model in a sub-directory `MH0` of the `pracann/exp` directory is described below.

To refer to the directory containing the alignments via a shell variable `pracgmmexp`, first set

```
pracgmmexp=~/MLMI2/pracgmm/exp/MFC_E_D_A_Z_FlatStart
```

and then

```
cd ~/MLMI2/pracann/exp
../tools/steps/step-dnntrain -vvv ../convert/mfc13d/env/environment_E_D_A_Z \
     $pracgmmexp/align-mono-hmm84/align/timit_train.mlf $pracgmmexp/mono/hmm84/MMF \
     $pracgmmexp/mono/hmms.mlist MH0/dnntrain
```

Note that alignments from MFCC-based GMM-HMMs can be used for both `MFCC` and `FBANK` parameterisations.

The above command specifies the most detailed level of trace information to be printed (via `-vvv`) which is useful for debugging. Note that with either `-v` only warnings are printed and with `-vv` general information about progress.

Note that if the K40 GPU is to be used, it is necessary to supply the option `-GPUID 0` to `step-dnntrain`. The GPU should be used for training where possible when using `mlsalt-cpu1` and `mlsalt-cpu2`. The status of the GPU can be found with the command `nvidia-smi`.

The `step-dnntrain` command will create a complete DNN system with the supplied alignments based on the initial GMM-HMM MMF specified. The MMF along with the associated model-list is used to determine the names set of models to be trained, and their structure.

After the example `step-dnntrain` has been run (or while it is running) take a look through the directory that has been created in `MH0/dnntrain`. As this is a ReLU system, there are no pre-training stages. The final structure with 5 hidden layers is created in the `MH0/dnntrain/hmm0` directory and epochs of SGD training are performed and the final DNN model will be `MH0/dnntrain/hmm0/MMF` There is a `LOG` file in `MH0/dnntrain/hmm0` that you should examine. This gives the training set and cross-validation (CV) set frame level accuracies. Note that the CV set is a 5% randomly selected portion of the training set. You should examine these accuracies as they are a useful guide to how well the model has trained.

The above has trained a fairly complex model, and gives an example of how the various steps are performed and uses a default ANN configuration. To change the default `step-dnntrain` takes a `-MODELINI` option which is used to specify an alternative ANN configuration. Some example `.ini` files are given in the `tools/htefiles` directory. Information on the format of .ini files is in A.5. The default `.ini` is a 7 layer (5 hidden layer) ReLU activation function network which is in the file (exact name depends on feature type) `tools/htefiles/DNN-7L.ReLU.*.ini`. Using one of the other pre-supplied files, or modifying those supplied will allow you to investigate other options.

Once the model has been trained it can be tested. In this case, the following command would be executed:

```
../tools/steps/step-decode $PWD/MH0/dnntrain hmm0 MH0/decode
```

Recall that the first argument needs to have an full absolute path, the second argument is the HMM sub-directory where to find the MMF to test, and the final argument the directory in which to run the test. The performance of this model (i.e. the output of `HResults`) can be found in `MH0/decode/test/LOG`.

Note that, as in the GMM-HMM part of the practical, it is useful to **roughly** tune the insertion penalty that is used in decoding. It saves time during tuning to only use the core test subset of the test set which is specified by using the `-CORETEST` option to `step-decode`.

Note also that in some cases comparisons of performance on the training set are useful, and this can be done using a sub-set of the training set using the `-SUBTRAIN` option to `step-decode`.

6.3 Single Hidden Layer Experiments

In this section, you should use a single hidden layer model with the context independent targets used above. The same alignment file should be used for all parts of this section.

You will need to make your own copy of `DNN-7L.ReLU.*.ini`, and convert it to have a single layer only. Note that this requires changes to the `[NeuralNet:DNN]` section and in the `[Layer:layerout]` section change the `FeatureElement1.Source` value to `layerin`.

The aim of this section is to investigate jointly the following:

- Context Width
- Feature Type

In each case investigated you will need to alter the `.ini` files as needed. Note that you should choose what you feel are interesting combinations to investigate, but start with no additional acoustic context to help classify the current frame (i.e. just a single frame at the input to the ANN). How much performance difference do differential coefficients make in this case? How does this vary as the context widow is extended?

Determine the relative performance of `MFCC` and `FBANK` features.

In all cases examine both the training/CV frame classification performance and the final phone recognition rate after running `step-decode`.

6.4 Adding Multiple Layers

Choose an input feature parameterisation (possibly with differentials), and a context shift set for further experiments.

In this section you should investigate adding further hidden layers to the DNN structure you are training by modifying the `.ini` file. Investigate the performance of adding layers to the initial single hidden layer structure to determine a suitable depth for the DNN. It is again suggested that you keep to 500 hidden layer nodes in order to keep computation reasonably fast.

As the number of layers is increased, the total number of parameters in the system increases: does this cause issues in parameter estimation?[16]

As before examine both the training/CV frame classification performance and the final phone recognition rate after running `step-decode` on both the test data and the training data subset.

6.5 Triphone Target Units

Once you have determined a suitable configuration, in terms of input features, amount of acoustic context and number of hidden layers, experiment with changing the output layer to be more context dependent.

If the targets change then so must the state-level alignments. Use a triphone model set that was trained in GMM-HMM part of the practical to generate suitable triphone state-level alignments with `step-align`. Use these alignments, along with the corresponding triphone GMM-HMM MMF and model list to train DNN-HMMs with triphone state targets with `step-dnntrain`, and evaluate the recognition performance with `step-decode`.

Note that for a context-dependent output layer, the number of DNN weights again substantially increases: does this cause issues in parameter estimation?

6.6 RNN-HMMs

The final section of the practical has a brief introduction to simple RNN models. The RNN at each frame takes in input of the current frame t and the output of a hidden vector at time $t-1$. The models are trained using truncated back-propagation through time (truncated BPTT) in which the model is *unfolded* for a fixed number of steps to form a feed-forward network. The advantage of unfolding in this way is that training can still use frame-level randomisation which is important for SGD training on speech data.

To investigate RNNs, choose of of the `RNN-1L.ReLU.*.ini` files. You should investigate the effect of

- The amount of unfolding that is applied. Note that the default is 20 steps but what happens if 5 or 10 steps are used instead?
- Comparing the use of context-independent or triphone state targets

[16] In the .ini file, the `WEIGHTDECAY` determines the amount of $L2$ regularisation supplied. The default value should be suitable.

7 Practical Extensions

There are a large number of possible extensions that can be performed. If time permits, you should investigate **one** of the extensions below. If you wish to work an alternative extension then you should discuss it with a demonstrator.

7.1 Extension 1: Bigram Language Model

In the main part of the practical a phone-loop grammar was used. In order to create and test a bigram language model:

- Create a phone-level back-off bigram language model using HLStats (use the `-o` option. Note that the language model is to be built over context-independent phones and not context independent phones (i.e. it should be trained from a context-independent label files) and should use the labels that are used for training (`train.mlf`).
- Build a bigram network with `HBuild` for use with HVite. Note that it is necessary to also add in additional "ENTER" and "EXIT" symbols into the dictionary used and ensure that these output a `sil` model. For instance add the lines:

```
!ENTER [sil]   sil
!EXIT  [sil]   sil
```

- Modify the HTE file used with `step-decode` : note that an alternative HTE file is optionally supplied.
- Roughly optimise the grammar scale factor with the bigram model. It is adequate to set the insertion penalty to zero.

What can you conclude about using the bigram model? Does the performance improvement differ between GMM-HMMs, DNN-HMMs and RNN-HMMs? Does this depend on the acoustic context used at the input to the model and/or the amount of context dependency at the output of the model? Would you expect to get further improvements in performance with a higher-order n-gram model?

7.2 Extension 2: Extended RNN models

There are a large number of possible extensions to the simple RNNs used so far. Two possible extensions to the simple RNN are

- Add more fully connected layers either at the input before the recurrent layer or before the output.
- Add further recurrent layers in addition to the single recurrent layer investigated earlier.

As an alternative it is also possible to investigate an LSTM based model. The definition for a single layer LSTM is in `LSTM-1L.FBANK_D_Z.ini`. This can be compared to a simple RNN as well as using, for example, more recurrent layers.

7.3 Extension 3: Time Delay Neural Networks

The time delay neural network can be used as an alternative acoustic model. The definition is given in `TDNN.ReLU.FBANK_D_Z.ini`. The performance of this model can be investigated, along with various possible modifications to the structure.

8 Write-Up

Your report is expected to be about 4,000 words long. It must be submitted no later than the date given on the front of this practical handout.

It should include **Experimental Findings and Conclusions**. Describe the work done, and the results for the various sections supported directly by the supplied scripts and for any extension attempted. For each different set-up, make concise tables/graphs giving your results. Comment on the results obtained and try to explain the patterns found. You may wish to examine the difference between performance on the training sub-set and test sets for some of the different HMMs. Discuss the key issues of generalisation, modelling accuracy, and the number of parameters estimated. What is the overall "best" system that you have found?

P.C. Woodland & C. Zhang,
October 2021

Acknowledgements:
C. Zhang developed many of the scripts and the infrastructure used for this practical.
The experiment and handout draws on previous practicals developed by M.J.F. Gales, T. Hain, P.C. Woodland & S.J. Young.

A Supplied Tools and Resources

A.1 Data Preparation

The original 61 TIMIT phones were mapped to the CMU 48 phones which are used for building acoustic models and language modelling. These are then further grouped into 39 distinct phones when scoring (by using `HResults` equivalence mappings). [17]

The TIMIT data names were changed to "accentid_speakerid_textid.featuretype", where "accentid", "speakerid", and "textid" are the original TIMIT ids, and "featuretype" is a 3-letter HTK feature format extension, all fields are in lower case.

A.2 System structure

The key components of the TIMIT system are defined in an environment file, which is associated with the input feature type and differentials. The variables listed in Table 2.

Variable	Function
TIMITTOOLS	tools directory with the tools and scripts
FEATYPE	the feature type id for the distinguish of different features
FEADIFF	the differential type used to extend the features
FEADIM	the dimension size of an extended feature vector
TIMITLIB	the library path associated with the feature type

Table 2: TIMIT system environment variables.

The final input feature to the HTK systems are formed by appending `FEADIFF` to `FEATYPE`. `TIMITLIB` contains all the resources for the system building with current `FEATYPE`. Therefore, a different environment is supposed to be used once a different type of input feature is used.

The tools directory defined by `TIMITTOOLS` has the tools and scripts:

1. `steps`: scripts for all steps of system building.
2. `scripts`: functional scripts used by the steps scripts.
3. `htkbin`: HTK binaries.
4. `htksrc`: HTK source code.

When performing basic experiments, the `steps` scripts should be sufficient, and they provide basic HTK system building and evaluation functions. The other scripts are used by the `steps` script functions, and can be reused once the users are exploring more complicated systems. The tools directory can be modified by each user and specified through either `TIMITLIB` variable or `steps` script run time option `-TOOLS`.

[17] Note that in the mapping to the 48 phone set sequences of stop-closure and a following consonant (to mark the release) are mapped to the release only (since the closure always occurs). This means that to correctly score the final output in a form that is comparable to published work that recognised consonants that correspond to a release need to be re-expanded to have both a stop consonant and a release. This expansion uses the `HLEd` script `edfiles/phoneexpand.led` The supplied decoding scripts include this step of phone (re-)expansion.

A.3 Resource files

The resource files in `TIMITLIB` are described in Table 3.

These resource files can be classified into two categories: feature dependent and feature independent. The feature dependent files include the feature file list, HTK config files *etc*. These files would need to be re-created if a new type of feature is included, and a new environment file made.

File	Shared†	Purpose
`cfgs/basic_*.cfg`	false	basic HTK config of the data type
`cfgs/coding.cfg`	false	HTK config for coding with current data type
`cfgs/cvn_*.cfg`	false	`HCompV` config for computing global variance vectors
`cfgs/herest_*.cfg`	false	HTK config for GMM-HMM estimation using `HERest`
`cfgs/viterbi-mono.cfg`	true	monophone system HVite config
`cfgs/viterbi-xwtri.cfg`	true	cross-word triphone system HVite config (logical label)
`cvn/cvn_/*/*`	false	`HCompV` vectors for global variance normalisation
`dicts/phone.dct`	true	`HVite` phone dictionary for decoding and alignment
`edfiles/clone_xwtri.hed`	true	`HHEd` config for cross-word triphone HMM generation
`edfiles/cluster_ROVAL_TBVAL.hed`	true	`HHEd` config template with questions for tree tying
`edfiles/make_xwtri.led`	true	`HLEd` config for MLF cross-word triphone extension
`edfiles/phone.led`	true	`HLEd` rules for TIMIT 61 to 48 phone mapping
`edfiles/phoneexpand.led`	true	`HLEd` rules for re-expanding stop consonants to include closures
`edfiles/tie_alltrans.hed`	true	`HHEd` config to tie HMM transition matrices
`eqsets/phone48_39.eqs`	false	39 phone clustering rules for `HResults` scoring
`flists/work/train.all.hcopy.scp`	false	original train set (with _sa*) `HCopy` file list
`flists/work/test.all.hcopy.scp`	false	original test set (with _sa*) `HCopy` file list
`flists/train.scp`	false	full train set (without _sa*) file list for GMM-HMMs
`flists/train.sub.scp`	false	subset train set file list for testing GMM-HMMs
`flists/test.scp`	false	full test set (without _sa*) file list
`flists/test.core.scp`	false	core test set (without _sa*) file list
`flists/dnn.train.scp`	false	DNN-HMM train set (without _sa*) file list
`flists/dnn.cv.scp`	false	DNN-HMM validation set (without _sa*) file list
`htefiles/HTE.mono`	true	monophone GMM-HMM training variable template
`htefiles/HTE.xwtri`	true	cross-word GMM-HMM training variable template
`htefiles/HTE.phoneloop`	true	variable template for phone loop decoding
`htefiles/*.ini`	true	ANN-HMM configuration definition
`htefiles/HTE.align`	true	variable template for state level alignment
`info/ident_cvn_*`	false	identity vectors as variance normalisation targets
`mlists/mono+sil.mlist`	true	48 phone (including sil) list
`mlists/mono+xwtri.mlist`	true	full cross-word triphone list (no triphone variants of sil)
`wdnets/phoneloop.net`	true	`HVite` phone loop decoding network (48 phones)

Table 3: `FEATYPE` library file descriptions (`convert/FEATYPE/lib/*`). "Shared†" shows whether the files are shared by different `FEATYPE`s.

A.4 Tools and scripts

Details of the HTK tools can be found in the HTKBook. A brief descriptions of each of the scripts in "steps" and "scripts" is given in Table 4.

Script	Function
`steps/step-mono`	building monophone GMM-HMMs
`steps/step-xwtri`	building decision tree based cross-word triphone GMM-HMMs
`steps/step-decode`	monophone/triphone GMM-HMM and ANN-HMM decoding
`steps/step-align`	monophone/triphone GMM-HMM and ANN-HMM alignment
`steps/step-dnntrain`	building ANN-HMMs with different kinds of targets
`scripts/herest`	single pass GMM-HMM training using `HERest`
`scripts/hbuild`	multi-pass GMM-HMM training using `HERest`
`scripts/hedit`	GMM-HMM/ANN-HMM model edit with `HHEd`
`scripts/hvite`	GMM-HMM/ANN-HMM decoding and alignment with `HVite`
`scripts/copydnn.py`	Copy ANN parameters from between ANN structures

Table 4: GMM- and ANN-HMM system scripts (`steps/*`) & the related functional scripts (`scripts/*`).

The HTE file `htefiles/HTE.phoneloop` can be modified when tuning decoding. The particular HTE file can be specified (other than the default file) using -DECODEHTE option in the decoding script. Some important variables in these HTE file are presented in Table 5. A more detailed descriptions of each of the configurable variables are given as comments at the beginning of the scripts in the "scripts" directory. Note that it also possible to over-ride these variable settings in `steps/step-decode` which provides an easier mechanism to tune these parameters.

Variable	HTE File	Function	Option Name
`HVPRUNE`	`HTE.phoneloop`	`HVite -t` option for beam search pruning	`-BEAMWIDTH`
`HVGSCALE`	`HTE.phoneloop`	`HVite -s` option as the grammar scaling factor	`-GRAMMARSCALE`
`HVIMPROB`	`HTE.phoneloop`	`HVite -p` option to penalise the phone insertions	`-INSWORD`

Table 5: Selected HTE file variables for training and decoding.

A.5 PyHTK Model Definitions

The Python script `step-dnntrain.py` (linked to `step-dnntrain`) makes calls to the `PyHTK` library. This in turn uses a configuration language to define the ANN models to be trained along with various parameters for training.

Some notes on `PyHTK` are given below.

(a) PyHTK uses the standard python config file parser for model definition (see python 3.5 online documentation https://docs.python.org/3.5/ for syntax *etc.*). Since in HTK, the config file (`.cfg`) is generally used for setting binary configuration variables, in PyHTK we refer to them as initial files (`.ini`).

(b) Like the standard python config file parser, a PyHTK `.ini` file comprises of multiple sections started with `[type: name]`, and each section name should be unique. Each section can have multiple variables with unique names, and assign each of them information for a model definition. A special type of variable allowed by PyHTK is the macro variable, which starts with symbol "`@`", and the value can be quoted anywhere using the macro name. PyHTK `.ini` file sections are divided into two types at the moment.

- Model definition: Can be used for both GMM-HMM definitions (not discussed further here as not being used by the current scripts) and ANN model definitions. The general input feature type *etc.* are defined in the "`ModelSet`" section, and for ANN models, at least one section of "`NeuralNet`" type is required. For example, `[NeuralNet: LSTM1]` adds the layer based definition of a neural network model named "`LSTM1`" into the model set. Each layer can be defined using a "`Layer`" type section.
- HTK configuration variables: An example of this type of section is the `[FrameLevelTraining]` section, which can have any config variables and those defined will be directly put into the `pretrain.cfg` or `finetune.cfg` files for frame level ANN model training. A special variable associated with this section is the "Initialisation" variable, which defines how this model is initialised. For example, if the value is "Random:HHEd", it will use HHEd to do the initialisation; if its value points to another ini file, then this model will be initialised by that model. This can be carried out recursively to support pre-training in a very flexible way (in terms of both model structure and pre-training configs).

A brief list of the basic `.ini` ANN model sections and the associated variables that are listed in the table below.

Category	Item	Description
Section	[ModelSet]	The general model information (e.g., input feature) definition section.
Variable	InputObservation.Type	The input feature type, e.g., <FBANK_D_Z> section.
Variable	InputObservation.Dim	The size of the input feature vector.
Section	[NeuralNet:*name*]	Contains a list of layer names given by variables named following the rule "Layer*idx*=*name*", where *idx* is the layer index ranging from 2 and *name* is the macro name of the corresponding layer.
Section	[Layer:*name*]	This section defines the layer and is the most complicated type of section. It is first divided into different cases according to the layer kinds, which is set by the variable.
Variable	Kind	Sets layer kind. Values include: FC/DNN (fully connected), LSTM, GRU, Pooling, and CNN.
Variable	FeatureMixture.Num	Set the number of feature elements.
Variable	FeatureMixture.Operation	Set the operation for combining the feature elements, and options include: Cat (default), Add, Mul, and Max.
Variable	FeatureElement*idx*.Dim	*idx* is index of feature element (from 1). This variable sets dimension (before stacking due to context shift) of this element.
Variable	FeatureElement*idx*.ContextShift	Sets integer set defining the way to stack the element.
Variable	FeatureElement*idx*.Source	Specifies source element, e.g., a vector, a layer, input feature etc.
Variable	OutputDim	The output dimension of a layer.
Variable	ActivationFunction	The activation function of the layer.
Variable	Weight.Name	Specifies the matrix name to be used as the weight of the layer.
Variable	HasBias	Specifies whether the layer have a bias vector.
Variable	Bias.Name	Specifies the vector name to be used as the bias of the layer.
Variable	MemoryCell.Weight.Name	This specifies the LSTM layer memory cell matrix name.
Variable	MemoryCell.XForm	Specifies transform for LSTM layer memory cell.
Variable	MemoryCell.HasBias	Specifies if LSTM layer memory cell has a bias vector.
Variable	InputGate.Weight.Name	Gives name of weight matrix of LSTM layer input gate.
Variable	InputGate.HasBias	Specifies whether LSTM layer input gate has a bias vector.
Variable	InputGate.HasPeepHole	Specifies if LSTM layer input input gate is associated with a peep hole vector
Variable	InputGate.Peephole.Name	Specifies vector name of LSTM layer input gate peep hole
Variable	InputGate.XForm	Specifies transform for input gate. Note such LSTM layer input gate variables exist also for "ForgetGate" and "OutputGate".
Variable	InputFeatureMap2D	Set the input feature map of the CNN layer, which is in the form of a 2D tuple i.e., (*val*1, *val*2), where *val*1 and *val*2 are two integers.
Variable	ZeroPadding2D	Set the zero padding dims of the CNN layer, which is also in the form of a 2D tuple.
Variable	Stride2D	Set the CNN layer stride as a 2D tuple.
Variable	FilterKernel2D	Set the CNN layer filter kernel as a 2D tuple. The above four CNN variables are also existed in the pooling layer.
Section	[NMatrix:*name*]	This section defines a NMatrix object that can be quoted/shared by the layers.
Variable	RowDim	Row number of the matrix (the output dimension).
Variable	ColumnDim	Column number of the matrix (the input dimension).
Variable	Values	Set some initial values for the matrix.
Section	[NVector:*name*]	This section defines a NVector object that can be quoted/shared by the layers.
Variable	Length	The length of the vector.
Variable	Values	Set some intial values for the vector.

B Acoustic Phonetic Symbols used in TIMIT

This appendix describes the phonemic and phonetic symbols used in the TIMIT phonetic transcriptions. These include the following symbols:

1. the closure intervals of stops which are distinguished from the stop release. The closure symbols for the stops `b,d,g,p,t,k` are `bcl,dcl,gcl,pcl,tck,kcl`, respectively. The closure portions of `jh` and `ch`, are `dcl` and `tcl`.
2. allophones that do not occur in the lexicon. The use of a given allophone may be dependent on the speaker, dialect, speaking rate, and phonemic context, among other factors. Since the use of these allophones is difficult to predict, they have not been used in the phonemic transcriptions in the lexicon.
   - flap `dx`, such as in words "muddy" or "dirty"
   - nasal flap `nx`, as in "winner"
   - glottal stop `q`, which may be an allophone of `t`, or may mark an initial vowel or a vowel-vowel boundary
   - voiced-h `hv`, a voiced allophone of `h`, typically found intervocalically
   - fronted-u `ux`, allophone of `uw`, typically found in alveolar context
   - devoiced-schwa `ax-h`, very short, devoiced vowel, typically occurring for reduced vowels surrounded by voiceless consonants
3. other symbols include two types of silence; `pau`, marking a pause, and `epi`, denoting epenthetic silence which is often found between a fricative and a semivowel or nasal, as in "slow", and `h#`, used to mark the silence and/or non-speech events found at the beginning and end of the signal.

SYMBOL	POSSIBLE EXAMPLE WORD	PHONETIC TRANSCRIPTION
Stops:		
b	bee	BCL B iy
d	day	DCL D ey
g	gay	GCL G ey
p	pea	PCL P iy
t	tea	TCL T iy
k	key	KCL K iy
dx	muddy, dirty	m ah DX iy, dcl d er DX iy
q	bat	bcl b ae Q
Affricates:		
jh	joke	DCL JH ow kcl k
ch	choke	TCL CH ow kcl k
Fricatives:		
s	sea	S iy
sh	she	SH iy
z	zone	Z ow n
zh	azure	ae ZH er
f	fin	F ih n
th	thin	TH ih n
v	van	V ae n
dh	then	DH e n
Nasals:		
m	mom	M aa M
n	noon	N uw N
ng	sing	s ih NG
em	bottom	b aa tcl t EM
en	button	b ah q EN
eng	washington	w aa sh ENG tcl t ax n
nx	winner	w ih NX axr
Semivowels and Glides:		
l	lay	L ey
r	ray	R ey
w	way	W ey
y	yacht	Y aa tcl t
hh	hay	HH ey
hv	ahead	ax HV eh dcl d
el	bottle	bcl b aa tcl t EL

Vowels:

iy	beet	bcl b IY tcl t
ih	bit	bcl b IH tcl t
eh	bet	bcl b EH tcl t
ey	bait	bcl b EY tcl t
ae	bat	bcl b AE tcl t
aa	bott	bcl b AA tcl t
aw	bout	bcl b AW tcl t
ay	bite	bcl b AY tcl t
ah	but	bcl b AH tcl t
ao	bought	bcl b AO tcl t
oy	boy	bcl b OY
ow	boat	bcl b OW tcl t
uh	book	bcl b UH kcl k
uw	boot	bcl b UW tcl t
ux	toot	tcl t UX tcl t
er	bird	bcl b ER dcl d
ax	about	AX bcl b aw tcl t
ix	debit	dcl d eh bcl b IX tcl t
axr	butter	bcl b ah dx AXR
ax-h	suspect	s AX-H s pcl p eh kcl k tcl t

Others:

SYMBOL	DESCRIPTION
pau	pause
epi	epenthetic silence
h#	begin/end marker (non-speech events)
1	primary stress marker
2	secondary stress marker