Indian Institute of Technology Madras

Tamil Speech Synthesis Using DNNs

BTech. Project Report by:

Kannan V

EE13B031

Department of Electrical Engineering

Abstract

We introduce the use of Merlin speech synthesis toolkit for neural network-based speech synthesis of Tamil- an Indian language. The system takes linguistic features as input, and employs neural networks to predict acoustic features, which are then passed to a vocoder to produce the speech waveform. Various neural network architectures are implemented, including a standard feedforward neural network, mixture density neural network, recurrent neural network (RNN), long short-term memory (LSTM) recurrent neural network, amongst others. The toolkit is Open Source, written in Python, and is extensible.

Index Terms: Speech synthesis, deep learning, neural network, Open Source, toolkit, Tamil synthesis

Introduction

Text-to-speech (TTS) synthesis involves generating a speech waveform, given textual input. Freely-available toolkits are available for two of the most widely used methods: waveform concatenation, and HMM-based statistical parametric speech synthesis, or simply SPSS. Even though the naturalness of good waveform concatenation speech continues to be generally significantly better than that of waveforms generated via SPSS using a vocoder, the advantages of flexibility, control, and small footprint mean that SPSS remains an attractive proposition.

In SPSS, one of the most important factors that limits the naturalness of the synthesized speech is the so-called acoustic model, which learns the relationship between linguistic and acoustic features: this is a complex and non-linear regression problem. For the past decade, hidden Markov models (HMMs) have dominated acoustic modelling. The way that the HMMs are parametrized is critical, and almost universally this entails clustering (or 'tying') groups of models for acoustically- and linguistically-related contexts, using a regression tree. However, the necessary across-context averaging considerably degrades the quality of synthesized speech. One might reasonably say that HMM-based SPSS would be more accurately called regression tree-based SPSS, and then the obvious question to ask is: why not use a more powerful regression model than a tree?

Recently, neural networks have been 'rediscovered' as acoustic models for SPSS. In the 1990s, neural networks had already been used to learn the relationship between linguistic and acoustic features, as duration models to predict segment durations, and to extract linguistic features from raw text input. The main differences between today and the 1990s are: more hidden layers, more training data, more advanced computational resource, more advanced training algorithms, and significant advancements in the various other techniques needed for a complete parametric speech synthesizer: the vocoder, and parameter compensation/enhancement/postfiltering techniques.

Recent Work

In the recent studies, restricted Boltzmann machines (RBMs) were used to replace Gaussian mixture models to model the distribution of acoustic features. The work claims that RBMs can model spectral details, and result in better quality of synthesised speech. Deep belief networks (DBNs) as deep generative model were employed to model the relationship between linguistic and acoustic features jointly. Deep mixture density networks and trajectory real-valued neural autoregressive density estimators were also employed to predict the probability density function over acoustic features.

Deep feedforward neural networks (DNNs) as a deep conditional model are the model popular model in the literature to map linguistic features to acoustic features directly. The DNNs can be viewed as replacement for the decision tree used in the HMM-based speech as detailed in earlier studies. It can also be used to model high-dimensional spectra directly. In the feedforward framework, several techniques, such multitask learning, minimum generation error, have been applied to improve the performance. However, DNNs perform the mapping frame by frame without considering contextual constraints, even though stacked bottleneck features can include some short-term contextual information.

To include contextual constraints, a bidirectional long short-term memory (LSTM) based recurrent neural network (RNN) was employed to formulate TTS as a sequence to sequence mapping problem, that is to map a sequence of linguistic features to the corresponding sequence of acoustic features. LSTM with a recurrent output layer was proposed to include contextual constraints. LSTM and gated recurrent unit (GRU) based RNNs are combined with mixture density model to predict a sequence of

probability density functions. A systematic analysis of LSTM-based RNN was presented to provide a better understanding of LSTM.

Merlin

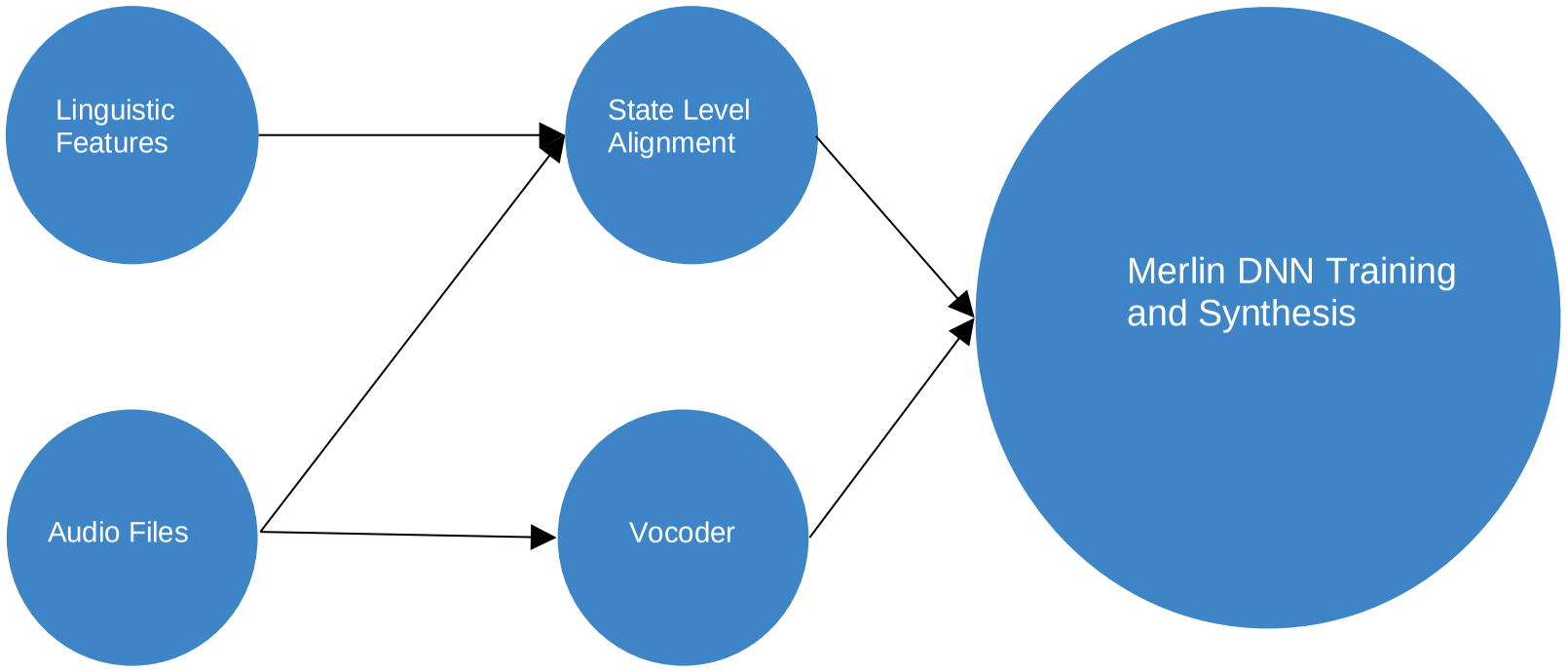
Recently, even though there has been an explosion in the use of neural networks for speech synthesis, a truly Open Source toolkit is missing. Such a toolkit would underpin reproducible research and allow for more accurate cross-comparisons of competing techniques, in very much the same way that the HTS toolkit has done for HMM-based work. In this paper, we introduce Merlin, which is an Open Source neural network based speech synthesis system.

The system has already been extensively used for the work reported in a number of recent research papers. This project will briefly introduce the design and implementation of the toolkit and provide benchmarking results on a freely-available speech corpus in Tamil. In addition to the results here and in the above list of previously-published papers, Merlin is the DNN benchmark system for the 2016 Blizzard Challenge. There, it is used in combination with the Ossian front-end 2 and the WORLD vocoder, both of which are also Open Source and can be used without restriction, to provide an easily-reproducible system.

Implementation Steps

Like HTS, Merlin is not a complete TTS system. It provides the core acoustic modelling functions: linguistic feature vectorisation, acoustic and linguistic feature normalisation, neural network acoustic model training, and generation. Currently, the waveform generation module supports two vocoders: STRAIGHT and WORLD but the toolkit is easily extensible to other vocoders in the future. It is equally easy to interface to different front-end text processors. Merlin is written in Python, based on the theano library. It comes with documentation for the source code and a set of 'recipes' for various system configurations.

The following figure shows the process diagram for this implementation:


Linguistic Features
State Level Alignment
Merlin DNN Training and Synthesis
Audio Files
Vocoder

1.Linguistic Features

Merlin requires an external front-end, such as Festival or Ossian. The front-end output must currently be formatted as HTS style labels with state-level alignment. The toolkit converts such labels into vectors of binary and continuous features for neural network input. The features are derived from the label files using HTS-style questions. It is also possible to directly provide already-vectorised input features if this HTS-like workflow is not convenient.

Festival

Festival offers a general framework for building speech synthesis systems as well as including examples of various modules. As a whole it offers full text to speech through a number APIs: from shell level, though a Scheme command interpreter, as a C++ library, from Java, and an Emacs interface. Festival is multi-lingual (currently English (British and American), and Spanish) though English is the most advanced. Other groups release new languages for the system. And full tools and documentation for build new voices are available through Carnegie Mellon's FestVox project (http://festvox.org)

The system is written in C++ and uses the Edinburgh Speech Tools Library for low level architecture and has a Scheme (SIOD) based command interpreter for control. Documentation is given in the FSF texinfo format which can generate, a printed manual, info files and HTML.

Festival is free software. Festival and the speech tools are distributed under an X11-type licence allowing unrestricted commercial and non-commercial use alike.

Lexicon

A Lexicon in Festival is a subsystem that provides pronunciations for words. It can consist of three distinct parts: an addenda, typically short consisting of hand added words; a compiled lexicon, typically large (10,000s of words) which sits on disk somewhere; and a method for dealing with words not in either list.

Lexical entries consist of three basic parts, a head word, a part of speech and a pronunciation. The headword is what you might normally think of as a word e.g. ‘walk’, ‘chairs’ etc. but it might be any token.

The part-of-speech field currently consist of a simple atom (or nil if none is specified). Of course, there are many part of speech tag sets and whatever you mark in your lexicon must be compatible with the subsystems that use that information. You can optionally set a part of speech tag mapping for each lexicon. The value should be a reverse assoc-list of the following form

```
(lex.set.pos.map
  '((( punc fpunc) punc)
    (( nn nnp nns nnps ) n)))
```

All part of speech tags not appearing in the left hand side of a pos map are left unchanged.

The third field contains the actual pronunciation of the word. This is an arbitrary Lisp S-expression. In many of the lexicons distributed with Festival this entry has internal format, identifying syllable structure, stress markigns and of course the phones themselves. In some of our other lexicons we simply list the phones with stress marking on each vowel.

Some typical example entries are

```
( "walkers" n ((( w oo ) 1) (( k @ z ) 0)) )
( "present" v ((( p r e ) 0) (( z @ n t ) 1)) )
( "monument" n ((( m o ) 1) (( n y u ) 0) (( m @ n t ) 0)) )
( "lives" n ((( l ai v z ) 1)) )
( "lives" v ((( l i v z ) 1)) )
```

By current conventions, single syllable function words should have no stress marking, while single syllable content words should be stressed.

Each lexicon in the system has a name which allows different lexicons to be selected from efficiently when switching between voices during synthesis. The basic steps involved in a lexicon definition are as follows.

First a new lexicon must be created with a new name

(lex.create "cstrlex")

A phone set must be declared for the lexicon, to allow both checks on the entries themselves and to allow phone mapping between different phone sets used in the system

(lex.set.phoneset "mrpa")

The phone set must be already declared in the system.

A compiled lexicon, the construction of which is described below, may be optionally specified

(lex.set.compile.file "/projects/festival/lib/dicts/cstrlex.out")

The method for dealing with unknown words, See section Letter to sound rules, may be set

(lex.set.lts.method 'lts_rules)

(lex.set.lts.ruleset 'nrl)

In this case we are specifying the use of a set of letter to sound rules originally developed by the U.S. Naval Research Laboratories. The default method is to give an error if a word is not found in the addenda or compiled lexicon. (This and other options are discussed more fully below.)

Finally addenda items may be added for words that are known to be common, but not in the lexicon and cannot reasonably be analysed by the letter to sound rules.

(lex.add.entry

'("awb" n (((ei) 1) ((d uh) 1) ((b @ l) 0) ((y uu) 0) ((b ii) 1))))

(lex.add.entry

'("cstr" n (((s ii) 1) ((e s) 1) ((t ii) 1) ((aa) 1))))

(lex.add.entry

'("Edinburgh" n (((e m) 1) ((b r @) 0)))))

Using lex.add.entry again for the same word and part of speech will redefine the current pronunciation. Note these add entries to the current lexicon so its a good idea to explicitly select the lexicon before you add addenda entries, particularly if you are doing this in your own ‘.festivalrc’ file.

For large lists, compiled lexicons are best. The function lex.compile takes two filename arguments, a file name containing a list of lexical entries and an output file where the compiled lexicon will be saved.

Compilation can take some time and may require lots of memory, as all entries are loaded in, checked and then sorted before being written out again. During compilation if some entry is malformed the reading process halts with a not so useful message. Note that if any of your entries include quote or double quotes the entries will probably be misparsed and cause such a weird error. In such cases try setting

(debug_output t)

before compilation. This will print out each entry as it is read in which should help to narrow down where the error is.

When looking up a word, either through the C++ interface, or Lisp interface, a word is identified by its headword and part of speech. If no part of speech is specified, nil is assumed which matches any part of speech tag.

The lexicon look up process first checks the addenda, if there is a full match (head word plus part of speech) it is returned. If there is an addenda entry whose head word matches and whose part of speech is nil that entry is returned.

If no match is found in the addenda, the compiled lexicon, if present, is checked. Again a match is when both head word and part of speech tag match, or either the word being searched for has a part of speech nil or an entry has its tag as nil. Unlike the addenda, if no full head word and part of speech tag match is found, the first word in the lexicon whose head word matches is returned. The rationale is that the letter to sound rules (the next defence) are unlikely to be better than an given alternate pronunciation for a the word but different part of speech. Even more so given that as there is an entry with the head word but a different part of speech this word may have an unusual pronunciation that the letter to sound rules will have no chance in producing.

Finally if the word is not found in the compiled lexicon it is passed to whatever method is defined for unknown words. This is most likely a letter to sound module. See section Letter to sound rules.

Optional pre- and post-lookup hooks can be specified for a lexicon. As a single (or list of) Lisp functions. The pre-hooks will be called with two arguments (word and features) and should return a pair (word and features). The post-hooks will be given a lexical entry and should return a lexical entry. The pre- and post-hooks do nothing by default.

Compiled lexicons may be created from lists of lexical entries. A compiled lexicon is much more efficient for look up than the addenda. Compiled lexicons use a binary search method while the addenda is searched linearly. Also it would take a prohibitively long time to load in a typical full lexicon as an addenda. If you have more than a few hundred entries in your addenda you should seriously consider adding them to your compiled lexicon.

Because many publicly available lexicons do not have syllable markings for entries the compilation method supports automatic syllabification. Thus for lexicon entries for compilation, two forms for the pronunciation field are supported: the standard full syllabified and stressed form and a simpler linear form found in at least the BEEP and CMU lexicons. If the pronunciation field is a flat atomic list it is assumed syllabification is required.

Syllabification is done by finding the minimum sonorant position between vowels. It is not guaranteed to be accurate but does give a solution that is sufficient for many purposes. A little work would probably improve this significantly. Of course syllabification requires the entry's phones to be in the current phone set. The sonorant values are calculated from the vc, ctype, and cvox features for the current phoneset. See 'src/arch/festival/Phone.cc:ph_sonority()' for actual definition.

Additionally in this flat structure vowels (atoms starting with a, e, i, o or u) may have 1 2 or 0 appended marking stress. This is again following the form found in the BEEP and CMU lexicons.

Each lexicon may define what action to take when a word cannot be found in the addenda or the compiled lexicon. There are a number of options which will hopefully be added to as more general letter to sound rule systems are added.

The method is set by the command

(lex.set.lts.method METHOD)

Where METHOD can be any of the following

'Error'

Throw an error when an unknown word is found (default).

'lts_rules'

Use externally specified set of letter to sound rules (described below). The name of the rule set to use is defined with the lex.lts.ruleset function. This method runs one set of rules on an exploded form of the word and assumes the rules return a list of phonemes

(in the appropriate set). If multiple instances of rules are required use the function method described next.

‘none’

This returns an entry with a nil pronunciation field. This will only be valid in very special circumstances.

‘FUNCTIONNAME’

Call this as a LISP function function name. This function is given two arguments: the word and the part of speech. It should return a valid lexical entry.

The basic letter to sound rule system is very simple but is powerful enough to build reasonably complex letter to sound rules. Although we’ve found trained LTS rules better than hand written ones (for complex languages) where no data is available and rules must be hand written the following rule formalism is much easier to use than that generated by the LTS training system (described in the next section).

The basic form of a rule is as follows

(LEFTCONTEXT [ITEMS] RIGHTCONTEXT = NEWITEMS)

This interpretation is that if ITEMS appear in the specified right and left context then the output string is to contain NEWITEMS. Any of LEFTCONTEXT, RIGHTCONTEXT or NEWITEMS may be empty. Note that NEWITEMS is written to a different "tape" and hence cannot feed further rules (within this ruleset). An example is

(# [c h] C = k)

The special character # denotes a word boundary, and the symbol C denotes the set of all consonants, sets are declared before rules. This rule states that a ch at the start of a word followed by a consonant is to be rendered as the k phoneme. Symbols in contexts may be followed by the symbol * for zero or more occurrences, or + for one or more occurrences.

The symbols in the rules are treated as set names if they are declared as such or as symbols in the input/output alphabets. The symbols may be more than one character long and the names are case sensitive.

The rules are tried in order until one matches the first (or more) symbol of the tape. The rule is applied adding the right hand side to the output tape. The rules are again applied from the start of the list of rules.

The function used to apply a set of rules if given an atom will explode it into a list of single characters, while if given a list will use it as is. This reflects the common usage of wishing to re-write the individual letters in a word to phonemes but without excluding the possibility of using the system for more complex manipulations, such as multi-pass LTS systems and phoneme conversion.

From lisp there are three basic access functions, there are corresponding functions in the C/C++ domain.

(lts.ruleset NAME SETS RULES)

Define a new set of lts rules. Where NAME is the name for this rule, SETS is a list of set definitions of the form (SETNAME e0 e1 ...) and RULES are a list of rules as described above.

(lts.apply WORD RULESETNAME)

Apply the set of rules named RULESETNAME to WORD. If WORD is a symbol it is exploded into a list of the individual characters in its print name. If WORD is a list it is used as is. If the rules cannot be successfully applied an error is given. The result of (successful) application is returned in a list.

(lts.check_alpha WORD RULESETNAME)

The symbols in WORD are checked against the input alphabet of the rules named RULESETNAME. If they are all contained in that alphabet t is returned, else nil. Note this does not necessarily mean the rules will successfully apply (contexts may restrict the application of the rules), but it allows general checking like numerals, punctuation etc, allowing application of appropriate rule sets.

The letter to sound rule system may be used directly from Lisp and can easily be used to do relatively complex operations for analyzing words without requiring modification of the C/C++ system. For example the Welsh letter to sound rule system consists or three rule sets, first to explicitly identify epenthesis, then identify stressed vowels, and finally rewrite this augmented letter string to phonemes. This is achieved by the following function

```
(define (welsh_lts word features)
  (let (epen str wel)
    (set! epen (lts.apply (downcase word) 'newepen))
    (set! str (lts.apply epen 'newwelstr))
```

```
(set! wel (lts.apply str 'newwel))
(list word
    nil
    (lex.syllabify.phstress wel))))
```

The LTS method for the Welsh lexicon is set to welsh_lts, so this function is called when a word is not found in the lexicon. The above function first downcases the word and then applies the rulesets in turn, finally calling the syllabification process and returns a constructed lexically entry.

The Code

We first generate phone level alignments using Festvox code and unified parser from Indic TTS. We need to prepare lexicon for the appropriate language and also a phoneme dictionary.

Go To:

tts_merlin_tamil > iitm_tamil_merlin

Run:

../../s1/festival/bin/festival -b festvox/build_clunits.scm '(build_prompts "etc/txt.done.data")'

If you get errors delete and retype all spaces.
This code generates prompt-lab and prompt-utt directory and the .lab and .utt files respectively for each audio clip.

For more information see:

https://www.iitm.ac.in/donlab/tts/downloads/synthesisDocs/1.Voice_building_initial_and_hybrid_segmentation.pdf

2. State Level Alignments

Kaldi

Monophone training and alignment

Take subset of data for monophone training.

The monophone models are the first part of the training procedure. We will only train a subset of the data mainly for efficiency. Reasonable monophone models can be obtained with little data, and these models are mainly used to bootstrap training for

later-models.

The listed argument options for this script indicate that we will take the first part of the dataset, followed by the location the data currently resides in, followed by the number of data points we will take (10,000), followed by the destination directory for the training data.

```
cd>mycorpus
utils/subset_data_dir.sh --first data/train 10000 data/train_10k
```

Train monophones

Each of the training scripts takes a similar baseline argument structure with optional arguments preceding those. The one exception is the first monophone training pass. Since a model does not yet exist, there is no source directory specifically for the model. The required arguments are always:

- Location of the acoustic data: data/train
- Location of the lexicon: data/lang
- Source directory for the model: exp/lastmodel
- Destination directory for the model: exp/currentmodel

The argument --cmd "$train_cmd" designates which machine should handle the processing. Recall from above that we specified this variable in the file cmd.sh. The argument --nj should be familiar at this point and stands for the number of jobs. Since this is only a subset of the data, we have reduced the number of jobs from 16 to 10. Boost silence is included as standard protocol for this training.

```
steps/train_mono.sh --boost-silence 1.25 --nj 10 --cmd "$train_cmd" \
data/train_10k data/lang exp/mono_10k
```

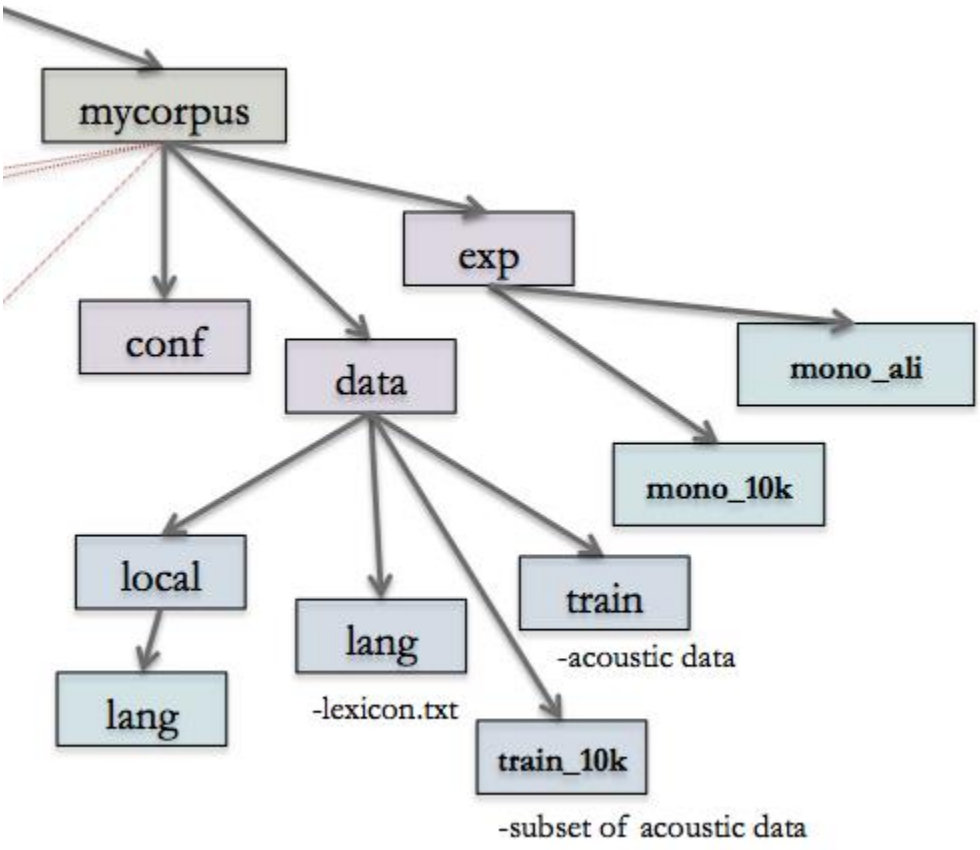
Align monophones

Just like the training scripts, the alignment scripts also adhere to the same argument structure. The required arguments are always:

- Location of the acoustic data: data/train
- Location of the lexicon: data/lang
- Source directory for the model: exp/currentmodel

Destination directory for the alignment: exp/currentmodel_ali
steps/align_si.sh --boost-silence 1.25 --nj 16 --cmd "$train_cmd" \
data/train data/lang exp/mono_10k exp/mono_ali || exit 1;

The directory structure should now look something like this:



Train delta-based triphones

Training the triphone model includes additional arguments for the number of leaves, or HMM states, on the decision tree and the number of Gaussians. In this command, we specify 2000 HMM states and 10000 Gaussians. As an example of what this means, assume there are 50 phonemes in our lexicon. We could have one HMM state per phoneme, but we know that phonemes will vary considerably depending on if they are at the beginning, middle or end of a word. We would therefore want *at least* three different HMM states for each phoneme. This brings us to a minimum of 150 HMM states to model just that variation. With 2000 HMM states, the model can decide if it may be better to allocate a unique HMM state to more refined allophones

of the original phone. This phoneme splitting is decided by the phonetic questions in questions.txt and extra_questions.txt. The allophones are also referred to as subphones, senones, HMM states, or leaves.

The exact number of leaves and Gaussians is often decided based on heuristics. The numbers will largely depend on the amount of data, number of phonetic questions, and goal of the model. There is also the constraint that the number of Gaussians should always exceed the number of leaves. As you'll see, these numbers increase as we refine our model with further training algorithms.

```
steps/train_deltas.sh --boost-silence 1.25 --cmd "$train_cmd" \
2000 10000 data/train data/lang exp/mono_ali exp/tri1 || exit 1;
```

Align delta-based triphones

```
steps/align_si.sh --nj 24 --cmd "$train_cmd" \
data/train data/lang exp/tri1 exp/tri1_ali || exit 1;
```

Train delta + delta-delta triphones

```
steps/train_deltas.sh --cmd "$train_cmd" \
2500 15000 data/train data/lang exp/tri1_ali exp/tri2a || exit 1;
```

Align delta + delta-delta triphones

```
steps/align_si.sh --nj 24 --cmd "$train_cmd" \
--use-graphs true data/train data/lang exp/tri2a exp/tri2a_ali || exit 1;
```

Train LDA-MLLT triphones

```
steps/train_lda_mllt.sh --cmd "$train_cmd" \
3500 20000 data/train data/lang exp/tri2a_ali exp/tri3a || exit 1;
```

Align LDA-MLLT triphones with FMLLR

```
steps/align_fmllr.sh --nj 32 --cmd "$train_cmd" \
data/train data/lang exp/tri3a exp/tri3a_ali || exit 1;
```

Train SAT triphones

steps/train_sat.sh --cmd "$train_cmd" \
4200 40000 data/train data/lang exp/tri3a_ali exp/tri4a || exit 1;

Align SAT triphones with FMLLR

steps/align_fmllr.sh --cmd "$train_cmd" \
data/train data/lang exp/tri4a exp/tri4a_ali || exit 1;

The Code

Here we train a GMM-HMM model with text, audio and phoneme transcription in order to get the state intervals and transition IDs.
We then intelligently combine the state level intervals and transitions with phone level alignment generated before using a python script to generate state level alignments.

Training the model to get the states:

Go To:

tamil_tts_segement

Decoding (optional to improve performance):

Generate separate train and test directories go to the data folder and use

utils/subset_data_dir_tr_cv.sh data/train_full data/train data/test

Run:

(LDA-MLLT does not work on the main node so we need to go to a subnode first)

ssh d3 or ssh d4

nohup bash run.sh &

If decoding,
To check results use

grep WER exp/tri1/decode/wer_*

For only the minimum error rate use

```
grep WER exp/tri1/decode/wer_* | utils/best_wer.sh
```

After the model and tuned for max accuracy and its run we need to extract the states. Use:

```
. ./path.sh

show-transitions data/lang/phones.txt exp/mono/final.mdl exp/mono/final.occs > mono_trans

convert-ali exp/tri2/final.mdl exp/mono/final.mdl exp/mono/tree ark:exp/tri2_ali/

convert-ali exp/tri2/final.mdl exp/mono/final.mdl exp/mono/tree "ark:gunzip -c exp/tri2_ali/ali.1.gz
|" ark:- | ali-to-phones --per-frame=true exp/mono/final.mdl ark:- ark,t:- | utils/int2sym.pl -f 2-
data/lang/phones.txt > mono_pnone_ali


convert-ali exp/tri2/final.mdl exp/mono/final.mdl exp/mono/tree "ark:gunzip -c exp/tri2_ali/ali.1.gz
|" ark,t:mono_ali
```

Copy the files mono_trans, mono_ali, mono_phone_ali to tts_tamil_merlin folder
Copy phone level alignments (.lab files) to
tts_tamil_merlin/experiments/tamil_merlin/test_synthesis/prompt-lab/full

Now we run the python script to get the state level alignments:

```
python  ali_lab.py mono_trans mono_phone_ali mono_ali
experiments/tamil_merlin/test_synthesis/prompt-lab/full
```

Copy the generated lab files to the acoustic and duration models of Merlin

3. Vocoding

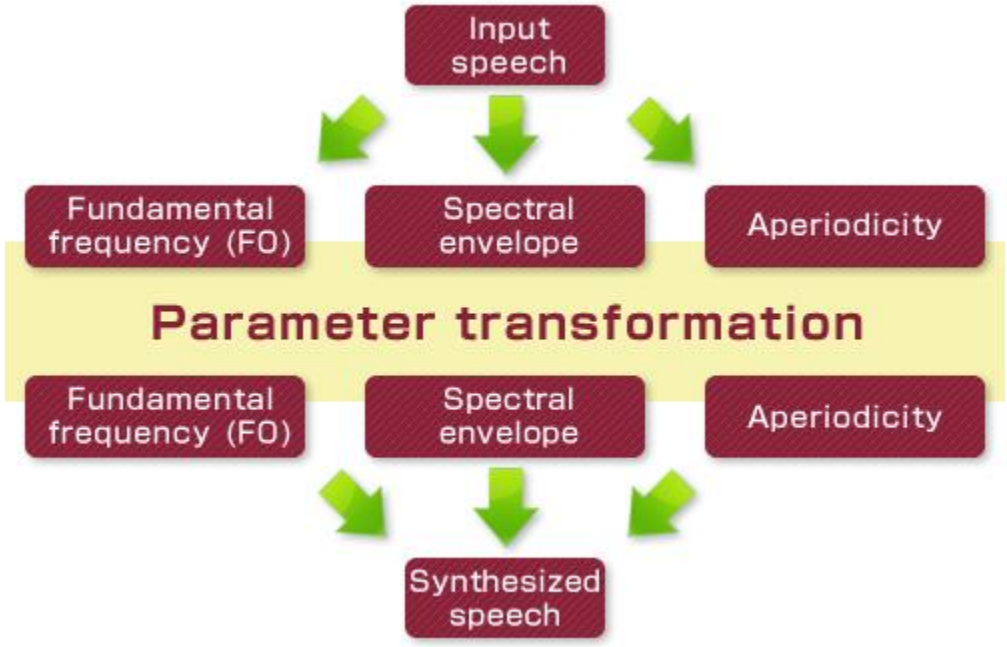
Currently, the system supports two vocoders: STRAIGHT (the C language version) and WORLD. STRAIGHT cannot be included in the distribution because it is not Open Source, but the Merlin distribution does include a modified version of the WORLD vocoder. The modifications add separate analysis and synthesis executables, as is necessary for SPSS. It is not difficult to support some other vocoder, and details on how to do this can be found in the included documentation.

We use the vocoder WORLD

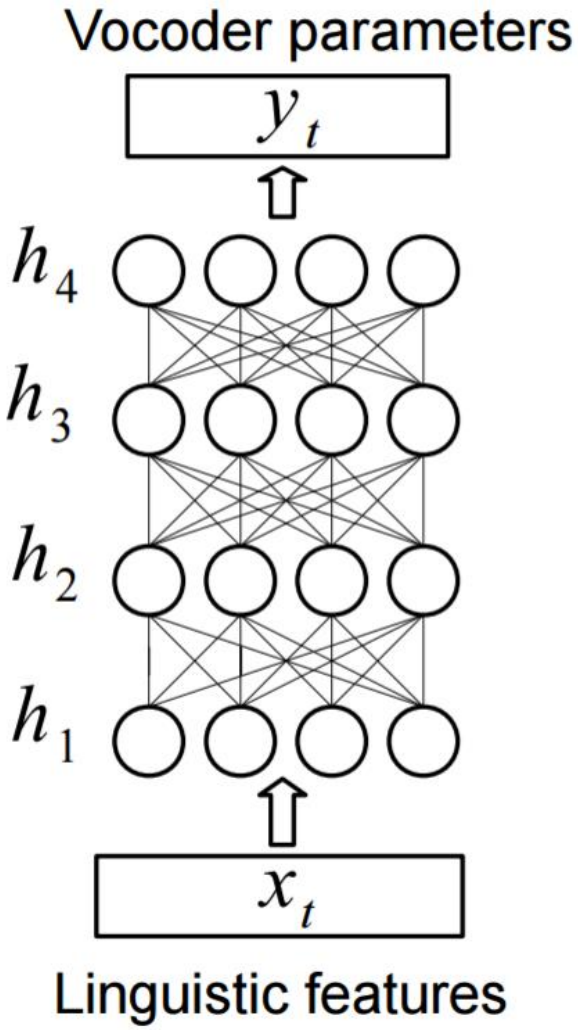
World

WORLD was proposed to synthesize high-quality speech as natural as the input speech. The purpose of WORLD is reducing the computational cost of TANDEM-STRAIGHT without deterioration. WORLD is superior to TANDEM-STRAIGHT in implementing the real-time singing synthesis, whereas it is inferior to TANDEM-STRAIGHT in manipulating consonant flexibly. Since the concept of WORLD differs from that of TANDEM-STRAIGHT, you should select them based on your purpose.

Figure illustrates the speech processing by WORLD. WORLD decomposes input speech into three parameters: Fundamental frequency (F0), spectral envelope and aperiodicity (Note: excitation signal employed in the previous version was destroyed in version 0.2.0). We can manipulate three parameters and generate the speech from them. Three parameters are effective as the parameters to analyse para- and non-linguistic information.



4. DNN Training and Synthesis



Feature normalisation

Before training a neural network, it is important to normalise features. The toolkit supports two normalisation methods: minmax, and mean-variance. The min-max normalisation will normalise features to the range of [0.01 0.99], while the meanvariance normalisation will normalise features to zero mean and unit variance. Currently, by default the linguistic features undergo min-max normalisation, while output acoustic features have mean-variance normalisation applied.

Acoustic modelling

Merlin includes implementations of several currently-popular acoustic models, each of which comes with an example 'recipe' to demonstrate its use.

Feedforward neural network

A feedforward neural network is the simplest type of network. With enough layers, this architecture is usually called a Deep Neural Network (DNN). The input is used to predict the output via several layers of hidden units, each of which performs a nonlinear function, as follows:

$$h_t = H(W_{xh} x_t + b_h) \quad (1)$$

$$y_t = W_{hy} h_t + b_y, \quad (2)$$

where $H(\cdot)$ is a nonlinear activation function in a hidden layer, W_{xh} and W_{hy} are the weight matrices, b_h and b_y are bias vectors, and $W_{hy} h_t$ is a linear regression to predict target features from the activations in the preceding hidden layer. Fig. 1 is an illustration of a feedforward neural network. It takes linguistic features as input and predicts the vocoder parameters through several hidden layers (in the figure, four hidden layers). In the remainder of this paper, we will use DNN to indicate a feedforward neural network of this general type. In the toolkit, sigmoid and hyperbolic tangent activation functions are supported for the hidden layers.

The Code

Before using merlin please go through the following tutorial for a simple demo:

http://jrmeyer.github.io/merlin/2017/02/14/Installing-Merlin.html

Only gpu 5 allows the use of merlin

Once you you have understood the toolkit use run_full_voice.sh script to train both models and synthesise voice.

Conclusion

Tamil voice is synthesized, with legibility and accuracy. Potential future improvements include:

- Trying other types of Neural Networks (LSTMs, DLSTMs)
- Multilanguage synthesis

Credits

- Sandeep Kothinti, RA at Speech Lab for his continued guidance
- Prof. S. Umesh for the opportunity and guidance
- Carnegie Mellon University, Alan W Black and Kevin Lenzo for front-end linguistic feature extraction
- Zhizheng Wu, Oliver Watts, Simon King, "Merlin: An Open Source Neural Network Speech Synthesis System" in Proc. 9th ISCA Speech Synthesis Workshop (SSW9), September 2016, Sunnyvale, CA, USA
- Speech and Music Technology Lab, IITM for the unified parser
- Speech Lab, IITM for various computational resources

References

Z.-H. Ling, S.-Y. Kang, H. Zen, A. Senior, M. Schuster, X.-J. Qian, H. M. Meng, and L. Deng, "Deep learning for acoustic modeling in parametric speech generation: A systematic review of existing techniques and future trends," IEEE Signal Processing Magazine, vol. 32, no. 3, pp. 35–52, 2015.

H. Zen, "Acoustic modeling in statistical parametric speech synthesis - from HMM to LSTM-RNN," in Proc. MLSLP, 2015, invited paper.

T. Weijters and J. Thole, "Speech synthesis with artificial neural networks," in Proc. Int. Conf. on Neural Networks, 1993, pp. 1764–1769.

G. Cawley and P. Noakes, "LSP speech synthesis using backpropagation networks," in Proc. Third Int. Conf. on Artificial Neural Networks, 1993, pp. 291–294.

C. Tuerk and T. Robinson, "Speech synthesis using artificial neural networks trained on cepstral coefficients." in Proc. European Conference on Speech Communication and Technology (Eurospeech), 1993, pp. 4–7.

M. Riedi, "A neural-network-based model of segmental duration for speech synthesis," in Proc. European Conference on Speech Communication and Technology (Eurospeech), 1995, pp. 599–602.

O. Karaali, G. Corrigan, N. Massey, C. Miller, O. Schnurr, and A. Mackie, "A high quality text-to-speech system composed of multiple neural networks," in Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP), vol. 2, 1998, pp. 1237–1240.

Z.-H. Ling, L. Deng, and D. Yu, "Modeling spectral envelopes using Restricted Boltzmann Machines and Deep Belief Networks for statistical parametric speech synthesis," IEEE Transactions on Audio, Speech, and Language Processing, vol. 21, no. 10, pp. 2129–2139, 2013.

S. Kang, X. Qian, and H. Meng, "Multi-distribution deep belief network for speech synthesis," in Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP), 2013, pp. 8012–8016.

S. Kang and H. Meng, "Statistical parametric speech synthesis using weighted multi-distribution deep belief network," in Proc. Interspeech, 2014, pp. 1959–1963.

O. Watts, G. E. Henter, T. Merritt, Z. Wu, and S. King, “From HMMs to DNNs: where do the improvements come from?” in Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP), 2016.

C. Valentini-Botinhao, Z. Wu, and S. King, “Towards minimum perceptual error training for DNN-based speech synthesis,” in Proc. Interspeech, 2015, pp. 869–873.

Z. Wu and S. King, “Minimum trajectory error training for deep neural networks, combined with stacked bottleneck features,” in Proc. Interspeech, 2015, pp. 309–313.

Y. Fan, Y. Qian, F. K. Soong, and L. He, “Sequence generation error (SGE) minimization based deep neural networks training for text-to-speech synthesis,” in Proc. Interspeech, 2015, pp. 864–868.

Z. Wu and S. King, “Improving trajectory modelling for dnn-based speech synthesis by using stacked bottleneck features and minimum generation error training,” IEEE Trans. Audio, Speech and Language Processing, 2016.

Y. Fan, Y. Qian, F. Xie, and F. K. Soong, “TTS synthesis with bidirectional LSTM based recurrent neural networks,” in Proc. Interspeech, 2014, pp. 1964–1968.

H. Zen and H. Sak, “Unidirectional long short-term memory recurrent neural network with recurrent output layer for low-latency speech synthesis,” in Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP), 2015, pp. 4470–4474.

B. X. Wenfu Wang, Shuang Xu, “Gating recurrent mixture density networks for acoustic modeling in statistical parametric speech synthesis,” in Proc. IEEE Int. Conf.