L21: HTK

Introduction
Building an HTK recognizer
Data preparation
Creating monophone HMMs
Creating tied-state triphones
Recognizer evaluation
Adapting the HMMs

This lecture is based on The HTK Book, v3.4 [Young et al., 2009]
Introduction

What is HTK?

- HTK is a toolkit for building Hidden Markov Models
- HTK is primarily designed for building speech recognizers
  - Estimating HMM parameters from a set of training utterances
  - Transcribing unknown utterances
Available HTK tools

- Data preparation tools
  - Convert speech waveforms into parametric format (e.g. MFCC)
  - Convert the associated transcriptions into appropriate format (e.g., phone or word labels)

- Training
  - Define the topology of the HMMs (i.e., prototypes)
  - Initialize models (e.g., bootstrap, flat start)
  - Train models (e.g., parameter tying, Baum-Welch, adaptation)

- Testing
  - Viterbi based recognizer (HVite) – can also be used for forced alignment
  - Decoder for large vocabulary speech recognition (HDecode)

- Analysis
  - Evaluate model performance (e.g., WER, ROC, ...)
Using HTK

- HTK consists of a set of tools to be run with a command-line interface
  - Each tool contains a set of required arguments and optional arguments
  - Optional arguments are always prefixed by a minus sign

```
HFoo -T 1 -f 34.3 -a -s myfile file1 file2
```

Optional arguments (4)  Main arguments (2)

- HTK tools can also be controlled by parameters in a configuration file

```
HFoo -C config1 -C config2 -f 34.3 -a -s myfile file1 file2
```

Configuration files (2)
Building an HTK recognizer

A tutorial example

– For the remainder of this lecture, we will introduce HTK by constructing a recognizer for a simple voice dialing application
  • Corpus will consist of continuously spoken digits and proper names
  • Though the task is simple, the recognizer will be sub-word-based so it can be easily expanded
  • HMMs will be continuous Gaussian mixture tied-state triphone with clustering performed using phonetic decision trees
Data preparation

Step 1 – the Task Grammar

– Application: voice-operated interface for phone dialing
– ASR must handle digit strings and personal names such as
  • “Dial nine zero four one oh nine”
  • “Phone Woodland”
– HTK provides a grammar definition language for simple tasks, consisting of variable definitions and regular expressions
  • Vertical bars denote alternatives
  • Square brackets denote optional items
  • Angle braces denote one or more repetitions
gram

$digit = \text{ONE} | \text{TWO} | \text{THREE} | \text{FOUR} | \text{FIVE} | \text{SIX} | \text{SEVEN} | \text{EIGHT} | \text{NINE} | \text{OH} | \text{ZERO};$

$name = \{ \text{JOOP} \} \text{ JANSSEN} | \{ \text{JULIAN} \} \text{ ODELL} | \{ \text{DAVE} \} \text{ OLLASON} | \{ \text{PHIL} \} \text{ WOODLAND} | \{ \text{STEVE} \} \text{ YOUNG};$

( SENT-START ( DIAL <$digit> | (PHONE|CALL) $name) SENT-END )
The HTK recognizer will require a word network, which can be created automatically from the grammar above using the HParse tool:

```
HParse gram wdnet
```

- where ‘gram’ contains the above grammar
Step 2 – the Dictionary

- Create a sorted list of all required words (file ‘wlist’)
  - For our grammar, this can be done manually
- Obtain a pronunciation dictionary (file ‘beep’)
  - Publicly available; see p. 27 for URL
- The HTK tool HDMan will then create a new dictionary by finding pronunciations for each word in ‘wlist’
  
  \[
  \text{HDMan} \ -m \ -w \ \text{wlist} \ -n \ \text{monophones1} \ -l \ \text{dlog} \ \text{dict} \ \text{beep} \ \text{names}
  \]
  - ‘names’: phonetic transcription of all proper names in our grammar
  - ‘global.ded’: edit script with additional commands (p. 27)
  - ‘monophones1’: list of phones used (output)

- The general format for each dictionary entry will be
  
  \[
  \text{WORD} \ [\text{outsym}] \ \text{p1} \ \text{p2} \ \text{p3} \ \ldots
  \]
Fig. 3.3 Step 2
Step 3 – Recording the Data

- Generate list of prompts for training and test sentences with HSGen
  
  ```bash
  HSGen -l -n 200 wdnet dict > testprompts
  ```
  
  - which will randomly traverse the word network, generate 200 numbered utterances, and pipe them to file ‘testprompts’

- Record training and test sentences
  
  - You can use HTK tool HSLab or other audio recording program

Fig. 3.4 Step 3
Step 4 – Creating the Transcription Files

– The first step is to create an orthographic transcription in HTK label format (MLF), which can be done with Perl script ‘prompts2mlf’

  prompts2mlf words.mlf trainingprompts

  • ‘trainingprompts’: list of training utterances
  • ‘words.mlf’: orthographic transcription (output)
    – This is an example of a Master Label File (MLF), a single file containing a complete set of transcriptions (HTK allows each individual transcription to be stored in its own file but it is more efficient to use an MLF)

– The second step is to generate phone-level MLFs, using HLEd

  HLEd -l '*' -d dict -i phones0.mlf mkphones0.led words.mlf

  • ‘phones0.mlf’: phone-level transcription
  • ‘mkphones0.led’: edit script (see p. 30), which commands HLEd to
    – Replace every word in ‘words.mlf’ with its pronunciation in ‘dict’
    – Insert a silence model at the start and end of every utterance, and
    – Delete all short-pause labels
trainingprompts

S0001 ONE VALIDATED ACTS OF SCHOOL DISTRICTS
S0002 TWO OTHER CASES ALSO WERE UNDER ADVISEMENT
S0003 BOTH FIGURES WOULD GO HIGHER IN LATER YEARS
S0004 THIS IS NOT A PROGRAM OF SOCIALIZED MEDICINE
etc

words.mlf

#!MLF#!
"*/S0001.lab"
ONE
VALIDATED
ACTS
OF
SCHOOL
DISTRICTS
.
"*/S0002.lab"
TWO
OTHER
CASES
ALSO
WERE
UNDER
ADVISEMENT
.

phones0.mlf

#!MLF#!
"*/S0001.lab"
sil
w
ah
n
v
ae
l
ih
d..
.. etc

Fig. 3.5 Step 4
Step 5 – Coding the data

– The final stage of data preparation is to parameterize the speech into sequence of feature vectors
  • HTK supports both FFT-based and LPC-based analysis
  • Here we will use MFCCs

– Coding is performed with the HTK tool HCopy

  \texttt{HCopy -T 1 -C config -S codetr.scp}

  • ‘config’: specifies all the conversion parameters
  • ‘codetr.scp’: script file, containing list of source files and their corresponding outputs

– The output is a separate MFCC file (*.mfc) for each audio file (*.wav) in the script file ‘codetr.scp’
config

# Coding parameters
TARGETKIND = MFCC_0
TARGETRATE = 100000.0
SAVECOMPRESSED = T
SAVEWITHCRC = T
WINDOWSIZE = 250000.0
USEHAMMING = T
PREEMCOEF = 0.97
NUMCHANS = 26
CEPLIFTER = 22
NUMCEPS = 12
ENORMALISE = F

codetr scp

/root/sjy/waves/S0001.wav /root/sjy/train/S0001.mfc
/root/sjy/waves/S0002.wav /root/sjy/train/S0002.mfc
/root/sjy/waves/S0003.wav /root/sjy/train/S0003.mfc
/root/sjy/waves/S0004.wav /root/sjy/train/S0004.mfc
Creating monophone HMMs

Introduction

– In this step, we create a set of identical monophone HMMs and train them, realign the training utterances, and retrain the HMMs

Step 6 – Creating flat-start HMMs

– Define prototype model containing HMM topology (file ‘proto’)  
  • For phone-based systems, a 3-state left-right with no skips is appropriate

– Compute global mean and variance of data, and initialize HMM proto

  `HCompV -C config -f 0.01 -m -S train.scp -M hmm0 proto`
  • ‘train.scp’: script containing the list of all training WAV files
  • ‘hmm0’: directory where new HMM proto with global mean and variance will be saved
    – HCompV also creates file ‘vFloor’ containing a variance floor for the HMMs

– Manually generate two files and save them on ‘hmm0’  
  • ‘macro’: contains global-options macro and the variance floor macro generated earlier (see p. 34)
  • ‘hmmdefs’: contains a copy of ‘proto’ for each phoneme, including ‘sil’
13 MFCC + Δ + Δ²

proto

```
"o <VecSize> 39 <MFCC_0_D_A>
"h "proto"
 BEGINHMM
 <NumStates> 5
 <State> 2
   <Mean> 39
     0.0 0.0 0.0 ...
   <Variance> 39
     1.0 1.0 1.0 ...
 <State> 3
   <Mean> 39
     0.0 0.0 0.0 ...
   <Variance> 39
     1.0 1.0 1.0 ...
 <State> 4
   <Mean> 39
     0.0 0.0 0.0 ...
   <Variance> 39
     1.0 1.0 1.0 ...
 <TransP> 5
  0.0 1.0 0.0 0.0 0.0 0.0
  0.0 0.6 0.4 0.0 0.0
  0.0 0.0 0.6 0.4 0.0
  0.0 0.0 0.0 0.7 0.3
  0.0 0.0 0.0 0.0 0.0
 ENDHMM
```

macros

```
~O
  <VecSize> 39
  <MFCC_0_D_A>
~V "varFloorl"
  <Variance> 39
    0.0012 0.0003 ...
```

hmmdefs

```
~h "aa"
  BEGINHMM ...
  ENDHMM
~h "eh"
  BEGINHMM ...
  ENDHMM
  ... etc
```
– Re-estimate flat-start monophone HMMs in directory ‘hmm0’

```
HERest -C config -I phones0.mlf -t 250.0 150.0 1000.0 -S
train.scp -H hmm0/macros -H hmm0/hmmdefs -M hmml monophones0
```

  • ‘monophones0’: same as ‘monophones1’ without short-pause (sp)
  • Results will be saved in new directory ‘hmm1’

– Repeat HERest twice more, generating directories ‘hmm2’ and ‘hmm3’
Step 7 – Fixing the Silence Models

- In this step, we make the models more robust by
  - Adding transitions to/from states 2 and 4 in the silence model,
  - Creating a 1-state short pause (sp) model tied to the center state of ‘sil’

- This is done in two steps
  - Manually edit ‘hmm3/hmmdefs’ to add a new (sp) model, and save it in a
    new directory ‘hmm4’ (see p. 35)
  - Run tool HHEd to add extra transitions and tie the (sp) model

    \[ \text{HHEd} \ -H \ hmm4/macros \ -H \ hmm4/hmmdefs \ -M \ hmm5 \ sil.hed \]
    \[ \text{monophones1} \]
    - ‘sil.hed’: script containing code to add transitions and tie states
  - Repeat HERest twice more, generating directories ‘hmm6’ and ‘hmm7’
Fig. 3.10 Step 7
Step 8 – Realigning the Training Data

- Realign training data and create new transcriptions

\texttt{HVite} -l '/*' -o SWT -b silence -C config -a -H hmm7/macros -H hmm7/hmmdefs -i aligned.mlf -m -t 250.0 -y lab -I words.mlf -S train.scp dict monophones1

- ‘aligned.mlf’: will contain the realigned utterances, in this case considering the best fit of all possible pronunciations in the dictionary
- Before doing this, we will need to manually insert an entry ‘silence sil’ at the end of the dictionary file ‘dict’

- Repeat HERest twice more, generating directories ‘hmm8’ and ‘hmm9’
Creating Tied-State Triphones

Introduction

The last step of model building is to transform the monophone HMMs into context-dependent triphone HMMs, which is done in two steps:

- First, convert monophone transcriptions into triphone transcriptions, create a new set of triphones (by copying monophones), and reestimating.
- Second, tie similar acoustic states (to ensure robust estimation).

Step 9 – Making Triphones from Monophones

Generate triphones transcriptions for training data:

```
HLEd -n triphones1 -l '*' -i wintri.mlf mktri.led aligned.mlf
```

- ‘mktri.led’: edit script explaining how to handle pauses (p. 38)
- ‘wintri.mlf’: word-internal triphone transcriptions (output)
- ‘triphones1’: list of triphones (output)
Generate context-dependent triphones by cloning monophones

```
HHEd -B -H hmm9/macros -H hmm9/hmmdefs -M hmm10 mktri.hed monophones1
```

- ‘mktri.hed’: edit script describing the procedure for HHEd (p. 39)

Reestimate (twice) the triphone set with HERest

```
HERest -B -C config -I wintri.mlf -t 250.0 150.0 1000.0 -s stats -S train.scp -H hmm11/macros -H hmm11/hmmdefs -M hmm12 triphones1
```

- ‘stats’: state occupation statistics (output), to be used during the state-clustering process (step 10)
Fig. 3.13 Step 9
Step 10 – Making Tied-State Triphones

– The last step in model building is to tie states within triphone sets in order to share data and make robust parameter estimates.

– Here we use a method based on decision trees, which is based on asking questions about the left and right context of each triphone:

```
HHEd -B -H hmm12/macros -H hmm12/hmmdefs -M hmm13 tree.hed triphones1 > log
```

• ‘tree.hed’: edit script describing which context to examine and what results to save in output files (p. 41)
Prior to executing HHEd, we will need to generate a list of all possible triphones on the entire dictionary, not just those on the training set (this is needed for recognition purposes)

```
HDMan -b sp -n fulllist -g global.ded -l flog beep-tri beep
```

- ‘global.ded’: global command TC (p. 42)
- ‘fulllist’: full list of all triphones (output)
- ‘beep-tri’: triphone transcription of all words in grammar (output)
- ‘tiedlist’: list of all tied states (output)
- ‘trees’: list of all trees (output)

Repeat HERest twice more, generating directories ‘hmm14’ and ‘hmm15’
Fig. 3.14 Step 10
Recognizer evaluation

Step 11 – Recognizing the Test Data

- First, run the recognizer on test data

```
HVite -C config -H hmm15/macros -H hmm15/hmmdefs -S test.scp -l '*' -i recout.mlf -w wdnet -p 0.0 -s 5.0 dict tiedlist
```

- ‘config’: configuration file to allow word-internal expansion (p. 43)
- ‘test.scp’: list of test files (MFC)
- ‘recout.mlf’: transcription output

- Finally, compare recognizer output against ground truth

```
HResults -I testref.mlf tiedlist recout.mlf
```

- ‘testref.mlf’: word-level transcription for each test file (ground truth)