Chapter 11

Development using Sphinx

Section Author: Yitao Sun and Arthur Chan

11.1 Sphinx's recognizer Application Programming Interface (API)

In previous chapters, the property of a speech recognizer was described. It is important to realize that that is mainly for the core speech recognition engine. Using the speech recognizer in real-life application has other considerations:

1. **Resource allocation** Resource may be required to be change dynamically as a per utterance perspective.
2. **Real-time response** The system is required to provide quick and natural response.
3. **Programming Interface** The developer can uses the functionalities of the recognizer through a set of application programmer interface (API).

This section discuss these issues and provide a brief introduction of the programming using Sphinx. We will describe each issue in detail.

11.2 Issues in live-mode recognition

We will describe three important issues in using the live mode decoder, that include how to segment the speech, the impact of recognizer's speed and the consideration of delay.

11.2.1 Issues in speech segmentation

One of the major difference between the batch mode recognizer and the live mode recognizer is that a clear begin and end time of the whole utterance is usually well-defined before the decooding. While, the live mode decoder may have no knowledge about the length of the whole utterances. For example, in so-called continous recording mode of the recognizer. The audio data was continuously captured and feed into the the live mode recognizer. To avoid delay, usually the application developer will define a fix and usually short speech segment's size when inputting the speech signal into the recognizer.

Another way to decide which segment is correct is to parse the sentence with and end-pointer. The segmented output will be put into the recognizer. Internally, sphinx used an energy-histogram-based end-pointer to decide an end-pointer. This is usually not the optimal way to do end-pointing. Ziad Al Bawab and Arthur Chan have created a version of Sphinx 3.X that incorporated and end-pointer, ep. You can find this information at,

www.cs.cmu.edu/ archan/share/s3ep.tgz .

11.2.2 Issues in speed and accuracy performance

Usually, real-life application require fast response from the speech recognizer. This requires so called real-time processing from the speech recognizer. As the design of Sphinx give the programmer the freedom to use different configuration of acoustic models and tune the system themselves. It also become the users or developers responsiblities to make sure the recognizer can be ran in real-time or faster than real-time.

One useful measure to allow us to have some feeling of the speed of the code is the xRT scale. For example, for a 1s waveform, if the processing requires N s to process, then we call it the system as NxRT system. Obviously, to have real-time response, N has to be a number smaller than 1 or else the response of the recognizer would be slower than its input.

The first important consideration of building a real-time speech recognition system is to decide whether it is feasible to build such a system. In 2004, it is possible to tune live_pretend (using techniques described in Chapter 8) to be as fast as smaller than 1xRT for a task which has 20k words with perplexity around 80 using trigrams language model.

The system will still give very reasonable performance compared to the un-tuned system. (within 5% relative degradation)

However, it is not always possible to make the recognizer less than 1xRT with reasonable performance due to limitation of computation power. For example, in 2004, broadcast news (65k words) is one task that a 1-pass speech recognizer such as Sphinx will have difficulty to give reasonable performance in 1xRT. For example, in some task, 10 fold speed-up would mean 20determined by experiments.

It is important to realize that it is always possible to the recognizer down to a certain speed, this will , however, always cause degradation of accuracy of the recognizer.

As another sidenote, usually speech recognition will not be the only component of a dialogue system. Other components such as end-pointer, speech synthesis can also require amount of computation. Therefore, it is important to profile individual components in troubleshooting your system.

11.2.3 Issues in delay in response

Delay of a speech recognizer can be caused by several reasons. Obviously, if the speed of the recognizer cannot catch up with the audio capturing routine, it will cause a substantial delay. This sub-section assumes the users of Sphinx have already tuned the system under 1xRT.

The most major factor that affect the delay of speech recognition system is the mechanism how the audio-capturing routines and the recognizer interact with each other. For example, if the audio routine only send data to recognizer every 10s, the system will bound to have long delay.

Internally, sphinx use a set of reasonably portable libraries for audio capturing. However, by its own design, capturing the audio data from the audio device will become a blocking call. This can also cause unreasonable delay. Therefore, these routines were not exposed to the public.

We, therefore, recommend the user take care of the problem of audio capturing separate from using the recognizer. For PC hardware, these problems can be solved by using libraries such as portaudio.

www.portaudio.org

Portaudio provides a set of mechanism to reduce the delay of the system, please consult its documentation to learn the howto.

11.3 Using the live mode recognizer

There are several major important process in using Sphinx's live mode recognizer.

1. **Resource Allocation/Deallocation** In a recognizer life-time, several important resources need to load in such as dictionary and acoustic model. This initialization/uninitialization from the command-line front-end is done by calling ld_init/ld_finish.
2. **Decoding Session Begin/End** [Need to rewrite] For every utterance, Sphinx start recognition by starting ld_begin_utt and end that by ld_end_utt. At every utterance, it is possible to set dynamic resource such as language models and transformation matrices for adaptation.
3. **Decoding process** The actual Viterbi decoding is done at ld_process_raw.

11.3.1 Programs

```
if (ld_init(&decoder))
printf("Initialization failed.
n");
return -1;
if (ld_begin_utt(&decoder, 0))
printf("Cannot start decoding
n");
return -1;
in_ad = ad_open_sps(samprate);
ad_start_rec(in_ad);
while (1)
num_samples = ad_read(in_ad, samples, BUFSIZE);
if (num_samples > 0)
/** dump the recorded audio to disk */
if (ld_process_raw(&decoder, samples, num_samples) ¡ 0)
printf("Data processing error.
n");
return -1;
ad_stop_rec(in_ad);
ad_close(in_ad);
```

```
if (ld_end_utt(&decoder))
printf(”Cannot end decoding.
n”);
return -1;
ld_finish(&decoder);
```

11.4 Common problems in setting up the live-mode speech recognizer

Are you doing the right thing? For example, are you sure you recording routine is operated correctly? This can be checked by dumping the audio data to a wave file.

In our experience in CMU, 95has a problem, it was caused by model/feature mismatch.

What is model/feature mismatch? As you already know, a frame is represented as a vector of floating point number in a speech recognizer. This representation process is called speech encoding. There are many speech encoding schemes if the world. Sphinx use MFCC and Sphinx allow the users a lot of freedom to tweak this encoding scheme.

Here comes the problem, there are several natural places in the training process, you need to specify the parameters of the encoding scheme. For example, in training, when one tries to convert the wave file to cepstrum, one need to specify all these parameters. Now, in decoding, either live-mode or batch mode, if these parameters were not used. There will be drammatic numerical difference between the parameter used in training the model and the parameter used in decoding. In that case, the pattern recognition algorithm will be totally messed up and will not give you any meaningful results.

For one practical example, if you are using the broadcast news model for regression test for our default testing grammar, if there is model/feature mismatch, you will see that the result will become ,

¡s¿ ¡/s¿

all the time.

Whereas, the correct result is actually

¡s¿ P I T T S B U R G H ¡/s¿

Other symptoms of model/feature mismatch includes repeated short outputs is always recognized. Or there is always no output even other parts of the system is diagnosed to be correct.

11.5 livedecodeAPI.c doc generated by doc++

Contents

1

int **ld_set_uttid** (live_decoder_t* decoder, char* uttid, int autogen)

Utility function declarations

Utility function declarations

2

int **ld_init** (live_decoder_t* decoder)

Initializes the live-decoder.

Initializes the live-decoder. Internal modules, including the cepstra-generating front-end, the language model, and the accoustic models are initialized, and live-decoder internal variables are set to a starting state.

This version of the live-decoder assumes the user has externally parsed arguments using <I>cmd_ln_parse()</I> or <I>cmd_ln_parse_file()</I>. The user is responsible for calling <I>cmd_ln_free()</I> when he/she is done with the live-decoder.

Return Value: `0` for success. -1 for failure.

Parameters: `decoder` Pointer to the decoder.

See Also: ld_finish (→4, *page 79*)

3

int **ld_init_with_args** (live_decoder_t* decoder, int argc, char** argv)

Initializes the live-decoder.

Initializes the live-decoder. Internal modules, including the cepstrum-generating front-end, the language model, and the accoustic models are initialized, and live-decoder internal variables are set to a starting state.

This version uses the <I>cmd_ln.h</I> interface internally. Arguments are parsed and stored internally, and freed later when <I> (→ 4, *page* 79)</I> is called.

Return Value:	`0`	for success. -1 for failure.
Parameters:	`decoder`	Pointer to the decoder.
	`argc`	Argument count.
	`argv`	Argument string array.

See Also: ld_finish (→4, *page 79*)

4

int **ld_finish** (live_decoder_t* decoder)

Wraps up the live-decoder.

Wraps up the live-decoder. All internal modules are closed or unloaded. Internal variables are either freed or set to a finishing state. This function should be called once the user is finished with the live-decoder.

Return Value: `Always` return 0 (for success).

Parameters: `decoder` Pointer to the decoder.

See Also: ld_init
ld_init_with_args (→3, *page 78*)

5

int **ld_begin_utt** (live_decoder_t* decoder, char* uttid)

Marks the start of the current utterance.

Marks the start of the current utterance. An utterance is a session of speech decoding that starts with a call to <I>ld_begin_utt()</I> and ends with a call to <I> (→ 6, *page* 81)</I>. In the duration of an utterance, speech data is processed with either <I> (→ 7, *page* 82)</I> or <I> (→ 8, *page* 83)</I>. Decoding results (hypothesis) can be retrieved any time after the start of an utterance using <I> (→ 9, *page* 84)</I>. However, all previous results will be clobbered at the start of a new utterance.

At the moment, there is an undocumented time limit to the length of an utterance.

Return Value: `0` for success. -1 for failure.

Parameters:

- `decoder` Pointer to the decoder.
- `uttid` Utterance ID string. If <I>null</I>, the utterance ID isignored.

See Also:

ld_end_utt
ld_process_raw
ld_process_ceps
ld_retrieve_hyps (→9, *page 84*)

6

int **ld_end_utt** (live_decoder_t* decoder)

Marks the end of the current utterance.

Marks the end of the current utterance. The Live-Decode API can no longer process speech data until the start of the next utterance. Any hypothesis retrieved prior to the end of the utterance is called a partial hypothesis. Any hypothesis retrieved after the end of the utterance is called the final hypothesis. See <I> (→ 9, *page* 84)</I> on how to retrieve hypothesis.

Return Value: `0` for success. -1 for failure.

Parameters: `decoder` Pointer to the decoder

See Also:
ld_begin_utt
ld_process_raw
ld_process_ceps
ld_retrieve_hyps (→9, *page 84*)

7

int **ld_process_raw** (live_decoder_t* decoder, int16* samples, int32 num_samples)

Process raw 16-bit samples for the current utterance decoding.

Process raw 16-bit samples for the current utterance decoding. This function has to be called in the duration of an utterance. That is, in between calls to <I> (→ 5, *page* 80)</I> and <I> (→ 6, *page* 81)</I>.

Return Value: `0` for success. -1 for failure.

Parameters:

`decoder`	Pointer to the decoder.
`samples`	Buffer of int16 audio samples.
`num_samples`	Number of samples in the buffer.

See Also:
ld_begin_utt
ld_end_utt
ld_process_ceps (→8, *page 83*)

8

int **ld_process_ceps** (live_decoder_t* decoder, float32** frames, int32 num_frames)

Process a buffer of cepstrum frames for the current utterance.

Process a buffer of cepstrum frames for the current utterance. To use this function, make sure that the parameters to the cepstra-generating front-end that matches the parameters to the decoder's accoustic model. This function has to be called in the duration of an utterance. That is, in between calls to <I> (→ 5, *page* 80)</I> and <I> (→ 6, *page* 81)</I>.

Return Value: `0` for success. -1 for failure.

Parameters:

`decoder`	Pointer to the decoder.
`frames`	Buffer of audio feature frames.
`num_frames`	Number of frames in the buffer.

See Also:
ld_begin_utt
ld_end_utt
ld_process_ceps (→8, *page 83*)

9

int **ld_retrieve_hyps** (live_decoder_t* decoder, char** hyp_str, hyp_t*** hyp_segs)

Retrieve partial or final decoding results (hypothesis).

Retrieve partial or final decoding results (hypothesis). Any hypothesis retrieved prior to the end of the utterance is called a partial hypothesis. Any hypothesis retrieved after the end of the utterance is called the final hypothesis. The hypothesis can be returned in a plain READ-ONLY string and/or an array of READ-ONLY word segments. In the plain string result, all filler and end words are filtered out as well as the pronouciation information. What is left is a very readable string representation of the decoding result. There is no filtering in the word segment result.

Here is an example on how to use the result returned by <I>ld_retrieve_hyps</I>:

<PRE> live_decoder_t d; char *str; hyp_t **segs;

...

ld_retrieve_hyps(&d, &str, &segs); for (; *segs; segs++) </PRE>

Return Value: `0` for success. -1 for failure.

Parameters:

`decoder`	Pointer to the decoder.
`hyp_str`	Returned pointer to a READ-ONLY string. If <I>null</I>,the string is not returned.
`hyp_segs`	Returned pointer to a null-terminated array of wordsegments. If <I>null</I>, the array is not returned.

10

int **ld_abort_utt** (live_decoder_t* decoder)

Abort the current decoding process immediately.

Abort the current decoding process immediately. As opposed to <I> (→ 6, *page* 81)</I>. Retrieving the hypothesis after an abort is not guaranteed.

<EM>!!! NOT IMPLEMENTED YET !!!</EM>

Return Value: `0` for success. -1 for failure.

Parameters: `decoder` Pointer to the decoder.

See Also: ld_end_utt (→6, *page 81*)