Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS
Pihlna do soute SVO 2014

Ondej Plk

Rozpoznni pomocLDI

Speech recognition using KALDI

Institute of Formal and Applied Linguistics
Supervisor: Ing. Mgr. Filip Jurk, Ph.D.
Study branch: Theoretical Computer Science

Prague 2014

I would like to thank my supervisor, Ing. Mgr. Filip Jurk, Ph.D., for his advice, guidence and for keeping me motivated. I would like to thank namely, Matj Korvas for HTK scripts results and advice, Lukilka, David Marek and Ondej Duek for hacks in Vim, Bash and Perl, Marek Vaut for advice with shared library linking, Tomartinec for C++ advice and Pavel Mencl for proof-reading. I am also very grateful to the Kaldi team, which was very responsive and helpful. Expecially, Daniel Povey and Vassil Panayotov. Last but not least, I would like to thank my parents and Ad ihvr all the help and support.

Nv pr: Rozpoznni pomocldi
Autor: Ondej Plk
Katedra: tav formaplikovanngvistiky
Vedoucplomov: Ing. Mgr. Filip Jurk, Ph.D.
E-mail vedouc: jurcicek@ufal.mff.cuni.cz

Abstrakt: Ttem t pr je implementace vkonn rozpozne v open-source syst trv ASR Kaldi (http://kaldi.sourceforge.net/) pro dialogovst. Kaldi ji obsahuje ASR dekod, kterak nejsou vhodno dialogovst. Hlavn dvody jsou jejich malti-malizace na rychlost a jejich velkodnenerov vsledku po ukonenomluvy. Cm t pr je proto vyvinutal-time rozpozne pro dialogovst optimalizovan na rychlost a minimalizujho zpodnrychlene bt realizov napad pomoclti-vlov dek nebo s vyuitrafickch karet pro obecnpoty. Sou je takava akustick modelu a testov ve vyvnialogovyst Vystadial.

Klvova: ASR,rozpoznnuveni, dekod

Title: Automatic speech recognition using Kaldi
Author: Ondej Plk
Department: tav formplikovanngvistiky
Supervisor: Ing. Mgr. Filip Jurk, Ph.D.
Supervisor's e-mail address: jurcicek@ufal.mff.cuni.cz

Abstract: The topic of this thesis is to implement efficient decoder for speech recognition training system ASR Kaldi (http://kaldi.sourceforge.net/). Kaldi is already deployed with decoders, but they are not convenient for dialogue systems. The main goal of this thesis to develop a real-time decoder for a dialogue system, which minimize latency and optimize speed. Methods used for speeding up the decoder are not limited to multi-threading decoding or usage of GPU cards for general computations. Part of this work is devoted to training an acoustic model and also testing it in the "Vystadial" dialogue system.

Keywords: ASR,speech recognition, decoder

Contents

1. Introduction

Spoken dialogue is the most intuitive form of communication among people. Spoken dialogue systems allow people to communicate with machines in an natural way. The quality of a dialogue with a spoken dialogue system significantly depends on speech recognition accuracy.

ASR! (**ASR!**) in a spoken dialogue system converts speech to text so that the dialogue system is able to extract semantic meaning from the text. Dialogue systems are able to exploit multiple alternative text hypotheses. State-of-the-art speech recognisers are able to extract multiple hypotheses in real time. We prefer extracting multiple hypotheses in form of a word lattice because it efficiently represents multiple hypotheses.

The Alex dialogue system had used the **HTK!** (**HTK!**) toolkit [**?**] and the OpenJulius [**?**] lattice speech recogniser to train acoustic models and to decode lattices in real time respectively. Unfortunately, OpenJulius crashes when extracting lattices. Fixing OpenJulius's complicated source code seemed unrealistic due to lack of documentation and community support. As a result, we decided abandon OpenJulius and **HTK!**.

As and alternative, we decided to use the Kaldi toolkit [**?**] because its speech recognisers are able to produce high-quality lattices and are sufficiently fast[1] for real-time recognition.[**?**] In addition, the Kaldi toolkit is actively maintained, and is distributed under the permissive Apache 2.0 license[2]. We still need to implement a speech recogniser which supports incremental speech processing, prepare acoustic modeling scripts and evaluate the developed recogniser, so that the Kaldi toolkit can be used in Alex dialogue system.

1.1 The goals of the thesis

The goals of the thesis are:

1. to build **AM!**s (**AM!**s) using the Kaldi toolkit,
2. to develop new real-time recogniser which supports incremental speech recognition,
3. to integrate the recogniser into our Alex **SDS!** (**SDS!**) written in Python and evaluate its performance.

1.1.1 Training acoustic models

A Kaldi speech recogniser requires statistical models, an **AM!** and a **LM!**. We focus on finding the best **AM!**s. The developed acoustic models will be compared with the **AM!**s trained with the **HTK!** toolkit. The models will be trained for Czech and English acoustic data.

[1]So far, the Kaldi developers focused on improving acoustic model training. However, In August 2012 a Kaldi team published a demo version of an on-line one-best hypothesis speech recogniser.

[2]`http://www.apache.org/licenses/LICENSE-2.0`

1.1.2 Development real-time speech recogniser

We will modify a Kaldi speech recogniser in order to allow incremental speech recognition. The resulting incremental interface will be simple yet allow state-of-the-art performance. In addition, we will implement such speech parametrisation and feature transformation preprocessing, so high-quality acoustic models can be used. Finally, we will implement posterior lattice computation the posterior probabilities of the word lattice representing multiple **ASR!** hypotheses.

In addition, we may suggest potential speed improvements e.g. approximations, use of **GPU!** (**GPU!**) or **DNN!** (**DNN!**) [?].

1.1.3 Integration into Alex SDS!s framework

As Alex **SDS!** is implemented in Python, we will develop a thin Python wrapper which efficiently exposes the speech recognition interfaces. The resulting recogniser will be integrated into Alex **SDS!** and the decoding parameters will be tuned to obtain best performance. The evaluation of the speech recognition setup is an important part of the integration.

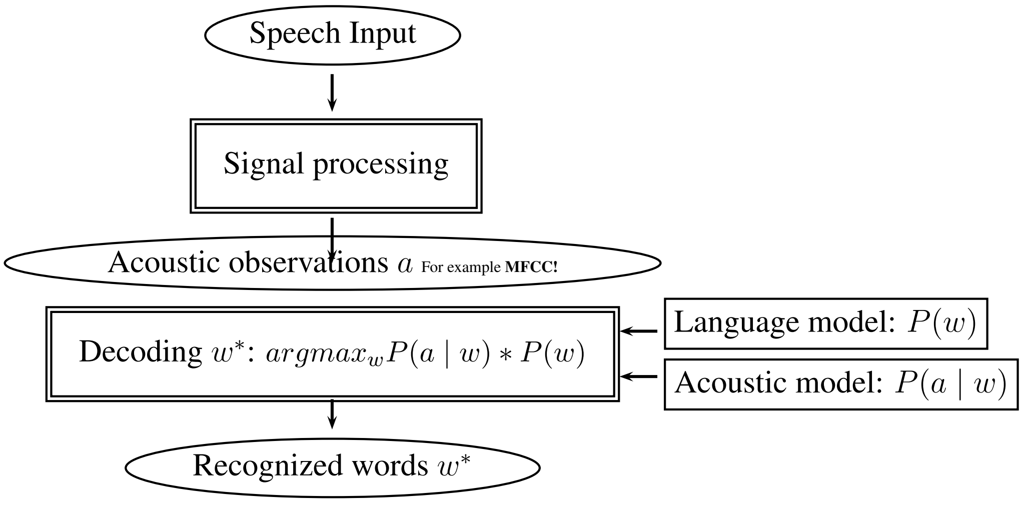
Thesis outline

In Chapter **??** we introduce a fundamental theory of speech recognition for related areas to our work. In Sections **??** and **??** we describe alternatives to Kaldi speech recognition toolkit. At the end of the chapter, we present OpenFST framework which allows the Kaldi library effectively implement many standard speech recognition operations. To obtain high-quality **AM!**s, we develop training scripts described in Chapter **??**. In addition, we compare acoustic models trained by Kaldi and previously used **HTK!** toolkit. Chapter **??** presents in detail the new Kaldi real-time recogniser and discuss its on-line properties. We distinguish the original work done by the Kaldi team and our improvements. Then in Chapter **??**, we describe deployment of the real-time recogniser into dialogue system Alex, we suggest evaluation criteria and also evaluate the integrated recogniser accordingly. Finally, Chapter **??** summarises the thesis and concludes with future research directions.

2. Background

This chapter introduce the basics of speech recognition related to this work. Section **??** introduces speech preprocessing, **AM!** (**AM!**) and **LM!** (**LM!**) training, and explains important aspects of speech decoding. Following sections describe specific speech recognition software implementations. The Kaldi toolkit is described in Section **??**, the **HTK!** toolkit in Section **??** and the Julius decoder in Section **??**.

The statistical methods for continuous speech recognition were established more than 30 years ago. The most popular statistical methods are based on acoustic modelling using **HMM!**s (**HMM!**s) and n-gram **LM!**s, which are also used in Kaldi, the toolkit of our choice. We introduce principles of speech recognition and present techniques which are used in Kaldi.



Speech Input
Signal processing
Acoustic observations a For example **MFCC!**
Decoding w^*: $argmax_w P(a \mid w) * P(w)$
Language model: $P(w)$
Acoustic model: $P(a \mid w)$
Recognized words w^*

Figure 2.1: Architecture of statistical speech recognizer[**?**]

2.1 Automatic speech recognition

The goal of statistical **ASR!** is to decode the most likely word sequence given some speech. The term *decoding* finds its origin in **HMM!** terminology. In speech recognition it is equivalent to *recognizing* the word sequence from the speech. Formally, we search for the most probable sequence of words w^* given the acoustic observations a as described in Equation **??**. The best word sequence w^* does not depend on probability of the acoustic features $P(a)$ so it can be eliminated as shown on the second row of the equation.

$$\begin{aligned} w^* = argmax_w\{P(w \mid a)\} &= argmax_w\{\frac{P(a \mid w) * P(w)}{P(a)}\} \\ &= argmax_w\{P(a \mid w) * P(w)\} \end{aligned} \tag{2.1}$$

The task of acoustic modelling is to estimate the parameters θ of a model so that the probability $P(a \mid w; \theta)$ is as accurate as possible.[1] Similarly, the **LM!** represents the probability $P(w)$. [2]

frames The Figure **??** illustrates the process of decoding the most probable word hypotheses w^* for a given speech utterance. First, the sampled audio signal is processed by speech parametrisation and feature transformations, so the decoding itself takes acoustic features a as input. The acoustic features a are computed on small overlapping windows of audio signal. The acoustic signal in one window is known as a frame.

The decoding itself is performed time synchronously frame by frame using beam search. The beam search expands hypotheses from the previous step by taking into account the new frame features, and it computes probabilities of the expanded hypotheses using the **AM!** and the **LM!**. If the number of hypotheses exceeds the beam, the low probable hypotheses are pruned.

After decoding the last audio frame, all hypotheses represent the whole utterance. The word labels w^* are extracted from the most probable hypothesis which survived the beam search.

Improving the accuracy of a speech recognition engine depends mainly on improving the **AM!** and the **LM!** and also on parameters of the beam search such as threshold how many hypotheses are allowed at maximum.

2.1.1 Speech parameterisation

The goal of speech parameterization is to reduce the negative environmental influences on speech recognition. Speech varies in a number of aspects. Some of them are listed below:

- Differences among speakers' pronunciation depends on gender, dialect, voice, etc.
- Environmental noises. In the dialogue system Alex where our **ASR!** implementation is used, the speech is typically recorded in a noisy street environment.
- The recorded channel. For example the telephone signal is reduced to frequency band between 300 to 3000Hz. The quality of mobile phone signal also influences the quality of the audio signal.

Different speech parametrisation may improve robustness of speech recognition for different recording conditions.

acoustic features Speech parametrisation extracts speech-distinctive acoustic features from the raw waveform. The two most successful methods for speech parametrisation in the last decades are **MFCC!** (**MFCC!**)[?] and **PLP!** (**PLP!**)[?].

Both **MFCC!** and **PLP!** transformations are applied on a sampled and quantized audio signal.[3] For each window **MFCC!** or **PLP!** efficiently computes statistics with a reduced dimension. The methods are very computationally effective and significantly improve the quality of recognised speech.

[1]Acoustic modelling is described in Section **??**.

[2]We describe language modelling in Section **??**.

[3]In our experiments we use 16 kHz sampling frequency and 16 bit samples.

The toolkits used in our dialogue system, Kaldi and **HTK!**, compute **MFCC!** coefficients for given audio input in a similar way.[4] Therefore, we choose **MFCC!** as the speech parametrisation technique for both toolkits, so we can compare them.

The **MFCC!** statistics are computed for each frame. In Figure **??** there are 7 windows — frames with length of 25 ms and frame shift of 10 ms. The whole utterance lasts 85 ms.

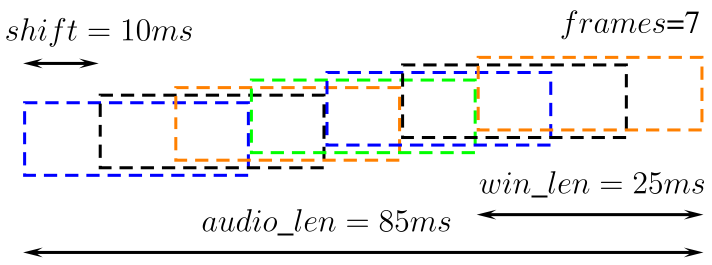


Figure 2.2: **PLP!** or **MFCC!** features are computed every 10 ms seconds in 25 ms windows. Audio length is $(frames - 1) * shift + win\_len = 85ms$

Let us describe the **MFCC!** computation for a 25 ms window shifted by 10 ms and 16kHz audio sampling frequency. The $16000 * 0.025 = 400$ samples in one window are reduced to 13 static cepstral coefficients.

The **MFCC!** static features are usually extended by time derivatives $\Delta + \Delta\Delta$ features [**?**]. As a result, **MFCC!** $\Delta + \Delta\Delta$ extracts $13 + 13 + 13 = 39$ acoustic features for one frame. The original vector of 400 audio samples in one frame is reduced to vector of 39 **MFCC!** $\Delta + \Delta\Delta$ acoustic features.

The **MFCC!** features are computed by the following steps:

1. The audio samples are transformed into the *frequency domain* by **DFT!** (**DFT!**) in a window.
2. The frequency spectrum from the previous step is transformed onto the mel scale, a perceptual scale of frequencies, using triangular overlapping filters.
3. The logs of the powers are taken from each of the mel frequencies.
4. At the end, the discrete cosine transform is applied on the list of mel log powers.
5. The **MFCC!** coefficients are the amplitudes of the resulting spectrum.
6. The $\Delta + \Delta\Delta$ coefficients are computed from the current and previous static features. See Figure **??**.

Feature space transformations

Feature space transformations are usually applied in addition to **MFCC!** or **PLP!** parametrisation. The feature space transformations are also typically

frame

[4]The subtle differences are caused by implementation approaches, but do not affect the quality of **MFCC!** coefficients in a significant way.

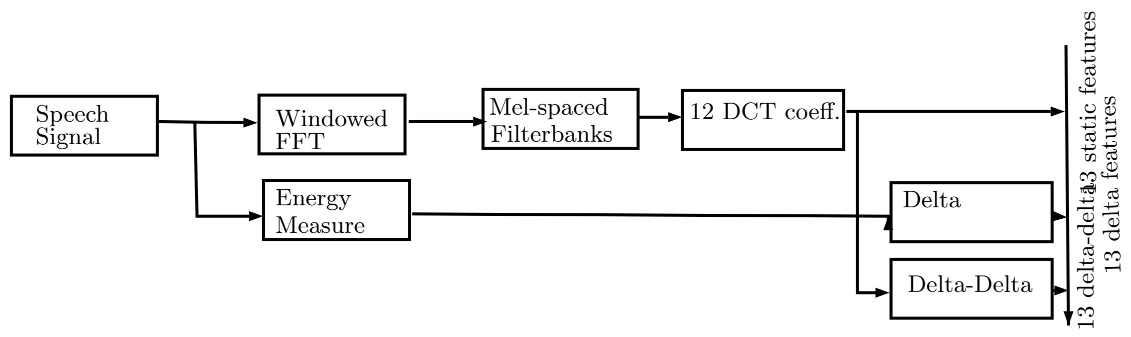


Figure 2.3: Typical setup with 39 features using **MFCC!**.

applied per frame, but they usually take into account context of several preceding (left context) and consecutive frames (right context).

The linear and affine transformations are expressed by the matrix multiplications Ax and Ax^+, respectively. The matrix A represents the transformation. The input vector is x and Ax are the transformed features. The affine transformation uses the extended vector $(x^+)^T = (x_1, \dots, x_n, 1)$ and the matrix $A : (n+1) * (n+1)$.

There is a large variety of available transformations. Depending on the acoustic data, one should choose the most appropriate transformation. Some transformations are estimated discriminatively, some use generative models. Some are speaker dependent, some speaker independent.

We list some of Kaldi transformations in order to illustrate the rich choice of feature transformations in the Kaldi toolkit.

- **HLDA!** (**HLDA!**)[?].
- **LDA!** (**LDA!**)[?] is typically used with **MLLT!** for speaker independent training.
- **MLLT!** (**MLLT!**) also known as **STC!** (**STC!**)[?]
- **ET!** (**ET!**)[?] uses a small number of speaker specific parameters for adaptation to the speaker.
- **CMVN!** (**CMVN!**)[?] typically normalises the cepstrum mean and variance per speaker.

In our acoustic modelling scripts, see Chapter **??**, we use two non-speaker adaptive feature transformations, which can be computed with very small context. The first transformation, $\Delta + \Delta\Delta$ for **MFCC!** coefficients, was already introduced. The second transformation, **LDA!** and **MLLT!**, is described briefly below.

LDA! and MLLT! feature transformation

The **LDA!**+**MLLT!** is an alternative setup to $\Delta + \Delta\Delta$ transformation in our training scripts. We use it also on top of **MFCC!** coefficients. Using several spliced **MFCC!** vectors, the **LDA!**+**MLLT!** searches for the best dynamic transformation.

The combination of **LDA!** and **MLLT!** applies the feature transformation in two steps: **LDA!** reduces the feature dimension and **MLLT!** applies the linear simple transformation[?]. Whereas, the **HLDA!** estimates dimension reduction and space transformation in one step.[?] The **LDA!** + **MLLT!** combination performs very a similar feature transformation to **HLDA!** and gains significant improvements over the $\Delta + \Delta\Delta$ transformation similarly as **HLDA!**[?][?].

2.1.2 Acoustic modelling

Acoustic modelling is arguably the heart of speech recognition. The **AM!** estimates the probability $P(a|w;\theta)$ of generating acoustic features a for given words w and thus directly affects speech recognition quality as seen in Equation **??**.

Acoustic modelling has only partial information available for training the **AM!** parameters θ because the corresponding textual transcription is time-unaligned. The hidden information of the word (time) alignment in an utterance makes acoustic model training more challenging. Modern speech recognition toolkits use **HMM!**s for modelling uncertainty between acoustic features and the corresponding transcription.

Choice of training units

The most successful acoustic modelling methods do not estimate the $P(a|w)$ directly, but estimate the probability $P(a|f_1 f_2 f_3 f_4)$ of generating acoustic features a for phones $w = f_1 f_2 f_3 f_4$ which forms the pronunciation of the word w. Moreover, triphones are used even more successfully for estimating the probability of acoustic features given a word pronunciation.

phone The phone is the smallest contrastive unit of speech. Let us see a few examples of words and their phonetic transcriptions according CMU dictionary[?].

- *youngest* & *Y AH1 NG G AH0 S T*
- *youngman* & *Y AH1 NG M AE2 N*
- *earned* & *ER1 N D*
- *ear* with two transcribed pronunciations *IY1 R* and *IH1 R*

The CMU dictionary distinguishes among several variations for each vowel e.g. *AH1* and *AH0*. It also stores two possible pronunciations for the word *ear*.

The acoustic features for a phone significantly depend on its context. The preceeding and following phones strongly influence the sound of the middle phone.

triphone The triphone is a sequence of three phones and captures the context of single phone. As a result, the acoustic properties of triphones vary much less according to the context than phones. Let us note that certain combinations of prefixes have the same effect on the central phone, e.g. q and k have the same effect on i. In order to reduce the number of triphones for acoustic modelling, these triphones are clustered together.

HMM!s (HMM!s)

The **HMM!** is a very powerful statistical method for characterizing observed data samples of a discrete-time series with an unknown state. [?]. In case of speech recognition the hidden states typically represent monophones or triphones and we observe samples of the acoustic features.

transition probability

Hidden Markov Models have two type of parameters *transition probabilities among states* and *probabilistic distribution for generating an observation in given state.* These parameters need to be estimated in **AM!** training.[5]

The transition probability is the probability of changing state q to state u. Each transition is represented as an arc $e = qu$ between the states q and u, see **??**. This probability is typically represented as the weight w_e of arc e.

Importantly, an **HMM!** uses a self loop arc $e = uu$ for each state to model acoustic features which are generated several times from the same state u. As a result, an **HMM!** is able to model variable length of phones.

Gaussian HMM

The Markov model emits an observation during traversal over its arcs. The **HMM!** emits the observation stochastically based on the probabilistic distribution related to the visited state. In speech recognition, a multivariate Gaussian distribution is typically used to model observation probabilities of **HMM!** states. The Gaussian distribution models the probability of emitting acoustic features in given state. The parameters of the Gaussian distribution are estimated for each state individually. However, states are usually clustered during **AM!** training and the states within a cluster share the same parameters for the Gaussian distribution.

Training HMM!

Kaldi uses Viterbi training and **HTK!** uses **EM!** algorithms to train **HMM! AM!**s. Both toolkits model observation probabilities using multivariate Gaussian distributions with the dimension of the acoustic features a.

Typically, the transition probabilities are initialised with values uniformly distributed. The observation probabilities are usually initialized by multivariate Gaussian distributions with μ and Σ set to the global mean and global covariance matrix estimated on all training acoustic data.

EM

Let us describe how the **EM!** (**EM!**) algorithm operates for one pair of training data consisting of acoustic features a and corresponding text speech transcription t. We create an **HMM!** t', where each state represents one monophone.[6] The monophones are extracted from the transcription t using the pronunciation dictionary. In Figure **??** the utterance *how do you do* was expanded to a monophone **HMM!** model. Given the **HMM!** model for transcription t and acoustic features a the parameters of the model are

[5] Both kind of parameters are denoted together as θ in Equation **??**.

[6] We describe the identical training procedure for simplicity on monophones. State-of-the-art **AM!**s use triphones.

estimated. It should be obvious that only states representing phones in the transcription can be trained by the training pair (a, t). Consequently, one needs a lot of training data to robustly estimate the parameters of every state.

The **EM!** algorithm iterates over the following steps in order to update the parameters of transition and observation probabilities:

- The observation probabilities are computed using the **HMM!** t'.
- **E-step**: Based on the observation probabilities, the observations are align to the states of **HMM!** t'.
- **M-step**: Based on the alignment of observations to states, the parameters of t' are re-estimated.

Baum-Welch

The **E-step** finds a distribution for the alignment between the **HMM!** t' and transcription t using **MLE!** (**MLE!**)[?] and observation probabilities. **MLE!** takes into account all possible alignments and their probabilities to compute the resulting distribution. The Baum-Welch equation can be derived from the fact that the **MLE!** criterion is also used for finding the most probable distribution in the **M-step**.[?]

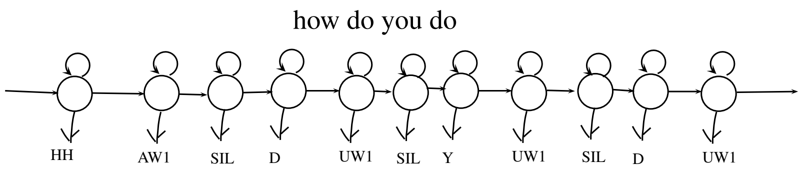


Figure 2.4: Markov monophone model for four words. Such an **HMM!** is constructed for monophone Viterbi training and reference transcriptions *how do you do*. The parameters of the **HMM!** model are updated according Equation **??**, **??** and **??**.

MLE! method

The **MLE!** is a general approach to setting statistical model parameters. It searches for the best parameters θ^* in order to maximize the likelihood function f for **IID!** (**IID!**) training data, as illustrated in Equation **??**. For **IID!** training data Equation **??** holds, describing the joint probability of the data.

$$f(x_1, x_2, x_3, \ldots, x_n|\theta) = f(x_1|\theta) * f(x_2|\theta) * \ldots * f(x_n|\theta) \tag{2.2}$$

The likelihood function can be derived from Equation **??** assuming fixed training data and the free parameter θ, as described in Equation **??**.

$$\mathcal{L}(\theta \,|\, x_1, \ldots, x_n) = \sum_{i=1}^{n} log(f(x_i|\theta)) \tag{2.3}$$

$$\theta^* = argmax_\theta \mathcal{L}(\theta \,|\, x_1, \ldots, x_n) \tag{2.4}$$

Viterbi training of acoustic models

On the other hand, the Kaldi toolkit applies the Viterbi criterion in assigning the acoustic observations to **HMM!** states. Viterbi training approximates the **EM!** algorithm by choosing the single best alignment and maximizing the posterior probability for the chosen alignment. Latest work suggests that Viterbi training is just as effective for continuous speech recognition as the Baum-Welch algorithm [?]. Moreover, Viterbi training needs much fewer computational resources.

We detail the Viterbi training since it is used in the Kaldi toolkit for acoustic model training, and a very similar algorithm is used for Viterbi decoding.

Given a set of training observations $O^r, 1 \leq r \leq R$ and an **HMM!** state sequence $1 < j < N$ the observation sequence is aligned to the state sequence via Viterbi alignment.[?] The best alignment T results from maximising Equation **??** for $1 < i < N$.

$$\phi_N(T) = max_i[\phi_i(T)a_{iN}] \tag{2.5}$$

The $\phi_i(o_t)$ from Equation **??** is computed recursively according to Equation **??**

$$\phi_i(o_t) = b_j(o_t) max \begin{cases} \phi_j(t-1)a_{jj} \\ \phi_{j-1}(t-1)a_{(j-1)j} \end{cases} \tag{2.6}$$

The initial conditions are $\phi_1(1) = 1$ and $\phi_j(1) = a_{1j}b_j(o_1)$, for $1 < j < N$. In our case the likelihoods are modeled as mixture Gaussian densities, so the output probability $b_j(o_t)$ is defined as in Equation **??**.

$$b_j(o_t) = \sum_{m=1}^{M_j} c_{jm} \mathcal{N}(o_t; \mu_{jm}, \Sigma_{jm}) \tag{2.7}$$

The M_j represents number of mixture components in state j, c_{jm} is the weight of m^{th} component and $\mathcal{N}(o_t; \mu_{jm}, \Sigma_{jm})$ is a multivariate Gausian with mean vector μ and covariance Σ.

Firstly, model parameters are updated based on the single-best alignment of individual observations to states and Gaussian components within states. Secondly, transition probabilities are estimated from the relative frequencies from Equation **??**, where A_{ij} denotes the number of transitions from state i to state j.

$$\hat{a}_{ij} = \frac{A_{ij}}{\sum_{k=2}^{N} A_{ik}} \tag{2.8}$$

The indicator function $\psi_{jm}^r(t)$ is used for updating the means and the covariance matrix from statistics. It returns one if o_t^r is associated with mixture component m of state j and zero otherwise. The mean vector and covariance matrix are updated according to Equations **??** and **??**.

$$\hat{\mu_{jm}} = \frac{\sum_{r=1}^{R} \sum_{t=1}^{T_r} \psi_{jm}^r(t) o_t^r}{\sum_{r=1}^{R} \sum_{t=1}^{T_r} \psi_{jm}^r(t)} \tag{2.9}$$

$$\hat{\Sigma_{jm}} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi_{jm}^{r}(t)(o_t^r - \hat{\mu_{jm}})(o_t^r - \hat{\mu_{jm}})'}{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi_{jm}^{r}(t)} \tag{2.10}$$

Finally, the mixture weights are computed based on the number of observations allocated to each component.[7]

$$c_{jm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi_{jm}^{r}(t)}{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\sum_{l=1}^{M}\psi_{jl}^{r}(t)} \tag{2.11}$$

generative training

To conclude, **AM!**s are trained using **MLE!** or Viterbi training, which approximates the theoretically optimal **MLE!** Baum-Welch training; however, in practice Viterbi training performs as well as **MLE!** modelling. Baum-Welch and Viterbi training aim at modelling the likelihood of a spoken utterance and perform so called generative training. However, discriminative methods, which re-estimate generative **AM!**s, perform better.

Discriminative training

discriminative training

Discriminative training uses an objective function and likelihoods from a generative model to discriminate – boost differences between high probable and low probable hypotheses. Discriminative training is typically initialised by an acoustic generative model from Baum-Welch or Viterbi training. Then, the likelihood from the generative model is boosted according an objective function, and the **AM!** is re-estimated. The following objective functions and accordingly named discriminative training methods are used in our training scripts:

- **MMI!**[?]
- **bMMI!**[?]
- **MPE!**[?]

For details on how the methods are initialized and their usage in Kaldi, see Chapter **??**.

2.1.3 Language modelling

A **LM!** effectively reduces and more importantly prioritises the **AM!** hypothesis. The probability of acoustic features given a transcription of words $P(a|w)$ (estimated by the **AM!**) is combined with the probability of the transcription of words $P(w)$ (estimated by the **LM!** for a given domain) in order to compute the posterior probability of the transcription given the features $P(w|a) = \frac{P(a|w)*P(w)}{P(a)}$.

The statistical **LM!** assigns a given word sequence its probability according Equation **??**. The most used n-gram **LM!** computes the probability of length k word sequence W according to Equation **??**.[?] The Markov assumption approximates the probability by assuming that only the most recent $n-1$ words are relevant when predicting next word. We call the number n the

***LM!** order*

[7]The Viterbi equations has the same notation as in [?].

order of the **LM!**.

$$P(W) = P(w_k, w_{k-1}, w_{k-2}, ..., w_1) \approx \prod_{i=1}^{k} P(w_i | w_{i-n+1}^{i-1}) \tag{2.12}$$

The probability $P(w_i|w_{i-n+1}^{i-1})$ for each word w_i is estimated using relative frequencies of the n-grams, (n-1)-grams, (n-2)-grams, . . . etc, on training data. The **MLE!** is used for estimating relative frequencies r according to Equation **??**.

$$r(w_i|w_{i-n+1}^{i-1}) = \frac{f(w_{i-n+1}^{i})}{f(w_{i-n+1}^{i-1})} \tag{2.13}$$

Equation **??** is intuitive but many valid and even reasonable utterances are missing or too few. Consequently, the numerator might be zero and the relative frequency may be undefined. This is known as sparse data problem. Smoothing techniques are often used to estimate the higher n-gram relative frequencies based on the lower frequencies.[**?**]. In principle, the predictive accuracy of the language model can be improved by increasing the order of the n-gram. However, doing so further exacerbates the sparse data problem.[**?**]

sparse data

LM smoothing

The **LM!** estimates the probability by counting the relative frequencies from a text corpus which is typically chosen according the targeted **ASR!** domain. For example, in the training scripts which are described in Chapter **??** we train the **LM!** only on text transcriptions from the training data using Witten-Bell smoothing.[**?**]

2.1.4 Speech decoding

The speech **HMM!** decoders find the most probable word sequences by searching phone sequences which correspond to the words. Phones are typically represented as triphones in an **AM!**.

Using a combination of **AM!** and **LM!** probabilities as described in Equation **??** does not produce the most accurate speech transcriptions. Typically a **LMW!** (**LMW!**) w_{lm} is used to improve speech recognition accuracy. It is tuned on the development set and balances the impact of the two models. Using the **LMW!** the best word sequence is found according to Equation **??**.

$$w^* = argmax_w\{P(w \mid a)\} = argmax_w\{P(a \mid w) * P(w)^{w_{lm}}\} \tag{2.14}$$

ASR! is a pattern recognition task as well as a search problem. In speech recognition, making a search decision is also referred to as decoding.[**?**]

For word recognition, the **AM!** limits the possible phone sequences to only words in the lexicon — the words in the training data. Word recognition is nowadays the most successful form of **ASR!**. An **HMM!** sequence represents the phone sequence which forms a word, as illustrated on Figures **??** and **??**. Words are connected via **HMM!** states which represent inter-word silence.

For isolated word recognition the **HMM!**s are evaluated for each word possibility. Using the forward algorithm for each **HMM!** h_w, we are able to compute the probability of every word w given the acoustic observations. The isolated word recognition becomes a simple recognition problem, where we select the most probable **HMM!** h^* from a finite set of word **HMM!**s.

Note that **HMM!** training is identical for continuous **ASR!** and isolated word recognition, but decoding is more complicated for continuous **ASR!** where we aim to decode word sequences.

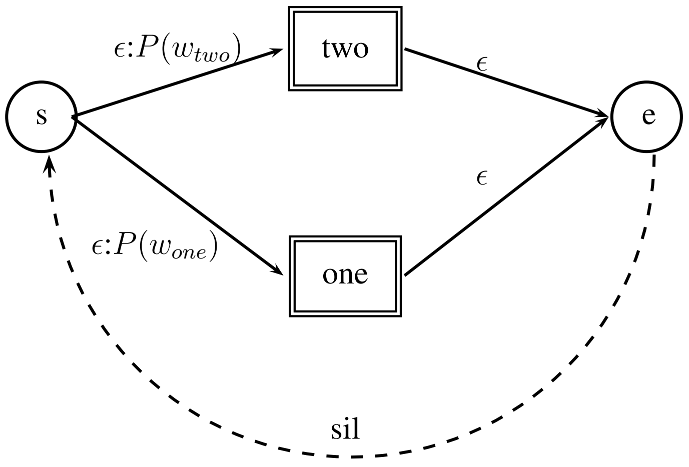


Figure 2.5: Diagram of how a **LM!** is combined with an **HMM!**s.

Let us introduce a simple example of continuous word **ASR!**. Imagine an **LM!** of order 1 modelling only two words - *one* and *two* each uniformly distributed[8]. We want to decode any possible sequence of words *one*, *two*. The ϵ transition at the end of each word's **HMM!**s to the final state e allows us to introduce an **HMM!** silence model which connects the final state e with the start state s. Consequently, the words are chained using a silence **HMM!** model as illustrated in Figure **??**. An expanded monophone **HMM!** for words *one* and *two* is shown in Figure **??**. Note that the **LM!** weight $P(w)$ can be stored on the ϵ transition at the beginning.

Even the simple **HMM!** network in Figure **??** can become a large search problem partially because the search space of words grows exponentially in the number of words in the utterance and partially because the word boundaries are unknown. Each word can begin at any moment and last with decreasing probability *ad infinitum*, so the search space explodes. In addition, higher order **LM!**s increase the decoding complexity even more.

One can see that the search space of a speech recognition problem is enormous and still has to be solved in very short time for real-time applications.

A natural choice for the one-best hypothesis is Viterbi beam search[?]. It uses a dynamic programming algorithm to compute the new best partial hypotheses for new audio data based on partial hypotheses from the previous step.

The Viterbi algorithm is a breadth-first search algorithm where a beam is used to limit the number of nodes which are expanded from the current

[8]If a **LM!** of order 1 assigns to every word equal probability, we say it has order 0

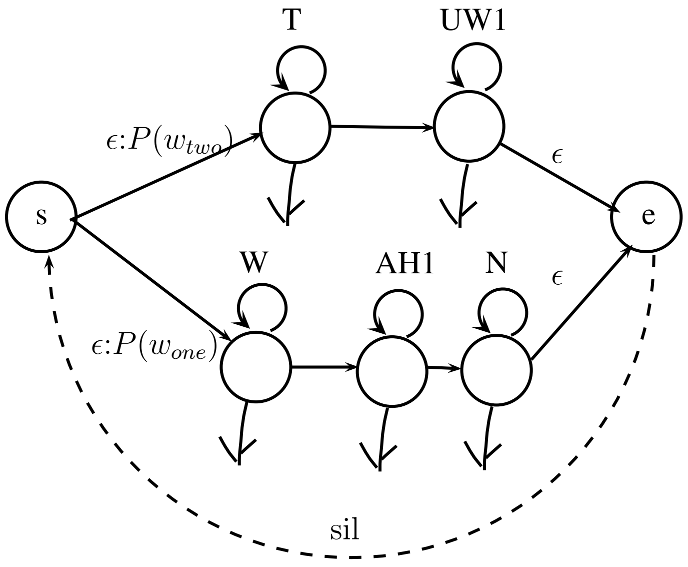


Figure 2.6: Expanded **HMM!**s for words *one* and *two*. The arrows at **HMM!** states illustrate that every observation of acoustic features can be generated according to the statistical distribution. Note that if a speaker says *two* a **HMM!** model with well trained parameters should output higher probability for the **HMM!** representing *two*. For longer speaker sequences e.g. *two one one two ...* the **HMM!**s are connected over the ϵ transitions, and a search is used for selecting the most probable sequence.

set of nodes to next iteration. We list a few alternatives of how to set up the beam for speech decoding.

- *Fixed beam* guarantees maximum size of memory footprint and fast decoding.

- *Relative one-best hypothesis comparison* effectively discards most of the improbable hypotheses if the one-best hypothesis is significantly better than the alternatives but it keeps a lot of alternatives if the one-best hypothesis is weak. The relative one-best hypothesis comparison naturally broadens the beam in uncertain regions, but does not guarantee any hard limits e.g., maximum number of nodes expanded.

- *Combination* of methods applies the strictest criteria on beam in each iteration.

Numerical stability

The hypotheses which are represented as the paths of states are typically rather long in the search graph and a lot of hypothesis are assigned tiny probabilities. In order to ensure numeric stability, probabilities are expressed in logarithmic scale.

In order to use the shortest distance measure to find the most probable path we use formula **??** derived from equation **??**. Both $C(a \mid w)$ and $C(w)$ are costs with ranges between zero and one, where a cost of one corresponds to zero probability $C(1) \cong P(0)$ and a cost of infinity corresponds to a probability of one $C(\infty) \cong P(1)$.

$$\begin{aligned} w^* &= argmin_w\{log(\frac{1}{C(a \mid w) * C(w)^{w_{lm}}})\} \\ &= argmin_w\{-log(C(a \mid w) * C(w)^{w_{lm}})\} \\ &= argmin_w\{-log(C(a \mid w)) - w_{lm} * log(C(w))\} \end{aligned} \tag{2.15}$$

Decoding formats

The one-best hypothesis outputs only single sequence of words despite the fact that other sequences of words are often almost as probable as the best hypothesis. Formats which are able to represent alternative hypothesis provide better results for further processing than one-best hypothesis because the alternatives may cover almost all probable hypotheses. We present n-best list and lattice formats, which both are able to represent alternative hypothesis.

n-best list

N-best list is an extension to the one-best hypothesis format. In n-best list is included apart from the most probable word sequence, also the second, third, . . . , n-th most probable hypothesis.

```
0.5 hi how are you
0.2 hi where are you
0.1 bey how are you
```

Figure 2.7: Example of 3-best list output with posterior probability for each path. N-best list in Kaldi can be easily extracted from lattices. Corresponding example lattice is in Figure **??**.

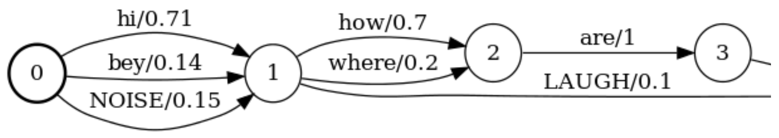


Figure 2.8: Word posterior lattice. Common parts of hypotheses are effectively represented. All outgoing arcs for each node sum to 1.0.

lattice

A lattice is a convenient type of **ASR!** output. It effectively represents the alternative hypotheses by sharing their common parts. Example of word lattice in Figure **??** shows word posterior lattice. Each hypothesis is represented as sequence of arcs from starts to final node. The words and their weights are associated with the arcs. The posterior probability of a hypothesis is computed as a product of posterior probabilities of each word in the hypothesis.

It is useful to capture how the quality of each hypothesis contrasts to its alternatives or even provide an absolute quality measure. Typically likelihood and posterior probability is associated with each word sequence to express its quality. The likelihood measure can be used only for relative comparison, whereas posterior probability is an normalised absolute measure. For some applications the likelihood measure is sufficient, other applications for example dialogue systems prefer posterior probabilities. Note that posterior probabilities for n-best lists typically do not sum to one and may need to be renormalised, because n-best list omits some hypothesis which are used to compute the posterior probability in lattices. See Figure **??** for such example for 3-best list.

2.1.5 Evaluating ASR! quality

WER The accuracy of a speech recognizer is typically measured using **WER!**. The **WER!** (**WER!**) measure is computed on one-best **ASR!** hypotheses and their human transcriptions. The **WER!** is computed as a minimum edit distance on words between the **ASR!** output and reference transcription. Following edit operations are used *substitution, deletion, insertion* to compute the minimum edit distance as illustrated in **??**. The effective implementation for computing WER uses dynamic programing and is not computationally intensive because **ASR!** hypotheses are typically quite short.

$$WER = 100 * \frac{min\_dist(decoded_{AM,LM}(a), t, edit\_operation = \{Subs, Del, Ins\})}{\#\ words\ in\ t} \tag{2.16}$$

reference Note that **WER!** is an error function so the ideal value is zero because for $WER = 0$ the one-best hypothesis *decoded*(a) and the reference transcription t are identical. The **WER!** value of 100 show that every single word is different between *decoded*(a) and reference t if the number of words in **ASR!** output and reference are equal. Despite the fact that **WER!** resembles percentage format, it can be bigger than 100. See the third example in Figure **??**.

```
decoded(a) = 'hi hi hi hi'
t='hi hi ha ha'
WER = 100 * ( 2 / 4) = 50

decoded(a) = 'how do you do'
t='how do you do''
WER = 100 * ( 0 / 4) = 0

decoded(a) = 'hi hi hi hi'
t='hello'
WER = 100 * ( 4 / 1) = 400
```

Figure 2.9: **WER!** captures the **ASR!** one-best hypotheses accuracy.

Note that the data used for evaluation should not be used in **AM!** training because we are evaluating the ability to decode unknown speech. We should also measure the **ASR!** quality on speech from a speaker who does not appear in speech training data because we usually want to decode speech of an unheard speaker.

Alternative measures

SER

The **SER!** measures how many decoded utterances $decoded(a)$ match exactly its reference t for all pairs (a, t) in test set T.

$$SER = \frac{\sum_{\{(a,t)\in T; decoded(a)=t\}} 1}{|T|} \tag{2.17}$$

oracle WER

If the n-best list or lattice is used the one-best hypothesis is extracted to compute **WER!** or **SER!** (**SER!**). On the other hand, we are using n-best lists or lattice because the one-best hypothesis might be wrong and the alternative hypothesis may be closer to the reference. In order to evaluate quality of alternative hypotheses one may use oracle **WER!** which reports the WER of the best hypotheses in n-best list or in lattice. The lattices with rich alternatives gain much lower oracle **WER!** than short n-best lists or even one-best hypotheses. The rich alternatives contain additional information and for example a dialogue system **SLU!** component may exploit the alternatives.

Measuring speed

In this thesis we are especially concerned about the speed of speech decoding because the implemented decoder is used in a real-time **SDS!**.

RTF!

A very natural measure of a speech decoding speed is **RTF!**, which expresses how much the recognizer decodes slower than the user speaks. We measure the **RTF!** (**RTF!**) for each recording as described in Equation **??**.

$$RTF = \frac{time(decode(a))}{length(a)} \tag{2.18}$$

For real-time decoding in a dialogue system we need smaller than one $RTF < 1.0$. In other words, the decoding of an utterance should take less time than a user needed for pronouncing the utterance. With $RTF < 1.0$ the hypothesis is decoded immediately after the user finishes the speech.

The decoding is performed while user is speaking, but extracting the **ASR!** hypothesis output is triggered at the very end of the speech. The users have to wait at least the time when the **ASR!** hypotheses is extracted.

latency

In real-time **SDS!** the critical measure is a delay how long the user has to wait for its answer. The latency measures the time between the end of the user speech and the time when a decoder returns the hypothesis, which is the most important speed measure for **ASR!** component in **SDS!**. Note if $RTF < 1.0$ then the latency corresponds to time of **ASR!** hypotheses extraction.

2.2 HTK!

The **HTK!** toolkit is a set of command line tools, sample scripts and library for training and decoding **HMM!** focused on speech recognition. With the toolkit are distributed two decoders *HVite* and *HDecode*, which are not designed for real-time applications.

Functionality of the core library can be accessed through command line executables. The command line programs are typically combined in training scripts to train acoustic and language models. In Figure **??** the acoustic models are labelled as "HMMs" and the language models in **HTK!** are represented in "Networks". The trained models are used in one of **HTK!** decoders e.g. *HVite* for decoding transcriptions, which can be evaluated using *HResults*.

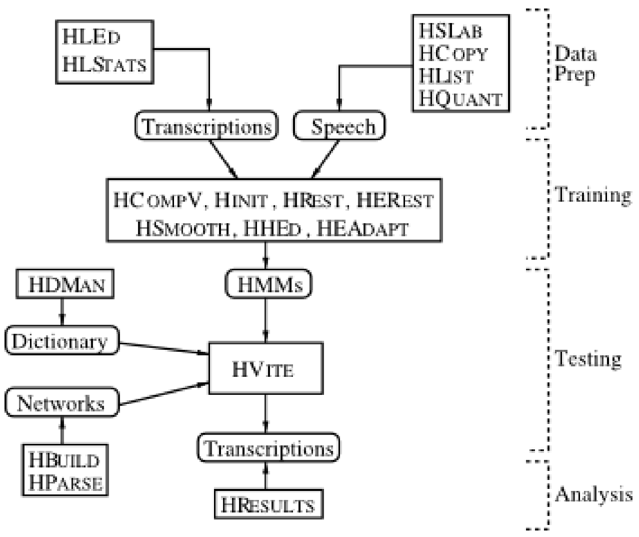


Figure 2.10: Figure 2.2 from HTK Book 3.4[?]

The **HTK!** library use Baum-Welch algorithm to train acoustic models. The *HVite* decoder uses token passing algorithm and Viterbi criterion.[?] Only unigram and bigram **LM!**s can be used with *HVite*. The termHDecode decoder can handle bigram or trigram language models.

Let us stress that we use high quality Bash and Perl scripts for training **HTK! AM!** from Vertanen improved by Matj Korvas.[?][?]

The **HTK!** toolkit is licensed under a special license[9]. The *HDecode* has very similar license condition but can be only used for research purposes.[10]

2.3 Julius decoding engine

Julius is a large vocabulary continuous speech decoder which can use **AM!** in **HTK!** format for decoding.[?] Julius is BSD licensed[11] and performs almost real-time decoding.

Julius is a two pass decoder. In the first pass, the decoding is performed using time synchronous beam search. The second pass re-ranks and further

[9] `http://htk.eng.cam.ac.uk/docs/license.shtml`

[10] You need to register even to see the license: `http://htk.eng.cam.ac.uk/prot-docs/hdecode_register.shtml`

[11] `http://www.linfo.org/bsdlicense.html`

prunes the extracted hypothesis from the pass one. Bigram **LM!** is used for the first pass and more complex trigram **LM!** is used for re-ranking.

Before the implementation of this thesis was finished the Alex **SDS!** team had been interested in Julius because its ability of real-time decoding and confusion network[12] output format.

The Alex team abandoned the Julius decoder for software issues e.g., crashes of the decoder. The crashes appeared during extracting confusion networks from Julius. In addition, the crashes were hard to detect because Julius used to run in a separate process.

2.4 Kaldi

Kaldi is a speech recognition toolkit consisting of a library, command line programs and scripts for acoustic modelling. Kaldi deploys several decoders for evaluation Kaldi **AM!**s. Kaldi uses Viterbi training for estimating **AM!**s. Only in special cases of speaker adaptive discriminative training the extended Baum-Welch algorithm is also used[**?**].

The architecture of the Kaldi toolkit could be separated to Kaldi library and training scripts. The scripts access the functionality of Kaldi library through command line programs. The C++ Kaldi library is based on the *OpenFST*[**?**] library and it uses optimized libraries for linear algebra such as BLAS and LAPACK. Related functionality is usually grouped in one namespace in C++ code, which corresponds to one directory on file system. The examples of the namespaces or directories can be seen in Figure **??**

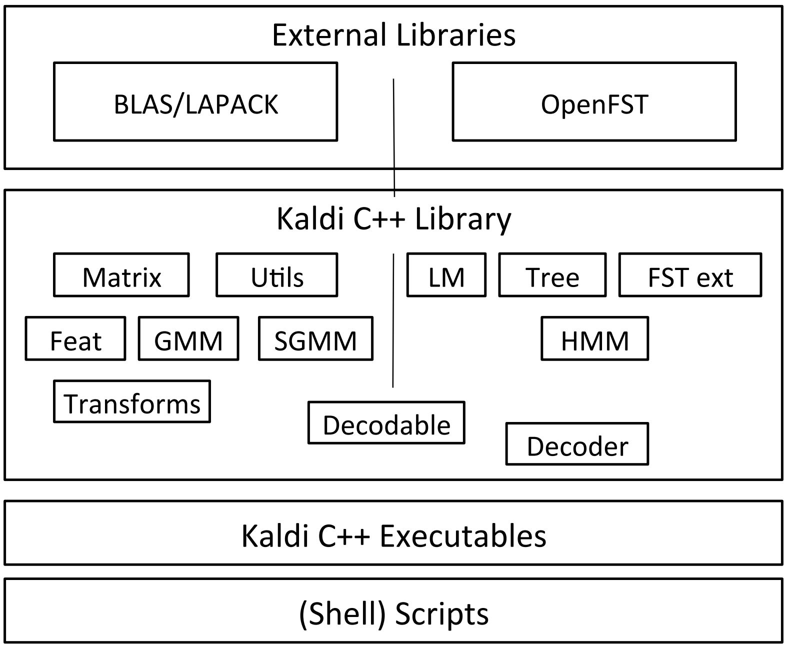


Figure 2.11: Kaldi toolkit architecture[**?**]

Kaldi uses executables which load its input from files and typically store

[12]A confusion network is approximation of a lattice described in Section **??**.

results again to files. Alternatively, the output of one Kaldi program can be feed into next command using system pipes. There are usually many alternatives for every speech recognition tasks as seen in list of executables below:

1. Speech parametrisation
    - *apply-mfcc*
    - *compute-mfcc-feats*
    - *compute-plp-feats*
    - ...
2. Feature transformation
    - *apply-cmvn*
    - *compute-cmvn-stats*
    - *acc-lda*
    - *fmpe-apply-transform*
    - ...
3. Decoders
    - *gmm-latgen-faster*
    - *gmm-latgen-faster-parallel*
    - *gmm-latgen-biglm-faster*
    - ...
4. Evaluation and utilities
    - compute-wer
    - show-alignments
    - ...

In addition, Kaldi provides very useful standardized scripts which wrap Kaldi executables or add new functionality. The scripts are located in *utils* and *steps* directories and are used in many training scripts recipes for different corpus data. In this thesis we created a new training recipe using the Kaldi infrastructure and Czech and English training corpus [?]. The recipe, the data and acoustic modelling scripts are described in Chapter **??**.

2.4.1 Finite State Transducers

The **FST!** framework and its implementation OpenFST determines the shape of the Kaldi data structures. Kaldi uses **FST!** (**FST!**) as underlaying representation for **LM!**, partially for **AM!**, lexicon and also for representing transformation between text, pronunciation and triphones.

The **FST!** framework provides well studied graph operations[?] which can be effectively used for acoustic modelling. Using the **FST!** framework the speech decoding task is expressed as a beam search in a graph, which is well studied problem.

The **FST!** graphs used for **AM!** model training and speech decoding can be constructed as sequence of standardized OpenFST operations.[**?**]. Decoding is performed on so called *decoding graph HCLG* which is constructed from simple **FST!** graphs as illustrated in Equation **??**.

$$HCLG = H \circ C \circ L \circ G \tag{2.19}$$

. The symbol $\circ$ represents an associative binary operation of composition on **FST!**s. We briefly explain the functionality of the transducers from Equation **??**:

1. G is an acceptor that encodes the grammar or language model.
2. L represents the lexicon. Its input symbols are phones. Its output symbols are words.
3. C represents the relationship between context-dependent phones on input and phones on output.
4. H contains the **HMM!** definitions, that take as input id number of **PDF!**s (**PDF!**s) and return context-dependent phones.

Following one liner illustrates how Kaldi decoding graph is created using standard **FST!** operations[13].[**?**]

$$HCLG = asl(min(rds(det(H'omin(det(Comin(det(LoG)))))))) \tag{2.20}$$

Semiring

Most of the operations operate on paths in the decoding graph. Path is a sequence of edges which have weights and an input and an output labels. Based on the weight type and weight path operations we distinguish several semirings.

Formally, a *semiring* $(\mathcal{K}, \oplus, \otimes, \bar{0}, \bar{1})$ is an algebraic structure on set $\mathcal{K}$ with operations $\oplus$ and $\otimes$. The binary operations multiplication $\oplus$ and addition $\otimes$ have identity element $\bar{0}$ respectively $\bar{1}$. The $(\mathcal{K}, \oplus)$ forms commutative monoid and $(\mathcal{K}, \otimes)$ forms just a monoid. The multiplication is left and right distributive over addition. Moreover, multiplication by $\bar{0}$ annihilates any member of $\mathcal{K}$ to *zero*. Table **??** shows useful semirings in OpenFST.

Name	$\mathcal{K}$	$\oplus$	$\otimes$	$\bar{0}$	$\bar{1}$
Real	$[0, \infty)$	$+$	$*$	0	1
Log	$(-\infty, \infty)$	$-log(e^{-x} + e^{-y})$	$+$	∞	0
Tropical	$(-\infty, \infty)$	min	$+$	∞	0

Table 2.1: Semirings used in speech recognition.[**?**]

[13]Kaldi tutorial on building *HCLG*: `http://kaldi.sourceforge.net/graph_recipe_test.html`

3. Acoustic model training

This chapter presents Kaldi acoustic modelling scripts for free Czech and English "*Vystadial*" data. The scripts were developed as part of this thesis, they are licensed under the Apache 2.0 license and are publicly available in the Kaldi repository[1]. The **AM!** (**AM!**) trained using these scripts can be used for both batch speech recognition with common Kaldi decoders and our *OnlineLatgenRecognizer*, which performs on-line decoding described in Chapter **??**.

The first Section **??** describes the used data. The chapter continues by presenting the **AM!**s training in Section **??**. Later, in Section **??** we evaluate trained **AM!**s and also compare them to generative **HTK! AM!**s which are trained using state of art **HTK!** scripts.

3.1 Vystadial acoustic data

The data were collected in Vystadial project[2], and they are released under the Creative Commons Share-alike (CC-BY-SA 3.0) license. The Czech[3]and English[4] data are available online in the Lindat repository[5][6].

The English acoustic data consists of recorded phone calls among humans and the **SDS!**, which was designed to provide the user with information on a suitable dining venue in the town. Most of the data was spoken in American English. The typical sentences recorded from users were queries for the dialogue system e.g.,

```
I NEED A CHINESE TAKE AWAY RESTAURANT IN THE CHEAP PRICE RANGE
I'M LOOKING FOR AN INTERNATIONAL RESTAURANT
I NEED TO FIND A PUB IT SHOULD ALLOW CHILDREN AND HAVE A TELEVISION
```

On the other hand, the Czech recordings were collected in three different ways[**?**]:

1. using a free Call Friend phone service
2. using the Repeat After Me speech data collecting process,
3. from the telephone interactions with the Alex **SDS!** in a **PTI!** (**PTI!**) domain.

In the Call Friend service native Czech speakers were invited to make free calls. In Repeat After Me process volunteers called a number where they were asked to repeat sentences synthesized by a **TTS!** (**TTS!**).

[1]http://sourceforge.net/p/kaldi/code/HEAD/tree/sandbox/oplatek2/egs/vystadial/
[2]http://ufal.mff.cuni.cz/grants/vystadial
[3]Czech data: `http://hdl.handle.net/11858/00-097C-0000-0023-4670-6`
[4]English data: `http://hdl.handle.net/11858/00-097C-0000-0023-4671-4`
[5]http://lindat.mff.cuni.cz/repository/
[6]A previous version of our training scripts is published with the data in the Lindat repository and described in work [**?**].

The user language differs significantly in dialogues with Alex system and the other two settings. The sentences in Alex's **PTI!** domain, as seen in the first paragraph, are shorter and contain noises. The speech is spontaneous and proper names are frequently used. On the other hand, the other two recording tasks, as seen in the second paragraph, have much broader vocabulary with less named entities, and the ideas are expressed in longer sentences.

PTI samples

```
A DAL
_NOISE_
JO DKUJU MOC TO JSEM CHTL VDT
ZE ZASTVKY DEJVICK
```

Call Friend samples

```
PRY S TYRANY A ZRDCI VEMI
UTRHNE SI KVT Z KYTICE A ODCHZ
DY TO JE HORNE ZVE
O LIBERALIZMU TEHDY NEBYLO EI
CO BY TAM S TEBOU DLALI
```

The **AM!**s for Czech are trained on acoustic data from all the three very different domains, because there is only two hours of in-domain data available in the Alex's public transport domain. The evaluation for Czech data in Section **??** is performed on a Vystadial test set combined from all three domains. The English **AM!**s are trained and tested on the data collected from the Venue domain using **SDS!**. The summary of audio sizes in training, development and test set are presented in Table **??**. Both Czech and English orthographic speech transcriptions were transcribed by humans.

dataset	audio[hour]	# sentences	# words
English			
training	41:30	47,463	178,110
development	01:45	2,000	7,376
test	01:46	2,000	7,772
Czech			
training	15:25	22,567	126,333
development	01:23	2,000	11,478
test	01:22	2,000	11,204

Table 3.1: Size of the data: length of the audio (hours:minutes), number of sentences (which is the same as the number of recordings), number of words in the transcriptions.[?]

3.2 Acoustic modelling scripts

We search for the best non-speaker adaptive **AM!**s in our scripts for **AM!** training. In this section, the explored methods and their settings are described, and the Section **??** presents the results for both Czech and English datasets. The Czech and English training scripts differ only in using a different phonetic dictionary, but otherwise the scripts remains exactly the same.

The recordings and their transcriptions from training dataset are used for acoustic modelling. The estimated **AM!**s are evaluated on the test set. The decoding of the test utterances is performed always with the same parameters, so that different **AM!**s can be compared. The Figure **??** lists all acoustic models trained in our scripts. An advanced **AM!** is always initiated by audio alignments (respectively acoustic features alignments) using a simpler **AM!**.

In paragraphs below, the organisation of acoustic model training is described. The used methods are listed in Figure **??** together with their hierarchy. The hierarchy shows that a more advanced method typically reuses initial values from previously trained simpler **AM!**.

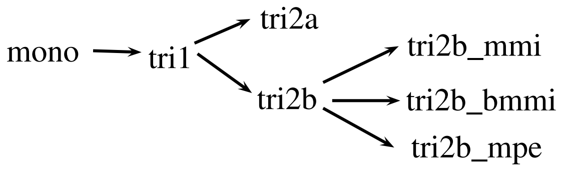
At first, a mono-phone model is trained from flat start using the MFCCs, Δ and $\Delta\Delta$ features. We force-align the feature vectors to HMM states using utterances' transcriptions. Secondly, we retrain the triphone **AM!** (*tri1a*). One branch of experiments finishes by training **MFCC!** $\Delta + \Delta\Delta$ triphone **AM!** (*tri2a*).

On the other hand, the second branch instead of $\Delta + \Delta\Delta$ transformation uses **LDA!**+**MLLT!** to train **AM!** (*tri2b*). Using the **AM!** *tri2b* three **AM!**s are discriminatively trained using the following objective functions:

1. **MMI!**[?][7]. The model *tri2b_mmi* is trained in four loops.

2. **bMMI!**[?]. The model *tri2b_bmmi* is trained in four loops with parameter 0.05.

3. **MPE!**[?]. The model *tri2b_mpe* is also retrained in four loops.



Training method name	Script shortcut
Monophone	mono
Triphone	tri1
$\Delta + \Delta\Delta$	tri2a
LDA!+**MLLT!**	tri2b
LDA!+**MLLT!**+**MMI!**	tri2b_mmi
LDA!+**MLLT!**+**bMMI!**	tri2b_bmmi
MPE!	tri2b_mpe

Figure 3.1: Training partial order among **AM!** in our training scripts

The acoustic models *mono*, *tri1*, *tri2a* and *tri2b* are trained generatively. The models *tri2b_mmi*, *tri2b_bmmi* and *tri2b_mpe* are trained discriminatively in four iterations. The discriminative models yield better results than generative models if enough data is available. See Figure **??** for evidence.

[7]Note the **MMI!** (**MMI!**) function is implemented as **bMMI!** with boosted parameter set to 0.

over-fitting The discriminative models from may significantly over-fit to the training data. Discriminative training uses a unigram **LM!** estimated on training dataset in order to compute their objective function, each iteration adapts more to the training data. We used four iterations for discriminative models training, and we have not experienced such behaviour.

Setup for feature transformations

We explored not only **AM!** training methods but also experimented with two feature transformation techniques. First, the $\Delta + \Delta\Delta$ triples the number of 13 **MFCC!** features by computing the first and the second derivatives from **MFCC!** coefficients. The computation of **MFCC!** coefficients with the derivatives produce 39 features per frame in total.

Second, the combination of **LDA!** and **MLLT!** is computed from 9 spliced frames consisting of 13 **MFCC!** features. The default context window of 9 frames takes current frame, four frames from the left context and four frames from the right context. The **LDA!** and **MLLT!** feature transformation gains substantial improvement over $\Delta + \Delta\Delta$ transformation. See Figure **??**.

Decoding setup

We use the trained **AM!**s described above for decoding the utterances from the test dataset. For each trained **AM!** we use the same speech parametrisation and feature transformation method as was used for the given **AM!** at training time. We experiment with all trained **AM!**s with both zerogram and bigram **LM!**.

The default bigram and zerogram **LM!**s for are built from orthographic transcriptions. The bigram **LM!** is estimated from the training data transcriptions. Consequently, in a test set appear unknown words, so called **OOV!**. The zerogram is extracted from a test set transcriptions. The zerogram is a list of words with probabilities uniformly distributed, so it helps decoding just by limiting the vocabulary size. The bigram **LM!** contains 17433 unigrams and 79333 bigrams for Czech and 936 unigrams and 5521 bigrams for English. The zerogram **LM!** is limited to 2944 words for Czech and to 302 words for English.

OOV

zerogram

The speech recognition parameters are set to default values; the exceptions are decoding parameters: *beam=12.0, lattice-beam=6.0, max-active-states=14000* and **LMW!**. The **LMW!** parameter sets the weight of a **LM!**, i.e., it regulates how much the **LM!** is used to help **AM!** in decoding. The **LMW!** value is estimated on the development set and the best value is used for decoding on the test dataset. The details about *beam=12.0, lattice-beam=6.0 and max-active-states=14000* can be found in Subsection **??**. Section **??** evaluates the **ASR!** performance for this parameters.

The *gmm-latgen-faster* decoder is used for the evaluation on testing data. It generates a word level lattice for each utterance and the one-best hypothesis is extracted from the decoded lattice and evaluated by **WER!** and **SER!** metrics against the reference transcription.

Note, that we are able to exactly reproduce the results of *gmm-latgen-faster* decoder with our *OnlineLatgenRecogniser*. The *gmm-latgen-faster* was used for evaluation in the scripts, so the Kaldi users do not have to install our extension.

3.3 Evaluation

The experiments focus on comparing the quality of ASR hypothesis measured by **WER!** on **AM!**s trained by different methods. We are not interested in absolute numbers since we model the language using a weak **LM!** focusing on the acoustic modelling. By training only simple bigram **LM!** we let the **AM!** influence the recognition quality more significantly. The same motivation lead us to use zerogram **LM!** which just limits vocabulary in the decoding task. Consequently, the best words are chosen among all hypotheses only by acoustic similarity.

We concentrate o
am_1, am_2 are traine
gains lower **WER!** th
experiment with a ric
AM! ($wer_1^{rich} < wer$
First, we show ho
by **WER!**. Second, t
section **??**, the best]
by well-written **HTK**
Vystadial dataset[?].

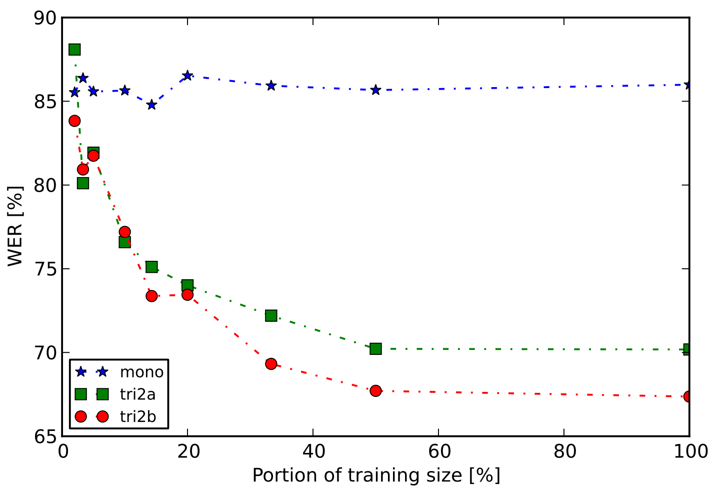


Figure 3.2: The figure displays improving performance of Czech generative **AM!**s based on growing size of training data for acoustic modelling. The zerogram **LM!** allows to evaluate only acoustic modelling, but causes a high **WER!**.

The Figure **??** describes how the amount of acoustic data influences the **WER!**. We illustrate that even with small datasets like Vystadial the high quality **AM!** can be trained. The WER decreases significantly

if new data are added to small dataset, but only small **WER!** reduction is achieved when the last 50% of data is added. One can also see that the $\Delta + \Delta\Delta$ feature transformation is clearly outperformed on full data by **LDA!**+**MLLT!** setup. Note also that the monophone **AM!** is typically used for the initialisation of triphone models and requires small portion of data to reach its limit. The WER is rather high due to the use of zerogram **LM!**. We evaluate only generative **LM!**s since we would have to have a fixed LM for discriminative m
build one.

It may seem that
discriminative trainin
transcribed data a b
shows the effect of in-
The **AM!** *tri2b_bmn*
were performed with
size. Note that this
was run on different
built also from that i

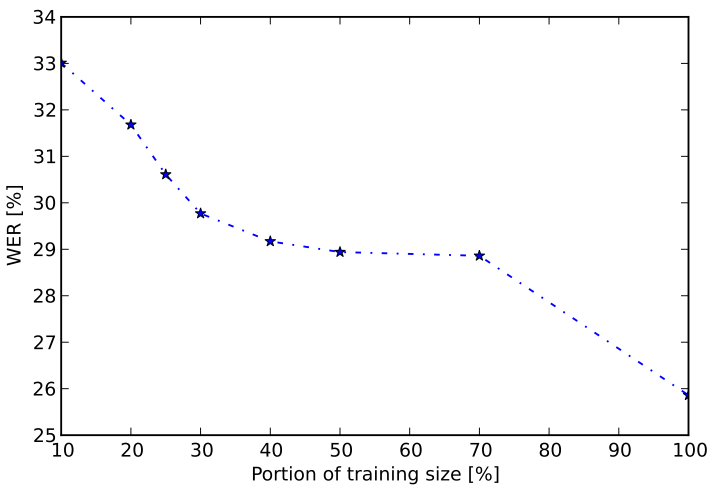


Figure 3.3: Influence of in-domain text size of **LM!** on speech recognition quality. The **AM!** *tri2b_bmmi* and parameters are fixed and only **LM!** training size varies.

To conclude we are able to train reasonable **AM!** with relatively small dataset such as Vystadial. On the other hand, additional data should improve speech recognition accuracy because

- the language domain changes in time and the new data reflect the differences,
- more data still may improve the best discriminatively trained **AM!**,

[8]Ondej Duek used our scripts developed for Alex dialogue system for the **PTI!** domain for the experiment.

- and last but not least the speech recogniser is more robust to new speakers.

3.3.1 Results

In this section we present the results of different acoustic training methods and we choose the best non-speaker adaptive setup. The Table **??** presents **AM!**s results.

language/method	**zerogram**	**bigram**
Czech		
tri $\Delta + \Delta\Delta$	70.7	56.6
tri LDA+MLLT	68.2	53.9
tri LDA+MLLT+MMI	65.3	49.5
tri LDA+MLLT+bMMI	65.3	49.3
tri LDA+MLLT+MPE	63.8	49.2
English		
tri $\Delta + \Delta\Delta$	35.7	16.2
tri LDA+MLLT	33.28	15.8
tri LDA+MLLT+MMI	25.01	10.4
tri LDA+MLLT+bMMI	23.9	10.2
tri LDA+MLLT+MPE	22.41	11.1

Table 3.2: Word error rates for zerogram and bigram LM for different training triphone methods. The 'tri $\Delta+\Delta\Delta$' row shows results for a generative model which is comparable to the model trained using the HTK scripts.

The complexity of the Czech data is clearly much larger than the complexity of the English data. The high **WER!** on the Czech dataset may be explained be following reasons:

- The mix of a very different domain and recording conditions is difficult to model by both **AM!** and **LM!**.
- The *Call Friend* and *Repeat After Me* collections task have a really broad domain which affect language modelling.
- The flective languages such as Czech have larger vocabulary and higher **OOV!**s (**OOV!**s) since one word may have several inflected forms.

The **WER!** on the Vystadial English data is lower than 20% for discriminative methods, which is reasonable, given the broad domain.

The discriminative training methods clearly outperformed the generative **AM!**s, and also the **LDA!**+**MLLT!** is more effective feature transformation than using $\Delta + \Delta\Delta$ features. On the other hand, there are subtle differences among the three discriminatively trained **AM!** in terms of performance. As a result, we choose **AM!** (*tri2b_bmmi*) discriminatively trained by **bMMI!** (**bMMI!**) with **MFCC!**, and **LDA!**+**MLLT!** preprocessing because informal experiments shows that decoding with **MPE!** (**MPE!**) **AM!** is slightly more computationally demanding compared with **bMMI!** **AM!**.

3.3.2 Kaldi and previous HTK! results comparison

We compared Kaldi and HTK on the Vystadial Czech and English datasets, to confirm that the Kaldi toolkit is a good alternative for **HTK!**. In addition, by using state of the art **HTK!** scripts we saw that the complexity of the Vystadial datasets is higher than in other datasets trained with the HTK scripts.[9]

We present results for triphone **AM!** estimated using Baum-Welch iterative training on zerogram and bigram **LM!**s. The *HVite* **HTK!** decoder was used to perform the decoding with the same **LM!**s as used in Kaldi scripts. The training procedure is further described in work [**?**].

language/method	**zerogram**	**bigram**
Czech		
tri $\Delta + \Delta\Delta$	64.5	60.4
English		
tri $\Delta + \Delta\Delta$	50.0	17.5

Table 3.3: HTK results: Word error rates on test set are obtained by both a zerogram and a bigram LM. The **AM!**s can be compared with the basic *tri* $\Delta + \Delta\Delta$ Kaldi setup in Table **??**.

The results suggest that Kaldi achieves similar **WER!** compared to **HTK!** when using standard generative training methods and bigram **LM!**s. Furthermore, one may obtain a substantial reduction in **WER!** by using more advanced discriminative training methods.

The experiment using **MFCC!**, **LDA!** & **MLLT!** and **bMMI!** discriminative training is a state of the art set up for speaker independent speech recognition [**?**] and outperforms **HTK!** models.

Furthermore, in Chapter **??**, we evaluate the trained Czech **AM!**s on **PTI!** domain on a different test set with a fine tuned **LM!** and the best **AM!** from list in Figure **??**. The best **AM!** is selected based on the results in Section **??**.

[9]Unfortunately, the dataset is not publicly available.

4. Real time recogniser

This chapter presents the *OnlineLatgenRecogniser*, the new on-line Kaldi recogniser which can be used in real-time applications. Section **??** describes the implementation of the *OnlineLatgenRecogniser*. Next Section **??** introduces *PyOnlineLatgenRecogniser*, a Python extension of C++ *OnlineLatgenRecogniser*. Finally, Section **??** summarizes properties of the new implemented Kaldi recogniser.

We implemented a lightweight modification of the *LatticeFasterDecoder* from the Kaldi toolkit, improved on-line speech parametrisation and feature processing in order to create an *OnlineLatgenRecogniser*. The Kaldi *OnlineLatgenRecogniser* implements on-line interface which allows incremental speech processing, and it is able to process the incoming speech in small chunks incrementally. As a result, the real-time speech decoding can be performed while a user is speaking and the ASR output is obtained with a minimal latency.

The implementation of the recogniser was motivated by the lack of an on-line recognition support in Kaldi toolkit. Therefore, the toolkit decoders could not be used in applications such as spoken dialogue systems. Although Kaldi included an on-line recognition application; hard-wired timeout exceptions, audio source fixed to a sound card, and a specialised 1-best decoder limit its use only to demonstration of Kaldi recognition capabilities.

Our on-line recogniser uses acoustic models trained using the state-of-the-art techniques, such as Linear Discriminant Analysis (LDA), Maximum Likelihood Linear Transform (MLLT), Boosted Maximum Mutual Information (BMMI) and Minimum Phone Error (MPE). It produces word posterior lattices which can be easily converted into high quality n-best lists.

The recogniser's speed and latency can be effectively controlled off-line by optimising a language model. At runtime the speed of decoding is controlled by a beam threshold. The latency depends on the amount of time spent on word posterior lattice extraction from the recogniser, which can be regulated by a level of approximations used during the word lattice creation.

4.1 OnlineLatgenRecogniser

The standard Kaldi executables which implements speech parametrisation, feature transformations and decoder are using a batch file interface. Each executable loads the input from a file, processes a whole utterance and saves its output to another file. However, in real-time applications one would like to take advantage of the fact that an acoustic signal of an utterance is recorded in small chunks and can be processed incrementally.

We reimplemented speech parameterisation and feature transformations in order to fit on-line OnlineLatgenRecogniser's interface, which can process audio features incrementally. In addition, we subclassed *LatticeFasterDecoder* and reorganized its original batch interface, so that it supports on-line decoding. Such implementation almost eliminates latency of a recogniser since almost all of the decoding can be performed while the user is still

LatticeFasterDecoder

speaking.

First, we present the public on-line interface of *OnlineLatgenRecogniser* and in next subsections we introduce its components. The Subsection **??** describes the decoder, the core component. Subsection **??** introduces on-line speech parametrisation and feature transformations and the Subsection **??** discusses word posterior lattice extraction.

4.1.1 *OnlineLatgenRecogniser* interface

The *OnlineLatgenRecogniser* makes use of the incremental speech pre-processing and modified *LatticeFasterDecoder* in order to provide the following speech recognition interface:

- *AudioIn* – queueing new audio for pre-processing,
- *Decode* – decoding a fixed number of audio frames,
- *PruneFinal* – preparing internal data structures for lattice extraction,
- *GetLattice* – extracting a word posterior lattice and returning log likelihood of processed audio,
- *GetBestPath* – extracting a one best word sequence,
- *Reset* – preparing the recogniser for a new utterance,

The interface is influenced by the decoder interface and the preprocessing of the utterance is completely hidden for the user of *OnlineLatgenRecogniser*. The *AudioIn* is the only method which is not related to the decoder functionality.

The `C++` example in Listing **??** shows a typical use of *OnlineLatgenRecogniser*. When audio data becomes available, it is queued into the recogniser's buffer (line 11) and immediately decoded (lines 12-14). If the audio data is supplied in sufficiently small chunks, the decoding of queued data is finished before new data arrives. When the recognition is finished, the recogniser prepares for lattice extraction (line 16). Line 20 shows how to obtain word posterior lattice as an OpenFST object. The auxiliary *getAudio()* function represents a separate process supplying speech data. Please note that the recogniser's latency is mainly determined by the time spent in the *GetLattice* function since the whole loop is processed while the user is speaking.

We designed the interface with following criteria in mind:

- Passing the audio in the recogniser should accept any size of audio input.
- Decoding should return a number of actually decoded frames. The recogniser may decode less frames than requested if not enough audio is available.
- Decoding should be called frequently on small chunks, which guaranties quick response times of the *Decode* method.

Listing 4.1: Example of the decoder usage

```
OnlineLatgenRecogniser rec;
rec.Setup(...);

size_t decoded_now = 0;
size_t max_decode = 10;
char *audio_array = NULL;

while (recognitionOn())
{
  size_t audio_len = getAudio(audio_array);
  rec.AudioIn(audio_array, audio_len);
  do {
    decoded_now = rec.Decode(max_decode);
  } while(decoded_now > 0);
}
rec.PruneFinal();

double tot_lik;
fst::VectorFst<fst::LogArc> word_post_lat;
rec.GetLattice(&word_post_lat, &tot_lik);

rec.Reset();
```

Consequently, *OnlineLatgenRecogniser* does not block a process to either load audio or decode an utterance. The loading of audio and the decoding can be easily alternated back and forth. Obviously, the decoding of single utterance can be separated into number of parts and other tasks can be run in a single process with speech recognition in order to allow an application to stay responsive. We are able to decode the utterance while the user speaks.

On the other hand, extracting the word posterior lattice may block the process since it is very computationally demanding. It lasts several tens of milliseconds. However, it is called only at the end of each utterance. Extracting one best word sequence is much faster and can be called at any time.

4.1.2 *OnlLatticeFasterDecoder*

In the *OnlLatticeFasterDecoder* implementation we reorganised the code of base class *LatticeFasterDecoder*. The *LatticeFasterDecoder::Decode* function runs a beam search from frame 0 to the end of each utterance. In addition, a pruning is triggered periodically in the function. In *OnlineLatgenRecogniser*, we split the LatticeFasterDecoder::Decode method which performed several tasks into three methods in order to control beam search:

- *Decode* – decoding a fixed number of audio frames instead of decoding whole utterance, pruning is triggered periodically,
- *PruneFinal* – run final pruning and so prepare the internal data structures for lattice extraction,
- *Reset* – preparing the recogniser for a new utterance.

In the *PruneFinal* function, which is called at the end of an utterance, the states are pruned by beam search with the knowledge that no further search will be performed, so more states can be safely discarded.

The decoding is performed on request by calling the *Decode* method with a parameter (int max_frames) which limits the number of decoded frames. It returns the number of frames which were actually decoded, which is always smaller or equal to *max_frames* value. The *OnlLatticeFasterDecoder::Decode* method performs decoding frame by frame using the Viterbi beam search. The speed of the Viterbi search is highly predictable for fixed settings of the recogniser. As a result, the *max_frames* parameter effectively limits the amount of time in the *Decode* method. Repeated calls of *Decode* with small values of *max_frames* keep the recognition responsive as implemented in Listing **??**.

The **ASR!** output is extracted by the original methods of *LatticeFasterDecoder*:

- *GetRawLattice* returns state-level lattice,
- *GetLattice* extracts from state-level lattice word lattice which is returned,
- *GetBestPath* returns just one-best path hypothesis.

The state-level lattice, which is returned from the *GetRawLattice* method, can be understood as lattice on triphone level. In the state-level lattice, a single word hypothesis is typically can be obtained from multiple state-level hypotheses due to different word alignments, i.e., the same words sequences were pronounced with different timing.

The decoding of *LatticeFasterDecoder* as well as lattice extraction can be controlled by several parameters. We mention the most important parameters which affect both speed and the **ASR!** output quality. The parameters either increase speed and decrease **ASR!** output quality or vice versa.

decoding parameters The *beam* and *max-active-states* parameters directly affect the speed of decoding. The *beam* parameter affects the speed of all utterances, whereas the *max-active-states* parameter plays its role for noisy utterances with uncertainty in beam search. In fact, the *max-active-states* is a threshold for worst case scenarios. The *lattice-beam* influences speed of lattice extraction.

The properties of the parameters and its relationship to **ASR!** output quality is described in detail in Section **??** where we evaluate the recogniser.

4.1.3 On-line feature pre-processing

This section describes audio signal buffering, **MFCC!** feature extraction and feature transformation. The resulting acoustic features are then used with an **AM!** in *OnlLatticeFasterDecoder* to obtain likelihood of each state explored by Viterbi search. *OnlineLatgenRecogniser* only uses the likelihood to run Viterbi search. The likelihood itself is extracted from **AM!** based on the acoustic features by *DecodableInterface*.

When a decoder is asked to perform decoding it needs to estimate likelihood for the states which should be explored, and so it requests the *DecodableInterface*. We implemented on-line version of *DecodableInterface* which let the decoder ask for likelihoods of new acoustic features frame by frame.

The decoder, the pre-processing pipeline and the data flow between the components are illustrated in Figure **??**. We briefly describe one step of Viterbi search:

- Audio is extracted from a buffer.
- The **MFCC!** features are computed on overlapping audio window. The new audio is used for shifting the audio window.
- Applying feature transformation on top of **MFCC!** features.
  - Δ+$\Delta\Delta$ requires at least two previous frames, if available the acoustic features a are returned.
  - The LDA+$MLLT$ is computed using context, which by default is set to four previous and four future frames. If context is available, the acoustic features a are returned.

  Note, that the $LDA + MLLT$ and the $\Delta + \Delta\Delta$ transformations are complementary.
- The *OnlDecodableDiagGmmScaled* queries the **AM!** for the likelihood of acoustic features and given state.
- The decoder itself performs the search in state level space having the probabilities from the *Decodable* interface.

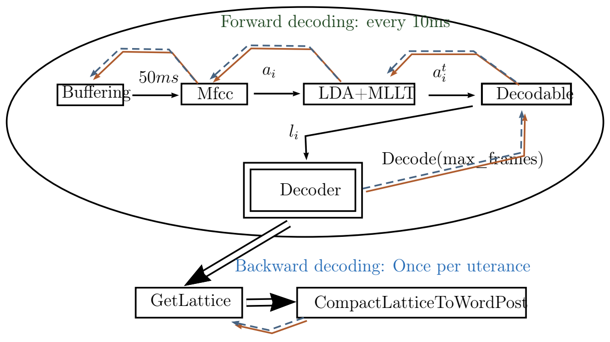


Figure 4.1: Components for on-line decoding

Each step in Figure **??** is implemented as a separate C++ class. *OnlineLatticeRecogniser* instantiate each class during the setup.

The on-line implementation *OnlDecodableDiagGmmScaled* of *DecodableInterface* easily handles missing audio data. If the likelihood for new frame is requested and the *OnlDecodableDiagGmmScaled* cannot obtain new acoustic features it returns default empty value. Then, the decoder's method *Decode(int max_frames)* returns zero indicating that no frames were decoded. Similarly, the speech parametrisation and feature transformations components returns their default empty value if they cannot compute its output. Consequently, *OnlineLatgenRecogniser* either decodes few frames or immediately returns zero indicating that no frames were decoded.

We have not experimented speech parametrisation settings. We used the recommended values, which are tested in tens of Kaldi recipes. The list the most important parameters:

- The frame width (set to 25 ms),
- the frame shift (set to 10 ms),
- and the frame splicing used for **LDA!**+**MLLT!** (nine frames are spliced).

4.1.4 Post-processing the lattice

The *OnlineLatgenRecogniser* not only extracts word lattice using *OnlLattice-FasterDecoder::GetLattice* function, but also computes posterior probabilities for the word lattice. The *OnlLatticeFasterDecoder* returns word lattice with alignments in form of *CompactLattice*. The *CompactLattice* determinised at state level still may contain multiple paths for each word sequence encoded in the lattice. The *CompactLattice* distinguishes each path not only according to the word labels on the path, but also according to the alignments. In order to obtain only the word lattice, we discard the alignments.

Compact-Lattice The steps of converting *CompactLattice* to word posterior lattice are listed below. For the implementation details see Listing **??**:

- Joining multiple word sequences which differer in word alignments is performed in two steps:
  - Discarding the alignments from *CompactLattice*.
  - Converting the lattice to its minimal lattice representation with no alternatives for one word hypothesis.
- The computing of the posterior probabilities through a standard forward-backward algorithm, which is implemented in two steps:
  - Computing α and β data structures for which a Kaldi implementation is reused.
  - Updating the lattice weights from likelihood to posterior probabilities based on α and β, which we implemented in the *MovePostToArcs* function.

The word posterior probability is converted from the likelihood of the words in word lattice. The word lattice obviously contains alternatives which were explored by the beam search during decoding the utterance. Consequently, the posterior probability is an approximation because the very low probable alternatives discarded by beam search are not considered. On the other hand, the discarded alternatives are so improbable so they almost do not influence the posterior probability.

Presumably, the word posterior values are more impacted by inaccurate likelihood values taken from the **AM!**. Generative models are improved so the likelihood match the reality as much as possible. On the other hand, the discriminative **AM!** models deliberately favour the most probable hypothesis by boosting the likelihood of the most probable hypothesis. As

Listing 4.2: Converting *CompactLattice* to posterior word lattice

```
double CompactLatticeToWordsPost(CompactLattice &clat,
                                  fst::VectorFst<fst::LogArc> *pst) {
  {
    Lattice lat;
    fst::VectorFst<fst::StdArc> t_std;
    RemoveAlignmentsFromCompactLattice(&clat); // remove the alignments
    ConvertLattice(clat, &lat); // convert to non-compact form.. no new
        ↪states
    ConvertLattice(lat, &t_std); // this adds up the (lm,acoustic) costs
    fst::Cast(t_std, pst);  // reinterpret the inner implementations
  }
  fst::Project(pst, fst::PROJECT_OUTPUT);
  fst::Minimize(pst);
  fst::ArcMap(pst, fst::SuperFinalMapper<fst::LogArc>());
  fst::TopSort(pst);
  std::vector<double> alpha, beta;
  double tot_lik = ComputeLatticeAlphasAndBetas(*pst, &alpha, &beta);
  MovePostToArcs(pst, alpha, beta);
  return tot_lik;
}
```

a result, the word posterior probability for the best hypothesis is artificially boosted. At the moment, we do not calibrate the word posterior probabilities in extracted lattices.

4.2 PyOnlineLatgenRecogniser

We also developed a Python extension, *PyOnlineLatgenRecogniser*, exporting the *OnlineLatgenRecogniser* C++ interface to Python. It can be used as an example of bringing Kaldi's on-line speech recognition functionality to higher-level programming languages. We extended also PyFST library[?], which interfaces OpenFST C++ template library into Python because we need to process further the OpenFST lattices produced by *PyOnlineLatgenRecogniser* in Python. Consequently, the recogniser as well as its input and output can be seamlessly used both from C++ and Python.

PyFST

PyOnlineLatgenRecogniser is a thin wrapper around *OnlineLatgenRecogniser* implemented using Cython[?]. The Cython compiler is well known for generating fast code when interfacing Python and C++ and the wrapper causes no measurable overhead.

We implemented conversion of the word posterior lattices to an n-best list. The implementation is efficient since the OpenFST shortest path algorithm is used on small lattices.

The minimalistic Python example in Listing **??** shows usage of the *PyOnlineLatgenRecogniser* and the decoding of a single utterance.

The audio is passed to the recogniser in small chunks (line 4), so the decoding (line 5 and 8) can be performed while the user is speaking. When no more audio data is available a likelihood and a word posterior lattice is extracted from the recogniser(line 10).

Note that *PyOnlineLatgenRecogniser* and *OnlineLatgenRecogniser* are initialised by string vector of arguments in command line format. The parameters are parsed using Kaldi's command line parser and options affect behaviour speech parametrisation, feature transformations and the *OnlLat-*

Listing 4.3: Fully functional example of the *PyOnlineLatgenRecogniser* interface

```
d = PyOnlineLatgenRecogniser()
d.setup(argv)
while audio_to_process():
    d.audio_in(get_raw_pcm_audio())
    dec_t = d.decode(max_frames=10)
    while dec_t > 0:
        decoded_frames += dec_t
        dec_t = d.decode(max_frames=10)
d.prune_final()
lik, lat = d.get_lattice()
```

ticeFasterDecoder. In addition, exactly the same parameters can be parsed by standard Kaldi utilities. We created demos[1] which use the same parameters for speech recognition using:

- standard Kaldi executables and scripts
- *PyOnlineLatgenRecogniser*
- *OnlineLatgenRecogniser*

The alternatives produce exactly the same results.

4.3 Summary

The *OnlLatticeFasterDecoder* performs the on-line speech recognition. We suggest exploiting the *OnlineLatgenRecogniser* and decoding utterances in small chunks and pass the audio to the recogniser immediately as it is available. The speech recognition parameters are initialized with reasonable default values and the parameters are the same as used in Kaldi executables. As a result, one can use the parameters from any Kaldi recipe to obtain the exactly same high quality results in the on-line speech recognition setting.

The implemented minimal on-line interface which supports **MFCC!** speech parametrisation, $\Delta - \Delta\Delta$ feature transformation or **LDA!**+**MLLT!** and both generative training and discriminative training using **bMMI!** and **MPE!**. The **MFCC!**, **LDA!**+**MLLT!** and **bMMI!** is one of the best setup for the speaker independent speech recognition. To conclude, we reimplemented Kaldi batch speech recognition, so that it can perform on-line real-time speech recognition and still maintain its high quality. The next Chapter **??** evaluates in detail the recognisers' real-time performance in the Alex Dialogue Systems Framework.

[1] `https://github.com/UFAL-DSG/pykaldi/tree/master/egs/vystadial/online_demo`

5. Kaldi ASR! in Alex SDS!

This chapter discuss the details of deploying *OnlineLatgenRecogniser* into Alex dialogue system. The *OnlineLatgenRecogniser* is used in Alex dialogue system for Czech **PTI!** (**PTI!**) domain available on public toll-free (+420) 800 899 998 line.

First, the architecture of Alex **SDS!** (**SDS!**) is described. Second, Section **??** presents how the wrapper *PyOnlineLatgenRecogniser* is integrated into **SDS!** Alex. Finally, Section **??** evaluates the decoder in Alex dialogue system on Czech **PTI!** domain.

5.1 Alex dialogue system architecture

The Alex dialogue system has a speech to speech user interface. The Alex dialogue system is developed in Python programming language and consists of six major components.

1. **VAD!** (**VAD!**)
2. **ASR!** (**ASR!**)
3. **SLU!** (**SLU!**)
4. **DM!** (**DM!**)
5. **NLG!** (**NLG!**)
6. **TTS!** (**TTS!**)

The system interacts with a user in *turns*. The schema in Figure **??** illustrates how the user's input is processed in single turn. The spoken input is passed to **ASR!** component which generates corresponding textual representation. **SLU!** extracts semantic meaning from the text and **DM!** decides which response to present. The **NLG!** component generates textual response from an internal representation of **DM!** and finally the **TTS!** read the text with human voice.

Each of the Alex's component runs in separate process in order parallelize the input data processing and output data generation. The components communicates among themselves through system pipes.

In order to prepare **ASR!** unit for *PyOnlineLatgenRecogniser*, we have implemented not only the wrapper itself, but also scripts for building decoding graph and evaluation. In addition, we integrated **AM!** training scripts to Alex framework. Let us introduce the framework organisation, so we can better explain how our scripts are used. The framework is separated into several logical parts:

- The core library is located at *alex/components/*. The library is domain and language independent. All components in Figure **??** are implemented in this core library.

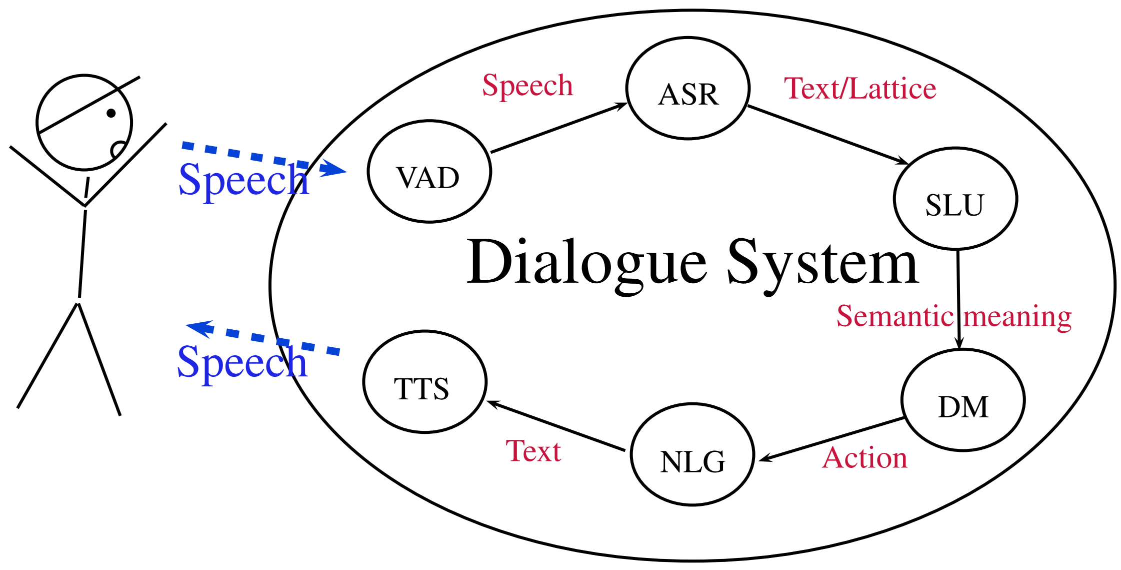


Figure 5.1: Single turn in Alex dialogue system

- Settings and scripts for specific domain applications are located in *alex/applications/*. For example, application for **PTI!** domain can be found in *alex/applications/PublicTransportInfoCS/* directory.
- The scripts which use external tools or data can be found in:
  - *alex/corpustools/* directory which focuses on formatting and organising the collected data,
  - and alex/tools/ directory which stores code for modelling **VAD!**, **ASR!**, *SIP* client, etc.
- Integration tests are stored in *alex/tests/*.
- The *alex/utils/* directory contains simple utilities for various purposes.

The components depicted in Figure **??** are represented as Python modules under the *alex/components/* directory. The source code of the components is very modular, so each component may support multiple implementations. For example the **ASR!** component currently supports several **ASR!** recognisers. The recognisers implement a common base class *ASRInterface* which is presented in Listing **??**. The supported speech recognisers are:

- OpenJulius (*alex/components/asr/julius.py*) interfaces OpenJulius decoder through sockets for on-line recognition.
- Google (*alex/components/asr/google.py*) uses cloud service for batch decoding.
- Kaldi (*alex/components/asr/kaldi.py*) imports *PyOnlineLatgenRecogniser* class and uses its functionality for on-line decoding.

One can easily choose an **ASR!** recogniser in Alex configuration file. The configuration file is also the right place to specify **AM!** and **LM!** and the speech recognition parameters if necessary.

In order to prepare a specific application with *PyOnlineLatgenRecogniser* one need to train **AM!** and **LM!**. One need to always train **SLU!** unit based on the **ASR!** unit outputs. The **LM!** model training and consequently **SLU!** training is very domain specific so the scripts are deployed for each application separately. For example, the scripts for **LM!** and **SLU!** model training for **PTI!** domain are located under directory *alex/applications/PublicTransportInfoCS/* in directories *lm/* and *slu/*.

5.2 Kaldi integration into Alex's SDS! framework

Integration of the Kaldi real-time recognizer into Alex's framework required implementing following features:

1. The *kaldi.py* module which exploits functionality of *PyOnlineLatgenRecogniser* and implements the abstract *ASRInterface*.

2. The training scripts for **AM!**s.

3. The scripts for building custom decoding graph *HCLG*. *HCLG* graph is a Kaldi effective representation of **AM!** and **LM!** used for decoding.

4. Evaluation of the **ASR!** recogniser in Alex, so the best speech recogniser can be selected.

The *PyOnlineLatgenRecogniser* integration is described in Subsection **??**. The training scripts for training **AM!**s were described in Chapter **??**. However, note that we adjusted their directory structure and copied them into *alex/tools/kaldi* directory so they nicely integrate in the Alex **SDS!**. The scripts for building the *HCLG* decoding graph are introduced in Subsection **??**. They are stored at *alex/applications/PublicTransportInfoCS/hclg/*. Finally, the Section **??** evaluates the performance of *PyOnlineLatgenRecogniser* in Alex **SDS!** and briefly compares it with Google speech recognition service used through Python module *alex/components/asr/google.py*.

5.2.1 *PyOnlineLatgenRecogniser* in Alex

The **ASR!** component in the Alex dialogue system runs as separate process, and the speech recognition is triggered based on **VAD!** decisions.

If **VAD!** detects start of speech in the input audio stream, it sends the speech signal to **ASR!** component and the *rec_in* method is called. The *rec_in* method is a part of Alex abstract *ASRInterface* illustrated in Listing **??**. See Listing **??**. In Kaldi implementation of *rec_in*, the audio is decoded using beam search while the user is speaking, i.e., the method *rec_in* gradually adds the new audio to *PyOnlineLatgenRecogniser's* buffer and immediately decodes it.[1]

If **VAD!** recognises end of speech, no more data are sent to *PyOnlineLatgenRecogniser* engine and *hyp_out* method is called in order to extracted word posterior lattice. Then, the word posterior lattice is converted to an n-best list.[2]

The *flush* method is used only if the speech recogniser wants to throw away the buffered audio input and reset the decoding.

Listing 5.1: ASRInterface

```
class ASRInterface(object):

    def rec_in(self, frame):

    def flush(self):

    def hyp_out(self):

    def rec_wav(self, pcm):
        self.rec_in(pcm)
        return self.hyp_out()
```

The method *rec_wav* from Alex's *ASRInterface* nicely illustrates how the two methods *rec_in* and *hyp_out* are used for decoding. Since the method is used only for testing purposes, it sends all input audio to the speech recogniser at once. However, in real-time application the audio is passed to *PyOnlineLatgenRecogniser* in small chunks, so the decoding can run as a user speaks.

In the on-line Kaldi settings, latency of the **ASR!** unit depends mostly on the time spent in *hyp_out* method. In the *hyp_out* method a word posterior lattice is extracted using the *PyOnlineLatgenRecogniser::GetLattice* method as described in Subsection **??**. For most cases the latency is well below 200 ms for our settings as illustrated in Figure **??**.

The Alex dialogue system frequently handles several spoken requests immediately one after another. At the end of each utterance the *hyp_out* method is called and the ASR hypothesis is extracted. Since the user already speaks when the lattice is extracted, the processor time which is already used for lattice extraction cannot be used for decoding. Consequently, the *rec_in* must decode the audio faster than the user speaks otherwise the audio cumulates in the recogniser's buffer.

We noticed the problem for chains of noises detected in **VAD!** components as multiple short utterances. The *hyp_out* method was called so often that almost no decoding was performed.

We solved the problem by improving **VAD!** so the *hyp_out* method is triggered less often. We reserved circa 100 ms for decoding between calls of the *hyp_out* method. All the utterance which are classified as speech are longer than 200 ms. As a result, the utterances can be decoded as they arrive

[1]If the **ASR!** component is busy with decoding the audio just waits in **VAD!** buffer instead of in *PyOnlineLatgenRecogniser*'s buffer.

[2]We would like to implement direct keyword spotting from *pyfst* lattices in Alex **SLU!** unit in future.

because the decoding runs almost twice as fast as user speaks and the time for decoding is at least half the time of the utterance.

5.2.2 Building in-domain decoding graph

HCLG

A decoding graph is a graph represented as an OpenFst object. It stores all the **LM!** model information and part of information for acoustic modelling. The decoding graph is necessary for decoding with Kaldi decoders. We build the *HCLG* graph using standard OpenFst operations which are implemented in Kaldi utilities.

We designed our scripts so they automatically update newly built **AM!**s and **LM!**s and create all files necessary for decoding with *OnlineLatgenRecogniser* including *HCLG* graph. The same files can also be used with standard Kaldi decoders or *PyOnlineLatgenRecogniser*.

The *HCLG* build script requires:

- **LM!**
- **AM!**
- Acoustic phonetic decision tree
- Phonetic dictionary

In addition to building *HCLG*, the script also copies necessary files for decoding from **AM!** and the *HCLG* graph to one directory. To sum up, following files are necessary for decoding with Kaldi decoders:

- Decoding graph *HCLG*,
- **AM!**,
- a matrix which defines feature transformations,
- a configuration file for speech parametrisation and feature transformations with the same settings as used for **AM!** training,
- and a **WST!** (**WST!**) — a file containing mapping between integer labels.

We also developed evaluation scripts which simply compute the statistics of measures which are evaluated given **AM!**, **LM!** and parameters in Section **??**. Both the evaluation scripts and build *HCLG* script are located in the *alex/applications/PublicTransportInfoCS/hclg/* directory.

Acoustic and language models for PTI! domain

PTI domain

The *OnlineLatgenRecogniser* is evaluated on a corpus of audio data from the **PTI!** (**PTI!**) domain. In PTI, users can interact in Czech language with a telephone-based dialogue system to find public transport connections [?]. The PTI corpus consist of approximately 12,000 user utterances with a length varying between 0.4 s and 18 s with median around 3 s. The data

Figure 5.2: The upper graph (a) shows that WER decreases with increasing *beam* and the average RTF linearly grows with the beam. The growth of the 95th RTF percentile is limited at 0.6 by setting *max-active-states* to 2000, because the *max-active-states* parameters influence presumably the worst cases with large search space. The lower graph (b) shows latency growth in response to increasing *lattice-beam*.

were divided into training, development, and test data where the corresponding data sizes are 9496, 1188, 1188 utterances respectively. For evaluation, a domain specific class-based language model with a vocabulary size of approximately 52,000 and 559,000 n-grams was estimated from the training data. Named entities e.g., cities or bus stops, in class-based language model are expanded before building a decoding graph. The perplexity of the resulting language model evaluated on the development data is about 48.

Since the PTI acoustic data amounts to less then 5 hours, the acoustic training data was extended by additional 15 hours of telephone out-of-domain data from VYSTADIAL 2013 - Czech corpus [?]. The acoustic models were obtained by BMMI discriminative training with LDA and MLLT feature transformations. A detailed description of the training procedure is given in Chapter **??**.

5.3 Evaluation of *PyOnlineLatgenRecogniser* in Alex

We focus on evaluating the speed of the *OnlineLatgenRecogniser* and its relationship with the accuracy of the decoder. We evaluate following measures:

- Real Time Factor (RTF) of decoding – the ratio of the recognition time to the duration of the audio input,
- Latency – the delay between utterance end and the availability of the recognition results,
- Word Error Rate (WER).

Accuracy and speed of the *OnlineLatgenRecogniser* are controlled by the *max-active-states*, *beam*, and *lattice-beam* parameters [?]. *Max-active-states* limits the maximum number of active tokens during decoding. *Beam* is used during graph search to prune ASR hypotheses at the state level. *Lattice-beam* is used when producing word level lattices after the decoding is finished. It is crucial to tune these parameters to obtain good results.

In general, one aims for a **RTF!** smaller than 1.0. Moreover, it is useful in practice if the RTF is even smaller because other processes running on the machine can influence the amount of available computational resources. Therefore, we target the RTF with value of 0.6, which was estimated as sufficient by informal experiments.

Figure 5.3: The percentile graphs show RTF and Latency scores for test data for *max-active-sates*=2000, *beam*=13, *lattice-beam*=5. Note that 95 % of utterances were decoded with the latency lower that 200ms.

Figure 5.4: Almost constant latency of on-line decoder (OnlineLatgenRecogniser) and linearly growing latency of cloud based speech recogniser (Google ASR service) for increasing utterance length.

We used grid search on the test set to identify the optimal parameters values. Figure **??** (a) shows the impact of the *beam* on the WER and RTF measures. In this case, we set *max-active-states* to 2000 in order to limit the worst case RTF to 0.6. Observing Figure **??** (a), we chose *beam* of value 13 for further experiments as this setting balances the **WER!**. Figure **??** (b) shows the impact of the *lattice-beam* on WER and latency when *beam* is fixed to 13. We set *lattice-beam* to 5 based on Figure **??** (b) to obtain the 95th latency percentile of 200 ms, which is considered natural in a dialogue [**?**]. *Lattice-beam* does not affect WER, but larger *lattice-beam* improves the oracle WER of generated lattices [**?**]. Richer lattices may improve **SLU!** performance.

Figure **??** shows the percentile graphs of the RTF and latency measures over the test set. The 95th percentile is the value of a measure such that 95% of the data has the measure below that value. One can see from Figure **??** that 95% of test utterances is decoded with RTF under 0.6 and latency under 200 ms. The extreme values for 5% of test utterances are in most cases caused by decoding long noisy utterances where uncertainty in decoding increase the search space slows down the recogniser. Using *beam* of 13, the *lattice-beam* of 5 and 2000 *max-active-states*, the *OnlineLatgenRecogniser* decodes the test utterances with a WER of about 21%.

percentile

In addition, we have also evaluated Google ASR service as we used it previously in Alex **SDS!**. The Google ASR service decoded the test utterances from the PTI domain with 95% latency percentile of 1900ms and it reached WER about 48%. The high latency is presumably caused by the batch processing of audio data and network latency, and the high WER is likely caused by a mismatch between Google's acoustic and language models and the test data.

Results

To conclude, we implemented **ASR!** component based on *OnlineLatgenRecogniser*. We also implemented scripts which allow easy **AM!** training and testing, and **LM!** evaluation for Kaldi speech recognition in Alex **SDS!**.

Based on evaluation, we selected the best setup[3] for ASR component in Alex Dialogue System Framework with WER under 22 %, latency less tha

[3]Setup: *beam* 12, *lattice-beam* 5, *max-active-states* 2000.

200 ms and RTF under 0.6 on **PTI!** domain. As a results, the *OnlineLatgen-Recogniser* performs significantly better than the previous **ASR!** engines.

6. Conclusion

This work presented the *OnlineLatgenRecogniser*, an extension of the Kaldi automatic speech recognition toolkit. The recogniser and its Python extension is stable and intensively used in a publicly available **SDS!** Alex[?]. The recogniser produces high quality word posterior lattices thanks to the use of a standard Kaldi lattice decoder. Scripts for **AM!** training and evaluation in Alex **SDS!** were prepared.

The training scripts[1] as well as the source code of the *OnlineLatgenRecogniser*[2] are currently merged into Kaldi repository. The Alex dialogue system and the integration of *OnlineLatgenRecogniser* is Apache, 2.0 licensed and freely available on Github[3]. The training scripts, the *OnlineLatgenRecogniser* and its Python wrapper *PyOnlineLatgenRecogniser* were developed also under Apache, 2.0 license on Github[4]

The goals set in introduction were achieved. We have successfully trained acoustic models, designed and also implemented speech recogniser and improved real-time decoder. Furthermore, we integrated the C++ *OnlineLatgenRecogniser* into Alex dialogue system written in Python. The recogniser's parameters were tuned and evaluated on **PTI!** domain. A state-of-the-art performance of speaker independent real-time recognition was achieved. As a result, the recogniser is deployed in publicly available **SDS!** Alex[5].

In addition to our implementation effort, we have also co-authored an article which uses **AM!** training scripts described in Chapter **??**. The article[?] describes the Czech and English Vystadial data sets as well as its acoustic modelling scripts in Kaldi and **HTK!**. We also submitted an article about *OnlineLatgenRecogniser's* implementation and properties to the Sigdial conference[6]. The article is currently in a review process.

Future plans include implementing more sophisticated speech parameterisation interface and feature transformations, implementing normalisation of word posterior lattices and exploring acoustic modelling based on **DNN!**.

Acknowledgments

This research was partly funded by the MEYS of the Czech Republic under the grant agreement LK11221 and core research funding of Charles University in Prague. The work described herein uses language resources hosted by the LINDAT/CLARIN repository, funded by the project LM2010013 of the MEYS of the Czech Republic. We would also like to thank Daniel Povey, Vassil Panayotov, Pavel Mencl, Ondej Duek, Matj Korvas, Lukilka, David Marek and Tomartinec for their useful comments and discussions.

[1] `http://sourceforge.net/p/kaldi/code/HEAD/tree/trunk/egs/vystadial_en/` and `http://sourceforge.net/p/kaldi/code/HEAD/tree/trunk/egs/vystadial_cz/`

[2] `http://sourceforge.net/p/kaldi/code/HEAD/tree/sandbox/oplatek2/src/dec-wrap/`

[3] `https://github.com/UFAL-DSG/alex`

[4] `https://github.com/UFAL-DSG/pykaldi`, `https://github.com/UFAL-DSG/pyfst`

[5] Alex provides **PTI!** on toll-free line 800 899 998 in Czech.

[6] `http://www.sigdial.org/`

A. Acronyms

DNN	Deep Neural Networks
SLU	Spoken Language Understanding
ASR	Automatic Speech Recognition
FST	Finite State Transducer
DFT	Discrete Fourier Transformation
GPU	Graphics Processing Unit
HTK	Hidden Markov Model Toolkit
EM	Expectation Maximization
PDF	Probability Density Function
OOV	Out of Vocabulary Word
RTF	Real Time Factor
HLDA	Heteroscedastic Linear Discriminant Analysis
HMM	Hidden Markov Model
LDA	Linear Discriminant Analysis
LM	Language Model
AM	Acoustic Model
MFCC	Mel Frequency Cepstral Coefficients
PLP	Perceptual Linear Prediction
PTI	Public Transport Information
IID	Independent and Identically Distributed
MLE	Maximum Likelihood Estimation
MLLT	Maximum Likelihood Linear Transform
CMVN	Cepstral Mean and Variance Normalisation
STC	Semi-Tied Covariance
ET	Exponential Transform
MMI	Maximum Mutual Information
bMMI	Boosted Maximum Mutual Information
PDF	Probability Density Function
MPE	Minimum Phone Error
PLP	Perceptual Linear Prediction
SER	Sentence Error Rate
SDS	Spoken Dialogue System
WER	Word Error Rate
LMW	Language Model Weight

VAD	Voice Activity Detection
DM	Dialogue Manager
TTS	Text to Speech
NLG	Natural Language Generation
WST	Word Symbol Table

B. CD content

The CD contains source code of projects developed, extended or modified as implementation part of this thesis. The thesis texts describes my work on projects listed below:

- Alex — Alex Dialogue System Framework where I added following files and directories:
  - *alex/components/asr/kaldi.py* — ASR component interfacing *PyOnlineLatgenRecogniser*
  - *alex/tools/kaldi/* — Kaldi training scripts modified for Alex
  - *alex/applications/PublicTransportInfoCs/hclg/* — Decoding graph (*HCLG*) scripts, and scripts for **ASR!** evaluation.
- The Kaldi toolkit — Speech recognition toolkit where I added directories:
  - *src/onl-rec* — Implementation of *OnlineLatgenRecogniser* and utilities
  - *src/pykaldi* — Python wrapper *PyOnlineLatgenRecogniser* and utilities
  - *egs/vystadial/s5* — Training scripts for acoustic modelling[1]
  - *egs/vystadial/online_demo* — Demos using using *OnlineLatgenRecogniser* and *PyOnlineLatgenRecogniser*.
- Pyfst — Python wrapper of OpenFst, where I improved installation and addedd several simple functions. Note I forked the original pyfst library.
- Pykaldi-eval — Repository for evaluation OnlineLatgenRecogniser written in IPython notebook. See interesting graphs.
- thesis.pdf
- Reference documentation for C++ code in *kaldi/src/onl-rec*.
- Reference documentation for Python code in *kaldi/src/pykaldi*.
- The reference documentation for my code in Alex.

[1]The same scripts were integrated into Kaldi svn trunk repository. However, the scripts are separated for Czech and English data. See `http://sourceforge.net/p/kaldi/code/HEAD/tree/trunk/egs/vystadial_cz/` and `http://sourceforge.net/p/kaldi/code/HEAD/tree/trunk/egs/vystadial_en/`

Bibliography

[1] ADSF, *The Alex Dialogue Systems Framework*, April 2014, https://github.com/UFAL-DSG/alex.

[2] Lee Akinobu, *Open-Source Large Vocabulary CSR Engine Julius*, April 2014, http://julius.sourceforge.jp/en_index.php.

[3] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri, *OpenFst: A general and efficient weighted finite-state transducer library*, Implementation and Application of Automata, Springer, 2007, pp. 11–23.

[4] Stefan Behnel, Robert Bradshaw, Lisandro Dalc Mark Florisson, Vitja Makarov, and Dag Seljebotn, *Cython: C-Extensions for Python*, 2014, `http://cython.org/`.

[5] Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean, *Large language models in machine translation*, In Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, Citeseer, 2007.

[6] Senaka Buthpitiya, Ian Lane, and Jike Chong, *A parallel implementation of Viterbi training for acoustic models using graphics processing units*, Innovative Parallel Computing (InPar), 2012, IEEE, 2012, pp. 1–10.

[7] Victor Chahuneau and Ondrej Platek, *The PyFst library: OpenFst in Python*, 2014, `https://github.com/UFAL-DSG/pyfst`.

[8] Y-L Chow, *Maximum mutual information estimation of HMM parameters for continuous speech recognition using the< e1> N</e1>-best algorithm*, Acoustics, Speech, and Signal Processing, 1990. ICASSP-90., 1990 International Conference on, IEEE, 1990, pp. 701–704.

[9] Steven Davis and Paul Mermelstein, *Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences*, Acoustics, Speech and Signal Processing, IEEE Transactions on **28** (1980), no. 4, 357–366.

[10] Mark JF Gales, *Semi-tied covariance matrices for hidden Markov models*, Speech and Audio Processing, IEEE Transactions on **7** (1999), no. 3, 272–281.

[11] Zoubin Ghahramani, *Unsupervised learning*, Advanced Lectures on Machine Learning, Springer, 2004, pp. 72–112.

[12] Joshua T Goodman, *A bit of progress in language modeling*, Computer Speech & Language **15** (2001), no. 4, 403–434.

[13] Ramesh A Gopinath, *Maximum likelihood modeling with Gaussian distributions for classification*, Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on, vol. 2, IEEE, 1998, pp. 661–664.

[14] Hynek Hermansky, *Perceptual linear predictive (PLP) analysis of speech*, The Journal of the Acoustical Society of America **87** (1990), 1738.

[15] Xuedong Huang, Alejandro Acero, Hsiao-Wuen Hon, et al., *Spoken language processing*, vol. 15, Prentice Hall PTR New Jersey, 2001.

[16] David Huggins-Daines, Mohit Kumar, Arthur Chan, Alan W Black, Mosur Ravishankar, and Alex I Rudnicky, *Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices*, Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on, vol. 1, IEEE, 2006, pp. I–I.

[17] Abdul J Jerri, *The shannon sampling theoremits various extensions and applications: A tutorial review*, Proceedings of the IEEE **65** (1977), no. 11, 1565–1596.

[18] Filip Jurk, *VYSTADIAL: Development of statistical methods for spoken dialogue systems*, April 2014, http://ufal.mff.cuni.cz/grants/vystadial.

[19] Matěj Korvas, Ondřej Plátek, Ondřej Dušek, Lukáš Žilka, and Filip Jurčíček, *Free English and Czech telephone speech corpus shared under the CC-BY-SA 3.0 license*, Proceedings of the Eigth International Conference on Language Resources and Evaluation (LREC 2014), 2014, p. To Appear.

[20] Akinobu Lee and Tatsuya Kawahara, *Recent development of open-source speech recognition engine julius*, 2009.

[21] Mehryar Mohri, Fernando Pereira, and Michael Riley, *Weighted finite-state transducers in speech recognition*, Computer Speech & Language **16** (2002), no. 1, 69–88.

[22] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar, *Foundations of machine learning*, The MIT Press, 2012.

[23] Sirko Molau, Florian Hilger, and Hermann Ney, *Feature space normalization in adverse acoustic conditions*, Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on, vol. 1, IEEE, 2003, pp. I–656.

[24] Fabrizio Morbini, Kartik Audhkhasi, Kenji Sagae, Ron Artstein, Dogan Can, Panayiotis Georgiou, Shri Narayanan, Anton Leuski, and David Traum, *Which ASR should I choose for my dialogue system?*, Proceedings of the SIGDIAL 2013 Conference (Metz, France, 2013, pp. 394–403.

[25] Hermann Ney, *Acoustic modeling of phoneme units for continuous speech recognition*, Proc. Fifth Europ. Signal Processing Conf, 1990, pp. 65–72.

[26] Daniel Povey, *The Kaldi ASR toolkit*, April 2014, http://sourceforge.net/projects/kaldi.

[27] Daniel Povey, Lukas Burget, Mohit Agarwal, Pinar Akyazi, Kai Feng, Arnab Ghoshal, Ondrej Glembek, Nagendra K Goel, Martin Karafiát, Ariya Rastrow, et al., *Subspace Gaussian mixture models for speech recognition*, Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on, IEEE, 2010, pp. 4330–4333.

[28] Daniel Povey, Mark JF Gales, Do Yeong Kim, and Philip C Woodland, *MMI-MAP and MPE-MAP for acoustic model adaptation.*, INTERSPEECH, 2003.

[29] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely, *The Kaldi speech recognition toolkit*, IEEE 2011 Workshop on Automatic Speech Recognition and Understanding, IEEE Signal Processing Society, December 2011, IEEE Catalog No.: CFP11SRW-USB.

[30] Daniel Povey, Mirko Hannemann, Gilles Boulianne, Lukas Burget, Arnab Ghoshal, Milos Janda, Martin Karafiát, Stefan Kombrink, Petr Motlicek, Yanmin Qian, et al., *Generating exact lattices in the WFST framework*, Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on, IEEE, 2012, pp. 4213–4216.

[31] Daniel Povey, Dimitri Kanevsky, Brian Kingsbury, Bhuvana Ramabhadran, George Saon, and Karthik Visweswariah, *Boosted MMI for model and feature-space discriminative training*, Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on, IEEE, 2008, pp. 4057–4060.

[32] Daniel Povey and Brian Kingsbury, *Evaluation of proposed modifications to MPE for large scale discriminative training*, Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on, vol. 4, IEEE, 2007, pp. IV–321.

[33] Daniel Povey, Brian Kingsbury, Lidia Mangu, George Saon, Hagen Soltau, and Geoffrey Zweig, *fMPE: Discriminatively trained features for speech recognition*, Proc. ICASSP, vol. 1, Philadelphia, 2005, pp. 961–964.

[34] Daniel Povey, G. Zweig, and A. Acero, *The Exponential Transform as a generic substitute for VTLN*, IEEE ASRU, 2011.

[35] Josef Psutka, *Benefit of maximum likelihood linear transform (MLLT) used at different levels of covariance matrices clustering in ASR systems*, Text, Speech and Dialogue, Springer, 2007, pp. 431–438.

[36] Josef Psutka, Ludek Müller, and Josef V Psutka, *Comparison of MFCC and PLP parameterizations in the speaker independent continuous speech recognition task.*, INTERSPEECH, 2001, pp. 1813–1816.

[37] Michael Riley, *OpenFst Quick Tour*, April 2014, http://www.openfst.org/twiki/bin/view/FST/FstQuickTour.

[38] Luis Javier Rodríguez and Inés Torres, *Comparative study of the Baum-Welch and Viterbi training algorithms applied to read and spontaneous speech recognition*, Pattern Recognition and Image Analysis, Springer, 2003, pp. 847–857.

[39] David Rybach, Stefan Hahn, Patrick Lehnen, David Nolden, Martin Sundermeyer, Zoltan Tüske, Siemon Wiesler, Ralf Schlüter, and Hermann Ney, *RASR-The RWTH Aachen University open source speech recognition toolkit*, Proc. IEEE Automatic Speech Recognition and Understanding Workshop, 2011.

[40] Gabriel Skantze and David Schlangen, *Incremental dialogue processing in a micro-domain*, Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics, Association for Computational Linguistics, 2009, pp. 745–753.

[41] UFAL-DSG, *The Alex Dialogue Systems Framework - Public Transport Information*, April 2014, http://ufal.mff.cuni.cz/alex-dialogue-systems-framework/.

[42] Keith Vertanen, *Baseline WSJ acoustic models for HTK and Sphinx: Training recipes and recognition experiments*, Tech. report, Cavendish Laboratory, University of Cambridge, 2006.

[43] Karel Veselỳ, Arnab Ghoshal, Lukáš Burget, and Daniel Povey, *Sequence discriminative training of deep neural networks*, Proc. INTERSPEECH, 2013, pp. 2345–2349.

[44] R Weide, *The cmu pronunciation dictionary, release 0.7a*, Carnegie Mellon University, 1998.

[45] Ian H Witten and Timothy Bell, *The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression*, Information Theory, IEEE Transactions on **37** (1991), no. 4, 1085–1094.

[46] Xuchen Yao, Pravin Bhutada, Kallirroi Georgila, Kenji Sagae, Ron Artstein, and David R Traum, *Practical evaluation of speech recognizers for virtual human dialogue systems.*, LREC, Citeseer, 2010.

[47] Jinjin Ye, *Speech recognition using time domain features from phase space reconstructions*, Ph.D. thesis, Marquette University Milwaukee, Wisconsin, 2004.

[48] SJ Young, *The HTK Hidden Markov Model Toolkit: Design and Philosophy*, vol. 2, 1994, pp. 2–44.

[49] Steve Young, Gunnar Evermann, Mark Gales, Thomas Hain, Dan Kershaw, Xunying Liu, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, et al., *The HTK book (for HTK version 3.4)*, Cambridge university engineering department **2** (2006), no. 2, 2–3.

[50] Xiaohui Zhang, Jan Trmal, Daniel Povey, and Sanjeev Khudanpur, *Improving deep neural network acoustic models using generalized maxout networks*, submitted to ICASSP (2014).