DEEP NEURAL NETWORK APPLICATIONS FOR AUTOMATIC SPEECH RECOGNITION SYSTEMS

A THESIS

submitted by

ANGEL BUENO RODRIGUEZ

for the award of the degree

of

DUAL DEGREE

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2014

THESIS CERTIFICATE

This is to certify that the thesis titled **Deep Neural Network Applications for Speech Recognition Systems**, submitted by **Angel Bueno Rodriguez** , to the Indian Institute of Technology, Madras for the award of the degree **Dual Degree**, is a bonafide record of the research work done by him under my supervision. The contents of the thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Umesh S
Research Guide
Professor
Department of Electrical Engineering
IIT Madras, 600 036

Place: Chennai
Date: 13th June 2014

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my project advisor Prof.S. Umesh for his inestimable guidance and encouragement. Without his assistance, support and dedicated involvement in every step throughout the project, this research would have never been accomplished. I could not have imagined having a better advisor and mentor for my Dual Degree project. I also wish to thank my professors from University of Granada, Prof. Antonio Garcia Rios for his invaluable support, Prof. Carmen Benitez for introducing me to the field of Signal Processing and Prof Luz Garcia Martinez for channeling my early enthusiasm for neural networks and being always inspirational.

I thank my lab colleagues Bhargav, Basil Abraham, Neethu, Jayesh, Vishnu P, Navneeth, Sekhar, Swetha, Sriranjani and Adarsh for providing a stimulating and pleasant lab work environment. It is also my pleasure to acknowledge the priceless presence of our lab administrator Prabhakaran who maintained all the machines in the lab so efficiently. I would like to thank N. Vishnu Prasad for providing me all the necessary support, advice and help at my arrival in India. I will be eternally grateful to Vinay N. K. who has been unwavering in his personal and professional support during the time we spent together at this institute and for being a source of motivation by stimulating my curiosity on difficult programming tasks. I am grateful to Karthick B.M. for his practical advice, for helping me to enrich my ideas and for all the countless hours that we spent together discussing regarding my project. I have no words to my great friend Clement Grozel for all his support, encouragement, advice, moments we shared in India.

Most importantly, none of this could have happened without my parents. I am always indebted to them. Without their love and support throughout my entire life I will not be what I am today. This thesis stands as a testament to your unconditional love and encouragement.

ABSTRACT

KEYWORDS: DNN; HMM; CDHMM; GPU

Abstract

This thesis investigates how Deep Neural Networks can improve the performance of Speech Recognition Systems. At the beginning, most recognition systems were based on hidden Markov Models (HMMs) as a statistical framework which models the sequential structure of speech signals. Despite their good performance, the non-optimal assumptions made by HMMs limit its capabilities. Few years back, the Gaussian Mixture Models (GMMs) have replaced the HMMs as the dominant technique for speech recognition by modeling the HMMs states with a mixture of Gaussian functions. But even this technique has its limitations as it requires detailed assumptions about the data distribution in order to estimate the posterior probabilities. Those systems evolved together as the only way to do speech recognition in a time when computers were not fast enough to achieve highly computational efficient methods. Using a mathematical representation of a biological neuron, Neural Networks (NNs) were born as a model that avoid many of those wrong assumptions, being able to learn complex functions, tolerate noise and support parallelism. Deep Neural Networks (DNNs) take those characteristics from NNs and go one step beyond. The main idea behind "deep" is that every layer is able to provide a higher level representation and a better classification of the input. If all those properties of DNNs are applied in acoustic modeling techniques, we can design a very reliable Automatic Speech Recognition System.

The basic of this project is the research about performance of this new acoustic modeling technique based on a deep learning approach and its applications on Automatic Speech Recognition Systems. Two different ways to implement deep neural networks for acoustic modeling were explored: A processor architec-

ture (CPU) and a graphic architecture (GPU). The advantages of GPU in terms of computational cost, accuracy and speed combined with the characteristics of DNNs lead into outperforming results compared with other conventional techniques. For Resource Management Database, a relative improvement of 43.6 % over Continuous Density Hidden Markov Model (CDHMM) has been achieved on DNN in a CPU, meanwhile 48.9 % relative improvement was obtained in GPU approach.

Contents

List of Tables

List of Figures

ABBREVIATIONS

ASR Automatic Speech Recognition

ANN Artificial Neural Network

CDHMM Continuous Density Hidden Markov Model

CMVN Ceptral Mean and Variance Normalization

CPU Central Processing Unit

CD Contrastive Divergence

DBN Deep Belief Network

DNN Deep Neural Network

FMLLR Feature-Space Maximum Likelihood Linear Regression

GMM Gaussian Mixture Model

GPU Graphic Processing Unit

HMM Hidden Markov Model

LDA Linear Discriminative Analysis

MFCC Mel-frequency Cepstral Coefficients

MLP Multi Layer Perceptron

RBM Restricted Boltzmann Machine

RM Resource Management

SAT Speaker Adaptation Techniques

SGMM Subspace Gaussian Mixture Models

WER Word Error Rate

Chapter 1

Introduction

Speech is the natural ability that humans have to communicate among themselves. Speech is derived from a highly complex brain structure that allows human beings to produce a language model that describes the environment surrounding them. No other living species has the anatomy and the brain mechanisms to produce speech. This capacity of speech sets us apart from all creatures known, even those closest to us on the evolutionary scale. Truly, this faculty of human speech and language is one of the most astonishing attributes of mankind.

The most important skills are learned during our early childhood, without any instruction. When we start to grow, we use speech communication through the rest of our lives. The speech process is so natural that we can not even realize the complexity of the phenomena. As speech is related with human brain and processes of the unconscious concerning the human biology, there are many factors that can affect speech production: Accent, articulation, pronunciation, nasality, pitch, volume, speed, background noise distortion are only a few examples. Furthermore, the speech will be directly influenced by external sources as transmission media if speech signal is used in electronic devices. All the variability intrinsic to speech production nature makes speech recognition a very complicated task.

Automatic Speech Recognition (ASR) is the scientific field of study where the mathematical models employed to convert human speech into machine understandable text are studied. The parameters related with speech recognition are so many that the mathematical models developed are very complex. Despite of that complexity, we have been always fascinated of how humans and machines can communicate. ASR systems have a special place in science fiction. It will always be in the memory of many the film " 2001: A space odyssey " where Captain Bowman has a perfect conversation with the supercomputer HAL 9000 or J.A.R.V.I.S in Iron man talking with Tony Stark about how to improve the Iron man armor. The

ultimate goal of ASR systems is to create such communication between humans and machines in the most effective way possible. But it is still impossible to replicate the natural language patterns between humans when talking to a machine due to the limiting factors previously mentioned.

The classical approach for ASR systems based on statistical Gaussian models have reached a limit in terms of computational power and performance. One single question made a big change in ASR systems: What makes people so good at recognizing speech?

The brain is wired in a way that enables humans to understand the speech with the associated knowledge related to the language they communicate. But brain and computers do not follow the same computational paradigm. In a computer, a clock frequency synchronizes the program instructions with the central processor and data is stored in an addressable memory. The human brain uses a massively parallel connection of slow and simple processing elements known as neurons which are connected by weights (synapses) whose strengths are modified by experience, creating an associative memory. Taking this as an inspiration, many researches and literature have been generated since then. The most significative stage was in 1957 with the perceptron proposed by Rosenblatt. The perceptron was the simplest model built to imitate the connectivity between neurons. But recent advances on machine learning algorithms have lead into deep learning techniques to provide high-level abstractions in the input data by using architectures composed of multiple non-linear transformations.

DNNs were proposed in 1985 by G.Hinton combining the machine learning techniques with the perceptron structure. The evolution of hardware and the availability of programming languages introducing parallel computing give us the capacity to rebuild the brain computational model. DNNs are able to create a higher level representation of the input and all the units can be updated jointly making easier to train the model with limited resources. Thus, DNNs stand as the state-of-the-art in speech recognition systems. The main motivation of this thesis is therefore to study the performance of DNNs in ASR systems. Furthermore,this thesis will focus on DNNs applications for Indian languages, where less

data resources are available. Also, a GPU hardware architecture is investigated to give the closest approach to the computational brain model and see if results are improved.

The rest of the thesis is structured as follows:

- Chapter 2 summarizes a brief overview in speech recognition and will provide an introduction into acoustic modeling techniques.
- Chapter 3 provides the essential framework of neural networks and its application on speech recognition.
- Chapter 4 presents our research with deep neural networks and shows the results for different language databases.
- Chapter 5 presents the conclusions of this thesis and provides future lines of action.

Chapter 2

Review of Speech Recognition

In this chapter a brief review of speech recognition field will be described. This introduction will be followed by a discussion on Hidden Markov Models in detail and a summary of the algorithms related with this technology will be discussed.

2.1 Fundamentals of Speech Recognition

Think about a bird singing on a tree early on the morning, the sound of the ocean waves hitting rocks or the ignition of a rocket being launched to the space. All this sounds are generated continuously in time according to certain attributes as frequency or sound intensity. Every single process in nature will generate a set of continuous values on a continuous interval of time. This is the definition of an analog signal. But we are limited by many factors and sadly, we can not capture all the information in an analog signal. In practical applications, a finite number of samples can be extracted from the signal. The speech signal is highly random signal and it can be affected by many factors. Due to its nature, speech signals are highly variational on time and non-stationary. According to Rabiner and Schafer (2010), we can consider the signal as stationary process on a *25ms* duration window. Thus, we can assume the speech signals as pseudo-stationary signals. Hence, all the main characteristics of them can be studied.

Speech recognition can be considered a pattern recognition multilevel task in which all the signal will be structured in a hierarchy of subword units which phonemes as basic units. From phonemes, morphemes, words, phrases and sentences can be built. Therefore, we can consider phonemes as the smallest linguistic unit. The layer model has some advantages in speech recognition as we can add additional constraints per layer to limit the error or uncertainties at lower lay-

ers, e.g., if a sentence is well pronounced or the speaker has accent that make recognition more complicated.

At this point, one question needs to be answered: How we convert the speech sound to basic subunits words? The basic structure of a standard speech recognition system is shown in figure 2.1. A more detailed explanation about the process can be found in Rabiner and Schafer (2010) .

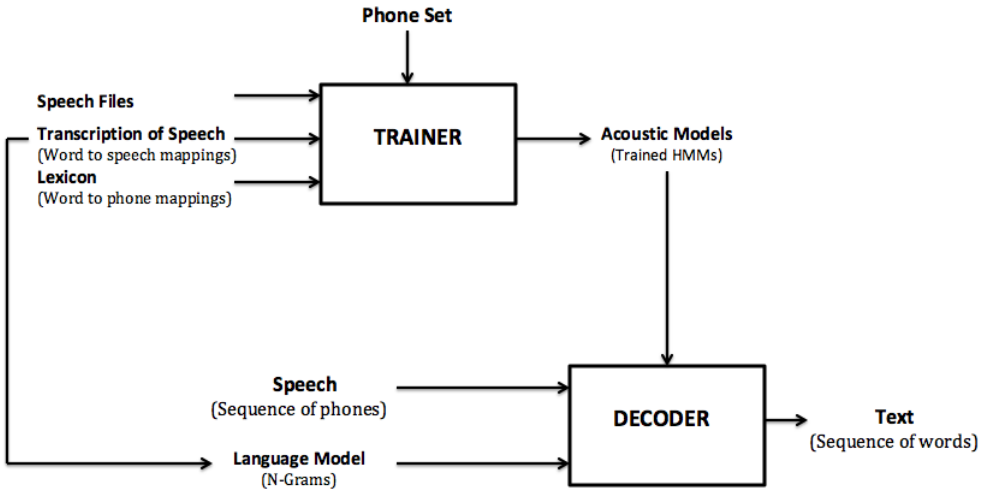

Figure 2.1: Structure of speech recognition systems

The first step is to sample the signal. Typically, this is done at high frequency, e.g, 16KHz over a microphone or 8KHz on a telephone. As a result, we will get a largely array of speech samples. From the speech samples,the conversion process from the sampled signal into a sequence of feature vectors to capture the spectral variability of the data is known as *Feature Extraction.*

Once all the features have been processed, a statistical framework known as Hidden Markov Model (HMM) is used to model those features. The modulation is done during the training stage if enough amount of transcribed training speech is available. Dictionary model is defined as the associative model that correlates the words with the sound.

After the training stage, testing is done with identical feature extraction process. In order to capture the properties of a language and give a probability to the next word in a speech sequence a language model is defined. Viterbi algorithm

is used for decoding in combination with dictionary, language model and the test data. After Viterbi decoding, the text output can be obtained. ASR systems need large amount of vocabulary. Hence, triphones are built. A triphone is the acoustic realization of a phoneme when surrounded by two specific phones. In the Viterbi decoding, those triphones will be chained together according to some given probabilities and words are formed.

2.1.1 Hidden Markov Models

The HMM has been considered as the most successful milestone in speech recognition field. HMMs, as explained in Rabiner and Juang (1986), is defined as a set of states connected by transition probabilities. In each state, a process of measurable observations is represented. The transitions between states is ruled by a finite state Markov chain. If a transition between states happens, then one output symbol will be generated in that state. The concept behind "*hidden*" is, for a given individual output sequence, we can not immediately identify the states sequence. In common language, we can consider the HMM as a "*black box*" where the input is given and the output is obtained, but we can not see "*whats happening inside*".

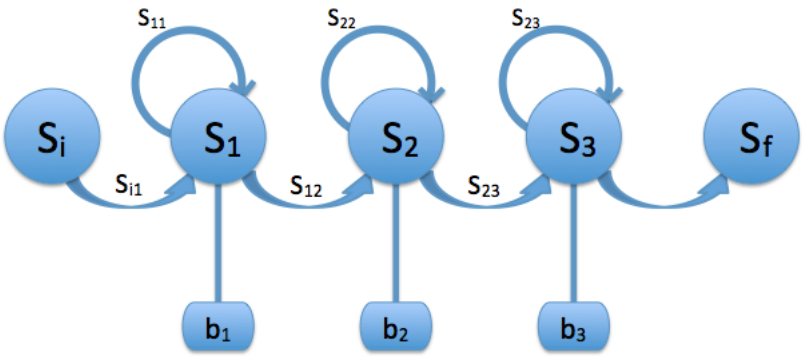

Figure 2.2: Hidden Markov Model

From a mathematical point of view the HMM can be defined as $\lambda = (\Pi, A, B)$ where :

- Π is the initial state distribution

- A is the state transition matrix where each element of the matrix a_{ij} represents the probability of going from state i to state j.
- B is the emission probability matrix where each element $b_i(u)$ is the probability of emitting a sound while in state i

At discrete intervals of time, a transition between state "one" to "two" will take place following the transition matrix A and a output will be generated. Speech sentences are spoken from left to right, so HMMs will always go forward or have some loops that represent the emission of the same sound in a spoken sentence. HMMs can also be structured in a hierarchy level from phoneme level to sentence level. But there are some problems associated with HMMs according to Rabiner and Juang (1986) that must be solved in order to have real applications. Those problems and their solutions can be briefly summarized as :

- **Problem 1 :** For an output sequence O in the model $\lambda = (\Pi, A, B)$, how do we compute the probability $Pr(O|\lambda)$ of the output sequence?

The main issue with this problem is related with matching the model with a given observation sequence. It is necessary to find an algorithm with a high efficiency in terms of computation. This algorithm is well-known as "*the forward procedure*". Given the state sequence i and the output sequence O in a model λ, the probability $Pr(O|\lambda)$ can be calculated as the product of the observation probabilities at every t through the forward procedure:

$$Pr(O|\lambda) = \sum_{all\, i} Pr(O|i, \lambda) \cdot Pr(i|\lambda) \tag{2.1}$$

$$Pr(O|\lambda) = \sum_{all\, i} \pi_{i1} b_{i1}(O_{i1}) a_{i1i2} b_{12}(O_2) ... a_{it-1} b_{it}(O_t) \tag{2.2}$$

The implementation of (2.2) is highly computational and not feasible as it increases the number of operations required to estimate the result and it has to compute all the probabilities in the state sequence. An efficient implementation is made taking the maximum probability in the state sequence, as:

$$Pr(O|\lambda) = \max_{i} \pi_{i1} b_{i1}(O_{i1}) a_{i1i2} b_{12}(O_2) ... a_{it-1} b_{it}(O_t) \tag{2.3}$$

The (2.3) solution is called the Viterbi algorithm and it can be easily implemented in ASR systems due to its low computational cost.

- **Problem 2 :** Given a set of outputs $O = O_1, O_2, O_3, ..., O_T$, we must figure out how to choose the best optimal states.

Forward procedure algorithm allows to calculate the probability of a HMM with respect to sequence of observations O, but we remain clueless about the states sequences that gave the output. The Viterbi algorithm is employed here as a mathematical tool in which the main difference is how probabilities are chosen. Viterbi is very similar to forward procedure but whereas the forward algorithm takes the sum of all the probabilities, the Viterbi algorithm takes the maximum probability. Therefore, the best path in a given set of states $i = i_1, i_2, i_3, ..., i_T$ will be given by:

$$argmax\, P(i|O, \lambda) = arg_i max\, \frac{P(i|\lambda) \cdot P(O|i, \lambda)}{P(O|\lambda)} \tag{2.4}$$

The above (2.4) is another way to write the Viterbi algorithm. The power of Viterbi lies on its simplicity to calculate the best path with the maximum probability with a low computational cost compared to the forward procedure.

- **Problem 3 :** How is it possible to maximize the probability $Pr(O|\lambda)$?

The most complicated problem related with HMM is due to the mathematical nature of the model. There is no direct solution for maximizing this probability and iterative procedures as gradient techniques or Baum-Welch method have to be used.

It is shown in that the common and natural representation of HMM is done through weighted finite-state transducers (WFSTs). WFST framework provides not only a representation of HMM, but also context-dependencies, dictionaries,

grammars and other outputs. They are connected by weights that can be optimized by efficient algorithms whose ultimate goal is to make weights distribution optimal. A further explanation can be found in Mohri *et al.* (2002).

2.1.2 Hybrid Systems: GMM-HMM

The states of an HMM have to be implemented by a probability distribution per state. The most common approach is made by direct modeling of the probabilistic distribution in each state. Typically, the best approximation is made by taking a K mixture of Gaussian distributions over the state space. A detailed implementation about this approach can be obtained in Woodland *et al.* (1994). The $b_j(u)$ in a HMM can be obtained jointly with a GMM as:

$$b_j(u) = \sum_{k=1}^{K} c_{jk} G(u, \mu_{jk}, U_{jk}) \tag{2.5}$$

Where c_{jk} is the weight factor of the Gaussian G characterized by its covariance matrix U_{jk} and meanμ_{jk}. This mathematical approach is known as Continuous Density Hidden Markov Model or CDHMM. The disadvantage of this model is that parameters are not shared by states.Therefore, for many states, a large value of K Gaussian might result in many parameters. If the value of K is decreased, a mediocre modulation of the Gaussian will be done and it may yield into a bad performance of the system.

2.1.3 Mel-Frequency Cepstra Coefficients (MFCC) features

Mel-Frequency Cepstra Coefficients (MFFCs) are the most used features in the state-of-art speech recognition systems. The steps can be summarized in Fig. 2.3:

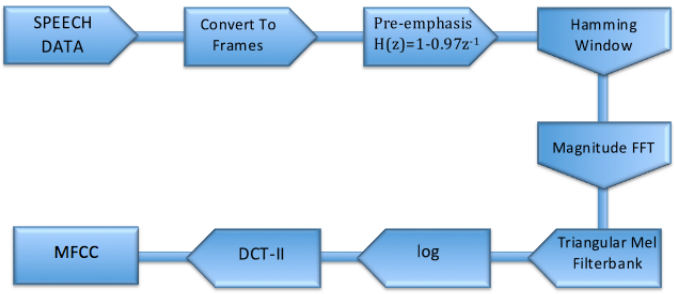

Figure 2.3: MFFC features extraction

The first stage is done through a short time processing of the input signal. The ultimate goal of this stage is to capture the spectral envelope of the signal. Speech input is analyzed by short analysis window (typically *25 ms* and *50%* frame overlap) in order to make a frame conversion. A pre-emphasis filter is applied to give weights to higher formants. Taking a Hamming window to minimize noise, the spectrum is obtained using DFT.

In second stage Mel Filterbank Analysis (MFB) is performed. Based on human perception experiments, the human ear act as a filter that concentrates on only a certain frequency range. Those filters are non-uniformly spaced on the frequency axis and concentrates more filters in the low frequency regions. This can be done by a triangular filter bank with Mel-wrapping. As a results, for each triangular filter an output coefficient will be obtained. Then,apply a log operation on each coefficient to reduce dimensionality. Finally, a DCT is performed to compress the information and uncorrelated the vector. Some additional processing as cepstra liftering is done to give same weight for all the coefficients. Additional processing as cepstra mean subtraction (CMS) is also performed to reduce any unwanted effect during the recording stage.

2.1.4 Cepstral Mean and Variance Normalization (CMVN)

MFCC feature extraction is very sensitive to the presence of background noise. This can be a serious problem that yields in a performance degradation of the system, specially in the testing phase. Previous solutions like CMS are not good

enough as they are also highly affected by noise background. A robust feature extraction approach is done through cepstral mean and variance normalization (CMVN) . This technique uses both the sample mean and standard deviation to normalize the cepstral sequence. Given a set of feature MFCC vectors $O^T = o_1, o_2, o_3, ..., o_T$, the mean and variance vectors are computed as:

$$\sigma^2 = \frac{1}{T}\sum_{t=0}^{T-1} diag(o_t o_t^T)$$

$$\mu = \frac{1}{T}\sum_{t=0}^{T-1} o_T$$

$$\tilde{O_t(d)} = \frac{o_t - \mu}{\sigma} \tag{2.6}$$

At the end of the normalization operation, the mean of the cepstral sequence must be zero and the variance equal to one. CMVN is applied to the full cepstra vector and it is not associated with any particular type of distortion. It provides robustness against speaker variability and additive noise background. There is many techniques in feature extraction,as proposed by De la Torre *et al.* (2005) based on Histogram Equalization (HEQ).

2.2 Brief review of speech techniques

Some preprocessing techniques must be done in order to start the training of neural networks. The last part of this chapter will be dedicated to give a brief review on those techniques.

2.2.1 Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) is an application of pattern classification field into speech recognition. For a given set of input features X in a n-dimensional space, the mathematical challenge is to find a linear transformation for feature

vectors Y in an m-dimensional space (with $m < n$) such that the separability between classes is maximum. A linear combination with a vector w^t will be given by :

$$Y = w^t X \tag{2.7}$$

From geometry, if $|w| = 1$ then, each y_i is the projection of x_i onto a line in direction w. Therefore, what we are interested to compute in LDA analysis is not the magnitude, but the best direction of the w vector as it will give the optimum discrimination. Lets assume that our input data X must be classified into J classes with meanμ_j and shared variance matrix $\sum$. Let it be three possible matrices W: within-class, B : between-class and T: total scatter matrix. The transformation will be done by enhancing the total scater matrix T keeping the within-class matrix W constant. The matrix T, W and B are defined as:

$$T = \sum_{k=1}^{J} \sum_{g_i=1} (X_i - \bar{x})(X_i - \bar{x})^T \tag{2.8}$$

$$W_j = \sum_{g_i=j} (X_i - \bar{x})(X_i - \bar{x})^T \Rightarrow W = \sum_{g_i=1} W_j \tag{2.9}$$

$$B = T - W \tag{2.10}$$

Thus, the maximization of the ratio between B and W matrices will lead into the following equation:

$$\hat{W} = argmax(\frac{w^t B w}{w^t W w}) \tag{2.11}$$

It is shown in Haeb-Umbach and Ney (1992) that the best solution to maximize those equations yields in a projection of X into the subspace of those m eigenvectors in $W^{-1}B$ which correspond to the m largest eigenvalues. LDA is a robust model to any non linear transformation in the feature spaces as well as invariant

to any linear transformation previously made. A set of new features with smaller dimensionality is obtained.

2.2.2 Maximum Likelihood Linear Regression (MLLR)

Maximum Likelihood Linear Regression MLLR is a based likelihood technique used for adapting Gaussian mean vector in HMM systems. The basic procedure is detailed in Woodland *et al.* (1994). Starting from the adaptation data from a new speaker, MLLR will update the model mean parameters to maximize the likelihood of the adaptation data. Assuming a CDHMM model with a given extended mean vector ξ_s, the adapted mean vector $\bar{\mu_s}$ will be calculated as :

$$\bar{\mu}_s = W_s \xi_s \tag{2.12}$$

where W_s is a $n \times (n+1)$ dimension matrix with n as the dimension of the observation vector O. The matrix W_s maximizes the likelihood of the adaptation data. The extended mean vector ξ_s is defined as $\xi_s = [\omega, \mu_s^T]$ where ω is just a parameter to control the offset. Therefore, the adapted Gaussian probability density function ($\bar{\mu}_s$, $\sum_s$) becomes:

$$b_{adapt}(o) = \frac{1}{(2\pi)^{n/2} |\sum_s|^{1/2}} \cdot \exp^{-\frac{1}{2}(O - W_s\xi_s)'(\sum_s)^{-1}(O - W_s\xi_s)} \tag{2.13}$$

This methodology depends on the Gaussian m and it can be updated depending on the presence of silence or non-silence.

2.2.3 Feature-space MLLR (FMLLR)

Feature-space MLLR (fMLLR) technique is employed in ASR systems to reduce the mismatch between the adapted models and the acoustic data for a given speaker. The algorithm is fully described in Ghoshal *et al.* (2010) and its mathematical roots lie in Constrained MLLR. For what concerns,in a GMM-HMM

model, the full covariance matrix is used and a Hessian computation in the transformed space is done in order to find the gradient with a number of pre-established iterations. For each in-speaker transform W^s, the update rule will be given as:

$$W^s \leftarrow W^s_{n-1} + k\triangle \tag{2.14}$$

where $\triangle$ is the proposed estimation in W and k the step size. As $\triangle$ is proportional to the gradient, we can rewrite the previous equation as:

$$W^s \leftarrow W^s_{n-1} + k(\frac{1}{\beta}\tilde{P}) \tag{2.15}$$

Where the matrix $\tilde{P}$ is the Hessian transformation of a previous matrix P which is the transformed gradient matrix in the adapted space. The constant $\frac{1}{\beta}$ is needed due to the expected Hessian is a per-observation quantity. This technique stands as an excellent optimization method and provides consistent results if it is done by taking all the available data for a particular speaker and gives poor results if an adaptation per utterance is done.

2.2.4 Speaker Adaptation (SAT)

Let it be a speech recognition system in which a high accuracy for one speaker is achieved. The motivation in Speaker Adaptative Training (SAT) is the improvement of the accuracy for another speaker taking only the utterances worth his/her speech data by maximizing the likelihood of the training data given the MLLR-adapted models. A further and detailed explanation can be found in Anastasakos *et al.* (1996). For what concerns us, lets assume a set of speakers R and a initial CDHMM modelλ. Each speaker will generate a sequence of observations O. The mathematical formulation suggest to find a mapping function G that convert features from initial model to an optimal model λ_c given by:

$$(\lambda_c, G) = \underset{(\lambda_c, G)}{arg\,max} \prod_{r=1}^{R} L(O^r; G^r(\lambda_c)) \tag{2.16}$$

where $L()$ is the likelihood estimator. In SAT, both λ_c and G are estimated jointly. This frameworks has some problems in terms of computational cost. From the stored standard diagonal statistics for each speaker, the estimation of optimal model and mapping function is done jointly, by keeping the mean and the variance updated for each speaker. Through each computation, mean and variance must be read from disk and updated. Infeasibility can arise due to excessive use of disk space and the computation of per-speaker statistics. Alternative solutions have been proposed by Povey *et al.* (2008). The most common technique is diagonal SAT, where only the diagonal quadratic term in the mean's objective function is stored. This fast training SAT technique will save disk space and computation time, leading into a fast and efficient method to estimate the optimal model.

Chapter 3

Neural Networks

In this chapter, we will briefly review the fundamentals of Neural Networks (NNs). Since its beginning, NNs have evolved following a specific design for the application they required. But there are some specific elements in common among all of them:

- A set of processing units that deal with the computational procedure.
- A set of connections characterized by a value (or strength) which describes how they interact.
- A training procedure that involves preparation of a neural network for a particular task.

A theoretical discussion, in relation to different configurations of neural network architectures and its applications in speech recognition systems, will be held in the upcoming sections of this chapter.

3.1 The first generation of neural networks

In a neural network, a huge amount of smallest processing units can be found. Those smallest units are known as *neurons* as per the biological model. In a human brain, a huge amount of those *neurons* are connected working jointly in parallel. Tree-like nerve fibers called *dendrites* are associated with the cell-body. The signal from the others *neurons* are received by the *dendrites*. At the same cell, a long fiber called the *axon* branches itself to the synaptic zones. The *synapse* is a chemical process activated by the increment of chemical potential according to a threshold value in which *neurons* communicate with each other through electrical pulses. When many pulses are received, their effects accumulate. Therefore, the *neuron* acts as a kind of "adding" device. The first mathematical model was

developed by McCulloch and Pitts (1943) as a *binary threshold unit* that can give one or zero as output. The model is shown in Figure 3.1:

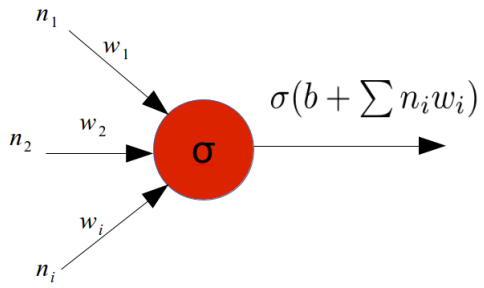

Figure 3.1: McCulloch&Pitts neuron model

For a sum of inputs n_i with associated weights w_i and a threshold value b, an activation threshold function $\sigma()$ must be computed, given as a result a binary output s that can be used as a decision boundary for classification task. The formula is given as in:

$$s = \sigma(b + \sum n_i w_i) \tag{3.1}$$

The limitations of this model are clear as binary decision boundary is not good enough for any task which involves huge amount of data and complex dimension spaces.

This mathematical model leads into the Rosenblatts perceptron model published in Rosenblatt (1958) where the combination of simple units will lead into more complex feature spaces based on the idea that each neuron would split the feature space with a hyperplane. A layer of hand-coded features is used to recognize words. A schematic overview of the model is given in Fig.3.2:

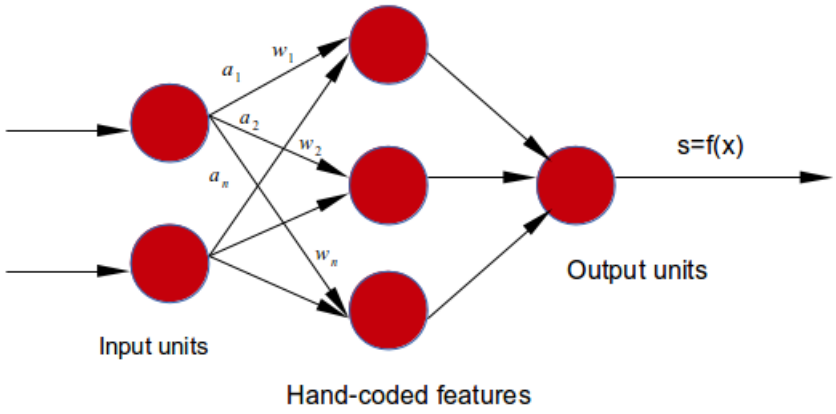

Figure 3.2: Perceptron model

A target output b is defined and our binary output $s = f(x)$ is compared with b. The resulting error $\delta = b - s$ is used to adjust the weights (connections) between the input and the middle layer according to the rule:

$$\triangle w_i = \eta \delta a_i \tag{3.2}$$

The way in which a neural network adjusts its weights is called the learning rule. But how is a perceptron able to learn? From a geometric point of view, we define the weight space as the space with one dimension per weight where each point in the space represents a particular setting of all the weights. As an example of the complexity attached with learning, notice that it can be fourteen dimension space if fourteen weights are present. If the threshold is removed, each neuron will split the region in a hyperplane. The challenge is to find the correct side of this hyperplane in which the weight must lie. It can be demonstrated Minsky and Papert (1987) that the correct direction will be computed regarding the angle of the scalar product between the input with the weight vector. Thus, for a large number of neurons a hyper-cone of feasible solutions will be formed. An interesting and detailed explanation of geometry in perceptrons can be read in Minsky and Papert (1987). After some iterations, the correct feature vector must lie in the feasible solution region if it exist.

Notice that perceptrons are limited as the solution region must exist. The existence of this region is related with the input features.Hence,features limitation is a problem in perceptrons as they need to learn the correct features. Even if hand-coded features are used, the existence of such region is not guaranteed. In case this region does not exist, the weights in the perceptron will not converge. Thus, perceptrons have a very limited capability to solve changing problems. In ASR systems, a large amount of vocabulary is employed and the objective function keeps changing. This will yield into an infeasible effective representation of the desired target.

3.2 The second generation of neural networks

The second generation of neural networks are well known as Multilayer Perceptrons (MLP). They were developed to solve the limitations of the previous generation. A new set of elements known as *"hidden layers"* were incorporated. Those hidden layers are an adaptive non-linear unit that will allow an efficient learning. At the hidden units is we do not know what they have to do, but a computation regarding the change in an error function when we modify the hidden activity can be done very efficiently. In MLP, neurons in one layer are connected to neurons in the next layer. Both properties, architecture and connections, lead in a higher-order and more complex representation of the input vector. For a given set of inputs $x_0, x_1, ...x_P$, we must define the desired output function $f(x, \vec{\theta})$ with $\vec{\theta} = \{w_{ij}, w_{jk,}, w_{kl}\}$ the set of weights on each layer. The MLP model is given in Fig. 3.3:

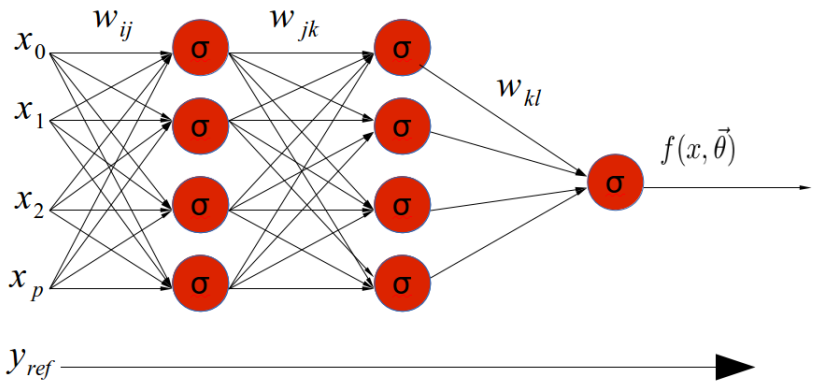

Figure 3.3: Multilayer Perceptron Model

Thus, for a reference model y_{ref} a cost function must be defined. Regarding the task, for classification problems, cross-entropy cost function is used, meanwhile for regression problems it is better sum-of-squared error function. In speech recognition, as we are classifying phones, we will use cross-entropy function. Thus, for $y_{ref}, f(x,\vec{\theta})$ the cost function L is defined as :

$$L = -\sum y_n log\, f_n(x,\vec{\theta}) \tag{3.3}$$

The training procedure must minimize the cross-entropy function. The *log* function is taken for numerical stability and math simplicity. Once we have a defined loss function, a first order optimization known as Stochastic Gradient Descent (SGD) can be computed.

3.2.1 Stochastic Gradient Descent (SGD)

The Stochastic Gradient Descent is an optimization method used for updating the weights in a neural network. To train an MLP, we learn all parameters of the model. Once we have defined the objective function $f(x,\vec{\theta})$, a forward propagation must be done to initialize the weights. Then, we compute the gradient with respect to the objective function and propagate the gradient backwards to update each layer. From a geometrical perspective, this algorithm will repeatedly make small steps (learning rate η) downward on an error surface defined by the loss function

L. In each step, the weights will be updated. The mathematical rule for the new weights will be given by:

$$w_{kl}^{'} = w_{kl} - \eta \frac{\delta L}{\delta w_{kl}}$$

$$w_{jk}^{'} = w_{jk} - \eta \frac{\delta L}{\delta w_{jk}}$$

$$w_{ij}^{'} = w_{ij} - \eta \frac{\delta L}{\delta w_{ij}} \tag{3.4}$$

Where $\{w_{ij}, w_{jk,}, w_{kl}\}$ are the previous weights, $\{w_{ij}^{'}, w_{jk,}^{'}, w_{kl}^{'}\}$ the updated weights, η the learning rate and $\{\frac{\delta L}{\delta w_{ij}}, \frac{\delta L}{\delta w_{jk}}, \frac{\delta L}{\delta w_{ij}}\}$ the gradient operator. Notice that SGD does not need to remember which examples were used in previous iterations, and it can compute features on the fly. There is one alternative called pre-conditioned SGD that we will use later in our software receipt. The use of a preconditioning matrix is a well-known technique to accelerate optimization methods. This technique is fully explained in Zhang *et al.* (2014). Instead of using a fixed learning rate, a symmetric positive definite matrix-value learning rate is defined. Also, its eigenvalues must be limited. Thus, the eigenvalues of this matrix will decrease during the training stage. This matrix shall not depend on the current training sample or we can get a non desired direction in the feature space. SGD is not the only method to train neural networks. Second order methods such as computing the Hessian matrix are also possible, but computationally expensive.

A main issue with SGD is when the neural network becomes very deep. In that case, SGD is ineffective as it can be stuck in local minima and the objective function can diverge. An extra problem related with the training stage is the dataset itself. The training data can contain information about mapping the input with the output, but some noise can be present. Once the model is fit, the noise will be still there and we are clueless about its origin: Does it belong to the data, to a particular dataset, or both?. This problem is known as "*overfitting*" and it can lead into a noise modeling. Despite of large number of solutions have been proposed, overfitting is still a problem in this generation of neural networks.

The most popular approach is the L2 regression. It suppress overfitting and it does not add too much complexity as is easy to calculate.

3.2.2 The softmax layer

For a classification problem we need to have a probabilistic output that lies on the interval [0,1]. This can be done by forcing the output of the last layer to represent a probability distribution with discrete values. Thus, final non-linearity can be solved by the softmax layer. The final MLP model is as shown in Fig. 3.4

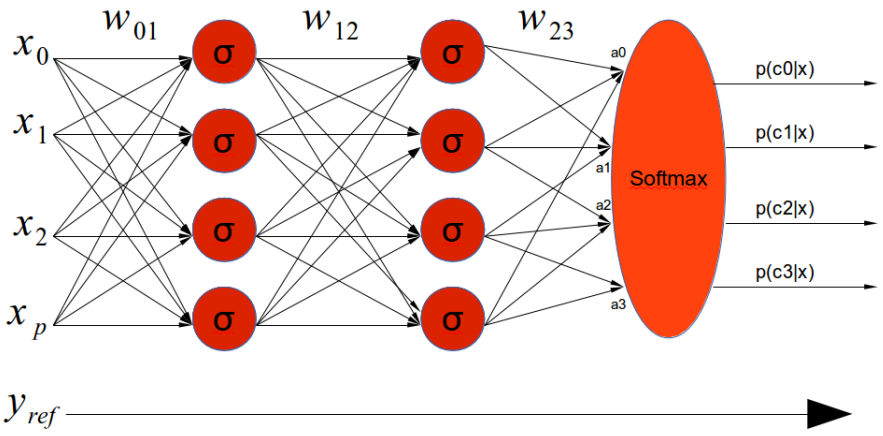

Figure 3.4: MLP with softmax layer

With softmax layer we can compute the class posterior probability $p(c_i|x)$ as:

$$p(c_i|x) = \frac{\exp(-a_i)}{\sum_{j=1}^{n} \exp(-a_j)} \tag{3.5}$$

where a_i is the *logit* (or accumulative input for the layer below) and a_j represents the *logit* for all the neurons in the softmax group. Thus, as our objective function $f(x, \vec{\theta})$ is now a posterior probability $p(c_i|x)$, we must re-define the loss function L as:

$$L = -\sum y_i^{ref} log\, p(c_i|x) \tag{3.6}$$

with$p(c_i|x)$ and y_i^{ref} belonging to the interval $[0,1]^N$. Therefore,neural networks are trained to minimize that cross-entropy function. This result leads us to use each class c_i as the basic unit to model HMM. By the Bayes' theorem, we can calculate the likelihood $p(x|c_i)$ as:

$$p(x|c_i) = \frac{p(c_i|x)}{p(c_i)} \tag{3.7}$$

The likelihood in equation 2.4 can replace the likelihood defined in GMM and it is the same equation. Thus, we can conclude that neural networks can be used as acoustic models.

3.3 The third generation of neural networks

The third generation of neural networks are called deep neural networks (DNN). DNNs were born as a result of advances in pattern recognition techniques and computer hardware. As defined in Hinton *et al.* (2012), a DNN is an MLP with more than one hidden unit between the input and the output layer. The performance of MLPs is limited by the training procedure: The objective function could diverge and SGD training is done serially on a machine, leading into a very slow solution on normal CPUs. The key to success in DNNs is the effective training procedure which is divided into two main stages:

- First, a set of units known as Restricted Boltzmann Machines are stacked together layer by layer. Pre-training is performed via unsupervised learning. The weights will be located in a good initial feature space for a posterior optimization. The stack of RBMs can be combined to build a single model defined as Deep Belief Networks (DBNs)
- Second,a *softmax* layer is added to the previous DBNs. This will create an architecture known as DNN-DBN. A "*fine tuning*" through mini-batch SGD is performed in order to adjust the pre-trained weights.

The final model will be used to predict each possible state on the HMM with the central frame of the input features. The rest of the chapter will be related with the description of the building methodology and training procedure in DNN-DBN.

3.3.1 Restricted Boltzmann Machines (RBM)

Restricted Boltzmann Machines (RBMs), deeply explained in Hinton (2010), are a graphical model that define a probabilistic function over a set of stochastic units. The upper layer is related with the learning procedure and is composed by smaller "hidden" units h that receive the data from the lower layer of visible units v. We say RBMs are "*restricted*" because there are no connections between hidden or visible units in order to make the learning process easier. The model is given as shown in

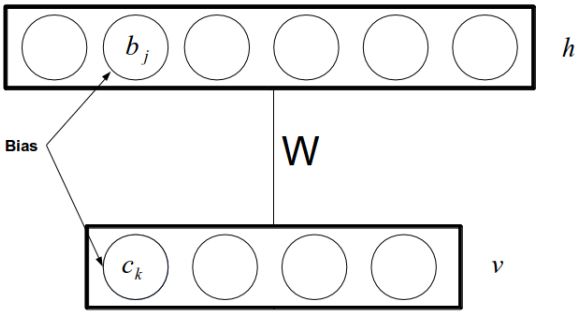

Figure 3.5: RBM architecture

Every joint configuration between visible and hidden units has an energy. Thus, we can define the energy $E(v, h)$ of a RBM model as:

$$E(v, h) = -h^T W v - c^T v - b^T h \tag{3.8}$$

where W is a matrix in which each element w_{ij} is the connection between the visible unit i and hidden unit j. The parameter h is the binary array of hidden units, v is the binary array for the visible units and c_k, b_j are the bias variables.The use of biases in a neural network increases the capacity of the network to solve problems by allowing the hyperplanes that separate individual classes to be offset for superior positioning. We can rewrite equation 3.9 as:

$$E(v, h) = -\sum_i \sum_j h_j w_{ij} v_i - \sum_i c_k v_i - \sum_j b_j h_j \tag{3.9}$$

The joint probability $p(v, h)$ within units is proportional to the energy and is defined as:

$$p(v, h) = \exp(-E(v, h))/Z \tag{3.10}$$

With $1/Z$, the "partition function" or the addition for all the pairs of hidden and visible vectors. Consequently, the probability in a given configuration of visible units $p(v)$ is computed by summing all the hidden vectors:

$$p(v) = \frac{1}{Z}\sum_{h} \exp(-E(v, h)) \tag{3.11}$$

The model will give an output probability according to the energy distribution. Furthermore, we must go a step beyond and maximize $log\, p(v)$ in order to get an effective training procedure for RBMs. Hence, by taking the average negative log-likelihood (NLL), we can maximize $log\, p(v)$ by employing the gradient operator. The cost function L can be defined as:

$$L = \frac{1}{N}\sum_{h} - log\, p(v) \tag{3.12}$$

with N the step size .Thus, we can write the gradient as:

$$-\frac{\delta \log\, p(v)}{\delta w_{ij}} = E_h[\frac{\delta E(v, h)}{\delta w_{ij}}|_v] - E_{v,h}[\frac{\delta E(v, h)}{\delta w_{ij}}] \tag{3.13}$$

where the operator $E_h[]$ is the expected value for the hidden layer given the visible units and $E_{v,h}[]$ the expected value for both, visible and hidden layers. Therefore, we have two expressions, one regarding the data and another one related with the observation. Alternatively, we can rewrite equation (3.13) as in Hinton *et al.* (2012):

$$\frac{\delta \log\, p(v)}{\delta w_{ij}} =< v_i h_j >_{data} - < v_i h_j >_{model} \tag{3.14}$$

where the operator $< . >$ is the expected value. The data part of the equation is easy to calculate. The unbiased samples of $v_i h_j$ can be obtained by computing the conditional probability within units. Due to limitations in the connections between hidden units, learning is simpler. Visible units are conditionally independent given the visible units, and hidden units are conditionally independent given the visible units. As a consequence, the conditional probability in each hidden unit h_j can be calculated as:

$$p(h_j = 1|v) = \frac{1}{1 + \exp[-(\sum_i v_i w_{ij} + b_j)]} \tag{3.15}$$

Similarly, the conditional probability in each visible unit v_i will be:

$$p(v_i = 1|h) = \frac{1}{1 + \exp[-(\sum_j h_j w_{ij} + b_i)]} \tag{3.16}$$

In contrast, the unbiased sample dealing with the model part of the equation (3.14) is more complicated. Notice from equation (3.13) that we have an exponential summatory over both vectors, v and h. As a consequence, it is computationally intractable. We need an effective method to approximate $E_{v,h}[\frac{\delta E(v,h)}{\delta w_{ij}}]$ in order to apply SGD efficiently. An efficient training method called Contrastive Divergence (CD) was proposed by Carreira-Perpinan and Hinton (2005) based on Gibbs sampling. For a training set of vectors $\{v\}$, we update all the hidden units in parallel using equation (3.15). Consequently, we must update all the visible units in parallel using equation (3.16). This procedure can be done in k steps (noted as CD_k) as shown in Fig. 3.6

As commented in Hinton *et al.* (2012) even if a large number of Gibbs sampling steps are run in order to learn better generative models, it is not an efficient procedure for pre-training because all the required parameters can be learned only in one Gibbs step. This method is called Contrastive Divergence One (CD1) and was also proposed by Carreira-Perpinan and Hinton (2005). It is faster than normal Gibbs sampling. As only one step is run, the procedure is as follows:

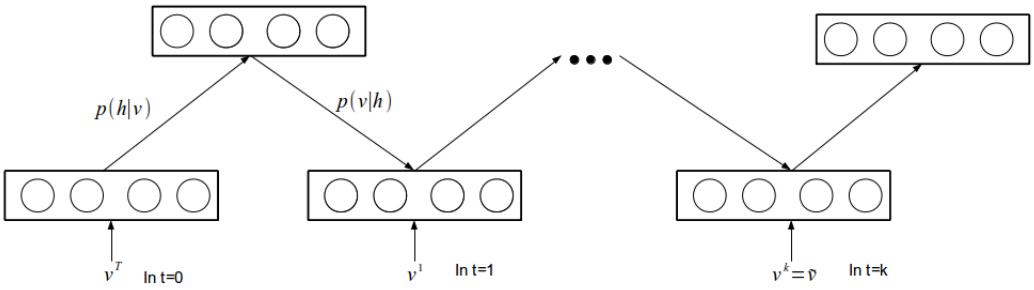

Figure 3.6: Gibbs sampling

- First, we give the training data vector to the visible units v.
- Second, the binary states in the hidden layer are changed by employing equation (3.15) .
- After, the sampled model can be obtained by setting the visible units to one and using equation (3.16) .
- Finally, the states in the hidden units will be updated once again.

For the reconstructed samples, we can derive the learning rule. Hence, the weights in the RBM will change according to:

$$W^n = W^{n-1} + \epsilon(< v_i h_j >_{data} - < v_i h_j >_{sampled}) \tag{3.17}$$

A more detailed derivation of equation (3.17) is given in Appendix A.1. CD is an effective training procedure for a RBM. Overfitting problem can be avoided in the first update of the hidden layer by taking the sampled binary values from the visible units. This is due to the property of the sampling noise as an effective noise regularizer. The second update can be computed using the real-valued probabilities instead of binary values.

3.3.2 Real data and Restricted Boltzmann Machines

Real data follow different probability distributions. Therefore, we must change the distribution in the first RBM in order to adapt the network for the input features. The inputs are generally MFCC features which can be parameterized far better with a Gaussian distribution. Hence, we must write the energy in this RBM as:

$$E(v,h) = -\sum_i \sum_j h_j w_{ij} \frac{v_i}{\sigma_i} - \sum_i \frac{(v_i - c_i)^2}{2\sigma_i^2} - \sum_j b_j h_j \tag{3.18}$$

where σ_i is the standard deviation of the Gaussian. Thus, if the energy changes, the conditional probabilities must be rewritten as:

$$p(h_j = 1|v) = \frac{1}{1 + \exp[-(\sum_i \frac{v_i}{\sigma_i} w_{ij} + b_j)]} \tag{3.19}$$

$$p(v_i = 1|h) = \mathbb{G}(c_i + \sigma_i \sum_j h_j w_{ij}, \sigma_i^2) \tag{3.20}$$

With $G(\mu, \sigma^2)$ as the Gaussian distribution. Because of the conditional probabilities, this type of RBM is called a Gaussian-Bernoulli RBM (gRBM). According to Hinton (2010), the issue in this model is related with the learning of the standard deviation as it becomes very complicated if CD1 is used. In practical applications, the normal procedure is to perform a data normalization to achieve a zero mean and unit variance. As a consequence, the variance in equation (3.18) is set to one. Thus, we can compute a reconstructed sample from the posterior probability in (3.20) without any noise.

As a final remark, the RBMs initialization has a huge impact about how the DBN-DNN behaviors. Taking equation 3.8, we can make a representation of the energy in equation 3.8 as a function of parameters. For a set of initial conditions, we can represent the energy as in Figure 3.7:

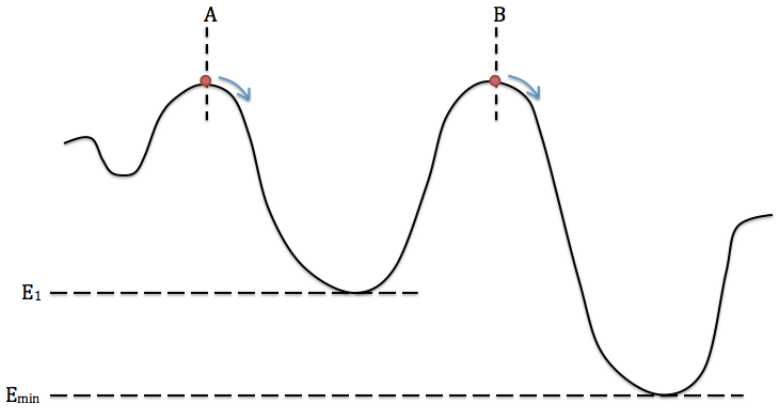

Figure 3.7: Energy function outlook

Where the red dot represents the initials conditions. We can make an analogy with "a ball" moving in the graph. Once the ball is placed in A, we want to move it to the lowest point. It is clear that when it moves, the ball can be stuck in any of the depressions without any chance to go up an reach the lowest point. If we push the ball very hard, it will reach a far point in the graph that it can not be the lowest one. In RBMs happens the same. Given a set of initial conditions (learning rate and initial parameters), the neural network tries to minimize the energy. Hence, the initial conditions must be placed in a place "nearby" the lowest point. Then, computing the gradient, it will move forward that minimum. How "fast" the ball moves in the graph is given by the learning rate. Therefore, setting an optimal initial conditions is key to guarantee a good convergence. An in depth lecture about this topic can be read in Hansen and Salamon (1990).

3.3.3 Deep Belief Network

Once we have an initial RBM, we can build a Deep Belief Network (DBN). As defined in Hinton *et al.* (2012), DBN is a "*single, multilayer generative model*". Each layer of the DBN is a RBM whose posterior probability over its hidden layer is the input for the next RBM. Thus, the idea of stacking RBMs is to improve the prior probability on the last layer just by adding another hidden layer. At the top, the last two layers are going to be connected undirected. The rest of layers will have top-down directed generative connections. Hence, the DBN is

a generative model that mix both, directed and undirected connections between variables. Finally, once the DBN is built with initial parameters, a fine-tuning operation is performed in order to adjust the weights for a particular task. A schematic overview of the building process is given in Figure 3.8

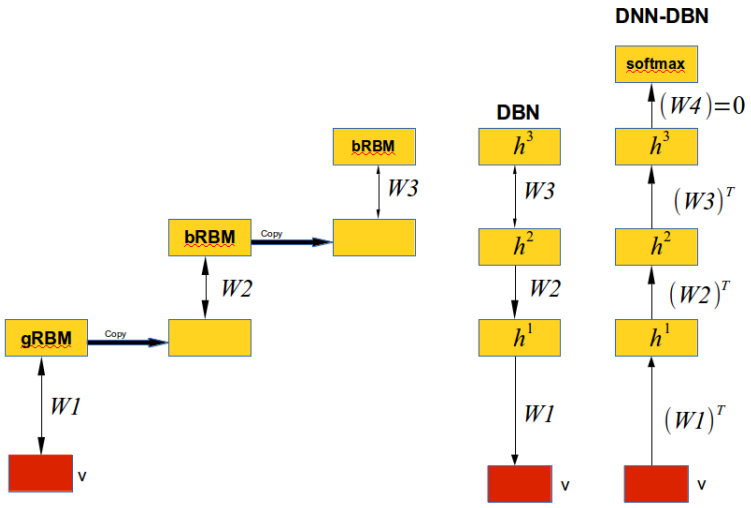

Figure 3.8: DBN building procedure

Analysis of Figure 3.8 shows a building process for a DBN with three hidden layers as proposed in Hinton *et al.* (2012). The first RBM must be a Gaussian RBM as the input are MFCC features. At this first stage, we learn W_1 by assuming that all the weights are tied. Then, we "*freeze*" W_1 and we address $(W_1)^T$ as the input data to the first hidden layer, h^1. As indicated by Hinton *et al.* (2012), we can rewrite equation (3.11) with a direct dependence of weight matrix W,

$$p(v;W) = \sum_{h} p(h;W)\, p(v|h,W) \tag{3.21}$$

where $p(h;W)$ is as in equation (3.11) but with changed roles of hidden and visible units. The training process can be done by freezing the $p(v|h,W)$ and learning the remaining weights. This is equivalent to learn another RBM using the aggregated conditional probability on h_1 as the input data. Furthermore, the resulting aggregated probability can be computed from a training sample and infering a hidden vector according to equation(3.19). This procedure will be repeated from a bottom up approach, forcing in each layer to learn "features of the

features". Each RBM can be trained using CD1 as explained in Section 3.3.2. At the end of the training, we will have a DBN with undirected connections on the two top layers and downward connections (top-down) in the remaining layers.

Stacking RBM in order to build a DNN is so successful because, as exposed inHinton (2010), each time that we add another layer of features, an improvement on the variational lower bound of the log probability is achieved. Furthermore, the bound will be defined as:

$$log\, p(x) \geq \sum_{all\, h} q(h^1|v)\, [log\, p(v) + log\, p(v|h^1) - \sum_{h^1} q(h^1|v)\, log\, q(h^1, v) \qquad (3.22)$$

where h^1 is the binary configuration on the first hidden layer, p(h^1) its prior probability and $q(\cdot|v)$ is the probability distribution defined on the binary configurations in h^1 hidden layer.Therefore, we can understand $q(h^1|v)$ as and approximation of $p(h^1|v)$. The closer $p(h^1|v)$ and $q(h^1|v)$ are, the stronger bound will be. Notice that if both are same, equation (3.22) becomes an equality. Moreover, if the difference between the two terms is very large, the approximation $q(h^1|v)$ might not be good and the bound will be weak. On the training procedure, when we freeze $p(v|h, W)$, we also keep constant the probability distribution $q(\cdot|v)$ and $p(v|h^1)$. On these conditions, the derivative of the bound will be calculated as:

$$\sum_{all\, h^1} q(h^1|v)\, log\, p(h^1) \qquad (3.23)$$

The previous equation (3.23) is very similar to train an RBM on data generated from $q(h^1|v)$. If the bound becomes very strong, the $log\, p(h^1)$ might decrease even if the lower bound increases. But the new decreased value will be limited when the probability is fixed at the beginning. At this point, the bound is tight and it will not decrease again. Therefore, notice that for many layers, a set of weights can be learn and the bound will always increment. This powerful property makes possible the improvement on the variational lower bound of the log probability.

Finally, to convert the pre-trained DBN into a DNN, we must drop all the calculated weights W_k and take the inverse direction, which is given by W_k^T. We have to add a softmax (3.8)to compute the posterior probabilities for each state in the HMMThen, we can train the DNN discriminately and use SGD in order to optimize the training.

3.3.4 Deep Neural Networks and Hidden Markov Models

DNNs can be applied as acoustic models if a previously discriminative training is performed in order to give the posterior probability for each of the states in the HMM model. As exposed in Hinton *et al.* (2012), the "*output probability for a given observation o_t at time t in utterance u_t for the HMM state s is given by*":

$$p(s|o_t) = \frac{\exp(a_u(s))}{\sum_{s^*} \exp(a_u(s*))} \tag{3.24}$$

where a_u is the activation output layer corresponding to state s. A log-likelihood for state s in observation o_t is used for recognition:

$$log\, p(o_t|s) = log\, p(s|o_t) - log\, P(s) \tag{3.25}$$

and $P(s)$ as the prior probability obtained from the data. A further explanation about how to compute $P(s)$ probability can be read in Hinton *et al.* (2012). Once we have defined $log\, p(o_t|s)$, we can apply the back propagation algorithm jointly with SGD to model the HMM states.

Chapter 4

Results

4.1 Databases Used

The performance of DNN has been tested on TIMIT, Resource Management (RM) and Mandi. RM is a standard database (Price *et al.* (1988)). It consists of 3990 training sentences related with naval resource management. It was recorded among by 168 speakers: 109 for training and 59 for testing. The sampling rates is 16Khz. The test set has been supplemented by periodic releases of speaker-independent testing data for comparative evaluation. We have implemented all these set as a whole testing set. The TIMIT database (Garofolo *et al.* (1993)) is recorded on eight principal dialects of American English. It is composed by a total number of 490 speakers: 462 speakers for training and 28 speakers for testing.

The Mandi data was collected for building Automatic IVR systems in order to get the price of Agricultural commodities in Indian languages. The speech data were collected among various states of India. This database comprises of six major Indian languages: Tamil, Telugu, Hindi, Bengali, Assamese and Marathi. Among these six languages, Hindi has been chosen for this study. All the recorded data is recorded with sampling rate of 8 KHz. The speakers were selected among farmers as the database was recorded on an rural environment. In our experiments the full databases have been separated into 1 hour, 3 hour, 5 hour and 22 hour sets to probe the performance of DNN.

4.2 The Kaldi Software Toolkit

4.2.1 What is Kaldi and why to use it

Kaldi is an open source and multi-purpose software toolkit for speech recognition. Kaldi is under Apache License v2.0. Its core is written in *C++* with some auxiliary libraries in Perl and Python. The top functions are *Shell* scripts. Kaldi has some advantages with respect to other classic toolkits such HTK or Sphinx:

- The Kaldi code is very simple and easy to understand. The design is based on compact and modular functions (feature extraction, training and decoding) which can be easily adapted and modified.
- Kaldi has complete recipes for building speech recognition systems. Thus, Kaldi can work with most of the databases in the Linguistic Data Consortium (LDC)
- The main core of Kaldi is integrated with many extensible libraries. A Finite State Transducer (FST) framework for decoding purpose is incorporated. Also, linear algebra libraries with CUDA language programming are incorporated in order to enable parallel computing in a GPU (Graphic Processor Unit)
- The vanilla version is tested against failures. The algorithms implemented are the best known solutions for speech recognition. Alternative procedures that do not work on certain cases are avoided. Hence, Kaldi is very reliable software.
- Kaldi provides integration with HTK features. HKT features can be used to train our systems in Kaldi.

All these advantages make Kaldi a very powerful tool for speech recognition and research purposes. In the subsequent sections, the main aspects of Kaldi related with data preparation, feature extraction and deep neural networks are covered.

4.2.2 Data preparation and language modeling

Before running any script, we must prepare the data. This step will allow not only to run our own databases, but also the standard ones. As commented in 4.2.1,

Kaldi is based on FST-framework. Thus, any language that can be represented as a FST graph can be used in Kaldi. If any language can not be represented as a FST, we can convert it by employing IRSTLM toolkit. Further information about this extension can be read in Povey *et al.* (2011).

Data preparation is an essential step to ensure a proper functionality of the scripts. In order to clarify further explanations, the tree directory for any generic database *"SS"* is given in Figure 4.1:

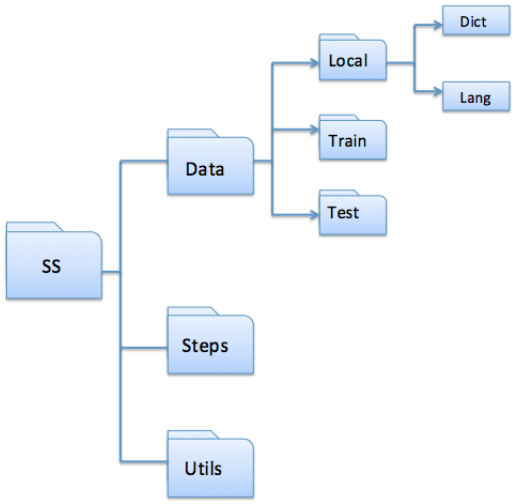

Figure 4.1: Folder tree structure in KALDI

The first stage is the configuration of the execution variables. In *SS* folder, we must configure:

- The script "*cmd.sh*" links hardware and software setup. If we are running on *IITM Libra Cluster* (refer as *Libra*), all the variables on this script must point to a file called "*queue.pl*". I would like to make an appreciation about "*queue.pl*" script. Kaldi is optimized for Sun GridEngine (SGE), an administrative solution for executing batch jobs (mainly, shell scripts) in UNIX. Thus,for *Libra*, we must check that our "*queue.pl*" file is configured with the proper SGE commands. Our laboratory configuration for "*queue.pl*" will not work as it is based on TORQUE, a different solution for queuing the jobs with different commands.
- The script "*path.sh*" links the program with the folder where Kaldi was compiled. All the path variables must point to that folder.

Once we have configured the execution variables, we must setup the data preparation. Inside "*/data*" folder, three main files have to be configured:

- The "*/local*" folder contains the dictionary for the current database. As seen in Figure 4.1, we can distinguish:
  - The "*/local/dict*" folder. Several "*.txt*" files are located inside: "*lexicon.txt*", "*non-silence.txt*" and "*silence.txt*". On these files, we can define the lexicon, the silence and non silence phonemes.
  - The "*/local/lang*" folder. It can be created using the script "*utils/prepare_lang.sh*". Here, "*words.txt*" and "*phones.txt*" are located. Based on OpenFst libraries, they represent the conversion from string to integers and back. It is needed for the FST framework. Further information about FST can be read in Mohri *et al.* (2002).

- Folder "*/train*" is related with the train dataset. Three key files are found in this folder: "*utt2spk*" gives information about the number of speakers that pronounce an utterance, "*wav.scp*" links the database with the main program when extracting features and "*text*" contains the transcriptions of each utterance.

- Structure in "*/test*" folder is similar to "*/train*", but it contains the data for testing stage.

Regarding the other folders in Figure 4.1, "*/tools*" has extra add-ons to manipulate the output. Folder "*/steps*" has all the execution files.

4.2.3 Feature extraction in Kaldi

The feature extraction in Kaldi is focused on MFCC features. Features are computed as explained in Chapter 2. Before running the extraction algorithm, we must sort all the links to the database in the file "*wav.scp*". If case we do not do this step, as random inputs are given to the program, MFCC coefficients will present small variations. This will yield in a mismatch between the baseline and the obtained results. The algorithm can be described as follows:

1. Frame the input signal with a window of 25 ms with 10 ms shift.
2. For each frame:
   - Frame preparation: Extract data, dithering, pre-emphasis, DC offset suppression and multiplication by Hamming window.
   - Compute FFT and power spectrum.
   - Triangular Mel-Filterbank with 23 triangular filter.
   - Compute the log and take 13 MFCC coefficients.
   - Liftering operation to give equal weights for all the coefficients.

After the extraction, we will get 13 dimensional MFCC features. Cepstral mean and variance normalization (CMVN) is applied to those features to achieve noise robustness.

4.2.4 Features pre-processing and models building

Neural networks will give better accuracy if MFCC features are improved. Hence, a "*pre-processing*" stage is necessary. This procedure consists on several speech recognition techniques applied on MFCC features. These techniques are explained in Chapter 2. A schematic overview of the process is given in Figure 4.2

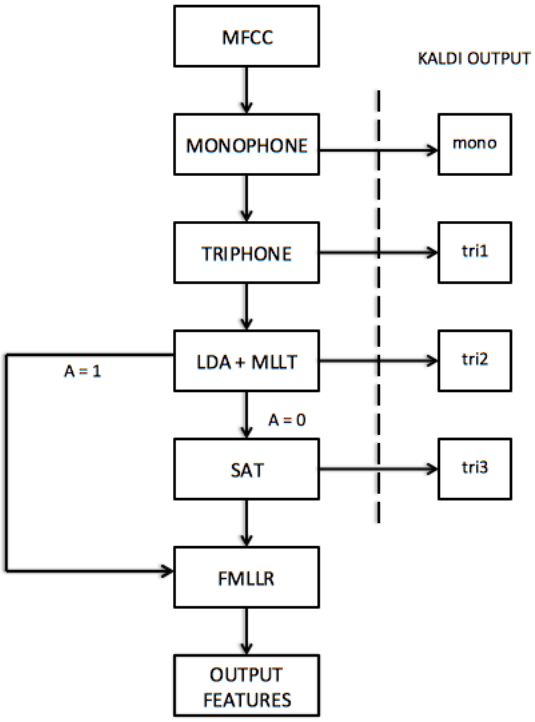

Figure 4.2: Features pre-processing

We would like to remark that procedure on Figure 4.2 corresponds to the flow execution diagram in our scripts. From the 13 MFCC features, monophone model is computed. In Kaldi, monophone system is treated as a special case of a context-dependent system, with zero phones of left and right context. Once the monophone is trained, we must get the alignment with the data in order to train the CDHMM system. This is done using expectation maximization algorithm.

Afterwards, splicing is done over these features with 9 frames before and after each center frame. LDA analysis is performed to get discriminant features with 40 dimensions and also to decorrelate the input. The output will be decorrelated again using MLLT. At this stage, a speaker adaptation (SAT) should be done. However, we must introduce a control mechanism:

- If the database is Hindi (A=1), the utterances are very short (maximum one second) and SAT algorithm does not have enough data to make a good estimation. In addition, our databases do not have any speaker label for the utterances, yielding into an imprecise adaptation.
- If we are using RM or TIMIT (A=0) we can perform SAT as the utterances are longer and the speaker labels utterances are present.

Afterwards, in both cases, we compute fMLLR to normalize the speaker variation. The whole procedure yields into a baseline of 40 dimensional fMLLR features. This baseline will be the input in the deep neural network scripts.

4.2.5 Deep Neural Networks in Kaldi

Deep Neural Networks are implemented in Kaldi from two different computational approaches: A first script for CPU architectures and a alternative one which takes advantage of parallel programming in a graphic processor or GPU.

4.2.5.1 DNN implementation on CPU

CPU implementation performs DNN training using fMLLR features extracted in Section 4.2.4. The first step is to accumulate LDA statistics. As commented in Povey *et al.* (2011), this LDA is a Kaldi script which implements a slightly different analysis fromHaeb-Umbach and Ney (1992): it is done to scale down the dimension and decorrelate the input. The main reason of this previous step is to reduce the "non-informative" part of the data which degrades the performance. By making it smaller, SGD will just ignore it.

The input features are now dumped into the disk. This will lead into several files, copied from our extracted features, and with extension "$X.Y.ark$". The X

refers to the job index and Y is the iteration index. It depends on the data and will be used during the training stage to perform that given number of iterations.

Afterwards, second stage is the neural network initialization. The activation function for each neuron is "*tanh*" type. The initial neural network will have one hidden layer and will be increased by two each iteration. The information about the initial neural network (number of nodes, activation function , mini-batch size) can be configured in an file called: *"nnet.conf"*. In addition, we can also configure the particularities of each new hidden layer in a file named "*hidden.conf*".

Next step is the training, performed by pre-conditioned SGD (refer Section 3.2.1). It is computed as a loop. The number of iterations are defined as the number of epochs plus an extra margin. Our default number of iterations are 20. On each iteration, we use different data in different jobs. We also calculate the cross-entropy yielding into an exponential decay of the learning rate to improve the accuracy. The smaller step size in the features space, more accuracy is gained. In the last iterations, it will keep constant. At the final step, we will average the models and the final fully trained deep neural network is obtained. A pseudo-likelihood is computed to use as state emission probabilities in the HMM.

4.2.5.2 DNN implementation on GPU

GPU scripts implement DNN as explained in Section 3.3.3. Training procedure is performed in CUDA. For a further explanation about the code and how deep neural network model is copied into GPU memory, please refer the Appendix A.2.

First, several files are created to split the data: 90 % of the train data is used as a proper training set and the remaining 10 % is used as cross-validation set. The activation function is given as a sigmoid. Afterwards, by taking the fMLLR features computed in Section 4.2.4, we start to stack RBMs. As the input features are Gaussian, the first RBM employs a Gaussian-Bernoulli distribution with lower learning rate. Remaining RBMs are Bernoulli-Bernoulli with same learning rate. To maintain the learning rate as constant during the training, we have to use a momentum from 0.5 to 0.9 that will be used to rescale the learning rate as:

$1 - m$. Contrastive Divergence with one Gibbs sampling step (CD1) is computed during all RBM training. Jointly, samples are taken from the data by performing sentence-level and frame-level shuffling. In addition, a L2 regularization to avoid overfitting: with a penalty factor of 0.0002 is used.

Second part of the code performs frame-level cross-entropy training. By employing the SGD with a given learning rate and a mini-batch size, we train the DNN to classify frames intro triphone states. We use sentence-level and frame-level shuffling to pick up the samples from the training dataset. When DNN improvement between two iteration is less than 0.5%: learning rate is reduced by half and the DNN is re-trained again. This procedure is repeated until improvement is less than 0.1%. The final step is to compute the posterior probability of each HMM state.

4.3 Hardware Setup

All the experiments and simulations have been run on IITM Libra Cluster. The job queue management system is based on Oracle SGE with 4.4.6 GCC compiler version. DNN computations have been performed on three NVIDIA Tesla M2070 GPU, with 6GB graphic memory and 448 CUDA cores. Feature extraction and rest of procedures were computed using Intel Xeon x5675 with 24 CPUs working at a frequency of 3.07 GHz and 1288 KB cache size.

4.4 Experimental Setup

4.4.1 Baseline Parameters

Table 4.1 shows the baseline systems for RM, TIMIT and Hindi. For all the databases, the total number of HMM states was 8. The silence is configured as context-independent phone with 5 states and the remaining states are belong to the context-dependent phones.The total number of CPU jobs was set to 20. Baseline

systems were built with a specific number of tied-states and Gaussian mixtures. For each language, the number of tied states, Gaussian mixtures, number of phones and Word Error Rate (WER) are listed in Table 4.1:

Dataset	**#Ph**	**CDHMM**							
		Triphone				**LDA+MLLT**			
		#Ts	**#Gauss**	**#Ps**	**% WER**	**#Ts**	**#Gauss**	**#Ps**	**% WER**
RM	47	1449	9017	0.71	***3.41***	1479	9020	0.71	***2.74***
TIMIT	38	402	2710	0.22	***28.38***	395	2705	0.22	***25.45***
Hindi (1hr)	42	383	1803	0.14	***14.92***	382	1806	0.14	***14.31***
Hindi (3hr)		454	2206	0.17	***11.59***	470	2204	0.17	***10.77***
Hindi (5hr)		571	4216	0.33	***9.10***	577	4212	0.33	***8.44***
Hindi (22hr)		1061	10834	0.86	***5.75***	1090	10823	0.86	***5.68***

#Ph - Number of Phones, #Ts - Number of tied states, #Gauss - Number of Gaussians, #Ps - Parameters (million)

Table 4.1: CDHMM baseline parameters

4.4.2 DNN simulations

The following subsections disseminate the obtained different results. To avoid overwhelm the reader, we have only selected the relevant results. More detailed and extended results are included in the Appendix A.3. DNN performance was evaluated for GPU and CPU on TIMIT, Hindi and RM databases. In addition, DNN on Hindi database has been studied for different parameter variations. The time has been measured for all the training stage. CPU jobs were fixed to 20. Mini-batch size was 256 an initial learning rate was 0.008 for GPU and 0.002 for CPU. Following the same pre-processing procedure discussed in Section.4.2.4, before extracting the fMLLR features, SAT was performed only in RM and TIMIT databases. Several simulations for number of parameters, hidden layers, simulation time and learning rate were conducted:

- **DNN - CPU baseline configuration**

Dataset	DNN (CPU)						
	#Hn	#Hl	#Id	#Od	#Ps	Time (min)	%WER
RM	1126	4	360	1495	7.2	143	1.92
TIMIT	793			1057	4.8	1505	21.57
Hindi (1hr)	250			384	11.1	70	13.79
Hindi (3 hr)	1761			470	11.9	126	9.86
Hindi (5 hr)	1181			472	7.6	252	6.65
Hindi (22 hr)	1741			1090	12.1	970	3.89

Hn - Number of Hidden Nodes, # Hl - Number of Hidden layers, # Id - Input dimension, # Od - Output dimension, #Ps - Parameters (million)

Table 4.2: DNN trained on CPU baseline results

- **DNN - GPU baseline configuration**

Dataset	DNN (GPU)						
	#Hn	#Hl	#Id	#Od	#Ps	Time (min)	%WER
RM	1024	6	440	1495	7.2	43	**1.74**
TIMIT	1024	6	360	1940	4.7	242	**21.39**
Hindi (1hr)	2000	5	440	384	3.6	32	**14.26**
Hindi (3 hr)	2000	7	440	470	10.6	63	**10.30**
Hindi (5 hr)	2048	6	440	698	19.11	123	**6.60**
Hindi (22 hr)	2048	6	440	712	21.3	275	**3.74**

Hn - Number of Hidden Nodes, # Hl - Number of Hidden layers, # Id - Input dimension, # Od - Output dimension, #Ps - Parameters (million)

Table 4.3: DNN trained on GPU baseline results

- **Variation of number of input parameters and simulation time**

# Ni	Hindi (1hr)		Hindi (3hr)		Hindi (5hr)		Hindi (22hr)	
	Time (min)	% WER	Time (min)	% WER	Time (min)	% WER	Time (min)	% WER
3	45	15.95	105	10.57	109	7.00	719	3.98
4	54	14.06	123	9.86	240	6.77	838	3.97
6	70	13.79	166	10.31	306	7.71	1190	3.89
8	92	16.58	248	10.53	345	7.14	1360	3.91
10	110	14.37	271	10.29	472	7.20	1549	3.92

Ni - Number of input parameters (million)

Table 4.4: DNN trained on GPU baseline results

- **Tuning the learning rate**

# Lr	Hindi (1hr)			Hindi (3hr)			Hindi (5hr)		
	#Ps	Time (min)	% WER	#Ps	Time (min)	% WER	#Ps	Time (min)	% WER
0.001	11.1	77	15.66	15.8	158	11.69	7.6	171	6.82
0.002	11.1	70	13.79	11.9	126	9.86	7.6	252	6.65
0.003	8.5	123	17.21	15.9	156	9.99	7.6	191	6.74
0.004	8.4	60	14.53	15.9	159	10.39	7.6	124	6.98

#Lr - Number of input parameters (million), #Ps - Number of input parameters (million)

Table 4.5: Variation of learning rate

4.5 Discussion

4.5.1 Baseline CDHMM results

The Table 4.1 shows the baseline parameters for the CDHMM system. The results for *"LDA+MLLT"* give improved results when compared to triphone model for all of the cases. As instance, for Hindi 5 hours, we achieve an improvement of 7.25% over basic triphone model. Also, as the database grows in time, the number of parameters also increases as the data needed to be estimated is larger. This baseline is used as basic features to build the DNN systems.

4.5.2 DNN baseline results

Table 4.1 and 4.3 gives the best DNN results for the baseline. Substantial reductions in the WER are achieved in DNN. A closer look at Table 4.2, when comparing DNN with *"LDA+MLLT"* system, the DNN on a CPU gives a relative improvement of 15.24% and 29.92% for RM and TIMIT respectively. Moreover, if we contrast the results with triphone model, an enhancement of 43.69% for RM and 23.99% in TIMIT are obtained. For Hindi databases, DNN also improves the baseline results. Comparing with "*LDA+MLLT*", in small datasets, Hindi 1 hour improves 3.63% and Hindi 3 hours does 8.44%. Furthermore**,** improvement is up to 21.21% and 31.51% for 5 and 22 hours respectively.

But the most astonishing results are for the GPU implementation in bigger datasets. An almost identical results of 1.74% and 0.35% were achieved in Hindi 1 hour and Hindi 3 Hours when compared with "*LDA+MLLT*". However, the DNNs that performed best on the validation belong to larger databases. Comparing with "*LDA+MLLT*" system, relative improvement in Hindi 22 hrs goes up to 21.8%, meanwhile in Hindi 5 hours is 34.15%.

Furthermore, the number of parameters give an important observation. First, the number of parameters needed in the models will increase as the database becomes larger. This is due to the estimation of DNN to converge into a solution with successful accuracy.

We notice that for Hindi in 1 hour on GPU, even for a good performance, DNN parameters decrease up to 3.6 million, meanwhile in CPU the number of parameter is 11.1 million. This is due to the configuration of the scripts. As commented in Section4.2.5, in CPU we add two hidden layer per iteration. If the improvement is good, then we continue the training, if not, we re-initialize the weights. The procedure of adding layers an re-estimating parameters yields into a slower, but better convergence. In contrast, when we train on GPU, it will take full advantage of the parallelism and it will converge faster. For Hindi 1 and 3 hours, the difference in performance is due to the random initialization of the weights done by Kaldi at the beginning of the training. Even if we have learning rate or

topology constant, it leads into an unwanted convergence. That can be a problem for small datasets. Notice that for small datasets, the way GPU script uses to initialize the neural network is not good and performance is degraded. When the data is increased, the GPU computation is able to do a better estimation as it has enough data to re-train the DNN effectively.

Notice the huge improvement regarding the simulation time between CPU and GPU for all the databases. From Table 4.2 and Table 4.3, we can observe that the simulation time is almost halved. Experiments show that DNN computation on GPU when the dataset is large is a better option: Less number of parameters and better accuracy can be achieved.

As a remark, notice that in Table 4.2 and Table 4.3 when the number of parameter increases, the complexity of the network will also increase. This heads into an increment of the model size when dumping models on disk. We must be extremely careful about this fact if we are training the DNN on a GPU. Kaldi allocates the whole DNN with the initial number of parameters inside the GPU memory (refer Appendix A.2). When stacking the RBMs, the model size increases. If we do not have enough GPU memory, a segmentation error is thrown. As we are simulating on *Libra* with three graphic cards, the total memory is up to 18 GB and it will not be a major issue. But if the simulation is done on a single GPU with less graphic memory, training the DNN can become impossible. A good practice to avoid this is to train the stack of RBMs on the CPU and, if size of the neural network is less than the GPU memory, then allocate the resulting DBN inside the GPU.

4.5.3 Initialization and Convergence

The simulations in Table 4.4 are conducted to determine how the accuracy of the DNN changes regarding the number of input parameters. The learning rate was 0.002 and the number of initial hidden layers are fixed by 2. The time needed for convergence is also recorded. We can observe two important facts. As we increase the number of parameters, the time to reach stable results will also increase.

Second, the initial number of parameters also plays a fundamental role in DNN behavior. As discussed in Figure 3.7, if we initialize the neural network with some random parameters that are not near the convergence region, it will be stuck in any local minima. Simulations in Table 4.4 show that for Hindi language. As an illustration, we can notice that for every number of parameters, in Hindi 5 hours dataset, it get stuck in local minima, except in 4 million parameter which is the optimal case. Further simulations (refer Appendix A.3) do not show any major improvement as we are introducing redundant information and it will increase the simulation time. Same discussion is extendable for the other datasets. In addition, notice the big performance jump in Hindi 1 hour with 8 million parameter as initial configuration. In this case, performance is very poor due to an overfitting problem. When shuffling frames at the beginning of the training, it is taking noisy frames, leading into a noise modulation. This yields in wrong initial configuration and hence, in an erroneous convergence.

4.5.4 Variation of learning rate and hidden layers.

Experiments on Table 4.5 shows the variation on the learning rate, simulation time and accuracy. These simulations were made by taking the best DNN architecture in Table 4.2. Notice that the change is very dramatic. As commented in Chapter 3, the learning rate gives the magnitude of the jump in the error surface of the cost function. If the learning rate is increased, the weights of the neural network oscillates, moving downhill or uphill, leading into an erroneous convergence. For example, for Hindi 3 hours, the number of parameters oscillates around 15.9 million, and only at the best result case, the parameters decrease properly because the solution is optimal. In addition, if decreased, oscillations happen too. Moreover, if the learning rate is too low, the jump in the error surface is smaller and simulation time will grow.

Chapter 5

Conclusions

This thesis has addressed the question of whether deep neural networks can be useful as acoustic models in automatic speech recognition systems. We succeed in that task by showing that they can if they are optimized properly. For many years, HMM's solutions have been proved to be effective. However, they suffer several drawbacks which yield into poor discrimination. Given those drawbacks, new alternatives have to be studied.

Deep Neural Networks are becoming the state-of-the-art technique in speech recognition. By imitating the human brain, they are well known for their ability to learn from the features, tolerate noise, and support parallelism. Thus, DNN's will be a future success in ASR systems.

We have explored two different approaches to use DNNs as acoustic models. The first approach is based on CPU computation. Experiments conducted on this architecture shows a relative improvement of 43.6 % over CDHMM's solutions for Resource Management and 23.9 % for TIMIT. For a small Hindi dataset of 1 hour, a improvement of 7.59 % is achieved with respect CDHMM.

The second approach is based on parallel computing on a GPU. This approach proved much more successful in terms of results and speed. DNNs takes advantage of parallel computing and outperforms the previous results on CPU. A relative improvement of 48.9 % for RM and 24.63 % for TIMIT on GPU architecture has been achieved. In addition, for Hindi language, an improvement of in Hindi 5 hrs with 25.3 % and Hindi 22 hours with 34.4 % were achieved. With our results, we support the previous research on this field and we show that DNNs stand as an excellent solution for building ASR systems in Hindi language.

Future lines of research can focus on the implementation of better training procedures, feature improvement and neural networks architectures. However, we

believe that deeper research into neuro-science theories about how human brain learns will lead into new computational techniques that will overtake the actual solutions.

Appendix A

A.1 Derivation of the learning rule for RBM

The learning rule for an RBM in equation 3.17 can be expressed as:

$$W^n = W^{n-1} - \epsilon(\nabla_W(-log\, p(x))) \tag{A.1}$$

According to 3.13, we can rewrite:

$$W^n = W^{n-1} - \epsilon(E_h[\frac{\delta E(v,h)}{\delta w_{ij}}|_v] - E_{v,h}[\frac{\delta E(v,h)}{\delta w_{ij}}]) \tag{A.2}$$

where the first term is related with the data and the second concerns the model. We commented in 3.14 the need to make a sample from the model in order to compute equation A.2. Thus,

$$W^n = W^{n-1} - \epsilon(E_h[\frac{\delta E(v,h)}{\delta w_{ij}}|_v] - E_h[\frac{\delta E(v,h)}{\delta w_{ij}}|_{\tilde{v}}]) \tag{A.3}$$

In order to derive the learning rule, we need to compute the derivative of the energy. Thus, taking the energy function in 3.9:

$$\frac{\delta E(v,h)}{\delta w_{ij}} = \frac{\delta}{\delta w_{ij}}(-\sum_i\sum_j h_j w_{ij} v_i - \sum_i c_k v_i - \sum_j b_j h_j) \tag{A.4}$$

As the derivative of the biases do not depend on the weights , they will be zero. EquationA.4 becomes:

$$\frac{\delta E(v,h)}{\delta w_{ij}} = \frac{\delta}{\delta w_{ij}}(-\sum_i\sum_j h_j w_{ij} v_i) \tag{A.5}$$

Therefore

$$\frac{\delta E(v,h)}{\delta w_{ij}} = -h_j v_i \tag{A.6}$$

Hence, from A.2, we can say :

$$\begin{aligned} E_h[\frac{\delta E(v,h)}{\delta w_{ij}}|_v] &= E_h[\frac{\delta E(v,h)}{\delta w_{ij}}|_v], \\ &= \sum_{h_j} -h_j v_i p(h_j|v) \end{aligned} \tag{A.7}$$

By defining $h(v)$ as the activation sigmoid function, from previous results in equationA.7 we can rewrite A.3 as :

$$W^n = W^{n-1} + \epsilon(h(v)v^T - h(\tilde{v})\tilde{v}^T) \tag{A.8}$$

which is the same as equation 3.17

A.2 Parallel Computation: A CUDA tutorial

In this section, we present a brief overview of CUDA programming application in Kaldi. More specifically, we will focus on how the matrix is allocated on the GPU. It does not intend to be an in depth description, but a helpful one for any reader who wants to work with Kaldi and DNN on a GPU giving some basic idea. For further explanations, we suggest to visit read related literature (Nukada and Matsuoka (2009))

A.2.1 CUDA brief overview

CUDA or **C**ompute **U**nified **D**evice **A**rchitecture is a C/C++ extension optimized for GPU computing exclusively on NVIDIA graphic cards. CUDA defines a computing dual paradigm : Part of the code which is run by the host (CPU) and remaining part of the code is run by the device (GPU) in a "*kernel*". It also proposes a hierarchical execution model:

- The *kernel* or function executed on the GPU as an array of threads in parallel with an unique ID.
- Hierarchy can be resumed as:
- When we are computing matrix multiplication, it takes not single elements but full rows and columns, which are processed in parallel.

A memory model divided in Shared (only host can write) , Local (only for devices) and Global (host and devices) memory is also defined.

A.2.2 Copying the DNN into the GPU

The DNN and the RBM must be allocated to inside the GPU if trained. The transfer to the memory happens when a class object CuMatrix is created. If we are working with vectors, we use Then, we copy the data into the GPU memory using the special command cudaMemcpy. The general procedure used by Kaldi when is trying to allocate a matrix "data" inside the GPU is shown below:

Algorithm A.1 Kaldi GPU allocation

We check if there is any GPU active CuDevice(*char * name, int32 len, int32 dev*)

if (device)

- Select the GPU *CuSelectGPUId(* GPU_id *)*
- Copy the matrix *cudaMemCopy(data , sizeof (data), cudaMemCpyhostToDevice)*
- Perform computations on GPU by executing kernels.
- Get the data back *cudaMemCopy (data , sizeof (data), cudaMemCpydeviceToHost)*
- Clean the memory *cudaFree(* GPU_id *)*

A.3 Additional tables

Tables regarding the initial number of parameters and the simulation time for datasets.

# Ni	Hindi (1hr)		Hindi (3hr)		Hindi (5hr)	
	Time (min)	% WER	Time (min)	% WER	Time (min)	% WER
1	28	14.78	86	10.14	157	7.21
2	39	14.32	88	10.45	187	6.60
3	45	15.95	105	10.57	109	7.00
4	54	14.06	123	9.86	252	6.65
5	62	14.69	138	10.49	289	7.19
6	70	13.79	166	10.31	306	7.71
7	85	14.38	217	11.46	323	7.50
8	92	16.58	248	10.53	345	7.14
9	95	14.88	242	10.41	404	7.12
10	110	14.37	271	10.29	472	7.20

Ni - Number of input parameters per million.

Table A.1: WER variation for Hindi - 1hr, Hindi - 3hr and Hindi - 5hr

# Ni	Hindi (22hr)	
	Time (min)	% WER
3	719	3.98
4	838	3.97
5	965	3.89
10	2110	3.92
20	2900	4.21
25	2983	4.17

Table A.2: WER variation for Hindi - 1hr, Hindi - 3hr and Hindi - 5hr

Bibliography

1. **Anastasakos, T.**, **J. McDonough**, **R. Schwartz**, and **J. Makhoul**, A compact model for speaker-adaptive training. *In Spoken Language, 1996. ICSLP 96. Proceedings., Fourth International Conference on*, volume 2. IEEE, 1996.

2. **Carreira-Perpinan, M. A.** and **G. E. Hinton**, On contrastive divergence learning. *In Proceedings of the tenth international workshop on artificial intelligence and statistics*. Society for Artificial Intelligence and Statistics NP, 2005.

3. **De la Torre, A.**, **A. M. Peinado**, **J. C. Segura**, **J. L. Pérez-Córdoba**, **M. C. Benítez**, and **A. J. Rubio** (2005). Histogram equalization of speech representation for robust speech recognition. *Speech and Audio Processing, IEEE Transactions on*, **13**(3), 355–366.

4. **Garofolo, J. S.**, **L. D. Consortium**, *et al.*, *TIMIT: acoustic-phonetic continuous speech corpus*. Linguistic Data Consortium, 1993.

5. **Ghoshal, A.**, **D. Povey**, **M. Agarwal**, **P. Akyazi**, **L. Burget**, **K. Feng**, **O. Glembek**, **N. Goel**, **M. Karafiát**, **A. Rastrow**, *et al.*, A novel estimation of feature-space mllr for full-covariance models. *In Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE, 2010.

6. **Haeb-Umbach, R.** and **H. Ney**, Linear discriminant analysis for improved large vocabulary continuous speech recognition. *In Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 1. IEEE, 1992.

7. **Hansen, L.** and **P. Salamon** (1990). Neural network ensembles. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **12**(10), 993–1001. ISSN 0162-8828.

8. **Hinton, G.** (2010). A practical guide to training restricted boltzmann machines. *Momentum*, **9**(1), 926.

9. **Hinton, G.**, **L. Deng**, **D. Yu**, **G. E. Dahl**, **A.-r. Mohamed**, **N. Jaitly**, **A. Senior**, **V. Vanhoucke**, **P. Nguyen**, **T. N. Sainath**, *et al.* (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, **29**(6), 82–97.

10. **Leggetter, C. J.** and **P. Woodland** (1995). Maximum likelihood linear regression for speaker adaptation of continuous density hidden markov models. *Computer Speech & Language*, **9**(2), 171–185.

11. **McCulloch, W. S.** and **W. Pitts** (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, **5**(4), 115–133.

12. **Minsky, M. L.** and **S. A. Papert**, *Perceptrons - Expanded Edition: An Introduction to Computational Geometry*. MIT press Boston, MA:, 1987.

13. **Mohamed, A.-r.**, **G. E. Dahl**, and **G. Hinton** (2012). Acoustic modeling using deep belief networks. *Audio, Speech, and Language Processing, IEEE Transactions on*, **20**(1), 14–22.

14. **Mohri, M.**, **F. Pereira**, and **M. Riley** (2002). Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, **16**(1), 69–88.

15. **Nukada, A.** and **S. Matsuoka**, Auto-tuning 3-d fft library for cuda gpus. *In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009.

16. **Povey, D.**, **A. Ghoshal**, **G. Boulianne**, **L. Burget**, **O. Glembek**, **N. Goel**, **M. Hannemann**, **P. Motlicek**, **Y. Qian**, **P. Schwarz**, *et al.*, The kaldi speech recognition toolkit. *In Proc. ASRU*. 2011.

17. **Povey, D.**, **H.-K. J. Kuo**, and **H. Soltau**, Fast speaker adaptive training for speech recognition. *In INTERSPEECH*. 2008.

18. **Price, P.**, **W. M. Fisher**, **J. Bernstein**, and **D. S. Pallett**, The darpa 1000-word resource management database for continuous speech recognition. *In Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*. IEEE, 1988.

19. **Rabiner, L.** and **B.-H. Juang** (1986). An introduction to hidden markov models. *ASSP Magazine, IEEE*, **3**(1), 4–16.

20. **Rabiner, L.** and **R. Schafer**, *Theory and Applications of Digital Speech Processing*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2010, 1st edition. ISBN 0136034284, 9780136034285.

21. **Rosenblatt, F.** (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, **65**(6), 386.

22. **Woodland, P.**, **J. J. Odell**, **V. Valtchev**, and **S. J. Young**, Large vocabulary continuous speech recognition using htk. *In Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, volume ii. 1994. ISSN 1520-6149.

23. **Zhang, X.**, **J. Trmal**, **D. Povey**, and **S. Khudanpur** (2014). Improving deep neural network acoustic models using generalized maxout networks. *submitted to ICASSP*.