National Technical University of Athens
School of Electrical and Computer Engineering
Division of Signals, Control and Robotics

A manifold-regularized, deep neural network acoustic model for automatic speech recognition

Diploma Thesis

of

IOANNIS M. CHALKIADAKIS

Supervisor: Alexandros Potamianos
Professor, N.T.U. of Athens

Athens, EXAM DATE

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Signals, Control and Robotics

A manifold-regularized, deep neural network acoustic model for automatic speech recognition

Diploma Thesis

of

IOANNIS M. CHALKIADAKIS

Supervisor: Alexandros Potamianos
Professor, N.T.U. of Athens

Approved by the examining committee on EXAM DATE.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Alexandros Potamianos	Petros Maragos	Shrikanth S. Narayanan
Professor, N.T.U. of Athens	Professor, N.T.U of Athens	Professor, U.S. California

Athens, EXAM DATE

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Signals, Control and Robotics

.....
Ioannis M. Chalkiadakis
Diploma of Electrical and Computer Engineering, National Technical University of Athens

Acknowledgements

Having finished the current project I would like to express my gratitude towards and thank everyone who helped me during my work.

First of all I would like to thank my supervisor Professor Alexandros Potamianos for his support for the choice of the project, his advice and help as well as trust and continuous support at times when progress was not as fruitful as expected.

Furthermore, I would like to thank Professor Aris Koziris, and the members of the Computing Systems Laboratory, especially Dimitris Siakavaras, who granted me access to their technical equipment and helped me set up the necessary experiments. Their contribution was crucial to the completion of the project.

I would also like to thank Yannis Klasinas for his technical advice on an important part of the project, which allowed me to progress faster.

Finally, I would like to deeply thank my family and friends for their help, support and patience over the whole course of my studies.

Abstract

The goal of the current project was to study deep architectures of neural networks which have received tremendous attention during the past few years, because of their success in tasks of interest to the machine learning community.

The application field that we selected was automatic speech recognition, given that most breakthroughs in the deep learning research have first occurred in speech recognition tasks. In addition, we adopted a manifold approach for the regularization of the training criterion of the network. The idea (Tomar and Rose, 2014) is that, if we manage to maintain the manifold-constrained relationships of speech input data through the network, we will learn a more accurate and robust against noise distribution over speech units. The algorithm that will maintain the manifold-imposed relations uses classes of speech units and distances between speech features to learn the underlying manifold.

The first chapter is a mathematical background refresher to introduce the notion of the manifold, in order to help with the understanding of the manifold regularization and why it works.

Next, we proceed to present a general, brief overview of automatic speech recognition, as well as current approaches in the field.

The third chapter is an introduction to manifold learning, where we define the area and give an overview of popular manifold learning algorithms. In the last section of this chapter, we present the work of Tomar and Rose, 2014, on which the current project was based.

Chapter four presents deep neural networks; it starts with the history and motivation behind deep architectures in speech recognition and proceeds with the description of the deep multilayer perceptron, which was the architecture of choice in the project. Practical tips and tricks are also given as they were collected during the development and experimental part of the project. The chapter concludes by briefly presenting recurrent neural networks, a type of network that has received a lot of attention of the deep learning community.

The last chapter describes in detail the manifold regularized network we built, the way we incorporated the manifold criterion in the deep neural network as well

as challenges we faced during development. Finally, experimental results and subsequent remarks are presented.

Keywords

deep neural networks, machine learning, manifold learning, manifold regularization, intrinsic graph, penalty graph, approximate nearest neighbors, kd-trees, coordinate patch, automatic speech recognition, large vocabulary continuous speech recognition, acoustic modeling, tandem acoustic modeling, hybrid acoustic modeling, Theano, Julia, Kaldi

Contents

List of Figures

Chapter 1

Mathematical Background

1.1 Introduction

An important part of every pattern recognition project handling large amounts of data, is the dimensionality reduction problem, that is, the transfer of the input data into a lower dimensional space, which will allow for more efficient processing and lower model error, as fewer parameters will have to be estimated. In order to develop algorithms for this task, one must be able to understand basic notions often arising in such context, e.g. Hilbert space, convergence, metric, manifolds. The purpose of this chapter is to offer a brief introduction to these concepts in connection with the broader topic of the project([Kre78]).

1.2 Metric Spaces

A *space* in the broader mathematical sense, is a set of elements X with some added structure.

Specifically, we will be working on *metric spaces*. Prior to defining a *metric space* we must first define what a *metric* is.

Definition 1. *A metric d on a set X (or distance function on X) is a function defined on $X \times X$, such that for all x,y,z $\in X$,the following properties hold true:*

- *d is real-valued, finite and non-negative*
- *d(x,y) = 0 iff $x = y$*
- *d(x,y) = d(y,x) (Symmetry)*
- *d(x,y) $\leq$ d(x,z) + d(z,y) (Triangle Inequality)*

Definition 2. *A metric space is a pair (X,d) where X is a set and d is a metric on X.*

Set X is also called the *underlying set* of (X,d).For the fixed points x,y the non-negative number *d(x,y)* is called the distance from x to y.

Examples of metric spaces are the well known $\mathbb{R}$ with the distance function

$$d(x, y) = |x - y|$$

, and the Euclidean space $\mathbb{R}^2$ with the Euclidean metric, defined by

$$d(x, y) = \sqrt{(x_1 - y_1) + (x_2 - y_2)}$$

for elements $x = (x_1, x_2)$ and $y = (y_1, y_2)$.

Spaces and their structure play a vital role in dimensionality reduction.The goal of this process is to transfer the input space into a lower dimensional one, yet at the same time retain the original relations between input data. In order to do this, we need to make sure that the space we move to, has certain properties.We are especially interested in spaces where the triangular inequality is satisfied, so that we can take advantage of the convergence properties it offers to the space.This is the reason why we built on a metric space, adding properties and operations, to derive new spaces which offer the required structure.

We will now introduce some auxiliary concepts which will enable a smooth transition into further defining kinds of spaces.

Definition 3. *Given a point $x_0 \in X$ and a real number $r > 0$, we define three types of sets:*

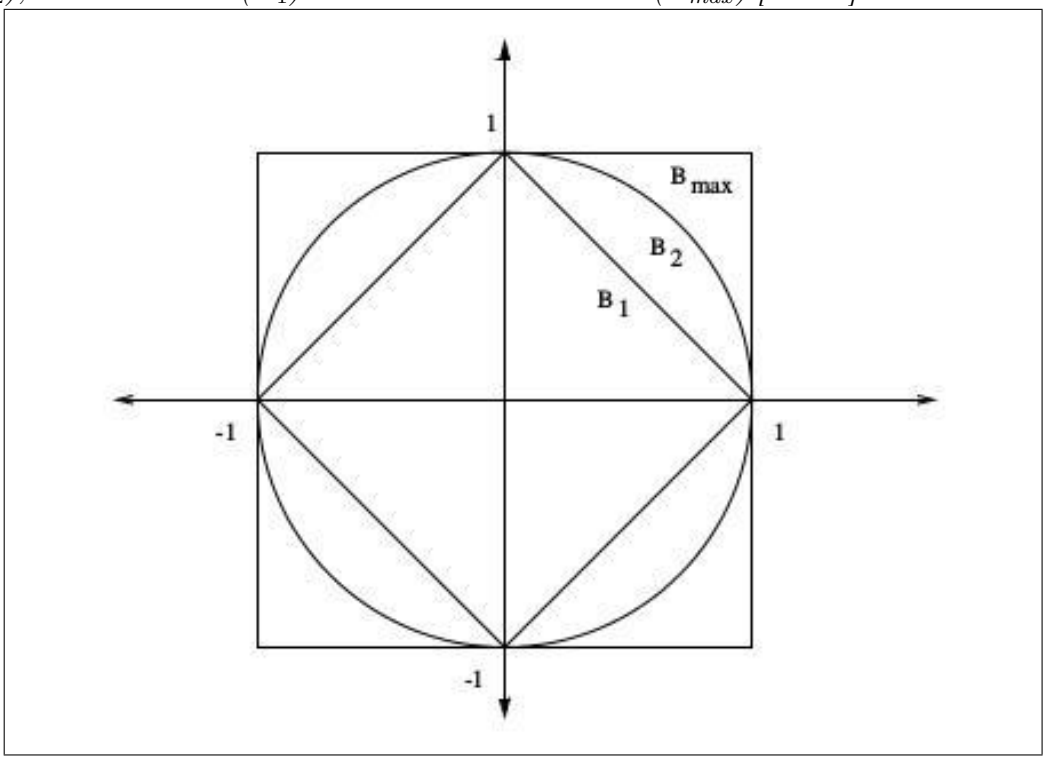
- *$B(x_0;r) = \{x \in X \mid d(x,x_0) < r\}$ (Open Ball)*
- *$B(x_0;r) = \{x \in X \mid d(x,x_0) \leq r\}$ (Closed Ball)*
- *$S(x_0;r) = \{x \in X \mid d(x,x_0) = r\}$ (Sphere)*

where point x_0 is called the center and r the radius.

Given the definition of a *ball*, we can define the notion of a neighborhood of a point.

Definition 4. *An open ball $B(x_0, \varepsilon)$, $\varepsilon > 0$, is called an ε - neighborhood of x_0. A neighborhood of x_0 is any subset of X which contains an ε - neighborhood of x_0.*

Figure 1.1: *The unit balls in the euclidean space defined using the euclidean norm (B_2), the sum norm (B_1) and the maximum norm (B_{max}).[Kre78]*



Since the process of dimensionality reduction is ,in its essence, a mapping from one space to another, we proceed by defining the notion of a continuous mapping.

Definition 5. *Let X = (X,d) and Y = (Y,$\bar{d}$) be metric spaces.A mapping $T : X \to Y$ is said to be continuous at a point $x_0 \in X$ if for every $\varepsilon > 0$ there exists a $\delta > 0$ such that*

$$\bar{d}(Tx, Tx_0) \ < \ \varepsilon \ \forall \ x \ satisfying \ d(x, x_0) \ < \ \delta$$

T is said to be continuous if it is continuous at every point of X.

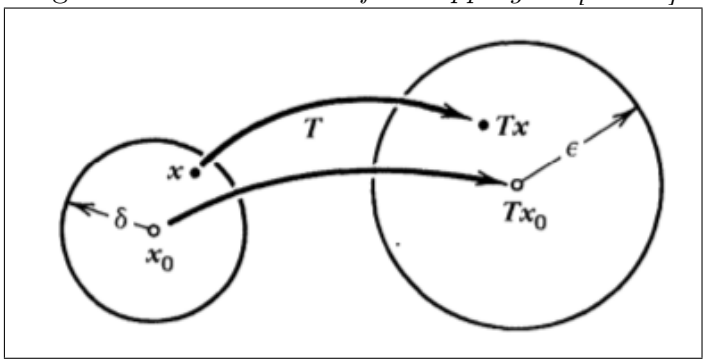
Having presented the ideas of a metric space and some of its subsets, as well as defined a continuous mapping, we proceed to remind the notions of convergence and completeness which will be used to move on to further spaces.

1.3 Convergence and completeness

Definition 6. *A sequence (x_n) in a metric space X = (X,d) is said to converge or to be convergent if there is a $x \in X$ such that*

$$\lim_{n\to\infty} d(x_n, x) = 0.$$

Figure 1.2: *Illustration of a mapping T. [Kre78]*



x is called the limit of x_n and we write

$$\lim_{n\to\infty} x_n = x \text{ or } x_n \to x.$$

It is obvious now why we need to be in a metric space to define the convergence of x_n: the metric d produces the sequence $a_n = d(x_n, x)$ whose convergence defines that of x_n.Furthermore, using the triangular inequality we can prove the following lemma:

Lemma 1. *Let X=(X,d) be a metric space.Then:*

- *A convergent sequence in X is bounded and its limit is unique.*
- *If $x_n \to x$ and $y_n \to y \in X$, then d(x_n, y_n)→ d(x,y).*

Convergence plays an important role in the definition of completeness which will help us later to define Banach and Hilbert spaces (where we mostly work).

Definition 7. *Let x_n be a sequence in $\mathbb{R}$ or $\mathbb{C}$. x_n converges and we call it a Cauchy sequence if and only if it satisfies the Cauchy convergence criterion,that is, if and only if for every given $\varepsilon > 0$ there is a N=N(ε) such that*

$$|x_m - x_n| < \varepsilon \quad \forall m, n > N.$$

However, this is the case only for $\mathbb{R}$ and $\mathbb{C}$. Given that most pattern recognition tasks work on multidimensional spaces, we have to generalize this convergence property in such spaces.

Definition 8. *A sequence x_n in a metric space X = (X,d) is said to be Cauchy (or fundamental),if for every $\varepsilon > 0$ there is a N=N(ε) such that*

$$d(x_m, x_n) < \varepsilon \quad \forall m, n > N.$$

This generalization allows us to give the following definition of completeness:

Definition 9. *A metric space X = (X,d) is said to be complete if every Cauchy sequence in X converges (that is, it has a limit which is an element of X).*

It is important to notice that convergence is not a property of the sequence by itself, but it also depends on the metric space that the sequence lies: the limit of the sequence must be *in the space*.

(EXAMPLES NEEDED??)

1.4 Vector spaces and Banach spaces

As we explore the structure of spaces so that we reach a suitable one for our problem, we come across the notion of *vector spaces*.

Definition 10. *A vector space (or linear space) over a field K is a non-empty set X of elements x,y,... ,which are called vectors, together with two algebraic operations : vector addition and multiplication of vectors by scalars, that is by elements of K.*

Vector addition associates with every ordered pair (x,y) of vectors, a vector x+y, called the sum of x+y, in such a way that the following properties are satisfied:

$$x + y = y + x$$

$$x + (y + z) = (x + y) + z$$

Furthermore, there exists a vector θ, called the zero vector, and for every vector x there exists a vector -x, such that for all vectors we have:

$$x + \theta = x$$

$$x + (-x) = \theta$$

Vector multiplication by scalars associates with every vector x and scalar α a vector αx (or $x\alpha$) called the product of α and x in such a way that for all vectors x,y and scalars α, β the following hold:

$$\alpha(\beta x) = (\alpha\beta)x$$

$$1x = x$$

$$\alpha(x + y) = \alpha x + \alpha y$$

$$(\alpha + \beta)x = \alpha x + \beta x$$

A vector space X exactly as defined above, may or may not be a metric space. To make sure that a relation between the algebraic structure of X and the metric exists, and thus be able to combine algebraic and metric concepts, we have to define on X a metric d based on a *norm*.

Definition 11. *A norm on a vector space X (over* $\mathbb{R}$ *or* $\mathbb{C}$*) is a real-valued function on X whose value at a* $x \in X$ *is denoted by*

$$\|x\|$$

and which has the properties

$$\|x\| \geq 0$$

$$\|x\| = 0 \Rightarrow x = 0$$

$$\|\alpha x\| = |\alpha|\|x\|$$

$$\|x + y\| \leq \|x\| + \|y\|$$

where x,y $\in X$ *and* $\alpha \in K$.

A norm on X defines a metric d on X which is given by

$$d(x, y) = \|x - y\| \quad x, y \in X$$

and is called the metric induced by the norm.

A vector space X with a norm defined on it, is called a normed space and is denoted by X or $(X,\|\cdot\|)$.

If a normed space is complete in the metric defined by the norm, we call it a *Banach space*.

1.5 Inner product spaces and Hilbert spaces

In a vector space, vectors can be added and multiplicated giving the space its algebraic properties. By defining a norm on such a space, the concept of the length of a vector is generalized allowing us to define a metric and establish a relation between the algebraic and geometrical structure of the space.

To connect the above with pattern recognition, one should notice that when the problem is transferred to a lower dimensional space, it is often desirable to keep not only the length/magnitude relations between the data (as represented by the distance/norm for example) but also the angle between them.This reminds us of the inner product of a Euclidean space, which is what we will expand over normed spaces.

Definition 12. *An inner product on a vector space X is a mapping of X×X into the scalar field K of X; that is, with every pair of vectors x and y, there is associated a scalar which is written*

$$\langle x, y\rangle$$

and is called the inner product of x and y, such that for all vectors x,y,z and scalar α the following properties hold:

$$\langle x + y, z\rangle = \langle x, z\rangle + \langle y, z\rangle$$

$$\langle \alpha x, y\rangle = \alpha\langle x, y\rangle$$

$$\langle x, y\rangle = \overline{\langle y, x\rangle}$$

$$\langle x, x\rangle \geq 0$$

$$\langle x, x\rangle \iff x = 0$$

An inner product on X defines a norm on X given by

$$\|x\| = \sqrt{\langle x, x\rangle}$$

and a metric induced by the above norm given by

$$d(x, y) = \|x - y\| = \sqrt{\langle x - y, x - y\rangle}$$

A vector space X with an inner product defined on it, is called an *inner product space* (or *pre-Hilbert space*).An inner product space which is complete in the metric defined by the inner product, is called a *Hilbert space*.

It is apparent that an inner product space is a normed space and a Hilbert space is a Banach space.

An important theorem of Banach spaces, and thus Hilbert spaces, is the *Fixed point theorem* or *Contraction theorem*.Before we present the theorem and its useful proof we have to give the following definitions.

Definition 13. *A fixed point of a mapping T:X → X of a set X into itself, is a x ∈ X which is mapped onto itself (it is kept "fixed" by T), that is,*

$$Tx = x$$

the image Tx coincides with x.

Definition 14. *Let X = (X,d) be a metric space. A mapping T:X → X is called a contraction on X if there is a positive real number $\alpha < 1$ such that for all x,y ∈ X*

$$d(Tx, Ty) \leq \alpha d(x, y)$$

that is, the images of any points x,y are closer together that the points x,y.

The Banach fixed point theorem is an existence and uniqueness theorem for fixed points of certain mappings, and its proof gives a constructive procedure for getting closer and closer to the fixed point starting from an initial approximation.Thinking this idea in connection with pattern recognition one should think of the fixed point as the pattern to be found and the initial approximation as the observation available.

Theorem 1. ***Banach fixed point theorem.***

Consider a metric space X = (X,d) where X ≠ ∅. Suppose that X is complete and let T:X→ X be a contraction on X. Then T has precisely one fixed point.

Proof Idea: We construct a sequence (x_n) and show that it is Cauchy so that it converges in the complete space X.Then we prove that its limit X is a fixed point of T and T has no further fixed points.

We choose any $x_0 \in$ X and define the iterative sequence (x_n) by

$$x_0\ ,\ x_1 = Tx_0\ ,\ x_2 = Tx_1 = T^2x_0\ ,...,\ x^n = T^nx_0,\ ... \tag{1.1}$$

which is the sequence of the images of x under repeated application of T. We now show that (x_n) is Cauchy. Since T is a contraction we have:

$$\begin{aligned} d(x_{m+1}, x_m) &= d(Tx_m, Tx_{m-1}) \\ &\leq \alpha d(x_m, x_{m-1}) = \alpha d(Tx_{m-1}, Tx_{m-2}) \\ &\leq \alpha^2 d(x_{m-1}, x_{m-2}) \\ ... &\leq \alpha^m d(x_1, x_0). \end{aligned}$$

Hence, using the triangular inequality we obtain for n>m:

$$\begin{aligned} d(x_m, x_n) &\leq d(x_m, x_{m+1}) + ... + d(x_{n-1}, x_n) \\ &\leq (\alpha^m + \alpha^{m+1} + ... + \alpha^{n-1}) d(x_0, x_1) \\ &= \alpha^m (\frac{1-\alpha^{n-m}}{1-\alpha}) d(x_1, x_0) \end{aligned}$$

Since 0< α <1, in the numerator we have 1 - $\alpha^{n-m} < 1$.Consequently,

$$d(x_m, x_n) \leq (\frac{\alpha^m}{1-\alpha}) d(x_1, x_0)$$

On the right 0< α <1 and $d(x_0, x_1)$ is fixed, so that we can make the right-hand side as small as we please by taking m sufficiently large (and $n > m$).This proves that (x_m) is Cauchy. Since X is complete, (x_m) converges, say $x_m \to x$. We now have to show that this limit x is a fixed point of the mapping T.

From the triangle inequality and the definition of contraction we have:

$$\begin{aligned} d(x, Tx) &\leq d(x, x_m) + d(x_m, Tx) \\ &\leq d(x, x_m) + \alpha d(x_{m-1}, x) \end{aligned}$$

The sum on the second line can be made smaller than any preassigned $\varepsilon > 0$ because $x_m \to$ x.We draw the conclusion that *d(x,Tx)=0)* and consequently $x = Tx$, which means that x is a fixed point of T.

We now have to show that this fixed point is unique. Let there be a second fixed point $\bar{x}$.Then

$$d(x, \bar{x}) = d(Tx, T\bar{x}) \leq \alpha d(x, \bar{x})$$

which implies that *d(x,$\bar{x}$)=0* since $\alpha < 1$. Hence x = $\bar{x}$ and the proof of the theorem ends here.

1.6 Manifolds

An important notion that we will come across very often in the present work is the notion of a *manifold*.

An intuitive representation of a manifold is the following: suppose we have a set of data vectors in $\mathbb{R}^n$, and that a subset of their dimensions represents certain features of interest.If we focus on these dimensions only,"keeping" the rest "steady", we notice that the data vectors move along a certain path in $\mathbb{R}^n$. This path, or curve formed in the space, is the *manifold* that the features lie on in $\mathbb{R}^n$.

For example, in figure 1.3 we see a curve in $\mathcal{R}^3$, which has zero volume and zero area. It can therefore be parameterized by a single variable. Consequently, despite being in the three-dimensional space, the curve has an intrinsic dimensionality of one, since it locally resembles $\mathcal{R}^1$. A manifold has a *locally* Euclidean geometry,in a neighborhood around each point, but on a global scale the relations between its elements are non metric.

We will give a formal definition of manifolds, when we talk about their use in representation learning.

1.7 Summary

In this chapter some basic ideas about spaces and their properties were presented.Starting from the metric and metric space, we moved on to the norm and vector space and based on these we defined the Hilbert space which is the closest generalization of the well-known Euclidean space. Hilbert spaces provide us with the tools we need in our work: convergence, completeness, contractions etc.

Figure 1.3: *A one-dimensional manifold embedded in the three-dimensional space. [Cay05]*

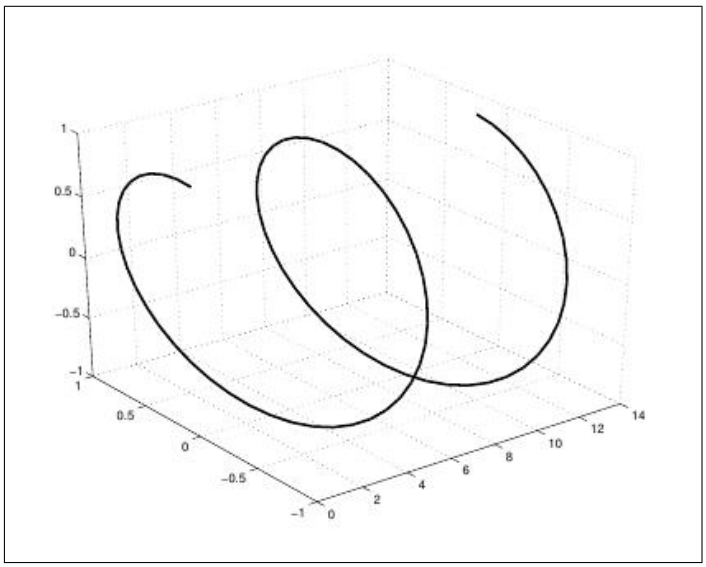
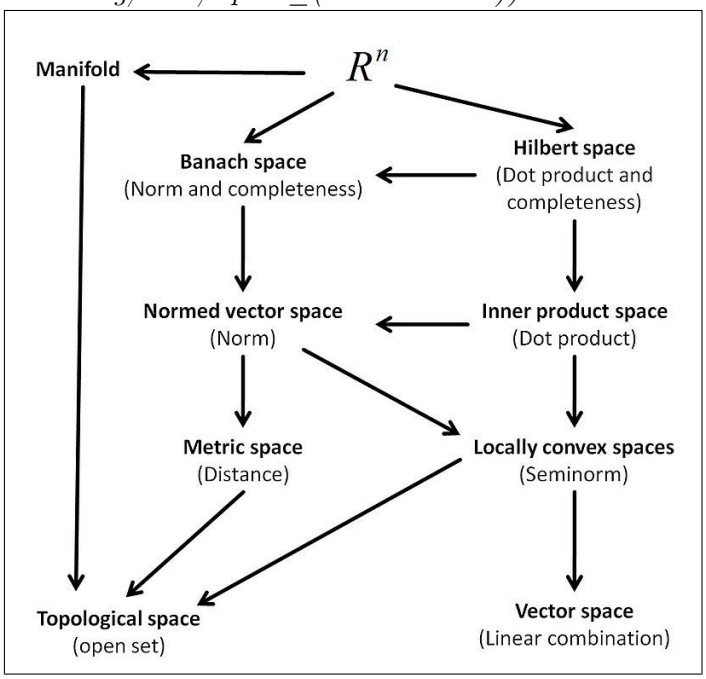


Figure 1.4: *Overview of various spaces and their relations.(Image taken from:https://en.wiki2.org/wiki/Space_(mathematics))*



Chapter 2

Automatic Speech Recognition

2.1 Introduction

The aim of the work on Automatic Speech Recognition (*ASR*) is to build systems that recognize spoken speech, that is, they are able to map acoustic signals to strings of words. In contrast to natural language processing, ASR does not try to determine the meaning behind speech or find the multiple meanings of words; it merely tries to recognize *which* words were spoken.

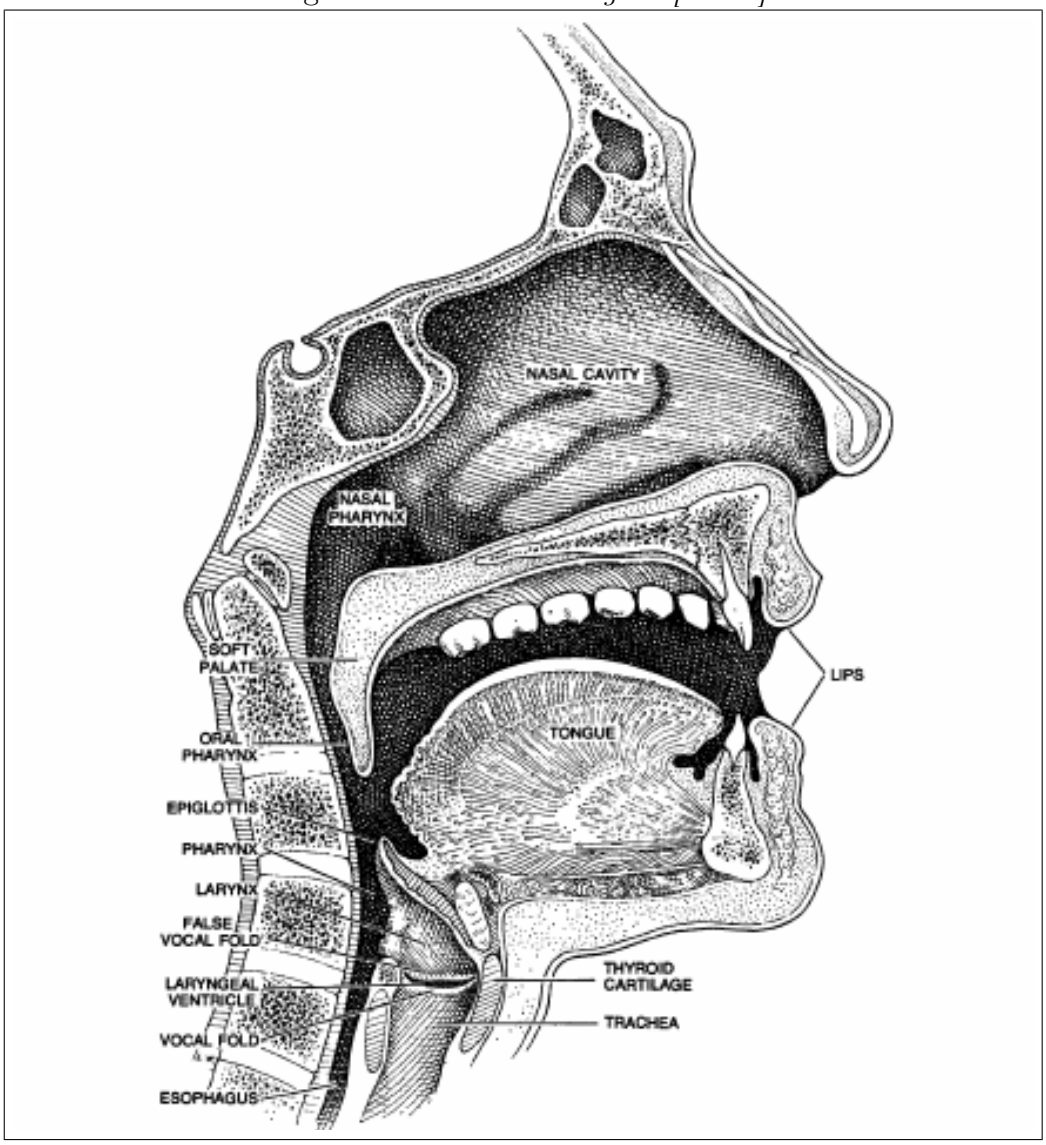
Research in ASR has come a long way in the last few years, which has allowed us to take advantage of it in multiple areas with very satisfying results: human-computer interaction (speech-only or multimodal interfaces), telephony (information passing, call routing) and dictation are examples where ASR systems can perform well irrespective of the speaker or their environment.

We will begin by providing a quick description of the way human speech is produced and move on to present the concept and mechanisms behind Automatic Speech Recognition.

2.2 Fundamentals of the speech production mechanism

[JM09] Sound is produced by the rapid movement of air from the lungs through the “windpipe", also called the *trachea* and out of the mouth or nose. As it flows through the trachea, it passes through the *larynx* (or *Adam's apple*) and there it affects the position of two small folds of muscle which are known as *vocal folds* or *vocal chords*. The possible movements of these muscles are either moving closer together or apart; the space between them is called the *glottis*. If the vocal folds are close together they will vibrate as air passes through the glottis and produce sounds

Figure 2.1: *The vocal organs.[JM09]*



that are known as *voiced* sounds. The vowels are examples of voiced sounds. On the other hand, if they are far apart they will not vibrate and the sounds produced are called *unvoiced*. Examples of unvoiced sounds are [p], [t] and [k].

After passing through the trachea and before exiting the body, air passes through the vocal tract, which consists of the *nasal* and *oral* tract. The vocal tract will act as a filter on the speech signal and the output of the filter will be the sound we will produce. The speech signal will vary according to what obstacles the air meets on its way out : the tongue, the lips or the teeth. These obstacles will define the vocal tract filter applied on the speech signal.

2.3 Design parameters of an ASR system

This section attempts to provide a short introduction into Automatic Speech Recognition and the most common architectures behind ASR systems[JM09].

However, before deciding on the architecture of a speech recognition system, one has to take into account its application domain.The most decisive parameters are:

- The vocabulary size,which is the number of distinct words that the system should be able to recognize. Few words imply relatively easy set-up and training of the system, whereas systems recognizing thousands of words, as in a broadcasting news vocabulary, are more complicated and harder to train.

- The fluency of the speech that the system will be asked to recognize.This includes whether the speech will be continuous or just isolated words as well as the speed and clarity of the speaker.Isolated word recognition systems, e.g. recognizing commands to a computer, are easier than ASR systems for continuous, conversational speech, e.g. a telephone conversation between humans.

- The environment where the recognition might have to take place. Systems designed to perform well in noisy environments with high distortion in the speech recorded are more demanding than systems that can clearly capture the speech for recognition in an isolated environment.

- The speaker variability. Speech recognition is easier if the system is expected to recognize the speech of a limited number of people. On the contrary, a general ASR system that should work with any speaker, regardless of sex, age, or accent is much more difficult to implement.

2.4 ASR system architecture

As mentioned in the introduction, the problem of ASR is, in principle, a structured sequence classification task, where a (relatively long) sequence of acoustic data is used to infer a (relatively short) sequence of the linguistic units such as words. Modern ASR systems use the model of a *noisy channel* to deal with the classification task. The idea behind this model is to think of the input signal as a distorted version of the corresponding words, which was produced as they passed through a noisy communications channel. If we manage to understand how the channel affected the signal, we can then match it to the original, noise-free set of words, by passing every acceptable -by the grammar of the language- sentence though the channel to get its distorted version and see which matches best the initial input signal.

Figure 2.2: *The noisy channel model. The decoder searches through all possible sentences and finds the one that after passing through the channel best matches the initial input signal[JM09].*

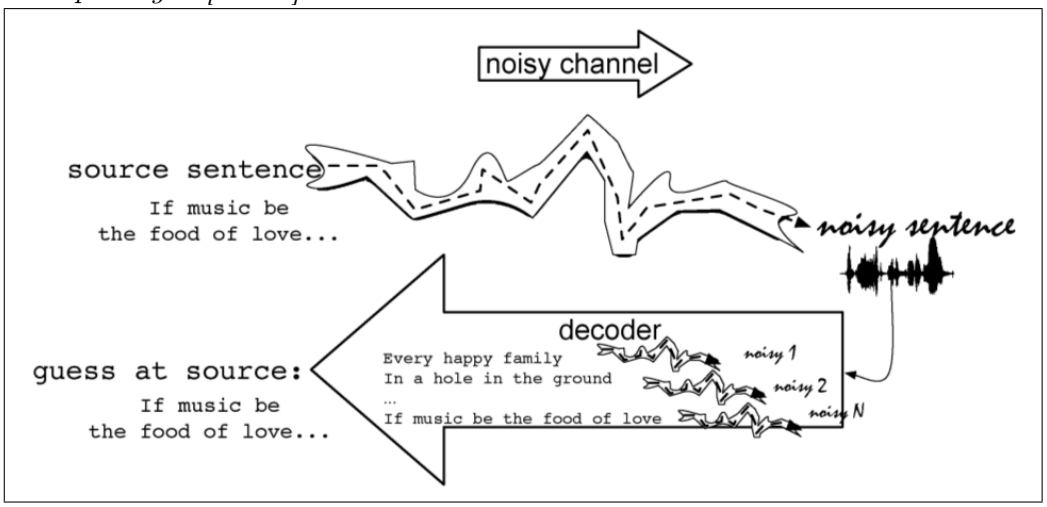
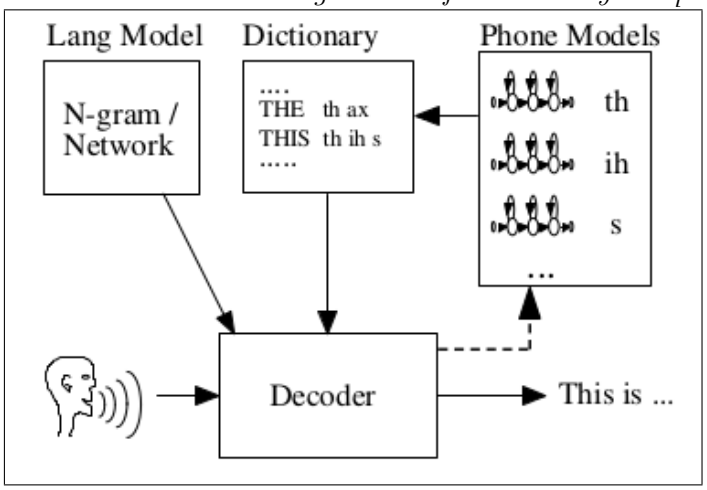


Figure 2.3: *The main building blocks of an ASR system[You02].*



In order to achieve the modelling of the noisy channel and the subsequent decoding of a new acoustic signal, we will need to have at our disposal the following tools: the prior probability of each sentence of the language, the probability of words being the concatenation of certain speech units and the probability of these speech units being realised as acoustic or spectral features, which are drawn from the input signal.

These tools define the main components of a modern automatic speech recognition system, which will be presented below, following the formulation of the computational/mathematical problem of Automatic Speech Recognition.

2.4.1 Computational formulation of ASR

We will now use mathematical notation and probability theory to answer the basic question in ASR: ''What is the most likely sentence Ŵ out of all sentences belonging to language $\mathbb{L}$ given an acoustic signal O? ''.

The acoustic input signal O is a sequence of observations o_i,

$$O = o_1, o_2, o_3, ...o_t$$

each one representing features of a specific part of the input speech, which is usually split into overlapping parts of a certain duration (*frames*). In the same way, if we treat each sentence of the language as a string of words,

$$W = w_1, w_2, w_3, ...w_n$$

we can write the answer to the ASR question in the following way:

$$\hat{W} = \underset{W \in \mathbb{L}}{\operatorname{argmax}} P(W|O)$$

, which we cannot compute directly. However, if we apply Bayes' rule, the equation takes the following form:

$$\hat{W} = \underset{W \in \mathbb{L}}{\operatorname{argmax}} \frac{P(O|W)P(W)}{P(O)}$$

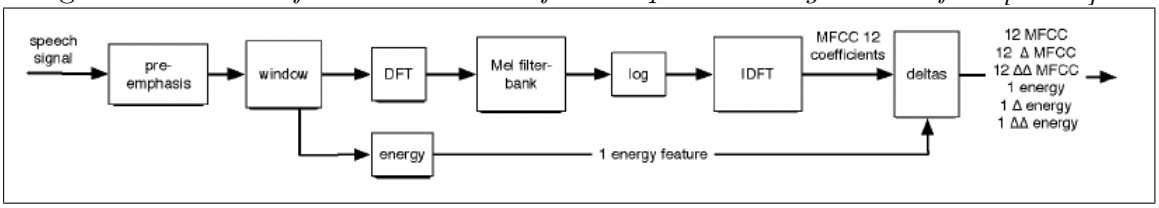
We can simplify the computation even further, if we consider that the probability of the observation in the denominator, *P(O)*, does not affect the maximization with respect to *W*, since the observation signal stays the same as we search over the sentences space. Consequently, the answer to our problem can be computed by the following form:

$$\hat{W} = \underset{W \in \mathbb{L}}{\operatorname{argmax}} P(O|W)P(W)$$

The two probabilities on the right hand side of the equation, represent the tools we need to address the recognition problem, as we have mentioned above: *P(W)*, which is the prior probability of each sentence of the language, is computed by the *language model*, whereas $P(O|W)$, which includes the probability of words being the concatenation of certain speech units and the probability of these speech units being realized as certain features, is calculated by the *acoustic model*.

Having presented the computational formulation of the automatic speech recognition problem, we can move on to present the required steps to be taken and models to be constructed in order to build a speech recognition system.

Figure 2.4: *MFCC feature extraction from a quantized digital waveform[JM09].*



2.4.2 Feature Extraction

[JM09] The first issue we have to address is how and in what format do we "insert" the speech waveform into our system. This process, known as *feature extraction* results in the extraction from the speech waveform of a sequence of acoustic *feature vectors*, each of them representing the information included in a small time window of the signal. These feature vectors are further preprocessed and finally presented as the input to the ASR system.

Mel Frequency Cepstral Coefficients

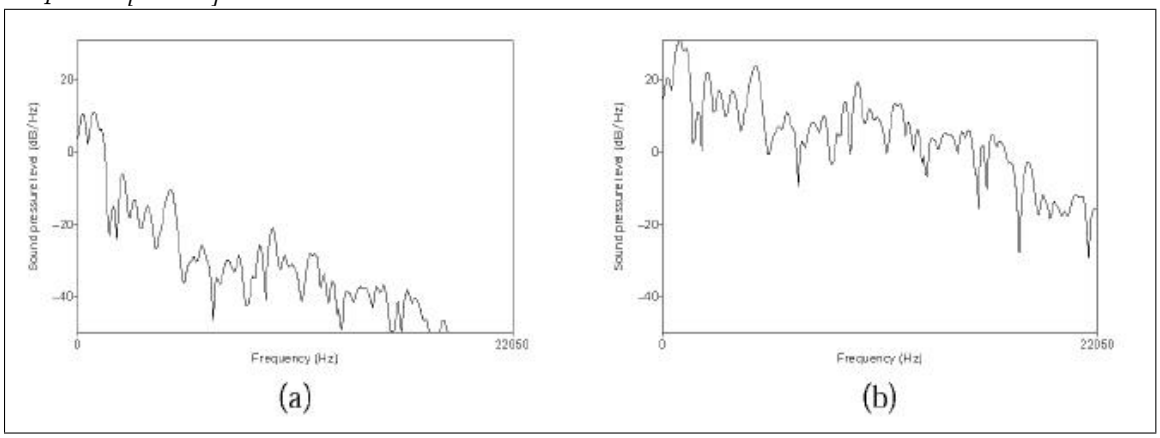
Although research in feature extraction moves towards using the raw waveform directly as input to the system, so far the most common feature representation in speech recognition systems is the Mel Frequency Cepstral Coefficients (*MFCC*).The steps involved in extracting the MFCC feature vectors follow the analog to digital conversion of the speech signal (sampling and quantization) and are outlined in the next paragraphs:

Pre-emphasis In the pre-emphasis step we want to amplify the amount of energy in the high frequencies of the signal. If we take a look at the spectrum of a vowel we will note a drop in energy as we move on to higher frequencies; this is known as *spectral tilt* and is due to the nature of the glottal pulse. **(EXPLAIN STH????)** Amplifying the energy of higher frequencies will improve phone detection accuracy as it will provide more information to the acoustic model coming from the boosted higher formants. In essence, pre-emphasis is applying to the signal a first order, high-pass filter whose equation is:

$$y[n] = x[n] - \alpha x[n-1],\ 0.9 \leq \alpha \leq 1.0$$

Windowing Given that the feature vectors we want to extract will be used to train phone classifiers, i.e. the acoustic model, we want them to able to capture the spectral properties corresponding to these fundamental speech units.Consequently, since speech is a non-stationary signal - its statistical properties are not constant

Figure 2.5: *Part of the spectrum of vowel [aa] before (a) and after (b) pre-emphasis[JM09].*



across time - we extract the feature vectors from a small window of the speech signal that corresponds to a phone or subphone, where the signal can be considered stationary - its statistical properties stay constant across time.

The windowing process comes down to applying a filter to the signal that is non-zero inside some region and zero elsewhere, moving this filter along the speech signal and extracting segments of the signal (or *frames*).The window is characterized by its *width* (in milliseconds, also called *frame size*), the *overlap* between successive windows (usually a percentage of the width) and its *shape*, e.g. rectangular, Hamming etc. The Hamming window is usually preferred over the rectangular as it gradually reduces the signal values at the boundaries of the window towards zero, thus avoiding discontinuities which cause problems during the next step of feature extraction (Fourier analysis):

$$w_{hamming}[n] = \begin{cases} 0.54 - 0.46\cos(\frac{2\pi n}{L}),\ 0 \leq n \leq L-1 \\ 0\ otherwise \end{cases}$$

The application of the filter is an element-wise multiplication of the signal values at each time step n with the values of the window:

$$y[n] = w[n]s[n]$$

where w is the window and s the signal.

Fourier Analysis The next step in the feature extraction process is to acquire the spectral information included in the windowed segments. The tool to extract such information, e.g. the amount of energy included in different discrete frequency bands of each segment, is the Discrete Fourier Transform.Given a part of a signal, the DFT will produce a complex number representing the magnitude and phase of

Figure 2.6: *Rectangular and Hamming windows and their effect on the signal[JM09].*

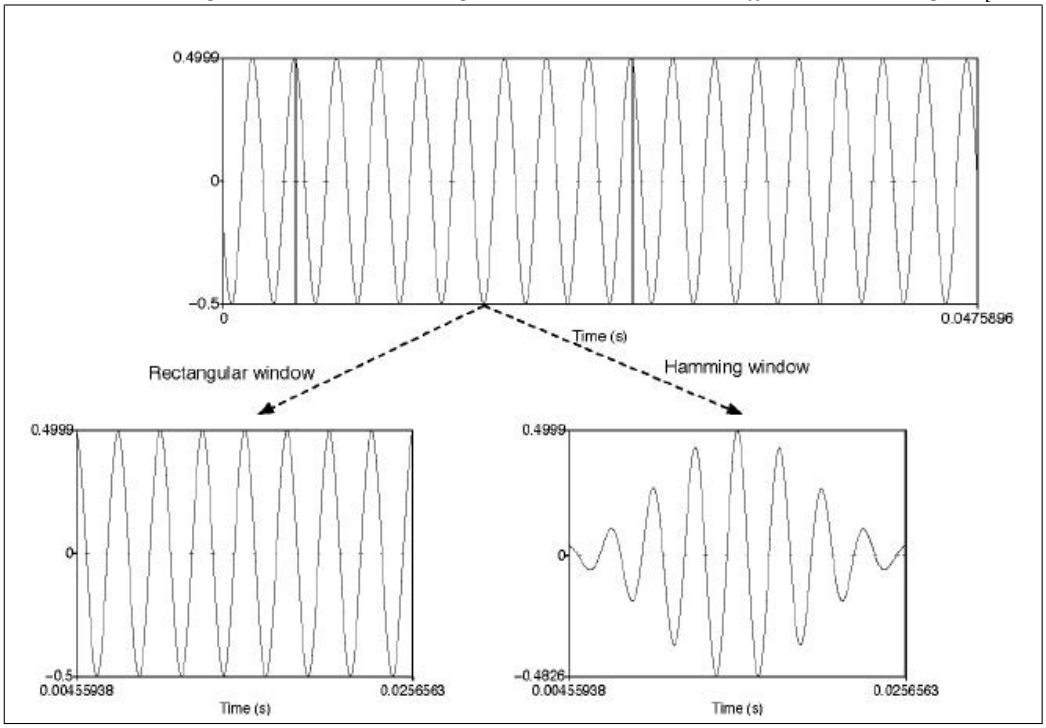


Figure 2.7: *Windowing process with a rectangular window.After a figure by Brian Pellom[JM09].*

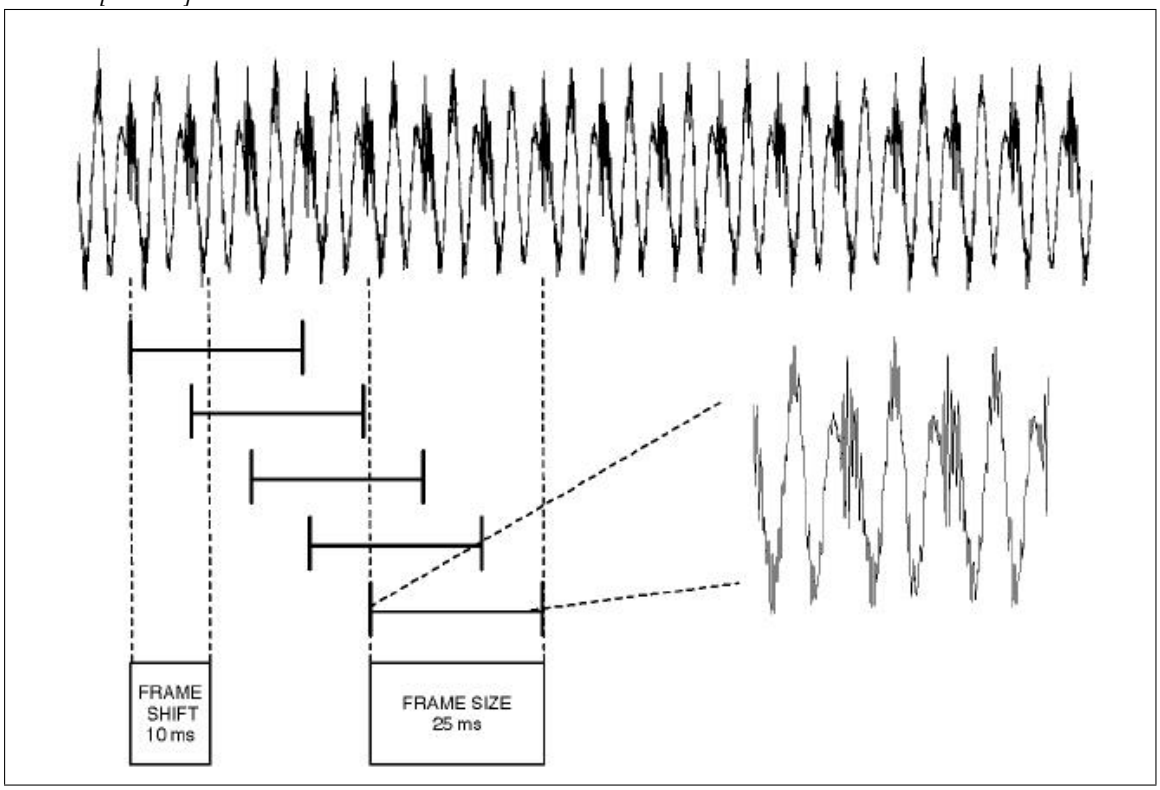
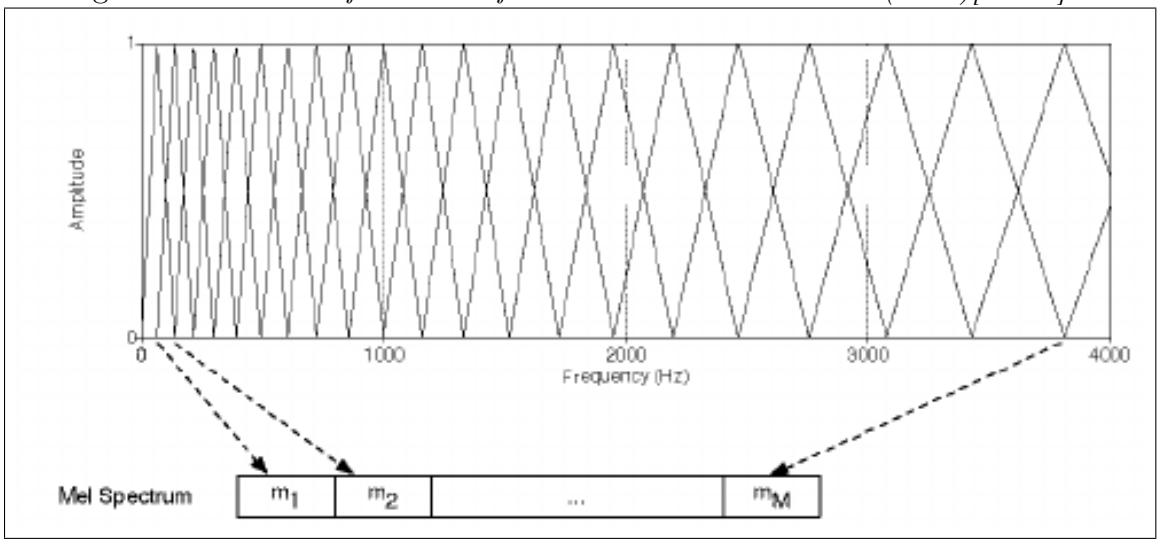


Figure 2.8: *The mel filterbank after Davis and Mermelstein (1980)[JM09].*



each frequency component of the corresponding segment.

Mel Filterbank Research has shown that the human hearing is not equally sensitive at all frequency bands - it is less sensitive at higher frequencies (above 1000 Hz).Moreover, humans are less sensitive to differences in amplitude at high amplitudes than at low amplitudes. Consequently, if we model this information and introduce it into the ASR system, we will improve its performance.

The way we take advantage of this information during the feature extraction process is by passing the DFT of the windowed signal parts through an array of triangular filters (a *filterbank*), which collect energy from each frequency band. These filters have their center frequencies spread on a *mel* scale, that is, 10 of them are spaced linearly below 1000 Hz and the remaining filters of the bank are spaced logarithmically above 1000 Hz. A *mel* is a unit of pitch defined so that pairs of sounds which are perceptually equidistant in pitch are separated by an equal number of mels[SVN37].The mapping between frequency in Hz and the mel scale is described by the relation:

$$mel(f) = 1127 \ln(1 + \frac{f}{700})$$

Finally, having passed the DFT of the signal through the filterbank, the final step is to take the logarithm of the mel spectrum values, which is a form of normalization to make the feature estimates less sensitive to variations in input.

The Cepstrum The next step in the MFCC feature extraction process is the computation of the *cepstrum*.As mentioned before, the speech waveform is created when a glottal source waveform is passed through the vocal tract which acts as a filter. As we have already mentioned, the shape of the vocal tract will determine the

Figure 2.9: *Effect of vocal tract on source signal.Tomi H. Kinnunen, Speech Technology Workshop*

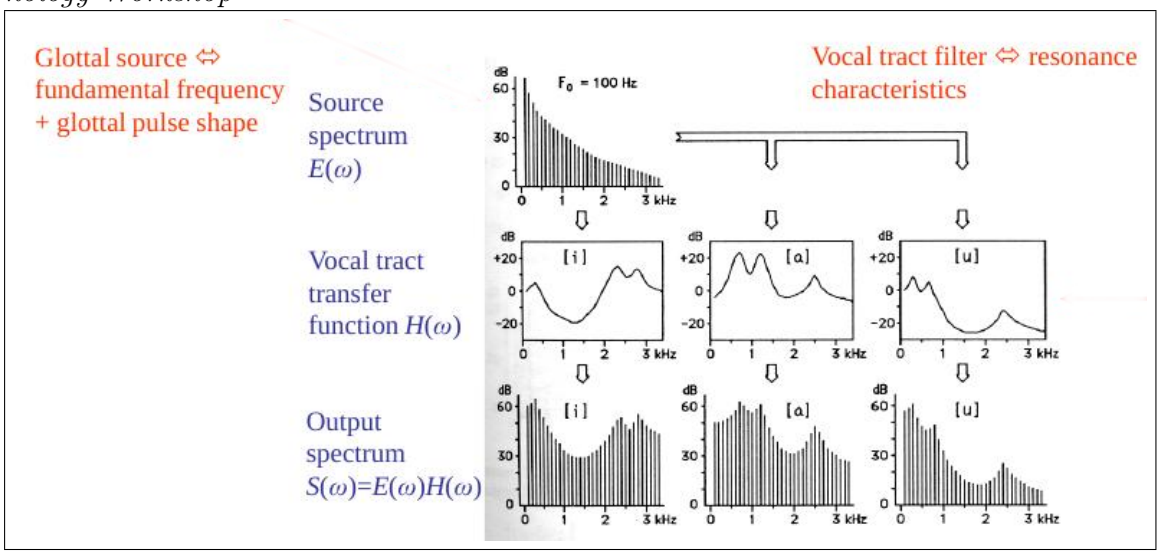
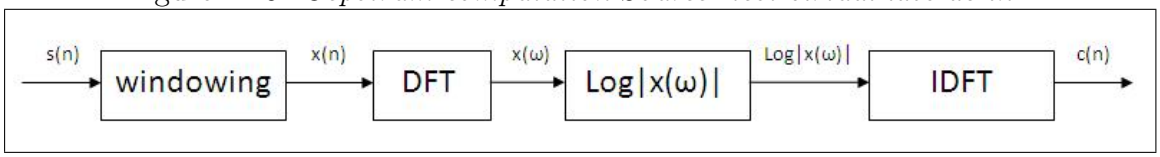


Figure 2.10: *Cepstrum computation.Source: test.virtual-labs.ac.in*

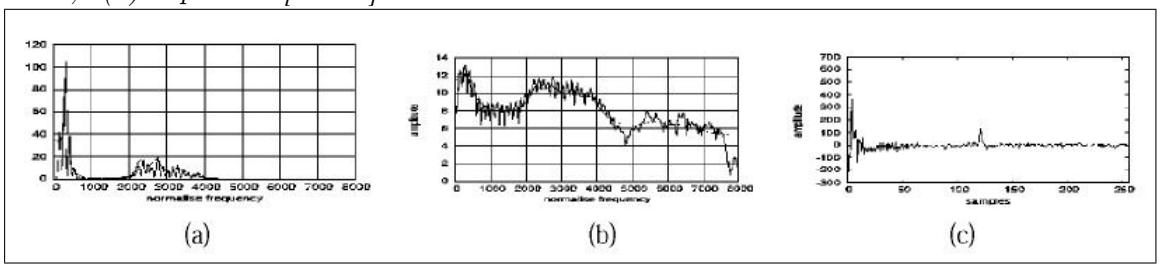


outcome of this filtering process, i.e. the sound, the phone that will be produced.

Therefore we aim to have some information about the vocal tract inserted into the feature vectors. The cepstrum provides us with a way of separating the glottal source from the vocal tract filter.

The peaks at lower values on the x-axis in the cepstrum of Figure 9 (c) correspond to the vocal tract characteristics whereas peaks at higher values correspond to the glottal source. Therefore, since we need information about the way a phone was produced, i.e. about the vocal tract shape, for the feature vectors we want to extract, we will keep a few of the first cepstral values (usually 12). Furthermore, an

Figure 2.11: *Cepstrum example: (a) magnitude spectrum, (b) log magnitude spectrum, (c) cepstrum.[JM09].*



important property of the cepstral coefficients is that their variance is uncorrelated, contrary to spectral coefficients, which are correlated at different frequency bands. This is extremely important for acoustic models based on Gaussian Mixture Models, as it allows us to keep the number of their parameters low.

Feature vectors The cepstral coefficients extracted from the previous process are just a part of the feature vectors. They are further enhanced by adding a few extra coefficients that provide more information helpful towards determining speech units such as phones.

The first extra piece of information we include by adding one more coefficient is the energy of the frame, which is defined as:

$$Energy = \sum_{t=t_1}^{t_2} s^2[t],$$

where s is the speech signal and t_1, t_2 are the boundaries of the frame.

Energy is useful for phone detection since it correlates with phone identity: higher energy reveals the presence of e.g. a vowel whereas very low energy could identify a pause in speech.

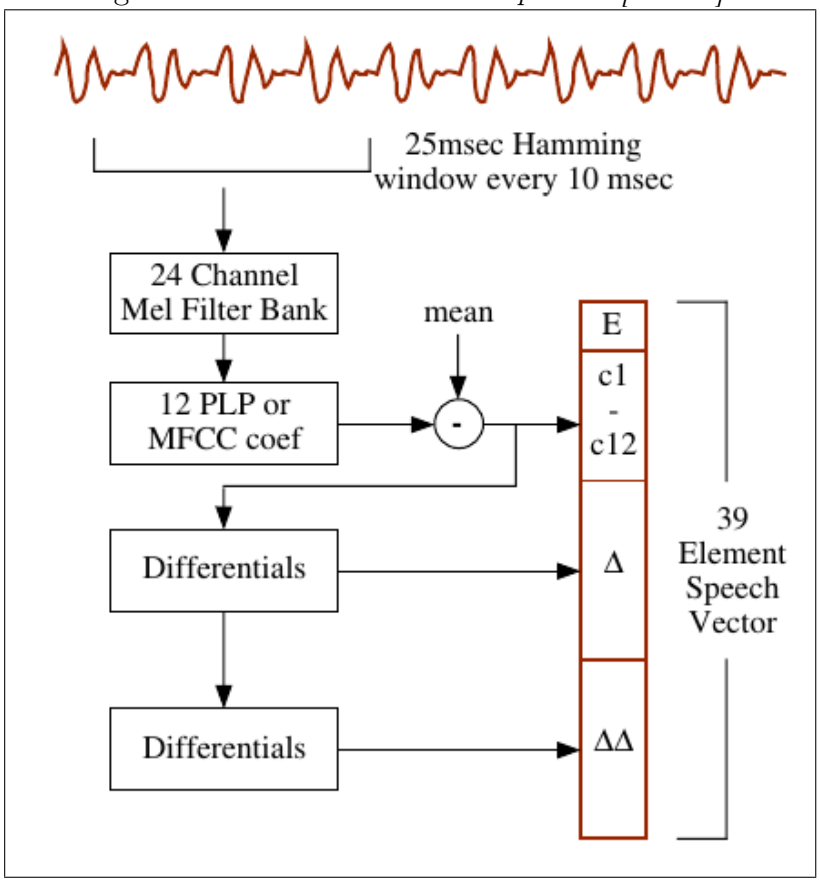
Considering that speech properties change from frame to frame it is reasonable to expect that capturing these changes would provide more information about the nature of the speech signal. This motivated the use of *delta* and *delta-delta* coefficients for each one of the cepstral coefficients and the energy. The *delta* coefficients capture the change of the corresponding feature between successive frames whereas the *delta-delta* coefficients capture the change of the *delta* features between successive frames. The simplest way to compute these *delta* and *delta-delta* coefficients is by taking the difference of the corresponding features between successive frames:

$$d(t) = \frac{c(t+1) - c(t-1)}{2},$$

where c is the cepstral feature with delta coefficient d at time t.

This concludes the construction of the feature vectors.As mentioned before, we usually pick 12 cepstral coefficients plus an energy coefficient, thus we will have 39-dimensional feature vectors: 13 + 13 *delta* +13 *delta-delta* coefficients.It is also common to concatenate the cepstral coefficients to produce higher dimensional vectors.This way we manage to include contextual information in the feature vectors.To deal with computational problems arising from the increase in the number of dimensions, various dimensionality reduction techniques are used, from simple ones such as Linear Discriminant Analysis or Principal Component Analysis, to more sophisticated such as techniques aiming to discover the lower dimensional manifold on which the feature vectors lie.

Figure 2.12: *Feature extraction process.[You02].*



2.4.3 The Language Model

The language model (LM) expresses how likely a given string of words is, taken into consideration certain linguistic constraints. In order to do this, we build on the idea of predicting the next word in a sequence of words, which is formalized with probabilistic models called *N-gram models*.

N-grams

An *N-gram* is a sequence of N words, e.g. a 2-gram or bigram is a sequence of two words, a 3-gram or trigram is a sequence of three words etc. An *N-gram model* is a probabilistic model which computes the N^{th} word of a sequence of N words given the previous N-1.

The power of N-grams becomes evident in areas such as speech or handwriting recognition, machine translation, spelling correction and natural language processing tasks. What all these areas have in common is that they might have to deal with noisy or ambiguous input. N-grams can deal with ambiguity by assigning a higher likelihood to word sequences that are valid according to the language constraints.

Since these models are capable of assigning a conditional probability to the next possible word in a group, we can exploit them to compute the joint probability of a sequence of words,i.e. a sentence, which is what we were aiming to from the beginning.

Suppose we have a sequence of n words $W = w_1, w_2, w_3, ..., w_n$. Then, the probability *P(W)* can be computed in the following way:

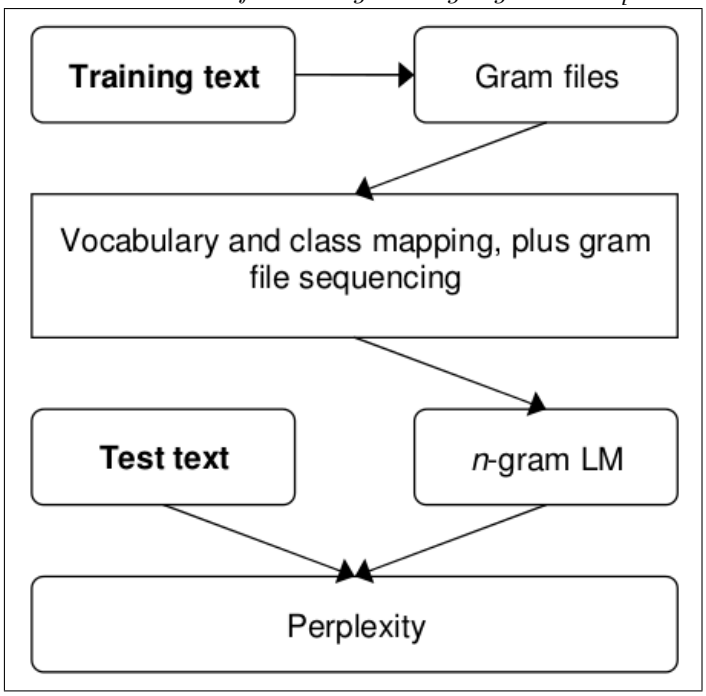
$$P(W) = P(w_1, w_2, w_3, ..., w_n) = \prod_{i=1}^{n} P(w_i|w_1, w_2, w_3, ..., w_{i-1})$$

As this computation for every possible word sequence in the language is very difficult if not impossible, we make the assumption that the i^{th} word depends only on the previous N-1 words (its *history*). At this point we can take advantage of the intuition behind the N-gram model:

$$P(W) = \prod_{i=1}^{n} P(w_i|w_1, w_2, w_3, ..., w_{i-1}) \approx \prod_{i=1}^{n} P(w_i|w_{i-N+1}, ..., w_{i-1})$$

to approximately compute the probability of the sequence. The expressions on the two sides would be exactly equal for sufficiently high n and if the language were ergodic, that is, the probability of any word could be estimated from sufficient history independent of the starting conditions.

Figure 2.13: *Process of building a language model.[YEK+02].*



Building a language model based on N-grams

[YEK+02]

Considering the intuition of N-grams and assuming that the probability of an N-gram occurring in an unknown text can be estimated from its frequency in a given training text, we can build language models based on N-grams.

The construction of such an LM can be broken down into three stages:

- Collect and store the N-grams of the training text (*corpus*)
- Possibly map some words into classes, e.g. out-of-vocabulary class mapping ???????????
- Count the N-grams and compute the N-gram probabilities

The last step of computing the probabilities is based on maximum likelihood estimation:

$$\hat{P}(w_i|w_{i-N+1},...,w_{i-1}) = \frac{C(w_{i-N+1},...,w_i)}{C(w_{i-N+1},...,w_{i-1})}$$

where *C(.)* is the count of a given word sequence extracted from the training text.

When building a language model based on N-grams, there are several factors one has to take into account before they decide on N. Resources constraints (e.g. storage) and size of the *vocabulary* (i.e. the set of distinct words in the language) will play a major role in deciding on N, as the number of parameters of the model grows

exponentially with N : $|V|^N$, where V is the vocabulary. However, storage needs because of parameters' size are lower than one would expect, because not all N-word combinations are acceptable/valid sequences in the language. What increases storage and computational needs however, is the large training sets required, so that our model estimates parameters with a minimum acceptable degree of confidence. Apart from processing, also acquiring these training sets might be difficult, especially for domain-specific applications, where training sets have to be specifically constructed.

Data sparsity and smoothing

[JM09] However, since the training set will always be finite, one can never have a sufficient number of N-grams for every valid N word sequence of the language. This issue faced when developing LMs is known as *data sparsity* and the technique used to deal with it is called *smoothing*. Smoothing aims to increase the robustness of the language model by redistributing the probability mass assigned by the maximum likelihood estimates: it removes some of it from higher counts of N-grams and assigns it to very low or zero counts, in order to ''smooth'' the distribution.

Laplace Smoothing. *Laplace* or *add-one* smoothing is the simplest form of smoothing : we just add one to all the counts. Before we compute the ML probabilities, one has to to take into account the extra $|V|$ ''words'' that we added, in order to maintain the sum of all probabilities equal to 1. As this last step is necessary and because Laplace smoothing does not perform well, it is more convenient to use an *adjusted count*

$$c_i^* = (c_i + 1)\frac{K}{K+V}$$

where c_i is the original count and K is the number of word tokens.

Discounting. Instead of adding the same amount of probability mass to all N-gram probabilities, a different approach would be to remove some mass out of the higher counts and assign it to lower ones. We can therefore define a *relative discount* as the ratio of the new counts and the originals:

$$d_c = \frac{c^*}{c}$$

One algorithm applying discounting, is the *Good-Turing discounting*.The intuition behind it is that we use the MLE of N-grams occurring c+1 times in the training set, to define the MLE of N-grams occurring c times. The new smooth count c^* is thus defined as:

$$c^* = (c+1)\frac{N_{c+1}}{N_c}$$

where N_i is the number of different N-grams occurring i times in the training set.

Back-off and interpolation. Discounting techniques allow us to distribute some probability mass equally to the unseen events. However, we can distribute it fairer if we take into consideration information from lower or higher order N-grams. This is the idea of *back-off* smoothing. In particular, *Katz back-off* always resorts to the (i-1)-gram if the i-gram has zero counts (i starting from our originally selected N). From a different point of view, we back-off to lower order N-grams only if we have zero evidence for a higher order one. On the other hand, *interpolation* deals with zero counts by summing estimates of all N-grams using weights (e.g. interpolate the estimates of unigrams,bigrams and trigrams).

Language model evaluation

Given the numerous applications of N-grams and LMs as well as their inherent drawbacks (static and finite vocabulary, finite training sets, more N-grams than can ever be collected and utilized), it is evident that we must have a way to compare LMs and evaluate their performance.

The most obvious way to compare two different language models would be to use them in our application and see which gives the best results. However, this way is expensive and time consuming as it is based on training and evaluating systems using huge speech datasets.

Another way to compare two LMs independent of the application, is to use the *perplexity* metric. The perplexity of an LM on a test set is a function of the probability that the LM assigns on it and is defined as:

$$PP(W) = P(w_1 w_2 w_3 ... w_N)^{-\frac{1}{N}}$$

where $W = w_1 w_2 w_3 ... w_N$ is the test set.

The best LM would be the one that has the minimum perplexity, since that would mean that it maximizes the test set probability, i.e. it better predicts the details of the test set. Another way to look at perplexity is as the weighted average branching factor of the language, that is, the number of best possible words following a word sequence. The smaller that number is, the better the work of the LM on coping with the ambiguity of the language.

Finally, when comparing language models using the perplexity metric, one should take care to use the *same vocabulary* for both LMs and evaluate them on the same test set which will be presented to the system for the first time during evaluation.

Recent advances in language modeling

In a paper presented recently [BDVJ03], a novel approach to building language models was presented which takes advantage of neural networks. The motivation

behind using NNs to build language models is based on the following problems associated with N-gram models:

- a huge amount of training data is needed to train LMs which will still have limited context capabilities (1-2 words)
- N-gram models ignore word similarity, which makes generalizing difficult

To deal with these issues becomes even more important when one considers that language models are probabilistic models using discrete random variables (words) which largely increases the amount of free parameters they need.

The writers suggest using distributed representations of words (i.e. real-valued feature vectors) which allows them 1) to identify similarities between words, since similar words will have similar feature vectors and 2) to exploit the smooth probability function modeled by a neural network in order to generalize: in this way, each vector representing a word will be able to provide information about a huge number of similar words, i.e. its "neighbors" in the feature space.In their work they present a neural network which simultaneously learns its parameters and the feature vectors associated with each word in the training set, and most importantly, the number of parameters it uses scales linearly with the vocabulary and context size. Due to the high computational cost of the training (higher than N-gram based models), they use parallel methods to efficiently train the model.

In recent years the use of neural networks for language modeling has included using recurrent neural networks which can take advantage of arbitrarily long contexts for each word (like humans do), something that was not possible with feedforward NNs [MKB+10].

2.4.4 The Acoustic Model

[You02]

According to the computational formulation of the ASR problem, we need the likelihood of the observed data (i.e. the acoustic signal) given the word sequence $P(O|W)$.However, it would be impractical and inefficient if we tried to compute this likelihood by building a separate model for each word in the language, since sub-word units are shared among different words. Instead, as mentioned before, the *acoustic model* calculates the probability of words being the concatenation of certain speech units and combines it with the probability of these speech units being realized as certain features, to produce the desired likelihood for a word.

We will first present some basic notions in acoustic modeling and then we will go into more details about this important part of the system.

Phones

The basic unit of speech analysis we use, is the *phone* which is the smallest identifiable unit we find in a stream of speech. The sequence of phones that constitutes each word in the training dataset is determined by a *pronouncing dictionary*. Using a phone sequence to represent each word makes it easy to add new words in the dataset just by adding them and their phone sequence to the dictionary.

Context-dependent phones

Given that there are thousands of words in a language, but just tens of phones (e.g. 44 in the English language) the computational and storage gain acquired from the use of phones as the basis of the acoustic model becomes immediately apparent. However, contextual effects like co-articulation cause large variations in the way that different sounds are produced even if in principle they correspond to the same phone. Hence, to achieve good phonetic discrimination, we build *context-dependent* phone models, with the most common being the *triphone*: for each phone there is a different model for every unique pair of left and right neighboring phones.

There are two dominant triphone models:

- *cross-word triphones*, which include phones of the previous and following words in the first and last triphones of the word of interest:

$$ten\ pots \rightarrow sil\_{sil}t_{e}\_{t}e_{n}\_{e}n_{p}\_{n}p_{o}\_{p}o_{t}\_{o}t_{s}\_{t}s_{sil}\_sil$$

  The advantage of this approach is that they model co-articulation across word boundaries, but on the other hand, they complicate the decoding process since the phone models of each word depend on the following and preceding words as well.

- *word-internal triphones*, which explicitly encode word boundaries, thus making decoding easier:

$$ten\ pots \rightarrow sil\_{sil}t_{e}\_{t}e_{n}\_{e}n\_\ \_p_{o}\_{p}o_{t}\_{o}t_{s}\_{t}s_{sil}\_sil$$

State of the art systems use mostly cross word triphones because of their ability to model contextual effects.

As a consequence, the number of distinct triphones greatly increases and the number of parameters for such systems can grow up to hundreds of millions, while at the same time we have too little training data in our disposal. In addition, we might

have unseen triphones appearing in evaluation tasks. To deal with these problems we have developed smoothing techniques, just as was the case with language models.

Smoothing techniques

Back-off and interpolation. When too little data is available for the training of a context-dependent model of a particular order, one can instead use a model of lower order at the expense of some inaccuracy in the modeling of the context: e.g. use a biphone or a context-independent phone (monophone) when we cannot use a triphone. In order to implement a more robust model one can use a weighted combination of models with various levels of context dependency (*interpolation*).

Parameter tying. An alternative that offers a greater degree of flexibility while maintaining the high level of context-dependency in the model, is the technique of *parameter-tying*, in which parameters of context-dependent phone models that are acoustically indistinguishable are tied together, to facilitate training in case there is little training data available.Before tying parameters together one has to apply some form of clustering to build the groups of phone models that will share their parameters.In practice, the most commonly used clustering technique is the *phonetic desicion tree*, where a binary tree is build for each phone model and its leaves contain the parameters to be shared.

Acoustic Modeling with Gaussian Mixture and Hidden Markov models

State-of-the-art ASR systems use Hidden Markov Models to represent each phone in conjunction with Gaussian Mixture Models to determine the probability that an acoustic observation was produced by a certain phone. The high representational capabilities and ease of training of these models are what has made them prevalent in ASR. We will present the GMM/HMM acoustic model and in the next chapters we will examine their most recent competitor, that is , deep neural networks.

Gaussian Mixture Models.[You02][YD14]

Multivariate Gaussian random variables and Mixture Models.

Given that the feature vector corresponding to an acoustic observation is multi-dimensional (usually 39-dimensional as was presented above), we will have to treat it as a multivariate random variable and use a multivariate distribution to assign a probability to it.The reason why we choose the Gaussian distribution is not only its desirable computational properties but also its ability to approximate many real-world data (owing to the law of large numbers), such as speech features.

Supposing that $\mathbf{\Sigma}$ is the co-variance matrix, $\boldsymbol{\mu}$ is the mean vector and d is the

number of dimensions of the feature vector, the multivariate Gaussian distribution is defined as:

$$p(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2} \left|\boldsymbol{\Sigma}\right|^{1/2}} \exp(-1/2(\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\vec{x} - \boldsymbol{\mu}) = \mathcal{N}(x|\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

However, because of the inherent multimodality of the speech features, a single Gaussian distribution is insufficient to describe them. Therefore, we use a *mixture* of Gaussian distributions:

$$p(\boldsymbol{x}) = \sum_{i=1}^{M} c_i \mathcal{N}(x|\boldsymbol{\mu_i}, \boldsymbol{\Sigma_i})$$

where c_i are the positive mixture weights and $\sum_{i=1}^{M} c_i = 1$.Usually, the number of the mixture components is chosen according to the nature of the problem and the information we have about the data.The variability and multimodality of the features may be due to multiple latent causes; provided we can identify these underlying causes we can match each one to the corresponding mixture component in the distribution.

As mentioned earlier, the Gaussian distribution is favorable both for its modeling and its computational properties. GMMs can model complex, multimodal distributions to any required level of accuracy and they can be trained using standard maximum likelihood approaches.Their attractiveness is also due to research into GMM training having come up with approaches to optimize the trade-off between their modeling effectiveness and the amount of training time and data needed. For example, we have the ability to reduce the number of free parameters (from M $\times d^2$ down to M) while still achieving high performance, if instead of using full covariance matrices we opt for diagonal Σ or even use the same matrix for all mixture components. The use of diagonal co-variance matrices has been thought to impose uncorrelatedness among features, but, given that a mixture of Gaussians with diagonal Σ can at least effectively describe the correlations modeled by a single full co-variance Gaussian, this thought has been misleading.

Specifically for speech recognition, a number of ways has been proposed to improve recognition accuracy of a GMM system.We can discriminatively train the system after the generative maximum-likelihood training, so that we maximize the probability of generating the observed speech features in the training data, or augment the input speech features with bottleneck features acquired using neural networks (the latter will be examined later on in this project).

The set of free parameters to be estimated for a Gaussian-mixture distribution is denoted by $\boldsymbol{\Theta}$ and consists of : $\{c_i, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\}$. In order to acquire the parameters

we rely on maximum likelihood methods and in particular on the Expectation-Maximization (EM) algorithm [DLR77]. The EM algorithm is used to find locally maximum likelihood parameter estimates of statistical models when the equations cannot be solved directly.It is especially useful for models involving latent variables apart from the parameters-to-be-estimated and the observable data. A GMM can be treated as such a model if we assume that each observable data point has a corresponding hidden data point specifying the component of the mixture that each point belongs to.Furthermore, the EM algorithm provides us with closed-form expressions for the computation of the estimates in the M-step:

$$c_i^{(j+1)} = \frac{1}{N}\sum_{t=1}^{N} h_i^{(j)}(t),$$

$$\boldsymbol{\mu}_i^{(j+1)} = \frac{\sum_{t=1}^{N} h_i^{(j)}(t)x^{(t)}}{\sum_{t=1}^{N} h_i^{(j)}(t)},$$

$$\boldsymbol{\Sigma}_i^{j+1} = \frac{\sum_{t=1}^{N} h_i^{(j)}(t)[\boldsymbol{x}^{(t)} - \boldsymbol{\mu}_i^{(j)}][\boldsymbol{x}^{(t)} - \boldsymbol{\mu}_i^{(j)}]^T}{\sum_{t=1}^{N} h_i^{(j)}(t)}$$

where the posterior probabilities, i.e. the "latent" variables corresponding to the mixture components, computed in the E-step are:

$$h_i^{(j)}(t) = \frac{c_i^{(j)}\mathcal{N}(\boldsymbol{x}^{(t)}|\boldsymbol{\mu}_i^{(j)}, \boldsymbol{\Sigma}_i^{(j)})}{\sum_{m=1}^{n} c_m^{(j)}\mathcal{N}(\boldsymbol{x}^{(t)}|\boldsymbol{\mu}_m^{(j)}, \boldsymbol{\Sigma}_m^{(j)})}$$

The last equation computes the conditional probability for a given data point $\boldsymbol{x}^{(t)}, t = 1, ..., N$ being generated from mixture component i using the current (denoted by j) parameter estimate.

Despite the ease of training of GMMs, they have two serious drawbacks when it comes to speech recognition systems.The first one is that they cannot model the sequence information contained in speech features. To balance their inability, we combine GMMs with more general models able to capture sequence information: the Hidden Markov Models, which will be presented next. The second disadvantage, is that, in spite of their huge modeling capabilities, GMMs are statistically inefficient for modeling data lying on or near a nonlinear manifold in the data space; this is the case however for speech features, despite their seemingly high dimensionality.To deal with this matter, there are a number of techniques we can apply to extract

the lower dimensional manifold of the features. We will go into more details about manifolds and speech features in the following chapter.

Hidden Markov Models and acoustic modeling.[YD14]

As we have already mentioned, mixture-of-Gaussian random variables (single- or multidimensional) lack a “temporal" dimension, which would make the length of the random vectors variable, in order to follow the length of the speech sequence we intent to model. Therefore, although Gaussian mixture models are appropriate for short-term sound patterns, we will need to introduce a new model appropriate for sequences of speech acoustic vectors.

Extending the notion of the random variable to the discrete-time random sequence will provide us with the necessary tool to model acoustic vector sequences of variable length. A discrete-time random sequence is a collection with variable length, consisting of random variables indexed by uniformly spaced discrete times.We will focus on the most commonly used class of random sequences which is the Markov sequences.

Markov sequences

The concept of *state* is a key point in Markov sequences. If we think of a system functioning as a Markov sequence generating random variables, then the configuration of the system at each time step is defined by a specific *state* of the sequence. If the state of the Markov sequence is confined to be discrete, then the Markov sequence is called a *Markov chain* and the possible values of each discrete state constitute the discrete state space. When each discrete state value is generalized to be a new random variable (either discrete or continuous) the Markov chain is generalized to the *Hidden Markov Sequence*, also called *Hidden Markov Model* when it characterizes statistical properties of real-world data sequences.The Hidden Markov Model is the tool that we will use to model speech units used, such as sub-phones.

Markov chains

A Markov chain is a discrete-time Markov sequence.Its state space is of discrete nature, finite and each element of the space is associated with a state in the chain:

$$q_t \in s^{(j)}, j \ = \ 1, 2, ..., N$$

where q_t symbols a state.

A Markov chain denoted by $q_1^T = q_1, q_2, ..., 1_T$, is completely characterized by the initial state distribution probabilities (*priors*) and the transition probabilities defined by:

$$P(q_t = s^{(j)} | q_{t-1} = s^{(i)}) \doteq a_{ij}(t), \quad i, j \ = \ 1, 2, ..., N$$

Given the transition probabilities of a Markov chain, the state-occupation probability

$$p_j \doteq P[q_t = s^{(j)}]$$

can be recursively computed by

$$p_i(t) = \sum_{j=1}^{N} a_{ij} p_j(t-1), \quad \forall i$$

Hidden Markov Models[RJ86][Rab89][DHS00]

If the states of a Markov chain are *emitting*, that is, they are able to generate observational output variables, then we call the chain an observable Markov sequence. However, since there is an one-to-one correspondence between the output of the chain and the states, the model is inadequate to describe real-world informational sources such as sequences of speech features.To overcome this limitation, we will add randomness to the Markov chain by associating each state with an observation probability distribution, thus creating the hidden Markov sequence. It is called hidden because the underlying Markov chain is no longer directly observable but it can be observed only through a separate random function characterized by the observation probability distributions, which overlap across the states.

A Hidden Markov Model (*HMM*) is characterized by:

- N, the number of states in the model
- K, the number of distinct observation symbols per state
- The transition probabilities, $\mathbf{A} = [\alpha_{ij}]$, i,j = 1,2,...,N where

  $$\alpha_{ij} = P(q_t = j | q_{t-1} = i), \; i, j \; = \; 1, 2, ..., N$$

- The initial Markov chain prior probabilities

  $$\pi \; = \; [\pi_i], \; i \; = \; 1, 2, ..., N$$

  where π_i = P(q_1=i)

- The observation probability distribution, P($\boldsymbol{o}_t | s^{(i)}$), i = 1,2,...,N.

  If $\boldsymbol{o}_t$ is discrete, the distribution associated with each state gives the probabilities of symbolic observation $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_k$:

  $$b_i(k) = P[\boldsymbol{o}_t = \boldsymbol{v}_k | q_t = i], \quad i \; = \; 1, 2, ..., N.$$

  If the observation probability distribution is continuous, then the parameters $\boldsymbol{\Theta}_i$ in the p.d.f. characterize state i in the HMM.The most common p.d.f used

Figure 2.14: *Generate observation sequence from an HMM .[YD14]*

procedure DRAWFROMHMM(A, π, $P(\mathbf{o}_t|s^{(i)})$)
▷ A is the transition probability
▷ π is the initial state occupation probability
▷ $P(\mathbf{o}_t|s^{(i)})$ is the observation probability given a state (either Eq. 3.7 if discrete or Eq. 3.8 if continuous
Select an initial state $q_1 = s^{(i)}$ by drawing from the discrete distribution π
for $t \leftarrow 1; t \leq T; t \leftarrow t+1$ **do**
Draw an observation $\mathbf{o}_t$ based on $P(\mathbf{o}_t|s^{(i)})$
Make a Markov-chain transition from the current state $q_t = s^{(i)}$ to a new state $q_{t+1} = s^{(j)}$ according to the transition probability a_{ij}, and assign $i \leftarrow j$.
end for
end procedure

in speech processing is, as we have seen, the multivariate mixture of Gaussian distributions:

$$b_i(\boldsymbol{o}_t) = \sum_{m=1}^{M} c_{i,m}\mathcal{N}(\boldsymbol{o}_t|\boldsymbol{\mu_{i,m}}, \boldsymbol{\Sigma_{i,m}})$$

with $\boldsymbol{\Theta}_i = \{c_{i,m}, \boldsymbol{\mu}_{i,m}, \boldsymbol{\Sigma}_{i,m}\}$

Given these parameters one could consider the HMM as a generative model producing a sequence of observational data, $\boldsymbol{o_t}$, $t = 1, 2, ...T$. According to this perspective, the data at each time t is generated from the model according to:

$$\boldsymbol{o_t} = \boldsymbol{\mu_i} + \boldsymbol{r_t}(\boldsymbol{\Sigma_i})$$

where state i at a given time t is determined by the evolution of the Markov chain characterized by α_{ij} and

$$\boldsymbol{r_t}(\boldsymbol{\Sigma_i}) = \mathcal{N}(0, \boldsymbol{\Sigma_i})$$

is a zero-mean, independent and identically distributed (*IID*) residual sequence.Given that $\boldsymbol{\mu_i}$ is constant, the observation $\boldsymbol{o_t}$ is also IID given the state. Consequently, the HMM would produce locally stationary sequences making it appropriate to model sub-phone units.A procedure to generate sequences of observations from an HMM is described in the figure 2.14.

The three basic problems for an HMM.

Given the HMM model as presented above, there are three main problems associated with it that apply to real-world problems:

- The evaluation problem: Suppose we have an HMM $(A_{ij}, b_{ik}, \Theta_i)$.How do we determine the probability that a given sequence of observations was generated by that model?

- The decoding problem: Suppose we have an HMM $(A_{ij}, b_{ik}, \Theta_i)$.How do we determine the most likely hidden state sequence that led to the generation of a given observation sequence?

- The parameter estimation problem: Given the basic structure of an HMM (number of states and number of distinct observation symbols) as well as a set of training observations, how do we determine the parameters $(A_{ij}, b_{ik}, \Theta_i)$?

The evaluation problem. Let $\boldsymbol{q}_1^T$ be a finite length sequence of states in a Gaussian-mixture HMM and $P(\boldsymbol{o}_1^T, \boldsymbol{q}_1^T)$ be the joint likelihood of the observation sequence $\boldsymbol{o}_1^T$ and the state sequence $\boldsymbol{q}_1^T$.

Then, $P(\boldsymbol{o}_1^T|\boldsymbol{q}_1^T)$ denotes the likelihood that the observation sequence $\boldsymbol{o}_1^T$ is generated by the model conditioned on the state sequence $\boldsymbol{q}_1^T$ and is in the form of:

$$\prod_{i=1}^{T} b_i(\boldsymbol{o}_t)$$

whereas the probability of state sequence $\boldsymbol{q}_1^T$ is the product of transition probabilities:

$$P(\boldsymbol{q}_1^T) = \pi_{q_1} \prod_{t=1}^{T-1} a_{q_t q_{t+1}}$$

The joint likelihood $P(\boldsymbol{o}_1^T, \boldsymbol{q}_1^T)$ can be obtained as:

$$P(\boldsymbol{o}_1^T, \boldsymbol{q}_1^T) = P(\boldsymbol{o}_1^T|\boldsymbol{q}_1^T)\boldsymbol{q}_1^T$$

Since the hidden state sequence $\boldsymbol{q}_1^T$ is not known, we will have to sum over all possible state sequences in order to compute the desired probability:

$$P(\boldsymbol{o}_1^T) = \sum_{\boldsymbol{q}_1^T} P(\boldsymbol{o}_1^T, \boldsymbol{q}_1^T)$$

The amount of this computation though, is exponential in the length T of the observation sequence. However, an efficient algorithm (linear complexity in T) to evaluate the above expression has been found, based on the principle of optimality (dynamic programming) [Bel03].The algorithm, known as *Forward algorithm* is described in figure 2.15

The decoding problem. The decoding problem consists of finding the most probable sequence of HMM hidden states given a sequence of observations.It is essentially a path-finding optimization problem that will be dealt with using again the dynamic programming paradigm.In fact, the decoding algorithm, also known as the *Viterbi algorithm* (figure 2.16) is very similar to the *Forward* algorithm presented above.

The Viterbi algorithm returns the maximum joint likelihood of the observation and state sequence as well as the corresponding state transition path.The optimal path for a left-to-right HMM, i.e. an HMM where transitions are only allowed in the forward direction, is equivalent to the information required to determine the optimal segmentation of the HMM states to match the observation sequence.

Figure 2.15: *The Forward algorithm for HMM probability evaluation.[JM09]*

function FORWARD(*observations* of len T, *state-graph* of len N) **returns** *forward-prob*

create a probability matrix *forward[N+2,T]*
for each state s **from** 1 **to** N **do** ; initialization step
$forward[s,1] \leftarrow a_{0,s} * b_s(o_1)$
for each time step t **from** 2 **to** T **do** ; recursion step
for each state s **from** 1 **to** N **do**

$$forward[s,t] \leftarrow \sum_{s'=1}^{N} forward[s',t-1] * a_{s',s} * b_s(o_t)$$

$$forward[q_F,\mathrm{T}] \leftarrow \sum_{s=1}^{N} forward[s,T] * a_{s,q_F}$$; termination step

return $forward[q_F,T]$

Figure 2.16: *The Viterbi algorithm for HMM decoding.[YD14]*

function VITERBI(*observations* of len T, *state-graph* of len N) **returns** *best-path*

create a path probability matrix *viterbi[N+2,T]*
for each state s **from** 1 **to** N **do** ; initialization step
$viterbi[s,1] \leftarrow a_{0,s} * b_s(o_1)$
$backpointer[s,1] \leftarrow 0$
for each time step t **from** 2 **to** T **do** ; recursion step
for each state s **from** 1 **to** N **do**

$$viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$$

$$backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s}$$

$$viterbi[q_F,T] \leftarrow \max_{s=1}^{N} viterbi[s,T] * a_{s,q_F}$$; termination step

$$backpointer[q_F,T] \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T] * a_{s,q_F}$$; termination step

return the backtrace path by following backpointers to states back in time from $backpointer[q_F,T]$

Figure 2.17: *The Backward algorithm for HMM decoding.[DHS00]*

1 **initialize** $\omega(T), t = T, a_{ij}, b_{jk}$, visible sequence V^T
2 **for** $t \leftarrow t - 1$;
4 $\beta_j(t) \leftarrow \sum_{i=1}^{c} \beta_i(t+1) a_{ij} b_{jk} v(t+1)$
5 **until** $t = 1$
7 **return** $P(V^T) \leftarrow \beta_i(0)$ for the known initial state
8 **end**

The parameter estimation problem. The goal in HMM training, is to extract the model parameters so as to minimize the empirical risk with respect to the joint likelihood loss, involving a sequence of acoustic data and their corresponding linguistic labels. To estimate the parameters of an HMM model given training data, we will apply the Expectation-Maximization algorithm, also known as *Baum-Welch algorithm* in the context of HMM training.First however, we will introduce the *Backward algorithm*, which is a part of the EM computation for HMMs.

The *Backward algorithm* (figure 2.17) is very similar to the *Forward* but now we are moving backwards, that is, the algorithm computes

$$\beta_t(i) = P(\boldsymbol{o}_{t+1}^T | q_t = i), \ t \ = \ 1, ..., T-1$$

The EM algorithm uses both the Forward and Backward algorithms in the expectation E-step in order to obtain:

- the posterior state transition probabilities in the HMM

$$\xi_t(i,j) = \frac{\alpha_t(i)\beta_{t+1}(j)\alpha_{ij} \exp N_{t+1}(j)}{P(\boldsymbol{o}_1^T|\theta_0)}, \quad t \ = \ 1, ..., T-1$$

  where $N_t(i)$ is the logarithm of the Gaussian p.d.f. associated with state i,

- the posterior state occupancy probabilities

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j)$$

In the maximization M-step the parameters are computed using the current estimates of ξ and γ:

$$\hat{\alpha}_i j = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\hat{\Sigma}_i = \frac{\sum_{t=1}^{T} \gamma_t(i)(\boldsymbol{o}_t - \hat{\boldsymbol{\mu}}_i)(\boldsymbol{o}_t - \hat{\boldsymbol{\mu}}_i)^T}{\sum_{t=1}^{T} \gamma_t(i)}$$

$$\hat{\mu}_i = \frac{\sum_{t=1}^{T} \gamma_t(i)\boldsymbol{o}_t}{\sum_{t=1}^{T} \gamma_t(i)}$$

for each state i.

HMMs in speech modeling and recognition.[YD14][ST04] We have already seen that HMMs can be used as generative models to produce sequences of observations. They are able to produce sequences of variable length, which is of utmost importance for speech modeling and recognition, and they have proven to be good models for the statistical distribution of sequence data of speech acoustics. Consequently, HMMs have become very popular in the ASR community.

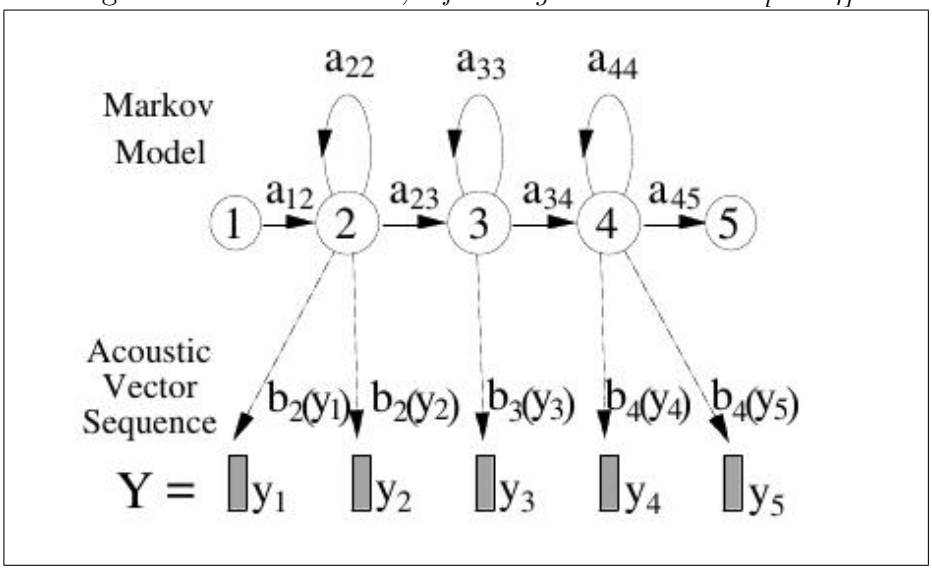
GMM/HMMs in ASR. As we have seen, a GMM/HMM is a statistical model that describes two dependent random processes, an observable and a hidden, where the observable is assumed to be generated by a hidden state according to a Gaussian mixture p.d.f.

In the context of speech, we think of a single HMM as a system generating acoustic features of a modeled speech unit, which can be a word, a syllable, a single phone , or, usually a context-dependent phone. The motivation behind choosing context-dependent phones, and therefore states, stems from the effort to reduce output variability of speech feature vectors associated with each state, leading to more detailed generative modeling. However this leads to an expansion of the state space, which we have seen how it is dealt with, at the beginning of the chapter.

The most common HMM model is the three-state left-to-right HMM (figure 2.18). The number of states is chosen based on the behavior of the vocal tract. It goes through three states when uttering a phone : changing from the previous phone, steady pronunciation of the current phone and changing to the next phone.

However, despite their advantages, HMMs have been found to have several weaknesses. The temporal independence of speech data conditioned on the HMM states and the lack of proven correlation between acoustic features and ways in which speech sounds are produced (e.g. speaking rate and style) have motivated the replacement of GMMs associated with each state by more realistic, temporally correlated dynamic systems containing hidden, continuous-valued dynamic structure (e.g. [Bil03]).

Figure 2.18: *Three-state, left-to-right HMM model.[ST04]*



2.4.5 The Decoder

[MPR01] The components described above are finally combined in the decoder which will find the most likely word sequence given a sequence of feature vectors. In order to define a common framework for the representation and use of the aforementioned models in LVCSR, we represent them by Weighted Finite State Transducers; an approach which provides significant algorithmic and engineering benefits.

A *finite-state transducer* is a finite automaton whose state transitions are labeled with both input and output symbols.Consequently, a path through the transducer encodes a mapping from an input to an output symbol sequence. *Weighted finite-state transducers (WFSTs)* , in addition to input/output symbols, have weights on the transitions which accumulate along paths to compute the total cost of a mapping from an input to an output symbol sequence.Thus, seeing the components of an ASR system as WFSTs and mathematical operations on them, allows for generalizing and efficiently implementing many of the common processing methods in speech recognition. We will briefly present the WFST framework in speech recognition, yet first we will introduce some necessary notation and algorithms.

Notation and Algorithms

- A *semi-ring* $(\mathbb{K}, \bigoplus, \bigodot, \bar{0}, \bar{1})$ is defined by a set of values $\mathbb{K}$, two binary operations of addition ($\bigoplus$) and multiplication ($\bigodot$) and two designated values $\bar{0}$ and $\bar{1}$. The addition operation is associative, commutative and has $\bar{0}$ as the identity element. The multiplication operation is associative, has $\bar{1}$ as the identity element, is distributive with respect to addition and has $\bar{0}$ as the annihilator

Figure 2.19: *Composition algorithm .[MPR01].*

```
WEIGHTED-COMPOSITION(T_1, T_2)
 1  Q ← I_1 × I_2
 2  S ← I_1 × I_2
 3  while S ≠ ∅ do
 4       (q_1, q_2) ← HEAD(S)
 5       DEQUEUE(S)
 6       if (q_1, q_2) ∈ I_1 × I_2 then
 7            I ← I ∪ {(q_1, q_2)}
 8            λ(q_1, q_2) ← λ_1(q_1) ⊗ λ_2(q_2)
 9       if (q_1, q_2) ∈ F_1 × F_2 then
10            F ← F ∪ {(q_1, q_2)}
11            ρ(q_1, q_2) ← ρ_1(q_1) ⊗ ρ_2(q_2)
12       for each (e_1, e_2) ∈ E[q_1] × E[q_2] such that o[e_1] = i[e_2] do
13            if (n[e_1], n[e_2]) ∉ Q then
14                 Q ← Q ∪ {(n[e_1], n[e_2])}
15                 ENQUEUE(S, (n[e_1], n[e_2]))
16            E ← E ∪ {((q_1, q_2), i[e_1], o[e_2], w[e_1] ⊗ w[e_2], (n[e_1], n[e_2]))}
17  return T
```

element: $\forall a \in \mathbb{K}, a \bigodot \bar{0} = \bar{0} \bigodot a = \bar{0}$. If $\bigodot$ is also commutative the semi-ring is called *commutative*, which will be the case for all the semi-rings mentioned later.

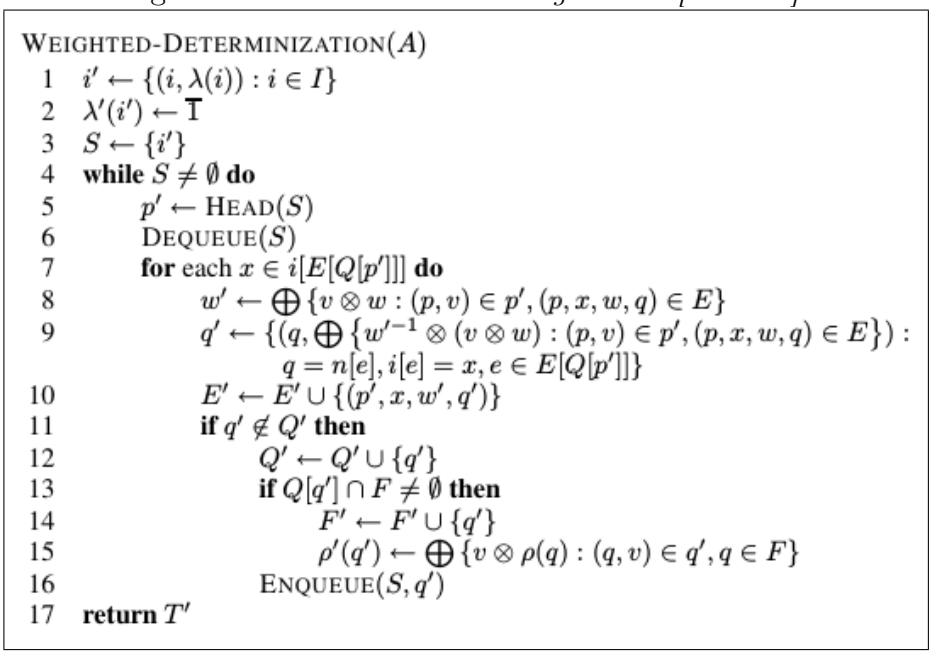
- A WFST $T = (\mathcal{A}, \mathcal{B}, \mathcal{Q}, \mathcal{I}, \mathcal{F}, \mathcal{E}, \lambda, \rho)$ over a semi-ring $\mathbb{K}$ is specified by a finite input alphabet $\mathcal{A}$, a finite output alphabet $\mathcal{B}$, a finite set of states $\mathcal{Q}$, a set of initial states $\mathcal{I} \subseteq \mathcal{Q}$, a set of final states $\mathcal{F} \subseteq \mathcal{Q}$, a finite set of transitions $\mathcal{E} \subseteq Q \times (\mathcal{A} \cup \epsilon) \times (\mathcal{B} \cup \epsilon) \times \mathcal{K} \times \mathcal{Q}$, an initial state weight assignment $\lambda : \mathcal{I} \to \mathcal{K}$ and a final state weight assignment $\rho : \mathcal{F} \to \mathcal{K}$. $\mathcal{E}[q]$ denotes the sum of the number of states and transitions of $\mathcal{T}$.

Based on the notation just introduced we will present the operations on WFSTs that will be used in speech recognition applications.

- *Composition* is the basic operation that allows us to create complex WFSTs from simpler ones, thus putting together all the fundamental components of an ASR system.

- *Determinization* removes non-determinancy from the WFST by ensuring that each state has no more than a single output transition for a given input label.Not every WFST is determinizable, however, there is a *pre-determinization* algorithm that can be used to make determinizable an arbitrary WFST over the tropical semi-ring $((\mathcal{R} \cup \{-\infty, +\infty\}, min, +, +\infty, 0))$ by inserting transitions labeled with special symbols.The determinization operation is particularly important in ASR considering the redundancy found in e.g. the WFST

Figure 2.20: *Determinization algorithm .[MPR01].*



```
WEIGHTED-DETERMINIZATION(A)
 1  i' ← {(i, λ(i)) : i ∈ I}
 2  λ'(i') ← T̄
 3  S ← {i'}
 4  while S ≠ ∅ do
 5      p' ← HEAD(S)
 6      DEQUEUE(S)
 7      for each x ∈ i[E[Q[p']]] do
 8          w' ← ⊕{v ⊗ w : (p, v) ∈ p', (p, x, w, q) ∈ E}
 9          q' ← {(q, ⊕{w'^-1 ⊗ (v ⊗ w) : (p, v) ∈ p', (p, x, w, q) ∈ E}) :
                q = n[e], i[e] = x, e ∈ E[Q[p']]}
10          E' ← E' ∪ {(p', x, w', q')}
11          if q' ∉ Q' then
12              Q' ← Q' ∪ {q'}
13              if Q[q'] ∩ F ≠ ∅ then
14                  F' ← F' ∪ {q'}
15                  ρ'(q') ← ⊕{v ⊗ ρ(q) : (q, v) ∈ q', q ∈ F}
16              ENQUEUE(S, q')
17  return T'
```

representing the pronunciation lexicon.A deterministic lexicon WFST will contain at most one path for any input string, thus less time and space will be needed to process the string (figure 2.20).

- *Minimization* transforms a WFST to an equivalent one with the fewest possible states and transitions which saves both space and time during its processing. Before minimising the transducer, a form of re-weighting (*weight-pushing*) is performed to redistribute weight among transitions as well as to improve search operations.

WFST models in speech recognition

There are four principal models of weighted finite-state transducers that are used in speech recognition:

- G : the word level grammar
- L : the pronunciation lexicon
- C : the context-dependency transducer
- H : the HMM transducer

The word level *grammar transducer G* has a state w_i for each word and transitions are added according to the N-gram model used in the grammar. For example,

a bigram grammar has a transition from state w_1 to w_2 for every bigram w_1w_2 seen in the training corpus.The label of the transition is then w_2 and the weight is the negative logarithm of the transition probability $(-\log(\hat{p}(w_2|w_1)))$.To deal with unseen N-grams while keeping the complexity of constructing the WFST low, we introduce a back-off state b.An unseen bigram w_1w_3 is then represented as two transitions: an ϵ transition w_1b with weight $-\log(\beta(w_1))$ and a transition bw_3 with weight $-\log(\hat{p}(w_3))$. Because ϵ transitions introduce non-determinism in the WFST, we can treat ϵ labels as normal symbols during determinization, thus keeping the number of transitions low - otherwise transitions become quadratic with respect to the size of the vocabulary after determinization.

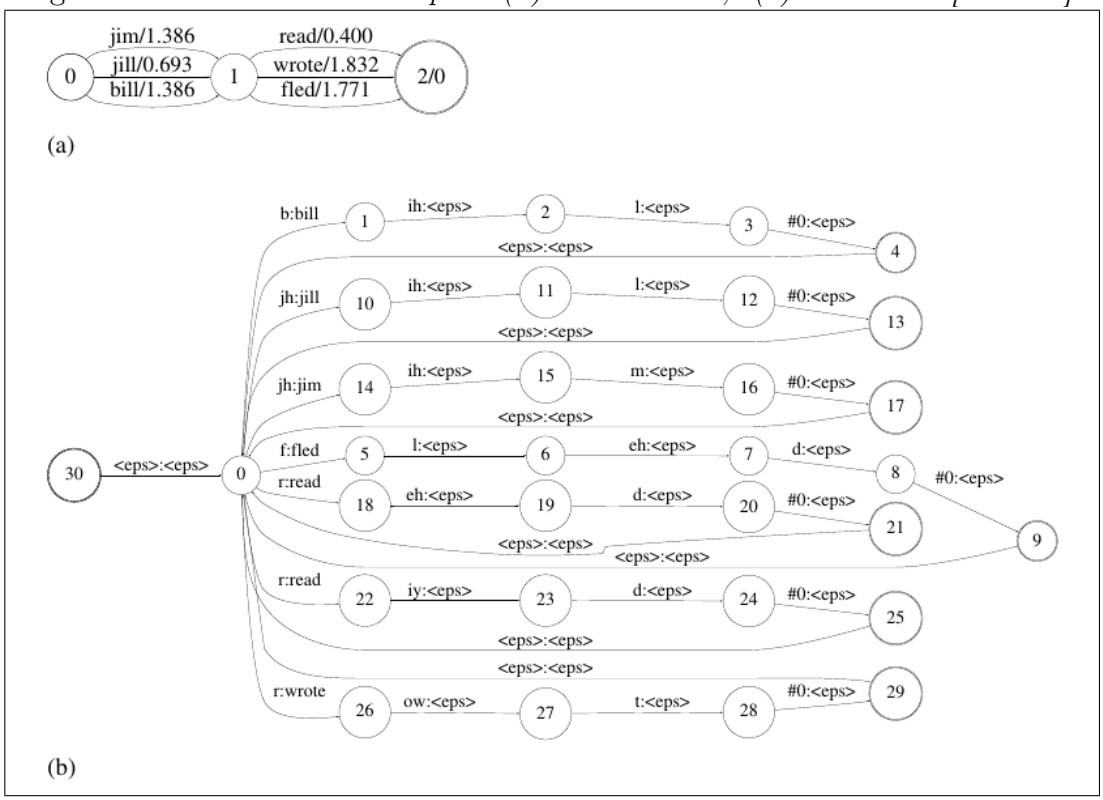
The *pronunciation lexicon transducer* L is the Kleene closure of the union of individual word pronunciations.It is easy to see that L is, in general, not determizable, considering the existence of homophones and the fact that the first word of the output string might be impossible to determine before the entire phone string is scanned. To make L determinizable we add a number of disambiguation symbols $(\#_i)$ as well as a symbol $(\#_0)$ to mark the end of the phonetic transcription of each word. The new lexicon transducer after the addition of these symbols is determinizable and is denoted by $\bar{L}$.

The *context dependency transducer* represents a mapping from context independent phones to context dependent units.The transducer has a state for every pair of phones with label (a, b) ,where a is the past and b is the future phone, and transitions marked as *a:phone/left context_ right context*.To apply the context dependent triphone models often used in ASR in the WFST framework, we need to be able to compose a context dependency transducer with the lexicon transducer introduced earlier.In order to make this composition feasible we first invert the context dependency transducer (interchange input and output labels) and create the transducer C which maps from context dependent triphones to context independent phones.

The final transducer that is used in the decoding process is the *Hidden Markov Models transducer* H which is the closure of the union of the individual HMMs of the acoustic model.

Applying the composition operation on the transducers presented here will output the decoding transducer which we will further optimize to help decoding and make it as efficient as possible.First, composing the lexicon and grammar transducers gives a new transducer that maps from phones to word strings restricted to the grammar $(L \circ G)$.The resulting transducer is then composed with the context dependency transducer C and the resulting transducer maps from context dependent phones to word strings restricted to the grammar $(C \circ L \circ G)$.Finally, $C \circ L \circ G$ is composed with the HMM transducer H and the outcome is a transducer that maps

Figure 2.21: *Transducer example: (a) Grammar G, (b) Lexicon $\bar{L}$ [MPR01].*



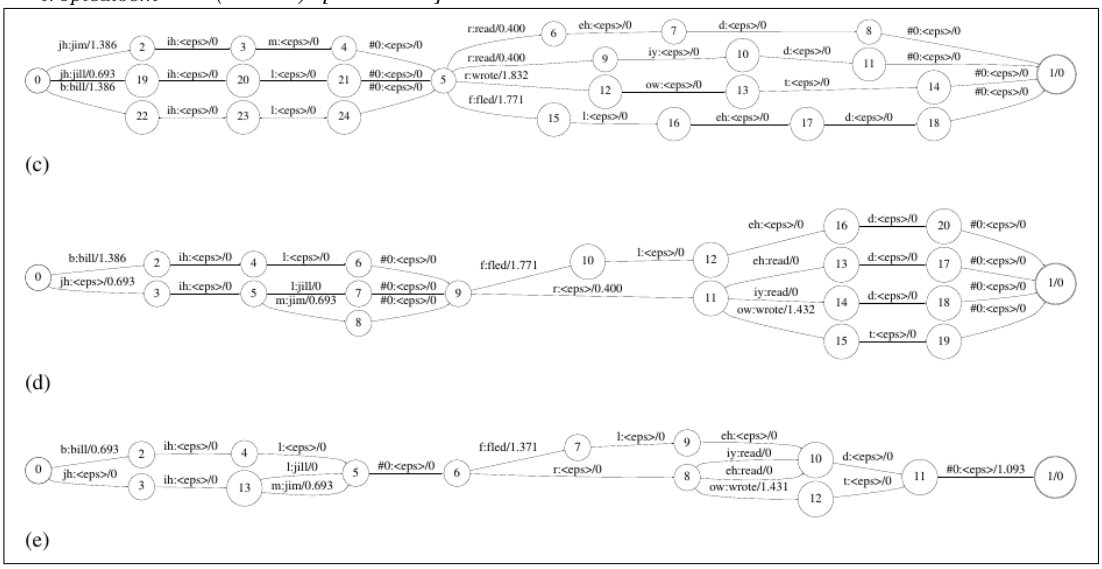
from the identifiers of context-dependent HMM states to word strings restricted to G $(H \circ C \circ L \circ G)$.

After the construction of the final transducer we have to determinize and minimize it so that we have the optimal transducer for recognition.By determinizing it we eliminate redundant paths which reduces recognition time.Moreover, if the determinization is applied after each composition stage during the construction of the graph, the composition operations that follow are performed more efficiently and the total size of the transducer is reduced.Once we have determinized the transducer we can further optimize it by minimization and the weight pushing process that precedes it. As far as weigh pushing is concerned, provided we are using the -log semi-ring, we can have large efficiency gains during the Viterbi beam search, and make sure that all weights leaving a state sum up to one (which is desirable for a language model).

Given the *HCLG* transducer and an utterance of N frames, how do we decode it, namely how do we find the most likely word sequence and its corresponding state-level alignment?

The first step is to construct an acceptor U of the utterance, which is a WFST with identical input and output symbols. The acceptor has $N+1$ states with an arc for each (time,context-dependent HMM state) combination and weights on these

Figure 2.22: *Transducer example: (c)* $\bar{L} \circ G$*, (d)*$\bar{L} \circ G$ *determinized, (e)* $min_{tropicalsem}$ *det(*$\bar{L} \circ G$*) [MPR01].*



arcs the scaled negated acoustic log likelihoods.Following the construction of U, we compose it with the decoding WFST and get a new WFST S:

$$S = U \circ HCLG$$

The decoding problem now reduces to finding the best path through S.The input symbol sequence of the best path is the state-level alignment and the output sequence is the corresponding sentence [PHB+12].

2.4.6 Evaluation

The performance of a large vocabulary continuous speech recognition system is evaluated based on the Word Error Rate (WER) metric. There are three types of possible errors when recognizing continuous speech: (a) substitution errors, i.e. the wrong word is recognized, (b) word deletions, namely the presence of a word is not recognized at all and (c) word insertions, meaning that an extra word is recognized.If we define the number of words in the text speech as N and denote with $C(\cdot)$ the number of errors of each type, then WER is defined as:

$$WER = \frac{C(substitutions) + C(deletions) + C(insertions)}{N}$$

ADD STATE OF THE ART WER + MENTION SYSTEM AND DATASET

Chapter 3

Manifold and Representation learning

In the first chapter we briefly saw an intuitive definition of manifolds. In this chapter we will give a mathematical definition of manifolds and examine the role that they play in representation learning.

Representation learning is the process of learning good representations of the data, that make it easier to extract information that will be useful when we build classifiers or other predictors. A good representation is one that captures the posterior distribution of the latent explanatory factors that led to the observed data and as such, it is particularly effective as input to a supervised predictor [BCV13].

3.1 Manifold learning

[BGC15]

Manifold learning is an approach to representation learning that builds on the *manifold hypothesis*([Cay05]). According to this hypothesis, real-world data presented in high dimensional spaces is expected to concentrate on the vicinity of a manifold $\mathcal{M}$ of much lower dimensionality $d_{\mathcal{M}}$, embedded in high dimensional input space $\mathcal{R}^{d_x}$.

This hypothesis is a kind of prior assumption about the data generating distribution that seems particularly fit to artificial intelligence tasks involving data, such as speech, text, music and images. The common thing about such data, is that if we choose configurations of the observed variables at random, according to a factored distribution (e.g. a uniform distribution), it is very unlikely to generate the kind of observations we want to model. For example, if we uniformly pick values for an acoustic signal, we will most likely create an unnatural speech sample. Consequently, manifold learning algorithms try to discover where probability concentrates in the

input space, as it will be a very small region of the total space of configurations.

Furthermore, making very small changes to the input data, e.g the values of the pixels of an image, will produce natural-looking data similar to the original input. This is another aspect of the manifold hypothesis, that is, probable configurations are likely to be surrounded by other possible configurations.

3.1.1 Mathematical formulation of manifold learning

[Cay05]

We will now give the mathematical definition of manifolds and mathematically formulate manifold learning.

Definition 15. *A homeomorphism is a continuous function whose inverse is also continuous.*

Definition 16. *A d-dimensional manifold $\mathcal{M}$ is a set that is locally homeomorphic with $\mathcal{R}^d$. That is, $\forall\ x \in \mathcal{M}$ there is an open neighborhood around x, $\mathcal{N}_x$, and a homeomophism $f : \mathcal{N}_x \to \mathcal{R}^d$. These neighborhoods are known as coordinate patches and the map as a coordinate chart. The image of the coordinate charts is referred to as the parameter space.*

Intuitively, the dimension d of the manifold indicates the number of independent ways by which a probable configuration can be locally transformed into another probable configuration (remember the example of tiny changes in pixel values of an image). Another concept associated with this intuition is the set of *tangent planes* of a manifold. A tangent plane at a point x on a d-dimensional manifold is given by d basis vectors which cover the local dimensions of variation on the manifold, i.e. they show how x can be changed while staying on the manifold (figure 3.1).

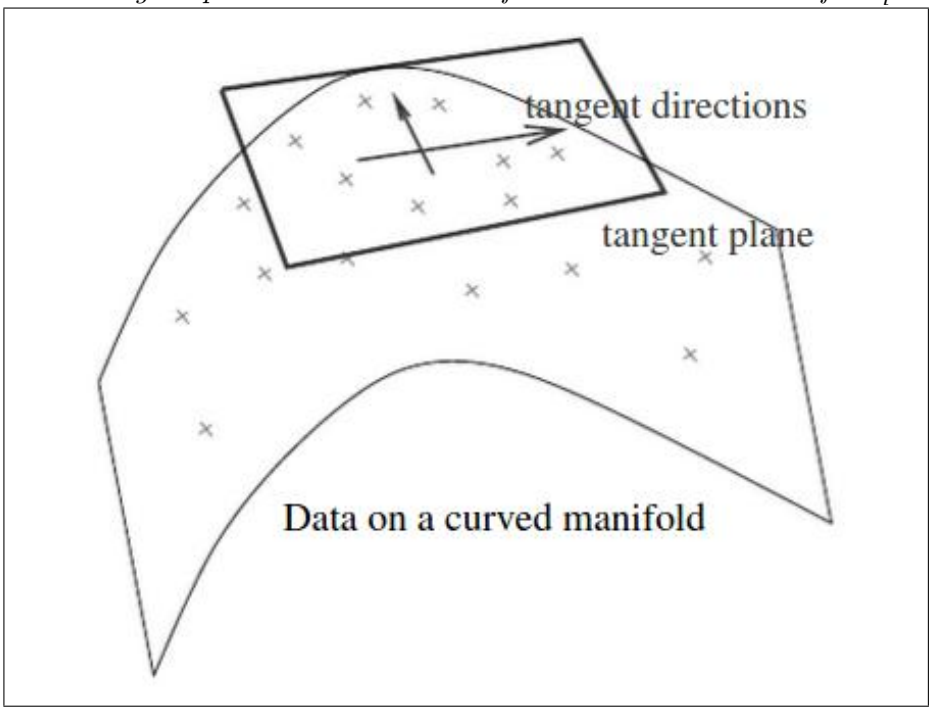
We will only deal with manifolds which are subsets of $\mathcal{R}^D$ where $D >> d$. That is, the manifold will lie in $\mathcal{R}^D$ but will be homeomorphic to a lower dimensional space $\mathcal{R}^d$.

Before moving on to manifold learning we will further constrain the manifolds of interest.

Definition 17. *A smooth or differentiable manifold is a manifold such that each coordinate chart is differentiable with a differentiable inverse. We say that such a chart is a diffeomorphism.*

Definition 18. *An embedding of a manifold $\mathcal{M}$ into $\mathcal{R}^D$ is a smooth homeomoprhism from M to a subset of $\mathcal{R}^D$.*

Figure 3.1: *Tangent plane and directions of variation on the manifold [BGC15].*



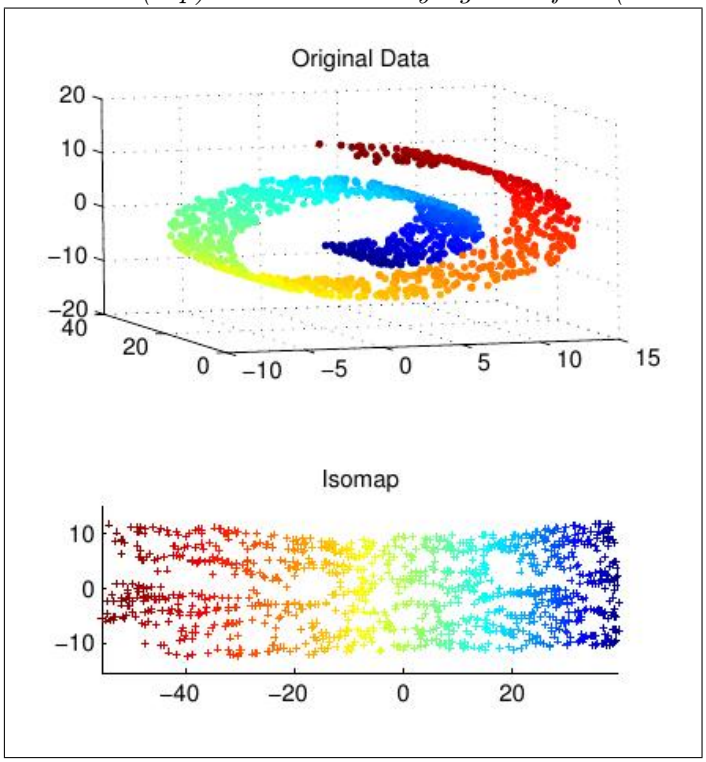
An embedding is an important concept that characterizes a manifold $\mathcal{M}$ and is a representation of the data points on or projected on the manifold. It is usually given by a low-dimensional vector, with less dimensions than the surrounding space, in which the manifold lies. The learning algorithms that will be presented later will learn either directly an embedding for each training data point, or a more general mapping, also called encoder or representation function, that maps any point in the high-dimensional input space to its embedding.

Definition 19. *Suppose we are given a set of $\mathcal{N}$ points $x_1, x_2, ..., x_N \in \mathcal{R}^D$ that lie on a d-dimensional manifold M which can be described by a single coordinate chart $f : \mathcal{M} \to \mathcal{R}^d$. Manifold learning is the process of finding $y_1, y_2, ..., y_N \in \mathcal{R}^d$, where $y_i \doteq f(x_i)$.*

In other words, manifold learning is the process of trying to learn the manifold of a set of available points, or find an embedding of the underlying manifold. The most frequently used example in the manifold literature is the Swiss roll, which is a two-dimensional manifold embedded in the three-dimensional space ($\mathcal{R}^3$). What is important to note is the fact that the distances between points in the original dataset are maintained in the two-dimensional dataset. This is due to the chart f between the two datasets being a homeomorphism.

The fact that the point distances between the datasets are preserved is a basic characteristic of the manifold learning algorithms that will be presented in the next

Figure 3.2: *Swiss roll (top) and the underlying manifold (bottom). [Cay05].*



section.

3.2 Manifold learning algorithms

3.2.1 Dimensionality reduction

[Ver08] We know that most learning algorithms performing well in low-dimensional datasets, perform poorly when applied to high-dimensional datasets. This phenomenon is known as the *curse of dimensionality*. In order to effectively apply learning algorithms in high-dimensional data, we would like first to map it to a low-dimensional space, while preserving much of the important information, and then run the algorithms in the projected space. This process is known as *dimensionality reduction*.

One way to verify the quality of a dimensionality reduction technique is to test how well the mapping preserves pairwise distances. This idea is based on the assumption that the distances between points in space relate to the dissimilarity between the corresponding observations. It is evident that this distance preservation would be much easier if we could find the underlying manifold structure of the data; we would then limit our work on points lying on the manifold and not in the whole ambient space.

A question that arises here, is, whether we can project any dataset on a lower-dimensional space, which maintains structure. The *Johnson-Lindenstrauss lemma* states that any n points in high dimensional euclidean space can be mapped onto k dimensions where $k \geq \mathcal{O}(\frac{\log n}{\epsilon^2})$ without distorting the euclidean distance between any two points more than a factor of $1 \pm \epsilon$ [Mah09], [DG03].

Lemma 2. *For any $0 < \epsilon < 1$ and any integer n, let k be a positive integer such that $k \geq 4(\frac{\epsilon^2}{2} - \frac{\epsilon^3}{3})^{-1} \ln n$.*

Then for any set V of n points in $\mathcal{R}^d$ there is a map $f : \mathcal{R}^d \to \mathcal{R}^k$ such that $\forall u, u \in V$:

$$(1 - \epsilon)\|u - v\|^2 \leq \|f(u) - f(v)\|^2 \leq (1 + \epsilon)\|u - v\|^2$$

This map can be found in randomized polynomial time.

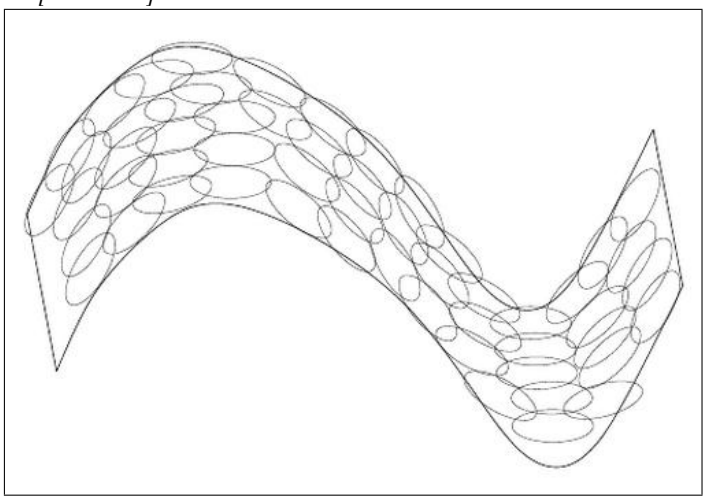
A general outline of the proof is the following:

- Construct a random projection over k-dimensional subspaces.
- Prove that the expected value of the euclidean distance of the random projection is equal to the euclidean distance of the orginal subspace.
- Prove that the variance of the euclidean distance is greater than the specified error factor only with a probability $\frac{2}{n^2}$, such that the union bound of this probability across all points is less than $1 - \frac{1}{n}$.

3.2.2 Algorithms

So far, most manifold learning algorithms are unsupervised learning procedures attempting to uncover the underlying manifold structure. They use a nearest-neighbor graph which has one node per training example and edges connecting near neighbors. Each node is associated with a tangent plane which spans the directions of variations associated with a neighborhood of the graph and a coordinate system that associates each training example with an embedding. This coordinate system can be thought of as a local Euclidean system or a locally flat Gaussian, with very small variance in the directions orthogonal to the plane, and a very large variance in the directions defining the local coordinate system (figure 3.3). With this approach, a generalization is possible to new examples by a form of interpolation between neighbors and a global coordinate system can be obtained through an optimization or solving a linear system. Furthermore, by formulating a problem in terms of graph structures one avoids making any assumptions about the data distribution ([YXZ+07]).

Figure 3.3: *Tangent planes are tiled together to cover the manifold, forming a global coordinate system [BGC15].*



However, the methods based on nearest-neighbor graph work well provided we have a very large number of training examples to cover all the curves and twists of the underlying manifold. This need for huge amounts of data, as well as the fact that manifolds of interest in AI, such as the ones underlying in data from speech and images, have many curves and twists, have motivated the use of deep learning methods and distributed representations to uncover the manifold structure. These methods try to learn a coordinate system for the main latent factors that explain the structure of the underlying generating distribution.

Furthermore, such algorithms work in batch mode and they do not provide a way of mapping new points from the high to the low dimensional space without re-running the algorithm from scratch; to tackle this issue, a number of approaches have been suggested ([AE], [SR03],[BPV03]).

We will now introduce some basic manifold learning algorithms.

Isomap

Isometric feature mapping (*ISOMAP*) was one of the first manifold learning algorithms. Isomap contains the following three steps:

- Construct the k nearest-neighbor graph. The number of neighbors k is defined after trying with various values and comparing the results. It is also possible to find the neighbors with an approximate nearest-neighbor procedure, e.g. use neighborhoods with a certain ϵ radius.

- Estimate the distances along the manifold (geodesic distances) between input points using shortest-path distances on the neighbor graph constructed in the

Figure 3.4: *Classical Multidimensional Scaling [Cay05].*

classical Multidimensional Scaling
input: $D \in \mathbb{R}^{n\times n}(D_{ii} = 0,\ \ D_{ij} \geq 0),\ \ d \in \{1, \ldots, n\}$

1. Set $B := -\frac{1}{2}HDH$, where $H = I - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$ is the centering matrix.
2. Compute the spectral decomposition of B: $B = U\Lambda U^\top$.
3. Form Λ_+ by setting $[\Lambda_+]_{ij} := \max\{\Lambda_{ij}, 0\}$.
4. Set $X := U\Lambda_+^{1/2}$.
5. Return $[X]_{n\times d}$.

previous step.

Isomap assumes that there is an isometric chart mapping the input points to the low-dimensional space, i.e. it preserves the distances between points. If the manifold is smooth enough then the geodesic distance between nearby points is almost linear. Given that the manifold locally resembles the Euclidean space, the Euclidean distance can be used to estimate distances. On the other hand, for points that are far apart from each other, we cannot estimate distances in the same way, because the manifold structure is not the same globally as it is locally. To compute such distances we use shortest-paths algorithms on the neighbors graph we have constructed.

- Using Multidimensional Scaling find points in lower-dimensional Euclidean space whose interpoint distances match the geodesic distances found in the previous step.

  In the third step Isomap finds points whose Euclidean distances equal the geodesic distances found earlier. Supposing that the manifold is isometrically embedded, we are certain that such points exist and are unique to translation and rotation. To find such points we apply a technique called *Multidimensional scaling* (figure 3.4), which, given a matrix $D \in \mathcal{R}^{n\times v}$ of dissimilarities constructs a set of points whose Euclidean distances from each other match closely those in D.

Isomap has the following two important properties. First, it automatically estimates the dimensionality of the underlying manifold: the number of non-zero eigenvalues found by MDS is the underlying dimensionality.

Second, under the following assumptions, Isomap is guaranteed to recover the parameterization of a manifold.

- The manifold is isometrically embedded into $\mathcal{R}^D$.

Figure 3.5: *ISOMAP [Cay05].*

Isomap
input: $x_1, \ldots, x_n \in \mathbb{R}^D$, k

1. Form the k-nearest neighbor graph with edge weights $W_{ij} := \|x_i - x_j\|$ for neighboring points x_i, x_j.
2. Compute the shortest path distances between all pairs of points using Dijkstra's or Floyd's algorithm. Store the squares of these distances in D.
3. Return $Y :=$cMDS(D).

- The underlying parameter space is convex, or intuitively, the parameter space of the manifold cannot contain any holes.
- The manifold is compact and well-sampled everywhere.

Locally Linear Embedding

The intuition behind Locally Linear Embedding (LLE) stems from thinking of the manifold as a collection of overlapping cooordinate patches. If the neighborhood sizes are small and the manifold does not have many curves or twists, then these patches will be approximately linear. The concept is to find these patches, discover their geometry and find a mapping from the manifold to $\mathcal{R}^d$ that preserves the local geometry and is approximately linear. Since these patches are overlapping, the local reconstructions will combine into a global one.

The first step of LLE tries to model the manifold as a collection of linear patches and uncover their geometry. In order to do so, it represents each point x_i as a weighted, convex combination of its nearest neighbors. The weights are chosen based on minimizing the squared error:

$$\|x_i - \sum_{j \in N(x_i)} W_{ij} x_j\|^2$$

where $N(i)$ is the set of nearest-neighbors of x_i.

The weight matrix W reveals the layout of the neighbors around each point; thus it is W that captures the local geometry of each patch. Apart from the convexity, we impose the following constraint on the weights:

$$W_{ij} = 0 \; if \; j \notin N(i)$$

These two constraints have an important physical meaning: the convexity makes the weights invariant to global rotation, translation and scaling, e.g.

$$\|x_i + a - \sum_{j \in N(x_i)} W_{ij}(x_j + a)\|^2 = \|x_i - \sum_{j \in N(x_i)} W_{ij} x_j\|^2$$

whereas the second enforces locality on the patches.

The above minimization problem has a closed-form solution for the weight matrix which, for each x_i is given by:

$$\hat{W}_i = \frac{\sum_k C_{jk}^{-1}}{\sum_{lm} C_{lm}^{-1}}$$

where C is the local covariance matrix

$$C_{jk} \doteq (x_i - \eta_j)^T (x_i - \eta_k)$$

and η_j, η_k are neighbors of x_i. $\hat{W} \in \mathcal{R}^{n \times k}$ is then transformed into the sparse $W \in \mathcal{R}^{n \times n}$ by setting $W_{ij} = \hat{W}_{il}$ if x_j is the l^{th} neighbor of x_i and $W_{ij} = 0$ if $j \notin N(i)$.

The second step in LLE is to find a configuration of points in the d-dimensional parameter space whose local geometry is described well by W. Contrary to Isomap LLE cannot discover the number of dimensions d; we have to provide it based on prior knowledge about the parameter space or estimate it using other techniques (e.g. Conformal Eigenmaps, [SS05]). Such a configuration is found by minimizing the cost function:

$$\sum_i \|y_i - \sum_j W_{ij} y_j\|^2 = Y^T[(I - W)^T(I - W)]Y = Y^T M Y$$

with respect to $y_1, y_2, ..., y_n \in \mathcal{R}^d$ and under constraints: $Y^T Y = I$ and $\sum_i Y_i = \mathbf{0}$. The first constraint forces the solution to be of rank d whereas the second centers the embedding around the beginning of the axes. Minimizing the cost function corresponds to setting the columns of Y to the eigenvectors that correspond to the minimum d eigenvalues of $\mathcal{M}$. However, the minimum (eigenvector,eigenvalue) pair is ($\mathbf{1}$,0). Therefore, to avoid the last coordinate being identical for all points, we pick the minimum d eigenvectors that are not constant.

Laplacian Eigenmaps

The Laplacian Eigenmaps algorithm is based on the concept of the Laplacian of a graph: Given a graph and the corresponding matrix containing edge weights, the Laplacian of the graph is defined as $L \doteq D - W$, where D is defined is the diagonal

Figure 3.6: *LLE [Cay05].*

Locally Linear Embedding
input: $x_1, \ldots, x_n \in \mathbb{R}^D$, d, k

1. *Compute reconstruction weights.* For each point x_i, set
$$W_i := \frac{\sum_k C_{jk}^{-1}}{\sum_{lm} C_{lm}^{-1}}.$$
2. *Compute the low-dimensional embedding.*
   - Let U be the matrix whose columns are the eigenvectors of $(I - W)^\top (I - W)$ with nonzero accompanying eigenvalues.
   - Return $Y := [U]_{n \times d}$.

matrix defined as $D_{ii} = \sum_j W_{ij}$. The eigenvalues of L reveal information about the structure of the graph, such as whether it is complete or connected. Since in our case the weight matrix contains information about the neighborhoods of the points, its Laplacian will help us discover local information about the underlying manifold.

In Laplacian Eigenmaps we have two choices for constructing for the weight matrix W.Either use the simple scheme:

$$W_{ij} = \begin{cases} 1 & \text{if } j \in N(i) \\ 0 & \text{if } j \notin N(i) \end{cases}$$

or use the Gaussian heat kernel:

$$W_{ij} = \begin{cases} e^{\frac{-\|x_i - x_j\|^2}{2\sigma^2}} & \text{if } j \in N(i) \\ 0 & \text{if } j \notin N(i) \end{cases}$$

Like LLE we know use W to find points in the d-dimensional parameter space (d is set or estimated in advance) that preserve the relations of their original counterparts.

The cost function we are aiming to minimize now is based on the fact that we want to maintain the distance between points:

$$\sum_{ij} W_{ij} \|y_i - y_j\|^2 = tr(Y^T L Y)$$

To avoid getting to a solution with less than d dimensions, we enforce the constraint $Y^T D Y = I$. In this way we will ensure that the rank of Y is d. We now have to solve the generalized eigenvalue problem $L\boldsymbol{y} = \lambda D\boldsymbol{y}$ and populate Y with the eigenvectors that correspond to the d smallest, non-zero eigenvalues.

Figure 3.7: *Laplacian Eigenmaps [Cay05].*

Laplacian Eigenmaps
input: $x_1 \dots x_n \in \mathbb{R}^D$, d, k, σ.

1. Set $W_{ij} = \begin{cases} e^{-\|x_i - x_j\|^2/2\sigma^2} & \text{if } x_j \in N(i) \\ 0 & \text{otherwise.} \end{cases}$
2. Let U be the matrix whose columns are the eigenvectors of $L\mathbf{y} = \lambda D\mathbf{y}$ with nonzero accompanying eigenvalues.
3. Return $Y := [U]_{n \times d}$.

3.3 Discriminative manifold learning algorithms in ASR

The reason why we presented some basic ideas and algorithms coming from the area of manifold learning is the fact that speech sounds are suspected to lie on a low-dimensional manifold, which we can exploit to reduce the dimensionality of speech feature vectors. Approaches to automatic speech recognition using manifold learning is an active field of research that has so far produced promising results.

3.3.1 Manifolds and speech data

We had mentioned earlier that natural data is usually high dimensional, yet the latent factors that generate it may lie in lower-dimensional spaces. Intuitively one can see why the manifold assumption is true for certain classes of speech: given that there is only a small finite number of articulatory systems involved (tracheia, glottis, vocal tract, tongue, lips, teeth), it is reasonable to expect the existence of a low-dimensional structure in speech data.

In their work, Jansen and Niyogi ([JN05]) consider wave propagation in acoustic tube models and present a derivation of the sounds generated by it. They represent steady-state sounds produced by such models with the magnitude of their Fourier coefficients and thus, as a point in a high dimensional space. They prove that certain classes of sounds generated, as the configuration of the articulatory system varies, lie on a low dimensional manifold embedded in the ambient high-dimensional space.

We will outline the analysis in their paper ([JN05]) when a single tube is modeling the vocal tract.

In order to be able to arrive at a tractable problem they introduce the following

approximations: the vocal tract walls are rigid, their impedance is much greater than that of air and the transverse dimension of the vocal tract is much smaller than the signal's wavelength. Based on these assumptions the problem of acoustically analysing the vocal tract resonator comes down to the analysis of planar waves in one spatial dimension.

The notation used is:

- p is the air pressure
- ρ is the density
- u is the velocity
- $A(x)$ is the cross-section area of the tube as a function of the position
- $U = Au$ is the volume velocity of the air perturbations

Starting from the continuity equation for compressible fluid flow and the preservation of momentum, we can arrive at a partial differential equation with respect to the volume velocity U as a function of space and time, which can be further reduced to the standard wave equation in free space:

$$\frac{\partial^2 U}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 U}{\partial t^2}$$

or, using pressure p

$$\frac{\partial^2 p}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2}$$

with the boundary conditions

$$U(0,t) = s(t)$$

where $s(t)$ represents the glottal vibrations as a function of time, and

$$\hat{U}(L,\omega) = \frac{\hat{p}(L,\omega)}{\boldsymbol{Z}_r(\omega)}$$

where the second condition is in the frequency domain and $\boldsymbol{Z}_r(\omega)$ is the acoustic impedance at the open end of the tube ($x = L$).

The solution $U(x,t)$ to the above equation can be expressed using its Fourier transform $\hat{U}(x,\omega)$:

$$U(x,t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{U}(x,\omega)e^{i\omega t}d\omega$$

Substituting in the wave equation and solving we get the general solution for the volume and (after working in a similar way) pressure respectively:

$$\hat{U}(x,\omega) = U_1(\omega)e^{-ikx} + U_2(\omega)e^{ikx}$$

$$\hat{p}(x,\omega) = p_1(\omega)e^{-ikx} + p_2(\omega)e^{ikx}$$

The detailed computations for the magnitudes can be found in [JN05]. Eventually we arrive at the general solutions for U and p:

$$\hat{U}(x,\omega) = \hat{s}(\omega)\left[\frac{e^{-ikx}}{1+\frac{B+1}{B-1}e^{-i2kL}} + \frac{e^{ikx}}{1+\frac{B-1}{B+1}e^{i2kL}}\right]$$

$$\hat{p}(x,\omega) = \hat{s}(\omega)\boldsymbol{Z}_r(\omega)\left[\frac{e^{-ikx}}{1+\frac{B+1}{B-1}e^{-i2kL}} + \frac{e^{ikx}}{1+\frac{B-1}{B+1}e^{i2kL}}\right]$$

where $B = \frac{\rho_0 c}{A\boldsymbol{Z}_r(\omega)}$, ρ_0 is the equilibrium density air value.

Since any odd periodic function can be expressed as a Fourier series of sinusoidal sources, we pick a single-frequency sinusoidal as the source function:

$$s(t) = U_0 sin\omega_0 t$$

If we compute its Fourier transform, substitute in the previous solutions and take the inverse Fourier, we arrive at the final form of the solution in the time-domain:

$$p(L,t) = \frac{U_0}{2i}\Big[f(\omega_0,L,A)e^{i\omega_0 t} - f(-\omega_0,L,A)e^{-i\omega_0 t} = Im(U_0 f(\omega_0,L,A)e^{i\omega_0 t})$$

where $f(\omega,L,A) = \boldsymbol{Z}_r(\omega)\Big[coskL - i\frac{A\boldsymbol{Z}_r(\omega)}{\rho_0 c}sinkL\Big]^{-1}$ and we arrived at the second form given that $\boldsymbol{Z}_r(\omega) = \boldsymbol{Z}_r^*(-\omega)$ and consequently $f^*(\omega,L,A) = f(-\omega,L,A)$.

If we use a linear combination of H harmonics as the source function

$$s(t) = \sum_{n=1}^{H} a_n sin(n\omega_0 t)$$

then at the output we expect a solution of the form

$$p(L,t) = \sum_{n=1}^{H} \beta_n sin(n\omega_0 t + \phi_n)$$

where for each n

$$\beta_n sin(n\omega_0 t + \phi_n) = Im(\alpha_n f(n\omega_0,L,A)e^{in\omega_0 t})$$

Consequently, the output Fourier coefficients β_n are given by

$$\beta_n(L,A) = \alpha_n |f(n\omega_0,L,A)|$$

If we now define a subset of $\mathcal{R}^H$ for a given set $\alpha_i | i = 1,...,H]$ by

$$\mathcal{M}_1(L_1,L_2) = (\beta_1,\beta_2,...,\beta_H) | L \in (L_1,L_2)$$

then $\mathcal{M}$ traces-out a one-dimensional curve in the ambient Fourier space $\mathcal{R}^H$ with the following properties:

- There exists a diffeomorphism $\phi : (L_1, L_2) \in \mathcal{R}^1 \rightarrow \mathcal{M}_1(L_1, L_2) \in \mathcal{R}^H$ for L_1, L_2 in the range of human vocal tract lengths.

- The diffeomorphism $\phi^{-1} : \mathcal{M}_1(L_1, L_2) \rightarrow \mathcal{R}^1$ is a coordinate chart on the set $\mathcal{M}_1(L_1, L_2)$.

- The set $\mathcal{M}_1(L_1, L_2)$ is open.

These properties tell us that the set $\mathcal{M}_1(L_1, L_2)$ is a smooth and open one dimensional manifold embedded in $\mathcal{R}^H$, which is not a straight line and spans the whole ambient space.

The manifold structure of data can be exploited in several ways: improve supervised learning techniques by applying regularization based on the manifold of speech data (*manifold regularization*) and find new methods of dimensionality reduction. One such method will be presented next.

3.3.2 Discriminative manifold learning

[TR14a][TR13a]

In the context of automatic speech recognition we usually want to classify our data into the corresponding phoneme categories, e.g. biphones or triphones. It has recently been established ([SR03]) that the geometric and local structure of the space the data lie on, is important for classification tasks. Consequently, if we could identify the underlying manifold of the available data and apply a discriminative feature transformation, we should see an improvement in our classification efforts.

However, discriminative transformations alone are incapable of capturing the geometric and local distributional structure of the data space, whereas on the other hand, manifold learning algorithms that do uncover the geometric properties of the space, are unsupervised and non-discriminative. Rose and Tomar propose a new framework ([TR14a]) that introduces a discriminative element in manifold learning techniques: they aim to maximize the separability between different classes while at the same time preserving the manifold-constraint relationships between data points belonging to the same class.

Their discriminative manifold learning framework starts -similarly to the already presented techniques- by embedding feature vectors into two high-dimensional graphs connecting neighborhoods of vectors, followed by an optimization of their structure according to certain constraints. These weighted, undirected graphs characterize the class-specific manifolds the feature vectors lie on. One of them, called the *intrinsic* graph, defines the relationships between same-class feature vectors

($\mathcal{G}_{int} = \{\boldsymbol{X}, \boldsymbol{W}_{int}\}$), whereas the other, called *penalty* graph, defines the relationships between feature vectors belonging to different classes ($\mathcal{G}_{pen} = \{\boldsymbol{X}, \boldsymbol{W}_{pen}\}$).

The characteristics of the two graphs, such as structure, connectivity and compactness, are mostly influenced by the weights on their edges, which are calculated according to the distance between points. Therefore, intuitively we understand that the graphs express the geometry of the data space.

Based on the metrics used to calculate distance, the writers propose two different approaches, both resulting in the estimation of a projection matrix $\mathcal{P} \in R^{d \times m}, m \ll d$, which maximizes the sub-manifold class discrimination in the projected space while at the same time it preserves the within-sub-manifold intrinsic data relations.

Locality Preserving Discriminant Analysis

[TR12a]

In Locality Preserving Discriminant Analysis (*LPDA*) the distance metric used is the Euclidean distance. The elements of the weight matrices $\boldsymbol{W}_{int}$ and $\boldsymbol{W}_{pen}$ are calculated using the Gaussian heat kernel:

$$w_{ij}^{int} = \begin{cases} e^{\frac{-\|x_i - x_j\|^2}{\rho}} & \text{if } C(x_i) = C(x_j),\ e(x_i, x_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$w_{ij}^{pen} = \begin{cases} e^{\frac{-\|x_i - x_j\|^2}{\rho}} & \text{if } C(x_i) \neq C(x_j),\ e(x_i, x_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

where ρ is the heat kernel scale parameter, $C(x_i)$ refers to the class of vector x_i and $e(x_i, x_j) = 1$ indicates that x_i is in the near neighborhood of x_j.The neighbors of vector x_i can be discovered either by an exact or by an approximate k-nearest neighbors method, such as a range search for points lying in a ball of radius r around x_i.

In essence, the meaning of the aforementioned weights assignment is that in graph $\mathcal{G}_{int}$ a node x_i is connected to its k_{int}-nearest neighbors having the same label, whereas in $\mathcal{G}_{pen}$ a node x_i is connected to its k_{pen}-nearest neighbors that have a different label than itself. The values of k_{int} and k_{pen} are determined empirically.

We can define a scatter measure for a graph G as follows:

$$F_G(\boldsymbol{P}) = \sum_{i \neq j} \|y_i - y_j\|^2 w_{ij} = 2\boldsymbol{P}^{\boldsymbol{T}} \boldsymbol{X}(\boldsymbol{D} - \boldsymbol{W})\boldsymbol{X}^{\boldsymbol{T}} \boldsymbol{P}$$

where $\boldsymbol{D}$ is a diagonal matrix defined as $D_{ii} = \sum_j w_{ij}$. Depending on whether we want to retain or discard graph properties, the optimal projection matrix $\boldsymbol{P}$ can be obtained by minimizing or maximizing F_G.

Specifically for LPDA, one would want to reinforce the fact that points belonging to different classes are conceptually far away from each other and on the other hand strengthen the relations between points of the same class. Intuitively it follows that one should aim to maximize the scatter of the penalty graph $F_{pen}(\boldsymbol{P})$ and minimize the scatter of the intrinsic graph $F_{int}(\boldsymbol{P})$. To combine the two scatters into a single measure the writers define their ratio as a measure of class-separability and graph preservation:

$$F(\boldsymbol{P}) = \frac{F_p(\boldsymbol{P})}{F_i(\boldsymbol{P})} = \frac{\boldsymbol{P}^T\boldsymbol{X}(\boldsymbol{D}_p - \boldsymbol{W}_p)\boldsymbol{X}^T\boldsymbol{P}}{\boldsymbol{P}^T\boldsymbol{X}(\boldsymbol{D}_i - \boldsymbol{W}_i)\boldsymbol{X}^T\boldsymbol{P}}$$

where i denotes *intrinsic* and p *penalty.*

The optimal $\boldsymbol{P}$ is the one to maximize $F(\boldsymbol{P})$:

$$\boldsymbol{P}_{LPDA} = \underset{\boldsymbol{P}}{\operatorname{argmax}}\, F(\boldsymbol{P}) = \underset{\boldsymbol{P}}{\operatorname{argmax}} \left\{ tr\Big((\boldsymbol{X}(\boldsymbol{D}_i - \boldsymbol{W}_i)\boldsymbol{X}^T\boldsymbol{P})^{-1}(\boldsymbol{P}^T\boldsymbol{X}(\boldsymbol{D}_p - \boldsymbol{W}_p)\boldsymbol{X}^T\boldsymbol{P}) \Big) \right\}$$

This maximization problem can be brought down to solving the following generalized eigenvalue problem:

$$(\boldsymbol{X}(\boldsymbol{D}_p - \boldsymbol{W}_p)\boldsymbol{X}^T)\boldsymbol{p}^j_{lpda} = \lambda_j(\boldsymbol{X}(\boldsymbol{D}_i - \boldsymbol{W}_i)\boldsymbol{X}^T)\boldsymbol{p}^j_{lpda}$$

where $\boldsymbol{p}^j_{lpda}$ is the j^{th} column of the transformation matrix $\boldsymbol{P}_{lpda} \in R^{d\times m}$ and is the eigenvector associated with the j_{th} largest eigenvalue. Therefore $\boldsymbol{P}^{optimal}_{lpda}$ contains the m eigenvectors associated with the m largest eigenvalues.

Correlation Preserving Discriminant Analysis

[TR13b][TR12b]

The second approach, Correlation Preserving Discriminant Analysis (CPDA), uses the cosine-correlation, i.e. normalized inner-product, as distance metric. This is motivated by the fact that models based on Euclidean distance are susceptible to noise, whereas cosine-correlation based models are expected to be less influenced by noise, given the robustness of the angles between cepstrum vectors.

Similarly to LPDA, CPDA uses two undirected graphs , ($\mathcal{G}_{int} = \{\boldsymbol{X}, \boldsymbol{W}_{int}\}$), ($\mathcal{G}_{pen} = \{\boldsymbol{X}, \boldsymbol{W}_{pen}\}$), to embed the feature vectors, yet this time the similarity between nodes is calculated based on the cosine-correlation distance:

$$w^{int}_{ij} = \begin{cases} e^{\frac{\langle x_i, x_j\rangle - 1}{\rho}} & \text{if } C(x_i) = C(x_j),\ e(x_i, x_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$w^{pen}_{ij} = \begin{cases} e^{\frac{\langle x_i, x_j\rangle - 1}{\rho}} & \text{if } C(x_i) \neq C(x_j),\ e(x_i, x_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

where the notation is the same as above. The scatter measure used in CPDA is the same as in LDPA, yet CPDA first projects the data points onto the surface of a unit hypersphere, thus discarding magnitude information and keeping correlation-based information about data points. Thus, the projection matrix $\mathcal{P} \in R^{d \times m}, m \ll d$, eventually estimated, will non-linearly project the features from the original d-dimensional hypersphere onto the target m-dimensional hypersphere.

The scatter for CPDA is:

$$F_G(\boldsymbol{P}) = \sum_{i \neq j} \|y_i - y_j\|^2 w_{ij} \overset{\boldsymbol{y}=\boldsymbol{P}^T\boldsymbol{x}/\|\boldsymbol{P}^T\boldsymbol{x}\|}{=} \sum_{i \neq j} \left\| \frac{\boldsymbol{P}^T\boldsymbol{x}_i}{\|\boldsymbol{P}^T\boldsymbol{x}_i\|} - \frac{\boldsymbol{P}^T\boldsymbol{x}_j}{\|\boldsymbol{P}^T\boldsymbol{x}_j\|} \right\| w_{ij} = 2 \sum_{i \neq j} \Big(1 - \frac{f_{ij}}{f_i f_j}\Big) w_{ij}$$

where for two arbitrary vectors x_u, x_v:

$$f_u = \sqrt{x_u^T \boldsymbol{P}\boldsymbol{P}^T x_u}$$

and

$$f_{uv} = \sqrt{x_u^T \boldsymbol{P}\boldsymbol{P}^T x_v}$$

As previously, we want to maximize the scatter of the penalty graph and minimize the scatter of the intrinsic graph. However, in CPDA the writers have chosen the difference of the two scatter measures as a measure of manifold separability and graph preservation:

$$F(\boldsymbol{P}) = F_{pen}(\boldsymbol{P}) - F_{int}(\boldsymbol{P}) = 2 \sum_{i \neq j} (1 - \frac{f_{ij}}{f_i f_j}) w_{ij}^{p-i}$$

where $w_{ij}^{p-i} = w_{ij}^{pen} - w_{ij}^{int}$. The optimal projection matrix $\boldsymbol{P}_{cpda}^{opt}$ is obtained by maximizing $F(\boldsymbol{P})$:

$$\boldsymbol{P}_{CPDA} = \underset{\boldsymbol{P}}{\operatorname{argmax}}\, F(\boldsymbol{P})$$

Because of the projection of the feature vectors onto the unit hypersphere, the maximization problem cannot be solved by solving an equivalent generalized eigenvalue problem. Therefore, an iterative procedure is chosen, the well known gradient ascent:

$$P := P + \alpha \nabla_P F$$

where

$$\nabla_P F = 2 \sum_{i \neq j} \Big[\frac{f_{ij} x_i x_i^T}{f_i^3 f_j} + \frac{f_{ij} x_j x_j^T}{f_i f_j^3} - \frac{x_i x_j^T + x_j x_i^T}{f_i f_j} \Big] \boldsymbol{P} w_{ij}^{p-i}$$

where α is the gradient scaling factor and $\nabla_P F$ represents the gradient of the scatter measure F with respect to $\boldsymbol{P}$. Given that F is non-linear and non-convex, a good initialization is important to avoid reaching just a local maximum. Such an

initialization can be obtained by neglecting the projection on the unit hypersphere, approximate a linear transformation:

$$\boldsymbol{y}_i = \boldsymbol{P}^T x_i$$

and eventually acquire $\boldsymbol{P}_{init}$ by solving the generalized eigenvalue problem:

$$(\boldsymbol{X}(\boldsymbol{D}_p - \boldsymbol{W}_p)\boldsymbol{X}^T)\boldsymbol{p}^j = \lambda_j(\boldsymbol{X}(\boldsymbol{D}_i - \boldsymbol{W}_i)\boldsymbol{X}^T)\boldsymbol{p}^j$$

where $\boldsymbol{D}$ is a diagonal matrix with elements $D_{ii} = \sum_i w_{ij}$ and $\boldsymbol{X}$ contains the normalized feature vectors. As before, $\boldsymbol{P}_{cpda}^{init}$ is composed of the eigenvectors corresponding to the m largest eigenvalues.

Noise Aware Manifold Learning

[TR13c]

Furthermore, in their work Rose and Tomar explore a new method to increase noise robustness of manifold learning methods. They point out that since edge weights, which depend on the Gaussian heat kernel size and shape, characterize local neighborhoods, there is a strong relation between kernel size and the robustness to background noise.

The kernel size governs the compactness of the neighborhood and the smoothness of the manifolds, and in turn, it is governed by the scale parameter ρ. Therefore, the choice of ρ is important to the definition of local neighborhoods and thus to the characteristics of the transformation. If ρ had a value that is too large, then the kernel would tend to flatten, and all data pairs on the graph would have the same importance; on the other hand, if it had a too small value, the manifold would lack smoothing and thus the kernel would be too sensitive to noise. Therefore, the writers claim that the optimal value for ρ is dependent on the SNR level of the speech signal, and support their claims with successful experimental results.

They propose that multiple scale parameters be used, each specific to a noise condition. First, during training, they gather multiple SNR dependent LPDA or CPDA projections and Continuous Density HMMs. This procedure is broken down into three steps:

- Based on a set of ρ values, determine the set of the corresponding projection matrices
- Heuristically identify the optimal value of rho and the corresponding transformation that maximizes ASR performance for a given SNR level

- Train CDHMM models from the features obtained using the set of projection matrices previously acquired

Having this set of SNR-dependent LPDA/CPDA transformations, SNR is estimated for each test utterance and the feature space projection is performed using the corresponding LPDA/CPDA projection matrix. Eventually, the corresponding CDHMM model is used in the ASR application.

This procedure is referred to as Noise-Aware LPDA/CPDA (*N-LPDA* or *N-CPDA*).

Results

All of the proposed techniques are verified by experiments the writers conducted using the Aurora-2 and Aurora-4 datasets.

LPDA and CPDA show a relative WER improvement ranging from 6 to 30% compared to conventional techniques, such as Linear Discriminant Analysis or Locality Preserving Projections. This verifies the assertion that manifold and discriminative learning result in a well-behaved and robust feature-space transformation.

Furthermore, CPDA performs better that LPDA under high noise conditions, supporting the noise robustness of cosine-correlation distance compared to Euclidean. As far as Noise-Aware manifold learning is concerned, N-LPDA performs slightly better on average as compared to LPDA for most noise conditions (0.10-0.20 reduced WER).

At this point it is important to point out the huge computational complexity involved in computing the weight matrices, given the high dimensionality of the feature space. To tackle this issue the writers propose using novel approximate nearest-neighbors techniques such as Locality Sensitive Hashing ([TR13b]).

Chapter 4

Deep Neural Networks

In this chapter we will give a general introduction to deep neural networks (*DNNs*). We will begin by presenting the ideas and motivation behind deep architectures and then move on to describe a few of these models as well as practical issues concerning building and training such models.

4.1 History and Motivation

[DY14]

Until recently, shallow-structured architectures were prevalent in machine learning and signal processing. These include Gaussian Mixture Models, Conditional Random Fields, Support Vector Machines, logistic regression and the well-known multilayer perceptron, all of which have been successful in solving well-constrained problems, yet their limited representational power does not allow them to effectively deal with applications involving natural signals such as human speech, natural sound and image.

In order to deal with such tasks efficiently, researchers had to improve the information processing mechanisms involved in the systems and they turned to the human brain for inspiration. Human information processing mechanisms (such as vision and audition) suggest the need for "deep", hierarchical architectures ([SKK+07]) to capture non-linear, complex structure and build internal distributed representations from sensory inputs. Researchers however, have been stressing out recently that deep learning, i.e. the application of deep architectures to extract information from data, is only *inspired* by the way the brain works, rather than mimicking it. For one thing, the human brain is full of different types of neurons each one serving a different purpose, whereas deep architectures have a single type of neuron performing the same task. Moreover, humans learn mostly in an unsupervised manner, contrary to deep models which usually need labeled examples to learn from.

Although deep architectures recently gained in popularity, the main idea behind them, that is to learn lower-level representations from raw input data, is not new. As early as 1998, Yann LeCun et al. ([LBBH98]) developed a Convolution Neural Network to recognize handwritten digits on banking checks. The novelty of LeNet-5 (the model proposed in the paper) was that it learned features from raw pixel images, as opposed to receiving hand-crafted features in the input end. The power of deep models lies exactly at that point: they can extract features from raw input and do discrimination jointly ([Ben09]).

However, at that time, it was not easy to implement a deep architecture onto , for example, a feed-forward neural network. The back-propagation algorithm, which had been very popular until then, did not work well in practice when more than a small number of hidden layers were involved. Given the presence of too many local optima in the non-convex objective function of such models, as well as the problems of diminishing or exploding gradients and the computational issues arising when the parameters' number rose too much, most research turned to shallow models with convex loss functions (e.g. SVMs, CRFs). It was not until recently that new theoretical breakthroughs appeared (e.g. RBM pretraining, primal-dual update algorithm, new non-linear activation functions) and together with the boom of computationally efficient hardware usage (GPUs, CPU/GPU clusters) allowed researchers to turn to deep (many hidden layers) and wide (many neurons) models.

4.1.1 Deep learning in Automatic Speech Recognition

Neural networks had appeared in the ASR field as early as 1980 with the Time Delay Neural Network ([WHH+90]), which introduced time-delay units, in addition to sigmoids, in order to take into account past history together with the current input. Hybrid approaches of (shallow) NNs with Hidden Markov Models were also introduced early ([MB90]) however at the time they were abandoned due to worse performance in comparison with GMM/HMM systems.

The interest in neural networks for ASR was refueled again in 2009 at the NIPS Workshop on Deep Learning, where a team of researchers from the University of Toronto proposed a DNN/HMM system for phone recognition ([rMDH]). The combination of the DNN with the HMM was the same as the first hybrid approach in 1990, yet now a deep model was used instead of a shallow and the team managed to achieve a 23.0% phone error rate on TIMIT as opposed to 27.7% of the GMM-HMM system.

Around the same time researchers started looking into raw features instead of MFCCs and deep autoencoders were tried for binary feature encoding and feature extraction. The promising results that these attempts yielded, turned the spotlight

on DNNs for ASR, and further successful experiments on large vocabulary tasks followed. The work presented in [SLY] by Seide et al. exhibited a 1/3 cut of error over the corresponding GMM/HMM system. Most of the ideas tried were not actually new; however, due to limited computing resources and training data it was difficult if not impossible before to train the large models that were tried at the time.

The experiments at that time started clarifying the conditions under which a context-dependent deep neural network together with an HMM system can be successfully applied to ASR:

- having a deep architecture
- using tied triphone states as targets (*senones*)
- use a contextual window of features as input

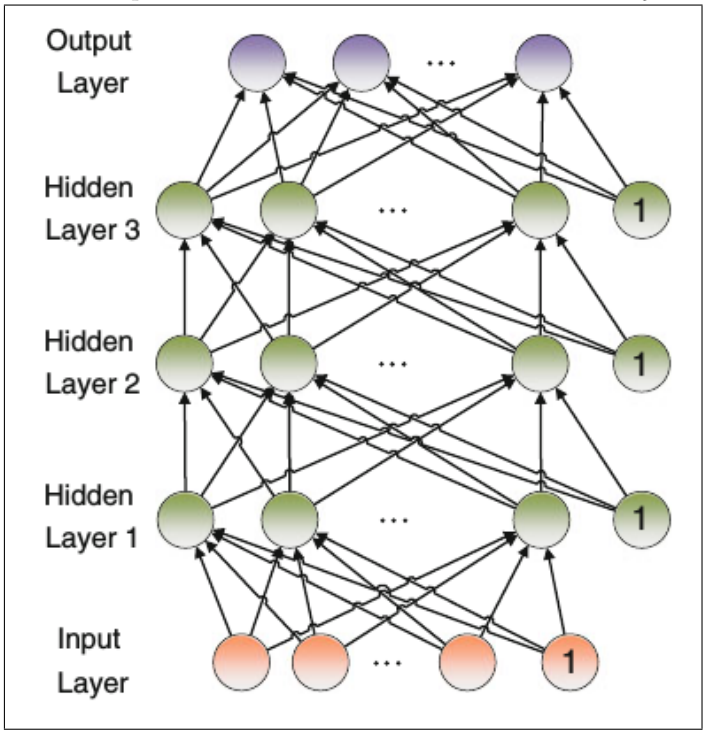
Furthermore, it was found that initializing layer weights through pre-training might help, but it is not crucial for good performance.

In parallel with research on deep models used in acoustic modeling, there was work going on on the features fed into the models. It was found that DNNs did not need some of the feature pre-processing steps necessary for HMM/GMM systems (e.g. HLDA, fMLLR). This was attributed first to the many layers in the architecture, which applied a nonlinear feature transformation on the input, second to the softmax layer which acts as a log-linear classifier and third to the joint optimization of the two. Furthermore, DNNs can handle correlated input and hence, correlated features that without pre-processing could not be used for GMM/HMM training can now be exploited in a DNN system without any previous processing.

Recent work on feature learning in the context of deep neural networks which was presented in 2013 ([SKrMR13]) showed that DNNs can extract the Mel-scale filterbank from the FFT spectrum given as input. Another team demonstrated that even raw time signal can be used as input to the DNN for acoustic modeling ([TGSN14]). Finally, in 2014 a team from IBM combined CNN,DNN and i-vector based adaptation techniques to improve the acoustic model and managed to reduce the WER on the Switchboard Hub5'00 dataset to 10.4% (as opposed to 14.5% achieved by the best GMM/HMM system). This error rate can be further reduced to less than 10% by improving the language model, either with recent neural-network approaches or with large-scale N-grams.

However, state-of-the-art systems with or without DNNs still perform poorly when faced with tasks involving far field microphones, very noisy conditions, speech with accent or multiple speakers and spontaneous, interrupted speech. To deal

Figure 4.1: *A deep neural network with three hidden layers [YD14].*



with such tasks research has focused on improving acoustic models and developing dynamic ASR systems with recurrent feedback, as well as incorporating semantic understanding which will assist in pruning the search space of sentences.

4.2 Constructing DNNs: the Multilayer Perceptron

[YD14]

Although initially the term "deep neural network" was used to identify a multilayer perceptron with many hidden layers, it has now come to name a whole class of neural network models having a "deep" architecture, i.e. having at least three hidden layers. However, we will use the conventional multilayer perceptron to present the basic deep architecture and cover the training issues associated with it.

4.2.1 Architecture of the Deep Multilayer Perceptron

As already mentioned, the simplest deep neural network is a multilayer perceptron with at least three hidden layers, that is, excluding the input and output layers. A DNN with a total of five layers can be seen in figure 4.1

We can denote the input layer as layer 0 and the output layer as layer L for an $L+1$-layer DNN.

The neurons in the first layer correspond to each feature (or dimension) in the input vector, i.e.

$$\boldsymbol{v}^0 = \boldsymbol{x}_{input}$$

whereas for the first L layers the output is calculated as follows:

$$\boldsymbol{v}^l = f(\boldsymbol{z}^l) = f\Big(\boldsymbol{W}^l \boldsymbol{v}^{l-1} + \boldsymbol{b}^l\Big),\ 0 < l < L$$

where

$$\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{v}^{l-1} + \boldsymbol{b}^l \in \mathcal{R}^{N_l \times 1}$$

$$\boldsymbol{v}^l \in \mathcal{R}^{N_l \times 1}$$

$$\boldsymbol{W}^l \in \mathcal{R}^{N_l \times N_{l-1}}$$

$$\boldsymbol{b}^l \in \mathcal{R}^{N_l \times 1}$$

and $N_l \in \mathcal{R}$, are respectively the excitation vector, the activation vector, the weight matrix, the bias vector and the number of neurons at layer l.

The function $f(\cdot) : \mathcal{R}^{N_l \times 1} \rightarrow \mathcal{R}^{N_l \times 1}$ is the activation function of the corresponding neuron that is applied element-wise to the excitation vector. The most common activation function is the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

followed by the second most common, the hyperbolic tangent

$$\tanh(z) = \frac{e^z + e^{-z}}{e^z + e^{-z}}$$

which is a rescaled version of the former. Both have the same modeling power, but the output range of the hyperbolic tangent is (-1,+1) , therefore the activation values are symmetric and this was believed to help during training. On the other hand the sigmoid function has an output range of (0,1), which stimulates sparse but asymmetric activation values. Recently, another type of activation function, the rectified linear unit (ReLU) has been suggested:

$$ReLu(z) = max(0, z)$$

ReLu has been gaining in popularity since it enforces sparse activation values and has a simple gradient:

$$\nabla_{ReLu}(z) = max(0, sgn(z))$$

As far as the output layer is concerned, its form depends on the task of interest.

Figure 4.2: *Computation of DNN output [YD14].*

```
1: procedure FORWARDCOMPUTATION(O)
                                          ▷ Each column of O is an observation vector
2:     V^0 ← O
3:     for ℓ ← 1; ℓ < L; ℓ ← ℓ + 1 do          ▷ L is the total number of layers
4:         Z^ℓ ← W^ℓ V^{ℓ−1} + B^ℓ              ▷ Each column of B^ℓ is b^ℓ
5:         V^ℓ ← f(Z^ℓ)          ▷ f(.) can be sigmoid, tanh, ReLU, or other functions
6:     end for
7:     Z^L ← W^L V^{L−1} + B^L
8:     if regression then                                  ▷ regression task
9:         V^L ← Z^L
10:    else                                              ▷ classification task
11:        V^L ← softmax(Z^L)                  ▷ Apply softmax column-wise
12:    end if
13:    Return V^L
14: end procedure
```

In case of regression tasks, a linear layer is used to generate the output vector $\boldsymbol{v}^L \in \mathcal{R}^{N_l}$, where N_l is the output dimension:

$$\boldsymbol{v}^L = \boldsymbol{z}^L = \boldsymbol{W}^L \boldsymbol{v}^{L-1} + \boldsymbol{b}^L$$

In case of classification tasks, each output neuron represents a class $i \in 1, ..., C$. where $C = N_L$ is the number of classes. In this case, the output of the DNN will be a probability distribution over the possible classes. The value of the i^{th} output neuron represents the probability $P_{dnn}(i|\boldsymbol{x}_i)$ that the input vector belongs to class i.

Since we want the output vector to be a valid probability distribution we impose the following constraints on it

$$v_i^L \geq 0$$

and

$$\sum_{i=1}^{C} v_i^L = 1$$

and use as output function the softmax function:

$$v_i^L = P_{dnn}(i|\boldsymbol{x}_i) = softmax(\boldsymbol{z}^L) = \frac{e^{z_i^L}}{\sum_{j=1}^{C} e^{z_j^L}}$$

where z_i^L is the i^{th} element of the excitation vector $\boldsymbol{z}^L$.

Given an input matrix with observations $\boldsymbol{O}$ the output of the DNN can be computed as described in figure 4.2

4.2.2 Training and related issues

Given that a multilayer perceptron with a sufficiently large hidden layer can approximate any function ([HSW89]), it is evident that a multilayer perceptron

with more than one hidden layers can serve as a universal approximator as well, that is, compute any mapping

$$g : \mathcal{R}^D \rightarrow \mathcal{R}^C$$

from the input space $\mathcal{R}^D$ to the output space $\mathcal{R}^C$. Given a set of training samples we can estimate the network parameters $\{\boldsymbol{W}, \boldsymbol{b}\}$ with a training process, characterized by a training criterion and a learning algorithm.

Training criterion

The training criterion is chosen so that an improvement in it reflects an improvement in the final evaluation score. Therefore, it should somehow express the final goal of the task, yet at the same time be easy to evaluate.

Ideally, we would like to estimate the DNN parameters that minimize the expected cost:

$$J_E = \mathrm{E}(J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}_i, \boldsymbol{y}))$$

where J is the cost (or loss) function given the model parameters $\{\boldsymbol{W}, \boldsymbol{b}\}$ together with the input vector $\boldsymbol{x}_i$ and the corresponding output $\boldsymbol{y}$. However, since we cannot know in advance the output for unseen samples, we can only optimize empirical criteria based on the training set.

The two most popular empirical criteria used in DNN training are the mean square error (MSE) for regression tasks and the cross-entropy (CE) for classification tasks, which are defined as follows:

$$J_{MSE}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}_{i,train}, \boldsymbol{y}_{o,train}) = \frac{1}{2M} \sum_{m=1}^{M} \|\boldsymbol{v}^L - \boldsymbol{y}\|^2$$

and

$$J_{CE}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}_{i,train}, \boldsymbol{y}_{o,train}) = \frac{1}{M} \sum_{m=1}^{M} (- \sum_{i=1}^{C} y_i \log(v_i^L))$$

where y_i and v_i in J_{CE} are respectively, the probability observed in the training set that $\boldsymbol{x}$ belongs to class i ($y_i = P_{empirical}(i|\boldsymbol{x})$) and the same probability estimated from the DNN $v_i^L = P_{DNN}(i|\boldsymbol{x})$).

The minimization of the cross-entropy criterion is equivalent to minimizing the the Kullback-Leibler divergence between the empirical probability distribution and the probability distribution estimated from the deep neural network. One crucial advantage of using cross-entropy as cost function against mean square error, is that it has better convergence properties than MSE and the network learns faster from its classification mistakes during training. This happens because the derivative of CE is proportional to the difference between DNN estimation and the actual target value,

Figure 4.3: *Back-propagation algorithm [YD14].*

procedure BACKPROPAGATION($\mathbb{S} = \{(\mathbf{o}^m, \mathbf{y}^m) \,|0 \leq m < M\}$)
▷ $\mathbb{S}$ is the training set with M samples
  Randomly initialize $\left\{\mathbf{W}_0^\ell, \mathbf{b}_0^\ell\right\}, 0 < \ell \leq L$ ▷ L is the total number of layers
  while Stopping Criterion Not Met **do**
    ▷ Stop if reached max iterations or the training criterion improvement is small
    Randomly select a minibatch $\mathbf{O}, \mathbf{Y}$ with M_b samples.
    Call ForwardComputation($\mathbf{O}$)
    $\mathbf{E}_t^L \leftarrow \mathbf{V}_t^L - \mathbf{Y}$ ▷ Each column of $\mathbf{E}_t^L$ is $\mathbf{e}_t^L$
    $\mathbf{G}_t^L \leftarrow \mathbf{E}_t^L$
    for $\ell \leftarrow L; \ell > 0; \ell \leftarrow \ell - 1$ **do**
      $\nabla_{\mathbf{W}_t^\ell} \leftarrow \mathbf{G}_t^\ell \left(\mathbf{v}_t^{\ell-1}\right)^{\mathrm{T}}$
      $\nabla_{\mathbf{b}_t^\ell} \leftarrow \mathbf{G}_t^\ell$
      $\mathbf{W}_{t+1}^\ell \leftarrow \mathbf{W}_t^\ell - \frac{\varepsilon}{M_b} \nabla \mathbf{W}_t^\ell$ ▷ Update $\mathbf{W}$
      $\mathbf{b}_{t+1}^\ell \leftarrow \mathbf{b}_t^\ell - \frac{\varepsilon}{M_b} \nabla \mathbf{b}_t^\ell$ ▷ Update $\mathbf{b}$
      $\mathbf{E}_t^{\ell-1} \leftarrow \left(\mathbf{W}_t^\ell\right)^{\mathrm{T}} \mathbf{G}_t^\ell$ ▷ Error backpropagation
      if $\ell > 1$ **then**
        $\mathbf{G}_t^{\ell-1} \leftarrow f'\left(\mathbf{Z}_t^{\ell-1}\right) \bullet \mathbf{E}_t^{\ell-1}$
      end if
    end for
  end while
  Return $dnn = \left\{\mathbf{W}^\ell, \mathbf{b}^\ell\right\}, 0 < \ell \leq L$
end procedure

and therefore the change in the network parameters is greater when the estimated value diverges a great deal from the target. **JUSTIFIED FOR MULTINOMIAL CASE??**

Training algorithm and practical considerations

Having selected an appropriate training criterion, we can train the network using the well known back-propagation algorithm which is based on the chain rule for gradient computation. The algorithm is summarized in figure 4.3.

Despite the simplicity of the algorithm, its application to DNNs gives rise to many problems, the most important ones being the speed of convergence and the vanishing or exploding gradients that get in the way of the training. It is therefore crucial to address these issues by specifically tuning the DNN and the training set to avoid such drawbacks.

Data preprocessing Data preprocessing is an important part of the pre-training phase in the DNN construction. There are two principal ways of preprocessing each one attacking a different problem that might arise during training.

The first preprocessing technique is referred to as per-sample feature normalization and it results in reducing the variability in the final feature presented to the DNN. This is desired since we want to drop any variation that is not important to the final decision made by the model and would otherwise add extra difficulties both at the output stage as well as during training, as the network would try to learn

too much redundant information. For example, in image recognition, it is desired to reduce the variability introduced by the brightness, or in speech recognition we want to avoid acoustic channel distortions. In this last case we can achieve our goal by calculating the per-utterance mean for each feature and subtracting it from all frames in the utterance (cepstral mean normalization).

The second preprocessing technique is the global feature standardization and it aims to scale the data along each dimension with a global transformation so that the final data vectors lie in a similar range of numerical values. The global transformation is estimated from the training data and is then applied to both training and test sets. Global feature standardization will improve performance in later steps of the training process. As far as DNN training is concerned, it allows us to use the same learning rate across all weight dimensions and still achieve good performance.

A common global transformation in speech recognition is to standardize each dimension of the feature vector to have zero mean and unit variance. If we use MFCC vectors without standardization, the energy component will dominate the learning procedure since its values are much greater than the rest of the coefficients.

Initialization of model parameters The back-propagation algorithm starts given a set of initial network parameters. Since the DNN is highly non-linear and the training criterion with respect to the model parameters is non-convex, initialization of the model parameters can greatly affect the end model. There are many suggestions for the initialization of a DNN model, all of which are based on two assumptions.

First, the weights should be initialized so that each neuron operates in the linear range of the activation function (e.g. sigmoid) at the start of the learning. If the weights were very large, neurons would saturate, i.e. take an extreme value (close to zero or one in the case of a sigmoid function) leading to vanishing gradients. This becomes evident in the case of the sigmoid as we can observe from its derivative:

$$\sigma'(\boldsymbol{z}_t^l) = (1 - \sigma(\boldsymbol{z}_t^l))\sigma(\boldsymbol{z}_t^l) = (1 - \boldsymbol{v}_t^l)\boldsymbol{v}_t^l$$

On the contrary, if the neurons operate in the linear range, the gradients are large enough and learning can proceed smoothly. It is important to point out that the excitation vector depends on both the weights and the input values. If we have standardized the input values in the preprocessing stage, then the weight initialization can become easier.

Second, the weights should be initialized at random. Since neurons in DNNs are symmetric and interchangeable, random initialization serves the purpose of symmetry breaking. If all neurons were initialized with the same values then they would have the same output thus detect the same patterns in the lower layers.

Bengio et. al. ([GB10]) have suggested that the weight values should be uniformly sampled from an interval depending on the activation function. For the sigmoid function they suggested the interval

$$\Big[-4\sqrt{\frac{6}{fan_{in}+fan_{out}}}, 4\sqrt{\frac{6}{fan_{in}+fan_{out}}}\Big]$$

where fan_{in} is the number of units in the $(i-1)^{th}$ layer and fan_{out} is the number of units in the i^{th} layer. This approach is based on the idea that neurons with more inputs should have smaller weights ([Ben12]).

LeCun et. al. ([LBOM98]) suggested to draw the initial weight values for layer l from a zero-mean Gaussian distribution with standard deviation $\sigma_{\boldsymbol{W}^{l+1}} = \frac{1}{\sqrt{N_l}}$, where N_l is the number of connections feeding into the node.

When it comes to speech recognition applications, usually each DNN layer has 1000-2000 neurons and initializing the weights either from a Gaussian distribution $\mathcal{N}(w; 0, 0.05)$ or from a uniform distribution in the range of [-0.05,0.05] we can achieve good results.

As far as the bias vectors $\boldsymbol{b}^l$ are concerned, they can be initialized to zero.

Regularization Taking into consideration the huge number of parameters in a DNN it is easy to understand the risk of overfitting the model to the training data. Overfitting occurs because although we aim to minimize the expected loss, we actually minimize the empirical loss defined on the training set. To control overfitting we try to affect the model parameters through the training function so that they do not fit the training data too well. This process is known as regularization or weight decay. Regularization aims to help the model learn patterns often seen in the training data yet not learn every noise peculiarity present in the training set ([Nie15]).

The most common forms of regularization are based on the L_1 and L_2 norms respectively:

$$R_1(\boldsymbol{W}) = \|vec(\boldsymbol{W})\|_1 = \sum_{l=1}^{L} \|vec(\boldsymbol{W}^l)\|_1$$

$$R_2(\boldsymbol{W}) = \|vec(\boldsymbol{W})\|_2^2 = \sum_{l=1}^{L} \|vec(\boldsymbol{W}^l)\|_2^2$$

Including the regularization terms the training criterion becomes

$$J_R(\boldsymbol{W}, \boldsymbol{b}; \mathcal{S}_{train}) = J(\boldsymbol{W}, \boldsymbol{b}; \mathcal{S}_{train}) + \lambda R(\boldsymbol{W})$$

where λ is an interpolation weight called regularization weight. Regularization is particularly helpful when the training set size is small compared to the number of parameters of the DNN. For DNNs used in speech recognition, there are usually

millions of parameters in the model and the regularization weight should be small ($\approx 10^{-4}$) or zero when the training set is large.

In Bayesian context, the regularization term is the negative log-prior $-logP(\theta)$ on the parameters θ. Adding the regularization term makes the training criterion correspond to the negative joint likelihood of training data and parameters

$$-logP(data, \theta) = -logP(data|\theta) - logP(\theta)$$

with the loss function $L(z, \theta)$ being interpreted as $-logP(z|\theta)$ and $-logP(data|\theta) = -\sum_{t=1}^{T} L(z_t, \theta)$, where $z_t,\ t = 1, ..., T$ are the training data. In case we are training our model with a stochastic gradient based method, as is usually the case, we want to use an unbiased estimation of the gradient of the total training criterion including both the loss function and the regularization term. In a mini-batch or online learning approach it is not trivial to include the regularization in the above sum, since it is different from the sum of the loss function on all examples and the regularization term. In these cases, the regularization term should be properly weighted not only with λ but also with the inverse of the number of updates needed to see all the training data ([Ben12]).

Another popular regularization method is dropout. The basic idea behind dropout is to randomly omit a certain percentage (e.g. p) of the neurons in each hidden layer for each presentation of the samples during training. This would mean that each random combination of the remaining neurons needs to perform well during training, which implies that each neuron depends less on other neurons to detect patterns.

One way to look at dropout is as a technique to add random noise to the training data, since each neuron takes input from a random collection of the neurons in the previous layer. Consequently, the excitation value will be different even if the same input is fed to the DNN. The improvement in generalization comes from the fact that the capacity of the DNN is reduced since some weights must be dedicated to removing the noise inserted by dropout.

A different perspective of dropout is as a *bagging* technique. *Bagging* refers to the combination of different classifiers, each one performing better than the rest at identifying a particular pattern in the data. The resemblance of dropout to bagging comes from the fact that omitting neurons creates in essence a different layer, thus a different classifier, during each feed-forward pass. After multiple passes it is as if we are averaging the outputs of these "different" classifiers to produce the final output of our model.

It should be noted that regularization affects only the weights and not the biases of the layers. This is because having a large bias does not make the network as sensitive to its input as having large weights would.

Batch size The term *batch size* refers to the number of samples we observe before we make an update to the model parameters. This number will affect the convergence speed as well as the end model.

The simplest choice of batch size is the whole training set. This approach, also known as batch training, has several advantages: it has good convergence properties, there are many techniques that can speed it up like conjugate gradient ([HS52]) and L-BFGS ([LN89]) and it allows for easy parallelization. On the other hand, seeing the whole dataset before a parameter update, is prohibitive for large datasets, which are often met in speech processing tasks.

Another approach to setting the batch size is to set it equal to one. In this approach, commonly referred to as stochastic gradient descent (*SGD*) or online learning, we update the model parameters after seeing just a single sample from the training set. If the sample is independently and identically distributed, it can be shown that the gradient of the training criterion estimated from it is an unbiased estimation of the gradient on the whole training set, despite the fact that it is noisy. This noise is the advantage of the SGD over the batch training approach. Since deep neural networks are highly non-linear and non-convex, the loss function contains many local minima, many of which are poor. Batch learning will converge to the minimum of whatever basin the initial model parameters are in and therefore it makes the model highly dependent on its initialization. On the other hand, the noise present in the gradient estimation of the SGD approach can help the algorithm move away from a poor local minimum and go closer to a better one. This property is reminiscent of simulated annealing, which allows the model parameters to move in a direction locally poorer but globally better ([KGV83]).

Moreover, SGD often converges much faster than batch learning, especially in large datasets. This is because first, it does not waste time in redundant samples in the dataset as batch learning does, and second because in each update the new gradient is estimated based on the new model rather than the old. This means that SGD moves faster towards the best model.

However, SGD cannot be easily parallelized and it will fluctuate around the local minimum due to the noise present in the gradient estimation, without fully converging to it. This fluctuation, can be sometimes useful, as it reduces overfitting.

The third approach to batch size is to set it to a number, usually small, between one and the training set size. This approach, called minibatch training, estimates the gradient based on a small number (a batch) of *randomly* drawn training samples. It must be noted that drawing the samples at random is crucial to the minibatch and the SGD approaches to training.

Similar to SGD, the minibatch-estimated gradient is also unbiased, yet the vari-

ance of the estimation is smaller than that in the SGD algorithm. Furthermore, we can easily parallelize the computations within the minibatch, which makes this approach converge faster than SGD.

Care should be taken as to how to choose the appropriate minibatch size, yet fortunately it can be chosen independently of the rest of the learning parameters. It can be determined automatically considering the variance of the gradient estimation, or it can be empirically determined by searching on a small subset of samples in each epoch. As far as speech recognition tasks are concerned, a minibatch size of 64-256 is good for the early stages of the training, which can be expanded to 1024-8096 in later stages.

Momentum In neural network training adding a momentum term to the update of the model parameters is known to improve the convergence speed. This is due to the fact that the parameters' updates will be based on all the previous gradients instead of only the current one and the oscillation problems commonly seen in the back-propagation algorithm will be reduced.

Learning rate and stopping criterion One of the most difficult to choose parameters in DNN training, yet maybe the most crucial, is the learning rate, as it can determine whether the training will convergence to an optimum, as well as the convergence speed. Most learning rate strategies choose an appropriate initial learning rate and modify it during training. The modifications are based on a strategy usually defined empirically.

A common practice to determine the initial optimal learning rate is to set it to the largest value that does not make the training criterion diverge. Consequently, one can start with a large value, observe the development of the training criterion and if it diverges, try setting the learning rate to a value three times smaller than the initial and proceed likewise until there is no divergence ([Ben12]).

In the case of batch learning, an empirical suggestion for a learning strategy is to decrease the learning rate when the weight vectors oscillate and increase it when the weights follow a relatively steady course. However, this approach cannot be applied to SGD learning since the weights continuously fluctuate ([LBOM98]).

It has been proven that when the learning rate is set to

$$\epsilon = \frac{c}{t}$$

where t is the number of samples seen by the network and c is a constant, or generally

$$\epsilon_t = \frac{\epsilon_0 \tau}{max(t, \tau)}$$

which maintains a constant learning rate for the first τ steps and then decreases it in $\mathcal{O}(\frac{1}{t})$, SGD converges asymptotically, although convergence will be slow since ϵ will quickly become too small ([YD14],[LBOM98]).

Network Architecture Network architecture, meaning the number of hidden layers and the number of neurons in each layer, is an important model parameter that can affect the computational complexity and consequently the training speed, as well as the performance of the end model itself.

Each layer in the network works in essence as a feature extractor of the previous layer. Consequently, one must make sure that the number of neurons in it is large enough to capture the patterns of interest of the input vector. This becomes especially important in the case of the lower layers, since lower layer features are more variable, and more neurons are required to model the hidden patterns. However, having too many neurons encloses the danger of overfitting. As a rule of thumb, the wide, shallow networks are easier to overfit whereas the narrow, deep networks are easier to underfit to the training data.

Taking these into consideration, it is often the case that a different number of neurons is used in each layer, with wide layers with many neurons being placed close to the network input and narrow layers with less neurons being placed close to the output. If however all layers have the same number of neurons, simply adding more layers in some cases might cause the network to underfit since the extra layers add extra constraints to the model parameters.

On the other hand, Bengio ([Ben12]) reports that networks using same-size layers perform better or the same as using decreasing or increasing size layers. He mentions though, that this could depend on the data used.

Regarding speech recognition tasks, it has been found that 5-7 layer DNNs with 1000-3000 neurons per layer work well.

4.3 Pretraining

Considering the above discussion and the difficulties in DNN training, it must have already been clear that the initialization of the model parameters is of utmost importance to the performance of the network. We will now briefly examine some pretraining techniques which aim at properly initializing a DNN to facilitate training and improve performance.

Generative pretraining

The idea of generative pretraining stems from the fact that generative models are able to identify more complex relationships between data and target labels than discriminative models. Before describing this pretraining method it makes sense to introduce the generative models that it is based on.

Energy-based models[Ben09] Energy-based models are models basing their learning procedure on the idea of *energy*. Energy is a scalar function of each configuration of the variables of interest in our model. Such models learn by modifying the shape of the energy function so that desirable configurations of their variables have low energy.

The probability distribution of energy-based models assumes the following form:

$$P(x) = \frac{e^{-Energy(x)}}{Z}$$

where Z, called the *partition function*, is

$$Z = \sum_x e^{-Energy(x)}$$

If the energy function is a sum of functions f_i

$$Energy(x) = \sum_i f_i(x)$$

then the probability distribution becomes

$$P(x) \propto \prod_i P_i(x) \prod_i e^{-f_i(x)}$$

In that case, if we assume that f_i is an "expert" imposing a constraint on x, then $P(x)$ is a "product of experts" and all f_i form a distributed representation of the configuration space, where more than one expert can contribute to each region in space. In contrast, if a different energy form leads to a weighted sum of experts, i.e. $P(x)$ is a "mixture of experts", then the experts do not create a distributed representation of the configuration space, since belonging to a region excludes all the rest. Distributed representations of data, are internal representations of the observed data in such away that they are modeled as being explained by the interactions of many latent factors ([DY14]). Such representations provide robustness in representing the internal structure of data and are capable to generalize concepts and relations.

In case there are unobservable components in our data, or we want to introduce hidden components in order to enhance the expressive power of the model, we can inset a hidden part h in the energy function. The probability distribution then becomes

$$P(v, h) = \frac{e^{-Energy(v,h)}}{Z}$$

where

$$Z = \sum_{v,h} e^{-Energy(v,h)}$$

and

$$P(v) = \sum_h \frac{e^{-Energy(v,h)}}{Z}$$

where v is the visible component of the data. If we define the quantity of "free energy" as

$$F_e(v) = -log(\sum_h e^{-Energy(v,h)})$$

then the probability distribution function of the visible part of the data takes the form of an energy-based model p.d.f. without latent components:

$$P(v) = \frac{e^{-F_e(v)}}{Z}$$

and

$$Z = \sum_v e^{-F_e(v)}$$

Boltzmann Machines A Boltzmann Machine is a network of symmetrically connected units that make stochastic decisions about whether to be on or off ([DY14]).

The energy function in a Boltzmann Machine is a general second-order polynomial:

$$Energy(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{a}'\boldsymbol{v} - \boldsymbol{b}'\boldsymbol{h} - \boldsymbol{h}'\boldsymbol{W}\boldsymbol{v} - \boldsymbol{v}'\boldsymbol{D}\boldsymbol{v} - \boldsymbol{h}'\boldsymbol{R}\boldsymbol{h}$$

where the offsets $\boldsymbol{a}_i, \boldsymbol{b}_i$, each associated with a single element of vector $\boldsymbol{x}$ or $\boldsymbol{h}$, and the weight matrices $\boldsymbol{W}, \boldsymbol{D}, \boldsymbol{R}$, each associated with a pair of units are the model parameters ([DY14],[Ben09]).

Restricted Boltzmann Machines A special type of Boltzmann Machine is the Restricted Boltzmann Machine (RBM). It is an undirected graphical model built by a layer of stochastic visible neurons and a layer of stochastic hidden neurons, which together form a bipartite graph with no visible-visible or hidden-hidden connections (figure 4.4).

The energy function of the RBM has the same form of the energy of a Boltzmann Machine, only now $\boldsymbol{D} = \boldsymbol{R} = \boldsymbol{0}$ given the special architecture of the RBM.

For the Bernoulli-Bernoulli RBM, in which $\boldsymbol{v} \in \{0,1\}^{N_v \times 1}$ and $\boldsymbol{h} \in \{0,1\}^{N_h \times 1}$ the energy is

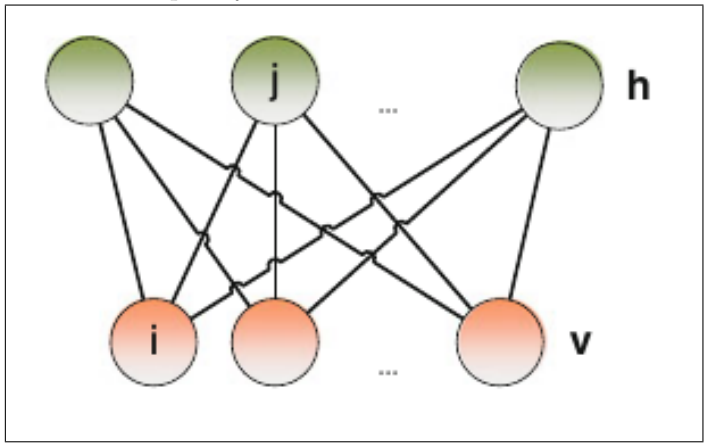
$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{a}^T\boldsymbol{v} - \boldsymbol{b}^T\boldsymbol{h} - \boldsymbol{h}^T\boldsymbol{W}\boldsymbol{v}$$

where N_v, N_h are the number of visible and hidden neurons respectively, $\boldsymbol{W} \in \mathcal{R}^{N_h \times N_v}$ is the weight matrix connecting visible and hidden neurons and $\boldsymbol{a} \in \mathcal{R}^{N_v \times 1}, \boldsymbol{b} \in \mathcal{R}^{N_h \times 1}$ are the bias vectors for the visible and hidden layers respectively.

If the visible neurons take real values, i.e. $\boldsymbol{v} \in \mathcal{R}^{N_v \times 1}$, the RBM formed is called Gaussian-Bernoulli RBM and its energy function is

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\frac{1}{2}(\boldsymbol{v} - \boldsymbol{a})^T(\boldsymbol{v} - \boldsymbol{a}) - \boldsymbol{b}^T\boldsymbol{h} - \boldsymbol{h}^T\boldsymbol{W}\boldsymbol{v}$$

Figure 4.4: *An example of a Restricted Boltzmann Machine [YD14].*



Since RBM is an energy model, each configuration of its units $(\boldsymbol{v}, \boldsymbol{h})$ is associated with a probability

$$P(v, h) = \frac{e^{-E(v,h)}}{Z}$$

where

$$Z = \sum_{v,h} e^{-E(v,h)}$$

Because there are no connections between units of the same layer the posterior probabilities $P(\boldsymbol{v}|\boldsymbol{h})$ and $P(\boldsymbol{h}|\boldsymbol{v})$ can be efficiently computed. For the Bernoulli-Bernoulli RBM we can easily arrive at the expression for the posterior

$$P(\boldsymbol{h}|\boldsymbol{v}) = \prod_i \frac{e^{b_i h_i + h_i \boldsymbol{W}_{i,*}\boldsymbol{v}}}{\sum_{\bar{h}_i} e^{b_i \bar{h}_i + \bar{h}_i \boldsymbol{W}_{i,*}\boldsymbol{v}}} = \prod_i P(h_i|\boldsymbol{v})$$

where $\boldsymbol{W}_{i,*}$ is the i^{th} row of $\boldsymbol{W}$. The above expression reveals that the hidden units are conditionally independent given the visible vector. Since $h_i \in \{0, 1\}$

$$P(\boldsymbol{h} = \mathbf{1}|\boldsymbol{v}) = \sigma(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b})$$

where $\sigma(\cdot)$ is the element-wise logistic sigmoid function, and for the binary visible neurons we symmetrically obtain

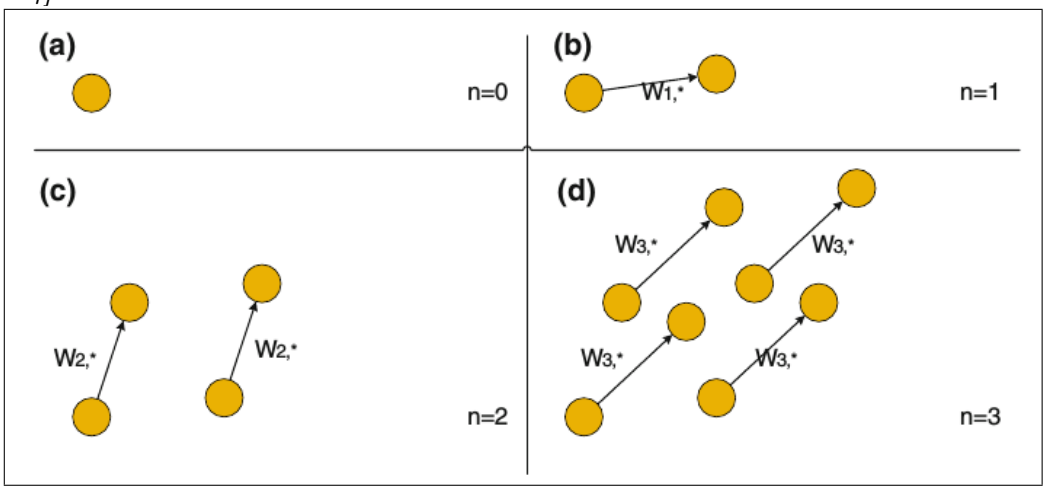
$$P(\boldsymbol{v} = \mathbf{1}|\boldsymbol{h}) = \sigma(\boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{a})$$

In the case of the Gaussian-Bernoulli RBM, the $P(\boldsymbol{h} = \mathbf{1}|\boldsymbol{v})$ is the same as above and

$$P(\boldsymbol{v}|\boldsymbol{h}) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{a}, I)$$

where I is the identity covariance matrix.

Figure 4.5: *Marginal probability density function of a Gaussian-Bernoulli RBM [YD14].*



At this point it is important to note that the posterior probability of the hidden units given the visible, independently of the input values, has the same form as the activation function of the deep multilayer perceptron (DNN) with sigmoidal hidden units. This is where the idea of generative pretraining is based, since we can equate the inference for RBM hidden units with forward computation in a DNN and use the weights of an RBM to initialize a feed-forward DNN with sigmoidal hidden units.

Using the free energy defined earlier we can express the marginal probability $P(\boldsymbol{v})$ as

$$P(\boldsymbol{v}) = \sum_{\boldsymbol{h}} P(\boldsymbol{v}, \boldsymbol{h}) = \frac{e^{-F(\boldsymbol{v})}}{\sum_{v} e^{-F(v)}}$$

For a Gaussian-Bernoulli RBM with no hidden neurons, the marginal probability density function is

$$p_0(\boldsymbol{v}) = \frac{e^{-\frac{1}{2}(\boldsymbol{v}-\boldsymbol{a})^T(\boldsymbol{v}-\boldsymbol{a})}}{Z_0}$$

which is a unit-variance Gaussian distribution centered at vector $\boldsymbol{a}$ (figure 4.5, part (a)).

It is easy to show ([YD14]) that when a new hidden neuron is added and the rest of the model parameters are fixed, the initial distribution is scaled and a copy of it is placed along the direction determined by $\boldsymbol{W}_{n,*}$ (figure 4.5, parts (b)-(d)). This means that RBMs represent their visible inputs as a Gaussian Mixture Model with an exponential number of unit-variance Gaussians. Consequently, RBMs can be used in generative models replacing GMMs, since, for example, a Gaussian-Bernoulli RBM can represent real-valued data distributions in a similar way to GMMs.

Another useful property of RBMs is that if we present the training data to the network, it can learn the correlation between different dimensions of the feature

vectors, i.e. represent inter-visible connections though the hidden neurons they connect to. However, this also implies that the hidden layers can be used to learn a different representation of the raw input.

When it comes to learning the RBM parameters, one can perform stochastic gradient descent using as training criterion the minimization of the negative log-likelihood:

$$J_{NLL}(\boldsymbol{W}, \boldsymbol{a}, \boldsymbol{b}; \boldsymbol{v}) = -logP(\boldsymbol{v}) = F(\boldsymbol{v}) + log \sum_{v} e^{F(v)}$$

However, the computation of the gradient of the log-likelihood of the data is infeasible to compute exactly:

$$\nabla_\theta J_{NLL}(\boldsymbol{W}, \boldsymbol{a}, \boldsymbol{b}; \boldsymbol{v}) = -\Big[\Big\langle \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \theta} \Big\rangle_{data} - \Big\langle \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \theta} \Big\rangle_{model}\Big]$$

where θ is some model parameter and $\langle\cdot\rangle_{data}$ and $\langle\cdot\rangle_{model}$ are the expectation of the argument estimated from the training data and the model respectively. The first expectation $\langle\cdot\rangle_{data}$ can be computed from the training set, but $\langle\cdot\rangle_{model}$ takes exponential time to compute exactly when the hidden values are unknown. Consequently, we have to use approximated methods for RBM training, with the most widely used being the Contrastive Divergence learning algorithm ([Hin02]).

The Contrastive Divergence algorithm aims to locally approximate the gradient of the training criterion around a training point. Its goal is to discover a decision surface to separate high- from low-probability regions by comparing training samples and samples generated by the model. The term "contrastive" comes exactly from the fact that it builds on the contrast between these two classes of samples.

The algorithm starts by initializing a sampling process (*Gibbs sampling*) at a training data sample. It then generates a hidden sample from the visible sample based on the posterior probability $P(\boldsymbol{h}|\boldsymbol{v})$, defined accordingly to the RBM model used (Gaussian-Bernoulli or Bernoulli-Bernoulli). This hidden sample is further used to generate a visible sample based on the posterior probability $P(\boldsymbol{v}|\boldsymbol{h})$. This process may continue for many steps. If it continues for an infinite number of steps the true expectation $\langle\cdot\rangle_{model}$ can be estimated. However, it has been found that even one step of the algorithm is enough to provide us with a good estimate of $\langle\cdot\rangle_{model}$ and consequently of the gradient of the training criterion ([Hin12]).

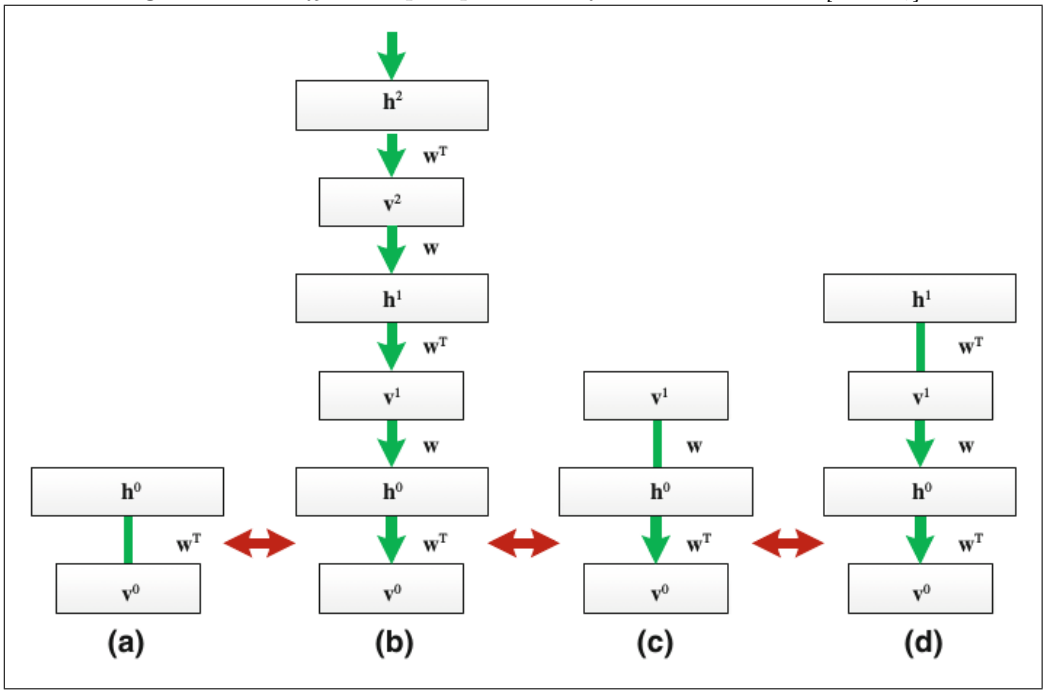
Deep Belief Networks We can view an RBM as a generative model with infinite number of layers all of which share the same weight matrix (figure 4.7, parts (a)-(b)).

If we separate the bottom layer from the deep model of figure 4.7(b) the remaining layers form another generative model with infinite number of layers sharing the same weight matrices, i.e. they are equivalent to another RBM whose visible

Figure 4.6: *Contrastive Divergence algorithm for RBM training [YD14].*

procedure TRAINRBMWITHCD($\mathbb{S} = \{\mathbf{o}^m | 0 \leq m < M\}, N$)
▷ $\mathbb{S}$ is the training set with M samples, N is the CD steps
  Randomly initialize $\{\mathbf{W_0}, \mathbf{a}_0, \mathbf{b}_0\}$
  while Stopping Criterion Not Met **do**
    ▷ Stop if reached max iterations or the training criterion improvement is small
    Randomly select a minibatch $\mathbf{O}$ with M_b samples.
    $\mathbf{V}^0 \leftarrow \mathbf{O}$ ▷ Positive phase
    $\mathbf{H}^0 \leftarrow P(\mathbf{H}|\mathbf{V}^0)$ ▷ Applied column-wise
    $\nabla_{\mathbf{W}} J \leftarrow \mathbf{H}^0 \left(\mathbf{V}^0\right)^{\mathrm{T}}$
    $\nabla_{\mathbf{a}} J \leftarrow \text{sumrow}\left(\mathbf{V}^0\right)$ ▷ Sum along rows
    $\nabla_{\mathbf{b}} J \leftarrow \text{sumrow}\left(\mathbf{H}^0\right)$
    for $n \leftarrow 0; n < N; n \leftarrow n + 1$ **do** ▷ Negative phase
      $\mathbf{H}^n \leftarrow \mathbb{I}\left(\mathbf{H}^n > \text{rand}\,(0, 1)\right)$ ▷ Sampling, $\mathbb{I}(\bullet)$ is the indicator function
      $\mathbf{V}^{n+1} \leftarrow P(\mathbf{V}|\mathbf{H}^n)$
      $\mathbf{H}^{n+1} \leftarrow P(\mathbf{H}|\mathbf{V}^{n+1})$
    end for
    $\nabla_{\mathbf{W}} J \leftarrow \nabla_{\mathbf{W}} J - \mathbf{H}^N \left(\mathbf{V}^N\right)^{\mathrm{T}}$ ▷ Subtract negative statistics
    $\nabla_{\mathbf{a}} J \leftarrow \nabla_{\mathbf{a}} J - \text{sumrow}\left(\mathbf{V}^0\right)$
    $\nabla_{\mathbf{b}} J \leftarrow \nabla_{\mathbf{b}} J - \text{sumrow}\left(\mathbf{H}^0\right)$
    $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t + \frac{\varepsilon}{M_b} \triangle \mathbf{W}_t$ ▷ Update $\mathbf{W}$
    $\mathbf{a}_{t+1} \leftarrow \mathbf{a}_t + \frac{\varepsilon}{M_b} \triangle \mathbf{a}_t$ ▷ Update $\mathbf{a}$
    $\mathbf{b}_{t+1} \leftarrow \mathbf{b}_t + \frac{\varepsilon}{M_b} \triangle \mathbf{b}_t$ ▷ Update $\mathbf{b}$
  end while
  Return $rbm = \{\mathbf{W}, \mathbf{a}, \mathbf{b}\}$
end procedure

Figure 4.7: *Different perspectives of the same RBM [YD14].*



layer and hidden layer are switched (figure 4.7, part (c)). This model is a special generative model called Deep Belief Network (*DBN*), where the bottom layers form a directed generative model and the top layer is an undirected RBM. Thinking in the same way we can derive the equivalent model shown in figure 4.7(d).

Because the DBN is related closely to the RBM, we can apply a layer-wise process to train deep generative models. We start by training an RBM, which, after the training, can discover a different representation of the feature vectors. Hence, for each vector $\boldsymbol{v}$ we compute a vector of expected hidden neuron activations $\boldsymbol{h}$. These hidden expectations can be fed to a new RBM as training data. Continuing in this way we can use each set of RBM weights to extract features from the output of the previous layer. When we do not need to train more RBMs, we can use the weight matrices to initialize the hidden layers of a DBN with as many hidden layers as the RBMs we have trained. The final DBN can be further fine-tuned.

This procedure can be used to stack RBMs with different or the same number of dimensions which allows us to improve the flexibility and performance of the DBN.

Weight initialization of a DNN The DBN weights can be used to initialize the weights of a sigmoidal DNN. This is due to the fact that $P(\boldsymbol{h}|\boldsymbol{v})$ in the RBM has the same form as that in the DNN provided the layers use the sigmoidal activation function. The DNN can be viewed as a statistical graphical model where each hidden layer $0 < l < L$ models posterior probabilities of conditionally independent hidden binary neurons $\boldsymbol{h}^l$ given input vectors $\boldsymbol{v}^{l-1}$ as Bernoulli distribution

$$P(\boldsymbol{h}^l|\boldsymbol{v}^{l-1}) = \sigma(\boldsymbol{z}^l) = \sigma(\boldsymbol{W}^l\boldsymbol{v}^{l-1} + \boldsymbol{b}^l)$$

and the output layer approximates the label $\boldsymbol{y}$ as

$$P(\boldsymbol{y}|\boldsymbol{v}^{L-1}) = softmax(\boldsymbol{z}^L) = softmax(\boldsymbol{W}^L\boldsymbol{v}^{L-1} + \boldsymbol{b}^L)$$

Given the observation vector $\boldsymbol{o}$ and its label $\boldsymbol{y}$, the precise modeling of $P(\boldsymbol{y}|\boldsymbol{o})$ requires integration over all possible values of $\boldsymbol{h}$ across all layers. However, because this is infeasible, we replace the marginalization with the mean-field approximation:

$$\boldsymbol{v}^l = \mathrm{E}(\boldsymbol{h}^l|\boldsymbol{v}^{l-1}) = P(\boldsymbol{h}^l|\boldsymbol{v}^{l-1}) = \sigma(\boldsymbol{W}^l\boldsymbol{v}^{l-1} + \boldsymbol{b}^l)$$

which is the non-stochastic description of the DNN previously presented.

Weight initialization via generative pretraining can improve the DNN performance on the test set. This is due to three main reasons. First, the initial point of the learning algorithm can greatly affect the end model, especially in batch training. Second, since we only need labels for the fine-tuning part of the training, we can utilize a large amount of unlabeled data, which would be useless without pretraining. Third, the generative criterion used in pretraining is different from the

discriminative criterion used in fine-tuning, which means that pretraining can act as a data-dependent regularizer.

Bengio in [Ben09] debates whether unsupervised (generative) pretraining has in principle a regularization or optimization effect.

Given that generative pretraining is equivalent to imposing a constraint on where in the parameter space a solution is allowed, i.e. near solutions of the unsupervised training criterion that capture important statistical structure in the data, it can be seen as a form of regularization. Furthermore, generative pretraining plays a more important role in the initialization of the lower layers. Experiments have shown that when there are no constraints on the number of hidden neurons at the top two layers of the network, the representation computed by the lower layers is unimportant: the network can still be trained to minimize the training error as much as desired - despite the fact that it may generalize poorly. However, if we have to use smaller top layers, then pretraining of the lower layers becomes necessary to get a low training error and better generalization.

The optimization perspective of unsupervised pretraining requires to focus on tuning the lower layers while the top two layers are kept small in terms of the number of neurons or of the magnitude of their weights. Bengio suggests that if unsupervised pretraining worked purely as a regularizer, then if we had an infinite stream of training data and performed online learning - in which case we would in essence minimize the generalization error - then with or without pretraining the error would converge to the same level. However, after emulating such a setting, he discovered that the pretrained network achieved a lower minimum, which suggests that there is an optimization component in generative pretraining.

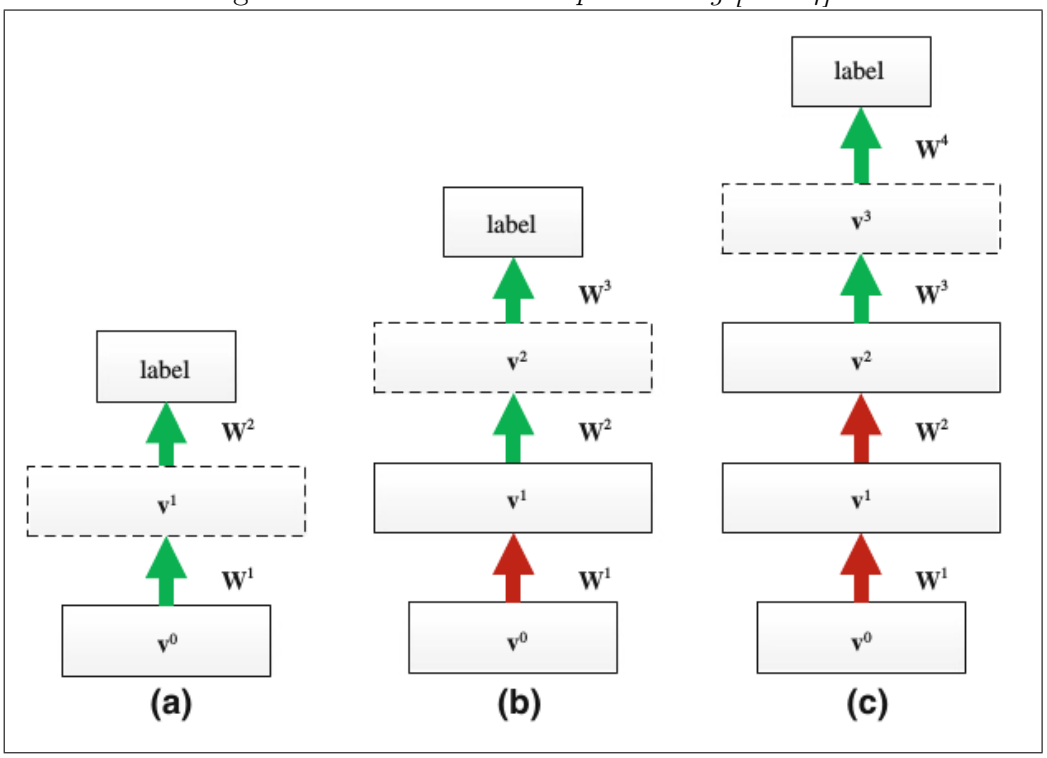
In any case, there has been no evidence so far of generative pretraining hurting the DNN training.

Apparently, generative pretraining is highly dependent on the task. It also works best with two hidden layers and is of trivial importance when the network has only one hidden layer. If we add more hidden layers effectiveness often decreases, since modeling errors introduced by the mean-field approximation and the contrastive divergence algorithm accumulate ([YD14]).

Discriminative pretraining

A different, "incremental", approach to pretraining is to discriminatively train each layer using back-propagation. Pretraining begins by training a one-hidden-layer network to convergence, using labels. Then, we insert another hidden layer before the output layer, randomly initialized, and retrain the whole network to convergence. We continue in the same way until the network reaches the desired

Figure 4.8: *Discriminative pretraining [YD14].*



number of hidden layers (figure 4.8)

It is important to note that all hidden layers are updated during training and not only the layer added last. Because of this, if we are planning to add more hidden layers, the network should not be trained to full convergence, to avoid having some hidden neurons operating in the saturation region and thus being unable to further update. A common heuristic is to go through the training set $\frac{1}{L}$ times of the total number of the data passes needed to convergence, where L is total number of layers in the final model.

Discriminative pretraining aims to bring the weights close to a good local optimum and it does not have the regularization effect of generative pretraining. Therefore, it is best used when large amounts of training data are available.

Hybrid pretraining

Both pretraining techniques presented work well, however they both have some drawbacks.

Generative pretraining is not directly related to the task-specific objective function, hence it is not guaranteed to help the discriminative fine-tuning phase. However, it does help to reduce overfitting. On the other hand, discriminative pretraining minimizes the training criterion, yet it entails the danger of over-tuning the lower

layers to the final objective thus making them unable to learn when new hidden layer are added.

To tackle these issues, it has been suggested to optimize a weighed sum of the generative and discriminative pretraining criteria. We thus have a hybrid criterion to optimize:

$$J_{HYB}(\boldsymbol{W}, \boldsymbol{b}; \mathcal{D}_{train}) = J_{DISC}(\boldsymbol{W}, \boldsymbol{b}; \mathcal{D}_{train}) + \alpha J_{GEN}(\boldsymbol{W}, \boldsymbol{b}; \mathcal{D}_{train})$$

The discriminative criterion can be cross-entropy or mean-square error whereas the generative can be negative log-likelihood. Intuitively, the generative criterion acts as a data-dependent regularizer for the discriminative.

Hybrid pretraining can outperform both the generative and the discriminative pretraining approaches. Despite the fact the the importance of pretraining diminishes as the amount of training data increases, pretraining can still help to make the training procedure more robust.

Dropout pretraining

We have already seen that dropout can be considered a bagging technique, thus generate a smoother objective surface. This implies that dropout could be used to pretrain a DNN to easily find a good starting point on the smoothed objective surface and then fine-tune the model without dropout.

Dropout requires labeled data and achieves similar performance to the discriminative pretraining, as well as being easier to implement and control.

4.4 Recurrent Neural Networks

[YD14]

Research in ASR has been working towards building models that can adequately emulate the human speech production and perception system. However, as we have seen, GMM/HMM models fail to model the dynamic and hierarchical structure of the human speech system and DNN/HMM models exhibit the same type of limitations. A Recurrent Neural Network (*RNN*) is a class of neural network models where many connections among its neurons form a directed cycle (hence the name *recurrent*), thus providing the network with a structure of internal states, or memory, which helps it exhibit a dynamic temporal behavior missing from the DNN.

The internal representation of dynamic speech features in the RNN is discriminatively formed by feeding the low-level features into the hidden layers, together with the hidden features from the past history. The RNN implements a time-delay

operation over the temporal dimension which allows its internal states to function as memory and the network to exhibit a dynamic temporal behavior.

RNNs are considered deep models since if we unfold the network in time, we will create as many layers in the network as the length of the input speech utterance.

In this section we will present the fundamental RNN model and the two basic ways to train it, as well as a model which has received a lot of attention recently, the *Long-Short-Term Memory*.

4.4.1 State-Space formulation of the basic RNN

The RNN differs from the DNN in that it operates not only on its inputs but also on its internal states, which encode the past information that has already been processed.

We will express the one-hidden layer RNN in terms of the noise-free nonlinear state-space model often used in signal processing, which will enable the learning of sequentially extended dependencies over a long time span. If $\boldsymbol{x}_t$ is the $K \times 1$ input vector, $\boldsymbol{h}_t$ is the $N \times 1$ vector of hidden state values and $\boldsymbol{y}_t$ is the $L \times 1$ output vector, the one-hidden layer RNN can be described as

$$\boldsymbol{h}_t = f(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}) \ (state\ equation)$$

$$\boldsymbol{y}_t = g(\boldsymbol{W}_{hy}\boldsymbol{h}_t) \ (observation\ equation)$$

where $\boldsymbol{W}_{hy}$ is the $L \times N$ weight matrix connecting the N hidden units to the L outputs, $\boldsymbol{W}_{xh}$ is the $N \times K$ weight matrix connecting the K inputs to the N hidden units, and $\boldsymbol{W}_{hh}$ is the $N \times N$ weight matrix connecting the N hidden units from time $t-1$ to time t.

Let us further define

$$\boldsymbol{u}_t = \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}$$

as the $N \times 1$ vector of hidden layer potentials and

$$\boldsymbol{v}_t = \boldsymbol{W}_{hy}\boldsymbol{h}_t$$

as the $L \times 1$ vector of output layer potentials.

Then, $f(\boldsymbol{u}_t)$ is the hidden layer activation function and $g(\boldsymbol{v}_t)$ is the output layer activation function. Commonly used activation functions in the hidden layers are the *sigmoid*, *tanh* and rectified linear units (*ReLU*), whereas typical output layer activation functions are the *linear* and $softmax$ functions.

In case we want to use the output from previous time frames to update the state vector the state equation becomes

$$\boldsymbol{h}_t = f(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{yh}\boldsymbol{y}_{t-1})$$

where $\boldsymbol{W}_{yh}$ is the weight matrix connecting the output layer to the hidden layer.

Figure 4.9: *Backpropagation-through-time [YD14].*

procedure BPTT($\{\mathbf{x}_t, \mathbf{I_t}\}\ 1 \leq t \leq T$)
▷ $\mathbf{x}_t$ is the input feature sequence
▷ $\mathbf{I_t}$ is the label sequence
▷ forward computation
for $t \leftarrow 1; t \leq T; t \leftarrow t+1$ **do**
  $\mathbf{u}_t \leftarrow \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}$
  $\mathbf{h}_t \leftarrow f(\mathbf{u}_t)$
  $\mathbf{v}_t \leftarrow \mathbf{W}_{hy}\mathbf{h}_t$
  $\mathbf{y}_t \leftarrow g(\mathbf{v}_t)$
end for
▷ backpropagation through time
$\delta_T^y \leftarrow (\mathbf{l}_T - \mathbf{y}_T) \bullet g'(\mathbf{v}_T)$
▷ •: element-wise multiplication
$\delta_T^h \leftarrow \mathbf{W}_{hy}^T \delta_T^y \bullet f'(\mathbf{u}_T)$
for $t \leftarrow T-1; t \geq 1T; t \leftarrow t-1$ **do**
  $\delta_t^y \leftarrow (\mathbf{l}_t - \mathbf{y}_t) \bullet g'(\mathbf{v}_t)$
  $\delta_t^h \leftarrow \left[\mathbf{W}_{hh}^T \delta_{t+1}^h + \mathbf{W}_{hy}^T \delta_t^y\right] \bullet f'(\mathbf{u}_t)$
▷ propagate from δ_t^y and δ_{t+1}^h
end for
▷ model update
$\mathbf{W}_{hy} \leftarrow \mathbf{W}_{hy} + \gamma \sum_{t=1}^{T} \delta_y^t \mathbf{h}_t^T$
$\mathbf{W}_{hh} \leftarrow \mathbf{W}_{hh} + \gamma \sum_{t=1}^{T} \delta_h^t \mathbf{h}_{t-1}^T$
end procedure

4.4.2 Training

Back-propagation-through-time algorithm

The back-propagation-through-time ($BPTT$) method ([Bod01]) manages to learn the weight matrices of the RNN by first unfolding the network in time and then propagating errors backwards through time. The algorithm is an extension of the original back-propagation algorithm for feed-forward neural networks, yet now the stacked hidden layers for the same training frame t have been replaced by the T same single hidden layers across time $t = 1, 2, ..., T$.

The training criterion of the BPTT algorithm is the well-known sum-square error

$$E = \frac{1}{2} \sum_{t=1}^{T} \|\boldsymbol{l}_t - \boldsymbol{y}_t\|^2$$

where $\boldsymbol{l}_t$ is the target vector and $\boldsymbol{y}_t$ is the output vector over all time frames. Our goal is to minimize this cost with respect to the weights and for that we will use the gradient descent rule. The algorithm is presented in figure 4.9

It should be noted that contrary to the DNN backpropagation algorithm, the RNN weight matrices are spatially duplicated for an arbitrary number of time steps, i.e. they are *tied*. That is the reason why in the update step of the algorithm we sum over all time frames.

The computational complexity of the BPTT is $O(M^2)$ per time step, where $M = LN + NK + N^2$ is the total number of the model parameters. It converges

slower the the original back-propagation algorithm due to dependencies between frames, and is highly likely to converge to a poor local optimum due to exploding or vanishing gradients. Although we can limit the past history to the last p time steps and thus improve speed, it is still difficult to achieve good results without much experimentation and tuning.

Primal-dual learning algorithm

The well-known problems of the back-propagation algorithm in conjunction with gradient computation are even more evident in the case of RNNs. Learning RNNs is particularly difficult because of the exploding (too big) and vanishing (too small) gradients.

A sufficient condition for vanishing gradients to occur is $\|\boldsymbol{W}_{hh}\| < d$, where d is a constant dependent on the activation function of the hidden units ($d = 4$ for sigmoidal units and $d = 1$ for linear) and $\|\boldsymbol{W}_{hh}\|$ is the largest singular value of the weight matrix $\boldsymbol{W}_{hh}$.

Similarly, a necessary condition for exploding gradients is $\|\boldsymbol{W}_{hh}\| > d$.

Therefore, in order to effectively learn an RNN, we have to impose certain constraints on the weight matrix $\|\boldsymbol{W}_{hh}\| > d$. For example, to avoid vanishing gradients, we could add a regularization term to increase the gradient or exploit curvature-related information about the regularization function; to avoid exploding gradients we could clip the gradient at an empirically-determined threshold.

A more rigorous approach to RNN learning would be to directly exploit the conditions described above regarding $\boldsymbol{W}_{hh}$, and this is the focus of the primal-dual algorithm for training RNNs. This algorithm tries to preserve the *echo state property* which is closely related to the previous constraints and states that *if the network has been run for a very long time, the current network state is uniquely determined by the history of the input and the teacher-forced output.* Mathematically, a sufficient condition for the network to satisfy the echo state property is

$$\|\boldsymbol{W}_{hh}\|_\infty < d$$

where $\|\boldsymbol{W}_{hh}\|_\infty$ is the maximum absolute sum of $\boldsymbol{W}_{hh}$, $d = 4$ for sigmoidal units and $d = 1$ for hyperbolic tangent units.

Given that we want to preserve the echo state property, RNN learning can be formulated as a constraint optimization problem:

$$\min_{\Theta} E(\Theta) = \min_{\Theta} E(\boldsymbol{W}_{hh}, \boldsymbol{W}_{xh}, \boldsymbol{W}_{hy})$$

subject to

$$\|\boldsymbol{W}_{hh}\|_\infty \leq d$$

The Lagrangian of this problem can be formulated as:

$$L(\Theta, \boldsymbol{\lambda}) = E(\boldsymbol{W}_{hh}, \boldsymbol{W}_{xh}, \boldsymbol{W}_{hy}) + \sum_{i=1}^{N} \lambda_i \Big(\sum_{j=1}^{N} |W_{ij}| - d \Big)$$

where $\lambda_i \geq 0$ is the ith entry of the Lagrange vector λ.

Then, the dual function $q(\lambda)$ is the solution to the following unconstrained optimization problem

$$q(\lambda) = \min_{\Theta} L(\Theta, \lambda)$$

and is a lower bound of the original constrained optimization problem, i.e.

$$q(\lambda) \leq E(\Theta^*)$$

Maximizing $q(\lambda)$ will be the best lower bound that can be obtained for $E(\Theta)$. This is called the *dual problem* of the original optimization problem:

$$\max_{\boldsymbol{\lambda}} q(\boldsymbol{\lambda}), \;\; \lambda_i \geq 0, \;\; i = 1, ..., N$$

and is a convex optimization problem since we aim to maximize a concave objective with linear equality constraints.

Having solved the dual problem for λ^* we substitute the optimal dual variables into the Lagrangian and solve for the set of parameters Θ that minimize $L(\Theta, \lambda^*)$:

$$\Theta^0 = \operatorname*{argmin}_{\Theta} L(\Theta, \lambda^*)$$

This solution will be an approximation to an optimal solution to the original constrained optimization problem. However, in general non-convex problems where acquiring a globally optimal solution is infeasible, this approximation will be good enough.

Following the two-stepped solution, the updates of the RNN parameters happen in two steps:

- primal update: minimize $L(\Theta, \lambda^*)$ w.r.t. Θ
- dual update: maximize $L(\Theta^*, \lambda)$ w.r.t. λ

The standard gradient descent algorithm can be applied to the update rules with some improvements being possible due to the structure in the objective function.

4.4.3 RNNs with Long-Short-Term Memory cells

Motivation

The basic RNN presented so far does not have the necessary structure to model complex temporal dynamics and is thus incapable of looking far back into the past in many types of input sequences.

One way to solve this problem is to introduce a memory structure into the RNN which leads to the incorporation of *long-short-term memory cells* (LSTM)([HS97]). Due to its structure, the LSTM-RNN can recognize temporally extended patterns in noisy input sequences and temporal order of widely separated events in noisy input streams. Like the basic RNN, the LSTM-RNN is a universal computing machine, provided it has enough network units and receives proper training, and, in addition, it can learn from input sequence data to classify, process and predict time series with very long time lags of unknown lengths between important events.

The LSTM-RNN has been shown to perform very well in handwriting and phone recognition, keyword spotting, reinforcement learning for robot localization and control, online learning for protein structure prediction, learning music composition and grammars, language identification, speech synthesis and environment-robust speech recognition.

Architecture of LSTM cells

The basic notion behind LSTM cells in the RNN is to use various types of element-wise multiplication (or *gating*) structure to control the information flow in the network. Such a cell can be viewed as a complex and smart network unit able to remember information for a long time. This is the job of the gating structure which is in charge of determining when the input is important enough to remember, when the cell should continue to remember or forget the information and when it should output the information.

An LSTM cell is mathematically described by the following forward operations iteratively over time $t = 1, 2, ..., T$:

$$\boldsymbol{i}_t = \sigma\Big(\boldsymbol{W}^{(xi)}\boldsymbol{x}_t + \boldsymbol{W}^{(hi)}\boldsymbol{h}_{t-1} + \boldsymbol{W}^{(ci)}\boldsymbol{c}_{t-1} + \boldsymbol{b}^{(i)}\Big)$$

$$\boldsymbol{f}_t = \sigma\Big(\boldsymbol{W}^{(xf)}\boldsymbol{x}_t + \boldsymbol{W}^{(hf)}\boldsymbol{h}_{t-1} + \boldsymbol{W}^{(cf)}\boldsymbol{c}_{t-1} + \boldsymbol{b}^{(f)}\Big)$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \bullet \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \bullet \tanh\Big(\boldsymbol{W}^{(xc)}\boldsymbol{x}_t + \boldsymbol{W}^{(hc)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(c)}\Big)$$

$$\boldsymbol{o}_t = \sigma\Big(\boldsymbol{W}^{(xo)}\boldsymbol{x}_t + \boldsymbol{W}^{(ho)}\boldsymbol{h}_{t-1} + \boldsymbol{W}^{(co)}\boldsymbol{c}_t + \boldsymbol{b}^{(o)}\Big)$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \bullet \tanh(\boldsymbol{c}_t)$$

where $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, $\boldsymbol{c}_t$, $\boldsymbol{o}_t$ and $\boldsymbol{h}_t$ are vectors, all with the same dimensionality, which represent five different types of information at time t of the input gate, forget gate, cell activation, output gate and hidden layer respectively; $\sigma(\cdot)$ is the logistic sigmoid function, $\boldsymbol{W}$'s are the weight matrices connecting different gates and $\boldsymbol{b}$'s are the corresponding bias vectors. The weight matrices are full except for $\boldsymbol{W}^{(ci)}$ that is diagonal. In addition to the above set, an additional output layer has to be provided on top of the LSTM-RNN's hidden layer.

Training

The Back-Propagation-Through-Time algorithm introduced for RNN training can be also used in the case of LSTM-RNNs in order to compute error derivatives (although the computation is more complex) and then apply stochastic gradient descent methods to update the weight parameters.

However, the problems of vanishing and exploding gradients encountered in RNN training are less evident here. This is because the error signals back-propagated from the output become trapped in the memory part of the LSTM cells. In this way, meaningful error signals are constantly fed back to each of the gates until the RNN parameters are well trained.

Chapter 5

Manifold regularized deep neural networks in ASR

This chapter covers the process followed to develop an acoustic model using deep neural networks and incorporate a manifold regularization term in the objective function used for training.

It starts by describing the preparatory work that had to be done before proceeding to use the deep network architecture, then it moves on to the incorporation of the DNN in the ASR system and the manifold term in the DNN as described in [TR14b], and finally it presents the experimental results acquired with the above systems.

5.1 Preparatory work

5.1.1 DNN in acoustic modeling

There are two main ways in which we can use deep neural networks in acoustic modeling [YD14]:

- a *hybrid* approach, where we directly compute the observation probability used in the Hidden Markov Model of a previously trained GMM-HMM automatic speech recognition system, i.e. compute the posterior probability of the HMM's state given the acoustic observation

- a *tandem* approach, where we extract a representation of the training features from one of the DNN's layers and feed it to a GMM-HMM system

Hybrid approach

The DNN-HMM system that is the outcome of the hybrid approach combines the strength of the DNN, that is, its representational power, with the benefit of the HMM, that is, its sequential modeling ability.

As we have already seen the combined use of neural networks and HMMs started between the end of the 1980s and the beginning of the 1990s, however, they were only applied to small vocabulary tasks. Research in the area resurrected again after DNNs exhibited their strong representational power and such systems performed well in large vocabulary continuous speech recognition applications.

In these combined systems the dynamics of the speech signal are modeled with the HMMs and the observation probabilities are estimated through the deep network: each output neuron is trained to estimate the posterior probability of a continuous density HMMs' state given an acoustic observation. Mathematically the output of the DNN can be formulated as:

$$P(q_t = s|\mathbf{x}_t), \forall s \in [1, \mathcal{S}]$$

where s is the state of the HMM and $\mathbf{x}_t$ is the input frame of acoustic features. Although in the first hybrid approaches s were the monophone states, in more recent systems DNNs directly model *senones*, i.e. tied triphone states. This has not only improved performance, but it also comes with two extra benefits: first, a DNN-HMM system can be built from an existing GMM-HMM requiring only minimal modifications, and second, any breakthrough in the modeling units of a GMM-HMM can be easily incorporated in the DNN-HMM system, since the improvement will reflect directly in the output units.

Given that the HMM requires the likelihood $p(\mathbf{x_t}|q_t)$ instead of the posterior probability during the decoding process, the DNN output has to be converted as follows:

$$p(\mathbf{x_t}|q_t = s) = \frac{P(q_t = s|\mathbf{x}_t)P(\mathbf{x}_t)}{P(s)}$$

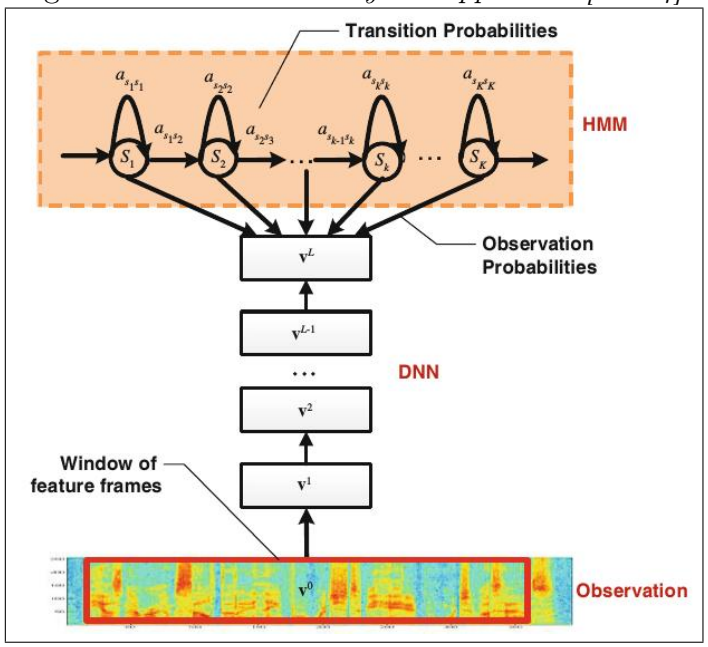
where $p(s)$ is the prior probability of each senone estimated from the training set as

$$P(s) = \frac{T_s}{T}$$

with T being the total number of frames, T_s the total number of frames labeled as s and $p(\mathbf{x}_t)$ an independent of the word sequence term, which can be ignored. The contribution of the prior likelihood is not great to the recognition accuracy but it can be important in reducing the label bias problem.

Taken the above into consideration, the ASR problem with a hybrid DNN-HMM

Figure 5.1: *DNN-HMM hybrid approach. [YD14].*



approach can be formulated in the following way:

$$\hat{w} = \underset{w}{\operatorname{argmax}} P(w|\mathbf{x}) = \underset{w}{\operatorname{argmax}} \frac{p(\mathbf{x}|w)P(w)}{P(\mathbf{x})} = \underset{w}{\operatorname{argmax}}\, p(\mathbf{x}|w)P(w)$$

where $P(w)$ is the language model probability and

$$p(\mathbf{x}|w) = \sum_{q} p(\mathbf{x}|q,w)p(q|w) \approx max\pi(q_0) \prod_{t=1}^{T} a_{q_{t-1}q_t} \prod_{t=1}^{T} \frac{p(q_t|\mathbf{x}_t)}{p(q_t)}$$

is the acoustic modeling probability. In the equation above $p(q_t|\mathbf{x}_t)$ is computed by the DNN, $p(q_t)$ is the prior estimated from the training set, $\pi(q_0)$ is the initial state probability and $a_{q_{t-1}q_t}$ is the state transition probability, both of which are determined by the HMM. Similar to traditional GMM-HMM systems, we can also include a weight λ to balance between the acoustic and language model scores:

$$\hat{w} = \underset{w}{\operatorname{argmax}}[\log p(\mathbf{x}|w) + \lambda \log P(w)]$$

According to a series of studies on LVCSR DNN-HMM systems, the components that have contributed the most to the success of such systems are:

- *the depth of the neural networks*, i.e. the many layers in the architecture. It has been experimentally verified that deeper models have a stronger discriminative ability than shallow models. Bengio in his monograph ([Ben09]) mentions that even though we can achieve the same accuracy by using just a single

Figure 5.2: *WER on Hub5 '00 - Switchboard, 309h training data. Summarized from Seidel et al. 2011* [YD14]

LxN	DBN-PT (%)	1xN	DBN-PT (%)
1×2k	24.2		
2×2k	20.4		
3×2k	18.4		
4×2k	17.8		
5×2k	17.2	1×3,772	22.5
7×2k	17.1	1×4,634	22.6
		1×16K	22.1

wide layer with thousands of units, when there are limitations to the number of parameters, much better performance can be attained by a deep model. In figure 5.2 we see that WER decreases as the number of layers increases, and, even if two networks have the same number of parameters, the deeper model performs better. However, after a certain number of layers, the network saturates and performance starts to decrease. Consequently, a trade-off has to be made between WER improvement on the one hand, and training and decoding cost on the other.

- *the use of a contextual window* as input to the network. Since DNNs are able to capture correlations between neighboring frames and exploit information included in them, it is crucial to use a window of frames (usually 9-13) as input to the network. Apart from capturing extra information though, this also allows us to compensate for the independence assumption made in the HMMs, which states that each feature frame is independent from the rest. However, this is in fact untrue since neighboring frames correlate with each other given the same state.

- *the use of senones as target labels.* Modeling senones allows the model to exploit information encoded in the fine-grained labels and reduce overfitting. Although using senones as targets implies that the number of output nodes will explode and thus classification accuracy will decrease, decoding performance is improved since the state transitions that would be prone to confusion are reduced. According to [YD14], using senone labels as targets has been the largest source of improvement of all design decisions.

It is worth mentioning that pretraining is not as critical to train deep networks as previously thought. It might be beneficial to small networks (less than five layers), however as the number of layers increases, the benefits diminish.

This is partly due to the fact that stochastic gradient descent can escape from local optima and consequently a good initialization for the weights is not of the utmost importance. In addition, the contrastive divergence algorithm employed in

the pretraining phase introduces modeling errors for each layer, which accumulate as the number of layers increases and thus hurt the effectiveness of pretraining.

Finally, although pretraining is deemed to contribute to reduce overfitting, its contribution is not that crucial, provided a huge amount of training data is used.

On the other hand though, even if the improvement in decoding accuracy is small, pretraining can act as an implicit regularizer on the training data, help towards achieving a more robust training and avoid bad weights initialization; thus it might help to achieve good performance with a small training set.

To train a hybrid DNN-HMM system, one first needs to train a conventional GMM-HMM system, since the hybrid system shares with it the phoneme-tying structure and the Hidden Markov Model. The latter will be used both for the final decoding phase and for the provision of the initial training targets. It is therefore crucial to train a good GMM-HMM system, in order to achieve good performance with the DNN. For the experiments conducted in the current project, the Kaldi speech recognition toolkit was used (see appendix A).

Depending on the decoder that will be used, a mapping from states to senones, which will be the training targets, might have to be build. Having built the GMM-HMM and the mapping if needed, the Viterbi algorithm is used on the training set to generate a forced alignment so as to acquire the targets for training. The senone labels are also used to estimate the priors that will be needed to convert the posteriors produced by the DNN to likelihoods that will be used by the HMM for the decoding.

Tandem approach

In the tandem approach, a set of features is extracted from the deep neural network and is then used to train a GMM-HMM ASR system. In this final stage one can either use only the features extracted from the network or use them complementary to traditional features used in ASR, e.g. MFCCs or filterbank.

The idea behind the tandem approach lies in the fact that DNNs can take as input a high-dimensional vector made of many observed variables correlated with each other, and, after passing it through all their layers of computation, extract a more abstract representation out of it. Consequently, it is assumed that DNNs can discover the underlying factors that have lead to the creation of the data in their original form ([Ben09]). Such models can be seen as combining a non-linear transformation model with a classification model, thus being able to transform input features into a discriminative representation that is invariant to factors of variability in speech recognition, such as different speakers and environmental noise [YSL+13]
.

The feature extraction can occur either at the last hidden layer before the classification layer, or at the classification layer itself. In the latter case, it is suggested ([YD14]) to use as targets monophone states in order keep the number of dimensions of the new feature vectors to a low enough level. In general, it is common practice to use a lot less units in the extraction layer, so as to force the network to compute a more compact representation of the salient information in the features, yet still highly discriminative. The extraction is usually followed by a dimensionality reduction algorithm, e.g. PCA, in order to keep only the directions of the highest variability.

The idea of extracting a new representation of the features is similar to the same task using an autoencoder. The difference between that approach and the one described here using a DNN, is that features coming from an autoencoder are often not discriminative, contrary to features coming from a DNN which has been trained to discriminate between phonetic units.

The difficulty of the tandem approach lies in the fact that one cannot know in advance which layer will produce the best features or how many activation units that layer should have; one has to experiment extensively in order to determine the best architecture and extraction layer. At this point it is worth mentioning that in order for the feature extraction to work, firstly a huge amount of training data is needed, and secondly, the test data should not deviate largely from the training set [ESS01].

Comparison of the two approaches

In general, both approaches have equal performance when considered over different tasks. However, the hybrid approach is much easier to implement and train in practice.

The main difference between the two is in the classifier: the hybrid approach uses a log-linear classifier, that is, the softmax layer at the output, whereas the tandem approach uses a GMM, which provides it with the advantage of being able to use numerous existing methods and tools for training a GMM/HMM ASR system.

5.1.2 Training of GMM-HMM

As mentioned, the training of the GMM-HMM used in this project was trained using the Kaldi speech recognition toolkit. The dataset we used comprised the 2000 shortest utterances of the Wall Street Journal corpus (see appendix B).

The GMM-HMM training begins by training a monophone system, that is, a model that uses a context-independent HMM. The training uses traditional 39-

dimensional MFCCs (including deltas and delta-deltas) as input features, which have normalized means and variances. During training, we first build the phonetic decision tree, which, in the monophone case has no splits, and then, using the tree, we compile the Finite State Transducer for each training utterance (*training graph*). The training graph encodes the HMM structure for the utterance it corresponds to; it includes the source and destination HMM state, the input and output symbols and the cost of the transition. In Kaldi's case, output symbols are words and input symbols are *transition-ids*, which roughly correspond to arcs in HMMs (for more details see the description of Kaldi's transition model in the appendix). The training continues by producing a first set of equally spaced alignments, which are refined in the following iterations using the Viterbi algorithm.

For the construction of a triphone condext-dependent system, one could build a model for every possible combination of three phones; that however would make the number of HMM models explode, as for 10 phones for instance, $10\times10\times10$ models would have to be built. Instead, we build a decision tree for each monophone of the already trained monophone system, by asking questions about the left and right context of each monophone. The triphones that had been seen would correspond to the leaves of the tree, and we would build an HMM model for each leaf. In order to accumulate sufficient statistics to train a GMM for each HMM state, we use the previously trained monophone system to acquire alignments for more data. As before, training proceeds by building the decision tree, which now maps from a pair (window of three phones, HMM state) to an integer identifier for a probability distribution function (*pdf-id* in Kaldi terminology). The tree is used to initialize the triphone model and subsequently train it.

5.2 Incorporating the Deep Neural Network

In order to use a deep neural network acoustic model we first have to select the development environment and language both of the DNN and of the initial GMM/HMM, which also determines the decoder. As already mentioned, the GMM/HMM system was trained in Kaldi, which will also be used for the decoding process. As far as the DNN is concerned, the Python programming language including the Theano library were used (see appendix C) for its development and training.

The main issues that needed to be dealt with for the training and decoding were:

- the format of input features
- the classification output format
- the integration of the trained DNN into the Kaldi model for decoding

Table 5.1: PFile format

Sentence	Frame	Feature Vector	Label
0	0	[0.2, 0.3, 0.5, 1.4, 1.8, 2.5]	10
0	1	[1.3, 2.1, 0.3, 0.1, 1.4, 0.9]	179
1	0	[0.3, 0.5, 0.5, 1.4, 0.8, 1.4]	32

5.2.1 Input features

The 39-dimensional MFCCs (including energy, deltas and delta-deltas) were selected as input features. As we have already seen, DNN perform best when each input frame of features is presented in its context, i.e. with neighboring frames. Therefore, and following the advice here [TR14b], we fed the network with a window of 9 frames in total: each frame together with 4 neighboring frames on the left and right.

The 2000 shortest WSJ utterances were split into training (95%, about 1880 utterances) and validation sets (5%, about 120 utterances). Most of the input module was taken from the project *KALDI-PDNN* by Y. Miao ([Mia14] and also see appendix C .

The feature frames are extracted directly from the corresponding Kaldi files, context is added to them and they are saved in *PFile* format. *PFile* ([pfib]) is a binary file format used to store feature frames and their corresponding labels. It was developed in the International Computer Science Institute (ICSI) and was intended mainly for ASR and machine learning tasks. Each file starts with a fixed length ASCII header followed by zero or more variable length binary sections. These sections are divided into *sentences*, each of which contains a sequence of *frames*. Each frame is associated with a feature vector and one or more labels. In the context of ASR applications, sentences and frames correspond to utterances and frames respectively. The frames are indexed within each sentence; however fake indices can be used for both sentences and frames. An example of the format can be seen in table 5.1.

The standard toolkit for handling PFiles is located here [pfia]; however, it is also included in Kaldi's distributions. It contains executables to create, read and print information about a PFile. Using this toolkit, together with an auxiliary Kaldi script we create two PFiles: one for the training and one for the validation set. All features are standardized to have zero mean and unit variance.

5.2.2 Architecture and training

In this section we will present the issues concerned with the training of the deep network. We will present them in the same order as they were introduced in chapter 4.

Training criterion

The training criterion used for the baseline deep network was the cross entropy loss function, due to its faster and more robust convergence. Although derivation of this particular cost function is easy to implement in Python, we did not need to do it manually, as the Theano framework allows for automatic differentiation.

Training algorithm and batch size

The implemented algorithm used for training is the standard back-propagation algorithm. The cost function is minimized using minibatch gradient descent (mini-batch training) to move on the surface defined by it. The batch size remained constant throughout training and various sizes were tried with the most successful appearing to be a size of 256 samples.This is in agreement with the literature on batch size for ASR tasks.

Initialization of weights and Regularization

For the weight initialization we have used the random initialization proposed in [GB10]. Using *PDNN* ([Mia14]) we also tried to initialize the weights with an RBM network however the improvement noticed was not adequate to justify the extra burden of training the RBM. (**CONFIRM???**)

As far as regularization is concerned, the method that resulted in huge improvements in the learning process is dropout.

The widespread use of dropout is also due to its easy implementation: the dropped out neurons have their activation set to zero and consequently no error signal passes through. Therefore, no other change to the training procedure or the network is needed other than randomly selecting the neurons to be dropped out. It should be noted, however, that dropout is used only during training; at test time the average of all possible combinations is used. There are two ways to accomplish this:

- at the end of the training, compensate all weights involved in training by multiplying them by (1-dropout_factor) and use the resulting model as a normal DNN

- during training multiply the input from the previous layer by $\frac{1}{1-r}$, where r is the dropout rate for the previous layer.

In the current project we used the latter implementation thus the activation $\mathbf{y}_t$ of layer t during forward propagation is:

$$\mathbf{y}_t = f(\frac{1}{1-r}\mathbf{y}_{t-1} * \mathbf{mW} + \mathbf{b})$$

where f is the t^{th} layer's activation function, $\mathbf{W}$ is the weight matrix, $\mathbf{b}$ the bias matrix, $*$ denotes element-wise multiplication and $\mathbf{m}$ is a binary mask whose elements are drawn from a Bernoulli(1-r) distribution

$$f(k,(1-r)) = (1-r)^k[1-(1-r)]^{1-k}, k \in 0,1$$

and indicate which neurons are not dropped out ([DSH13]).

On the other hand, regularization using L1 or L2 norms did not provide almost any improvement. However, an implicit weight decay was imposed on the weights by clipping and scaling the columns of the weight matrices as described here [caf].

Learning rate and momentum

Carefully selecting the learning rate is important for the convergence and convergence speed of the training. In out approach, different learning rates were tried and two strategies were used for its decrease: either decreased it by half or exponentially every time the validation error rate rose.

The exponential reduction followed the rule:

$$learning_rate = \frac{learning\_rate\_init}{1 + decay\_factor * epoch}$$

A momentum term was included in the update of the model hoping to improve convergence speed and stabilize the back-propagation algorithm, which it did. The momentum schema implemented is the following:

$$\upsilon_{t+1} = \mu\upsilon_t + (1-\mu)\epsilon\nabla f(\theta_t)$$

$$\theta_{t+1} = \theta_t + \upsilon_{t+1}$$

where ϵ is the learning rate, μ is the momentum term, f is the cost function and θ is the parameter to be updated. In addition, μ gradually increases to a maximum value based on the training epoch.

Network architecture

The network we used for the acoustic modeling is a standard multi-layer perceptron with at least four hidden layers ('Deep Neural Network'). We tried various different layer sizes, yet all hidden layers had the same size; no 'pyramid' shaped networks were tried as they do not improve performance according to the literature. The input layer had 117 dimensions ((4+1+4)×13 features).The output layer was a soft-max layer for classification purposes and its architecture is something that needs closer attention.

The output targets and consequently the dimensions of the classification layer depend on the decoding framework. Most state-of-the-art speech recognition systems use as targets identifiers for the GMMs involved; Kaldi for instance, uses *pdf-id*s (see transition modeling in appendix A) during decoding and therefore the classification targets in our project are *pdf-id*s. This corresponds to a few hundred output units in the monophone case and a few thousand in the triphone. Using *pdf-id*s allows us to directly export a trained DNN model to Kaldi and use it for the forward pass of the test datasets through the network (export module taken from [Mia14]).

A DNN model in Kaldi is described by a text file that contains the following information:

- layer type and input/output dimensions, for instance: *<Sigmoid> 1024 1024*
- weights for each layer

The decoding script ([Mia14]) is based on a DNN recipe in Kaldi (*dnn1*,[kalb]) which, at the time of the project, did not support Rectified Linear units (ReLU). In order to use the decoder with rectifiers, a ReLU component had to be included in Kaldi; the implementation was taken from here [kalc].

5.3 Manifold regularization

why hybrid approach

Introduction

The task of the current project was to implement the ideas described in 3.3.2 onto an LVCSR task using part of the Wall Street Journal corpus, as described in a previous section of the current chapter.

Although the initial paper by Tomar and Rose ([TR14b]) follows the tandem approach, we implemented the hybrid. This was mainly due to the dataset used. For a dataset like WSJ, which contains long utterances and has a large lexicon, the

acoustic model plays a less important role than for a dataset like Aurora, which was used in the original paper and which consists only of utterances containing digits. Therefore, a manifold regularization in the DNN extracting features will have a greater impact on decoding in the case in Aurora but not in WSJ, where the language model prevails to the acoustic.

This was also empirically confirmed, as we tried to extract bottleneck features for a tandem approach, yet decoding efforts failed.

Discovering the manifold structure

Like most manifold learning algorithms, the approach described in [TR14b] and [TR14a] begins by constructing two neighbourhood graphs which will contain the manifold constrained relations between data points.

As already described in 3.3.2, the first graph ($\mathbf{W}_{int}$) is constructed by taking into account only same label neighbours of each point, whereas the second ($\mathbf{W}_{pen}$) contains only neighbours that belong to a different phonetic class.

The labels that are used for the construction of the graphs are of the utmost importance for the success of the discovery of the manifold and the performance of the regularized system. One would assume that the labels would be the targets that the DNN will use during training; however, this is only correct if the decoder uses -and thus the targets are- the physical triphone states of the Hidden Markov Models. In our case, where we used Kaldi's decoder which is based on identifiers of Gaussian distributions (see Kaldi transition modeling in appendix A) and they were the targets for DNN training, the aforementioned assumption does not hold. This is due to the fact that Gaussian identifiers have no physical meaning or 'presence' on the manifold of the phonetic units and there does not exist a one-to-one mapping between triphone states and Gaussian distributions. Therefore, they are unable to help in the discrimination between the phonetic classes.

The alternatives labels we could use were either the phones or the physical HMM states of the triphones. To determine the appropriate label for discrimination we tried the algorithm for the reduction of dimensions as described in [TR14a] and visualized the results. It is obvious that only a small portion of the data was used (2500 frames) which contained samples from a small number of classes.

The feature space before the application of LPDA can be visualized in two dimensions using PCA, as depicted in figure 5.3. There are five phonetic classes in the data subset, however they are overlapping and discrimination between them is difficult.

Using phones as labels during the construction of the affinity matrices we can acquire the projection in two and three dimensions as seen in figures 5.4 and 5.5.

Figure 5.3: *Projection in 2D of 2.5k MFCC and energy feature vectors using PCA.*

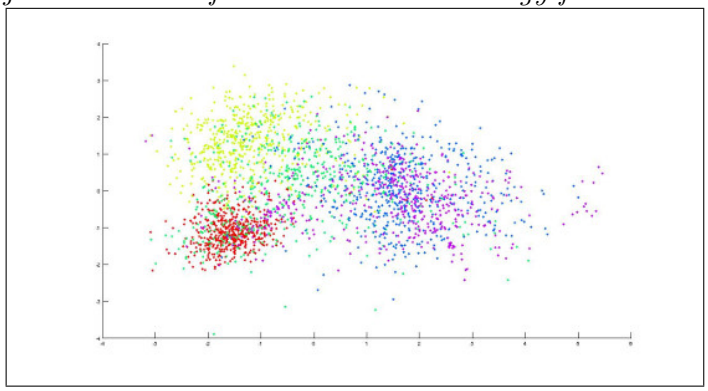
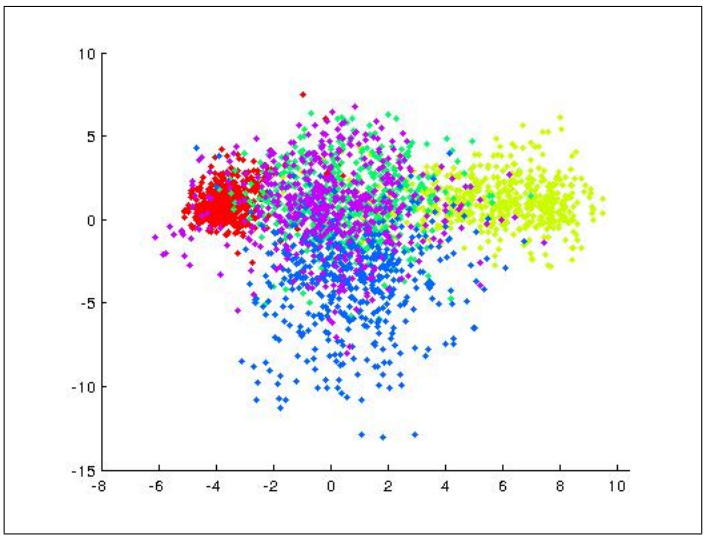


Figure 5.4: *LPDA, 2D projection, kpen=kint=400, Rint=850,Rpen=3000, 2.5k data, 5 phones*



It is evident that the projection using information from the manifold does help to discriminate between classes. Even if some classes overlap in the 2D space, they are separable when projected in three dimensions. Furthermore, as a consequence of the optimization criterion used in the reduction algorithm, which includes the minimization of the within-class scatter measure, the phone clusters formed in the projected space are more compact. Increasing the number of neighbors (see figures 5.6 and 5.7) used to capture the coordinate patches of the manifold, will improve the acquired projection; therefore, a balance has to be found between the computational cost to build the manifold graphs and the discrimination improvement.

If we build the manifold graphs using the HMM states of the triphones as targets, we acquire the visualizations as shown in figures 5.8 - 5.11 . The discrimination is not more evident in the 2D projections, yet in the 3D plots we can clearly see that using (phone-HMM state) pairs as labels, helps to discriminate between phonetic

Figure 5.5: *LPDA, 3D projection, kpen=kint=400, Rint=850,Rpen=3000, 2.5k data, 5 phones*

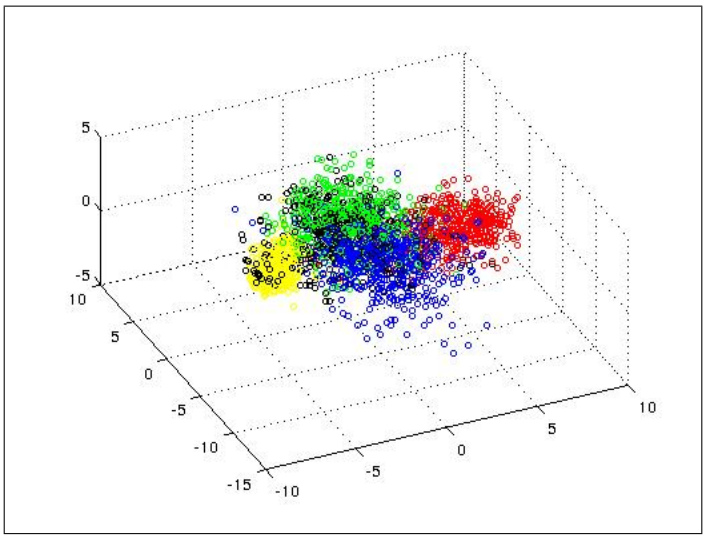


Figure 5.6: *LPDA, 3D projection, kpen=500,kint=600, Rint=850,Rpen=3000, 2.5k data, 5 phones*

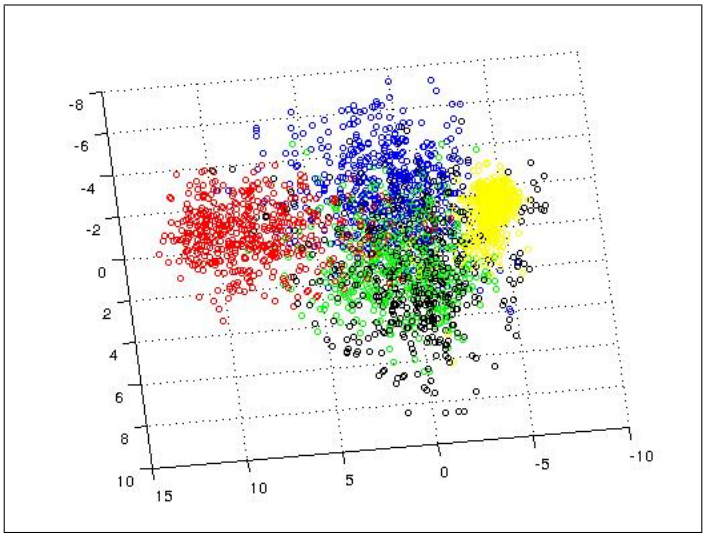
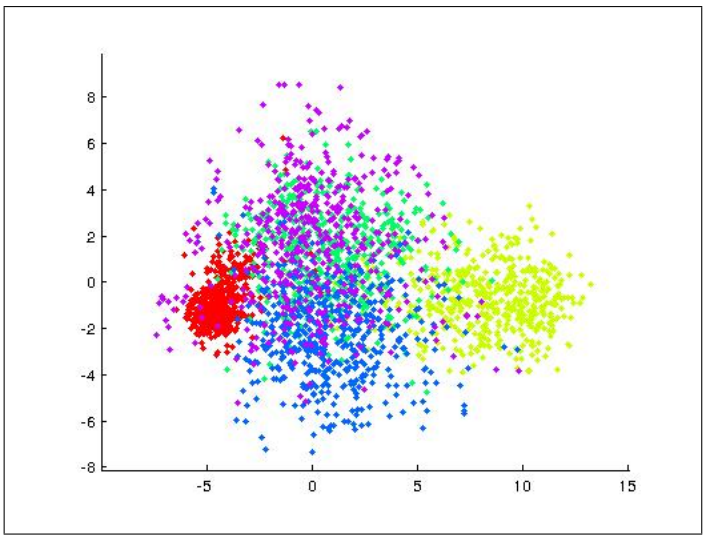


Figure 5.7: *LPDA, 2D projection, kpen=500,kint=600, Rint=850,Rpen=3000, 2.5k data, 5 phones*



classes.

Consequently, just for the graph building, we use (phone - HMM state) pairs as labels, and during DNN training for the hybrid approach, Gaussian identifiers as targets.

For the weights that describe the relations between data points we also used the Locality Preserving Discriminant Analysis and the weight matrices were populated in the following way:

$$w_{ij}^{int} = \begin{cases} e^{\frac{-\|x_i - x_j\|^2}{\rho}} & \text{if } C(x_i) = C(x_j),\ e(x_i, x_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$w_{ij}^{pen} = \begin{cases} e^{\frac{-\|x_i - x_j\|^2}{\rho}} & \text{if } C(x_i) \neq C(x_j),\ e(x_i, x_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

where ρ is the heat kernel scale parameter, $C(x_i)$ refers to the class of vector x_i and $e(x_i, x_j) = 1$ indicates that x_i is in the near neighborhood of x_j.

Constructing the neighborhoods - kd-tree

It is evident that discovering the manifold requires discovering the neighborhoods around each data point that constitute the coordinate patches depicted in figure 3.3. Given a big training set though, determining the nearest neighbors (either exact or approximate) of a data point can become a difficult problem. In the current project we used the m approximate nearest neighbors, i.e. m points that lie within a radius r of the point in question, to build each coordinate patch.

Figure 5.8: *LPDA, 3D projection, kpen=kint = 400, Rint=850,Rpen=3000, 2.5k data, phone-HMMstate label*

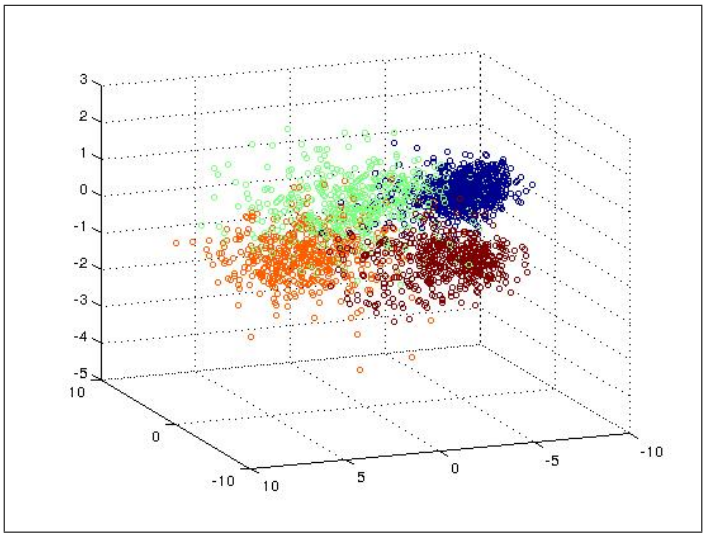


Figure 5.9: *LPDA, 2D projection, kpen=kint = 400, Rint=850,Rpen=3000, 2.5k data, phone-HMMstate label*

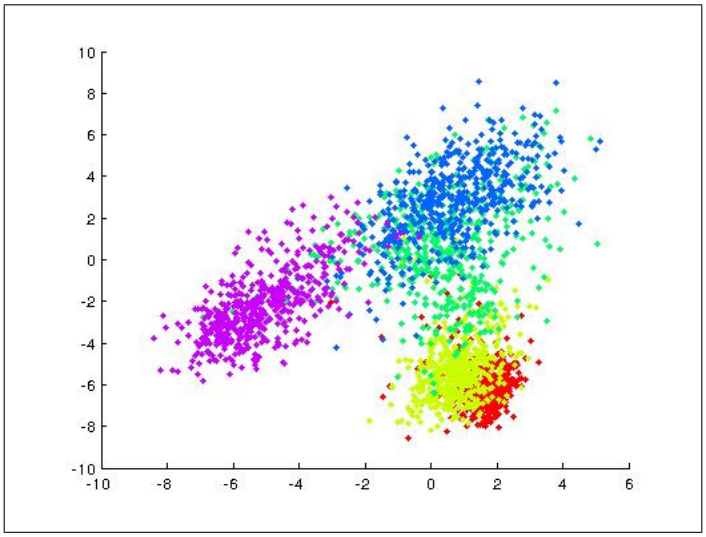


Figure 5.10: *LPDA, 3D projection, kpen=kint = 600, Rint=850,Rpen=3000, 2.5k data, phone-HMMstate label*

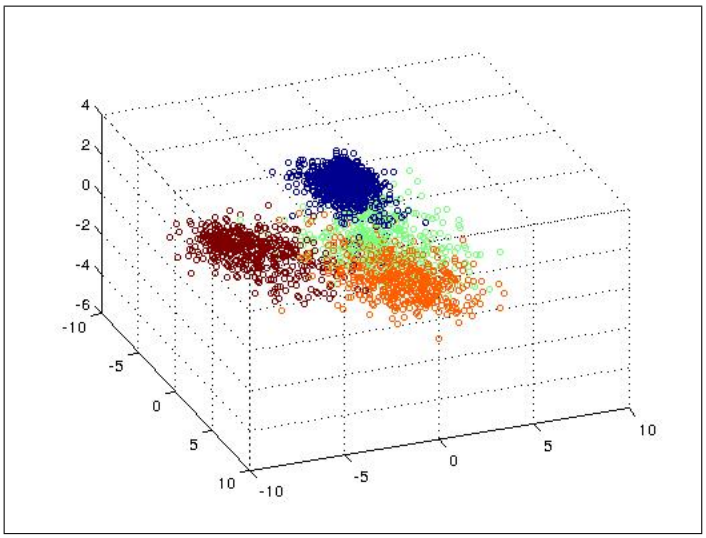
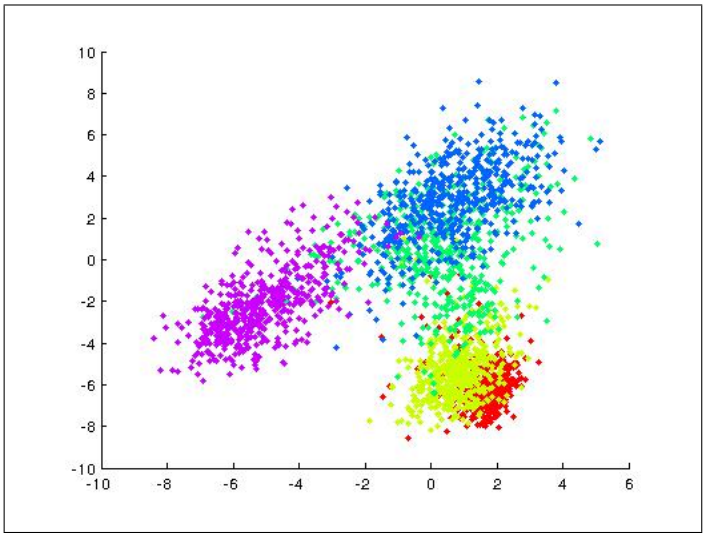


Figure 5.11: *LPDA, 2D projection, kpen=kint = 600, Rint=850,Rpen=3000, 2.5k data, phone-HMMstate label*



To determine the m approximate neighbors we used the kd-tree data structure. A kd-tree stores a finite set of points from a k-dimensional space (in our case, the 13-dimensional space of MFCC) [Ben79]. The basic idea behind kd-trees is to split the input space by alternating between axes, using everytime the plane that is perpendicular to the axis in question and passes through the median of the For instance, in the 2-dimensional case the corresponding tree is constructed as follows:

Algorithm 1 Build 2-d tree

procedure BUILD2DTREE(S, DEPTH)

if *S contains only one point* **then return** leaf containing the point

else if *DEPTH is even* **then**

Split S with a perpendicular to the x-axis line at the median x-coordinate of the points, into two subsets P_1 containing points left or on the line, and P_2 containing points to the right of the line

else

Split S with a perpendicular to the y-axis line at the median y-coordinate of the points into two subsets P_1 containing points left or on the line, and P_2 containing points to the right of the line

n_{left} = Build2DTree(P_1, depth+1)

n_{right} = Build2DTree(P_2, depth+1)

Create node n_{root} storing the splitting coordinate, make n_{left} the left child and n_{right} the right child of n

return n_{root}

If N is the number of points in S, a kd-tree can be built in $\mathcal{O}(N \log N)$ time.

In order to determine the m neighbors of each point that lie within range R of it the following procedure is performed:

where $region(node)$ is the area of the space that contains $node$ and is bounded by the closest splitting planes, e.g. see figure 5.12.

Running a query for m approximate nearest neighbors in a balanced kd-tree takes $\mathcal{O}(n^{1-\frac{1}{k}} + m)$ time.

The programming approach to the construction of the manifold graphs involved a clustering stage to further ease and speed up the process.

The clustering to produce the penalty graph was a simple *k-means*, where the initial cluster centers were randomly chosen from the samples and the number of clusters was manually set. Having acquired the cluster to which each point belongs, one can search for nearest neighbors only within that cluster.

As far as the intrinsic graph is concerned, the clustering was based on the labels of each sample: each cluster contained all same-label samples, among which we

Algorithm 2 Search k-d tree

procedure SEARCHTREE($node$, R)

if *node* is a leaf **then return** point stored at *node* if it lies within ball of radius R

else

if $region(node_{leftChild})$ is fully contained within ball of radius R **then return** $node_{leftChild}$

else if $region(node_{leftChild})$ intersects ball of radius R **then return** SearchTree($node_{leftChild}$, R)

if $region(node_{rightChild})$ is fully contained within ball of radius R **then return** $node_{rightChild}$

else if $region(node_{rightChild})$ intersects ball of radius R **then return** SearchTree($node_{rightChild}$, R)

Figure 5.12: *Region of node v, picture taken from Computational Geometry class in [kdt].*

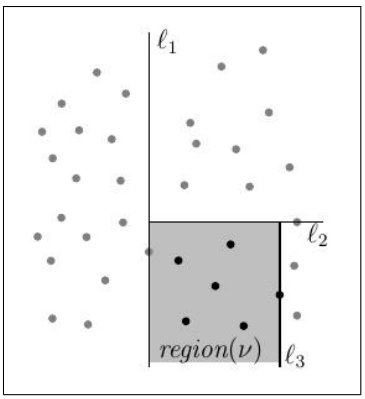
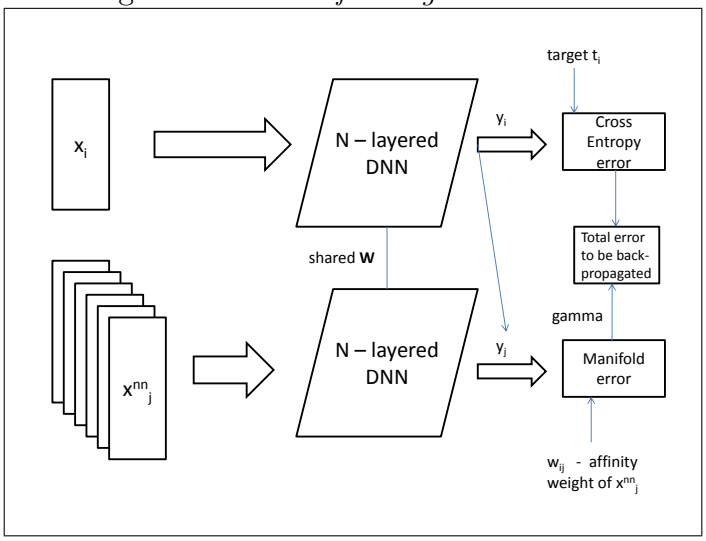


Figure 5.13: *Manifold regularized DNN*



subsequently searched for neighbors for each sample belonging in the cluster.

The construction of both graphs was performed using the *Julia* (see appendix D) programming language with the *Clustering* ([julc]) and *KDTrees* ([julb]) packages.

Architecture of the manifold regularized network

In figure 5.13 we can see the architecture of the manifold regularized neural network.

The first input to the network is a window of frames which is centered around each training frame. A window of 9 frames was used in this project, i.e. the central frame together with four adjacent frames on the left and four on the right. The second input is a neighborhood of frames of each training frame, which describe the manifold structure around it. Given that the second input has to pass through the same network, each frame is passed in a window of the same size as the window of the first input.

This kind of architecture is dictated by the criterion used during the network's training phase:

$$\mathcal{F}(\mathbf{W};\mathbf{Y}) = \frac{1}{N}\sum_{i=1}^{N}\left\{V(\mathbf{x}_i,\mathbf{t}_i,f) + gamma\sum_{j=1}^{2k}\|\mathbf{y}_i-\mathbf{y}_j\|^2 w_{ij}\right\}$$

where V is the training criterion used for the baseline system, i.e. the cross-entropy loss in our case, and the rest is the term that imposes the manifold regularization: *gamma* is the regularization weight, $\mathbf{y}_i$ is the output of the network that corresponds to the input window $\mathbf{x}_i$, $\mathbf{y}_j$ is the output of the network that corresponds to the manifold neighbor $\mathbf{x}_j$,$j = 1, ..., K$ including neighbors in both graphs, and w_{ij} is the

weight that describes the relationship between $\mathbf{x}_i$ and $\mathbf{x}_j$ and is the difference of the corresponding entries in the manifold graphs:

$$w_{ij} = w_{ij}^{int} - w_{ij}^{pen}$$

The gradient of the regularized training function, which will be used to update the network's parameters is then computed as follows:

$$\nabla_{\Theta_{n,m}} \mathcal{F} = \nabla_{\Theta_{n,m}} V + C \sum_{j=1}^{K} w_{ij}(y_{i,m} - y_{j,m}) \left(\frac{\partial y_{i,m}}{\partial \theta_{n,m}} - \frac{\partial y_{j,m}}{\partial \theta_{n,m}} \right)$$

We have already mentioned that the input format to the neural network is the *PFile*. To facilitate the manifold regularization, we had to build a second input file in the same format, which contained as 'features' the k nearest neighbors (from both graphs) in their 9-frame window and as label the weight w_{ij} as defined above. Since the weight term is of type float, the standard *PFile* toolkit ([pfia]) had to be modified to support float labels and not just integers as it originally had.

One of the most challenging parts of the project was how to pass the manifold-required data of each minibatch *and* the minibatch through the network and in such a way so as to allow computation using the GPU. In order to do this, we had to set up the data-loading module in such a way so as to load onto the GPU the manifold data required just for the current minibatch and load the rest onto the CPU memory. Consequently, this caused a loss in execution speed but it allowed us to perform the manifold regularization which would have been impossible otherwise.

5.4 Decoding and experimental results

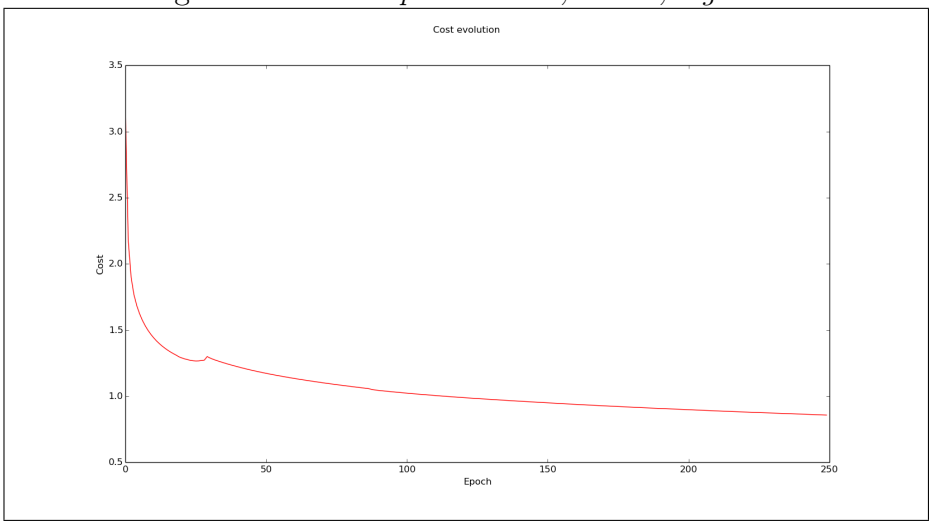
cost-valid plots, observations/conclusion

In this section we will present the results of the performed experiments and summarize the observations made. We will begin with the evaluations using the unregularized network and then proceed to the manifold regularized system.

The baseline for the comparison is Kaldi's GMM/HMM system, trained on the 2000 shortest utterances of the WSJ corpus and evaluated on the two accompanying datasets: *dev93* and *eval92*.

The decoding accuracy achieved by the triphone system is 76.15% on *dev93* and 82.62% on *eval92*, whereas the monophone system achieved correspondingly 64.87% and 74.45% on the two test sets.

Figure 5.14: *Monophone DNN, 5x600, sigmoid*



Monophone DNN

- A deep MLP consisting of 5 layers with 600 sigmoid units each was trained for 250 epochs, with the initial learning rate set at 0.06. The learning rate was decaying exponentially at the end of each epoch and the training strategy was minibatch gradient descent (GD) with batch size 256 samples and momentum starting at 0.8 and gradually increasing to 0.99. The cost and validation plots are shown in figures 5.14 and 5.15. The decoding accuracy on dev93 dataset was 69.6% and on eval92 78.84%.

- A deep MLP consisting of 4 layers with 1024 sigmoid units each was trained for 100 epochs, with initial learning rate set at 0.08. The learning rate was decaying exponentially at the end of each epoch and the training strategy was minibatch GD with batch size 256 samples and momentum starting at 0.8 and gradually increasing to 0.99. The cost and validation plots are shown in figures 5.16 and 5.17. The decoding accuracy on dev93 dataset was 69.34% and on eval92 79.83%.

- A deep MLP consisting of 4 layers with 1024 hyperbolic tangent (tanh) units each was trained for 20 epochs, with initial learning rate set at 0.08. The learning rate was decaying exponentially at the end of each epoch and the training strategy was minibatch GD with batch size 256 samples and momentum starting at 0.8 and gradually increasing to 0.99. The cost and validation plots are shown in figures 5.18 and 5.19. The decoding accuracy on dev93 dataset was 67.97% and on eval92 77.30%.

- A deep MLP consisting of 4 layers with 1024 rectified linear units each was

Figure 5.15: *Monophone DNN, 5x600, sigmoid*

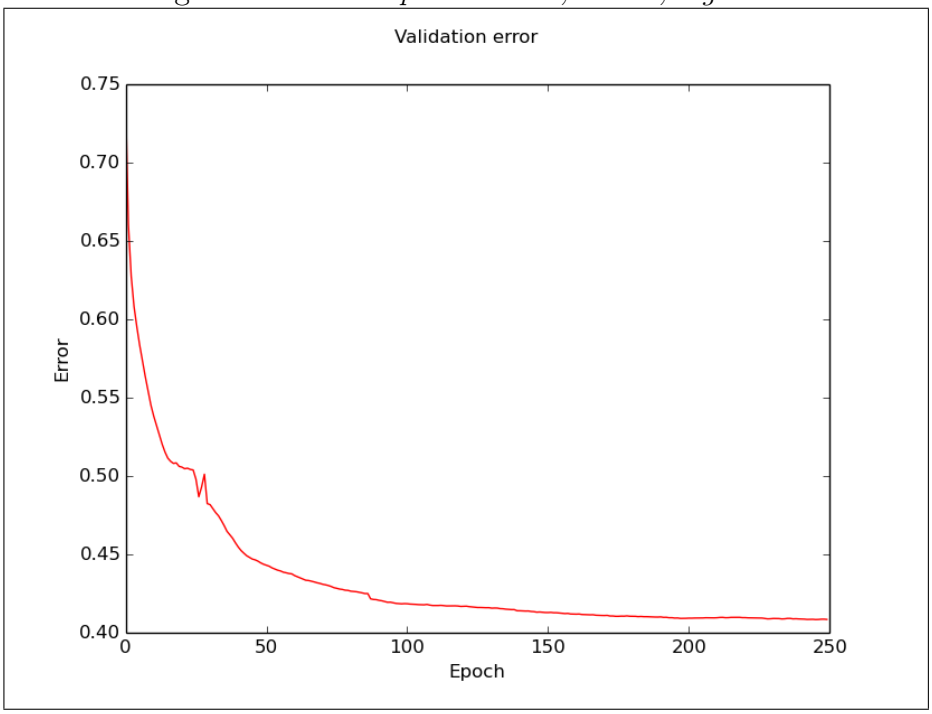


Figure 5.16: *Monophone DNN, 4x1024, sigmoid*

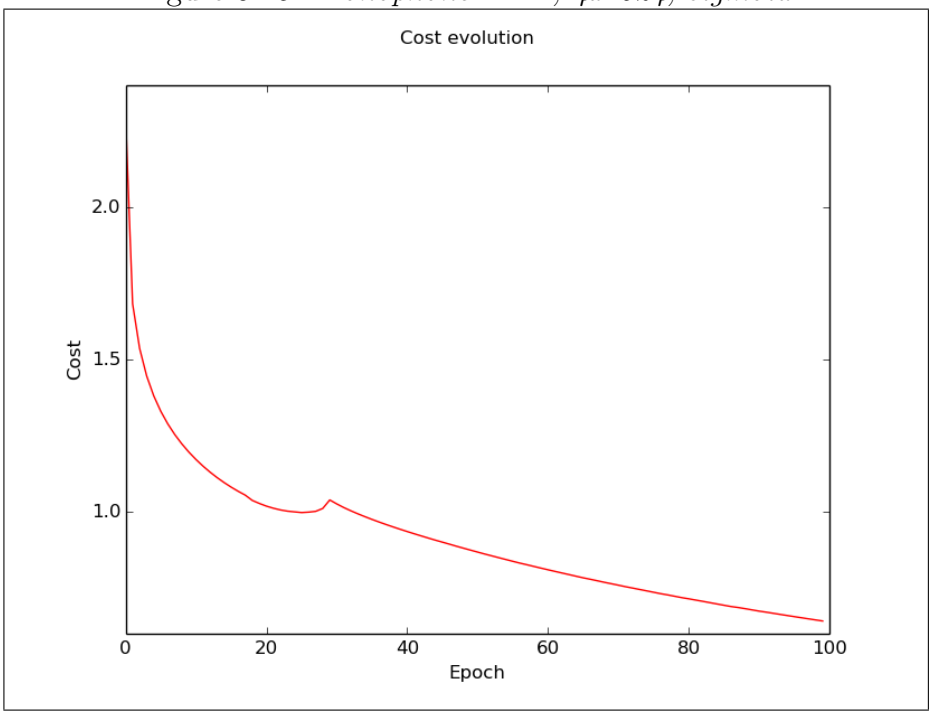


Figure 5.17: *Monophone DNN, 4x1024, sigmoid*

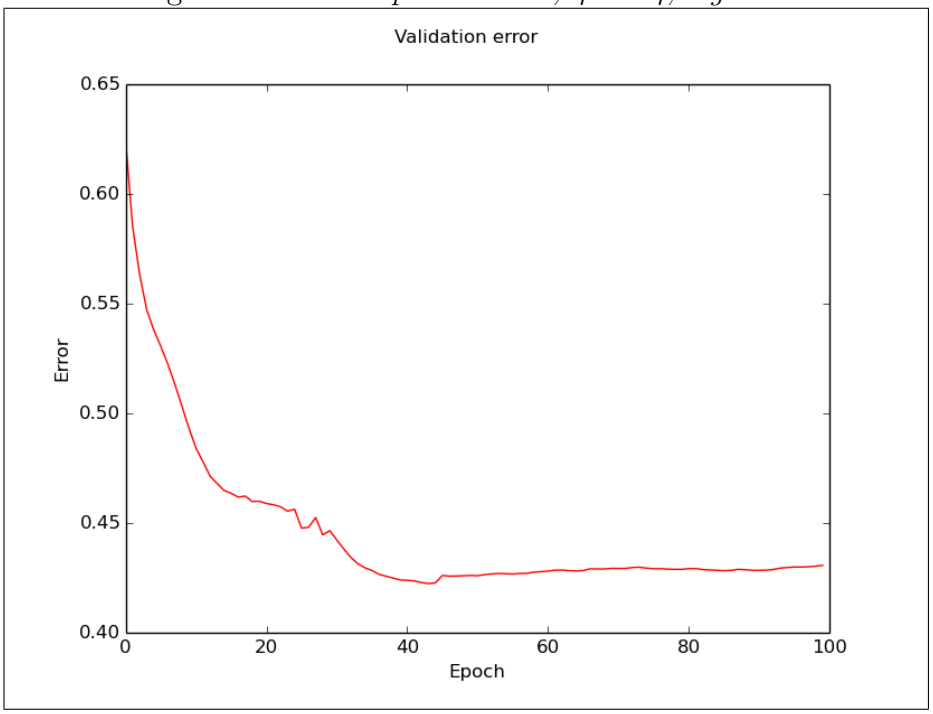


Figure 5.18: *Monophone DNN, 4x1024, tanh*

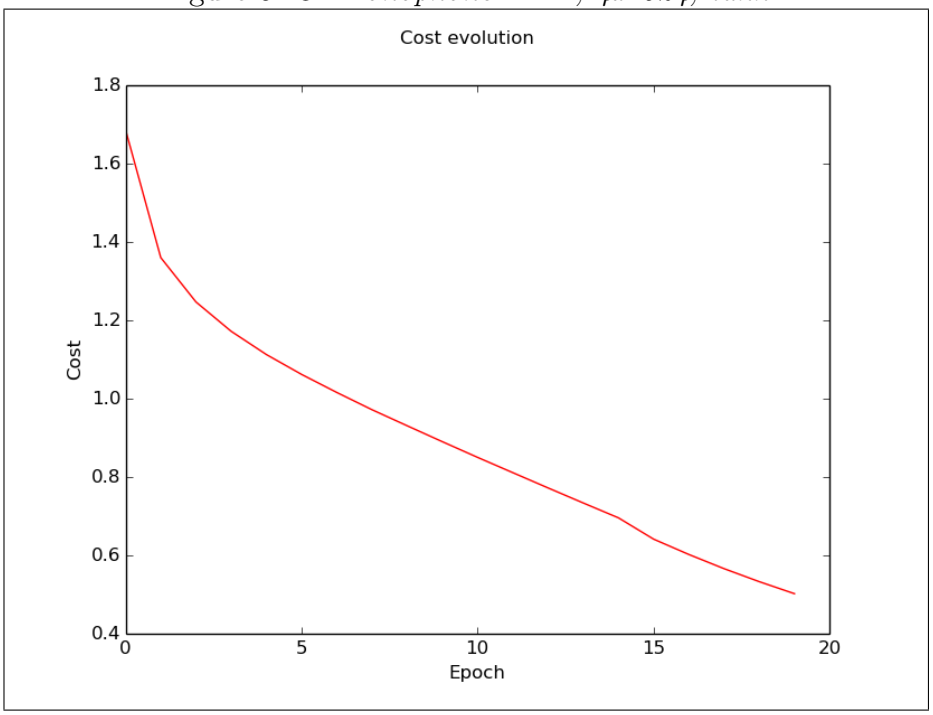
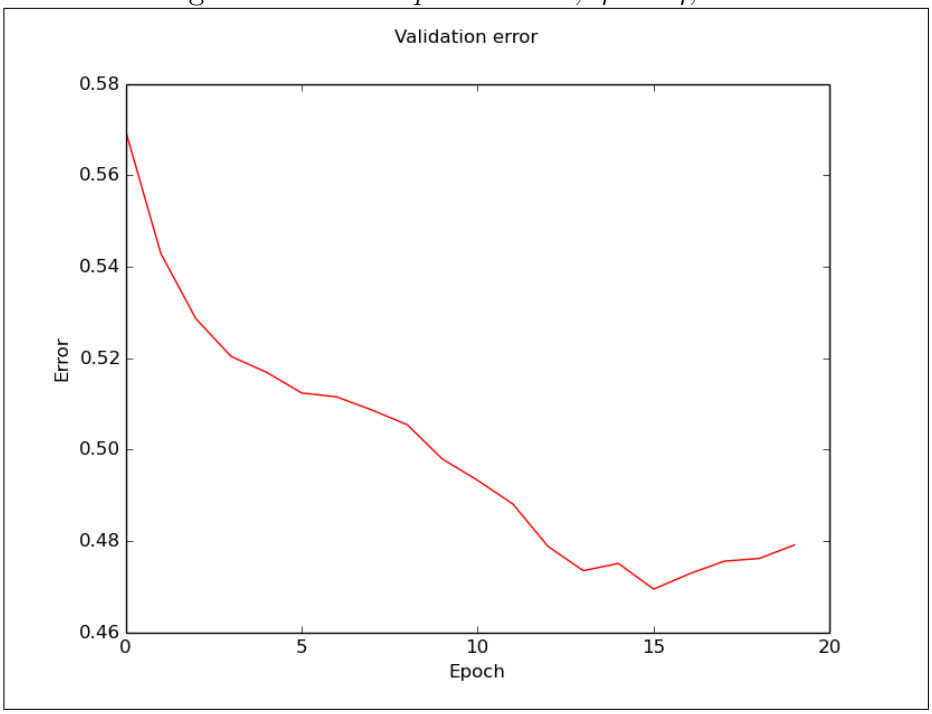


Figure 5.19: *Monophone DNN, 4x1024, tanh*



trained for 30 epochs, with initial learning rate set at 0.08. The learning rate was halved every time the validation error rose and the training strategy was minibatch GD with batch size 256 samples and momentum starting at 0.8 and gradually increasing to 0.99. Dropout was used as a regularization method. The dropout rate was set at 0.4 for the hidden layers and there was no dropout in the input layer. The cost and validation plots are shown in figures 5.20 and 5.21. The decoding accuracy on dev93 dataset was 71.1% and on eval92 80.99%.

Triphone DNN

- A deep MLP consisting of 6 layers with 2048 ReLU units each was trained for 80 epochs, with initial learning rate set at 0.07. The learning rate was decaying exponentially every time the validation error increased and the training strategy was minibatch GD with batch size 256 samples and momentum starting at 0.9 and gradually increasing to 0.99. Dropout was used as a regularization method. The dropout rate was set at 0.4 for the hidden layers and there was no dropout in the input layer.The cost and validation plots are shown in figures 5.22 and 5.23. The decoding accuracy on dev93 dataset was 81.56% and on eval92 88.64%.

Figure 5.20: *Monophone DNN, 4x1024, ReLU*

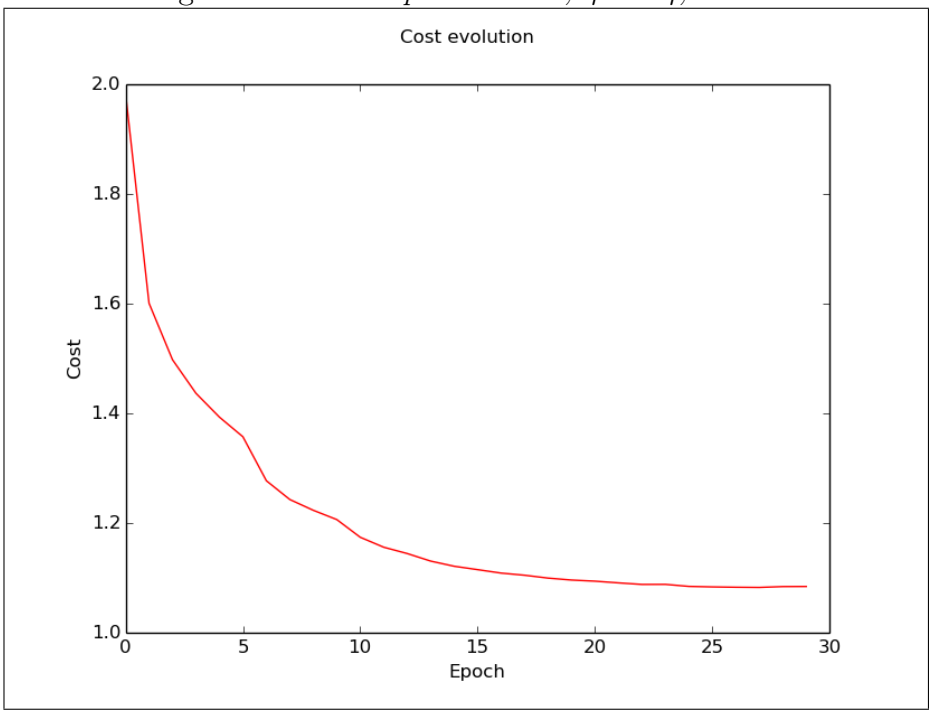


Figure 5.21: *Monophone DNN, 4x1024, ReLU*

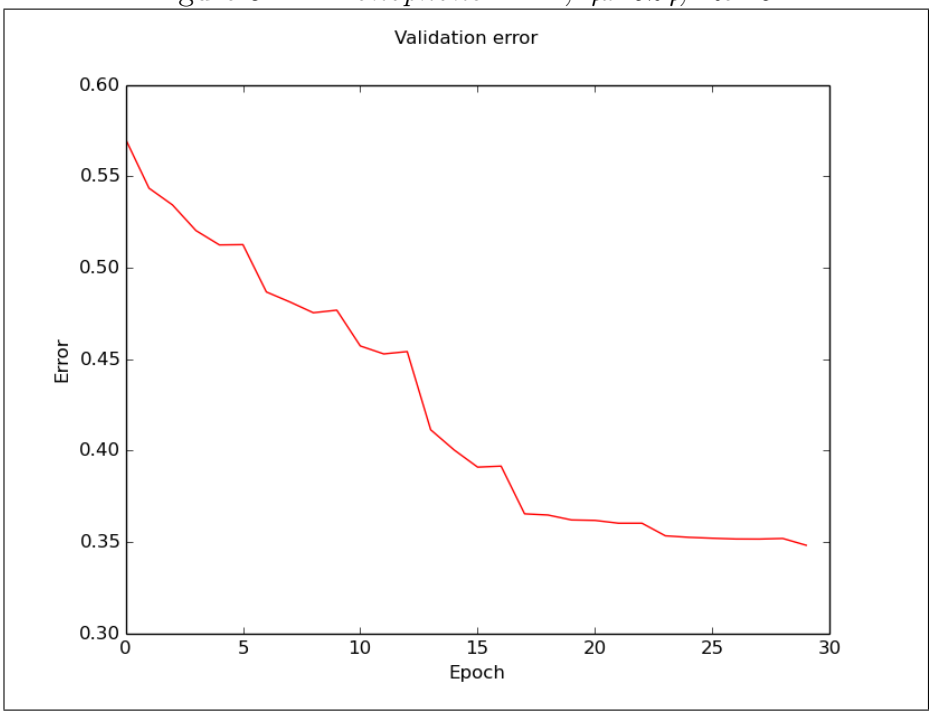


Figure 5.22: *Triphone DNN, 6x2048, ReLU*

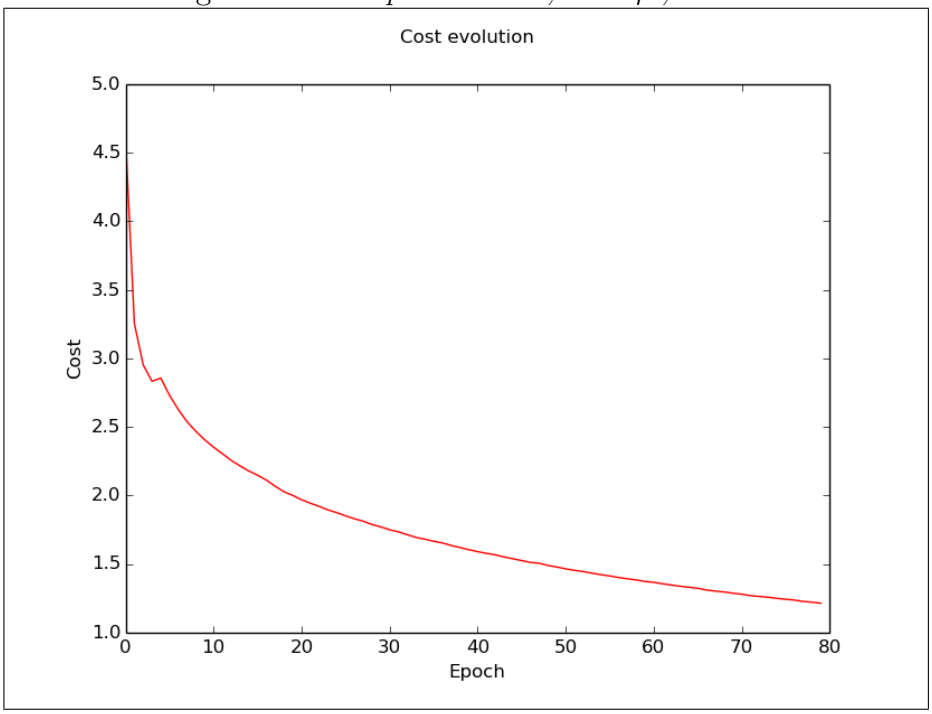


Figure 5.23: *Triphone DNN, 6x2048, ReLU*

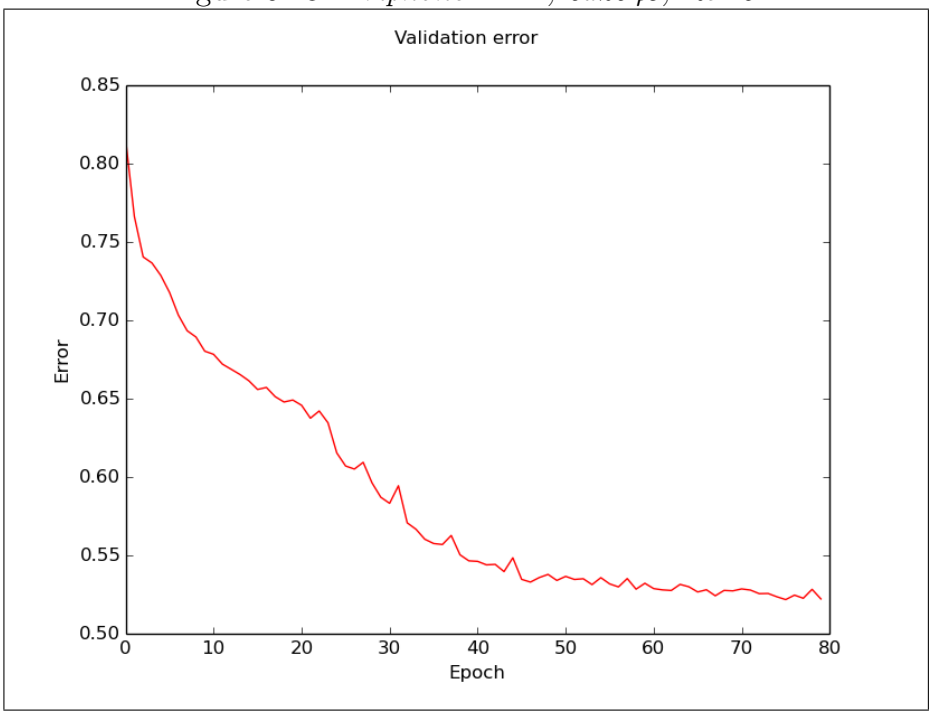
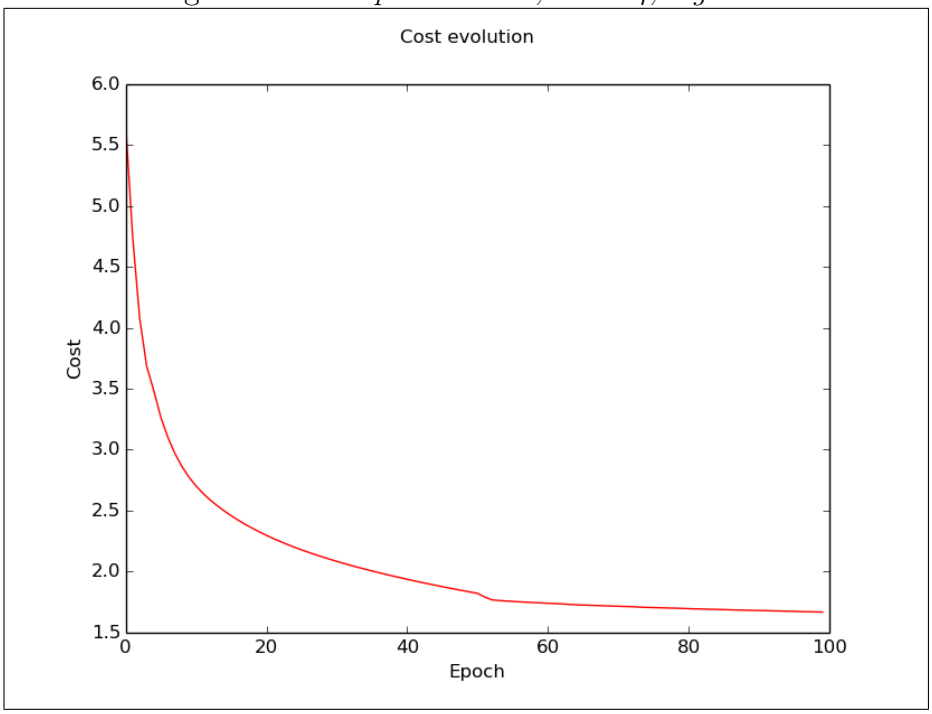


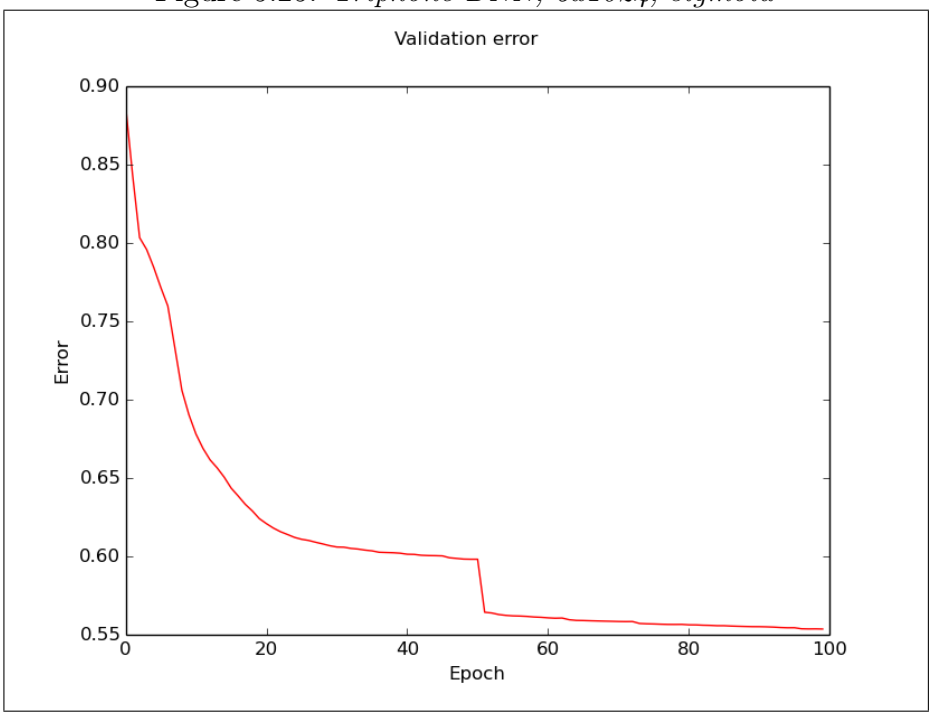
Figure 5.24: *Triphone DNN, 5x1024, sigmoid*



- A deep MLP consisting of 5 layers with 1024 sigmoid units each was trained for 100 epochs, with initial learning rate set at 0.05. The learning rate was decaying exponentially every time the validation error increased and the training strategy was minibatch GD with batch size 256 samples and momentum starting at 0.9 and gradually increasing to 0.99. The cost and validation plots are shown in figures 5.24 and 5.25. The decoding accuracy on dev93 dataset was 78.43% and on eval92 85.11%.

  The trained system set the initialization point in the weights space for the manifold-regularized DNN.

Figure 5.25: *Triphone DNN, 5x1024, sigmoid*



Appendix A

Kaldi Speech Recognition Toolkit

[PGB+11]

Kaldi is an open source speech recognition toolkit implementing state-of-the-art algorithms for feature extraction, acoustic modeling and decoding. It originated in Johns Hopkins University in 2009, and its name comes from the Ethiopian goatherder who discovered the coffee plant.

Its codebase is written entirely in C++ and the corresponding executables are invoked from bash scripts called "recipes" which depend on the dataset they process.

The main benefits of Kaldi are:

- Code-level integration with Finite State Transducers, since it compiles against the OpenFst library
- Extensive matrix library using BLAS and LAPACK
- Generically developed modules to easily support extensions
- Open license (Apache v2.0)
- Complete recipes that span from feature extraction to decoding

Installation[leca]

In order to install Kaldi you need to clone the repository where it resides:

"git clone https://github.com/kaldi-asr/kaldi.git kaldi-trunk –origin golden"

The INSTALL file contains all the required information about the installation process.

Getting started

Source code and executables are organized according to the purpose they serve (decoding, feature extraction, neural network configuration etc.) and are placed in

kaldi-trunk/src. Invoking a script without arguments will print instructions on how to run it and what arguments it actually needs.

Example recipes for well-known datasets are inside kaldi-trunk/egs. The latest recipe for each dataset is inside the s5 folder. For example, the latest recipe for the Wall Street Journal dataset is kaldi-trunk/egs/wsj/s5/run.sh and it is executed by running the script *run.sh* while being inside that directory.

Going though the WSJ recipe

To give an idea of Kaldi's recipes, we will focus on the Wall Street Journal recipe for the rest of the chapter, referring to the *run.sh* script. Apart from it, inside *kaldi-trunk/egs/wsj/s5* are folders (*local*, *steps*, *utils*) containing dataset-dependent scripts.

- Data preparation [kala]
  - The script begins by setting the variables which determine the dataset location (*wsj0,wsj1*).
  - Invoke dataset dependent script (*wsj_data_prep.sh*) to make sure all data needed are available and to build auxiliary scripts for the training steps (e.g. spk2utt, utt2spk scripts). It also extracts the language model which (unless we want to use a different one) is distributed with the WSJ dataset.
  - Proceed to create the dictionary (*wsj_prepare_dict.sh*) and the language directory, which contains extra information regarding the language model (*prepare_lang.sh*).
  - Reorganize the data directory to facilitate the next steps (*wsj_format_data.sh*).
- Feature extraction
  - Extract MFCC features (*steps/make_mfcc.sh*). Make sure that the data are where the script expects them to be and in the correct form (case-sensitive to file names).
- Split the dataset into smaller chunks which will be used for training various systems (*utils/subset_data_dir.sh*).
- Training
  - The usual procedure to evaluate an ASR system starts with system training (e.g. *train_mono.sh*), proceeds to build the decoding graph HCLG

Table A.1: Kaldi ASR accuracy on WSJ

GMM/HMM system	test_eval92	test_dev93
monophone	25.55 %	35.13 %
delta + delta-delta	11.84 %	18.27 %
LDA + MLLT	10.81 %	16.89 %
LDA + MLLT + SAT	8.77 %	14.63 %
LDA + MLLT + SAT (more data)	7.73 %	12.58 %

utterance-id	[frame1 features
	frame2 features
	. . .]

Table A.2: Kaldi table format

(*utils/mkgraph.sh*) and ends with system evaluation (*steps/decode.sh*) [lecd]. The final result of the evaluation is inside exp/decode.../scoring_-kaldi/best_wer.

– The system just created can be used to extract the observations-phones alignment, which will be used to initiate the training of e.g. a triphone model [lecb],[lecc].

The decoding accuracy on WSJ for various systems is summarized on the following table:

Useful scripts

Kaldi provides various executables that can prove useful not only for ASR system training but also for debugging/testing purpose. In particular, inside kaldi-trunk/src/bin:

- *ali-to-phones* : converts alignments from Kaldi raw format ("transition-ids") to phones. To view the alignments use *show-alignments*.

- *copy-matrix* (or *featbin/copy-feats*): copies a Kaldi matrix. During copying we can save the matrix in text format. The read and write specifiers that are referred to in the scripts are "ark" ,"t" and "scp" for binary ,text and scp (readable by text editors) scripts.

  In general, Kaldi processes scp scripts (ending in .scp) and binary data. Kaldi matrices have the form seen in the table and are saved into binary format (.ark files).

Kaldi transition modeling

The basic transition model is as follows. Each phone has a HMM topology and each HMM-state of each of these phones has a number of transitions out of it. Each HMM-state has an associated 'pdf_class' which gets replaced with an integer identifier, a *pdf-id*, via the tree. The transition model associates the transition probabilities with a triple (phone, HMM-state, pdf-id). We associate with each such triple a transition-state. Each transition-state has a number of associated probabilities to estimate.Each probability has an associated transition-index. We associate with each (transition-state, transition-index) a unique *transition-id*, which is the output of the alignments. Each individual probability estimated by the transition-model is associated with a transition-id. A list of the terms found in Kaldi's transition model is:

- *pdf-id*: a number output by the Compute function of ContextDependency (it indexes probability distribution functions). Zero-based.
- *transition-state*: the states for which we estimate transition probabilities for transitions out of them. In some topologies, will map one-to-one with pdf-ids. One-based, since it appears on FSTs.
- *transition-index*: identifier of a transition (or final-prob) in the HMM. Zero-based.
- *transition-id*: identifier of a unique parameter of the TransitionModel. Associated with a (transition-state, transition-index) pair. One-based, since it appears on FSTs.

List of the supported mappings in Kaldi:

- (phone, HMM-state, pdf-id) -> transition-state
- (transition-state, transition-index) -> transition-id
- transition-id -> transition-state
- transition-id -> transition-index
- transition-state -> phone
- transition-state -> HMM-state
- transition-state -> pdf-id

Appendix B

Wall Street Journal corpus

The Wall Street Journal corpus is a large dataset of sentences of North American English collected from the 1987-89 editions of the corresponding newspaper. The dataset is divided in two subsets, WSJ0 and WSJ1, and it was collected to support the development and evaluation of large vocabulary, speaker-independent, continuous speech recognition systems.

WSJ0 [wsjb]

The Wall Street Journal Phase I (CSR-WSJ0) corpus was designed by the DARPA Corpus Coordinating Committee and was collected in 1991 at the Massachusetts Institute of Technology Laboratory for Computer Science, SRI International, and Texas Instruments (TI) in late 1991.

The test material in WSJ0 contains 5.000-word and 20.000-word WSJ vocabulary read tests, as well as tests using spontaneous dictation. Each set of test material is segmented into utterances and each utterance was recorded with two microphones, a Sennheiser close-talking microphone and a secondary microphone of varying type.

So that storage requirements are minimized, the waveforms have been compressed using the SPHERE-embedded 'Shorten' compression algorithm which was developed at Cambridge University, and the storage requirements for the corpora have been approximately halved.

WSJ1 [wsja]

WSJ1 was collected in 1992 and contains about 78.000 training utterances (73 hours of speech), and about 8.200 (5.000-word and 20.000-word vocabulary) development test utterances (8 hours of speech). Like WSJ0, the training portion of the corpus was recorded using 2 microphones: a Sennheiser close-talking head-mounted microphone, and a secondary microphone of varying types, and the waveforms are stored using the 'Shorten' compression algorithm.

Appendix C

Python-Theano

C.1 Python

Python is a popular high-level, general-purpose, programming language that was developed in the 90s in Netherlands by Guido van Rossum.

It is interpreted and its strict syntax allows for clear, easy-to-read programs. Python is a multi-paradigm programming language: it allows for object-oriented programming, structured programming, and some of its modules allow for functional programming ideas to be used. This versatility, simplicity of syntax and large collection of available libraries are the reason why it is so popular among software developers in many different fields.

In this project we used Python version 2.7.6.

C.2 Theano

[BLP+12],[BBB+10]

Theano is a Python library intended for optimization and evaluation of mathematical expressions involving arrays. These expressions can also contain symbolic variables which makes Theano indispensable for neural network programming in Python, as it makes it easy for the programmer to define the training criterion and perform the necessary calculations.

Key advantages of Theano include:

- easy integration with *NumPy*, another Python library for mathematical operations and arrays
- transparent use of a Graphics Processing Unit (GPU); Theano will automatically detect a GPU - if available - and use it to speed up calculations (use CUDA/OpenCL), otherwise it will fall back onto the CPU

- symbolic differentiation; one has to simply express a function using symbolic variables, state with respect to which variable the derivative will be taken, and Theano will compute it
- numerical stability optimizations
- dynamic C code generation, which improves performance
- many available tools for unit-testing and self-verification

Theano tutorials and material relevant to its machine learning applications can be found here [the].

C.2.1 Exploiting the GPU

When developing Python programs using Theano, one has to consider the following, as to what can be optimized on the GPU:

- Computations with *float32* data type; support for *float64* is expected
- Matrix operations such as multiplication, convolution, and element-wise operations. However, the speed-up is much greater (5-50x) if the arguments are large enough to make full use of all GPU cores

On the other hand, indexing, dimension shuffling and constant-time reshaping will be equally fast on GPU as on CPU, whereas summing over rows/columns of tensors (symbolic arrays) can be slower on the GPU than on the CPU.

Care should also be taken when it comes to moving data to and from the GPU, as is can be quite slow and thus cancel most of the benefits of GPU computations.

To ensure that the GPU will be used for computations, one should firstly include *floatX=float32* in the configuration file of Theano, i.e. *.theanorc*. Then, whenever a float type variable should be declared in a script, the expression *theano.config.floatX* should be used instead. Secondly, one should ensure that all output variables have *float32* type. Setting the *floatX* entry in *.theanorc* is a good way to make sure that all float variables that will be processed by the GPU are of the correct type. Finally, there are many more flags that will prove useful for profiling the program's execution, such as *profile=True* or *assert_no_cpu_op*.

C.3 Kaldi-PDNN

[Mia14]

Python DNN (PDNN) is a deep learning toolkit developed in Python. It contains many different neural network architectures (DNN,CNN, RBM) as well as a number of supporting tools for reading and storing data and exporting model parameters. In the current project, the input module for reading PFiles, as well as the output module for saving Kaldi DNN models were based on PDNN.

Kaldi-PDNN constructs state-of-the-art DNN acoustic models using PDNN and builds complete ASR systems by combining the DNN model with Kaldi.

The general process has three steps:

- Initial GMM models are built with Kaldi standard recipes
- Acoustic models using a deep architecture (DNN/CNN) are trained using PDNN
- Trained models are imported into Kaldi for direct decoding or building a tandem system using DNN-extracted features

Appendix D

Julia

[jula]

Julia is an open-source, high-level programming language for technical computing, developed in MIT. Its design and just-in-time compiler allow it to match or outperform other common programming languages for scientific and numerical computing.

Julia features an active community contributing to a huge number of external packages, as well as the following key benefits:

- multiple dispatch, i.e. allows the programmer to define a function's behavior across many combinations of argument types
- Easy integration and call of Python and C functions
- Strong shell-like capabilities for handling other processes
- Support of parallel and distributed computation
- Ability to combine low- and high-level programming in the same script

Bibliography

[AE] John McKenna Andrew Errity. An investigation of manifold learning for speech analysis.

[BBB+10] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[BCV13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, August 2013.

[BDVJ03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[Bel03] Richard Ernest Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[Ben79] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng.*, 5(4):333–340, July 1979.

[Ben09] Yoshua Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009.

[Ben12] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012.

[BGC15] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015.

[Bil03] Jeff A. Bilmes. Buried markov models: a graphical-modeling approach to automatic speech recognition. *Computer Speech and Language*,

17(2.3):213 – 231, 2003. New Computational Paradigms for Acoustic Modeling in Speech Recognition.

[BLP+12] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[Bod01] Mikael Boden. A guide to recurrent neural networks and backpropagation, 2001.

[BPV03] Yoshua Bengio, Jean-Francois Paiement, and Pascal Vincent. Out-of-sample extensions for lle, isomap, mds, eigenmaps, and spectral clustering. In *In Advances in Neural Information Processing Systems*, pages 177–184. MIT Press, 2003.

[caf] https://github.com/bvlc/caffe/issues/109.

[Cay05] L. Cayton. Algorithms for manifold learning. *Univ. of California at San Diego Tech. Rep*, 2005.

[DG03] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Struct. Algorithms*, 22(1):60–65, January 2003.

[DHS00] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

[DSH13] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *ICASSP*, pages 8609–8613, 2013.

[DY14] Li Deng and Dong Yu. Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7(3–4):197–387, June 2014.

[ESS01] Daniel P. W. Ellis, Rita Singh, and Sunil Sivadas. Tandem acoustic modeling in large-vocabulary recognition. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2001, 7-11 May, 2001, Salt Palace Convention Center, Salt Lake City, Utah, USA, Proceedings*, pages 517–520, 2001.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[Hin02] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8):1771–1800, August 2002.

[Hin12] Geoffrey E. Hinton. A practical guide to training restricted boltzmann machines. In Gregoire Montavon, Genevieve B. Orr, and Klaus-Robert MG'Oller, editors, *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 599–619. Springer, 2012.

[HS52] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–436, 1952.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[JM09] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[JN05] A. Jansen and P. Niyogi. A geometric perspective on speech sounds. Technical report, 2005.

[jula] http://julialang.org/.

[julb] https://github.com/juliageometry/kdtrees.jl.

[julc] https://github.com/juliastats/clustering.jl.

[kala] http://kaldi-asr.org/.

[kalb] http://kaldi.sourceforge.net/dnn1.html.

[kalc] https://github.com/naxingyu/kaldi-nn/tree/master/src/nnet.

[kdt] https://www.cise.ufl.edu.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[Kre78] E. Kreyszig. *Introductory Functional Analysis With Applications*. Wiley Classics Library. John Wiley & Sons, 1978.

[LBBH98] Yann Lecun, LG•on Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[LBOM98] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Effiicient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag.

[leca] http://www.danielpovey.com/files/lecture1.pdf.

[lecb] http://www.danielpovey.com/files/lecture2.pdf.

[lecc] http://www.danielpovey.com/files/lecture3.pdf.

[lecd] http://www.danielpovey.com/files/lecture4.pdf.

[LN89] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Math. Program.*, 45(3):503–528, December 1989.

[Mah09] Michael Mahoney. Algorithms for massive datasets analysis, lecture1 notes, 2009.

[MB90] Nelson Morgan and Herve Bourlard. Continuous speech recognition using multilayer perceptrons with hidden Markov models, 1990.

[Mia14] Yajie Miao. Kaldi+pdnn: Building dnn-based ASR systems with kaldi and PDNN. *CoRR*, abs/1401.6984, 2014.

[MKB+10] Tomas Mikolov, Martin Karafiat, Lukas Burget, Jan Cernocky, and Sanjeev Khudanpur. Recurrent neural network based language model. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH*, pages 1045–1048. ISCA, 2010.

[MPR01] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition, 2001.

[Nie15] Michael Nielsen. Neural Networks and deep learning,. , 2015.

[pfia] https://code.google.com/archive/p/pfile-utilities/.

[pfib] https://martin-thoma.com/what-are-pfiles/.

[PGB+11] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The kaldi speech recognition toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, December 2011. IEEE Catalog No.: CFP11SRW-USB.

[PHB+12] Daniel Povey, Mirko Hannemann, Gilles Boulianne, Lukas Burget, Arnab Ghoshal, Milos Janda, Martin Karafiat, Stefan Kombrink, Petr Motlicek, Yanmin Qian, Korbinian Riedhammer, Karel Vesely, and Ngoc Thang Vu. Generating exact lattices in the wfst framework. In *ICASSP*, pages 4213–4216. IEEE, 2012.

[Rab89] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *PROCEEDINGS OF THE IEEE*, pages 257–286, 1989.

[RJ86] L. Rabiner and B. Juang. An introduction to hidden markov models. *IEEE Acoutics, Speech and Signal Processing Magazine*, 3:4–16, 1986.

[rMDH] Abdel rahman Mohamed, George Dahl, and Geoffrey Hinton. Deep belief networks for phone recognition.

[SKK+07] Thomas Serre, Gabriel Kreiman, Minjoon Kouh, Charles Cadieu, Ulf Knoblich, and Tomaso Poggio. A quantitative theory of immediate visual recognition. *PROG BRAIN RES*, pages 33–56, 2007.

[SKrMR13] Tara N. Sainath, Brian Kingsbury, Abdel rahman Mohamed, and Bhuvana Ramabhadran. Learning filter banks within a deep neural network framework. In *ASRU*, pages 297–302. IEEE, 2013.

[SLY] Frank Seide, Gang Li, and Dong Yu. Conversational speech transcription using context-dependent deep neural networks. In *in Proc. Interspeech 2011*, pages 437–440.

[SR03] Lawrence K. Saul and Sam T. Roweis. Think globally, fit locally: Unsupervised learning of low dimensional manifolds. *J. Mach. Learn. Res.*, 4:119–155, December 2003.

[SS05] Fei Sha and Lawrence K. Saul. Analysis and extension of spectral methods for nonlinear dimensionality reduction. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 784–791, New York, NY, USA, 2005. ACM.

[ST04] Darius Silingas and Laimutis Telksnys. Specifics of hidden markov model modifications for large vocabulary continuous speech recognition. *Informatica, Lith. Acad. Sci.*, 15(1):93–110, 2004.

[SVN37] S. S. Stevens, J. Volkmann, and E. B. Newman. A scale for the measurement of the psychological magnitude pitch. *The Journal of the Acoustical Society of America*, 8(3):185–190, 1937.

[TGSN14] Zoltán Tüske, Pavel Golik, Ralf Schlüter, and Hermann Ney. Acoustic modeling with deep neural networks using raw time signal for LVCSR. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, pages 890–894, 2014.

[the] http://deeplearning.net/software/theano/.

[TR12a] Vikrant Singh Tomar and Richard C. Rose. Application of a locality preserving discriminant analysis approach to ASR. In *11th International Conference on Information Science, Signal Processing and their Applications, ISSPA 2012, Montreal, QC, Canada, July 2-5, 2012*, pages 103–107, 2012.

[TR12b] Vikrant Singh Tomar and Richard C. Rose. A correlational discriminant approach to feature extraction for robust speech recognition. In *INTERSPEECH 2012, 13th Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012*, pages 555–558, 2012.

[TR13a] Vikrant Singh Tomar and Richard C. Rose. Efficient manifold learning for speech recognition using locality sensitive hashing. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 6995–6999, 2013.

[TR13b] Vikrant Singh Tomar and Richard C. Rose. Locality sensitive hashing for fast computation of correlational manifold learning based feature

space transformations. In *INTERSPEECH 2013, 14th Annual Conference of the International Speech Communication Association, Lyon, France, August 25-29, 2013*, pages 1776–1780, 2013.

[TR13c] Vikrant Singh Tomar and Richard C. Rose. Noise aware manifold learning for robust speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 7087–7091, 2013.

[TR14a] Vikrant Singh Tomar and Richard C. Rose. A family of discriminative manifold learning algorithms and their application to speech recognition. *IEEE/ACM Transactions on Audio, Speech & Language Processing*, 22(1):161–171, 2014.

[TR14b] Vikrant Singh Tomar and Richard C. Rose. Manifold regularized deep neural networks. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, pages 348–352, 2014.

[Ver08] Nakul Verma. Mathematical advances in manifold learning, 2008.

[WHH+90] Alexander Waibel, Toshiyuki Hanazawa, Geofrey Hinton, Kiyohiro Shikano, and Kevin J. Lang. Readings in speech recognition. chapter Phoneme Recognition Using Time-delay Neural Networks, pages 393–404. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[wsja] https://catalog.ldc.upenn.edu/docs/ldc94s13a/wsj1.txt.

[wsjb] https://catalog.ldc.upenn.edu/ldc93s6a.

[YD14] Dong Yu and Li Deng. *Automatic Speech Recognition: A Deep Learning Approach*. Springer Publishing Company, Incorporated, 2014.

[YEK+02] S. Young, G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, V. Valtchev, and P. Woodland. The htk book. *Cambridge University Engineering Department*, 3, 2002.

[You02] Steve Young. Acoustic modelling for large vocabulary continuous speech recognition. *Cambridge University Engineering Department*, 2002.

[YSL+13] Dong Yu, Michael L. Seltzer, Jinyu Li, Jui-Ting Huang, and Frank Seide. Feature learning in deep neural networks - A study on speech recognition tasks. *CoRR*, abs/1301.3605, 2013.

[YXZ+07] Shuicheng Yan, Dong Xu, Benyu Zhang, Hong-Jiang Zhang, Qiang Yang, and Stephen Lin. Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(1):40–51, January 2007.