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

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.

1.1 Preparatory work

1.1.1 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:

$$p(\mathbf{x_t}|q_t = s) = \frac{P(q_t = s|\mathbf{x}_t)P(\mathbf{x}_t)}{P(s)}$$

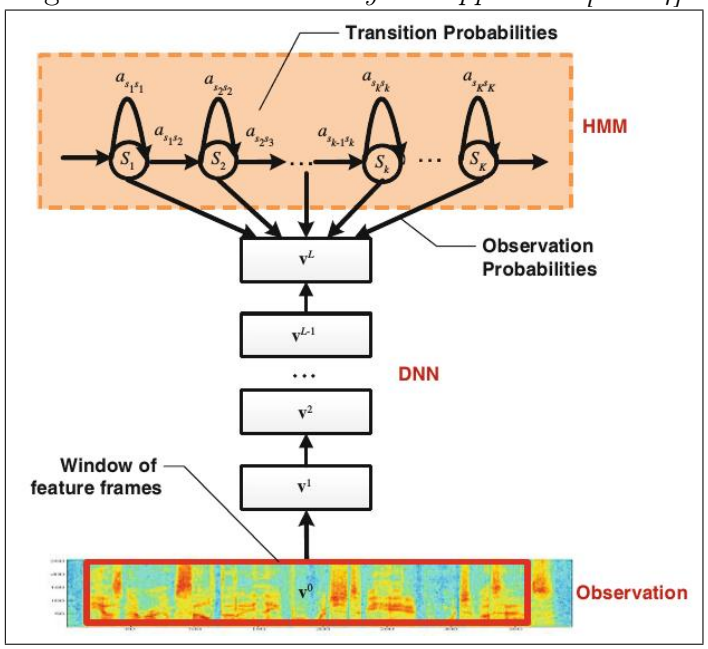
where $p(s)$ is the prior probability of each senone estimated from the training set as

$$P(s) = \frac{T_s}{T}$$

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 1.1: *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 1.2: *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 1.2 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.

1.1.2 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.

1.2 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 1.1: 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

1.2.1 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 1.1.

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.

1.2.2 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 ??.

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].

1.3 Manifold regularization

why hybrid approach

Introduction

The task of the current project was to implement the ideas described in **??** 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 **??**, 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 1.3. 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 1.4 and 1.5.

Figure 1.3: *Projection in 2D of 2.5k MFCC and energy feature vectors using PCA.*

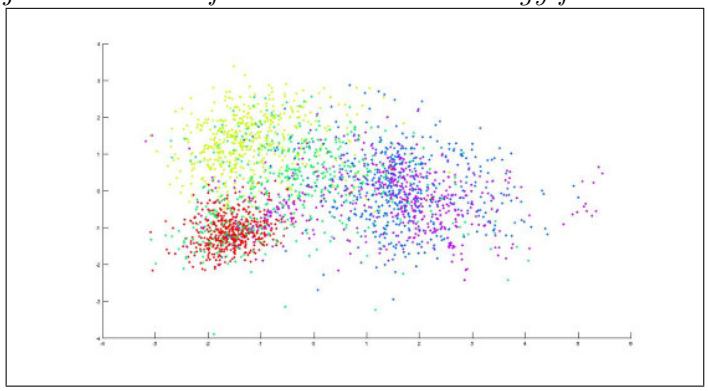
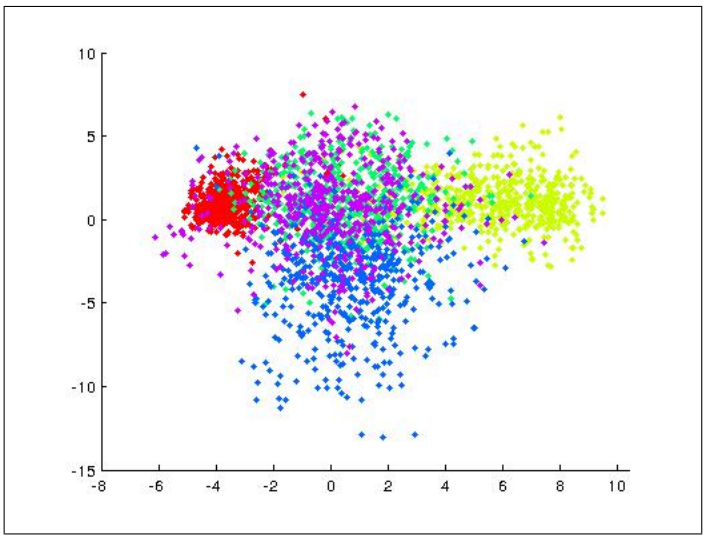


Figure 1.4: *LPDA, 2D projection, kpen=kint=400, Rint=850,Rpen=3000, 2.5k data, 5 phones*



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 1.6 and 1.7) 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 1.8 - 1.11 . 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 1.5: *LPDA, 3D projection, kpen=kint=400, Rint=850,Rpen=3000, 2.5k data, 5 phones*

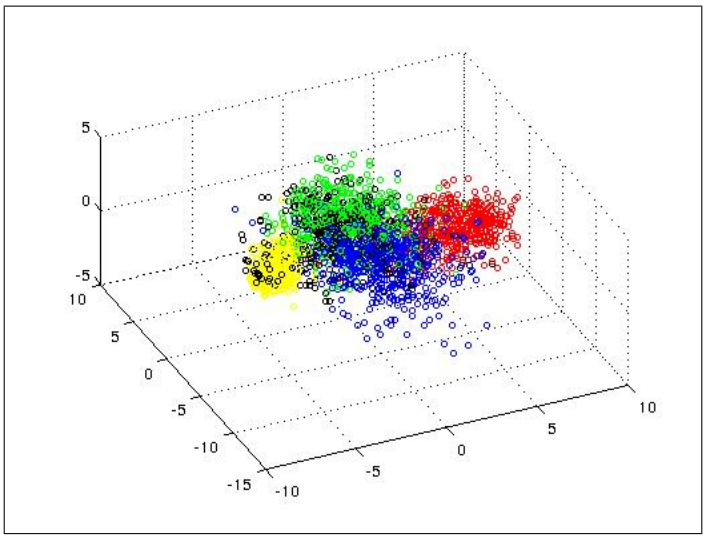


Figure 1.6: *LPDA, 3D projection, kpen=500,kint=600, Rint=850,Rpen=3000, 2.5k data, 5 phones*

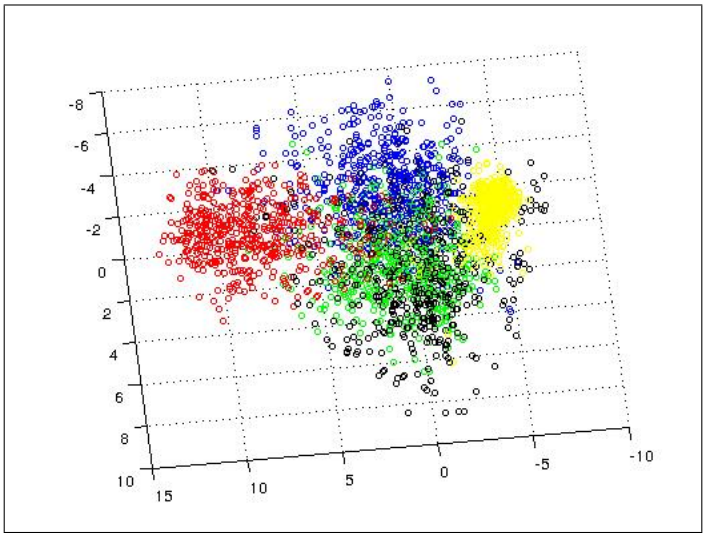
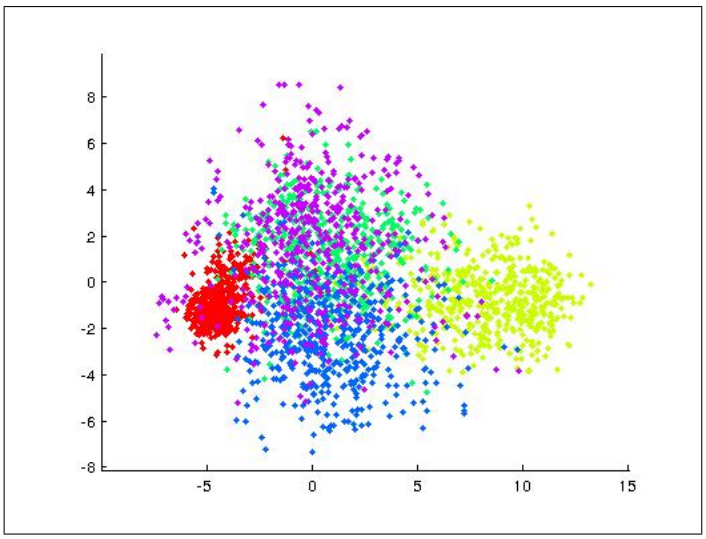


Figure 1.7: *LPDA, 2D projection, kpen=500,kint=600, Rint=850,Rpen=3000, 2.5k data, 5 phones*



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 **??**. 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 1.8: *LPDA, 3D projection, kpen=kint = 400, Rint=850,Rpen=3000, 2.5k data, phone-HMMstate label*

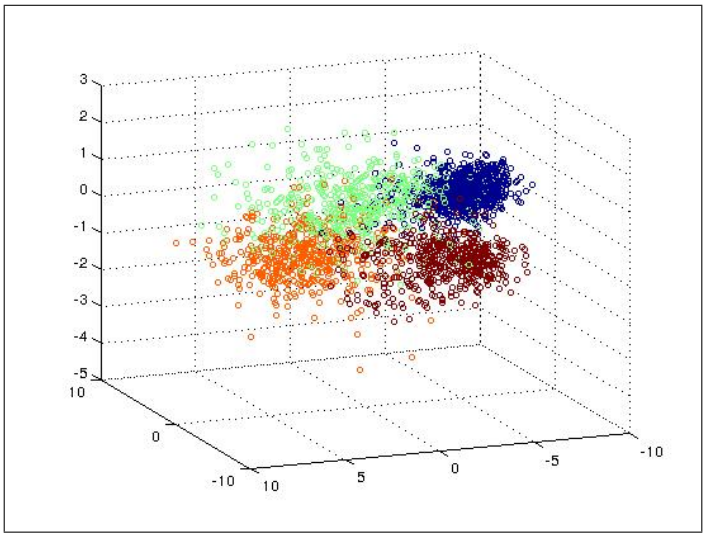


Figure 1.9: *LPDA, 2D projection, kpen=kint = 400, Rint=850,Rpen=3000, 2.5k data, phone-HMMstate label*

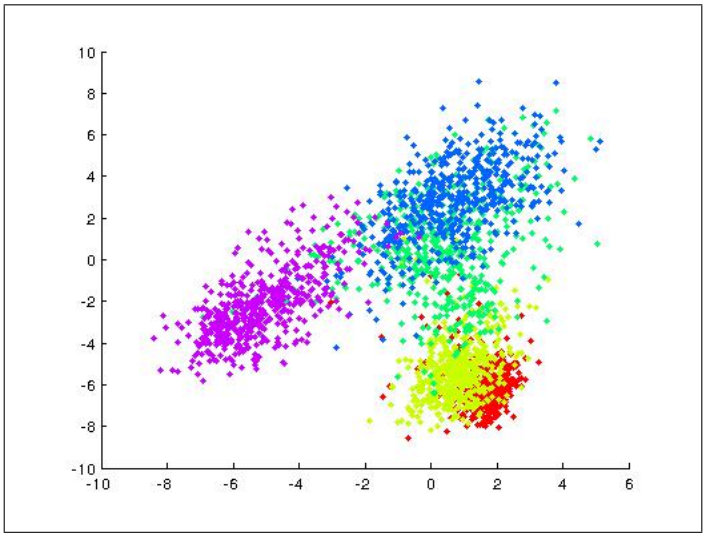


Figure 1.10: *LPDA, 3D projection, kpen=kint = 600, Rint=850,Rpen=3000, 2.5k data, phone-HMMstate label*

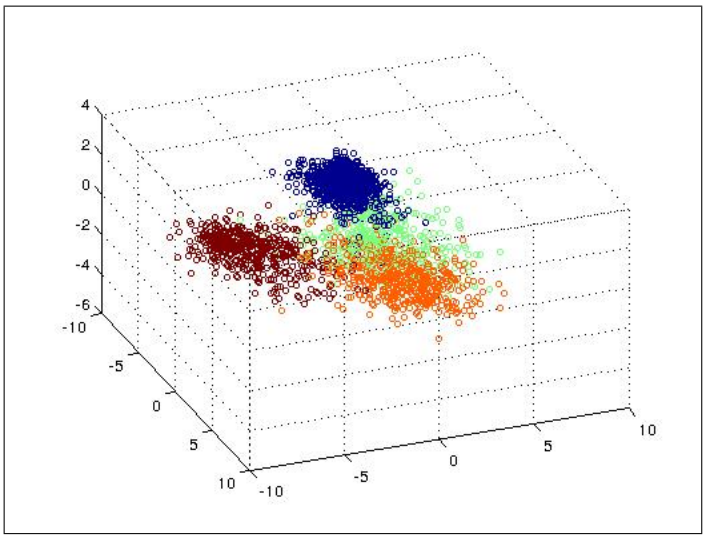
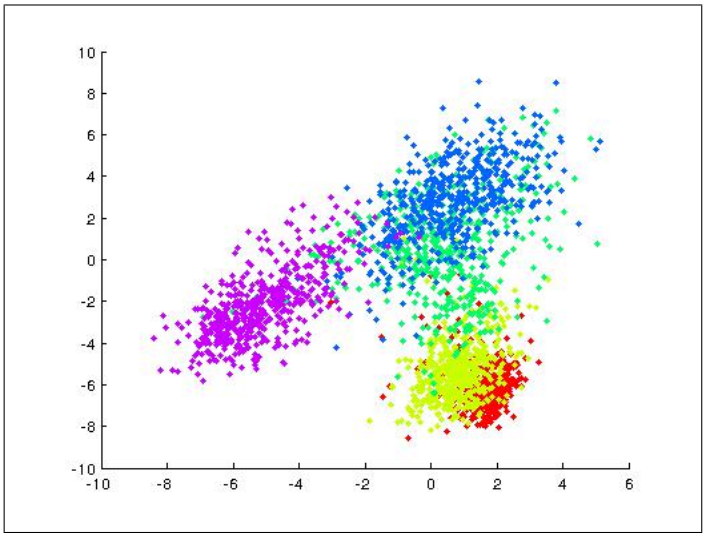


Figure 1.11: *LPDA, 2D projection, kpen=kint = 600, Rint=850,Rpen=3000, 2.5k data, phone-HMMstate label*



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 1.12.

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 1.12: *Region of node v, picture taken from Computational Geometry class in [kdt].*

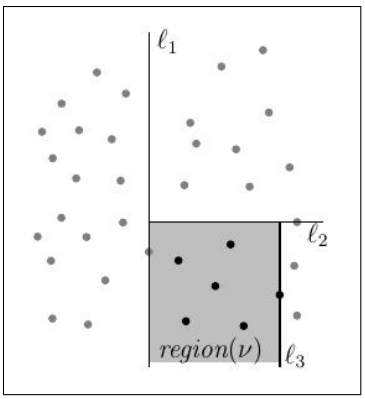
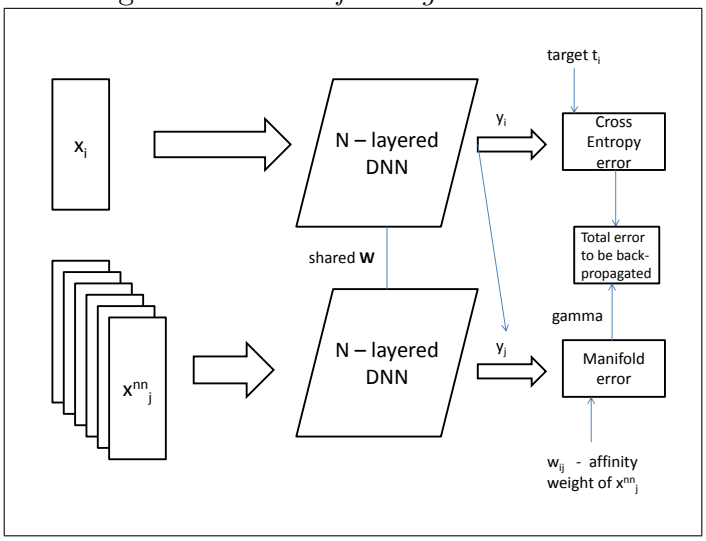


Figure 1.13: *Manifold regularized DNN*



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 1.13 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.

1.4 Decoding and experimental results

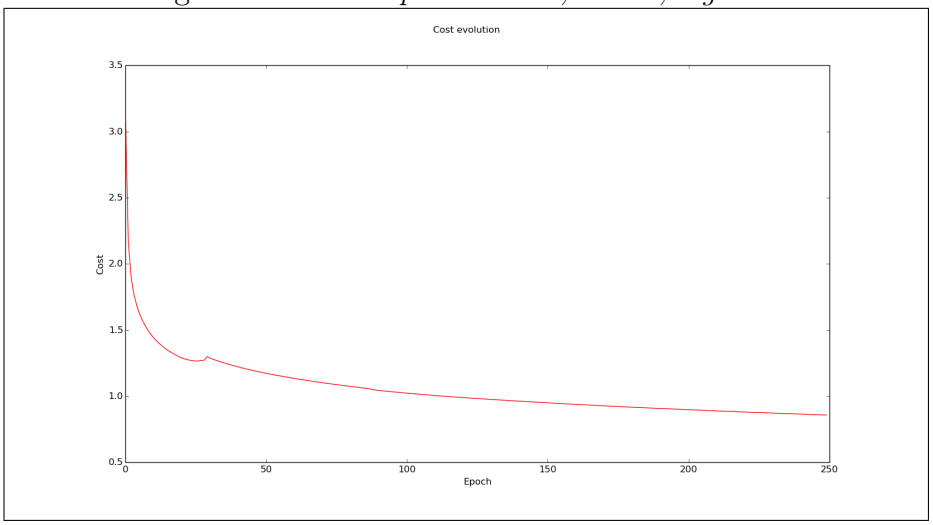
cost-valid plots, observations/conclusion

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 1.14: *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 1.14 and 1.15. 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 1.16 and 1.17. 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 1.18 and 1.19. 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 1.15: *Monophone DNN, 5x600, sigmoid*

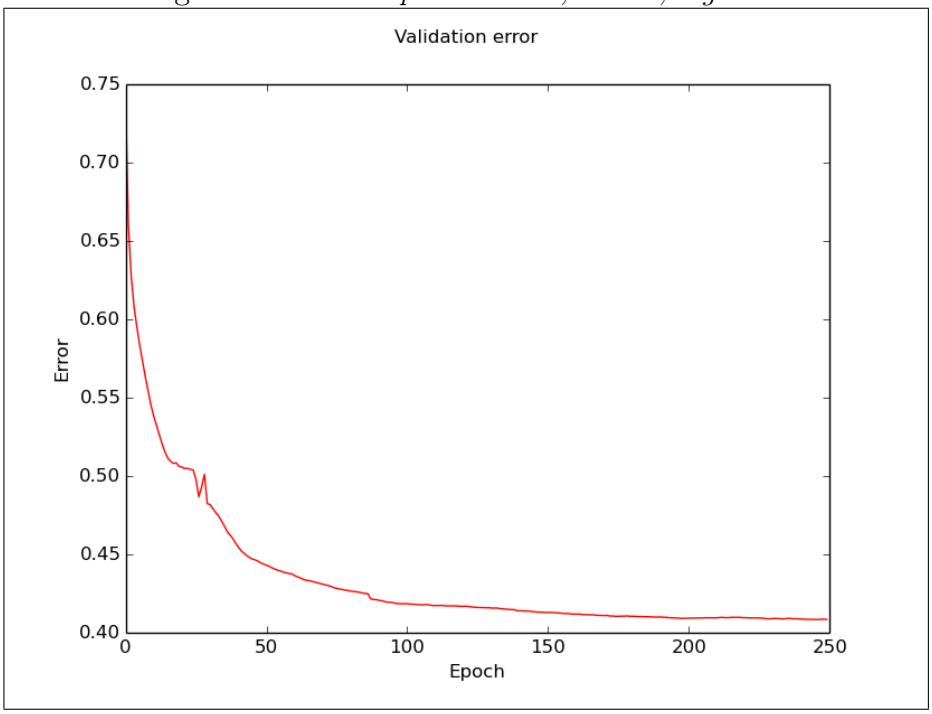


Figure 1.16: *Monophone DNN, 4x1024, sigmoid*

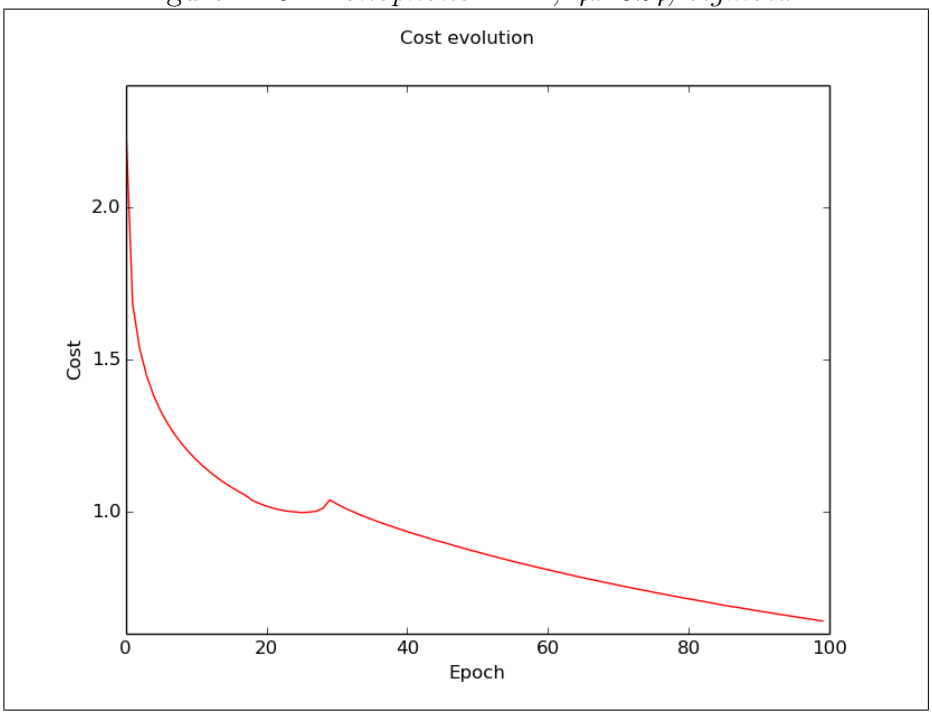


Figure 1.17: *Monophone DNN, 4x1024, sigmoid*

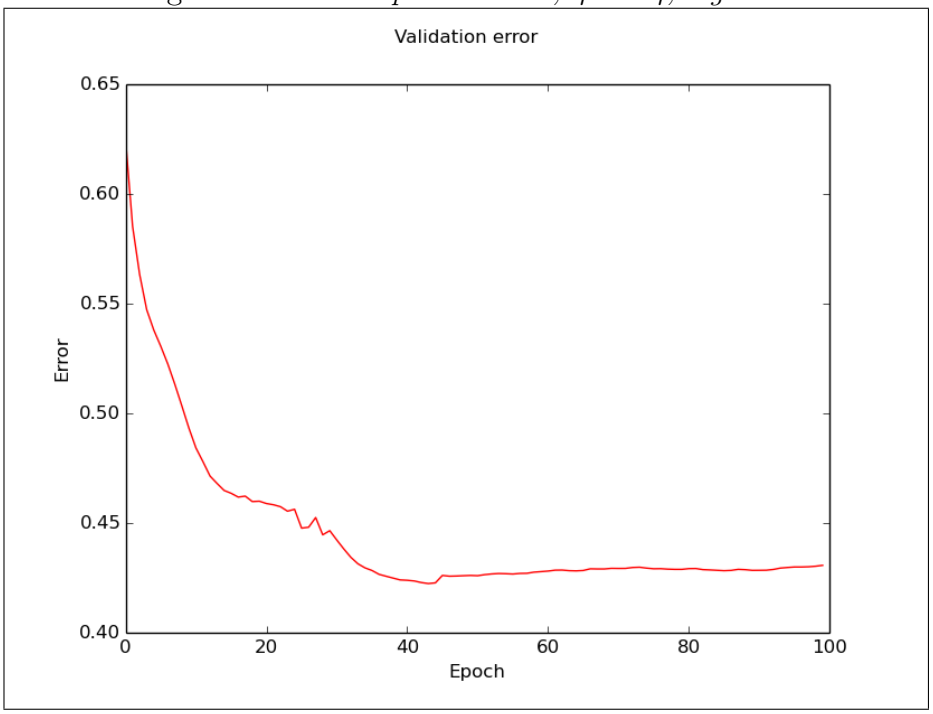


Figure 1.18: *Monophone DNN, 4x1024, tanh*

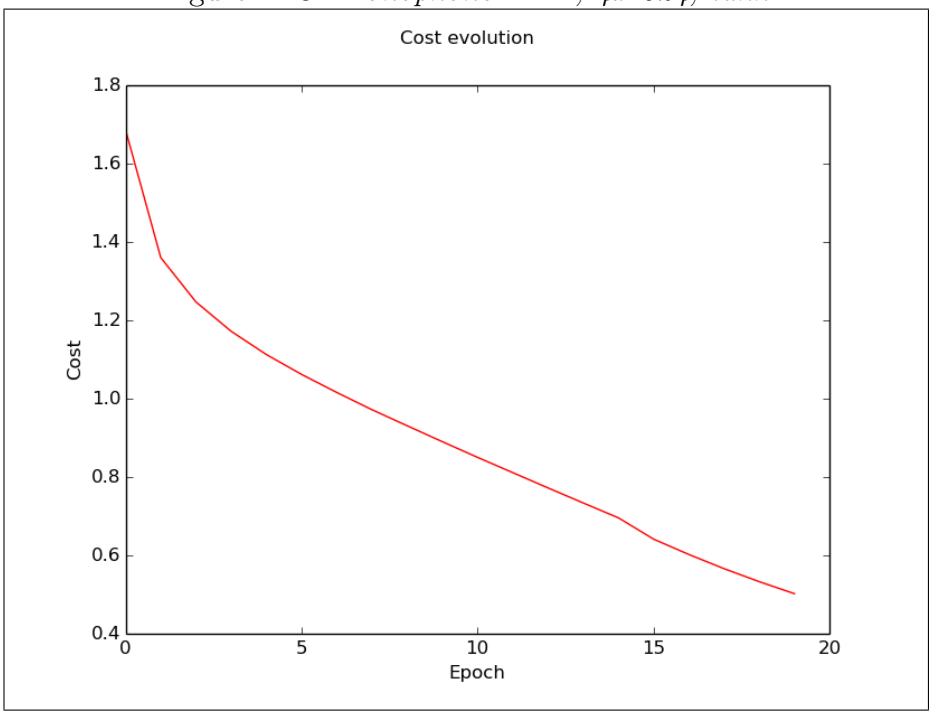
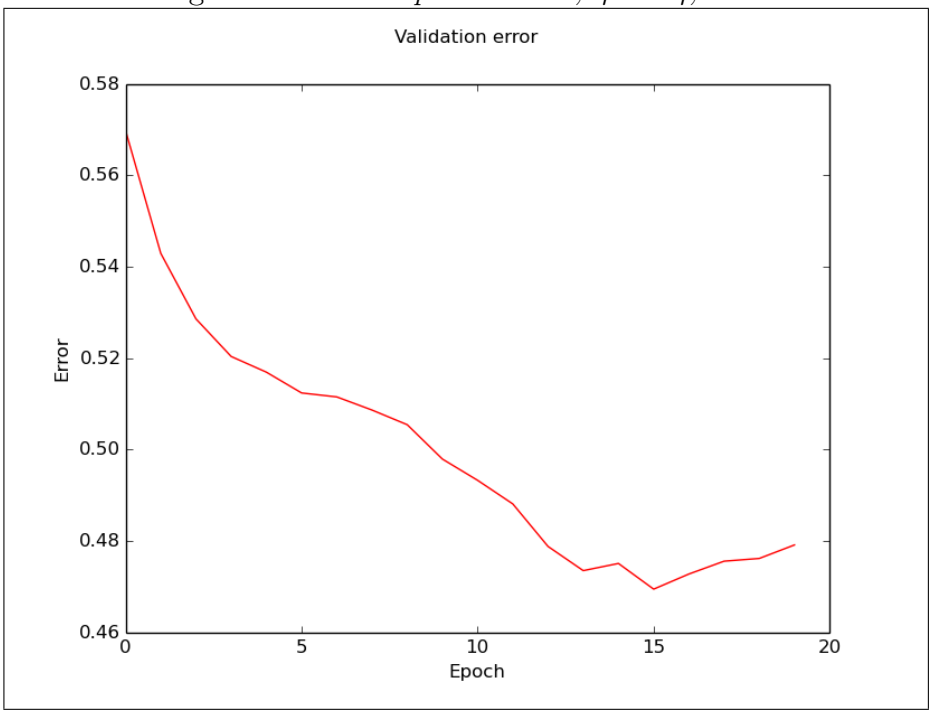


Figure 1.19: *Monophone DNN, 4x1024, tanh*



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 1.20 and 1.21. 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 1.22 and 1.23. The decoding accuracy on dev93 dataset was 81.56% and on eval92 88.64%.

Figure 1.20: *Monophone DNN, 4x1024, ReLU*

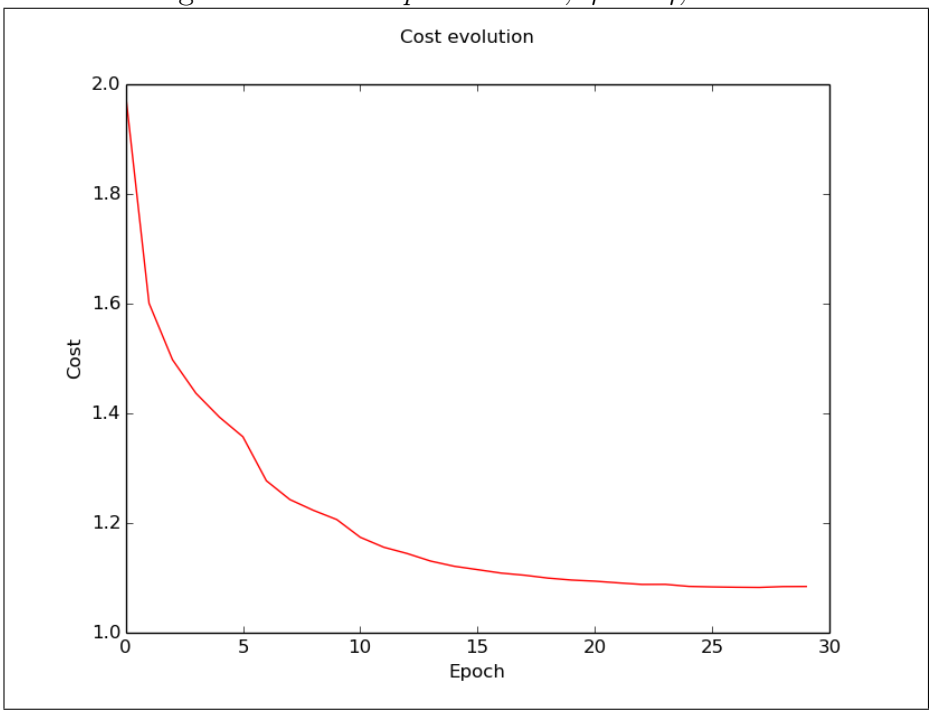


Figure 1.21: *Monophone DNN, 4x1024, ReLU*

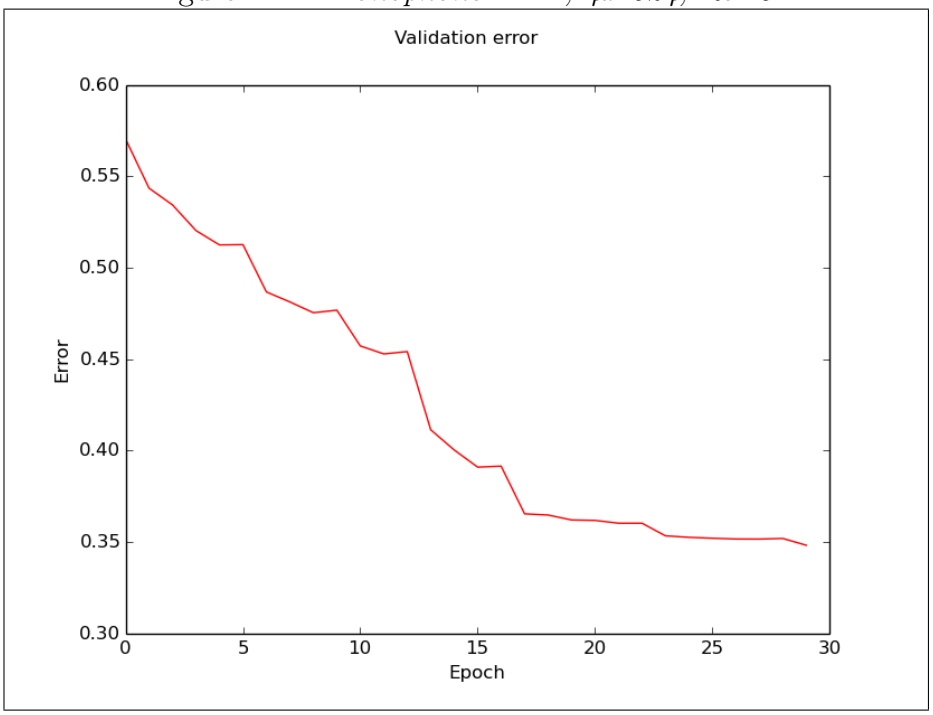


Figure 1.22: *Triphone DNN, 6x2048, ReLU*

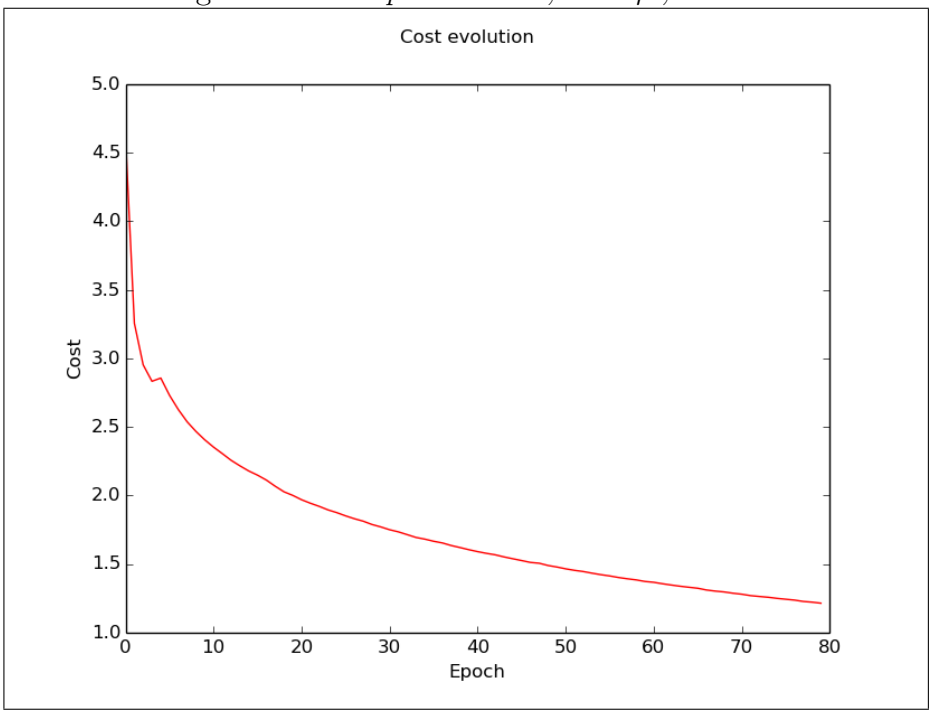


Figure 1.23: *Triphone DNN, 6x2048, ReLU*

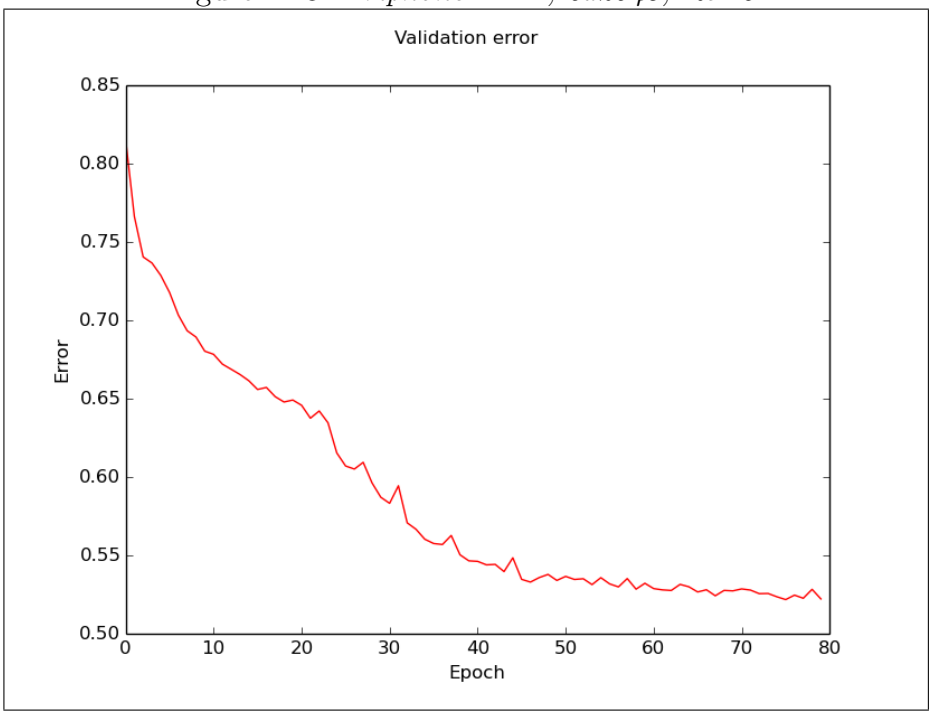
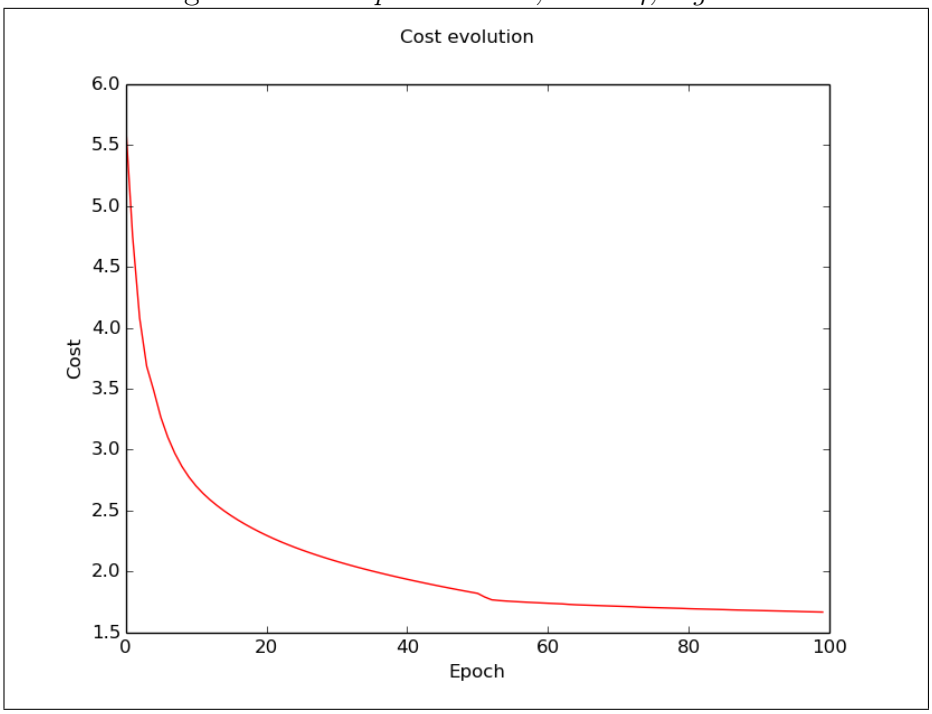


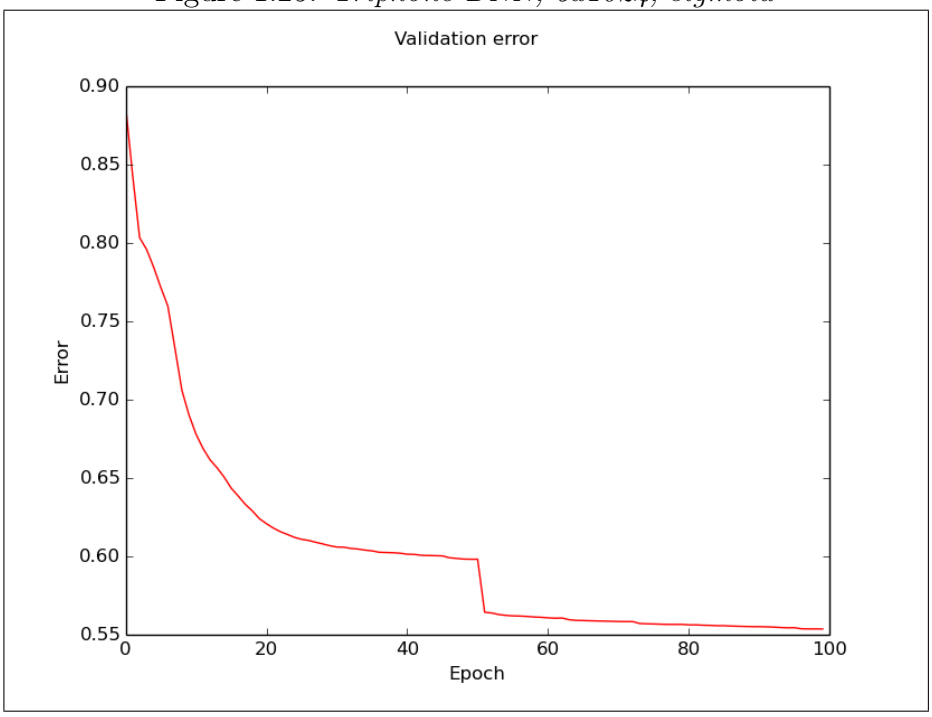
Figure 1.24: *Triphone DNN, 5x1024, sigmoid*



- 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 1.24 and 1.25. 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 1.25: *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

[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.

[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.

[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.

[caf] https://github.com/bvlc/caffe/issues/109.

[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.

[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.

[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.

[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.

[Mia14] Yajie Miao. Kaldi+pdnn: Building dnn-based ASR systems with kaldi and PDNN. *CoRR*, abs/1401.6984, 2014.

[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.

[the] http://deeplearning.net/software/theano/.

[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.

[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.

[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.