NTNU
Det skapende universitet

Dype Nevronett for Talegjenkjenning

Jonathan Halstensen
Lepsøy

Master i elektronikk
Innlevert: juni 2015
Hovedveileder: Torbjørn Svendsen, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Project Description

Artificial neural networks have been good tools for machine learning and pattern recognition for the past decades. In practice, these have been limited to three "layers", i.e. one input layer, one so called hidden layer, and an output layer. Through the past few years there has been a breakthrough for training techniques for "deep learning", i.e. neural networks with more than one hidden layer. This type of neural network has shown significant improvement of performance as compared to traditional neural networks in a range of areas, including speech recognition.

In speech recognition, Deep Neural Networks (DNN) have primarily been used for estimation of observation probabilities. There are however several other possible applications, e.g. detection based speech recognition and acoustic-articulatory inversion.

In the first part of the task, the properties of a deep neural network as an estimator of observation probabilities in a speech recognition system will be investigated. The speech recognizer is based on Hidden Markov Models (HMM). More specifically, it will be investigated if there are language dependent differences on an acoustic-phonetic level through comparing the performance of a speech recognition system on a Norwegian and an American-English database.

In the second part of this task, it will be investigated if deep neural networks can be applied for acoustic-articulatory inversion. The acoustic pressure wave which is produced when we speak is a result of how we use the articulators, i.e. lips, tongue, jaws, and vocal cord. An articulatoric description of the speech production is universal, since all humans have the same apparatus for general speech, and is therefor attractive to create models for speech production. Acoustic-articulatory inversion is based on estimating the position and movement of the articulators from the speech signal's waveform.

Abstract

In this work we measure the performance of a Hidden Markov Model-based phone recognition system on the Norwegian speech database NAFTA. The Hidden Markov Model posterior probabilities are trained using a Deep Neural Network and the results are compared to results from experiments on the American-English database TIMIT. We find comparable results and find the same deficiencies in both cases, i.e. frequent confusion pairs. Based on the results, we further explore Acoustic-articulatory inversion mapping by implementing a Deep Belief Network to investigate if the performance is sufficient for use in a hybrid recognition system to cope with the discovered deficiencies. Our implementation obtained an average root mean square error of 1.348mm on the mngu0 test dataset showing that sufficient precision is achievable.

Sammendrag

I dette arbeidet tester vi ytelsen til et fonemgjenkjenningssystem, basert på skjulte markovmodeller, på den norske taledatabasen NAFTA. De skjulte markovmodellenes aposteriori sannsynligheter blir trent ved hjelp av et dypt kunstig nevronett og resultatene sammenlignes med resultatene fra eksperimenter gjort på en amerikansk-engelsk database kalt TIMIT. Resultatene viser fonemfeilrate i samme størrelsesorden og vi finner de samme svakhetene i begge tilfeller, i.e. par av fonemer som ofte forveksles. Basert på disse resultatene utforsker vi videre akustisk-artikulatorinversjon ved hjelp av et dypt nevronett. Målet er å undersøke om disse kan oppnå tilstrekkelig ytelse for å danne et hybrid-system sammen med en fonemgjenkjenner for å omgå de nevnte svakhetene. Vår implementasjon oppnådde en root mean square error på 1.348mm på mngu0 testdatasettet, hvilket viser at tilstrekkelig ytelse er oppnåelig.

Acknowledgements

I would like to thank my supervisor Professor Torbjørn Svendsen for insightful suggestions and guidance. Without our frequent discussions, this work would not be possible to complete. In addition, I would like to thank Associate Professor Magne Hallstein Johnsen for valuable help during the most stressful periods.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Artificial intelligence (AI) is a broad field of study where one aims to make computers model our world based on digital input and to act in an intelligent manner based on this input. There are many interpretations of what true intelligence really is, but that is not a concern of this paper. A widely used definition of the field, is that it is the study of developing intelligent agents that perceive their environment and takes actions that maximize their chance of success. For a computer to model our world in a sufficient way to make correct decisions it will have to process and store large amounts of data from several types of input such as sound and picture. To formalize rules and logic for every given input, both input that has already been discovered and also prepare it for future, so far potentially undiscovered inputs is almost impossible. A possible solution to this problem is learning algorithms. These algorithms attempt to capture details and information from vast amounts of data and improve their performance by adapting as new information arrives. To a certain extent these algorithms perform well within several fields, pattern recognition and classification to name a few. However, many of the learning algorithms that exist today fail to utilize semantics which is an essential part of humans' way of communicating. Other challenges are to make a computer socially intelligent and be able to express and interpret emotions and make it attain the ability to be creative. A benchmark for AI is for the computer to be able to communicate with humans in a way that makes it seem natural and indistinguishable from a human being.

Speech recognition can be done in many ways, but in essence it involves taking some sort of representation of a speech waveform and producing a transcription, typically on a word level basis as a decoding of the speech. To create a system which can model all the different words in a language can be hard and impractical due to the size of the dictionary for any given language. However, all the words in every language are built by a limited set of building blocks called phonemes. Such a set is easier to model and also more universal across languages. Due to this, systems that perform phone recognition have been developed. Phone recognition systems have been highly successful and come in a wide range of variations, the most widely used of which, is utilizing Hidden Markov Models (HMMs) [6].

Ever since its *discovery* in the 1940's [7], the development of its mathematical models [8] and the development of a sufficient training algorithm [9], Artificial Neural Networks (ANNs) have been a part of speech recognition research and was applied successfully in a range of applications. Due to several difficulties, in particular with training such networks, these networks were shallow architectures, i.e. only one or two hidden layers. These difficulties included lack of sufficiently large training data, slow training algorithms and, due to the gradient-relying training algorithms, local minimum converging learning. Due to these apparent issues, researchers moved away from neural networks to pursue more promising alternatives. This was true until what many refer to as the renaissance of Deep Neural Networks (DNNs) occurred. In 2006, G. E. Hinton et al. [10] introduced the Deep Belief Network and a fast, unsupervised, training algorithm for it. In addition to only requiring unlabeled data, the algorithm also countered the issue with local minimum if used as an initialization step for a regular DNN. The unsupervised learning accompanied with the increase in speed allowed for training of deeper architectures which in turn unleashed the true power of neural networks. Deep neural networks have since then been successfully applied in a multitude of disciplines achieving comparable, and even better results than current state-of-the-art methods [11].

1.2 Motivation

The experiments documented in this dissertation can be divided into two main parts. In the first, a database consisting of Norwegian speech was used to train

a state-of-the-art phone recognition system to compare its performance on Norwegian speech to the results already documented on the TIMIT which consists of American-English speech. The experiments yielded similar, however significantly poorer results, and the system suffers from the same deficiencies for both languages. In chapter 5, tables of the most common mistakes made on both the Norwegian and English speech are shown and the major difficulty for the system becomes obvious. A wide range of phonemes have very similar spectral content and thus even speaker individual differences can confuse the system. E.g. in the case of speech-to-text systems, this is not acceptable and state-of-the-art speech recognition systems report phone error rates of the order of 20% [12].

Speech is produced by the passage of air through the vocal tract, and the sound produced is determined by the position of the vocal folds and shape of the vocal tract. The vocal tract articulators include, but are not limited to, the jaw, lips and tongue, as shown in figure 2.5. The Acoustical-articulatory inversion mapping problem involves inferring the position of the vocal tract articulators from an acoustic speech signal. It is possible to produce the same, or highly similar sounds, with several articulator configurations. This fact, along with the inversion problem being a non-linear one, makes the task a challenging one. However, the fact is in some sense the opposite problem to that of phone recognition. Assume for a second that this challenge can be overcome and the articulator positions can be estimated. Further, recall the confusion pairs from the tables in chapter 5. One of the pairs that appears for both the English and Norwegian datasets is the confusion between */m/* and */n/*. By saying these two sounds out loud and noticing the position of the mentioned articulators it becomes obvious how different the positioning is even though the sounds are perceived fairly similar in a context. The most noticeable difference is that the lips are closed for the */m/* and open for the */n/*. So while still keeping in mind our assumption of being able to infer the position, it would now be possible to detect if the lips are open or closed and thus also discriminate between the two sounds. By doing similar tests it should be obvious that this could potentially reduce the phone error rate by a large margin. This is the main motivation for the second part of the experiments, namely performing the inversion mapping.

The motivation for these experiments is not to create a stand-alone recognition system. It is rather to investigate if articulatory inversion mapping can discriminate between phonemes which the phone-recognition system can not. Due to

this, only the inversion mapping is performed in these experiments. The goal is to analyze if such a system can achieve precise enough mapping to be used in a hybrid system with a phone-recognizer. Encouraged by the results documented in Benigno Uría's Master's thesis [5], a deep neural network was chosen for the task.

The rest of this dissertation is organized as follows: in chapter 2, we will review the theory behind neural networks, how they work, their different training methods and some of the variations that exist. In chapter 3, we will introduce the different coding libraries and databases utilized for the experiments and give a brief description of their functionality and content respectively. In chapter 4, we will detail our experimental setups in terms of general approach, implementation and data setup. Further, we will display our results from the different experiments, analyze their performance, compare them with each other and discuss their significance. To finish, in chapter 5 we will draw some conclusions and outline future research directions we intend to pursue encouraged by the results documented in this dissertation.

Chapter 2

Theory

2.1 Linear Perceptron

The single layer perceptron is one of the simplest neural networks and is the building block for several other variants. A simple net can learn to recognize simple patterns and do tasks like classifying and clustering. Figure 2.1 illustrates the practical connectivity for one class only. The dark gray nodes comprise the input layer and the light gray node comprises the output layer. A network with K classes will have one node in the output layer for each class and every input node will be connected to each output node.

We assume a set of K classes, $\omega_k, k = 1, ..., K$. The system is defined by its linear discriminant function:

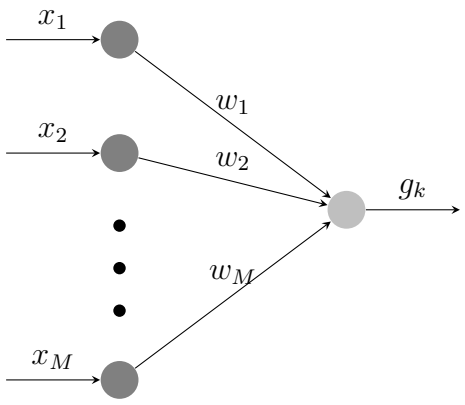

FIGURE 2.1: Graphical representation of a perceptron

$$g_k(x) = \mathbf{w}_k^T \mathbf{x} + b \tag{2.1}$$

where:

$$\mathbf{x} = \{x_1, x_2, ..., x_M\}^T$$

is the input vector transposed,

$$\mathbf{w}_k = \{w_{k1}, w_{k2}, ..., w_{kM}\}^T$$

is the weight vector transposed, corresponding to the respective class, and b is a bias term. It is common practice to append an element with the value 1 to the input vector and extend the weight vector with one element to remove the bias term and simplifying the terms. Doing this and collecting all the discriminant functions from equation (2.1) leads to:

$$g(\mathbf{x}) = W^T \mathbf{x}_n \tag{2.2}$$

where:

$$g(x) = \{g_1(x), g_2(x), ..., g_K\}^T, \qquad W = \{\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_K\}$$

Classification is done based on the rule:

$$if \quad g_i(x) \geq g_j(x) \quad \forall i \neq j \quad then \quad x \in \omega_i \tag{2.3}$$

It is useful to consider the two-dimensional 2-class case with only two inputs for illustrational purposes. The slope of the decision line is determined by the weights and the offset from the origin is determined by the bias term. The decision line divides the space in two halves as shown in figure 2.2 and in this 2-class scenario, any values above the line will lead to class A responses and any below will lead to class B responses. Decisions for values on the line can be solved in any way desirable.

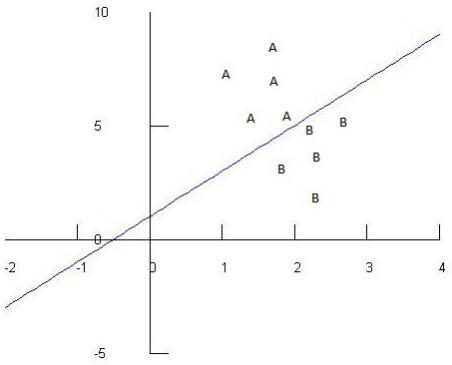

FIGURE 2.2: Two-input, two-class classification problem with decision border

The weights and biases can be either fixed or adapted. There exists several different algorithms to train the network which offer good results for datasets with certain properties. The original training algorithm for perceptrons was developed by Rosenblatt [13]. This simple algorithm is guaranteed to converge if the dataset is linearly separable. The network, when trained, separates the space into K different convex regions corresponding to each class. The decision boundaries are defined by $y_i(x) = y_j(x)$ and are thus composed of segments of hyperplanes. As in the simple case shown in figure 2.2, the biases allow segments of hyperplanes that do not contain the origin.

Early reasearch on perceptrons show the versatility of the perceptron, but also some of its limitations. The perceptron can only partition the space with linear decision boundaries and it has thus been shown that the perceptron training algorithm does not converge for datasets that are not linearly separable [14]. A solution for the convergence problem is to use a Least Mean Square (LMS) criterion. This will not guarantee that 2.3 will hold for all input vectors, even for linearly separable data sets, but it will allow for convergence even if the set is not linearly separable. It has been shown that this approach may be superior in some situations [14].

Training a neural network is based on updating the weights by optimizing a cost function with respect to the weights. In the LMS approach for perceptrons, the updated weights are calculated from a preclassified training set by minimizing the cost function E for all input vectors in the set and all classes. The cost function, or Mean Square Error (MSE) is defined as:

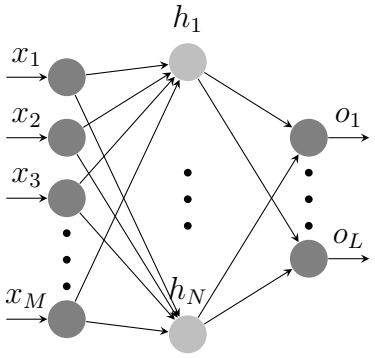

FIGURE 2.3: Graphical representation of a Multi-Layer Perceptron with M inputs, N hidden units and L classes

$$E = \sum_{k=1}^{K} \sum_{\mathbf{x} \in \omega_k} ||g(\mathbf{x}) - \Delta k||^2 \tag{2.4}$$

In the MSE, the output is matched to the expected result, in most cases, 1 if the input vector belongs to the given class and 0 if the vector does not belong to the class. Δk is a vector consisting of zero-components except from the k-th one which is 1 and corresponding to the correct class. Minimizing E with respect to all the weights provides the updates to the weights.

2.2 Multi-Layer Perceptron (MLP)

Multi-layer perceptrons are feed-forward networks with an arbitrary number of layers of nodes between the input and the output layers described in section 2.1. These layers are called hidden layers and consist of one or more hidden nodes with full inter-layer connectivity and no intra-layer connectivity. An example of a network with one hidden layer can be seen in figure 2.3.

The number of input nodes must correspond to the size of the input, e.g. for an image of size P x Q, $M = P \cdot Q$, and the number of output nodes must correspond to the desired output, i.e. the number of classes in a classification scenario. The number of hidden nodes can be arbitrarily large and must be either estimated or chosen with respect to other specifications like computational complexity and size of the training set. The main benefit from using an MLP is to overcome the issues with linearity. Adding several layers of linear units will, once the network

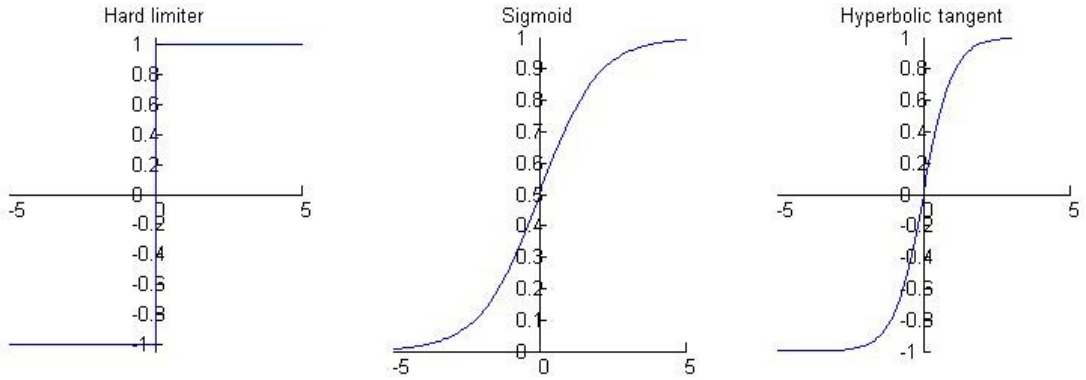

FIGURE 2.4: Non-linear operators

has been trained, be equivalent to a single layer linear system, possibly with a loss of rank. It has been shown, however, that adding a nonlinear operator on the output of each layer will allow the system to perform *any* nonlinear mapping from the input to the output. In fact, a system with only one hidden layer can, with a sufficient amount of hidden units, represent any continuous function with arbitrary accuracy. To extend this further it has also been shown that with two layers, even discontinuous functions can be represented [15]. It is important to note that even if the number of hidden units is theoretically possible, it might be impractically large.

There are several different nonlinear operators to consider for the MLP. The most common ones are the hard limiter, the sigmoid, and the hyperbolic tangent shown in figure 2.4. The ideal operator would be the hard limiter, but because of the gradient based training methods (see section 2.3) the operator has to be differentiable. The most widely used operators are the sigmoid and the hyperbolic tangent. Both have the useful property that the slope is maximum around zero and the gradient will peak in this area, i.e. close to the decision border, which is the most critical area. The main difference between the two is that the sigmoid's dynamic area is $(0, 1)$ while the hyperbolic tangent's is $(-1, 1)$. Here, we will focus on the sigmoid function:

$$F(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

Modifying the LMS criterion we get:

$$E = \sum_{k=1}^{K} \sum_{x \in \omega_k} ||F[g(\mathbf{x})] - \Delta_k||^2 \tag{2.6}$$

where k refers to the output node index for the current layer. The sigmoid function is applied element wise and the modified discriminant function is thus a vector with one element for each output node g:

$$F_i[g(\mathbf{x})] = \frac{1}{1 + exp(-g_i(\mathbf{x}))} \tag{2.7}$$

An n-layered MLP consists of $(n+1)$ layers L_l where L_0 corresponds to the input-layer, L_n corresponds to the output layer and $L_1, ..., L_{n-1}$ correspond to the hidden layers. The state-to-state transition equation is given as:

$$\mathbf{h}_l = F(W_l^T \mathbf{h}_{(l-1)}) \tag{2.8}$$

where F is the nonlinear operator described above.

2.3 Backpropagation & gradient descent

Using the LMS criterion as cost function requires *a priori* knowledge of the expected output of the layer, i.e. the Δk from equation (2.6). For the output layer the expected result is given by the training vector label, but for the intermediate hidden layers this is unknown. As the output label is the only knowledge we have on the expected output, a scheme was developed to iteratively update one layer at a time, beginning with the output layer, and sending the error back to the previous level until it reaches the input layer. This method is called backpropagation of error. In this section we will adopt some of the notation from [16], but will not go in as great detail when it comes to derivations.

As we know the expected result on the output layer we will derive the update rule for the output layer and later generalize the result to apply for the hidden layers. We denote the expected output for each unit d_k where k is the unit index. We also define a new cost function:

$$E = \frac{1}{2}\sum_{k}(d_k - o_k)^2 \tag{2.9}$$

where o_k is the actual result on the output node and the vector $\mathbf{o} = h_n$ is calculated from equation (2.8). The goal is to minimize the error and the negative gradient is thus calculated with respect to each weight:

$$-\frac{\partial E}{\partial w_{kj}^o} = (d_k - o_k)\frac{\partial F_k^o}{\partial g_k^o}\frac{\partial g_k^o}{\partial w_{kj}^o} \tag{2.10}$$

We set the weight change to be proportional to the gradient with a parameter α which is called the Learning Rate(LR). This parameter will vary with application but is typically below 1 and positive. Inserting this and evaluating the differentials above leads to:

$$\Delta_{kj}^o = \alpha i_j (d_k - o_k) F_k^{o\prime}(g_k^o) \tag{2.11}$$

and:

$$\delta_k^o = (d_k - o_k) F_k^{o\prime}(g_k^o) \tag{2.12}$$

where the derivative $F_k^{o\prime}$ is not evaluated to generalize for different operators F. i_j is the output from previous-layer node j. The weight update rule at iteration t for the output layer is thus:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \alpha\delta_{kj}^o i_j \tag{2.13}$$

As mentioned earlier, the issue with training the hidden layers is that d_k is unknown. It is intuitive to expect that the output error E depends on the outputs of the hidden layers. By expressing the actual result o_k with respect to the result from the previous layer:

$$o_k = F_k^o(W_{kj}^T \mathbf{h}_{(n-1)}) \tag{2.14}$$

we can evaluate the negative gradient which leads to:

$$-\frac{\partial E}{\partial w_{ji}^h} = F_j^{h\prime}(g_j^h) i_i \sum_k (y_k - o_k) F_k^{o\prime}(g_k^o) w_{kj}^o \tag{2.15}$$

Again we choose the change of weights to be proportional to the negative derivative and define the absolute value of change by inserting from equation(2.12) to be:

$$\Delta_{ji}^h = \alpha F_j^{h\prime}(g_j^h) i_i \sum_k \delta_k^o w_{kj}^o \tag{2.16}$$

and:

$$\delta_j^h = F_j^{h\prime}(g_j^h) \sum_k \delta_k^o w_{kj}^o \tag{2.17}$$

Equation (2.16) illustrates why the method is called backpropagation. The errors from the output layer is *propagated back* to the previous layer using the weights between the layers *before* they are updated in the current iteration. From this the update rule for each hidden layer is defined as:

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \alpha \delta_j^h i_i \tag{2.18}$$

For each vector in the training set the vector is sent through the network to produce the output layer result. The error is calculated and propagated back through all the layers until it reaches the input layer. Next the weights are updated according to the update rule and in the end the cost function is calculated. If the estimated error is smaller than a set threshold, the training can be concluded.

2.4 Practical considerations

There are certain practical elements that must be considered before a deep neural network can be implemented. In the section above there are several parameters that have not been discussed properly. It was mentioned that the number of hidden layers can, in theory, be arbitrarily large and the same goes for the hidden units in each respective layer. The size of a network depends on the application, data set, and computational power and there is no clearly defined optimal size.

A general idea is to use as few hidden units in each layer as possible to limit the load on the CPU during simulations. The number is usually determined by a combination of experience and testing on the actual system. If the network fails to converge, the number of units required might be larger than what was attempted. However, if the network does converge, one might try fewer units to optimize for the computational load. If one examines the weights regularly while training, it might be possible to spot units that are not participating much in the training. This would be units that do not change much in value from the initial values. It is also important to consider the chance for over- and under-fitting. The number of units that can be trained is directly related to the size of the training set. An upper bound is thus put on the number of units and exceeding this will cause under-fitting. Over-fitting is a result of training a network for too long or with too many training examples compared to hidden units, in particular for Autoencoders, or with examples that are not general enough. This will lead to poor generalization because it adjusts to very specific features in the set and the performance on unseen data which deviate substantially from the training set will be poor.

It is obvious that the selection of training data is important and should be considered while deciding for other parameters in the network. In general, one can use as many training examples as are available, but this may not be efficient nor ideal. In many cases a small dataset is all that is needed for convergence of the network. A neural network generalizes well, i.e. it learns to ignore irrelevant data and learns the key features quickly. If the input to the network after training might contain noise, one should include some noisy inputs during training. In fact, including some noisy inputs might help the network to converge even if no noise is expected [16]. It is important to note that a neural network does not extrapolate well, meaning that if the network has received an insufficient set of training data without certain important features, the classification result may be unreliable. As with the size of a network, the size of the training data and the content of the training set must be decided for empirically unless there are some constraints posed on the network. In several applications, for instance classification, it is imperative for the training examples belonging to the different classes to be applied at random. If the network is trained for each class sequentially, it will only ”learn” the last class and ”forget” the previous classes.

Next step after the network size has been determined is to initialize the weights and

choosing learning parameters. There are many ways the weights can be initialized, but they are typically set to small values, e.g. ± 0.5, and usually chosen at random. As opposed to the weights, the choice of learning rate is critical for the network performance. If the value is chosen too large, the network might not settle for a solution because the large step size may cause the network to bounce back and forth. Due to this, the value is chosen small, typically in the order of 0.5 to 0.25. This might cause the network to take a long time to converge, but given a large enough training set, it will converge.

Since the update rule applies a gradient descent method, there is a large pitfall for the algorithm. As the algorithm descends down the slope in weight space it might encounter a local minimum which, if the learning rate is too small, it will not be able to get out of. If the error at this point is smaller than the set threshold this does not matter, but if not, there are a few ways to solve the problem. The easiest solution is to reset the network with new weights and hope that the algorithm will converge to a sufficient solution. At the same time it might be worth considering a change of system parameters. This may be time consuming and no matter how many times the algorithm is run with new initial values, there is no way to guarantee that the solution will converge to the global minimum. In 2006 G. E. Hinton et. al. observed that it is possible to train a neural network greedily, one layer at a time, where each of these layers are trained as a Restricted Boltzmann Machine (RBM) where the output of each layer acts as the visible layer of the next. The idea with applying this method is to move the network to an area in weight space that is closer to the global optimum by training it in an unsupervised manner and later fine-tuning the network with the backpropagation algorithm.

2.5 Energy-based models

An RBM is a so called energy-based model. These models associate an energy, a scalar value, to each configuration of the variables one wants to optimize. Training such a model involves modifying the function which determines the energy so that it has certain properties. For neural networks we want desirable configurations to have low energy and vice-versa. In the case of energy-based probability functions one defines the probability distribution through an energy function:

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{Z} \tag{2.19}$$

where:

$$Z = \sum_{x} e^{-E(x)} \tag{2.20}$$

and $E(\mathbf{x})$ being the energy function. To relate this function to a neural network we introduce a hidden part of the function $\mathbf{h}$ and a visible part $\mathbf{x}$. By principles of marginalization we can write the probability distribution as:

$$P(\mathbf{x}) = \sum_{\mathbf{h}} P(\mathbf{x}, \mathbf{h}) = \sum_{\mathbf{h}} \frac{e^{-E(\mathbf{x},\mathbf{h})}}{Z} \tag{2.21}$$

We define a new variable which we will call free energy, inspired from physics:

$$F(\mathbf{x}) = -log \sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})} \tag{2.22}$$

Inserting 2.22 into 2.21 we get:

$$P(\mathbf{x}) = \frac{e^{-F(\mathbf{x})}}{Z} \tag{2.23}$$

where:

$$Z = \sum_{\mathbf{x}} e^{-F(\mathbf{x})} \tag{2.24}$$

Training such a model involves minimizing the energy-based probability function by applying stochastic gradient descent. The negative log-likelihood gradient of $P(\mathbf{x})$ can be calculated to be [11]:

$$\frac{\partial log P(\mathbf{x})}{\partial \theta} = -\frac{\partial F(\mathbf{x})}{\partial \theta} + \sum_{\tilde{\mathbf{x}}} P(\tilde{\mathbf{x}}) \frac{\partial F(\tilde{\mathbf{x}})}{\partial \theta} \tag{2.25}$$

where $\tilde{x}$ denotes values drawn from the model distribution and θ denotes the model parameters.

2.5.1 Boltzmann Machines

The Boltzmann machine is a type of energy-based model which includes hidden variables. The energy function for these models is defined as:

$$E(\mathbf{x}, \mathbf{h}) = -\mathbf{b}'\mathbf{x} - \mathbf{c}'\mathbf{h} - \mathbf{h}'W\mathbf{x} - \mathbf{x}'U\mathbf{x} - \mathbf{h}'V\mathbf{h} \tag{2.26}$$

where b is the bias in the input layer, c is the bias in the hidden layer, W are the weights connecting the hidden and visible units, U are the weights between intra-connections in the visible layer and V are the weights between intra-connections in the hidden layer. The gradient of the log-likelihood of the Boltzmann Machine can be written as [11]:

$$\begin{aligned}
\frac{\partial P(\mathbf{x})}{\partial \theta} &= \frac{\partial log \sum_h e^{-E(\mathbf{x},\mathbf{h})}}{\partial \theta} - \frac{\partial log \sum_{\tilde{\mathbf{x}},\mathbf{h}} e^{-E(\tilde{\mathbf{x}},\mathbf{h})}}{\partial \theta} \\
&= -\frac{1}{\sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})}} \sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})} \frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \theta} \\
&+ \frac{1}{\sum_{\tilde{\mathbf{x}},\mathbf{h}} e^{-E(\tilde{\mathbf{x}},\mathbf{h})}} \sum_{\tilde{\mathbf{x}},\mathbf{h}} e^{-E(\tilde{\mathbf{x}},\mathbf{h})} \frac{\partial E(\tilde{\mathbf{x}}, \mathbf{h})}{\partial \theta} \\
&= -\sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{x}) \frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \theta} + \sum_{\tilde{x},\mathbf{h}} P(\tilde{x}, \mathbf{h}) \frac{\partial E(\tilde{x}, \mathbf{h})}{\partial \theta}
\end{aligned} \tag{2.27}$$

which gives us the average log-likelihood gradient over the training set as:

$$E_{\hat{P}}\left[\frac{\partial log P(\mathbf{x})}{\partial \theta}\right] = -E_{\hat{P}}\left[\frac{\partial E(\mathbf{x})}{\partial \theta}\right] + E_P\left[\frac{E(\mathbf{x})}{\partial \theta}\right] \tag{2.28}$$

where $\hat{P}$ is the training set empirical distribution and P being the model distribution. From this it is clear that if we can sample from the model distribution, we have a Monte-Carlo method to obtain a stochastic estimator of the log-likelihood gradient.

2.5.2 Gibbs sampling

To be able to obtain the estimator of the log-likelihood we need a method to sample from the model itself. The solution for this is called Gibbs sampling which is a Markov Chain Monte Carlo (MCMC) algorithm. For a fully connected Boltzmann Machine, Gibbs sampling is slow. However, due to the factorization RBMs provide, we do not need to sample in the positive phase because the gradient is calculated analytically. Because of this, a very efficient Gibbs sampling can be performed and by starting it from a training vector one can ensure convergence in few steps. For k Gibbs steps, starting from a training example, i.e. sampling from $\hat{P}$, and using the conditional probabilities obtained above, the chain will look as follows [11]:

$$\begin{aligned} \mathbf{x_1} &\sim \hat{P}(\mathbf{x}) \\ \mathbf{h_1} &\sim P(\mathbf{h} \mid \mathbf{x}_1) \\ \mathbf{x_2} &\sim P(\mathbf{x} \mid \mathbf{h}_1) \\ \mathbf{h_2} &\sim P(\mathbf{h} \mid \mathbf{x}_2) \\ &\vdots \\ \mathbf{x}_{k+1} &\sim P(\mathbf{x} \mid \mathbf{h}_k) \end{aligned} \tag{2.29}$$

where $\mathbf{x}_{k+1} = \tilde{x}$ as seen in (2.27). From the first training sample, calculate the conditional probability and draw the first hidden vector from the distribution. From this, calculate the conditional probability for the input vector from the hidden vector sample and sample the second input vector. Alternate between sampling input vector and hidden vector until convergence. The reasoning for starting with a training vector is that as the model learns, it will capture the features in the training set; and the model distribution P and the training distribution $\hat{P}$ will become more similar in a statistical sense. Hence, starting with a training vector will ensure a quicker convergence in terms of steps.

2.5.3 Contrastive Divergence (CD)

Contrastive Divergence is an approximation of the log-likelihood gradient which uses Gibbs sampling. It has been shown that this method is a successful update rule for training RBMs [17]. With CD we replace the expectation in (2.28) over all possible configurations with one sample obtained from the Gibbs chain. This

approximation adds some extra variance due to only taking one or a few MCMC samples instead of doing the complete sum. However, this does not hurt much as long as it is comparable or smaller than the variance added due to online gradient descent. It is still very expensive to run a long MCMC chain even though we only run a few and thus the idea of a k-step CD (CD-k) was introduced which involves another approximation. The general idea is to stop the Gibbs chain before convergence, i.e. after k steps. The update rule obtained from CD-k is given by:

$$\begin{aligned} \Delta W_{ij}^0 &\propto \langle v_i h_j \rangle_0 - \langle v_i h_j \rangle_k \\ \Delta b_i^0 &\propto \langle v_0 \rangle_0 - \langle v_i \rangle_k \\ \Delta b_i^1 &\propto \langle h_i^1 \rangle_0 - \langle h_i^1 \rangle_k \end{aligned}$$

where:

- $\langle \cdot \rangle_0$ denotes the average when the visible units take input values and the hidden units are sampled from their conditional distribution (2.33).
- $\langle \cdot \rangle_k$ denotes the average after applying input data values on the visible layer and performing a k-step Gibbs sampling to each layer.

Performing only k steps introduces a certain bias to the gradient, but we know that as $k \to \infty$, the bias goes to zero. As mentioned earlier, when the model distribution is close to the empirical distribution from the training set, convergence will occur after few steps. When we start the chain from a training vector, the MCMC has already converged and thus we only need one step to obtain an unbiased sample from P. A surprising result found in [17] is that even CD-1 often gives good results. They show that a larger k gives a more precise result, but the gain from increasing k is not always worth the increased computational cost as very good approximations can be obtained with CD-1.

2.5.4 Restricted Boltzmann Machines (RBM)

In this report we will only focus on the binary case. For more detailed derivations and information about the non-binary case, the reader is referred to [10][18]. The

Restricted Boltzmann Machine is a variant of the Boltzmann Machine where matrices U and V from (2.26) are null, i.e. no intra-layer connections are allowed, only inter-layer connections. The free energy for such a model is given by:

$$E(\mathbf{x}) = -\mathbf{b}'\mathbf{x} - \sum_i log \sum_{\mathbf{h}_i} e^{\mathbf{h}_i(\mathbf{c}_i + W_i\mathbf{x})} \tag{2.30}$$

where W_i is the row vector corresponding to the ith row of W. By using a factorization trick shown in [11] we can readily derive an expression for the conditional probability:

$$\begin{aligned} P(\mathbf{h}|\mathbf{x}) &= \frac{\exp(\mathbf{b}'\mathbf{x} + \mathbf{c}'\mathbf{h} + \mathbf{h}'W\mathbf{x})}{\sum_{\tilde{\mathbf{h}}} \exp(\mathbf{b}'\mathbf{x} + \mathbf{c}'\tilde{\mathbf{h}} + \tilde{\mathbf{h}}'\mathbf{W}\mathbf{x})} \\ &= \frac{\prod_i \exp(\mathbf{c}_i\mathbf{h}_i + \mathbf{h}_i W_i\mathbf{x})}{\prod_i \sum_{\tilde{\mathbf{h}}_i} \exp(\mathbf{c}_i\tilde{\mathbf{h}}_i + \tilde{\mathbf{h}}_i W_i\mathbf{x})} \\ &= \prod_i \frac{\exp(\mathbf{h}_i(\mathbf{x}_i + W_i\mathbf{x}))}{\sum_{\tilde{\mathbf{h}}_i} \exp(\tilde{\mathbf{h}}_i(\mathbf{c}_i + W_i\mathbf{x}))} \\ &= \prod_i P(\mathbf{h}_i|\mathbf{x}) \end{aligned} \tag{2.31}$$

From the energy function it is obvious that $\mathbf{x}$ and $\mathbf{h}$ play a symmetric role and from a similar derivation one can calculate the conditional probability for $\mathbf{x}$ given $\mathbf{h}$ to be:

$$P(\mathbf{x}|\mathbf{h}) = \Pi_i P(\mathbf{x}_i|\mathbf{h}) \tag{2.32}$$

In the binary case, i.e. $h \in \{0, 1\}$, we see by insertion that we obtain the usual neuron equation in both cases and can express the conditional probabilities as:

$$P(\mathbf{h}_i = 1|\mathbf{x}) = \frac{e^{\mathbf{c}_i + W_i\mathbf{x}}}{1 + e^{\mathbf{c}_i + W_i\mathbf{x}}} = \text{sigm}(\mathbf{c}_i + W_i\mathbf{x}) \tag{2.33}$$

$$P(\mathbf{x}_i = 1|\mathbf{h}) = \text{sigm}(\mathbf{b}_j + W'_{.j}\mathbf{x}) \tag{2.34}$$

where $W_{.j}$ is the jth column of W.

2.5.5 Gaussian-Bernoulli RBM (GBRBM)

With the binary units introduced for RBMs, one has to "cheat" in order to handle continuous-valued inputs. When scaling the input vector to the interval (0,1) by subtracting the training set mean for each respective input neuron, the input continuous value can be used as the probability for a binary random variable to take the value 1. This has been a useful approach for pixel gray levels, but may be inappropriate for other signals which are less coarsely quantized. The sampling introduces a lot of noise and is thus not appropriate for the experiments documented in this report. An alternative approach is the use of Gaussian-Bernoulli RBMs [19]. The difference between the GBRBM and the common RBM is that the input to the RBM follows a Gaussian distribution. In both types of RBMs, the hidden layer consists of normal binary units which follow a Bernoulli distribution.

To obtain Gaussian distributed units, the binary visible units are replaced by linear ones with independent Gaussian noise. Doing so adds a quadratic expression to the energy and the energy function is modified to be[18]:

$$E(\mathbf{v}, \mathbf{h}) = \sum_{i \in vis} \frac{(v_i - a_i)^2}{2\sigma_i^2} - \sum_{j \in hid} b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij} \tag{2.35}$$

It is in essence possible to learn the variance of the noise for each visible unit, but this is difficult using CD-1 and it is thus common to z-score normalize the input, i.e. subtract the mean and divide by the standard deviation, estimated over the entire input dataset for each respective neuron, leaving them with zero mean and unit variance. By inserting the zero mean and unit variance, and converting the energy equation to a matrix equation, the energy reduces to [5]:

$$E(\mathbf{v}, \mathbf{h}) = \frac{1}{2}(\mathbf{v} - \mathbf{b})^T(\mathbf{v} - \mathbf{b}) - \mathbf{v}^T W \mathbf{h} - \mathbf{c}^T \mathbf{h} \tag{2.36}$$

Similar to the standard RBM, the conditional probability factorizes to become:

$$P(v_j|\mathbf{h}) \sim \mathcal{N}(-\mathbf{b} - W_{.j}\mathbf{h}, 1) \tag{2.37}$$

The reconstruction value of the Gaussian visible units is then equal to their down-propagated input plus its bias, which is normal distributed with 0 mean and unit

variance. Instead of sampling from the Gaussian distribution, using the mean vectors as visible activations is commonly used. This is due to the fact that the standard deviations are not updated, and thus, the samples of the visible units are either dominated by the noise or affected only slightly by the standard deviations. It is important to note that the learning rate when training GBRBMs is required to be 1-2 orders of magnitude smaller due to the fact that there is no upper bound to the size of a component in the reconstruction. If one component becomes very large, the weights connected to the unit will receive a large learning signal and thus has to be countered by the smaller learning rate.

2.5.6 Deep Belief Networks (DBN)

A Deep Belief Network [10] is a deep generative architecture built in a layer-wise manner where each layer consists of stochastic binary units. The bottom layer is a visible layer denoted $\mathbf{v}$, while the rest of the layers are hidden layers denoted $\mathbf{h}^1, \mathbf{h}^2, ..., \mathbf{h}^n$. The top two layers $(\mathbf{h}^{l-1}, \mathbf{h}^l)$ are connected in an undirected manner and form an RBM while the rest of the layers are connected in a top-down directed fashion. In practice, each layer of a DBN shares parametrization with an RBM and one can thus visualize a DBN as a stack of RBMs.

The probability of activation of a unit in any layer below the top two is conditionally independent of the layers below given the state of the layer directly on top of it due to the top-down directed structure. The joint probability of a DBN thus factorizes as:

$$P(\mathbf{v}, \mathbf{h}^1, ..., \mathbf{h}^n) = \left(\prod_{0}^{l-2} P(\mathbf{h}^k | \mathbf{h}^{k+1}) \right) P(\mathbf{h}^{l-1}, \mathbf{h}^l) \tag{2.38}$$

where $\mathbf{v} = \mathbf{h}^0$, $P(\mathbf{h}^k|\mathbf{h}^{k+1})$ is a visible-given-hidden conditional distribution in an RBM (2.34) associated with layer k of the DBN, and $P(\mathbf{h}^{l-1}, \mathbf{h}^l)$ is the joint distribution of the top level RBM. Due to these properties, sampling from a DBN can be done by performing Gibbs sampling on the top two layers, alternating until the stationary state has been reached. Furthermore, we can propagate the activations down by performing ancestral sampling until we reach the visible layer by using equation (2.34), and obtain the DBN sample.

Training a DBN is done in a greedy layer-wise manner. The DBN will be initialized with only two layers which are connected in an undirected manner as they are the two top layers and thus comprise an RBM. The net is trained as an RBM in the usual unsupervised manner. When training is complete, the parameters are fixed and the connections are transformed into a top-down connectivity. Next, a new layer is appended on top of the net and connected to the previous top layer with undirected connections, again making the top two layers comprise an RBM. These two top layers are, again, trained as an RBM, however with a twist. The input data to the *visible* layer of the RBM, i.e. the previous top layer, is obtained by passing the input data through the previous layers of the DBN using equation (2.33). This process is repeated until the desired number of layers is reached.

2.5.7 Using a DBN for regression

The training method outlined in section 2.5.6 is a purely unsupervised training method and can thus not be used directly for regression or classification. The reasoning for training a DBN for regression, however, is that through the unsupervised training, the DBN learns the parameter space topography. The goal is to obtain a model, which captures a representation of the data which corresponds to higher levels of abstraction. There are many ways a DBN can be transformed into a classification or regression architecture, but only a regression architecture will be utilized in this work.

As mentioned in section 2.4, a major issue with DNNs is that the popular training method, gradient descent, is susceptible to local minimum in the parameter space topography. A solution is to initialize the DNN as a DBN before performing gradient descent and thus enhancing performance significantly [10]. The transformation from a DBN to a DNN performing regression is simple. A linear regression layer is added on top of the DBN and the DBN weights and biases are used as initial parameters for the DNN. The top linear regression layer poses a linear optimization problem and can be trained, for example, by using the pseudo-inverse method:

$$\begin{aligned} \mathbf{b}^y &= \bar{y} \\ W^y &= \left(\mathbf{h}^n \mathbf{h}^{n^T}\right)^{-1} \mathbf{h}^n \mathbf{y}^r \end{aligned}$$

where $\mathbf{b}^y$ represents the bias of the output layer units, $\bar{y}$ the mean of the outputs in the dataset, W^y the weights from the top hidden layer to the output layer, $\mathbf{h}^n$ the top hidden layer states for all datapoints in the dataset, and y^r the mean-subtracted outputs in the dataset.

Since the DBN was initially trained in an unsupervised manner, the performance is typically not very good and further training is needed. This fine-tuning can be done using the back-propagation algorithm, i.e. gradient descent.

2.6 MFCC, Deltas and Delta-Deltas

Mel-frequency cepstral coefficients (MFCCs) are representations of the short-term power spectrum of a signal. The estimation of these coefficients can be summed up in four steps

1. Calculate the fourier transform of a windowed (short-term) signal
2. Map the powers into the mel-scale, typically by using overlapping triangular windows
3. Take the logarithm of the powers at each mel-frequency
4. Calculate the discrete cosine transform of the mel log powers

These coefficients are widely used in speech processing because they capture the spectral content of the signal which can be used to create a model for speakers.

The cepstral features provide static information for the signal, but in many applications, speech processing in particular, it is useful to look at the dynamic information as well. As the MFCCs are discrete, the exact derivative of the function does not exist, however there exists several approximations to this. The first and second derivative of the MFCCs are called Deltas and Delta-Deltas respectively. A widely used equation for the Deltas estimation is as follows

$$\mathbf{d}_t = \frac{\sum_{k=-K}^{K} k \cdot \mathbf{c}_{t+k}}{2\sum_{k=-K}^{K} k^2} \tag{2.39}$$

It is important to note that the MFCCs are not replaced by the Deltas and Delta-Deltas, but come in addition. This provides information about time-dependencies in the first and second order. The Delta-Deltas can be calculated in several different ways, one being the use of equation 2.39, but in many cases a simpler approach is used like the one below

$$\mathbf{dd}_t = \mathbf{d}_{t+m} - \mathbf{d}_{t-m} \tag{2.40}$$

2.7 Linear Discriminant Analysis (LDA)

The goal of LDA is to reduce the dimensionality while preserving as much class discriminatory information as possible. It tries to find a linear combination of features which characterize or separate classes. More formally, given a number of independent features describing the data, LDA creates a linear combination of these to maximize the separation of classes with respect to a criterion. The criterion used in LDA is the difference between the means of each class. We define two measures [20], the first being the *within-class* scatter matrix given by

$$S_w = \sum_{i=1}^{c} \sum_{j=1}^{N_j} (x_j^i - \mu_i)(x_j^i - \mu_i)^T \tag{2.41}$$

where x_j^i is the jth sample of class i, μ_i is the mean of class i, c is the number of classes and N_i is the number of samples in class i. The second measure is the *between-class* scatter matrix given by

$$S_b = \sum_{i=1}^{c} (\mu_i - \mu)(\mu_i - \mu)^T \tag{2.42}$$

where μ is the mean over all classes. Finally, the goal of the LDA is to maximize the between-class measure while minimizing the within-class measure. There are several ways to execute this optimization, but one way to do it is by maximizing the ratio given by [21]

$$U_{opt} = \arg\max_U \frac{\mid U^T S_b U \mid}{\mid U^T S_w U \mid} \tag{2.43}$$

The solution matrix U is a set of generalized eigenvectors of S_b and S_w. In many cases there is an issue with obtaining singularity and Principle component analysis (PCA) can be performed as a pre-processing step before applying LDA.

2.8 Hidden Markov Models(HMMs)

Many speech recognition systems represent their acoustic models based on HMMs. These are statistical Markov models where the system is assumed to be a Markov model with unobserved or *hidden* states [22]. The states contain a probability distribution describing the state value and the states are linked together with transition probabilities. The probability densities are assumed to be Gaussian Mixture Models (GMMs) which are in essence a sum of Gaussian distributions with different parameters. A general GMM with parameters given by $\Theta = \{c_j, \mu_j, \Sigma_j\}_{j=1}^{M}$ is of the form

$$p(x \mid \Theta) = \sum_{j=1}^{M} c_j N(x; \mu_j, \Sigma_j) \tag{2.44}$$

In the case of speech recognition, the HMM is used to estimate the most likely path through the network of states given the input signal. Before an HMM can be used for recognition it must be trained for the task at hand. Learning for an HMM is essentially modifying the GMM parameters and transition probabilities to obtain the best sequence of states for the given input. This task is usually done by estimating the Maximum Likelihood (ML) of the HMM parameters given the output sequence. Estimating the ML is typically done using the Expectation-Maximization (EM) algorithm to optimize the *local* maximum likelihood.

2.9 Maximum Likelihood Linear Transform (MLLT)

In general, optimizing for a larger set of variables is a larger computational burden and requires more training data. As mentioned in 2.8, the Gaussian distributions comprising the GMMs are determined by their mean value and covariance matrix Σ. In most applications, these are assumed to be of a diagonal form due to lower computational burden and robust parameter estimation. The MLLT is a

representation which allows several distributions to share a few covariance matrices instead of having one covariance matrix for each distribution, thus lowering the dimensionality of the parameter space. Instead of a distinct covariance matrix for each distribution, each distribution consists of two elements; a non-singular linear transform matrix W^T which is shared across several distributions and the diagonal elements in the matrix Λ_j. The inverse of the covariance matrix for each distribution can then be represented as

$$\Sigma_j^{-1} \approx W \Lambda_j W^T \tag{2.45}$$

The EM algorithm mentioned in section 2.8 can be extended to include these parameters in the optimization process. The effect of the MLLT is a clustering of statistically similar states; in speech recognition these states can be different phones that are statistically similar. This leads to a reduction in feature space which is generally more easily trained. Experiments comparing MLLT to other linear transforms for speech recognition purposes can be seen in [23].

2.10 Speaker Adaptive Training (SAT)

We can separate speech recognition in three categories: Speaker independent (SI), which aims to recognize speech from any speaker. Speaker dependent (SD), which has models for unique speakers. Speaker adaptive (SA), which builds on an SI system, but can achieve comparable results to SD with less speaker specific training data. In speaker adaption, a linear transform is found so to maximize the likelihood of the adaption data. This is done by applying the transform to the Gaussian means, or as in feature-space Maximum Likelihood Linear Regression (fMLLR), to the feature vectors themselves.

The main difference between standard MLLR and fMLLR is that MLLR adapts the model while the fMLLR adapts the observation vectors. It can however be shown that applying so called *constrained* MLLR on the model parameters is equivalent to applying fMLLR on the feature vectors when considering the Gaussian likelihoods and we will thus only define the cMLLR in this setting. The constraint imposed

on the standard MLLR is that the transformation applied on the variance must correspond to the transform applied on the mean. This results in the general formulation of cMLLR given by the update functions

$$\hat{\mu} = A'\mu - b' \tag{2.46}$$

and

$$\hat{\Sigma} = A'\Sigma A'^{T} \tag{2.47}$$

A more detailed description, the solution for the problem and a simple recipe for converting cMLLR to fMLLR can be found in [24]. The estimation of the linear parameters can be done using the EM algorithm.

2.11 Articulators

Human speech production is based around pressing air through a non-uniform pipe and changing the shape of the pipe affects the produced sound. In general, the sounds can be divided into two main groups, namely voiced and unvoiced sounds. A sound lands in the category voiced if its production involves vibration of the vocal cords. When air has passed through the vocal cords, the sound produced is shaped by the different speech organs, or *articulators*, in the vocal tract. The particular configuration of these articulators determines which sound is produced. Even though two different sounds may sound similar to a listener, e.g. the nasals */n/* and */m/*, the configuration of articulators may be differing significantly. In this case a major difference is the lip and tongue positions. For */m/*, the lips are closed and the tongue touches the lower incisors, while for */n/*, the lips are apart and the tongue touches the upper incisors as well. As can be seen in chapter 4, the main confusion pairs for a phone recognition system are sounds which are close both in temporal and spectral content. As just explained, information about the articulator positions can help distinguish these pairs and a system which can detect this information may boost the performance of regular speech recognition systems by creating a hybrid system.

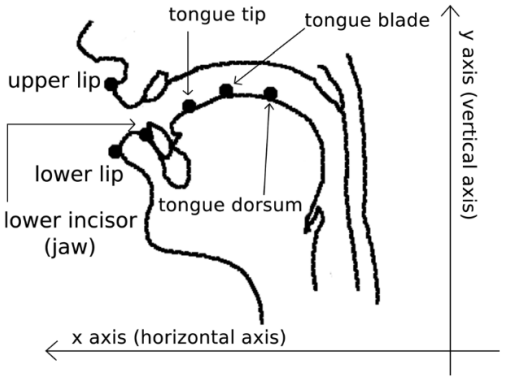

FIGURE 2.5: Positioning of sensors on articulators seen on the midsagittal plane. Image borrowed from [4]

2.11.1 Electromagnetic Articulography (EMA)

Electromagnetic Articulography is a technique where electromagnetic transducer coils are mounted to the articulators of interest to record precise measurements of their position. This information is recorded in parallel with recording of the waveform to provide synchronous acoustic and articulatory information. The position is acquired by the transmitter coils generating alternating magnetic fields which in turn induce alternating voltages in receiver coils. These receiver coils are connected to a system which processes these voltages and provides measures of transducer locations in three dimensions as a function of time. The tissue inside the vocal tract allows a certain degree of movement of the transducers and they may also change shape. This can cause the receiving- and transmitting transducers to undergo a rotational misalignment which must be accounted for to avoid erroneous data. This is typically done by using either two biaxial transducers or three single-axis transducers as reference. For more detailed information, the reader is referred to [25].

There are many different articulators in the human vocal tract which all influence the sound produced to a certain degree. Some of these are either impractical to measure, contain mutually redundant information with other articulators, or do not move significantly. In many cases, only the most active articulators are of interest and in this work only six will be considered; upper lip (UL), lower lip, (LL), lower incisor (LI/JAW), tongue tip (TT), tongue blade (TB), and tongue dorsum (TD). In addition, only the midsagittal plane will be considered due to the very small movements in the z-direction, i.e. perpendicular to the midsagittal plane. The recording setup can be seen in figure 2.5.

Chapter 3

Coding libraries and databases

There exists several different libraries for implementing neural networks. Many of these are not designed specifically for neural networks, but the general mathematical structure of neural networks allows them to be implemented efficiently by these libraries. Desired features in such libraries are the ability to do matrix operations efficiently, efficient ways of computing gradients with respect to a high number of variables, and ways to speed up computation, e.g. the ability to use the GPU. For the purposes of this project, several different databases and libraries were used due to the varying nature of the experiments. Some of the sections in this chapter are recaps of the respective papers or information from the respective webpages and although small rephrasing may occur, the entirety of the information is credited the respective author as referenced in the title.

3.1 Kaldi[1]

Kaldi is a speech recognition toolkit and is written in C++ [26]. It is similar in aim and scope as the more widely known Hidden Markov Model Toolkit (HTK). Kaldi contains a large range of features, but some of the more notable are as follows: Integration with Finite State Transducers(FST) by using the OpenFst toolkit as a library, linear algebra support by including a matrix library, essentially providing a C++ wrapper for standard Basic Linear Algebra Subprogram (BLAS) and Linear Algebra Package (LAPACK) routines and, as far as possible, provide algorithms in the most generic form possible. The main goal of Kaldi is to provide complete recipes for building speech recognition systems. The developers of Kaldi aim to

avoid limiting the toolkit to speech recognition and also to create recipes that are not dataset dependent. It has also been put emphasis on making the toolkit easy to reuse and refactor by having loose coupling between directories. In short, the aim is that any given header should need to include as few other header files as possible. Kaldi also allows for computation using NVIDIA produced GPUs through CUDA. CUDA stands for Compute Unified Device Architecture and is a parallel computing platform and application programming interface (API) model created by NVIDIA. This allows developers to use the GPU for general purpose processing which dramatically increases computing performance.

By installing Kaldi, one also installs a large library of example directories. These consist of high-level shell scripts which execute several different speech recognition algorithms step by step on pre-defined datasets which have to be supplied by the user. These examples are good tools to learn how a program is built in Kaldi and to get experience with the toolkit. Furthermore, one can reuse code from these examples to implement new algorithms or do modifications to tailor them for a certain task, e.g. to be able to input different datasets.

3.2 DeepLearnToolbox

The DeepLearnToolbox is a MATLAB/Octave toolbox for deep learning developed by Rasmus Berg Palm [27][28]. The toolbox includes deep belief networks, stacked autoencoders, convolutional neural nets, convolutional autoencoders, and vanilla neural nets. In addition, each method has example codes to help understand how to use it. The dataset used in the examples is the common MNIST database of handwritten digits and is used for classification purposes. The implementation is fairly straight-forward and utilizes an object oriented approach.

3.3 TIMIT[2]

TIMIT is an acoustic-phonetic continuous speech corpus designed as a combined effort among the Massachusetts Institute of Technology (MIT), Stanford Research Institute (SRI), and Texas Instruments inc. (TI). The corpus contains a total of 6300 utterances, 10 sentences spoken by each of the 630 speakers. The speakers are

chosen from each of the eight major dialects of American English. Each speaker has been assigned a unique speaker-ID, and each utterance has been assigned a unique utterance-ID. TIMIT includes time-aligned orthographic, phonetic, and word transcriptions as well as a 16-bit speech waveform, sampled at 16kHz, for each utterance. It is important to note that the wav-format used for encoding is the NIST/Sphere format. The data source is microphone speech and the recordings were done at TI, while the transcription was done at MIT. It has later been maintained, verified, and prepared for CD-ROM production by the National Institute of Standards and Technology (NIST). Full documentation can be found at the webpage of the Linguistic Data Consortium [2].

3.4 Norwegian acoustic-phonetic speech database (NAFTA)

The corpus contains a total of 4800 utterances, 20 sentences spoken by each of the 240 speakers. The 20 sentences are divided into three groups, 3 sentences which are read by all speakers, 12 sentences that are read by three unique speakers, and 5 sentences that are unique for each speaker. Both TIMIT and NAFTA suggest a test set and a developments set, however the labeling differs between the two. In TIMIT, each speaker has a unique ID, but this is not the case in NAFTA. Each utterance is marked with geographical region, gender, and age group. Within each age group, a speaker is labeled with an internally unique value, but non-unique across age groups. Furthermore, each utterance is marked to tell if it belongs to the test set or the development set, which of the three groups of sentences it belongs to and finally a unique sentence ID. In addition to the difference in labeling, the two databases also differ in phonetic inventory. While TIMIT limits itself to transcriptions with allophones, NAFTA utilizes the Extended Speech Assessment Methods Phonetic Alphabet (X-SAMPA) standard which includes symbols that are not allophones, e.g. "@" and "}", which give rise to problems in the recipe provided by KALDI. It also includes information about intonation which is not supported by the script. The wav-format used for encoding in this database is the Microsoft and IBM audio file format standard.

3.5 Mngu0[3]

This corpus will ultimately consist of a collection of multiple sources of articulatory data acquired from a single speaker: electromagnetic articulography (EMA), audio, video, volumetric MRI scans, and 3D scans of dental impressions. In this work we will only use the EMA and audio data and these are thus the only data discussed in this context. For a more detailed description of the database, the reader is referred to [3]. At the time of writing, there is only a *Day 1* subset available. This subset contains 1354 utterances, giving a total of approximately 67 minutes of speech data, discounting initial and final silences. The sentences were collected from newspaper text chosen with an algorithm which aimed to maximize coverage of context-specific diphones in as few sentences as possible. Sentence lengths range from 1 to 48 words, and comprise a variety of types, such as questions, statements, exclamations, etc.

The raw EMA data was recorded using the AG500 which allowed for tracking of sensors in 3D space with two angles of rotation, resulting in 5 measurements per sensor coil. In addition to these 5 coordinates, another 2 reliability indicators are added for each of the 12 coils used, giving a total size of 84 data points per sample. The database also provides a processed version of the EMA data. This subset contains only the x- and y-coordinates in the midsaggital plane. Hence, the processed version consists of 12 channels of EMA, sampled at 200 Hz, with initial and final silences omitted. These have been z-score normalized by subtracting the respective global mean from each channel, and then divided by 4 times each channel's respective global standard deviation.

In addition to the raw audio data which is captured using a Sennheiser MKH50 hypercardioid, the database comprises a processed version. The audio data has been converted to frequency-warped line spectral frequencies (LSF) [29] of order 40 plus a gain value. Furthermore, the audio has been lowpass filtered and downsampled to 16kHz. The frame shift used is 5 msec to match the EMA sample rate. Finally, the LSF feature vectors have also been z-score normalized in the same way as the EMA data.

Three subsets have been used in previous inversion mapping work using mngu0 data: A validation and test set, each containing 63 utterances, and a training set containing the rest. The list of which utterances have been used in each subset is available.

Chapter 4

Implementation and Results

4.1 TIMIT and NAFTA phone recognition

The experiments described in this section are based around the KALDI library. KALDI contains a folder with a wide range of example scripts or recipes which use the functionality of the KALDI library to perform speech recognition tasks on different databases. The recipe chosen for this project can be found in the "egs/-timit/s5" folder within the KALDI directory. The database used is the TIMIT corpus and speech recognition is performed using HMMs trained with a range of methods, in particular two different implementations of DNNs. The recipe is composed of a hierarchy of shell scripts which do file and folder handling, and file editing on the higher levels and calls the KALDI C++ functions at the lower levels to execute the algorithms used.

The codebases for the two neural networks are developed by two different authors. One is maintained by Karel Vesely while the other is maintained by Daniel Povey. Povey's codebase was originally based on an earlier version of Vesely's code, but has later been extensively rewritten. Vesely's code generally gives slightly better results, but runs on a single CPU card or a single CPU which makes it slow. Povey's code allows for use of multiple CPUs with multiple threads and even GPUs which allows for a lot higher run speeds. In these experiments, Povey's codebase has been used due to the substantial run-speed improvement which allows for quicker testing. The scope of these experiments is not to achieve the optimal phone error rates, but rather compare phone error rates for two databases composed from different languages, using the same recognition system.

The system can be partitioned into two main sections. The first section involves data, dictionary, and language preparation followed by feature extraction and initialization of the HMM. The second section consists of four different approaches to improve the performance of the HMM which is done sequentially. Each of the four algorithms are initialized with the results from the previous algorithm in the chain. The first step is training using deltas and delta-deltas of the MFCC coefficients. The second step involves training using Linear Discriminant Analysis (LDA) to reduce dimensionality, followed by Maximum Likelihood Linear Transform (MLLT), and the third step applies Speaker Adaptive Training (SAT). The final step in the training process initializes a deep neural network and trains it to estimate the posterior likelihoods for the HMM. A flow chart for the phone recognition system is shown in figure 4.1.

In order to use the same code for a different database one has to either modify the code to work with a database of another hierarchical structure or modify the database structure to match that of TIMIT. Due to the structure of the NAFTA database it was concluded that modifying the database structure would be the most efficient and least error prone way to proceed.

As mentioned in chapter 3, the wav-format for the two databases is different, varying both in sampling rate and encoding. To transform between different formats a command line tool called Sound eXchange (SoX) was used. TIMIT is organized as a hierarchy of folders starting with the test and train directories, each containing a folder for each group. In TIMIT these groups consist of speakers from each of the eight different major dialects of American English. Within each group there is a folder for each speaker containing the transcription files (.wrd, .phn, .txt) and sound file (.wav) for each of the utterances corresponding to the speaker. NAFTA however, is structured in a less hierarchical manner. All the transcription information for all the utterances in the database is stored in a single document. The general content in the transcription files is the same, but there are two important differences. The dictionary of phonemes is different and the time stamps used in NAFTA are displayed as milliseconds while the time stamps in TIMIT are displayed as sample number.

As mentioned above, the NAFTA transcription file contains the information to create three separate transcription files for each utterance. The .phn file contains the information of the phoneme transcription with corresponding time stamps for start and stop of each phoneme. The .wrd file contains the information of the

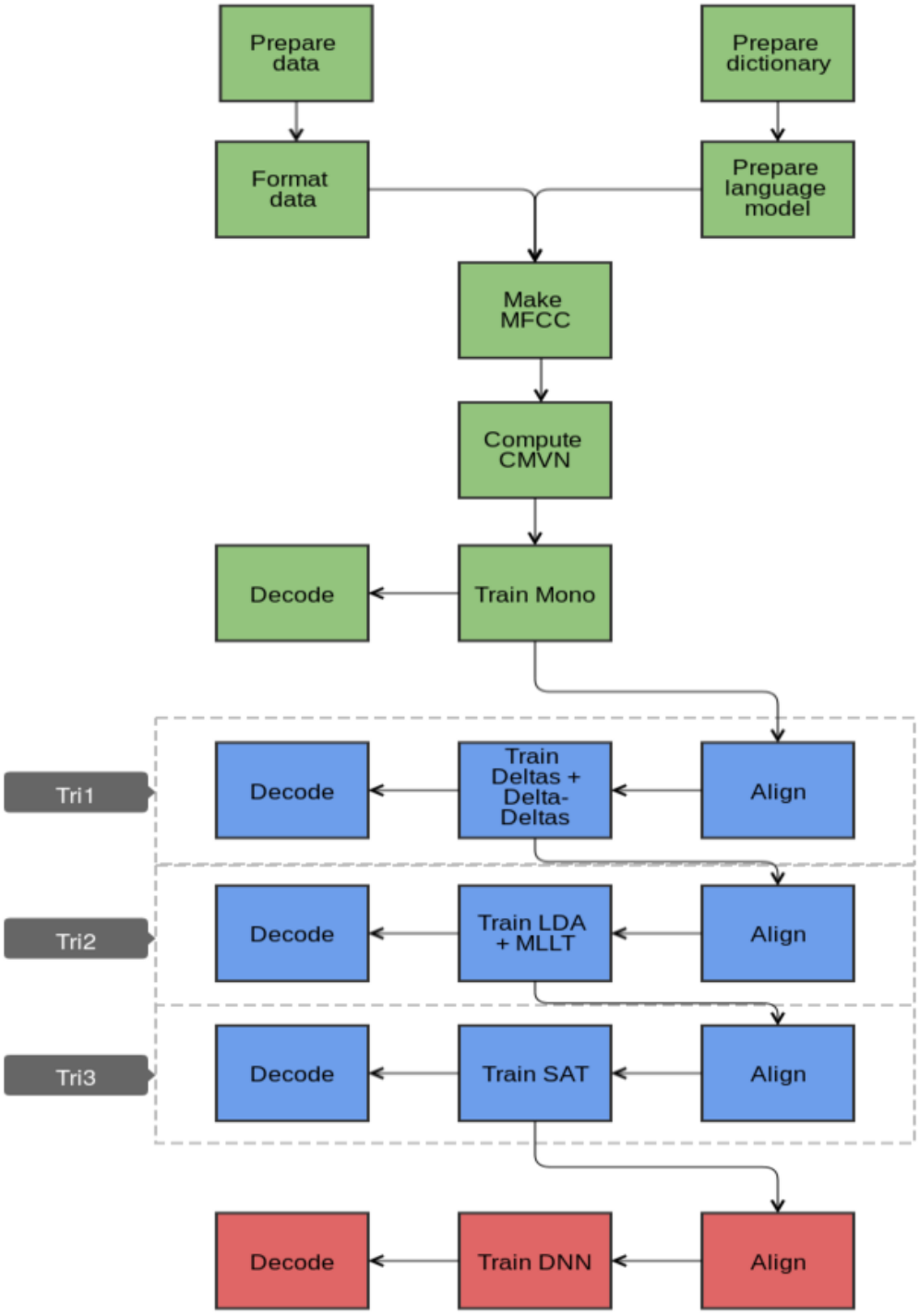

FIGURE 4.1: Flow diagram of the phone recognition system used for NAFTA and TIMIT experiments

word transcription with corresponding time stamps for start and stop of each word. The .txt file contains the information of the complete sentence with corresponding time stamps for start and stop of the sentence, including initial and terminating silence. As the Kaldi recipe is written with Shell scripts, certain modifications had to be done to the phonetic dictionary used in the NAFTA database due to complications with some of the punctuation used in the annotation. A smaller set of phonemes was drawn and the transcription file was translated using Perl scripts. Furthermore, the transcription file was separated into the three necessary files and the hierarchy of directories created using a combination of Perl and Shell scripts.

During the initial data preparation, the TIMIT dictionary is limited from 60 to 48 phonemes for training. At the time of scoring, the dictionary is further limited from 48 to 39 phonemes by introducing "don't-care" pairs. These are pairs which are considered equivalent at scoring. These dictionary size reductions are described in more detail in [30]. The same experiment was performed on the NAFTA database with two different initial dictionaries and two different limitation schemes. For the first experiment the dictionary was limited from 94 to 53 phonemes for training. The dictionary was not reduced further for scoring, i.e. 53 phonemes were used, and the results are shown in table 4.4. It became obvious from the results that the scoring dictionary was too large and a second experiment was run using an initial dictionary of 50 phonemes for training and a scoring dictionary of 39. The "don't-care" pairs used to reduce the size of the decoding dictionary are shown in table 4.1.

A	<==>	A:
ae	<==>	ae:
O	<==>	O:
e	<==>	e:
i	<==>	i:
oe	<==>	oe:
u	<==>	u:
Y	<==>	Y:
ou	<==>	ou:
tS	<==>	S
C	<==>	S
C	<==>	tS

TABLE 4.1: "Don't-care" pairs for NAFTA 50-39 experiment

4.1.1 Results and discussion

For this section three experiments were done, all varying either in terms of database, phonetic dictionary, or both. Two tables are shown below for each experiment, one displaying the different error rates, e.g. substitutions, deletions, and insertions, and another displaying the top 25 confusion pairs.

The first experiment was done on the TIMIT database with an original dictionary size of 48 and a decoding dictionary size of 39. These results were used as a baseline to be compared with the results obtained from test runs on the NAFTA database and can be seen in table 4.2 for the error rates and table 4.3 for the confusion pairs. The best phone error rate achieved on the TIMIT database was 20.4% at the seventh iteration. The main confusion pairs are different vowels and certain consonant pairs all of which are to be expected. The recognition is based on MFCC, i.e. the spectral information of the waveform and thus phones with similar spectral content will be harder to separate. Since all vowels are voiced sounds, they will appear similar in the frequency plane and even more so when uttered as part of a word due to the fact that people leave out parts of the phones and concatenate others in everyday speech. In addition to observing that vowels are confused with vowels, and consonants with consonant, there is a distinct pattern in confusion pairs within the consonant group. Stops are confused with stops, fricatives with fricatives, and so forth and within each of these groups the confusion is typically between the voiced and unvoiced version of bilabial, alveolar, velar, and dental sounds respectively. The *Err* column shows the percentage phone error rate at each iteration through the training set. It turns out that the error rate follows a convex curve with a minimum around iteration seven and increasing afterwards. This is an important discovery and shows clear signs of over-fitting.

The second and third experiment was done on the NAFTA database. This database was acquired in a similar manner to TIMIT, but the phonetic dictionary is different. In its original state, the NAFTA dictionary is composed of 94 phonemes including a glottal stop which is removed prior to testing. The first experiment was conducted without a reduction of dictionary size at time of decoding. The results show a substantially worse performance and the reason is obvious. In Norwegian language there are *long vowels* which means that several words can be written identically, but are pronounced differently in terms of the length of the vowel. In table 4.5 the long vowels are indicated by a ":" appended to the vowel. Except

timit_48_39	**Corr**	**Sub**	**Del**	**Ins**	**Err**	**S.Err**
Score_1	82.9	13.2	3.9	5.9	23	100
Score_2	82.9	13	4.1	5.0	22.1	100
Score_3	82.6	12.8	4.6	4.0	21.4	100
Score_4	82.3	12.7	5.0	3.5	21.2	100
Score_5	82.1	12.6	5.3	2.9	20.8	100
Score_6	81.8	12.6	5.6	2.4	20.6	99.5
Score_7	81.7	12.5	5.9	2.1	20.4	99.5
Score_8	81.2	12.6	6.2	1.8	20.6	99.5
Score_9	80.6	12.8	6.6	1.4	20.8	99.5
Score_10	80.3	12.4	7.3	1.3	21	99.5

TABLE 4.2: Decoding results after DNN training with TIMIT database as input

	Occurrences	Original		Confusion
1:	36	ah	==>	ih
2:	34	ih	==>	ah
3:	27	ih	==>	iy
4:	25	eh	==>	ih
5:	25	z	==>	s
6:	24	er	==>	r
7:	21	m	==>	n
8:	18	ah	==>	eh
9:	16	ae	==>	eh
10:	16	iy	==>	ih
11:	15	d	==>	t
12:	14	r	==>	er
13:	13	n	==>	m
14:	12	ah	==>	er
15:	12	eh	==>	ah
16:	12	s	==>	z
17:	11	ng	==>	n
18:	11	ow	==>	ah
19:	10	ih	==>	eh
20:	9	ah	==>	aa
21:	9	er	==>	ih
22:	8	b	==>	p
23:	8	eh	==>	ae
24:	8	ey	==>	ih
25:	8	g	==>	d

TABLE 4.3: TIMIT-48-39-confusion pairs

from the long vowels, the confusion pairs follow a similar pattern to the ones from the TIMIT experiments. The phones displayed in the *Original* column are the correct phones and the ones shown in the *Confusion* column are the ones inferred by the system. Table 4.4 shows the error rates and it is obvious that the phone error rates for this experiment also follow a similar curve across iterations, but the signs of over-fitting appear earlier and the system performs overall poorer.

To make the results more comparable, the dictionary for decoding was reduced to the same size as for the TIMIT experiments and the training dictionary was reduced by three entries. To reduce the decoding dictionary, a set of don't-care pairs were introduced as shown in table 4.1. The results from this experiment can be seen in terms of error rates in table 4.6 and in terms of confusion pairs in table 4.7. It is immediately obvious that the reduced decoding dictionary improved performance significantly. Again, the phone error rate follows the same curve and shows signs of over-fitting. Despite of the improved performance, the system is still 2.5 percentage points off the results of the TIMIT experiments. Because the experiments are performed on different databases which are recorded under different circumstances, a certain difference in performance must be expected. The main difference between the two databases, however, is that they contain data spoken in two different languages. Every language in the world can be divided into a set of phonemes, but the dictionary of phonemes and their statistical distribution varies for each language. This fact impacts a phone-recognition system greatly. As mentioned earlier, the occurrences of long vowels in Norwegian is a big issue for the recognizer which does not occur in English. It is obvious from the results that the confusion pairs are fairly predictable and the frequency of occurrence of these phonemes in each language will thus impact the recognition error accordingly. In addition, the size of the training, testing, and validation sets are significantly different which will affect the results.

The confusion pairs follow the same pattern as for the TIMIT experiment and the results thus suggest that the approach is viable for different languages, however it might require steps to achieve comparable performance. The results also display the obvious disadvantage a phone model encounters, which is the lack of ability to distinguish between phones that are similar both in waveform and spectral content. It is also important to note the clear case of over-fitting for all of the three cases. Over-fitting is seen as an iteratively larger phone error rate as training proceeds. In fact, for the experiment with the NAFTA database using 53 phonemes at decoding

time, over-fitting occurs already after the fifth epoch. It is highly likely that all the experiments will produce better results by applying the dropout method outlined in [31]. This allows the system to keep training for a longer time before over-fitting occurs and usually results in better generalization.

NAFTA_53_53	**Corr**	**Sub**	**Del**	**Ins**	**Err**	**S.Err**
Score_1	77.4	16.9	5.8	8.5	31.2	100
Score_2	77.3	16.3	6.4	6.7	29.4	100
Score_3	77.1	15.8	7.0	5.3	28.2	100
Score_4	76.9	15.4	7.8	4.3	27.4	100
Score_5	76.5	15.1	8.5	3.6	27.1	100
Score_6	75.9	14.9	9.2	3.0	27.1	100
Score_7	75.3	14.8	9.9	2.6	27.3	100
Score_8	74.8	14.6	10.6	2.2	27.4	100
Score_9	74.1	14.5	11.4	1.9	27.8	100
Score_10	73.3	14.3	12.4	1.6	28.3	100

TABLE 4.4: Decoding results after DNN training with NAFTA database(54-53-53) as input

	Occurrences	Original		Confusion
1:	234	i:	==>	i
2:	224	a:	==>	a
3:	208	a	==>	a:
4:	165	m	==>	n
5:	163	o	==>	o:
6:	162	i	==>	i:
7:	133	n	==>	m
8:	131	e:	==>	e
9:	122	o	==>	a
10:	120	o:	==>	o
11:	107	e	==>	e:
12:	99	y	==>	i
13:	98	r	==>	a
14:	85	d	==>	t
15:	81	l	==>	r
16:	75	ou	==>	ou:
17:	73	g	==>	k
18:	72	u:	==>	u
19:	71	u	==>	o:
20:	70	rn	==>	n
21:	63	rl	==>	l
22:	61	b	==>	p
23:	60	a}	==>	a
24:	60	ou:	==>	ou
25:	60	u	==>	u:

TABLE 4.5: NAFTA-53-53-confusion pairs

NAFTA_50_39	Corr	Sub	Del	Ins	Err	S.Err
Score_1	82.1	12.3	5.6	8.5	26.4	100
Score_2	81.8	11.9	6.3	6.8	24.9	100
Score_3	81.5	11.5	7.0	5.4	23.9	100
Score_4	81.1	11.1	7.7	4.3	23.1	100
Score_5	80.7	10.9	8.5	3.5	22.9	100
Score_6	80.1	10.7	9.2	2.9	22.8	100
Score_7	79.6	10.5	9.9	2.5	22.9	100
Score_8	78.9	10.4	10.6	2.1	23.2	99.9
Score_9	78.3	10.4	11.4	1.8	23.6	99.9
Score_10	77.6	10.3	12.2	1.6	24.0	99.9

TABLE 4.6: Decoding results after DNN training with NAFTA database(50-50-39) as input

	Occurrences	Original		Confusion
1:	189	y	==>	i
2:	181	o	==>	a
3:	177	m	==>	n
4:	147	o	==>	u
5:	144	n	==>	m
6:	132	u	==>	o
7:	102	r	==>	a
8:	100	a	==>	o
9:	96	e	==>	i
10:	89	d	==>	t
11:	87	l	==>	r
12:	79	rn	==>	n
13:	74	i	==>	y
14:	71	i	==>	e
15:	71	rl	==>	l
16:	67	g	==>	k
17:	62	r	==>	l
18:	61	a	==>	ae
19:	61	b	==>	p
20:	57	ae	==>	a
21:	57	ae	==>	e
22:	56	oe	==>	e
23:	55	r	==>	v
24:	52	r	==>	o
25:	51	e	==>	ae

TABLE 4.7: NAFTA-50-39-confusion pairs

4.2 Acoustic-articulatory inversion using DNNs

To establish a baseline for future experiments, the experiments described in Benigno Uría's Master's thesis [5] were attempted to be reproduced. These experiments were to give a reference performance as well as verify the implementation of the network structure.

The implementation for these experiments was done in MATLAB. The framework used was the DeepLearnToolbox developed by Rasmus Berg Palm [28] and is described in chapter 3. The toolbox is developed for general deep learning, but is to a certain extent focused around classification of mnist handwritten digits. To be able to perform regression instead of classification, modifications were done to the testing function so it would simply compare predicted output with target output and return the RMSE value. For these experiments, continuous values were used as input which were not exclusively in the range (0,1). To cope with this, an implementation of a Gaussian-Bernoulli RBM was added. Instead of using binary units in the hidden layer, the output sample was chosen to be the probability of activation after being passed through the sigmoid function. The toolbox implements L2-regularization for the neural network, but not for the RBM. The RBM training was thus modified to allow for this regularization.

When unfolding a DBN to a DNN, a linear output layer is added on top. Generally, this layer is initialized with random values. However, through doing so, and applying backpropagation of error, the structure built through pre-training with a DBN can get distorted significantly. Instead, initialization is done by applying the pseudo-inverse method described in chapter 2.5.7. As a final modification, the hidden biases for DBN training were initialized to be -4 and the initial weights are drawn from the normal distribution given by $\mathcal{N}(0, 0.01)$.

4.2.1 Data setup

The dataset used for these experiments has been described in section 3.5. In addition to supplying the raw data, the database also consists of transformed data to make reproduction of results achieved with the database easier. From the 7 datapoints for each of the 6 coils, only two are used, meaning each EMA frame consists of 12 coordinates. Even though the EMA data is 3-dimensional, the choice

of only using x- and y-dimensions in the midsaggital plane can be explained by the movement perpendicular to the midsaggital plane being very small, which is intuitive. The acoustic data consists of frames of 40 frequency-warped LSFs[29] and a gain value. The frame shift is set to 5msec to match the sampling frequency of the EMA of 200Hz. The 6 coils used are shown in figure 2.5 and are mounted on the upper lip (UL), lower lip (LL), lower incisor (LI/JAW), tongue tip (TP), tongue blade (TB), and tongue dorsum (TD).

As input, a context window of 10 acoustic frames is used, selecting only each second frame making the context window span a total of 100msec. As output, or target values, we use the EMA frame that corresponds to the middle of the context window, i.e. between acoustic frame 5 and 6. The dataset is divided into three subsets following the suggestion provided in the database, namely a validation and a test set comprising 63 utterances each and a training set comprising the last 1137 utterances.

4.2.2 Pre-training of the DNN

Due to the nature of the backpropagation method for training DNNs, pre-training has proven to yield good results. In these experiments, the DNN was first trained as a DBN before getting unfolded into a DNN. The visible layer, or input layer, is by definition the size of the input vectors, i.e. 41 in this case. Furthermore, three hidden layers were sequentially trained and added on top to construct a 41-300-300-300 DBN. In its original state, a DBN is trained by stacking RBMs which have binary activations in both visible and hidden neurons. The input data is not binary and we thus chose to use a GBRBM for our first layer and regular RBMs for the following two layers. The RBMS were trained using the Contrastive Divergence criterion and only performing one step (CD-1) as this has shown good results. The parameters for the training configurations were chosen equal to the ones in [5] and are shown in table 4.8 for the GBRBM, and table 4.9 for the two following RBMs. Training proceeded by first training the GBRBM with the input data, then freezing its weights and putting an additional layer on top. The top two layers then formed a Bernoulli-Bernoulli RBM and was trained with the activation of the previously top layer after propagating the input data from the input layer. This was repeated for the last layer to finalize the training.

Parameter	Value
Learning rate	0.00001
Momentum	0.5 (first 10 epochs) 0.9 (rest of epochs)
Total epochs	200
Minibatch size	100
Initial weights	$\mathcal{N}(0, 0.01)$
Initial visible bias	0
Initial hidden bias	-4
Weight decay	0.0001

TABLE 4.8: Initial parameters for the Gaussian-Bernoulli RBM training of the first pre-training layer

Parameter	Value
Learning rate	0.0001
Momentum	0.5 (first 10 epochs) 0.9 (rest of epochs)
Total epochs	200
Minibatch size	100
Initial weights	$\mathcal{N}(0, 0.01)$
Initial visible bias	0
Initial hidden bias	-4
Weight decay	0.0001

TABLE 4.9: Initial parameters for the Bernoulli-Bernoulli RBM training of the two pre-training layers succeeding the GBRBM-layer

4.2.3 Fine-tuning the DNN

When pre-training was done, the DBN had to be unfolded into a DNN for fine-tuning. However, a linear regression layer had to be added on top of the network as outlined in section 2.5.7 before fine-tuning could commence. The output layer had to be the same size as the target vector we wish to infer, i.e. 12 in this case and the weights were attempted initialized by applying the pseudo-inverse method. The next and final step of the training phase was applying the back-propagation method with learning parameters as shown in table 4.10.

4.2.4 Investigating non-satisfactory results

Due to not having access to machines with GPU-processing options, the experiments were not run with the full dataset. To verify the implementation, the

Parameter	Value
Learning rate	0.0001
Momentum	0.5 (first 10 epochs) 0.9 (rest of epochs)
Total epochs	150
Minibatch size	100
Weight decay	0.0001

TABLE 4.10: Initial parameters for the fine-tuning phase of the DNN training

experiment was run using a smaller subset of the data. When running the above mentioned experiments, the results were far from satisfactory. For each epoch, the mean reconstruction error and the RMSE was monitored for the DBN pre-training and the DNN fine-tuning respectively. For each epoch, the reconstruction error of each of the RBMs was reduced as intended. The issues, however, started appearing when the fine-tuning of the DNN had commenced. The DNN would achieve reduced RMSE for the first three epoch, after which, the RMSE would converge and oscillate around this RMSE value. Seemingly, the network hit a local minimum which it was not able to get out of.

To locate the error, every step of the code was monitored in detail, only making one change at the time, to make sure that if a change occurred, the source would be known. Throughout the code for pre-training, no real error was encountered, but it appeared that the weights, even after many epochs of training, were still negligible in size as compared to the hidden bias, which was initialized to -4.

A major issue was discovered when monitoring the unfolding of the DBN into a DNN. As described earlier, the initialization of the linear regression layer was done by using the pseudo-inverse method. It turned out that the weights were initialized with values in the order of 1e8-1e10 which, obviously, are way too large. The learning rate used for the fine-tuning is 0.0001, meaning that even with an extremely large dataset, the values would never reach reasonable values. Furthermore, when passing a training vector through the network and multiplying the hidden layer activations with the output weight matrix, the output values would become extremely large. This leads to an enormous error and thus also an enormous gradient. Even though the error is reduced when propagating it down to the hidden layers, the gradient would still be too large and the fine structure created by the DBN would be destroyed. All the weights would become so large that when passing a training vector through, the input to the sigmoid functions would be so

large, either in positive or negative sense, that the output would become either 0 or 1. An interesting discovery was that, even though the RMSE was extremely large in this case, the fine-tuning of the network would hit a local minimum after the same number of epochs and oscillate back and forth with a step size in the range of (-1,1), a mere fraction of the actual error.

Assuming the initialization of the output layer being the problem, the pseudo-inverse method was replaced with a randomized initialization. The dataset used for debugging consisted of a training set with 52,000 training vectors and a test set of 16,600 test vectors. The final results obtained after this modification was in place seemed reasonable at first glance with a mean RMSE for each output unit of 0.206 centimeters, but for some reason, the training still converged after three training epochs. By further investigation of the activations for each layer, it turned out that each value was very close to 0.018.

By reverse engineering, it became obvious that the initial hidden biases were too large and dominating the sum after forward-propagating activations to the next layer. To recap, the activations of a layer is given by $sigm(xW - b)$, where x is the activations of the previous layer, W is the weight matrix and b is the hidden biases. The weights are initialized by drawing from a normal distribution with zero mean and 0.01 variance. As an example, we will show the activation of a single neuron in the hidden layer. The activation will be given by $\sum_{i=1}^{41} x_i * w_i - 4$, where x has zero mean and unit variance and w is initialized as mentioned above. This sum will, in a highly unlikely case, equate as much as $41 * 1 * 0.01 = 0.41$. In the average case, however, the sum will be very close to zero. In fact, the expected value is exactly zero due to the zero mean. This leaves the input to the sigmoid to equal ≈-4 for all units, giving an activation of roughly 0.018.

The main issue with the input to the sigmoid being such large negative values is that the gradient of the sigmoid for very large or very small input values is really small. Thus, gradient training will only cause very small updates of the weights and if the system ends up in a local minimum, it will not be able to get out.

4.2.5 Further experimentation to improve performance

Due to the issues with reproducing the results show in [5], we had to take a step back, start with simpler structures and, step by step, modify one parameter at

a time to build a system which could perform at a comparable level. The size of the structure was reduced to only comprising one hidden layer to speed up the experiments so that time would allow testing of many different configurations of parameters. In table 4.11, the results from experiments with varying number of epochs and initial hidden bias are shown. All results shown in this table are acquired through training with the debugging set mentioned above.

The first modification done was initializing the hidden biases, which can be seen in the first column, to zero. By doing so, the RMSE was reduced and the system was able to escape, or avoid completely, the local minimum encountered as seen for the case with -4 initialization. Due to this discovery, several experiments were run with an increasing number of epochs both for DBN and DNN training. It appeared that by increasing the number of epochs above 200 for the case of DBN training, and 150 for DNN training, the RMSE would not drop, but in fact increase slightly. After establishing a temporary "optimal" number of epochs, several different initializations of the hidden biases were attempted. It turned out that training with initial values of -2 provided a slightly better performance and thus an increased number of DNN training epochs was tested which produced the so far best RMSE of 0.1595, as seen in column four.

It was hypothesized that the dataset size could influence which initialization was optimal for the given experiment. To test this, experiments with larger datasets were run for different hidden bias initializations. The results are shown in table 4.12. All results shown in this table are acquired through training with a subset of the full training and test set respectively. For the training and test vectors, the notation 52k-104k means all vectors from the 52000nd to the 104000nd vector are used.

A peculiar observation is that the performance using the small debugging set is better than using a training and test set of approximately four times the size. A possible explanation for this is that the debugging train and test sets are both taken from the training subset of the full dataset. The training set consists of utterances 1-9 and the test set consists of utterances 12-14. This means that they are relatively close in recording time and thus might also contain fairly similar content. If the test set consists of features that has not yet been learned by the net, it will lead to large errors and thus choosing utterances close in recording time is more likely to show better results due to similar content and recording conditions. To validate this theory, two experiments were run with the same parameters, only

Bias init.	**epochs DBN**	**epochs DNN**	**RMSE**
-4	30	15	0.2061
0	30	15	0.1875
0	50	30	0.1716
0	200	150	0.1639
0	300	200	0.1645
$\mathcal{N}(0, 0.1)$	200	150	0.1640
-0.5	200	150	0.1641
-2	200	150	0.1634
-2	200	500	0.1595
-4	200	150	0.1778
-4	300	200	0.1743

TABLE 4.11: Experiments run with the debugging training set (52k training vectors and 16600 test vectors) and a 41-300-12 net structure

Bias init.	**Train vectors**	**Test vectors**	**RMSE**
-4	1-200k	1-40k	0.1782
-2	1-200k	1-40k	0.1770
0	1-200k	1-40k	0.1788
0	52k-104k	16.6k-33.2k	0.1895
0	1-52k	63k-79.k (from train)	0.1703
-2	1-3.5kk	1-300k	0.1602
-2	1-2kk	1-100k	0.1582

TABLE 4.12: Experiments run with a varying subset of the full dataset. Train vectors are taken from the full train set and test vectors from the full test set unless specified otherwise. The number of DBN epochs was 200 and the number of DNN epochs was 150. The net structure used was 41-300-12.

changing which subsets to use for training and testing, one with testing set taken from the full test set, and one with testing set taken from the full train set. The system performs significantly better in the latter case and confirms our suspicion. Due to this fact, the debugging training set was used for further experiments, not to force better results, but due to the assumption that for experiments with the full dataset, the system will have learned close to all features and provide similar performance when tested on the full test set.

With these observations concerning the subsets used for training, an experiment with large amounts of data was performed to see if this would allow a significant improvement. An improvement was seen, however not as large as first anticipated. Again, the test set consisted of utterances recorded at a later time than the training

data, but due to the large training set it was assumed that most of the features were learned. To check if this assumption held, another experiment was run with a smaller training set and a highly reduced test set, removing most of the utterances recorded at a later time than the training utterances. By doing so, the performance was increased, which backs up our theory.

A very interesting discovery is that with only one hidden layer and with a high enough number of epochs in the DBN training, the DNN fine-tuning RMSE does not get stuck and actually improves for each training epoch even when initializing the hidden biases to -4. Even though the system still performs worse with an initial bias of -4, it may seem that, either the reduced number of layers or the increased amount of training data with respect to system parameters may have solved the convergence problem. To determine the root cause of the improvement, a large scale experiment with three layers with initial hidden biases of -4 should be tested with a large subset of, or ideally the whole, training set.

Even though improvements are seen by tuning the different parameters of the training, the performance is still far off what has been reported on similar networks. In the beginning of this section, we explained why the pseudo-inverse method was used for initialization. Initializing the weights of the output layer with random small values and performing back-propagation will distort the fine and fragile structure built by the DBN pre-training. So far, all the experiments were run without any sort of initialization on the output layer other than random, and this fact may explain the mediocre results. The issues with the pseudo-inverse method may have several causes, but our experiments unveil one in particular. Every ten rows of the matrices used to create the square matrix to be inverted are equal. This, in addition to values being relatively small, leads to a matrix which is close to singular and thus a poor inverse. Instead of the pseudo-inverse method, we propose an approach which won't suffer from this issue. After unfolding the DBN into a DNN, the output layer is initialized with random values. Furthermore, the weights from all layers but the output layer are frozen and when performing back-propagation updates, only the output layer is updated for the first epochs. Since the output layer is linear, the number of epochs does not need to be large, in fact, only one epoch might be enough. When the output layer is initialized properly, the rest of the training is executed in the regular manner with back-propagation for all layers.

Net structure	**Number of pre-training epochs**	**total number of back-prop epochs**	**RMSE**
41-100-12	0	150	0.1641
41-100-12	0	200	0.1634
41-100-12	0	500	0.1629
41-100-12	2	150	0.1749
41-100-12	10	150	0.1761
41-100-12	10	160	0.1754
41-100-12	100	200	0.1797

TABLE 4.13: RMSE results when first freezing the hidden layer weights and performing back-propagation as pre-training for the output layer weights. The number of DBN training epochs is 200 in each case. The dataset used is the debugging set and hidden layer initial bias is $\mathcal{N}(0, 0.01)$.

To speed up simulations, hidden layers of 100 neurons instead of 300 was investigated and the results are shown in table 4.13. As seen in the first column, all the experiments here were performed with one hidden layer. Furthermore, the number of training epochs where only the output layer was trained is seen in column two, the total number of backpropagation training epochs is shown in column three, and the RMSE for each experiment is shown in column four. For a network with one hidden layer and the initial setup of 200 DBN training epochs and 150 back-propagation epochs, the RMSE is approximately the same as for a network with 300 hidden neurons. In addition, the proposed pre-processing step of the output regression weights was implemented and tested for a different number of pre-training epochs. Unfortunately, the results showed decreasing overall performance, the more pre-training epochs were performed. It may seem that the large gradient from the first training epochs in fact did not mess up the fine structure created by DBN training, but rather updated them more appropriately. This is a peculiar find, but it can be explained by the size of the learning rate used in these experiments. If the learning rate is too small, the backpropagation will only do very small modifications to the hidden layer. However, experiments on a single hidden layer structure are not enough to draw a conclusion for deep architectures and so similar experiments had to be performed for deeper networks.

The experiments for one, two, and three hidden layers were all first run with 200 DBN training epochs, 150 backpropagation epochs and $\mathcal{N}(0, 0.01)$ initialization of the hidden bias to have a baseline to compare the structures with. Already with two hidden layers it became obvious that changes to the dataset had to be done

Net structure	Number of pre-training epochs	total number of back-prop epochs	RMSE
41-100-100-12	0	150	0.1760
41-100-100-12	0	300	0.1700
41-100-100-12	0	500	0.1648
41-100-100-12	0	800	0.1614
41-100-100-12	2	150	0.1762
41-100-100-12	2	500	0.1651
41-100-100-12	10	150	0.2059
41-100-100-12	100	200	0.2055

TABLE 4.14: RMSE results when first freezing the hidden layer weights and performing back-propagation as pre-training for the output layer weights. The number of DBN training epochs is 200 in each case. The dataset used is the debugging set and hidden layer initial bias is $\mathcal{N}(0, 0.01)$.

as compared to the single hidden layer case. The results for the two-hidden layer case are shown in table 4.14. The first column shows the network structure, the second shows the number of pre-training epochs for the output layer in increasing order, the third shows the total number of backpropagation epochs, and the fourth shows the RMSE of each experiment. The performance was significantly worse and even when using 500 backpropagation epochs, the performance was still a little off from what was obtained with a single hidden layer. Due to this result, a larger number of epochs was used to test the performance when including the step of pre-processing the output layer weights.

An interesting observation is that in this case, performance using two epochs of output layer pre-training was approximately as good as without any pre-training. This is a significant improvement from the one-hidden layer experiments. Both for the case with 10 and 100 pre-training epochs, the network got stuck in the local minimum encountered in the initial experiment, i.e. RMSE ≈ 0.206. The improved performance of the pre-training with an increased number of layers encouraged experimenting with yet another hidden layer, however for these tests the debugging set was expected to be inadequate. The number of epochs required to achieve comparable results to the one-hidden layer case, suggests that a larger dataset has to be used for further experiments.

To confirm our suspicion, a three-hidden layer network was trained with the debugging set with varying initialization of the hidden biases. The results, as displayed in table 4.15, show poor performance as expected. Both the experiments with -4

Hidden bias initialization	**Number of DBN training epochs**	**Number of DNN back-prop epochs**	**RMSE**
$\mathcal{N}(0, 0.01)$	200	150	0.2046
-2	200	150	0.1972
-2	200	300	0.1932
-4	200	150	0.2063
-4	300	150	0.2060

TABLE 4.15: RMSE results when initializing the output regression layer with random weights and using a network structure of 41-100-100-100-12. The dataset used is the debugging set.

initialization and random initialization got stuck in the same local minimum as encountered before, i.e. with an RMSE of roughly 0.206. A surprising observation however, was that initializing the hidden biases to -2 allowed the network to escape the local minimum and thus it might seem that this initialization is superior to the others. Due to this observation, further experiments with three layers and a larger dataset should be performed. However, because such experiments have a long run time, these were not prioritized and further experimentation on one-hidden layer networks was focused.

4.2.6 Low-pass filtering the predictions

So far, the focus has only been revolving around RMSE values, but this is just one of many performance measures and can, in some cases, be deceiving. To get a better picture of the system performance, plotting the prediction and the target values can often reveal information that RMSE values can not. An excerpt from testing with the debugging testing set can be seen in figure 4.2.

It is apparent that the prediction follows the general curvature of the trajectory, but it has a high frequency component with a large amplitude. This leads to the hypothesis that by removing the high frequency components, the RMSE can be reduced. A simple finite impulse response (FIR) filter was used to create a low-pass(LP) filter. The order of the LP filter was chosen by comparing three different cases for different cut-off values. When the order was chosen, a smaller frequency interval around the best performing cut-off frequency was tested with a finer quantization. The results can be seen in table 4.16. The first three rows show the RMSE for seven different cut-off frequencies when using an FIR filter

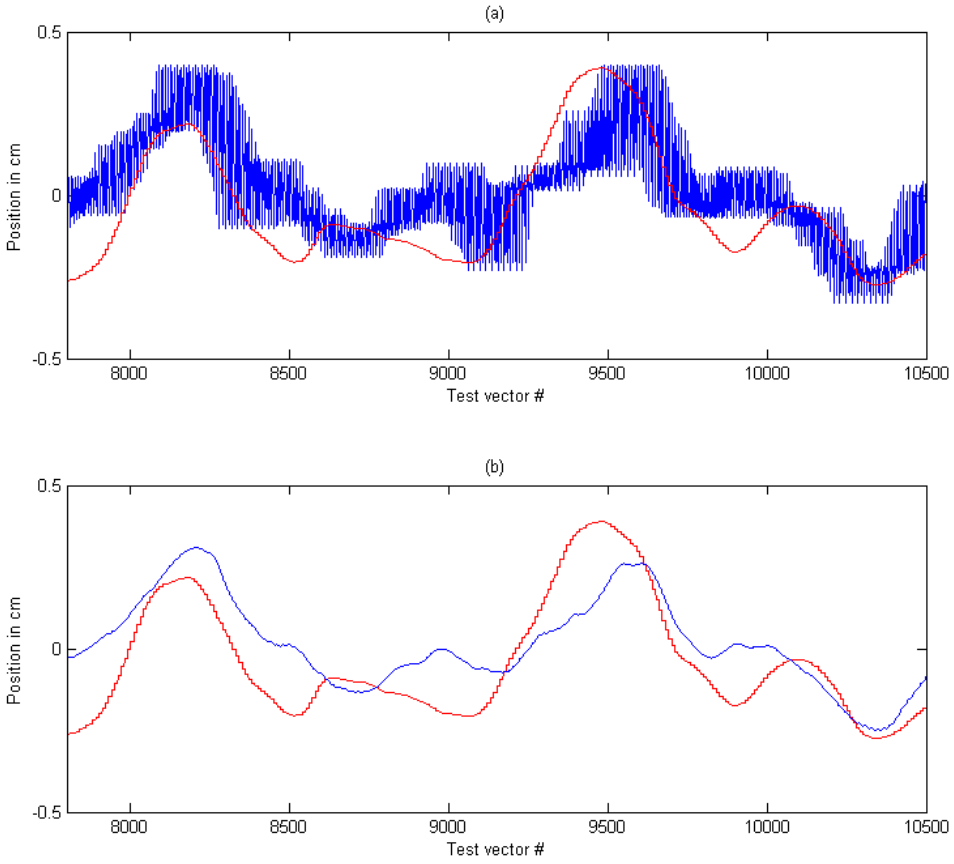

FIGURE 4.2: Excerpt of articulator trajectory for a single articulator using the debugging test set. (a): Predicted articulator trajectory (blue) and target trajectory (red). (b): LP-filtered predicted articulator trajectory (blue), using an order 30 fir-filter with cut-off frequency 0.1 times Fs/2, and target trajectory (red)

of order 10, 20, and 30 respectively. Both the filter with order 20 and 30 achieve the shared optimal performance. However, the filter of order 30 performs slightly better for almost any cut-off frequency, and this order was thus chosen for further experimentation. To decide for a cut-off frequency, the order was set to 30 and new experiments were run with a smaller step size in cut-off frequency and centered around the frequency that had performed the best so far. The results can be seen in the bottom two rows and it turned out that both increasing, and decreasing, the cut-off did not improve the performance and it was thus concluded that 0.1 was the best choice for further experimentation. It is important to note that the actual cut-off frequency is obtained by multiplying with Fs/2.

By low-pass filtering the predictions, we were able to improve the performance

Cut-off:	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**
Order 10:	0.1501	0.1517	0.1544	0.1574	0.1595	0.1608	0.1617
Order 20:	0.1484	0.1510	0.1577	0.1602	0.1615	0.1623	0.1629
Order 30:	0.1484	0.1505	0.1584	0.1593	0.1612	0.1613	0.1621
Cut-off:	**0.03**	**0.04**	**0.05**	**0.06**	**0.07**	**0.08**	**0.09**
Order 30:	0.1484	0.1484	0.1484	0.1484	0.1484	0.1484	0.1484
Cut-off:	**0.11**	**0.12**	**0.13**	**0.14**	**0.15**	**0.16**	**0.17**
Order 30:	0.1484	0.1484	0.1484	0.1485	0.1485	0.1487	0.1489

TABLE 4.16: RMSE after low-pass filtering the predictions with a simple low-pass finite impulse response filter. A 41-300-12 structure is used, hidden bias initialization is $\mathcal{N}(0, 0.01)$, 200 epochs for DBN and 150 epoch for DNN. The training set used is the debugging set.

of the system with approximately 10%. In addition to showing a smaller RMSE, another benefit from the filtering is the reduced difference between neighboring samples. In the excerpt displayed, this difference could be as large as 0.4 cm, which means that even though an RMSE value of 0.1640 was decent, the actual error at any point could be unacceptably large. Even though the filtering made some samples deviate more from the actual value, a smooth curve is more applicable due to its predictable nature and consistency.

4.2.7 Investigating a solution for the instability

Low-pass filtering the predictions improved the performance greatly, but did not deal with the root cause of the problem. By locating the issue and solving it, the performance is likely to improve even more. From figure 4.2(a), it is obvious that small variations in the input signal cause a big change in the output prediction. This is a non-linear behavior and the issue is thus likely to lie somewhere between the input layer and the hidden layer. The sigmoid function is a non-linearity and its largest gradient is around the input value 0. In this area, a small change in input provides a large change in output. This means that if the weights between the input and the hidden layer are very small, the sum being fed into the sigmoid will be very small and cause the instability we see. To investigate this, a histogram was made from the weights after DBN pre-training to visualize the values they take. This histogram is shown in figure 4.3.

It is obvious from figure 4.3 that the weights between the input layer and the hidden layer are very small. As the input is z-normalized, the input values are

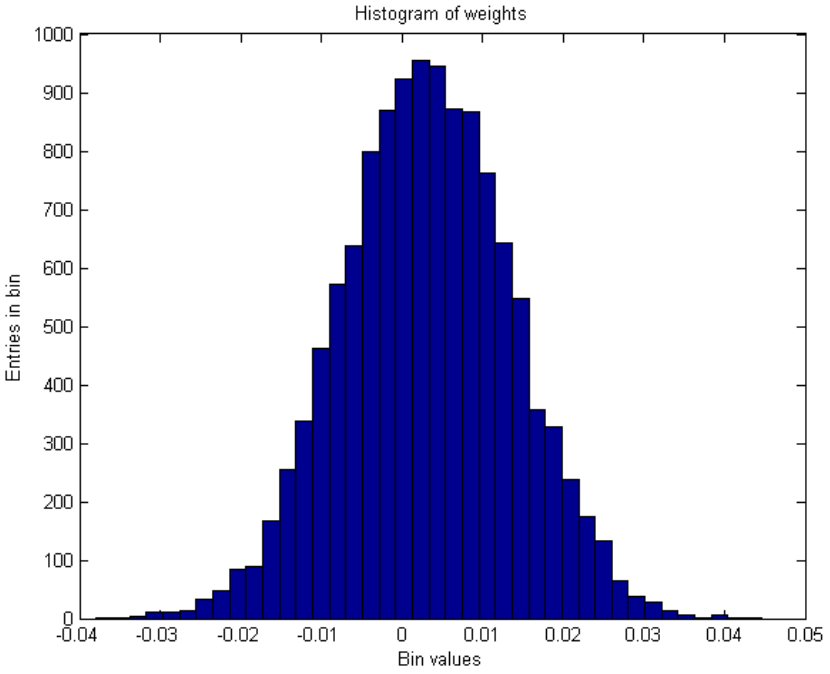

FIGURE 4.3: Histogram of the weights between the input layer and hidden layer after DBN pre-training a 41-300-12 network with an LR of 0.00001.

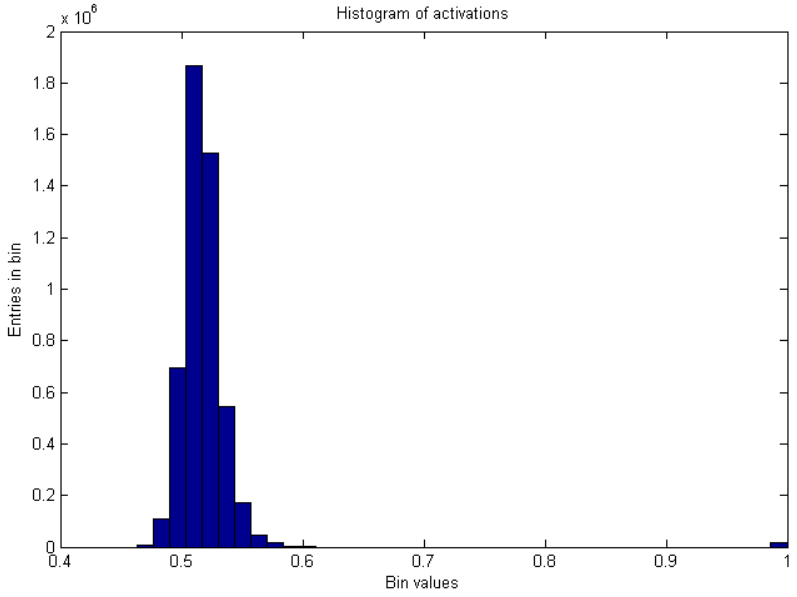

FIGURE 4.4: Histogram of the activations of the hidden layer after DBN pre-training a 41-300-12 network with an LR of 0.00001.

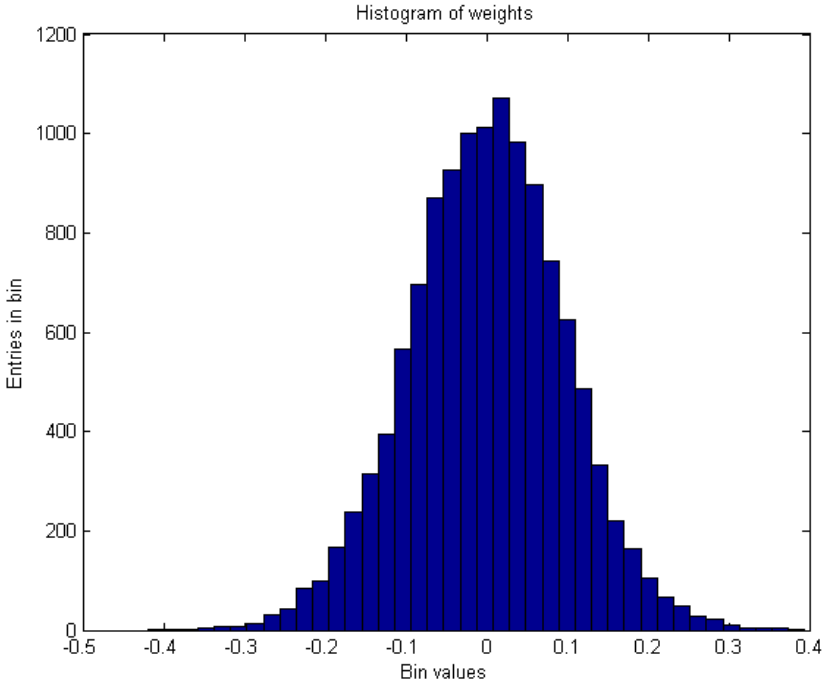

FIGURE 4.5: Histogram of the weights of the hidden layer after DBN pre-training a 41-300-12 network with a learning rate of 0.01.

roughly between (-1,1) and thus the product of the multiplication with the input and weights will be even smaller than what is shown in the histogram. In addition, the values are both positive and negative, meaning that several of the terms will most likely cancel out. The activations in the hidden layer after the sigmoid non-linearity can be seen in figure 4.4. The activations are focused around the value 0.5, which is the output from a sigmoid with input 0. It has a slight offset to the right of 0.5 which can also be seen in the weight histogram. The one large spike around 1 is the appended bias term. This discovery shows that the DBN pre-training does not work exactly as intended. The weights are not moved far enough in the parameter space away from their original position, i.e. not trained properly. So far, experiments have been done with a fairly low learning rate. This might cause the training to converge to a local minimum due to the small step size, or just not be trained long enough to be able to reach a sufficiently good minimum. Further experiments were thus conducted with varying learning rates to investigate if this could improve the pre-training performance.

The learning rate chosen for these experiments was the one outlined in Uría's thesis. In his work, a dataset approximately 140 times larger was used and thus,

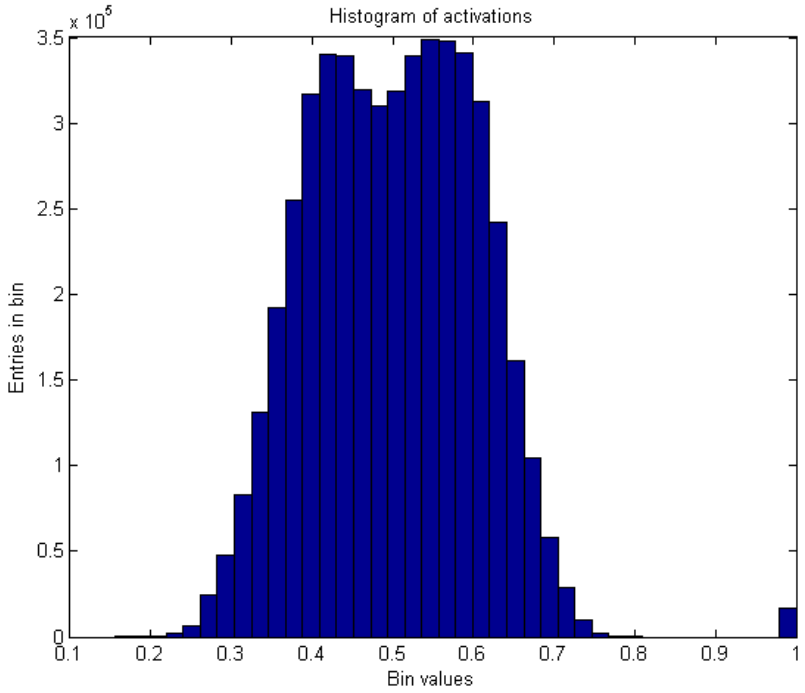

FIGURE 4.6: Histogram of the activations of the hidden layer after DBN pre-training a 41-300-12 network with a learning rate of 0.01.

a smaller learning rate can be used successfully. The learning rate was increased by a factor of 100, 1000 and 5000, and the RMSE was calculated with respect to the debugging test set, however without any back-propagation training. The results can be seen in table 4.17. Surprisingly, when increasing the learning rate by a factor of 100, the RMSE was increased. However, by increasing the learning rate even further, an RMSE gain was clearly visible. The GBRBM is less stable than the regular RBM and thus at some point, the learning rate would blow the training out of proportion. In fact, 0.05, which is shown in the table, was the largest learning rate tested that would not cause the training to go out of bounds.

Learning rate(LR)	0.00001	0.001	0.01	0.05
RMSE	0.3005	0.3090	0.2899	0.2113

TABLE 4.17: RMSE after unfolding a pre-trained DBN without applying back-propagation. The net structure was 41-300-12 and DBN pre-training was done for 200 epochs on the debugging set.

In figure 4.5, a histogram of the weights of the GBRBM after training with a learning rate of 0.01 are shown. The shape of the histogram is similar to that of the initial weight histogram in figure 4.3. However, the width is approximately 10

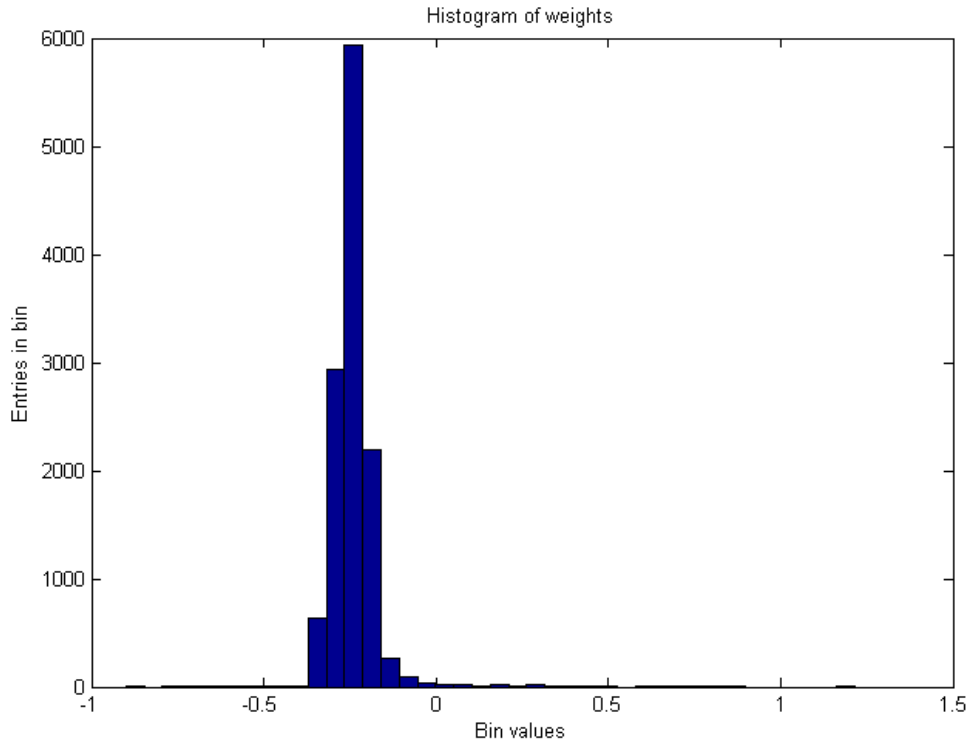

FIGURE 4.7: Histogram of the weights of the hidden layer after DBN pre-training a 41-300-12 network with a learning rate of 0.05.

times larger. This means that the weights have been moved significantly further in parameter space and the size of the weights is now large enough to avoid getting stuck around 0 after multiplication and summation with the input. In figure 4.6, a histogram of the activations in the hidden layer are shown and it is clear that the activations now clearly deviate from 0.5, which means that the dynamic area of the sigmoid is better utilized. This in turn should mean less instability on the output since the gradient of the sigmoid decreases the further away from zero-input it gets. This observations clearly shows that training is performed to a larger degree than what was done with a lower learning rate.

In table 4.17 we saw a large dip in RMSE when the network was trained with a learning rate of 0.05. The performance in this case, even before backpropagation and with random initialization on the output, was almost as good as the first results obtained in this dissertation, i.e when the backpropagation got stuck in a local minimum. It is, however, not the goal of DBN pre-training to achieve a low RMSE for regression, but rather to learn the general features of the input space. The exact RMSE after DBN training is thus not necessarily a sufficient measure

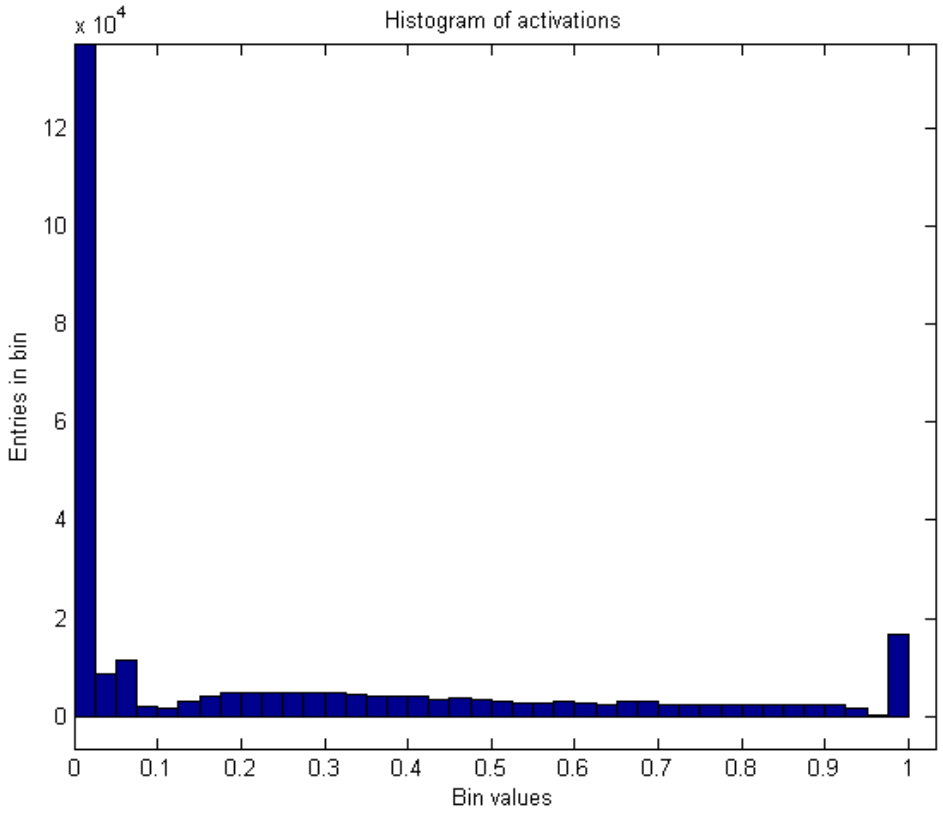

FIGURE 4.8: Zoomed histogram of the activations of the hidden layer after DBN pre-training a 41-300-12 network with a learning rate of 0.05.

for its performance. The weight histogram using this learning rate can be seen in figure 4.7. Surprisingly, the vast majority of the weights are now on the negative side and, as opposed to the case of an LR of 0.01, the weights now seem to be focused in a smaller dynamic area again. A histogram of the activations can be seen in figure 4.8. It is important to note that this histogram has been zoomed in due to the size of the 0-th bin. The activations are now spread over the entire dynamic area, i.e (0,1), but most of the entries are located around 0 due to the grouping of weights around -0.2. Because the RMSE value is so low, a learning rate of 0.05 seems to be a good choice, however the clustering of weights around a single value is somewhat alarming.

So far, the experiments with varying learning rate were evaluated before backpropagation had been performed. To see what impact changing the learning rate for the DBN made, the results after backpropagation also had to be evaluated. Surprisingly, the RMSE turned out to be exactly the same regardless of DBN-learning rate. Even though this was a discouraging discovery, the increased learning rate

was not discarded due to its provable advantage over a lower learning rate before backpropagation. Furthermore, when increasing the learning rate of the DBN provided improved results, one might ask, why couldn't this be the case for the backpropagation training as well. To check this, a 41-300-12 network was pre-trained as a DBN with a learning rate of 0.01, then unfolded to an NN and fine-tuned with backpropagation with different learning rates. The reason for choosing 0.01 learning rate for the DBN is due to the nicer spread of weights and activations. The results from these experiments can be seen in table 4.18. The learning rate for the backpropagation training is displayed in the third column and the number of epochs of pre-training for the output layer is shown in column four. By monitoring the RMSE which is shown in the first column, it is obvious that increasing the learning rate for backpropagation did not make any difference, except for the case where the learning rate got too large. At this point, the network got stuck in the local minimum encountered several times before. It is important to note that the learning rate when performing backpropagation was set to be the same value for both the hidden layer and the output layer.

RMSE	**DBN-LR**	**NN-LR**	**out-layer pre-training epochs**
0.1642	0.01	0.0001	0
0.1642	0.01	0.0001	2
0.1642	0.01	0.01	0
0.1642	0.01	0.01	2
0.2062	0.01	0.1	0

TABLE 4.18: RMSE after unfolding a pre-trained DBN with a larger learning rate and fine-tuning the NN with varying learning rate and pre-training of the output layer. The net structure was 41-300-12, 200 DBN training epochs, 150 NN training epochs and done with the debugging set.

When training a neural network, in particular a deep neural network, with backpropagation, the change for each iteration is largest for the output layer and gets smaller and smaller the further back the error is propagated. The magnitude of this reduction can be several orders and thus increasing the learning rate by 100 times for both the output layer and the hidden layer might not be sufficient. This increase is likely to be sufficient for the output layer given the results shown in table 4.18, but they are not likely to be sufficient for the hidden layer. To check this, a final experiment was carried out, gradually increasing the learning rate of the hidden layer while keeping the output layer learning rate unchanged. Due to time constraints, this effect was not thoroughly tested, but the interesting results

encourage further investigation and this is thus suggested for future research. The results obtained are shown in table 4.19.

RMSE	Hidden layer LR	Output layer LR
0.1642	0.0001	0.0001
0.1592	0.1	0.0001
0.1556	0.5	0.0001
0.1560	0.6	0.0001
0.1560	1.0	0.0001
0.1536	0.5	0.01

TABLE 4.19: RMSE for different hidden layer learning rates during backpropagation fine-tuning. The net structure was 41-300-12, the DBN LR was 0.01, 200 DBN training epochs, 150 NN training epochs and done with the debugging set.

It is obvious from table 4.19 that improved performance can be achieved by increasing the learning rate for lower layers during backpropagation. This is intuitive because the error after backpropagation will be reduced for each layer and thus also the gradient. Due to this fact, the improvement is likely to be even larger when applying this to a deeper structure with more layers. A decrease in RMSE was seen both for the tests with a learning rate of 0.1 and 0.5. However, when increasing the learning rate to 1.0, the RMSE had increased by a small margin. A learning rate of 0.6 was thus tested to see if a learning rate between 0.5 and 1.0 would preform even better. As seen in the table, this was not the case and a learning rate of 0.5 was concluded to be the best for this exact setup of the experiment.

Out of curiosity, the best learning rate for the hidden layer was chosen and the learning rate of the output layer was increased. It turned out that this combination improved performance even further and achieved the best results so far throughout all the experiments described in this dissertation. Again, by increasing the output layer learning rate further, the system became unstable and performed severely worse.

4.2.8 Testing the final model

Combining everything that was learned so far, a final model was put together to measure how well this improved network could perform. The network size used

was 41-300-12, the number of DBN pre-training epochs was 300 and the number of backpropagation fine-tuning epochs was 300. The initial hidden bias values of the DBN were drawn from the $\mathcal{N}(0, 0.01)$ distribution and the DBN learning rate was set to 0.01. Initialization with -2 was tested as well, but this failed to show improved performance for this setup. After unfolding the DBN to an NN, the output layer learning rate was set to 0.01 and the hidden layer learning rate to 0.5. After training was complete, predictions were made and low-pass filtered with an order 30 FIR filter with 0.1·Fs/2 cut-off frequency, and the RMSE was calculated with respect to the corresponding target values. The average RMSE for each of the articulators in both x- and y-direction are shown in table 4.20 as well as the total average RMSE. The final prediction and target trajectories for articulator TD_x can be seen in figure 4.9 as well as their absolute difference at every sample point.

Articulator	**RMSE**
TD_x	0.1503
TD_y	0.1931
TB_x	0.1647
TB_y	0.1713
TT_x	0.1867
TT_y	0.2101
JAW_x	0.0785
JAW_y	0.1004
UL_x	0.0570
UL_y	0.0596
LL_x	0.0915
LL_y	0.1547
Average	0.1348

TABLE 4.20: RMSE for each articulator obtained from the final model.

The performance of the network proves to be significantly better than what was initially achieved in the beginning of these experiments. In fact, the RMSE precision for some of the articulators is down to half a millimeter. Articulator TD_x was chosen for display due to it being closest in RMSE to the mean RMSE of all the articulators. It is obvious from figure 4.9 that the prediction follows roughly the same curvature and in periods provides a very accurate match to the real target values. It is interesting to see however, how the network performs a lot better for large peaks above zero than below. In fact, in some cases, the network predicts a positive spike when a negative one occurs. This might be due to the sigmoid's output dynamic range, i.e. (0,1). An input value close to 0 will contribute less to the total input sum of the sigmoid and thus also less to the gradient. This suggests that a better choice of non-linear operator would be the hyperbolic tangent which

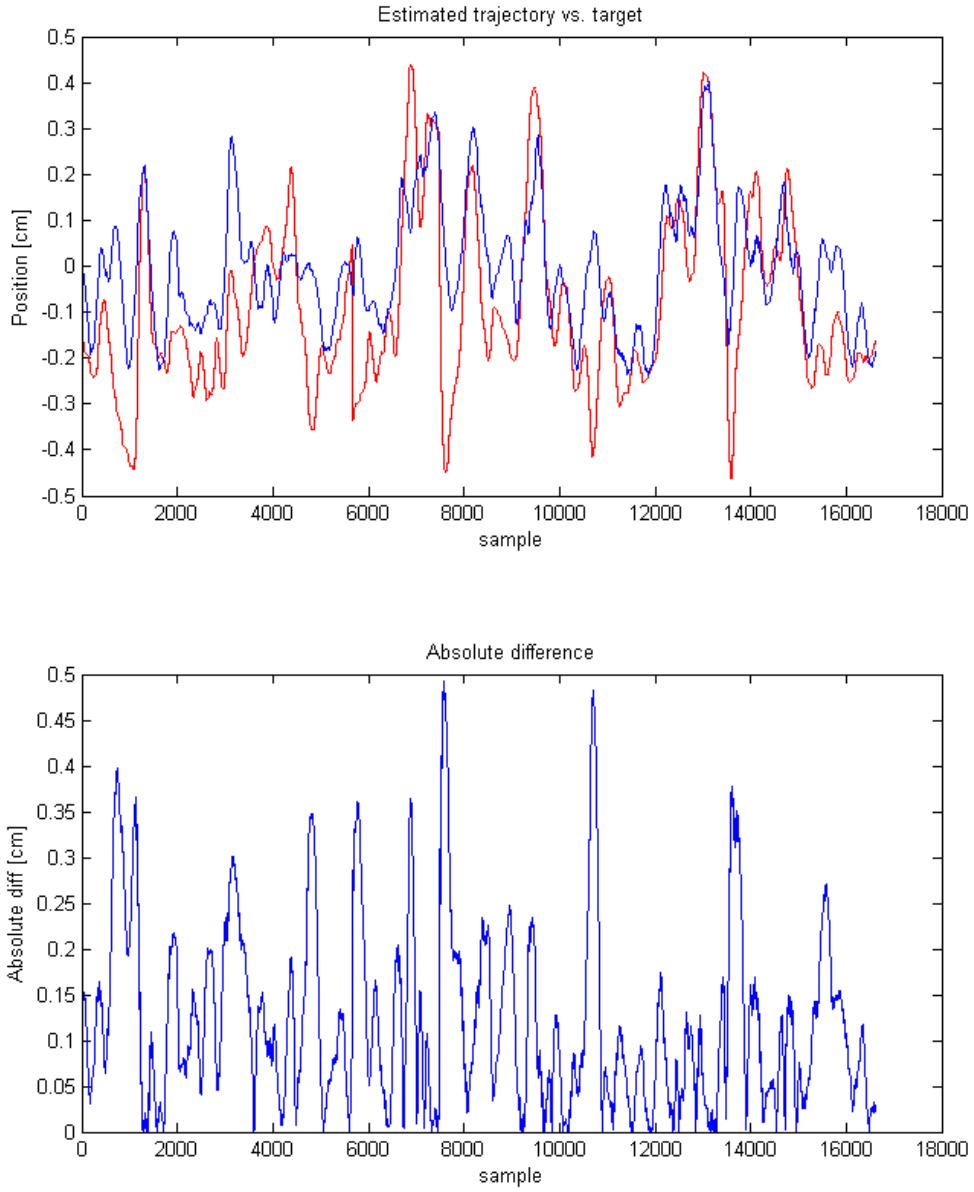

FIGURE 4.9: (Top): Plot of the predicted trajectory (blue) and the target (red) for articulator TD_x. (Bottom): Plot of the absolute difference between prediction and target.

has a dynamic range of (-1,1). This will make the values that previously were close to 0 be close to -1 and thus contribute equally, though in the opposite direction, of values close to 1.

Figure 4.10 and 4.11, show the trajectories for each of the 6 articulators plotted for two different utterances. Each figure contains all 6 articulators for both utterances in the x- and y-direction respectively. These trajectories have been z-score normalized with respect to the entire dataset, i.e. mean subtracted and divided by 4 times the standard deviation (STD). The STDs for each articulator in each dimension is shown in table 4.21 and 4.22. All the values shown, both the STDs and the plots are in a centimeter scale. Comparison of any two utterances in the dataset produce similar results and even though only a comparison of two utterances is shown here, the same conclusions can be drawn for an arbitrary pair of utterances.

It is clear from the STDs that the movement in the mid-saggital plane is in the order of several millimeters in each direction from the mean value. In fact, by inspecting the trajectories, the movement can even surpass a centimeter, which is intuitive and can be verified by "talking to a mirror". Even though there are areas of overlap in the trajectories, the difference is for the majority of the time several millimeters. This indicates that with a precision of even 2-3 millimeters, one can discriminate between a wide range of phones.

Articulators	TD-x	TB-x	TT-x	JAW-x	UL-x	LL-x
STDs (σ)	0.2345	0.2633	0.2964	0.1096	0.0617	0.1372

TABLE 4.21: Standard deviations over the entire dataset for each articulator in the x-direction

Articulators	TD-y	TB-y	TT-y	JAW-y	UL-y	LL-y
STDs (σ)	0.3647	0.3606	0.3518	0.1778	0.0837	0.2525

TABLE 4.22: Standard deviations over the entire dataset for each articulator in the y-direction

The original setup of these experiments was based on the ones outlined in [5] and we will thus use the results displayed in this thesis as a baseline for comparison. With a network of 41-300-12, Uría achieved an RMSE of 1.0787, however the RMSE of each individual articulator for this experiment setup is not provided. Thus, for comparison we will have to compare with the results for the case of a

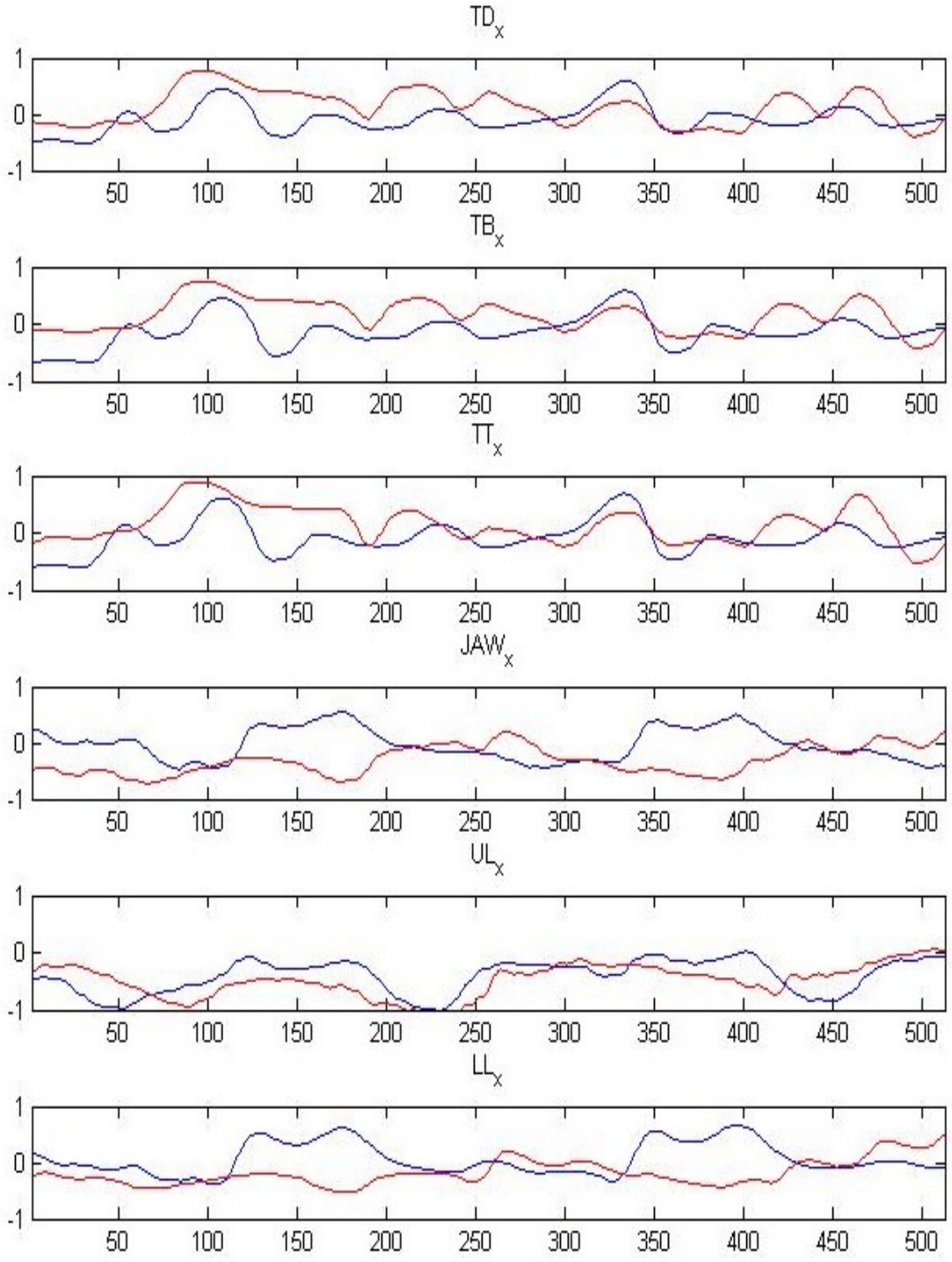

FIGURE 4.10: Articulator trajectories in the x-direction, blue from utterance mngu0-s1-0001 and red from utterance mngu0-s1-0002

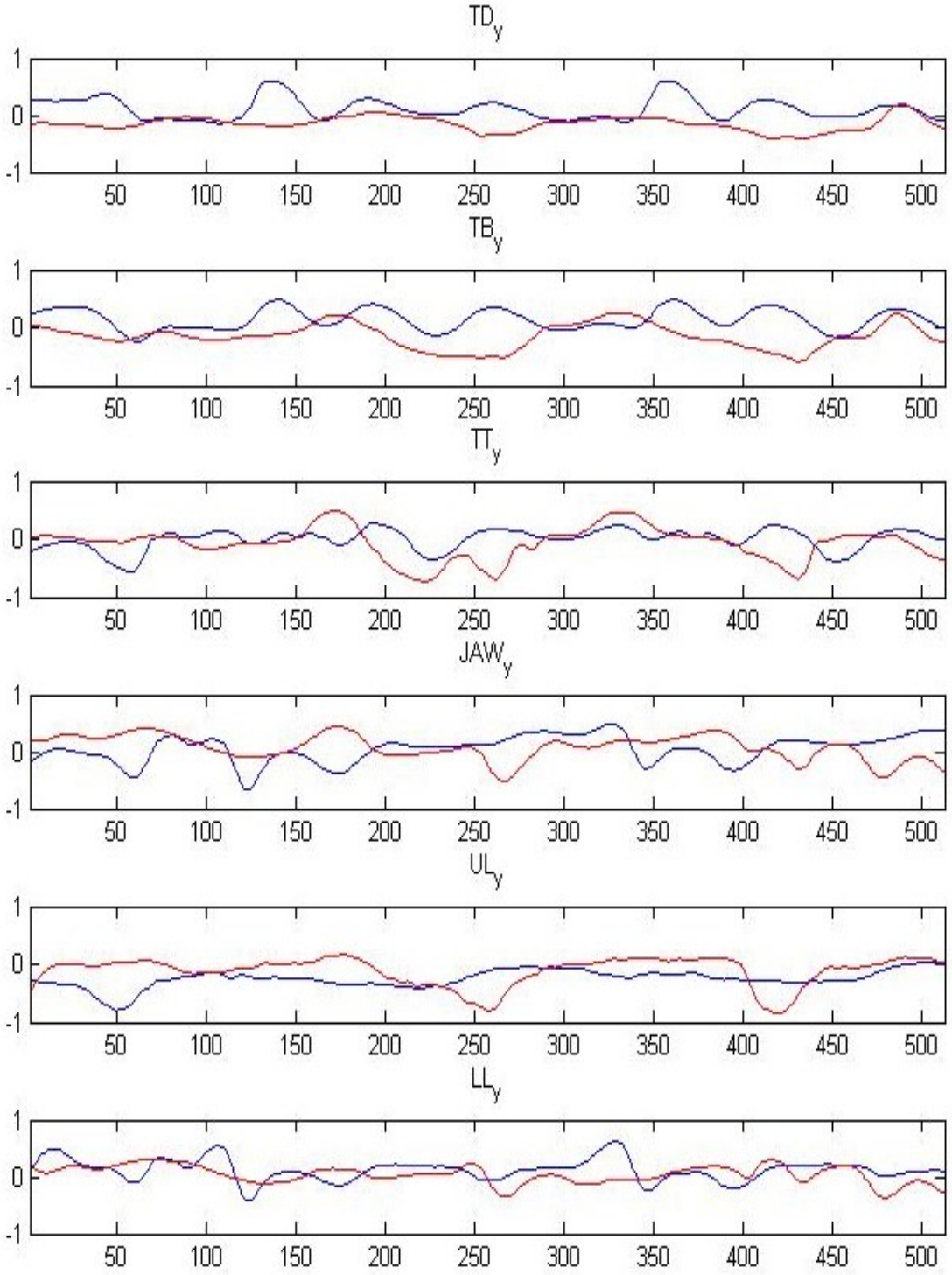

FIGURE 4.11: Articulator trajectories in the y-direction, blue from utterance mngu0-s1-0001 and red from utterance mngu0-s1-0002

network with structure 41-300-300-300-12. Even though the absolute difference between the articulators for the two experiments will not be directly comparable, the intra-difference between articulators in each respective experiment is still of interest. Uría's results for each articulator as well as the mean are shown in table 4.23. It is obvious that five outputs achieve a significantly lower RMSE value than the rest. The five from JAW_x down to LL_x are in both experiments significantly lower. These five comprise the lips and the jaw out of which the lower lip and jaw in the y-direction are the hardest to predict. Furthermore, the magnitude of each articulator with respect to the rest in each respective experiment is strikingly similar, with the exception of TT_y.

The superb performance of the five articulators mentioned above is interesting, however not very surprising. By comparing which articulators are easily predicted to their standard deviations in tables 4.21 and 4.22, the reason becomes obvious. All the articulators performing well also have a small standard deviation. This is, of course, intuitive because a variation over a smaller area is easier to track. The largest mistakes made by the network are, as seen in figure 4.9, when the largest movements away from the calibrated zero-position occur. The tongue moves both faster, and changes direction more frequently than the jaw and lips and thus has a higher standard deviation. In addition, as supported by both the standard deviation and intuition, the movement of the lower lip and lower incisor in the y-direction is far greater than of their movement in the x-direction due to the anatomy of the mouth cavity. All of this concurs with the results shown in these two dissertations.

Articulator	**RMSE**
TD_x	0.1143
TD_y	0.1503
TB_x	0.1268
TB_y	0.1225
TT_x	0.1221
TT_y	0.1277
JAW_x	0.0581
JAW_y	0.0722
UL_x	0.0324
UL_y	0.0454
LL_x	0.0647
LL_y	0.1083
Average	0.0954

TABLE 4.23: RMSE for each articulator obtained from Benigno Uría's Master's thesis [5].

Chapter 5

Conclusion

In this work we have tested how well a state-of-the-art implementation of a phone recognition system performs on a database comprised of Norwegian speech called NAFTA. The same system was used to run tests on the TIMIT database and the performance on the two databases was compared. The system performed significantly worse on the Norwegian dataset due to the so called long vowels which are observed in the Norwegian language. By introducing don't-care pairs which neglected these long vowels, the performance was improved by a large margin and achieved a phone error rate of 22.8%. The phone error rate obtained from the experiments run on TIMIT was 20.4% and it is obvious from these results that good performance is achievable, also when performing recognition on Norwegian speech. The 2.4 percentage point difference can be explained partially by the statistical distribution of the different phones for the different languages and also the different phonetic dictionaries used for the two datasets.

The confusion pairs observed for both languages follow a similar pattern where vowels dominate and certain consonant pairs are particularly frequent. In short, the main confusion within the consonant group is between the voiced and unvoiced version of each sound. These observations encouraged investigating the acoustic-articulatory problem. In this work we implemented a deep neural network which was pre-trained as a deep belief network and fine-tuned with the backpropagation algorithm to be used for regression. The output layer was set to be of size 12 to infer the position of 6 different articulators in the x- and y-direction in the midsaggital plane. Due to issues encountered when training the deep architecture in combination with the long run time of the training, the number of hidden layers

was reduced to 1. After overcoming these issues, a one-hidden layer network was trained successfully for regression and achieved an impressive RMSE precision of 1.348mm on the mngu0 dataset.

By comparing this RMSE value to the trajectory of each articulator it is clear that this is sufficient precision to be able to discriminate between a range of phones. Further experimentation is however needed to determine which phones can be reliably discriminated between before a similar system can be used in hybrid with a phone recognition system. The results displayed in this dissertation show that sufficient performance for a acoustic-articulatory inversion system is achievable and thus encourages further experimentation on combining articulator inversion with phone recognition to improve the state-of-the-art performance in this area.

5.1 Limitations

Due to the long run time of the programs implemented, the number of times each experiment could be run was limited. In order to obtain more precise results, each experiment should be run several times to obtain averages of the performance. It is however important to note that the scope of the articulatory inversion experiments in this work was not to push for the state-of-the-art performance, but rather to determine if the path of acoustic-articulatory inversion using (deep) neural networks is worth pursuing.

In this work, the articulatory inversion system is trained and tested with speech from a single individual. Due to large individual differences, size of the vocal tract in particular, the precision is likely to be reduced significantly when trained with several individuals. For a large individual, the standard deviations will likely be larger and has to be taken into account.

As explained in section 4.2.1, the target values for each frame was chosen to be the target values corresponding to the middle frame in the context window it belonged to. Strictly speaking, by doing so, one does not perform real regression. Another approach that may show better results, is to use linear interpolation between the target values used in these experiments to obtain the target. Alternatively, one can apply smoothing, e.g. low-pass filtering on the target values, but in this case use the correct target value for each frame instead of the context window value.

5.2 Future research

Despite not achieving as good results as the state-of-the-art implementations, the results from the articulatory inversion are promising and a hybrid system of articulatory inversion and phone recognition should be implemented. The reason for using a hybrid system instead of direct recognition by inversion is twofold. As mentioned in the previous section, the precision is likely to be reduced in a more realistic scenario and thus may not be accurate enough for recognition. Furthermore, several sounds can be produced with similar configurations of articulators, which is likely to lead to miss-classification. However, an articulatory inversion system can be used to limit the set of possible phones and potentially discriminate between pairs of phones that normally are subject to confusion.

To improve the results shown in this dissertation, an adaptive learning rate should be tested and also different learning rates for each layer should be implemented. When applying the backpropagation algorithm, the error, and thus also the gradient, will be smaller by several orders of magnitude as the error is propagated back. This leads to a very small update of the lower layers and hence little learning is achieved. In addition, in some areas of the parameter space, the gradient is very small which leads to slow training. If areas like this are detected, the learning rate can be increased so that the system can pass such a *plateau* more quickly and thus not "waste" epochs on little to no learning.

Another interesting research path is to use other representations of the data as input. More specifically, deep neural networks have shown good results within pattern recognition. A popular representation of speech is the spectrogram, a visual representation of the frequencies in a sound. A deep neural network could be trained to recognize patterns in the spectrograms and from that potentially achieve better performance in terms of phone discrimination.

Finally, due to the high temporal correlation between articulator samples, implementing a deep architecture which can capture temporal dependencies is likely to outperform the simple DBN-DNN structure implemented in this work. A very interesting network structure called CLDNN has proven to out-perform other deep architectures in a range of large vocabulary tasks [32]. The CLDNN is a combination of a Convolutional Neural Network (CNN), which reduces the spectral variation of the input feature, Long Short-Term Memory (LSTM) to perform temporal modeling, and finally a DNN which with the given input produces a feature

representation that is more easily separable. An architecture such as this is likely to be able to capture variations a normal DNN can not and thus potentially yield better results also for the articulatory inversion problem.

Appendix A

Matlab code

The main code base used for the articulatory inversion experiments was borrowed from the DeepLearnToolbox which is described in chapter 3. In this appendix, all the functions from this toolbox that were modified significantly are included. Aside from the main function, all the functions borrowed have their original function names and are credited to the developer of this toolbox. Any functions used in the functions shown here that are not included in this appendix are located in the DeepLearnToolbox and have only been modified slightly, or not at all.

A.1 Main script

```
function [er, nn] = test_DBN_NN
% Runs pre-training, training and test phase

% mngu0_prepared is a small dataset used as a dummy for fast results
load mngu0_prepared;
train_x = double(train_x);
test_x = double(test_x);
% valid_x = double(valid_x);
train_y = double(train_y);
test_y  = double(test_y);
% valid_y = double(valid_y);


%%  train a 300-300-300 hidden unit DBN and use its weights to initialize
```

```
%   an NN
rand('state',0)
%% Train DBN
dbn.sizes = [300];
opts.numepochs =   300;
opts.batchsize =   100;
opts.momentum  =   0.5;
opts.alpha     =   0.1;
opts.weightDecay = 0.0001;
dbn = dbnsetup(dbn, train_x, opts);
dbn = dbntrain(dbn, train_x, opts);

%% The following three lines are used to save and load pre-trained DBNs
% save('dbn_trained_400k', 'dbn', '-v7.3');
% nn=0;
% load('dbn_trained_400k');

%% unfold dbn to nn
nn = dbnunfoldtonn(dbn, 12, train_x, train_y);

%% DropoutFraction only used in case of dropout training.
% nn.dropoutFraction = 0.5;

%% train nn
opts.numepochs =   300;
opts.batchsize =   100;
nn = nntrain(nn, train_x, train_y, opts);

%% test nn
[er, nn] = nntest(nn, test_x, test_y);

assert(er < 0.2, 'Too big error');
```

A.2 DBN functions

```
function dbn = dbnsetup(dbn, x, opts)
    % Changed the initial weights to be drawn from N(0,0.01)
    % Changed the initial visible bias to be -4, -2 or N(0,0.01)
    n = size(x, 2);
    dbn.sizes = [n, dbn.sizes];
```

```
    for u = 1 : numel(dbn.sizes) - 1
        dbn.rbm{u}.alpha    = opts.alpha;
        dbn.rbm{u}.momentum = opts.momentum;
        dbn.rbm{u}.weightDecay = opts.weightDecay;

        dbn.rbm{u}.W  = randn(dbn.sizes(u + 1), dbn.sizes(u)).*0.01;
        dbn.rbm{u}.vW = zeros(dbn.sizes(u + 1), dbn.sizes(u));

        dbn.rbm{u}.b  = zeros(dbn.sizes(u), 1);
        dbn.rbm{u}.vb = zeros(dbn.sizes(u), 1);

%         dbn.rbm{u}.c  = zeros(dbn.sizes(u + 1), 1) - 2;
        dbn.rbm{u}.c = randn(dbn.sizes(u+1), 1).*0.01;
        dbn.rbm{u}.vc = zeros(dbn.sizes(u + 1), 1);
    end

end
```

```
function dbn = dbntrain(dbn, x, opts)
    % Changed the first rbm to be a gbrbm
    n = numel(dbn.rbm);

    dbn.rbm{1} = gbrbmtrain(dbn.rbm{1}, x, opts);
    for i = 2 : n
        opts.momentum = 0.5;
        %Propagate training input through previous RBM(s)
        x = rbmup(dbn.rbm{i - 1}, x);
        %Train next RBM
        dbn.rbm{i} = rbmtrain(dbn.rbm{i}, x, opts);
    end

end
```

```
function [nn] = dbnunfoldtonn(dbn, outputsize, x, y)
%DBNUNFOLDTONN Unfolds a DBN to a NN
%   dbnunfoldtonn(dbn, outputsize ) returns the unfolded dbn with a final
%   layer of size outputsize added.
    if(exist('outputsize','var'))
        sizez = [dbn.sizes outputsize];
    else
```

```
        sizez = [dbn.sizes];
    end
    nn = nnsetup(sizez);

    for i = 1 : numel(dbn.rbm)
        nn.W{i} = [dbn.rbm{i}.c dbn.rbm{i}.W];
    end

    %Performing pseudo-inverse method for regression layer initialization
    %This initializes the weights of the top layer to extreme values (~1e9)
    %Some sources suggest that the reason is the activations being too
    %small. The approach, however, is supported by many.
    %Due to this, currently initializing with random values
%     [activations, ~] = nnpredict(nn, x);
%     bias = mean(y);
%
%     hn = activations{end-1}(5:10:end,:);
%     yr = y(5:10:end,:);
%
%     for i = 1:size(yr,1)
%        yr(i,:) = yr(i,:) - bias;
%     end
%     nn.W{end} = (pinv(hn)*yr)';

end
```

```
function rbm = gbrbmtrain(rbm, x, opts)
    % The only difference from rbmtrain() is the calculation of v2
    % Also changed the reconstruction error to be RMSE
    % Momentum changes after epoch 10
    % gbrbm alpha = rbm.alpha/10 due to the sensitive gaussian layer
    % Using the probabilities as samples instead of binary values for the
    % bernoulli-layers
    % Added weight decay for the weight training
    assert(isfloat(x), 'x must be a float');
    m = size(x, 1);
    numbatches = floor(m / opts.batchsize);

    assert(rem(numbatches, 1) == 0, 'numbatches not integer');

    for i = 1 : opts.numepochs
        kk = randperm(m);
```

```
        err = 0;
        for l = 1 : numbatches
            batch = x(kk((l - 1) * opts.batchsize + 1 : l * ...
                opts.batchsize), :);

            v1 = batch;
            %v2h pass
            h1 = sigm(repmat(rbm.c', opts.batchsize, 1) + v1 * rbm.W');
            %h2v pass
            v2 = (repmat(rbm.b', opts.batchsize, 1) + h1 * rbm.W);
            %final v2h pass
            h2 = sigm(repmat(rbm.c', opts.batchsize, 1) + v2 * rbm.W');

            c1 = h1' * v1;
            c2 = h2' * v2;

            %Estimating numerical gradients
            rbm.vW = rbm.momentum * rbm.vW + (rbm.alpha/10) * ...
                ((c1 - c2) / opts.batchsize - rbm.weightDecay * rbm.W);
            rbm.vb = rbm.momentum * rbm.vb + (rbm.alpha/10) * ...
                sum(v1 - v2)' / opts.batchsize;
            rbm.vc = rbm.momentum * rbm.vc + (rbm.alpha/10) * ...
                sum(h1 - h2)' / opts.batchsize;

            %Updating weights
            rbm.W = rbm.W + rbm.vW;
            rbm.b = rbm.b + rbm.vb;
            rbm.c = rbm.c + rbm.vc;

            err = err + mean(sqrt(mean((v1 - v2) .^ 2)));
        end

        disp(['epoch ' num2str(i) ...
            '/' num2str(opts.numepochs) ...
            '. Average reconstruction error is: ' ...
            num2str(err / numbatches)]);

        if i == 10
           opts.momentum = 0.9;
        end
    end
end
```

```
function rbm = rbmtrain(rbm, x, opts)
    % Changed the reconstruction error to be RMSE
    % Momentum changes after epoch 10 to 0.9
    % Using the probabilities as samples instead of binary values for the
    % bernoulli-layers
    % Adde weight decay for the weight training

    assert(isfloat(x), 'x must be a float');
    m = size(x, 1);
    numbatches = floor(m / opts.batchsize);

    assert(rem(numbatches, 1) == 0, 'numbatches not integer');

    for i = 1 : opts.numepochs
        kk = randperm(m);
        err = 0;
        for l = 1 : numbatches
            batch = x(kk((l - 1) * opts.batchsize + 1 : l * ...
                opts.batchsize), :);

            v1 = batch;
            %v2h pass
            h1 = sigm(repmat(rbm.c', opts.batchsize, 1) + v1 * rbm.W');
            %h2v pass
            v2 = sigm(repmat(rbm.b', opts.batchsize, 1) + h1 * rbm.W);
            %final v2h pass
            h2 = sigm(repmat(rbm.c', opts.batchsize, 1) + v2 * rbm.W');

            c1 = h1' * v1;
            c2 = h2' * v2;

            %Estimating numerical gradients
            rbm.vW = rbm.momentum * rbm.vW + rbm.alpha * ((c1 - c2) / ...
                opts.batchsize - rbm.weightDecay * rbm.W);
            rbm.vb = rbm.momentum * rbm.vb + rbm.alpha * sum(v1 - v2)' /...
                opts.batchsize;
            rbm.vc = rbm.momentum * rbm.vc + rbm.alpha * sum(h1 - h2)' /...
                opts.batchsize;

            %updating weights
            rbm.W = rbm.W + rbm.vW;
            rbm.b = rbm.b + rbm.vb;
            rbm.c = rbm.c + rbm.vc;
```

```
            err = err + mean(sqrt(mean((v1 - v2) .^ 2)));
        end

        disp(['epoch ' num2str(i) ...
            '/' num2str(opts.numepochs) ...
            '. Average reconstruction error is: ' ...
            num2str(err / numbatches)]);

        if i == 10
           opts.momentum = 0.9;
        end
    end
end
```

```
function createWeightHistogram( dbn )

W = reshape(dbn.rbm{1}.W,1,[]);
figure
hist(W,40), title('Histogram of weights'), xlabel('Bin values'), ...
    ylabel('Entries in bin');

end
```

A.3 NN functions

```
function nn = nnapplygrads(nn, i)
%NNAPPLYGRADS updates weights and biases with calculated gradients
% nn = nnapplygrads(nn) returns an neural network structure with updated
% weights and biases

    % Applies back-propagation only to the output layer for N epochs, where
    % N is given by i < N+1
    if i < 1
        temp = nn.n-1;
        if(nn.weightPenaltyL2>0)
            dW = nn.dW{temp} + nn.weightPenaltyL2 * ...
                [zeros(size(nn.W{temp},1),1) nn.W{temp}(:,2:end)];
        else
```

```
            dW = nn.dW{temp};
        end

        dW = nn.learningRate * dW;

        if(nn.momentum>0)
            nn.vW{temp} = nn.momentum*nn.vW{temp} + dW;
            dW = nn.vW{temp};
        end

        nn.W{temp} = nn.W{temp} - dW;
    else
        for i = 1 : (nn.n - 1)
            %The following varies the learning rate for the different
            %layers
            if i == 1
                nn.learningRate = 0.5;
            else
                nn.learningRate = 0.01;
            end
            if(nn.weightPenaltyL2>0)
                dW = nn.dW{i} + nn.weightPenaltyL2 * ...
                    [zeros(size(nn.W{i},1),1) nn.W{i}(:,2:end)];
            else
                dW = nn.dW{i};
            end

            dW = nn.learningRate * dW;

            if(nn.momentum>0)
                nn.vW{i} = nn.momentum*nn.vW{i} + dW;
                dW = nn.vW{i};
            end

            nn.W{i} = nn.W{i} - dW;
        end
    end
end
```

```
function [activations, nn] = nnpredict(nn, x)
    %NNPREDICT performs a feed-forward pass to obtain output activations
    %It returns activations for all layers for debugging purposes
```

```
    nn.testing = 1;
    nn = nnff(nn, x, zeros(size(x,1), nn.size(end)));
    nn.testing = 0;

    activations = nn.a;
end
```

```
function [er, nn] = nntest(nn, x, y)

    % Estimate predictions
    [labels, nn] = nnpredict(nn, x);

    % Filter predictions
    filt = fir1(30,0.1);
    activations = [];
    for i = 1:size(labels{end},2)
        activations = [activations filter(filt,1,labels{end}(:,i))];
    end
    difference = (y-activations);

    %Plotting the estimated trajectory versus the real and also plotting
    %the difference in a separate subplot
    figure
    subplot(2,1,1)
    plot(y(:,1), 'r'), title('Estimated trajectory vs. target'), ...
        xlabel('sample'), ylabel('Position [cm]');
    hold on
    plot(activations(:,1), 'b');
    subplot(2,1,2)
    plot(abs(difference(:,1))), title('Absolute difference'), ...
        xlabel('sample'), ylabel('Absolute diff [cm]')

    % Multiply by the standard deviations from before z-scoring to obtain
    % the error in cm
    load('ema_stds.mat');
    for i = 1:size(y,1)
        difference(i,:) = difference(i,:) .* (4*stds(1:12));
    end
    er =  sqrt(mean(difference .^ 2));

end
```

```
function nnCreateWeightHistogram( nn )

W = reshape(nn.W{1,1},1,[]);
figure
hist(W,15), title('Histogram of weights'), xlabel('Bin values'), ...
    ylabel('Entries in bin');

end
```

```
function createActivHistogram( nn )

a = reshape(nn.a{2},1,[]);
figure
hist(a,15), title('Histogram of activations'), xlabel('Bin values'), ...
    ylabel('Entries in bin');

end
```

A.4 Data preparation

```
close all
clear
clc

matFileName = 'trainOutput';
outFileName = strcat(matFileName, '.txt');
trainFileName = 'trainfiles.txt';
testFileName = 'testfiles.txt';
validFileName = 'validationfiles.txt';

train_x = []; %output matrix for LSFs
train_y = []; %output matrix for labels

test_x = []; %output matrix for LSFs
test_y = []; %output matrix for labels

valid_x = []; %output matrix for LSFs
valid_y = []; %output matrix for labels
```

```
tic
disp('... Retrieving training data');
fid = fopen(trainFileName);
tline = fgetl(fid);

while ( ischar(tline) )

    i_lsf = strcat(tline, '.lsf');
    i_ema = strcat(tline, '.ema');

    [lsf, ~, ~] = estload(i_lsf);
    [ema, ~, ~] = estload(i_ema);

    for j = 1:size(lsf,1)
        for k = 1:10
            train_x = [train_x; lsf(j,(k-1)*41+1:k*41)];
            train_y = [train_y; ema(j,1:12)];
        end
    end
    t = toc;
    disp(strcat(tline, ' ... ', num2str(t)));
    tline = fgetl(fid);
end

fclose(fid);
t = toc;
disp(strcat('Training set extraction took ', num2str(t), 'seconds'));

tic
disp('... Retrieving validation data');
fid = fopen(validFileName);
tline = fgetl(fid);

while ( ischar(tline) )
    j_lsf = strcat(tline, '.lsf');
    j_ema = strcat(tline, '.ema');
    [lsf, ~, ~] = estload(j_lsf);
    [ema, ~, ~] = estload(j_ema);

    for j = 1:size(lsf,1)
        for k = 1:10
            valid_x = [valid_x; lsf(j,(k-1)*41+1:k*41)];
            valid_y = [valid_y; ema(j,1:12)];
```

```
        end
    end
    t = toc;
    disp(strcat(tline, ' ... ', num2str(t)));
    tline = fgetl(fid);
end

fclose(fid);
t = toc;
disp(strcat('Validation set extraction took ', num2str(t), ' seconds'));

tic
disp('... Retrieving test data');
fid = fopen(testFileName);
tline = fgetl(fid);

while ( ischar(tline) )
    k_lsf = strcat(tline, '.lsf');
    k_ema = strcat(tline, '.ema');
    [lsf, ~, ~] = estload(k_lsf);
    [ema, ~, ~] = estload(k_ema);

    for j = 1:size(lsf,1)
        for k = 1:10
            test_x = [test_x; lsf(j,(k-1)*41+1:k*41)];
            test_y = [test_y; ema(j,1:12)];
        end
    end
    t = toc;
    disp(strcat(tline, ' ... ', num2str(t)));
    tline = fgetl(fid);
end
fclose(fid);
t = toc;

disp(strcat('Test set extraction took ', num2str(t), 'seconds'));
save('mngu0_prepared', 'train_x', 'train_y', 'valid_x', 'valid_y', ...
    'test_x', 'test_y');
```

```
function createTestFromTrain(filename)

load(filename)
```

```
new_train_x = [];
new_train_y = [];
new_test_x = [];
new_test_y = [];
for i = 1:(size(train_x,1)/10)
    new_train_x = [new_train_x; train_x(10*(i-1)+1:10*(i-1)+8,:)];
    new_train_y = [new_train_y; train_y(10*(i-1)+1:10*(i-1)+8,:)];
    new_test_x = [new_test_x; train_x(10*(i-1)+9:10*(i-1)+10,:)];
    new_test_y = [new_test_y; train_y(10*(i-1)+9:10*(i-1)+10,:)];
end

train_x = new_train_x;
train_y = new_train_y;
test_x = new_test_x;
test_y = new_test_y;

save('mngu0_8_2_separated_train', 'train_x', 'train_y', 'test_x', 'test_y')
end
```

Bibliography

[1] KALDI at sourceforge. http://kaldi.sourceforge.net/index.html. Accessed: 2015-04-29.

[2] TIMIT acoustic-phonetic continuous speech corpus. https://catalog.ldc.upenn.edu/LDC93S1. Accessed: 2014-12-04.

[3] Korin Richmond, Phil Hoole, and Simon King. Announcing the electromagnetic articulography (day 1) subset of the mngu0 articulatory corpus. In *Proc. Interspeech*, pages 1505–1508, august 2011.

[4] Jangwon Kim, Asterios Toutios, Sungbok Lee, and Shrikanth S. Narayanan. A kinematic study of critical and non-critical articulators in emotional speech production. *The Journal of the Acoustical Society of America*, 137(3):1411–1429, 2015.

[5] Benigno Uría. A deep belief network for the acoustic-articulatory inversion mapping problem. Master's thesis, University of Edinburgh, 2011.

[6] J. Baker, L. Deng, J. Glass, S. Khudanpur, C.-H. Lee, N. Morgan, and D. O'Shaughnessy. Developments and directions in speech recognition and understanding, part 1. *Signal Processing Magazine, IEEE*, 26(3):75–80, 2009.

[7] D. O. Hebb. *The Organization of behavior*. Wiley & Sons, 1949.

[8] R. Rosenblatt. *Principles of Neurodynamics: perceptrons and the theory of brain mechanisms*. Spartan Books, 1959.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, (323):533 – 536, 1986.

[10] G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, (18):1527–1554, 2006.

[11] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009. doi: 10.1561/2200000006. Also published as a book. Now Publishers, 2009.

[12] P. Huang, H. Avron, T. Sainath, V. Sindhwani, and B. Ramabhad ran. Kernel methods match deep neural networks on TIMIT. *IEEE International Conference on Acoustics, Speech, and Signal Processing(ICASSP)*, 2014.

[13] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[14] Hervé Bourlard and Nelson Morgan. *Connectionist speech recognition: a hybrid approach*. Kluwer Academic Publishers, 1994.

[15] Stuart J. Russel and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Pearson Education, 3 edition, 2010.

[16] James A. Freeman and David M. Skapura. *Neural Networks: Algorithms, Applications and Programming Techniques*. Addison-Wesley Publishing Company, inc., 1991.

[17] M. A. Carreira-Perpinan and G. E. Hinton. On contrastive divergence learning. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS'05)*, pages 33–40. Society for Artificial Intelligence and Statistics, 2005.

[18] G. E. Hinton. *A practical guide to Training Restricted Boltzmann Machines*, 1. edition, 2010.

[19] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems*, (19):153–160, 2007.

[20] Aleix M. Martínez and Avinash C. Kak. PCA versus LDA, journal = IEEE transactions on pattern analysis and machine intelligence, volume = 23, number = 2, year = 2001, pages = 228–233.

[21] Tae-Kyun Kim and Josef Kittler. Locally linear discriminant analysis for multimodally distributed classes for face recognition with a single model image. *IEEE transactions on pattern analysis and machine intelligence*, 27(3): 318–327, 2005.

[22] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, pages 257–286. IEEE, 1989.

[23] J. V. Psutka and A. Prazak. Multiple application of the MLLT based on clustering supported by phonetic knowledge. In *9th International Conference on Signal Processing*. IEEE, 2008.

[24] Mark JF Gales. Maximum likelihood linear transformations for HMM-based speech recognition. *Computer speech & language*, 12(2):75–98, 1998.

[25] J. S. Perkell, M. H. Cohen, M. A. Svirsky, M. L. Matthies, I. Garabieta, and M. T. Jackson. Electromagnetic midsagittal articulometer systems for transducing speech articulatory movements. *The Journal of the Acoustical Society of America*, 92(6):3078–3096, 1992.

[26] 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.

[27] Deeplearntoolbox at github. https://github.com/rasmusbergpalm/DeepLearnToolbox. Accessed: 2015-04-30.

[28] R. B. Palm. Prediction as a candidate for learning deep hierarchical models of data. Master's thesis, 2012.

[29] H.W. Strube. Linear prediction on a warped frequency scale. *The Journal of the Acoustical Society of America*, 68:1071, 1980.

[30] K-F Lee and H-W Hon. Speaker-independent phone recognition using hidden Markov models. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(11):1641–1648, 1989.

[31] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overitting. *Journal of Machine Learning Research*, (15):1929–1958, 2014.

[32] T.N. Sainath, O. Vinyals, A. Senior, and H. Sak. Convolutional, long short-term memory, fully connected deep neural networks. In *40th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.