NTNU

Norwegian University of
Science and Technology

Speech Enhancement with Deep Neural Networks

Olav Klungsøyr Melve

Master of Science in Electronics
Submission date: June 2016
Supervisor: Magne Hallstein Johnsen, IET
Co-supervisor: Øystein Birkenes, Cisco

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Master Thesis spring 2016

for

Olav Klungsøyr Melve

June 10, 2016

Speech Enhancement with Deep Neural Networks

Many speech based applications are hampered by unwanted signals. This applies to speech communication (telephone), recognition, diarization, meetings, industrial environments etc. The sources of the unwanted signals can be other persons (overlapping speech) or different kind of noises. Thus noise reduction has been a major research area for many years.

Conventional signal processing methods like Wiener filters, spectral subtraction, active noise control, voice activity detection etc. have been widely applied. Further, the methods should show strong correlation between objecive and subjective quality criterias.

The performance of most speech based applications have recently shown a significant improvement due to the introduction of machine learning approaches. This improvement is mainly due to the recent success of learning in so called deep neural networks (DNN).

In this master task, suggested by Cisco, the student shall implement and evaluate one such DNN approach to speech enhancement [1]. Both performance for white noise and robustness with respect to different kinds of noise sources [2] shall be investigated. Further the DNN solution should be robust over different signal to noise ratios. The quality of the method shall be evaluated both by objective measures like segmental SNR and a measure correlated by subjective quality scores PESQ [3]. Finally the quality shall be compared to a standard signal processing method called OM-LSA [4].

Supervisor 1: Associate professor Magne Hallstein Johnsen, NTNU
Supervisor 2: Software engineer Øystein Birkenes, Cisco Systems Norway

Summary

This master thesis describes the implementation and evaluation of a promising approach to speech enhancement based on deep neural networks. A baseline system was implemented and trained using noisy data synthesized by combining speech from the TIMIT database with white Gaussian noise and recorded background noise signals from the Aurora2 database. Several techniques for improving the system, some proposed in other papers and some original, were implemented and evaluated. The quality of the enhanced speech has been assessed by comparison with the reference clean speech using mean square error (MSE) in the short time log-magnitude spectrum, segmental signal-to-noise-ratio estimates on the waveforms, and a ITU-T standard method called Perceptual Evaluation of Speech Quality (PESQ).

Of the implemented techniques, using *dropout* during training was shown in a small experiment to give better results for the MSE, but worse or no better results in terms of PESQ. Another technique called *global variance equalization* had the opposite effect, negatively affecting MSE, but significantly improving the PESQ results. An experiment replacing the sigmoid activation functions of the deep neural network with the increasingly popular *rectified linear units* indicated that the latter setup could achieve as good or better performance without using greedy layer-wise pretraining.

In addition to comparing variations of the deep neural network, the speech enhancement system was compared to a standard signal processing method called OM-LSA. The deep neural network based system giving best performance resulted in superior PESQ score, but, in some cases, worse segmental SNR than what was achieved with OM-LSA. Two hybrid systems combining OM-LSA with the DNN system were proposed, and one showed a significant improvement in MSE for test set with unseen noise over the DNN alone. In terms of PESQ, however, using the DNN alone gave better results than both hybrid systems.

Testing the DNN systems performance for different sound classes gave some more insight into what the method is good, and less good, at. An attempt to understand the parameters of the trained DNN led to a new interpretation of the system as one identifying speech features in noise and choosing from a set of ”basis vectors” how to best estimate the clean speech log-magnitude spectrum. This was considered an interesting perspective even if it might not be the ”correct” interpretation.

Some limited subjective evaluation was performed by the student by listening to files from the test set enhanced by the system. This revealed that the system performs very well in certain cases, also for unseen noise, but results in distorted speech of low experienced quality in other. This was especially true for files dominated by noise that were mismatched to the training data.

Sammendrag

Denne masteroppgaven beskriver implementasjon og testing av en lovende teknikk for taleforbedring med bruk av dype nevrale nettverk. Et grunnleggende system ble implementert og trent med støyet data syntetisert ved å kombinere tale fra TIMIT databasen med hvit gaussisk støy og opptak av bakgrunnsstøy fra Aurora2 databasen. Flere teknikker for forbedring av systemet ble implementert og testet, enkelte foreslått andre artikler og enkelte originale. Kvaliteten til talen etter støyfjerning ble målt ved sammenligning av ren of støyet tale med middels kvadratisk avvik i log-magnitude spekteret til rammer av signalene, segmentelt signal-til-støy-forhold estimert basert på bølgeformen og en standard utviklet av ITU-T kalt Perceptual Evaluation of Speech Quality (PESQ).

Av de implementerte teknikkene førte *dropout*, basert på et lite eksperiment, til en forbedring i midlere kvadratisk avvik, men ingen forbedring i PESQ. En annen teknikk, *global variance equalization* viste den motsatte tendensen: en negativ effekt på det kvadratiske avviket og en forbedring av PESQ målet. Et eksperiment hvor aktiveringsfunksjonen i nettverket ble byttet ut med en mer moderne likerettet lineær funksjon indikerte at dette oppsettet kunne resultere i like god eller bedre ytelse uten å bruke en mye brukt teknikk for lagvis pretrening av nettverket.

I tillegg til å sammenligne variasjoner av det dype nevrale nettverket, ble taleforbedringssystemet også sammenlignet med en tradisjonell signalprosesseringsteknikk kalt OM-LSA. Det dype nevrale nettverket som hadde beste testresultater resulterte i forbedret tale med høyere PESQ, men for noen data dårligere segmentelt signal-til-støy-forhold enn OM-LSA. To hybridsystem som kombinerte de to metodene, OM-LSA og DNN, ble presentert og testet. Det ene systemet viste signifikant forbedring i kvadratisk avvik for test data med usette støytyper. For PESQ målet viste det seg imidlertid å være bedre å bruke DNN systemet alene.

Testing av systemet på ulike lydklasser gav økt innsikt i hva systemet er bra og mindre bra til. Et forsøkt på å forstå de trente parametrene til det dype nevrale nettverket resulterte i en ny forståelse av estimeringsprosessen som en kombinasjon av ”basisvektorer” som til sammen danner det beste estimatet av utgangsrammen. Uansett om dette er en riktig tolkning av nettverket eller ikke ble det vurdert som en interessant måte å tenke om systemet på.

Noen subjektive vurderinger basert på lytting til resultatet av de forskjellige teknikkene ble foretatt for enkelte filer fra testsettene. Disse testene indikerte at systemet har god ytelse i mange tilfeller, men resulterer i tale med dårlig kvalitet i andre tilfeller. Dette er særlig tilfellet for filer hvor signalet domineres av støytyper som er veldig ulik de som ble inkludert i treningen av nettverket.

Preface

The following text is written by Olav Klungsøyr Melve for my master thesis in the Electronics Engineering study program with specialization in Signal Processing at the Norwegian University of Science and Technology. The work was carried out during the fall semester of 2016. The master task was formulated by Cisco Systems Norway and Øystein Birkenes has been my external supervisor and contact person at Cisco, together with Svein Gunnar Pettersen. My supervisor at NTNU has been Magne Hallstein Johnsen. I want to thank all three for supervising and helping me throughout the course of the work with the master project. Without their guidance and feedback this document would not have seen the light of day.

June 10, 2016

Table of Contents

List of Tables

List of Figures

Abbreviations

ANN	=	artificial neural network
NN	=	neural network
DNN	=	deep neural network
GPU	=	graphics processing unit
MLP	=	multilayer perceptron
RBM	=	restricted Boltzmann machine
GB-RBM	=	Gaussian-Bernoulli restricted Boltzmann machine
ReLU	=	rectified linear units
MSE	=	mean-squared error
SSE	=	sum of squared errors
PESQ	=	perceptual evaluation of speech quality
SNR	=	signal-to-noise ratio
MOS	=	mean opinion score
GVE	=	global variance equalization
NAT	=	noise aware training

Chapter 1

Introduction

In communication systems and other speech based applications, the signal of interest – speech – will often be corrupted by background noise and other interfering signals. The field of speech enhancement aims to remove this corruption by use of signal processing algorithms, improving the quality of speech signals as a result. Many traditional techniques such as the Wiener filter, spectral subtraction, voice activity detection and more, have found widespread use and give good performance when the signal-to-noise-ratio (SNR) is relatively high. For low SNRs, however, the traditional methods often lead to annoying residual noise, like the so-called ”musical noise”. Traditional approaches also frequently make assumptions about the corruption that might not hold in all scenarios. For example, a much used assumption of stationary or slowly varying noise statistics in classical speech enhancement methods results in these algorithms having poor performance for non-stationary or transient noises.

With the growing popularity of machine learning in deep neural networks (DNN), new approaches have been shown to result in state-of-the-art performance for many traditional signal processing problems. Instead of finding analytical solution, based on assumptions and approximations, these methods use enormous sets of data to train complex models that implements the wanted functionality. Also in the field of speech enhancement, several papers have reported good results using these kinds of machine learning techniques. In one particular paper [1], a DNN based system trained to minimize the mean square error of noisy speech in the log-magnitude spectrum demonstrated very promising noise reduction capabilities for non-stationary noise, without introducing the mentioned ”musical noise”. this system.

The work described in this master thesis is a continuation of a preliminary project completed by the student in the preceding semester, implementing a limited version of the system from [1] for enhancement of speech corrupted by additive white Gaussian noise. The objective of the master work was to build on this, in order to implement and evaluate a more complete version of the DNN based speech enhancement system. Part of the practical challenge involved transitioning to more appropriate software for machine learning than what was used in the preliminary project. This gave the possibility of massively

accelerating the model training by use of a GPU.

Recreating the results from [1] was not considered realistic under the scope of the master task. Rather, the contribution of this master thesis was intended to be a more in-depth understanding of the system and how the different variations affect the performance. Some possible alternative setups outside of [1] were also discussed, and an attempt was made to interpret some of the learned parameters based on knowledge of the input and output signals.

The document is structured as follows:

Chapter 2: Contains the basic theory needed to understand the rest of the thesis. It focuses mainly on artificial neural networks and deep learning, but also contains a short description of a traditional speech enhancement algorithm used for comparison and the quality measures used for testing.

Chapter 3: Contains a short description of the databases used, how noisy data was synthesized and what software tools and resources were used.

Chapter 4: Contains a description of the speech enhancement system, the general training and testing procedure, techniques implemented to improve the performance of the system and proposed variations in the setup of the system.

Chapter 5: Contains an analysis of the system through presentation and discussion of the results from the experiments.

Chapter 6: Contains suggestions for future work.

Chapter 7: Contains the conclusion.

Chapter 2

Basic Theory

2.1 Overview

This chapter contains the theoretical basis for the work presented in the thesis. The focus has been on explaining the central concepts of artificial neural networks including some details regarding architecture and training procedures. As the master thesis is a continuation of project [5] completed by the student in the previous semester, much what was written in the theory chapter there is also relevant for this thesis. The sections 2.2, 2.3 and 2.6 are copied directly from the project report with minor corrections and a few appended paragraphs. The remaining sections are either extensively rewritten and reorganized or did not appear in [5].

2.2 Artificial Neural Networks

Artificial neural networks (ANNs) are a family of machine learning models that can be trained to perform many complicated tasks. Originally inspired by our understanding of biological nervous systems, ANNs are mostly made up of simple computational elements (neurons) in a densely connected network structure [8]. Neural networks are used to perform non-linear mappings from input to output data, which can be trained to realize a wide variety of functions. There are a number of different families of neural networks that vary in the operations performed in individual nodes, in how the nodes are connected together, and in the algorithms and procedures used for training.

The basic building block of most ANNs are neurons like the one illustrated in Figure 2.1. This neuron receives inputs from other nodes in the neural network through the incoming connections, including a bias (or offset) that is not connected to any other node. Each of the connections have a weight that is multiplied with the incoming value they are communicating to the neuron. In the figure the bias is illustrated as a connection having weight w_0 (the value of the bias) and input always equal to 1. The sum of weighted inputs to the neuron is passed to an *activation function* that computes the output, or activation, of

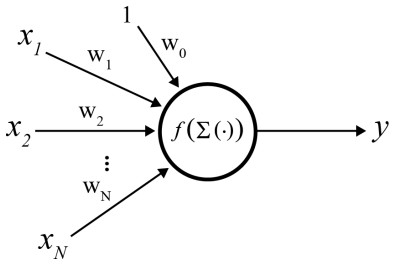


Figure 2.1: A single neuron. [5]

the neuron (Eq. 2.1).

$$y = f\left(\sum_{i=1}^{N} w_i x_i + w_0\right) \tag{2.1}$$

By defining data and weight vectors $\mathbf{x} = [1, x_1, x_2, ..., x_N]^T$ and $\mathbf{w} = [w_0, w_1, w_2, ..., w_N]^T$ we can write Equation 2.1 using vector notation:

$$y = f\left(\mathbf{w}^T \mathbf{x}\right) \tag{2.2}$$

Traditionally, common choices of f include the sigmoid function in Equation 2.3 and the hyperbolic tangent among others [9]. In some cases the neurons might simply be linear, meaning the activation function output equals its input: $f(\mu) = \mu$.

$$f(\mu) = \frac{1}{1 + e^{-\mu}} \tag{2.3}$$

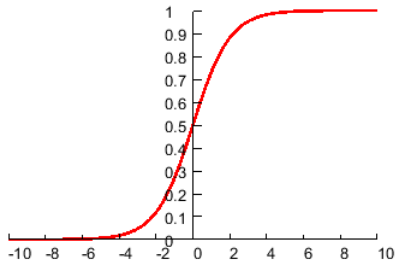


Figure 2.2: The sigmoid function. [5]

If the absolute values of the weights are big and the input values not too small, neurons with activation functions like the sigmoid can be thought of to be either ”on” or ”off”, corresponding to an output that is either 1 or 0 (see Figure 2.2). For small weights, however, the input sum might be close to zero, meaning the output of the node will vary, approximately linearly, between values close to 0 and 1.

Ignoring the transitional part of the function, the negative of the bias value acts as a threshold that the weighted sum of inputs must be greater than to turn on the neuron (making the argument in Equation 2.1 greater than 0).

In recent years rectified non-linearities like the rectified linear unit (ReLU) in Equation 2.4 and Figure 2.3 have become very popular, especially for classification tasks. In [10] it was found that *"using a rectifying non-linearity is the single most important factor in improving the performance of a recognition system"*. Others have found that these types of activation functions work equally well or better than hyperbolic tangent functions, and that they are interesting because they provide a better model of biological neurons [11]. [12] calls the replacement of sigmoid units with rectified linear units one of two major algorithmic changes that has *"greatly improved the performance of feedforward networks"*. Generalizations of the rectifier function in Equation 2.4 exist that have a non-zero gradient for negative input values. An example is the one used in so-called *leaky ReLUs*, defined in Equation 2.5. Here α is a small positive constant like 0.01 [12]. The gradient for this function is 1 for positive μ and α for negative μ, and by setting $\alpha = 0$ we get the standard rectifier function.

$$f(\mu) = max(\mu, 0) \tag{2.4}$$

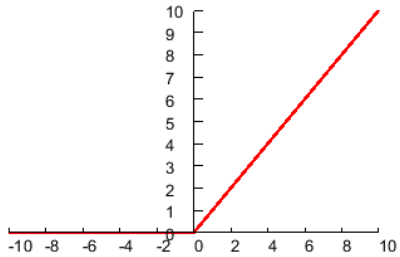


Figure 2.3: The rectified linear function is a leaky ReLU with $\alpha = 0$

$$f(\mu) = max(\mu, 0) + \alpha min(\mu, 0) \tag{2.5}$$

An important subclass of ANNs are so called feedforward neural networks, briefly mentioned in the previous paragraph. In these networks the nodes are organized in a directed acyclic graph from input nodes to output nodes. A special case of these kinds of networks is the multilayer perceptron (MLP). Here the nodes are organized in layers with connections only between neighboring layers [9]. The nodes in the first layer are simply the input features that are fed to the network, meaning they do not perform any computations. The final layer is the output layer and any layers preceding this – except the input layer – are called *hidden layers*. A simple MLP is illustrated in Figure 2.4. The biases of the neurons are drawn in red and are again represented as ones connected to the neurons through weighted connections.

When numbering the layers in a MLP many choose to omit the input layer and only count the layers containing the neurons that perform computations. By this convention the neural network in Figure 2.4 is a two-layer MLP. The nodes in the hidden and output layers are of the kind illustrated in Figure 2.1.

We can organize the output values from one layer in a vector and use notation similar to Equation 2.2, only now using weight matrices for every layer of connections. Figure

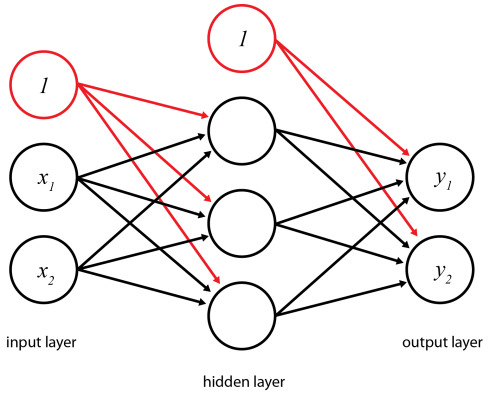


Figure 2.4: A simple multilayer perceptron (MLP). Previously featured in [5]

2.5 shows an alternative representation of the MLP from Figure 2.4, using vectors and matrices to describe the network. The input vector $\mathbf{x} = [1, x_1, x_2]^T$ is multiplied by the transposed weight matrix for the first layer of connections $\mathbf{W}_1 = [\mathbf{w}_1^{(1)}, \mathbf{w}_2^{(1)}, \mathbf{w}_3^{(1)}]$. Each column in $\mathbf{W}_1$ is a weight vector, like the one in Equation 2.2, describing the connections to one of the nodes in the hidden layer. The subscript indicates which of the three nodes the connections lead to and the superscript $^{(1)}$ indicates that it's the first layer of connections. The matrix-vector product is stored in $\mathbf{r}$ and the activation function of the neurons is applied to the elements of this vector to compute the layer output, as written in Equation 2.6.

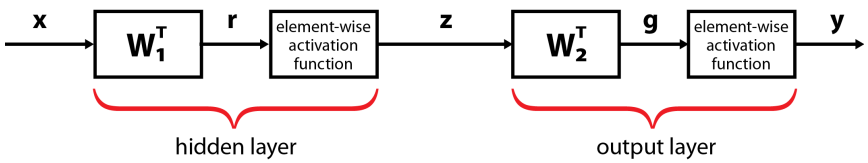


Figure 2.5: Block diagram for MLP in Figure 2.4. [5] Based on [6]

$$\tilde{\mathbf{z}} = f_1(\mathbf{r}) = f_1\left(\mathbf{W}_1^T\mathbf{x}\right) \tag{2.6}$$

To include possible biases a 1 is concatenated with the output vector: $\mathbf{z} = [1, \tilde{\mathbf{z}}^T]^T$. This is not illustrated in the figure. The concatenated vector serves as input to the next layer, and the output of the MLP is computed as:

$$\mathbf{y} = f_2(\mathbf{g}) = f_2\left(\mathbf{W}_2^T\mathbf{z}\right) \tag{2.7}$$

Note that here the same activation function is used for all neurons in the same layer, but different functions might be used for the hidden and output layers. For both layers, and in general, the activation function is computed element-wise on the matrix-vector product.

2.3 Backpropagation

Multilayer perceptrons are usually trained in a supervised manner, using labeled training data. The set of training data is a collection of input-output pairs $[\{\mathbf{x}_k, \mathbf{t}_k\}, k = 1, 2, ..., M]$ where $\mathbf{t}_k$ is the ideal output vector, or *target*, for input vector $\mathbf{x}_k$, and M is the size of the training set. An objective function is defined for measuring the disparity between the actual output $\mathbf{y}_k$ and the ideal output $\mathbf{t}_k$ for every input vector $\mathbf{x}_k$ in the set. The training procedure aims to minimize this objective function for the data in the training set by updating the parameters θ of the network. A few popular choices for the objective function are the sum of squared errors (SSE), mean-squared-error (MSE) and cross-entropy error function [9]. For an output layer having N elements the SSE function is written in Equation 2.8. The θ is the set of parameters of the MLP. In [9] the MSE function is defined as in Equation 2.9, which is simply the SSE function normalized by the size of the training set and number of output elements. However, objective functions normalizing over the training set, but not the number of vector elements, are also often referred to as MSE, e.g. in [1]. The factor $\frac{1}{2}$ is not necessary, but included to get nicer derivatives of the functions.

$$J_{SSE}(\theta) = \frac{1}{2}\sum_{k=1}^{M}\sum_{i=1}^{N}(y_{ki} - t_{ki})^2 \tag{2.8}$$

$$J_{MSE}(\theta) = \frac{1}{2NM}\sum_{k=1}^{M}\sum_{i=1}^{N}(y_{ki} - t_{ki})^2 = \frac{1}{NM}J_{SSE}(\theta) \tag{2.9}$$

The most used algorithm for training MLPs is the backpropagation algorithm. It computes the derivatives of the objective function with respect to the weights of the network and use them to update the weights in a gradient descent like manner [9]. This means the weight update for the connection going from node i in layer $l-1$ to node j in layer l is set proportional to the negative of the objective function J differentiated with respect to the weight of that connection. Assuming the notation $\Delta w = w_{new} - w_{old}$ this can be written as in Equation 2.10. The parameter α is a small positive value called the *learning rate*.

$$\Delta w_{ji}^{(l)} = -\alpha\frac{\partial J(\theta)}{\partial w_{ji}^{(l)}} \tag{2.10}$$

Since backpropagation uses a gradient based update rule it is clear that the activation function must be differentiable. Rectified linear units have previously been avoided because of the non-differentiable point for inputs equaling zero [12], however, in practice this has been shown not to be a problem. Both the sigmoid (Eq. 2.3) and the hyperbolic tangent also have simple, continuous derivatives and are much used in feedforward networks trained by backpropagation. In particular, if $y = f(\mu)$ where f is the sigmoid function we can write its derivative as:

$$\frac{dy}{d\mu} = y(1 - y) \tag{2.11}$$

Assuming some initial value for the weights in the network, the backpropagation algorithm first calculates the output vector for one, several or all of the input vectors in the

training set. This is called the forward pass. To understand how the backpropagation algorithm works it is useful to take a closer look at the last layer of the network. In the following the y's are neuron outputs and g's are the sums of weighted inputs to the neurons. Subscripts of these variables denote the neuron number and the superscripts the layer number. The following derivations are based on [13].

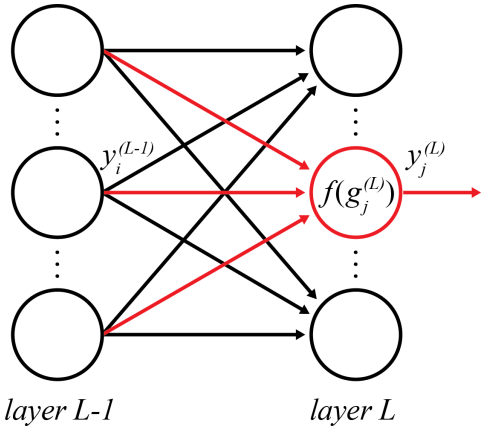


Figure 2.6: A closer look at the last layer of the MLP [5].

Depending on the objective function, the differentiation of objective function J with respect to the MLP output elements will be some kind of error term. In Figure 2.6 the j'th neuron in the MLP output layer is emphasized. If we use the SSE objective function of Equation 2.8, but for simplicity set the size of the training data to be $M = 1$, we have the following derivative for output j:

$$\frac{\partial J_{SSE}(\theta)}{\partial y_j^{(L)}} = \frac{\partial}{\partial y_j^{(L)}} \frac{1}{2} \sum_{i=1}^{N} \left(y_i^{(L)} - t_i \right)^2 = \left(y_j^{(L)} - t_j \right) \tag{2.12}$$

These terms are 'propagated' backwards in the network using the chain rule to obtain the partial derivatives in Equation 2.10. First, using the error term of Equation 2.12, the derivative of J with respect to $g_j^{(L)}$ (the input sum for node j) can be found. Assuming the activation function f in the last layer is a sigmoid, we have:

$$\frac{\partial J_{SSE}(\theta)}{\partial g_j^{(L)}} = \frac{\partial y_j^{(L)}}{\partial g_j^{(L)}} \frac{\partial J_{SSE}(\theta)}{\partial y_j^{(L)}} = y_j^{(L)} \left(1 - y_j^{(L)} \right) \frac{\partial J_{SSE}(\theta)}{\partial y_j^{(L)}} \tag{2.13}$$

This can be used to find the derivatives needed in the weight update:

$$\frac{\partial J_{SSE}(\theta)}{\partial w_{ji}^{(L)}} = \frac{\partial g_j^{(L)}}{\partial w_{ji}^{(L)}} \frac{\partial J_{SSE}(\theta)}{\partial g_j^{(L)}} \tag{2.14}$$

Using the three previous equations and also noting that $g_j^{(L)} = \sum_i w_{ji}^{(L)} y_i^{(L-1)}$ we have:

$$\frac{\partial J_{SSE}(\theta)}{\partial w_{ji}^{(L)}} = y_i^{(L-1)} \frac{\partial J_{SSE}(\theta)}{\partial g_j^{(L)}} = y_i^{(L-1)} y_j^{(L)} \left(1 - y_j^{(L)}\right) \left(y_j^{(L)} - t_j\right) \tag{2.15}$$

Using Equation 2.15 one can update all the weights of layer L using Equation 2.10.

To update the weights of the preceding layers you need error terms like the one in Equation 2.12. Since there are no target values for nodes in the hidden layers, these error terms are also acquired using the chain rule. The outputs of layer $L-1$, to the left in Figure 2.6, are the values $y_i^{(L-1)}$, meaning the error terms are:

$$\frac{\partial J_{SSE}(\theta)}{\partial y_i^{(L-1)}} = \sum_j \frac{\partial g_j^{(L)}}{\partial y_i^{(L-1)}} \frac{\partial J_{SSE}(\theta)}{\partial g_j^{(L)}} = \sum_j w_{ji}^{(L)} \frac{\partial J_{SSE}(\theta)}{\partial g_j^{(L)}} \tag{2.16}$$

This can be used with Equations 2.13 and 2.14 for layer $L-1$ to get the updates of the weights connected to this layer. This process is repeated going backwards through all the layers until all the weights of the network have been updated.

With matrix notation the gradient used in the update rule, using a single input vector, can be written as in Equation 2.17 [6].

$$\nabla_{\mathbf{W}_l} J(\theta) = \mathbf{y}^{(l-1)} \left[\mathbf{y}^{(l)} .* (1 - \mathbf{y}^{(l)}) .* \Delta\mathbf{y}^{(l)}\right]^T \tag{2.17}$$

Here $.*$ denotes element-wise multiplication and $\Delta\mathbf{y}^{(l)}$ is the error term for layer l. For the last layer we have $\Delta\mathbf{y}^{(L)} = \mathbf{y}^{(L)} - \mathbf{t}$. For other layers with sigmoid activation functions the error terms are given by Equation 2.18. The update rule for the weight matrix in layer l is the same as in Equation 2.10, only for weight matrices: $\Delta\mathbf{W}_l = -\alpha \nabla_{\mathbf{W}_l} J(\theta)$.

$$\Delta\mathbf{y}^{(l)} = \mathbf{W}_{l+1} \left[\mathbf{y}^{(l+1)} .* (1 - \mathbf{y}^{(l+1)}) .* \Delta\mathbf{y}^{(l+1)}\right] \tag{2.18}$$

In all practical cases the training set size $M \gg 1$ and to get the correct derivatives of objective functions like equations 2.8 and 2.9 a summation over the gradient term for every training vector is necessary. In practical applications many choose to update the weights using only a subset or even a single term. This is called mini-batch and on-line training respectively, as opposed to full-batch learning which uses the whole training set for every weight update.

2.4 Regularization and modifications of the update rule

To successfully train a neural network one needs a big enough model to implement the desired mapping and enough representative data for the learned mapping to perform well also for new, unseen data. If this is the case, the network is said to *generalize* well. However, if the number of parameters (weights) of the neural network is too large compared to the number of training examples, or the data is not representative, there's a risk of *overfitting* the network parameters to the training data. This means that the neural network learns a mapping that is specific to the data it has seen, leading to a large gap between performance

on training data and test data. Reducing the number of parameters in the model will make overfitting less likely, but this obviously also reduces the capacity of the neural network, which could lead to an inability to learn the desired mapping. An obvious remedy is to use a big model and simply increase the amount of training data. However, since labeled data in most cases is a limited resource, this is not always a realistic solution. Luckily there exist a number of alternative strategies to try to keep neural networks from overfitting. These methods are collectively referred to as *regularization* [12].

One way to combat overfitting to try to keep the weights of the network from growing too big. This is essentially making the learning prefer "smoother" functions, which can be compared to using a simpler model (less parameters) [9]. This can be accomplished in several ways, but among the most usual are penalizing the norm of the parameters. An example is the much used L_2 regularization which adds a parameter times the sum of squared weight to the objective function. Intuitively, if this parameter has a big value, the only way to minimize the objective function is to make sure the sum of squared weights is a small number. Adding this term to the objective is equivalent [9] to subtracting a parameter times the weight matrix from the weight matrix update term:

$$\Delta \mathbf{W}_l' = -\alpha \nabla_{\mathbf{W}_l} J(\theta) - \alpha\kappa \mathbf{W}_l \tag{2.19}$$

Writing out the expression to be assigned to the weight matrix it's easily seen that the old parameter values are scaled by a number smaller than 1 for every update (Eq. 2.21). For this reason L_2 regularization is also referred to as *weight decay*.

$$\mathbf{W}_l \leftarrow \mathbf{W}_l + \Delta \mathbf{W}_l' \tag{2.20}$$

$$\mathbf{W}_l \leftarrow (1 - \alpha\kappa)\mathbf{W}_l - \alpha \nabla_{\mathbf{W}_l} J(\theta) \tag{2.21}$$

Weight decay makes sure weights with small or zero gradient terms are slowly scaled toward zero, essentially reducing unnecessary parameters and thus simplifying the model.

For weights with nonzero gradient terms it's clear that the modified weight update $\Delta \mathbf{W}_l'$ will be zero before the minimum of the original objective function is reached (and the gradient is zero). In that sense there exists a certain trade-off in choosing the value of κ, between penalizing large weights and reaching a minimum of the (original) objective function. Regularization can therefore be seen as methods that seek to improve performance on unseen test data at the expense of (possibly) worsening performance on the training data.

A form of regularization similar to weight decay penalizes the sum of absolute values of the weights and is therefore called L_1 regularization. According to [12] this type of regularization typically leads to solutions with increased sparsity, meaning it may cause several of the weight matrix elements to have an optimal value of zero. This might be preferred for some types of problems.

Instead of adding a penalty term to the cost function it is possible to implement explicit constraints of the parameters of the neural network. An example of this is constraining the norm of each column of each weight matrix to be less than or equal to some value c. If the norm exceeds this value, all elements in the column vector are scaled by the same value, projecting the vector onto a N-dimensional sphere with radius c [12], N being the number of elements in the column. Since the elements in a column are the incoming weights

to a single node in the next layer this type of regularization keeps the individual nodes from having too large incoming weights. This seems reasonable, especially when using saturating activation functions like sigmoid and the hyperbolic tangent. In [14] this type of regularization is chosen because it allows the use of a large learning rate at the start of training without big parameter updates causing the weights to "blow up". This regularization technique is called *max-norm* in [15], where it is recommended in combination with *dropout*, which is described in Section 2.5.

There exists other modifications to the weight matrix update rule that can improve learning and generalization. A popular method called *momentum* is designed to speed up learning by stabilizing the trajectory of the weight matrix update [9]. Instead of using the gradient terms directly in the weight update, momentum accumulates the gradients in a velocity matrix, defined in Equation 2.22.

$$\mathbf{V}_l[n] = \omega \mathbf{V}_l[n-1] - \alpha \nabla_{\mathbf{W}_l} J(\theta[n-1]) \tag{2.22}$$

Here the velocity of the *n*'th epoch is set equal to a parameter ω times the previous velocity, minus the learning rate times the gradient of the objective using the parameters after the previous update $\theta[n-1]$. Then the weight update term of the *n*'th epoch is set equal to the velocity at time *n*: $\Delta \mathbf{W}_l[n] = \mathbf{V}_l[n]$. The momentum parameter ω is chosen to take values the range $[0, 1)$, meaning the velocity term will contain a sum of exponentially decayed past gradient terms.

A slightly different implementation of momentum that was used in the thesis work is presented in equations 2.23 and 2.23. Here the the gradient term is scaled with $1 - \omega$ and added to the previous velocity to form the new one. The parameters are updated by subtracting the velocity times the learning rate. The only change is the additional scaling of the gradient, which in practice leads to using a lower learning rate $\alpha' = \alpha(1 - \omega)$.

$$\mathbf{V}_l[n] = \omega \mathbf{V}_l[n-1] + (1-\omega) \nabla_{\mathbf{W}_l} J(\theta[n-1]) \tag{2.23}$$

$$\mathbf{W}_l[n] = -\alpha \mathbf{V}_l[n] \tag{2.24}$$

2.5 Dropout

A simple and effective way to improve the performance of neural networks on unseen data is to train many different networks and average the predictions they make for a given input. According to [12] this kind of *model averaging* is a very powerful and reliable method for improving generalization. To train many different big neural networks to convergence can however be very computationally expensive.

Dropout is a powerful way to prevent large neural network architectures from overfitting that can be thought of as an approximation of model averaging using a big number of different neural networks [15]. Dropout works by randomly dropping nodes and their connections during the training of a neural network. This is equivalent to sampling among an exponential set of smaller sub-networks of the bigger architecture, as illustrated in Figure 2.7. A new sub-network is sampled for every weight update, which means that every unique model might just be trained a few times, if at all, during the training procedure. However, since many of the smaller models will contain the same connections, all the

weight of the original neural network will still – with high probability – be updated many times throughout the training.

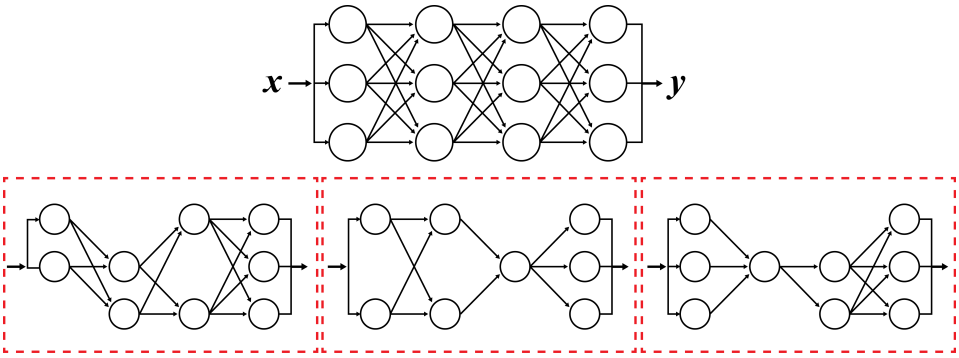


Figure 2.7: Three possible sub-nets, sampled from the original neural network by dropping nodes with a certain probability

Dropout can be implemented by multiplying a binary mask with the activations, or outputs, of every layer except the final one. The elements of the mask vector are typically Bernoulli random variables that take the value 1 with probability p [15]. The inverse probability, $1 - p$, will be referred to as the dropout rate of the nodes. According to [12] it is normal to use a value of 0.5 for p for nodes in the hidden layers and 0.8 for nodes in the input layer. The bias-nodes of the neural network are not dropped.

After training with dropout, one does not actually average the predictions of all the sampled networks. Instead the complete architecture containing all the nodes is used, with weights going out from a node scaled by multiplying them with the probability, *p*, of keeping that node during training [15].

Dropout is thought to reduce overfitting by preventing the neurons forming complex co-adaptions that tend to fit the training data well, but are less likely to generalize to unseen examples [14]. Since a different set of nodes will be present for every weight update, the neurons will be forced toward discovering useful features of their inputs on their own, without relying too much on the other nodes of the network. Removing nodes will however reduce the capacity of the neural network, something that can be remedied by increasing the size of the model [12].

Dropout as it has been described here works by multiplying the node outputs in a layer with independent Bernoulli random variables. Even though this method can be understood as sampling and training sub-networks to approximate model averaging, it was found in [15] that other random variables than Bernoulli can work as well. Especially random variables drawn from a Gaussian distribution with unit mean and variance was found to work *"just as well, or perhaps better"* than Bernoulli variables, even though the nodes are no longer *dropped* from the network in the same way. Making the variance $\sigma^2 = \frac{1-p}{p}$ gives the capability of scaling the "dropout rate" similar to the Bernoulli method. Since the expected values of the Gaussian variables are 1, unlike the Bernoulli variables, it is not necessary to scale the weights by *p* after training.

Dropout is a type of regularization seemingly very different from the previous methods

described. Dropout can be combined with these other regularization techniques to further improve generalization. As mentioned, the max-norm technique described in the previous chapter was found to be especially well suited for dropout in [15].

2.6 Restricted Boltzmann Machines

Restricted Boltzmann machines (RBM) are a special class of energy-based probabilistic models called Boltzmann machines. They are graphical models with hidden and visible nodes, much like the neural networks previously described. They are described as energy-based because the configuration of the nodes in the model has an associated energy. The probability of this configuration is set proportional to the exponential of the negative energy (Eq. 2.26) [16]. RBMs are restricted in that the hidden and visible nodes are organized in a bipartite graph structure with connections only between pairs of hidden and visible nodes. This can be illustrated by placing the visible nodes in one layer and the hidden nodes in another and only having connections going from one layer to the other, as shown in Figure 2.8. RBMs are trained with unlabeled data and can be used as building blocks for deep architectures [17]. Using the weights and biases of stacked RBMs as initial values for the parameters of deep neural networks led to breakthrough that caused a revival of interest in neural networks in 2006. This is described further in the next section.

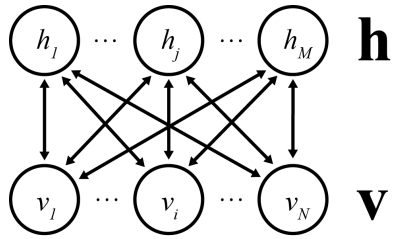


Figure 2.8: Graphical model of a Restricted Boltzmann machine. Values of the visible and hidden nodes are contained in the vectors $\mathbf{v}$ and $\mathbf{h}$ respectively. [5]

In the standard RBM all nodes are stochastic, binary units and they are therefore different from the neurons in Section 2.2. Another difference from the neural networks described earlier is the fact that the connections are bidirectional, so values can be communicated both ways between a pair of connected nodes. The connections still have a weight associated with them, which is symmetric, meaning the connection has the same weight going from the hidden to visible node as going from the visible to hidden node. Every node has an additional bias term, though this is not illustrated in Figure 2.8.

The energy of a configuration $(\mathbf{h}, \mathbf{v})$ is given by the expression in Equation 2.25 [18]. Here b_j is the bias of hidden node *j* and c_i the bias of visible node *i*. The factor w_{ij} is the weight of the connection between hidden node *j* and visible node *i*.

$$Energy(\mathbf{h}, \mathbf{v}) = -\sum_{i \in visible} c_i v_i - \sum_{j \in hidden} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{2.25}$$

Equation 2.26 gives the probability of a configuration $(\mathbf{h}, \mathbf{v})$ for the visible and hidden

variables of the RBM. The denominator Z is a normalizing constant, often referred to as partition function, defined in 2.27. This is simply a sum over all possible configurations of the nodes, necessary for the probabilities to sum to 1. θ in these equations is the set of parameters of the RBM, containing the weight matrix and biases of both layers.

$$P(\mathbf{h}, \mathbf{v}; \theta) = \frac{e^{-Energy(\mathbf{h},\mathbf{v})}}{Z} \tag{2.26}$$

$$Z = \sum_{\mathbf{v}} \sum_{\mathbf{h}} e^{-Energy(\mathbf{h},\mathbf{v})} \tag{2.27}$$

The exponent in 2.26 is the negative of the energy of the configuration, defined in Equation 2.25. It's clear that in order to have high probability, a configuration $(\mathbf{h}, \mathbf{v})$ of the nodes should have low energy. To get the probability of the data in the visible nodes, a sum over all possible vectors $\mathbf{h}$ is necessary:

$$P(\mathbf{v}; \theta) = \sum_{\mathbf{h}} \frac{e^{-Energy(\mathbf{h},\mathbf{v})}}{Z} \tag{2.28}$$

Inspired by physics [16], it is normal to define the *free energy* of the RBM to be:

$$FreeEnergy(\mathbf{v}; \theta) = -\log\left(\sum_{\mathbf{h}} e^{-Energy(\mathbf{h},\mathbf{v})}\right) \tag{2.29}$$

Equation 2.28 then can be written on the same form as 2.26:

$$P(\mathbf{v}; \theta) = \frac{e^{-FreeEnergy(\mathbf{v};\theta)}}{Z} \tag{2.30}$$

Since the RBM has no direct connections between pairs of hidden nodes or pairs of visible nodes, the variables in one layer are independent when conditioned on the state of the other layer [16]. For the hidden layer this can be written as in Equation 2.31.

$$P(\mathbf{h}|\mathbf{v}; \theta) = \prod_{j \in hidden} P(h_j|\mathbf{v}; \theta) \tag{2.31}$$

Here the value of the *j*'th node in the hidden layer, h_j, is a binary random variable that takes the value 1 with the probability in Equation 2.32. The function σ is the sigmoid, defined in Equation 2.3. An almost identical expression in Equation 2.33 gives the probability of a visible unit being equal to 1, given $\mathbf{h}$.

$$P(h_j = 1|\mathbf{v}; \theta) = \sigma\left(\sum_i v_i w_{ij} + b_i\right) \tag{2.32}$$

$$P(v_i = 1|\mathbf{h}; \theta) = \sigma\left(\sum_j w_{ij} h_j + c_i\right) \tag{2.33}$$

The goal of RBM training is to update the parameters (weights and biases) using a set of unlabeled training data, so that the model can generate this data with high probability. This is done by lowering the energy of the node configurations ($\mathbf{h}$,$\mathbf{v}$) with the training data at the visible nodes, and increasing the energy of other (unseen) configurations [18]. This has the effect of increasing the numerator in Equation 2.28 for values of $\mathbf{v}$ in the training set, and reducing terms in the partition function (Eq. 2.27) for other configurations. The result is a higher probability for the data vector $\mathbf{v}$, or rather a higher likelihood of the RBM parameters θ for the training data. Essentially this is describing maximum likelihood learning, which aims to maximize Equation 2.28 for training data $\{\mathbf{v}\}$ by updating the parameters θ. To maximize the likelihood, one needs the gradient with respect to the parameters. It turns out that the derivative of the log-likelihood with respect to the weights is the relatively simple expression in Equation 2.34 [18]. The E denotes expectation over the distribution specified in the subscript.

$$\frac{\partial \, log\,(P(\mathbf{v};\theta))}{\partial w_{ij}} = E_{data}\{v_i h_j\} - E_{model}\{v_i h_j\} \tag{2.34}$$

Unfortunately, getting numerical values for 2.34 takes many iterations of alternating Gibbs sampling, which is computationally inefficient in practical applications [18].

A learning procedure called *Contrastive Divergence* (CD_1 for short) using a single Gibbs sampling step has been shown to work well for training RBMs. This is an computationally efficient, although not very accurate, approximation of the log-likelihood gradient described in 2.34. As illustrated in Figure 2.9, it starts with a vector from the training set in the visible layer and then updates the hidden nodes by sampling from a Bernoulli distribution with probability in 2.32. Having set the values of the hidden layer, the visible nodes are sampled in the same fashion, creating a 'reconstruction' of the visible data. Finally, a second update of the hidden units (given the reconstruction of the visible nodes) is performed. Note that to avoid unnecessary variance in the update, the hidden values are not actually sampled during this last step, but set equal to their probabilities of having the value 1 (Eq. 2.32) [18]. The update rules are given in equations 2.35 – 2.37 [18]. The subscript denotes node number and the superscript the update number in the sampling chain, as in Figure 2.9. α is again a learning rate or step factor.

$$w_{ij} = w_{ij} + \alpha(v_i^{(1)} h_j^{(1)} - v_i^{(2)} h_j^{(2)}) \tag{2.35}$$

$$b_j = b_j + \alpha(h_j^{(1)} - h_j^{(2)}) \tag{2.36}$$

$$c_i = c_i + \alpha(v_i^{(1)} - v_i^{(2)}) \tag{2.37}$$

In many cases, doing more than one step of Gibbs sampling can improve the quality of the RBM update. In general, the CD_k algorithm performs k steps of conditionally updating the visible and hidden states of the RBM, and uses statistics from the final update as the second term in the equations 2.35 – 2.37 [18].

For continuous input data, using binary visible nodes is not the best solution. An alternative to the binary RBM, called Gaussian-Bernoulli RBM (GB-RBM) uses binary hidden nodes, but real-valued, gaussian visible nodes. This means the hidden nodes are still updated by sampling from a Bernoulli distribution with probability as in Equation

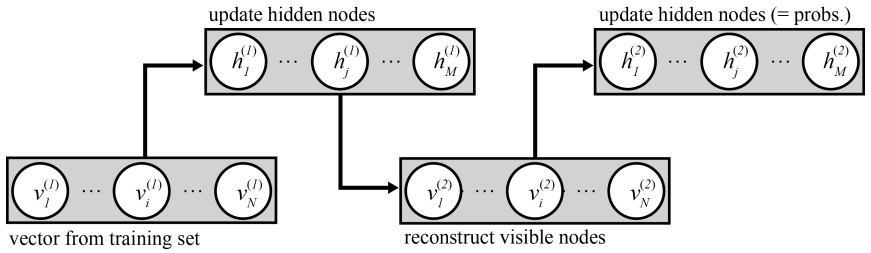


Figure 2.9: Illustration of sampling steps in Contrastive Divergence algorithm CD_1. [5]

2.32, but that the visible nodes are updated by sampling from a gaussion distribution. If the input features are normalized to have unit variance, the distribution of the visible nodes is written in Equation 2.38 [19]. This is equivalent to having linear visible nodes with added gaussian noise: $v_i = \left(c_i + \sum_{j=1}^{H} w_{ij} h_j\right) + n$, where $n \sim \mathcal{N}(0, 1)$. In [18], however, it is written that in practice it is easier to use noise-free reconstructions when sampling the visible units.

$$P(v_i|\mathbf{h};\theta) = \mathcal{N}\left(c_i + \sum_j w_{ij} h_j, 1\right) \tag{2.38}$$

The learning rate should be smaller for the GB-RBM, as the real-valued visible units are not bounded in size like the binary units are [18]. Aside from these differences, the learning procedure for GB-RBM is the same as for the binary (Bernoulli-Bernoulli) RBM, although the energy function is slightly different from 2.25.

Just like for neural networks, it is possible for RBMs to overfit the training data. The free energy from Equation 2.29 can be used to monitor the overfitting of the model. The procedure calls for calculating the ratio of the probability the RBM assigns to training data $\mathbf{v}_{train}$ over that which it assigns validation data $\mathbf{v}_{valid}$. Using the ratio of the probabilities causes the partition function Z to cancel out:

$$\frac{P(\mathbf{v}_{train};\theta)}{P(\mathbf{v}_{valid};\theta)} = \frac{e^{-FreeEnergy(\mathbf{v}_{train};\theta)}}{e^{-FreeEnergy(\mathbf{v}_{valid};\theta)}} \tag{2.39}$$

Taking the logarithm of this ratio reduces the expression to the difference in free energy for the two datasets:

$$\log\left(\frac{P(\mathbf{v}_{train};\theta)}{P(\mathbf{v}_{valid};\theta)}\right) = FreeEnergy(\mathbf{v}_{valid};\theta) - FreeEnergy(\mathbf{v}_{train};\theta) \tag{2.40}$$

[18] recommends finding the free energy of a representative subset of the training data and a validation set every few epochs during the training. If, after a number of epochs have passed, the difference starts to increase, this might indicate that the likelihood of the RBM is improving for the training data, but not the validation data.

When doing iterative gradient-based training we generally want to monitor the training objective as the learning is progressing. For RBM the objective we try to maximize,

the likelihood in Equation 2.28, is difficult or intractable to calculate because of the many terms in the partition function. In lack of a better measure it is normal to monitor the reconstruction error when training RBMs [18], and see whether this is being reduced. Steadily increasing reconstruction cost is a bad sign, but even when this measure is decreasing it is hard to know if the likelihood is being increased as well. In fact, in [20] it was observed that the likelihood might be steadily decreasing even when the reconstruction error is improving. First of all, this reinforces that the reconstruction error is a bad measure of how the learning is progressing. Secondly, it shows that the use of an approximate learning algorithm, like CD_k, might diverge without it being detected.

2.7 Deep Learning

The term deep learning is often used to describe machine learning approaches using deep architectures, meaning models "*... composed of multiple levels of non-linear operations*" [16]. Within the domain of neural networks one example of such architectures are multilayer perceptrons having two or more hidden layers. These are often referred to as deep neural networks (DNN). Deep architectures can provide multiple levels of hierarchical abstraction of the input data, something that is believed to be a central part of how the human brain works and therefore necessary for many artificial intelligence tasks [16]. One can think about the hidden layers of a DNN as learning a representation of the data that makes the action of the last layer (typically a linear classifier or regression layer) easier [12].

It can be proven that an MLP with one hidden layer is an universal approximator, meaning it can theoretically approximate any (compact) function with error arbitrarily close to zero [9]. There is, however, no guarantee that one can learn the proper solution, or that it will be computationally efficient. Indeed it might take an enormous amount of nodes in the hidden layer to do so. In general, deep architectures like DNNs are much more efficient for realizing highly varying functions [16].

For many years, DNNs were considered very difficult to train by the backpropagation algorithm described earlier. A commonly held belief was that the gradient based trained found local minima with high cost compared to the global minimum. According to [12], many experts in the field no longer consider local minima a big problem: *"...experts now suspect that, for sufficiently large neural networks, most local minima have a low cost fuction value..."*. It is, however, stated that this is still actively being researched. Another problem for DNNs using sigmoid or similar neurons with weights initialized to small random values, is that the error terms tend to become vanishingly small as they are backpropagated through the architecture. The result is poor adaption of the first layers of a DNN. This leads to *under-fitting*, meaning the model is unable to learn the mapping implied by the labeled data. As opposed to the over-fitting described earlier, where the DNN performed well for the training data, but not for test data, under-fitting means the network performs bad for all data.

In 2006 a new approach, using an unsupervised *pretraining* procedure, stacking several individually trained RBMs to create a deep architecture, was shown to be successful for training models called Deep Belief Networks (DBN) [21]. In [17] the same procedure was used successfully for initializing weight matrices in a feed-forward network, followed by *fine-tuning* using labeled data and the backpropagation algorithm. The procedure, which

has been used in this master thesis, can be described in the following steps:

1. Having the size of the final DNN, initialize a restricted Boltzmann machine for every layer of connections, except the last one going to the output layer.
2. Using unlabeled data, train the RBM for the first layer using contrastive divergence as described in 2.6
3. Use the activations of the hidden nodes, when driving the visible nodes of the trained RBM with training data, as input data for the next level RBM
4. Train all RBMs in the same manner, using data from the hidden layer of RBM at the previous level as input
5. Unfold the stacked RBMs to a DNN by copying the pre-trained weight matrices to the MLP structure
6. Fine-tune the complete network using backpropagation with labeled data

Why this method of greedy, layer-wise pretraining works is not fully understood. Experiment published in [22] suggest that pre-training acts as a form of regularization, by "*...minimizing variance and introducing bias towards configurations of the parameter space that are useful for unsupervised learning*", but other plausible explanations also exist. Another potential benefit of pre-training is that the learning is unsupervised, allowing use of unlabeled data in the network training procedure.

While RBMs were the first models used in this pretraining technique, others have since been used successfully in much the same manner. One notable example is the *denoising autoencoder* [23]. Explained in simple terms, an autoencoder is a feedforward neural network that is trained to reconstruct its input. For pretraining, *shallow* autoencoders with a single hidden layer are used. By using the input data as target for the output, this type of neural network is said to *encode* the input going to the hidden layer, and then *decode* it by reconstructing it as close to the original as possible. An optional constraint often used with autoencoders is to *tie* the two weight matrices [23]. This means constraining the weight matrix going from hidden layer to output of the autoencoder to be equal to the transpose of the weight matrix going from the input to the hidden layer. This sounds very similar to the RBM, which also reconstructs the input data in some sense by using the transpose of the weight matrix going from visible to hidden units. An autoencoder, however, is trained to do make this reconstruction as good as possible, measured by some objective like MSE or cross-entropy. Also, unlike the RBM, the autoencoder does not perform sampling in the hidden units.

The denoising autoencoder is an autoencoder that stochastically corrupts the input, but keeps the original input as target [23]. It can be compared to using dropout, but only for the input layer. This corruption, or added noise, is useful for keeping the autoencoder from learning to simply copy the data from one layer to the other, or similar trivial solutions. Rather than this, we want it to learn a representation of the data in the hidden layer that, when using it as a building block for deep network, makes subsequent learning easier [12]. Denoising autoencoders have been found to work as well as, or even better than, RBMs for greedy layer-wise pretraining [23]. Considering that RBMs are less intuitive and more difficult to train also make denoising autoencoders an attractive alternative for pretraining.

Since the breakthrough with greedy layer-wise pretraining in 2006 it has been shown that it is possible to successfully train DNNs from random initializations. One example came in 2010, when a method of *Hessian-free optimization* was shown to give superior results to the pretraining method presented in [17] on the same problems [24]. In [25] the same level of performance was achieved, without pretraining, using gradient descent from a more carefully chosen random initialization with a gradually increasing momentum value. In the same paper it is suggested that, with the right type of parameter initialization, training deep neural networks is much simpler than previously believed. Other papers report that using rectified linear units instead of sigmoid units result in DNNs that *"can reach their best performance without requiring any unsupervised pre-training"* [11]. In cases with little labeled data, but large amounts of unlabeled data, pretraining can still benefit these networks as well.

Some other recent techniques that might be worth mentioning include *residual learning* and *batch-normalization*. Both techniques are reported to be able to greatly improve the performance of deep neural networks. Batch-normalization deals with the problem that the input distribution to the individual layers of a DNN will change as the parameters of the preceding layers are updated. This means the parameters of a layer will need to adapt to a continuously evolving distributions during learning [26]. This is a problem that is more severe for deeper networks. The method first uses mean and standard deviation estimates to normalize the mini-batch activations for every layer. The normalized activations are then shifted and scaled by two learnable parameters per activation, that introduces the possibility of undoing the preceding normalization (in case that turns out to be the optimal solution). The reported results state that batch-normalization *"dramatically accelerates the training of deep neural nets"* [26].

Residual learning is a method of adding new layers to a model without risking higher training error as a result (under-fitting). It introduces a "shortcut" around stacks of layers, adding the input of those layers to their output. This means the layers are not learning the same mapping from input to output, but instead the residual of the mapping. Using this method for image recognition, a very deep neural network with 152 layers was successfully trained and used to win an image classification competition, as described in [27].

2.8 OM-LSA

OM-LSA as it is used in this text refers to a speech enhancement method combining a *optimally-modified log-spectral amplitude* (OM-LSA) speech estimator with a *minima controlled recursive averaging* (MCRA) noise estimator as described in [4]. The method assumes that the corrupted speech takes the form $y(n) = x(n) + d(n)$, where $x(n)$ is clean speech and $d(n)$ is uncorrelated noise. The algorithm operates on frames of speech using the short-time Fourier transform (STFT) in Equation 2.41.

$$Y(k,l) = \sum_{n=0}^{N-1} y(n + lM)h(n)e^{-j(2\pi/N)nk} \tag{2.41}$$

Here k is the index of the frequency bin, l is the frame index and h is a window function. The clean speech frames are estimated by calculating a gain $G(k,l)$ for every frame and

frequency bin, which is multiplied with the input:

$$\hat{X}(k,l) = G(k,l)Y(k,l) \tag{2.42}$$

The log-spectral amplitude estimator is defined to minimize the mean square error:

$$E\{(\log A(k,l) - \log \hat{A}(k,l))^2\} \tag{2.43}$$

where $A(k,l) = |X(k,l)|$, under the assumption of Gaussian distributed STFT coefficients [4]. Operating with a binary hypothesis model for whether speech is present or absent, the gain is given by:

$$G(k,l) = \{G_{H_1}(k,l)\}^{p(k,l)} G_{min}^{1-p(k,l)} \tag{2.44}$$

where G_{H_1} is the conditional gain for when speech is present and G_{min} is a lower bound for the gain for when speech is absent, chosen subjectively for *"noise naturalness"* [4]. $p(k,l)$ is the speech presence probability, conditioned on $Y(k,l)$. The expression for G_{H_1} is not given here (see [4]), but it uses a noise power estimate based on the MCRA estimator mentioned initially. An improved version (IMCRA) is described in [28].

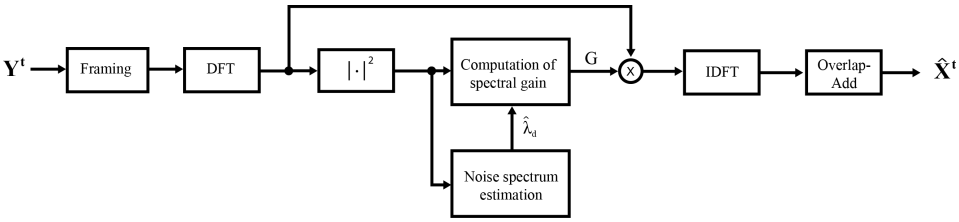


Figure 2.10: Block diagram of OM-LSA. Taken from [4]

A block diagram of the OM-LSA method is illustrated in 2.10. Note that the method only modifies the magnitude of the noisy speech spectrum, meaning the gain $G(k,l)$ is real and positive.

In this master thesis an implementation of OM-LSA found in [29] was used for comparison with the DNN based system. This is appropriate as both techniques attempt to minimize essentially the same error, namely the mean square error of the log-magnitude spectrum of the frames. In [1] an improved version of OM-LSA called logMMSE was also used for comparison with the DNN system.

2.9 Measures of speech quality

When designing a speech enhancement system the objective generally is to improve the quality of speech signals as experienced by the users of the system. One way to measure speech quality is to use listening tests and collect mean opinion scores (MOS), but this can be very costly and time-consuming. It is also a very impractical method during development, as one might have many small variations of the system to compare in many stages

of the process. Therefore there is a great need for an objective, calculable metric that is highly correlated with subjective experience of quality.

Perceptual evaluation of speech quality (PESQ) is an ITU-T standard method for speech quality assessment of speech codecs and telephone networks [30]. It compares clean input speech with (possibly) distorted output speech and gives a score ranging from -0.5 to 4.5 that is reported to have high correlation with MOS for certain application. The method involves time-alignment of the signals, followed by a psychoacoustic model using a modified Bark scale that is equalized and mapped to (Sone) loudness.These transformed signals are used to form two disturbance measures, one symmetric and one asymmetric, that are fed to a cognitive model that outputs the method's MOS estimate [3]. This model is the simple linear combination in Equation 2.45, which is the result of linear regression on data from 30 subjective tests[3].

$$PESQMOS = 4.5 - 0.1d_{SYM} - 0.0309d_{ASYM} \tag{2.45}$$

PESQ is the main measure used in [1] and [31] to evaluate the performance of the speech enhancement systems, and for that reason it was also used in this work.

As briefly mentioned, the objective function that the DNNs in [1] and [31], and also in this work, are trained to minimize is the mean square error of clean and noisy speech in the log-magnitude spectral domain. Minimizing this objective is shown in [32] to have some consistency with the human auditory system. Since it is pretty straight-forward to compute, the MSE in the log-spectral domain might also be a useful measure of speech quality, relative to the clean speech. Although it is considered less correlated with subjectively experienced quality than PESQ, it is included also in the testing of the DNN systems to get an idea of the quality of the enhancement.

The segmental SNR measure from [33], which is defined in Equation 2.46, was used as an additional objective measure of speech quality for some of the experiments. This, again, is not expected to give as good an idea of the subjective quality as PESQ, but it is useful for measuring the degree of noise reduction. [1]

$$SNR_{seg} = \frac{10}{M}\sum_{m=0}^{M-1}\log_{10}\frac{\sum_{i=Nm}^{Nm+N-1}x(i)^2}{\sum_{i=Nm}^{Nm+N-1}(x(i)-\hat{x}(i))^2} \tag{2.46}$$

Here $x(i)$ are the samples of clean speech while $\hat{x}(i)$ were samples of enhanced or noisy speech, depending on whether the input or output segmental SNR was being calculated.

Chapter 3

Data and software

3.1 Overview

This chapter contains a short description of the database used for training, how noisy data was synthesized and what software tools and resources were used in the project.

3.2 TIMIT

The speech data used for training and testing the deep neural networks was extracted from the TIMIT corpus. TIMIT is an American English speech database containing in total 6300 sentences, read by 630 different speakers. Of these sentences there are 2342 unique texts organized in such a way that 2 of them are read by all 630 speakers, 450 are read by 7 speakers each and the remaining 1890 are only read by 1 person per sentence [34]. Every read sentence is stored as an individual audio file, organized in two different folders; one for training and one for testing. All the audio files are labeled with their phonetic content for use in speech technology applications like automatic speech recognition

The files contained in the TIMIT training folder was split into two sets, 3465 files to be used for training and 1155 to be used for validation. Another 1344 files from the test folder were reserved for testing. A list of all the file names for each of the three sets were constructed and shuffled to not have all sentences from one speaker in direct succession. When loading data for training or testing, file names were read from these list until as many frames as specified by the user were obtained.

The audio files in the TIMIT database are sampled at $16kHz$ and were decimated to $8kHz$ before being used in the project.

3.3 Aurora 2

Aurora 2 is a framework and a database designed for evaluation of speech recognition systems and algorithms under noisy conditions [2]. Among other things it contains a number

of audio clips of natural background noise, which were used in this work. There are 6 types of noise: *babble*, *restaurant*, *street*, *airport*, *subway*, *exhibition* and *car*. Spectrograms of these noises can be seen in figure 3.1. The duration of the different noise signals, rounded to nearest whole second, are also given in the figure. All noise recordings have sampling frequency $8kHz$.

3.4 Synthesizing noisy speech

The noisy speech that was used as input to the speech enhancement system was synthesized by adding different types of noise to the decimated TIMIT-files, scaled to different SNR levels. This operation and the following feature extraction was performed in MATLAB. The typical SNR levels used were 0, 5, 10 and 15 dB. When scaling the noise, a global average of the power of all TIMIT training files was used as the signal power. This, together with the target SNR value, was used to calculate the wanted noise power level. Since this global average was used, the actual SNR of the individual files would vary around the target SNR value, providing a range of different file-based SNR levels. For the test set this is discussed further in 5.2.

For most of the training, the 4 noise types *AWGN*, *babble*, *restaurant* and *street* were used for training, in some cases including *airport* and *subway*. The types *car* and *exhibition* were reserved for testing.

For every new SNR level for a speech-noise combination, a new noise-signal was sampled. For additive white Gaussian noise *(AWGN)* this was done using the *randn()* function in MATLAB. For the noise signals from the Aurora 2 database a segment of length equaling that of the speech signal was selected by randomly choosing a start index for the segment that fulfilled the condition that start index + segment length would not extend beyond the end of the noise signal. Studying the lengths of the noise signals in figure 3.1 it is clear that noise sampled from the shorter signals like *subway*, *exhibition* or *car* would have a bigger probability of reusing the same noise samples multiple times. Because either the speech or the SNR will be different for every time a noise segment is sampled, this is not considered a big problem. One can argue that since the TIMIT-files contains multiple speakers reading the same sentence, there is a very small probability that the exact same noise signal could be combined twice with the same sentence, spoken by a new person. Since these speaker will have different voice characteristics, dialects and/or speak with different tempo, even this extreme case can probably be considered to provide two unique signals.

3.5 Theano

Theano [35] is a Python library originating from the Machine Learning Laboratory (MILA) at University of Montreal. It allows users to define symbolic mathematical expressions and compile them efficiently using either CPU or GPU. The expressions are stored as graphs, which allows Theano to simplify and optimize calculations and, among other things, easily find the gradient of expressions with respect to the specified variables. Theano is a very popular choice for machine learning applications, and is the foundation for other machine

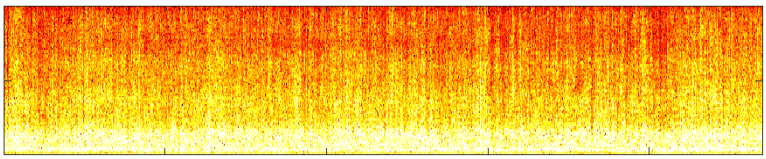
learning solutions that specialize in neural networks, like Keras [36], Lasagne [37] and Blocks [38]. Theano, running on an nVidia GPU, was used extensively for training the neural networks in the master thesis project work.

3.6 MATLAB and the DeepLearn Toolbox

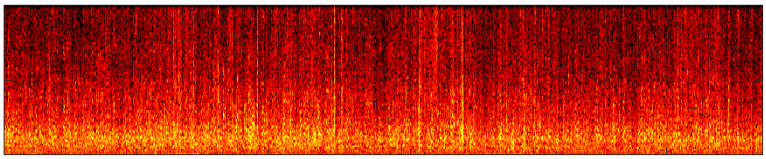
Although Python and Theano was used for training the deep neural networks, MATLAB was used for synthesizing the noisy speech, as mentioned previously, for extracting features, and for testing the trained DNNs. The feature-extraction was only performed once, storing the features of noisy and clean speech in binary files to be loaded in Python before training. The testing of the trained models used an open source collection of scripts and functions called the DeepLearn Toolbox [39]. This toolbox was also used in the project leading up to the master thesis, but it is no longer maintained. That this toolbox might no longer be up to date was not a problem for the purpose of testing, since only simple functionality, like computing the feedforward pass, was needed.

3.7 Online code resources

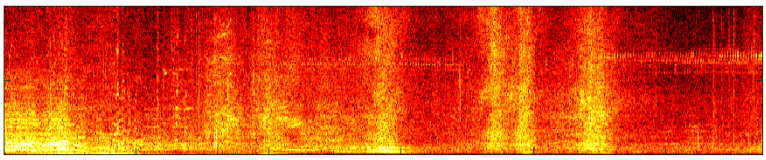
A lot of freely available Theano code found online was used as basis or inspiration for the code written in this master thesis. The Deep Learning tutorial from [40] formed the main basis for the project although some code from the Lasagne documentation [37] was also used. The implementation of dropout (see section 2.5) was based on code found on [41]. A program implementing the PESQ measure was compiled from source code found on [42].



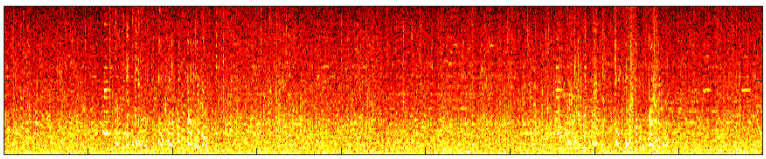
(a) Babble, length 235 sec



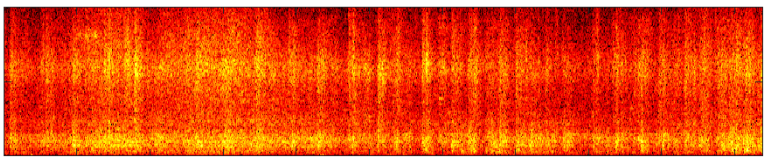
(b) Restaurant, length 284 sec



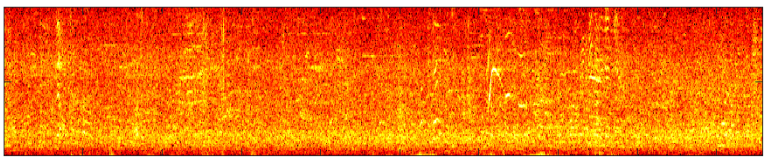
(c) Street, length 54 sec



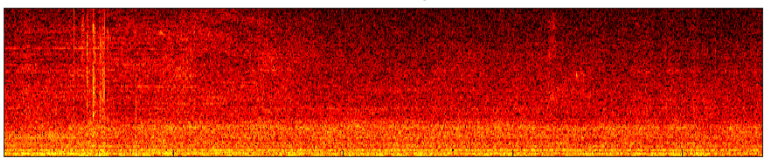
(d) Airport, length 180 sec



(e) Subway, length 21 sec



(f) Exhibition, length 19 sec



(g) Car, length 22 sec

Figure 3.1: Spectrograms of the noises in the Aurora 2 database

Chapter 4

Implementation and experiments

4.1 Overview

This chapter starts with a high level description of the speech enhancement system and an explanation of how the DNN at the core of the system was trained. Following this, a description of techniques from [1] that were implemented to improve the system are given. Investigating changes beyond what was done in [1], an alternative DNN based on rectified linear units is discussed, as well as a hybrid system combining the DNN approach with the OM-LSA method. At the end of the chapter the testing procedures are described.

4.2 Speech enhancement system

The speech enhancement system was based on [1] and is illustrated in Figure 4.1. Enhancement is performed by a deep neural network on the log-magnitude spectrum of windowed frames from the input signal. The system is designed for speech signals sampled at $F_s = 8kHz$.

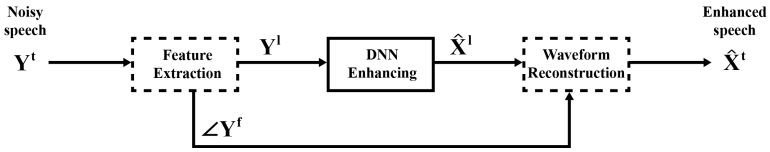


Figure 4.1: Speech enhancement system [5]. Based on [1]

The feature extraction before the DNN is depicted in Figure 4.2. The input signal, Y^t, in the discrete time domain, is divided into frames using a Hamming window of length 32 msec. Every new frame is extracted by shifting the window by half this value, 16 msec. The discrete Fourier transform (DFT) is used to get the samples of the frames' frequency spectrum. These two steps are essentially an implementation of the short-time

Fourier transform briefly mentioned in Section 2.8, Equation 2.41. The frame spectra are separated into their magnitude and phase components. The enhancement mapping is only performed for the magnitude, and so the noisy phase, $\angle Y^f$, is used directly in the reconstruction of the frames. Because of symmetry, only the one-sided magnitude spectrum is sent to the DNN, which for the chosen window length and sampling frequency results in 129 frequency samples. The natural logarithm is applied to the magnitude components which then are scaled to have zero mean and unit variance, using mean and standard deviation estimates found for the training data. To improve the performance of the system some acoustic context is provided with the frame to be enhanced. This involves combining the scaled log-magnitude spectrum of the current frame with some preceding and following frames. For simplicity, these are referred to as the left and right context frames respectively. The DNN performs a mapping from the input features, Y^l, (containing the log-magnitude components of several frames of noisy speech) to the log-magnitude components of a single enhanced frame, $\hat{X}^l$.

The waveform reconstruction from output vectors of the DNN is depicted in Figure 4.3. The output features from the DNN, $\hat{X}^l$, are multiplied with an optional Global Variance equalization factor and scaled back using the same mean and standard deviation estimates as in the feature extraction. The inverse of the natural logarithm is applied and the one-sided magnitude spectrum estimate is mirrored and combined with the noisy phase to construct the complete linear frequency spectrum of the enhanced frame. The inverse discrete fourier transform (IDFT) is then applied and the frames are combined with overlap-add to form the complete enhanced waveform. .

The size of the input of the DNN was $N = 129N_{fr}$, with N_{fr} being the size of the acoustic context, including the current frame. The output was always a single frame with length $M = 129$. The activation function used in the hidden layers was the sigmoid (Eq. 2.3) except for some experiments using leaky ReLUs (Eq. 2.5), described in Section 4.5. The output layer was comprised of linear neurons, meaning no activation function was used.

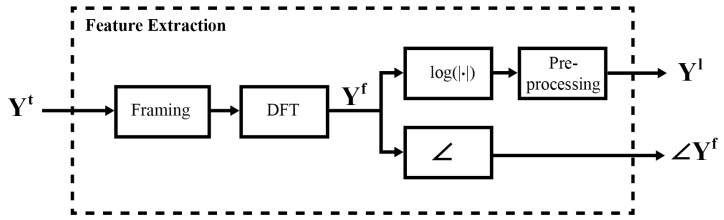


Figure 4.2: Feature extraction block diagram [5]. Based on [1]

4.3 General training procedure

Presented here is the core training procedure, adapted from [31] and [1], as it was used in the project [5] and for most of this master thesis. Variations of the described method

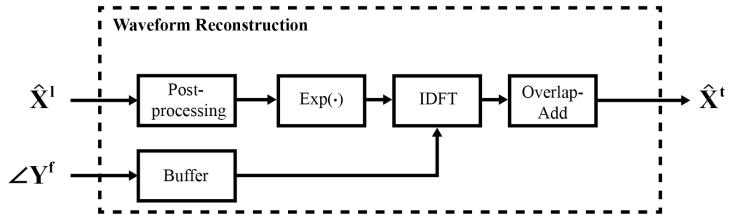


Figure 4.3: Waveform reconstruction block diagram [5]. Based on [1]

are given in the following sections as these in general were compared to this procedure. All hidden layers were initialized using the greedy layer-wise pretraining described in Section 2.7. For the first layer, having the normalized log spectral features as input, a Gaussian-Bernoulli RBM (GB-RBM) was used. For the remaining layers, binary (Bernoulli-Bernoulli) RBMs were used. The pretraining was done using the same data, without targets, as in the supervised fine-tuning.

After pretraining for a set number of epochs, the parameters of the hidden layers of the DNN, $\{\mathbf{W}\}_1^{L-1}$, were set to the values of the corresponding parameters for the RBM. The weights going to the output layer, $\mathbf{W}_L$, were initialized to be zeros. The DNN was then trained in a supervised manner using backpropagation on mini-batches of the training data. An illustration of the described training procedure is shown in Figure 4.4.

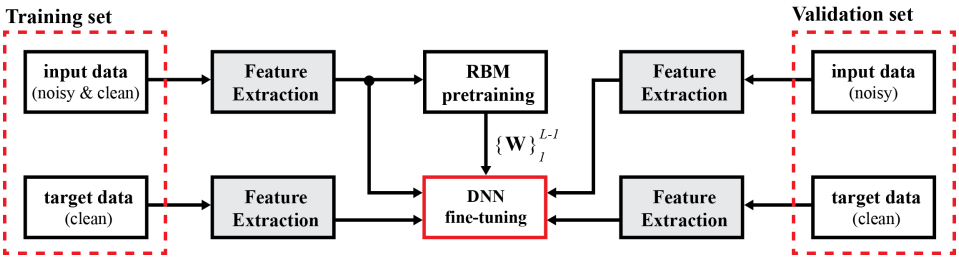


Figure 4.4: Block diagram of training procedure from [5]

The objective function minimized during training was the sum of squared errors of Equation 2.8, normalized by the number of vectors in the mini-batch. The expression is given in Equation 4.1 and will be referred to as mean-square error (MSE), even though the averaging is over mini-batches only and not output elements. This MSE is therefore slightly different than the one defined in the theory section in Equation 2.9, but is one widely applied in the field. The size of the mini-batch used for updating the weights M in Equation 4.1 was set to be $M = 128$ as in [1].

$$J(\theta) = \frac{1}{2M} \sum_{k=1}^{M} \sum_{i=1}^{N} (y_{ki} - t_{ki})^2 \tag{4.1}$$

The update rule for the velocity of layer l is given in Equation 4.2, where $\mathbf{V}_l[n]$ is the

after the n'th epoch for layer l and $\nabla_{\mathbf{W}_l} J(\theta[n-1])$ is the gradient of the objective function with regard to the weight matrix. The velocity is then used to update the weight matrix as described in Equation 4.3. The function $f_{MN}(\psi, c)$ is the max-norm regularization that constraints the columns of ψ to have an L_2-norm less than or equal to the parameter c. This was usually set to 2.6 or 3. The momentum parameter, ω, was initialized to 0.5 and linearly increased to some maximum value, usually set to be 0.9. The learning rate, α, was generally set to be 0.01 but decayed by a multiplicative factor (usually 0.95) after the maximum momentum was reached.

$$\mathbf{V}_l[n] = \omega \mathbf{V}_l[n-1] + (1-\omega)\nabla_{\mathbf{W}_l} J(\theta[n-1]), 1 \leq l \leq L \tag{4.2}$$

$$\mathbf{W}_l[n] = f_{MN}\left(\mathbf{W}_l[n-1] - \alpha \mathbf{V}_l[n], c\right) \tag{4.3}$$

The training ran for a specified number of epochs. For every completed epoch the MSE on a validation set was calculated. If the value was the best seen so far, the current parameters were saved. At the end of training the parameters giving the lowest validation MSE were kept as the final solution.

4.4 Improving the performance of the DNN

There are many ways of improving the DNN in the described speech enhancement system. The most obvious one is to **increase the amount of training data**, but there are several approaches to doing this. In the preceding project work only Gaussian white noise was added to the TIMIT-speech files. One could increase the data set by adding more of the same noise to new clean speech files, or to the existing ones at new SNRs. What was considered more interesting for this master thesis was to introduce new *types* of noise. This was done by using the Aurora2 database described in Section 3.3. Initially the 3 noise types *babble*, *restaurant* and *street* where used for training, together with *AWGN*, although some bigger experiments included the types *airport* and *subway* as well.

The effect of introducing new noise types in the training set, on test sets with seen and unseen data, was analyzed by training two identical DNN with two different datasets. The architectures had 5 frames in the input layer, meaning 645 nodes, three hidden layers with 500 nodes in each, and, as always, 129 nodes in the output layer. Both DNNs were trained using the same 905 files from the TIMIT-database. For the first network every speech file was added white gaussian noise at SNR levels 0,5 and 10 dB, while for the second network every file was added either white gaussian noise or one of the three noise types mentioned above: *babble*, *restaurant* and *street*, from the Aurora2 database, at the same SNR levels. The clean case was also included, which resulted in about 3 hours of speech data for both cases. The results from this experiment motivated a second one, where the two identical DNNs were trained again using 364 TIMIT-files as basis for the training sets. This time, however, every file used for the more general training set were added all four noise types including the clean version, and not just one noise type per file. The *AWGN* case used only two fifths of this data, namely the clean and *AWGN* versions of the same files. The same three SNR levels were used for these training sets. The resulting data amounted to about 4 hours of speech for the more general set and about 1 hour and 20 minutes for the purely *AWGN* set. The results are presented and discussed in Section 5.3.

Another way to improve the speech enhancement system is to **increase the acoustic context** provided with every input frame. This increases the amount of data that the DNN can use to estimate the clean output frame. Increasing the nodes in the first layer does, however, introduce more parameters that the network needs to learn, which could lead to the training becoming more difficult. So increasing the size of the input in this way is not guaranteed to improve performance, especially if the additional information is of little use in the prediction task. Having more frames in the input might also be a factor increasing what [1] refers to as over-smoothing. The effect of different context sizes was studied in [1], where 11 frames in the input was found to be the best alternative. At the start of this thesis 5 frames chosen symmetrically around the current frame were used. Later on this was increased to 9, meaning 4 left and 4 right context frames for every one frame enhanced. When using a first hidden layer of size N_1, this introduces $4*129*N_1 = 516N_1$ new parameters to the model.

Not a lot of effort was spent comparing the performance of using 5 and 9 frames, since the details around implementing this change are pretty trivial and the effect on performance was well documented in [1]. Some experiments with choosing asymmetric context were performed as an initial investigation into the possibility of a real time version of the system. Enough experiments were not performed to conclude anything meaningful and so they are not included in the analysis. A short discussion of the limited results can be found in Appendix A for those interested.

In the paper [1] three techniques are presented for improving the speech enhancement system initially introduced in [31]. These are dropout, global variance equalization and noise aware training. The two first of these were also implemented in this master thesis.

Dropout has already been described in the theory Section 2.5. In [1] a dropout rate of 0.1 was used for the input layer, and 0.2 for all hidden layers. This is a bit lower than the values usually recommended in literature so a few experiments were performed using higher rates than in the paper. Most experiments did however use the same dropout rates as [1]. To investigate the effect of dropout as a regularization technique three different DNNs were trained. The two first had the same architecture: 645 nodes (5 frames) in the input and three hidden layers with 500 nodes in each. The first was trained without dropout and the second with the dropout rates mentioned above. The third DNN had the same dropout rates and the same size for the input layer, but increased the number of nodes in the hidden layers to 645. This was done so that the expected number of nodes in the hidden layer would equal the number of nodes in the DNN trained without dropout: $N_{new}p = 625(1 - 0.2) = 500 = N_{old}$. The learning rates for the DNNs were 0.0005 for pre-training and 0.1 for the fine-tuning. All networks were pre-trained for 50 epochs and then fine-tuned for 200 epochs, storing the parameters giving the best validation performance. To investigate how the technique could combat over-fitting, the models were trained with a relatively small data set equaling about 30 minutes of noisy, and clean speech and the max-norm regularization threshold was increased to 4.

Global Variance equalization (GVE) is a post-processing technique that aims to combat a problem in the enhancement system referred to as *over-smoothing*. This problem is characterized by a suppression of the formant peaks, especially for high frequencies, which can lead to muffled speech [1]. Two definitions of global variance are given in [1]. The first is the frequency dependent version in Equation 4.4 where $\hat{X}^l(d, n)$ is the d'th frequency

sample of the n'th output vector $\hat{\mathbf{X}}^{\mathbf{l}}$. A number of of M estimated frames are used in the computation.

$$GV_{est}(d) = \frac{1}{M}\sum_{n=1}^{M}\left(\hat{X}^l(d,n) - \frac{1}{M}\sum_{n=1}^{M}\hat{X}^l(d,n)\right)^2 \tag{4.4}$$

A frequency independent, or constant, global variance is defined in Equation 4.5, the only difference being that the averaging is over the $D = 129$ frequency bins as well.

$$GV_{est} = \frac{1}{MD}\sum_{d=1}^{D}\sum_{n=1}^{M}\left(\hat{X}^l(d,n) - \frac{1}{MD}\sum_{d=1}^{D}\sum_{n=1}^{M}\hat{X}^l(d,n)\right)^2 \tag{4.5}$$

An identical global variance is calculated for the reference by using the target frame instead of the estimate $\hat{X}^l(d,n)$. Using these values, two equalization factors are defined: the frequency dependent $\alpha(d)$ in Equation 4.6 and the frequency independent β in Equation 4.7.

$$\alpha(d) = \sqrt{\frac{GV_{ref}(d)}{GV_{est}(d)}} \tag{4.6}$$

$$\beta = \sqrt{\frac{GV_{ref}}{GV_{est}}} \tag{4.7}$$

Both factors are used by replacing η in Equation 4.8 to create the re-scaled, GVE post-processed frame estimate $\hat{X}''(d,n)$

$$\hat{X}''(d,n) = \hat{X}^l(d,n) * \eta * \sigma_N(d) + \mu_N(d) \tag{4.8}$$

Enhancement with and without both GVE factors were performed in most experiments, but usually only the results for the factor giving best PESQ score are presented. Section 5.5 features a comparison of both β and a modified version of $\alpha(d)$ tested for a trained DNN. This DNN had 9 frames in the input and three hidden layers with 700 nodes each. It was trained using dropout as previously described and otherwise following the general training procedure described in Section 4.3. 10 hours of data containing the 4 noise types *AWGN*, *babble*, *restaurant* and *street* was used.

4.5 Changing the hidden nodes of the DNN

When studying the literature, the typical modern DNN seem to have moved away from both using sigmoid activation functions as the standard choice for neurons, and from using greedy layer-wise pre-training as the standard initialization technique. Both sigmoid units and pre-training are still in use, but [12] recommends rectified linear units as the new standard neurons in hidden layers, and also warns about using pre-training unless it is known to be appropriate for the problem. Having trained multiple networks with sigmoid units and RBM pretraining it was interesting to test a variation of the system using these alternative activation functions and training purely by backpropagation (with dropout) from

a random initialization. The leaky ReLUs with activation function in Equation 2.5 were chosen as the new units. The slope (α) of the leaky rectifier for negative inputs was set to 0.01 to have non-zero gradients also for negative inputs. The biases for the ReLUs were initialized to 0.1 to increase the chance of the units being activated for the initial inputs.

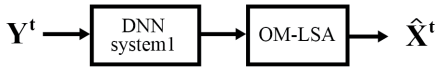
Some initial experiments with smaller architectures were first carried out, until designing a large DNN for both the standard and this alternative setup. The model had 1161 elements (9 frames) in the input, 4 hidden layers with 600 nodes each and the linear output layer with 129 elements. Batches containing about 2 hours of data were loaded 10 times for every epoch, meaning a total of 20 hours of noisy speech data was used to train the networks. Mini-batches with 128 frames were chosen as usual from the 2 hour batch currently in memory. The 6 noise types *AWGN*, *babble*, *restaurant*, *street*, *airport* and *subway* were added to the clean speech from the TIMIT database at SNR levels 0,5,10 and 15 dB. Two noise types at all SNR levels were added to every clean TIMIT file used. The frames within the 2 hour batch were shuffled before being divided into mini-batches. Dropout rates for both DNNs were 0.1 for the first layer and 0.2 for the hidden layers, and the max-norm threshold was set to 3. Momentum was used as described in the general procedure. The learning rate for the sigmoid DNN was 0.0005 for pre-training and 0.1 for fine-tuning with dropout. After the maximum value for the momentum parameter was reached, the learning rate was reduced by 5% for every epoch. The learning rate for the ReLU based DNN was 0.005 and followed the same scaling procedure as the sigmoid DNN. Results from testing the networks using PESQ, MSE and segmental SNR are given in Section 5.7.

4.6 A hybrid speech enhancement system

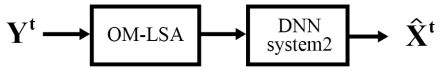
When comparing the DNN based enhancement system and the OM-LSA method from [4] it seemed that both methods were better than the other on different types of noisy speech. It was therefore of interest to see if a combination of the two system would provide an improvement over using the systems on their own. Two cases were tested and compared with the standard DNN speech enhancement system. The first case, called hybrid system 1, was simply to use the OM-LSA method on the reconstructed waveforms from the regular DNN based system. This is illustrated in Figure 4.5b. The advantage of this approach is that it is not necessary to train a new DNN since the input data is the same as before. The second case, called hybrid system 2, is in Figure 4.5c and reverses the order of the two methods. In this case the input data to the DNN will be quite different from the standard noisy input, and therefore a new DNN with the same architecture and amount of training data was designed for this system.

The DNNs of the two methods had 9 frames in the input layer, making 1161 nodes, with the context frames chosen symmetrically around the current frames. Three hidden layers were used, with 700 nodes in each. 10 hours of speech data containing TIMIT speech files added one of the four noisetypes *AWGN*, *babble*, *restaurant* and *street* at the SNR levels 0, 5, 10 and 15 dB were used for the DNN of hybrid system 1. The clean version of every file was also included. For hybrid system 2 the same TIMIT files where added re-sampled noise of the same types as the first DNN, at the same SNR levels. The resulting files were enhanced by the OM-LSA method before used as input data to the DNN. The exception is the clean versions of the files which were input directly to the DNN

(a) The regular DNN speech enhancement system



(b) Hybrid system 1



(c) Hybrid system 2

Figure 4.5: Two possible implementations of OM-LSA DNN hybrid systems.

without pre-processing with OM-LSA. Both DNNs were trained with dropout, using rates of 0.1 for the input and 0.2 for the hidden layers. The systems were tested both with and without global variance equalization.

4.7 General testing procedure

The main quality metric used for testing the speech enhancement systems was the PESQ measure, briefly described in Section 2.9, but the objective of the training, the MSE in Equation 4.1, was also calculated, as well as the segmental SNR in some cases. For testing, three different data sets were generally used:

- a *matched* set, sometimes referred to as **test set 1**, containing a number of TIMIT files from the test folder added some, or all, of the types of noise that were used in training at the 4 SNR levels 0, 5, 10 and 15 dB
- a *mismatched* set, sometimes referred to as **test set 2**, containing the same TIMIT files as test set 1 added the unseen noise types *car* and *exhibition* at SNR levels 0, 5, 10 and 15 dB
- a subset of the training files

The noisy test files were synthesized as specified in Section 3.4 using the global signal power of the training files and storing files with the extracted features for the different files with all noise and SNR combinations. Using a global SNR had some consequences that are discussed in Section 5.2. The pre-processed files to be used in the data sets were chosen by specifying a number, N_{test}, of frames to be included and then reading TIMIT file-names in random order from a list. With each new file-name, only one of the noise types was chosen.

Having selected a TIMIT source file and a noise type, the pre-processed data for all SNR levels of this sentence-noise combination, including the clean case, were loaded. New files were loaded until having at least N_{test} number of frames. When calculating the MSE for the test sets, only the N_{test} first frames were used. In most cases this meant not including all the frames of the last file. For the PESQ and segmental SNR measures, which operate on the waveform, every file that had been selected was used in full at every SNR level. Specifying N_{test} to equal 20 minutes worth of frames led to using about 80 TIMIT files. With 4 SNR levels this meant each set included about 320 noisy sentences. This amount was chosen because it was believed to give a sufficient impression of the performance while not taking too much time to test. In retrospect, much more data could have been used for the MSE calculation, which was quite efficient compared to reconstructing all the files and finding their PESQ score.

The PESQ and segmental SNR for clean files were assessed in some cases, but it was not included when calculating the average score over all test files. The data sets for the MSE did, however, include the clean frames, as this class was part of the data set for which the MSE was being minimized during training.

All testing of the trained DNNs were performed using the DeepLearn Toolbox [39] in MATLAB. The reason for choosing MATLAB was twofold. Firstly: this meant using an environment the student was familiar with from the previous project, and secondly: testing could be performed on a different machine than the one used for training, which was the only one with Theano installed. The last point was convenient because this machine was also used by several other people.

When evaluating the results, especially for the PESQ measure, it was not known how big a change could be considered significant. This made it hard to conclude anything from comparison of methods giving very small performance gains. When this is the case it is noted in the discussion that the results might be too small for meaningful conclusions.

4.8 Testing performance for different sound classes

Since the TIMIT speech files have been labeled according to their phonetic content, it was possible test how the speech enhancement system performed for different sound classes like vowels, fricatives and so on. To do this, a hash table mapping from phonetic label into bigger sound classes was constructed according to the TIMIT documentation. 8 different sound classes were used: *stops*, *affricates*, *fricatives*, *nasals*, *semivowels and glides*, *vowels* and *silence*. Note that both *stops* and *affricates* include the closures preceding the sound.

Since the signals were decimated and some initial frames were dropped during the enhancement (the left context of the first frame and the right context of the last frame), the indices for the labels in the database had to be scaled and shifted appropriately before the phones could be extracted. Having found a segment in the speech waveform, the phonetic label was mapped to the sound class label and the segment was concatenated with other extracted segments of the same class. This was done for clean, noisy and enhanced speech. The quality of the different classes of noisy and enhanced speech were then compared to the corresponding classes of clean speech. To do this, the concatenated arrays were treated as regular speech signals and given as input to the program calculating the PESQ score.

This was done for about 80 files from the TIMIT test set using noise at the 4 SNRs 0, 5, 10 and 15 dB. In addition to the PESQ measure, a segmental SNR measure defined in Equation 4.9 was used. This is similar to the one defined in Equation 2.46, but allows segments of different length.

$$SNR_{seg} = \frac{10}{M} \sum_{j=1}^{M} \log_{10} \frac{\sum_{i=1}^{N_j} x_j(i)^2}{\sum_{i=1}^{N_j} (x_j(i) - \hat{x}_j(i))^2} \tag{4.9}$$

Here x_j is the *j'th* clean speech segment of some sound class, and $\hat{x}_j$ is the corresponding segment of either noisy or enhanced speech. N_j is the length of the current segment and M is the number of segments of the current sound class, extracted from all the processed test files.

The sound class specific testing was only performed for a single DNN and was not the basis for comparing different DNNs or the DNN with the OM-LSA method. The results are presented and discussed in Section 5.8.

Chapter 5

Analysis

5.1 Overview

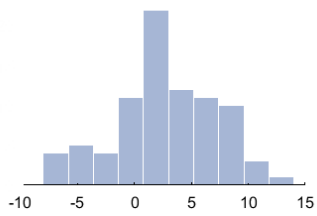
This chapter contains the analysis of the DNN based speech enhancement system. Results from the experiments described in the preceding chapter are presented and discussed in sections 5.3 to 5.8. The first of these sections analyze, by comparison of differently trained DNNs, the benefit or disadvantages of using the presented techniques and changes to the system. Section 5.7 also compares two different DNN setups with the OM-LSA method described in Section 2.8. The two last sections analyze a single trained DNN, in an attempt to learn more about how the system works. Section 5.8 does this by studying the performance on different sound classes, while Section 5.9 contains an interpretation of the learned weights of one of the DNNs.

5.2 A note about the test data

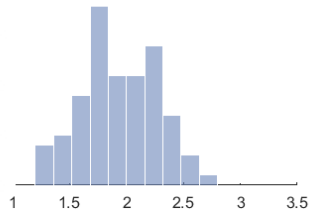
As mentioned in the previous chapter the different DNNs were tested using data with seen and unseen noise added to globally set SNR levels. Many of the results presented in this chapter give the performance measures for the different globally set SNR levels separately. This is done even though the SNR of individual files will vary around the stated SNR value value depending on the signal power of the clean speech relative to the global average. To get an idea of the distribution of the file based SNR values, a histogram of these values for the test set with seen noise at global SNR level 5dB is plotted in Figure 5.1a.

Studying Figure 5.1a, it is clear that the variation in SNR for the files is quite large. Figures 5.1b and 5.1c show histograms of the PESQ scores for the same files as in 5.1a, before and after enhancement with the DNN system. Noting that the PESQ measure returns values in the range $[-0.5, 4.5]$ these figures also show what should be considered a significant variation in quality for the different files.

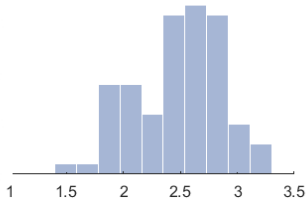
In Figure 5.2 every file's SNR and its PESQ score before enhancement are plotted. Not surprisingly there seems to be a positive correlation between SNR of the file and its



(a) SNR of input files in dB



(b) PESQ of input files



(c) PESQ of enhanced files

Figure 5.1: Distribution of SNR of input files and PESQ of output and enhanced files in test set 1 for global SNR 5 dB. The enhanced files are from using the ReLU-DNN in Section 5.7

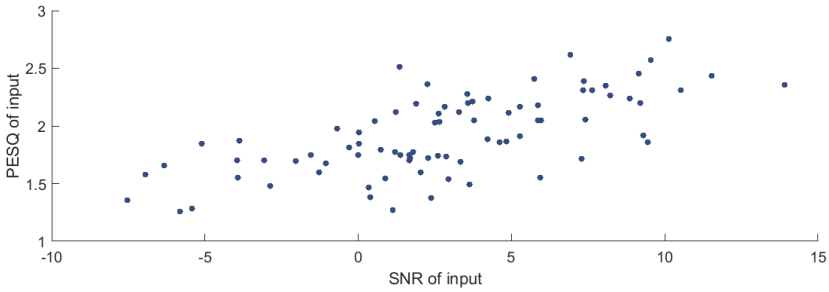


Figure 5.2: Relationship between SNR and PESQ of the input files

perceived quality as measured by the PESQ. Figure 5.3 gives the same plot for SNR of input files against their PESQ score after enhancement. These seem to be have a slightly higher correlation. This indicates that the spread in SNR for the input files can be assumed to be partly responsible for the spread in PESQ for the enhanced files.

A spread in SNR and PESQ might not be a big problem if the average PESQ for the variable SNR files matches the average for files having exactly the global SNR level in question. When choosing individual files from the different SNR groups to listen to, however, one could not expect the results to be representative for that particular SNR level. To illustrate this, the files with best, worst and closest to the average PESQ were stored when testing the DNNs in section 5.7. The results showed that the best file from the SNR 0

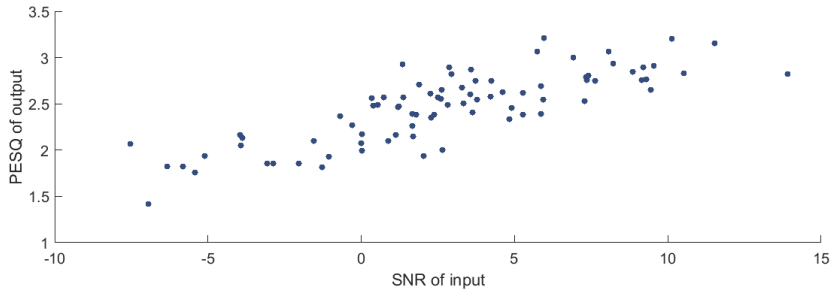


Figure 5.3: Relationship between SNR of the input files and PESQ of the resulting enhanced files

dB class had higher file based SNR before enhancement (and higher PESQ after enhancement) than the worst result in the SNR 15 dB class (results from this experiment are given in Appendix B). Considering this it might have been better to set the SNR individually for every file when adding noise for testing to get results that were more representative for the listed SNR. Because of limited time this was, however, not done. Note that using a globally set SNR for the training files probably is still considered a good idea since this means a wide range of SNR levels are represented in the data.

In addition to the spread in SNR, the subset of test files that has been used for testing can be seen in Figure 5.1a to be centered around a SNR value lower than the global average. In fact, as given in Table 5.1, the average SNR levels of the test subset was found to be 2.2 dB under the global average value for every SNR. This is a result of using a globally set SNR and then choosing only a subset of these files for the actual testing. That the average SNR is lower than expected might have had a bigger effect on the results than the variation in file based SNR around this average had. Again, because limited time did not allow all the experiments to be repeated, the reader is advised to keep these results in mind while reading the following results.

Table 5.1: Global average SNR used when adding noise to files and average SNR of chosen testfiles

Global average SNR	0	5	10	15
Test set average SNR	-2.2	2.8	7.8	12.8

5.3 Adding new types of noise

This section gives the results for the experiment evaluating the effect of introducing new types of noise to the training set, as described in Section 4.4. Two datasets were used for testing: one matched case, in which 81 TIMIT test files were added white gaussian noise to SNR levels 0,5,10 and 15 dB, and one where the same files where added the unseen noise types *car* and *train* from the Aurora2 database. In Figure 5.4 the PESQ score for the two DNNs on the two test sets is shown, including the PESQ score before enhancing for

Figure 5.4: Average PESQ score using two different test sets for DNNs trained with either all *AWGN* or *AWGN* and three types of Aurora2 noise. The DNN with Aurora2 noise chooses one of the 4 types to add to every TIMIT speech file.

comparison. The Table 5.2 shows the MSE of the enhanced frames of the same test sets. The leftmost columns of the two bar plots show the average PESQ of the noisy input files, which is the same for both DNNs on the same test set.

Table 5.2: MSE on two different test sets for DNNs trained with either all *AWGN* or *AWGN* and three types of Aurora2 noise. The DNN with Aurora2 noise chooses one of the 4 types to add to every TIMIT speech file.

Noise types in test set	**DNN trained with AWGN only**	**DNN trained with 4 noisetypes**
AWGN	**35.38**	38.13
unseen noise	86.96	**57.43**

The two right columns of the plot to the left show that the DNN trained with only *AWGN* gives a slightly bigger improvement in PESQ-score on the test set featuring *AWGN* only. This difference is also found in the MSE results in the first row of Table 5.2. This could be because the purely *AWGN* trained DNN had training data with this type of noise added to all the files, while the other DNN only used *AWGN* for about a quarter of the files. Using more training data of this type should be expected to improve performance on the matched test set. Another possible explanation of why the purely *AWGN* trained DNN works better for this test set is that it is more specialized for this type of noise. Since the DNN trained with 4 noisetypes uses a more diverse or general training set, it is possible that it might not learn mappings that work especially well for white noise, but bad for other types, if any such mappings exist. These results motivated the second experiment described in section 4.4, where the more general set was chosen to contain

the same amount of files with *AWGN* as the purely *AWGN* set. The results are given in PESQ-scores in Figure 5.5 and MSE in Table 5.3.

Studying the results of this second experiment the PESQ-scores look very similar to Figure 5.4 with the *AWGN* DNN again performing slightly better for test data with this noise type. The difference in MSE is bigger in this case, as seen in the first row of Table 5.3, with the specialized DNN performing better for the *AWGN* test set. It looks like adding more noise types to the same TIMIT-files is making the DNN worse at enhancing the *AWGN* test data. This seems to confirm that the *AWGN* trained DNN in the first experiment is better at this type of noise mainly because it is more specialized and not just because it had more training data with *AWGN* than the other neural network.

Figure 5.5: Average PESQ score using two different test sets for DNNs trained with either all *AWGN* or *AWGN* and three types of Aurora2 noise. The DNN with Aurora2 noise adds all 4 types to every TIMIT speech file.

Table 5.3: MSE on two different test sets for DNNs trained with either all *AWGN* or *AWGN* and three types of Aurora2 noise. The DNN with Aurora2 noise adds all 4 types to every TIMIT speech file.

Noise types in test set	**DNN trained with AWGN only**	**DNN trained with 4 noisetypes**
AWGN	**34.48**	55.34
unseen noise	93.45	**87.09**

Of course, if we want to design a general speech enhancement system the matched test set is not the most interesting case to study. The benefit of using 4 noise types instead of just one is seen when testing on unseen noise types, illustrated in the bar plot to the right of both figures 5.4 and 5.5. While the AWGN trained DNN doesn't look to do much at all, the DNN trained with 4 types gives a noticeable increase in PESQ-score. Again the difference

is greater in terms of MSE, especially for the first experiment given in the second row of Table 5.2.

A last thing to note about these experiments is the difference in MSE for the DNN trained with 4 noise types between the first and second experiment. Comparing the second column of tables 5.2 and 5.3, it is clear that the first DNN trained with about 3 hours of speech data performs significantly better than the second DNN trained with 4 hours of speech data. The difference here is that the first training set, although shorter in duration, is made by adding noise to 905 TIMIT-files, while the second only uses 364, but adds all noise types to every file. This means that 13 input vectors of the second training set have the same target vector: one clean version and four different noisy versions of the same frame at the three SNR levels 0,5 and 10 dB. The first case had only 4 cases of input vectors sharing a target vector: one clean and three SNR levels of the single noise type chosen to be added to that file. The conclusion is that a smaller, but more diverse training set can result in a DNN with better performance than one with a larger, but more redundant training set. This is useful to be aware of when wanting to add more data to the training set. Since more data gives an increase in training time, it is probably a good idea to try to prioritize diversity when expanding the training set.

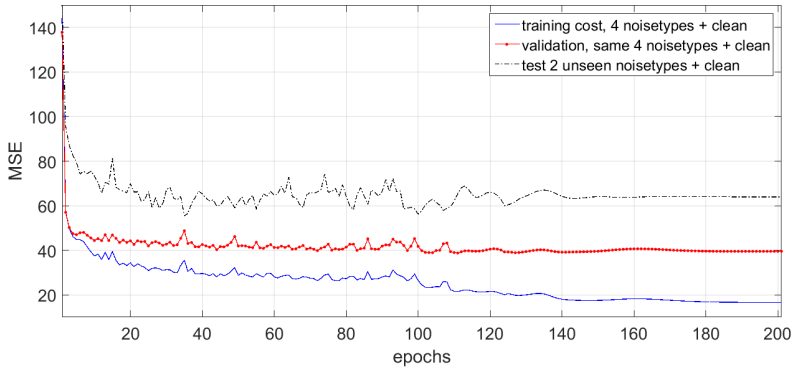
5.4 Using dropout for increased generalization

As described in Section 4.4 an experiment of training three DNNs on the same data with and without dropout was performed. To get an idea of how dropout affected the learning, the cost (MSE) for training, validation and an unmatched test set was calculated for every epoch. The results are plotted in Figure 5.6. The curves in Figure 5.6a and Figure 5.6b are for the same DNN architecture without and with dropout respectively. It looks like introducing dropout stabilizes the learning in some way, leading to slightly less noisy curves. In addition, the curves for the cost of the different data sets have been brought closer together. This could indicate that the network is generalizing better. It is clear that this is a result both of higher training error and lower cost on the validation and test sets.

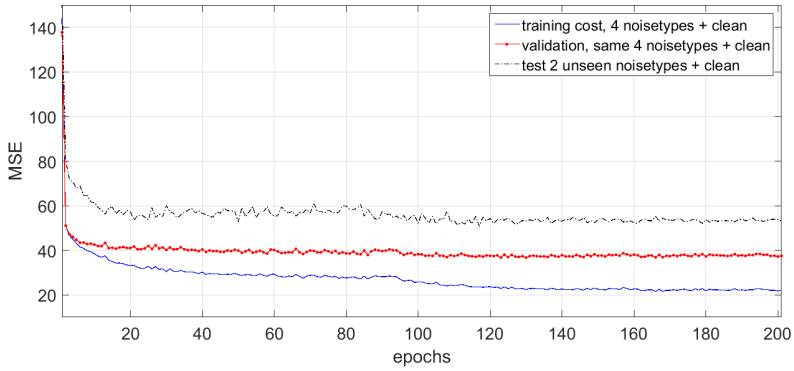
The third DNN with increased model size gives the curves in Figure 5.6c that look very similar to the ones for the smaller model with dropout. From looking at these figures, increasing the number of parameters seems not to have negatively affected the training in any big way, except possibly a slightly higher test cost.

Table 5.4: MSE and PESQ on test sets for the discussed DNNs using half an hour of data, with and without dropout

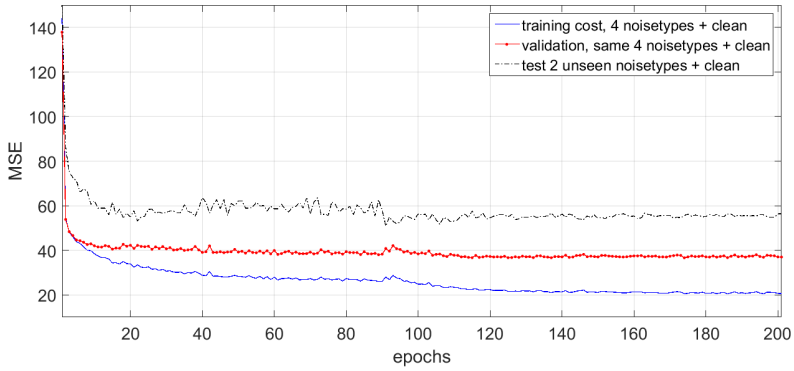
	Seen noise		**Unseen noise**	
DNN trained...	**MSE**	**PESQ**	**MSE**	**PESQ**
without dropout	41.29	**2.33**	66.96	**2.29**
with dropout and same architecture	**37.18**	2.29	**54.40**	2.25
with dropout and bigger architecture	**37.18**	2.32	55.79	2.28



(a) Without dropout



(b) With dropout, same architecture



(c) With dropout, bigger architecture

Figure 5.6: Cost for training, validation and test sets during learning, with and without dropout.

In Table 5.4 the results in MSE and PESQ for the parameters giving the lowest validation cost during training for the three DNNs are given. These are from testing on the same unseen test set as previously described and a test set with seen noise now featuring all 4 noise types used in training. Again the effect of dropout on the MSE is noticeable, this being lowered for both DNNs trained with dropout on both test sets. A somewhat surprising result was that the average PESQ scores on the test files were worse after introducing dropout. This problem seems to be somewhat alleviated when increasing the model size, although the PESQ results for the third DNN are still slightly below the first one. In [1] there is, contrary to this result, reported an *improvement* in PESQ after implementing dropout, so this result did not seem to make sense. One possible explanation is that only the sizes of the hidden layers in the third architecture were increased to have the same expected number as the DNN trained without dropout, and nothing was done with the input layer. Since nodes are also dropped from the input, the first layer of connections has a lower expected value for the number of parameters than the ”dropout-free” DNN had. Instead of having the same expected number of nodes in the individual hidden layers, maybe the network should have been scaled to have the same expected number of parameters in total, or more. Assuming nothing was wrong with the implementation, the main difference from [1] was in model size and amount of training data, so this supports the idea that the bigger third model might still have been too small. Whatever the explanation, dropout was used in spite of these results for the remaining experiments.

5.5 Using Global Variance equalization

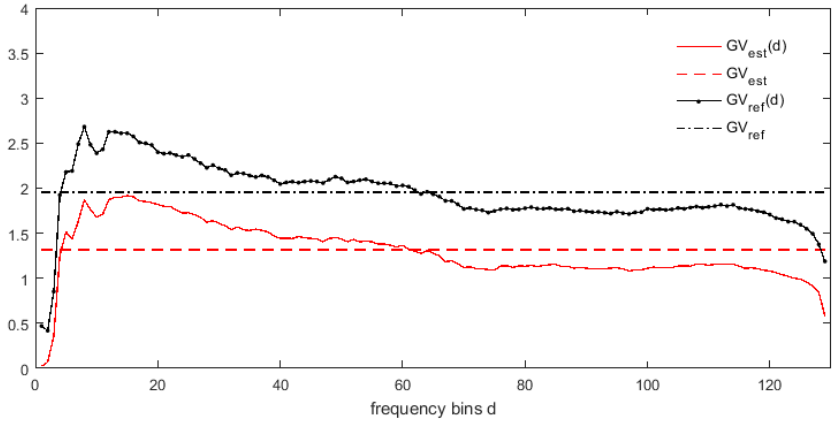
In Figure 5.7a the constant and frequency dependent global variances of the target and estimated clean speech features are illustrated. The values for the estimated clean speech will of course be slightly different depending on the DNN and what data is used. These results were found using a subset of the training data for a 4 layer DNN trained with 10 hours of speech data.

Calculating the frequency dependent global variance factor, $\alpha(d)$, results in much higher values for low frequencies, as illustrated in Figure 5.7b. For this reason a modified factor, $\tilde{\alpha}(d)$ was used in practice. Here the values for the first 6 frequency bins was set equal to the values of bins 7-11 of $\alpha(d)$, in reverse order. In the figure it is clear that $\tilde{\alpha}(d)$ and β are quite similar.

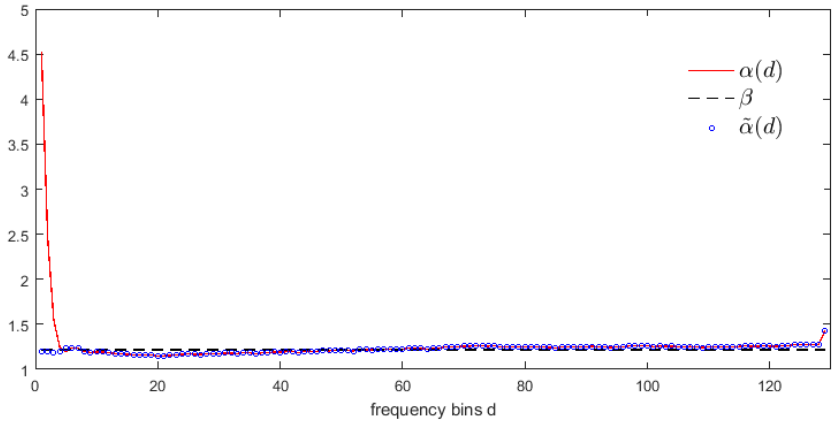
Table 5.5: Effect of global variance equalization (GVE) on **PESQ** using the ReLU-DNN from Section 4.5

Test set	Input	Enhanced without GVE	Enhanced with $\hat{\alpha}(d)$ GVE	Enhanced with β GVE
seen noise	2.09	2.55	**2.62**	2.61
unseen noise	2.07	2.42	2.52	**2.53**

In Table 5.5 the average PESQ scores before and after speech enhancement using a DNN with and without global variance equalization are given. Clearly, using GVE has an impact on the PESQ results, improving the score for both test sets using both the frequency



(a) Global variance for the reference and estimated clean speech



(b) Global variance equalization factors

Figure 5.7: The different global variances of the clean and estimated speech and the resulting equalization factors used in post-processing

dependent and independent factors. It is not clear here which of the two factors are the best as both perform slightly better than the other for one of the test sets.

Listening to some of the results the effect of GVE was not always noticeable, but in some cases it seemed to remove some of the residual noise that often is present after enhancement. In a few of the examples, the energetic voiced sounds were also noticeably louder compared to the results without GVE. In [1] the frequency independent β was reported to give better results for testing on unseen noise environments, which also is the tendency in Table 5.5, if a difference of 0.01 can be considered significant. This was however not always the case for other experiments, and whichever of the factors that gave the best results for a particular DNN was chosen as the standard for that DNN.

Since the GVE-factors are multiplied with the output before re-scaling using mean and

Table 5.6: Effect of global variance equalization (GVE) on **MSE** using the ReLU-DNN from Section 4.5

Test set	Input	Enhanced without GVE	Enhanced with $\hat{\alpha}(d)$ GVE	Enhanced with β GVE
seen noise	129.93	**37.10**	42.57	42.33
unseen noise	122.68	67.89	**64.25**	**64.25**

standard deviation estimates, one can find the MSE between the output-GVE product and the target. The results, for the same DNN as above, are given in Table 5.6. Although there is an improvement for unseen noise, for the test set with seen noise the equalization actually results in higher MSE. The purpose of global variance equalization is, however, to improve the subjectively experienced quality of the speech and the PESQ score is assumed to be a better measure of this than the MSE of the log-magnitude features. For this reason the effect of GVE on the MSE has largely been ignored when considering the different DNN systems.

5.6 Testing the hybrid system approach

As described in Section 4.6, two ways of combining the OM-LSA method with the DNN based speech enhancement were implemented and tested. The two methods calls for different DNNs, here simply called DNN1 and DNN2, trained on features from noisy speech and features from OM-LSA enhanced noisy speech respectively. The results for the DNN are without global variance post-processing unless stated otherwise.

In Table 5.7 and Table 5.8 the PESQ results for hybrid system 1, using test files with seen and unseen noise respectively, are given. The first column gives the average PESQ of the noisy input speech at different SNR levels. The second column shows the result after the first enhancement-stage, which here is the DNN1 system. The last column show results after the files are further enhanced by the OM-LSA method.

Table 5.7: Average PESQ for test set with **seen** noise after the two stages of hybrid system 1 from figure 4.5b

	Input	After DNN1	After DNN1 and OM-LSA
SNR 15	2.57	**2.88**	2.74
SNR 10	2.24	**2.63**	2.46
SNR 5	1.93	**2.35**	2.13
SNR 0	1.64	**2.00**	1.74
Average	2.09	**2.47**	2.27

For both tables 5.7 and 5.8 the highest average PESQ scores result from only using the DNN1 system. This indicates that using OM-LSA for post-processing not only fails to provide additional performance gains, it actually seems to result in lower speech quality, as measured by the PESQ score. The numbers are not given here, but using Global variance

Table 5.8: Average PESQ for test set with **unseen** noise after the two stages of hybrid system 1 from figure 4.5b

	Input	**After DNN1**	**After DNN1 and OM-LSA**
SNR 15	2.55	**2.77**	2.73
SNR 10	2.22	**2.53**	2.45
SNR 5	1.90	**2.20**	2.12
SNR 0	1.62	**1.90**	1.75
Average	2.07	**2.35**	2.26

equalization for this system actually resulted in lower PESQ after using OM-LSA, even though it improved the results after the DNN1 alone. When listening to some of the files it seemed like the OM-LSA was successful in removing some of the residual noise from the DNN, but in some cases it alse introduced some distortion of the speech signal. Based on the numbers and listening tests it seems possible that using OM-LSA for post-processing can improve the results in some cases, but in general it seems that the DNN1 alone outperforms hybrid system 1.

In Table 5.9 and Table 5.10 the PESQ results for hybrid system 2 are given. Again the results after the first part and after the whole hybrid system are provided. The results for the DNN1 alone are also included for comparison. First, comparing the result after just using the OM-LSA (column two) to the whole hybrid system (column three), the addition of the DNN after the OM-LSA results in better PESQ score in all cases except for SNR 15. For this highest SNR the DNN gives no improvement for seen noise and lower PESQ score for unseen noise. In general, the improvement in PESQ of using the DNN seems to be bigger for the lower SNRs than for the higher.

Further, comparing the hybrid system to using only the DNN1 system (columns three and four) the hybrid system is seen to give lower PESQ score for test files with seen noise, but higher for files with added unseen noise. When Global variance equalization was implemented, however, the pure DNN gave equal or better PESQ score for the data with unseen noise also. The results with GVE are given in Table 5.11.

Table 5.9: Average PESQ for test set with **seen** noise after the two stages of hybrid system 2 from figure 4.5c

	input	**After OM-LSA**	**After OM-LSA and DNN2**	**DNN1 only**
SNR 15	2.57	2.87	2.87	**2.88**
SNR 10	2.24	2.50	2.60	**2.63**
SNR 5	1.93	2.10	2.31	**2.35**
SNR 0	1.64	1.65	1.95	**2.00**
Average	2.09	2.28	2.44	**2.47**

In Table 5.12 the MSE results for DNN1 and DNN2 are given for the same two test sets used in the PESQ-testing, and a subset of the training set. The MSEs of the inputs

Table 5.10: Average PESQ for test set with **unseen** noise after the two stages of hybrid system 2 from figure 4.5c

	input	After OM-LSA	After OM-LSA and DNN2	DNN1 only
SNR 15	2.55	**2.92**	2.85	2.77
SNR 10	2.22	2.56	**2.58**	2.53
SNR 5	1.90	2.16	**2.25**	2.20
SNR 0	1.62	1.74	**1.93**	1.90
Average	2.07	2.35	**2.40**	2.35

Table 5.11: Average PESQ for hybrid system 2 and DNN1 for the two test sets when using Global Variance Equalization

	Hybrid system 2 seen noise	DNN1 only seen noise	Hybrid system 2 unseen noise	DNN1 only unseen noise
SNR 15	2.94	**2.97**	**2.91**	**2.91**
SNR 10	2.63	**2.69**	2.61	**2.64**
SNR 5	2.30	**2.36**	2.27	**2.29**
SNR 0	1.95	**2.01**	1.92	**1.97**
Average	2.46	**2.51**	2.43	**2.45**

are also given, which for DNN1 is the noisy speech features and for DNN2 is the noisy speech features after OM-LSA enhancement. Comparing the input columns we see that using OM-LSA for pre-processing gives considerably lower MSE for all three sets. Using DNN2 on this data further improves the results and gives lower MSE than using the DNN1 alone. The difference in MSE after enhancement between the two DNNs is greatest for the unseen noise case, where DNN1 performs considerably worse than DNN2. In fact the MSE after DNN1 is higher than the MSE of the input for DNN2 (after OM-LSA only). Consequently, the performance gap of the DNNs between the sets with seen and unseen noise is smaller for the hybrid system than the DNN1 system. One possible explanation is simply that the SNR of the input files are higher for the OM-LSA pre-processed features than for the ”raw” noisy features, making the speech enhancement in the DNN easier for DNN2. This interpretation seems very plausible considering the big difference in MSE for the input signals of the two DNNs. An alternative explanation is that the residual noises after OM-LSA method are more similar to one another than the original noise types were, which makes the mismatched test data seem more like the data the DNN2 has seen during training. Another way of formulating this is that the OM-LSA method might have managed to remove parts of the noise that were the most mismatched to the training data, removing some of the disparity between the test sets.

The PESQ results indicate DNN1 with GV gives better results in than hybrid system 2, while the MSE shows the opposite tendency. One explanation is that the OM-LSA will not only remove the noise, but might distort the speech in a way that affects the PESQ measure more than the MSE. A possible way to avoid speech distortion from OM-LSA, but use the method together with the DNN, is discussed under future work in Chapter 6.

Table 5.12: MSE before and after DNN enhancement in the two hybrid systems

	Input		Enhanced	
Data set	**DNN2**	**DNN1**	**DNN2**	**DNN1**
subset of training set	53.39	135.23	**28.11**	32.54
test set with seen noise	52.66	129.48	**29.26**	33.13
test set with unseen noise	50.98	137.85	**32.06**	56.97

5.7 Using rectified linear units and comparing with OM-LSA

As mentioned in Section 4.5 some smaller experiments with the leaky-ReLU replacing the sigmoid neurons were done initially. These models, in general, performed slightly worse than the pre-trained sigmoid models both in terms of higher MSE and lower average PESQ. For bigger architectures with more data the results from the two networks seemed to be more comparable. In this section the result from the 5 layer architecture with 600 nodes in the hidden layers and 9 frames in the input, trained with 20 hours of data, are given. For the test set with seen noise only the four noisetypes *AWGN*, *babble*, *restaurant* and *street* were used, leaving out the two last types used in the training set. The mismatched test set was the same as previously.

The tables 5.13 and 5.14 give PESQ result for both test sets enhancing with the two DNN systems and the OM-LSA method, including the average PESQ scores of the input data and the optimal PESQ score. The ”optimal” column gives the average PESQ of the waveforms reconstructed by combining the optimal magnitude (the clean target frame) with the phase of the noisy input. Since the system only enhances the magnitude this is a better measure for the best attainable performance than the maximum PESQ value of 4.5. The results for the clean input case are also provided, but these are not included in the averages given in the bottom row of the tables. Studying the columns with PESQ scores for enhanced test files we see that both systems lead to a significant increase in the PESQ score compared to the noisy input, but a big decrease for the clean case. Both DNNs also outperform the OM-LSA method for the noisy data. For the test set with seen noise the ReLU-DNN gives better PESQ results than the pre-trained sigmoid-DNN for all SNR and the clean case. For the unseen noise it performs about the same, or better by an insignificant amount.

In the tables 5.15 and 5.16 the segmental SNR of Equation 2.46 is used on the input and enhanced files from both systems. Both DNN systems leads to an improvement which again seems to be bigger for the ReLU-DNN. Comparing with the OM-LSA method shows that this system results in higher score than the DNNs on the mismatched test set and the SNR 15 dB case for the test set with seen noise. It is possible that the OM-LSA method has been able to remove more noise in the second test set, but also negatively affected the speech leading to a higher segmental SNR, but lower PESQ compared to the DNN systems.

For completeness, the MSE calculated on the log-magnitude spectrum of the frames are given in Table 5.17. Here it is seen that the ReLU-DNN results in a slightly lower MSE

Table 5.13: Average PESQ for test set with **seen** noise added at different SNR using a DNN with sigmoid units and one with rectified linear units.

	Input	Enhanced sigmoid-DNN	Enhanced ReLU-DNN	Enhanced OM-LSA	Optimal
clean	4.50	3.54	3.59	**4.01**	4.50
SNR 15	2.57	3.01	**3.07**	2.84	3.81
SNR 10	2.24	2.73	**2.81**	2.49	3.63
SNR 5	1.93	2.40	**2.47**	2.08	3.49
SNR 0	1.64	2.06	**2.11**	1.65	3.33
Average	2.09	2.55	**2.62**	2.26	3.57

Table 5.14: Average PESQ for test set with **unseen** noise added at different SNR using a DNN with sigmoid units and one with rectified linear units.

	Input	Enhanced sigmoid-DNN	Enhanced ReLU-DNN	Enhanced OM-LSA	Optimal
clean	4.50	3.54	3.59	**4.01**	4.50
SNR 15	2.55	2.95	**2.97**	2.88	3.79
SNR 10	2.22	2.70	**2.71**	2.53	3.65
SNR 5	1.90	**2.36**	**2.36**	2.14	3.50
SNR 0	1.62	**2.06**	**2.06**	1.71	3.35
Average	2.07	**2.52**	**2.52**	2.31	3.57

for the test set with seen noise and the training set, but a bigger MSE for the test set with unseen noise. This differs from the results for the PESQ and segmental SNR measure. This disparity between the different measures has been encountered multiple times when comparing the performance of differently trained DNNs. For example, comparing the MSE and PESQ results of the DNNs in this chapter to the results presented in earlier sections, we see that other DNNs often perform better in terms of MSE than the ones in this section, but that they have worse PESQ results. A good explanation of this cannot be provided by the student.

Some listening tests were performed by the student for the three methods compared

Table 5.15: Average segmental SNR for test set with **seen** noise added at different SNR using a DNN with sigmoid units and one with rectified linear units.

	Input	Enhanced sigmoid-DNN	Enhanced ReLU-DNN	Enhanced OM-LSA
SNR 15	3.00	4.66	5.38	**6.22**
SNR 10	-2.03	3.51	**4.29**	2.91
SNR 5	-6.98	2.19	**2.92**	-0.52
SNR 0	-11.98	0.83	**1.31**	-4.04
Average	-4.50	2.80	**3.48**	1.14

Table 5.16: Average segmental SNR for test set with **unseen** noise added at different SNR using a DNN with sigmoid units and one with rectified linear units.

	Input	**Enhanced sigmoid-DNN**	**Enhanced ReLU-DNN**	**Enhanced OM-LSA**
SNR 15	2.68	4.11	4.44	**6.57**
SNR 10	-2.35	2.29	2.45	**3.57**
SNR 5	-7.26	0.13	0.21	**0.58**
SNR 0	-12.29	-2.56	**-2.44**	-2.50
Average	-4.80	0.99	1.16	**2.05**

Table 5.17: MSE for the DNNs with ReLU and sigmoid units

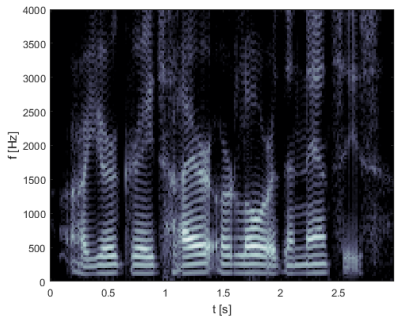
Data set	**Sigmoid-DNN**	**ReLU-DNN**
subset of training set	36.55	**34.96**
test set with seen noise	38.27	**37.10**
test set with unseen noise	**62.17**	67.89

in this section. As mentioned in Section 5.2 the files with the PESQ scores that were highest, lowest and closest to the average, out of all the test files, for every SNR, were stored. Quite often, but not always, the same files were found to be best and worst in terms of PESQ for the two variations of the DNN system. Which files that were considered ”most average” differed quite a lot more. Comparing the enhanced versions of the files that were the same for the two systems, it was not easy to claim that one was better than the other. It was also not always easy to decide if the DNN based systems were better than the OM-LSA either, especially for the worst case files of the lower SNR levels, where the speech was almost impossible to understand for all systems. As indicated in the plot of Figure 5.3, the files with lowest PESQ would generally be ones with very low SNR. For removing non-stationary or transient noises, the DNN systems were much more effective than OM-LSA, which was also noted in [1]. Another big difference between the OM-LSA and DNN systems is in the nature of the residual noise. While the OM-LSA often would fail to remove much of the non-stationary noises, the DNN would in some cases introduce annoying background noise very different from what was present in the noisy input speech.

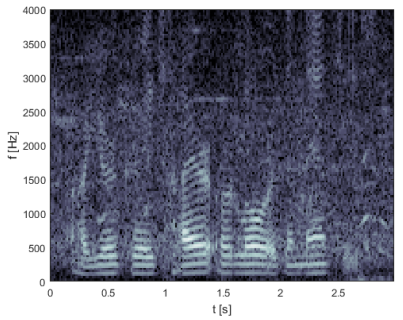
In addition to listening to files from the test sets, some audio files with noisy speech used for demonstration purposes for papers [1, 31] were downloaded from [7] and [43] and enhanced using the ReLU-DNN from this section. The enhanced speech was compared subjectively by the student with the enhanced results from the systems in [1] and [31], which were also given on [7] and [43]. On these files the results achieved with the ReLU-DNN was considered comparable to those from [7], but on the files from [43] the system from [1] was superior, as expected. Spectrograms for one of the files from the website [7], with the unseen *exhibition* noise added to 10 dB SNR are illustrated in Figure 5.8. In Appendix C links to the audio examples are provided.

It should also be noted that, because of the increased training time for the DNNs in this section it was too time-consuming to try many different choices for all the hyper-parameters. That the ReLU based DNN seems to work better here might therefore be a

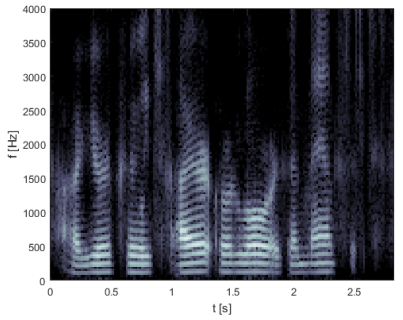
case of the sigmoid-based one not being as well optimized. However, the results do imply that the ReLU-DNN trained by backpropagation alone might be an equally good choice as a sigmoid-based DNN with RBM pre-training. ReLU based networks are reportedly [44] easier to optimize and faster to train, which, in addition to not needing pretraining makes them attractive for further development.



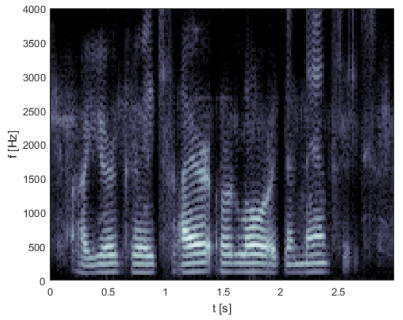
(a) Clean speech



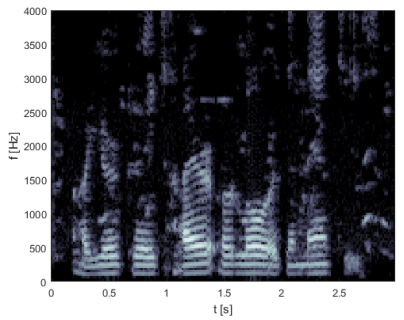
(b) Noisy speech



(c) Speech enhanced with ReLU-DNN



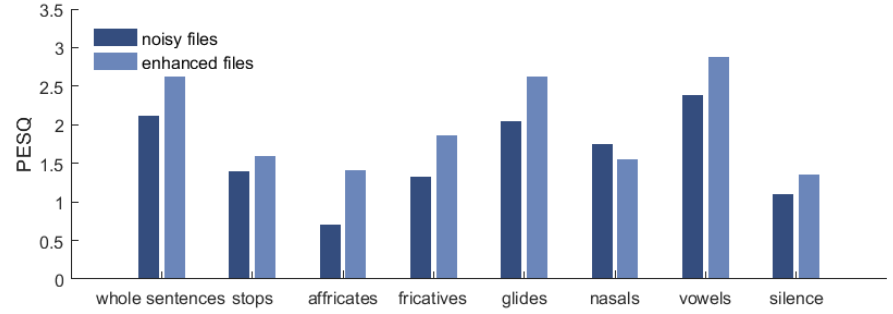
(d) Enhanced speech from [31]



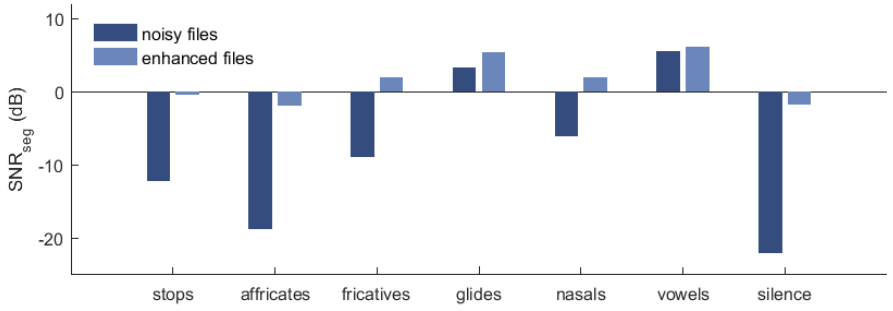
(e) Enhanced speech using OM-LSA

Figure 5.8: Spectrograms using files from [7].

5.8 Performance on different sound classes



(a) Average PESQ

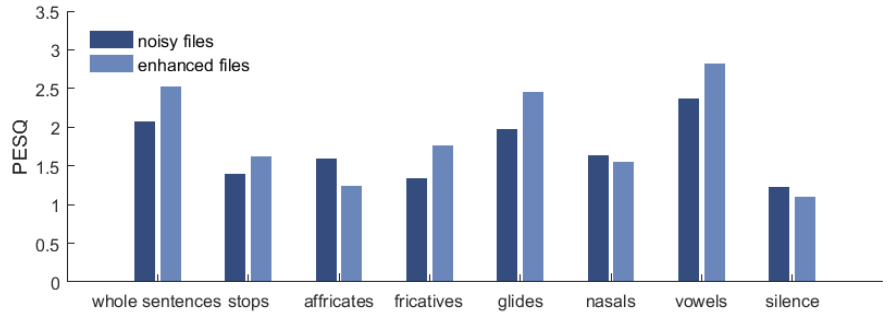


(b) Segmental SNR

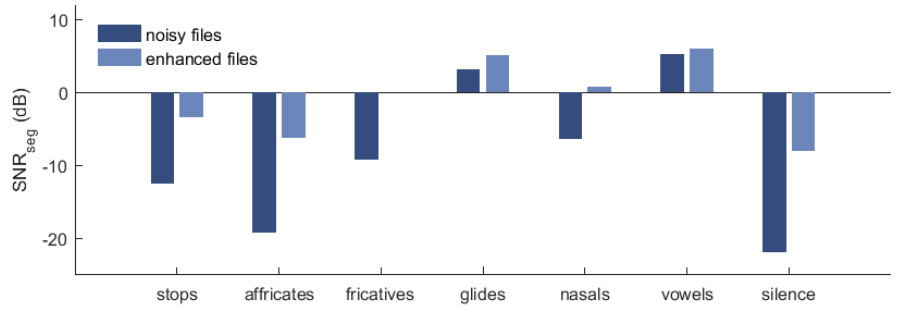
Figure 5.9: Comparison of clean enhanced sound classes from test files with seen noise

Figure 5.9 show results for the selected test files with added *AWGN*, *babble*, *restaurant*, *street*, *airport* and *subway* noise. For the network tested here these are the same types of noise used in the training set. Figure 5.10 show results for the same TIMIT files with the added unseen noisetypes *car* and *exhibition*. Note that the class *semivowels and glides* is simply labeled as *glides* in the figures, although it contains both types of sounds. For the PESQ results in figures 5.9a and 5.10a, the average PESQ scores for whole sentences are also included. Looking at these figures, it is clear that *vowels* and *semivowels and glides* are the sound classes with the highest PESQ score for both noisy and enhanced speech. This seems reasonable considering they are sounds with high energy, meaning they have a higher SNR relative to other sound classes. This is also confirmed looking at the corresponding segmental SNR values in figure 5.9b and 5.10b, where these two classes are the only two that have a positive SNR for the noisy speech. The fact that the segmental SNR measure for the *silence* class is a number, and not $-\infty$, illustrates that the extracted segments contain some non-zero samples like traces of ends or starts of other sounds and possibly some background noise. The power of the noisy signal will, however, be dominated by the noise, which is why the segmental SNR value is so low for this

class. The same applies to *stops* and *affricates* since these classes include the longer silent closures before the more energetic release of the sound.



(a) Average PESQ

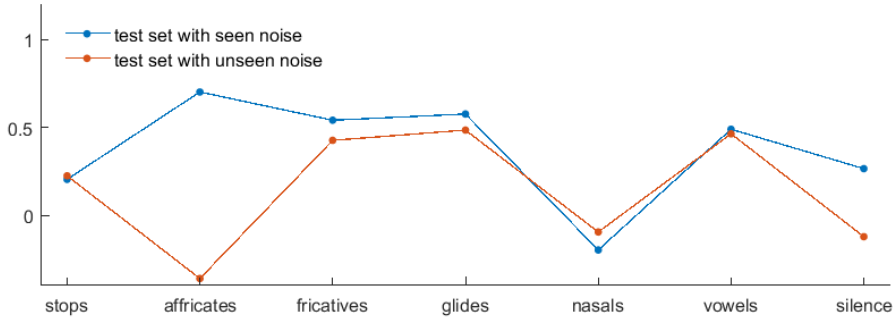


(b) Segmental SNR

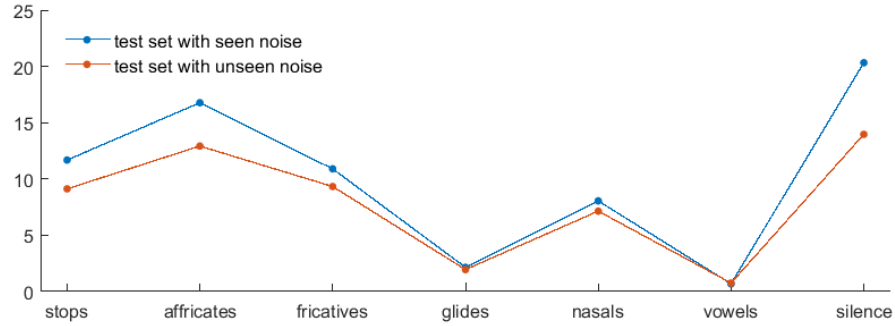
Figure 5.10: Comparison of clean enhanced sound classes from test files with unseen noise

In the segmental SNR figures for both test sets there is an improvement going from noisy to enhanced speech for all sound classes. For PESQ, however, *nasals* for seen noise and *affricates*, *nasals* and *silence* for unseen noise stand out as having worse PESQ score after speech enhancement. Studying the difference in PESQ and segmental SNR within all the sound classes, it is evident that the speech enhancement system does not seem to work equally well for all types. To get a better idea of this, the differences in performance for enhanced and noisy speech are plotted in figure 5.11. Classes *semivowels and glides* and *vowels* are seen to be among the most improved in terms of PESQ (5.11a), but the least improved in terms of segmental SNR (5.11b). The difference in improvement between the two test sets is largest for the *silence* and *affricates* classes for both PESQ and segmental SNR. *Affricates* in particular shows an extreme difference in the PESQ improvement, going from having the biggest increase for the seen noise to having the largest reduction for the unseen noise.

It is important to consider the reliability of the performance measures as they are used in this section. Clearly PESQ was designed to be used on spoken sentences and not long



(a) Average PESQ



(b) Segmental SNR

Figure 5.11: Improvement after enhancement in PESQ and segmental SNR

concatenated arrays of instances from a single speech sound class. Therefore it is hard to know whether or not the PESQ results make sense when used in this way. The segmental SNR is easier to understand, but it is a completely objective measure and probably less correlated with the subjective experience of speech quality. These measures on the individual sound classes have therefore not be considered when comparing different DNNs. Another issue is that the length and frequency of use of the different sound classes vary greatly. This means the length of the segment arrays used for calculating the performance will be very different depending on the sound class. For the results given here, for example, the test files contained about 6 minutes of concatenated *vowels*, but only 50 seconds of *affricates*.

5.9 Studying the learned weights of the DNN

Deep neural networks are often described as ”black box” models, since it can be very difficult to understand what is happening within the nodes of the network. Nonetheless, it can be interesting to try to interpret what the network has learned even though it is hard, or impossible, to fully understand it. One approach is to try to make sense of what the

different nodes of the first hidden layers are activated by in the input data. In the case of the described speech enhancement system the input is the log-magnitude spectrum of several frames concatenated to form one vector. This is illustrated in Figure 5.12 with 4 past and future context-frames included in the input for frame k.

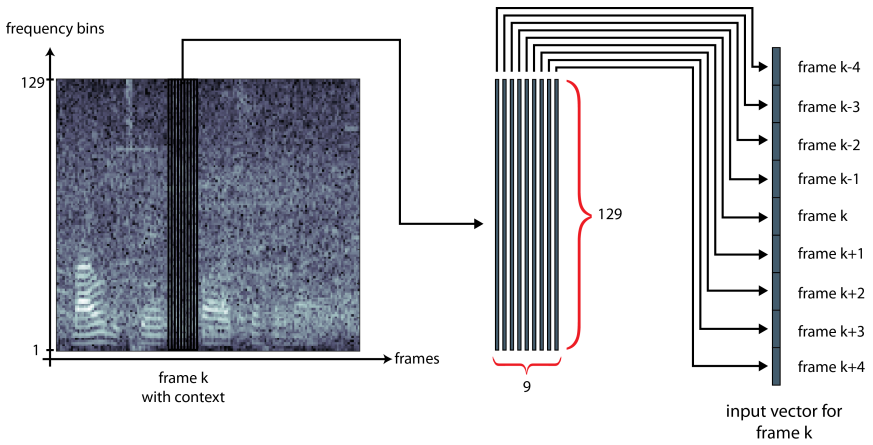


Figure 5.12: Choosing group of 9 frames from spectrogram to make input vector

In a densely connected MLP, every node in the first hidden layer will have a connection to every element of the input vector. Knowing that the input is a concatenation of several frames, the weights going to a single node can be separated based on which part of the input vector they are connected to. Taking the node j as an example, this means dividing the j'th column of the weight matrix into as many parts as there are frames in the input. These can be organized the same way the input data was in the spectrogram of Figure 5.12 to get an idea of what kind of features the first node "looks for" in the input data. This is illustrated in Figure 5.13.

To activate a sigmoid or rectifier neuron we need the weighted sum of input elements to be greater than the negative bias value. This means the weights should overlap with the part of the spectrogram that makes up the input in such a way that big positive weights are multiplied with positive values, and big negative weights with negative values. Therefore, by plotting the generated "weight images", scaled such that the biggest weight are represented by white pixels and the smallest (most negative) are represented by black, one can make some sense of what features the neural network has learned to recognize in the input data. In Figure 5.14 this is done for 30 nodes in the first layer of a well trained sigmoid DNN, using 9 frames in the input.

Studying Figure 5.14, it's seen that several of the nodes contain horizontal striped pattern that might indicate that they have learned to recognize the harmonics associated with voiced sounds. It's also interesting to see how some of these striped patterns span the width of the images, while others seem only focused on the middle frame. Since the middle frame of the input is the one the system is supposed to enhance it makes sense that more focus is placed on this rather than the context frames. When conducting experiments with non-symmetric context it was observed that this emphasis was shifted to the new

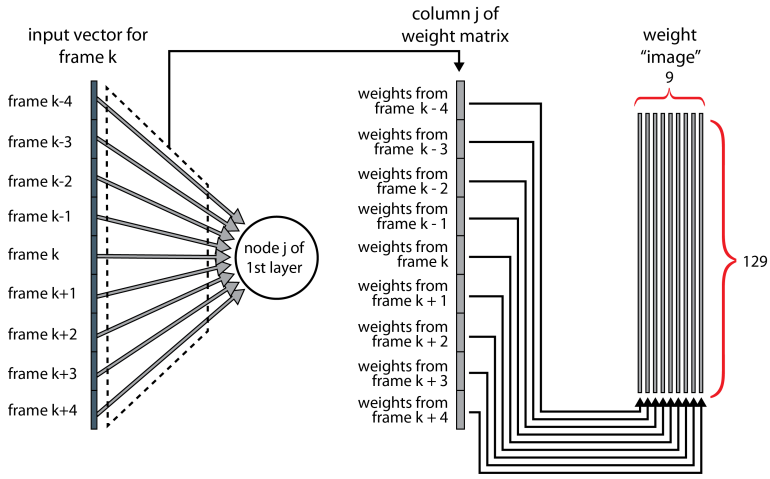


Figure 5.13: Dividing weight column into image based on which frames they are connected to in the input

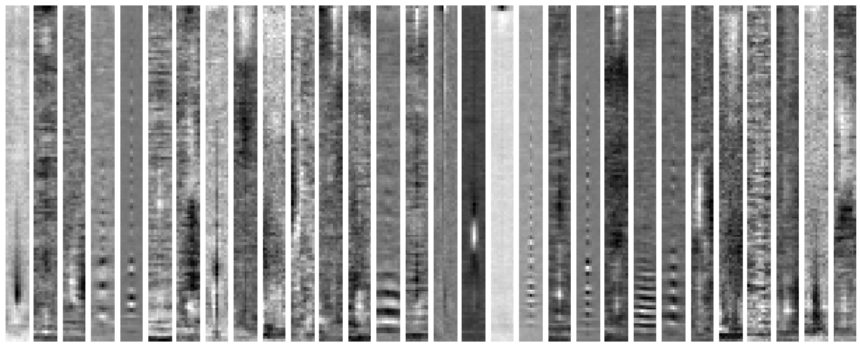
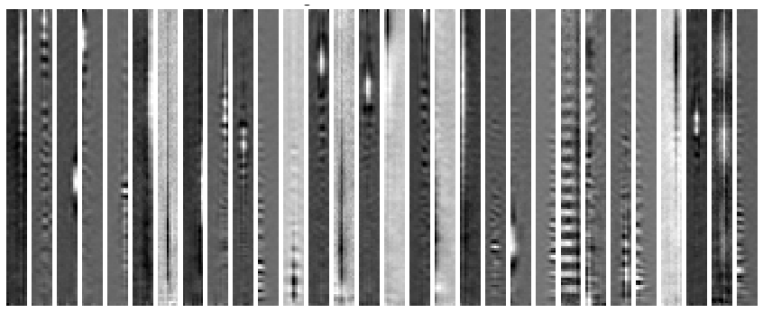


Figure 5.14: Incoming weights for nodes 361-390 in the first hidden layer. In all the images white or light grays mean values close to the maximum value for that node and black or dark grays are close to the minimum.

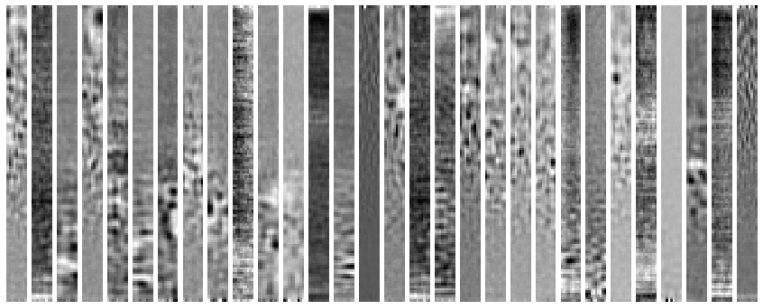
location of the current frame in the input. The information about where in the input vector the current frame is located comes from supervised training with the target frames, and is not something that can be learned in the unsupervised pre-training. For very deep networks with logistic neurons, which might experience vanishing gradients, it's possible that this information would not propagate all the way back to the first layer.

Assuming some of the neurons are trained to activate if certain patterns typical for speech appear in the spectrogram, maybe some neurons are purely "noise detectors" in much the same way. To get an idea whether this was the case, unsupervised learning using a shallow denoising autoencoder was performed for both clean input data and purely noisy data. The same dimension and scaling factors where used as for the complete DNN

presented earlier. The same kind of weight images were plotted for a subset of nodes for both autoencoders in Figure 5.15. Studying the weights for the speech autoencoder in Figure 5.15a we can see some similar patterns to what we saw in Figure 5.14, although less noisy and without the focus on the center frame that we saw earlier. The weights of the noisy autoencoder in Figure 5.15b look very different for the most part, although one can find some resemblance between a few of the weight images in Figure 5.14 and Figure 5.15.



(a) Weights learned for clean speech



(b) Weights learned for noise

Figure 5.15: Weigths for a subset of nodes learned by the denoising autoencoders using log-magnitude spectrum of frames of clean speech and noise.

A short experiment was conducted to explore whether these ”specialized” features in Figure 5.15 could be useful when training the speech enhancement system with noisy speech data. A DNN with two hidden layers containing 500 and 300 nodes respectively was used for the experiment. The 500 nodes of the first layer were split into two groups, 300 to be trained for speech detection and 200 for noise detection.Two denoising autoencoders with these dimensions of the hidden layers were trained using clean speech and noise respectively, in the same way as before. The weights and biases for the speech and noise autoencoders were then combined to form the first layer of the DNN. The other layers were pre-trained as denoising autoencoders using the preceding layer's activations for noisy input speech, before the complete DNN was fine-tuned using labeled data. For comparison an identical model was trained using the standard pre-training method (with denoising autoencoders instead of RBMs). When trained for the same number of epochs

this specialized first layer pretraining approach resulted in worse performance than for the standard setup. A possible explanation is that, because we use the logarithm of the magnitude-spectrum as input, the noisy speech features are not a linear combination of features from clean speech and noise, as the waveforms are. Therefore the specialized neurons might not be equipped to recognize the noise and clean-speech "parts" of the noisy speech features. The efforts were abandoned at that point, but the idea of utilizing the noise and speech data separately in some way could be interesting to look into more closely.

When it comes to the hidden layers not connected to the input, it is a bit more complicated to get an idea of what the weights have learned. Some work has been done in this field for computer vision tasks [45, 46], but none of those methods were applied here. Instead some focus was placed on the final layer, which produces the observed output and therefore might be easier to study. In the case of the speech enhancement system the final layer is doing the job of estimating the (normalized) clean log-magnitude spectrum of the frame being processed. The operations performed in this layer is illustrated in Figure 5.16.

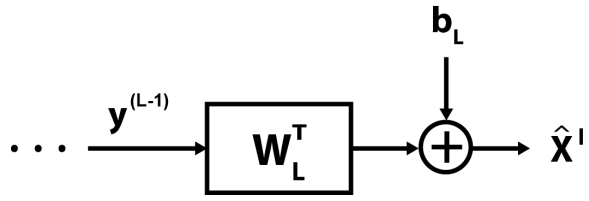


Figure 5.16: Last layer of the DNN

Denoting the transposed weight matrix by its column vectors $\mathbf{W}_L^T = [\mathbf{w}_{c1}, \mathbf{w}_{c2}, ..., \mathbf{w}_{cM}]$, and the activations of the previous layer by its elements $\mathbf{y}^{(L-1)} = [y_1^{(L-1)}, y_2^{(L-1)}, ..., y_M^{(L-1)}]^T$, the output of the DNN can be written as in Equation 5.1

$$\hat{\mathbf{X}}^{\mathbf{l}} = \sum_{i=1}^{M} y_i^{(L-1)} \mathbf{w}_{ci} + \mathbf{b}_L \tag{5.1}$$

To get the final estimate of the clean log-magnitude spectrum, $\hat{\mathbf{X}}'$, the output must be scaled back by element-wise multiplication (denoted by $.*$) with the standard deviation estimate, $\sigma_{\mathbf{N}}$, and addition of the mean estimate, $\mu_{\mathbf{N}}$, used in the pre-processing:

$$\hat{\mathbf{X}}' = \hat{\mathbf{X}}^{\mathbf{l}} .* \sigma_N + \mu_N \tag{5.2}$$

Inserting the sum from Equation 5.1 we get:

$$\hat{\mathbf{X}}' = \sum_{i=1}^{M} y_i^{(L-1)} \mathbf{w}_{ci} .* \sigma_N + \mathbf{b}_L .* \sigma_N + \mu_N \tag{5.3}$$

We see the final spectrum estimate has two parts. The first is a sum of the scaled column vectors $\mathbf{w}_{ci} .* \sigma_N$ weighted by the elements of the previous layer's output vector. The other is the term $\mathbf{b}_L .* \sigma_N + \mu_N$ which is independent of the input and therefore the

same for every frame estimate. In Figure 5.17 this last term is plotted together with μ_N, which is the mean log-magnitude spectrum of many frames of noisy speech, and the mean log-magnitude spectrum of many frames of clean speech. For this DNN at least, the scaled bias vector seems to have brought the mean vector, μ_N, closer to the mean of the target clean speech, μ_C, in addition to changing its shape somewhat.

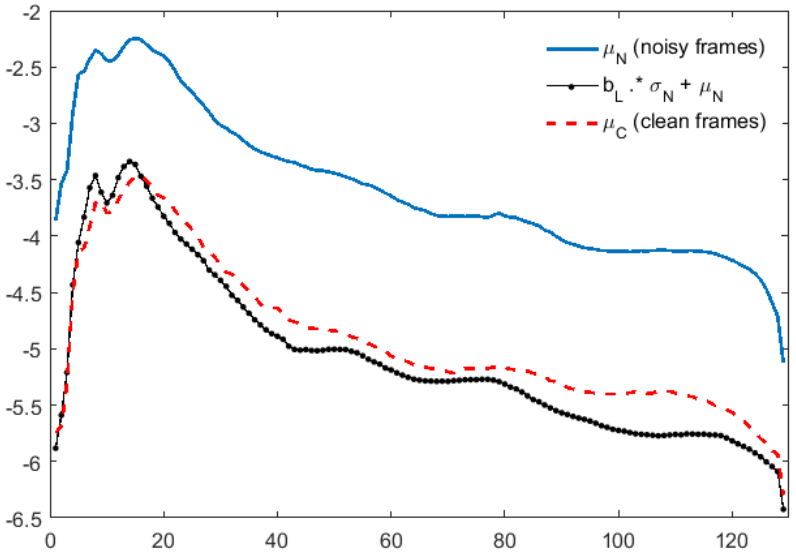


Figure 5.17: Scaled bias of last layer and mean log-magnitude spectrum for clean and noisy speech frames

The sum of scaled vectors in Equation 5.3 is the frame specific part of the output. When using sigmoid units in the hidden layers, the elements of the vector $\mathbf{y}^{(L-1)}$ will be somewhere in the range $[0, 1]$. One can think of the elements of this vector as choosing which columns vectors of the weight matrix will be used to estimate the frames clean log-magnitude spectrum. Thinking of it this way the scaled vectors $\{\mathbf{w}_{ci} . * \sigma_N\}_{i=1:M}$ must form some kind of "basis" for the clean speech log-magnitude spectrum (together with the input independent part). In Figure 5.18 some of these "basis vectors" are plotted, which gives the possibility of attempting to interpret the nature of the vectors. For example the top one, number 683 on the plot, might be used in the estimation of a voiced frame. Others, like number 677 and 678 might be used to introduce new formants to the spectrum, since they are mostly zero except for a single top centered around some frequency. Number 675 looks like it has the purpose of raising the high frequencies of the spectrum. To create silent frames in the output, maybe a combination of many vectors similar to number 679 and 680 is used to flatten the shape in Figure 5.17 and reduce its values. The dimension of the last layer in this example is 700, so there are many possible combinations that can be used to estimate the clean speech spectrum of the frame.

If the last layer is interpreted as described, it follows that the DNN up to that point performs the operation of finding the "coefficients" (elements of $\mathbf{y}^{(L-1)}$) that form the

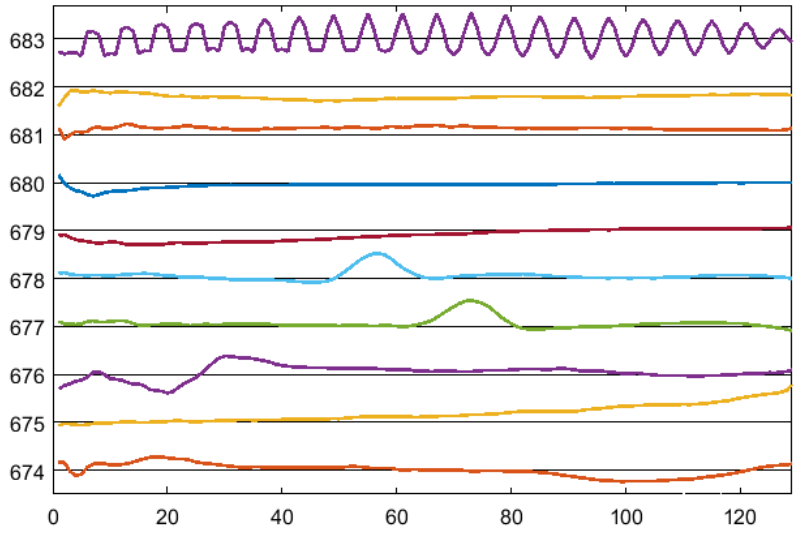


Figure 5.18: Some of the scaled column vectors of the last transposed weight matrix. Might be interpreted as basis vectors being combined to form the frame spectral estimate

closest approximation of the clean target vector. For example, if the nodes with weight patterns in Figure 5.14 that look like like voiced sounds are activated, this information might be propagated through the DNN resulting in a big value of the node being multiplied with a column vector like 683 in Figure 5.18. Thinking of the system in this way might be of use when trying to further improve the system. If nothing else, this was considered an interesting interpretation that gives a different insight into the operation of the speech enhancement system than the student had at the start of the master thesis work.

Chapter 6

Future work

6.1 Overview

This section contains some suggestions of future work and research based on the results presented in the master thesis. The content reflects what the student feels would have been most interesting to work with further if given more time.

6.2 Improving the hybrid system approach

Two hybrid systems combining the OM-LSA method with the DNN system were described in sections 4.6 and 5.6. The best results were found using a cascade of the methods with OM-LSA being applied before the DNN. In these experiments, the complete waveform was reconstructed in between the two systems. Since both methods use a STFT approach with enhancement of only the samples of the magnitude of the frames $|Y(k,l)|$, the waveform reconstruction after the OM-LSA method is not necessary. An alternative, more compact approach, is illustrated as a block diagram in Figure 6.1a. Note that in the figure, since the system is drawn as multiplying the spectral gain from the OM-LSA method to the squared magnitude (i.e. the energy spectral density) of the frames, the square of the gain is used. After the logarithm this amounts to a factor 2 which can be removed when scaling the features in the DNN pre-processing.

In 5.6 it was observed that the hybrid system gave a big improvement in MSE, but worse PESQ, compaed to using the DNN system alone. This result was interpreted as a result of the OM-LSA introducing some speech distortion that was hard for the DNN to undo. An alternative method of combining the OM-LSA method with the speech enhancement system, which could possibly improve the system, is to let the DNN itself ”decide” how to apply the spectral gain used in this method. Instead of using the multiplication in Figure 6.1a, the gain can be passed as input features together with the noisy spectral features to the DNN. The DNN cannot implement multiplication of the input elements with each other. However, since we apply the logarithm in the previ-

ous hybrid system, what the DNN in Figure 6.1a gets as input is a scaled verion of $\log(G(k,l)|Y(k,l)|) = \log(G(k,l)) + \log(|Y(k,l)|)$. So by providing both $\log(|Y(k,l)|)$ and $\log(G(k,l))$ to the DNN, the network can learn to apply the same operation as in the first hybrid system, but also other combinations. To see that this is the case, consider the activation of node j in the first hidden layer, receiving the weighted sum of the input features $\mathbf{y}^{(0)}$:

$$\{\mathbf{y}^{(1)}\}_j = f\left(\sum_{i=1}^{N} w_{ji}\{\mathbf{y}^{(0)}\}_i + w_{j0}\right) \tag{6.1}$$

Assume, for simplicity, that the input of the DNN in Figure 6.1a for frame l is comprised of just this frame's 129 frequency samples of OM-LSA pre-processed noisy features: $\mathbf{y}^{(0)} = [\log(G(1,l)|Y(1,l)|), ..., \log(G(129,l)|Y(129,l)|)]^T$. Then the activation can be written as in Equation 6.2.

$$\begin{aligned}\{\mathbf{y}^{(1)}\}_j &= f\left(\sum_{i=1}^{129} w_{ji}\log\left(G(i,l)|Y(i,l)|\right) + w_{j0}\right) \\ &= f\left(\sum_{i=1}^{129} w_{ji}\log\left(|Y(i,l)|\right) + \sum_{i=1}^{129} w_{ji}\log\left(G(i,l)\right) + w_{j0}\right)\end{aligned} \tag{6.2}$$

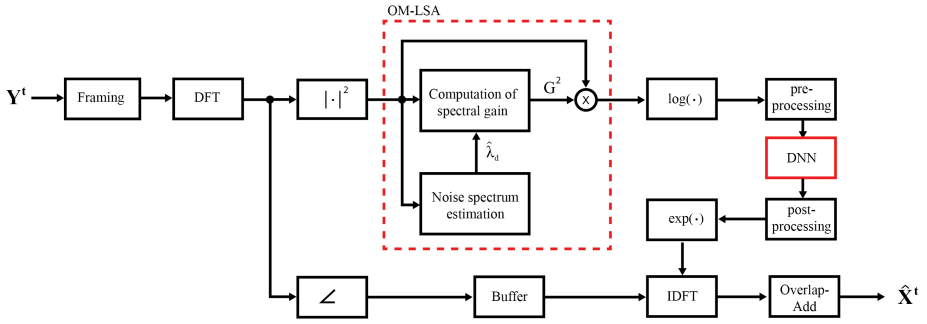
If the gains are concatenated with the log-spectral features the input vector will be $\mathbf{y}^{(0)} = [\log(|Y(1,l)|), ..., \log(|Y(129,l)|), \log(G(1,l)), ..., \log(G(129,l))]^T$. Then the activation can be written as:

$$\{\mathbf{y}^{(1)}\}_j = f\left(\sum_{i=1}^{129} w_{ji}\log\left(|Y(i,l)|\right) + \sum_{k=130}^{258} w_{jk}\log\left(G(k-129,l)\right) + w_{j0}\right) \tag{6.3}$$

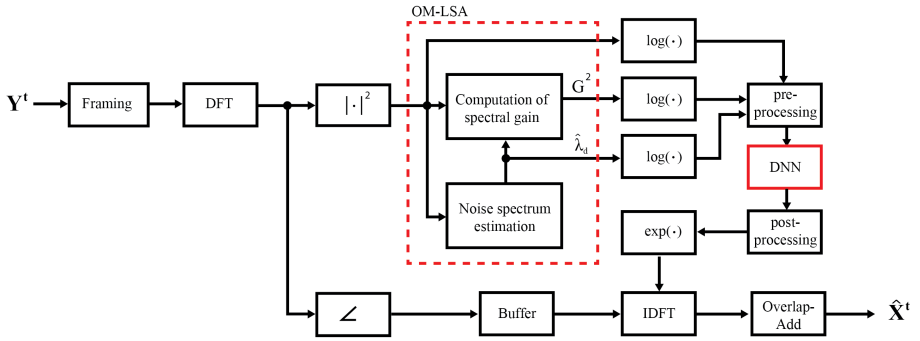
Comparing equations 6.2 and 6.3 it is clear that choosing the weights in the second sum to be equal to the weights of the first, $w_{jk} = w_{j(k-129)}$ for $k \in [130, 258]$, will make the activations in 6.2 and 6.3 the same. In practice the scaling in the pre-processing block will change this somewhat, but it still holds that the DNN in the second case, at least in theory, can learn to implement the OM-LSA exactly as it was used in Figure 6.1a. It might, however, also learn to better utilize information from this method.

In addition to the gain, the OM-LSA system produces a noise spectrum estimate $\hat{\lambda}_d$ that also could be useful to have as input to the DNN. In fact, one of the methods implemented for improving the DNN system in [1] is to provide the DNN with an estimate of the noise log-power spectrum. This was called noise aware training (NAT) and was shown in the paper to improve the speech enhancement system for most cases, even though the estimate used simply was an average of the first 6 frames (assumed to contain only noise). The OM-LSA estimate, $\hat{\lambda}_d$, uses a recursive averaging based on frames where speech is assumed to be absent, and is therefore updated throughout the enhancement. This adaptive-NAT might prove to be a better estimate than the the one in [1] for changing noise environments. An alternative hybrid system, that combines the proposed methods, is illustrated in Figure 6.1b. This might be a preferred solution to Figure 6.1a as the DNN here receives the non-processed magnitude of the frame, without possible distortions caused by the OM-LSA method, but it can still use the information provided by the OM-LSA method. The

downside is the increase in input layer size, which means more parameters to learn in the first layer. A possible trade-off would be to only include the 129 gain factors of the current frame and the 129 frequency bins of the noise estimate, meaning the input size would be $N = N_{fr}129+258$ with N_{fr} again being the size of acoustic context including the current frame.



(a) Compact version of hybrid system 2 (see Sec. 4.6)



(b) Alternative combination of the OM-LSA and DNN method

Figure 6.1: Block diagrams for hybrid OM-LSA and DNN speech enhancement systems.

6.3 Modifications of the existing system

There are many ways in which the speech enhancement system can be modified. The first thing would be to implement the noise aware training (NAT) that was presented in [1], that because of limited time was not included in this project. Training the system to also provide a noise estimate for the frame, or simply using the difference (in the the linear

spectral domain) of estimated output and noisy input, an adaptive NAT could possibly be implemented. This would need some changes in the training procedure, but might give some performance gains for changing noise environments over the NAT proposed in [1].

Moving beyond what is already done by the authors of [1], it could be interesting to test a recurrent neural network setup of the system. That is, a neural network containing directed cycles instead of strictly the feed-forward structure utilized in this project. Using a single frame as input, information from previous frames could then be ”stored” in the network. This means, in theory, that the network automatically can leverage past useful information without explicitly including a left context. Maybe a system like this can even learn to keep a running estimate of the noise, replacing the need of NAT. Recurrent models are often much harder to train than feed-forward network, but special versions like the so called Long Short Term Memory (LSTM) network have used successfully for many applications [12].

Starting from the interpretation of the weights of the last layer as forming a collection of basis vectors, presented in Section 5.9, it could be interesting to see if a robust and general basis could be explicitly trained, introducing a form for pretraining of the last layer. As mentioned in that section, it could also be interesting to see if the noise and speech data could be utilized separately for the training, in some way.

6.4 Moving towards a real-time implementation

In today’s society, where communication via speech or video on digital devices has become almost as widespread as face-to-face conversations, it is obvious that effective real-time speech enhancement systems are of great interest. Having a complex speech enhancement system running on a smart-phone or other device demands computational efficiency and minimized delay. Limiting delay by removing the right side context of the frames, with a thorough examination of how this affects performance, is an obvious first step in this regard. It is probable that much work is needed before the system is applicable for this scenario and so it is an interesting direction in which to continue the research.

Chapter 7

Conclusion

The task for this master thesis was to implement and evaluate a DNN based approach to speech enhancement as described in the paper [1]. Attaining the performance reported in [1] was not considered realistic so no direct comparison was performed with the results presented there. Rather the aim was to examine and understand in more detail the effect of the proposed techniques on the performance of the speech enhancement system. Among these, dropout has been implemented as a regularization technique during training, global variance equalization post-procesing was introduced to reduce a problem of over-smoothing, and more data combined with new types of noise, was used to increase performance on speech corrupted by unseen noise. Some changes outside those discussed in [1], including a hybrid OM-LSA DNN system, have been proposed and implemented. In addition, some novel evaluations of the system have been performed, examining performance on sub classes of speech as well as attempting to interpret the learned parameters of the model.

Implementing many of the techniques, and training on a data set containing 20 hours of speech corrupted by AWGN and 5 noise types from the Aurora2 database resulted in two variations of the system giving superior performance over the traditional OM-LSA method in terms of PESQ. From listening to a subset of test files it was harder to conclude a clear improvement for the DNN over OM-LSA, as both methods give poor performance on noisy speech with low SNRs.

Bibliography

[1] Y. Xu, J. Du, L.-R. Dai, and C.-H. Lee, "A regression approach to speech enhancement based on deep neural networks," *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, vol. 23, no. 1, pp. 7–19, 2015.

[2] H.-G. Hirsch and D. Pearce, "The aurora experimental framework for the performance evaluation of speech recognition systems under noisy conditions," in *ASR2000-Automatic Speech Recognition: Challenges for the new Millenium ISCA Tutorial and Research Workshop (ITRW)*, 2000.

[3] A. W. Rix, J. G. Beerends, M. P. Hollier, and A. P. Hekstra, "Perceptual evaluation of speech quality (pesq)-a new method for speech quality assessment of telephone networks and codecs," in *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, vol. 2, pp. 749–752, IEEE, 2001.

[4] I. Cohen and B. Berdugo, "Speech enhancement for non-stationary noise environments," *Signal processing*, vol. 81, no. 11, pp. 2403–2418, 2001.

[5] O. K. Melve, "Machine learning approach to speech enhancement." Project work leading up to Master thesis.

[6] M. H. Johnsen, "Artificial neural networks (ANN) used in automatic speech recognition (ASR)." Lecture notes [PDF slides] from course TTT16 at NTNU.

[7] "Demos for paper published on the ieee signal processing letters: An experimental study on speech enhancement based on deep neural networks." `http://home.ustc.edu.cn/~xuyong62/demo/SE_DNN.html`. Audio examples for paper [31].

[8] R. P. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Mag.*, pp. 4–22, 1987.

[9] R. D. Reed and R. J. Marks II, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. The MIT Press, 1999.

[10] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?," in *Proc. International Conference on Computer Vision (ICCV'09)*, IEEE, 2009.

[11] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)* (G. J. Gordon and D. B. Dunson, eds.), vol. 15, pp. 315–323, Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[12] Y. B. Ian Goodfellow and A. Courville, "Deep learning." Book in preparation for MIT Press, 2016.

[13] G. E. Hinton, "The backpropagation algorithm." `https://class.coursera.org/neuralnets-2012-001`. Lecture notes [Powerpoint slides] from course "Neural Networks for Machine Learning" at Coursera.org.

[14] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[16] Y. Bengio, "Learning deep architectures for AI," vol. 2, no. 1, 2009. Also published as a book. Now Publishers, 2009.

[17] G. E. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[18] G. E. Hinton, "A practical guide to training restricted boltzmann machines," 2010.

[19] A. Mohamed, G. Dahl, and G. E. Hinton, "Acoustic modeling using deep belief networks," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 20, no. 1, pp. 14–22, 2012.

[20] A. Fischer and C. Igel, "Empirical analysis of the divergence of gibbs sampling based learning algorithms for restricted boltzmann machines," in *ICANN (3)*, vol. 6354 of *Lecture Notes in Computer Science*, pp. 208–217, Springer, 2010.

[21] G. E. Hinton and S. Osindero, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, p. 2006, 2006.

[22] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?," vol. 11, pp. 625–660, Feb. 2010.

[23] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, pp. 3371–3408, Dec. 2010.

[24] J. Martens, "Deep learning via hessian-free optimization," in *ICML*, pp. 735–742, Omnipress, 2010.

[25] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (S. Dasgupta and D. Mcallester, eds.), vol. 28, pp. 1139–1147, JMLR Workshop and Conference Proceedings, May 2013.

[26] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[28] I. Cohen, "Noise spectrum estimation in adverse environments: Improved minima controlled recursive averaging," *Speech and Audio Processing, IEEE Transactions on*, vol. 11, no. 5, pp. 466–475, 2003.

[29] I. Cohen, "Free software implementation of the om-lsa speech enhancement method." `http://webee.technion.ac.il/people/IsraelCohen/`. Homepage of Israel Cohen.

[30] TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU, *Perceptual evaluation of speech quality (PESQ): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs*, 2001.

[31] Y. Xu, J. Du, L.-R. Dai, and C.-H. Lee, "An experimental study on speech enhancement based on deep neural networks," *Signal Processing Letters, IEEE*, vol. 21, no. 1, pp. 65–68, 2014.

[32] F. Xie and D. Van Compernolle, "A family of mlp based nonlinear spectral estimators for noise reduction," in *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, vol. ii, pp. II/53–II/56 vol.2, 1994.

[33] J. H. L. Hansen and B. L. Pellom, "An effective quality evaluation protocol for speech enhancement algorithms," in *IN PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SPEECH AND LANGUAGE PROCESSING*, pp. 2819–2822, 1998.

[34] *TIMIT documentation*, 1990.

[35] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[36] "Keras: Deep learning library for theano and tensorflow." `http://keras.io/`, 2008. [Online].

[37] "Lasagne documentation." `http://lasagne.readthedocs.io/en/latest/`. [Online].

[38] "Blocks documentation." `http://blocks.readthedocs.io/en/latest/`. [Online].

[39] R. B. Palm, "Prediction as a candidate for learning deep hierarchical models of data," Master's thesis, Technical University of Denmark, DTU Informatics, E-mail: reception@imm.dtu.dk, 2012.

[40] "Deep learning tutorial." http://deeplearning.net/tutorial/. [Online; accessed several times between 15.Jan.-10.June 2016].

[41] M. Denil, "A theano implementation of hinton's dropout.." https://github.com/mdenil/dropout. [Online; accessed 08. April 2016].

[42] "P.862 : Revised annex a – reference implementations and conformance testing for itu-t recs p.862, p.862.1 and p.862.2." https://www.itu.int/rec/T-REC-P.862-200511-I!Amd2/en. Source code for PESQ.

[43] "Demos for paper published on ieee/acm transactions on audio, speech, and language processing: A regression approach to speech enhancement based on deep neural networks." http://home.ustc.edu.cn/~xuyong62/demo/SE_DNN_taslp.html. Audio examples for paper [1].

[44] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, *et al.*, "On rectified linear units for speech processing," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 3517–3521, IEEE, 2013.

[45] D. Erhan, Y. Bengio, A. Courville, and P. Vincent, "Visualizing higher-layer features of a deep network," Tech. Rep. 1341, University of Montreal, June 2009. Also presented at the ICML 2009 Workshop on Learning Feature Hierarchies, Montréal, Canada.

[46] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," *arXiv preprint arXiv:1506.06579*, 2015.

Appendix

A Studying the effect of the context frames on performance

Considering possible real time use of the speech enhancement system it was interesting to see how limiting the right context affected the performance. Ideally the past context would be the most important, as this would limit the need for buffering future frames to process the current, and thereby reduce the delay. A DNN with 5 frames in the input layer and two hidden layers with 500 and 300 nodes were trained with about three hours of noisy and clean speech data. One standard net was trained where the target frame was the clean version of the frame in the center of the input. This gives a *symmetrical* context with 2 frames to the left and 2 to the right of the frame being enhanced. A second DNN was trained with the target shifted one frame to the right. This gave a *skewed* context, with 3 frames to the left and a single to the right. A third DNN was trained with the target frame shifted another frame to the right, meaning the last of the five frames was the one being enhanced. This meant a *one-sided* context with 4 frames to the left and 0 frames to the right of the frame to be enhanced. The results for shifting the context in a 5 frame input is given in table 1. It seems from this experiment that using a skewed context, with 3 left and 1 right frame, doesn't affect the PESQ results too much. The one-sided case, on the other hand, does seem to perform worse than the symmetrical case in terms of PESQ.

Table 1: MSE and average PESQ for two identical DNNs trained with 5 frames in the input layer chosen to be symmetric (left = 2, right = 2), skewed (left = 3, right = 1) or one-sided (left = 4, right = 0)

	Seen noise		**Unseen noise**	
Context	**MSE**	**PESQ**	**MSE**	**PESQ**
symmetric	**33.66**	**2.37**	55.04	**2.36**
skewed	34.05	2.36	56.78	**2.36**
one-sided	35.46	2.31	55.75	2.33

A bigger network with 9 frames in the input and three hidden layers with 700 neurons each was also trained using about 10 hours of noisy and clean speech data. For this only two cases were tested: the symmetrical one and a skewed version with 6 frames of left context and 2 to the right. The results for the bigger DNN with 9 frames in the input is given in table 2. With the exception of the MSE for the unseen noise, shifting the context does not seem to hurt the performance, and actually seems to improve it somewhat. The improvement is, however, very small and can probably not be considered significant. As mentioned, [1] found that increasing the context from 2 frames on either side (5 in

total) to 4 frames (9 in total) lead to better performance. These limited result seem to indicate that keeping the 2 frames of right sided context and simply increasing the left sided (past) context might lead to equivalent performance as the symmetrical increase. More experiments are needed before concluding if this is the case.

Table 2: MSE and average PESQ for two identical DNNs trained with 9 frames in the input layer chosen to be symmetric (left = 4, right = 4) or skewed (left = 6, right = 2)

	Seen noise		**Unseen noise**	
Context	**MSE**	**PESQ**	**MSE**	**PESQ**
symmetric	33.13	2.51	**56.97**	2.45
skewed	**32.95**	**2.53**	60.08	**2.47**

B The good, the bad and the average

SNR of noisy files and PESQ of enhanced files for the files having the worst, best and closest to the average PESQ score are given in tables 3 and 4

Table 3: PESQ scores and SNR for best and worst files from test set with seen noise

Global SNR	0	5	10	15
Best PESQ	3.06	3.21	3.48	3.66
SNR of file	5.11	5.94	15.05	20.16
Worst PESQ	0.98	1.41	2.06	2.35
SNR of file	-10.80	-6.95	-1.33	4.37
Average PESQ	2.11	2.47	2.81	3.07

Table 4: PESQ scores and SNR for best and worst files from test set with unseen noise

Global SNR	0	5	10	15
Best PESQ	3.10	3.26	3.53	3.64
SNR of file	5.26	10.25	15.23	20.23
Worst PESQ	0.98	1.44	1.53	2.18
SNR of file	-12.60	-4.02	-1.51	2.93
Average PESQ	2.06	2.36	2.71	2.97

C Link to audio demos of the ReLU based DNN

Url containing the audio files from [7] enhanced with the ReLU-DNN based system
`https://www.dropbox.com/sh/hodv3x08jmrfx05/AAAoFLwf4o_qFL9QmH4n46MFa?dl=0`

Url containing the audio files from [7] enhanced with the ReLU-DNN based system

https://www.dropbox.com/sh/ugecp32e334rueo/AABRDQ7T7P38ma_J936j3stqa?dl=0