THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

A Neural Network Based Implementation of Non-Negative Matrix Factorization for Targeted Speech Denoising

by
Matthew Smarsch

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

December 12, 2017

Professor Sam Keene, Advisor

THE COOPER UNION FOR THE
ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Dean, School of Engineering Date

Prof. Sam Keene, Thesis Advisor Date

Abstract

The cocktail party problem is the task of extracting a speaker's voice from a noisy environment containing the voices of other speakers. The human brain does this quite well in most cases, but hearing impaired individuals often struggle to hear others due to background noise in a crowded bar or restaurant. Previous research efforts in the fields of signal processing and machine learning have presented frameworks to solve this task in supervised and unsupervised environments. One such implementation is Non-Negative Matrix Factorization (NMF).

NMF is an algorithm for creating an approximation of a non-negative matrix into the product of two smaller, non-negative matrices. NMF is a linear dimensionality reduction technique that is often applied in recommendation systems, text mining, spectral data analysis, and audio analysis. This thesis expands upon previous work demonstrating the benefits of using a neural network implementation of the NMF algorithm for the tangential problem of source separation. We present an end-to-end framework for approaching a semi-supervised case of the cocktail party problem. In the semi-supervised case, we attempt to extract a target speaker's speech signal from a noisy mixture signal only with the knowledge of the presence of noise, not the type of noise in the mixture. The framework presented in this thesis transforms a raw audio signal into an non-negative magnitude spectrogram

and uses the NMF algorithm to extract the target speaker's contribution to the mixture spectrogram. The approximation of the speaker's spectrogram is inversely transformed into a raw audio signal using the phase of the noisy mixture. We present results demonstrating improved intelligibility of a target speakers voice using a neural network based NMF implementation in a semi-supervised case of the cocktail party problem.

Acknowledgments

This thesis would not have been possible without the close advisement of Professor Sam Keene. From the inception of my research into the Cocktail Party Problem during Senior Project to this thesis, he has been an extraordinary resource, mentor, and friend.

I would like to thank Frank Longueira for his help with my research since Selective Hearing and for his suggestions during our meetings with Professor Keene.

I would next like to thank Chris Curro for his valuable guidance as a friend and professor.

Thank you to all of the faculty and employees of The Cooper Union and to all my friends, classmates, and teammates for some of the best, toughest, and most rewarding years of my life.

Thank you Mom, Dad, and Nick for all of the love and encouragement you have shown me. You guys are simply the best family I could have ever asked for.

Last, but not least, thank you, Gina, for all of your love, support, and patience. It wasn't always easy, but you were always there to support me when I needed it the most. Thank you.

Contents

Chapter 1

Introduction

One of the largest issues facing hearing impaired individuals in their day to day lives is accurately recognizing speech in the presence of background noise [Healy et al., 2013]. While hearing aids do a good job of amplifying sound, they do not do enough to increase speech intelligibility. This does not cause a problem in quiet environments, but standard hearing aids will tend to underperform in the presence of noise, particularly in the presence of non-stationary noise like one might find in a crowded bar or restaurant.

While people without hearing impairments usually have no trouble focusing on a single speaker in the presence of multiple interfering voices, it is a much more difficult task for people with a hearing impairment [McDermott, 2009]. The problem of picking out on persons' speech in an environment with many speakers was dubbed the cocktail party problem [Cherry, 1953]. The paper asserts that humans are normally capable of separating multiple speakers and focusing on a single one [Cherry, 1953]. However, hearing impaired individuals may have issues when it comes to performing this same task. The cocktail party problem has been

approached using several different techniques, such as using microphone arrays, monaural algorithms, and computation auditory scene analysis (CASA) [Healy et al., 2013].

Modern hearing aids incorporate the microphone array strategy. They use beamforming to amplify sound coming from a specific direction and attenuate the sound coming from elsewhere. This technique comes with several drawbacks. In order for it to work, the speech the user is trying to focus on must come from a different direction than the noise. Difficulty will also arise when the source of the target speech changes location [Healy et al., 2013]. Monaural algorithms use a single microphone and are not dependent upon the location of the speech source and the noise. These algorithms attempt to estimate the clean speech signal after a statistical analysis of the speech and noise. These methods include spectral subtraction, Wiener filtering, and mean-square error estimation. Spectral subtraction removes the estimated power spectral density of the noise signal from the power spectral density of the noisy speech. Wiener filtering estimates the clean speech signal from the speech and mixture spectrum ratios. Mean-square error estimation estimates clean speech by modeling the speech and noise spectra as statistically independent Gaussian random variables. While monaural strategies can improve performance of automatic speech recognition systems and increase Signal-to-Noise Ratio (SNR), they have not been able to increase speech intelligibility for human listeners so far [Healy et al., 2013]. CASA has some promising results using ideal binary time-frequency masks to hide regions of the mixture where the SNR is below a certain threshold. However, this method of separating speech from noise requires prior knowledge of both, as the mask is created based off the relative

strengths of the speech signal and the noise [Healy et al., 2013].

This thesis seeks to build off of previous research into the performance of a technique known as Non-Negative Matrix Factorization (NMF) for source separation and speech denoising in supervised and unsupervised environments [Mohammadiha, 2017]. NMF has many applications ranging from recommendations to dimensionality reduction. As we will demonstrate in the next chapter, NMF can be applied in order to model the way that a particular person speaks and recognize the portion of a noisy signal that was contributed by a target speaker. We leverage the conveniences afforded by a neural network implementation of the NMF algorithm as demonstrated in prior research [Smaragdis, 2017].

We begin our research by presenting background information in Chapter 2 on Non-Negative Matrix Factorization, neural networks, and the benefits of a neural network based NMF implementation. In Chapter 3, we formally introduce the problem statement, the data being used, and the steps that were taken in our approach to the Cocktail Party Problem. Chapter 4 discusses the various neural network architectures and training paradigms that were investigated during this research. In Chapter 5, we introduce and discuss the results of our research by comparing the quality of the denoised speech signal using the Mean-Squared Error (MSE) as an evaluation metric. Finally, in Chapter 6 we make our conclusions and propose avenues for future research.

Chapter 2

Background

2.1 The Cocktail Party Problem

The cocktail party problem is the task of removing interference from human speech. In a crowded bar or restaurant, humans have a strong ability to focus on the speech of a desired target while ignoring sources of interference. Typically, this interference can be attributed to the voices of others and the vibrations caused by the acoustical interactions of sound sources and the environment. In these settings, various sounds and voices overlap in both frequency and time. This overlap causes classical signal processing techniques for speech denoising to provide poor performance. Statistical and signal processing methods that have been applied to the cocktail party problem include independent component's analysis, principal component's analysis, and auditory scene analysis [Bee, 2008].

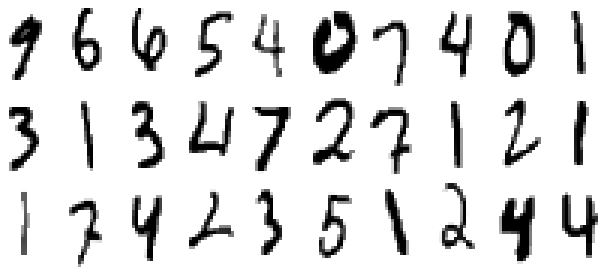

Figure 2.1: MNIST handwritten digit dataset [Deng, 2012]

2.2 Machine Learning

Machine learning is the process of finding patterns in data that enable a computer to perform a given task without being explicitly programmed to do so. The problems that we deal with in machine learning can be grouped into two classes: classification and regression. A classic machine learning classification problem is that of recognizing handwritten digits such as from the MNIST dataset shown in Figure 2.1 [Deng, 2012].

Figure 2.1 illustrates the complications to what seems like a simple problem at first glance. Specifically, we see that the same number can be written many different ways, in many different styles of handwriting. If we were to try and create a set of rules and patterns to classify the digits, we would soon find that the number of rules required to exhaustively solve this classification problem balloons to an infeasible amount.

Machine learning algorithms combat this problem much in the same way that humans first learn how to recognize hand written digits. Using a training set of examples, a machine learning model can be algorithmically tuned to generalize from this training set. We can then use this trained model to classify new examples

from a testing set.

2.3 Regression

The class of machine learning techniques applied to the cocktail party problem in this thesis is an example of the second class of machine learning problems: regression. Unlike in classification problems where we label a real valued input vector with a value from a discrete set of targets (the digits 0-9 in our example), regression problems have continuous outputs. In this thesis, we aim to take a real-valued input (a noisy speech signal) and output a clean/de-noised version of this speech signal.

A classic regression technique is known as linear regression. For an input vector of k independent observations, x_i, we would like to determine a scalar dependent variable, y. In linear regression, we assume a linear relationship betwen the input and the output. We formally define this relationship as:

$$y_i = w_1 x_{i1} + w_2 x_{i2} + ... + w_k x_{ik} + b_i, \tag{2.1}$$

$$\boldsymbol{y} = \boldsymbol{x^T w} + \boldsymbol{b} \tag{2.2}$$

where $i = 1, 2, ..., k$, $\boldsymbol{x^T}$ contains rows of input observations, $\boldsymbol{w}$ is a vector of weights/slopes, and $\boldsymbol{b}$ is a vector of biases/intercepts. This simple linear regression formulation will provide the foundation for our definition of neural networks in Section 2.4.

2.4 Neural Networks

Neural networks are a class of machine learning algorithms for performing classification and regression. Inspired by the neural architecture of the human brain, neural networks have been shown to produce state of the art results across many fields and applications. As described in linear regression, input vectors to neural networks are multiplied with a set of weights and summed with a bias vector to create a latent representation of the input data. Unlike linear regression, neural networks use various activation functions to introduce non-linearities into the model. The latent representation is passed through one of these activation functions (See Section 2.6). This allows neural networks to learn non-linear mappings and interactions of input features to better inform their output decisions. This is an example of a single layer of a neural network. As we will discuss, neural networks can be designed with one or (sometimes many) more layers. Neurons, activation functions, and network architectures are discussed below.

2.5 Neurons

Much like in our own brains, neurons are considered the basic building blocks of neural networks. A block diagram of a neuron is shown in Figure 2.2. The neuron is comprised of three components:

1. Weights
2. Accumulator
3. Activation Function

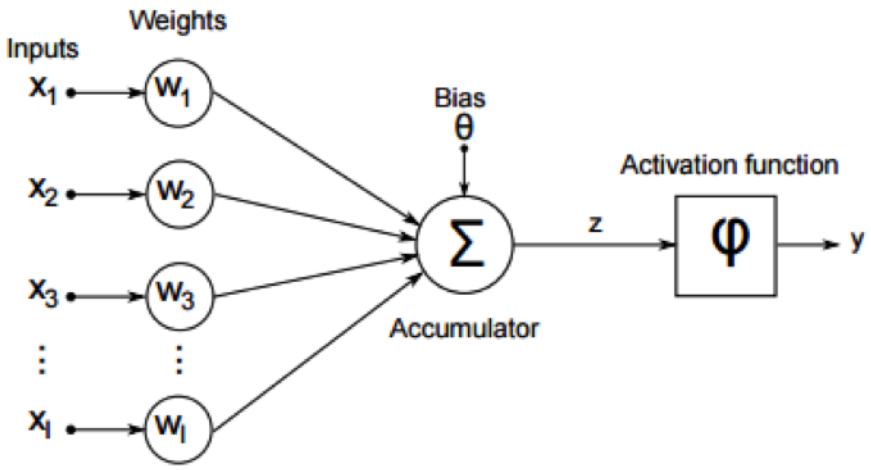

Figure 2.2: A block diagram of a neuron. $x_{1,\ldots,l}$ are the inputs to the neuron, $w_{1,\ldots,l}$ are the weights, and Φ is the activation function. The input to the activation function is z and the output is y [Jacobson et al., 2015].

The inputs to the neuron, $x_{1,\ldots,l}$, are either the input vectors of the problem or the output vector of a previous layer within the neural network. The weights, $w_{1,\ldots,l}$, are applied to the inputs of the neuron to perform scaling. This scaling can be thought of as determining which parts of the input are important within the context of that neuron, magnifying important input features and removing insignificant features. As we train our neural network with training data, these weight parameters are tuned to allow the network to adapt to the problem and training data. By determining the proper scaling of input features, we are able to better inform the model of the appropriate output decision to make given some input data.

After the inputs to the neuron are scaled by the weights, they are summed with a bias/intercept vector in the accumulator. So far, this mimics the linear regression process described in Section 2.3. Where the concept of a neuron deviates

from the concept of linear regression is in the introduction of an activation function.

2.6 Activation Functions

The ability to scale inputs according to their relevance to a desired output is useful if the desired output is a linear function of the inputs. In practice, there are often many non-linear interactions between the input features that cannot be modeled under this paradigm. By passing the output of the accumulator through a chosen activation function, we can introduce non-linear effects into our model. For example, in a binary classification problem, the activation function used in the output layer of a neural network might be a step function. Since the step function maps all inputs to either 0 or 1, this guarantees that the output of the network is a 0 or 1 binary classification. For purposes that will become apparent from our discussion of the formulation of NMF in Section 2.10, we will restrict our discussion of activation functions to functions with non-negative outputs. Three such activation functions that were investigated over the course of this project are:

Rectified Linear Unit (RELU):

$$g(x) = max(x, 0) \tag{2.3}$$

Softplus:

$$g(x) = log(1 + e^x) \tag{2.4}$$

Absolute Value:

$$g(x) = |x| \tag{2.5}$$

where g is the activation function, Φ.

2.7 Network Architectures

A neural network's architecture, or its arrangement of the neurons described in Section 2.5, vastly impact a networks suitability to a given problem. Neural networks are typically represented by directed graphs that can be cyclic or acyclic and fully or partially connected. Fully connected graphs are graphs in which each neuron of a given layer has a direct connection to each neuron of the next layer and so on. There are various types of network architectures that are categorized by their internal neuron connections between layers such as Feedforward, Convolutional, and Recurrent Neural Networks.

The Feedforward Neural Network (FNN) is the simplest neural network structure and is the one used in this thesis. As its name implies, the FNN is a fully connected, acyclic graph which only passes information in a single direction: forward. Figure 2.3 shows a two layer FNN with a single hidden layer.

The configuration of the network, from the number of hidden layers to the number of nodes in each layer, impacts the performance of the network for a given task. Each hidden layer of a network can be thought of as a feature extractor from the output of the previous layer or the input. The input to the first hidden layer is in the input space, as the number of inputs will be the same as the number of features. However, the output of the hidden layer will be represented in a feature space. The more units in a hidden layer there are, the more complex the decision boundary can be and the more effective the network can be in seperating features. However, adding hidden layers and extra units will not necessarily improve the

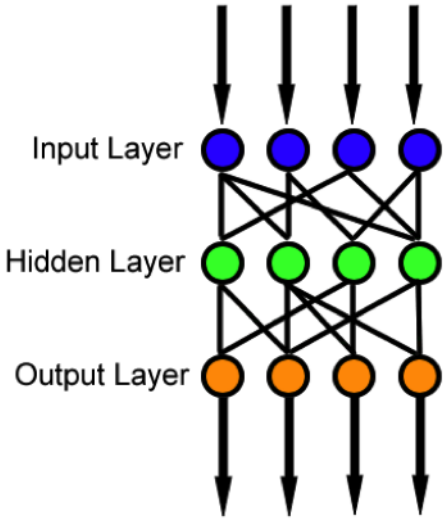

Figure 2.3: A two-layer Feedforward Neural Network [Auer et al., 2008]

performance of a network. The number of hidden units in a network is related to the networks ability to generalize. The risk of overfitting increases significantly if there are too many units relative to training data, while the risk of poor classification increases if there are too few. Also, the more hidden units there are in a network, the longer it takes to train and the greater a computational load is put on the processor [Jacobsen et al., 2015]. Adding hidden layers to a network may also have negative effects on its performance. Networks with several hidden layers have a higher likelihood of getting caught in a local minima during training and not reaching the optimal decision boundary [Hinton et al., 2006].

2.8 Training

In order for the network architecture described in Section 2.7 to be able to solve complex problems, it has to learn about the problem we are trying to solve. As we

previously mentioned, the way that we are able to teach our network is by adjusting the weights that scale the inputs to the neuron during a training phase. There are traditionally two types of training: Supervised and Unsupervised. In a supervised training scenario, we are providing our neural network with an observed set of inputs and a corresponding set of outputs. Since we have the truth labels at training time, we can compare the network's output to the true target using an error function. Common error functions are the Sum of Squares function and Cross Entropy function. By minimizing the error function of our network, we can algorithmically update the weights of the network to provide better target estimates based on the training data that we have observed. Weight updates are performed using an algorithm known as the back-propagation algorithm, which assigns responsibility for the observed error to particular network components/weights. The weights are finally adjusted using the gradient descent method [Jacobsen et al., 2015].

In the unsupervised scenario, we do not have the truth labels at training time. In many cases, such as clustering, we are simply training the neural network to learn the underlying patterns within the training data so that it can be separated into different clusters or classifications.

In the case of the cocktail party problem, the unsupervised case is when we have noisy target speaker data and attempt to train a network to denoise the noisy speech without any knowledge as to what type of noise is being applied to the clean speech. In the supervised case of our problem, we have knowledge of the type of noise being used on the clean speech and can train our model in the presence of such noise. One unconventional training scenario that we will explore in this thesis is the semi-supervised case. In the semi-supervised training scenario, we

have knowledge that there will be noise applied to our speech, but we do not have knowledge as to what type of noise this will be. Instead, we train using a general noise model that does not contain the noise model that will be used in testing.

2.8.1 Regularization

One common modification that is made to the loss function of a neural network is the addition of a regularization penalty. Regularization penalties are often used to prevent models from overfitting to the training data. By adding a penalty on the size of the weight parameters, we can ensure that the model remains sensitive to new input data and will hopefully generalize well to new data. Regularization can also be used to constrain the parameters of our model to behave in a particular way. As we will discuss, one such constraint is a sparsity constraint which encourages sparsity in our parameter matrices. This type of regularization is known as L1-Regularization [Bagnell and Bradley, 2009].

2.9 Autoencoder Networks

An autoencoder is a specific Feedforward Neural Network architecture formulation in which the goal of the network is to reconstruct its input at its output. The input and output layers are the same size and are connected by hidden layers. Figure 2.4 shows an autoencoder with three fully connected, hidden layers.

The autoencoder creates an output reconstruction by learning an efficient, smaller representation of its input. This stage of the autoencoder is known as the encoder and is comprised of one or more hidden layers. Once the encoder has mapped the original input into a lower dimensional space, a reciprocal procedure is

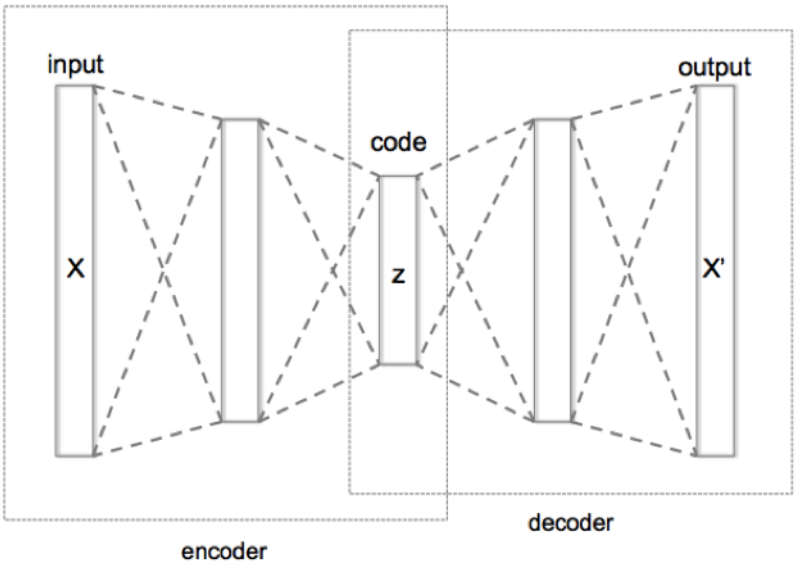

Figure 2.4: A fully-connected autoencoder with 3 hidden layers [Bengio, 2009]

performed in the decoder portion of the network. The decoder uses an architecture symmetric in layer topology to the encoder to reconstruct the input from the efficient mapping in the code layer. In the case of the autoencoder, the output is compared to the input through the use of an error function as mentioned in Section 2.8. We have seen in our prior research that autoencoders tend to perform well for audio processing tasks and specifically the task of denoising speech.

2.10 Non-Negative Matrix Factorization

As mentioned in the previous chapter, NMF is a decomposition algorithm that has been used in various applications. For the purposes of this research, we will describe NMF in the context of its application to targeted speech denoising. We

will begin our discussion of NMF with an introduction to the algorithm itself. Next, we present a formulation for a neural-network based implementation of the algorithm. Finally, we discuss the benefits and conveniences of a neural network NMF implementation.

2.10.1 NMF

Given a non-negative matrix, $\boldsymbol{X}$, the K-rank Non-Negative Matrix Factorization model aims to find non-negative matrix factors $\boldsymbol{W}$ and $\boldsymbol{H}$ such that:

$$\boldsymbol{X} \approx \boldsymbol{W}\boldsymbol{H} \tag{2.6}$$

where $\boldsymbol{X} \in \mathbb{R}_{\geq 0}^{M \times N}$ is a nonnegative input matrix, $\boldsymbol{W} \in \mathbb{R}_{\geq 0}^{M \times K}$, and $\boldsymbol{H} \in \mathbb{R}_{\geq 0}^{K \times N}$. For the purposes of this paper, we take $\mathbb{R}_{\geq 0}^{M \times N}$ to represent the set of all real, non-negative matrices of size $M \times N$. Equation 2.6 can be written in a column by column notation as

$$v \approx \boldsymbol{W}h \tag{2.7}$$

where v and h are the corresponding columns of $\boldsymbol{V}$ and $\boldsymbol{H}$. This can be interpreted as the columns of $\boldsymbol{V}$ being approximated by a linear combination of the columns of $\boldsymbol{W}$ being weighted by the elements of h. In this regard, we can consider $\boldsymbol{W}$ to be a set of basis vectors, each of which are activated by elements in the activation matrix, $\boldsymbol{H}$.

Now that we have described the parameters of the factorization, we discuss how the NMF algorithm ensures that the multiplication of the non-negative bases

matrix, $\boldsymbol{W}$, and the non-negative activation matrix, $\boldsymbol{H}$, successfully approximates the non-negative input matrix, $\boldsymbol{X}$. The method used in NMF is quite similar to the training mechanism introduced in Section 2.8. First, we must define a metric by which we can assess the quality of our approximation. Historically, the NMF algorithm has been implemented using an expansion upon the Kullback-Leibler divergence where the inputs are not constrained to sum to 1 [Lee and Seung, 2001]. This divergence between target and reconstruction is given by

$$D(\boldsymbol{X}, \hat{\boldsymbol{X}}) = \sum_{i,j} (\boldsymbol{X}_{i,j}[\log(\boldsymbol{X}_{i,j}) - \log(\hat{\boldsymbol{X}}_{i,j})] - \boldsymbol{X}_{i,j} + \hat{\boldsymbol{X}}_{i,j}) \tag{2.8}$$

By measuring the distance between the approximation and the true input matrix using some error/cost function, NMF algorithms apply multiplicative the bases and activations by a factor that depends on the quality of the reconstruction. Much like the backpropagation algorithm, these updates seek to adjust the model parameters to increase the quality of the reconstruction. It has been proven that the quality of this approximation improves monotonically with the application of multiplicative update rules in NMF algorithm [Lee and Seung, 2001].

As discussed in the next section, the similarities between the general framework of an NMF algorithm and neural networks allow us to directly and analogously implement an NMF algorithm using neural networks and an autoencoder.

2.10.2 A Neural Network Based Approach

Combining the discussion of Equation 2.6 and our introduction to autoencoders from Section 2.9, we introduce a neural network based implementation of the NMF algorithm. Equation 2.6 can be formulated as a linear autoencoder where:

$$\text{Encoding Layer:} \quad \boldsymbol{H} = \boldsymbol{W}^* \cdot \boldsymbol{X} \tag{2.9}$$

$$\text{Decoding Layer:} \quad \hat{\boldsymbol{X}} = \boldsymbol{W} \cdot \boldsymbol{H} \tag{2.10}$$

where once again $\boldsymbol{X} \in \mathbb{R}_{\geq 0}^{M \times N}$ is a nonnegative input matrix, $\boldsymbol{W} \in \mathbb{R}_{\geq 0}^{M \times K}$, and $\boldsymbol{H} \in \mathbb{R}_{\geq 0}^{K \times N}$. Here, $\boldsymbol{W}^*$ are the weights of the encoder layer and $\boldsymbol{W}$ are the weights of the encoder layer. $\boldsymbol{W}$ and $\boldsymbol{H}$ are the weights and bases matrices just as they were in the classic NMF formulation from Equation 2.6.

As presented, Equations 2.9 and 2.10 are equivalent to the NMF algorithm and do not add any additional benefit. Furthermore, the non-negativity constraints would be burdensome to implement under the proposed framework [Smaragdis, 2017]. As we have previously seen, however, leveraging activation functions not only introduces non-linearities that can be helpful for modeling purposes, but also allows us to transform our data as desired. The addition of an activation function, $g()$, brings us to our definition of a Non-Negative Autoencoder:

$$\text{Encoding Layer:} \quad \boldsymbol{H} = g(\boldsymbol{W}^* \cdot \boldsymbol{X}) \tag{2.11}$$

$$\text{Decoding Layer:} \quad \hat{\boldsymbol{X}} = g(\boldsymbol{W} \cdot \boldsymbol{H}) \tag{2.12}$$

where $g()$ is a function that maps its input into a non-negative output space such that $g : \mathbb{R}^{M \times N} \mapsto \mathbb{R}_{\geq 0}^{M \times N}$. Unlike the classic NMF implementation, we have loosened the nonnegativity constraint here as $\boldsymbol{W}$ is no longer to required to be

non-negative.

It should be apparent from our discussion of training schemas in Section 2.8 that we can use the backpropogation algorithm with gradient descent to iteratively update $\boldsymbol{W}$ and $\boldsymbol{H}$ such that we can train the autoencoder network to produce an accurate approximation for the input matrix. After training the network on an input matrix, we will have successfully found a bases matrix and activations matrix to explain the input sound. While this is certainly a neural network based implementation of NMF, it is not very useful on its own.

Now that we have a method for learning how to decompose a clean training sample into a matrix factorization, we extend this idea to approximate new input matrices with the same underlying structure as the matrices that we trained on. The key here is that there is some underlying structure in the input matrices under observation. This underlying structure is captured in the basis vectors of the $\boldsymbol{W}$ matrix and should theoretically be able to explain mmethod used in NMFs with the same latent structure. By fixing the bases matrix and learning a new activation matrix, we can use the bases matrix learned from one input matrix to explain a new input matrix with the same structure. That is, given an input matrix $\boldsymbol{X}$ and a learned model $\boldsymbol{W}$, we can estimate $\boldsymbol{H}$ such that $\hat{\boldsymbol{X}} \approx \boldsymbol{X}$. This estimation can be performed using a single-layer non-linear network given by:

$$\hat{\boldsymbol{X}} = g(\boldsymbol{W} \cdot \boldsymbol{H}) \tag{2.13}$$

We can estimate $\boldsymbol{H}$ once again using the back propagation algorithm with gradient descent. In summary, we have introduced a neural network based imple-

mentation of NMF that consists of a non-negative autoencoder and a single layer neural network. The autoencoder serves the purpose of learning a bases model for the class of input matrices under observation, while the single layer neural network allows us to explain a new input matrix by activating these fixed bases.

2.10.3 The Benefits of a Neural Network Based Approach

Now that we have defined a framework for learning a model for a class of input matrices and a method for explaining new input matrices with the same underlying structure, we must address the question of why this implementation is necessary or better than the classic NMF implementations. Beginning with the activation function that differentiates our denoising autoencoder from a linear autoencoder, we have introduced a non-linearity in our algorithm that allows us to model more complex patterns in our input matrices. By implementing the NMF algorithm with neural networks, we can introduce arbitrary complexities such as multiple encoding and decoding layers that are suited for the task at hand. Although not investigated in this thesis, this formulation also allows for various alternative neural network architectures to be analyzed in their effectiveness in NMF applications. Such alternative architectures include Convolutional and Recurrent Neural Networks, which are known for strong performance in image and audio processing due to their spatial and temporal dependencies, respectively [Krizhevsky et al., 2012][Mikolov, 2010].

Perhaps the most appealing benefit of this implementation is the level of support surrounding the implementation of neural networks. As we will discuss in Section 3, various libraries and frameworks have out-of-the-box neural network

implementations that allow for quick, modular experimentation. These packages provide implementations for various layer structures, activation functions, and auto-differentiation for backpropagation with gradient descent. These tools allow us to quickly experiment with various architectures to determine the optimal architecture for the problem at hand, without getting bogged down in the underlying implementation details. In addition to these structural and convenience improvements, neural network implementations have been demonstrated comparable performance to classical NMF implementations [Smaragdis, 2017].

2.11 NMF and Speech Denoising

Thus far, most of our discussion surrounding the theory of NMF has been presented for the general case for some non-negative input matrix. We now turn our attention here and for the rest of this thesis to the application of NMF for targeted speech denoising.

In the audio samples analyzed in this thesis, we are exclusively dealing with discrete-time, real-valued signals, denoted by $x[n]$, where each point in time, n, represents a sample of the continuous-time signal, $x(t)$. As mentioned, NMF only works for non-negative input matrices and we have already stated that $x[n]$ is a real-valued signal. Therefore, we must somehow transform our input audio vector such that it is represented by a non-negative matrix. The transform used in this thesis is known as the Discrete Time Fourier Transform (DTFT) given by

$$X(\omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{2.14}$$

The DTFT transforms the signal from the time domain into the frequency domain where the signal is observed as the sum of sinusoids given by $e^{-j\omega n}$, where ω represents the frequency of the sinusoid. The magnitude of the DTFT is given by

$$|X(\omega)| \tag{2.15}$$

and determines the presence of a sinusoid $e^{j\omega n}$ in $x[n]$. The phase of the DTFT is given by

$$\theta(\omega) = \angle X(\omega) \tag{2.16}$$

and represents the alignment of the sinusoids in relation to one another to make up $x[n]$.

Instead of analyzing the raw time-domain signal itself, we use the DTFT to produce a magnitude spectrogram, which is a non-negative matrix describing the time-frequency relationship of the signal. Once we form our approximation for the target spectrogram, we can then use the inverse DTFT to return to the time domain and reconstruct the target speech signal. More details on our construction of the magnitude spectrogram for an input audio wave will be detailed in Chapter 3.

Now that we have the ability to transform a real-valued speech signal into a non-negative matrix, we can pass magnitude spectrograms of a target speaker or source of sound and model it as demonstrated in Section 2.10.2. For demonstrative purposes, we will look at a simple example of the NMF algorithm's ability to learn how to model the notes of a piano.

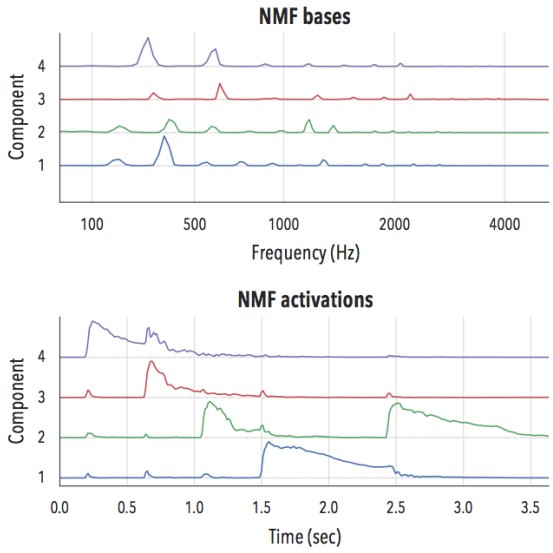

Figure 2.6: The bases and activation matrices obtained by performing the classic NMF algorithm with $K = 4$. [Smaragdis, 2017]

Example: NMF for Modeling Piano Notes

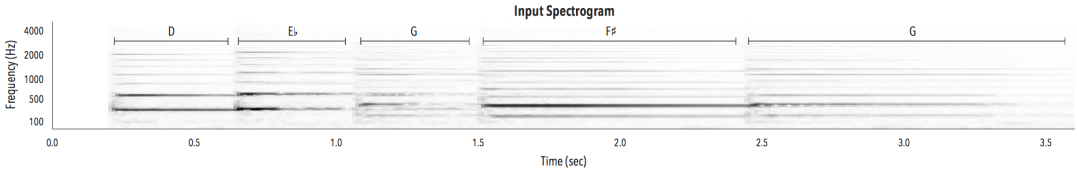

Figure 2.5: The input magnitude spectrogram of a piano playing 5 notes as labelled. [Smaragdis, 2017]

Figure 2.5 shows the resulting magnitude spectrogram from taking the DTFT of an audio clip of five notes being played consecutively on a piano. Since four distinct notes are played in the sequence, it makes sense to model the sound of the piano using an NMF algorithm with a rank of $K = 4$.

Figure 2.6 shows the bases and activations matrices that are obtained by performing the classic implementation of the NMF algorithm to factorize the input

spectrogram into bases and activations factors. As you can see, the activations in Figure 2.6 are clearly aligned to the boundaries of the notes in the sequence in Figure 2.5. Each vector in the activations matrix operates on a corresponding basis vector, implying that each note in the piano sequence is modeled by a single basis vector and activated by a single activation vector. Since notes are played by themselves in the sequence, only a single basis vector is ever activated at any one time.

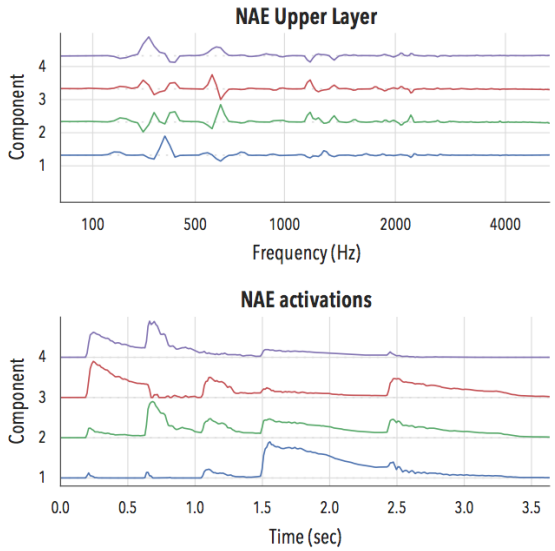

Figure 2.7: The bases and activation matrices obtained by performing the NMF algorithm with a neural network implementation with $K = 4$. [Smaragdis, 2017]

Figure 2.7 shows the bases and activation matrices obtained by using our neural network implementation of the NMF algorithm. While the multiplication of the bases and activation matrices will approximate the input spectrogram, they are distinctly different from the matrices that we found using the classic NMF algorithm. The reason for this is that we have relaxed the non-negativity constraint in our neural network implementation of the NMF algorithm by only requiring that

the input, latent state, and output of the autoencoder network be non-negative. This allows our bases matrix to contain negative values. This does not affect our approximation of the input, but it does allow for cross-cancellation between the basis resulting from multiple bases being unnecessarily activated at a given time. We can use the L1-Regularization penalty described in Section 2.8.1 to encourage sparsity in our activation matrix.

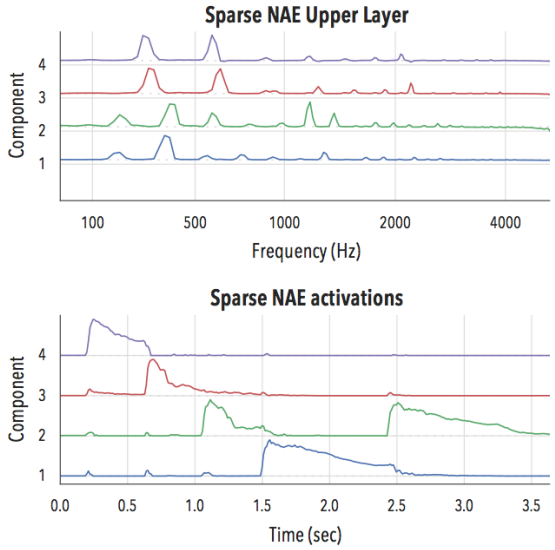

Figure 2.8: The bases and activation matrices obtained by performing the NMF algorithm with an L1-regularized neural network implementation with $K = 4$. [Smaragdis, 2017]

The results of approximating the input spectrogram with L1 regularization added to the neural network implementation of the NMF algorithm are shown in Figure 2.8. These learned matrices look much more like what we would expect, given the results shown in Figure 2.6. Now that we have learned a model for these 4 piano notes, we could fix the learned bases matrix and use the single layered neural network to learn an activation matrix to explain a completely new sequence

of these notes.

Chapter 3

Problem Statement and Implementation Details

Motivated by the demonstrated ability of NMF to model piano notes and, by extension, the voice of a human speaker, we aim to apply a neural network based NMF implementation to the Cocktail Party Problem. In this chapter, we will define a general framework and data model that we will use in the experiments described in Chapter 5.

3.1 Noisy Speech Generation

Given a clean speech signal from our target speaker, $x[n]$, we can create a noisy speech signal, $y[n]$, given by

$$y[n] = x[n] + N[n] \tag{3.1}$$

where $N[n]$ is additive noise that is applied to the clean speech signal to

create a noisy speech signal. We can measure the level at which the noise obfuscates the clean speech signal by measuring the Signal-to-Noise Ratio (SNR) of the noisy speech signal. SNR is essentially the ratio of the average power of the speech signal to the average power of the noise signal. We can then, therefore, adjust this SNR as we desire by scaling the magnitudes of the noise signal and thus scaling the average power of the noise signal.

The clean data sources that were used in this thesis are:

1. An Audiobook of "The History of the Four Georges" by Justin McCarthy
2. The CHiME Utterances Dataset
3. Clean sentences from 5 speakers [Smaragdis, 2017]

The noise sources that were considered are:

1. Bar Noise from YouTube
2. Babble noise formed by combining sentences from speakers other than the target speakers [Smaragdis, 2017]

As we are seeking to build off of the previous work of Paris Smaragdis, we will mostly limit our analysis to the set of speakers and sentences that he used in his papers.

3.2 Frequency Transform and Inverse Transform

As discussed in Section 2.11, we use a frequency transform known as the DTFT to represent the input data as a magnitude spectrogram. As is performed in the Smaragdis paper, we use a 512 DFT and applying a square root Hann window

with a hop size of 25%. We reconstruct our target speech signal by applying the following to the output of the model of each known source (See Section 4.1):

$$s_i(t) = STFT^{-1}(\frac{\hat{\boldsymbol{X}}_{\boldsymbol{i}}}{\sum_j \hat{\boldsymbol{X}}_{\boldsymbol{j}}} \odot \boldsymbol{X} \odot e^{i\Phi}) \tag{3.2}$$

where $STFT^{-1}()$ is the inverse spectrogram operator, $\odot$ indicates element wise multiplication, Φ is the phase of the noisy input mixture, and $\boldsymbol{X_i}$ is the estimated magnitude spectrogram of the i'th source.

3.3 Implementation Details

The experiments in this thesis were performed using Python3, Numpy, Tensorflow, and Librosa. Tensorflow is an neural network package that was internally developed and eventually open-sourced by Google [Abadi et al., 2015]. Numpy is a package known for scientific computing and matrix operations within python. Finally, Librosa is a python package for music and audio analysis [Mcafee et al., 2015]. All experiments were performed on a CPU with 16GB of RAM and a 2.6 GHz processor.

In order to quantitatively analyze our results, we leverage the Google Speech Recognizer to define a ground truth value for an audio clip. Using the Google Speech Recognizer transcription as a ground truth, we can compare the transcription of our reconstructions and compute a Word Error Rate (WER) to quantitatively assess the performance of our targeted speech denoising experiments. In situations where the Google Speech Recognizer was unable to present a ground truth value, we used the MSE as a quantitative measure of the performance of our estimate. Finally,

we evaluate our semi-supervised results using the perceptual evaluation of speech quality (PESQ), which is the telecommunications standard for the evaluation of intelligibility [Rix et al., 2001].

Chapter 4

Proposed Methods and Experiments

We will begin our discussion of our proposed methods and experiments by reviewing previous research done towards a neural network NMF implementation. The modeling techniques that have been presented herein are mostly taken directly from prior implementations, however our application is different since we would like to analyze the implementations performance in a semi-supervised setting. After a brief overview of the prior research, we will describe the approaches that were taken to research the effectiveness of a neural network based NMF algorithm for targeted speech denoising in a semi-supervised setting.

4.1 A Neural Network Based NMF Implementation for Source Separation

In previous work on neural network based NMF implementations, the modified NMF algorithm is applied to the problem of supervised source separation. Clean sentences from multiple speakers are used to learn speech models for each source in a mixture. What is of importance here is the fact that multiple speech models are being learned: one speech model for each speaker, meaning one set of basis vectors for each speaker in the mixture. New sentences, which have not been trained on, for each speaker are used to create a mixture signal by summing the signals together. Using the fixed, learned speech bases, a single layered neural network is built for each speaker to learn a new activation matrix. Finally, the reconstructions produced by each speakers' neural network are summed together to approximate the input mixture. This approximation is compared to the original mixture using the loss function of Equation 2.8. Based on this comparison of the sum of the individual sources to the original input mixture, each source's activation matrices are updated using gradient descent and the process is repeated for thousands of training epochs.

4.2 Application to Speech Denoising

The method proposed in the previous section works well for its intended purpose under most conditions. The obvious issue that we will run into in our application of this method is that all components of the input mixture must be known beforehand. In other words, the application of NMF to source separation is a completely

supervised problem. In the case of targeted speech denoising, we will rarely have any prior information about the additive noise that is corrupting our clean, target signal.

4.2.1 Unsupervised Approach

The unsupervised approach to the targeted speech denoising problem was the first solution that we tried in our experimentation. Unfortunately, due to the definition of the NMF algorithm's loss function, any unsupervised approach leads to the target speaker's bases being activated in an attempt to explain the entire mixture. However, attempts to solve the unsupervised problem were not wholly unsuccessful. In fact, the ability of the NMF bases to attempt to explain foreign signals was the motivation for the semi-supervised approach that we will be working towards over the next few sections. If one set of bases could be forced to explain all components of a mixture, we hypothesize that we would be able to use the target speaker's bases to explain the target signal and some general set of bases to explain everything else.

4.2.2 Semi-Supervised Targeted Speech Denoising

While we may not have any prior indication of what kind of noise is corrupting the clean target speech, we can make a general assumption that there is in fact noise corrupting the target speech. Making the assumption that there is some noise present allows us to create a universal noise model just as we would for a completely independent source. By modeling an unknown noise source with a general noise model, we are transforming the problem of targeted source separation

into a semi-supervised source separation problem. With this approach, we hope that the speaker's bases will be used to explain only the speaker's contribution to the noisy signal, while the bases of the general noise model will be able to explain any other noise that we encounter.

4.2.3 An Improved Cost Function

One observation that we made during our own implementation of the NMF algorithm was that the algorithm did not tend to perform well in the presence of inputs that were very similar to each other. This held true even in the fully supervised source separation problem. Given the presence of two similar sources in the mixture, such as two men with similar voices speaking, the bases of one speaker can be activated to reconstruct the speach of another. This results in the individual reconstruction components being correlated to one another. In order to address this, we propose a modification to the KL-Divergence cost function from Equation 2.8. The modified cost function is given by

$$D(\boldsymbol{X}, \hat{\boldsymbol{X}}) = \sum_{i,j}(\boldsymbol{X}_{i,j}[\log(\boldsymbol{X}_{i,j}) - \log(\hat{\boldsymbol{X}}_{i,j})] - \boldsymbol{X}_{i,j} + \hat{\boldsymbol{X}}_{i,j}) + \lambda \sum_{k} \|H_k\|_{L_1} + r^2 \quad (4.1)$$

where the second term is a sparsity regularizer and r^2 is the square of the Pearson Correlation of the individual mixture components of $\hat{\boldsymbol{X}}$. The pearson correlation component between two signals, X and Y, is given by

$$r = r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\,\sqrt{n\sum y_i^2 - (\sum y_i)^2}} \quad (4.2)$$

This correlation term acts as a penalty that discourages the model from learning activation functions that act on the bases of one component to explain another. This correlation component is shown to produce improved results in targeted speech denoising and source separation when the input mixture.

Chapter 5

Results and Discussion

Given the poor performance that was observed in an unsupervised setting, we then turn our focus to a ground-up approach. We start simply by analyzing our ability to model a speaker's speech and use this learned model to explain a new testing sentence. Next, we assess the ability of our model to perform the supervised source separation technique from prior neural network based NMF research in the presence of:

1. Additive White Gaussian Noise (AWGN)
2. Additional Speaker's
3. Bar Noise

Finally, we present results for numerous semi-supervised cases in which a target speaker model is found and a general noise model (one that was not trained with the class of noise used in the testing mixture) is used to explain the testing mixture. Unless otherwise noted, we use the data preprocessing techniques introduced in Chapter 3 along with the below choices for free parameters. These choices were

informed by either experimental confirmation or by the findings presented in previous research [Smaragdis, 2017].

Parameter	Value
Activation Function	SoftPlus
Learning Rate	0.01
Training Epochs	50000
L1-Regularization λ	0.001
NMF Rank	20
SNR	0 dB
Autoencoder Architecture	Shallow

5.1 Model Validation

By ensuring that we have the ability to reproduce comparable results to those presented in the Smaragdis paper, we can confirm that our neural network implementation of NMF is correct.

5.1.1 Learning a Speech Model

First, we learn speech models for each of the speakers that we will encounter in the mixture. We will refer to the speakers as Male 1, Male 2, Male 3, and Female 1. We present the results for Male 1 in this section. The procedure and results for learning the bases of the other speakers follow an identical procedure.

Using four concatenated sentences as training data, we first train a shallow non-negative autoencoder to find the bases matrix, $\boldsymbol{W}$, of Male 1. We train the autoencoder by feeding the entire training set as a single batch to the autoencoder

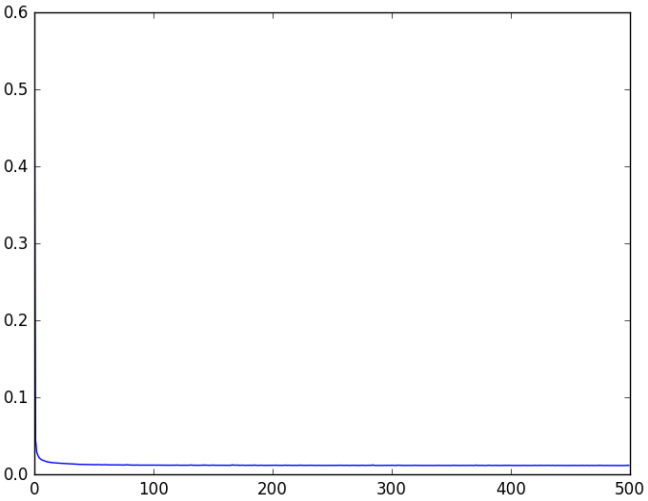

Figure 5.1: Loss value over the training phase of learning a speech model for Male 1. The autoencoder, despite being shallow, quickly converges to a minima and does so monotonically as expected.

network and using the Adam Optimizer to perform backpropagation with gradient descent for parameter updates [Kingma and Ba, 2014]. Previous research efforts have used the RProp algorithm for backpropagation, however, we did not observe any noticeable performance differences as a result of this change.

Figure 5.1 shows the value of our observed loss over the course of the training period. As mentioned earlier, the loss values are expected to decrease monotonically until they arrive at a minima. This is confirmed by the plot in Figure 5.1

5.1.2 Learning an Out-Of-Sample Activation Matrix

Now that we have learned bases for Male 1, we should confirm that we are able to approximate new testing sentences from Male 1 that we have not seen in the training of the bases. We confirm this by using the $\boldsymbol{W}$ matrix that we just learned as a fixed parameter in a single layer neural network. We again use the Adam

Optimizer to learn an activation matrix, $\boldsymbol{H}$, such that the activation vectors of $\boldsymbol{H}$ activate the learned bases to approximately reconstruct the new testing sentence. We present the results of this test for speaker 1 below and the observer WER across all speakers.

- Male 1 Ground Truth: "wipe the grease off his dirty face"
- Male 1 Reconstruction: "what's the grease off his dirty face"
- WER Across All Speakers: 6%

Qualitatively, the reconstructions of the out-of-sample testing sentences sound almost identical to the samples themselves. Quantitatively, we observed that the speech recognition system returned a missing, substituted, or deleted word 3 times out of 33 possible words in the speakers' testing sentences. We note that the ground truth labels were not always 100% accurate, perhaps elevating the true error. We also note that this is a fairly small sample size of testing sentences and we would expect further experimentation to drive the WER down. Nonetheless, we consider these results and our qualitative assessment of the reconstructions as validation that our model is working as intended.

5.2 Supervised Source Separation

After validating our model and implementation, we now turn our attention to the problem of supervised source separation as demonstrated in the Paris paper. As the Google Speech Recognition software was unable to produce a transcription of the audio signals, we abandon the WER metric in favor of the Mean-Squared Error (MSE) given by

Source 1	Source 2	r^2?	Source 1 MSE	Source 2 MSE	Mixture MSE
Male	AWGN	N	0.000292	0.471941	0.000168
Male	AWGN	Y	0.000292	0.471662	0.000169
Male	Bar Noise	N	0.000811	0.006157	0.000132
Male	Bar Noise	Y	0.000750	0.005967	0.000137
Male	Female	N	0.000609	0.000668	0.000073
Male	Female	Y	0.000617	0.000624	0.000105
Male 1	Male 2	N	0.001329	0.001434	0.000420
Male 1	Male 2	Y	0.001277	0.001483	0.000113

Table 5.1: MSE Values for the Case of Supervised Source Separation

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{5.1}$$

where $hatY_i$ is our approximation for Y_i.

The results of the supervised source separation are shown in Table 5.1. We present the source separations of multiple pairs of source mixtures. Once again, the supervised case performed as expected. The only exception this time is given by the source separation of the Male 1 speaker and the Male 2 speaker, which perform an order of magnitude worse than all other separations. This does not follow any of the results put forth by in prior research, however, we believe that this can be attributed to the underperformance of the NMF algorithm in the presence of a mixture comprised of correlated components. We observed that the individual components contained parts of each other's signal because their bases were being activated in order to explain both signals. This was the original motivation behind the addition of the Pearson correlation parameter. The addition of this parameter makes a big improvement in the accuracy of the mixture reconstruction and also reduces the crosstalk in the individual sound samples.

Target Source	Noise Source	Noise Bases	r^2?	Target MSE
Male	AWGN	Bar Noise	N	0.000541
Male	AWGN	Bar Noise	Y	0.000542
Male	AWGN	Female	N	0.000822
Male	AWGN	Female	Y	0.000640
Male	Bar Noise	AWGN	N	0.001240
Male	Bar Noise	AWGN	Y	0.001240
Male	Bar Noise	Female	N	0.001035
Male	Bar Noise	Female	Y	0.001239
Male	Female	AWGN	N	0.001189
Male	Female	AWGN	Y	0.001303
Male	Female	Bar Noise	N	0.000844
Male	Female	Bar Noise	Y	0.000781
Male	Male 2	AWGN	N	0.001408
Male	Male 2	AWGN	Y	0.001440
Male	Male 2	Bar Noise	N	0.001194
Male	Male 2	Bar Noise	Y	0.001191
Male	Male 2	Female 2	N	0.001060
Male	Male 2	Female 2	Y	0.001014

Table 5.2: MSE Values for the Case of Semi-Supervised

5.3 Semi-Supervised Targeted Speech Denoising

We finally move onto the case most related to the Cocktail Party Problem, which is the semi-supervised targeted speech denoising. We present results below for the MSE between the target speech signal and our approximation of it from a semi-supervised training of a neural network based NMF algorithm using the bases of noise classes other than the one being applied in the testing phase.

Table 5.2 shows a consistent improvement for the semi-supervised approach with the addition of the correlation penalty had on the target MSE for the case of both inputs of Male 1 and Male 2 being highly correlated.

Finally, we present results for another semi-supervised case where the general

	Noisy	Denoised
Target: Male Noise: AWGN	1.26	1.24
Target: Male Noise: Bar	0.86	1.19
Target: Male Noise: Female	1.23	1.21
Target: Male Noise: Male #2	0.94	1.32

Figure 5.2: PESQ results for the semi-supervised case with no correlation penalty

	Noisy	No Correlation	Correlation
Target: Male Noise: AWGN	1.26	1.24	1.45
Target: Male Noise: Bar	0.86	1.19	1.22
Target: Male Noise: Female	1.23	1.21	1.04
Target: Male Noise: Male #2	0.94	1.32	1.26

Figure 5.3: PESQ results for the semi-supervised case with the addition of the correlation penalty

noise model is trained using data from all sources except samples from the noise and target sources present in the mixture. Figures 5.2 and 5.3 show the PESQ results for this scenario. PESQ values range from -0.5 (worst) to 4.5 (best).

These results show mixed effects for the addition of the correlation penalty. While we mostly see increased performance over the base case, we do see some instances of generally decreased performance for the case of a female noise source. These results somewhat contradict a qualitative assessment of the intelligibility

of the denoising result, however, as the denoised reconstructions sound noticeably better than the original noisy mixture. We attribute this discrepancy to distortion introduced by the reconstruction algorithm, as well as phase artifacts that are introduced by applying the noisy mixture phase to the estimated target reconstruction.

Chapter 6

Future Work and Conclusions

The benefit offered by the addition of the correlation penalty for similar inputs leads me to believe that research in this direction could potentially allow the NMF algorithm to to be applied more easily to mixtures containing highly similar data. We have shown improvement over the baseline implementation using a naive addition of the correlation penalty. We believe that a cross-validation scheme to find an ideal weighting of the correlation penalty could possibly yield even greater results.

In a real world application of this procedure, we believe that a more comprehensive set of noise training data, consisting of common, everyday noise and situations, should be used. As this training set of noise data became more exhaustive over time, the problem would approach the supervised domain, where we were able to show good results for all cases of noise.

References

1. Cherry, Colin, and J. A. Bowles. "Contribution to a study of the Cocktail party problem." The Journal of the Acoustical Society of America 32.7 (1960): 884-884.
2. Mohammadiha, Nasser, Paris Smaragdis, and Arne Leijon. "Supervised and unsupervised speech enhancement using nonnegative matrix factorization." IEEE Transactions on Audio, Speech, and Language Processing 21.10 (2013): 2140-2151.
3. Bee, Mark A., and Christophe Micheyl. "The cocktail party problem: what is it? How can it be solved? And why should animal behaviorists study it?." Journal of comparative psychology 122.3 (2008): 235.
4. Venkataramani, Shrikant, Y. Cem Subakan, and Paris Smaragdis. "Neural network alternatives to convolutive audio models for source separation." arXiv preprint arXiv:1709.07908 (2017).
5. Lee, Daniel D., and H. Sebastian Seung. "Algorithms for non-negative matrix factorization." Advances in neural information processing systems. 2001.
6. Gillis, Nicolas. "The why and how of nonnegative matrix factorization." Regularization, Optimization, Kernels, and Support Vector Machines 12.257 (2014).
7. Bishop, Christopher M. Pattern Recognition and Machine Learning. Springer-Verlag New York, 2016.
8. Bagnell, J. Andrew, and David M. Bradley. "Differentiable sparse coding." Advances in neural information processing systems. 2009.
9. Healy, E. W., Yoho, S. E., Wang, Y., and Wang, D. (2013). An algorithm to improve speech recognition in noise for hearing-impaired listeners. Acoustical Society of America, 134(4):30293038.
10. Feng, X. Zhang, Y. Glass, J. (2014). Speech Feature Denoising and Dereverberation via Deep Autoencoders for Noisy Reverberant Speech Recognition. MIT Computer Science and Artificial Intelligence Laboratory Cambridge, MA, USA, 02139
11. Liu, D. Smaragdis, P. Kim, M. (2014). Experiments on Deep Learning for Speech Denoising. University of Illinois at Urbana-Champaign, USA Adobe Research, USA

12. Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. Science, 313(5786):504507.

13. Kaplan, G., Longueira, F., and Smarsch, M. (2016). Selective Hearing. The Cooper Union for the Advancement of Science and Art

14. D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, CoRR, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

15. Abadi, Martn, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." arXiv preprint arXiv:1603.04467 (2016).

16. McFee, Brian, et al. "librosa: Audio and music signal analysis in python." Proceedings of the 14th python in science conference. 2015.

17. Rix, Antony W., et al. "Perceptual evaluation of speech quality (PESQ)-a new method for speech quality assessment of telephone networks and codecs." Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on. Vol. 2. IEEE, 2001.

18. Deng, Li. "The MNIST database of handwritten digit images for machine learning research [best of the web]." IEEE Signal Processing Magazine 29.6 (2012): 141-142.

19. McDermott, Josh H. "The cocktail party problem." Current Biology 19.22 (2009): R1024-R1027.

20. Auer, Peter; Harald Burgsteiner; Wolfgang Maass (2008). "A learning rule for very simple universal approximators consisting of a single layer of perceptrons" (PDF). Neural Networks. 21 (5): 786795. doi:10.1016/j.neunet.2007.12.036. PMID 18249524

21. Bengio, Y. (2009). "Learning Deep Architectures for AI" (PDF). Foundations and Trends in Machine Learning. 2. doi:10.1561/2200000006.

22. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

23. Mikolov, Tomas, et al. "Recurrent neural network based language model." Interspeech. Vol. 2. 2010.

24. Abadi, Martn, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." arXiv preprint arXiv:1603.04467 (2016).

Appendix

Model File

```
## Matthew Smarsch
## Cooper Union Electrical Engineering Master's Thesis
## Advisor: Professor Sam Keene
## Neural Network NNMF Implementation for the Cocktail Party Problem

import argparse
import sys, os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
import tensorflow as tf
import math
import numpy as np
import datetime
import wave
import matplotlib.pyplot as plt
import pylab
import librosa
import librosa.display
import scipy
from scipy.io import wavfile
from scipy.fftpack import fft
from scipy import signal
from keras import backend as K

# Attempts to adjust the SNR to the targ_snr on the scale specified by scale
# Looks to adjust noise accordingly, starting with noise_arr, then clean_arr,
#
# IMPLEMENTATION REQUIRED: Scale both clean and noise if SNR can't be achieved
#                                                    by only adjusting one array
#
# Returns adjusted clean_arr and noise_arr
def adjust_SNR(clean_arr, noise_arr, targ_snr, scale = 'db'):
    db_snr, lin_snr = get_SNR(clean_arr, noise_arr)
    if scale == 'db':
        targ_snr = math.pow(10, targ_snr/10)
    if lin_snr == targ_snr:
        return clean_arr, noise_arr
    else:
        scale_factor = math.sqrt(lin_snr/targ_snr)
        noise_arr = np.multiply(noise_arr, scale_factor)
        return clean_arr, noise_arr

# Calculates SNR of a clean and noise array
#
# Return: SNR in dB and on a linear scale
def get_SNR(clean_arr, noise_arr):
    p_clean = np.mean(np.square(clean_arr))
    p_noise = np.mean(np.square(noise_arr))
    lin_snr = p_clean/p_noise
    db_snr = 10*math.log10(lin_snr)
    return db_snr, lin_snr

def correlation_coefficient_loss(y_true, y_pred):
    x = y_true
    y = y_pred
    mx = K.mean(x)
```

```
56      my = K.mean(y)
57      xm, ym = x-mx, y-my
58      r_num = K.sum(tf.multiply(xm,ym))
59      r_den = K.sqrt(tf.multiply(K.sum(K.square(xm)), K.sum(K.square(ym))))
60      r = r_num / r_den
61
62      r = K.maximum(K.minimum(r, 1.0), -1.0)
63      return K.square(r)
64
65  # Alias for wavfile.read -> reads .wav to numpy array
66  #
67  # Returns: Numpy array representation of .wav file
68  def read_wav(filename):
69      return librosa.load(filename, None)
70
71  def save_plot(values, filename):
72      plt.figure()
73      plt.plot(values)
74      plt.savefig(filename, bbox_inches='tight')
75
76  class SentenceMixture:
77      #def __init__(self, speaker_model, sess, speaker, noisy_speaker = None,
            bases_hack = None):
78      def __init__(self, sess, speaker, noisy_speaker = None, bases_hack = None):
79          #self._speaker_model = speaker_model
80          #self._rank = self._speaker_model._rank
81          self._rank = 20
82          self._sess = sess
83          self._speaker = speaker
84          self._bases_hack = bases_hack
85          self._noisy_speaker = noisy_speaker
86          #self._bases = self._speaker_model._bases
87          self._bases = bases_hack[0]
88          if noisy_speaker is None:
89              self._testing_sentence = self._speaker._testing_sentence
90              self._clean_testing_sentence = self._testing_sentence
91          else:
92              self._clean_testing_sentence, self._testing_sentence = self.
                    create_testing_sentence()
93          self._testing_fs = self._speaker._testing_fs
94          self.compute_testing_spectrogram()
95          self._testing_data = self._testing_spectrogram[0]
96          self._lambduh = 0
97          self._num_epochs = 10000
98          self._learning_rate = 0.01
99          #self.build_singular_model()
100         self.build_double_model()
101
102     def create_testing_sentence(self):
103         min_length = min(np.shape(self._speaker._testing_sentence)[0], np.shape(
                self._noisy_speaker._testing_sentence)[0])
104         clean_sentence = self._speaker._testing_sentence[:min_length]
105         #noise_sentence = signal.decimate(self._noisy_speaker._testing_sentence,
                6)[2000:2000+min_length]
106         noise_sentence = self._noisy_speaker._testing_sentence[:min_length]
107         clean_sentence, noise_sentence = adjust_SNR(clean_sentence, noise_sentence
                , 0)
108         noisy_sentence = clean_sentence + noise_sentence
109         #clean_sentence = self._speaker._testing_sentence
110         #print(np.shape(clean_sentence))
111         #print(np.shape(self._noisy_speaker._testing_sentence))
112         #noisy_sentence = np.append(clean_sentence, self._noisy_speaker.
```

```
            _testing_sentence)
113         librosa.output.write_wav('noisy.wav', noisy_sentence, int(self._speaker.
                _fs))
114         return clean_sentence, noisy_sentence
115
116     def compute_testing_spectrogram(self):
117         fft_size = 512
118         D = librosa.stft(self._testing_sentence, n_fft=fft_size, hop_length=int(
                fft_size/4))
119         mag, phase = librosa.magphase(D)
120         self._testing_spectrogram = (mag, phase)
121
122     def build_singular_model(self):
123         self._x = tf.placeholder(tf.float32)
124         self._w = tf.placeholder(tf.float32)
125         self._H_act = model_variable([self._rank, np.shape(self._testing_data)
                [1]], 'H_act_mix')
126         tf.add_to_collection('l1', tf.reduce_sum(tf.abs(self._H_act)))
127         self._reconstruction = tf.nn.softplus(tf.matmul(self._w, self._H_act))
128         self._cost_function = tf.reduce_mean(tf.add(tf.subtract(tf.multiply(self.
                _x, tf.subtract(tf.log(self._x), tf.log(self._reconstruction))), self.
                _x), self._reconstruction))
129         self._l1_penalty = tf.reduce_sum(tf.get_collection('l1'))
130         self._loss = self._cost_function + self._lambduh * self._l1_penalty
131
132     def build_double_model(self):
133         self._x_1 = tf.placeholder(tf.float32)
134         self._w_1 = tf.placeholder(tf.float32)
135         self._H_act_1 = model_variable([self._rank, np.shape(self._testing_data)
                [1]], 'H_act_1')
136         tf.add_to_collection('l1', tf.reduce_sum(tf.abs(self._H_act_1)))
137         self._x_2 = tf.placeholder(tf.float32)
138         self._w_2 = tf.placeholder(tf.float32)
139         self._H_act_2 = model_variable([self._rank, np.shape(self._testing_data)
                [1]], 'H_act_2')
140         tf.add_to_collection('l1', tf.reduce_sum(tf.abs(self._H_act_2)))
141         self._reconstruction_1 = tf.nn.softplus(tf.matmul(self._w_1, self._H_act_1
                ))
142         self._reconstruction_2 = tf.nn.softplus(tf.matmul(self._w_2, self._H_act_2
                ))
143         self._reconstruction = self._reconstruction_1 + self._reconstruction_2
144         self._cost_function = tf.reduce_mean(tf.add(tf.subtract(tf.multiply(self.
                _x_1, tf.subtract(tf.log(self._x_1), tf.log(self._reconstruction))),
                self._x_1), self._reconstruction))
145         self._l1_penalty = tf.reduce_sum(tf.get_collection('l1'))
146         self._correlation_penalty = correlation_coefficient_loss(self.
                _reconstruction_1, self._reconstruction_2)
147         #self._mse_penalty = tf.sqrt(tf.reduce_mean(tf.square(self.
                _reconstruction_1 - self._reconstruction_2)))
148         #self._loss = self._cost_function + self._lambduh * self._l1_penalty +
                self._correlation_penalty
149         self._loss = self._cost_function + self._lambduh * self._l1_penalty
150
151     def train_init(self):
152         model_variables = tf.get_collection('model_variables')
153         self._optim = tf.train.AdamOptimizer(self._learning_rate).minimize(self.
                _loss, var_list=model_variables)
154         self._sess.run(tf.global_variables_initializer())
155
156     def train_single(self):
157         loss_values = []
158         self.train_init()
```

```
159         print("BEGINNING MIXTURE TRAINING:")
160         print(self._testing_data)
161         for epoch in range(self._num_epochs):
162             iter_loss = self.train_single_iter(self._testing_data)
163             if epoch % 100 == 0:
164                 print('epoch: {}'.format(epoch))
165                 print('loss: {}'.format(iter_loss))
166                 loss_values.append(iter_loss)
167         return loss_values
168 
169     def train_single_iter(self, data):
170         loss, _ = self._sess.run([self._loss, self._optim], feed_dict={self._x:
                data, self._w: self._bases})
171         return loss
172 
173     def single_test(self):
174         return self._sess.run([self._reconstruction, self._H_act], feed_dict={self
                ._x: self._testing_data, self._w: self._bases})
175 
176     def train_double(self):
177         loss_values = []
178         self.train_init()
179         print("BEGINNING MIXTURE TRAINING:")
180         print(self._testing_data)
181         for epoch in range(self._num_epochs):
182             iter_loss = self.train_double_iter(self._testing_data)
183             if epoch % 100 == 0:
184                 print('epoch: {}'.format(epoch))
185                 print('loss: {}'.format(iter_loss))
186                 loss_values.append(iter_loss)
187         return loss_values
188 
189     def train_double_iter(self, data):
190         loss, _ = self._sess.run([self._loss, self._optim], feed_dict={self._x_1:
                data, self._w_1: self._bases_hack[0], self._x_2: data, self._w_2: self
                ._bases_hack[1]})
191         return loss
192 
193     def double_test(self):
194         return self._sess.run([self._x_1, self._reconstruction_1, self._w_1, self.
                _H_act_1, self._reconstruction_2, self._w_2, self._H_act_2], feed_dict
                ={self._x_1: self._testing_data, self._w_1: self._bases_hack[0], self.
                _x_2: self._testing_data, self._w_2: self._bases_hack[1]})
195 
196 class SentenceSpeaker:
197     def __init__(self, audio_directory, name):
198         self._audio_directory = audio_directory
199         self._name = name
200         self._sentence_1, self._fs_1 = self.preprocess_sentence(audio_directory +
                'Sentence_1.wav')
201         self._sentence_2, self._fs_2 = self.preprocess_sentence(audio_directory +
                'Sentence_2.wav')
202         self._sentence_3, self._fs_3 = self.preprocess_sentence(audio_directory +
                'Sentence_3.wav')
203         self._sentence_4, self._fs_4 = self.preprocess_sentence(audio_directory +
                'Sentence_4.wav')
204         self._sentence_5, self._fs_5 = self.preprocess_sentence(audio_directory +
                'Sentence_5.wav')
205         self._training_sentences = [self._sentence_1, self._sentence_2, self.
                _sentence_3, self._sentence_4]
206         self._testing_sentence = self._sentence_5
207         self._big_training_sentence =  self.get_big_training_sentence()
```

```
208             self._fs = self._fs_1
209             self._training_fs = [self._fs_1, self._fs_2, self._fs_3, self._fs_4]
210             self._testing_fs = self._fs_5
211             self._mixture_sentence = self._sentence_5
212 
213         def get_big_training_sentence(self):
214             reshaped_sentences = []
215             for training_sentence in self._training_sentences:
216                 reshaped_sentences.append(np.reshape(training_sentence, (-1,1)))
217             return np.reshape(scipy.vstack(reshaped_sentences), (-1,))
218 
219         def preprocess_sentence(self, sentence_file):
220             sentence, fs = read_wav(sentence_file)
221             return sentence, fs
222 
223         def compute_training_spectrograms(self):
224             fft_size = 512
225             self._training_spectrograms = []
226             for fs, sentence in zip(self._training_fs, self._training_sentences):
227                 D = librosa.stft(sentence, n_fft=fft_size, hop_length=int(fft_size/4))
228                 mag, phase = librosa.magphase(D)
229                 self._training_spectrograms.append((mag, phase))
230 
231         def compute_big_training_spectrogram(self):
232             fft_size = 512
233             D = librosa.stft(self._big_training_sentence, n_fft=fft_size, hop_length=
                    int(fft_size/4))
234             mag, phase = librosa.magphase(D)
235             self._big_training_spectrogram = (mag, phase)
236 
237         def plot_spectrograms(self):
238             for spectr in self._training_spectrograms:
239                 plt.figure(figsize=(12, 8))
240                 librosa.display.specshow(spectr[0], y_axis='linear', x_axis='time')
241                 plt.colorbar(format='%+2.0f')
242                 plt.title('Linear-frequency power spectrogram')
243                 plt.tight_layout()
244                 plt.show()
245 
246     def reconstruct_wave(magnitude, phase):
247         reconstr = librosa.istft(magnitude * phase)
248         return reconstr
249 
250     def reconstruct_target_wave(noisy_input, reconstruction_targ, reconstruction_noise
        , phase):
251         reconstr = librosa.istft((reconstruction_targ + reconstruction_noise) * phase)
252         reconstr1 = librosa.istft((reconstruction_targ) * phase)
253         reconstr2 = librosa.istft((reconstruction_noise) * phase)
254         #reconstr1 = librosa.istft(np.multiply(np.multiply(np.divide(
                reconstruction_targ, np.add(reconstruction_noise, reconstruction_targ)),
                noisy_input), np.exp(1j * phase)))
255         #reconstr2 = librosa.istft(np.multiply(np.multiply(np.divide(
                reconstruction_noise, np.add(reconstruction_noise, reconstruction_targ)),
                noisy_input), np.exp(1j * phase)))
256         return reconstr, reconstr1, reconstr2
257 
258     def model_variable(shape, name):
259         variable = tf.get_variable(name=name,
260                                    dtype=tf.float32,
261                                    initializer=tf.random_uniform(shape, -0.1, 0.1))
262         tf.add_to_collection('model_variables', variable)
263         return variable
```

```
264
265 class SpeechModel:
266
267     def __init__(self, sess, speaker, args):
268         self._sess = sess
269         self._speaker = speaker
270         self._activation = args.activation
271         self._rank = args.rank
272         self._learning_rate = args.alpha
273         self._num_epochs = args.epochs
274         self._lambduh = args.lambduh
275         self._training_data = speaker._big_training_spectrogram[0]
276         self.build_model()
277
278     def build_model(self):
279         self._x = tf.placeholder(tf.float32)
280         W_enc = model_variable([self._rank, np.shape(self._training_data)[0]], '
                W_enc')
281         if self._activation == 'SOFTPLUS':
282             self._H_act = tf.nn.softplus(tf.matmul(W_enc, self._x))
283         elif self._activation == 'RELU':
284             self._H_act = tf.nn.relu(tf.matmul(W_enc, self._x))
285         tf.add_to_collection('l1', tf.reduce_sum(tf.abs(self._H_act)))
286         self._W_dec = model_variable([np.shape(self._training_data)[0], self._rank
                ], 'W_dec')
287         if self._activation == 'SOFTPLUS':
288             self._reconstruction = tf.nn.softplus(tf.matmul(self._W_dec, self.
                    _H_act))
289         elif self._activation == 'RELU':
290             self._reconstruction = tf.nn.relu(tf.matmul(self._W_dec, self._H_act))
291         self._cost_function = tf.reduce_mean(tf.add(tf.subtract(tf.multiply(self.
                _x, tf.subtract(tf.log(self._x), tf.log(self._reconstruction))), self.
                _x), self._reconstruction))
292         self._l1_penalty = tf.reduce_sum(tf.get_collection('l1'))
293         self._loss = self._cost_function + self._lambduh * self._l1_penalty
294
295     def train_init(self):
296         model_variables = tf.get_collection('model_variables')
297         self._optim = tf.train.AdamOptimizer(self._learning_rate).minimize(self.
                _loss, var_list=model_variables)
298         self._sess.run(tf.global_variables_initializer())
299
300     def train(self, log_file):
301         loss_values = []
302         self.train_init()
303         print("BEGINNING TRAINING:", file=log_file)
304         print(self._training_data, file=log_file)
305         for epoch in range(self._num_epochs):
306             iter_loss = self.train_iter(self._training_data)
307             if epoch % 100 == 0:
308                 print('epoch: {}'.format(epoch), file=log_file)
309                 print('loss: {}'.format(iter_loss), file=log_file)
310                 loss_values.append(iter_loss)
311         return loss_values
312
313     def train_iter(self, data):
314         loss, _ = self._sess.run([self._loss, self._optim], feed_dict={self._x:
                data})
315         return loss
316
317     def test(self):
318         return self._sess.run([self._reconstruction, self._W_dec, self._H_act],
```

```
                feed_dict={self._x: self._training_data})
319
320
321 def main():
322
323     '''
324
325     ENVIRONMENT SETUP
326
327     '''
328
329     parser = argparse.ArgumentParser()
330     parser.add_argument('-a', '--alpha', dest='alpha', type=float, default=0.01,
            help='the learning rate for backpropagation')
331     parser.add_argument('-act', '--activation', dest='activation', type=str,
            choices=['RELU', 'SOFTPLUS'], default='SOFTPLUS', help='activation
            function for the speech model')
332     parser.add_argument('-c', '--clean', dest='clean_model', type=str, choices=['
            Sentences', 'Book', 'Piano'], default='Sentences', help='clean speech type
             that we are trying to model')
333     parser.add_argument('-e', '--epochs', dest='epochs', type=int, default=50000,
            help='number of passes through the training set')
334     parser.add_argument('-l', '--lambduh', dest='lambduh', type=float, default=0,
            help='L1 regularization coefficient')
335     parser.add_argument('-r', '--rank', dest='rank', type=int, default=20, help='
            number of spectral components in the W (bases) matrix')
336     args = parser.parse_args()
337
338     simulations_dir = '/Users/MSmarsch/Documents/Thesis/Simulations/' + datetime.
            datetime.now().strftime('%Y-%m-%d_%H-%M-%S') + '/'
339     os.mkdir(simulations_dir)
340     '''
341
342     AUDIO PREPROCESSING
343
344     '''
345
346     if args.clean_model == 'Sentences':
347         ## Load Sentences in
348         sentences_dir = '/Users/MSmarsch/Documents/Thesis/Audio/Clean_Speech/
                Sentences/'
349         male_speaker_1 = SentenceSpeaker(sentences_dir + 'Male_1/', 'Male_1')
350         male_speaker_2 = SentenceSpeaker(sentences_dir + 'Male_2/', 'Male_2')
351         male_speaker_3 = SentenceSpeaker(sentences_dir + 'Male_3/', 'Male_3')
352         female_speaker_1 = SentenceSpeaker(sentences_dir + 'Female_1/', 'Female_1'
                )
353         female_speaker_2 = SentenceSpeaker(sentences_dir + 'Female_2/', 'Female_2'
                )
354         bar_noise = SentenceSpeaker(sentences_dir + 'Noise/', 'Noise')
355         awgn_speaker = SentenceSpeaker(sentences_dir + 'AWGN/', 'AWGN')
356         #speakers = [bar_noise, awgn_speaker]
357         #speakers = [male_speaker_1, male_speaker_2, male_speaker_3,
                female_speaker_2]
358         speakers = [male_speaker_1, female_speaker_2]
359         #noisy_speaker = speakers[1]
360         noisy_speaker = male_speaker_2
361         bases_hack = []
362
363         ## Compute Training Spectrogram by taking 512 pt. DFT with hann window and
                25% overlap
364         '''
365         target_speaker_model = None
```

```
366         for speaker in speakers:
367             speaker_dir = simulations_dir + speaker._name + '/'
368             os.mkdir(speaker_dir)
369             audio_dir = speaker_dir + 'Audio/'
370             os.mkdir(audio_dir)
371             plot_dir = speaker_dir + 'Plots/'
372             os.mkdir(plot_dir)
373             model_dir = speaker_dir + 'Model/'
374             os.mkdir(model_dir)
375 
376             speaker.compute_big_training_spectrogram()
377             reconstr = reconstruct_wave(speaker._big_training_spectrogram[0],
                    speaker._big_training_spectrogram[1])
378             librosa.output.write_wav(audio_dir + speaker._name + '_Input.wav',
                    reconstr, int(speaker._fs))
379             with tf.Session() as sess:
380                 speech_model = SpeechModel(sess, speaker, args)
381                 log_file = open(speaker_dir + 'output.log', 'w+')
382                 loss_values = speech_model.train(log_file)
383                 prediction, bases, activations = speech_model.test()
384                 bases_hack.append(bases)
385                 reconstr = reconstruct_wave(prediction, speaker.
                        _big_training_spectrogram[1])
386                 librosa.output.write_wav(audio_dir + speaker._name + '
                        _Reconstruction.wav', reconstr, int(speaker._fs))
387                 loss_filename = plot_dir + 'Loss.png'
388                 save_plot(loss_values, loss_filename)
389                 bases_filename = plot_dir + 'Bases.png'
390                 save_plot(bases, bases_filename)
391                 activations_filename = plot_dir + 'Activations.png'
392                 save_plot(activations, activations_filename)
393                 base_matrix_filename = model_dir + 'Bases.npy'
394                 np.save(base_matrix_filename, bases)
395                 speaker._bases = bases
396                 speech_model._bases = speaker._bases
397                 if target_speaker_model is None:
398                     target_speaker_model = speech_model
399             tf.reset_default_graph()
400         '''
401         for speaker in speakers:
402             bases_hack.append(np.load("/Users/MSmarsch/Documents/Thesis/
                    Simulations/All_Speakers/" + speaker._name + "/Model/Bases.npy"))
403         noisy_speaker._bases = bases_hack[1]
404         for speaker in speakers:
405             speaker._bases = bases_hack[0]
406             with tf.Session() as sess:
407                 #sentence_mixture = SentenceMixture(target_speaker_model, sess,
                        speaker, noisy_speaker, bases_hack)
408                 sentence_mixture = SentenceMixture(sess, speaker, noisy_speaker,
                        bases_hack)
409                 #sentence_mixture = SentenceMixture(speech_model, sess, speaker)
410                 #speaker._speech_model = speech_model
411                 speaker._sentence_mixture = sentence_mixture
412                 loss_values = sentence_mixture.train_double()
413                 #loss_values = sentence_mixture.train_single()
414                 noisy_input, reconstruction_targ, bases_targ, activations_targ,
                        reconstruction_noise, bases_noise, activations_noise =
                        sentence_mixture.double_test()
415                 #reconstruction, activations = sentence_mixture.single_test()
416                 #reconstr = reconstruct_wave(reconstruction, sentence_mixture.
                        _testing_spectrogram[1])
417                 #librosa.output.write_wav(speaker._name + '_Test.wav', speaker.
```

```
                    _clean_testing_sentence, int(speaker._fs))
418                 #librosa.output.write_wav(speaker._name + '_Reconstruction.wav',
                        reconstr, int(speaker._fs))
419                 reconstr, reconstr1, reconstr2 =reconstruct_target_wave(
                        noisy_input, reconstruction_targ, reconstruction_noise,
                        sentence_mixture._testing_spectrogram[1])
420                 librosa.output.write_wav('Mixture_Reconstruction.wav', reconstr,
                        int(speaker._fs))
421                 librosa.output.write_wav('Target.wav', speaker._testing_sentence,
                        int(speaker._fs))
422                 librosa.output.write_wav('Mixture_Reconstruction_Target.wav',
                        reconstr1, int(speaker._fs))
423                 librosa.output.write_wav('Mixture_Reconstruction_Noise.wav',
                        reconstr2, int(speaker._fs))
424                 for variable in tf.trainable_variables():
425                     print(variable)
426                 print(bases_targ)
427                 print('----------------------------------------')
428                 print(speaker._bases)
429                 print('----------------------------------------')
430                 print(bases_noise)
431                 print('----------------------------------------')
432                 print(noisy_speaker._bases)
433                 sys.exit()
434             tf.reset_default_graph()
435
436
437 if __name__ == '__main__':
438     main()
```

MSE Calculation

```
1  import numpy as np
2  import librosa
3  import sys
4
5  def getMSE(file1, file2):
6      signal1, sr1 = librosa.load(file1, None)
7      signal2, sr2 = librosa.load(file2, None)
8      min_length = min(len(signal1), len(signal2))
9      signal1 = signal1[:min_length]
10     signal2 = signal2[:min_length]
11     #return ((signal1-signal2) ** 2).mean(axis=1)
12     #return np.shape(signal1)
13     return ((signal1-signal2) ** 2).mean(None)
14
15 def main():
16     print(str(getMSE(sys.argv[1], sys.argv[2])))
17
18 if __name__ == '__main__':
19     main()
```