THE COOPER UNION
FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

A Fully Convolutional Neural Network Approach to End-to-End Speech Enhancement

by
Frank Longueira

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

April 16, 2018

Professor Sam Keene, Advisor

THE COOPER UNION
FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

Acknowledgments

Thank you to Professor Sam Keene, for his inspiration, guidance, and support as advisor to this endeavor.

Thank you to Matthew Smarsch, for being a steadfast partner throughout my academic years and now co-worker. See you at work on Monday.

Thank you to Christopher Curro, for his inspiration, support, and vast knowledge in the field of deep learning.

Thank you to The Cooper Union's Electrical Engineering & Mathematics Departments, for providing me with a logical framework for maneuvering through life and the desire to teach others what has been taught to me.

Thank you to my family, for being a constant source of support and encouragement throughout my life.

Thank you to Starbucks, for their coffee, Wi-Fi, and unlimited refills.

Thank you to Peter Cooper, for his open mind, practicality, and generosity that has given myself and many others the opportunity to study free of financial burden. His life has provided me with a model for rising to intellectual, financial, and social prominence from humble means.

Abstract

Speech enhancement seeks to improve the quality of speech degraded by noise. Its importance can be found in applications such as mobile phone communication, speech recognition, and hearing aids. An example of speech enhancement relates to the famous cocktail party problem. This problem deals with extracting a target speaker's voice from a mixture of background conversations. In such a situation, the human brain tends to do a good job focusing in on the target speech while blocking out the noisy environment surrounding it. The goal of solving the cocktail party problem is to find a computer algorithm that functionally mimics how the brain extracts the target speaker's voice. In this master's thesis, a novel approach to solving the cocktail party problem is presented that relies on a fully convolutional neural network (FCN) architecture. The FCN takes noisy, raw audio data as input and performs nonlinear, filtering operations to produce clean, raw audio data of the target speech at the output. Results from experimentation indicate the ability to generalize to new speakers and robustness to new noise environments of varying signal-to-noise ratios.

Contents

List of Figures

List of Tables

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 [1]. While modern hearing aids do a good job of amplifying sound, they do not do enough to increase speech quality and intelligibility. This is not a problem in quiet environments, but a standard hearing aid that simply amplifies audio will fail to provide the user with a signal they can easily understand when the user is in a noisy environment [2]. The problem of speech intelligibility is even more difficult if the background noise is also speech, such as in a bar or restaurant with many patrons.

While people without hearing impairments usually have no trouble focusing on a single speaker out of multiple, it is a much more difficult task for people with a hearing impairment [3]. The problem of picking out one person's speech in an environment with many speakers was dubbed the cocktail party problem in a paper by Colin Cherry, published in 1953 [4]. The paper asserts that humans are normally capable of separating multiple speakers and focusing on a single one. However, hearing impaired individuals may have issues when it comes to performing this same task. A solution to the cocktail party problem would be an algorithm that a computer can employ in real-time to enhance the speech corrupted by babble (background noise from other speakers). Traditionally, the cocktail party problem has been approached using several different techniques, such as using microphone arrays, monaural algorithms involving signal processing techniques, and Computational Auditory Scene Analysis (CASA) [1].

Modern hearing aids incorporate the microphone array strategy. They use beamforming to amplify sound coming from a specific direction (the simplest algorithms assume directly in front of the user) and attenuate the sound coming from elsewhere [5]. 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 speech changes location.

Monaural algorithms use a single microphone and so are not dependent on 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. Traditional monaural algorithms include spectral subtraction and Wiener filtering [8] - [9] . 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 by employing an optimal LTI filter in the mean-squared error sense based on stochastic process assumptions on the noisy input signal. If the background noise is also speech, as in the cocktail party problem, these types of filtering techniques have difficulty extracting the target speech. This difficulty arises due to speech of different human speakers occupying overlapping frequency ranges in the frequency domain. While traditional monaural strategies have been shown to improve speech quality, they have struggled with improving speech intelligibility for human listeners [6].

Computational Auditory Scene Analysis (CASA) has some promising results using ideal binary time-frequency masks to hide regions of the speech mixture where the SNR is below a certain threshold [7]. However, this method of separating speech from noise requires prior knowledge of both, as the mask is created based off of the relative strengths of the speech signal and the noise. This strategy also faces difficulty if the noise and target speech occupy similar frequency ranges as is the case with babble noise.

More recent studies in speech enhancement related to the cocktail party problem fall in the domain of deep learning. With the advent of big data, more memory, and increased processing power, deep learning has completely revolutionized many domains such as speech recognition and object recognition. Deep neural networks are able to learn complex, nonlinear representations of data that tend to far exceed human crafted features. Deep learning approaches to the cocktail party problem tend to take noisy spectrograms as input and transform them to clean spectrograms. The use of deep convolutional neural networks and deep denoising autoencoders on spectrograms have proven to be powerful techniques in practice [10]. One drawback to the use of spectrograms as input is the computation of spectrograms tends to be high since the short-time Fourier transform has to be applied to the raw audio data. This prior computation before inputting into the network requires time and hence increases the difficulty of use in real-time applications. In addition, phase information of the input speech tends to be lost in many of these approaches since only the magnitude spectrum is used. This can cause degradation in quality at the output of the system [11].

This master's thesis is motivated by the deep learning community's recently focused efforts on end-to-end speech enhancement systems that take the raw time domain audio signal as input instead of frequency domain features [12] - [14]. The approach that will be described in this thesis involves the use of a fully convolutional neural network (FCN) applied to raw audio data and is motivated by prior work in the area [15]. The approach builds upon the work of [16] that shows pooling layers may not be necessary for audio processing tasks. The proposed FCN based algorithm in this paper is advantageous for many reasons when it comes to a solution to the cocktail party problem. One reason is an FCN can be viewed as performing filtering directly in the time-domain and the key idea is the FCN can learn optimal, nonlinear filters for the given task at hand. In addition, an FCN by definition has no fully connected layers and generally does a better job at maintaining local temporal correlations in the audio signal from input to output [15]. Lastly, an FCN will generally have far fewer parameters than other correspondingly similar deep neural networks due to parameter sharing. This allows for less memory usage and quicker computation which is ideal for real-time applications. Before reviewing the results of this approach, the proceeding pages will review the necessary background and present an overview of the system.

Chapter 2

Background

2.1 Speech & Signal Processing Fundamentals

This section will go over some fundamental information related to speech and signal processing. First, the basics of speech will be reviewed and a simple way of modeling speech is presented. Next, a discussion of time-dependent Fourier analysis will take place. Time-dependent Fourier analysis is used in many practical speech enhancement applications. After this, a measurement of speech degradation by background noise called signal-to-noise ratio (SNR) will be discussed. Finally, this background section ends with an introduction to filtering signals and a robust algorithm for perfectly reconstructing a time-domain signal after it has been processed by a system.

2.1.1 Basics of Speech

Speech is produced by excitation of an acoustic tube called the vocal tract. There are three basic classes of speech sounds:

- **Voiced sounds**: periodic pulses of airflow excite the vocal tract
- **Fricative sounds**: produced by constricting the vocal tract somewhat and forcing air through
- **Plosive sounds**: pressure is built up by completely closing off the vocal tract and is then released

Speech can be modeled as the response of an LTI system, namely the vocal tract [17]. The vocal tract transmits excitations (vibrations) generated in the larynx to the mouth. In normal speech, the vocal tract tends to change shape slowly with time and imposes its characteristic frequencies, called formants, on the excitation traveling through it. Through this view, the vocal tract is a slow, time-varying filter and a speech signal can be expressed mathematically as

$$s(t) = e(t) * v(t) \tag{2.1}$$

where $s(t)$ is the speech signal, $e(t)$ is the excitation signal, and $v(t)$ is the impulse response corresponding to the vocal tract.

In a statistical sense, speech is a non-stationary signal. This means that the statistics of speech generally change over time. When speech is viewed on the time-scale of 10 - 40 ms, the statistics can be assumed to be relatively constant and Fourier analysis can be applied [18]. The frequency content of speech is generally below 8 kHz and hence this implies that the sampling rate used in speech applications does not need to be higher than 16 kHz. In fact, digital telephone communication systems have used sampling rates of 8 kHz without loss of intelligibility [18].

2.1.2 Time-Dependent Fourier Analysis

Non-stationary signals, such as speech, have statistics (i.e. properties such as amplitude, frequency) that change over time. A useful representation of these types of signals is called the spectrogram [18]. A spectrogram provides a time-frequency representation of a signal by using a mathematical tool called the short-time Fourier transform:

$$X[n,\omega] = \sum_{m=-\infty}^{\infty} x[n+m]w[m]e^{-j\frac{2\pi\omega}{N}m} \tag{2.2}$$

where $x[n]$ is a discrete signal with N points, $w[m]$ is a windowing sequence generally of shorter length than $x[n]$, n is a discrete-valued variable representing time, and ω is a discrete variable representing frequency.

For discrete signals, the short-time Fourier transform (STFT) can be interpreted as a sliding (through time) discrete Fourier transform (DFT) applied to windowed chunks of the signal. For each windowed chunk of the signal, the DFT extracts frequency information. The use of a windowing sequence is used to break the signal up into "pieces" and ensure smooth transitions in frequency information through time. A popular windowing sequence used in practice is called the Hanning window and it is defined as:

$$w[n] = \begin{cases} 0.5 - 0.5\cos(\frac{2\pi n}{M}), & 0 \leq n \leq M \\ 0, & otherwise \end{cases} \tag{2.3}$$

A spectrogram plots the magnitude of $X[n,\omega]$ across time and across frequency in a 2-D representation. The value of the magnitude response is represented by various colors in this 2-D representation (white generally representing higher magnitudes, black representing lower magnitudes). For DFT application on real-valued finite discrete signals, the discrete valued frequency variable, ω, uniquely and exhaustively describes all frequency content of the the input signal when viewed on the domain of $\{0, 1, 2, ..., \frac{N}{2}\}$. The reason for this is found in the study of discrete sampling theory, including the Nyquist-Shannon Sampling Theorem [18]. Conceptually, the idea is to treat finite real-valued signals as cyclical in time and in order to represent the information present in the signal the sampling rate must be at least twice the maximum frequency present in the signal. These facts allow a spectrogram plot to have a finite frequency axis, as seen in the figure below.

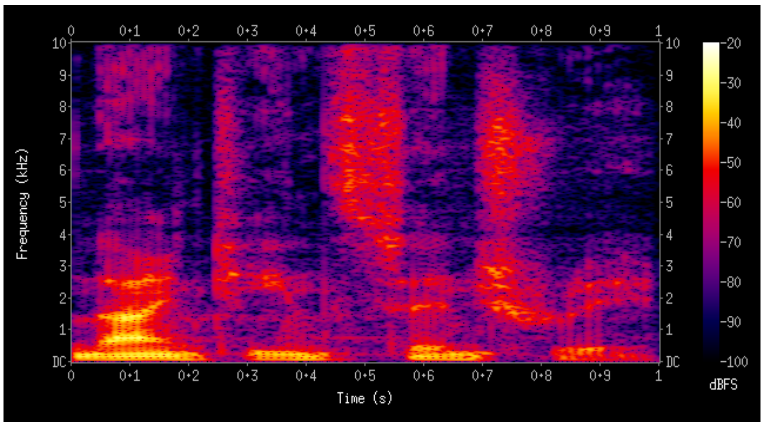

Figure 2.1: Spectrogram of the spoken words "nineteenth century" [19]

2.1.3 Signal-to-Noise Ratio (SNR)

A common measure for quantifying the amount a signal has been degraded by the presence of background noise is called the signal-to-noise ratio (SNR) [18].

$$SNR = 10 \log_{10} \frac{{\sigma_x}^2}{{\sigma_e}^2} \tag{2.4}$$

where ${\sigma_x}^2$ represents the variance of the signal and ${\sigma_e}^2$ represents the variance of the background noise. The units of SNR are named decibels (dB). If a signal is in the presence of background noise such that the SNR is equal to 0 dB, this implies that the relative power of each is about equal. A positive SNR indicates the signal power is stronger than the noise power, while a negative SNR indicates the noise power is stronger than the signal power.

2.1.4 Filtering

The concept of filtering in signal processing refers to the removal of unwanted frequency components from a signal. Commonly used filters in signal processing are found inside the class of linear-time invariant (LTI) systems. These filters are characterized entirely by their impulse response [18]. Specifically, the output signal can be expressed as a convolution of the filter's impulse response with the input signal. Many types of LTI filters exist with two popular ones being the ideal low-pass and ideal high-pass filters. The ideal low-pass filter is a system designed for removing frequency components above a specified cutoff frequency, while the ideal high-pass filter is a system designed for removing frequency components below a specified cutoff frequency. In practicality, ideal filters are not realizable but many approximations exist such as Butterworth filters and Chebyshev filters [18].

2.1.5 Overlap-Add Method of Reconstruction

In applications, such as speech enhancement and audio coding, where the input signal's time-dependent Fourier transform is modified, the overlap-add method of reconstruction provides a robust algorithm for perfectly reconstructing the output time domain signal [18].

Suppose that $R \leq L \leq N$. The following decomposition can be expressed:

$$x_r[m] = x[rR+m]w[m] = \frac{1}{N}\sum_{\omega=0}^{N-1} X_r[k]e^{j\frac{2\pi\omega}{N}m} \quad 0 \leq m \leq L-1 \tag{2.5}$$

where $x[n]$ is an N-point signal, $w[n]$ is an L-point windowing sequence, R represents the spacing between successive DFTs, and $x_r[n]$ represents the r^{th} recovered windowed slice of the signal $x[n]$. If the following condition is assumed about the windowing sequence:

$$\sum_{r=-\infty}^{\infty} w[n-rR] = 1 \tag{2.6}$$

Then $x[n]$ can be perfectly reconstructed by shifting the recovered segments to their original time locations and summing:

$$x[n] = \sum_{r=-\infty}^{\infty} x_r[n-rR] \tag{2.7}$$

An example of a windowing sequence that satisfies the above criteria is the Hanning window (discussed in Section 2.1.2) with length $L = M+1$ and $R = M/2$.

2.2 Traditional Speech Enhancement Methods

To get a better sense of the history of speech enhancement, this section will review a few traditional methods for removing background noise from a corrupted speech signal. These methods include spectral subtraction, Wiener filtering, and Ideal Binary Mask (IBM) estimation. In addition, a brief overview of popular evaluation metrics for speech enhancement systems will be presented. The metrics to be presented are named perceptual evaluation of speech quality (PESQ), short-time objective intelligibility (STOI), and word error rate (WER).

2.2.1 Spectral Subtraction

One of the first techniques introduced in the field of speech enhancement is called spectral subtraction [21]. The main idea of spectral subtraction is to obtain an estimate of the magnitude spectrum of the background noise and subtract this estimate from the magnitude spectrum of the combined target speech and background noise. The final result of this computation is an estimate of the target speech's magnitude spectrum which can be used to invert back into the time-domain.

Suppose a target speech signal $x[k]$ and statistically independent additive noise $n[k]$. Then speech corrupted by background noise, $y[k]$, can be represented as follows:

$$y[k] = x[k] + n[k] \tag{2.8}$$

This implies the following in the short-time Fourier domain:

$$X[k, \omega] = Y[k, \omega] - N[k, \omega] \tag{2.9}$$

where $X[k, \omega]$ is the STFT of $x[k]$, $Y[k, \omega]$ is the STFT of $y[k]$, and $N[k, \omega]$ is the STFT of $n[k]$. This can be equivalently expressed in polar form:

$$X[k, \omega] = |Y[k, \omega]| e^{j\phi_y(k,\omega)} - |N[k, \omega]| e^{j\phi_n(k,\omega)} \tag{2.10}$$

where $\phi_y(k, \omega)$ is the phase of $Y[k, \omega]$ and $\phi_n(k, \omega)$ is the phase of $N[k, \omega]$. In practice, it can be shown that the noise-free phase can be estimated by the noisy phase which implies:

$$\phi_y(k,\omega) \approx \phi_n(k,\omega) \tag{2.11}$$

This assumption leads to the following:

$$X[k,\omega] = (|Y[k,\omega]| - |N[k,\omega]|)e^{j\phi_y(k,\omega)} \tag{2.12}$$

Therefore, to obtain an estimate of the STFT of the target speech, $\hat{X}[k,\omega]$, an estimate of the magnitude of the STFT of the noise, $|\hat{N}[k,\omega]|$ is required:

$$\hat{X}[k,\omega] = (|Y[k,\omega]| - |\hat{N}[k,\omega]|)e^{j\phi_y(k,\omega)} \tag{2.13}$$

$\hat{X}[k,\omega]$ can finally be inverted back to the time domain with the help of the overlap-add method of reconstruction to recover an estimate of the target speech, $\hat{x}[k]$.

In practice, $|\hat{N}[k,\omega]|$ can be obtained by sampling the noise during pauses in the speech, computing the STFT of these samples, and then averaging the magnitude spectrums across these sampled STFTs to obtain an estimate of $|N[k,\omega]|$. The main drawback of the spectral subtraction algorithm is the limited ability to obtain a precise estimate of $|N[k,\omega]|$. This is especially a problem for background noise that is non-stationary, such as babble noise as illustrated in the cocktail party problem. A poor estimate of $|N[k,\omega]|$ will tend to cause errors in the subtraction step which can result in remnant noise and speech distortion of the target speech estimate, $\hat{x}[k]$.

2.2.2 Wiener Filter

In the study of LTI systems and filtering, a natural question arises pertaining to finding the minimum-mean-square-error (MMSE) filter of a wide-sense stationary (WSS) input process. This optimal MMSE filter is called the Wiener filter. The derivation for characterizing the Wiener filter (in discrete time) will be given below [22].

Suppose a WSS random process, $x[n]$. The goal is to determine the frequency response characterizing an LTI system, $h[n]$, that outputs a WSS process $\hat{y}[n]$ that is the minimum-mean-square-error (MMSE) estimate of some target process $y[n]$ that is jointly WSS with $x[n]$.

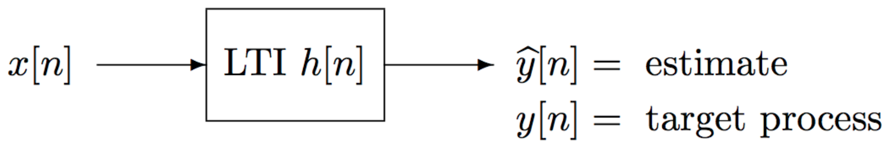

Figure 2.2: A diagram representing an input process, $x[n]$, passing through an LTI system, $h[n]$, that outputs an estimate $\hat{y}[n]$ of the target process $y[n]$ [22].

The error, $e[n]$, between the filter's output, $\hat{y}[n]$, and the target process, $y[n]$, is defined as follows:

$$e[n] \triangleq \hat{y}[n] - y[n] \tag{2.14}$$

An optimization problem can be written down that is solved by finding the LTI filter's impulse response, $h[n]$, (the Wiener filter) that satisfies the following criteria:

$$\underset{h[.]}{\text{minimize}} \quad E\{e^2[n]\} \tag{2.15}$$

First, the error criterion is expanded using the fact that the output of an LTI filter can be expressed as a convolution of its impulse response with the input signal:

$$\epsilon = E\{(\sum_{k=-\infty}^{\infty} h[k]x[n-k] - y[n])^2\} \tag{2.16}$$

The goal is to choose the values of $h[m]$ for all m that minimize this error criterion, ϵ. Multivariate optimization is applied to minimize ϵ by taking the partial derivative of ϵ with respect to $h[m]$ for each m and setting each of these expressions equal to zero.

$$\frac{\partial \epsilon}{\partial h[m]} = E\{2(\sum_{k} h[k]x[n-k] - y[n])x[n-m]\} = 0 \tag{2.17}$$

This implies the following:

$$R_{ex}[m] = E\{e[n]x[n-m]\} = 0 \quad for\ all\ m \tag{2.18}$$

By Equation 2.18 and the definition of orthogonality, it can be concluded that the error signal and the input signal are mutually orthogonal. This orthogonality condition can be equivalently re-written as follows:

$$R_{ex}[m] = E\{e[n]x[n-m]\} = E\{(\hat{y}[n] - y[n])x[n-m]\} = R_{\hat{y}x}[m] - R_{yx}[m] \tag{2.19}$$

Combining the orthogonality condition stated in Equation 2.18 with Equation 2.19, the following statement is true:

$$R_{\hat{y}x}[m] = R_{yx}[m] \quad for\ all\ m \tag{2.20}$$

Equation 2.20 says that the optimal filter's estimate of the target process has a cross-correlation with the input process that is equal to the cross-correlation of the target process' cross-correlation with the input process. Since the estimate, $\hat{y}[n]$ is obtained by inputting the input process $x[n]$ through an LTI filter, the following convolution relationship applies:

$$R_{\hat{y}x}[m] = h[m] * R_{xx}[m] \tag{2.21}$$

Combining Equation 2.20 and Equation 2.21 implies:

$$R_{yx}[m] = h[m] * R_{xx}[m] \tag{2.22}$$

Then taking the z-transform of both sides of Equation 2.22:

$$S_{yx}(z) = H(z)S_{xx}(z) \tag{2.23}$$

where $S_{yx}(z)$ is the cross-spectral density of $y[n]$ and $x[n]$ and $S_{xx}(z)$ is the power-spectral density of $x[n]$. Therefore, the optimal LTI filter in the MMSE sense (the Wiener filter), is characterized by the following equation:

$$H(z) = \frac{S_{yx}(z)}{S_{xx}(z)} \tag{2.24}$$

The Wiener filter tends to perform better than spectral subtraction in practice, but it suffers from the fact that it is constrained to be a linear estimator. A linear estimator may not have enough complexity to remove highly non-stationary background noise.

2.2.3 Ideal Binary Mask Estimation

Another common technique in the field of speech enhancement is based on the concept of an Ideal Binary Mask (IBM) [21]. The idea of an IBM arises from a model for human auditory perception called Auditory Scene Analysis (ASA). ASA can be broken down into two stages. The first stage, called the segmentation stage, involves the decomposition of an input signal into time-frequency units (T-F units). An example of an input signal can be speech or any other type of sound that enters the human auditory system. After the segmentation stage is the second stage called the grouping stage. The grouping stage involves grouping T-F units that are most likely to have been generated from the same source. This model, proposed by Albert Stanley Bregman in 1990, is theorized to model how the human auditory system separates sounds in an input signal mixture. ASA has inspired the field of Computational Auditory Scene Analysis (CASA). CASA's main focus is to find computational means of separating an input signal mixture similar to how a human does so [23]. In a typical CASA system, an input signal is first passed through a gammatone filter bank to generate a T-F representation that mimics the human auditory system. This T-F representation is called a cochleagram. The next goal in a typical CASA system is to use the cochleagram to separate an input signal mixture into groups. For speech enhancement, this process of separation brings up the concept of an Ideal Binary Mask. Put simply, an Ideal Binary Mask is a decision rule that determines whether a T-F unit in the T-F representation is dominated by the noise source or by the target speech. The IBM, $H[n, w]$, is defined as:

$$H[n,w] = \begin{cases} 1, & if \frac{\|X[n,w]\|^2}{\|N[n,w]\|^2} > \boldsymbol{\theta}, \\ 0, & otherwise \end{cases} \tag{2.25}$$

where $\|X[n,w]\|^2$ represents the energy in a speech T-F unit at position $[n,w]$, $\|N[n,w]\|^2$ represents the energy in a noise T-F unit at position $[n,w]$, and $\boldsymbol{\theta}$ is a threshold value. Conceptually, the IBM attempts to remove T-F units in which the noise signal's energy is higher than the speech signal's energy according to some threshold, $\boldsymbol{\theta}$. In theory, an IBM will preserve the T-F units that correspond to the target speech. Though in practice, one will not have direct access to both the target speech and noise sources and therefore an IBM will need to be estimated. Machine learning techniques, such as support vector machines and neural networks, have been employed in the field of IBM estimation in order to classify when a T-F unit has more target speech energy than background noise energy [24].

2.2.4 Performance Evaluation Measures (PESQ, STOI, WER)

Three popular measures for measuring performance of speech enhancement systems are: perceptual evaluation of speech quality (PESQ), subjective short-time objective intelligibility (STOI), and word error rate (WER).

PESQ was introduced as a reliable objective speech quality measurement for communication networks [25]. The key motivation for PESQ was to create a metric that was objective and correlated well with subjective human opinion of speech quality. PESQ computes a number on the range -0.5 to 4.5 called mean opinion score (MOS) that is a function of the target speech and filtered speech. The PESQ model takes the target speech and filtered speech as input and first aligns both signals to a standard listening level. Next, the signals are filtered with an input filter that mimics a standard telephone handset. These filtered outputs are then aligned and processed through an auditory transformation (attempting to mimic the human auditory system). Two distortion parameters are extracted from the disturbance (the difference between the transformed outputs) and mapped to an MOS that gives a measure of speech quality.

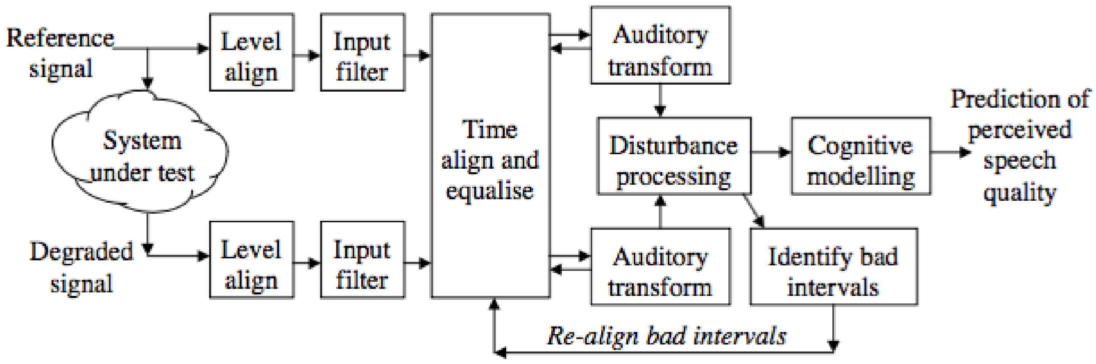

Figure 2.3: A diagram depicting the PESQ model [25]

STOI was introduced as a reliable objective speech intelligibility measurement for speech enhancement systems [26]. As with PESQ, STOI is a function of the target speech and filtered speech that produces a scalar value that is expected to correlate well with the average intelligibility (the percentage of correctly understood words averaged across a group of users) of the target speech. As an overview, STOI can be broken into three stages. The first stage involves a T-F decomposition of both signals corresponding to the properties of the human auditory system. Next, the T-F representations are segmented into short time intervals, normalized appropriately, and a correlation coefficient is computed between corresponding T-F representation intervals of the target speech and filtered speech. Finally, these computed correlation coefficients are averaged across all frequency bands and frames.

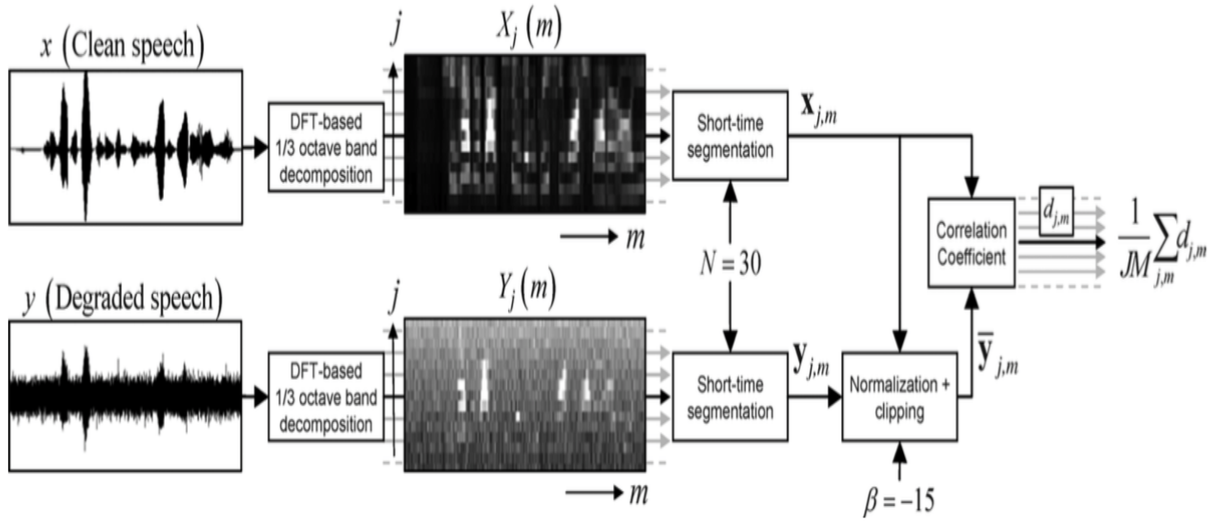

Figure 2.4: A diagram depicting the STOI model [26]

WER is another performance evaluation measure for characterizing intelligibility of speech directly [27]. It is primarily used in speech recognition to get a measure of improvement. It can also be used in speech enhancement to determine if a speech enhancement system improves speech intelligility relative to the degraded speech's intelligibility. In speech enhancement, a speech recognition system would first be applied to the target speech to obtain a predicted text-based word sequence. The target speech's text-based word sequence, call it $y[n]$, will be used as a reference in the WER calculation. Next, the speech recognition system will be applied to the degraded speech signal to obtain a predicted text-based word sequence, call it $y_{noisy}[n]$. Finally, the speech recognition system will be applied to the filtered speech signal (the output of the speech enhancement system when the degraded speech signal is used as input) to obtain a predicted text-based word sequence, call it $\hat{y}[n]$. A WER will be computed for the pair $y[n]$ and $y_{noisy}[n]$, as well as for the pair $y[n]$ and $\hat{y}[n]$. Before actually computing each pair of text-based sequences' WER, each pair of sequences will first be aligned with one another to minimize their edit distance. This minimization of edit distance is usually achieved via Viterbi algorithm. The WER for each pair is computed using the following definition:

$$WER = \frac{S + D + I}{H + S + D} \tag{2.26}$$

where H represents the number of word hits, S represents the number of substitutions, D represents the number of deletions, and I represents the number of insertions. An example using this terminology is below:

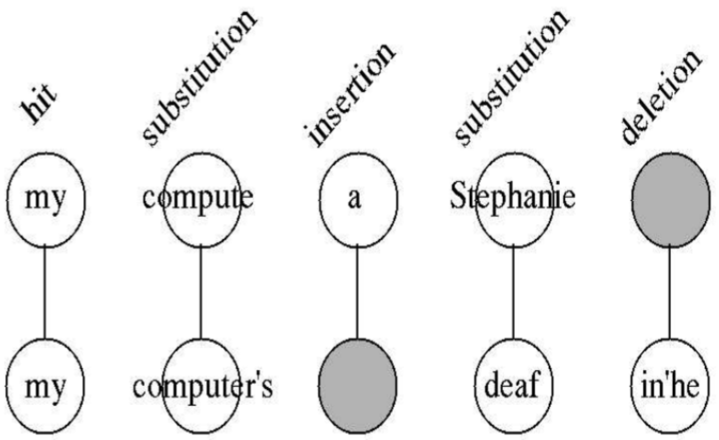

Figure 2.5: An example of calculating the number of hits (H), substitutions (S), deletions (D), and insertions (I) of two aligned text-based word sequences. The top sequence can be interpreted as either $y_{noisy}[n]$ or $\hat{y}[n]$, and the bottom sequence is the reference sequence $y[n]$ [27].

2.3 Machine Learning

The focus of this master's thesis relies on techniques in the domain of deep learning. Deep learning is a subdomain of the larger domain of machine learning. Therefore, this section will briefly review the fundamentals of machine learning.

2.3.1 Definition

Machine learning deals with algorithms that learn to perform a given task through experience. A machine learning algorithm can be viewed as a computer program that performs a task, but is not entirely specified by a computer programmer. Instead, the computer program relies on access to relevant data to specify itself and hence learn to perform the given task. Mitchell (1997) provides a simple, general definition that does a good job of characterizing a machine learning algorithm [28]:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

2.3.2 Example: Linear Regression

Due to the abstract nature of the definition presented in the previous section, a concrete example will prove to be helpful in understanding the terminology and goal of machine learning. One of the simplest examples of a machine learning algorithm is known as linear regression [28]. Linear regression takes a vector $\mathbf{x} \epsilon \mathbb{R}^n$ as input and attempts to predict a scalar output $y \epsilon \mathbb{R}$. The prediction, $\hat{y}$, is defined as follows:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} \tag{2.27}$$

where $\mathbf{w} \in \mathbb{R}^n$ is a vector of parameters.

In machine learning, $\mathbf{x}$ is referred to as a feature vector and y as the corresponding label. Each entry of $\mathbf{x}$ is a real number that is defined in a way that is useful for predicting y. As an example, one can imagine $\mathbf{x}$ containing measurements like age and weight of a person and from this wanting to predict the person's height, y. In linear regression, each entry of $\mathbf{w}$ acts as a coefficient for the corresponding entry of $\mathbf{x}$. In other words, the task, T, is to predict y by learning a linear transformation, parametrized by $\mathbf{w}$, that is applied to $\mathbf{x}$. This linear transformation is learned through experience, E, from a training set, $\mathbf{X}^{(train)} \in \mathbb{R}^{m \times n}$, of m examples with m labels specified as a vector $\mathbf{y}^{(train)} \in \mathbb{R}^m$. Using mean squared error as the performance measure, P, the following equation is sought to be minimized on the training set:

$$MSE_{train} = \frac{1}{m} \|\hat{\mathbf{y}}^{(train)} - \mathbf{y}^{(train)}\|_2{}^2 \tag{2.28}$$

Using tools from multi-variable calculus, the gradient of Equation 2.28 is taken with respect to $\mathbf{w}$ and set equal to zero in order to solve for the optimal parameter vector $\mathbf{w}$ that minimizes the equation.

$$\begin{aligned} \nabla_{\mathbf{w}}\, MSE_{train} &= 0 \\ \nabla_{\mathbf{w}} \frac{1}{m} \|\hat{\mathbf{y}}^{(train)} - \mathbf{y}^{(train)}\|_2{}^2 &= 0 \\ \frac{1}{m} \nabla_{\mathbf{w}} \|\mathbf{X}^{(train)}\mathbf{w} - \mathbf{y}^{(train)}\|_2{}^2 &= 0 \end{aligned} \tag{2.29}$$

Applying matrix algebra will yield the following solution to Equation 2.29:

$$\mathbf{w} = (\mathbf{X}^{(train)^\top}\mathbf{X}^{(train)})^{-1}\mathbf{X}^{(train)}\mathbf{y}^{(train)} \tag{2.30}$$

Linear regression is sometimes referred to as one-step learning due to the closed-form expression found in Equation 2.30. Many other types of machine learning algorithms do not have a closed-form expression and clever methods of optimization are needed. It is worth noting that linear regression is most often used with an intercept (bias) parameter, b, in order to learn an affine transformation rather than a more limited linear transformation as seen in Equation 2.27.

$$\hat{y} = \mathbf{w}^\top\mathbf{x} + b \tag{2.31}$$

where $\mathbf{w} \in \mathbb{R}^n$ is a vector of parameters and $b \in \mathbb{R}$ is a parameter. Equation 2.31 can be written to satisfy the functional form of Equation 2.27 by appending an entry of 1 to every feature vector $\mathbf{x}$ and the derivation for $\mathbf{w}$ follows with the bias b being implicitly included as an entry of $\mathbf{w}$. To conclude, linear regression is a simple, efficient algorithm for practical learning problems. It also acts as an effective introduction to the space of machine learning algorithms.

2.3.3 Unsupervised v.s. Supervised Learning

Machine learning can be broadly categorized into two classes: unsupervised learning and supervised learning [28]. The distinction between unsupervised learning and supervised learning is the type of experience the learning algorithm is allowed to have. In general, machine learning algorithms learn via a dataset of examples pertaining to a task that is to be learned. For example, one of the oldest datasets studied is called the Iris dataset. An example (data point) in the Iris dataset corresponds to measurements taken of a specific Iris plant. The dataset consists of three species of Iris plants and a common task pertaining to the Iris dataset is to classify a given example as one of the three species. The Iris dataset contains an entry for each example specifying which species of Iris plant the example belongs to. If a machine learning algorithm uses this information to accomplish the classification task, this is called supervised learning. The idea is the machine learning algorithm has explicit knowledge that a relationship exists between the measurements of the Iris plants and the species of the Iris plants. The algorithm is in essence being "supervised" during the learning process. If instead only the measurements of each plant are given but not the information specifying each example's species, this task is now referred to as unsupervised learning. The goal is to now cluster the examples together based on their measurements in hopes that this will reveal structure pertaining to the task of species classification.

Unsupervised and supervised learning can be spoken of informally using the language of probability. In unsupervised learning, an attempt is made to learn about the probability distribution $p(\mathbf{x})$ where $\mathbf{x}$ is a random vector corresponding to the features of an example (e.g. measurements of the Iris plant). In supervised learning, an attempt is made to learn about the probability distribution $p(y|\mathbf{x})$ where y is a random scalar (or vector in more general scenarios) corresponding to the labels of the examples (e.g. type of species) and $\mathbf{x}$ is again a random vector corresponding to the features of an example (e.g measurements of the Iris plant). In both of these scenarios, learning about the corresponding probability distributions will help in accomplishing related tasks (e.g. determining an Iris plant's species).

2.3.4 Overfitting v.s. Underfitting

A central challenge in machine learning relates to generalization. In the least, it is important to learn how to accomplish a task on the training set in possession. More importantly though, the goal is to learn how to accomplish the task in general (i.e. for data points outside of the training set). The concepts of overfitting and underfitting relate to this idea of generalization [28]. A machine learning algorithm is said to underfit a given problem when its model complexity is not high enough to estimate the underlying process representing the task. When a machine learning algorithm underfits, it is unable to perform well on the training set and hence will have poor performance generalizing to new data. A machine learning algorithm is said to overfit when its model complexity is too high and hence the algorithm is able to memorize the training set. This training set memorization enables error within the training set to be very low but the algorithm will tend to do perform poorly on new data points. Overfitting does not learn the true underlying process representing the task, but only how to perform the task on a finite number of data points. Many methodologies for quantifying model complexity exist with the most popular being the Vapnik-Chervonenkis dimension [29]. The underlying thought in machine learning is to pick a model complexity that is appropriate for the given task. The belief is there is an optimal point in terms of model complexity for achieving optimal generalization. The example to follow provides an intuitive notion of these ideas.

Suppose a training set of 8 data points in the Cartesian plane, each represented as a coordinate pair (x, y). It is believed there exists a relationship between x and y and the following three linear regression based models are chosen to try and model the relationship:

$$\begin{aligned} \hat{y} &= b + wx \\ \hat{y} &= b + w_1 x + w_2 x^2 \\ \hat{y} &= b + \sum_{i=1}^{9} w_i x^i \end{aligned} \tag{2.32}$$

The first model represents a hypothesis space (model space) of the family of polynomials with degree 1. The second model represents a hypothesis space of the family of polynomials with degree 2. Finally, the third model represents a hypothesis space of the family of polynomials with degree 9. These models clearly are increasing in complexity. Suppose it is known that the true relationship between x and y is quadratic (of course in practice one would not have access to this knowledge). Using the solution derived in Section 2.3.2 to solve for the best parameters for each of the models, one can imagine that the first model will underfit the problem since it does not have enough model complexity to describe the true underlying process. The second model is specified with optimal model complexity and this model will provide optimal generalization error. The third model is too complex and instead will overfit to the data points and thus lead to poor generalization performance. The following figure visually describes this phenomena.

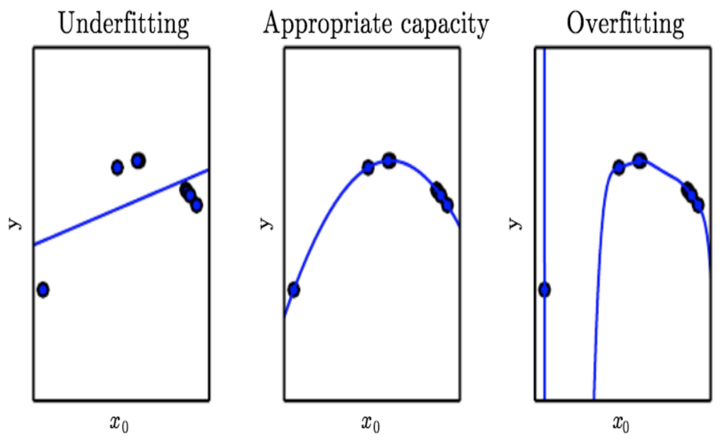

Figure 2.6: (LEFT) Represents the first model's fit, (CENTER) represents the second model's fit, and (RIGHT) represents the third model's fit. [28]

2.3.5 Regularization

The concept of regularization focuses on the issue observed in the previous section involving model complexity and generalization error. Regularization is any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error [28]. The fundamental idea behind regularization is based on the assumption that the model should be as simple as possible without losing predictive power. One of the most popular approaches to regularization is called weight decay. Weight decay directly modifies the cost function of the learning algorithm by adding a penalty that penalizes large parameter values:

$$J(\mathbf{w}) = MSE_{train} + \lambda \mathbf{w}^\top \mathbf{x} \tag{2.33}$$

where λ is a nonnegative real number that specifies how much regularization to impose and $\mathbf{w}$ represents the parameters of the model. When the learning algorithm minimizes $J(\mathbf{w})$ (such as in linear regression), it makes a tradeoff between minimizing the training set error and the size of the parameter values. Smaller parameter values tend to imply a smoother model and decrease the likelihood of overfitting. Suppose the model complexity example discussed in Section 2.3.4 and focus on the third model (family of polynomials of degree 9). Suppose that weight decay is added to the cost function for linear regression and then the best parameters are solved for using a modified form of the closed-form solution discussed in Section 2.3.2. The following figure visually depicts the various models that are fit depending on how much regularization is imposed.

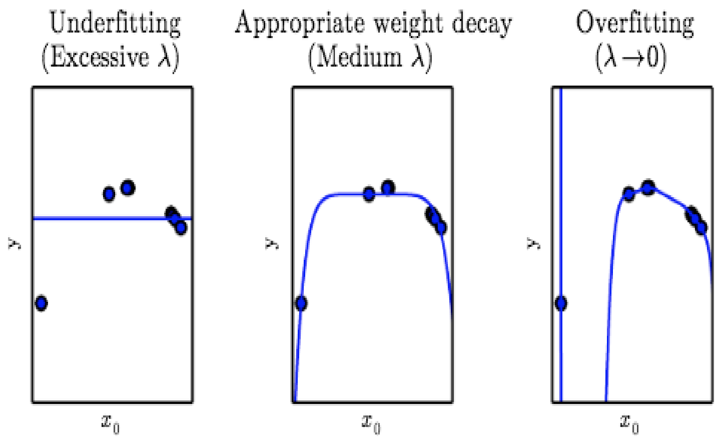

Figure 2.7: Varying λ and its effect on the model that is fit [28]

As λ increases, the linear regression algorithm seeks out a smoother solution that does a good job decreasing training error. This example motivates how regularization can be used as an effective means of limiting model complexity in a systematic manner.

2.3.6 Cross-Validation

Choosing the optimal λ for weight decay requires a systematic procedure. The goal is to choose the λ that results in the lowest generalization error. In practice, it is usually impossible to compute the theoretical generalization error but instead a validation set is used as a proxy for measuring generalization error. Common practice in machine learning is to split the initial training set into a new training set (using 80% of the initial training set) and a validation set (using the remaining 20% of the initial training set). The machine learning algorithm trains on this new training set and the fitted model is tested on the validation set to compute an estimate of the generalization error. Since this estimate is a function of the validation set chosen, the procedure of cross-validation allows for a more stable estimate of the generalization error. The most common form of cross-validation is called k-fold-cross-validation [28]. The idea is to perform the fitting and testing procedure k-times and take the average of the k generalization error estimates obtained. For example, given an initial training set one would randomly partition it into k mutually exclusive and exhaustive subsets. Next, one subset is chosen as the validation set and the other $k-1$ subsets are used as the training set. Once a model is trained on the training set, it is used to compute the error on the validation set. This procedure is repeated for each of the k subsets and the average of the errors computed on each respective validation set is taken as the generalization error estimate.

2.3.7 Principle of Maximum Likelihood

The principle of maximum likelihood gives a principled manner for selecting a model from a given model space [28]. The essential idea is to seek out the model that results in the training set having maximal likelihood of occurring. This is the most common principle used in machine learning to select a model.

Suppose a training set of m examples: $X = \{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ drawn independently from some true but unknown data-generating distribution $p_{data}(\mathbf{x})$ where a given $\mathbf{x}^{(i)}$ is represented as a vector of real numbers of some fixed length. Let $p_{model}(\mathbf{x}; \boldsymbol{\theta})$ be a family of probability distributions parametrized by $\boldsymbol{\theta}$. For a fixed $\boldsymbol{\theta}$, $p_{model}(\mathbf{x}; \boldsymbol{\theta})$ estimates $p_{data}(\mathbf{x})$. The maximum likelihood estimator for $\boldsymbol{\theta}$ is defined as:

$$\begin{aligned} \boldsymbol{\theta}_{ML} &= \arg\max{}_{\boldsymbol{\theta}}\ p_{model}(\mathbf{X}; \boldsymbol{\theta}) \\ &= \arg\max{}_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{model}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) \end{aligned} \tag{2.34}$$

Equation 2.34 specifies a criterion for selecting the parameter vector $\boldsymbol{\theta}$ that maximizes the likelihood of the training set. Due to Equation 2.34 being prone to numerical underflow (i.e. small probabilities being multiplied), it can be reformulated as follows:

$$\boldsymbol{\theta}_{ML} = \arg\max{}_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log(p_{model}(\mathbf{x}^{(i)}; \boldsymbol{\theta})) \tag{2.35}$$

Equation 2.35 can be rescaled such that each term is weighted by the frequency it occurs in the training set (this does not affect the arg max). This is done such that the maximization process can be viewed as minimizing the dissimilarity, as measured by KL divergence, between the empirical distribution $\hat{p}_{data}(\mathbf{x})$ defined by the training set and the model distribution $p_{model}(\mathbf{x};\boldsymbol{\theta})$.

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} E_{x\sim\hat{p}_{data}} \log(p_{model}(\mathbf{x}^{(i)};\boldsymbol{\theta})) \tag{2.36}$$

In other words, Equation 2.36 is saying that the principle of maximum likelihood is equivalent to minimizing the cross-entropy between the empirical distribution and the model's distribution. For example, using mean squared error as the cost function for linear regression, as discussed in Section 2.3.2, is equivalent to minimizing the cross-entropy between the empirical distribution and a homoscedastic, Gaussian probability model. The principle of maximum likelihood therefore provides a justification for using mean squared error as the cost function in linear regression and many other machine learning algorithms.

2.3.8 Bias-Variance Tradeoff

As discussed in Section 2.3.4, the tradeoff between overfitting and underfitting deals with the issue of model complexity. One should choose a model that is complex enough to describe the underlying task and generalize well to new data. If the model complexity is too low for the corresponding task to be learned, performance on the training set will be sub-optimal and hence generalizing will be out of the question. If the model complexity is too high, this increases the likelihood of overfitting to the training set, in essence memorizing the training set and hence leading to poor generalization. This tradeoff described can be analyzed through another theoretical abstraction called the bias-variance tradeoff [29]. The idea is to decompose the theoretical generalization error into two components: bias and variance. First, suppose the goal is to learn an estimate, $g^{(D)}$, of some function, f, describing an underlying task corresponding to a dataset, D, of N examples. Each example is assumed to be independently drawn from some joint probability distribution $p(\mathbf{x}, y)$ where $\mathbf{x} \epsilon \mathbb{R}^n$ and $y \epsilon \mathbb{R}$. The function, f, describes the true relationship between the input vector $\mathbf{x}$ and output label y. The expected out-of-sample mean squared error, E_{out}, dependent on the estimate, $g^{(D)}$, is defined as follows:

$$E_{out}(g^{(D)}) = E_{\mathbf{x}}[(g^{(D)}(\mathbf{x}) - f(\mathbf{x}))^2] \tag{2.37}$$

where the expectation operator $E_{\mathbf{x}}$ is taken over the underlying probability distribution on $\mathbf{x}$, $p(\mathbf{x})$. In order to define the full expected out-of-sample mean squared error that takes into account all possible realizations of the dataset D, integration is to be performed over the probability distribution $p(\mathbf{x}, y)$ that generates the realized dataset D. This is taken into account by applying the expectation operator E_D over $p(\mathbf{x}, y)$ on Equation 2.37.

$$\begin{aligned} E_D[E_{out}(g^{(D)})] &= E_D[E_{\mathbf{x}}[(g^{(D)}(\mathbf{x}) - f(\mathbf{x}))^2]] \\ &= E_{\mathbf{x}}[E_D[(g^{(D)}(\mathbf{x}) - f(\mathbf{x}))^2]] \end{aligned} \tag{2.38}$$

The expectation operators E_D and $E_{\mathbf{x}}$ in Equation 2.38 can be swapped due to the rules of multi-variable integration. Next, attention is focused on the expression inside $E_{\mathbf{x}}$ on the right-side of Equation 2.38. The goal is to decompose this expression into competing terms that provide an understanding of the tradeoff in generalization error encountered with increased model complexity. To enable this type of decomposition, first a function called the average hypothesis, $\bar{g}$, is defined:

$$\bar{g}(\mathbf{x}) = E_D[g^{(D)}(\mathbf{x})] \tag{2.39}$$

$\bar{g}$ can be interpreted as the average learned function obtained by taking a weighted average point-wise of learned functions $g^{(D)}$ across all possible datasets, D. Using this definition of an average hypothesis and some algebraic manipulations (removed for brevity but can be accessed via the reference [29]), the following decomposition is obtained:

$$E_D[(g^{(D)}(\mathbf{x}) - f(\mathbf{x}))^2] = E_D[(g^{(D)}(\mathbf{x}) - \bar{g}(\mathbf{x}))^2] + (\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2 \tag{2.40}$$

Motivated by the additive decomposition above, the following terms are defined:

$$\begin{aligned} var(\mathbf{x}) &= E_D[(g^{(D)}(\mathbf{x}) - \bar{g}(\mathbf{x}))^2] \\ bias(\mathbf{x}) &= (\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2 \end{aligned} \tag{2.41}$$

With the terms defined in Equation 2.41, Equation 2.40 can be re-written as follows:

$$E_D[(g^{(D)}(\mathbf{x}) - f(\mathbf{x}))^2] = var(\mathbf{x}) + bias(\mathbf{x}) \tag{2.42}$$

Using the decomposition provided by Equation 2.42 and returning to the equation for generalization error provided by Equation 2.38, the final decomposition of interest is obtained:

$$\begin{aligned} E_D[E_{out}(g^{(D)})] &= E_{\mathbf{x}}[E_D[(g^{(D)}(\mathbf{x}) - f(\mathbf{x}))^2]] \\ &= E_{\mathbf{x}}[var(\mathbf{x}) + bias(\mathbf{x})] \\ &= var + bias \end{aligned} \tag{2.43}$$

The term *var* in Equation 2.43 gives a measure of the average squared “distance” of the learned function estimated by a fixed, realized dataset, D, from a best hypothetical average function, $\bar{g}$, that is obtained using all realizations of D. The term *bias* in Equation 2.43 gives a measure of the average squared "distance" the best hypothetical average function $\bar{g}$ is from the true underlying function, f. Conceptually, if model complexity is very high this will tend to imply lower generalization error due to *bias* since $\bar{g}$ will tend to have the ability to approximate f. In turn, this will tend to lead to the tradeoff of having higher generalization error due to *var* since the learned estimate $g^{(D)}$ will tend to be very sensitive to the realized dataset, D. Likewise if model complexity is very low, this will tend to imply higher generalization error due to *bias* since the best hypothetical average function $\bar{g}$ will have a difficult time approximating the underlying function, f. In turn, this will tend to lead to the tradeoff of having lower generalization error due to *var* since the learned estimate $g^{(D)}$ will be less sensitive to the realized dataset, D. The bias-variance tradeoff gives a theoretical tool for formalizing the need for choosing model complexity appropriately when applying machine learning.

2.3.9 Bayesian Inference

Many machine learning algorithms rely on a frequentist's perspective on statistics. This type of approach involves finding one optimal set of parameters (based on some specified criterion such as minimizing mean squared error) in a specified model's parameter space. The principle of maximum likelihood, discussed in Section 2.3.7, is a form of this type of approach. A frequentist approach tends to make the assumption that there exists one true set of parameters, call it $\boldsymbol{\theta}$, underlying the task to be learned. The learned estimate, $\hat{\boldsymbol{\theta}}$, is a random variable that is a function of the realized dataset, D. The fundamental idea in Bayesian inference is to view all sets of the parameters in a specified model's parameter space as possible and use the dataset, D, to narrow in on the most likely ones [28]. This is usually done by first specifying a prior probability distribution, $p(\boldsymbol{\theta})$, on the parameters of the specified model space. This prior probability distribution is then updated to a new probability distribution, called the posterior distribution $p(\boldsymbol{\theta}|D)$. This posterior distribution represents a learned collection of models for future predictions (each weighted in importance based on $p(\boldsymbol{\theta}|D)$ rather than just one model as in the frequentist approach. A Bayesian approach to statistics develops a full probabilistic framework (using Bayes' law) for machine learning. Bayesian inference can be formalized with the following two equations. The first equation formalizes how to use Bayes' law of probability to update the prior distribution $p(\boldsymbol{\theta})$ given a dataset, $D = \{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$.

$$p(\boldsymbol{\theta}|\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}) = \frac{p(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)})|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)})} \tag{2.44}$$

The second equation formalizes how to compute the predictive distribution for a future data point, $\mathbf{x}^{(m+1)}$.

$$p(\mathbf{x}^{(m+1)}|\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}) = \int p(\mathbf{x}^{(m+1)}|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)})d\boldsymbol{\theta} \tag{2.45}$$

In practice, the given framework above tends to be computationally expensive. The idea of regularization, discussed in Section 2.3.5, can be viewed as a computationally feasible way of blending a Bayesian approach with a frequentist approach to machine learning.

2.4 Deep Learning

The approach to be discussed in this master's thesis employs a technique from the domain of deep learning. This technique is called convolutional neural networks. Therefore, this section will briefly review the fundamentals of deep learning.

2.4.1 Motivation

Deep learning was motivated by traditional machine learning algorithms failing to generalize well on artificial intelligence tasks, such as speech recognition and object recognition [28]. With the advent of increasingly larger datasets and higher computational power, deep learning has recently gained much traction for solving problems in the world of machine learning. Fundamentally, deep learning relies on the use of complex models, called neural networks, that assume many tasks in the natural world are generated by a composition of factors. The layers of a neural network represent a composition of functions that aim to learn the given composition of factors that generated the specific task. This composition of functions can be highly nonlinear and allows for approximating highly complex underlying functions. In practice, these models tend to generalize well to new data points given enough training data.

2.4.2 Deep Feedforward Networks

Deep feedforward networks are the most fundamental class of models in deep learning [28]. In general, the goal of neural networks is to approximate some function, f, that underlies a specific task (e.g. object recognition). A feedforward network can be defined as a mapping $\hat{y} = \hat{f}(\mathbf{x}; \boldsymbol{\theta})$ where $\mathbf{x}$ represents an input vector, $\hat{y}$ represents a scalar (or output vector in more general scenarios) estimating some true label y, and $\boldsymbol{\theta}$ represents the parametrization of $\hat{f}$ that is an approximation of f. This class of models is called feedforward because information propagates from the input of the network, through intermediate computations that define $\hat{f}$, and finally to the estimate $\hat{y}$. The use of the term networks corresponds to a model typically being represented as a composition or connection of many different functions. For example, suppose the following three functions: $\hat{f}^{(1)}$, $\hat{f}^{(2)}$, and $\hat{f}^{(3)}$. These functions can be composed in such a manner such that the deep feedforward network represents the function $\hat{f}(\mathbf{x}) = \hat{f}^{(3)}(\hat{f}^{(2)}(\hat{f}^{(1)}(\mathbf{x})))$. This network is referred to as having three layers and the term "deep" comes from this notion of depth that refers to the number of layers. As more interestingly behaved functions are chained together, the network is able to learn complex functions for approximating difficult tasks. Once a compositionally defined function $\hat{f}$ is defined parametrically, an optimal set of parameters can be found by using gradient-based optimization on a specified loss function (i..e. using the principle of maximum likelihood). The goal during training is to drive the function output of the network $\hat{f}(\mathbf{x})$ closer to $f(\mathbf{x})$ for each $\mathbf{x}$ in the training set. The output layer of the network is being forced to produce a result close to the label $y = f(\mathbf{x})$, but the intermediary computations done in previous layers are learned by the network. The intermediary computations are done between the input layer and output layer and are usually called the hidden layers. A fundamental question in deep learning relates to determining which parametrized families of functions are most useful for learning to accomplish real world tasks. One of the most commonly used parametric families, inspired by biological neurons, is the application of an affine transformation applied to the input vector followed by an element-wise nonlinearity, such as the sigmoid function. This type of layer is called a fully-connected layer. Let ϕ be a function from this parametric family and suppose an input vector $\mathbf{x} \in \mathbb{R}^n$. This function is formally defined as follows:

$$\phi(\mathbf{x}) = \sigma(\mathbf{W}^\top \mathbf{x} + \mathbf{b}) \tag{2.46}$$

where $\mathbf{W} \in \mathbb{R}^{n \times m}$, $\mathbf{b} \in \mathbb{R}^n$, and $\sigma(z) = \frac{1}{1+e^{-z}}$ is applied element-wise. The nonlinearity σ in Equation 2.47 is referred to as an activation function in deep learning because it "activates" dependent on the value of the affine transformed input signal. The affine transformed input signal can be viewed as weighing the different vector entries of the input signal dependent on the task at hand. Another activation function widely used in deep learning is called the rectified linear unit (ReLU).

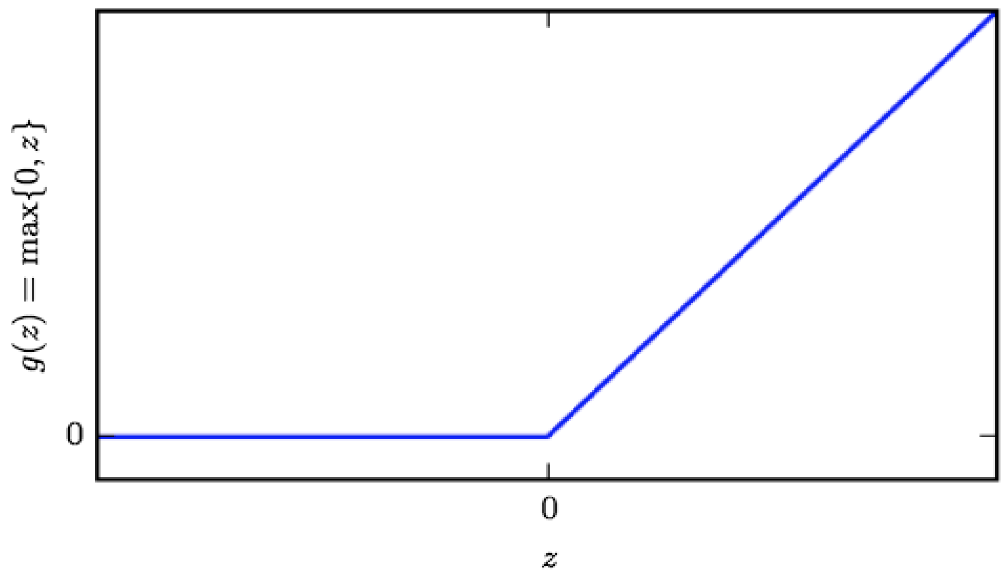

Figure 2.8: Graph of the rectified linear unit (ReLU) [28]

The class of deep feedforward networks provides a solid framework for learning arbitrarily complex models in a systematic fashion. A theoretical result known as the universal approximation theorem supports the use of these models [28]. Informally, it states that a single hidden layer feedforward network of sufficient functional complexity can approximate any continuous function on $\mathbb{R}^n$.

2.4.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a specialized kind of neural network for processing input data that has an inherent grid-like topology [28]. Generally, the input data to a CNN will have natural structure to it such that nearby entries are correlated. Examples of this type of data are 1-D audio time series data and 2-D images. The more formal definition of a CNN is a neural network that uses convolution in place of general matrix multiplication in at least one of its layers. Convolution of two discrete vectors, $x[n]$ and $w[n]$, is defined as follows:

$$y[n] = (x * w)[n] = \sum_{k=-\infty}^{\infty} x[k]w[n-k] \tag{2.47}$$

Convolution can be interpreted as fixing one vector in place, striding the other vector along it, and for each stride a dot product is computed. Each dot product produces one number that is an entry in the output vector. In the case of CNNs, a convolutional layer is generally composed of three stages. The first stage involves parametrized, learnable filters each performing a convolution in parallel. This convolution operation can be modified from the definition in Equation 2.47 in different ways, such as how much to stride before computing another dot product. The second stage involves an element-wise non-linearity similar to a fully-connected layer. Finally, the third stage is called pooling. Pooling is a method of downsampling the output vector of the second stage. One way to do this is called max-pooling in which the maximal element in a defined section of the output is taken to represent the entire section. To summarize, a convolutional layer tries to find local patterns in the input. Each filter in the first stage is learned during training in such a way that is task specific. In other words, the CNN attempts to find the most relevant patterns that help determine how to accomplish the given task. An equivalent way of thinking about CNNs is by imagining a convolutional layer as being a fully connected layer with an infinitely strong prior that says weights are shared across input data entries and a majority of them are zero [28]. In addition, it is important to note that the ideas discussed here generalize to N-dimensional, finite tensors.

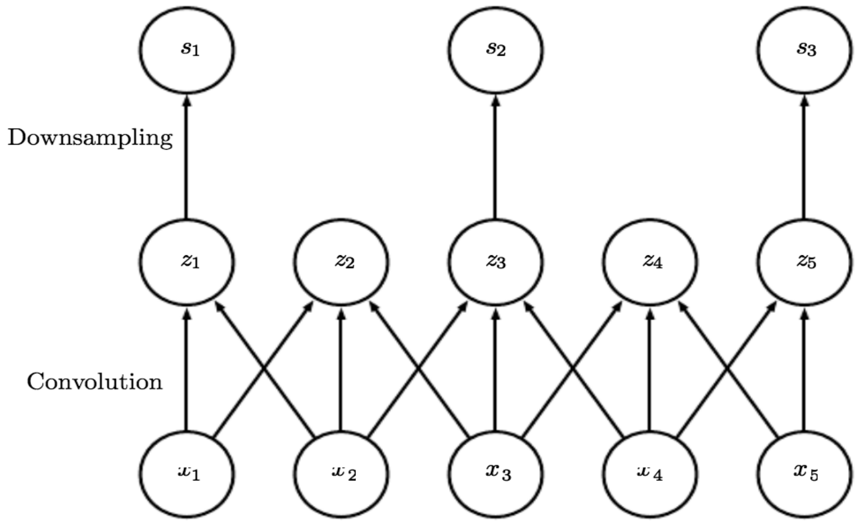

Figure 2.9: An example of a convolutional layer with a filter of kernel size 3 applied to an input vector $\mathbf{x}$. First, convolution is performed and a nonlinearity is applied element-wise to produce the vector $\mathbf{z}$. Then $\mathbf{z}$ is downsampled via a pooling operation to produce the final output vector $\mathbf{s}$ [28].

In practice, CNNs have been a powerful neural network architecture for object recognition and other related tasks. This can be attributed to CNNs having a neuroscientific basis in how the mammalian visual system works.

2.4.4 Gradient-based Optimization

A neural network is a function, $\hat{f}(\mathbf{x};\boldsymbol{\theta})$, from some specified parametric family defined by $\boldsymbol{\theta}$ that approximates some true underlying function, f. This approximation, $\hat{f}(\mathbf{x};\boldsymbol{\theta})$, is found by minimizing some defined cost function $J(\boldsymbol{\theta})$. The cost function is typically written as an average over the training set as follows:

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} L(\hat{f}(\mathbf{x}^{(i)};\boldsymbol{\theta}), y^{(i)}) \tag{2.48}$$

where N is the number of training examples, L is the per example loss function, $\hat{f}(\mathbf{x}^{(i)};\boldsymbol{\theta})$ is the output of the neural network with $\mathbf{x}^{(i)}$ as input and parameters $\boldsymbol{\theta}$, and $y^{(i)}$ is the corresponding label. Due to $J(\boldsymbol{\theta})$ generally being highly non-convex in the context of deep learning, a technique called gradient-based optimization is applied in order to minimize it [28]. Conceptually, results from multi-variable calculus show that the gradient of a function points in the direction of local maximal increase. It can be shown that the direction opposite this will point toward local maximal decrease. Therefore, by "following" the gradient by taking small steps in its opposite direction inside the input parameter space, $J(\boldsymbol{\theta})$ is slowly minimized until a good set of parameters are found to use to approximate the underlying function f. Formally, the parameter vector $\boldsymbol{\theta}$ is updated to a new parameter vector $\boldsymbol{\theta}'$ in an iterative, step-wise fashion:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \epsilon\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.49}$$

where ϵ is a positive real number referred to as the learning rate (typically between 0.001 - 0.1). In practice, as the number of examples N gets large, Equation 2.49 becomes computationally infeasible due to Equation 2.48. Therefore, a variant to gradient descent called stochastic gradient descent (SGD) is introduced. The key idea to SGD is for every iteration m examples are sampled from the N examples in the training set such that $m << N$. The idea here is to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data-generating distribution. By doing this, an estimate of the true gradient is obtained at a tremendously reduced computational cost compared to using all N examples from a possibly very large training set. In practice, SGD or some variant of SGD is generally always used without loss of optimization accuracy and improved optimization efficiency [30]. Here's the formal algorithm description for SGD [28]:

Algorithm 1 Stochastic gradient descent (SGD) update at training iteration k

```
Require: Learning rate $\epsilon_k$
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\mathbf{x}^{(1)}, ...., \mathbf{x}^{(m)}\}$
    with corresponding labels $y^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{n=1}^{m} L(\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
  end while
```

SGD is a useful gradient-based optimization algorithm. Many variants of SGD have been introduced, such as the Adam algorithm, that attempt to increase optimization performance [31]. The Adam algorithm uses information from the first and second moment of the gradient while also incorporating historical values of the gradient moments in order to achieve a momentum aspect. The momentum aspect is achieved by weighing current gradient moment estimates with an exponentially decaying history of the gradient moment estimates. The use of momentum aids in creating a smoother traversal in the input parameter space of a possibly very complex multivariate function. Adam can be viewed as an adaptive learning algorithm that adapts its learning rate for each parameter in accordance with the functional landscape that is encountered during optimization. Below is a formal algorithm description for the Adam algorithm [28].

Algorithm 2 The Adam algorithm

```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in [0, 1).
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$.
  Initialize time step $t = 0$.
  while stopping criterion not met do
      Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, ...., \boldsymbol{x}^{(m)}\}$ with
      corresponding labels $y^{(i)}$.
      Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{n=1}^{m} L(\hat{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
      $t \leftarrow t + 1$
      Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\, \hat{\boldsymbol{g}}$
      Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\, \hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}$
      Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-{\rho_1}^t}$
      Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-{\rho_2}^t}$
      Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

2.4.5 Regularization & Early Stopping

As in machine learning, regularization in deep learning deals with techniques that seek to improve generalization error possibly at the cost of increased training error. Some of the most popular regularization techniques deal with the concept of parameter norm penalties. These penalties are added to a specified cost function that is to be optimized, generally via some gradient descent based algorithm. The central idea here is to limit the effective capacity of a parametric model based on the prior belief that the parameter values should not be very large. Suppose a cost function $J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$ where $\boldsymbol{\theta}$ parametrizes a neural network, $\boldsymbol{X}$ corresponds to a training set of feature vectors, and $\boldsymbol{y}$ corresponds to the associated labels. A parameter norm penalty $\Omega(\boldsymbol{\theta})$ is added to the objective function, J, to get a new regularized objective function called $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \Omega(\boldsymbol{\theta}) \tag{2.50}$$

The most common parameter norm penalty is the L^2-norm. Another option is the L^1-norm, which also acts as a feature selection method since it strongly biases parameter values toward zero if they do not contribute enough meaningful information to the given task.

Another more general form of regularization is the technique of early stopping. Deep learning relies on gradient descent on a defined cost function over a training set. In general, the cost on the training set is to be minimized with the hope that this leads to a solution that generalizes well to the given task. This methodology is prone to overfitting since deep learning models tend to have a high effective capacity in terms of parameter space. Early stopping seeks to improve this methodology by computing the cost on a separate validation set every fixed number of training iterations. If the cost on the validation set is seemingly not decreasing, this implies that the generalization error is most likely not improving and that training should be halted. The model parameters from the best iteration with respect to validation set cost are saved.

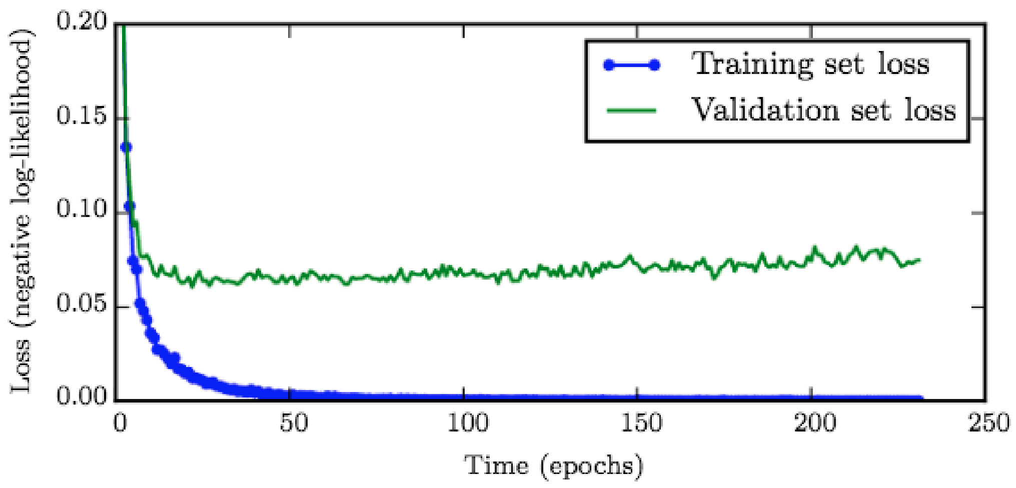

Figure 2.10: Learning curves showing how the loss (cost) on the training set and validation set change over training epochs when training a neural network on the famous MNIST dataset. Notice the training loss continues to decrease through 200 epochs, but the validation loss reaches a minimum value before 50 epochs [28].

Early stopping can be applied to any deep learning model and is extremely powerful in practice. Another thing to note is the cost computed on the validation set need not be the same cost function used for minimization on the training set. The cost function needed to be differentiable on the training set in order to use gradient-based optimization, but the cost function on the validation set need not be differentiable. Instead, a cost function on the validation set that more closely gives a measure of task performance can be used, such as accuracy in binary classification.

2.4.6 Batch Normalization

An issue that arises when training deep neural networks is known as internal covariate shift. The idea is that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This tends to slow down training, requires tuning learning rates appropriately, and being careful with parameter initialization. In practice, this problem has been shown to make training deep networks difficult. A method for solving this issue is called batch normalization [32]. Batch normalization relies on normalizing layer inputs for each training mini-batch that is seen. By performing this normalization, tuning learning rates becomes increasingly unnecessary as well as setting correct parameter initializations. Formally, the batch normalization transform is described as follows [32]:

Algorithm 3 Batch normalization transform, $\hat{y}_i$, of an activation x_i

```
Require: $m$ instances of a layer activation across a mini-batch, B = $x_{1...m}$
Require: Parameters to be learned: $\gamma$, $\beta$
Require: $\epsilon$ (a small positive constant for numerical stability)
    $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
    $\sigma^2{}_B \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
    $\hat{x}_i \leftarrow \frac{\hat{x}_i - \mu_B}{\sqrt{\sigma^2{}_B + \epsilon}}$
    $\hat{y}_i \leftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$
```

Once a network is trained, the batch normalization transform uses population statistics of the training set for computation during inference time. In practice, the use of batch normalization has shown to decrease training times and regularize models leading to improved generalization.

Chapter 3

A Fully Convolutional Neural Network Approach

Having covered the necessary background, this section will now describe this master's thesis's novel approach to speech enhancement that uses a fully convolutional neural network (FCN) architecture. The first part will discuss how the approach was motivated. The next part will discuss how the system was designed. The final two parts of this section deal with testing the speech enhancement system and present results in terms of the performance measures PESQ and WER [33] - [36].

3.1 Motivation

Deep learning has been very successful in learning how to complete complex tasks, such as speech recognition, object recognition, and speech enhancement. Many of the deep learning approaches in speech enhancement require feature extraction before inputting into a neural network, such as a denoising autoencoder or convolutional neural network. The most popular methodology is to use spectrograms as input to the neural network [35]. The drawback here is the extra prior computation of the STFT and loss of phase information at the output.

More recent deep learning approaches have considered an end-to-end approach to speech enhancement that requires no feature extraction [12] - [14]. The idea is to use the noisy time-domain audio signal as input to a neural network and obtain a filtered time-domain audio signal at the output. This methodology rids the need for a prior STFT computation and retains phase information at the output. This recent push in the deep learning community towards end-to-end speech enhancement systems is one of the motivations for this master's thesis's approach. The other large motivation comes from two papers dealing with the study of CNNs on raw audio data. In the first paper [15], the authors make a strong case for the lack of need for fully connected layers in a neural network that processes raw audio data at the input. Instead, they recommend the use of convolutional layers in order to maintain local correlations in the signal as it passes through the network. In addition, a fully convolutional network (i.e. a CNN with no fully connected layers) will generally have much fewer parameters than a correspondingly similar network that includes fully connected layers. This reduced model complexity is especially important for real-time application of the speech enhancement algorithm. In the second paper [16], the authors provide insight into the inner workings of convolutional layers applied to raw audio data. They make a strong case for the lack of need for pooling layers and emphasize the convolution theorem:

$$x * h = \mathcal{F}^{-1}\{\mathcal{F}\{x\} \cdot \mathcal{F}\{h\}\} \tag{3.1}$$

where x can be viewed as the input audio signal, h is a learned filter, and $\mathcal{F}$ is the Fourier transform operator. The convolution theorem allows an FCN to be viewed as a large, nonlinear filter bank. By maintaining the size of the raw audio input vector throughout intermediary computations, each filter's output can be viewed as providing a nonlinear filtered representation of the input vector. As the depth of the FCN increases, a larger number of nonlinear filtered representations is achieved. At the final filtering layer, these representations are combined in a matter that rids the input signal of the background noise representations and only keeps the target speech representations. With the motivation of the approach in mind, the next section will provide specific details of the system design.

3.2 System Design

The first step in designing the speech enhancement system is gathering data for training and validation purposes. An openly available audiobook (narrated by a speaker named Pamela) found online serves as the target speech for designing the system [37]. In addition, babble noise audio clips were found online to serve as background noise when additively combined to Pamela's speech [39] - [40]. All of these audio clips were downsampled to 16 kHz and to have only one audio channel (taking the element-wise average of the two channels if necessary). Table 3.1 concisely describes this data and how it is split for training and validation.

	Target Speech	Babble Noise	SNR	Time (Min:Sec)
Training Set	Chapter 1	Bar Noise	5 dB	35:37
Validation Set	Chapter 2	Cafe Noise	5 dB	5:04

Table 3.1: Data collection and splitting for system design purposes. Target speech refers to Pamela's narration of Chapters 1 - 2 in [37]. Babble noise refers to two different environments found online [39] - [40]. Each set of target speech is additively combined with its corresponding set of babble noise at an SNR of 5 dB.

It is important to note that the system is being designed around a single speaker (i.e. Pamela) and a single SNR of 5 dB. The reason for doing this is to first find a few reasonable FCN architectures for the task of denoising Pamela's speech that has been corrupted by babble noise at an SNR of 5 dB. After choosing a subset of FCN architectures, further exploration will be done for denoising Pamela's speech at SNRs of 0 dB and -5 dB in order to choose one FCN architecture for the system. Once a single FCN architecture is selected and fixed, the next step will involve exploring generalization to a new speaker and the system's robustness to different signal-to-noise ratios.

Having decided on training and validation data, designing the system relies on three main things: (1) a methodology for pre-processing the raw audio data for input into the FCN, (2) fixing an FCN architecture, (3) a methodology for post-processing the raw audio data that is output by the FCN. To begin, let's discuss (1). First, target speech is additively combined with its corresponding babble noise. Next, based on stationarity assumptions for speech, the noisy audio data is split into 20 ms frames in which consecutive frames overlap by 50%. Each noisy frame is to be multiplied by a corresponding Hanning window of equal length. To complete the pre-processing methodology, for each noisy frame the mean of the entire training target speech is element-wise subtracted and standard deviation of the entire training target speech is element-wise divided. Having fixed the methodology for pre-processing, the methodology for post-processing, (3), can also be fixed if it is assumed the FCN outputs a preprocessed filtered 20 ms frame of the preprocessed noisy 20 ms input frame. Given an output frame from the FCN, it is element-wise multiplied by the standard deviation of the entire training target speech and the mean of the entire training target speech is added element-wise. Next, the output from the FCN for the next frame (keeping in mind that these two consecutive frames overlap by 50%) is obtained and the same thing is done. Finally, the overlap-add method of reconstruction is applied to undo the Hanning window that was applied to both overlapping frames. This results in having reconstructed 30 ms of filtered raw audio data that is ready for playback. This post-processing methodology can be iteratively done for an arbitrary amount of noisy input audio data. The most important part of the system design deals with (2), fixing an FCN architecture.

Since a fully convolutional neural network contains only convolutional layers, the only things to be determined are how deep the network needs to be and the details of each layer (i.e. number of filters, kernel size, etc.). The output of the network is to be a one dimensional vector of the same length as the 20 ms input vector, so two things can be immediately concluded upon. The first conclusion is to use “same” padding in all layers to ensure the temporal length of the input vector remains the same throughout intermediary computations and therefore at the output. This allows the FCN to be viewed as a nonlinear, filter bank. The second conclusion is the output layer will be a convolutional layer with one filter and no activation function. This output layer is suitable for reconstructing audio data from its nonlinear representations (i.e. the output layer is able to match the range in which audio data exists) and allows for an output that is one-dimensional. Next, the kernel size for all filters in the network will initially be fixed at 5 ms in length, i.e. 25% of the input size. This can be tuned later on via the validation set. In addition, a dilation factor will not be used in any convolutional layer in order to better preserve local correlations. The structure of each hidden layer will be the following: convolution operation, batch normalization, and ReLU (or PReLU) activation. Batch normalization between the convolution operation and activation function tends to improve training time and generalization performance [32]. Also, the ReLU (or PReLU) activation tends to work well in general CNN practice [28]. With all of this covered, the only thing left to determine is how many hidden layers are necessary and how many filters per hidden layer. These hyperparameters will be determined by training different FCN architectures and comparing MSE performance on the validation set. All models to follow are trained with Adam SGD. In addition, all training employs early stopping that terminates after 20 epochs of no improvement and returns the parameters of the model with best validation loss during training.

To begin, an architecture with one hidden layer is trained to find the number of filters needed in a layer. The number of filters is slowly increased and with each new number of filters a model is trained and validation loss computed. This process provides an understanding of how much complexity, in terms of number of filters per layer, is needed for this task.

Number of Filters	Training Loss (MSE)	Validation Loss (MSE)
50	0.0401	0.0541
100	0.0384	0.0512
200	0.0398	0.0519
300	0.0383	0.0504
400	0.0372	0.0496
500	0.0377	0.0490
600	0.0381	0.0504
700	0.0384	0.0503
800	0.0375	0.0507
900	0.0391	0.0497
1000	0.0373	0.0498

Table 3.2: This table describes training loss & validation loss (MSE) of single hidden layer FCN architectures as the number of filters increases.

Table 3.2 shows that increasing the number of filters marginally helps reduce training loss and validation loss. With this result, a similar procedure will be followed to gain an understanding of how the depth of the network improves performance. The procedure involves first fixing the number of filters to be either 50, 100, or 200 filters per layer and then the depth of the network is increased.

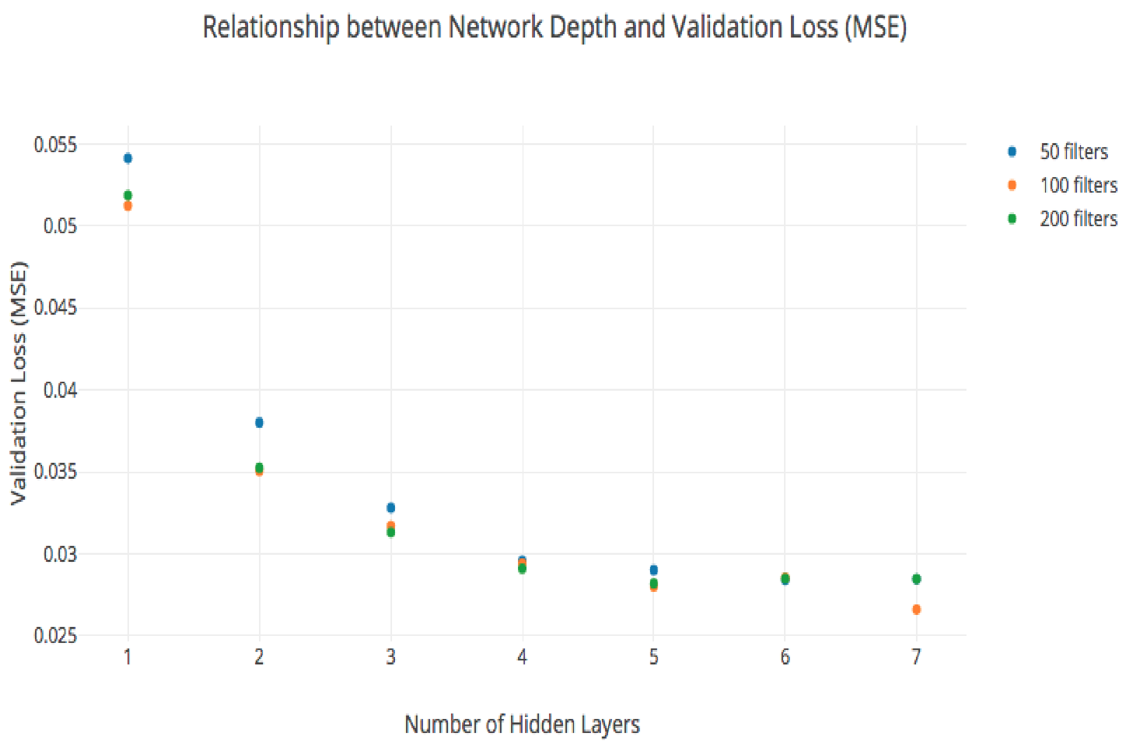

Figure 3.1: A plot that shows increasing network depth leads to large decreases in validation loss. Each curve was generated by fixing the number of filters per hidden layer (either 50, 100, or 200) and increasing the number of hidden layers in the network.

From Figure 3.1, it can be concluded that increasing depth largely helps decrease validation loss as compared to increasing the number of filters in a given layer. With the knowledge that between 1 to 200 filters in a layer and a depth of 5 to 6 layers tends to provide good validation loss, different network architectures are experimented with within this range. Architectures with different kernel sizes, activation functions, number of filters and regularization were experimented with with the goal of minimizing validation loss. Over 70 FCN architectures were trained and validated in total. From the over 70 FCN architectures, the top 13 FCN architectures that provide the lowest validation loss are taken to be studied further.

These architectures are detailed in Appendix A. The PESQ and WER of the filtered validation set is computed for each of the 13 architectures and compared to the PESQ and WER of the noisy validation set (at an SNR of 5 dB). The goal is to pick an architecture that provides a high PESQ (good speech quality), low WER (good intelligibility), and has a small number of parameters (low model complexity). The following table contains these results.

Model #	Number of Parameters	PESQ	WER
25	3,218,501	2.470	29.001%
26	12,837,001	2.445	28.591%
27	1,009,501	2.390	26.402%
30	1,209,751	2.441	31.464%
31	4,819,501	2.490	31.737%
33	2,142,896	2.421	28.728%
38	5,867,728	2.452	29.001%
41	4,682,896	2.422	28.454%
53	2,266,736	2.458	25.718%
64	761,251	2.437	29.275%
69	761,501	2.443	27.633%
70	1,562,101	2.451	29.412%
71	841,251	2.480	27.223%

Table 3.3: Number of parameters, PESQ, and WER of top 13 FCN architectures in terms of validation loss. The specific details of each architecture can be found in Appendix A. For comparison, the noisy validation set at an SNR of 5 dB has a PESQ of 1.764 and WER of 50.479%.

Using the results of Table 3.3, Model #53 and Model #71 are chosen to be studied further. Both of these models have good PESQ, WER (Model #53 has the best WER), and model complexities that are not too high relatively. Model

#53 is over twice as complex as Model #71, but this complexity may be needed to achieve generalization across different SNRs. This leads to experimenting with both models on the same validation set but at other SNRs rather than only 5 dB (specifically 0 dB and -5 dB). First, each architecture will be trained on the same training set but at 0 dB and then PESQ and WER will be computed on the 0 dB validation set. This process will be repeated for -5 dB. The results across these different SNRs for both models are displayed in the following table.

Model #	SNR	PESQ	WER
53	0	2.056	60.411%
71	0	2.037	63.699%
53	-5	1.395	94.110%
71	-5	1.314	96.849%

Table 3.4: Results of Model #53 and Model #71 trained and validated at SNRs of 0 dB and -5 dB. For comparison, the noisy validation set at an SNR of 0 dB has a PESQ of 1.382 and WER of 88.493% and at an SNR of -5 dB it has a PESQ of 1.103 and WER of 98.493%

Model #53 clearly outperforms Model #71 at both 0 dB and -5 dB. Therefore, Model #53 is chosen to be the FCN architecture for the speech enhancement system.

Layer Type	Output Shape	Number of Parameters
1-D Convolution	(320, 12)	972
Batch Normalization	(320, 12)	48
PReLU Activation	(320, 12)	3,840
1-D Convolution	(320, 25)	24,025
Batch Normalization	(320, 25)	100
PReLU Activation	(320, 25)	8,000
1-D Convolution	(320, 50)	100,050
Batch Normalization	(320, 50)	200
PReLU Activation	(320, 50)	16,000
1-D Convolution	(320, 100)	400,100
Batch Normalization	(320, 100)	400
PReLU Activation	(320, 100)	32,000
1-D Convolution	(320, 200)	1,600,200
Batch Normalization	(320, 200)	800
PReLU Activation	(320, 200)	64,000
1-D Convolution	(320, 1)	16,001

Table 3.5: A layer-by-layer description of Model #53's FCN architecture. More details on this architecture can be found in Appendix A.

Now that the FCN architecture is fixed and the speech enhancement system is completely specified, the next section will deal with testing the system on new audio data in order to measure its ability to generalize on the same speaker.

3.3 Testing Generalization on the Same Speaker

The first part of testing involves training the speech enhancement system on a speaker and testing it on the same speaker, but the speech will have never been seen by the system nor the babble environment. First, the speech enhancement system trained and validated from the previous section, i.e. using Model #53 as the FCN architecture, is used for testing. Five minutes from Chapter 3 from [37] will be used as the test target speech and five minutes of a new babble environment [41] will be used for the test background noise. To make sure the testing procedure is clear, consider the following walkthrough of the process.

First, using the training setup of the previous section, the speech enhancement system is trained using Chapter 1 from [37] and bar babble noise from [39] at a specific SNR, say 5 dB. The training process employs early stopping that uses 5 minutes from Chapter 2 from [37] and 5 minutes of cafe babble noise from [40] at the same SNR. Next, the trained system is tested on 5 minutes from Chapter 3 in [37] and 5 minutes of coffee shop babble noise from [41] at the same SNR (i.e. 5 dB), but also at 0 dB and -5 dB to get a measure of the system's robustness across SNRs. This process is repeated for an SNR of 0 dB and an SNR of -5 dB. Tables 3.6 & 3.7 report the results of this process.

	Testing SNR		
	5 dB	0 dB	-5 dB
Training SNR			
5 dB	2.478	1.917	1.350
0 dB	2.530	2.100	1.461
-5 dB	2.379	2.060	1.482
	(Noisy: 1.782)	(Noisy: 1.444)	(Noisy: 1.321)

Table 3.6: This table presents PESQ test results across different SNRs for testing on the same speaker. Each row represents the SNR that the speech enhancement system was trained at. Each column represents the SNR of the test set. For reference, the PESQ of the noisy test set is included for each SNR.

	Testing SNR		
	5 dB	0 dB	-5 dB
Training SNR			
5 dB	25.243%	60.888%	94.868%
0 dB	31.900%	58.391%	91.817%
-5 dB	52.705%	77.531%	94.452%
	(Noisy: 43.689%)	(Noisy: 86.269%)	(Noisy: 97.503%)

Table 3.7: This table presents WER test results across different SNRs for testing on the same speaker. Each row represents the SNR the speech enhancement system was trained at. Each column represents the SNR of the test set. For reference, the WER of the noisy test set is included for each SNR.

Table 3.6 and Table 3.7 provide some interesting insights into the generalizability of the speech enhancement system. The first key insight is that both PESQ and WER on the test set do a good job tracking the results for PESQ and WER on the validation set from the previous section. The other key insight is that the speech enhancement system trained at 0 dB is quite robust across SNRs, sometimes doing better in scenarios one would not expect. With promising results from testing on the same speaker, the second part of testing will study generalizability to a new speaker.

3.4 Testing Generalization on a New Speaker

The second part of testing will employ the same methodology as the first part of testing but the target speech will come from a new speaker. Audio data is acquired for a new speaker (specifically another female speaker by the name of Tricia) via another audiobook [38]. First, performance will be measured when the speech enhancement system is trained only on the speaker Pamela (as has been done up to this point) and tested on the speaker Tricia. All trained models (i.e. at each specific SNR) from the first part of testing are used again in the second part of testing. These trained models filter 5 minutes of speech from Chapter 1 of [38] corrupted by the same coffee shop babble noise [41] at SNRs of -5, 0, and 5 dB. Tables 3.8 & 3.9 report the results of this process.

	Testing SNR		
	5 dB	0 dB	-5 dB
Training SNR			
5 dB	2.215	1.781	1.291
0 dB	2.171	1.839	1.382
-5 dB	2.229	1.876	1.437
	(Noisy: 1.874)	(Noisy: 1.471)	(Noisy: 1.182)

Table 3.8: This table presents PESQ test results across different SNRs for training on one speaker and testing on a new speaker. Each row represents the SNR that the speech enhancement system was trained at. Each column represents the SNR of the test set. For reference, the PESQ of the noisy test set is included for each SNR.

	Testing SNR		
	5 dB	0 dB	-5 dB
Training SNR			
5 dB	25.134%	54.545%	89.037%
0 dB	29.412%	50.401%	84.358%
-5 dB	55.615%	67.380%	89.305%
	(Noisy: 33.556%)	(Noisy: 72.326%)	(Noisy: 94.652%)

Table 3.9: This table presents WER test results across different SNRs for training on one speaker and testing on a new speaker. Each row represents the SNR that the speech enhancement system was trained at. Each column represents the SNR of the test set. For reference, the WER of the noisy test set is included for each SNR.

When comparing Tables 3.6 & 3.7 with Tables 3.8 & 3.9, it is noticed that performance on the new speaker is good but does not quite track the performance of a system trained on a speaker and then tested on that same speaker. It is hypothesized that fine-tuning the parameters of the trained network with a few minutes of data from the new speaker should improve performance and more closely track performance on the same speaker. Therefore, an additional, disjoint 5 minutes of speech from Chapter 1 of [38] corrupted by 5 minutes of cafe babble noise from [40] at a given SNR is used to fine-tune the already trained model (i.e. trained on the speaker Pamela) by doing 5 epochs of gradient descent. The resulting trained model is then used to again filter 5 minutes of speech from Chapter 1 of [38] corrupted by the same coffee shop babble noise [41] at SNRs of -5, 0, and 5 dB. Tables 3.10 & 3.11 report the results of this process.

	Testing SNR		
	5 dB	0 dB	-5 dB
Training SNR			
5 dB	2.417	1.953	1.442
0 dB	2.378	2.025	1.571
-5 dB	2.283	2.026	1.619
	(Noisy: 1.874)	(Noisy: 1.471)	(Noisy: 1.182)

Table 3.10: This table presents PESQ test results across different SNRs for training on one speaker, fine-tuning on a new speaker, and then testing on that new speaker. Each row represents the SNR that the speech enhancement system was trained at. Each column represents the SNR of the test set. For reference, the PESQ of the noisy test set is included for each SNR.

	Testing SNR		
	5 dB	0 dB	-5 dB
Training SNR			
5 dB	21.791%	45.588%	87.166%
0 dB	28.342%	40.909%	76.337%
-5 dB	58.690%	79.813%	81.684%
	(Noisy: 33.556%)	(Noisy: 72.326%)	(Noisy: 94.652%)

Table 3.11: This table presents WER test results across different SNRs for training on one speaker, fine-tuning on a new speaker, and then testing on that new speaker. Each row represents the SNR that the speech enhancement system was trained at. Each column represents the SNR of the test set. For reference, the WER of the noisy test set is included for each SNR.

When comparing Tables 3.6 & 3.7 with Tables 3.10 & 3.11, it is noticed that performance on the new speaker now does a good job tracking the performance of a system trained on a speaker then tested on that same speaker. This proves the initial hypothesis and it can be concluded that the speech enhancement system trained at 0 dB is able to generalize to new speakers (via fine-tuning) and is markedly robust across different SNRs. Figure 3.2 contains spectrograms illustrating the workings of the speech enhancement system.

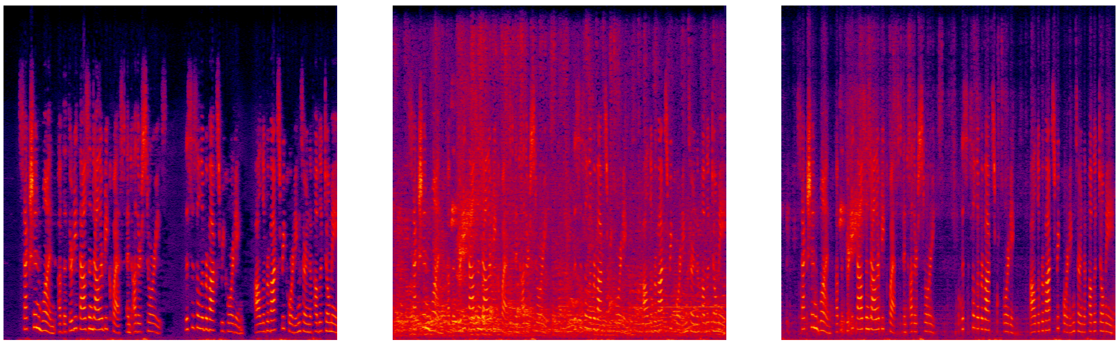

Figure 3.2: Clean (LEFT), noisy (CENTER), and filtered (RIGHT) spectrograms of 10 seconds of the new speaker's speech at 0 dB.

Chapter 4

Conclusions & Future Work

A fully convolutional neural network based end-to-end speech enhancement system that serves as a solution to the famous cocktail party problem has been presented. A strong ability to generalize to new speakers is presented by fine-tuning of the system with limited data. Test results show that the system is robust to different babble noise environments of varying SNRs. This speech enhancement system shows promising results objectively, using PESQ and WER measures, and subjectively by listening to the filtered audio. A few questions to consider for future research pertaining to this speech enhancement system are presented below:

1) What is the optimal model complexity for the task?

2) Does the system continue to generalize well to new environments and speakers?

3) What is the minimal computational/storage complexity needed to employ this system in real-time?

References

[1] E. Healy, et al., "An Algorithm to Improve Speech Recognition in Noise for Hearing-Impaired Listeners." *The Journal of the Acoustical Society of America*, vol. 134, no. 4, 2013, pp. 30293038., doi:10.1121/1.4820893.

[2] H. Dillon, *Hearing Aids.* Thieme, 2012.

[3] J. Mcdermott, "The Cocktail Party Problem." *Current Biology*, vol. 19, no. 22, 2009, doi:10.1016/j.cub.2009.09.005.

[4] C. Cherry, "Some Experiments on the Recognition of Speech, with One and with Two Ears" (PDF). *The Journal of the Acoustical Society of America.*, 1953, 25 (5): 97579. doi:10.1121/1.1907229.

[5] T. Rohdenburg, et al. "Robustness Analysis of Binaural Hearing Aid Beamformer Algorithms by Means of Objective Perceptual Quality Measures." 2007 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, 2007, doi:10.1109/aspaa.2007.4393016.

[6] Loizou, Philipos C., and Gibak Kim. "Reasons Why Current Speech-Enhancement Algorithms Do Not Improve Speech Intelligibility and Suggested Solutions." IEEE Transactions on Audio, Speech, and Language Processing, vol. 19, no. 1, 2011, pp. 4756., doi:10.1109/tasl.2010.2045180.

[7] C. Guo, et al. "A Speech Enhancement Algorithm Using Computational Auditory Scene Analysis with Spectral Subtraction." 2016 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), 2016, doi:10.1109/isspit.2016.7886000

[8] Wei, Cheng-Wen, et al. "Perceptual Multiband Spectral Subtraction for Noise Reduction in Hearing Aids." 2010 IEEE Asia Pacific Conference on Circuits and Systems, 2010, doi:10.1109/apccas.2010.5774973.

[9] Modhave, Nayan, et al. "Design of Multichannel Wiener Filter for Speech Enhancement in Hearing Aids and Noise Reduction Technique." 2016 Online

International Conference on Green Engineering and Technologies (IC-GET), 2016, doi:10.1109/get.2016.7916626.

[10] Simpson, and Andrew J. R., "Deep Transform: Cocktail Party Source Separation via Complex Convolution in a Deep Neural Network." 12 Apr. 2015, arxiv.org/abs/1504.02945

[11] K. Paliwal, et al., "The Importance of Phase in Speech Enhancement." *Speech Communication*, vol. 53, no. 4, 2011, pp. 465494., doi:10.1016/j.specom.2010.12.003.

[12] T. Sainath, R. Weiss, et al., "Learning the speech front-end with raw waveform cldnns." *In Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[13] G. Trigeorgis, F. Ringeval, et al., "Adieu features? End-to-end speech emotion recognition using a deep convolutional recurrent network." *In Acoustics, Speech and Signal Processing (ICASSP)*, 2016 IEEE International Conference, pp. 52005204. IEEE, 2016.

[14] J. Thickstun, Z. Harchaoui, and S. Kakade, "Learning features of music from scratch". arXiv preprint arXiv:1611.09827, 2016.

[15] S. Fu, et al, "Raw Waveform-Based Speech Enhancement by Fully Convolutional Networks." *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2017, doi:10.1109/apsipa.2017.8281993.

[16] Y. Gong and C. Poellabauer, "How do deep convolutional neural networks learn from raw audio waveforms?", 2018, Available: https://openreview.net/forum?id=S1Ow_eRb

[17] P. C. Loizou, *Speech Enhancement -Theory and Practice.* Taylor and Francis, 2nd Edition., 2013

[18] A. Oppenheim, et al. *Discrete-Time Signal Processing.* Dorling Kindersley / Pearson Education, 1999.

[19] Aquegg, "Spectrogram." *Wikipedia, Wikimedia Foundation*, 21 Dec. 2008, en.wikipedia.org/wiki/Spectrogram#/media/File:Spectrogram-19thC.png.

[20] *Butterworth filter design - MATLAB*, www.mathworks.com/help/signal/ref/butter.html

[21] A. Jacobsen, M. Kolbk, *Spectral Speech Enhancement using Deep Neural Networks*, 2015

[22] A. Oppenheim, G. Verghese, *Signals, Systems and Inference*, Pearson, 2017.

[23] D. Wang, and G. Brown, *Computational Auditory Scene Analysis: Principles, Algorithms, and Applications*. IEEE Press,2006.

[24] D. Wang. "Keynote Addresses: From Auditory Masking to Binary Classification: Machine Learning for Speech Separation." 2013 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, 2013, doi:10.1109/waspaa.2013.6701900

[25] Rix, A.W., et al., "Perceptual evaluation of speech quality (PESQ) - a new method for speech quality assessment of telephone networks and codecs." *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, doi:10.1109/icassp.2001.941023.

[26] C. Taal,, et al., "An Algorithm for Intelligibility Prediction of Time-Frequency Weighted Noisy Speech." *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 19, no. 7, 2011, pp. 2125-2136., doi:10.1109/tasl.2011.2114881.

[27] A. Morris, et al., "From WER and RIL to MER and WIL: improved evaluation measures for connected speech recognition." Institute of Phonetics at Saarland University, Germany.

[28] I. Goodfellow, et al., *Deep Learning*. MIT Press, 2016. Available online: http://www.deeplearningbook.org/

[29] Y. Abu-Mostafa, et al., *Learning From Data: A Short Course*. AMLbook.com, 2012.

[30] D. R. Wilson and T. R. Martinez, "The general inefficiency of batch training for gradient descent learning," Neural Netw., vol. 16, no. 10, pp. 1429-1451, Dec. 2003. [Online]. Available: http://dx.doi.org/10.1016/S0893-6080(03)00138-2

[31] 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

[32] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," CoRR, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[33] P.862 : Revised Annex A - Reference Implementations and Conformance Testing for ITU-T Recs P.862, P.862.1 and P.862.2. N.p., n.d. Web. 25 Mar. 2018. Available: http://www.itu.int/rec/T-REC-P.862-200511-I!Amd2/en

[34] “Belambert/asr-evaluation.” *GitHub*. N.p., n.d. Web. 25 Mar. 2018. Available: https://github.com/belambert/asr-evaluation

[35] X. Lu, et al., “Speech Enhancement Based on Deep Denoising Autoencoder”, 13 Aug. 2013. Interspeech 2013.

[36] *Speech to Text Demo*. IBM Watson, n.d. Web. 25 Mar. 2018. Available: https://speech-to-text-demo.ng.bluemix.net/

[37] J. McCharthy, *A History of the Four Georges.*, Vol. 1, Audiobook Available: librivox.org/a-history-of-the-four-georges-volume-1-by-justin-mccarthy/. Narrator: Pamela Nagami

[38] M. Twain, *The $30,000 Bequest and Other Stories.*, Audiobook Available: librivox.org/30000-bequest-and-other-stories-by-mark-twain/, Narrator: Tricia G.

[39] “BAR CROWD for 3 Hours Sound Effect.” *YouTube*, 10 Oct. 2013, www.youtube.com/watch?v=bCnJoaXYFGg.

[40] “The Noise Cafe 2 hours.” *YouTube*, 28 Apr. 2012, www.youtube.com/watch?v=KZV9FmHOsRg.

[41] “Coffee Shop Sounds Background Noise For Work.” *YouTube*, 23 Mar. 2016, www.youtube.com/watch?v=jBNoFk03vWk.

Appendix A

System Design: Top 13 FCN Architectures

FCN Name: Model #25
Number of Filters Per Hidden Layer: 100, 100, 100, 100, 100
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 29
Activation: ReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0149619835036
Validation Loss: 0.0280033512336
Number of Parameters: 3,218,501
Validation PESQ: 2.470
Validation WER: 29.001%

FCN Name: Model #26
Number of Filters Per Hidden Layer: 200, 200, 200, 200, 200
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples

SNR: 5 dB
Best Epoch: 35
Activation: ReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0146398135166
Validation Loss: 0.0281914634397
Number of Parameters: 12,837,001
Validation PESQ: 2.445
Validation WER: 28.591%

FCN Name: Model #27
Number of Filters Per Hidden Layer: 50, 50, 50, 50, 50, 50
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 35
Activation: ReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.016485615045
Validation Loss: 0.028403555044
Number of Parameters: 1,009,501
Validation PESQ: 2.390
Validation WER: 26.402%

FCN Name: Model #30
Number of Filters Per Hidden Layer: 50, 50, 50, 50, 50, 50, 50
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 20
Activation: ReLU

Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0179147224677
Validation Loss: 0.0284523007359
Number of Parameters: 1,209,751
Validation PESQ: 2.441
Validation WER: 31.464%

FCN Name: Model #31
Number of Filters Per Hidden Layer: 100, 100, 100, 100, 100, 100, 100
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 29
Activation: ReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0141895399333
Validation Loss: 0.0266052821079
Number of Parameters: 4,819,501
Validation PESQ: 2.490
Validation WER: 31.737%

FCN Name: Model #33
Number of Filters Per Hidden Layer: 12, 25, 50, 100, 200
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 69
Activation: ReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0150186144539

Validation Loss: 0.0284648104539
Number of Parameters: 2,142,896
Validation PESQ: 2.421
Validation WER: 28.728%

FCN Name: Model #38
Number of Filters Per Hidden Layer: 12, 25, 50, 100, 200
Filter Size Per Hidden Layer: 1.0, 2.0, 4.0, 8.0, 16.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 46
Activation: ReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0144523108945
Validation Loss: 0.027507254274
Number of Parameters: 5,867,728
Validation PESQ: 2.452
Validation WER: 29.001%

FCN Name: Model #41
Number of Filters Per Hidden Layer: 12, 25, 50, 100, 200
Filter Size Per Hidden Layer: 5.0, 5.0, 8.0, 8.0, 12.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 72
Activation: ReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0127350104231
Validation Loss: 0.0283544569146
Number of Parameters: 4,682,896
Validation PESQ: 2.422

Validation WER: 28.454%

FCN Name: Model #53
Number of Filters Per Hidden Layer: 12, 25, 50, 100, 200
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 125
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 29
Activation: PReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Number of Parameters: 2,266,736
Training Loss: 0.0153066175394
Validation Loss: 0.0276513302435
Validation PESQ: 2.458
Validation WER: 25.718%
Training Loss (0 dB): 0.0329275182994
Validation Loss (0 dB): 0.0830621913256
Validation PESQ (0 dB): 2.056
Validation WER (0 dB): 60.411%
Training Loss (-5 dB): 0.109046343979
Validation Loss (-5 dB): 0.237691486046
Validation PESQ (-5 dB): 1.395
Validation WER (-5 dB): 94.110%

FCN Name: Model #64
Number of Filters Per Hidden Layer: 25, 25, 50, 50, 100
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 150
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 35
Activation: ReLU
Batch Normalization: YES, between convolution & activation

Regularization: NONE
Training Loss: 0.0171705612745
Validation Loss: 0.0283289428108
Number of Parameters: 761,251
Validation PESQ: 2.437
Validation WER: 29.275%

FCN Name: Model #69
Number of Filters Per Hidden Layer: 25, 25, 50, 50, 100
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 300
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 36
Activation: PReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0167125569588
Validation Loss: 0.0286627422307
Number of Parameters: 761,501
Validation PESQ: 2.443
Validation WER: 27.633%

FCN Name: Model #70
Number of Filters Per Hidden Layer: 25, 25, 50, 50, 100, 100
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 300
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 24
Activation: PReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0159617189242
Validation Loss: 0.028751412553

Number of Parameters: 1,562,101
Validation PESQ: 2.451
Validation WER: 29.412%

FCN Name: Model #71
Number of Filters Per Hidden Layer: 25, 25, 50, 50, 100
Filter Size Per Hidden Layer: 5.0, 5.0, 5.0, 5.0, 5.0 ms
Filter Size for Output Layer: 5.0 ms
Input Frame Time: 20.0 ms
Total Epochs: 300
Batch Size: 100 examples
SNR: 5 dB
Best Epoch: 36
Activation: ReLU
Batch Normalization: YES, between convolution & activation
Regularization: NONE
Training Loss: 0.0157915527726
Validation Loss: 0.0285157191015
Number of Parameters: 841,251
Validation PESQ: 2.480
Validation WER: 27.223%
Training Loss (0 dB): 0.0352021876569
Validation Loss (0 dB): 0.0823882113324
Validation PESQ (0 dB): 2.037
Validation WER (0 dB): 63.699%
Training Loss (-5 dB): 0.119671967316
Validation Loss (-5 dB): 0.252999086193
Validation PESQ (-5 dB): 1.314
Validation WER (-5 dB): 96.849%

Appendix B

Python Code

B.1 audio_preprocessing.py

```
from os.path import join
import librosa
import librosa.display
import numpy as np
import scipy.signal
import scipy.io.wavfile
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
import math


def round_up_to_even(x):
    return int(math.ceil(x / 2.) * 2)

def next_pow2(x):
    return 2**(x-1).bit_length()

def apply_window(frame):

 M = frame.size
 window = scipy.signal.hann(M)

 return(frame*window)

def overlapp_add_reconstruction( frame1_windowed, frame2_windowed ):

 M = frame2_windowed.size
 R = M/2
 output = np.zeros(frame1_windowed.size + R)
 output[:frame1_windowed.size] = frame1_windowed
 output[(frame1_windowed.size-R):] = output[(frame1_windowed.size-R):] +
     frame2_windowed

 return(output)

def load_audio_files( audio_folder_path, chapter_names, noise_names ):

 chapters = {}
 for chapter_name in chapter_names:
  file_path = join(audio_folder_path, chapter_name + ".wav")
  fs, audio_time_series = scipy.io.wavfile.read(file_path)
  chapters[ chapter_name ] = (audio_time_series, fs)

 noise = {}
 for noise_name in noise_names:
  file_path = join(audio_folder_path, noise_name + ".wav")
  fs, audio_time_series = scipy.io.wavfile.read(file_path)
  noise[ noise_name ] = (audio_time_series, fs)

 return chapters, noise

def load_audio( audio_folder_path, audio_filename ):
 file_path = audio_folder_path + audio_filename
 fs, audio_time_series = scipy.io.wavfile.read(file_path)
 return audio_time_series, fs

def concatenate_audio( names, dict ):

 arrays_to_concatenate = []
 fs = dict[names[0]][1]
 for name in names:
  arrays_to_concatenate.append(dict[name][0])

 return(np.concatenate(arrays_to_concatenate) , fs)

def combine_clean_and_noise(audio_time_series_train, audio_time_series_noise, snr_db):
 if( audio_time_series_train.size <= audio_time_series_noise.size ):
  audio_time_series_noise = audio_time_series_noise[0:audio_time_series_train.size]
 else:
  audio_time_series_train = audio_time_series_train[0:audio_time_series_noise.size]

 audio_time_series_train = audio_time_series_train.astype('float')
 audio_time_series_noise = audio_time_series_noise.astype('float')

 A_train_2 = np.mean(np.power(np.absolute(audio_time_series_train),2))
 A_noise_2 = np.mean(np.power(np.absolute(audio_time_series_noise),2))
 A_noise_targ_2 = A_train_2 / (10**(snr_db/10.))

 scaling_coeff = np.sqrt(A_noise_targ_2) / np.sqrt(A_noise_2)

 combined_result = audio_time_series_train + (scaling_coeff * audio_time_series_noise)

 return(combined_result)

def downsample( audio, orig_sr, targ_sr):
 audio = audio.astype('float')
 audio_downsampled = librosa.resample(audio, orig_sr, targ_sr)
 audio_downsampled = audio_downsampled.astype('int16')
 return audio_downsampled, targ_sr

def generate_frames( audio_time_series_train, fs, frame_time, lag = 0.5 ):
 frame_length = round_up_to_even( frame_time * fs )
 total_time_steps = int((audio_time_series_train.size / (frame_length * lag)) - 1)

 x_train = np.zeros( shape = (frame_length, total_time_steps) )

 for i in range(0, (total_time_steps - 1)):
  x_train[:, i] =  apply_window( audio_time_series_train[ i*(frame_length / 2) : (
        i*(frame_length / 2) + frame_length) ] )

 return(x_train)

def generate_train_features( x_train ):
 n = next_pow2(x_train.shape[0])
 x_train_features = np.zeros( shape = (n,  x_train.shape[1]))
 for i in range(0, x_train.shape[1]):
  x_train_features[:, i] = np.abs(np.fft.fft( a = x_train[:, i], n = n ))
 return(x_train_features)

def scale_features( x_train_features, mu, std ):

 x_train_features = (x_train_features - mu) / float(std)

 return(x_train_features.T)

def rebuild_audio_from_indices( predicted_time_series_indices, x_train ):
 output = x_train[:, predicted_time_series_indices[0]]

 for i in predicted_time_series_indices[1:]:
  output = overlapp_add_reconstruction(output, x_train[:, i])

 return(output.astype('int16'))

def rebuild_audio( x_test ):
 output = x_test[0, :]
 for i in xrange(1, x_test.shape[0]-1):
  output = overlapp_add_reconstruction(output, x_test[i, :])

 return(output.astype('int16'))

def sdr_computation( target_speech, distorted_speech ):
 if( target_speech.size <= distorted_speech.size ):
  distorted_speech = distorted_speech[0:target_speech.size]
 else:
  target_speech = target_speech[0:distorted_speech.size]

 target_speech = target_speech.astype('float')
 distorted_speech = distorted_speech.astype('float')

 A_target_2 = np.mean(np.power(np.absolute(target_speech),2))
 A_noisedistortion_2 = np.mean(np.power(np.absolute(distorted_speech),2))

 sdr = 10*np.log10(A_target_2 / A_noisedistortion_2)

 return(sdr)

def generate_input( audio_time_series, fs, frame_time, train_mu, train_std ):

 x_frames = generate_frames( audio_time_series, fs, frame_time,  )
 x_frames_scaled = scale_features( x_frames, train_mu, train_std )
 x_frames_scaled_input = np.reshape(x_frames_scaled, (x_frames_scaled.shape[0],
     x_frames_scaled.shape[1], 1))

 return(x_frames_scaled_input)
```

B.2 cnn_model.py

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv1D, MaxPooling1D, UpSampling1D, Activation, LeakyReLU
    , PReLU, Dropout
from keras.layers.normalization import BatchNormalization
from keras import regularizers
from keras.models import load_model
from keras import backend as K
import numpy as np
from keras.callbacks import ModelCheckpoint, History, EarlyStopping

def create_model(input_shape, num_filters_per_hidden_layer,
    filter_size_per_hidden_layer, filter_size_output_layer ):

 K.clear_session()

 model = Sequential()

 model.add(Conv1D( filters = num_filters_per_hidden_layer[0], kernel_size =
     filter_size_per_hidden_layer[0], padding='same', input_shape = input_shape))
 model.add(BatchNormalization())
 model.add(PReLU())

 for num_filters, filter_size in zip(num_filters_per_hidden_layer[1:],
     filter_size_per_hidden_layer[1:]):
  model.add(Conv1D( filters = num_filters, kernel_size = filter_size, padding='same
      '))
  model.add(BatchNormalization())
  model.add(PReLU())


 model.add(Conv1D(1, kernel_size = filter_size_output_layer, padding='same'))

 return(model)

def train_model( model, train_inputs, train_labels, epochs, batch_size,
    validation_inputs, validation_labels, filepath, patience ):

 model.compile(optimizer = 'adam', loss='mean_squared_error')

 checkpointer = ModelCheckpoint(filepath = filepath, monitor = "val_loss", verbose
      = 1, mode = 'min', save_best_only = True)
 early_stopping = EarlyStopping(monitor = 'val_loss', min_delta = 0, patience =
     patience, verbose = 1, mode='auto')

 history = model.fit( train_inputs, train_labels,
                 epochs = epochs,
                    batch_size = batch_size,
                    shuffle = True,
                    validation_data = (validation_inputs, validation_labels),
                    callbacks = [checkpointer, early_stopping])

 model = load_model(filepath)

 return(model, history)

def train_model_finetune( model, train_inputs, train_labels, epochs, batch_size ):

 model.compile(optimizer = 'adam', loss='mean_squared_error')

 history = model.fit( train_inputs, train_labels,
                 epochs = epochs,
                    batch_size = batch_size,
                    shuffle = True)

 return(model, history)

def save_model( model, save_path ):
 model.save(save_path)

def load_model_( load_path ):
 model = load_model( load_path )
 return(model)

def predict_model( model, inputs ):
 predictions = model.predict(inputs, batch_size = None, verbose=0, steps=None)
 return(predictions)

def get_output( model, new_input ):
 get_output = K.function([model.layers[0].input, K.learning_phase()],
                                    [model.layers[ len(model.layers) - 1 ].output])
 layer_output = get_output([new_input, 0])[0]

 return(layer_output)

def get_output_multiple_batches(model, input_frames, batch_size = 100):

 batches_output_frames_holder = []
 for i in xrange(0, input_frames.shape[0], batch_size):
  batch_input_frames = input_frames[ i:i+batch_size , :, : ]
  batch_output_frames = get_output( model, batch_input_frames )
  batch_output_frames = np.reshape(batch_output_frames, (batch_output_frames.shape[0], -1))
  batches_output_frames_holder.append(batch_output_frames)

 output_frames_concatenated = np.concatenate( batches_output_frames_holder, axis = 0 )

 return(output_frames_concatenated)

def summary_statistics( filename, model_name, history, frame_time, snr_db,
      num_filters_per_hidden_layer, filter_size_per_hidden_layer, filter_size_output_layer,
      epochs, batch_size):



 best_val_loss = min( history.history["val_loss"] )
 best_epoch_index = history.history["val_loss"].index( best_val_loss )
 best_train_loss = history.history["loss"][ best_epoch_index ]

 print( "\tFCNN Name: " + model_name )
 print( "\tNumber of Filters Per Hidden Layer: " + ', '.join(map(str, num_filters_per_hidden_layer) ))
 print( "\tFilter Size Per Hidden Layer: " + ', '.join(map(str, list(np.array(filter_size_per_hidden_layer)*1000)))  + " ms" )
 print( "\tFilter Size for Output Layer: " + str( filter_size_output_layer*1000 ) + " ms")
 print( "\tFrame Time: " + str( frame_time*1000 ) + " ms")
 print( "\tTotal Epochs: " + str(epochs) )
 print( "\tBatch Size: " + str(batch_size) + " examples" )
 print( "\tSNR: " + str( snr_db ) + " dB")
 print( "\tBest Epoch: " + str(  best_epoch_index + 1 ) )
 print( "\tTraining Loss: " + str( best_train_loss ) )
 print( "\tValidation Loss: " + str( best_val_loss ) )
 print("\n")
```

```
    with open(filename, "a") as text_file:
        text_file.write( "FCNN Name: " + model_name )
        text_file.write( "\n" )
        text_file.write( "Number of Filters Per Hidden Layer: " + ', '.join(map(str, num_filters_per_hidden_layer) ))
        text_file.write( "\n" )
        text_file.write( "Filter Size Per Hidden Layer: " + ', '.join(map(str, list(np.array(filter_size_per_hidden_layer)*1000))) + " ms" )
        text_file.write( "\n" )
        text_file.write( "Filter Size for Output Layer: " + str( filter_size_output_layer*1000 ) + " ms")
        text_file.write( "\n" )
        text_file.write( "Frame Time: " + str( frame_time*1000 ) + " ms")
        text_file.write( "\n" )
        text_file.write( "Total Epochs: " + str(epochs) )
        text_file.write( "\n" )
        text_file.write( "Batch Size: " + str(batch_size) + " examples" )
        text_file.write( "\n" )
        text_file.write( "SNR: " + str( snr_db ) + " dB")
        text_file.write( "\n" )
        text_file.write( "Best Epoch: " + str(  best_epoch_index + 1 ) )
        text_file.write( "\n" )
        text_file.write( "Training Loss: " + str( best_train_loss ) )
        text_file.write( "\n" )
        text_file.write( "Validation Loss: " + str( best_val_loss ) )
        text_file.write( "\n\n" )
```

B.3 main.py

```
import audio_preprocessing as ap
import cnn_model as cnn
import numpy as np
import scipy.io.wavfile
import os
from subprocess import call

print("Getting paths to audio files...")
cwd = os.getcwd()
parent_cwd = os.path.abspath(os.path.join(cwd, os.pardir))
audio_folder_path = parent_cwd + "/Audio_Files/"

print("Creating training, validation, and test sets...")
snr_db = 5
frame_time = 0.020

# Generate training set
audio_time_series_train, fs = ap.load_audio( audio_folder_path, audio_filename = "Chapter1.wav")
audio_time_series_train_noise, fs = ap.load_audio( audio_folder_path, audio_filename = "Chapter1_Babble.wav")
audio_time_series_train_noisy = ap.combine_clean_and_noise(audio_time_series_train, audio_time_series_train_noise, snr_db)
train_mu = np.mean( audio_time_series_train )
train_std = np.std( audio_time_series_train )

train_clean = ap.generate_input(  audio_time_series_train, fs, frame_time, train_mu, train_std )
train_noisy = ap.generate_input(  audio_time_series_train_noisy, fs, frame_time, train_mu, train_std )

# Generate validation set
```

```
audio_time_series_validation , fs = ap.load_audio( audio_folder_path ,
    audio_filename = "Chapter2_5_Min.wav")
audio_time_series_validation_noise , fs = ap.load_audio( audio_folder_path ,
    audio_filename = "Chapter2_5_Min_Babble.wav")
audio_time_series_validation_noisy = ap.combine_clean_and_noise(
    audio_time_series_validation , audio_time_series_validation_noise , snr_db)

validation_clean = ap.generate_input(  audio_time_series_validation , fs ,
    frame_time , train_mu , train_std )
validation_noisy = ap.generate_input(  audio_time_series_validation_noisy , fs ,
    frame_time , train_mu , train_std )

# Generate test set
audio_time_series_test , fs = ap.load_audio( audio_folder_path , audio_filename = "
    Chapter3_5_Min.wav")
audio_time_series_test_noise , fs = ap.load_audio( audio_folder_path ,
    audio_filename = "Chapter3_5_Min_Babble.wav")
audio_time_series_test_noisy = ap.combine_clean_and_noise(audio_time_series_test ,
    audio_time_series_test_noise , snr_db)
scipy.io.wavfile.write( filename = parent_cwd + "/Audio_Files/Test_Files/" + "
    CleanTest_5min_" + str(snr_db) + "dB.wav", rate = fs , data =
    audio_time_series_test.astype('int16'))
scipy.io.wavfile.write( filename = parent_cwd + "/Audio_Files/Test_Files/" + "
    CleanTest_1min_" + str(snr_db) + "dB.wav", rate = fs , data =
    audio_time_series_test.astype('int16')[0:(60*fs)])
scipy.io.wavfile.write( filename = parent_cwd + "/Audio_Files/Test_Files/" + "
    NoisyTest_5min_" + str(snr_db) + "dB.wav", rate = fs , data =
    audio_time_series_test_noisy.astype('int16'))
scipy.io.wavfile.write( filename = parent_cwd + "/Audio_Files/Test_Files/" + "
    NoisyTest_1min_" + str(snr_db) + "dB.wav", rate = fs , data =
    audio_time_series_test_noisy.astype('int16')[0:(60*fs)])

test_clean = ap.generate_input(  audio_time_series_test , fs , frame_time , train_mu ,
     train_std )
test_noisy = ap.generate_input(  audio_time_series_test_noisy , fs , frame_time ,
    train_mu , train_std )


print("Preparing neural network for training...")
input_shape = (train_noisy.shape[1], 1)
epochs = 1
batch_size = 100
filter_size_per_hidden_layer = [0.005, 0.005, 0.005, 0.005, 0.005]
filter_size_output_layer = 0.005
num_filters_per_hidden_layer = [12, 25, 50, 100, 200]
patience = 20

model_name = "MODEL_NAME_HERE"
model_save_path = parent_cwd + "/Saved_Models/" + model_name
#model = cnn.load_model_(model_save_path)

model = cnn.create_model( input_shape , num_filters_per_hidden_layer , map(int , list
    (np.array(filter_size_per_hidden_layer)*fs)), int(filter_size_output_layer*fs)
     )
model, history = cnn.train_model(  model = model,
          train_inputs = train_noisy ,
          train_labels = train_clean ,
          epochs = epochs ,
          batch_size = batch_size ,
          validation_inputs = validation_noisy ,
          validation_labels = validation_clean ,
          filepath = model_save_path ,
```

```python
          patience = patience)

print("Getting CNN output for noisy test set input...")
test_filtered_frames = (train_std * cnn.get_output_multiple_batches(model, test_noisy)) + train_mu

print("Perfectly reconstructing filtered test set audio & saving to memory...")
test_filtered = ap.rebuild_audio( test_filtered_frames )
scipy.io.wavfile.write( filename = parent_cwd + "/Audio_Files/Test_Files/" + model_name + "_FilteredTest_" + str(snr_db) + "dB_5min.wav", rate = fs, data = test_filtered)
scipy.io.wavfile.write( filename = parent_cwd + "/Audio_Files/Test_Files/" + model_name + "_FilteredTest_" + str(snr_db) + "dB_1min.wav", rate = fs, data = test_filtered[0:(60*fs)])

os.chdir(parent_cwd + "/Audio_Files/Test_Files")
call( ["./PESQ", "+16000", "CleanTest_1min_" + str(snr_db) + "dB.wav", "NoisyTest_1min_" + str(snr_db) + "dB.wav"] )
call( ["./PESQ", "+16000", "CleanTest_1min_" + str(snr_db) + "dB.wav", model_name + "_FilteredTest_" + str(snr_db) + "dB_1min.wav"] )

summary_stats_filename = parent_cwd + "/Saved_Models/Model_Descriptions.txt"
cnn.summary_statistics( summary_stats_filename, model_name, history, frame_time, snr_db,
        num_filters_per_hidden_layer, filter_size_per_hidden_layer, filter_size_output_layer,
        epochs, batch_size)
```