香 港 大 學

THE UNIVERSITY OF HONG KONG

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

SENIOR DESIGN PROJECT

Speech Enhancement based on Stacked Denoising Autoencoder with Wiener Filter

Name: Chu Man Chung
UID: 3035101205
Submission Date: 10th April, 2017

Abstract

For the mobile communication application and speech recognition system which operate in relatively complex environments, background noise is severe and hinder the speech recognition systems from recognizing the speech correctly. As a result, it is necessary to apply noise reduction technic to remove the background noise and enhance the quality of speech.

In this project, a speech enhancement and noise removal model based on Stacked Denoising Autoencoder(SDAE), Signal to Noise Ratio Estimation(SNR) and Frequency Domain Wiener Filter is proposed. Leaky ReLU is used as activation in the encoding stage of autoencoder and the relationship between the power spectrum of clean speech, noisy speech and background noise is modelled by SDA. First, the sub-band power spectrum of background noise and clean speech are estimated by two different SDA model with the same noisy speech as input. Second, a priori SNR is obtained from the estimated spectrum of clean speech and background noise from SDAE with the *a Posteriori SNR Controlled Recursive Averaging* (PCRA) approach. Finally, the clean speech is obtained by Wiener filter in frequency domain with a priori SNR.

The performance evaluation shows that the model improves the speech quality by 0.7 PESQ scores.

In conclusions, the neural network based method can provide a satisfactory noise removal and enhance the speech for speech recognition system.

Acknowledgement

I want to acknowledge with my appreciation to all those who help me to complete the senior design project. A special gratitude I want to give to my supervisor of the project, Dr. Y. C. Wu, whose contribution in giving suggestions on the proposed model and encouragement, helped me to have a deeper understanding of my project.

Furthermore, I also want to express my appreciation to the pivotal role of Mr. Ray Z.X LI, who help me to access to the computer assigned for this project and how to run the program on it.

Table of Contents

List of Abbreviations

ATH	"Absolute Threshold for Hearing"
DAE	"Denoising Autoencoder"
DCT	"Discrete Cosine Transform"
DNN	"Deep Neural Network"
FFT	"Fast Fourier Transform"
LSD	"Log-Spectral Distortion"
IFFT	"Inverse Fast Fourier Transform"
MCRA	"Minima-Controlled Recursive Averaging"
MFCC	"Mel Frequency Cepstral Coefficients"
MMSE	"Minimum Mean Square Error"
MAE	"Mean Absolute Error"
MSE	"Mean Square Error"
OLA	"Overlap-Add method"
WSE	"Weight Square Error"
PCRA	"*a Posteriori* SNR Controlled Recursive Averaging"
PESQ	"Perceptual Evaluation of Speech Quality"
PESQI	"Perceptual Evaluation of Speech Quality Improvement"
SDAE	"Stacked Denoising Autoencoder"
SGD	"Stochastic Gradient Descent"
SNR	"Signal to Noise Ratio"
STFT	"Short Time Fourier Transform"

List of Figure

1. Introduction

1.1 Background of Speech Enhancement

For the mobile communication application and speech recognition system which operate in relatively complex environments, background noise is severe and hinder the speech recognition systems from recognizing the speech correctly. As a result, it is necessary to apply noise reduction technic to remove the background noise and improve the speech quality.

Model background noise reduction technic can back to 1970s when several powerful algorithms have been developed to remove noise, such as frequency domain Wiener Filter proposed by J. Lim and A. V. Oppenheim, MMSE Short Time Spectral Amplitude estimator proposed by Y. Ephraim and wavelet thresholding method developed by D. L. Donoho. These noise removal algorithms assume the additive nature and simple statistical difference of noise and speech. However, the relation between noise and speech is complex which make applying an adoptive and non-linear model for finding the relationship between speech and noise in time and frequency domain more reasonable. Neural network has these properties and as a result it has long been thought as better algorithms to enhance the speech quality.

Neural network has many different types of model which are suitable for different tasks and the one for noise reduction is called Denoising Autoencoder (DAE) which is a special version of Autoencoder and first introduced in 2008. Since adopting neural network requires huge computational power which could not be achieved in the past, DAE has showed its robustness in removing noise and enhancing speech quality in recent year with the advanced technology in graphic processing unit.

1.2 Project Goal and Scope

Goal of the project are:

- To design a model based on Stacked Denoising Autoencoder
- To maximize the background noise reduction
- To minimize the distortion between restored speech and clean speech
- To maximizer perceptual quality of estimated clean speech

Since most automatic speech recognition systems available online have some kind of noise reduction technics, it may be inaccurate to conduct experiment and get the Word Error Rate (WER) with the speech recognition system and the noise reduction model proposed in this project. Therefore, this project will focus more on speech enhancement part and Perceptual Evaluation of Speech Quality Improvement(PESQI) based on ITU-T Recommendation P.862 Standard, Log-Spectral Distortion(LSD) and Noise Reduction are proposed to access the performance of model.

Since this project focus on enhancement of speech and ability of noise removal, the model's ability on improving the speech recognition system's performance is undetermined.

1.3 Project Organization

The paper is organized as follow. First, the theory of methods, such as STFT, MFCC and SDAE, are introduced in part two. Second, two unsuccessful models proposed for the project are briefly introduced, evaluated and discussed in part three. Since two of them give relatively bad result, only the final model has detail comparison with different parameters. Third, the performance of the final models is evaluated with three evaluation criteria, PESQI, LSD and Noise Reduction. Then, the performance is deeply discussed and accessed. Finally, a conclusion of the project is drawn.

2. Background Theory

In this part, the background theory of each method ever used in the project will be introduced in order to provide solid ground to the proposed models.

2.1 Short-Time Fourier Transform

STFT which belongs to Fourier transform is common technic in speech enhancement task.

2.1.1 Fourier Transform

The Fourier Transform base on Fourier series which explains that a complicated and periodic functions can be considered as a combination of sines and cosines function with different frequency. Due to this property, the Fourier Transform first lengthened the non-periodic function and allow it to approach infinity. Then, the periodic function is decomposed into a number of sines and cosines representation which has different frequency. Usually, the combination of sines and cosine with the same frequency is represented with complex components by Euler's formula.

$$e^{2\pi i\theta} = \cos 2\pi\theta + i\sin 2\pi\theta$$

Thus, the resulting Fourier coefficient will be a set of complex number with different frequency.

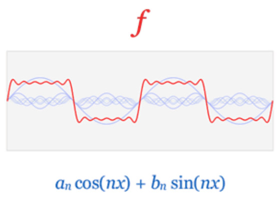


Figure 1: Signal in Time Domain

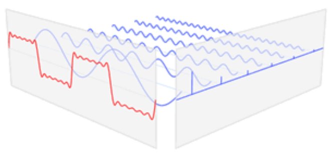


Figure 2: Decomposed Signal

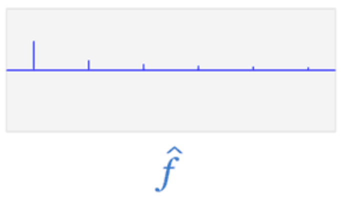


Figure 3: Signal in Frequency Domain

Reference to Wiki-Fourier Transform: https://en.wikipedia.org/wiki/Fourier_transform

The Fourier Transform decomposes a function of time, such as the speech signal in this project, into a function of frequency and this operation can be thought of mapping the function from time domain into frequency domain. Linear operations performed in one domain (time or frequency) have corresponding operations in the other domain, which are sometimes easier to perform. For example, the operation of differentiation in the time domain correspond to multiplication in frequency domain.

Here is the mathematics definition of Fourier Transform.

For continuous signal $x(t)$

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt$$

where ω is the frequency.

For discrete signal x[n]

$$X_{2\pi}(\omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}$$

2.1.2 STFT

STFT is one type of Fourier Transform and the variation of STFT is that STFT consider the non-stationary property of audio signal whose spectral content, or frequency content, changes over time. Since the Fourier coefficients are time-invariant function, applying original Fourier transform does not reveal the transitions in spectral content. To avoid this issue, the Fourier transform should be applied over a short time period that the audio speech is assumed stationary.

Usually, the audio signal is broken into 20ms-40ms frames with 50% or 75% overlap and the overlap is used to reduce artifacts at the boundary. Then, a window function, usually Hann and Hamming windows, is applied to each frames in order to provide a better frequency response since each frame is smoother and has less ripples. Then, Fourier transform is applied to each frame.

$$Hann\ Function: w[\lambda] = 0.5 - 0.5\cos\left(\frac{2\pi\lambda}{N-1}\right)$$

$$Hamming\ Function: w[\lambda] = 0.54 - 0.46\cos\left(\frac{2\pi\lambda}{N-1}\right)$$

where $1 \leq \lambda \leq N$ and N is the size of frames.

Here is the mathematics definition of discrete STFT.

$$X[\lambda,\delta] = \sum_{m=-\infty}^{\infty} x[m]w[\lambda - m]e^{-j\delta\lambda}$$

2.1.3 Power Spectrum

Power spectrum, or spectrogram, is defined as the power of STFT.

$$spectrogram = |X[\lambda,\delta]|^2 = X^2(\lambda,\delta)$$

where λ is frames index and δ is frequency index.

From the formula above, the spectrogram become real number but the phase information is not totally lost. Some phase information is contained in the spectrogram in another form, as time delay. Thus, it is possible to make a good estimation of the phase information of spectrogram.

Research [17] has suggested that the human ear extract most perceptual information from the power spectrum of speech signal. Thus, spectrogram is extensively used in speech enhancement tasks and signal processing while the phase information is much less important in speech enhancement. In this project, the phase information will not be modified in order to reduce the complexity of the models.

2.1.4 Inverse STFT

In this project, two inverse STFT methods are used and they are Overlap-Add method and Real Time Signal Estimation from Modified STFT Magnitude Spectra [19].

2.1.4.1 Overlap-Add Method

This method requires the modified STFT coefficients which have phase information. However, only the power spectrum of STFT coefficients will be used in the three models proposed in this project. Thus, the phase information used for overlap-add method is unmodified. In other word, it is noisy phase.

Overlap-Add method is an efficient way to recover the STFT coefficients back to real-time signal. Here is the equation.

First, inverse Fourier transform for each frame of STFT coefficients and it is multiplied by the window function.

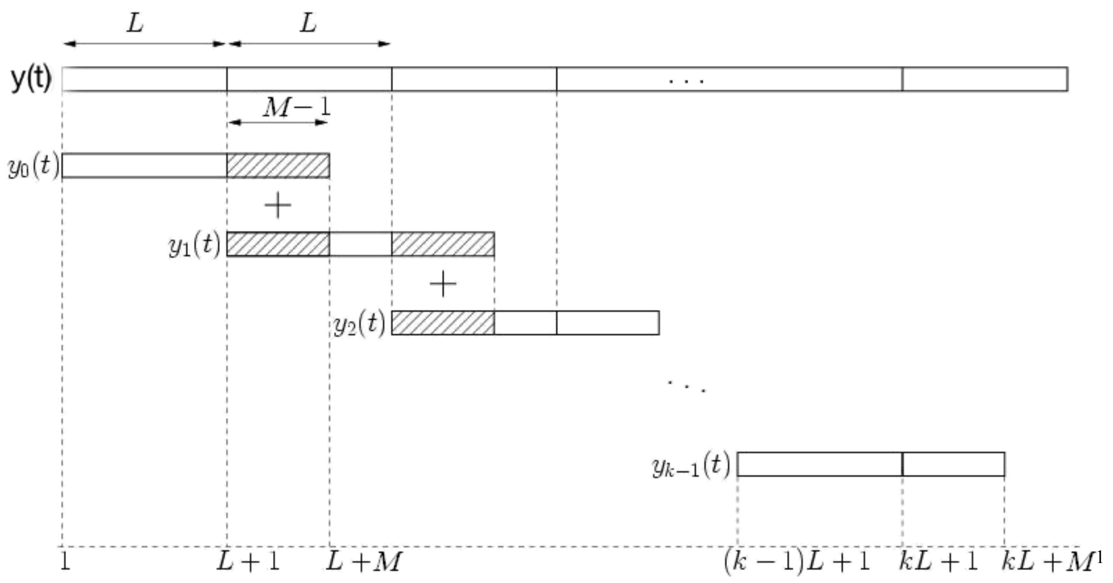
$$y_k[n] = IFFT(X_k[n]) * w_k[n]$$

where k is the frames index, $X_k[n]$ is the nth frames of STFT coefficients.

Since the STFT coefficients is usually overlapped, the overlap part is added together and the result is y[n]. The figure 4 can visually explain it.

For the window, a similar operation is needed and w[n] can be gotten.

$$x[n] = y[n]/w[n]$$



[1]

Figure 4: Overlap-Add method from Wiki-Overlap-Add: https://en.wikipedia.org/wiki/Overlap%E2%80%93add_method

2.1.4.2 Real Time Signal Estimation from Modified STFT Power Spectrum

Because this is only adopted in the first model and the program for it is already available, the theory is not deeply studied. The main procedure is transforming the spectrum with no phase information back to signal and then the signal is transformed to STFT spectrum. This repeats more than 10 times and if the overlapping is more than 90%, the estimation will be accurate.

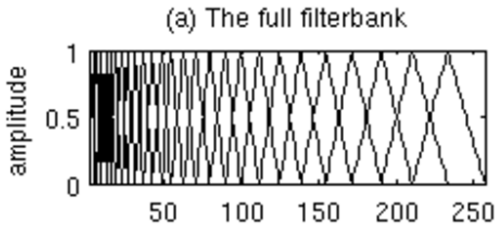
Since the procedure repeat many times, it takes long time to estimate the signal. In addition, if the overlapping is less than 75%, the speech will be highly distorted and inaccurate.

2.2 Mel Frequency Cepstral Coefficients

In this section, a brief introduction of MFCC will be given. They include mel frequency filter and the following operation on the coefficients

2.2.1 Mel Scale Filtering

The main idea behind mel scale filter is the different importance of each frequency. Human ear is most sensitive to 20Hz to 8000Hz and this frequency range contains most of the speech information. Thus, the STFT power spectrum is filtered and only the most important spectrum with certain frequency is kept. One common filter is triangular window as shown in figure 6.



The mel scale is calculated by the frequency

$$m = 2595log_{10}(1 + \frac{f}{700})$$

Figure 5: Mel-Scale Filter from Mel Frequency Cepstral Coefficient (MFCC) tutorial:

http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/

Here is the operation

First, the number of filter bank, lowest frequency and highest frequency need to be determined. For example, they are 10, 300Hz and 8000Hz.

Second, the low frequency and high frequency is transfer to mel scale. In this case, they are 401.25 and 2834.

In this example 10 filter banks are needed and 10+2=12 points are required. The 10 additional points are linearly spaced between 401.25 and 2834.

m[i] = [401.25, 622.50, 843.75, 1065.00, 1286.25, 1507.50, 1728.74,
1949.99, 2171.24, 2392.49, 2613.74, 2834.99]

Then, it is transformed back to Hz

h[i] = [300, 517.33, 781.90, 1103.97, 1496.04, 1973.32, 2554.33,
3261.62, 4122.63, 5170.76, 6446.70, 8000]

Since Discrete STFT power spectrum has frequency resolution, exact frequency above cannot be found in the power spectrum. Thus, the frequency need to be rounded to nearest FFT bin.

$$f[i] = (FFT\ size + 1) * h[i]/sample\ rate$$

f[i] = [9, 16, 25, 35, 47, 63, 81,104,132,165,206,256]

Then, the final filter bank will be

$$H_m(k) = \begin{cases} 0\ for\ k < f[m-1]\ or\ k > f[m+1] \\ \dfrac{k - f[m-1]}{f[m] - f[m-1]}\ for\ f[m-1] \le k \le f[m] \\ \dfrac{f[m+1] - k}{f[m+1] - f[m]}\ for\ f[m] \le k \le f[m+1] \end{cases}$$

where m=1 to 10. As a result, H(k) will be a 10 x FFT size matrix

2.2.2 MFCC

The main reason of using MFCC in signal processing is the robustness in representing the speech property and information. As mentioned in 2.2.1 Mel-Scale filter, only the coefficients with most information are kept and others are filtered. As a result, the input size to the model will be reduced and it will be computational efficient for automatic speech recognition system.

As shown in Figure 5, the operation of MFCC follow four steps.

Step 1: Take STFT and get the power spectrum

Step 2: Apply Mel-Scale Filter on the spectrum (e.g 26 filter bank) and get 26 coefficients

$$Mel - Frequency\ Spectrum\ =\ H(k) \cdot Power\ Spectrum$$

where · is dot operation.

Step 3: Take the logs of the spectrum on each frame and it gives the 26 logfbank coefficients

Step 4: Take the discrete cosine transform on the logfbank 26 coefficients to give 26 cepstral coefficients.

The resulting 26 coefficients are called Mel Frequency Cepstral Coefficients and for automatic speech recognition system, only the lower 12 or 13 coefficients are kept. In addition, the figure 5 shows that differentiation is performed on the cepstral coefficients. The resulting coefficients are called Deltas coefficient and it is not used in this project.

- The MFCC feature
- The logfbank feature

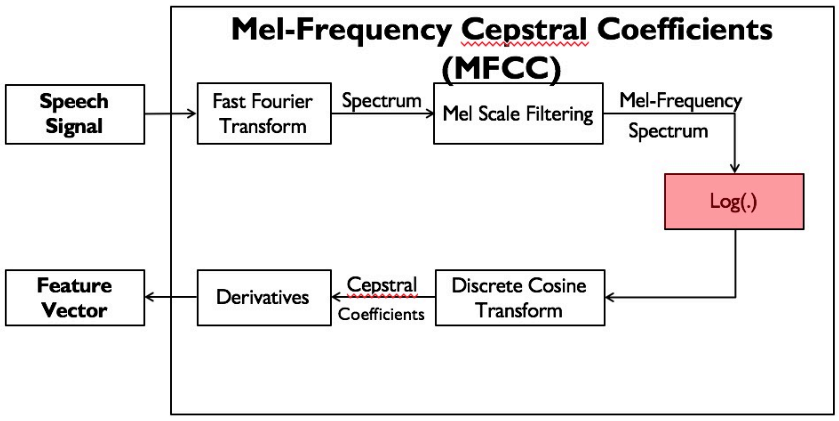


Figure 6: MFCC operation from「冠軍之道」Apparent Personality Analysis 競賽經驗分享: https://zhuanlan.zhihu.com/p/23176872

2.2.3 Inverse MFCC

$$Mel - Frequency\ Spectrum\ = \ H(k) \cdot Power\ Spectrum$$

$$H(k)^{-1} \cdot Mel - Frequency\ Spectrum\ = Power\ Spectrum$$

where the three variables are all matrix and $H(k)$ is (No. of filter bank x FFT size) matrix. Since $H(k)$ is not a square matrix, it is not possible to recover the original power spectrum with mel-frequency spectrum. In other word, performing MFCC is lossy operation.

In this project, Moore–Penrose pseudoinverse is used to estimate

$$H(k)^{-1} = H^{+} = (H^{T}H)^{-1}H^{T}$$

Here is the detail operation of Inverse MFCC.

Step 1: Take Inverse DCT on the modified MFCCs

$$Logfbank\ = \ IDCT(MFCCs)$$

Step 2: Take power

$$Mel\ Frequency\ Spectrum\ = \ 10^{Logfbank}$$

Step 3: Multiple the mel frequency spectrum with Moore–Penrose pseudoinverse of mel scale filter

$$STFT\ spectrum\ = \ H^{+} \cdot Mel\ Frequency\ Spectrum$$

Step 4: Perform OLA or Real Time Signal Estimation from Modified STFT Power Spectrum

2.3 Signal to Noise Ratio (SNR) Estimation

In this project, two methods of SNR estimation are proposed. The first one is simple SNR.

$$SNR = \frac{Estimated\ Clean\ Spectrum}{Estimated\ Noise\ Spectrum} = \frac{\hat{X}^2(\lambda,\delta)}{\hat{\sigma}_d^2(\lambda,\delta)}$$

Thus, the restored clean spectrum will be:

$$X_{restored}^2(\lambda,\delta) = Y^2(\lambda,\delta) * \frac{SNR}{1-SNR} = Y^2(\lambda,\delta) * G(\lambda,\delta)$$

The second one is a Posteriori SNR Controlled Recursive Averaging (PCRA) approach. Stacked Denoising Autoencoder is very powerful in removing the background noise. However, the direct use of restored clean power spectrum calculated above will result in distorted speech when the period of speech is highly covered by background noise. Thus, PCRA takes this consideration and recover the distorted by using the speech information of previous frames and the noisy speech.

Here is the detailed operation:

Step 1: *a Posterior* SNR take the consideration of noisy spectrum and noise spectrum

$$\gamma(\lambda,\delta) = \max\left(\frac{Y^2(\lambda,\delta)}{\hat{\sigma}_d^2(\lambda,\delta)}, 1\right)$$

where $\lambda\ and\ \delta$ are the frame index and frequency bin, $\hat{\sigma}_d^2(\lambda,\delta)$ is the restored noise spectrum and $Y^2(\lambda,\delta)$ is the noisy spectrum

Step 2: First Order Recursive Averaging is performed on the SNR for reducing the fast fluctuations along the time. And the compare the SNR with a predefined threshold T_γ and get $I(\lambda,\delta)$. It is assumed that if the speech component dominates on λth frame and ith bin, SNR will be high thus $I(\lambda,\delta) = 1$ indicate the presence of speech component. Then, first order recursive averaging is also performed on $I(\lambda,\delta)$ to reduce the fast fluctuations and it becomes probability function which indicate the probabilty of the speech component dominating on λth frame and δth bin,.

$\bar{\gamma}(\lambda) = \alpha_\gamma \bar{\gamma}(\lambda-1) + (1-\alpha_\gamma)\gamma(\lambda)$

$$I(\lambda,\delta) = \begin{cases} 0\ if\ \bar{\gamma}(\lambda,\delta) < T_\gamma \\ 1\ otherwise \end{cases}$$

$$p(\lambda,\delta) = \alpha_p p(\lambda-1,\delta) + (1-\alpha_p) I(\lambda,\delta)$$

where α_γ and α_p are smoothing factor

Step 3: In order to take the previous frames into consideration, the probability function $p(\lambda,\delta)$ will become a smoothing factor for step 4.

$$\alpha_\xi(\lambda,\delta) = \alpha_{\xi min} + (1-p(\lambda,\delta))(\alpha_{\xi max} - \alpha_{\xi min})$$

where $\alpha_{\xi min}$ and $\alpha_{\xi max}$ are the minimum and maximum values of the smoothing factor.

Step 4: The *a priori* SNR takes the consider of previous frames, the noisy spectrum and noise spectrum. Therefore, it is calculated as follow.

$$\bar{\xi}(\lambda,\delta) = \alpha_{\xi}(\lambda,\delta)\bar{\xi}(\lambda-1,\delta) + (1-\alpha_{\xi}(\lambda,\delta))[\beta\frac{\hat{X}^2(\lambda,\delta)}{\hat{\sigma}_d^2(\lambda,\delta)} + (1-\beta)(\gamma(\lambda,\delta)-1)]$$

in which β is the weighting factor. If β is smaller, more consideration of previous frame is taken.

From the two equations in step 3 and 4, *a priori* SNR is control by the probability function $p(\lambda,\delta)$. If $p(\lambda,\delta) = 0$, the *a priori* SNR of current frame will be closed to the previous frame. It is thought that the restored clean spectrum is over-attenuated and lost some speech information. However, the previous frame and SNR of noisy spectrum and noise spectrum may have the lost information and therefore they are taken into consideration to calculate the *a priori* SNR.

The advantage of PCRA is that it can recover the lost information but the disadvantage is that the method itself cannot accurately distinguish the noise information and speech information. As a result, both of them will be recovered. Therefore, using PCRA will reduce the noise reduction ability of the model and improve the preservation of speech component.

2.4 Wiener Filter

$$G(\lambda,\delta) = \frac{SNR}{SNR+1}$$

$$X_{restored}^2(\lambda,\delta) = Y^2(\lambda,\delta) * G(\lambda,\delta)$$

The Wiener Filter is used in order to take the noise spectrum into consideration. Here is a very simple example for illustration of simple Wiener Filter's effect.

$$X^2(\lambda,\delta) = 50000, Y^2(\lambda,\delta) = 70000 \; and \; \sigma^2(\lambda,\delta) = 20000$$

where $X^2(\lambda,\delta), Y^2(\lambda,\delta) \; and \; \sigma^2(\lambda,\delta)$ are clean, noisy and noise spectrum.

The construction error of estimated clean and noise spectrum may be large while the construction error of noise spectrum is usually smaller. For example, the estimated in this case

$$\hat{X}^2(\lambda,\delta) = 5000 \; and \; \hat{\sigma}^2(\lambda,\delta) = 12000$$

$$Y^2(\lambda,\delta) * G(\lambda,\delta) = \frac{Y^2(\lambda,\delta) * SNR}{SNR+1} = \frac{Y^2(\lambda,\delta)\hat{X}^2(\lambda,\delta)}{\hat{X}^2(\lambda,\delta) + \hat{\sigma}^2(\lambda,\delta)} = 20588$$

which is closer to $X^2(\lambda,\delta)$.

Since estimated noise spectrum is always more accurate than estimated clean spectrum, applying Wiener Filter may help to improve the construction error of estimated clean spectrum.

Although it is not rigorous proof of Wiener Filter's effect, the filter always improve the speech quality in real application.

2.5 Programming Language

In this project, python is used as the programming language because it has a nice application of GPU computation and many libraries about machine learning. The machine learning libraries mainly include Keras, Theano and Tensorflow.

2.6 Stacked Denoising Autoencoder

In section, a brief introduction of stacked denoising autoencoder(SDAE) will be given. The sub-section includes autoencoder, denoising autoencoder and SDAE.

2.6.1 Autoencoder

In few years ago, restrict bolzmann machine (RBM) was firstly introduced to build a deep belief network (DBN). However, it is difficult to use tradition optimization algorithms to train the network. As a substitute, the AE is used as an equivalent module to the RBM in building DAE. One advantage of AE is that many traditional optimization algorithms are already to be used in training.

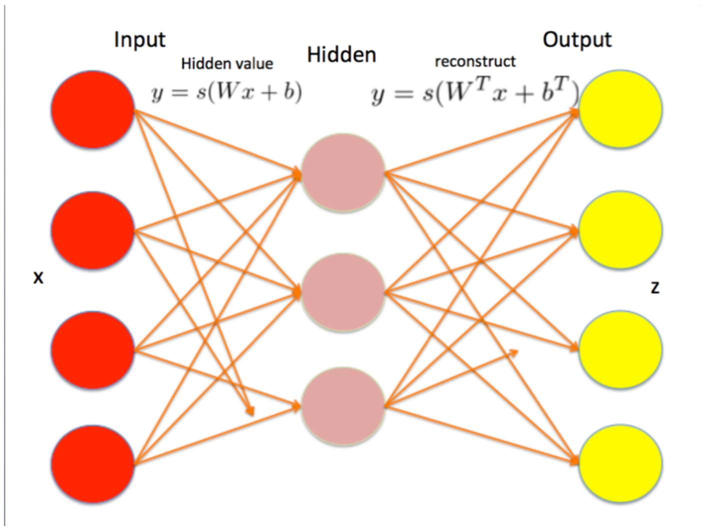


Figure 7: Autoencoder from Deep Learning Tutorials Denoising AutoEncoder, てれか

As shown in figure 7, Autoencoder is one type of fully connected neural network while the hidden layer's size is usually smaller than the input and output layer's size. It includes one nonlinear encoding stage and one linear decoding stage. In encoding stage, the inputs are compressed into some coefficients with smaller size while the coefficients are decompressed back to original inputs in decoding stage thus it can be thought of compression and decompression. It is thought that if the input can be compressed into coefficients with smaller size, it may be a good representation of the input.

Assuming the output is x and input is y.

$$h(y_i) = \sigma(W_1 y_i + b)$$

$$\hat{x} = W_2 h(y_i) + c$$

where W_1 and W_2 are encoding and decoding matrix for the neural network connection weight. b and c are the vector of bias for hidden and output layers. The weights and biases are determined by optimizing the mean square error loss

$$L(\Theta) = \frac{1}{N}\sum ||x_i - \hat{x}_i||_2^2$$

where Θ is the parameter set (W_1, W_2, b, c) and x_i is the desired coefficients while $\hat{x}_i$ is the predicted coefficients.

2.6.2 Denoising Autoencoder

Autoencder has a special version Denoising Autoencoder in which the input to DAE is a distorted output.

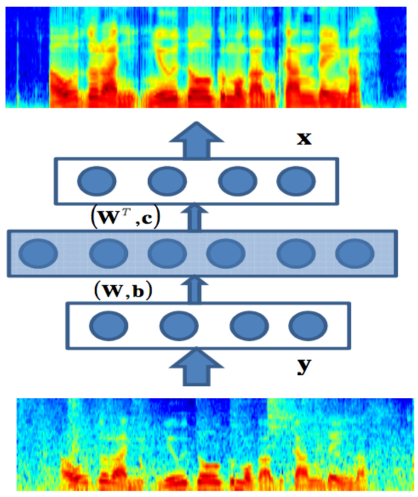


Figure 8: Denoising Autoencoder with noisy input y and clean output x from [17]

As shown in Figure 8, denoising Autoencoder is similar to Autoencoder except that the input and output are the same for Autoencoder while the input is noisy version of output for Denoising Autoencoder. It is thought that the noise is at higher dimension and if the input is compressed into coefficients with smaller size, the coefficients may mostly have the clean information and the noise information is filtered.

2.6.2 Stacked Denoising Autoencoder

By stacking several DAE, a SDAE is built as shown in figure 9. SDAE is commonly trained with pre-training. In pre-training stage, each layer of SDAE is separately trained as DAE. Detail refers to 2.7.7 pre-training part. After pre-training stage, the DAEs are stacked together and form a multiple layer neural network. Then, it is trained as normal multiple layer neural network and this stage is called fine tuning stage.

However, since pre-training may not be useful if the amount of training data is large, SDAE can also be trained without pre-training stage.

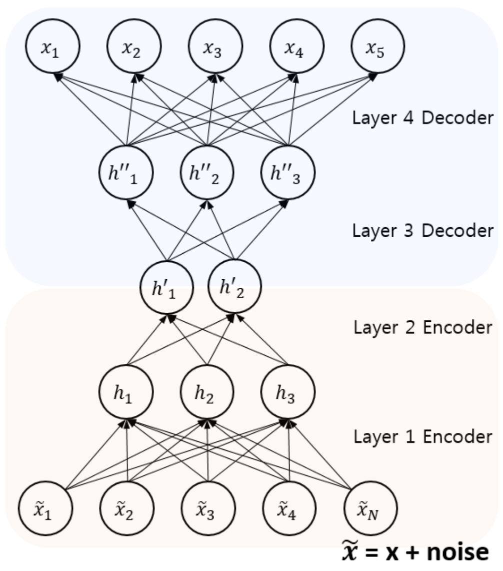


Figure 9: Stacked Denoising Autoencoder from Introduction Auto-Encoder: https://wikidocs.net/3413

Similar to Autoencoder, the activation in encoding part is non-linear function while the activation in decoding part is linear function.

$$h_i = h(y_i) = \sigma(W_1 y_i + b_i)$$
$$h'_i = h'(h_i) = \sigma(W_2 h_i + b'_i)$$
$$h''_i = h''(h'_i) = W_3 h'_i + b''_i$$
$$\hat{x}_i = x(h''_i) = W_4 h''_i + b'''_i$$

where W_i is weighting array respectively to layer i and y is noisy input.

The weights and biases of the neural network are determined by optimizing the following equation.

$$L = \frac{1}{N}\sum ||x_i - \hat{x}_i||^2$$

2.7 Training Method

In this section, the training methods will be introduced. They include objective, back propagation, Stochastic gradient decent, activation, regularizer, momemtum and pre-training.

2.7.1 Objective

Since the main task of Stacked Denoising Autoencoder is minimizing the statistical difference between desired output and actual output, mean square error(MSE) or mean absolute error(MAE) may be suitable for it.

$$MSE: L = \frac{1}{N}\sum ||x_i - \hat{x}_i||^2$$

$$MAE: L = \frac{1}{N}\sum |x_i - \hat{x}_i|$$

Moreover, another objective function, weighted square error(WMSE), is also proposed in this project. In MSE, all the input neurons have the same importance. However, different input neurons represent difference frequency for power spectrum as input. For example, a 44100Hz audio signal is transfer into STFT with 882(20ms) FFT size. Then, the frequency resolution is 44100/882=50Hz. In other word, the 1st spectrum coefficient is 0Hz and 2nd is 50Hz. Since sensitivity of human ear to different frequency is different, it is reasonable to weight different frequency.

$$WSE\colon L = \sum |F_i * |x_i - \hat{x}_i||^2$$

or in matrix form

$$WSE\colon L = ||F \circ (x - \hat{x})||^2$$

where $\circ$ is element wise multiplication and F is the weighting function.

In this project, Stochastic Back-Propagation training algorithm is applied to train the SDAE. The following show the effect of weighting function.

For simplicity, assume the DAE has one hidden layers, activation in encoding part is leaky relu and activation in decoding part is linear, the gradient of L with respective to

$$\hat{x} = a_2 = W_2 h + b_2$$

$$\frac{\partial L}{\partial a_2} = 2F \circ (\hat{x} - x)$$

The respective update of $W_2\ and\ b_2$ is:

$$\Delta W_2 = \varepsilon \frac{\partial L}{\partial W_2} = \varepsilon \frac{\partial L}{\partial a_2} \frac{\partial \hat{x}}{\partial W_2} = 2\varepsilon F \circ (\hat{x} - x) h^T$$

$$\Delta b_2 = \varepsilon \frac{\partial L}{\partial b_2} = \varepsilon \frac{\partial L}{\partial a_2} \frac{\partial \hat{x}}{\partial b_2} = 2\varepsilon F \circ (\hat{x} - x)$$

where ε is the learning rate

Since the activation of hidden is Leaky Relu, h is expressed as:

$$h(m) = \sigma(a_1(m)) = \begin{cases} a_1(m)\ if\ a_1(m) > 0 \\ \alpha a_1(m)\ otherwise \end{cases}$$

$$\frac{\partial h(m)}{\partial a_1(m)} = \begin{cases} 1\ if\ a_1(m) > 0 \\ \alpha\ otherwise \end{cases}$$

Thus, the gradient of L with respective to

$$u_1 = W_1 y + b_1$$

$$\frac{\partial L}{\partial a_1(m)} = \frac{\partial L}{\partial h(m)}\frac{\partial h(m)}{\partial a_1(m)} = \begin{cases} \frac{\partial L}{\partial h(m)}\ if\ a_1(m) > 0 \\ \alpha \frac{\partial L}{\partial h(m)}\ otherwise \end{cases}\ for\ m = 1,2,\dots,N_{hid}$$

where N_{hid} is the hidden layer size.

Reference to the back-propagation,

$$\frac{\partial L}{\partial h} = W_2^T \frac{\partial L}{\partial a_2} = W_2^T[2F \circ (\hat{x} - x)]$$

Then, the respective update of $W_1\ and\ b_1$ is:

$$\Delta W_1 = \varepsilon \frac{\partial L}{\partial W_1} = \varepsilon \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial W_1} = \varepsilon \frac{\partial L}{\partial a_1} y^T$$

$$\Delta b_1 = \varepsilon \frac{\partial L}{\partial b_1} = \varepsilon \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial b_1} = \varepsilon \frac{\partial L}{\partial a_1}$$

From the equation above, $\Delta W_1, \Delta b_1, \Delta W_2\ and\ \Delta b_2$ are controlled $\frac{\partial L}{\partial a_1}$ and $\frac{\partial L}{\partial h}$ which is proportional to the weighting function. Thus, they are controlled by the weighting function F which makes it possible to set an appropriate learning rate for each bin.

In this paper, the weighting function F is obtained by absolute threshold for hearing (ATH) which defines the minimum sound intensity in dB required to be detected by an average human ear with normal hearing in a quiet environment. The ATH is varied with different frequency and it is minimum for frequency around 3000Hz to 4000Hz. It is assumed that if ATH of a certain frequency is smaller which means the frequency is more detectable by human ear, the frequency is more important. The difference in importance of frequency can be used to construct the weighting function F.

Here is the detail operation.

Step 1: Since the frequency bins of spectrum represent different frequency, calculate ATH for different frequency bin.

For example, a 44100Hz audio signal is transformed to STFT spectrum with 882(20ms) FFT size. Then, the frequency resolution will be 44100Hz/882=50Hz.

$$Feq[\delta] = 50(\delta - 1)\ for\ 1 \le \delta \le 882$$

$Feq[i]$ just refer to the center frequency of each frequency bin instead of the magnitude of STFT spectrum.

$$ATH[\delta] = 3.64(\frac{Feq[\delta]}{1000})^{-0.8} - 6.5e^{-0.6(\frac{Feq[\delta]}{1000}-3.3)^2} + 10^{-3}(\frac{Feq[\delta]}{1000})^4\ for\ 1 \le \delta \le 882$$

Step 2: Shift the ATH for making the minimum 1 and Calculate the frequency importance weights $\mathcal{F}$

$$\mathcal{F} = \frac{1}{ATH[\delta]_{shifted}}$$

Since $Feq[i]$ only represent the center frequency of each frequency bin, the bin 1 not only refer to 0Hz. Thus, it is unreasonable that $w_g[1] = 0$. In order to avoid assigning $Feq[1] = 0, ATH[1] = \infty \; and \; w_g[1] = 0$ to frequency bin 1, the 4th $w_g[4]$ is assigned to bin 1. $w_g[1] = w_g[4]$

2.7.2 Shuffle Input

In training the neural network, overfitting should be prevented and the network should only learn the internal relationship between input and output or the distribution of input itself. Therefore, the sequence of training example should not be fixed in order to prevent that the network learn the sequential information between each training example.

In the project, all the training examples are shuffled first before going to the next epoch.

2.7.3 Stochastic Gradient Descent

The backpropagation training requires the calculation gradient of objective function for each single training example. However, it will be computational difficult with large amount of data. Therefore, the gradient is approximated with a large amount of training example, or m amount of training example where m is called mini-batch size.

The following is the step of updating weight and bias for mini batch.

Step 1: Input m set of training example to the network.

Step 2: Calculate the responding output error and error of each layer with backpropagation algorithm.

$$\delta^{i,L} = \nabla C_x \odot \hat{x}'$$
$$\delta^{i,l} = ((w^{l+1})^T \delta^{i,l+1} \odot \hat{x}')$$

where $1 \leq i \leq m, 1 \leq l \leq L$, m is mini-batch size and L is number of layer.

Step 3: Update the weight and bias with the average gradient

$$w^l -> w^l - \frac{\varepsilon}{m} \sum \delta^{i,l} \, (a^{i,l-1})^T$$

$$b^l -> b^l - \frac{\varepsilon}{m} \sum \delta^{i,l}$$

where $a^{i,l-1}$ is the output vector of previous layer.

2.7.4 Activation

In the project, several activations are proposed.

Linear Function:

$$h_i = a_i = W_i h_{i-1} + b_i$$

This activation is necessary for Stacked Denoising Autoencoder. Without this activation in decoding stage, the construction error is much larger.

Sigmoid Function:

$$h_i = S(a_i) = \frac{1}{1+e^{-a_i}} = \frac{1}{1+e^{-(W_i h_{i-1}+b_i)}}$$

The shape of this activation is shown in Figure10. This is a very popular activation and has an important property that the derivative of the activation is computational simple.

$$S'(a_i) = S(a_i)(1 - S(a_i))$$

Since back-propagation algorithm requires the calculation of $\sigma'(a_i)$ which is the derivation of activation, adopting sigmoid function as activation makes the training computational simple and faster.

In most research papers about Stacked Denoising Autoencoder, sigmoid is used as activation in encoding stage.

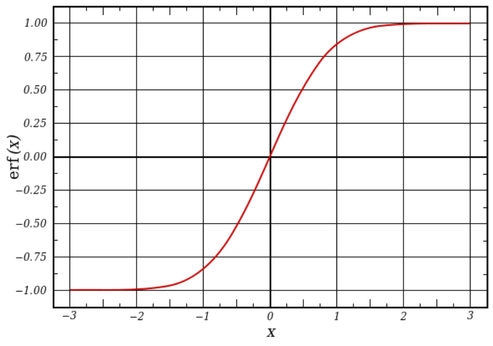


Figure 10: Sigmoid Function

ReLU:

$$\sigma(a_i) = max(0, a_i)$$

ReLU is first introduced by Hahnloser et al in a 2000 paper in Nature which he explains the strong biological motivation of using ReLU for neural network and human neuron acts similar to ReLU. If the biological signal to a neuron is less than some threshold, the neuron will not response to the signal. Similar to human neuron, the artificia neuron adopting ReLU as activation do not response to the input and output zero if the input is less than zero.

In addition, ReLU has an advantage of sparse representations of input.

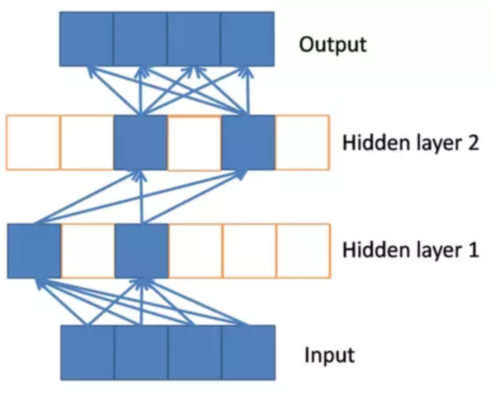


Figure 11: Sparsity of ReLU from Xavier Glorot, Antoine Bordes and Yoshua Bengio

Sparsity has some advantages in training the neural network. First, dense representations can become strongly correlated during training, and this causes the network to overfitting because the abstract features encoded in the hidden units become entangled. Second, a goal of a neural network is to have a distributed representation of input related to output and a sparse set may be more easily distributed because there are few interactions across the network.

Since ReLU is efficient and robust in enhancing training of neural network, it was the most popular activation for deep neural network in 2015 [7].

Leaky ReLU:

Original ReLU has a serious problem called Dying ReLU problem. The ReLU neurons sometimes are in the state that they become inactive and output zero for all inputs. In this state, gradients flow backward through the neurons is zero and thus the neurons stick to this point and "die". For stacked denoising autoencoder, the number of neurons in the middle is few and therefore Dying ReLU problem may lead to large construction error.

Leaky ReLU is an attempt to solve the problem.

$$\sigma(a_i) = \begin{cases} a_i \, for \, a_i > 0 \\ \alpha a_i \, otherwise \end{cases}$$

where α is a parameter and usually very small.

The neurons can have response respect to all input, as a result it has gradient flow backward through the neurons and the neurons will not stick to some points.

2.7.5 Regularization

In the project, L2 regularization is applied to regulate the weighting. The main idea of L2 regularization is to add an extra term, called the regularization term, to the objective function. Here's the regularized mean square loss:

$$L = \frac{1}{n}\sum ||x - \hat{x}||_2^2 + \frac{\lambda}{2n}\sum w^2 = L_{MSE} + \frac{\lambda}{2n}\sum w^2$$

$$\frac{\partial L}{\partial w} = \frac{\partial L_{MSE}}{\partial w} + \frac{\lambda w}{n}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L_{MSE}}{\partial b}$$

where the regularization term is the sum of the square of all the weights in the network.

The main purposes of L2 regularization are to make the neural network learning small weight and avoid overfitting. For example, the small weight will need to small loss if L_{MSE} remain the same. Large weight is possible when the large weight will improve L_{MSE} greatly. Therefore, L2 regularization can be thought of a compromise between finding a small weight and improve the loss of original objective function. Their relative importance depends on the regularization parameter λ. The larger parameter, the more importance of small weight. Small weight is preferable because it is thought that smaller weights are lower complexity and provide a simple as well as powerful representation of the data.

In addition to L2 regularization, a weight constraint is applied on the network.

$$w_{ij} = min(w_c, w_{ij})$$

where w_c is the maximum value for weight. Weight constraint is applied in order to prevent the weights becoming larger and larger because of unknown reason.

2.7.6 Momentum

In training neural network, traditional back propagation algorithm only considers the gradients decent of objective function respective to the variable, such as

$$w -> w' = w - \varepsilon \nabla L$$

where w is the variable matrix and $w_1, w_2, w_3 \ldots.$ are the variable the neural network is going to train. In this project, only the weight w and bias b are trained.

However, the rate of change of variable may have some useful information in training the variable. Thus, a new term called Momentum is introduced in the equation above.

$$v -> v' = \mu v - \varepsilon \nabla L$$

$$w -> w' = w + v'$$

where μ is the momentum coefficients.

From the equations above, if μ is non-zero, the previous gradients decent of objective function μv is also considered to train the variables w. As shown in figure 12, w, v, μ and $\varepsilon \nabla L$ can be thought of the position, velocity, friction of ball and a net force(gravity) on the ball. If the direction of ball is downward, the velocity will get larger and the ball will go to the bottom faster. If the direction of ball is upward and the ball overshoot the minimum point, the friction $0 < \mu < 1$ will slow down the ball and finally the ball will go to minimum point. Therefore, the introduction of momentum will make the training of neural network faster.

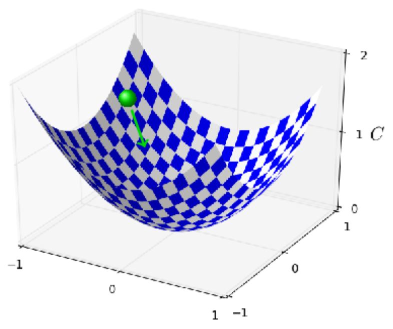


Figure 12: Training Graph from Michael Nielsen

2.7.7 Pre-Training

Pre-Training is a powerful technic to initialize the weights and bias for SDAE.

Assume a 3 hidden layers SDAE is pre-trained and the input, desired output and actual output are $y, x \text{ and } \hat{x}$ as shown in figure 13. Each layer, h1,h2 and h3, are trained as a signal hidden layer denoising autoencoder. For h1:

$$a_1 = \sigma(W_{11}y + b_{11})$$

$$\hat{x} = W_{12}a_1 + b_{12}$$

$$L = \frac{1}{N}\sum ||x - \hat{x}||^2$$

where $W_{11}, W_{12}, b_{11}, b_{12}$ are determined by minimizing the objective function.

Then, the hidden layer of h1 DA predict the input and desired output of the next DA with $y \text{ and } x$. Such as:

$$Input\ of\ h2\text{: } \sigma(W_{11}y + b_{11}) = a_1$$

$$Desired\ output\ of\ h2\text{: } \sigma(W_{11}x + b_{11}) = x_1$$

$$Actual\ output\ of\ h2\text{: } \hat{x}_1$$

$$a_2 = \sigma(W_{21}a_1 + b_{21})$$

$$\hat{x}_1 = W_{22}a_2 + b_{22}$$

$$L = \frac{1}{N}\sum ||x_1 - \hat{x}_1||^2$$

Similar to previous operation,

$$Input\ of\ h3\text{: } \sigma(W_{21}a_1 + b_{21}) = a_2$$

$$Desired\ output\ of\ h2\text{: } \sigma(W_{21}x_1 + b_{21}) = x_2$$

$$Actual\ output\ of\ h2\text{: } \hat{x}_2$$

$$a_3 = \sigma(W_{31}a_2 + b_{31})$$

$$\hat{x}_2 = W_{32}a_3 + b_{32}$$

$$L = \frac{1}{N}\sum ||x_2 - \hat{x}_2||^2$$

After pre-training the three layer, h1,h2,h3 will be stacked to form a SDAE and their respective W_{n1} *and* b_{n1} will be the initial weights and biases.

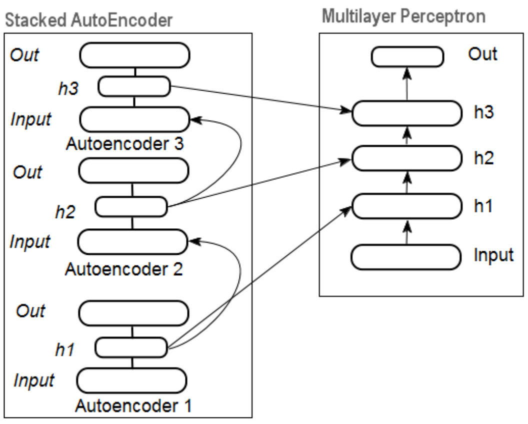


Figure 13: Pre-training of Stacked Autoencoder from THIRD GENERATION NEURAL NETWORKS: DEEP NETWORKS: https://www.mql5.com/en/articles/1103

2.8 Performance Evaluation

In the project, three different evaluations, PESQ, Log-Spectral Distortion and Noise Reduction, are proposed to evaluate the model performance on the perceptual difference between distorted speech and reference speech, level of distortion of speech and level of noise removal.

2.8.1 Perceptual Evaluation of Speech Quality Improvement

PESQ, comprising a testing methodology for automatic evaluation of the distorted speech quality as heard by a real listener, is a worldwide objective method applying to industry for objective speech quality testing. It has different standard and ITU-T Recommendation P.862.1 standard proposed by International Telecommunication Union is adopted in this project. The scoring of PESQ range from -0.5 to 4.5 which the high score represents better speech quality. Since the evaluation of PESQ requires 8000Hz or 16000Hz audio signal while the audio used in the project is 44100Hz, the audio file is down sample to 16000Hz.

In this project, the noise is randomly added to the clean speech and the PESQ of different noisy speech reference to clean speech may be different. Therefore, instead of PESQ, PESQ improvement (PESQI) will be used to evaluate the performance of model. The PESQ of noisy speech reference to clean speech as well as estimated clean speech reference to clean speech will be calculated and PESQI is improvement of PESQ.

The whole process of evaluating PESQ is done by a C# program provide by International Telecommunication Union.

2.8.2 Log-Spectral Distortion Improvement

In addition to PESQI, an objective and mathematical method to evaluate the statistical difference between distorted and original clean spectrum is proposed.

$$D_{LS} = \frac{1}{N_{frame}} \sum \sqrt{\frac{1}{N_{FFT}/2 + 1} \sum (10log_{10} \frac{X^2(\lambda, i)}{\hat{X}^2(\lambda, i)})^2}$$

where N_{frame} is number of frame and $N_{FFT}/2 + 1$ is FFT frequency bin. For example, $1 \leq \lambda \leq N_{frame}$ and $1 \leq i \leq N_{FFT}/2 + 1$. In addition, $\hat{X}^2(\lambda, i)$ can be estimated clean spectrum or noisy spectrum. The value is in dB.

With similar reason of PESQI, only the improvement is considered.

2.8.3 Noise Reduction

In addition to PESQ and evaluation of distortion, an evaluation on the noise reduction performance of model is proposed.

$$NR_{LS} = \frac{1}{N_{frame}} \frac{1}{N_{FFT}/2 + 1} \sum \sum (|\frac{X^2 - Y^2(\lambda, i)}{Y^2}|)$$

3. Methodology

In this section, the data base and source of noise will be explained first and the three models proposed in this project will also be introduced. Two of them are unsuccessful model and therefore a brief description, evaluation and discussion will be given while the final model has detail description and comparison of different setting. The evaluation and discussion of final model will be given in section 4.

3.1 Data Base

The data base, from Marathon Match - Contest: Spoken Languages 2, contain 57560 recorded speech files in 176 languages and they totally last for about 160 hours. Since the data base is large and the training of SDAE do not need all of the data, only 9600 recorded speech files randomly selected are used in the project. 7680 speech files are used for training and 1920 speech files are used for testing. In the comparison part in section 3, only one audio file randomly selected from the data base excluding the data for training is used for validation for fast comparison. In section 4, 20 audio file are selected and a more representative performance will be shown.

3.2 Type of Noise

In this project, only one noise audio file is used. The noise audio file lasts for 33 seconds consists of car and road noise. Since the noise is longer than the speech, a portion, about 9 to 10 seconds, of noise file is randomly picked and added to the speech. Thus, for all speech file, their background noise will be different. In this project, 5dB SNR will be used. Here is the equation of SNR in dB.

$$SNR = 20log_{10}(\frac{A_{signal}}{A_{noise}})$$

where A_{signal} and A_{noise} is the root mean square value of signal and noise.

$$A_{signal} = \sqrt{\frac{1}{N}\sum V_{signal}}$$

$$A_{noise} = \sqrt{\frac{1}{N}\sum V_{noise}}$$

where V is the amplitude of signal in time domain.

Therefore, A_{signal} will be calculated first and $A_{noise,desired}$ is gotten.

$$A_{noise,desired} = \frac{A_{signal}}{10^{\frac{5}{20}}}$$

Then, the amplitude of noise is multiple to meet the desired root mean square.

$$A_{noise,desired} = \sqrt{\frac{1}{N}\sum kV_{noise}} = \sqrt{multipler}\sqrt{\frac{1}{N}\sum V_{noise}}$$

$$multipler = (\frac{A_{noise,desired}}{A_{noise}})^2$$

$$V_{noise,5dB} = kV_{noise}$$

Since the root mean square of signal is calculated by the mean of whole speech, the sound intensity of signal is not constant and sometimes the noise may be small and sometimes it may be louder than the signal. This is for simulating the real situation.

In the project, background noise is considered as additive noise. Therefore, the noisy speech will be the combination of noise and speech.

$$V_{noisy} = V_{noise,5dB} + V_{signal}$$

3.3 Logfbank Model

The first model was proposed at the beginning of this project and an attempt to test the robustness of SDAE. With reference to [18], a similar model adopting Logfbank as input was proposed. First, the noisy speech is transformed into Logfbank coefficients as the input to SDAE which estimated clean Logfbank coefficents. Then, inverse transform is performed and the signal is estimated with Real Time Signal Estimation. The Figure 14 shows the detail of model. In the following subsection, a brief description, performance and discussion of the model will be given.

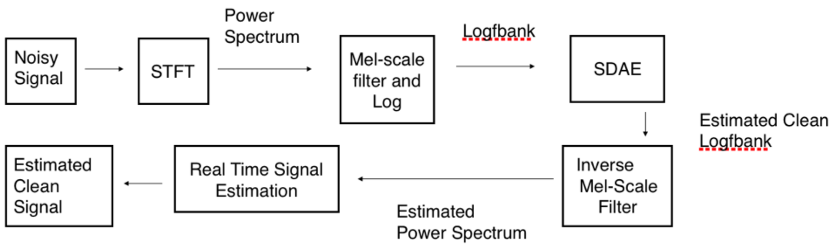


Figure 14: Logfbank Model

3.3.1 Logfbank Coefficients

The noisy signal is transformed to Logfbank. The following is the setting.

$$FFT\ size: 882\ (20ms)\ with\ 735\ overlapping$$

$$Mel - Scale\ Filter: 40\ with\ 20Hz\ Lowest\ Frequency\ and\ 8000Hz\ Highest\ Frequency$$

The detail of transformation can refer to section 2.1 and 2.2.

Thus, each frame has 40 coefficients. Then, 11 neighboring frames are combined together to form the input and desired output for SDAE.

$$y = [\delta_{noisy}[\lambda - 5], \ldots\ldots, \delta_{noisy}[\lambda], \ldots\ldots, \delta_{noisy}[\lambda + 5]$$

$$x = [\delta_{clean}[\lambda - 5], \ldots.., \delta_{clean}[\lambda], \ldots.., \delta_{clean}[\lambda + 5]$$

where $\delta[\lambda]$ indicates the Logfbank coefficients at $\lambda\, frame$

Therefore, the length of $x\ and\ y$ will be 440.

3.3.2 SDAE

The structure of SDAE is shown in Figure 15. However, Keras, the machine learning library provider, does not support Leaky ReLU as the activation in the first layer and thus it is replaced by linear function. It is expected the performance will be similar.

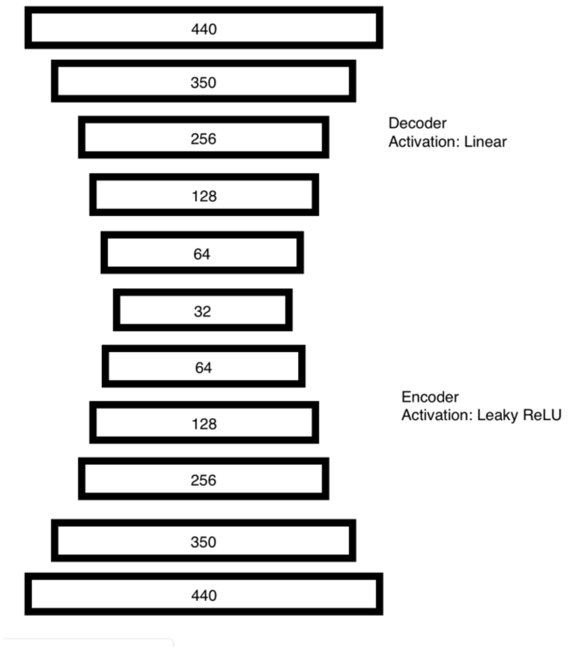


Figure 15: SDAE of Logfbank Model

Greedy layer-wise pre-Training is performed to minimize the construction error between each layer. The detail of pre-training refers to section 2.7.7.

The network is trained to minimize the MSE by Stochastic Gradient Descent with 0.005 learning rate, 0.9 momentum coefficient, 512 batch size, 0.0002 L2 coefficients, 0.9 weight constraint and 0.3 Leaky ReLU coefficients.

$$L = \frac{1}{N} \sum ||x - \hat{x}||^2$$

3.3.3 Inverse Logfbank

After getting the estimated logfbank coefficients from SDAE, inverse Mel-Scale Filter and Real Time Signal Estimation. Their respective detail refers to section 2.2.3 and 2.1.4.2.

3.3.4 Performance Evaluation

snr=5	PESQ (With original speech as reference)	LSD Improvement(dB)	Noise Reduction
Noisy speech	1.515		
Estimated Clean Signal	1.487	15.0817	0.911

Before calculating LSD Improvement and NR, it performs inverse logarithm for having same scale of value with power spectrum.

$$y_{evaluation} = 10^{y} \ x_{evaluation} = 10^{x} \ \hat{x}_{evaluation} = 10^{\hat{x}}$$

Therefore, the estimated clean speech from the model has a worse perceptual speech quality than the noisy speech. However, LSDI and Noise Reduction get a satisfactory result. The main reason may be the distortion by Mel-Scale filter.

3.3.5 Distortion by Mel-Scale Filter

Logfbank and MFCC are filtered by mel-scale filter and only the most important frequencies are kept while others are filtered. Therefore, they have most of the word information and advantage of low input size. For ASR system, they are very common and powerful feature of speech as input because their low input size can make the recognition faster while the recognition accuracy will not be affected.

However, for Logfbank and MFCC, the less important frequencies are filtered and this operation is non-inversible. Therefore, perceptually the restored speech is highly distorted even the word information is kept.

In order to examine the distortion of mel-scale filter, one speech file is randomly selected conduct an experiment.

Step 1: Transform the speech signal to STFT spectrum with 882 frame size and 735 overlapping.

Step 2: Take mel-scale filter with 40 mel-components and log on the spectrum

Step 3: Perform inverse operation and get the restored signal.

The resulting PESQ score is 1.88 while PESQ range from -0.5 to 4.5 and the higher score means better speech similarity with the reference speech. PESQ with 1.88 is definitely a bad result. To visual compare the two signal, the figure 16,17 shows the signal of original speech and restored speech. From the two figure, it is obvious that the overall shape of signal is kept while some detail of original signal is distorted.

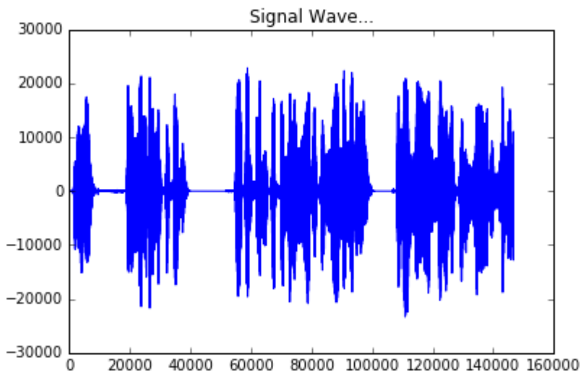


Figure 16: The original Signal

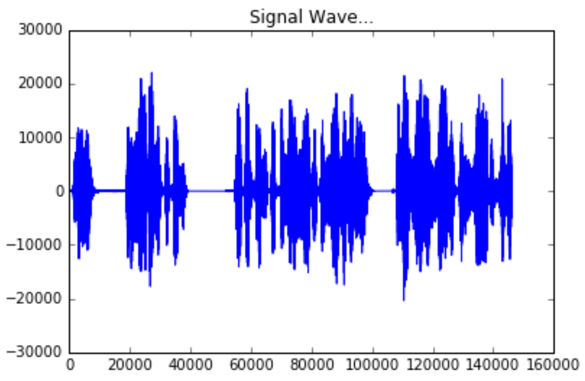


Figure 17: The Signal restored from logfbank

3.3.6 Discussion

The distortion of Mel-Scale Filter largely contributes to the low PESQ in the result. However, LSDI and Noise Reduction has a relatively good result which suggests SDAE can improve the quality of Logfbank coefficients and remove the noise. For ASR system, the input usually is MFCC and Logfbank coefficients and the performance of system may not be affected by the distortion of Mel-Scale Filter. Therefore, this model may have potential of noise removal for ASR system.

3.4 Second Model

The Second model was proposed after the problem mentioned in section 3.3.5 was fully understood. With reference to [3], a similar model adopting power spectrum as input was proposed. First, the noisy speech is transformed into STFT spectrum as the input to SDAE which estimated clean spectrum. Then, inverse transform is performed and the signal is estimated with Real Time Signal Estimation. The Figure 18 shows the detail of model. In the following subsection, a brief description, performance and discussion of the model will be given.

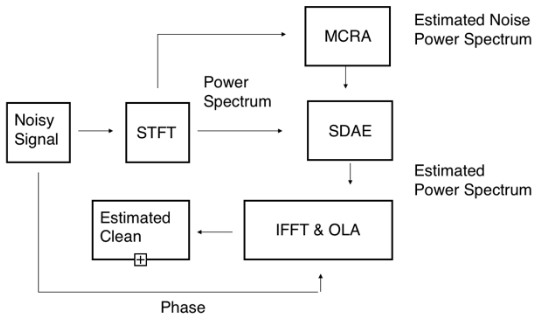


Figure 18: Spectrum Model

3.4.1 Type of Input

Since MFCC and Logfbank have the problem mentioned in section 3.3.5, power spectrum is considered as a better choice in speech enhancement.

Therefore, the power spectrum is selected as the input and the power spectrum is calculated with 882 frames size(20ms) and 441(10ms) overlapping.

$$spectrogram = |X[\lambda, i]|^2 = X^2(\lambda, i)$$

3.4.2 SDAE

The structure of SDAE is shown in Figure 19. However, Keras, the machine learning library provider, does not support Leaky ReLU as the activation in the first layer and thus it is replaced by linear function. It is expected the performance will be similar.

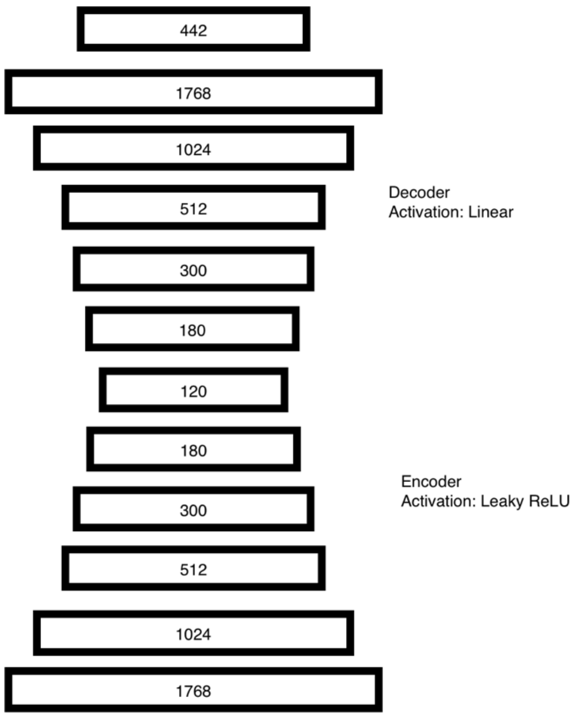


Figure 19: SDAE of Spectrum model

3.4.2.1 Input to SDAE

The input of SDAE is noisy spectrum and desired output is clean spectrum. Such that

$$Input = Y^2[\lambda,\delta] = y$$

$$Desired\ Output = X^2[\lambda,\delta] = x$$

However, the input and output is large numerically and the scale of difference between Input and Desired Output range from $10^1\ to\ 10^{10}$. At the beginning of training, the actual output $\hat{X}^2[\lambda,i]$ must be greatly different from the desired output when the speech is not presence in clean speech. For example, $x \approx 0$ because speech is not presence while $y \approx 10^{10}$ and $\hat{x} \approx 10^9$ because the noise is presence.

Therefore, the update of weight and bias will be large.

$$\Delta W_{l-1} = \varepsilon \frac{\partial L}{\partial W_{l-1}} = \varepsilon \frac{\partial L}{\partial a_{l-1}} \frac{\partial \hat{x}}{\partial W_{l-1}} = 2\varepsilon(\hat{x} - x)h^T$$

$$\Delta b_{l-1} = \varepsilon \frac{\partial L}{\partial b_{l-1}} = \varepsilon \frac{\partial L}{\partial a_{l-1}} \frac{\partial \hat{x}}{\partial b_{l-1}} = 2\varepsilon(\hat{x} - x)$$

where l is the number of layers.

Since $\hat{x} - x$ is large, the update of $\Delta W_{l-1}\ and\ \Delta b_{l-1}$ are also large. However, $\hat{x} - x$ may be small for when the speech is presence. This problem will need to unstable update of $\Delta W_{l-1}\ and\ \Delta b_{l-1}$ and finally the training loss will go to infinity.

In order to solve the unstable training:

$$Input = log_{10}(Y^2[\lambda,\delta]) = y$$

$$Desired\ Output = log_{10}(X^2[\lambda,\delta]) = x$$

Taking logarithm on input can minimize the scale of difference between Input and Desired Output and solve this problem.

3.4.2.2 Feature Expansion

For the SDAE estimating the clean spectrum, a feature expansion on input is adopted while the desired output remains the same.

$$Input = [y[\lambda - 1], y[\lambda], y[\lambda + 1], n[\lambda]]$$

$$Desired\ Output = x[\lambda]$$

The previous and following frame of noisy input are taken into consideration because they may have some information needed to estimate the current frame.

In addition, the input is expended by feeding the estimated noise spectrum based on the thought that feeding information of noise as the input to the neural network is beneficial for speech recognition [3]. This expansion is called noise aware and the noise spectrum is estimated with MCRA.

3.4.2.3 Training Method

The network is trained to minimize the Mean Square Error by Stochastic Gradient Descent with 0.0005 learning rate, 0.9 momentum coefficient, 512 batch size, 0.02 L2 coefficients, 0.9 weight constraint and 0.3 Leaky ReLU coefficients.

3.4.3 Inverse STFT with OLA

Real Time Signal Estimation is not used in this model and final model because it is computational difficult and it usually takes long time to estimate the phase. Besides, a good estimation of phase requires over 90% overlapping which makes the training data size becomes large. For example, a 1.8MB wav file will expend to 18MB data of power spectrum with 90% overlapping. This should be avoided because the data has large similarity and it may affect the training of neural network. Therefore, Overlap-Add method is used.

The merit of OLA is that it is computational simple and large overlapping is not required while the demerit is that the use of noisy phase may affect the speech quality. However, Dr. Wang and Lim [5] have public a paper in which they investigated the importance of phase information and they found that the spectrum is more important in restoring the signal than the phase. In order to prove their finding, a similar experiment was done.

The noisy speech is made by the clean speech combined with 5dB noise and they are transformed to STFT spectrum with 882 FFT size and 441 overlapping. Then, OLA and IFFT are performed on the clean spectrum with noisy phase extracted from the noisy STFT coefficients before taking absolute value. The restored clean speech and original clean speech are down-sampled to 16000Hz and PESQ evaluation is conducted.

The resulting PESQ score is 3.241. Although the result is not perfect, it is a tradeoff between the performance and the efficiency of model.

3.4.4 Performance Evaluation and Discussion

snr=5	PESQ (With original speech as reference)	LSD Improvement(dB)	Noise Reduction
Noisy speech	1.615		
Estimated Clean Signal	1.940	18.011	1.682

Comparing with the first model, this model can improve the perceptual speech quality and LSD and Noise Reduction is also improved. Meanwhile, it takes less time to estimate the clean speech because OLA and IFFT are adopted. However, the resulting PESQ improvement is 0.315 and it is unsatisfactory.

The main reason of low PESQ is that the SDAE over-attenuate the noisy spectrum and as a result some speech information is lost.

3.5 Final Model

With consideration of the problem found in spectrum model, the final model proposed in the project take log power spectrum as input to two SDAEs. One of SDAEs outputs estimated clean power spectrum and the another SDAE outputs estimated noise STFT spectrum. The estimated clean and noise STFT spectrum are used to estimate *a prior* SNR by PCRA whose detail refers to section 2.3. In the project, the *a prior* SNR is used to calculate Wiener Filter which multiply the noisy power spectrum to get the restored clean power spectrum. Finally, the phase of noisy speech is used for estimate the clean signal with the restored clean power spectrum and OLA. Figure 20 shows the detail of the model.

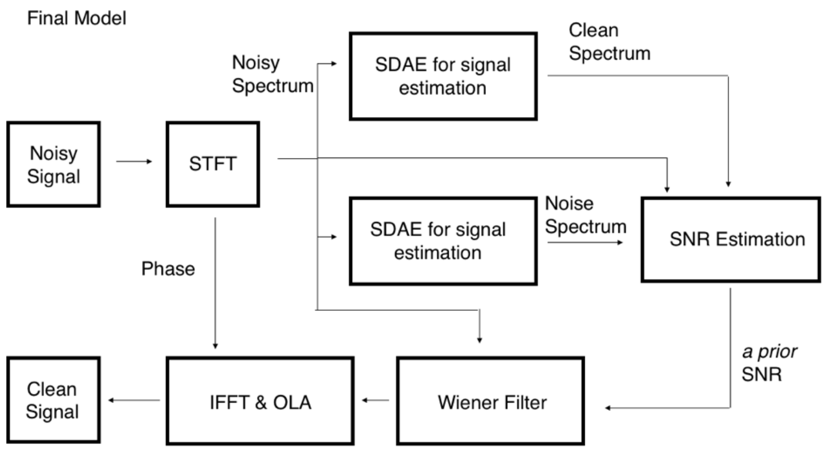


Figure 20: Final Model for Speech Enhancement

For each part of the model, a detail reason and comparison with other choices will be given in the sub-sections.

3.5.1 Type of Input

For the project, three types of input are possible and they are the power spectrum, logfbank and MFCC. Their respective background theory can refer to section 2.1 and 2.2.

With the same reason mentioned in section 3.4.1, the power spectrum which is invertible with OLA is selected as the input. The power spectrum is calculated with 882 frames size(20ms) and 441(10ms) overlapping.

$$spectrogram = |STFT(\lambda, i)|^2 = X^2(\lambda, i)$$

3.5.2 SDAE

In this section, the detail of SDAE and the reason of choosing certain settings as well as their comparison will be given. The following figures shows the overall structure of SDAE. However, Keras, the machine learning library provider, does not support Leaky ReLU as the activation in the first layer and thus it is replaced by linear function. It is expected the performance will be similar.

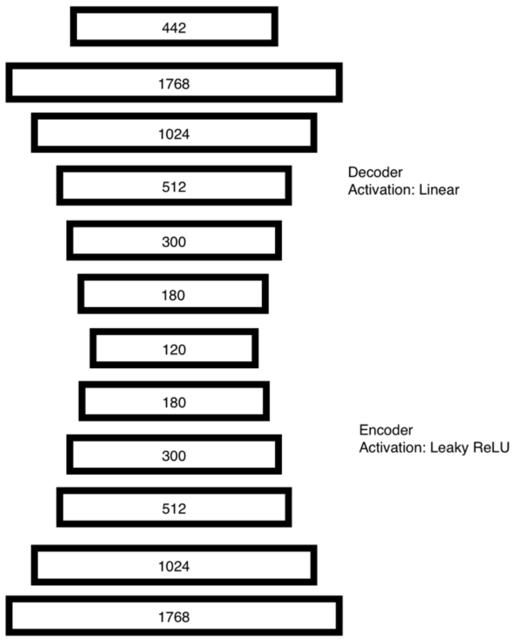


Figure 21: SDAE for estimating clean spectrum

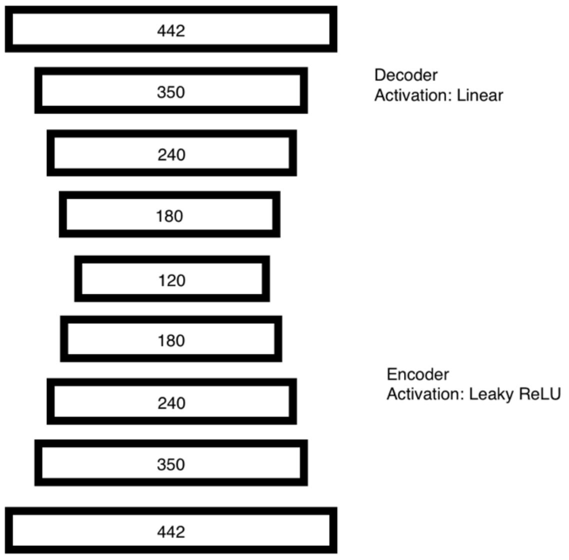


Figure 22: SDAE for estimating noise spectrum

3.5.2.1 SDAE for estimating noise spectrum

In this project, a prior SNR require an estimation of noise spectrum.

$$y[n] = x[n] + n[n]$$

$$Y^2[\lambda, \delta] = X^2[\lambda, \delta] + N^2[\lambda, \delta]$$

where Y^2, X^2 *and* N^2 are noisy, clean and noise spectrum. In the model, two SDAEs are proposed and one of which is for estimating the noise spectrum.

Comparing with another SDAE estimating the clean spectrum, this SDAE is less important in enhancement of speech quality. Therefore, the settings, such as activation, objective function and optimizer, of this SDAE are the same as another SDAE for simplicity except the numbers of hidden layer and neuron. For this SDAE, only the current frame will be used as input to network and therefore the input layer size will be 442. The following hidden layer is adjusted to match the input layer. The structure of two SDAEs are shown by Figure 21 and 22.

In addition, comparing with MCRA which is used to estimate the noise spectrum, SDAE perform better in term of mean square error.

$$MSE = \frac{1}{N * d} ||n - \hat{n}||^2$$

where N is the number of frames and d is (FFT size /2+1)

snr=5	MSE Loss
MCRA	1.831
SDAE	0.2998

3.5.2.2 Input to SDAE

With the same reason mentioned in section 3.4.2.1, the input to the two SDAEs are the log noisy spectrum frames and the desired output is either log noise spectrum frames or log clean spectrum frames. Such that

$$Input = log_{10}(Y^2[\lambda, i]) = y$$

$$Desired\ Output = log_{10}(X^2[\lambda, i]) = x\ or\ log_{10}(N^2[\lambda, i]) = n$$

Taking logarithm on input can minimize the scale of difference between Input and Desired Output and solve the problem of unstable training.

3.5.2.3 Feature Expansion

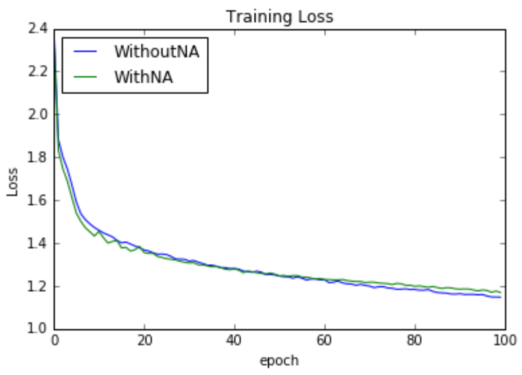
With the same reason mentioned in section 3.4.2.2, the SDAE estimating the clean spectrum, a feature expansion on input is adopted while the desired output remains the same.

$$Input = [y[\lambda - 1], y[\lambda], y[\lambda + 1], n[\lambda]]$$

$$Desired\ Output = x[\lambda]$$

In addition, the noise spectrum is estimated with the SDAE for estimating noise mentioned in section 3.5.2.1 and MCRA is not used.

Here is the training and validation loss with and without noise aware.



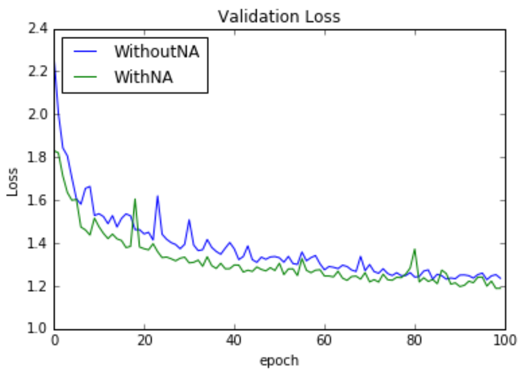


Figure 23: Training Loss of noise aware

Figure 24: Testing Loss of noise aware

From Figure 24, the noise aware seems to prevent overfitting and improve the testing loss of SDAE. Although the effect is not significant, this technic is adopted because the noise spectrum is already available from the output of the SDAE for estimating noise spectrum and the cost of using noise aware is low.

3.5.2.4 Pre-Training

For the model, greedy layer-wise pre-training is adopted as shown in figure 25. The detail of pre-training can refer to section 2.7.7.

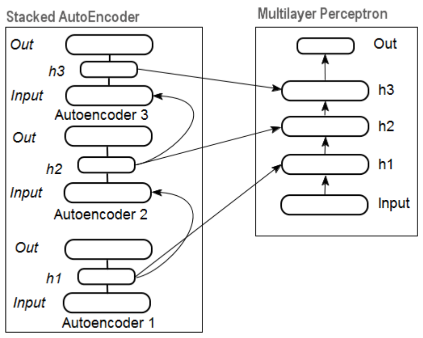


Figure 25: greedy layer-wise pre-training

3.5.2.5 Objective

In the model, two objective functions are proposed and they are MSE and WSE whose detail refer to section 2.7.1. Here is the result

	MSE	WSE
PESQI	0.939	0.742
LSD Improvement(dB)	17.22	16.60
Noise Reduction	0.6896	0.6943

Although it was expected that WSE can perform better MSE, the result shows that MSE finally get better performance. This result is unexpected and further investigation on it should be done.

3.5.2.6 Activation

Three different activations are proposed for SDAE and they are ReLU, LeakyReLU and Sigmoid. Their detail refers to section 2.7.4. For LeakyReLU, the coefficient = 0.01

$$\sigma(a_i) = \begin{cases} a_i \; for \; a_i > 0 \\ 0.01a_i \; otherwise \end{cases}$$

The following is the training loss and testing loss of each activation. However, Keras, the machine learning library provider, does not support Leaky ReLU as the activation in the first layer and thus it is replaced by linear function. It is expected the performance will be similar. For other activation, the first layer is ReLU or sigmoid.

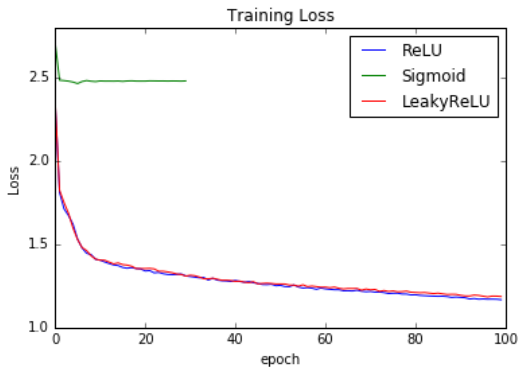


Figure 26: Training Loss of different activations

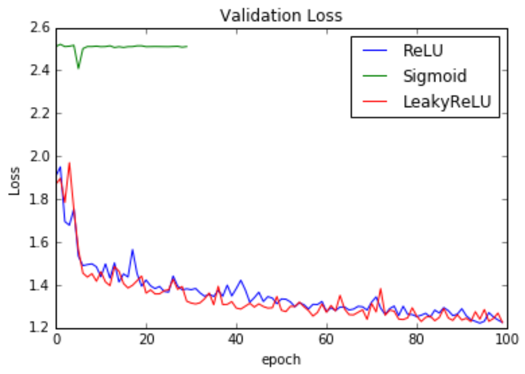


Figure 27: Testing Loss of different activations

Since sigmoid function perform seems to perform much worse than ReLU and LeakyReLU, the training is stop at 30 epochs. From figure 26 and 27, ReLU and LeakyReLU have similar performance. Since LeakyReLU is an attempt to solve the Dying ReLU problem, it is reason to assume that the model does not have this problem and therefore they have similar performance.

In the project, LeakyReLU is adopted as activation in encoding part.

In decoding part, Linear function is always adopted because it greatly reduces the loss.

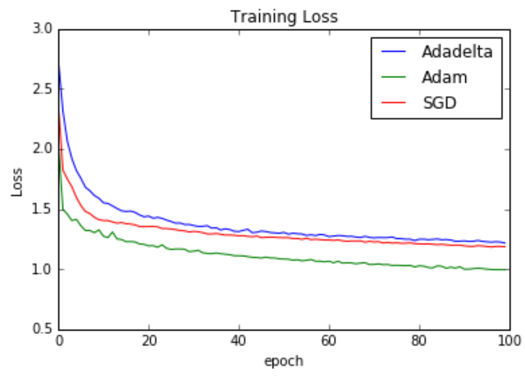
3.5.2.7 Regularization

L2 regularization and weight constraint is adopted for SDAE and the background detail refer to section 2.7.5. The L2 regulation parameter $\lambda = 0.02$ and the weight constraint $w_c = 0.9$.

3.5.2.8 Optimizer

In this model, three optimizers, SGD optimizer, Adadelta optimizer, Adam optimizer, are proposed.

Adadelta optimizer and Adam optimizer are special version of SGD optimzer and the difference is that Adadelta adopts adaptive learning rate while Adam adopts adaptive momentum values. In order to examine their ability in training the neural network, they are compared and the result is shown in the following figures. For SGD, the learning rate is 0.0001 and momentum coefficient is 0.9. For Adam and Adadelta, the default setting suggested by Keras, the machine learning library, is adopted.



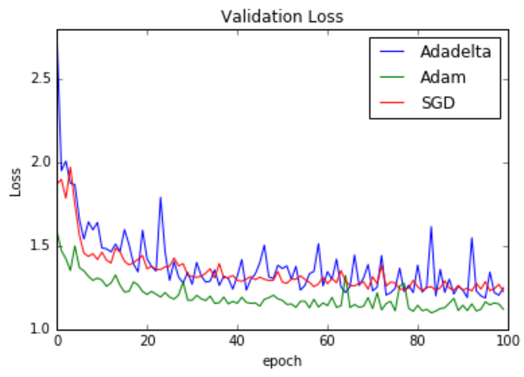


Figure 28: Training Loss of different optimizer

Figure 29: Testing Loss of different optimizer

The figure 28 and 29 suggest that Adam perform faster than SGD and therefore it is adopted as optimizer in this model.

3.5.3 SNR Estimation

In section 2.3, a detail operation of *a prior* SNR estimated by a Posteriori SNR Controlled Recursive Averaging (PCRA) is explained and only the equation is shown here.

Step 1: $\gamma(\lambda,\delta) = \max\left(\frac{Y^2(\lambda,\delta)}{\hat{\sigma}_d^2(\lambda,\delta)}, 1\right)$

Step 2: $\bar{\gamma}(\lambda,\delta) = \alpha_\gamma \bar{\gamma}(\lambda-1,\delta) + (1-\alpha_\gamma)\gamma(\lambda,\delta)$

$$I(\lambda,\delta) = 0 \; if \; \bar{\gamma}(\lambda,\delta) < T_\gamma$$

$$I(\lambda,\delta) = 1 \; if \; \bar{\gamma}(\lambda,\delta) \geq T_\gamma$$

$$p(\lambda,\delta) = \alpha_p p(\lambda-1,\delta) + (1-\alpha_p) I(\lambda,\delta)$$

Step 3: $\alpha_\xi(\lambda,\delta) = \alpha_{\xi min} + (1 - p(\lambda,\delta))(\alpha_{\xi max} - \alpha_{\xi min})$

Step 4: $\bar{\xi}(\lambda,\delta) = \alpha_\xi(\lambda,\delta)\bar{\xi}(\lambda-1,\delta) + (1-\alpha_\xi(\lambda,\delta))[\beta\frac{\hat{X}^2(\lambda,b)}{\hat{\sigma}_d^2(\lambda,b)} + (1-\beta)(\gamma(\lambda,\delta)-1)]$

where α_γ, T_γ, α_p, $\alpha_{\xi min}$, $\alpha_{\xi max}$ and β are the parameter to adjust.

The step 4 equation has three different parts

Part 1: $\bar{\xi}(\lambda-1,\delta)$ which is *a prior* SNR of previous frame. In the final model, it is assumed that the large distortion of some frames of estimated clean spectrum from SDAE partly comes from the missing information of current frame. However, the previous frame may have less distortion and the missing information of current frame. Therefore, depending of the probability function $p(\lambda,\delta)$ and $\alpha_\xi(\lambda,\delta)$ which indicate if the current is highly distorted, the information of previous frame $\bar{\xi}(\lambda-1,\delta)$ will be used.

Part 2: $\frac{\hat{X}^2(\lambda,\delta)}{\hat{\sigma}_d^2(\lambda,\delta)}$ is the estimated SNR of current frame

Part 3: $\gamma(\lambda,\delta)-1=\frac{max(0,Y^2(\lambda,\delta)-\hat{\sigma}_d^2(\lambda,\delta))}{\hat{\sigma}_d^2(\lambda,\delta)}$ is the Maximum Likelihood(ML) estimate of current frame. The idea of ML is based on the additive nature of noise.

$$\frac{max(0,Y^2-\hat{\sigma}_d^2)}{\hat{\sigma}_d^2}=\frac{X^2}{\hat{\sigma}_d^2}$$

if $\hat{\sigma}_d^2$ is accurate.

However, when it is not accurate, it will keep the noise. For example, when

$$X^2(\lambda,\delta)=0,\qquad Y^2(\lambda,\delta)=50000=\sigma_d^2(\lambda,\delta),\ \hat{\sigma}_d^2(\lambda,\delta)=30000$$

$$Y^2(\lambda,\delta)-\hat{\sigma}_d^2(\lambda,\delta)=20000$$

$$\gamma(\lambda,\delta)-1=SNR=\frac{2}{3}$$

$$\hat{X}^2(\lambda,\delta)=\ Y^2(\lambda,\delta)*\frac{SNR}{1+SNR}=20000$$

Therefore, in the part $\beta\frac{\hat{X}^2(\lambda,\delta)}{\hat{\sigma}_d^2(\lambda,\delta)}+(1-\beta)(\gamma(\lambda,\delta)-1)$, β indicate a tradeoff of distortion of speech and noise removal. The large β means larger noise while the information from ML estimate will also be smaller and it distorts the speech.

In order to examine the importance of PCRA and ML, three different settings are compared.

Settings 1: Reference to [5], a relatively high weighting of previous frame and ML is adopted.
$\alpha_\gamma=0.8$, $T_\gamma=1.5$, $\alpha_p=0.95$, $\alpha_{\xi min}=0.9$, $\alpha_{\xi max}=0.98$ and $\beta=0.75$

Setting 2: Since ML estimate will increase the noise level, it is not adopted. In addition, a relatively small weighting on previous frame is adopted
$\alpha_\gamma=0.8$, $T_\gamma=1.5$, $\alpha_p=0.95$, $\alpha_{\xi min}=0.3$, $\alpha_{\xi max}=0.15$ and $\beta=1$

Setting 3: Simple SNR estimation is adopted.

$$SNR=\frac{\hat{X}^2(\lambda,\delta)}{\hat{\sigma}_d^2(\lambda,\delta)}$$

Here is their performance.

SNR=5	PESQI	LSDI	Noise Reduction
Setting 1	0.168	3.87	0.50
Setting 2	0.849	16.32	0.68
Setting 3	0.775	16.41	0.808

The performance shows that the setting from [5] is not satisfactory and therefore ML should not be considered in the equation of step 4. In addition, setting 2 and setting 3 have different advantages and their application should depend on the real situation. In this project, setting 2 is adopted.

3.5.4 Wiener Filter

The main difference of second model and final model is the Wiener Filter. SDAE for estimating noise spectrum and *a prior* SNR estimation are adopted because they can be used for calculating the Wiener Filter. For evaluating performance of Wiener Filter, a comparison is done.

Setting 1: The estimated clean STFT spectrum is directly transformed back to signal with OLA and IFFT

Setting 2: The estimated clean STFT spectrum is multiplied by Wiener Filter with is composed with *a prior* SNR estimation setting 2 in section 3.5.3.

snr=5	PESQI	LSDI	Noise Reduction
Setting 1	0.352	17.02	1.89
Setting 2	0.849	16.32	0.68

The better performance on LSDI and Noise Reduction of setting 1 is expected because SDAE already minimize the different of estimated clean STFT spectrum and desired clean STFT spectrum which is curial to the two performance evaluation. However, the smaller difference does not necessary mean better perceptual speech quality as discussed in section 2.7.1 Objective/Loss which is also the main idea of WSE. For example, the human ear is more sensitive to frequency from 20Hz to 8000Hz and therefore the same improvement in this range will lead to better perceptual speech quality than the improvement in other range. Therefore, the use of Wiener Filter may distort the spectrum in high frequency which lead to worse performance in LSDI and Noise Reduction and improve the spectrum in low frequency which lead to better performance in PESQI.

Although setting 1 has its own advantage, the use of wiener filter is adopted in final model.

4. Performance Evaluation

In this part, performance of final model will be evaluated with different SNR of noise.

4.1 Experiment Setup

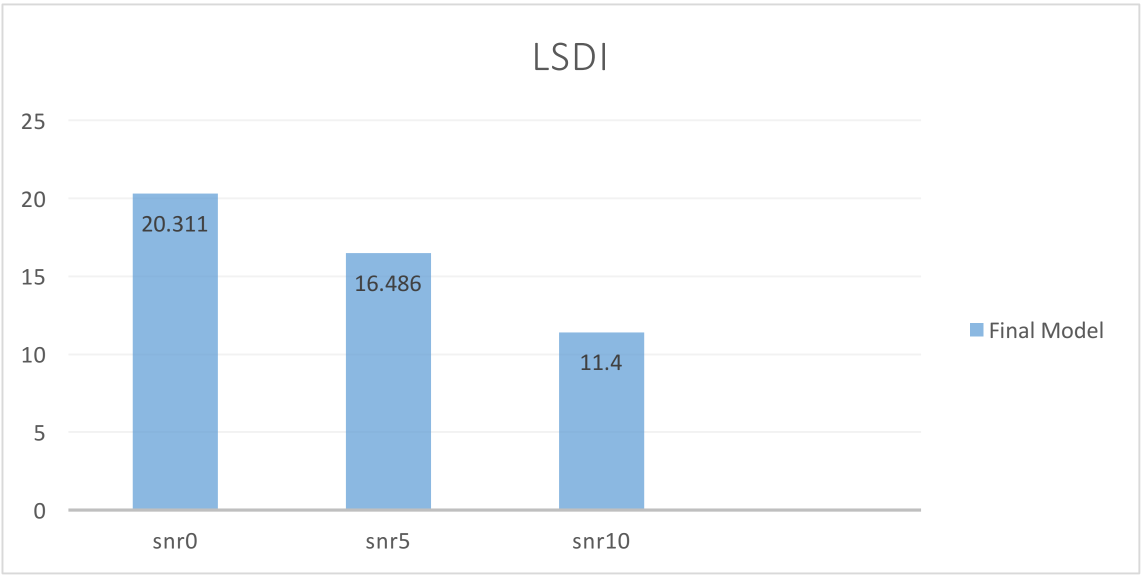
20 audio files are randomly selected from the data base excluding the audio file used for training and testing. The noise file is car and road noise last for 33 seconds and a portion of 9 to 10 seconds of the file is randomly selected. Then, the final model with setting 2 in section 3.5.3 is used to estimate the clean speech. All speech are down sampled to 16000Hz for PESQI.

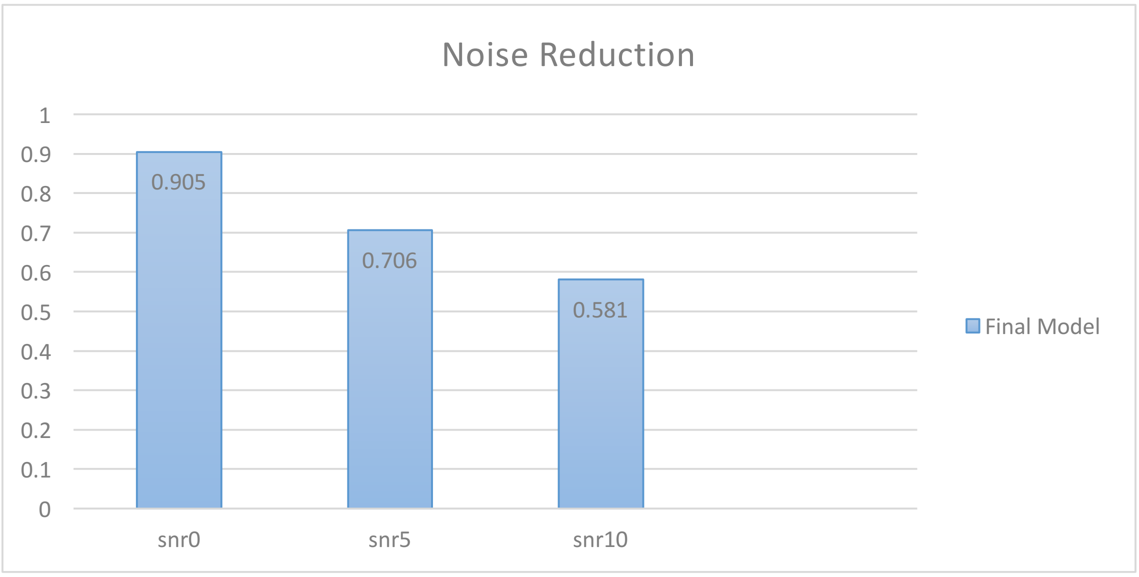
The final evaluation score is the average of each evaluation. In addition, the SDAEs are trained with 5dB noise, therefore, the performance of model with different noise ratio is expected worse.

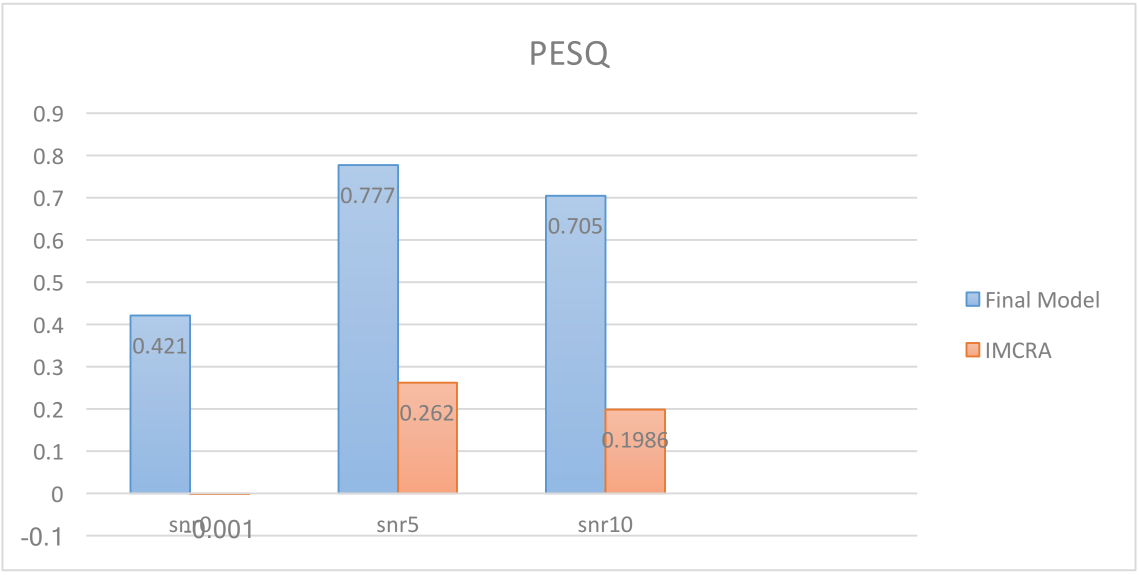
Moreover, the performance of traditional spectral subtraction algorithm with imcra noise estimation algorithms on PESQ is also evaluated in order to make a comparison with the final model.

4.2 Result

The detail of three evaluations can refer to section 2.8.







In addition to table, a speech is randomly selected from the 20 validation speech above and a graph of original speech, noisy speech and estimated speech are shown in the following.

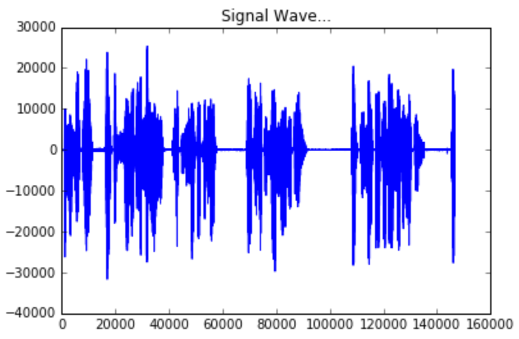


Figure 30: Original Speech

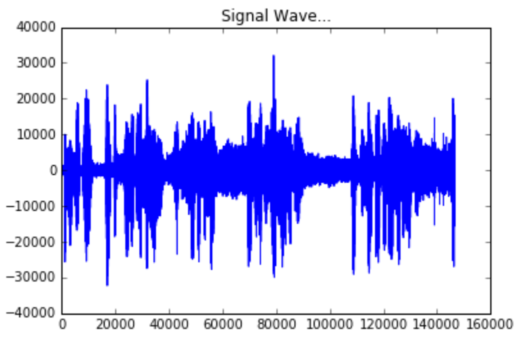


Figure 31: Noisy Speech

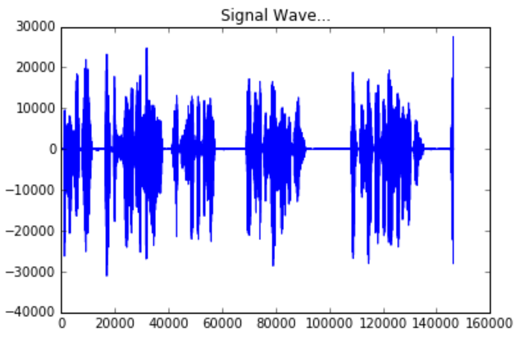


Figure 32: Estimated Speech

4.3 Discussion

4.3.1 PESQI

In the final performance evaluation, 20 audio files with different noise are used. It is important to note that the average amplitude of sound intensity of the noise is fixed to some value while the sound intensity can be fluctuating along the time. Therefore, some part of the noisy speech may have smaller SNR while some part may have larger SNR. As a result, some of them are less distorted by the noise when the portion of noise tend to be smooth which means the sound intensity keep similar along the time. When the portion of noise tend to be fluctuating and the snr may be sometimes 4-5 smaller than the fix snr.

In case of less distorted speech, the PESQI may be smaller since the PESQ of estimated clean speech has an upper limit in the final model because of the adoption of OLA and noisy phase which can refer to section 3.4.3.

Therefore, the PESQI of snr10 is worse than the PESQI of snr5 while the PESQI has a large improvement between snr0 and snr5.

In addition, the final PESQI evaluation of snr5 is worse than the comparison before because one audio file with a fixed portion of noise is used to evaluate the performance in section 3. Since the noisy speech selected in section 3 may have larger distortion, the PESQI is also larger. In this performance evaluation, some of the noisy speech may have less distortion and the estimated clean speech may reach their upper limit therefore PESQI is smaller. Since some of the PESQI is smaller, the average PESQI will also be reduced.

4.3.2 LSDI and Noise Reduction

It is expected that the two performance evaluation are reduced with higher snr because it is thought that if the noisy speech has large distortion and more noise, any improvement will have good performance. When the speech is only lightly distorted, which means small LSD, and the noise is small, it is difficult to have large LSD improvement and Noise Reduction.

4.3.3 Comment on Performance and Limitation

In section 3.5.2.8, the result suggests that Adam which adopt adaptive momentum learning perform better than traditional SGD and Adadelta. The reason may be the momentum is important to train the network and the best momentum may be varied with different stage of training. As a result, an adaptive momentum approach generates the best performance. In addition, the result of section 3.5.3 suggests that the ML of SNR may not be preferable in speech enhancement task although it has some useful information. Moreover, the result in section 3.5.2.5 suggest that MSE perform better than WSE although it was thought that WSE could have higher PESQI score than MSE.

The performance of model is satisfactory for removing the specific noise which is used in training the SDAEs. From figure 30, 31 and 32, the estimated speech is closed to the original speech. In addition, comparing with traditional noise removal method, the proposed model performs better in the three performance evaluation.

However, the model has some limitations because of SDAEs and noisy phase.

The performance of SDAE depends on the training data. If it is trained by one type of noisy speech, it will perform much worse when the noise is another type. For example, when the noise is people group talking noise, the result is the following.

SNR=5	PESQI	LSDI	Noise Reduction
people group talking noise	0.245	7.772	0.699

Comparing with the performance with car and road noisy speech as input, it is much worse.

In addition, the OLA method requires phase information and the noisy phase is used in this project for simplicity of model. However, it limits the performance of model and the best PESQ of estimated clean speech is around 3.2. In the performance evaluation of snr10, some of the estimated clean speech can get 3.1 to 3.4 PESQ score which may be very closed to their limit. Therefore, if better estimation of phase is adopted, the performance of model may be further improved.

4.4 Future Research Recommendation

The recommendations based on the problem of model are building different SDAEs for different noise type and estimating phase and they are not adopted in the project because of insufficient remaining time. However, they may have large potential in improving the model's performance.

4.4.1 Noise Identification

It is possible to make a model to identify the type of noise. Reference to [5], a Gaussian

Mixture Model(GMM) based noise classification can achieve more than 95% accuracy with 14 different type of noise. Since DNN is good at modeling the statistical relationship, it may be a good approach to identify the type of noise.

After identifying the type of noise, the noisy speech can get into the model and SDAE specific for the type of noise and the output should be a good estimation of clean speech. Even the noise type is untrained before, the noise classification can estimate which type of noise is most similar to the unseen noise and send it to the most suitable model. Therefore, the performance will not degrade largely.

4.4.2 Estimating Phase

Reference to section 3.4.3, the main reasons of adopting noisy phase are the computational difficulty of Real Time Signal Estimation and the large overlapping requirement. However, it may be possible to build a SDAE or other type of DNN for estimating clean phase. Comparing with Real Time Signal Estimation, DNN is relatively computational simple and fast while it does not have the requirement of large overlapping. Therefore, it may have the potential to improve the model performance by estimating the clean phase.

5. Conclusion

Our brain is incredible strong in handling background noise and speech separation which enable us to talk to each other in a noisy environment. However, the machine could not do it well before the full application of neural network. For investigating the power of neural network in term of noise reduction, my research tries to build a stacked denoising autoencoder which help to removal background noise of noisy observation for improving the speech quality and recognition accuracy of ASR system. This is complementary to the exiting speech recognition system in mobile device which operate in a relatively complex environment where background noise, such as car noise or street noise, is common and greatly affect the performance of ASR system. In addition to improving speech recognition accuracy, key contribution may include background noise removal for music or conversation.

With reference to the performance of model shown on section 4, the model improves the perceptual speech quality of 5dB snr by 0.7 PESQ score while LSDI is 16.486db and Noise Reduction is 0.706. However, the performance is degraded largely with untrained noise. While the recommendations in section 4.4 are possible methods further improve the performance, it makes the model bulky. For ASR system, simple and fast but robust model for noise removal is desirable because the main focus of the system should be the speech recognition. The model and recommendation proposed in the project may not be suitable for mobile application. However, when efficiency is not required, the model and recommendation propose in the project may be suitable. More research is needed for investigation the better model which is simple but robust.

6. Reference

[1] "Deep Learning Tutorials — DeepLearning 0.1 documentation", Deeplearning.net, 2017. [Online]. Available: http://deeplearning.net/tutorial/. [Accessed: 07- Apr- 2017].

[2] "Python tutorial", www.tutorialspoint.com, 2017. [Online]. Available: http://www.tutorialspoint.com/python/. [Accessed: 07- Apr- 2017].

[3] A. Kumar and D. Florencio, "Speech Enhancement In Multiple-Noise Conditions using Deep Neural Networks", 2016.

[4] B. Sh. Mahmood and N. N. Ibrahim, "Processing Mel Speech Power Spectrum for Speech Restoration", International Journal of Enhanced Research in Science, Technology & Engineering, vol. 5, no. 9, pp. 39-45, 2016.

[5] B. Xia and C. Bao, "Wiener filtering based speech enhancement with Weighted Denoising Auto-encoder and noise classification", Speech Communication, vol. 60, pp. 13-29, 2014.

[6] D. Griffin and Jae Lim, "Signal estimation from modified short-time Fourier transform", IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 32, no. 2, pp. 236-243, 1984.

[7] E. Boucheron and L. De Leon, "On the Inversion of Mel-Frequency Cepstral Coefficients for Speech Enhancement Applications", 2017.

[8] E. Plourde and B. Champagne, "Multidimensional STSA Estimators for Speech Enhancement With Correlated Spectral Components", IEEE Transactions on Signal Processing, vol. 59, no. 7, pp. 3013-3024, 2011.

[9] E. Plourde and B. Champagne, "Multidimensional STSA Estimators for Speech Enhancement With Correlated Spectral Components", IEEE Transactions on Signal Processing, vol. 59, no. 7, pp. 3013-3024, 2011.

[10] F. Chollet, "Building Autoencoders in Keras", Blog.keras.io, 2017. [Online]. Available: https://blog.keras.io/building-autoencoders-in-keras.html. [Accessed: 07- Apr- 2017].

[11] M. Nielsen, "Neural Networks and Deep Learning", Neuralnetworksanddeeplearning.com, 2017. [Online]. Available: http://neuralnetworksanddeeplearning.com/. [Accessed: 07- Apr- 2017].

[12] M. Zhao, D. Wang, Z. Zhang and X. Zhang, "Music Removal by Convolutional Denoising Autoencoder in Speech Recognition", 2015.

[13] P. Loizou, Speech enhancement, 1st ed. Boca Raton, Fla.: CRC Press, 2013.

[14] T. Gerkmann and M. Krawczyk, "MMSE-Optimal Spectral Amplitude Estimation Given

the STFT-Phase", IEEE Signal Processing Letters, vol. 20, no. 2, pp. 129-132, 2013.

[15] T. Ishii, H. Komiyama, T. Shinozaki, Y. Horiuchi and S. Kuroiwa, "Reverberant Speech Recognition Based on Denoising Autoencoder", INTERSPEECH, 2013.

[16] T. Sainburg, "Spectrograms, MFCCs, and Inversion in Python", Tim Sainburg, 2017. [Online]. Available: https://timsainb.github.io/spectrograms-mfccs-and-inversion-in-python.html. [Accessed: 07- Apr- 2017].

[17] Thomas F. Quatieri, "Discrete-Time Speech Signal Processing: Principles and Practice"

[18] X. Lu, Y. Tsao, S. Matsuda and C. Hori, "Speech Enhancement Based on Deep Denoising Autoencoder".

[19] X. Zhu, G. Beauregard and L. Wyse, "Real-Time Signal Estimation From Modified Short-Time Fourier Transform Magnitude Spectra", IEEE Transactions on Audio, Speech and Language Processing, vol. 15, no. 5, pp. 1645-1653, 2007.

[20] Z. Wu, S. Takaki and J. Yamagishi, "DEEP DENOISING AUTO-ENCODER FOR STATISTICAL SPEECH SYNTHESIS", 2015.

1. Appendix

7.1 Autoencoder_log_PS.py

Program 1: This program is used to build the SDAE estimating clean spectrum for final model

```python
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 28 16:38:06 2017

@author: u3510120
"""

import data_logPS
from keras.layers import Dense
from keras.models import Sequential
import numpy as np
from keras.optimizers import SGD
from keras import backend as K
from keras.models import model_from_json
import os
from keras.regularizers import l2
from keras.constraints import maxnorm
from keras.layers.advanced_activations import LeakyReLU
import math

path=os.getcwd()
fft_size = 882
step_size = 441
rate=44100
keeptrain = bool(int(raw_input("Keep Train?: ")))
epoch=int(raw_input("Please enter epoch: "))
p = int(raw_input("Please enter the proportion of training: "))
learn = float(raw_input("Please enter learn: "))
mon=float(raw_input("Please enter mon: "))
batch = 512
loss_function = "mse"

leaky_coefficient=0.01



#Loading Data

if keeptrain==True:
    json_file = open('Autoencoder_log_PS.json', 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    autoencoder = model_from_json(loaded_model_json)
    # load weights into new model
    autoencoder.load_weights("Autoencoder_log_PS.h5")
    print("Loaded model from disk")
else:
    w_coefficient = float(raw_input("Please enter w_coefficient: "))
    w_contraint = float(raw_input("Please enter w_contraint: "))
    print('Fine-tuning:')

    act=LeakyReLU(leaky_coefficient)
    autoencoder = Sequential()
    autoencoder.add(Dense(1024, input_shape=(1768,),W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(512, W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(300, W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(180, W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(120, W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(180,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(300,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(512,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(1024,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(1768,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(442,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))




op = SGD(lr=learn,momentum=mon)

def ath(nfft,rate):
    feq_base=float(rate/nfft)
    feq=[]
    for x in range(nfft/2+1):
        feq.append(feq_base*x)
    feq=np.asarray(feq)
    threshold=[]
    for x in range(nfft/2+1):
        if x ==0:
            num=3.64*math.pow(feq[3]/1000,-0.8)-6.5*math.exp(-0.6*math.pow(feq[3]/1000-3.3,2))+0.001*math.pow(feq[3]/1000,4)
            threshold.append(num)
        else:
            num=3.64*math.pow(feq[x]/1000,-0.8)-6.5*math.exp(-0.6*math.pow(feq[x]/1000-3.3,2))+0.001*math.pow(feq[x]/1000,4)
            threshold.append(num)
    threshold=np.asarray(threshold)
    threshold=threshold+np.abs(np.min(threshold))+1
    weight=1/threshold
    return weight

weight=ath(fft_size,rate)

def custom_objective(y_true, y_pred):
    cce=K.sum(K.square(K.abs(weight*(y_pred - y_true))),axis=-1)
    return cce

def dist(y_true, y_pred):
    return 10*np.log10(K.mean(np.abs(y_true - y_pred)))

def reduct(x_true,y_pred):
    return 10*np.log10(K.mean(np.abs(x_true - y_pred)))

autoencoder.compile(optimizer=op, loss='mse',metrics=[dist,reduct])
for x in range(epoch/50):
    x_train,x_train_noise,x_test,x_test_noise,x_validation_noise,x_validation_clean,x_train_noise_ps,x_test_noise_ps= data_logPS.loaddata()
    n_train = np.random.randint(x_train.shape[0]*0.65)
    n_test = np.random.randint(x_test.shape[0]*0.65)
    x_train=np.concatenate(x_train[n_train:n_train+x_train.shape[0]/3]).astype(None)
    x_train_noise=np.concatenate(x_train_noise[n_train:n_train+x_train_noise.shape[0]/3]).astype(None)
    x_test=np.concatenate(x_test[n_test:n_test+x_test.shape[0]/3]).astype(None)
    x_test_noise=np.concatenate(x_test_noise[n_test:n_test+x_test_noise.shape[0]/3]).astype(None)
    x_train_noise_ps=np.concatenate(x_train_noise_ps[n_train:n_train+x_train_noise_ps.shape[0]/3]).astype(None)
    x_test_noise_ps=np.concatenate(x_test_noise_ps[n_test:n_test+x_test_noise_ps.shape[0]/3]).astype(None)


    autoencoder.fit(np.concatenate((np.concatenate((np.zeros([1,442]),x_train_noise[0:-1,:]),axis=0),x_train_noise[:x_train_noise.shape[0]/p],np.concatenate((x_train_noise[1:,:],np.zeros([1,442])),axis=0),x_train_noise_ps),axis=1),
                    x_train[:x_train.shape[0]/p],
                    nb_epoch=50,
                    shuffle=True,
                    batch_size=batch,
                    verbose=1,
                    validation_data=(np.concatenate((np.concatenate((np.zeros([1,442]),x_test_noise[0:-1,:]),axis=0),x_test_noise[:x_test_noise.shape[0]/p],np.concatenate((x_test_noise[1:,:],np.zeros([1,442
                                                     ])),axis=0),x_test_noise_ps),axis=1)
                                      , x_test[:x_test.shape[0]/p])
                    )
        # serialize model to JSON
    model_json = autoencoder.to_json()
    with open("Autoencoder_log_PS.json", "w") as json_file:
```

```python
        json_file.write(model_json)
    # serialize weights to HDF5
    autoencoder.save_weights("Autoencoder_log_PS.h5",overwrite=True)
    print("Saved model to disk")

txt =",fft size: " + str(fft_size) + ", step_size: " + str(step_size) + "\n" + "batch_size: " + str(batch) +   ", learning rate: " + str(learn) + ", Momentum: " + str(mon) + ", Loss_function: "+loss_function+", leaky_coefficient:"+str(leaky_coefficient)+",w_contraint:"+str(w_contraint)+",w_coefficient:"+str(w_coefficient)+"\n"
model_config = str(autoencoder.get_config()) + "\n"
result = "result: " + str(autoencoder.evaluate(np.concatenate((np.concatenate((x_test_noise[-1:,:],x_test_noise[0:-1,:]),axis=0),x_test_noise[:x_test_noise.shape[0]/p],np.concatenate((x_test_noise[1:,:],x_test_noise[0,:].reshape(1,x_test_noise.shape[1])),axis=0),x_test_noise),axis=1), x_test[:x_test.shape[0]/p],batch_size=128))+"\n"
text =txt+model_config+result
with open("Autoencoder_log_PS.txt", "a") as text_file:
    text_file.write(text)
```

7.2 Autoencoder_logPS_estimate_noise.py

This program is used to build the SDAE estimating noise spectrum for final model

```python
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Wed March 25 16:38:06 2017

@author: u3510120
"""

import data_logPS_noisy
from keras.layers import Dense
from keras.models import Sequential
import numpy as np
from keras.optimizers import SGD
from keras import backend as K
from keras.models import model_from_json
import os
from keras.regularizers import l2
from keras.constraints import maxnorm
from keras.layers.advanced_activations import LeakyReLU

path=os.getcwd()
fft_size = 882
step_size = 441
rate=44100
keeptrain = bool(int(raw_input("Keep Train?: ")))
epoch=int(raw_input("Please enter epoch: "))
learn = float(raw_input("Please enter learn: "))
mon=float(raw_input("Please enter mon: "))
batch = 512
loss_function = "mse"

leaky_coefficient=0.3



#Loading Data

if keeptrain==True:
    json_file = open('Autoencoder_logPS_estimate_noise.json', 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    autoencoder = model_from_json(loaded_model_json)
    # load weights into new model
    autoencoder.load_weights("Autoencoder_logPS_estimate_noise.h5")
    print("Loaded model from disk")
else:
    w_coefficient = float(raw_input("Please enter w_coefficient: "))
    w_contraint = float(raw_input("Please enter w_contraint: "))
    print('Fine-tuning:')

    act=LeakyReLU(leaky_coefficient)
    autoencoder = Sequential()
    autoencoder.add(Dense(350, input_shape=(442,),W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(240, W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(180, W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(120, W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(80, W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(act)
    autoencoder.add(Dense(120,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(180,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(240,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(350,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))
    autoencoder.add(Dense(442,activation='linear', W_regularizer=l2(w_coefficient),W_constraint = maxnorm(w_contraint)))




op = SGD(lr=learn,momentum=mon)

def dist(y_true, y_pred):
    return 10*np.log10(K.mean(np.abs(y_true - y_pred)))

def reduct(x_true,y_pred):
    return 10*np.log10(K.mean(np.abs(x_true - y_pred)))

autoencoder.compile(optimizer=op, loss='mse',metrics=[dist,reduct])
for x in range(epoch/50):
    x_train,x_train_noise,x_test,x_test_noise,x_validation_noise,x_validation_clean= data_logPS_noisy.loaddata()
    x_train=np.concatenate(x_train).astype(None)
    x_train_noise=np.concatenate(x_train_noise).astype(None)
    x_test=np.concatenate(x_test).astype(None)
    x_test_noise=np.concatenate(x_test_noise).astype(None)



    autoencoder.fit(x_train_noise,
                    x_train,
                    nb_epoch=50,
                    shuffle=True,
                    batch_size=batch,
                    verbose=1,
                    validation_data=(x_test_noise,x_test)
                    )
        # serialize model to JSON
    model_json = autoencoder.to_json()
    with open("Autoencoder_logPS_estimate_noise.json", "w") as json_file:
        json_file.write(model_json)
    # serialize weights to HDF5
    autoencoder.save_weights("Autoencoder_logPS_estimate_noise.h5",overwrite=True)
    print("Saved model to disk")

model_config = str(autoencoder.get_config()) + "\n"
result = "result: " + str(autoencoder.evaluate(x_test_noise, x_test[:x_test.shape[0]],batch_size=128))+"\n"
text =model_config+result
with open("Autoencoder_logPS_estimate_noise.txt", "a") as text_file:
    text_file.write(text)
```

7.3 data_logPS

This program is used to load the audio files, transmit them log power spectrum and save the spectrum for Autoencoder_log_PS.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 25 15:14:03 2017

```

```
@author: u3510120
Apply car noise to audio signal
Apply the new database
SNR=5
"""

from logfbank2 import power
import scipy.io.wavfile as wav
import numpy as np
import os
import glob
from keras.models import model_from_json


def addnoise(a,b,snr=5):
    path = os.getcwd()
    rate=[]
    data=[]
    data2 = []
    fn=[x for x in glob.glob(path+'/trainingdata/*.wav')][a:b]
    for x in fn:
        temprate, tempdata = wav.read(x)
        rate.append(temprate)
        data.append(tempdata)
    data = np.asanyarray(data)
    temprate, noise = wav.read('salamisound-1020082-street-noise-cars-in-both.wav')
    for i in range(data.shape[0]):
        n=np.random.randint(noise.shape[0]-data[i].shape[0])
        tempnoise=noise[n:n+data[i].shape[0],:]
        rms_noise = np.sqrt(np.mean(np.square(tempnoise.astype('int32'))))
        rms_signal = np.sqrt(np.mean(np.square(data[i].astype('int32'))))
        amp = rms_signal/(np.power(10,np.divide(snr,20,dtype='float64')))/rms_noise
        tempnoise = tempnoise*(amp**2)
        data2.append(tempnoise)
    data2 = np.asarray(data2)+ data
    validation_noise = data2[-1]
    validation_clean = data[-1]
    return data,data2,validation_clean,validation_noise,rate[0]

def loadmodel(wav_spectrogram):
    json_file = open('Autoencoder_logPS_estimate_noise.json', 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    autoencoder = model_from_json(loaded_model_json)
    # load weights into new model
    autoencoder.load_weights("Autoencoder_logPS_estimate_noise.h5")
    print("Loaded model from disk")
    wav_spectrogram_ps=[]
    for x in range(wav_spectrogram.shape[0]):
        wav_spectrogram_ps.append(autoencoder.predict(wav_spectrogram[x]))
    wav_spectrogram_ps=np.asarray(wav_spectrogram_ps)
    return wav_spectrogram_ps


def loaddata():
    if not os.path.exists('x_train_logPS.npy') & os.path.exists('x_train_noise_logPS.npy') & os.path.exists('x_test_logPS.npy')
 & os.path.exists('x_test_noise_logPS.npy'):
        readfile(6401,7200)
        readfile(7201,8000)
    x_train=np.load('x_train_logPS.npy')
    x_train_noise=np.load('x_train_noise_logPS.npy')
    x_test=np.load('x_test_logPS.npy')
    x_test_noise=np.load('x_test_noise_logPS.npy')
    x_validation_clean = np.load('x_validation_logPS.npy')
    x_validation_noise = np.load('x_validation_noise_logPS.npy')
    x_train_noise_ps=np.load('x_train_logPS_noise_ps.npy')
    x_test_noise_ps=np.load('x_test_logPS_noise_ps.npy')
    return x_train,x_train_noise,x_test,x_test_noise,x_validation_noise,x_validation_clean,x_train_noise_ps,x_test_noise_ps

def readfile(a,b):
    clean_speech,corrupted_speech,validation_clean,validation_noise,rate=addnoise(a,b)
    clean_coefficient = []
    print('calculating clean speech logfbank')
    for x in range(clean_speech.shape[0]):
```

```python
        wav_spectrogram,phase = power(clean_speech[x], fft_size = 882,
                                    step_size = 441,
                                    spec_thresh = 0,
                                    lowcut = 0,
                                    highcut = 15000,
                                    samplerate = rate,
                                    noise = False,
                                    log = True
                                    )
        clean_coefficient.append(wav_spectrogram)
    clean_coefficient = np.asarray(clean_coefficient)
    validation_clean,phase=power(validation_clean, fft_size = 882,
                                    step_size = 441,
                                    spec_thresh = 0,
                                    lowcut = 0,
                                    highcut = 15000,
                                    samplerate = rate,
                                    noise = False,
                                    log = True
                                    )
    validation_noise,phase=power(validation_noise, fft_size = 882,
                                    step_size = 441,
                                    spec_thresh = 0,
                                    lowcut = 0,
                                    highcut = 15000,
                                    samplerate = rate,
                                    noise = False,
                                    log = True
                                    )

    corrupted_coefficient = []
    'noise_ps=[]'
    print('calculating corrupted speech coefficient')
    for x in range(corrupted_speech.shape[0]):
        wav_spectrogram,phase=power(corrupted_speech[x], fft_size = 882,
                                    step_size = 441,
                                    spec_thresh = 0,
                                    lowcut = 0,
                                    highcut = 15000,
                                    samplerate = rate,
                                    noise = False,
                                    log = True
                                    )
        'noise_ps.append(noise)'
        corrupted_coefficient.append(wav_spectrogram)

    corrupted_coefficient=np.asarray(corrupted_coefficient)
    noise_ps=loadmodel(corrupted_coefficient)
    'noise_ps=np.asarray(noise_ps)'


    print('Saving')
    clean_coefficient_train=clean_coefficient[0:clean_coefficient.shape[0]*8/10]
    corrupted_coefficient_train=corrupted_coefficient[0:corrupted_coefficient.shape[0]*8/10]
    clean_coefficient_test=clean_coefficient[clean_coefficient.shape[0]*8/10:clean_coefficient.shape[0]]
    corrupted_coefficient_test=corrupted_coefficient[corrupted_coefficient.shape[0]*8/10:corrupted_coefficient.shape[0]]
    noise_ps_train=noise_ps[0:noise_ps.shape[0]*8/10]
    noise_ps_test=noise_ps[noise_ps.shape[0]*8/10:noise_ps.shape[0]]

    if os.path.isfile('x_train_logPS.npy'):
        print('Next Round')
        clean_coefficient_train=np.concatenate((np.load('x_train_logPS.npy'),clean_coefficient_train),axis=0)
        corrupted_coefficient_train=np.concatenate((np.load('x_train_noise_logPS.npy'),corrupted_coefficient_train),axis=0)
        clean_coefficient_test=np.concatenate((np.load('x_test_logPS.npy'),clean_coefficient_test),axis=0)
        corrupted_coefficient_test=np.concatenate((np.load('x_test_noise_logPS.npy'),corrupted_coefficient_test),axis=0)
        noise_ps_train = np.concatenate((np.load('x_train_logPS_noise_ps.npy'),noise_ps_train),axis=0)
        noise_ps_test = np.concatenate((np.load('x_test_logPS_noise_ps.npy'),noise_ps_test),axis=0)
    np.save('x_train_logPS', clean_coefficient_train)
    np.save('x_train_noise_logPS', corrupted_coefficient_train)
    np.save('x_test_logPS', clean_coefficient_test)
    np.save('x_test_noise_logPS', corrupted_coefficient_test)
    np.save('x_train_logPS_noise_ps', noise_ps_train)
    np.save('x_test_logPS_noise_ps', noise_ps_test)
    np.save('x_validation_logPS',validation_clean)
    np.save('x_validation_noise_logPS',validation_noise)
```

7.4 data_logPS_noisy

This program is used to load the audio files, transmit them log power spectrum and save the spectrum for Autoencoder_logPS_estimate_noise

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Sat March 25 15:14:03 2017

@author: u3510120
Apply car noise to audio signal
Apply the new database
SNR=5
"""

from logfbank2 import power
import scipy.io.wavfile as wav
import numpy as np
import os
import glob
from keras.models import model_from_json


def addnoise(a,b,snr=5):
    path = os.getcwd()
    rate=[]
    data=[]
    data2 = []
    fn=[x for x in glob.glob(path+'/trainingdata/*.wav')][a:b]
    for x in fn:
        temprate, tempdata = wav.read(x)
        rate.append(temprate)
        data.append(tempdata)
    data = np.asanyarray(data)
    temprate, noise = wav.read('salamisound-1020082-street-noise-cars-in-both.wav')
    for i in range(data.shape[0]):
        n=np.random.randint(noise.shape[0]-data[i].shape[0])
        tempnoise=noise[n:n+data[i].shape[0],:]
        rms_noise = np.sqrt(np.mean(np.square(tempnoise.astype('int32'))))
        rms_signal = np.sqrt(np.mean(np.square(data[i].astype('int32'))))
        amp = rms_signal/(np.power(10,np.divide(snr,20,dtype='float64')))/rms_noise
        tempnoise = tempnoise*(amp**2)
        data2.append(tempnoise)
    data2 = np.asarray(data2)
    data3 = data2+ data
    validation_input = data3[-1]
    validation_output = data2[-1]
    return data2,data3,validation_output,validation_input,rate[0]

def loaddata(phase=False):
    if not os.path.exists('x_train_output_logPS_estimate_noise.npy') & os.path.exists('x_train_input_logPS_estimate_noise.npy') & os.path.exists('x_test_output_logPS_estimate_noise.npy') & os.path.exists('x_test_input_logPS_estimate_noise.npy'):

        readfile(1601,2400)
        readfile(2401,3200)
    x_train_output=np.load('x_train_output_logPS_estimate_noise.npy')
    x_train_input=np.load('x_train_input_logPS_estimate_noise.npy')
    x_test_output=np.load('x_test_output_logPS_estimate_noise.npy')
    x_test_input=np.load('x_test_input_logPS_estimate_noise.npy')
    x_validation_output = np.load('x_validation_output_logPS_estimate_noise.npy')
    x_validation_input = np.load('x_validation_input_logPS_estimate_noise.npy')
    if phase:
        x_validation_output_phase = np.load('x_validation_output_phase_logPS_estimate_noise.npy')
        x_validation_input_phase = np.load('x_validation_input_phase_logPS_estimate_noise.npy')
        return x_train_output,x_train_input,x_test_output,x_test_input,x_validation_input,x_validation_output,x_validation_input_phase,x_validation_output_phase
    else:
        return x_train_output,x_train_input,x_test_output,x_test_input,x_validation_input,x_validation_output
```

```python


def readfile(a,b):
    output_speech,input_speech,validation_output,validation_input,rate=addnoise(a,b)
    output_coefficient = []
    print('calculating output speech logfbank')
    for x in range(output_speech.shape[0]):
        wav_spectrogram,phase = power(output_speech[x], fft_size = 882,
                                    step_size = 441,
                                    spec_thresh = 0,
                                    lowcut = 0,
                                    highcut = 15000,
                                    samplerate = rate,
                                    noise = False,
                                    log = True
                                    )
        output_coefficient.append(wav_spectrogram)
    output_coefficient = np.asarray(output_coefficient)
    validation_output,validation_output_phase=power(validation_output, fft_size = 882,
                                    step_size = 441,
                                    spec_thresh = 0,
                                    lowcut = 0,
                                    highcut = 15000,
                                    samplerate = rate,
                                    noise = False,
                                    log = True
                                    )
    validation_input,validation_input_phase=power(validation_input, fft_size = 882,
                                    step_size = 441,
                                    spec_thresh = 0,
                                    lowcut = 0,
                                    highcut = 15000,
                                    samplerate = rate,
                                    noise = False,
                                    log = True
                                    )

    input_coefficient = []
    print('calculating input speech coefficient')
    for x in range(input_speech.shape[0]):
        wav_spectrogram,phase=power(input_speech[x], fft_size = 882,
                                    step_size = 441,
                                    spec_thresh = 0,
                                    lowcut = 0,
                                    highcut = 15000,
                                    samplerate = rate,
                                    noise = False,
                                    log = True
                                    )
        input_coefficient.append(wav_spectrogram)
    input_coefficient=np.asarray(input_coefficient)

    print('Saving')
    output_coefficient_train=output_coefficient[0:output_coefficient.shape[0]*8/10]
    input_coefficient_train=input_coefficient[0:input_coefficient.shape[0]*8/10]
    output_coefficient_test=output_coefficient[output_coefficient.shape[0]*8/10:output_coefficient.shape[0]]
    input_coefficient_test=input_coefficient[input_coefficient.shape[0]*8/10:input_coefficient.shape[0]]

    if os.path.isfile('x_train_input_logPS_estimate_noise.npy'):
        print('Next Round')
        output_coefficient_train=np.concatenate((np.load('x_train_output_logPS_estimate_noise.npy'),output_coefficient_train),axis=0)
        input_coefficient_train=np.concatenate((np.load('x_train_input_logPS_estimate_noise.npy'),input_coefficient_train),axis=0)
        output_coefficient_test=np.concatenate((np.load('x_test_output_logPS_estimate_noise.npy'),output_coefficient_test),axis=0)
        input_coefficient_test=np.concatenate((np.load('x_test_input_logPS_estimate_noise.npy'),input_coefficient_test),axis=0)

    np.save('x_train_output_logPS_estimate_noise', output_coefficient_train)
    np.save('x_train_input_logPS_estimate_noise', input_coefficient_train)
    np.save('x_test_output_logPS_estimate_noise', output_coefficient_test)
    np.save('x_test_input_logPS_estimate_noise', input_coefficient_test)
    np.save('x_validation_output_logPS_estimate_noise',validation_output)
    np.save('x_validation_input_logPS_estimate_noise',validation_input)
    np.save('x_validation_input_phase_logPS_estimate_noise',validation_input_phase)
    np.save('x_validation_output_phase_logPS_estimate_noise',validation_output_phase)


```

7.5 logfbank2

This program has the necessary function to compute MFCC, logfbank, MCRA, OLA, Real Time signal estimation and power spectrum. Some of the code taken from https://gist.github.com/kastnerkyle/179d6e9a88202ab0a2fe.

```python
# Packages we're using
import numpy as np
import matplotlib.pyplot as plt
import copy
from scipy.io import wavfile
from scipy.signal import butter, lfilter
import scipy.ndimage

### Parameters ###
fft_size = 2048 # window size for the FFT
step_size = fft_size/16 # distance to slide along the window (in time)
spec_thresh = 0 # threshold for spectrograms (lower filters out more noise)
lowcut = 0 # Hz # Low cut for our butter bandpass filter
highcut = 15000 # Hz # High cut for our butter bandpass filter
samplerate = 48000
# For mels
n_mel_freq_components = 40 # number of mel frequency channels
shorten_factor = 1 # how much should we compress the x-axis (time)
start_freq = 20 # Hz # What frequency to start sampling our melS from
end_freq = 8000 # Hz # What frequency to stop sampling our melS from


# Most of the Spectrograms and Inversion are taken from: https://gist.github.com/kastnerkyle/179d6e9a88202ab0a2fe

def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    return b, a

def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = lfilter(b, a, data)
    return y

def overlap(X, window_size, window_step):
    """
    Create an overlapped version of X
    Parameters
    ----------
    X : ndarray, shape=(n_samples,)
        Input signal to window and overlap
    window_size : int
        Size of windows to take
    window_step : int
        Step size between windows
    Returns
    -------
    X_strided : shape=(n_windows, window_size)
        2D array of overlapped X
    """
    if window_size % 2 != 0:
        raise ValueError("Window size must be even!")
    # Make sure there are an even number of windows before stridetricks
    append = np.zeros((window_size - len(X) % window_size))
    X = np.hstack((X, append))

    ws = window_size
    ss = window_step
    a = X

    valid = len(a) - ws
    nw = (valid) // ss
    out = np.ndarray((nw,ws),dtype = a.dtype)

    for i in xrange(nw):
        # "slide" the window along the samples
        start = i * ss
        stop = start + ws
        out[i] = a[start : stop]

    return out

def re_overlap(X, window_size, window_step):
    wave = np.zeros((X.shape[0]*window_step+window_size))
    for i in range(X.shape[0]):
        if i == X.shape[0]:
            wave[i*128:]=X[i][:]
        else:
            wave[i*128:i*128+127]=X[i][0:127]
    return wave


def stft(X, fftsize=fft_size, step=step_size, mean_normalize=False, real=False,
         compute_onesided=True):
    """
    Compute STFT for 1D real valued input X
    """
    if real:
        local_fft = np.fft.rfft
        cut = None
    else:
        local_fft = np.fft.fft
        cut = None
    if compute_onesided:
        cut = fftsize // 2 + 1
    if mean_normalize:
        X -= X.mean()

    X = overlap(X, fftsize, step)

    size = fftsize
    win = scipy.hanning(size+1)[:-1]
    X = X * win[None]
    X = local_fft(X)[:, :cut]
    phase = X/np.abs(X)
    return X,phase

def re_spectrogram(X,phase, fftsize=fft_size, step=step_size, mean_normalize=False, real=False,
         compute_onesided=True):
    X = phase * X
    if compute_onesided:
        temp = np.conjugate(X[:,-2:0:-1])
        X = np.concatenate([X, temp], axis=1)

    if real:
        local_ifft=np.fft.irfft
    else:
        local_ifft=np.fft.ifft


    X = np.real(local_ifft(X))
    win = 0.54 - .46 * np.cos(2 * np.pi * np.arange(fftsize) / (fftsize - 1))
    X = X / win[None]
    X = re_overlap(X,fftsize,step)
    return X


def pretty_spectrogram(d,log = False, thresh= 5, fft_size = 512, step_size = 64):
    """
    creates a spectrogram
    log: take the log of the spectrgram
    thresh: threshold minimum power for log spectrogram
    """
    specgram,phase = stft(d, fftsize=fft_size, step=step_size, real=True,
        compute_onesided=False)
    specgram = np.square(np.abs(specgram))
    if log == True: # volume normalize to max 1
        specgram = np.log10(specgram) # take log
        specgram[specgram < -thresh] = -thresh # set anything less than the threshold as the threshold
    else:
        specgram[specgram < thresh] = thresh # set anything less than the threshold as the threshold

    return specgram,phase

def mcra(specgram,log = False):
    #Initialize the parameters
    if log == True:
        specgram = np.power(10,specgram)
    length=specgram[0,:].size
    for x in range(specgram.shape[0]):
        if x == 0:
            noise_ps = specgram[x,:][:,None]
            P=specgram[x,:]
            Pmin=specgram[x,:]
            Ptmp=specgram[x,:]
            pk=np.zeros((length,1))
            ad=0.95
            a_s=0.8
            L=np.round(1000*2/20)
            delta=5
            ap=0.2
            result = noise_ps.T
        else:
            P=a_s*P+(1-a_s)*specgram[x,:]
            if np.remainder(x+1,L)==0:
                Pmin=np.minimum(Ptmp,P)
                Ptmp=P
            else:
                Pmin=np.minimum(Pmin,P)
                Ptmp=np.minimum(Ptmp,P)
            Srk=np.divide(P,Pmin)
            Ikl=np.zeros((length,1))
            ikl_index=np.nonzero(Srk>delta)
            Ikl[ikl_index]=1
            pk=ap*pk+(1-ap)*Ikl
            adk=ad+(1-ad)*pk
            noise_temp=np.multiply(adk,noise_ps)+np.multiply((1-adk),specgram[x,:][:,None])
            result=np.append(result,noise_temp.T,axis=0)
            noise_ps=noise_temp
    if log==True:
        result = np.log10(result)
    return result


def snr_estimation(mcra,restored_ps,noisy_ps,threshold,ay,ap,amax,amin,beta):
    one = np.zeros((noisy_ps/mcra).shape)+1

    post_snr_averaging=np.maximum([noisy_ps/mcra],one)
    post_snr_averaging=post_snr_averaging.reshape(post_snr_averaging.shape[1],post_snr_averaging.shape[2])
    for n in range(post_snr_averaging.shape[0]):
        if not n==0:
            post_snr_averaging[n,:]=ay*post_snr_averaging[n-1,:]+(1-ay)*post_snr_averaging[n,:]
    post_snr_averaging[post_snr_averaging<threshold]=0
    post_snr_averaging[post_snr_averaging>=threshold]=1
    for n in range(post_snr_averaging.shape[0]):
        if not n==0:
            post_snr_averaging[n,:]=ap*post_snr_averaging[n-1,:]+(1-ap)*post_snr_averaging[n,:]
    smoothing=amin+(1-post_snr_averaging)*(amax-amin)
    post_snr=np.maximum([noisy_ps/mcra],one)
    post_snr=post_snr.reshape(post_snr.shape[1],post_snr.shape[2])
    priori_snr=(1-smoothing)*(beta*(restored_ps/mcra)+(1-beta)*(post_snr-1))

    for n in range(post_snr_averaging.shape[0]):
        if not n==0:
            priori_snr[n,:]=smoothing[n,:]*priori_snr[n-1,:]+priori_snr[n,:]
    return priori_snr

def wiener_filter(priori_snr):
    return priori_snr/(1+priori_snr)
```

```python

def power_estimate(mcra,restored_ps,noisy_ps,threshold,log,noisy_input,pcra,ay,ap,amax,amin,beta):
    if log:
        mcra = np.power(10,mcra)
        restored_ps = np.power(10,restored_ps)
        noisy_ps = np.power(10,noisy_ps)
    if pcra:
        priori_snr = snr_estimation(mcra,restored_ps,noisy_ps,threshold,ay,ap,amax,amin,beta)
    else:
        priori_snr = restored_ps/(restored_ps+mcra)
    w_filter=wiener_filter(priori_snr)
    if noisy_input:
        clean_ps = noisy_ps * w_filter
    else:
        clean_ps = restored_ps * w_filter
    return clean_ps

def istft(X, phase, overlap=2):
    X = np.sqrt(X)
    X = phase * X
    fftsize=(X.shape[1]-1)*2
    hop = fftsize / overlap
    w = scipy.hanning(fftsize+1)[:-1]
    x = scipy.zeros(X.shape[0]*hop)
    wsum = scipy.zeros(X.shape[0]*hop)
    for n,i in enumerate(range(0, len(x)-fftsize, hop)):
        x[i:i+fftsize] += scipy.real(np.fft.irfft(X[n])) * w   # overlap-add
        wsum[i:i+fftsize] += w ** 2.
    pos = wsum != 0
    x[pos] /= wsum[pos]
    return x



# Also mostly modified or taken from https://gist.github.com/kastnerkyle/179d6e9a88202ab0a2fe
def invert_pretty_spectrogram(X_s, log = False, fft_size = 512,compute_onesided=False, step_size = 512/4, n_iter = 10):

    if log == True:
        X_s = np.power(10, X_s)
    X_s = np.sqrt(X_s)
    if compute_onesided:
        X_s = np.concatenate([X_s, X_s[:,-2:0:-1]], axis=1)
    X_t = iterate_invert_spectrogram(X_s, fft_size, step_size, n_iter=n_iter)
    return X_t

def iterate_invert_spectrogram(X_s, fftsize, step, n_iter=10, verbose=False):
    """
    Under MSR-LA License
    Based on MATLAB implementation from Spectrogram Inversion Toolbox
    References
    ----------
    D. Griffin and J. Lim. Signal estimation from modified
    short-time Fourier transform. IEEE Trans. Acoust. Speech
    Signal Process., 32(2):236-243, 1984.
    Malcolm Slaney, Daniel Naar and Richard F. Lyon. Auditory
    Model Inversion for Sound Separation. Proc. IEEE-ICASSP,
    Adelaide, 1994, II.77-80.
    Xinglei Zhu, G. Beauregard, L. Wyse. Real-Time Signal
    Estimation from Modified Short-Time Fourier Transform
    Magnitude Spectra. IEEE Transactions on Audio Speech and
    Language Processing, 08/2007.
    """
    X_best = copy.deepcopy(X_s)
    for i in range(n_iter):
        if verbose:
            print("Runnning iter %i" % i)
        if i == 0:
            X_t = invert_spectrogram(X_best, step, calculate_offset=True,
                                     set_zero_phase=True)
        else:
            # Calculate offset was False in the MATLAB version
            # but in mine it massively improves the result
            # Possible bug in my impl?
            X_t = invert_spectrogram(X_best, step, calculate_offset=True,
                                     set_zero_phase=False)
        est,phase= stft(X_t, fftsize=fftsize, step=step, compute_onesided=False)
        X_best = X_s * phase[:len(X_s)]
    X_t = invert_spectrogram(X_best, step, calculate_offset=True,
                             set_zero_phase=False)
    return np.real(X_t)

def invert_spectrogram(X_s, step, calculate_offset=True, set_zero_phase=True):
    """
    Under MSR-LA License
    Based on MATLAB implementation from Spectrogram Inversion Toolbox
    References
    ----------
    D. Griffin and J. Lim. Signal estimation from modified
    short-time Fourier transform. IEEE Trans. Acoust. Speech
    Signal Process., 32(2):236-243, 1984.
    Malcolm Slaney, Daniel Naar and Richard F. Lyon. Auditory
    Model Inversion for Sound Separation. Proc. IEEE-ICASSP,
    Adelaide, 1994, II.77-80.
    Xinglei Zhu, G. Beauregard, L. Wyse. Real-Time Signal
    Estimation from Modified Short-Time Fourier Transform
    Magnitude Spectra. IEEE Transactions on Audio Speech and
    Language Processing, 08/2007.
    """
    size = int(X_s.shape[1] // 2)
    wave = np.zeros((X_s.shape[0] * step + size))
    # Getting overflow warnings with 32 bit...
    wave = wave.astype('float64')
    total_windowing_sum = np.zeros((X_s.shape[0] * step + size))
    win = 0.54 - .46 * np.cos(2 * np.pi * np.arange(size) / (size - 1))

    est_start = int(size // 2) - 1
    est_end = est_start + size
    for i in range(X_s.shape[0]):
        wave_start = int(step * i)
        wave_end = wave_start + size
        if set_zero_phase:
            spectral_slice = X_s[i].real + 0j
        else:
            # already complex
            spectral_slice = X_s[i]

        # Don't need fftshift due to different impl.
        wave_est = np.real(np.fft.ifft(spectral_slice))[::-1]
        if calculate_offset and i > 0:
            offset_size = size - step
            if offset_size <= 0:
                offset_size = step
            offset = xcorr_offset(wave[wave_start:wave_start + offset_size],
                                  wave_est[est_start:est_start + offset_size])
        else:
            offset = 0
        wave[wave_start:wave_end] += win * wave_est[est_start - offset:est_end - offset]
        total_windowing_sum[wave_start:wave_end] += win
    wave = np.real(wave) / (total_windowing_sum + 1E-6)
    return wave

def xcorr_offset(x1, x2):
    """
    Under MSR-LA License
    Based on MATLAB implementation from Spectrogram Inversion Toolbox
    References
    ----------
    D. Griffin and J. Lim. Signal estimation from modified
    short-time Fourier transform. IEEE Trans. Acoust. Speech
    Signal Process., 32(2):236-243, 1984.
    Malcolm Slaney, Daniel Naar and Richard F. Lyon. Auditory
    Model Inversion for Sound Separation. Proc. IEEE-ICASSP,
    Adelaide, 1994, II.77-80.
    Xinglei Zhu, G. Beauregard, L. Wyse. Real-Time Signal
    Estimation from Modified Short-Time Fourier Transform
    Magnitude Spectra. IEEE Transactions on Audio Speech and
    Language Processing, 08/2007.
    """
    x1 = x1 - x1.mean()
    x2 = x2 - x2.mean()
    frame_size = len(x2)
    half = frame_size // 2 + 1
    corrs = np.convolve(x1.astype('float32'), x2[::-1].astype('float32'))
    corrs[:half] = -1E30
    corrs[-half:] = -1E30
    offset = corrs.argmax() - len(x1)
    return offset

# def freq_to_mel(f):
#     return 2595.*np.log10(1+(f/700.))
# def mel_to_freq(m):
#     return 700.0*(10.0**(m/2595.0)-1.0)

# def create_mel_filter(fft_size, n_freq_components = 64, start_freq = 300, end_freq = 8000):
#     """
#     Creates a filter to convolve with the spectrogram to get out mels

#     """
#     spec_size = fft_size/2
#     start_mel = freq_to_mel(start_freq)
#     end_mel = freq_to_mel(end_freq)
#     plt_spacing = []
#     # find our central channels from the spectrogram
#     for i in range(10000):
#         y = np.linspace(start_mel, end_mel, num=i, endpoint=False)
#         logp = mel_to_freq(y)
#         logp = logp/(rate/2/spec_size)
#         true_spacing = [int(i)-1 for i in np.ceil(logp)]
#         plt_spacing_mel = np.unique(true_spacing)
#         if len(plt_spacing_mel) == n_freq_components:
#             break
#     plt_spacing = plt_spacing_mel
#     if plt_spacing_mel[-1] == spec_size:
#         plt_spacing_mel[-1] = plt_spacing_mel[-1]-1
#     # make the filter
#     mel_filter = np.zeros((int(spec_size),n_freq_components))
#     # Create Filter
#     for i in range(len(plt_spacing)):
#         if i > 0:
#             if plt_spacing[i-1] < plt_spacing[i] - 1:
#                 # the first half of the window should start with zero
#                 mel_filter[plt_spacing[i-1]:plt_spacing[i], i] = np.arange(0,1,1./(plt_spacing[i]-plt_spacing[i-1]))
#         if i < n_freq_components-1:
#             if plt_spacing[i+1] > plt_spacing[i]+1:
#                 mel_filter[plt_spacing[i]:plt_spacing[i+1], i] = np.arange(0,1,1./(plt_spacing[i+1]-plt_spacing[i]))[::-1]
#         elif plt_spacing[i] < spec_size:
#             mel_filter[plt_spacing[i]:int(mel_to_freq(end_mel)/(rate/2/spec_size)), i] =  \
#                 np.arange(0,1,1./(int(mel_to_freq(end_mel)/(rate/2/spec_size))-plt_spacing[i]))[::-1]
#         mel_filter[plt_spacing[i], i] = 1
#     # Normalize filter
#     mel_filter = mel_filter / mel_filter.sum(axis=0)
#     # Create and normalize inversion filter
#     mel_inversion_filter = np.transpose(mel_filter) / np.transpose(mel_filter).sum(axis=0)
#     mel_inversion_filter[np.isnan(mel_inversion_filter)] = 0 # for when a row has a sum of 0

#     return mel_filter, mel_inversion_filter

def make_mel(spectrogram, mel_filter, shorten_factor = 1, log=False):
    mel_spec =np.transpose(mel_filter).dot(np.transpose(spectrogram))
    if log:
        return np.log10(mel_spec)
    else:
        return mel_spec


def mel_to_spectrogram(mel_spec, mel_inversion_filter, spec_thresh, shorten_factor,log=False):
    """
    takes in an mel spectrogram and returns a normal spectrogram for inversion
    """
    if log == True:
        mel_spec = np.power(10, mel_spec)
    mel_spec = (mel_spec+spec_thresh)
    uncompressed_spec = np.transpose(np.transpose(mel_spec).dot(mel_inversion_filter))
    uncompressed_spec = scipy.ndimage.zoom(uncompressed_spec, [1,shorten_factor])
    uncompressed_spec = uncompressed_spec -4
    return uncompressed_spec

# From https://github.com/jameslyons/python_speech_features

def hz2mel(hz):
    """Convert a value in Hertz to Mels
    :param hz: a value in Hz. This can also be a numpy array, conversion proceeds element-wise.
    :returns: a value in Mels. If an array was passed in, an identical sized array is returned.
    """
    return 2595 * np.log10(1+hz/700.)

def mel2hz(mel):
    """Convert a value in Mels to Hertz
    :param mel: a value in Mels. This can also be a numpy array, conversion proceeds element-wise.
    :returns: a value in Hertz. If an array was passed in, an identical sized array is returned.
    """
    return 700*(10**(mel/2595.0)-1)
'''''
def re_powspec(spectrogram,NFFT = fft_size):
    """Compute the power spectrum of each frame in frames. If frames is an NxD matrix, output will be NxNFFT.

    :param frames: the array of frames. Each row is a frame.
    :param NFFT: the FFT length to use. If NFFT > frame_len, the frames are zero-padded.
    :returns: If frames is an NxD matrix, output will be Nx(NFFT/2+1). Each row will be the power spectrum of the corresponding frame.
    """
    return np.sqrt(NFFT*spectrogram)

def powspec(spectrogram,NFFT = fft_size):
    """Compute the power spectrum of each frame in frames. If frames is an NxD matrix, output will be NxNFFT.

    :param frames: the array of frames. Each row is a frame.
    :param NFFT: the FFT length to use. If NFFT > frame_len, the frames are zero-padded.
    :returns: If frames is an NxD matrix, output will be Nx(NFFT/2+1). Each row will be the power spectrum of the corresponding frame.
    """
    return 1.0/NFFT * np.square(spectrogram)
'''
def get_filterbanks(nfilt=n_mel_freq_components,nfft=fft_size,samplerate=samplerate,lowfreq=0,highfreq=None):
    """Compute a Mel-filterbank. The filters are stored in the rows, the columns correspond
    to fft bins. The filters are returned as an array of size nfilt * (nfft/2 + 1)
    :param nfilt: the number of filters in the filterbank, default 20.
    :param nfft: the FFT size. Default is 512.
    :param samplerate: the samplerate of the signal we are working with. Affects mel spacing.
    :param lowfreq: lowest band edge of mel filters, default 0 Hz
    :param highfreq: highest band edge of mel filters, default samplerate/2
    :returns: A numpy array of size nfilt * (nfft/2 + 1) containing filterbank. Each row holds 1 filter.
    """
    highfreq= highfreq or samplerate/2
    assert highfreq <= samplerate/2, "highfreq is greater than samplerate/2"

    # compute points evenly spaced in mels
    lowmel = hz2mel(lowfreq)
    highmel = hz2mel(highfreq)
    melpoints = np.linspace(lowmel,highmel,nfilt+2)
    # our points are in Hz, but we use fft bins, so we have to convert
    #  from Hz to fft bin number
    bin = np.floor((nfft+1)*mel2hz(melpoints)/samplerate)

    fbank = np.zeros([nfilt,nfft//2+1])
    for j in range(0,nfilt):
        for i in range(int(bin[j]), int(bin[j+1])):
            fbank[j,i] = (i - bin[j]) / (bin[j+1]-bin[j])
        for i in range(int(bin[j+1]), int(bin[j+2])):
            fbank[j,i] = (bin[j+2]-i) / (bin[j+2]-bin[j+1])
    return fbank

def create_mel_filter(fft_size, n_freq_components = n_mel_freq_components, start_freq = start_freq, end_freq = end_freq, samplerate=samplerate):
    """
    Creates a filter to convolve with the spectrogram to get out mels

    """
    mel_inversion_filter = get_filterbanks(nfilt=n_freq_components,
                                           nfft=fft_size, samplerate=samplerate,
                                           lowfreq=start_freq, highfreq=end_freq)
    # Normalize filter
    mel_filter = mel_inversion_filter.T / mel_inversion_filter.sum(axis=1)

    return mel_filter, mel_inversion_filter


def fbank(data, fft_size = 2048,
        step_size = fft_size/16,
        spec_thresh = 0,
        lowcut = 0,
        highcut = 15000,
        samplerate = 48000,
        n_mel_freq_components = 40,
        shorten_factor = 1,
        start_freq = 20,
        end_freq = 8000,
        log = False
        ):
    wav_spectrogram,phase = pretty_spectrogram(data, fft_size = fft_size,
                                   step_size = step_size, log = False, thresh = spec_thresh)

    mel_filter, mel_inversion_filter = create_mel_filter(fft_size = fft_size,
                                                        n_freq_components = n_mel_freq_components,
                                                        start_freq = start_freq,
                                                        end_freq = end_freq,
                                                        samplerate=samplerate)

    mel_spec = np.transpose(make_mel(wav_spectrogram, mel_filter, shorten_factor = shorten_factor, log = log))
    mel_spec[mel_spec<0]=0
    return mel_spec

def invfbank(mel_spec, fft_size = 2048,
        step_size = fft_size/16,
        spec_thresh = 0,
        lowcut = 0,
        highcut = 15000,
        samplerate = 48000,
        n_mel_freq_components = 40,
        shorten_factor = 1,
        start_freq = 20,
        end_freq = 8000,
        log = False
        ):
    mel_filter, mel_inversion_filter = create_mel_filter(fft_size = fft_size,
                                                        n_freq_components = n_mel_freq_components,
                                                        start_freq = start_freq,
                                                        end_freq = end_freq,
                                                        samplerate=samplerate)
    re_mel_spec = mel_to_spectrogram(mel_spec, mel_inversion_filter, spec_thresh, shorten_factor,log=log)
    recovered_audio_orig = invert_pretty_spectrogram(re_mel_spec.T, fft_size = fft_size,
                                         step_size = step_size, log = False, n_iter = 10)
    return recovered_audio_orig



def power(data, fft_size = 2048,
        step_size = fft_size/16,
        spec_thresh = 0,
        lowcut = 0,
        highcut = 15000,
        samplerate = 48000,
        noise = False,
        log = False
        ):
    data = data[:,0].astype('float64')
    wav_spectrogram,phase = pretty_spectrogram(data, fft_size = fft_size,
                                   step_size = step_size, log = log, thresh = spec_thresh)
    if noise:
        noise_ps = mcra(wav_spectrogram,log=log)
        return wav_spectrogram,noise_ps,phase
    return wav_spectrogram,phase

def invpower(data, fft_size = 2048,
        step_size = fft_size/16,
        spec_thresh = 0,
        lowcut = 0,
        highcut = 15000,
        samplerate = 48000,
        n_mel_freq_components = 40,
        shorten_factor = 1,
        start_freq = 20,
        end_freq = 8000,
        log = False
        ):
    recovered_audio_orig = invert_pretty_spectrogram(data.T, fft_size = fft_size,
                                          step_size = step_size, log = log, n_iter = 300)
    return recovered_audio_orig
```