Indian Institute Of Technology Bombay

EE338: Digital Signal Processing

Application Assignment

Speech De-noising Web App

Group No.: 9
Conglomerate No.: 2
Group Member 1: Advait Kumar (18D070003)
Group Member 2: Janeel Patel (180020038)
Group Member 3: Prakhar Diwan (180100083)

December 5, 2020

Contents

1 Abstract

1.1 Initial Abstract

Ever since the onset of the coronavirus pandemic, India, as well as the rest of the world, have steadily moved towards the flipped classroom method of teaching. Corporate jobs have been replaced by 'work from home', and it is said that even after the situation stabilizes, this will no longer be a method that is frowned upon. Hence we felt that speech denoising will altogether hold greater importance in the post-COVID-19 era as well.
Speech denoising has been a mature area in the field of digital signal processing for a long time. Various methods/filters have been incorporated and the resulting solutions have been improved upon over the last half-century. However, the results are still not good enough to consider this problem statement to be solved. The current best methods all involve pure digital signal processing, which serves as further encouragement for us to also use digital signal processing to solve it. We will primarily be using Short Time Fourier Transforms(STFT) and Discrete Wavelet Transforms(DWT) as our baseline approaches. We will keep on updating our approach as we move ahead.

1.2 Additions made

After application of Digital Signal Processing concepts, we decided to use deep learning as a fine tuning step, and have made a comparison between not fine-tuned system and the fine-tuned version of it. We did it to check if any non linear transformations, other than thresholding and the normal digital signal processing steps, on the already transformed signal gave us improvement in Signal-to-Noise Ratio (SNR).
We also added implementation based on the Digital Signal Processing principle of Fast Fourier Transform (FFT) alongside the earlier mentioned Short-Time Fourier Transform and Discrete Wavelet Transform. We also made another set of comparisons amongst these implementations.
DISCLAIMER: The definitions and algorithms have been take from research papers, which have been mentioned in the References and Acknowledgements section.

2 Description of Speech Denoising Application

The problem statement for the application is described as follows:
Usually, in communication systems, the received signal are polluted with noise and distortion, which are mainly caused due to channel behaviour. Thus at the receiving end, the signal may lose it's information due to corruption. Thus, denoising of the received signal is essential for efficient communication in one-to-one as well as broadcast systems. It also aids the man to machine communication, since machine interpretation of noisy input signal is very poor. As one of the oldest signals, speech signal denoising is a mature area, but still improvements are being made.
Speech signal denoising, or noise reduction, is the removal of noise and distortions present in the signal for recovering back the original signal, free from noise. The objective is to improve the Signal-to-Noise Ratio (SNR) of the incoming speech audio. This is an important application of Digital Signal Processing, which has various uses such as in cellular phones, hearing aids, teleconferencing (has become widespread after COVID-19 pandemic) etc.
Previously, work in the area has been dominated by conventional methods such as Fourier Transform but also consists of non-conventional methods such as Wavelet Transform, due to the robust features it incorporates. Blind Adaptive Filtering (BAF) is another area, which utilizes wavelet transform for removal of unknown noises. Wavelet-based speech denoising has become the latest digital signal processing concept being used.
Fine-tuning of the system plays an important role in making it robust to noises and more accurate. In recent years with the advent of Deep Learning, the tuning is being done with the help of neural networks in most of the research related to speech denoising.
In this project, we worked on mainly three principles of Digital Signal Processing, namely, Fast Fourier Transform, Short-Time Fourier Transform and Discrete Wavelet Transform; along with exploring deep learning for fine-tuning our web application. We explored the 3 principles independently and made comparisons to identify that DWT outperforms FFT and STFT. For high utilization of our web application, we also used deep learning to fine-tune our model, making it capable to be used in real world.

3 Overview of our approach

For our application, first, we pre-process the incoming audio samples. The pre-processing part is of major importance and utilizes digital signal processing concepts. Here we used the dominant Fast Fourier Transform-based analysis in one of our implementations. This method is unable to provide the information on variation of spectrum with time, since speech signal contains non-stationary properties. After this we used Short-Time Fourier Transform to overcome this, since STFT (in a constant time and frequency resolution) can represent a signal in both the domains: frequency as well as time, with the help of time-windowing function. Here, we used the Hann window. After this we tried the latest Wavelet transform based approach, in which we used the Daubechies wavelets.

Next, we try to improve upon our approach by using deep learning. We use neural networks to find a transformation between the already pre-processed noisy clip and the clean clip. Since we pre-process our clips using three different transforms, we also used the same architecture of our neural network to train on these three transformed signals. Any non linear mapping between the input and the output frequency-time domains can be learnt using neural networks. This gives us improvement over the otherwise simpler 'thresholding method' of filtering which we employ when we use only digital signal processing.

The neural network comprises of a VGG-16 type of model, with dilated convolutional layers and adaptive batch normalisation. The model is trained using an Adam optimizer with an objective to minimise the L1 loss between the fourier transforms of the noisy and clean data. The exact details of the model as well as the training process (along with code) are discussed in the coming sections.

We finally incorporate whatever we have made into a web application, which we host publicly. A sound clip containing the noise is to be uploaded as an input. The best performing model out of the 3 models that we trained forms the backend of the application and by processing the input, it generates an output which is then uploaded on the website for the user to download. (demo in the video)

4 Principles and Theory

We used the following three methods, independent of each other, for pre-processing the audio samples, before giving them as inputs to neural network for training:

4.1 Fast Fourier Transform

Fourier transform is an essential tool in the area of signal processing and is used to obtain the frequency spectrum of a given time domain signal, it serves as a link between frequency and time domains. Upon moving to digital signal processing, we utilize the discrete Fourier transform (DFT), which is very important for real world applications. FFT is an algorithm for computation of DFT of a given sequence or it's inverse discrete Fourier transform (IDFT). It is a fast method computing DFT in $O(nlogn)$ time for an n-point sequence, instead of the earlier method of time complexity $O(n^2)$. After obtaining the spectrum, we filter off noise, broadly in our application.

4.1.1 Algorithm

Cooley-Turkey algorithm was used for FFT. This transform has a crucial dependence on the factorisation point noted by Carl Friedrich Gauss. It uses the divide and conquer technique, by dividing the discrete Fourier transform formulation with the help of division of odd and even points of sequence recursively till a single point is reached. The procedure is as shown below:

$$\begin{aligned}\sum_{n=0}^{N-1} a_n e^{-2\pi ink/N} &= \sum_{n=0}^{N/2-1} a_{2n} e^{-2\pi i(2n)k/N} + \sum_{n=0}^{N/2-1} a_{2n+1} e^{-2\pi i(2n+1)k/N} \\ &= \sum_{n=0}^{N/2-1} a_n^{even} e^{-2\pi ink/(N/2)} + e^{-2\pi ik/N} \sum_{n=0}^{N/2-1} a_n^{odd} e^{-2\pi ink/(N/2)}\end{aligned}$$

So, the transform keeps splitting the sequence into smaller sequences, and then combines the output of 2 N/2 point discrete Fourier transform for forming the N point discrete Fourier transform for the sequence. The procedure is depicted below:

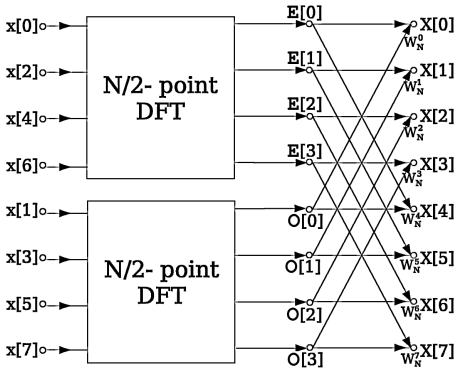

Figure 1: Fast Fourier Transform Method

4.1.2 Implementation

The input audio file was first converted into a numpy array using the librosa.load() function from the Librosa library. We then used the fft() function from the Numpy FFT library to transform this array (consisting of amplitudes in the time domain) to the frequency domain. We used the 2048 point FFT. We then use hard thresholding to try to filter out the high frequency noise components. The results are provided in the next section.

4.1.3 Other applications

As an essential part of mathematics and electrical engineering, fast Fourier transform and the discrete Fourier transform are largely the province of engineers and mathematicians looking for innovations in the world. Since, it's a computationally less expensive operation of a Discrete Fourier Transform, which is a central concept in signal processing, it gets used almost in every practical implementation of signal processing systems. In real life applications, fast Fourier transform might be helpful in sound engineering, seismology, in biomedical imaging, etc.

4.1.4 Advantages and Disadvantages

Discrete Fourier transform loses information in time domain, as it provides the frequency spectrum. Discrete Fourier transform is not able to track changes in frequency spectrum with respect to time. This leads to a decrease in quality in speech signal denoising, which is also our application, as speech signal is non-stationary. Fast Fourier Transform algorithm is a computationally less expensive version $O(n \log n)$ of the traditional discrete Fourier transform $O(n^2)$. Since, it is the one of the earliest algorithms, it is easily implementable and is a mature topic in signal processing. Though other methods for implementing the FFT do exist, such as direct implementation, recursive method, etc.

4.2 Short-Time Fourier Transform

The STFT represents a signal in the time-frequency domain by computing discrete Fourier transforms (DFT) over short overlapping windows. This helps in accommodating the frequency response changes with respect to time (i.e. tracks the change of frequency and phase content of local sections of signal) and thereby, allows an efficient an analysis of non-stationary signals, such as speech signals. In STFT, we trade frequency resolution for capturing the transient behaviour of the signal, which is made finer by using shorter time windowing. In our application, we have used the Hann window, which is shown below:

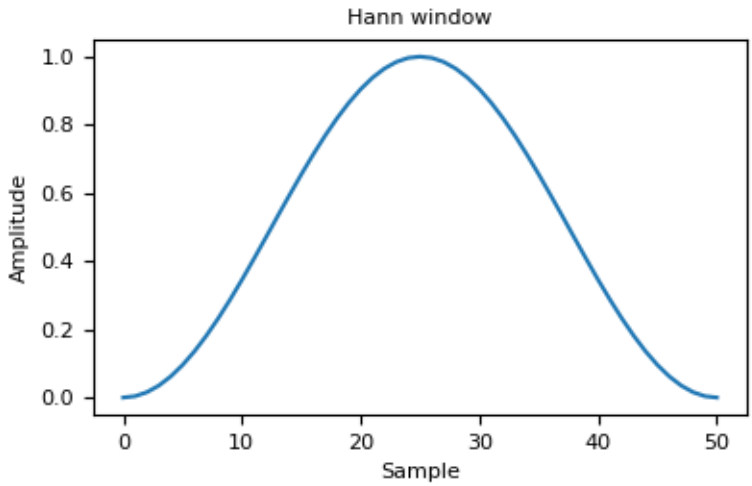

Figure 2: Hann Window

If we use window functions that are steep near the boundaries such as the rectangular window, this leads to a larger slope near that section's boundaries. This results in artifacts that are due to interferences that are spread across the entire frequency spectrum. These frequency components come from the window function instead of the original signal which is undesirable. Hence we use the Hann window, which is popularly used in signal processing. It is a raised cosine window, which goes to 0 very smoothly at the boundaries. This softens the artifacts as discussed above in the frequency domain. The Hann window is given by the following formula:

$$w[n] = \begin{cases} 0.5 - 0.5\cos(\frac{2\pi n}{N}) & -\frac{N}{2} \leq n \leq \frac{N}{2} \\ 0 & \text{o.w} \end{cases}$$

4.2.1 Algorithm

By definition, the Short-Time Fourier transform pair is given as follows:

$$X_{STFT}[m,n] = \sum_{k=0}^{L-1} x[k]g[k-m]e^{-i2\pi nk/L}$$
$$x[k] = \sum_{m}\sum_{n} X_{STFT}[m,n]g[k-m]e^{i2\pi nk/L}$$

where, $x[k]$ denotes a signal and $g[k]$ denotes an L-point window function.
So the computation is simply done by applying the window function in time domain and then taking the fast Fourier transform of the temporal divisions of the resultant sequence.
A visual representation of the procedure is shown below. Here the spectrogram, which is an intensity plot

of STFT magnitude over time is shown. The x-axis represents time and y-axis frequency. Figure 3 was obtained from [5] A compact view of Short-Time Fourier transformation is shown below:

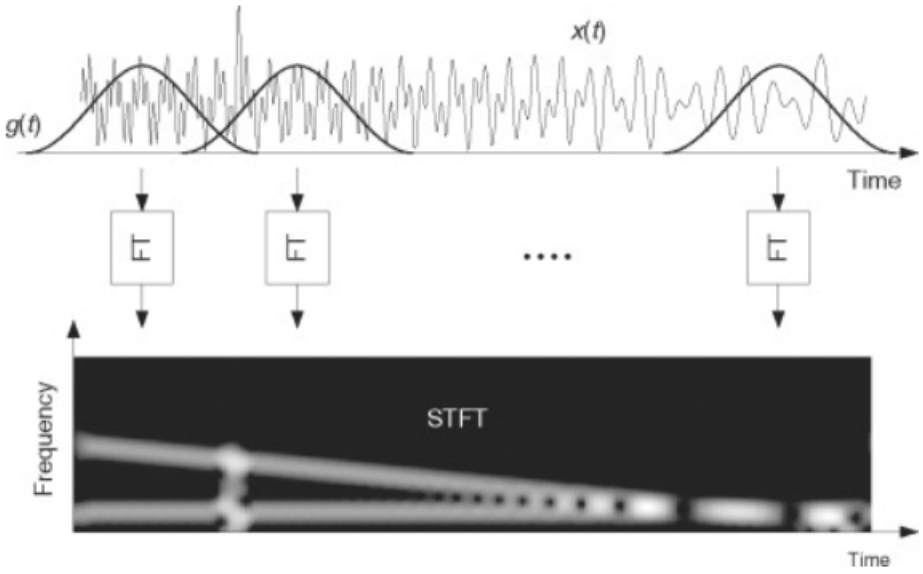

Figure 3: Application of STFT with Hann Window

Figure 4: Compact view of STFT

4.2.2 Implementation

For implementation of STFT, we used the stft() function from the Librosa library. The audio file was converted to a numpy array using the same procedure as mentioned for the fast fourier transform. We then use hard thresholding to try to filter out the high frequency noise components. The results are provided in the next section.

4.2.3 Other applications

In our application, we used it to analyse speech signal for efficient denoising. It is highly popular in audio engineering, since audio signals, which are non-stationary can be effectively analysed with the help of Short-Time Fourier transform.
Apart from denoising of speech signals, it is also used for analyzing music signals and filtering of them as well. This information can be used for tuning audio effects, which is a usual practice during music compositions. It can be utilized in almost every audio processing system.

4.2.4 Advantages and Disadvantages

Short-Time Fourier transform, being one of the earliest time dependent transforms, can be used in analysis of non-stationary signals As it is the one of the popular transforms in audio engineering, it's usage in our application seemed essential and being highly documented helped a lot during its implementation.
But there is a trade-off involved between the between temporal and frequency dimensions, i.e. if you get a finer resolution in time domain, you lose resolution in the frequency domain(uncertainty principle in signal processing). This factor makes it less efficient for analysing non-periodic signals with fast-transient features (i.e. high frequency content within shirt duration). It's usage has also been limited to analysis of speech and music signals.

4.3 Discrete Wavelet Transform

The Wavelet transform is either Continuous Wavelet transform (CWT) or Discrete Wavelet transform (DWT), based on application to the respective type of function/signal. In our application we used Discrete Wavelet tranform with Daubechies wavelets. Daubechies wavelets were invented by Ingrid Daubechies, and these are called compactly supported orthonormal wavelets, and made discrete wavelet analysis practically possible. Daubechies wavelet family is shown below, here in dBn, n denotes the order of wavelet. We used the eighth order Daubechies wavelet in our implementation. Denoising by DWT is based on the princi-

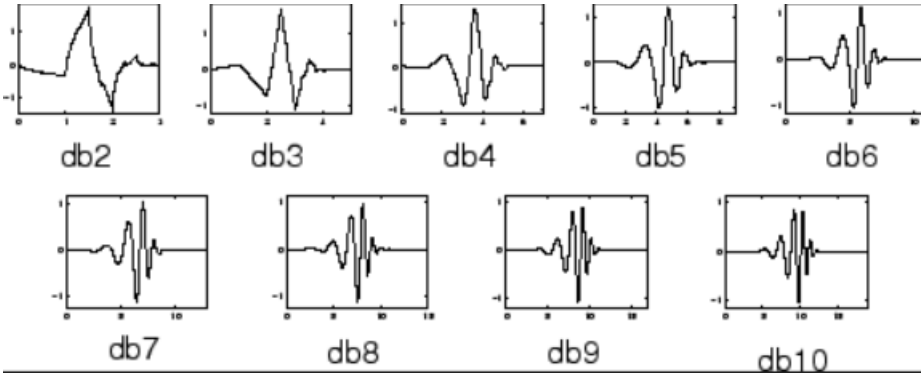

Figure 5: Daubechies Wavelet Family

ple that the input signal can be reconstructed by limited numbers of wavelet coefficients. Time computation for DWT is $O(N)$, where N is the number of points in the input sequence.

4.3.1 Algorithm and Definition

The CWT calculation is applied like STFT, but with 2 major differences. First, the window function is modified as the transform is computed for each single spectral component of the signal. Second, this windowed signal then is tranformed as follows, instead of Fourier transform:

$$CWT_x^{\Psi}(\tau, s) = \frac{1}{\sqrt{|s|}} \int x(t)\overline{\Psi(\frac{t-\tau}{s})}$$

Here, there are 2 variables in the output of CWT: τ is the translation parameter and s is the scale parameter. The novelty from STFT is s, which is a dilating or a compressing operation, depending on its value. $\Psi(t)$ is the mother wavelet, which is the transforming function. **[7849377]**
Now, Discrete Wavelet Transform is Continuous Wavelet Transform with discrete coefficients and lesser computational complexity. It is similar to set of inner products between a finite-length sequence and a discretised wavelet basis. A DWT coefficient results from each such inner product.
One level of DWT for a signal $x[n]$ is calculated by passing it through a low-pass and a high-pass filter, both of which are quadrature mirror filter of each other. The outputs give detail coefficients (from high-pass filter) and approximation coefficients (from low-pass filter).
As in both the outputs, the frequency spectrum is halved, thereby, both the output signals can be downsampled by factor of 2 and still be reconstructed (Nyquist's Sampling Theorem). This is a single level, for achieving a higher resolution, higher order Discrete-Wavelet Transform is used, in which the outputs are fed into new low-pass and high-pass filters to obtain next order Discrete-Wavelet Transform coefficients. A 3 layer Discrete-Wavelet Transformation is also shown below, where S is the original signal.

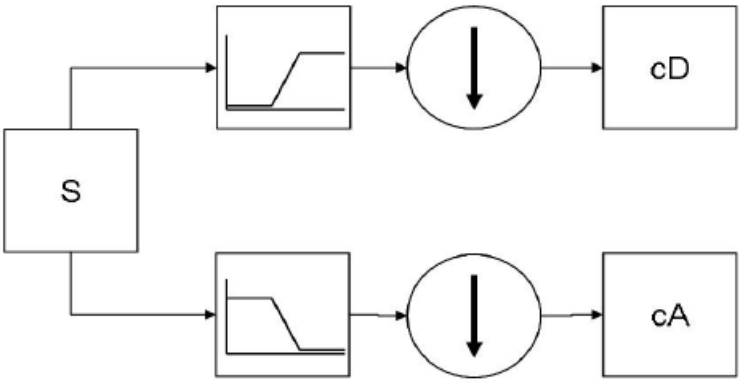

Figure 6: Discrete-Wavelet Transformation (single order)

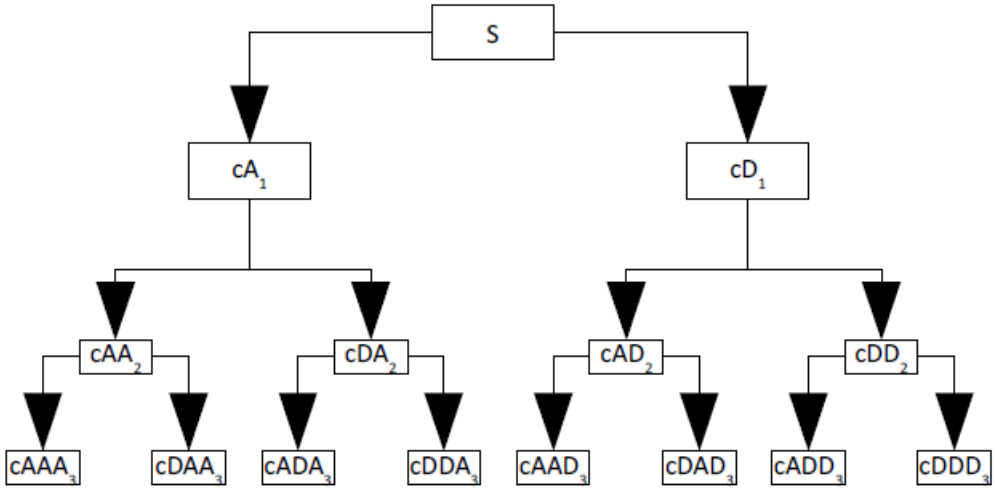

Figure 7: 3 layer Discrete-Wavelet Transformation

4.3.2 Implementation

For implementation of Discrete Wavelet Transform, we used the dwt() function from the PyWavelets library. We again use hard thresholding to threshold the high frequency components of the noisy signal that might be corresponding to the noise component. Other procedure remains the same as for the above 2 transforms

4.3.3 Other applications

The discrete wavelet transform has large number of applications in the area of signal coding, data compression, image processing, digital communications, digital signal processing etc.
One of its popular uses is in the field of image processing, where it's various order transforms work as features for several computer vision tasks such as image classification, segmentation etc. For these tasks 2-D wavelets are used instead of 1-D wavelets.
A sample signal processing utilizing Discrete-Wavelet transform is shown below:

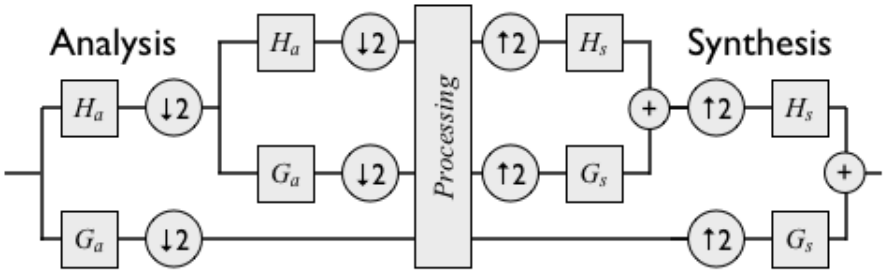

Figure 8: Overall view of a system which uses DWT

4.3.4 Advantages and Disadvantages

Wavelet transform extends the analysis ability to non-periodic signals with fast transient feature. As Wavelet transform keeps the localisation information of the prominent frequencies, whereas Fourier transform doesn't. That is if the high frequency components carry the important information, the time localization resolution will be more for higher frequencies since they are represented by larger number of samples. The time localisation will be of low resolution in the case, where main information lies in low frequency region.
As speech signal carries information in the high frequency region, it made DWT highly important to our application, since it also incorporates efficient analysis of transient signals.
However it has a few shotcomings. Firstly it has shift selectivity. It is shift variant due to downsampling that occurs in the process. In a L-level wavelet decomposition it is shift invariant only for shifts by multiples of 2^L. The second drawback is that it lacks directional selectivity. This is because we apply a DWT individually on rows and then columns of signal. Hence the resulting output is of rank one, which is very poor at separating directions other than horizontal or vertical. The final drawback is that it also lacks information about the phase. Being a discrete downsampling process information about the phase is quite often lost or neglected in favour of amplitude information. Hence the DSP procedures of soft or hard thresholding might not work in these cases.

4.4 Deep Learning

We now introduce deep learning into the picture. Neural networks have been widely used in the past to tackle a variety of problems in the fields of audio processing, image processing & video processing. They provide us a way to possible learn a non linear mapping relationship between the inputs and the outputs. The function or the 'model' is dynamic and its constituents parameters or 'weights' keep on changing over time or 'trained' as it encounters new inputs and learns new dependencies. The model is trained using a loss function that measures some form of the error between the expected output and the output of the network. As we decrease this error, the model keeps on becoming more and more accurate. This training is done with the help of an 'optimizer' and a 'loss function'.

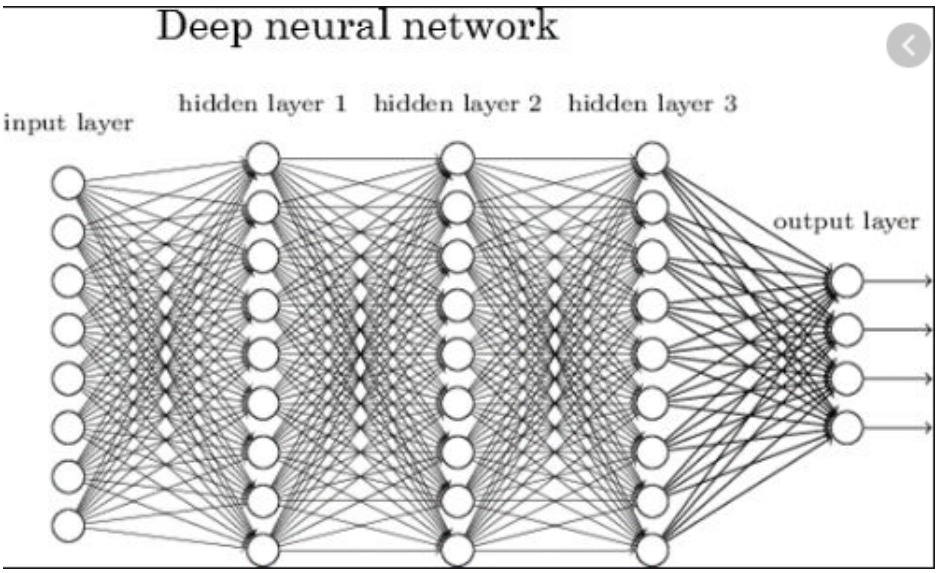

Figure 9: A deep neural network

4.4.1 Model Description

For our implementation we used a model that had an architecture similar to the VGG-19 network albeit with a few differences.
The first difference is that our network uses **dilated** convolutional layers instead of the normal convolutional layers used by the original model. This is to capture the longer temporal dependencies that may exist in the speech. Words that have been spoken in the past may be factors that govern the frequency of words that are being spoken currently. Hence it is important for our network to also learn these dependencies.
The second difference is that our model uses a different kind of normalisation layer called 'adaptive batch normalisation'. This normalisation keeps on learning and tuning the batch normalisation parameters as and when the model is training. It has been shown to give superior performance in other fields, hence we decided to use it here.
The third and the final difference or modification is that our model uses 'complex' weights instead of only real weights in the case of STFT and FFT. This is because after applying STFT and FFT, the frequency domain values are complex. Hence we felt that it would be best to directly try to learn a complex domain mapping between the input and the output rather than learning real and imaginary mappings separately and then trying to combine them. This approach would decrease the time required for training as well as memory.

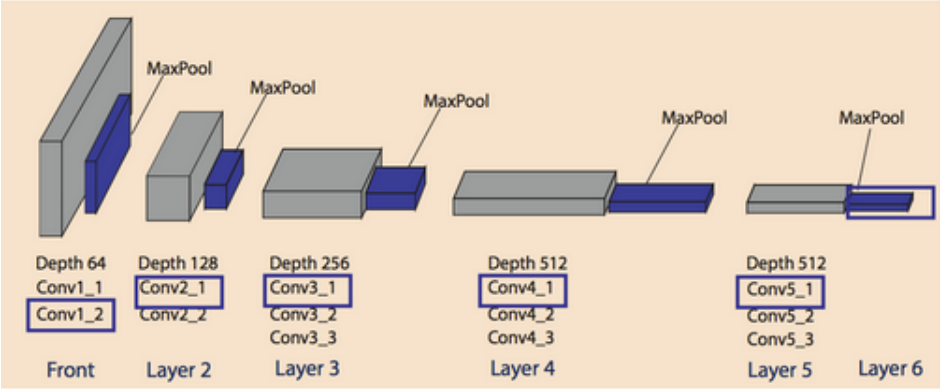

Figure 10: Architecture of the VGG model which we modified

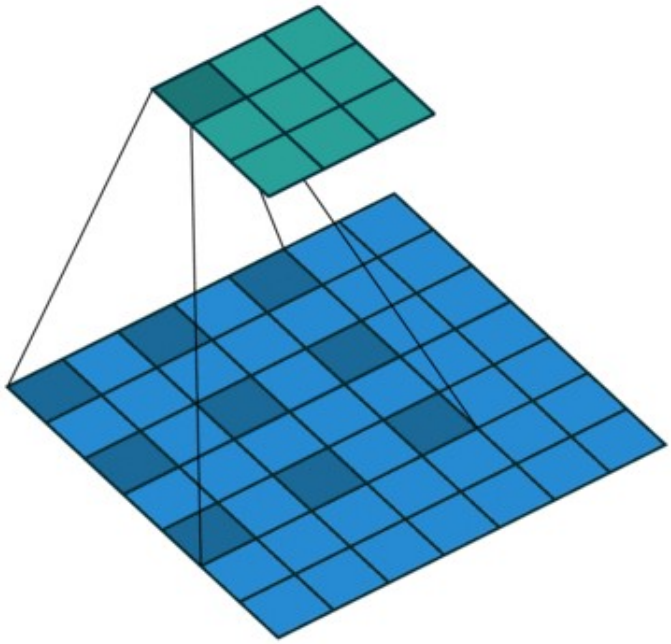

Figure 11: Dilated convolutional layers

4.4.2 Training Process

We trained the three models separately. We used ADAM optimizer to minimize the L1 loss [**germain2018speech**] between the expected and the model output. We trained all the three models for 100 epochs in about 20 hours using the free Tesla P80 GPUs provided by Google Colaboratory.

Method	A → W	D → W	W → D	A → D	D → A	W → A	Avg
AlexNet [Krizhevsky *et al.*, 2012]	61.6	95.4	99.0	63.8	51.1	49.8	70.1
DDC [Tzeng *et al.*, 2014]	61.8	95.0	98.5	64.4	52.1	52.2	70.6
DAN [Long *et al.*, 2015]	68.5	96.0	99.0	67.0	54.0	53.1	72.9
Deep CORAL [Sun and Saenko, 2016]	66.4	95.7	99.2	66.8	52.8	51.5	72.1
RevGrad [Ganin and Lempitsky, 2015]	73.0	96.4	99.2	-	-	-	-
Inception BN [Ioffe and Szegedy, 2015]	70.3	94.3	**100**	70.5	**60.1**	57.9	75.5
SA [Fernando *et al.*, 2013]	69.8	95.5	99.0	71.3	59.4	56.9	75.3
GFK [Gong *et al.*, 2012]	66.7	**97.0**	99.4	70.1	58.0	56.9	74.7
LSSA [Aljundi *et al.*, 2015]	67.7	96.1	98.4	71.3	57.8	57.8	74.9
CORAL [Sun *et al.*, 2016]	70.9	95.7	99.8	71.9	59.0	60.2	76.3
AdaBN	74.2	95.7	99.8	**73.1**	59.8	57.4	76.7
AdaBN + CORAL	**75.4**	96.2	99.6	72.7	59.0	**60.5**	**77.2**

Figure 12: Results of adaptive batch normalisation paper

5 Observations and Investigations

5.1 Comparison-I

Here, we compare the performances of pure DSP-based models of Fast Fourier Transform, Short Time Fourier Transform and Discrete Wavelet Transform.

Method	Accuracy(SNR)
FFT	6.86 DB
STFT	10.73 DB
DWT	**12.24 DB**

Table 1: Performance of purely DSP based methods

5.2 Comparison-II

Here, we compare the performances of fine-tuned (via deep learning) DSP-based models of Fast Fourier Transform, Short Time Fourier Transform and Discrete Wavelet Transform.

Method	Accuracy(SNR)
FFT	7.84 DB
STFT	12.36 DB
DWT	**13.86 DB**

Table 2: Performance of purely DSP based methods fine tuned with deep learning

5.3 Comparison-III

Here, we compare the performances amongst the pure DSP models with their respective fine-tuned version.

Method	DSP Accuracy(SNR)	DL + DSP Accuracy(SNR)
FFT	6.86 DB	7.84 DB
STFT	10.73 DB	12.36 DB
DWT	**12.24 DB**	**13.86 DB**

Table 3: Performance of purely DSP based method and DSP fine tuned with deep learning based methods

6 Conclusion

Hence it is evident from the above results that using digital signal processing gives good results for speech denoising. On cascading it with deep learning we obtain a significant improvement of more than 1 DB in SNR accuracy. Hence using deep learning is also helpful in this field. We finally notice that the Discrete Wavelet Transform gives the best results in both the above cases. This was expected as from our analysis we saw that DWT was able to improve upon the shortcomings of both the STFT and the FFT transforms. This is one of the reasons why DWT is also used in the preprocessing stages of a large number of problem statements in signal processing as well as deep learning.

7 Deliverables

Removal of noise from audio recordings made on mobile phones.

- Noise removal from pre-recorded online lectures
- Filtering noise from audio recorded on mobile phones
- Audio enhancement of old recordings, movies, etc.

Restoration of lost audio signals from corrupted media, old audio recordings, etc.
Singing and recording audios, can then be filtered for removal of background noise.
Noise free pre-recorded presentations for professionals, working from home during pandemic.

8 Contributions to Aatmanirbhar Bharat Initiative

We observed that most of the speech denoising software such as audacity (the US), wavepad (Australia), Sony Creative Noise Reduction (Japan), Solicall (Israel), etc. have been made and developed outside India. We did not come across any indigeneous (and cheap, open-source) speech noise removal software. Hence this will be our contribution.
Also, the majority of these software products are compatible only with a computer/laptop. Hence, if time permits, we wish to develop a WEB APP, so that it can be easily accessible to one and all at the touch of their fingers and the time taken to upload/download the files from a bigger device gets saved.
Since in India not everyone has a quiet environment fit for music composition, this app will help them record noise-free compositions. This will also be of help to uncover the musical talents, existing in India; since any person can send their de-noised composition for auditions, expanding their reach further.

9 References and Acknowledgements

References

[1] Ing Yann Soon, Soo Ngee Koh and Chai Kiat Yeo, "Wavelet for speech denoising," TENCON '97 Brisbane - Australia. Proceedings of IEEE TENCON '97. IEEE Region 10 Annual Conference. Speech and Image Technologies for Computing and Telecommunications (Cat. No.97CH36162), Brisbane, Queensland, Australia, 1997, pp. 479-482 vol.2, doi: 10.1109/TENCON.1997.648249.

[2] M. A. Oktar, M. Nibouche and Y. Baltaci, "Speech denoising using discrete wavelet packet decomposition technique," 2016 24th Signal Processing and Communication Application Conference (SIU), Zonguldak, 2016, pp. 817-820, doi: 10.1109/SIU.2016.7495865.

[3] Short Time Fourier Analysis,
`http://libvolume2.xyz/biomedical/btech/semester7/speechsignalprocessing/shorttimefourieranalysis/shorttimefourieranalysisnotes2.pdf`

[4] Audio AI: isolating vocals from stereo music using Convolutional Neural Networks
https://towardsdatascience.com/audio-ai-isolating-vocals-from-stereo-music-using-convolutional-neural-networks-210532383785

[5] Short-Time Fourier Transform
https://www.sciencedirect.com/topics/engineering/short-time-fourier-transform

[6] M. Parchami, W. Zhu, B. Champagne and E. Plourde, "Recent Developments in Speech Enhancement in the Short-Time Fourier Transform Domain," in IEEE Circuits and Systems Magazine , vol. 16, no. 3, pp. 45-77, thirdquarter 2016, doi: 10.1109/MCAS.2016.2583681

[7] Y. Cengiz and Y. D. U. Arıöz, "An Application for speech denoising using Discrete wavelet transform," 2016 20th National Biomedical Engineering Meeting (BIYOMUT) , Izmir, 2016, pp. 1-4, doi: 10.1109/BIYOMUT.2016.7849377.

[8] VGG Network
https://arxiv.org/abs/1409.1556

[9] L1 loss for speech denoising
https://arxiv.org/abs/1806.10522

[10] Dilated Convolutions
https://arxiv.org/abs/1511.07122

[11] Adaptive Batch Normalisation
https://arxiv.org/abs/1603.04779

10 Tools and Languages used

In the project, we mainly utilized python programming language along with its libraries: Librosa, Pytorch, PyWavelets, Numpy & ComplexPytorch. All these libraries are open source.

11 Appendix

The project will be made open-source as a contribution to the community of Digital Signal Processing. The codes utilized by us are given below:

11.1 Code For Complex Valued Batch Normalisation

```
class ComplexBatchNorm2d(_ComplexBatchNorm):

def forward(self, input_r, input_i):
    assert(input_r.size() == input_i.size())
    assert(len(input_r.shape) == 4)
    exponential_average_factor = 0.0


    if self.training and self.track_running_stats:
        if self.num_batches_tracked is not None:
            self.num_batches_tracked += 1
            if self.momentum is None:  # use cumulative moving average
                exponential_average_factor = 1.0 / float(self.num_batches_tracked)
            else:  # use exponential moving average
                exponential_average_factor = self.momentum
```

```
if self.training:

    # calculate mean of real and imaginary part
    mean_r = input_r.mean([0, 2, 3])
    mean_i = input_i.mean([0, 2, 3])


    mean = torch.stack((mean_r,mean_i),dim=1)

    # update running mean
    with torch.no_grad():
        self.running_mean = exponential_average_factor * mean\
            + (1 - exponential_average_factor) * self.running_mean

    input_r = input_r-mean_r[None, :, None, None]
    input_i = input_i-mean_i[None, :, None, None]

    # Elements of the covariance matrix (biased for train)
    n = input_r.numel() / input_r.size(1)
    Crr = 1./n*input_r.pow(2).sum(dim=[0,2,3])+self.eps
    Cii = 1./n*input_i.pow(2).sum(dim=[0,2,3])+self.eps
    Cri = (input_r.mul(input_i)).mean(dim=[0,2,3])

    with torch.no_grad():
        self.running_covar[:,0] = exponential_average_factor * Crr * n / (n - 1)\
            + (1 - exponential_average_factor) * self.running_covar[:,0]

        self.running_covar[:,1] = exponential_average_factor * Cii * n / (n - 1)\
            + (1 - exponential_average_factor) * self.running_covar[:,1]

        self.running_covar[:,2] = exponential_average_factor * Cri * n / (n - 1)\
            + (1 - exponential_average_factor) * self.running_covar[:,2]

else:
    mean = self.running_mean
    Crr = self.running_covar[:,0]+self.eps
    Cii = self.running_covar[:,1]+self.eps
    Cri = self.running_covar[:,2]#+self.eps

    input_r = input_r-mean[None,:,0,None,None]
    input_i = input_i-mean[None,:,1,None,None]

# calculate the inverse square root the covariance matrix
det = Crr*Cii-Cri.pow(2)
s = torch.sqrt(det)
t = torch.sqrt(Cii+Crr + 2 * s)
inverse_st = 1.0 / (s * t)
Rrr = (Cii + s) * inverse_st
Rii = (Crr + s) * inverse_st
Rri = -Cri * inverse_st

input_r, input_i = Rrr[None,:,None,None]*input_r+Rri[None,:,None,None]*input_i, \
                   Rii[None,:,None,None]*input_i+Rri[None,:,None,None]*input_r
```

```
        if self.affine:
            input_r, input_i = self.weight[None,:,0,None,None]*input_r+self.weight[None,:,2,None,None]*i
                               self.bias[None,:,0,None,None], \
                               self.weight[None,:,2,None,None]*input_r+self.weight[None,:,1,None,None]*i
                               self.bias[None,:,1,None,None]

        return input_r, input_i
```

11.2 Code for Complex Convolutional Layer

```
class ComplexConv2d(Module):

    def __init__(self,in_channels, out_channels, kernel_size=3, stride=1, padding = 0,
                 dilation=1, groups=1, bias=True):
        super(ComplexConv2d, self).__init__()
        self.conv_r = Conv2d(in_channels, out_channels, kernel_size, stride, padding, dilation, groups,
        self.conv_i = Conv2d(in_channels, out_channels, kernel_size, stride, padding, dilation, groups,

    def forward(self,input_r, input_i):
#         assert(input_r.size() == input_i.size())
        return self.conv_r(input_r)-self.conv_i(input_i), \
               self.conv_r(input_i)+self.conv_i(input_r)
```

11.3 Code For Adaptive Batch Normalisation

```
    class AdaptiveBatchNorm2d(nn.Module):
    def __init__(self, num_features, eps=1e-5, momentum=0.1, affine=True):
        super(AdaptiveBatchNorm2d, self).__init__()
        self.bn = ComplexBatchNorm2d(num_features, eps, momentum, affine)
        self.a = nn.Parameter(torch.FloatTensor(1, 1, 1, 1))
        self.b = nn.Parameter(torch.FloatTensor(1, 1, 1, 1))

    def forward(self, x_real, x_imag):
        output_real, output_imag = self.bn(x_real, x_imag)
        return self.a * x_real + self.b * output_real, self.a * x_imag + self.b * output_imag
```

11.4 Code For Model Architecture

```
    class vgg_type(nn.Module):
  def __init__(self):
    super(vgg_type, self).__init__()
    self.conv1 = ComplexConv2d(1, 64, [1,3], padding=[0,1], bias = False)
    self.norm1 = AdaptiveBatchNorm2d(64)
    self.relu = ComplexReLU()
    self.lrelu = ComplexReLU()

    self.conv2 = ComplexConv2d(64,64,[1,3], padding = [0,1], dilation = 1, bias = False)
    self.conv3 = ComplexConv2d(64,64,[1,3], padding = [0,2], dilation = 2, bias = False)
    self.conv4 = ComplexConv2d(64,64,[1,3], padding = [0,4], dilation = 4, bias = False)
    self.conv5 = ComplexConv2d(64,64,[1,3], padding = [0,8], dilation = 8, bias = False)
    self.conv6 = ComplexConv2d(64,64,[1,3], padding = [0,16], dilation = 16, bias = False)
```

```
    self.conv7 = ComplexConv2d(64,64,[1,3], padding = [0,32], dilation = 32, bias = False)
    self.conv8 = ComplexConv2d(64,64,[1,3], padding = [0,64], dilation = 64, bias = False)
    self.conv9 = ComplexConv2d(64,64,[1,3], padding = [0,128], dilation = 128, bias = False)
    self.conv10 = ComplexConv2d(64,64,[1,3], padding = [0,256], dilation = 256, bias = False)

    self.norm2 = AdaptiveBatchNorm2d(64)
    self.norm3 = AdaptiveBatchNorm2d(64)
    self.norm4 = AdaptiveBatchNorm2d(64)
    self.norm5 = AdaptiveBatchNorm2d(64)
    self.norm6 = AdaptiveBatchNorm2d(64)
    self.norm7 = AdaptiveBatchNorm2d(64)
    self.norm8 = AdaptiveBatchNorm2d(64)
    self.norm9 = AdaptiveBatchNorm2d(64)
    self.norm10 = AdaptiveBatchNorm2d(64)

    self.final = ComplexConv2d(64,1, [1,1])

  def forward(self, x):
    x_real, x_imag = self.conv1(x.real, x.imag)
    x_real, x_imag = self.norm1(x_real, x_imag)
    x_real, x_imag = self.relu(x_real, x_imag)

    x_real, x_imag = self.conv2(x_real, x_imag)
    x_real, x_imag = self.norm2(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)

    x_real, x_imag = self.conv3(x_real, x_imag)
    x_real, x_imag = self.norm3(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)

    x_real, x_imag = self.conv4(x_real, x_imag)
    x_real, x_imag = self.norm4(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)

    x_real, x_imag = self.conv5(x_real, x_imag)
    x_real, x_imag = self.norm5(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)

    x_real, x_imag = self.conv6(x_real, x_imag)
    x_real, x_imag = self.norm6(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)

    x_real, x_imag = self.conv7(x_real, x_imag)
    x_real, x_imag = self.norm7(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)

    x_real, x_imag = self.conv8(x_real, x_imag)
    x_real, x_imag = self.norm8(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)

    x_real, x_imag = self.conv9(x_real, x_imag)
    x_real, x_imag = self.norm9(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)
```

```
    x_real, x_imag = self.conv10(x_real, x_imag)
    x_real, x_imag = self.norm10(x_real, x_imag)
    x_real, x_imag = self.lrelu(x_real, x_imag)

    x_real, x_imag = self.final(x_real, x_imag)
    return x_real, x_imag
```

11.5 Code For Training

```
def train(dataloader, model, optimizer, criterion):
  model.train()
  train_losses.append(0)
  progbar = tqdm_notebook(total = len(dataloader), desc = 'Train')


  for i, (input, target) in enumerate(dataloader):
    optimizer.zero_grad()

    input, target = Variable(input.unsqueeze_(0).cuda()), Variable(target.unsqueeze_(0).cuda())

    input = input.type(torch.cfloat)

    output_real, output_imag = model(input)

    error_real = criterion(output_real, target.real)
    error_imag = criterion(output_imag, target.imag)

    error = error_imag + error_real

    error.backward()
    optimizer.step()

    train_losses[-1] = train_losses[-1] + error.data
    progbar.set_description('Train Angle (loss=%.4f)' % (train_losses[-1]/(i+1)))
    progbar.update(1)

  train_losses[-1] = train_losses[-1]/len(dataloader)
```

The remaining code for the web app as well as the entire training code is present at the following GitHub repository :- `https://github.com/janeelpatel/EE338WebApp`