



Multi-Microphone Speaker Separation based on Deep DOA Estimation

Project team:

Mor Zerahia

Idan Uri

Instructor: Mr. Shlomi Chazan

Academic Supervisor: Prof. Sharon Gannot

Dsp lab instructor: Mr. Pini Tandeitnik

Table of contents

Abstract	2
Motivation.....	3
Problem formulation.....	4
Short time fourier transform.....	6
W-disjoint orthogonality	8
Deep neural network	11
Direction of arrival	13
The separation algorithm.....	15
The U-NET architecture.....	18
Building the dataset	26
Data loader.....	29
Results	31
Acknowledgments.....	36
Code	37

Abstract

Our project is a multi-microphone speech separation based on masking inferred from the speaker's direction of arrival (DOA).

The masking task following the W-disjoint orthogonality property of speech signals that claim that in each time-frequency (TF) bin there is only one dominate speaker. Therefore, This TF bin can be associated with a single DOA.

In our procedure, we apply a deep neural network (DNN) with a U-net architecture to infer the DOA of each TF bin from a concatenated set of the spectra of the microphone signals.

In our project we will produce mask using spatial information to determine the DOA of each time frequency bin and we will obtain separation by multiplying the mask with the original spectrum of the speakers.

Motivation

In today's life, speech plays an essential role not only in human communication, but in different applications that supposed to make our life more comfortable like "SIRI" of Apple and "ALEXA" of Amazon. These two applications (and much more) work when someone talk directly to the device and ask it to do a certain thing that the application is capable to do.

The problem occurs when two people (or more) are speaking at the same time. In that case the device would not be able to understand what they say.

latest approaches had a dramatic impact on the single-microphone speech separation field. Most of them only exploit spectral information and ignored spatial information which contain more information about the signals that can be beneficial and useful in speaker separation domain.

In our project, instead of working with one microphone we will use an array of microphone so that we can use not only spectral information but spatial information as well. In that case we will separate the speakers and we will be able to determine and direction of arrivals (DOA) by addressing each time frequency bin to the right angle.

Problem formulation

In this section we will elaborate on the process of waves in acoustic channel as a spatial information.

Consider an array of M microphones capturing a mixture of N speech sources in a reverberant enclosure. The i -th speech signal $s^i(t)$ propagates through the acoustic channel before being captured by the m -th microphone.

In the STFT domain it can be written as:

$$z_m(l, k) = \sum_{i=1}^N s^i(l, k) h_m^i(l, k)$$

Where h_m^i is the room impulse response (RIR) relating the i -th speaker and the m -th microphone. and where l and k , are the time-frame and the frequency-bin (TF) indexes, respectively.

(RIR details are given in the "Building The Dataset" section)

In general, the room impulse response is the transfer function between the sound source and microphone. In order to recover the original sound source, the received microphone signal can be convolved with the inverse of the room impulse response function.

It is a complicated problem to determine the exact effect that reflections have on a direct signal. The resulting superposition (or cancellation) depends on both the relative amplitude and phase of each wave. The amplitude of the reflected signal (and phase rotation) can vary according to the distance travelled by the wave, the wavelength of the wave, the angle of incidence with each reflective surface, and the absorbent nature of the room among other things.

For better understanding of the process occurs in the acoustic channel lets take a look at the following figures:

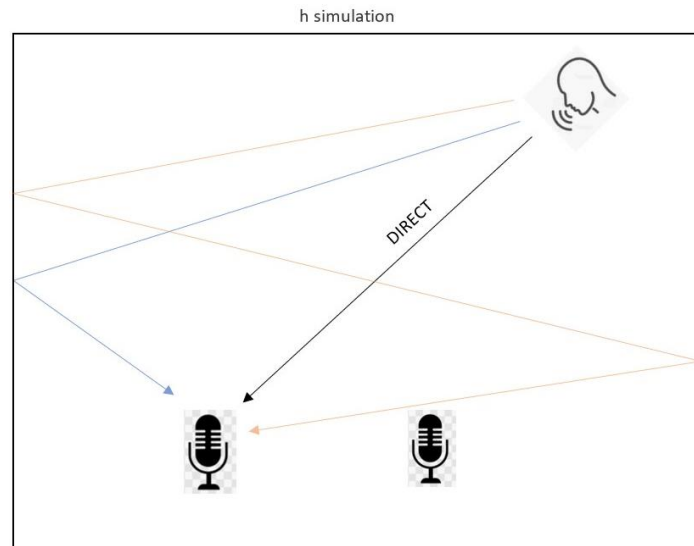


Fig.1

In order to witness the echo effect, we get from the waves that reflect through the wall we've made an experiment to see what it looks like when we clap our hand only one time.

We got the following pulses:

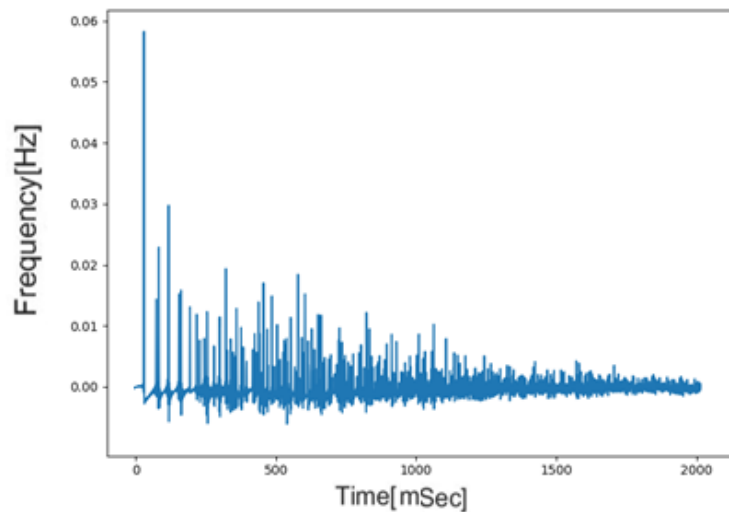


Fig.2

Where the first pulse is the direct wave, and the others are that reflected from the walls and echoed.

Theoretical Background

Short Time Fourier Transform

The Short-time Fourier transform (STFT), is a Fourier-related transform used to determine the sinusoidal frequency and phase content of local sections of a signal as it changes over time. In practice, the procedure for computing STFTs is to divide a longer time signal into shorter segments of equal length and then compute the Fourier transform separately on each shorter segment. This reveals the Fourier spectrum on each shorter segment. One then usually plots the changing spectra as a function of time, known as a spectrogram or waterfall plot.

Continuous-time STFT

in the continuous-time case, the function to be transformed is multiplied by a window function which is nonzero for only a short period of time. The Fourier transform (a one-dimensional function) of the resulting signal is taken as the window is slid along the time axis, resulting in a two-dimensional representation of the signal. Mathematically, this is written as:

$$\mathbf{STFT}\{x(t)\}(\tau, \omega) \equiv X(\tau, \omega) = \int_{-\infty}^{\infty} x(t)w(t - \tau)e^{-i\omega t} dt$$

where $w(t)$ is the window function and $x(t)$ is the signal to be transformed.

Discrete-time STFT

In the discrete time case, the data to be transformed could be broken up into chunks or frames. Each chunk is Fourier transformed, and the complex result is added to a matrix, which records magnitude and phase for each point in time and frequency. This can be expressed as:

$$\mathbf{STFT}\{x[n]\}(m, \omega) \equiv X(m, \omega) = \sum_{n=-\infty}^{\infty} x[n]w[n - m]e^{-j\omega n}$$

Likewise, with the signal $x[n]$ and the window $w[n]$.

After the STFT analysis we get the spectrogram in the Time Frequency domain. The spectrogram is a 3 dimension Graph where the x-axis represents the time, the y-axis represents the frequency and the color represents the power of the signal. An example of a spectrogram is given below:

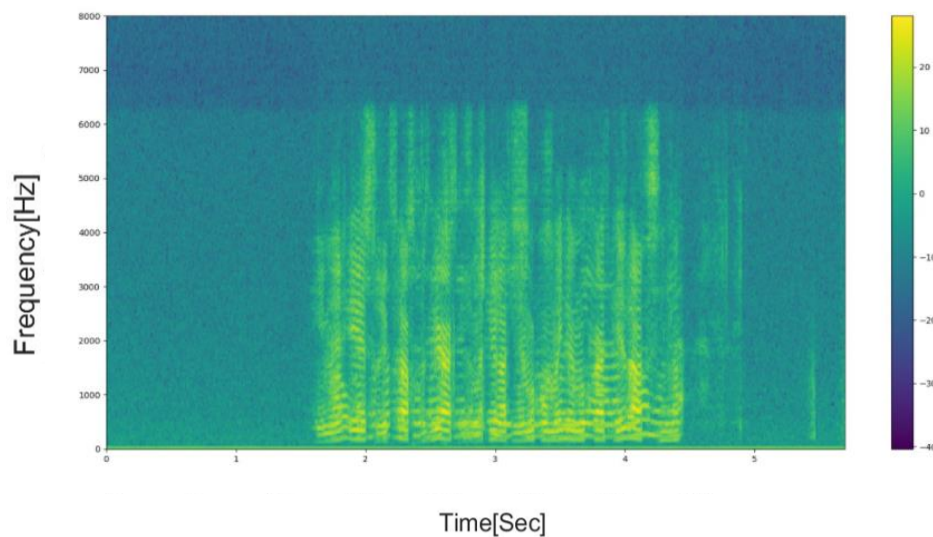


Fig.3

W-disjoint orthogonality

Reminder, our goal is to recover the original sources given only the mixtures.

As we already said, we obtain separation by creating a Mask for each speaker by clustering of TF bins to the various speakers in the scene, and a multiplication of the noisy spectrogram by '1' in TF bins clustered to the desired speaker, and '0' otherwise.

The underline assumption of these masking algorithm is the W-disjoint orthogonality principle.

The W-disjoint orthogonality principle, stating that each TF bin is dominated by a single speaker.

Let's have two signals $x_1(t)$ and $x_2(t)$.

We can say the the signals $x_1(t)$ and $x_2(t)$ are W-disjoint orthogonality if for a given window function, the support of the windowed Fourier transform of $x_1(t)$ and $x_2(t)$ are disjoint sets.

The windowed Fourier transform of $x_n(t)$ is defined as:

$$\mathcal{F}^w |x_n|(\omega, \tau) = \int_{-\infty}^{\infty} W(t - \tau) x_n(t) e^{-j\omega t} dt$$

Denote here as $s_n(\omega, \tau)$. The W-disjoint orthogonality can than be state as:

$$s_1(\omega, \tau) \cdot s_2(\omega, \tau) = 0 \quad \forall \omega, \tau$$

(In practice, however, this equation is rarely satisfied exactly. Instead, the term approximately W-disjoint orthogonal is introduced, which represents the level of orthogonality of sources).

W-disjoint orthogonality is different and in general stronger condition than statistical orthogonality.

As we already mentioned, our separation algorithm based on masking inferred from the speakers direction of arrival.

Mask is basically a matrix with '1' and '0',

Masking involves clustering of TF bins and a multiplication of the TF bins desired by '1' and '0' otherwise.

An example of mask is given below:

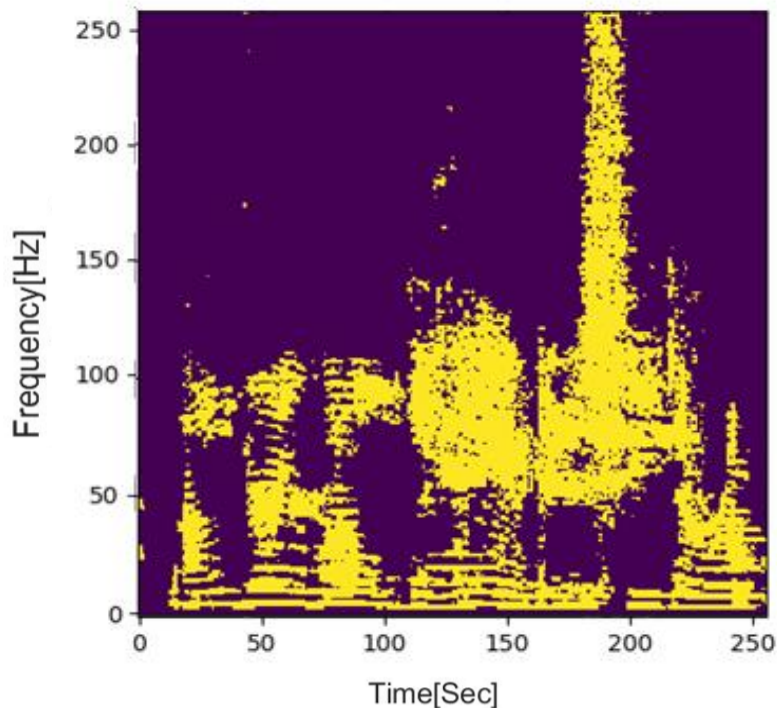


Fig.4

where the x-axis represents the time and the y-axis represents the frequency.

The color yellow represents the TF that have been clustered as '1' and the color purple represent the TF that have been clustered as '0'.

At the end of the process, in order to recover the original sources given the mixtures spectrogram, all we need to do is to multiply the mask with the mixture spectrogram.

For a better understanding of the mask role in the separation process, let's take a look in the next diagram:

Assume we got a mixture STFT from two speaker:

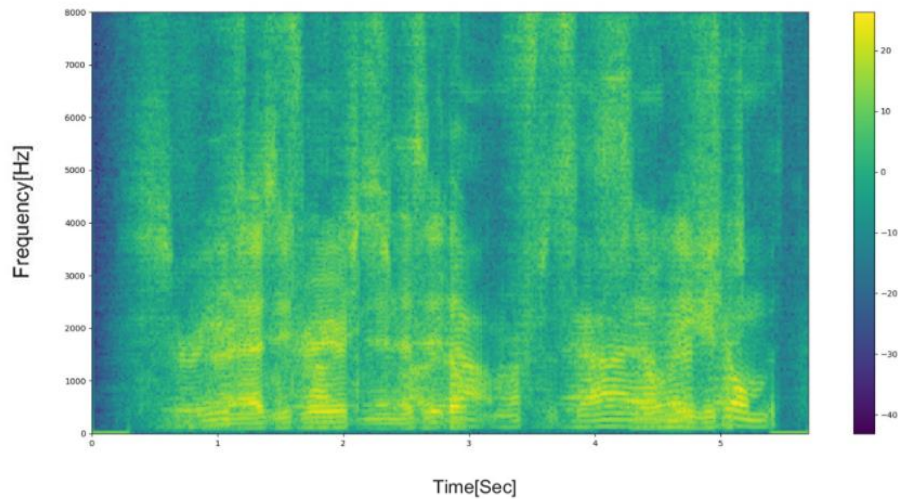


Fig.5

Now, according to the W-disjoint orthogonality we have been able to produce a mask with our U-NET for each of the speakers according to their DOA (details are given in the next sections).

All we need to do is to multiply the mixture STFT with the mask we saw on Fig.4 and we will get one of the speaker's STFT as shown in Fig.6, or in other words, separation.

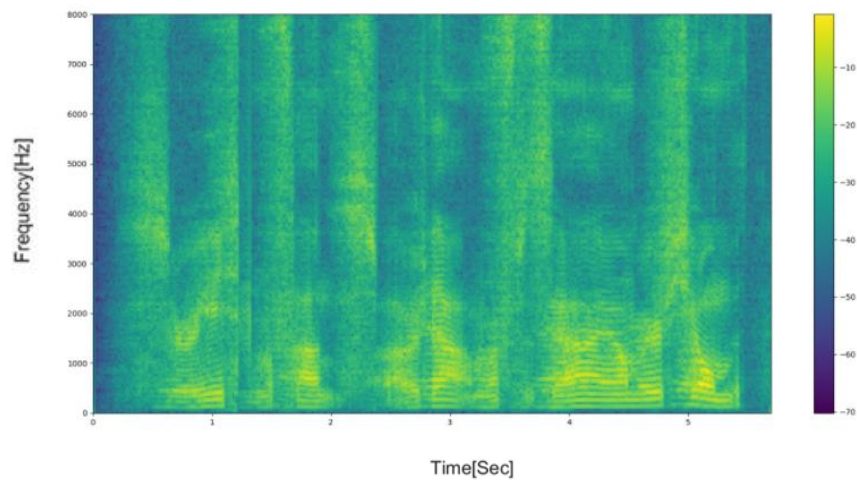
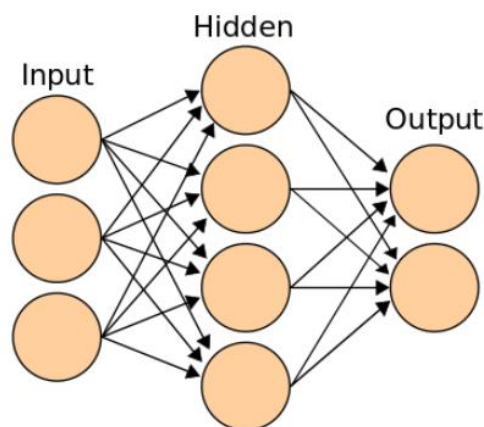


Fig.6

Deep Neural Network (DNN)

A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship. The network moves through the layers calculating the probability of each output.

An ANN is based on a collection of connected units or nodes called artificial neurons. Each connection can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.



Training

Neural networks learn (or are trained) by processing examples, each of which contains a known "input" and "result," forming probability-weighted associations between the two, which are stored within the data structure of the net itself. The training of a neural network from a given example is usually conducted by determining the difference between the processed output of the network (often a prediction) and a target output. This is the error. The network then adjusts its weighted associations according to a learning rule and using this error value. Successive adjustments will cause the neural network to produce output which is increasingly similar to the target output. After a sufficient number of these adjustments the training can be terminated based upon certain criteria. This is known as supervised learning.

Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules.

In our project we applied a deep neural network with a U-net architecture which we will elaborate about in the next chapters.

Direction of arrival (DOA)

One way to calculate the DOA is by finding the angle ' θ ' between the microphone and the speaker.

For better understanding lets simplify our situation by assuming we have only two microphones and one speaker as demonstrate below.

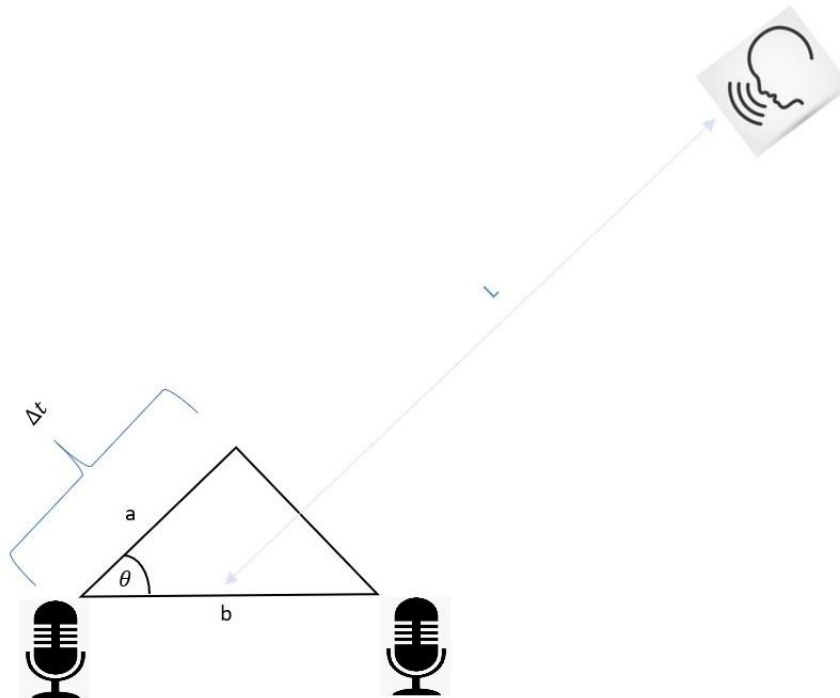


Fig.7

First, we assume that $L \gg b$ hence we have the same angle between the speaker and each one of the microphones.

$$a = c \cdot \Delta t$$

where c is the propagation speed of sound and Δt is the estimated time delay between the first and the second microphone. we use the cross-correlation function to measure when the similarity of the two signals is at the highest value.

Then we can calculate the angle by simple geometry:

$$\theta = \cos^{-1}\left(\frac{a}{b}\right)$$

where b is the distance between the microphones.

Note that this calculation carried on by the DNN itself and we are not doing it actively or training the net to do this particular calculation.

In our project we will work with a circular array of 7 microphones and one more microphone in the middle

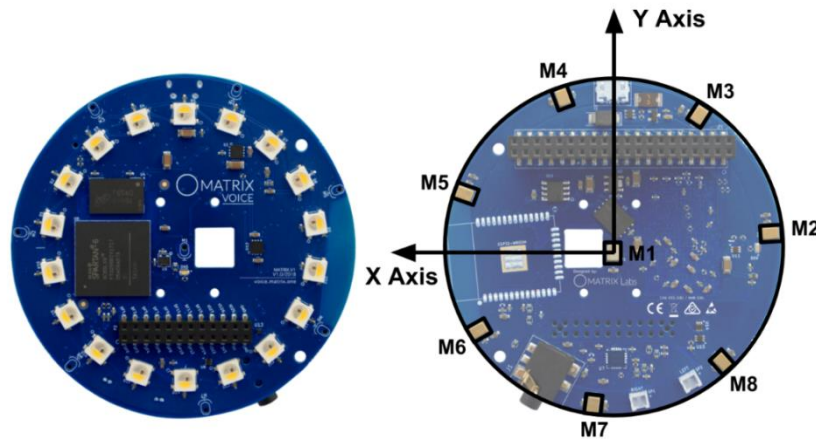


Fig.8

The separation algorithm

We try to estimate the DOA for each TF bin by a neural network and then separate the speakers by grouping these bins according to their estimated DOA.

The neural network uses the microphone signals to estimate the DOA at each TF bin of a given time-frequency image. The network input shape is a $L \times K$ where L is the number of time frames and K is the number of frequency bins.

Instead of feeding the raw microphone signals straight into the net, we decided to preprocess the information in order to help the net and make the learning faster and better. We used the centered microphone in the array as a reference microphone and divided all the other microphones respected to him to get the phase of the instantaneous relative transfer function (RTF) estimate. The phase angle is encoded as a point in the unit circle. The input features to the network, therefore, is an $L \times K$ matrix R where each (l, k) entry has M channels each correspond to a microphone:

$$r(l, k, m) = \left(\cos \left(\angle \frac{z_m(l, k)}{z_{ref}(l, k)} \right), \sin \left(\angle \frac{z_m(l, k)}{z_{ref}(l, k)} \right) \right)$$

As we said, Due to the W-disjoint assumption, the normalized features $r(l, k, m)$ are dominated by a single speaker and hence correspond to a specific DOA. Ideally, the speech contribution to $r(l, k, m)$ is negligible. Hence, it is expected that these are better features than the raw data for DOA estimation.

We form the DOA estimation as a classification task by discretizing the possible angles to be in the set $\Theta = \{0^\circ, 15^\circ, 30^\circ, \dots, 180^\circ\}$.

Let $y_{l,k}$ be a random variable indicating the active direction at bin (l, k) . The target of the network is to infer the conditional distribution of the discrete set of candidate DOAs in Θ for each TF bin, given the recorded signal:

$$p_{l,k}(\theta) = p(y_{l,k} = \theta | R), \quad \theta \in \Theta$$

where R is an $L \times K$ matrix of all the TF bins. The image-to-image DOA prediction task is implemented by a U-net, which details are given in the next section.

After DOA probabilities are set for each TF bin, the next step is to use the estimated DOA to form a mask for each detected speaker in the scene. After running the input to the net we will most of the TF bins will get high probabilities in a set of N specific DOAs Θ_n

The estimated mask of the i -th speaker is the U-net output:

$$M_i(l, k) = \operatorname{argmax}_{\theta_i \in \Theta_n} p_{l,k}(\theta_i)$$

and the absolute value of the i -th speaker signal is reconstructed as follows:

$$|\hat{s}^i(l, k)| = |z_{ref}(l, k)| \cdot \hat{M}_i(l, k).$$

The noisy phase is then used to reconstruct the separated signals in the time-domain, by the application of the inverse STFT. Note, that if a static acoustic scene can be assumed, namely that the sources do not significantly change their DOA during the entire utterance, permutation problems, which are typical to clustering-based approaches, are circumvented. Note that estimating the DOA is modeled here as a classification

problem and not as a regression task. We are not interested in finding the exact DOAs of the speakers in the scenario but rather, grouping them into distinct directions. That is, even with inaccurate DOA estimate, the speech separation can still work, provided that most TF bins are clustered to a mutually exclusive classes.

The U-NET Architecture

As we already said, in our project we apply a deep neural network with a U-net architecture to infer the DOA of each TF bin from a concatenated set of the spectra of the microphone signals.

We train a U-net to classify each TF bin of the multichannel STFT image to one of the DOA candidates.

The Network foundation looks like:

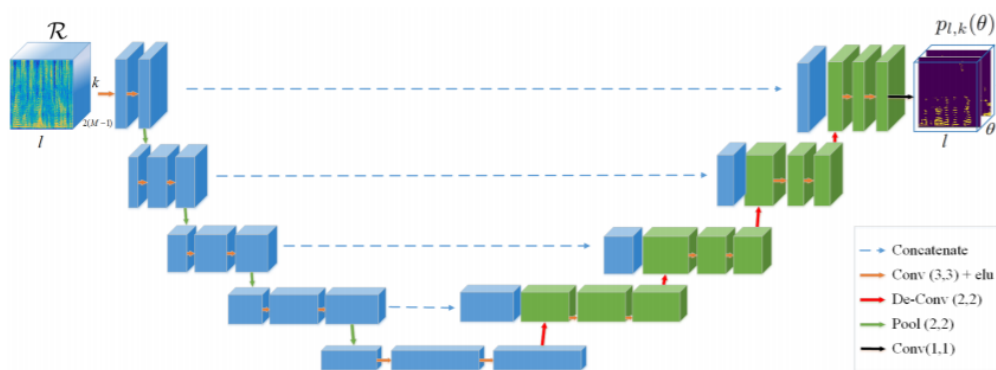


Fig.8

First sight, it has a “U” shape. The architecture is symmetric and consists of two major parts — the left part is called contracting path ("encoder"), which is constituted by the general convolutional process; the right part is expansive path ("decoder"), which is constituted by transposed 2d convolutional layers – a certain technic to upsample the data back to the original size.

Before we start expanding on the encoder and the decoder, let's take a look on some of the more basic elements that are part of the net.

Conv 2d

The input to this layer is three - dimensional data, and the output is three - dimensional data. The important parameters of this layer are 'filters' (the size of the green rectangle), and 'kernel_size' (the size of the dark blue rectangle), the data the layer is applied upon is represented by light blue. In the learning process, the model learns the optimal filter weights values.

Batch Normalization

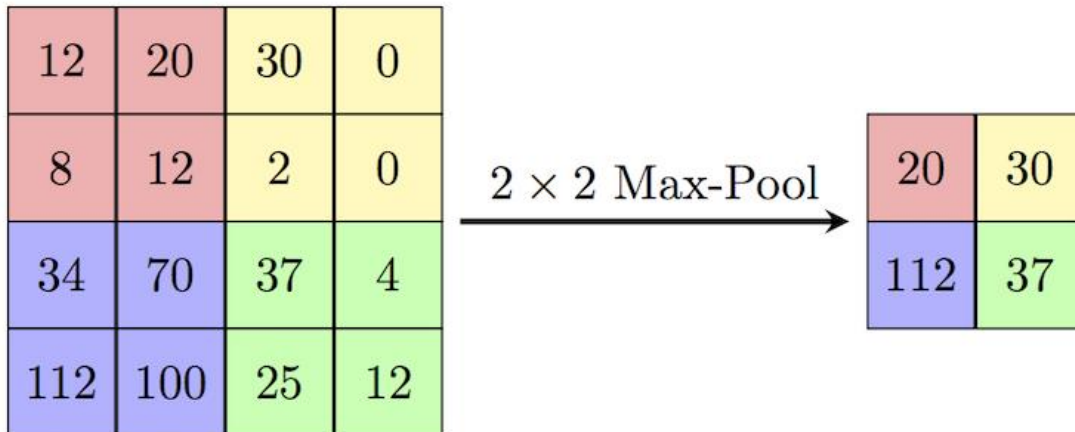
To increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers, that are changing all the time, and get 10 times or more improvement in the training speed. Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift).

Relu

In a neural network, the activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input. The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero.

MaxPool

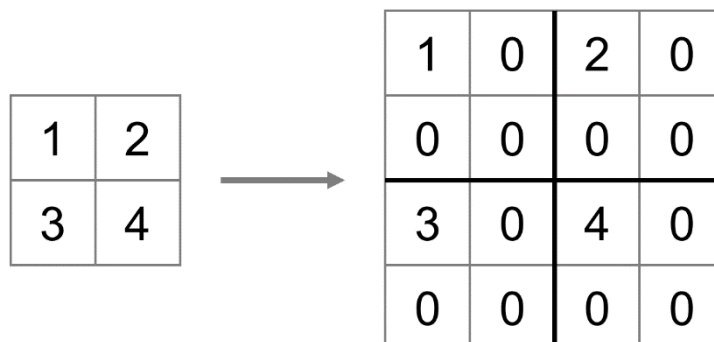
Best described by the picture below, MaxPool2D is a way of data downsampling. Using this layer both takes down computational load and regularize the model.



The most important parameters in this layer are 'pool_size' (which represent the are the max-filter is applied to), and 'strides' (which sets the number of steps taken in each axis).

Upsample

The same way that maxpool down samples the data on the encoder the upsample layer upsamples the data on the way up.



We use mostly in bilinear upsampling. After stuffing pixels with zeros it uses all nearby pixels to calculate the pixel's value, using linear interpolations.

Now that we know more about the layers that networks are made from, we can start looking at the unique architecture of our UNet.

The contracting path ("encoder")

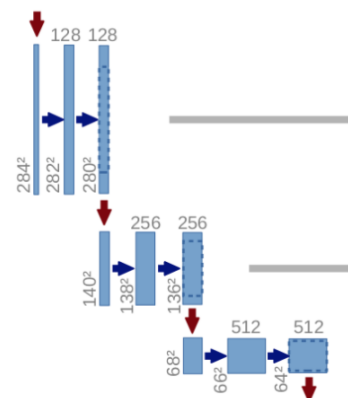
The contracting path follows the formula:

Maxpool → Conv → BatchNorm → Relu → Conv → BatchNorm → Relu

Each step down is made out from those layers.

Notice that each process constitutes two convolutional layers, and the number of channels gets bigger, as convolution process will increase the depth of the image. The red arrow pointing down is the max pooling process which halves downsize of image.

This process is repeated 4 times until the data is "encoded"



The expansive path ("decoder")

In the expansive path, the image is going to be upsized to its original size. The formula follows:

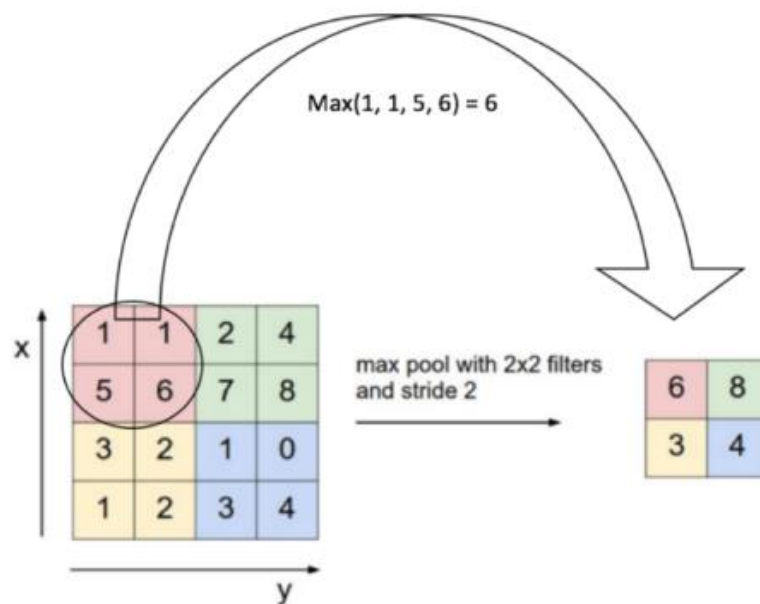
Upsample → Conv → BatchNorm → Relu → Conv → BatchNorm → Relu
→ Concatenate

After the transposed convolution, the image is upsized from $28 \times 28 \times 512 \rightarrow 56 \times 56 \times 216$, and then, this image is concatenated with the corresponding image from the contracting path and together makes an image of size $56 \times 56 \times 512$. The reason here is to combine the information from the previous layers in order to get a more precise prediction.

In the U-NET architecture presented in Fig.8 above, in the encoder part, the input image is squeezed into a bottleneck using 2x2 max pooling operations (down sample). Max pooling is done by applying a max filter to (usually) non-overlapping subregions of the initial representation.

For a better understanding of the max pooling concept let's say we have a 4x4 matrix representing our initial input. Let's say, as well, that we have a 2x2 filter that we'll run over our input. We'll have a stride of 2 (meaning the (dx, dy) for stepping over our input will be (2, 2)) and won't overlap regions.

For each of the regions represented by the filter, we will take the max of that region and create a new, output matrix where each element is the max of a region in the original input.



Now, in the decoder part it is upsample back to the original image shape. The main problem with this architecture is that during the pooling operation padding with zeros (or using

different approaches) will not give us the same image we had before the downsampling. In other words, important local information is lost.

In order to solve this problem, a U-shape architecture was developed. As we can see in Fig.8 The U-net connects between the layers with the same dimensions in the encoder and in the decoder.

In that way we take the information directly from the encoder to the decoder without going through the bottleneck and thus, alleviating the information loss problem.

The output DOA distribution is finally obtained by a softmax layer

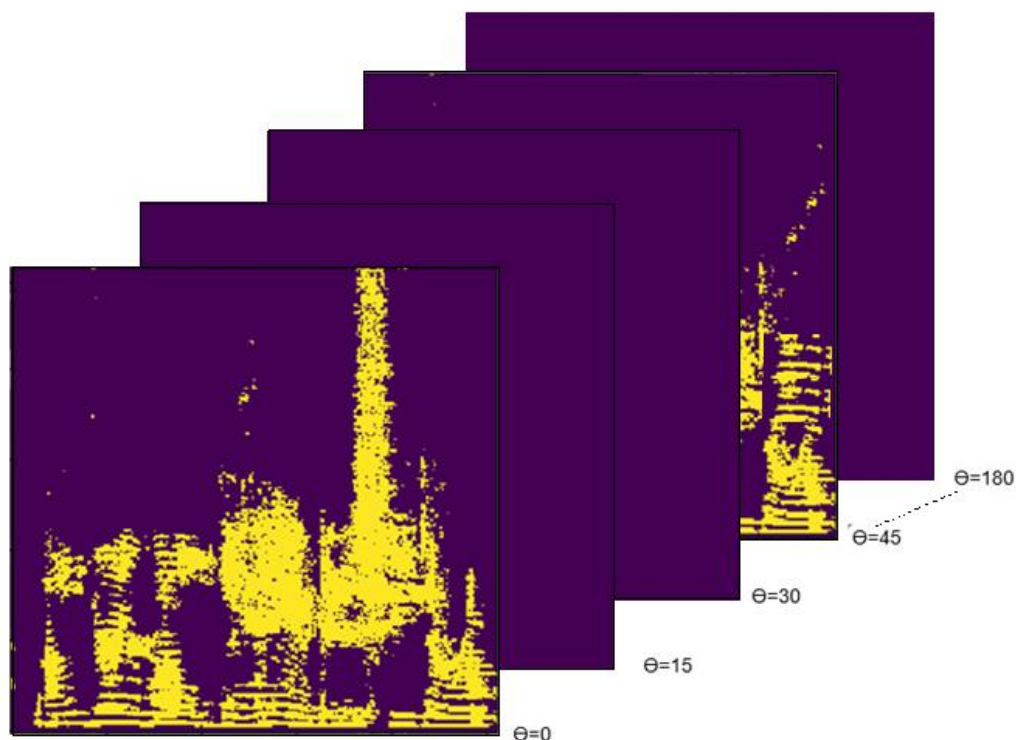
The raw data input is normalized to zero mean and unit variance.

To train the network we use a simulated data as recording of two speakers from "TIMIT" dataset, the angles of the two speakers in the room and the distance from the microphones. In that way we can easily find for each TF bin the dominant speaker and the corresponding DOA.

The network is trained to minimize the cross entropy between the correct and the estimated DOA.

The cross-entropy cost function is summed over all the images in the training set. The network was implemented with Tensor-Flow and training was done using the ADAM optimizer which is optimization algorithm that can be used to update network weights iterative.

The output of the U-net is an array of masks arranged by different DOA as presented in the figure below.



Building The Dataset

In order to get fine results, we need to train our U-NET with optimal examples.

in other words, we need to make enough samples of two speakers in an acoustic room that speak simultaneously while standing in different angles and in different distance from the microphones.

The dataset has been built in three stages:

Stage 0 – randomly choosing audio samples.

We are working with a database called TIMIT. That database contains thousands of audio samples of male and female speakers reading different sentences in different lengths.

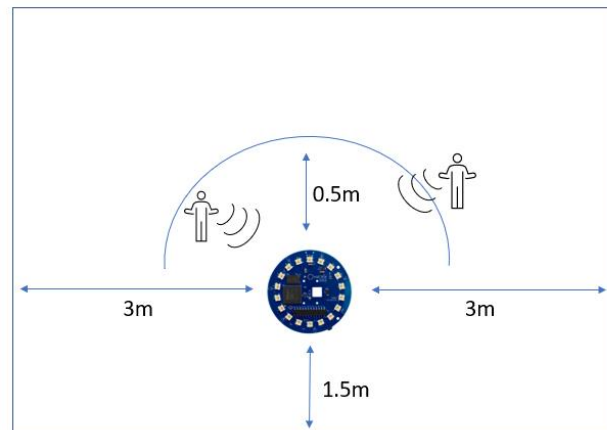
- * 16 speakers from 8 dialect regions
- * 1 male and 1 female from each dialect region
- * total 130 sentences (10 sentences per speaker; note that some sentences are shared among other speakers, sa1 and sa2 are spoken by all speakers.)
- * total 160 sentence recordings (10 recordings per speaker)
- * audio format: wav format, single channel, 16kHz sampling, 16 bit sample, PCM encoding

Stage 1 – RIR generator

As we already mentioned, the room impulse response (RIR) is the transfer function between the sound source and microphone that enables us to control the reflection order, room dimension and the microphone directivity.

We took the exact dimension of the room (distance between the walls and the ceiling height) and the microphones location and simulated it.

The room we simulated is 6x6x3 where the microphone is positioned in at 3x1.5x1.5. The speakers are about 0.5m from the microphone in different angels:



Stage 2 – random angles:

The output of the net is an array of masks arranged by different DOA by angle, in other words the net supposed to tell us the exact angles which the speech signal came from.

In order to train the net to manage this task, in each example we randomly initialized different angles to each one of the speakers. Then we use our rir generator to simulate 2 speakers standing in a room and talking from those angels.

Stage 3 – generating masks:

After we have two speakers talking from different angels in the room in the time domain, we use STFT get our signals in the TF domain. For each TF bin we compare the speakers to see which one is active at that TF, and that is how we create masks for each speaker.

Stage 4 – Our mixed signal:

We add the speakers signals to get our mixed signal, and then divide the mixed signal by its own std to normalize each sample. Then we use stft to get the mixed signal in the TF domain.

Stage 5 – VAD

Voice activity detector. Not all TF bins contains one of the speakers, some TF bins contain silence or maybe even noise. That is why we use something that is called VAD. For each of our generated samples we manually found a bound that every bin louder than that threshold is considered as a speaker. The rest of the bins are classified as Non active bins. We use VAD to make the training easier for the net. In the loss calculation we ignored bins which their VAD is set to '0'.

We do not care if the net will classify non active bin to the right angle because no one speaks at that bin. After marking Non active bins as not active we will tell the net to ignore those bins while training, and the net will have easier job classifying only the Active bins to the right angle.

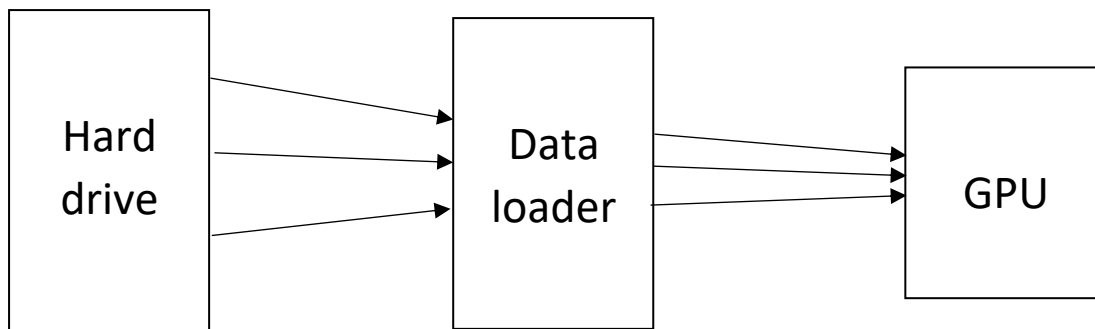
Stage 6 – net input:

The input for the net is the stft from the 8 microphones. First, we divide each track of the microphone by the track of the middle microphone. Doing that leaves each track relative information which make the data more informative and the training process easier.

After that we separate the input to its real and imaginary parts (because the net can't process complex numbers) and normalized each sample.

Data-loader

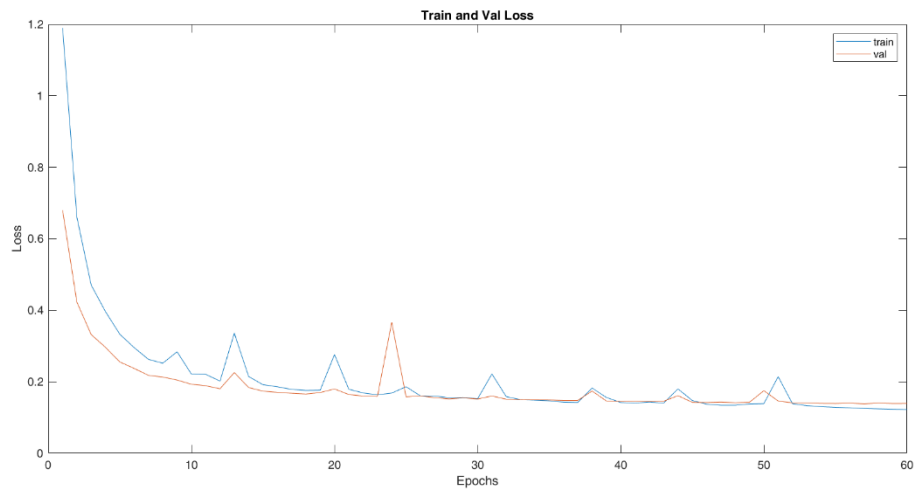
Data Loader is a client application for import or export of data between many common database formats.



Although the CPU has large amount of capacity space, the GPU has advanced calculation ability that accelerates the amount of data a CPU can process in a given amount of time. When there are specialized programs that require complex mathematical calculations, such as deep learning, those calculations can be offloaded by the GPU. This frees up time and resources for the CPU to complete other tasks more efficiently.

Training

We trained our net on 10,000 samples for 60 epochs:



Results

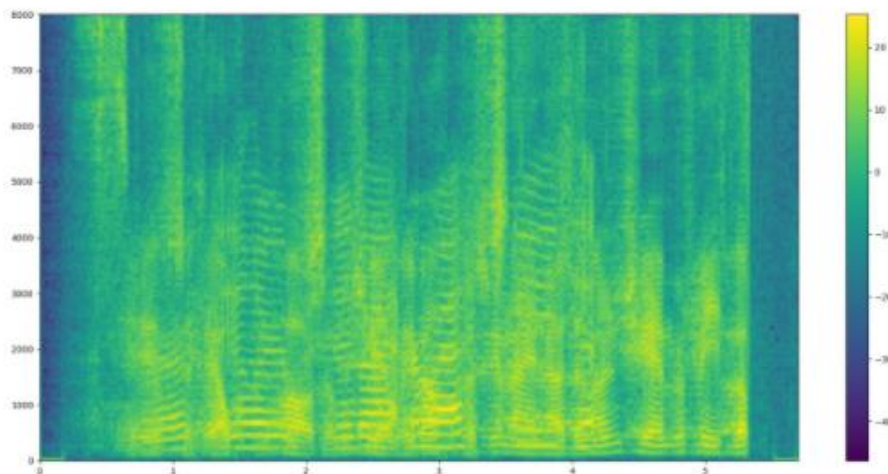
We tested our final net on 2 types of test samples:

Synthesized sample - created by the script we used to create our dataset. The net did not trained on that specific sample.

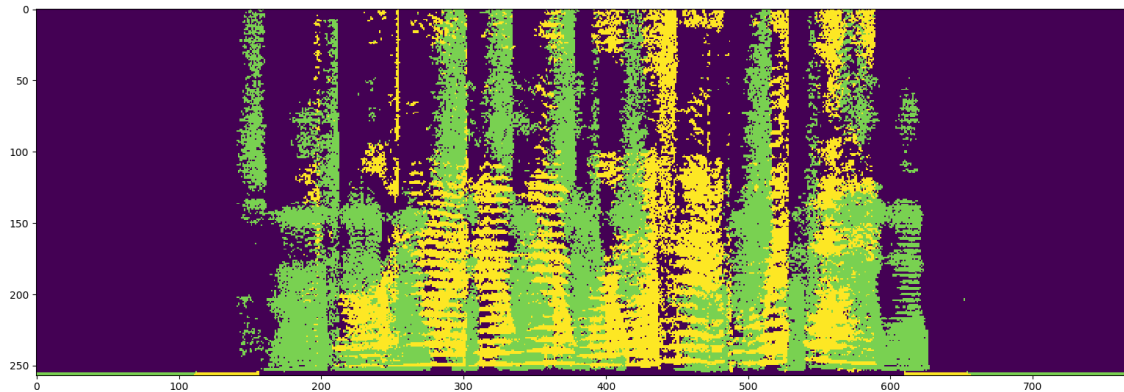
Real sample – we recorded a real sample on the microphone.

Synthesized samples:

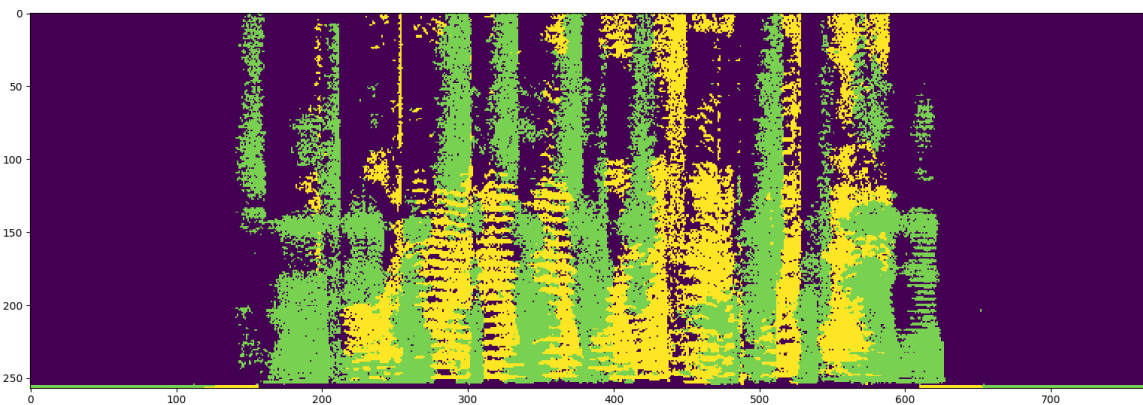
For the first example we took 2 speakers speaking at the same volume. Here is the the combiend stft:



Here are the masks we expected to get:

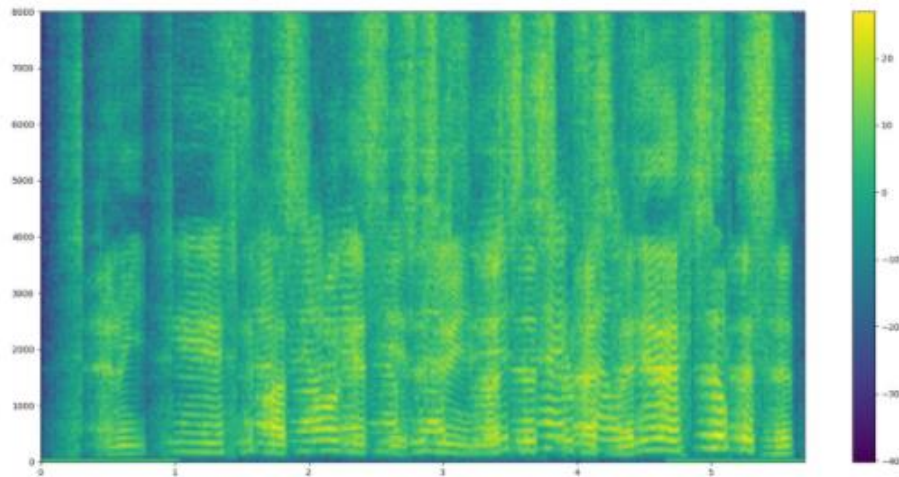


Here are the masks we got:

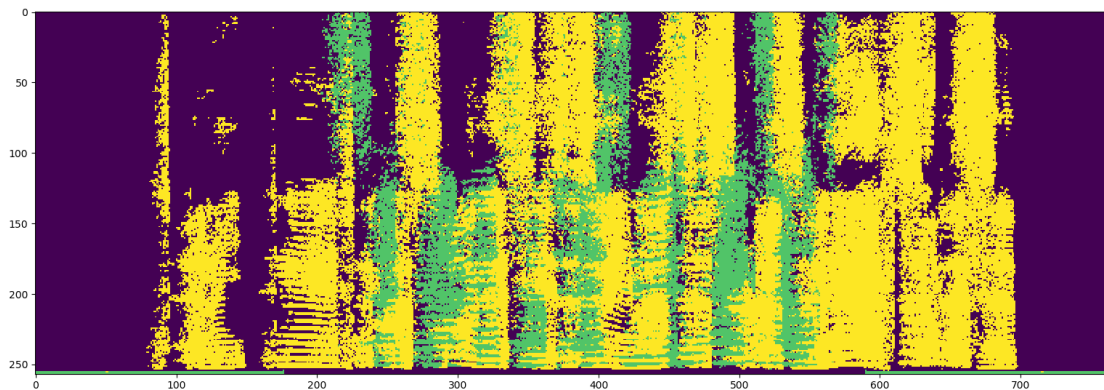


We got very good results, the net predicted the masks almost perfectly and we got a very good audio restoration as well.

We also tested the net on a sample where one speaker is louder than the other. Here is the stft:

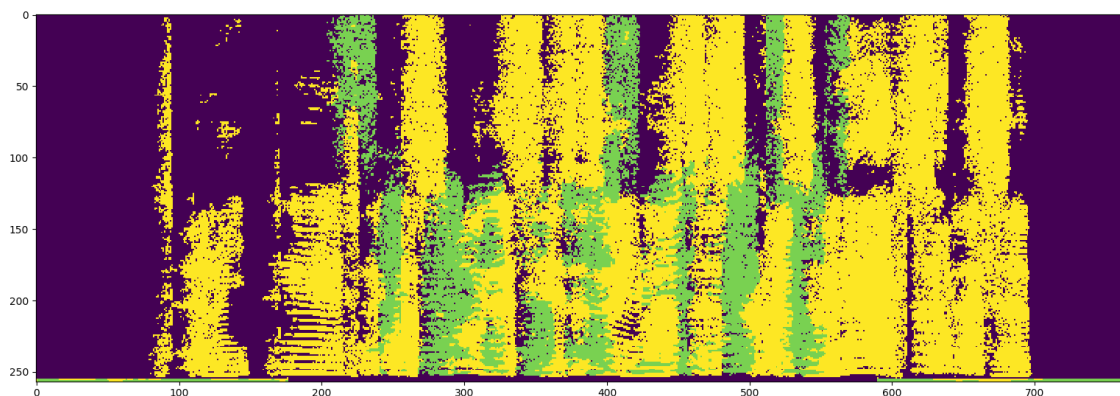


Here are the masks we expected to get:



As you can see, one of the speaker is significantly louder.

Here are the masks we got:

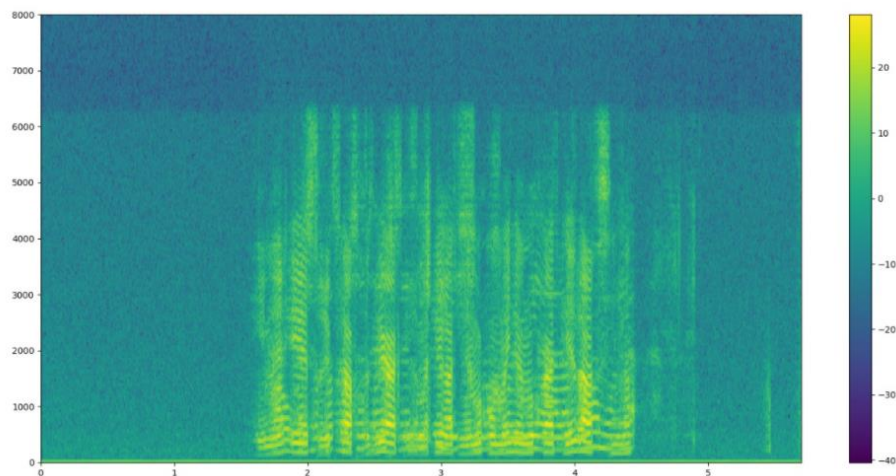


The net predications are still very good, the audio restoration is less clear for the quieter speaker but the separation is still working well.

Real samples:

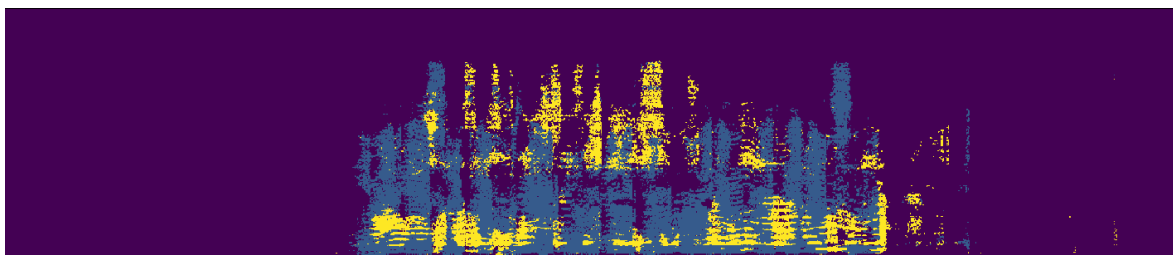
We also recorded real sample using the MARTIX Voice microphone. We recorded the sample in a different room then the one we simulated for the dataset.

Here is the stft:



Since it is a real sample, we don't have a correct predictions.

Here is the masks we got from the net:



The separation is not as clear as it was on the synthesized samples but the audio restoration is good. Each mask does a good job of filtering out the other speaker and leaving enough information so the sentence is clear.

Acknowledgments

This book presents a year long process which included studies and reaserch. It was accomplished using academic knowledge and skills we have obtained throughout our degree.

We would like to express our deep gratitude and admiration to Mr. Shlomi Chazan for the guidance and patience. There is no doubt that without his directions we wouldn't been able to accomplish this ptoject.

We would also like to thank Mr. Pini Tandeitnik for helping and supporting us with any problem we faced throughout this whole process.

Code:

Stft and istft:

```

1.
2. import scipy.io as sio
3. import numpy as np
4.
5. def pure_stft(z,K=512,overlap=0.75,RdB=80):
6.     K_int=K/2+1
7.     K_int=int(K_int)
8.
9.     sub_num=1/(1-overlap)-1
10.    SEG_NO=np.fix(len(z)/(K*(1-overlap)))-sub_num
11.    SEG_NO=int(SEG_NO)
12.    Z=np.zeros((K_int,SEG_NO),dtype=complex)
13.    P=np.zeros((K,SEG_NO))
14.    for seg in np.arange(1,SEG_NO+1):
15.        time_cal=np.arange((seg-1)*K*(1-overlap)+1,(seg-
16.        1)*K*(1-overlap)+K+1)-1
17.        time_cal=time_cal.astype('int')
18.        V=np.fft.fft(z[time_cal]*np.append(np.hanning(K-
19.        1),0))
20.        P[:,seg-1]=np.angle(V).reshape(K,)
21.        time_freq=np.arange(1,K_int+1)-1
22.        time_freq=time_freq.astype('int')
23.        Z[:,seg-1]=V[time_freq]
24.    return Z
25.
26. import scipy.io.wavfile as wav
27. import soundfile as sf
28.
29. import matplotlib.pyplot as plt
30.
31. mat=sio.loadmat('synt_win.mat')
32. synt_win=mat['synt_win']
33.
34. K=512
35. overlap=0.75
36. eps=2.2204*np.exp(-16)
37.
38. def istft(STFT):
39.     long=7*0.008*16000*len(STFT)+512 #missed a 2 factor
40.     long=int(long)
41.     s_est=np.zeros((long,1))
42.     overlap=0.75
43.     K=512
44.     stft_size=int(K/2+1)
45.     for seg in np.arange(1,STFT.shape[1]):
46.
47.         time_cal=np.arange((seg-1)*K*(1-overlap)+1,(seg-
48.         1)*K*(1-overlap)+K+1)-1
49.         time_cal=time_cal.astype('int64')
50.
51.         Ahat=(np.abs(STFT[:,seg-1]).reshape(stft_size,1))
52.         P1=np.angle(STFT[:,seg-1]).reshape(stft_size,1)
53.

```

```

54.
55.     inv_Ahat=Ahat[:, :-1]
56.     inv_Ahat=inv_Ahat[1:stft_size-1]
57.     Ahatfull=np.append(Ahat, inv_Ahat).reshape(K,1)
58.
59.     inv_Ahat_p=P1[:, :-1]
60.     inv_Ahat_p=inv_Ahat_p[1:stft_size-1]
61.     Ahatfull_p=np.append(P1, -inv_Ahat_p).reshape(K,1)
62. #     RR[:, seg-1]=Ahatfull.reshape(K,)
63.
64.     Z2=Ahatfull*np.exp(1j*Ahatfull_p)
65.     r1=np.real(np.fft.ifft(Z2.T))
66.     r2=r1.T*synt_win
67.     s_est[time_cal,:]=s_est[time_cal,:]+r2
68. #     const=max_s/np.max(np.abs(s_est))
69.     s_est=s_est/1.5/np.max(np.abs(s_est))
70.     return (s_est)

```

Creating the data set (helping functions):

```

1. import math
2. import matplotlib.pyplot as plt
3. import sounddevice as sd
4. from scipy.io.wavfile import write
5. import pyrirgen
6. import numpy as np
7. from numpy import diff
8. import h5py
9.
10. from stft_istft import *
11.
12. from random import randrange
13. from scipy.io import wavfile
14. from scipy import signal
15. import os, random
16. import webbrowser
17. import os, random
18. import webbrowser
19. import librosa
20. import scipy.io
21. #####
22. #functions:
23.
24. def rd_path():
25.     basedir1 = "C:\\Users\\USER1\\Spyder
26.     Project\\Week3Database\\TIMIT CD\\TIMIT\\TRAIN"
27.     basedir3 = basedir1 + '\\DR' + str(random.randint(1,8))
28.     random_man = random.choice([x[0] for x in
29.     os.walk(basedir3)][1:])
30.     filesList = [i for i in os.listdir(random_man) if
31.     os.path.isfile(os.path.join(random_man,i)) and
32.     '.WAV' in i and not 'SA' in i]
33.     #This line gets a list of WAV files without SA1/2
34.     from random_man
35.     Thefile = random.choice(filesList)
36.     return (random_man + '\\\'+ Thefile)
37. ###

```

```

34.
35. def rd_pos(num_of_speakers):
36.     num_of_microphones = 8
37.     can = 12
38.     r = np.random.normal(0, 0.1, num_of_speakers)
39.
40.     theta =
41.     np.sort(np.random.randint(1,11,size=num_of_speakers))
42.     while np.amin(diff(theta)) < 2:
43.         theta =
44.         np.sort(np.random.randint(1,11,size=num_of_speakers))
45.         thetal = theta*math.pi/can # Creating the angle in rad
46.         between 0 to pi
47.         R = 0.5 # Average distance
48.         x = 3 + (R-r)*np.cos(thetal)
49.         y = 1.5 + (R-r)*np.sin(thetal)
50.         s = [] # Source position
51.         [x y z] (m)
52.         for i in range(num_of_speakers):
53.             s.append([x[i], y[i], 1.5])
54.         return s,theta
55.     ###
56.
57. def microphons_pos(num_of_microphones):
58.     r =
59.     np.arange(3*num_of_microphones,dtype=float).reshape(num_of_micr
60.     ophones,3)
61.     r[0] = [3, 1.5, 1.5] #
62.     Receiver position [x y z] (m)
63.     r[1] = [3 - 0.03813, 1.5 + 0.00358, 1.5]
64.     r[2] = [3 - 0.02098, 1.5 + 0.03204, 1.5]
65.     r[3] = [3 + 0.01197, 1.5 + 0.03638, 1.5]
66.     r[4] = [3 + 0.03591, 1.5 + 0.01392, 1.5] #
67.     Receiver position [x y z] (m)
68.     r[5] = [3 + 0.03281, 1.5 - 0.01977, 1.5]
69.     r[6] = [3 + 0.005, 1.5 - 0.03797, 1.5]
70.     r[7] = [3 - 0.02657, 1.5 - 0.02758, 1.5]
71.
72.     return r
73.     ###
74.
75. def get_filters(s, r, n_o_s=2, n_o_m=4, fs = 16000):
76.
77.     c = 340 # Sound velocity (m/s)
78.     fs = 16000 # Sample frequency
79.     (samples/s)
80.     L = [6, 6, 3] # Room dimensions [x y
81.     z] (m)
82.     rt = 0.4 # Reverberation time (s)
83.     n = 2048
84.     h =
85.     np.zeros(n_o_s*n_o_m*n,dtype=float).reshape(n_o_s,n_o_m,n)
86.
87.     for j in range(n_o_s):
88.         for i in range(n_o_m):
89.             h[j][i] = pyrirgen.generateRir(L, s[j], r[i],
90.             soundVelocity=c, fs=fs, reverbTime=rt, nSamples=n)
91.

```

```

82.         return h
83.
84.     ###
85.
86.     def get_Signals(n_o_s = 2, fs = 16000):
87.
88.         sec = 1.95
89.         num_of_speakers = n_o_s
90.         signals = []
91.         length = []
92.         for i in range(num_of_speakers):
93.             signals.append(librosa.load(rd_path(), sr=fs)[0])
94.             while signals[i].size < int(1.2*sec*fs):
95.                 signals[i] = librosa.load(rd_path(), sr=fs)[0]
96.             length.append(signals[i].size)
97.
98.         max_l = np.argmax(np.asarray(length))
99.
100.        for i in range(num_of_speakers):
101.            buff = (signals[max_l].size - signals[i].size)/2
102.            signals[i] = np.pad(signals[i],
103.                (int(np.ceil(buff)),int(np.floor(buff))),
104.                'constant',constant_values=(0))
105.
106.        return signals
107.
108.    ###
109.
110.    def get_conv_Signals(n_o_s = 2, fs = 16000):
111.
112.        sec = 1.95
113.        num_of_microphones = 8
114.        num_of_speakers = n_o_s
115.        s_length = int(sec*fs) + 2047
116.
117.        signals = get_Signals(num_of_speakers,fs)
118.
119.        r = microphones_pos(num_of_microphones)
120.        s,theta = rd_pos(num_of_speakers)
121.        h = get_filters(s, r, num_of_speakers,
122.            num_of_microphones, fs)
123.
124.        # in here we are saving the sound signals of the speaker
125.        # after the conv
126.        # sound_mat[n][m] ; n is for speaker ; m is for
127.        # microphone
128.        sound_mat_n = np.convolve(signals[0],h[0][0]).shape[0]
129.        sound_mat =
130.            np.zeros(num_of_microphones*num_of_speakers*sound_mat_n,dtype=f
131.               loat).reshape(num_of_speakers,num_of_microphones,sound_mat_n)
132.        for n in range(num_of_speakers):
133.            for m in range(num_of_microphones):
134.                sound_mat[n][m] =
135.                    np.convolve(signals[n],h[n][m])
136.
137.
138.
139.
140.        buff2 = (s_length - (sound_mat.shape[2] % s_length))/2
141.        f_length = int(buff2*2 + sound_mat_n)

```

```

132.         padded_signals =
np.zeros(num_of_microphones*num_of_speakers*f_length, dtype=float).reshape(num_of_speakers, num_of_microphones, f_length)
133.     for n in range(num_of_speakers):
134.         padded_signals[n] = np.pad(sound_mat[n],
((0,0), (int(np.ceil(buff2)), int(np.floor(buff2))))),
'constant', constant_values=(0))
135.
136.     #return padded_signals, theta
137.     return np.where(padded_signals==0, 0.0001,
padded_signals), theta
138. ###
139.
140. def turn_signals_to_stft(signal_mat):
141.
142.     #signal_mat[n][m][t][1]:n-speakers, m-microphones, t-time
slices
143.     stft_shape =
pure_stft(signal_mat[0][0][0], K=512, overlap=0.75, RdB=80).shape
144.     s_shape = signal_mat.shape
145.
146.     stft_mat =
np.zeros(s_shape[0]*s_shape[2]*s_shape[1]*stft_shape[0]*stft_shape[1], dtype=complex).reshape(s_shape[0], s_shape[2], s_shape[1],
stft_shape[0], stft_shape[1])
147.
148.     for k in range(s_shape[0]): # speakers
149.         for i in range(s_shape[2]): # time slices
150.             for j in range(s_shape[1]): # microphones
151.                 stft_mat[k][i][j] =
pure_stft(signal_mat[k][j][i], K=512, overlap=0.75, RdB=80)
152.
153.     return np.squeeze(stft_mat)
154.
155. ###
156.
157. def turn_stft_to_signal(stft, a, num_of_speakers):
158.
159.     s = []
160.     b = []
161.     d = []
162.     for i in range(num_of_speakers):
163.         s.append((a == i+1)*1)
164.         b.append(stft*s[i])
165.
166.     for i in range(num_of_speakers):
167.         c = []
168.         for j in range(b[0].shape[0]): #time slice
169.             x = istft(b[i][j])
170.             c = np.append(c, x[0:31200])
171.         d.append(c)
172.
173.     return d
174.

```

Creating the dataset (script):

```

1. from rir_funcs import *
2. import wavio
3.
4. def make_data():
5.     fs = 16000
6.     sec = 1.95
7.     num_of_microphones = 8
8.     num_of_speakers = 2
9.     s_length = int(sec*fs) + 2047
10.
11.     signals,theta = get_conv_Signals(num_of_speakers, fs)
12.     signals_final_shape =
np.reshape(signals,(num_of_speakers,num_of_microphones,-
1,s_length))
13.     # signals_final_shape[n][m][t][l]. n - speakers, m -
microphones , t - time slices, l - slice length
14.
15.     ## in here we are saving the mixed sound signal of the
combined speakers
16.     mixed_sound_mat = np.sum(signals,axis=0)
17.     std = np.std(mixed_sound_mat, axis=1, dtype=np.float64)
18.     mixed_sound_mat = (mixed_sound_mat.T / std).T #dividing
the each track by it's own std
19.     mixed_sound_mat =
np.reshape(mixed_sound_mat,(1,num_of_microphones,-1,s_length))
20.     # mixed_sound_mat[1][m][t][l]. m - microphones , t - time
slices, l - slice length
21.
22.
23.     mixed_stft = turn_signals_to_stft(mixed_sound_mat)
24.     speakers_stft = turn_signals_to_stft(signals_final_shape)
25.     speakers_stft = speakers_stft[:, :, 0]
26.     # mixed_stft shape: [t][m][257][256]
27.     # speakers_stft shape: [n][t][257][256]
28.
29.     VAD = 0
30.     mask = np.argmax(abs(speakers_stft), axis=0) + 1
31.     VAD_mask =
(np.log10(abs(np.where(mixed_stft[:,0]==0,0.0001,mixed_stft[:,0
]))) > VAD)*1
32.     a = VAD_mask*mask
33.
34.     target = a - 1
35.     for i in range(num_of_speakers):
36.         target = np.where(target == i,theta[i],target)
37.         target = np.int8(target)
38.
39.     mixed_stft = np.where(mixed_stft==0,0.0001,mixed_stft)
40.     div_mat =
mixed_stft[:,1:num_of_microphones]/np.expand_dims(mixed_stft[:,0],axis=1)
41.     div_mat[np.isnan(div_mat)] = 0
42.     net_input =
np.float32(np.concatenate((div_mat.real,div_mat.imag),axis=1))
43.

```

```
44.         mean =
np.mean(net_input,axis=1)
45.     std = np.std(net_input,axis=1)
46.     final = np.zeros(net_input.shape,dtype=np.float32)
47.     for i in range(net_input.shape[0]):
48.         final[i] = net_input[i] - mean[i]
49.         final[i] = final[i]/std[i]
50.
51.
52.     res_signals = turn_stft_to_signal(mixed_stft[:,0], a,
num_of_speakers)
53.
54.     return final,target
55.
56. #####
#####
57.
58.
59. i = 0
60. while(i < 1995):
61.     stft, target = make_data()
62.     for j in range(0,stft.shape[0]):
63.         h5f = h5py.File('./trainMark8/Mark8sample' +
str(10005+i)+'.h5', 'w')
64.     h5f.create_dataset('Input',data=np.array(stft[j]),compression =
"gzip" )
65.     h5f.create_dataset('Target',data=np.array(target[j]),compressio
n = "gzip" )
66.         h5f.close()
67.         i = i+1
68.     print(i)
69.
```

Training the Net:

```
"""Model_simplerNet.ipynb
```

```
Automatically generated by Colaboratory.
```

```
Original file is located at
```

```
https://colab.research.google.com/drive/1GJPqbWgYtYY0QdglbLKU\_kAu354UmykU
"""
```

```
import numpy as np
import h5py
import matplotlib.pyplot as plt
import time
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils import data
from torch.utils.data import Dataset
from torch.autograd import Variable
import torchvision
from torchvision import transforms, datasets
from sklearn.model_selection import train_test_split
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
device1 = torch.device('cpu')
if torch.cuda.is_available():
    device = torch.device('cuda')    # Default CUDA device
```

```
print(device)
```

```
class My_Dataloader_Train(Dataset):
    def __init__(self):
        super().__init__()
        #self.input = h5py.File('/content/drive/My Drive/Project
Dataset/trainFile5000.h5', 'r')['Input'][0:4250]
        #self.target = h5py.File('/content/drive/My Drive/Project
Dataset/trainFile5000.h5', 'r')['Target'][0:4250]
        self.len = 4250

    def __getitem__(self,i):
```

```

        self.input =
h5py.File('/content/drive/My Drive/Project
Dataset/trainMark8/Mark8sample'+str(i)+'.h5', 'r')['Input'][::]
        self.target = h5py.File('/content/drive/My Drive/Project
Dataset/trainMark8/Mark8sample'+str(i)+'.h5', 'r')['Target'][::]
        X = torch.tensor(self.input)
        Y = torch.tensor(self.target)
        return X,Y

def __len__(self):
    return self.len

class My_Dataloader_Val(Dataset):
    def __init__(self):
        super().__init__()
        #self.input = h5py.File('/content/drive/My Drive/Project
Dataset/trainFile5000.h5', 'r')['Input'][4250:5000]
        #self.target = h5py.File('/content/drive/My Drive/Project
Dataset/trainFile5000.h5', 'r')['Target'][4250:5000]
        self.len = 750

    def __getitem__(self,i):
        self.input = h5py.File('/content/drive/My Drive/Project
Dataset/trainMark8/Mark8sample'+str(4250+i)+'.h5',
'r')['Input'][::]
        self.target = h5py.File('/content/drive/My Drive/Project
Dataset/trainMark8/Mark8sample'+str(4250+i)+'.h5',
'r')['Target'][::]
        X = torch.tensor(self.input)
        Y = torch.tensor(self.target)
        return X,Y

    def __len__(self):
        return self.len

""" Parts of the U-Net model """

class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels,
mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels

        self.double_conv = nn.Sequential(
padding=1),
            nn.Conv2d(in_channels, mid_channels, kernel_size=3,
nn.BatchNorm2d(mid_channels),

```

```

        nn.ReLU(inplace=True),
        nn.Conv2d(mid_channels, out_channels, kernel_size=3,
padding=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True),
    )

    def forward(self, x):
        return self.double_conv(x)

```

```

class Down(nn.Module):
    """Downscaling with maxpool then double conv"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)

```

```

class Up(nn.Module):
    """Upscaling then double conv"""

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the
        number of channels
        if bilinear:
            self.up = nn.Upsample(scale_factor=2,
mode='bilinear', align_corners=True)
            self.conv = DoubleConv(in_channels, out_channels //
2, in_channels // 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels ,
in_channels // 2, kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = torch.tensor([x2.size()[2] - x1.size()[2]])

```

```

diffX = torch.tensor([x2.size()[3] - x1.size()[3]])

x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
               diffY // 2, diffY - diffY // 2])
# if you have padding issues, see
# https://github.com/HaiyongJiang/U-Net-Pytorch-
Unstructured-
Buggy/commit/0e854509c2cea854e247a9c615f175f76fbb2e3a
# https://github.com/xiaopeng-liao/Pytorch-
UNet/commit/8ebac70e633bac59fc22bb5195e513d5832fb3bd
x = torch.cat([x2, x1], dim=1)
return self.conv(x)

class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
kernel_size=1)

    def forward(self, x):
        return self.conv(x)

class simpUNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=True):
        super(simpUNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)
        self.up1 = Up(1024, 512, bilinear)
        self.up2 = Up(512, 256, bilinear)
        self.up3 = Up(256, 128, bilinear)
        self.up4 = Up(128, 64 * factor, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)

```

```

    x = self.up3(x, x2)
    x = self.up4(x, x1)
    logits = self.outc(x)
    #logits = F.softmax(logits, dim=1)
    return logits

DB_T = My_Dataloader_Train()

DB_V = My_Dataloader_Val()

torch.cuda.empty_cache()

simpnet = simpUNet(14,12).float().to(device)

simpnet.load_state_dict(torch.load('/content/drive/My
Drive/Project Dataset/Mark8ParamsMark8.pt'))

train_loader_train = data.DataLoader(dataset = DB_T, batch_size =
32, num_workers=8)
train_loader_val = data.DataLoader(dataset = DB_V, batch_size =
32, num_workers=8)

optimizer = torch.optim.Adam(simpnet.parameters(), lr=0.0001)
#criterion = nn.MSELoss()
criterion = nn.CrossEntropyLoss(ignore_index=-1)
loss = Variable()
min_val_loss = 1000000
train_loader_val__avarage_vector = []

from torch import autograd
torch.cuda.empty_cache()

train_loss_vector = []
val_loss_vector = []

for j in np.arange(20):
    ##### Starting train
    epoch #####
    running_loss = 0

    start = time.time()
    simpnet.train() # setting the model for training mode

    for batch_inx, (specs,masks) in enumerate(train_loader_train):
        #print(f'Loading time is: {start - end}')
        #print(f'Batch number {i} is starting, Loading time is
        {start-end}')

```

```

specs = specs.to(device).float()
specs = specs.view(-1,14,257,256)
outputs = simpnet(specs)
masks = masks.to(device)
#print('Masks shape is: ' + str(masks.shape))
masks = masks.view(-1,257,256)

#print('Output shape is: ' + str(outputs.shape))
#print('Masks shape is: ' + str(masks.shape))
loss = criterion(outputs,masks.long())
#print(f'batch is {batch_inx} loss is: {str(loss)}')
running_loss += loss.item()
optimizer.zero_grad()
loss.backward()
#torch.nn.utils.clip_grad_norm_(simpnet.parameters(), 0.5)
optimizer.step()

del specs
del masks
del loss
del outputs
torch.cuda.empty_cache()

end = time.time()
#print(f'Time is {end-start}')
train_loss_vector.append(running_loss/len(train_loader_train))
#print(train_loss_vector)
##### Finished train
epoch #####

print(f'finished epoch: {j+1}, time was {end-start}')
print("loss of training = " +
str(running_loss/len(train_loader_train)))
#print('validating now...') # validating after each epoch

##### Starting val epoch
#####
simpnet.eval()
with torch.no_grad():
    running_vloss = 0

    for batch_inx, (specs,masks) in enumerate(train_loader_val):
# loop over the validation dataset multiple times

        specs = specs.to(device).float()
        #specs = specs.float()
        specs = specs.view(-1,14,257,256)
        outputs = simpnet(specs)
        masks = masks.to(device)

```

```

        masks = masks.view(-1,257,256)

        vloss = criterion(outputs,masks.long())
        running_vloss += vloss.item()

        del specs
        del masks
        del vloss
        del outputs
        torch.cuda.empty_cache()

    val_loss = running_vloss/len(train_loader_val)
    val_loss_vector.append(val_loss)

    if val_loss<min_val_loss:
        min_val_loss = val_loss
        torch.save(simpnet.state_dict(),'/content/drive/My
Drive/Project Dataset/Mark8ParamsMark9.pt' )
        pass

    print("loss of validation = " +
str(running_vloss/len(train_loader_val))+"\n")
    ##### Finished val epoch
    #####

    print("Finished Training")
    plt.plot(val_loss_vector)
    print("min loss of validation = " + str(min_val_loss))

    fig, ax1 = plt.subplots(1, figsize=(16, 8))

    ax1.plot(train_loss_vector)
    ax1.plot(val_loss_vector)
    ax1.legend(['train', 'val'])
    ax1.set_ylabel('validation accuracy')
    ax1.set_xlabel('Epochs')

    ax1.grid()
    plt.show()

    import pandas
    df = pandas.DataFrame(data={"train": train_loss_vector, "val":
val_loss_vector})
    df.to_csv('/content/drive/My Drive/Project
Dataset/Mark8ParamsMark9Loss.csv', sep=',',index=False)

```