Bachelor Project: Augmenting *pyroomacoustics* with machine learning utilities

MERMET Alexis

June 8, 2018

Contents

Chapter 1

Introduction

1.1 Objectives of the project

During this project, we want to implement new functionalities to the already existing Python library, *pyroomacoustics*[1]. These functionalities include a wrapper to Google's Speech Commands Dataset[2], utilities for augmenting datasets with the already-available room impulse response (RIR) generator, and scripts for evaluating the performance of single and multi-microphone processing for speaker recognition against a pre-trained model.

Before I start, I would like to thank Eric Bezzam for the help he gave me during all the semester, by meeting me every week and helping me really quick when I had a problem. I also would like to thank Robin Scheibler who gave me feedback before my presentations even though he is in Japan and never met me.

1.2 What is *pyroomacoustics*?

First of all *pyroomacoustics* is a library allowing us to make audio room simulation and also apply array processing algorithm in Python. Developed by former and current EPFL undergraduate and graduate students, the goal of this library is to aid in "the rapid development and testing of audio array processing algorithms."[1] There are three core components:

1. Object-oriented interface in Python for constructing 2D and 3D simulation scenarios;
2. A fast C implementation of the image source model for room impulse response (RIR) generation;
3. Reference implementations of popular algorithms for beamforming, direction finding, and adaptive filtering.

Before the start of this project, we could find five main classes in *pyroomacoustics*: The Room class, the SoundSource class, the MicrophoneArray class, the Beamformer class and the STFT class . Quickly after I began working, Robin Schleiber also added a Dataset class that helped me to start creating a wrapper for Google's Speech Commands Dataset (explained below).

With the Room class, you create an object that is a collection of Wall objects, a MicrophoneArray and a list of SoundSource(s). It can be either 2D or 3D. A SoundSource object has as attributes the location of the source itself and also all of its images sources. In general we create this list directly in the Room object that contains the source. Finally the MicrophoneArray class consist of an array of microphone locations together with a sampling frequency.

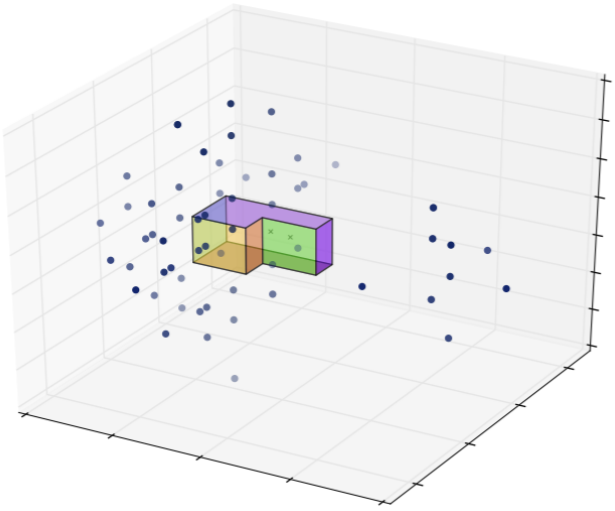


Figure 1.1: Example of a room in *pyroomacoustics*.

1.3 What is TensorFlow?

As they say on their website, TensorFlow is an"open source software library for high performance numerical computation."[3] We followed a TensorFlow tutorial called "Simple Audio Recognition" to create the neural networks we used during the whole project (we are going to explain how it was created in 2.1). We have also reimplemented some of their functions to be able to label sounds we have modified through processing (see 3.2).

Chapter 2

Theoretical knowledge

2.1 Training the Neural Network

In this section we're going to talk about how the neural network was trained. First off all, we download the "Google's Speech Commands"[3] dataset since we need it to train our network but to test the performance of our algorithms. According to the tutorial, this model is considered really simple but is also "quick to train and easy to understand".

This model works as a convolutional network (in fact this model is similar to the one you can use to do some image recognition). First of all a window of time is defined and the audio signal is converted to an "image", i.e. a 2D array, with the Short Time Fourier Transform (STFT). This is done by "grouping the incoming audio samples into short segments and calculating the strength of the frequencies across a set of bands". All the frequency strengths of a given segment will then be treated as a vector of values. These vectors are then ordered according to the time, forming the two-dimensional array known as a "spectrogram"

In the Figure 2.1, time is increasing from top to bottom and frequencies from left to right. We can also see different part of the sound that are probably specific part of it (like a syllable in our case of word).

After our "image" is created we can feed it into a multi-layer convolutional neural network (CNN), with a fully-connected layer followed by a softmax at the end. With a large amount of images and associated labels, we can train our network to classify different words. It took between 16h-20h to train the model and we're going to look at its accuracy later on in this report.

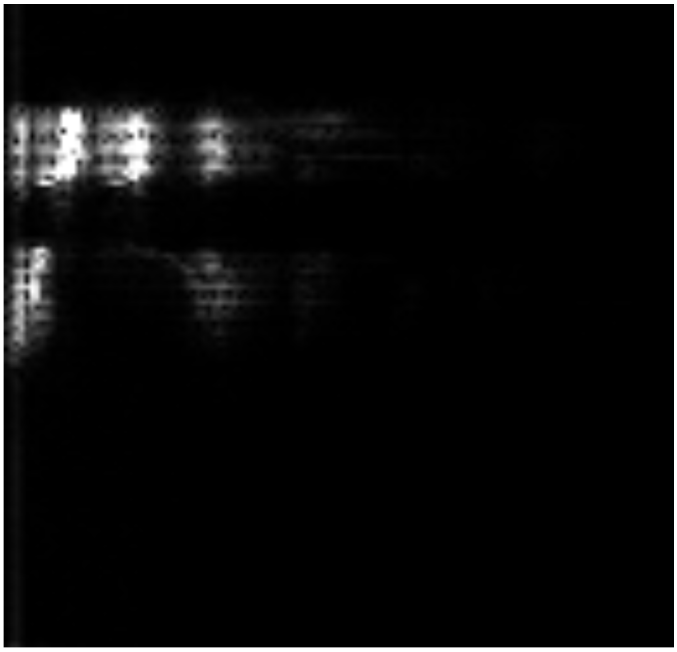


Figure 2.1: Spectrogram of one of our sample during the training (image of the TensorFlow website).

2.2 The GoogleSpeechCommands Dataset: Basic informations

Created by the TensorFlow and AIY teams, the Speech Commands dataset is used to add training and inference in TensorFlow. The dataset contains 65,000 one-second long sound of 30 short words, spoken by "thousands of different people".[2] This dataset is not fixed and will continue to increase with the contribution of users. It is designed to help a user to create his own basic voice recognition interface, with common words like 'yes', 'no', directions, etc. The network explained above is trained to recognize the following words:

1. Yes
2. No
3. Up
4. Down
5. Left
6. Right
7. On
8. Off
9. Stop
10. Go

2.3 Signal-to-noise ratio

In this section, we're going to talk about one of the most important metrics to quantify the performance of signal processing: the signal-to-noise ratio (SNR)[4]. Even though this idea is quite simple and well-known, we talk about it because it is important in our data augmentation and analysis. We are going to talk about it in the "Implementation" part (Chapter 4).

First of all, for a single sensor, considering the signal $y(t) = s(t) + n(t)$, with $s(t)$ being the sound of interest and $n(t)$ being the noise, the SNR is defined as the signal power over the noise power (assuming zero-mean):

$$SNR_{sensor} = \frac{E[|s(t)^2|]}{E[|n(t)^2|]} = \frac{\sigma_s^2}{\sigma_n^2} \tag{2.1}$$

with σ_s^2 being the power of the ignal of interest and σ_n^2 being the power of the noise.
Finally in this project we don't use the SNR under this form but express it in decibels (dB) such that we now have:

$$SNR_{dB} = 10\log_{10}\frac{\sigma_s^2}{\sigma_n^2} = 20\log_{10}\frac{\sigma_s}{\sigma_n} \tag{2.2}$$

(cause $\log_{10} t^2 = 2\log_{10} t$).

2.4 Algorithms

In this section we are going to talk about the different algorithms we used in this project.

2.4.1 Single Channel Noise Removal (SCNR)

SCNR is used to suppress the noise, stationary noise in particular[5]. We consider a noisy input signal $x[n]$ that becomes $X(k, i)$, with i referring to specific time chunk and f to frequency. This conversion to the frequency domain is done using the STFT (Short Time Fourier Transform) for overlapping chunks of audio. The noise suppressor removes the noise by applying a time-frequency-varying real-valued gain filter $G(k, i)$ to $X(k, i)$. We define this gain filter has follow:

- If there is no noise at a given time and frequency, the gain filter has value 1.
- If there is only noise at a given time and frequency, the gain filter has value G_{min}. We choose this value as the one at which the noise shall be attenuated. For example if we want to reduce by 10dB then

$$G_{min} = 10^{-10/20} \tag{2.3}$$

- If there is a mix of signal and noise at a given time and frequency, the gain filter has a value within $[G_{min}, 1]$.

For this algorithm, an estimation of the noise is needed, we have:

$$P(k,i) = E[|X(k,i)|^2]$$

as the estimate of the instantaneous signal + noise power, and we need to compute a noise estimate, $P_N(k,i)$. There is two ways to compute it:

1. If the noise is stationary: $P_N(k,i) = \min P(k,i)$ over some past period of time.
2. Use a voice detector to determine the most recent chuck corresponding to silence + noise: $P_N(k,i) = P(k,i)$ during this silence period.

In our implementation, we chose the first option, looking back for a fixed number of blocks B:

$$P_N(k,i) = \min_{[i-B,i]} P(k,i) \tag{2.4}$$

Now we can define our gain filter such that:

$$G(k,i) = \max[\frac{(P(k,i) - \beta P_N(k,i))^\alpha}{P(k,i)^\alpha}, G_{min}] \tag{2.5}$$

where β is an overestimation factor, often set to a value larger than 1 to ensure all noise is suppressed by a factor of G_{min}, and the exponent α controls the transition behaviour of the gain filter between G_{min} and 1.

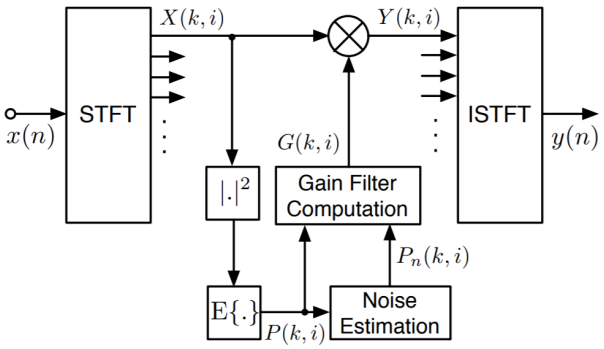


Figure 2.2: STFT-based stationary noise suppressor (from the Audio Signal Processing and Virtual Acoustics books[5]).

In Figure 2.2 you can see a representation of the noise suppressor we have implemented. First the signal x(n) is converted into X(k,i) in the frequency domain using the STFT. Then for each value X(k,i), the circuit compute its power. It is then used as we said before to estimate the noise

(equation 2.6) and create the filters G(k,i) (equation 2.7). Finally we use these filters in the frequency domain by multiplying each X(k,i) by G(k,i)to obtain Y(k,i) that will be then transformed back into the time domain (by using inverse STFT called here ISTFT).

2.4.2 Beamforming: Delay and Sum

In this section, we are going to talk about one of the most basic *beamforming* algorithm[6]. Beamforming tries to perform an intelligent combination of the sensor signals in order to increase the SNR.
If we consider an array of M sensors and define that the signal received at the sensor m is $y_m(t) = s_m(t) + n_m(t)$. We can write $z(t)$ as a weighted sum of $\{y_m t\}_{m=0}^{M-1}$:

$$z(t) = \sum_{m=0}^{M-1} w_m \cdot y_m(t - \Delta_m) = z_s(t) + z_n(t) \tag{2.6}$$

with w_m being the weight corresponding to the signal y_m and Δ_m being a delay to time-align separate microphones so that the signal arrives at the same moment to each sensor.
Now we can write the SNR of our *beamformed* signal $z(t)$ as:

$$SNR_{array} = \frac{E[|Z_s(t)^2|]}{E[|Z_n(t)^2|]} = \frac{|\sum_m w_m|^2 \sigma_s^2}{\sum_m |w_m|^2 \sigma_n^2} \tag{2.7}$$

where we have assumed that the noise is uncorrelated across channels.
Delay and Sum Weights also called Delay And Sum (DAS) looks like what we discussed above. This algorithm takes each individual microphone signal and put all of them in phase by doing a delay correction. Then it sums up the delayed signals and normalized by the number of microphone channels.
If we consider an array of M sensors and define that the signal received at the sensor m is $x_m(t)$ then we define $y(t)$ such that:

$$y(t) = \sum_{m=0}^{M-1} w_m \cdot x_m(t - \Delta_m) \tag{2.8}$$

with w_m corresponding to the weights, used to improve the quality of the recording for the m^{th} signal x_m and Δ_m corresponding to the delay chosen to maximize the array's sensitivity to waves propagating from a particular direction.

Chapter 3

Implementation

3.1 The GoogleSpeechCommands Dataset

The "GoogleSpeechCommands" dataset wrapper was created as a subclass of the "Dataset" class that was already implemented in *pyroomacoustics*. This class will load Google's Speech Commands Dataset in a structure that is convenient to be processed. It has four main attributes:

1. the directory where the dataset is located, the `basedir`.
2. A dictionary whose keys are word in the dataset. The values are the number of occurrences for that particular word in a dictionary called `size_by_samples`.
3. A list of subdirectories in `basedir`, where each sound type is the name of a subdirectory, called `subdirs`.
4. And finally `classes`, the list of all sounds, which is the same as the keys of `size_by_samples`.

There are multiple functions in this class and we're going to review them quickly to give you a general idea of what is possible with this wrapper.

1. we have the `init` function that is the builder of our class. When creating a structure containing the Google Speech Command dataset, the user can choose if he wants to download it or not. But he can also choose if he wants to construct just a subset of the whole dataset at the start.

```
def __init__(self, basedir=None, download=False, build=True, subset=None,
    seed=0, **kwargs):
```

Figure 3.1: The `init` function of a GoogleSpeechCommands structure

2. The `build corpus` function that allows the user to build the corpus with some filters, as for example the list of the desired words to be taken from the corpus.

```
def build_corpus(self, subset=None, **kwargs):
```

Figure 3.2: The `build corpus` function from the wrapper

3. The `filter` function that allows the user to filter the dataset and select samples that match the criterias provided.

```
def filter(self, **kwargs):
```

Figure 3.3: The `filter` function from the wrapper

Now that we talk about the wrapper, we need to present also the "GoogleSample" class that is inheriting from the class "AudioSample" created beforehand in *pyroomacoustics*. This class allows the user to create an audio object to print it in a nice way, plot the corresponding spectrogram, and listen to the file using the *sounddevice* library.[7]

3.2 How to label a file?

In this section, we are going to see how a user can label a file (following the example script available on my *pyroomacoustics* fork called `how_to_label_sa_file`). This example uses the "GoogleSpeechCommand" dataset and also the graph we obtained by training TensorFlow neural network. First of all we rewrote some function of TensorFlow such that we could access the result of labelling. These functions are:

1. `load_graph`, that is loading the graph used to label sounds.
2. `load_labels` loads the labels corresponding to the graph (for example: yes, no, etc...)
3. `run_graph` labels a sound and return the prediction (in percentage)
4. `label_wav`, the main function

In the script, the user needs to specify his label file and his graph file. In our example, he can choose one of the word from the list we saw in Section 2.2 and then label it using the `label_wav` function in the following way:

Here `destination` represents the directory in which the file to label is kept, labels_file the label file, graph_file the graph obtained from TensorFlow and finally word is the sound you except to obtain with this wav file.

```
#label your wavfile
score = label_wav(destination, labels_file, graph_file, word)
```

Figure 3.4: The labelling function in the *pyroomacoustics* example

Later on, this construction will become really important to us cause we will use it to test the efficiency of our signal processing algorithms for speech recognition 3.4.

3.3 How to synthesize noisy signals

Now we will learn how to synthesize noisy signals in *pyroomacoustics*. First of all we have implemented, in utils, two function to create noisy signal:

1. `modify_input_wav`
2. `modify_input_wav_multiple_mics`

We will only talk about the second one since the first function is just a special case of the second function. It can be done with the functions taking care of multiple microphones case (but we keep the first function since its way easier to use it in the single microphone case since you obtain only one noisy signal at the end and not an array of noisy signal you need to flatten). This function will first of all create two rooms, one for the sound source and

```
def modify_input_wav_beamforming(wav,noise,room_dim,max_order,snr_vals,mic_array,pos_source,pos_noise,N):
```

Figure 3.5: the modify_input_wav_multiple_mics functions from *pyroomacoustics*

one for the noise source, of the same dimension as specified by the argument `dimension`. We can separate them since the operations we are going to do are linear and the sound is independent from the noise. Moreover, we would like to separate them so that we can weight *just* the simulated propagated noise according to a specific SNR. After that the room simulation is done, we recover the signals obtained in both rooms. We normalize the noise signal obtained before creating noisy signals for all SNR values given to the function: the argument `snr_vals`.

```
for i,snr in enumerate(snr_vals):
    noise_std = np.linalg.norm(audio_reverb[0])/(10**(snr/20.))
    for m in range(shape[0]):

        final_noise = noise_normalized[m]*noise_std
        noisy_signal[i][m] = pra.normalize(audio_reverb[m] + final_noise)
```

Figure 3.6: How to synthesize the noisy signal for each SNR value

In the figure above, we compute the noisy signal corresponding to each SNR value. The new noise is obtained by multiplying the normalized noise by a coefficient corresponding to a specified SNR value we would like *at the*

microphone(s). We obtain this coefficient from the formula for the SNR (see 2.3).

$$SNR_{dB} = 20 \log_{10} \frac{\sigma_s}{\sigma_n} \tag{3.1}$$

$$\Leftrightarrow \frac{SNR_{dB}}{20} = \log_{10} \frac{\sigma_s}{\sigma_n} \tag{3.2}$$

$$\Leftrightarrow 10^{\frac{SNR_{dB}}{20}} = \frac{\sigma_s}{\sigma_n} \tag{3.3}$$

$$\Leftrightarrow \frac{\sigma_s}{10^{\frac{SNR_{dB}}{20}}} = \sigma_n \tag{3.4}$$

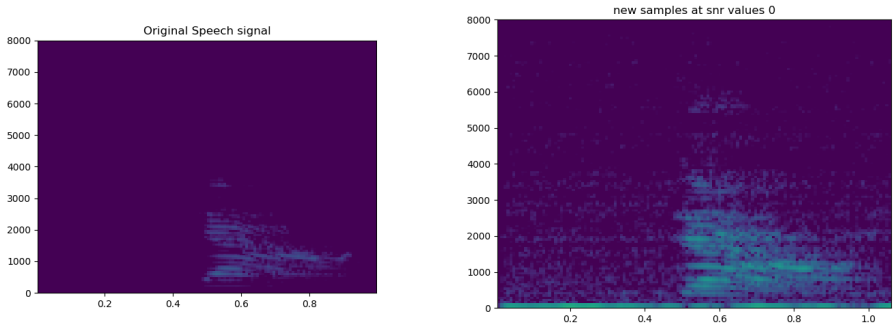
$$\tag{3.5}$$

with σ_n being the square root of the noise's power and also the coefficient we are looking for and σ_s is computed from the simulated source signal. With the obtained coefficient, we can compute the simulated signal with a desired SNR at the microphone(s) as:

$$y(t) = normalized(x(t) + \sigma_n * normalized(n(t))) \tag{3.6}$$

where $x(t)$ the simulated sound signal and $n(t)$ the simulated noise signal for our specified room.

You can see an example of how to use this function in my *pyroomacoustics* fork. It is called `filter`how_to_synthesize_a_signal. In this example, for a word from the "GoogleSpeechCommands" dataset ("no") and an SNR of 20, we obtained the following new noisy signal:



(a) original input signal's spectrogram (b) A noisy signal's spectrogram

Figure 3.7: original input signal's spectrogram compared to new noisy signal's spectrogram

3.4 Testing the performance of the algorithms

We have implemented one algorithm in this project, the Single Channel Noise Removal (SCNR). The Delay and Sum (DAS) was already implemented in the package with lots of other *beamforming* algorithm that incorporate more sophisticated approach but did not use. Nevertheless, the example we provide can be extended to these algorithms by modifying the algorithm used for processing.

3.4.1 SCNR

You can see how we implemented it in Figure 3.8 however we will explain how we did it. First of all we create an STFT object from the *pyroomacoustics* library that allows us to compute the STFT and to use it in an easy and efficient way. We prepare also an array that will contain all of our processed audio at the end.

```
'''
make an STFT object (these class are already implemented in Pyroomacoustics and have example showing how to use them)
'''

hop = fft_len//2
window = pra.hann(fft_len, flag='asymmetric', length='full')
stft = pra.realtime.STFT(fft_len, hop=hop, analysis_window=window, channels=1)

'''
Processing of our noisy signals contained in the noisy array.
'''

# collect the processed block for each of our noisy signal
processed_audio_array = np.zeros(noisy_single_mic.shape)

# we run the algorithm for each of our possible signal
for i, snr in enumerate(snr_vals):

    n = 0
    while len(noisy_single_mic[i]) - n > hop:
        # go to frequency domain
        stft.analysis(noisy_single_mic[i][n:(n+hop)])
        X = stft.X

        # estimate of signal + noise at current time
        P_sn = np.real(np.conj(X)*X)

        # estimate of noise level
        P_prev[:, 1] = P_sn
        P_n = np.min(P_prev, axis=1)

        # compute mask
        for k in range(n_fft_bins):
            G[k] = max((max(P_sn[k] - beta*P_n[k],0)/P_sn[k])**alpha, Gmin)

        # back to time domain
        processed_audio_array[i][n:n+hop] = stft.synthesis(G*X)

        # update step
        P_prev = np.roll(P_prev, -1, axis=1)
        n += hop
# we reset the STFT object
stft.reset()
```

Figure 3.8: SCNR algorithm implemented in *pyroomacoustics*

Then we implement the algorithm that we run for each SNR values given by the user in this case (example called `analyse_improvement_of_single_noise_channel_removal` in *pyroomacoustics*). We set a counter n to 0 at the start of the algorithm. It tells us in which chunk of the STFT we are at a given moment. After that, we enter in the while loop where we use the STFT object to our advantage. With it we can compute easily the STFT of our input signal (a noisy signal). Then we fill our matrix containing all the previous signal's power estimation and we select its minimum value as our noise's power estimation. Having this estimation we can now compute the value

of our filter, called mask in the figure, using the formula (2.6) (see 2.4.1). Finally we have to update our matrix containing the powers and our counter n before repeating the actions above.

3.4.2 DAS beamforming

Even though we have not implemented the algorithm itself, we have used the DAS algorithm as you can see in Figures 3.9 and 3.10. We have create a function working as the one presented in `How to synthesize noisy signals` as you will see below (see 3.3). In this function we directly compute the beamformed signal that then we can use in a script looking like the one presented before (see 3.2) to label it.

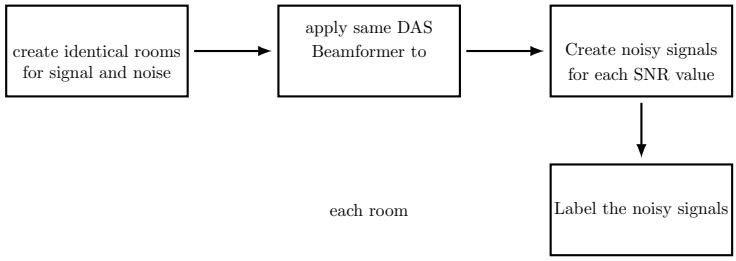


Figure 3.9: schema of how our beamforming script works

```
norm_fact = np.linalg.norm(noise_reverb[-1])
noise_normalized = output_noise / norm_fact

#initilialize the array of noisy_signal
noisy_signal = np.zeros([len(snr_vals),np.shape(output_signal)[0]])

for i,snr in enumerate(snr_vals):
    noise_std = np.linalg.norm(audio_reverb[-1])/(10**(snr/20.))
    final_noise = noise_normalized*noise_std
    noisy_signal[i] = pra.normalize(pra.highpass(output_signal + final_noise,fs_s))

return noisy_signal
```

Figure 3.10: how to create beamformed signals (utils package)

Indeed we create two rooms, one containing the signal and the other one noise. In these room, we have a circular array of microphone plus one placed at the centre of this circle. We simulate these two rooms before applying DAS on both of their output (since this algorithm is linear we can do so). Then we normalize our beamformed noise to be able to set the SNR according to a desired value. To do so we use the norm of the noise at the center microphone. Then we create our SNR factor using the norm of the center microphone in the signal room. Finally we compute our noisy signal as before (see 3.3).

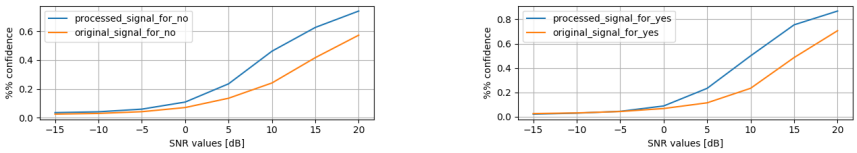
Chapter 4

Results

In this section we will look at the performance of the two algorithms when we try to label processed sound. First we will talk about SNCR and then about DAS.

4.1 Analyse improvement of Single Channel Noise Removal

We first tried the SNCR algorithm in a script called: `analyze_improvement_of_single_noise_channel_removal_fulldata` available in my *pyroomacoustics* fork. We ran it for subset of size 25 in the GoogleSpeechCommands dataset and for the words "no", "yes", "stop" and "up" that the TensorFlow neural network can recognize. This means we have created 1750 samples per word and 25 samples per SNR for a given word. We obtained the following results for the the word "no" and "yes":



(a) classification of the word no (b) classification of the word yes

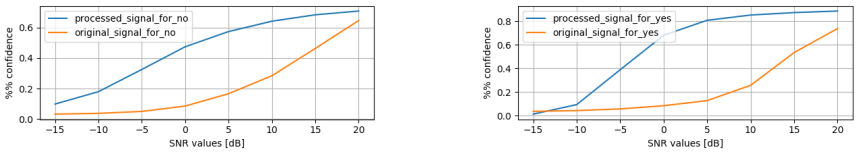
Figure 4.1: classification comparison between the original noisy signal and a processed version of it using SNCR for samples of size 25

We can see in the case of the word "no" that the algorithm improved the

recognition for low SNR (SNR that are the most plausible in real life). For example at a SNR of 10dB we have an improvement of 23% for the classification which is huge (we go from 25% of correct recognition by the model to 48%). We can also see that the recognition is always higher when we are using a processed signal. The result of this test is even more impressive when we are working with the word "yes" since we have an improvement of the classification of 29% at a SNR of 10dB (we go from 23% of correct recognition by the model to 52%).
We can conclude that the SNCR is a good algorithm for cleaning the data before giving it to a classifier. The improvement for low SNR are always close or above 20%. But we can't say this is perfect. Even if the algorithm is quick, we can see that we don't achieve more that 50% of correctness in a majority of the case for low SNR. This is not that good if we want to use this algorithm for a vocal recognition system.

4.2 Analyse improvement of Beamforming

Now we will look at the efficiency of our DAS algorithm. We use the script `analyze_improvement_of_beamforming_fulldata` available in my *pyroomacoustics* fork. We ran it for subset of size 25 in the GoogleSpeechCommands dataset and for all the word that the TensorFlow neural network can recognize. This means we have created 1750 samples per word and 25 samples per SNR for a given word. We obtained the following result:



(a) classification of the word no (b) classification of the word yes

Figure 4.2: classification comparison between the original noisy signal and a processed version of it using DAS for samples of size 25

In the case of the word "no" we can see that the algorithm improved a lot the recognition for low SNR. At 0dB we have an improvement of 30% for the classification. This is even better than the improvement of SNCR at higher SNR! In this case we go from 10% of correct recognition by the model to 48% (nearly 50%). Also the processed signal is always better recognize by

the model. We have even better result when we are working with the word "yes". We have in this case we have an improvement of 58% of the classification at a SNR of 10dB (we go from 10% of correct recognition by the model to 68%).
Now we can say that the DAS is an even better algorithm than the SNCR for cleaning the data before giving it to a classifier. The improvement at low SNR seems to be higher than 30% in general. One more time this algorithm is not perfect (what we want is more than 50% of good recognition at low SNR). But seeing its performance and knowing that DAS is the simplest form of Beamforming, we could say that beamforming seemed to be one solution to improve efficiently the vocal recognition of a system.

Chapter 5

Conclusion

5.1 Where are we now?

At the end of this project we were able to add a wrapper to Google's Speech Commands Dataset that is actually available in *pyroomacoustics* at the address `https://github.com/LCAV/pyroomacoustics/tree/master/pyroomacoustics/datasets` since we have done the pull request for it already(`https://github.com/LCAV/pyroomacoustics/pull/30`). We were able to add utilities for augmenting datasets and to test the efficiency of processing algorithm for improving the recognition's quality of a system. We have seen that SNCR is working quite well for what we are doing right now but we can surely do much better (but we will talk about this at the end). DAS beamforming also improved the recognition, but it would be interested to test the performance of more sophisticated algorithms. All the scripts we have done are now available on the GitHub of *pyroomacoustics* (my fork in fact) at the address `https://github.com/alexismermet/pyroomacoustics/tree/master/examples/final_scripts_for_final`.

5.2 What's next?

In the future and for the improvement of the *pyroomacoustics* library, other students could test and implement other signal processing algorithm (for example just test the other beamforming algorithms already available in the package). They could also create new wrapper for other datasets. Then they could also create a new dataset from scratch for *pyroomacoustics* which could help user to test their algorithms in the best condition possible (also it could be quite fun to record sound). Finally, they could try to find another machine learning model with better accuracy than the model we have used in this project.

Bibliography

[1] Robin Scheibler, Eric Bezzam, and Ivan Dokmanić. "Pyroomacoustics: A Python package for audio room simulations and array processing algorithms". In: *arXiv preprint arXiv:1710.04196* (2017).

[2] Google Brain Team Pete Warden Software Engineer. *Launching the Speech Command Dataset*. Ed. by Google AI Blog. 2017. URL: `https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html`.

[3] TensorFlow, ed. *Simple Audio Recognition*. 2018. URL: `https://www.tensorflow.org/versions/master/tutorials/audio_recognition`.

[4] Jørgen Grythe. *Array gain and reduction of self-noise*. report. Norsonic AS, Oslo, Norway, 2016.

[5] Christof Faller and Dirk Schröder. *Audio Signal Processing and Virtual Acoustics*. 2015.

[6] Robin Scheibler, Ivan Dokmanić, and Martin Vetterli. "Raking echoes in the time domain". In: *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE. 2015, pp. 554–558.

[7] Matthias Geier. *Play and Record Sound with Python*. 2018. URL: `https://python-sounddevice.readthedocs.io/en/0.3.11/`.