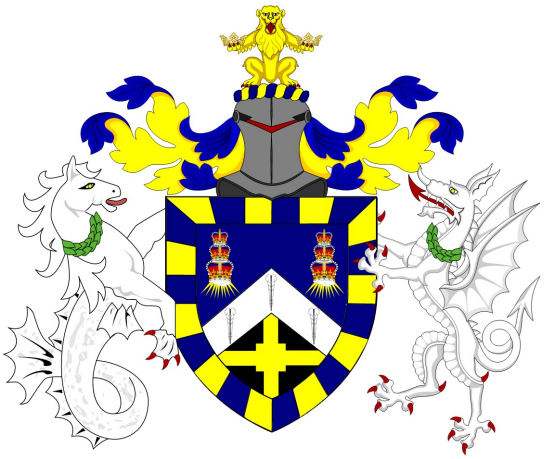
Sound and Music Computing MSc Dissertation ECS750P, 2018/19

# Real-time application of Denoising Autoencoders


**Franks Vladimir Chavez**

**180637487**

Supervisor: Prof. Dr Gyorgy Fazekas

A thesis presented for the degree of

Master of Science in *Sound and Music Computing*


School of Electronic Engineering and Computer Science

Queen Mary University of London

# Acknowledgements

For this project, I want to thank my tutor for the mentoring and technical support. I also want to thank my family for the motivational help they offered me.

# Abstract

Audio denoising is an ongoing challenge for the Machine Listening field. The proposed project aims to deliver a neural network based denoiser able to be called from a real-time environment like a VST plugin.
The report will explain how the denoising research has been carried out and how the obtained Python denoiser can be ported on to C++. Detailed results analysis under both statistical and psychoacoustical point of view are provided. Finally, a set of future improvements with related pros and cons are provided.

# Contents

# Chapter 1

# Introduction

## 1.1 Overview

This report aims to describe how a self made Neural Network-based de-noiser has been implemented as an audio plugin for real time applications.

## 1.2 Motivation

At the actual stage, there are no instrument-specific denoiser plugins on the mainstream market: all the analysed products behave under a "generic" denoising approach.
Another point to consider is the typology of solutions implemented by such algorithms: most of them use DSP based methods in either the frequency and time domains [4] (more datails in 2.1). A low budget AI based denoiser would be very useful in many audio technology based scenarios (both studio and live) due to the its ease of use and the un-required noiseprint training.

## 1.3 Objectives

Follows a list of goals the project aims to accomplish:

- Develop a classifier between noisy signal frames and noisy empty frames.
- Deploy an optimised denoising autoencoder model for the noisy signal frames.
- Train the model for different sorts of input instruments.
- Train the model of each instrument at different noise levels for improved specialisation.
- Implement the denoising models on a user friendly plugin.

# Chapter 2

# Background and Research

This chapter lists and briefly explains the required theoretical concepts needed to undertake the proposed problem.

## 2.1 Noise

This project aims to assess a quite common issue that can be found in the audio-related industry: noise [1][2]. Generically, by noise, we define any unwanted sound, however, by an engineering approach, noise refers to random fluctuations of data along the time axis.
In audio-recording applications, noise might arise in different cases. There are two main situations where noise might appear: interference-caused noise and gain staging provoked noise. The first type arises when to the original sound, an interference noise is involuntary added, this might happen when cables are not properly shielded or ground loops are created by badly managing the power supply units of the recording outboard. The second type of noise arises when an improper gain staging setting is applied to the audio chain: it might happen that different devices connected to each other do have way different SNR values. Such condition might lead to noisy or clip situations.

The classic ways to prevent the first typology of noise is by using balanced audio signals (TRS cables [3]) and shielding, for the second issue improving the gain stage settings of the audio chain or replace the components with a tight SNR.
It might happen that, inside a recording studio environment, this precautions are not taken too much into consideration, therefore, in order to save the recordings, some de-noising work must be done in post-processing.
By quoting Adnan Quadri [4], De-noising algorithms rely on few well-known techniques, where the main are: wavelet transform based de-noising, both standard and non-negative matrix factorisation, adaptive filter based techniques and least mean square algorithms.
The main goal of this project is to create a method which is capable of differentiate between stochastic distributions of data and actual signals containing some sort of relevant information. Ideally, the software should work in real time and be as optimised as possible under both performance and user-experience points of view.
The proposed method relies on a computer-vision based approach which manipulates the frequency domain transform of a given noisy signal. Mostly based on the work proposed in [15].
Before starting with the proposed method, we must discuss what frequency domain transforms are and see what particular features we might be interested in in order to tackle the noise problem.

## 2.2 Visualisation of Time and Frequency domain signals

In order to better understand what the effects of added noise are we must first graphically visualise the effects generated by the noisy interference.
We start by discussing the time vs amplitude representation. Like perceived naturally, we can represent any audible signal as an air wave fluctuating in amplitude over time.

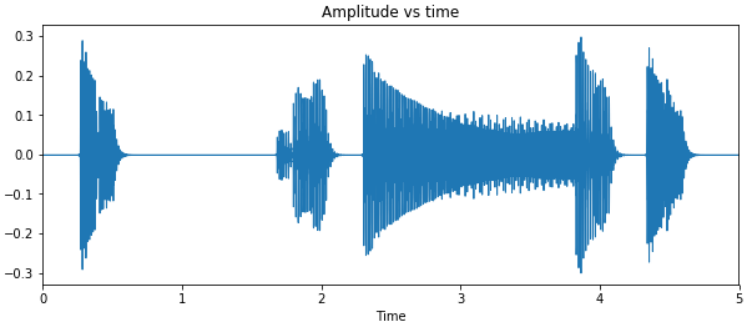



Figure 2.1: 5 seconds of clean bass recording showed as amplitude vs time

As we can see, the provided 5 seconds signal does not contain any evident quantitative of noise. The next figure shows how the noisy version of the previously given signal will look like:

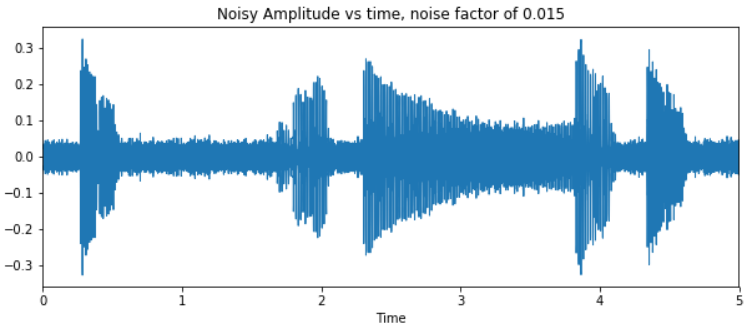



Figure 2.2: 5 seconds of noisy bass recording showed as amplitude vs time

As we can see from the given figure, by adding noise, disruption of the signal happens: silence parts start being disrupted as well as the crests of the audio wave.

We now introduce the Discrete Fourier Transform [5][6], a finite sum which goal is to decompose any given discrete amplitude vs time signal into a decomposition of both cosines and sines (frequency domain). These two sets of sinusoids represent both the real and imaginary parts of the decomposition in the frequency domain.

It is Given by the formula:

$$x(k) = \sum_{n=0}^{N-1} x(n) exp(\frac{-jnk2\pi}{N})$$

We now have a look at how the DFT of 2.1 will look like:

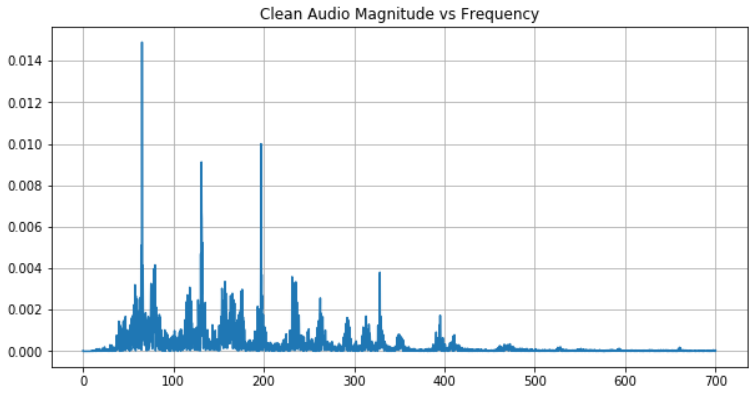



Figure 2.3: Real part of the DFT of 2.1

As shown by the figure, the bass recording contains high magnitude where its harmonics are, as the recording is quite clear, the overall noise floor is very little. We now introduce the frequency domain representation of the same signal but with added noise:

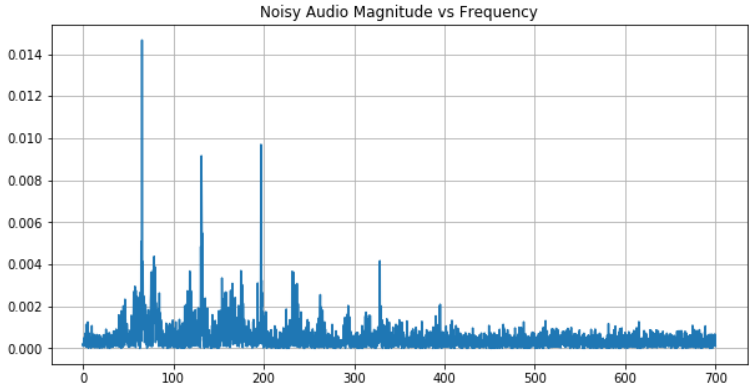



Figure 2.4: Real part of the DFT of 2.2

As shown before, the DFT highlights the main components of the bass recording, although, this time the overall noise floor is consistently higher. As our audio recording (2.1) changes significantly over time, the DFT representation of such signal will speak for its entire length, giving little relevance to the silence parts as well as to the quickly played notes.

We therefore might want to show a close series of DFTs during time in order to display the evolution in magnitude of any signal. The Short Time Fourier Transform (STFT) [7][8][9] comes into rescue, in few words, the way it works is to divide a given signal into frames and take the DFT of such frame. Mathematically, the STFT is given by:

$$X_m(\omega) = \sum_{n=-\infty}^{\infty} x(n)w(n - mR)exp(-j\omega n)$$

where:
x(n) = input signal at time n
w(n) = length M windowing function
$X_m(\omega)$ = DTFT of windowed data centered about time mR
R = hop size, in samples, between each single DTFT
The next figure shows the STFT of signal presented in 2.1 and 2.2:

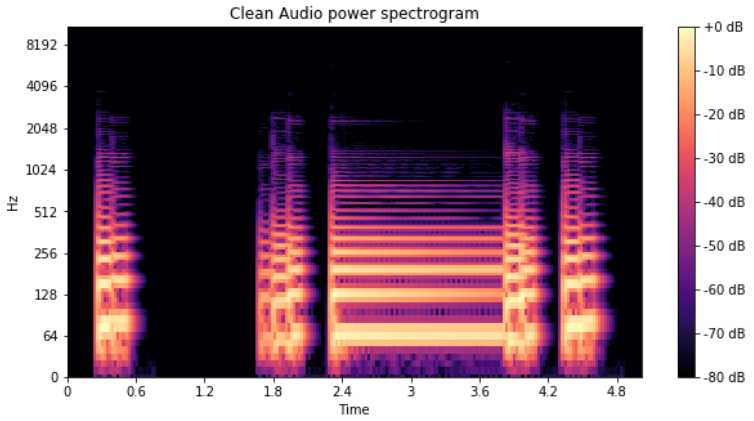



Figure 2.5: STFT of 2.1

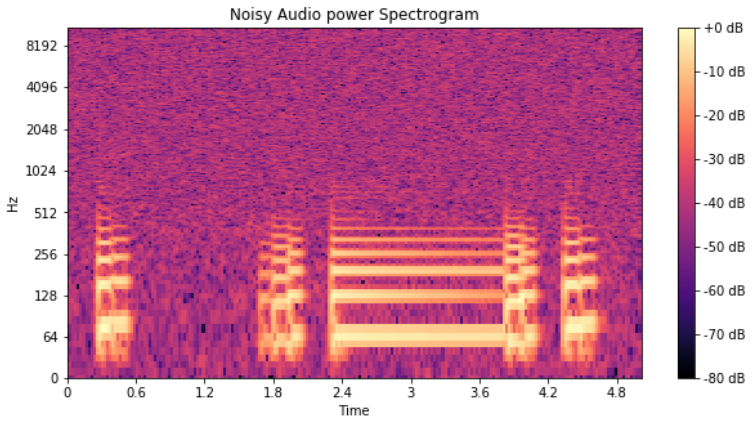



Figure 2.6: STFT of 2.2

As we can see from the before and after spectrograms, noise contaminated the entire audio signal, although the main energy bands of the original signal can still be recognised in 2.2. Unfortunately the upper harmonics are mostly gone, probably due to their weaker power. Now we might say that we do have a good noise visualiser, although, especially if applied to bass recordings, we can visualise such signals with a more suitable frequency scale.
Another point worth of notice is that, as we plot the STFT into a logarithmic scale, the lower frequencies look more "pixelated" compared to the higher frequencies, this because all the frequency bands are weighted equally. We now introduce a variation of the STFT, the Constant-Q Transform (CQT)[10][12], the main goal of such transform is to give the right amount of accuracy according to the logarithmic scale, therefore more accuracy to lower frequencies and less accuracy as the frequencies start increasing.
One thing worth noticing about the CQT is that, as the frequencies increase, the window functions get shorter, this for keeping the center frequencies geometrically spaced.
The CQT transform of any time-domain signal is given by the following formula:

$$X^{CQ}(k,n) = \sum_{j=n-[N_k/2]}^{n+[N_k/2]} x(j)a_k^*(j-n+N_k/2)$$

where:
k=1,2,...,K is the index for the frequency bins
$a_k^*(n)$ denotes the complex conjugate of $a_k(n)$ which represents the complex -valued waveforms defined by:

$$a_k(n) = \frac{1}{N_k}w(\frac{n}{N_k})exp[-i2\pi n\frac{f_k}{f_s}]$$

From a user point of view, in order to calculate the CQT we must set the following parameters:

- Sample Rate
- Number of bands per octave
- Lowest frequency
- Highest frequency

We now take a look on how the frequency domain plots (2.1 and 2.2) would look like if the CQT was used in place of the STFT:

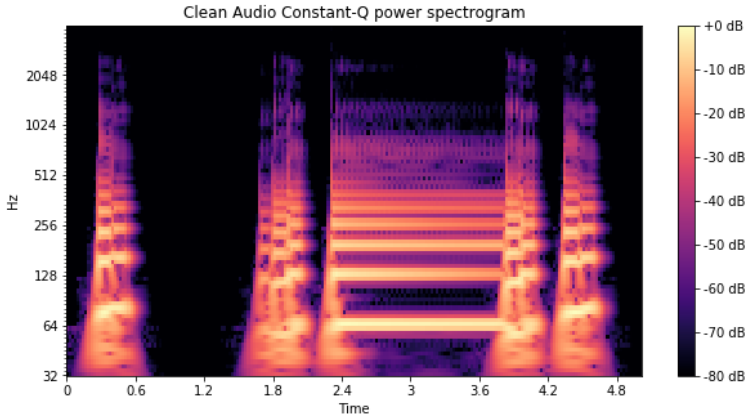



Figure 2.7: CQT of 2.1

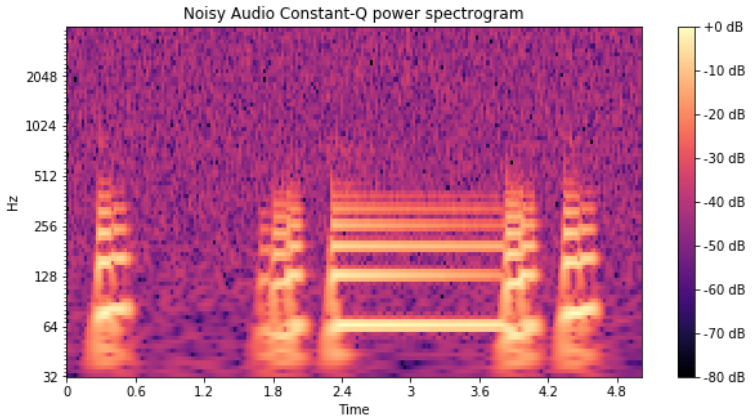



Figure 2.8: CQT of 2.2

As we can see from both 2.7 and 2.8, the CQT provides an even amount of resolution for a logarithmic frequency representation. This particular type of time-frequency domain representation is relatable to humans' hearing system [11]: the basilar membrane of the cochlea contains hair cells which resonate according to their resonating frequency, where the distribution of these frequencies follows a logarithmic scale.

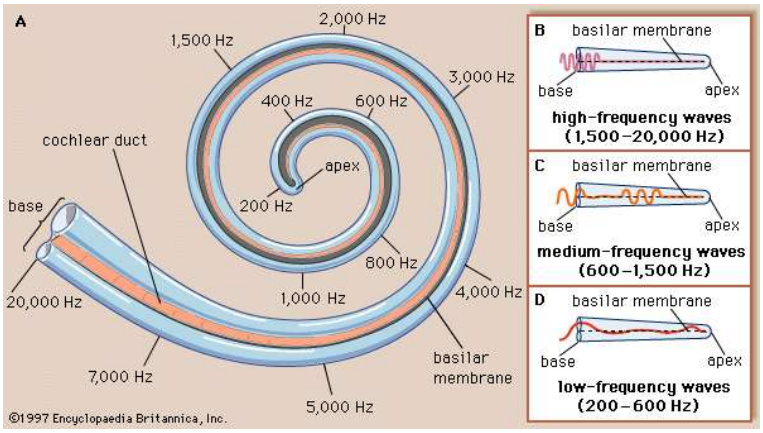



Figure 2.9: Model showing the distribution of the frequencies along the basilar membrane of the cochlea, referenced from [11]

## 2.3 Machine Learning, Deep Learning and Variational Autoencoders

Generically, the main goal of machine learning [15][38] is to meaningfully transform input data into useful representations of the data itself in order to simplify a specific subsequent task.
We can take a practical example where a given 2D plane contains binary data points represented by coordinates (x,y):

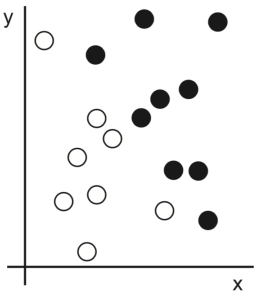



Figure 2.10: Binary sample data on a 2D space [38]

As shown in 2.10 we do have two kinds of data scattered across a plane. Let's say we want to make an algorithm able to discriminate between black and white points. Inputs would be the coordinates of each point and the expected output would be the colors of our points. We can measure the efficiency of the algorithm by checking the percentage of points that have been classified correctly.
We can do such transformation by hand as shown in 2.11:

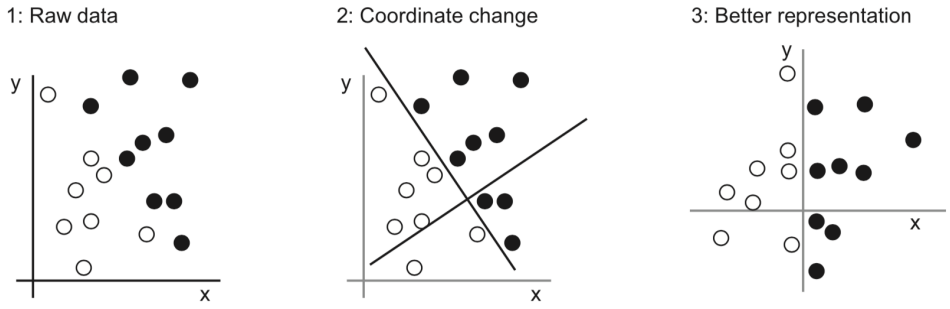



Figure 2.11: Coordinate change of 2.10 [38]

With the new representation, the classification problem can be expressed by the rule: "black points are such that $x > 0$" and "white points are such that $x < 0$".
For this very specific problem we got the solution by hand, however, if instead we tried to figure out an algorithm that recreates the approach taken by our brain we would use as feedback the percentage of points being correctly classified, this method defines machine learning: where the goal is to find a better representation.
Deep learning is a specific field of machine learning where the main goal is to learn representations from the input data and by feed forwarding the learnt weights to the successive layers add more meaningful representations.
Normally, the layered representations are learned by models known as neural networks. The amount of layers inside the model will set the depth of the

model.
The following figure shows how a 3-layers neural network works while trying to solve a classic computer vision problem: hand written digit classification:

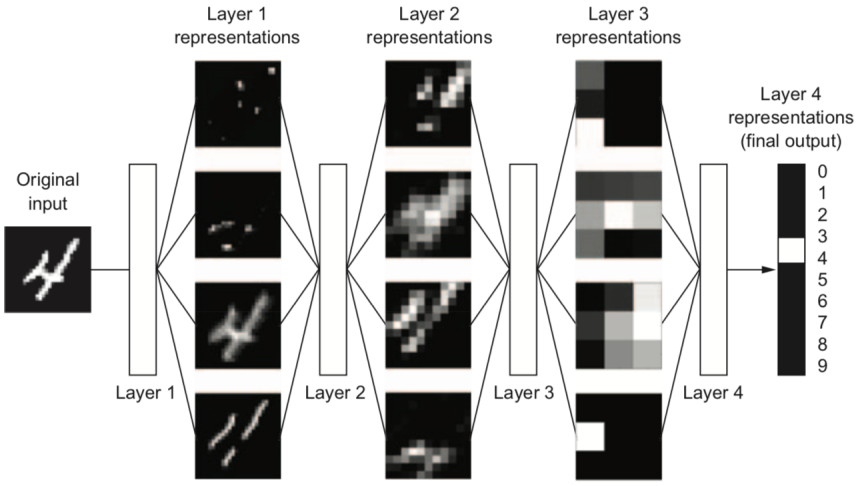



Figure 2.12: Deep representation model by a digit-classification model [38]

As shown in 2.12, the network takes the input image and transforms it into representations that are increasingly different from the original image and, at the same time, increasingly informative about the final output.
We now get into detail on how the training process inside neural networks actually happens. The layer's weights control the behaviour of the whole network, by learning we therefore mean finding a set of parameters for the weights that map example inputs to their associate targets. In order to control what the output of the network is after each training iteration, we must be able to measure the gap between the obtained output and the ideal one. The Loss function is used for such purpose: it takes the prediction of the network as well as the true target and computes a distance score, telling how well the network has done (loss score).
Another element comes now into place, the optimiser: its duty is to use the

score given by the loss function to adjust the value of the weights in order to decrease the loss score for the current training iteration. This task is known as backpropagation.

Initially, all weights are initialised with random values (far from the ideal weights), with every training iteration, the weights get tuned a little bit more towards the correct direction and, as consequence, the loss score decreases [14]. The described process is represented in the following figure:

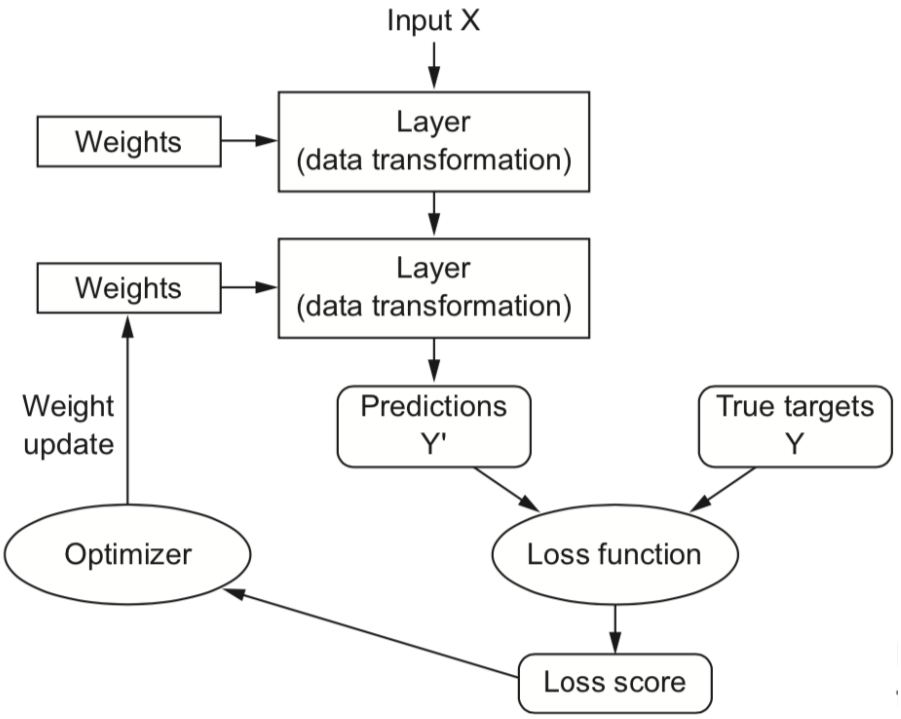



Figure 2.13: Basic diagram showing the main steps that involve training a neural network, sourced from [38]

So far we discussed about supervised learning problems where the main goal was to map a set of data to another, given enough examples of the mapping.
However, in other scenarios, we might want to generate new types of data or handle missing values. We therefore introduce the concept of unsupervised learning: a type of self-organised learning able to find unknown patterns in un-labelled data sets.
We now introduce the typology of Autoencoders neural networks [16][17][18[19][20]: a specific type of unsupervised neural network. The main goal of such networks is to be trained to attempt to copy its input to its output as precisely as possible. This type of network is composed by two main parts: dimensionality reduction and reconstruction, performed respectively by the encoder function $h = f(x)$ and decoder function $r = g(h)$. We must clarify that the goal of the autoencoder is not to map $g(f(x)) = x$ at all times, instead autoencoders are designed to be unable to learn to copy perfectly.

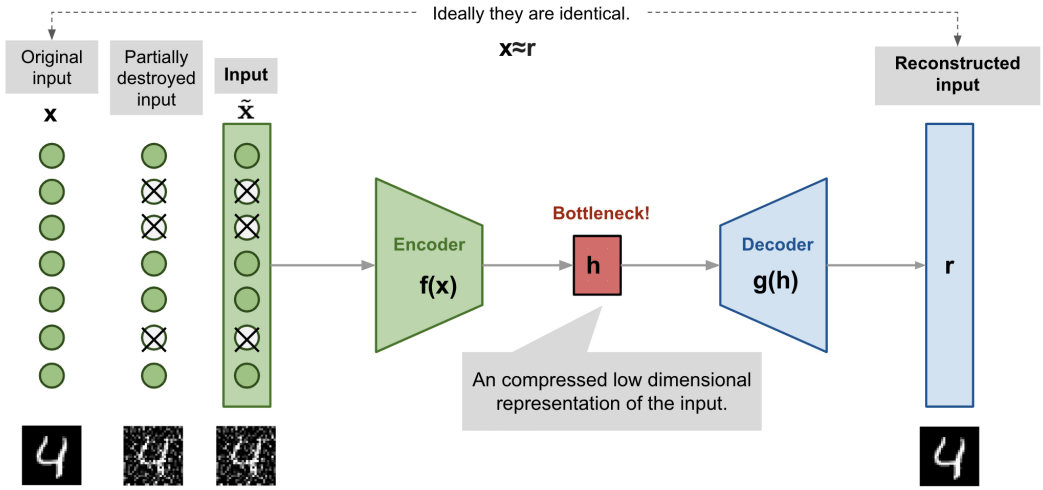



Figure 2.14: Diagram showing the processes involeved inside autoencoders: mapping an input x to an output r thorough an internal representation or code h

Next, we must discuss on how learning inside standard autoencoders is done. As previously stated learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data, therefore we must minimise the loss function by:

$$L(x, g(f(x))), \tag{2.1}$$

where L is a loss function penalising g(f(x))for being dissimilar from the input x.
We can use 2.1 for denoising purposes by minimising:

$$L(x, g(f(\tilde{x}))), \tag{2.2}$$

where $\tilde{x}$ is a corrupted version of the original input x, therefore, by changing the target, the new goal of the autoencoder will be to remove the added corruption rather than replicating the original input.
Inside training process, the denoising autoencoder (DAE) learns a reconstruction distribution $p_{reconstruct}(x|\tilde{x})$ estimated from training pairs $(x, \tilde{x})$ according to the following steps:

- Sample a training example x from the training data
- Sample a corrupted version of $\tilde{x}$ from $C(\tilde{x}|x = x)$
- Use $(x, \tilde{x})$ as a training example for estimating the autoencoder reconstruction distribution $p_{reconstruct}(x|\tilde{x}) = p_{decoder}(x|h)$ with h the output of the encoder $f(\tilde{x})$ and $p_{decoder}$ defined by a decoder $g(h)$.

## 2.4 Audio Plugins and the VST standard

In the audio industry, few components are essential to achieve a full mix ITB (in the box). Hardware-wise a computer, an audio interface and an output transductor (headphones or speakers) are essential. Software-wise, a DAW (digital audio workstation) containing at least the basic plugins is needed. Plugins [30] are a particular type of software that can be added to a specific DAW according to their extension. The following table shows the main available plugin extensions, their manufacturer and the platforms they support:

| Plugin extension types and compatibility, source: [30] | | |
|---|---|---|
| Name | Developer | Supported Platform |
| Virtual Studio Technology (VST) | Steinberg | MacOS, Windows and Linux |
| Audio Units (AU) | Apple | MacOS iOS tvOS |
| Real Time AudioSuite | Avid | MacOS and Windows |
| Avid Audio eXtension | Avid | MacOS and Windows |
| LADSPA | LGPL | MacOS, Windows and Linux |
| VAMP | vamp-plugins | MacOS, Windows and Linux |

For our purpose we focused on delivering a VST-based denoiser, the main reasons for such decision are that the VST standard [21] (now at its third main iteration: VST3) is by far the most known and therefore used audio plugin standard in the audio industry. The following list cites some of the DAWs (and other multimedia processing programs) that fully support the VST standard:

- Ableton Live
- Adobe Audition

- Adobe Premiere
- Audacity
- DaVinci Resolve
- Digital Performer
- FL Studio
- Harrison Mixbus
- iZotope RX
- Logic Pro
- Maschine
- Max MSP
- Reason
- Reaper
- Sony Vegas
- Steinberg Cubase/Nuendo

At this stage, one of the trade offs with the VST framework is related to its early stage development for Linux [22]. Steinberg announced its official support for such platform in 2017. Also, Juce [37] (the most common C++ library for compiling VST/AU plugins) has not been able yet to support the VST3 SDK for the Linux platform [39]. An improvement under such aspect would seamlessly blend the Deep Learning research environment (mainly based on Linux systems) with the VST3 plugins development, allowing a suitable environment for the creation of AI based VSTs.

# Chapter 3

# Proposed Work

## 3.1 User Requirements

The goal of the project is to deliver an user friendly autoencoder-based real time denoiser. Ideally the denoiser will be able to work on different type of musical audio sources when appropriately set by the user.
The following diagram shows how the user would interact with the final product:

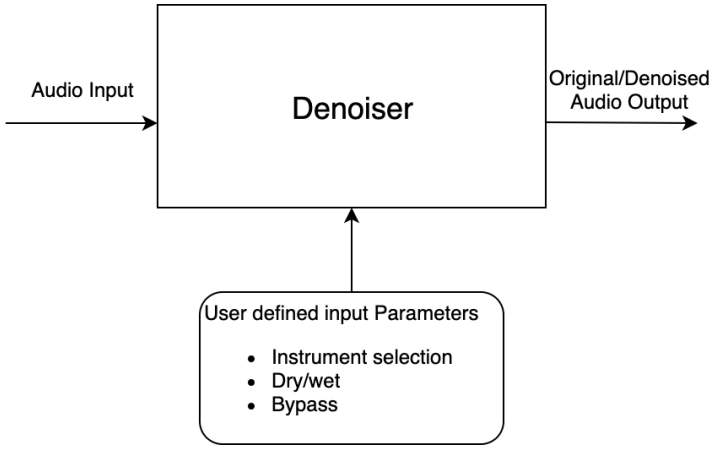



Figure 3.1: High level flow chart showing the final product functionality under a user point of view.

For the reasons mentioned in 2.4, the final plug-in will be delivered under the VST standard. Ideally, the installation process should follow a dran and drop sort of approach.

## 3.2 Business model

The main investment of the proposed project is the time required for researching the best techniques for both denoising and processing the audio data. There is no substantial economic investment for this type of product apart from a computer. All the third part libraries required for developing this project are free and open source [32][33][34][35][36][37]. Ideally, as the final product is addressed to a non-expert user audience, the ideal retail price is meant to lie on the "budget" price range.

## 3.3 System data flow

The following figure illustrates the data flow of the proposed project:

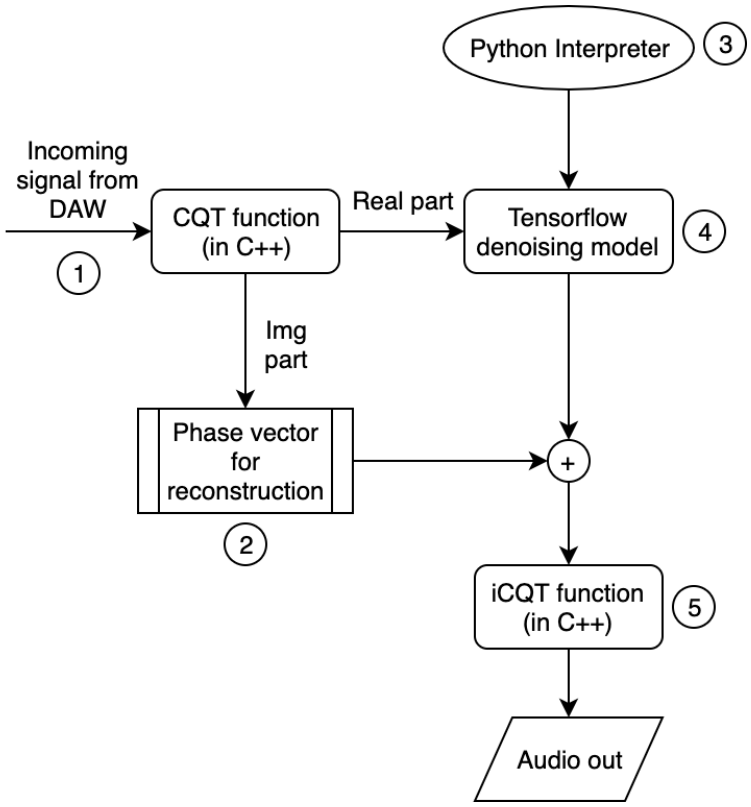



Figure 3.2: AED dataflow.

According to 3.2 we can specify the main data types used:

1. Float vector
2. Imaginary part saved as a float vector
3. Inherited from Python.h
4. Python .h5 model
5. From real+complex vector to a float amplitude vector

# Chapter 4

# Implementation

This Chapter discusses how the implementation process has been carried out: starting from neural network research all the way down to the vst deployment.

## 4.1 Neural Network

The first part of the implementation process discuss on how training and tuning the DAE can be done by using the Keras [33] environment on Python.
The autoencoder model will work on CQT transforms of single audio blocks at the time which normally range between 64 and 512 samples each according to the latency the user wants to get from the DAW. As the network works better with square-shaped input arrays, we will deal with 252x252 nunpy arrays for the Librosa CQT transform [32].
For obtaining such dimension we analyse 16100 samples-long frames sampled at 44.KHz. For the CQT we set the hop length to 64 samples and 36 bands per octave.

### 4.1.1 Original and Disrupted datasets

The first step consisted in creating randomly played bass audio files of about 10 seconds length each. Such files are created with Logic Pro X, which by default contains a satysfying set of acoustic bass synthesisers. To increase the variery of sounds, we set random parameters for the pedalboard, amp and pickups as well as by implementing random pitch shifting and BPM speed. At a first stage we came up with 60 different bass recordings of 10 seconds each.

Once all the recordings have been named with ascending values and stored in the same folder it is time to bring the files on Python for the dataset creation.

The following diagram shows how the dataset creation code works on Python:

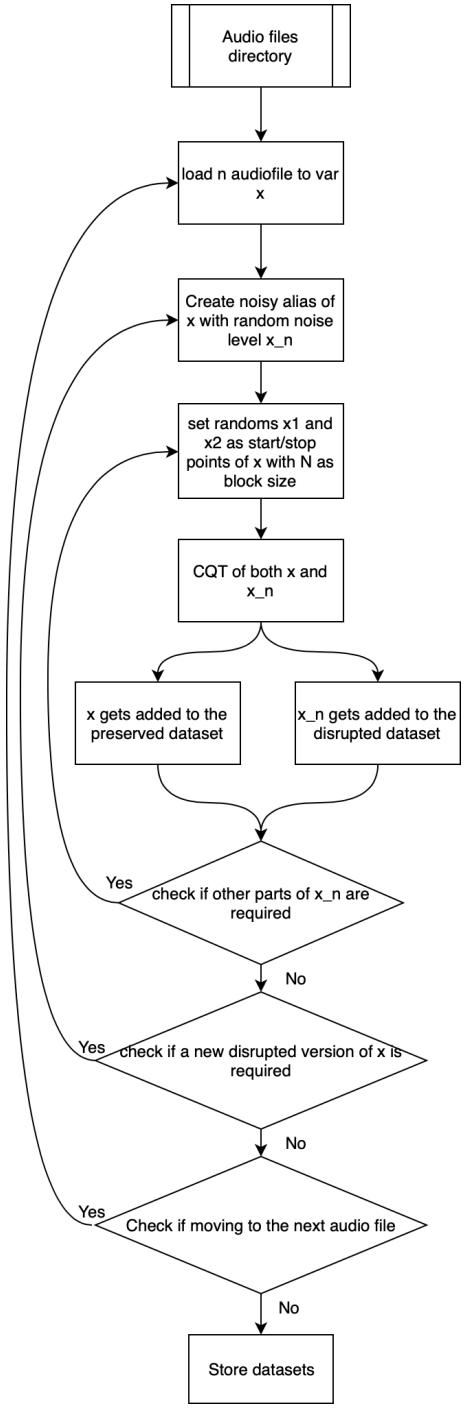



Figure 4.1: Dataset creation flowchart.

The method proposed for the dataset creation is very flexible as by just setting few parameters like the range of audio files we want to use and how many disrupted versions of each audio file we want to create we can exponentially increase the size of our training dataset. There are no constraints on the duration of the audio files as we randomly select a small part of it at every iteration. Also, if new recordings are added to the audio directory, all we need to do is to re-run the script for an updtaed dataset as the method goes through each file having a .wav extension (full code in 8.2).
We apply the same type of script for creating our testing dataset, although we use a smaller amout of data (usually by following the 70/30 rule).

### 4.1.2 Baseline Model

The main approach consists into taking a computer vision type of solution and use it for frequency domain processing. We take inspiration from Fracois Chollet's guide on how to build autoencoders on Keras [17]: this leads to the initial implementation. The following table shows how the MNIST denoising autoencoder is set for 28x28 b/w digits:

| MNIST Baseline model structure | | |
|---|---|---|
| Layer type | Output Shape | Number of parameters |
| input (InputLayer) | (None, 28, 28, 1) | 0 |
| Convolutional | (None, 28,28,32) | 320 |
| Max pooling | (None, 14,14,32) | 0 |
| Convolutional | (None, 14, 14, 32) | 9248 |
| Max pooling | (None, 7, 7, 32) | 0 |
| Convolutional | (None, 7, 7, 32) | 9248 |
| Up Sampling | (None, 14, 14, 32) | 0 |
| Convolutional | (None, 14, 14, 32) | 9248 |
| Up Sampling | (None, 28, 28, 32) | 0 |
| Convolutional | (None, 28, 28, 1) | 289 |

Effectively, the table just displayed is an implemented version of 2.14, data gets bottle necked until getting a dimension of 7x7.

### 4.1.3 Proposed Model

As already stated, for our case we will use larger arrays (the CQT function returns 252x252 vectors for 16100 samples long audio blocks sampled a 44100 Khz). As we do have bigger incoming arrays, we can get the Baseline Model and increase the amount of layers in order to extend the amount of learnable features by going deeper with the netwrok. The following table shows the Autoencoder model structure for our project:

| MNIST Baseline model structure | | |
|---|---|---|
| Layer type | Output Shape | Number of parameters |
| input (InputLayer) | (None, 252, 252, 1) | 0 |
| Convolutional | (None, 252, 252, 32) | 544 |
| Max pooling | (None, 126, 126, 32) | 0 |
| Convolutional | (None, 126, 126, 32) | 16416 |
| Max pooling | (None, 63, 63, 32) | 0 |
| Convolutional | (None, 63, 63, 32) | 16416 |
| Max pooling | (None, 21, 21, 32) | 0 |
| Convolutional | (None, 21, 21, 32) | 16416 |
| Max pooling | (None, 7, 7, 32) | 0 |
| Convolutional | (None, 7, 7, 32) | 16416 |
| Up Sampling | (None, 21, 21, 32) | 0 |
| Convolutional | (None, 21, 21, 32) | 16416 |
| Up Sampling | (None, 63, 63, 32) | 0 |
| Convolutional | (None, 63, 63, 32) | 16416 |
| Up Sampling | (None, 126, 126, 32) | 0 |
| Convolutional | (None, 126, 126, 32) | 16416 |
| Up Sampling | (None, 252, 252, 32) | 0 |
| Convolutional | (None, 252, 252, 1) | 513 |

We start from an initial 252x252 input array to a 7x7 bottle neck array: we are passing through 5 encoding layers and 4 deconding layers. More con-

volutional layers will enhance an higher amount of shapes and patterns inside the training CQTs. Section 7.2 discusses on how the technique proposed in this section might be improved.
We now list the training parameters used for the autoencoder (more in 8.4):

| Parameter | Value |
|---|---|
| Epochs | 300 |
| Batch Size | 20 |
| Shuffle | true |

## 4.2 Populated vs Non Populated Noisy Frames Binary Classifier

The deep denoising autoencoder has been designed to interact with CQT transforms of corrupted signals. However this might not always be the case, as in a real-life scenario the DAW might not receive any signals at all: in this case a white noise CQT would be sent to the DAE. In order to avoid such situation we designed a binary classifier able to distinguish between CQTs containing disrupted signals (populated) and CQTs of empty signals (non populated). This chapter shows how this classifier is going to be used in the audio procesing pipeline.
Following, two examples of populated and unpopulated audio frames:

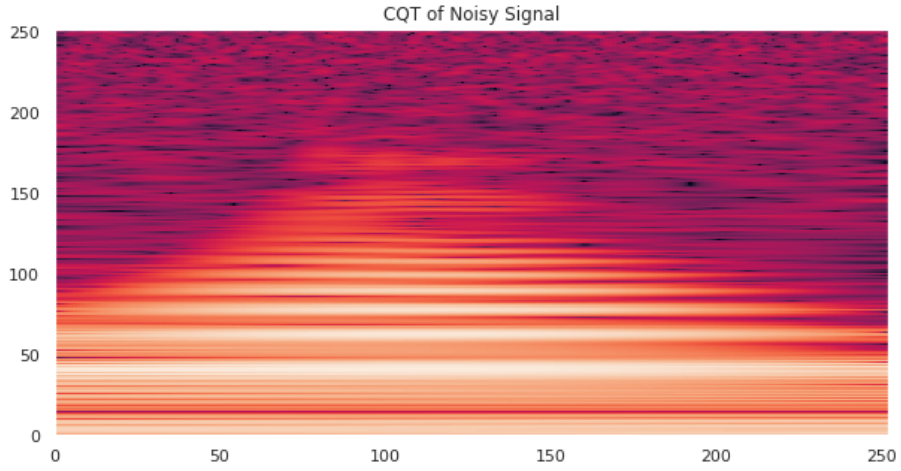



Figure 4.2: Populated CQT transform, contains actual frequency information

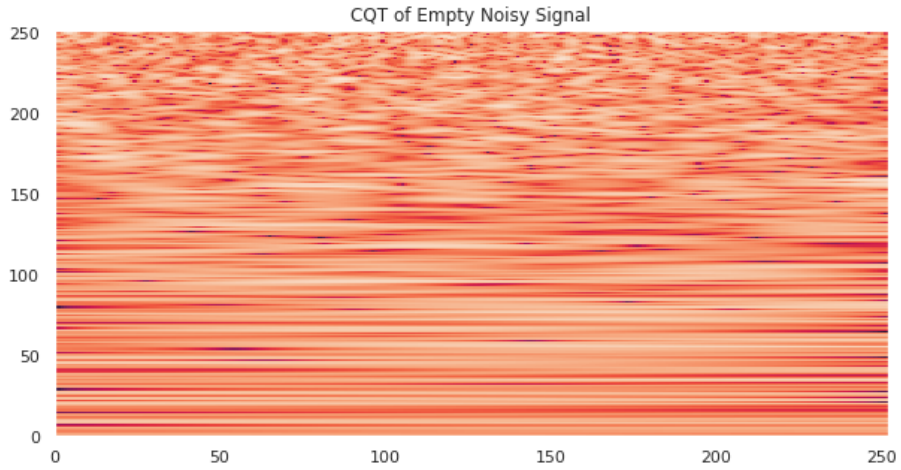



Figure 4.3: Non Populated CQT transform, only noise is stored

As we can see from the proposed figures, we can avoid trying to denoise the audio frame showed in 4.3 as the right output should be a silenced time domain signal.
The following diagram shows how the binary classifier is going to be used inside the Denoising Pipeline:

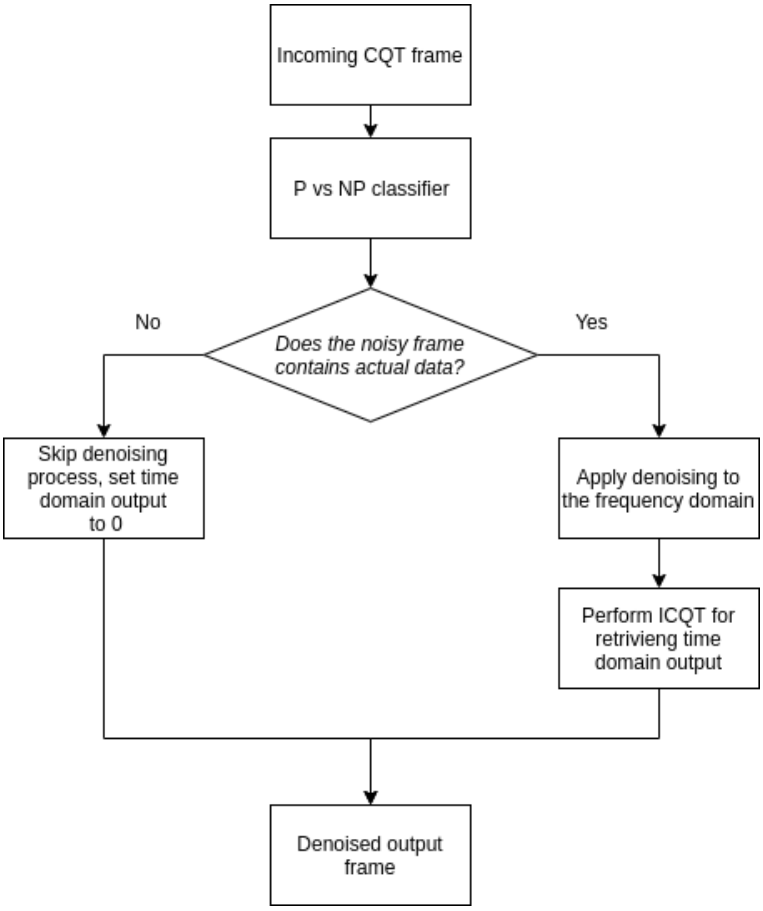



Figure 4.4: How the P vs NP binary classifier is used inside the Denoiser pipeline

The reference of the network structure has been taken from the article written by Francois Chollet on The Keras Blog titled ("Building powerful image classification models using very little data") [31], in such article the author deploys the following deep classifier convnet:

| Deep convnet binary classifier structure | | |
|---|---|---|
| Layer type | Output Shape | Number of parameters |
| input (InputLayer) | (None, 252, 252, 1) | 0 |
| Convolutional | (None, 251, 251, 32) | 160 |
| Activation (relu) | (None, 251, 251, 32) | 0) |
| Max pooling | (None, 125, 125, 32) | 0 |
| Convolutional | (None, 124, 124, 32) | 4128 |
| Activation (relu) | (None, 124, 124, 32) | 0 |
| Max pooling | (None, 62, 62, 32) | 0 |
| Convolutional | (None, 61, 61, 64) | 8250 |
| Activation (relu) | (None, 61,61,64) | 0 |
| Max pooling | (None, 30, 30, 64) | 0 |
| Flatten | (None, 57600) | 0 |
| Dense | (None, 64) | 0 |
| Activation (relu) | (None, 64) | 0 |
| Dropout | (None, 64) | 0 |
| Dense | (None, 1) | 65 |
| Activation (sigmoid) (output) | (None, 1) | 0 |

For the proposed model, we use the same structure inside the baseline model as from such network structure we obtained quite satysfying results(97% accuracy on validation data). Finally, in order to control the classifier training process, we set the earlystopping and ModelCheckpoint callbacks inside the fit function, we set as monitoring value the validation loss with a patience of 3 epochs, in this way, we avoid overfitting and save the best model (the one with the lowest validation loss). For the classifier training we set 25 epochs (even though we finished earlier thanks to the Early Stopping function), a batch size of 28 and enabled the shuffle option.
By using this approach we also save on energy consuption required by the GPU based training.

## 4.3 C++ binding for real time use

As we are saving the Python model as a .h5 file we wanto to find a way to load such model on C++ inside Juce. In order to embed the Python interpreter we are going to use the Pybind11 library. Crucial part is to set the proper linker and compiler flags in order to tell the C++ compiler that we want to embed the Python.h header. Such flags are obtained by running the following two commands on terminal:

```
/usr/local/bin/python3.7-config --cflags
/usr/local/bin/python3.7-config --lflags
```

Once obtained the flags we will have to import them inside the custom Xcode flags section of our Juce project.
The first listed approach runs the entire python interpreter inside the C++ project, such solution might be handy when many Python libraries are required at the same time, however, this might not always be the case as sometimes all we want to use from python are the stored keras models. We can use the kerasify library on C++ in order to load single keras models. Possibly, the second solution might be the lighter one under a computational point of view. At this stage we are still unable to perform the whole denoising process inside the C++ base plugin, however, for demonstration purposes we managed to call the Python interpreter and send the audio frame buffer to a Python module that changes the gain for such frame and sends it back to C++ for playback. Detailed description on the required headers and methods can be found in section 8.7.

## 4.4 Plugin UI

As initially mentioned, the aim of the plugin is to be as user accessible as possible. The user will be able to select the type of source to denoise, and, as generally the denoised output is slighlty quiter than the original versions, an output gain control allows to compensate for such issue. Finally, a Dry/Wet slider lets the user compare in real time the differences between the noisy and denoised tracks.

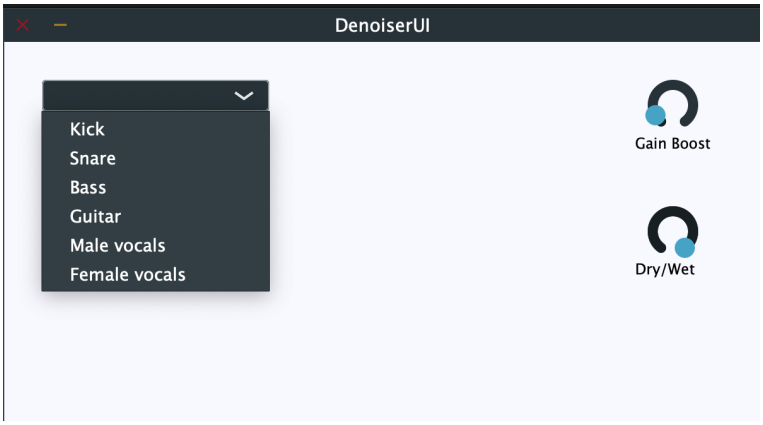



Figure 4.5: Screenshot of the denoiser user interface.

The Plug-in UI will get updated according to the newly added instruments support. As the only goal is to denoise, no particular UI components are required, such light interface will not overload the system CPU.

# Chapter 5

# Results

## 5.1 Magnitude Reconstruction Evaluation with 100 training samples

### 5.1.1 Statistical approach

In order to discuss the performance of our denoising network on real audio parts we introduce the concept of mean square error (MSE).
For our case scenario, we will use the MSE as a method to assess the quality of our estimator (the DAE).
As said by its own name, the MSE between two matrix Y and $\hat{Y}$ is obtained by calculating the average of the squared difference of the points $Y_i$ and $\hat{Y}_i$ respectively as given by the formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Following, we can introduce the peak signal to noise ratio (PSNR) which takes the MSE error and represents it under a logarithmic scale with the

highest possible value as the reference. The PSNR formula is given by:

$$PSNR = 20log_{10}(\frac{MAX}{MSE})$$

According to the given formula, for a perfectly reconstructed image (without any tipe of loss) the MSE will be zero and so the PSNR will be undefined. Follows an example of how the autoencoder worked on single audio frames. The first image represents the original signal, follows a disrupted version of the given signal and finally the reconstructed version.

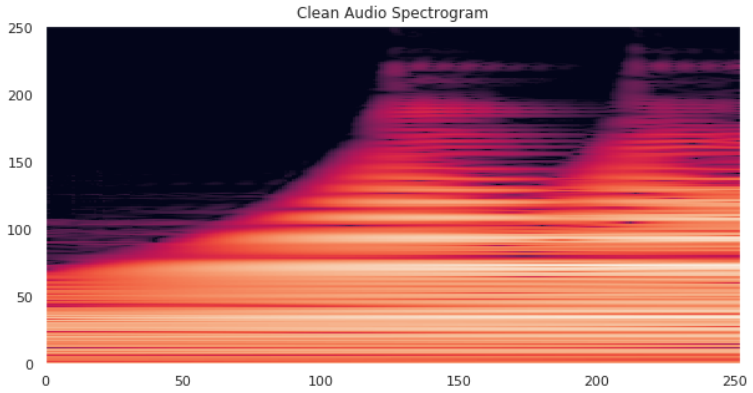



Figure 5.1: Clean audio, CQT audio frame.

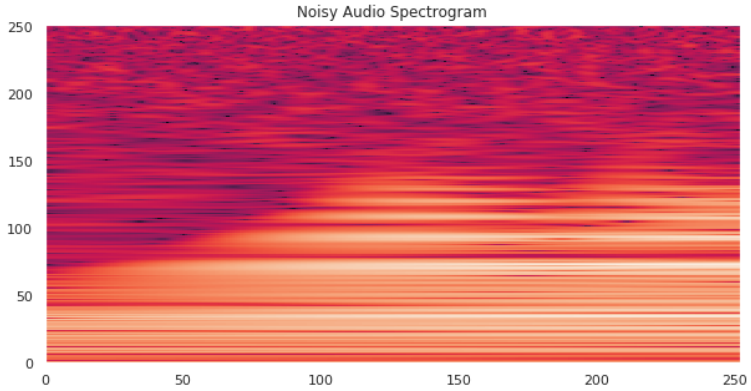



Figure 5.2: Noisy audio, CQT audio frame.

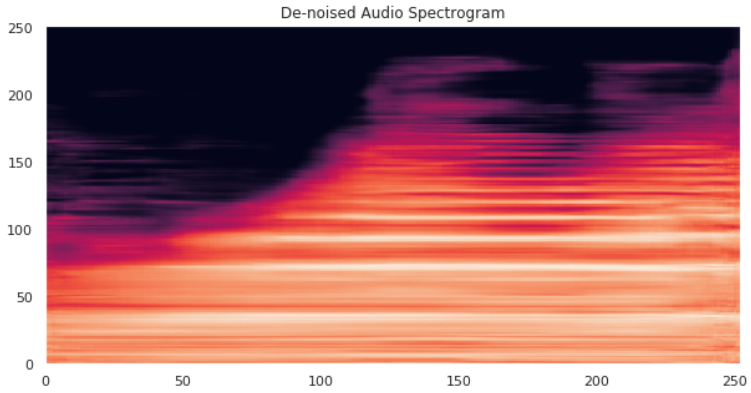



Figure 5.3: Reconstructed audio, CQT audio frame.

For the proposed figures (5.1, 5.2, 5.3) we obtained a MSE reconstruction value of 0.006 and a PSNR value of 21.7 dB. So far, for visualisation reasons, we analised the results for only a single audio CQT frame, although what we can also analyse the performance for large datsets and plot the distribution. We start by assessing the results obtained by denoising audio frames coming from already seen audio files datasets:

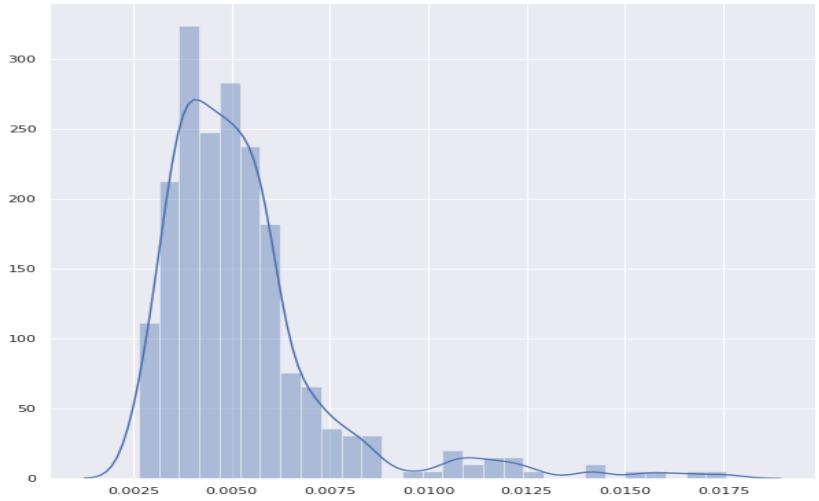



Figure 5.4: Seen Data MSE distribution of the Autoencoder's performance : x axis represents MSE values, y amount of frames with such MSE value.

We now check the PSNR for the same dataset:

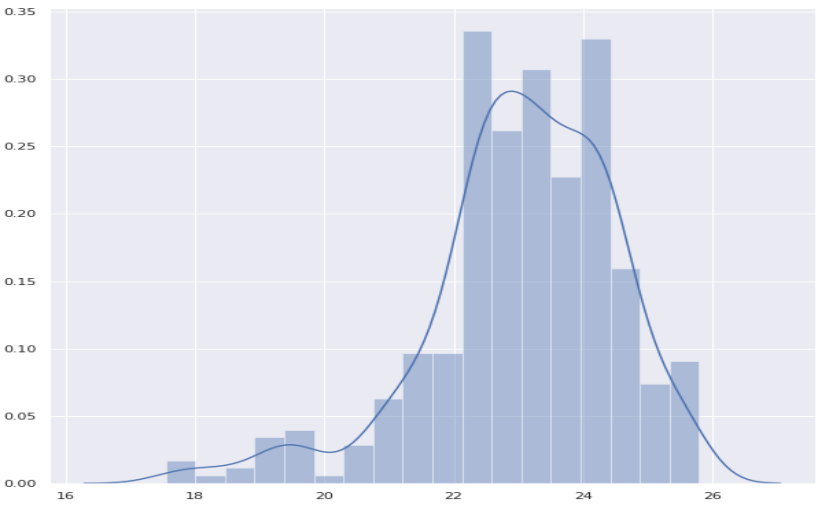



Figure 5.5: Seen Data PSNR distribution of the Autoencoder's performance : x axis represents PSNR result, y represents the probability to get such PSNR score.

We now take a look at the results coming from denoised frames of new audio files (unseen data [26]).

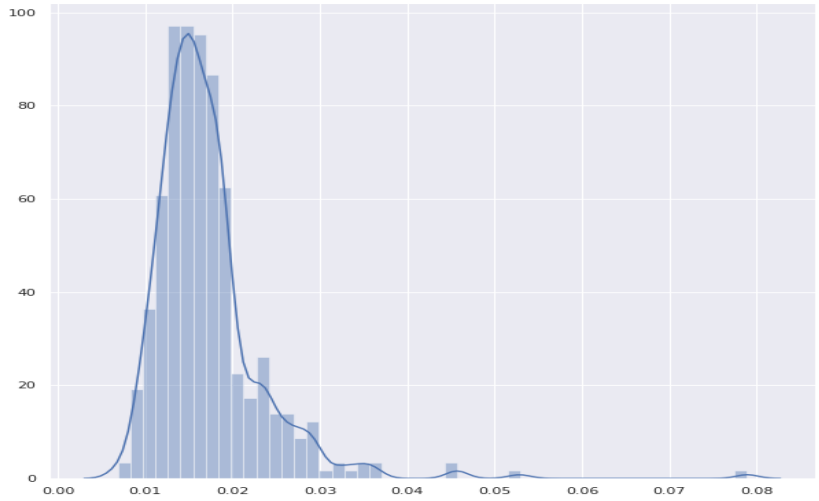



Figure 5.6: Unseen Data MSE distribution of the DAE's performance: x axis represents MSE values, y amount of frames with such MSE value.

And do the same for the PSNR distribution:

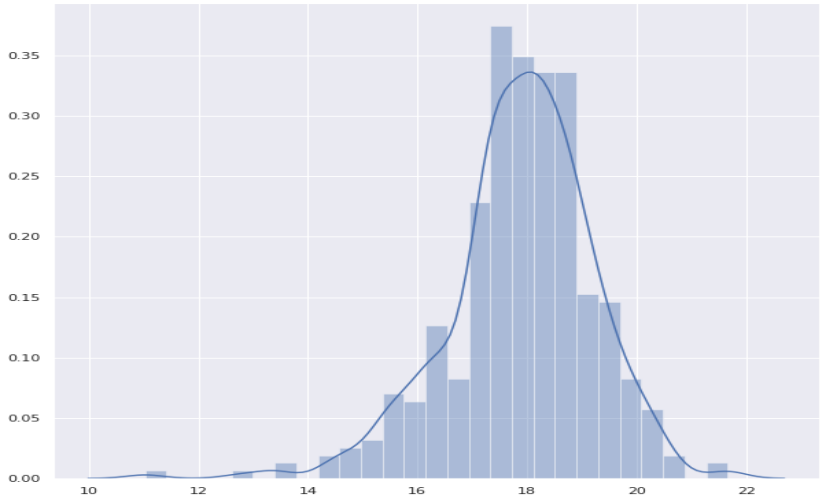



Figure 5.7: Unseen Data PSNR distribution of the Autoencoder's performance: x axis represents PSNR result, y represents the probability to get such PSNR score.

The first difference we notice between the seen and unseen datasets is that the PSNR for the seen data has an average value $\mu$ of around 23.5dB while the $\mu$ PSNR value for the unseen dataset goes around 18dB. Secondly, the distribution for the seen dataset does not follow a smooth curve, while the results distribution for the unseen data follows a more regular shape. These two factor are hints that tell us that during the network's training it might have ancountered some overfitting issues. Although, such problems did not constitue an issue while denoising newer audio frames (which is the of data our DAE will encounter during a real-life type of use).
We can also discuss about the fact that in both cases the distribution follows a negative skew type of behaviour, this is given by the fact that, during the assessment process, the model encountered some kinds of never seen CQT frames, thus the bad PSNR scores. At this stage such cases are more then acceptable as they are not caused by the network structure itself rather than a small training set.

## 5.2 Evaluation of Magnitude Reconstruction with a 18 thousand samples trained model

In the same way as section 5.2 we analyse the eventual improvements of the model presented in 4.1.3 trained with a considerabily bigger training set. We achieved an higher dataset by increasing the amount of sampled audio frames from each audio file (passing from 2 frames per audio file to 75 for each file).

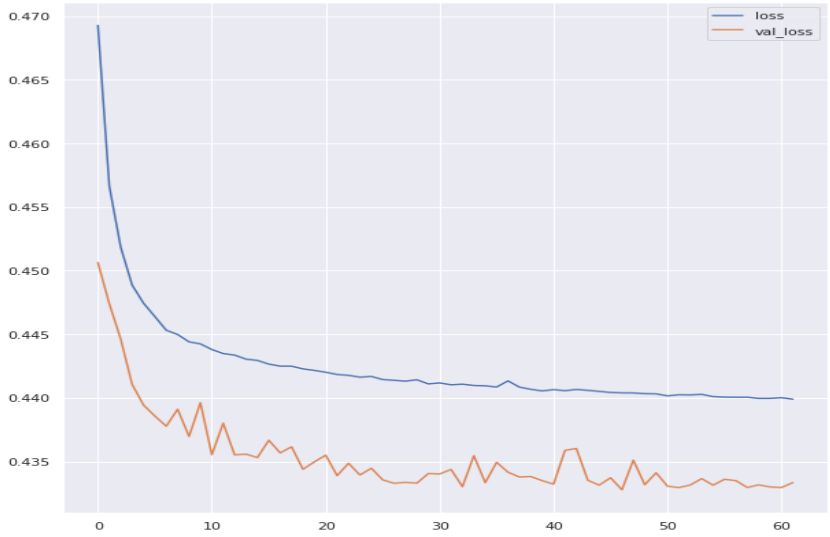



Figure 5.8: Improved model training graph, weights at epoch 45 have been saved as they provide the best validation loss therefore the best generalisation

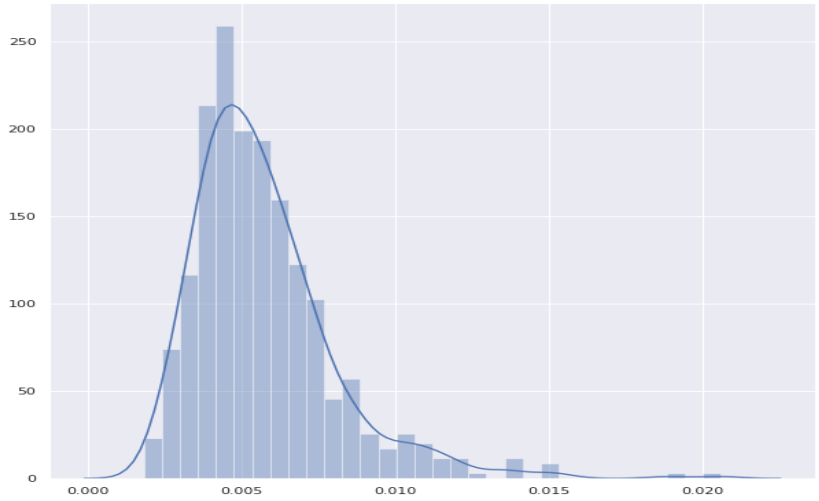



Figure 5.9: Improved model MSE, we can see an imporvement compared to 5.6.

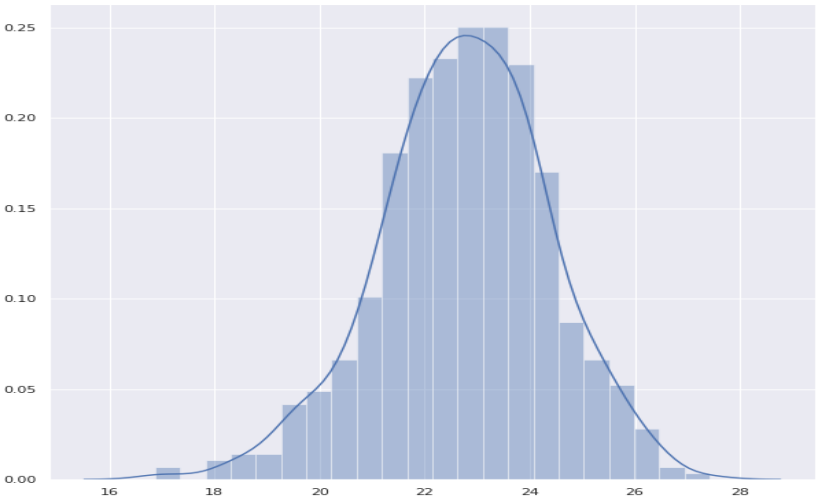


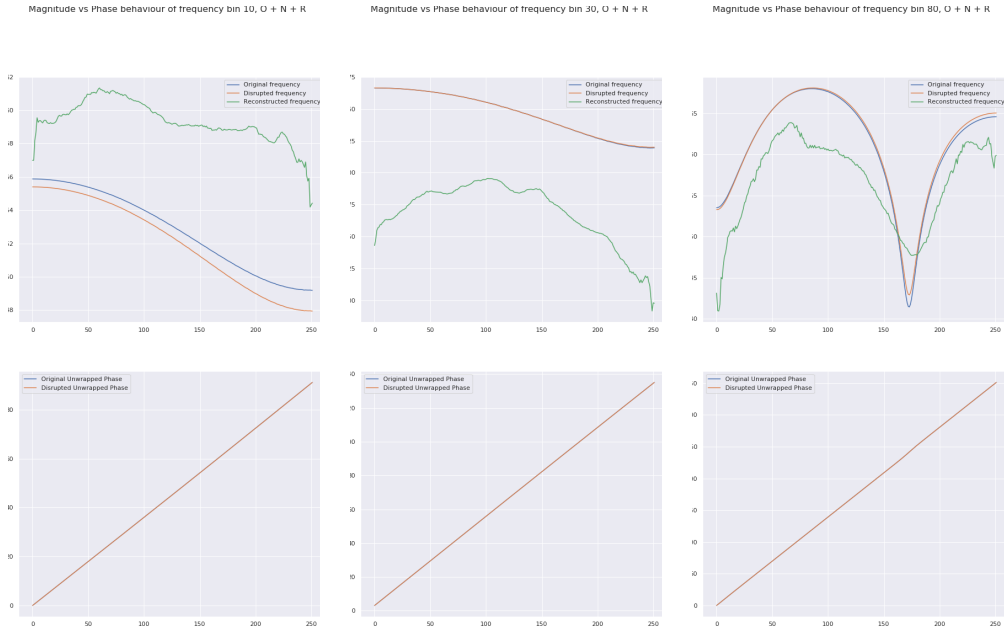

Figure 5.10: PSNR distribution for the improved model, we can see an increase of about +6dB compared to figure 5.7

As we can see from the given results distribution, an average increase of +6dB on the PSNR results is obtained, still, we are not obtaining out-

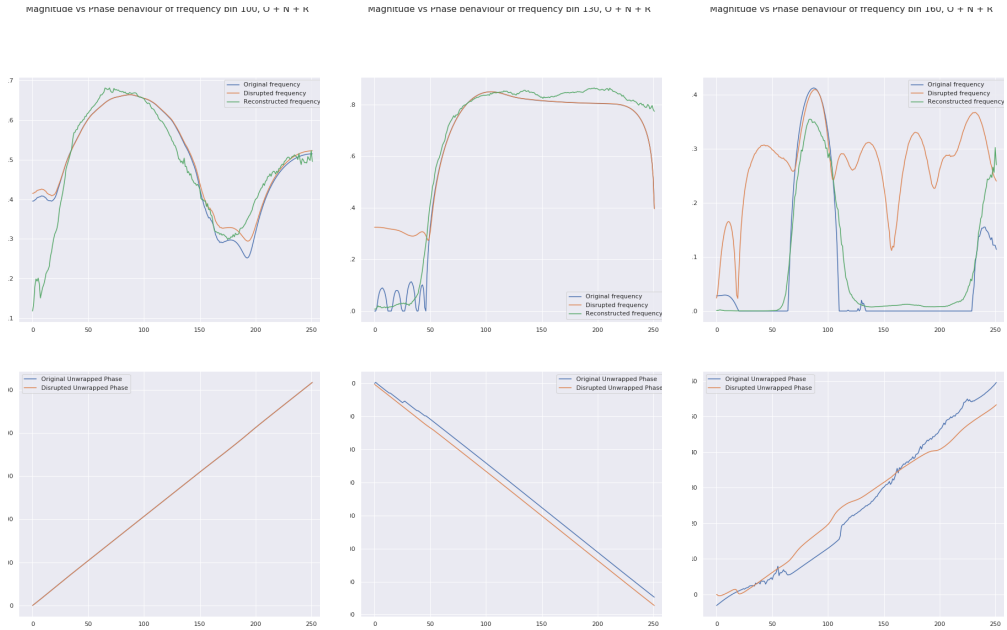
standing results (which ideally would range at about 40dB PSNR), however, these are encouraging results as by using an even bigger dataset and an improved network architecture (more explained in 7.2) we might aim towards the desired reconstruction results.

## 5.3 Magnitude vs Phase analysis for specific frequency Bins

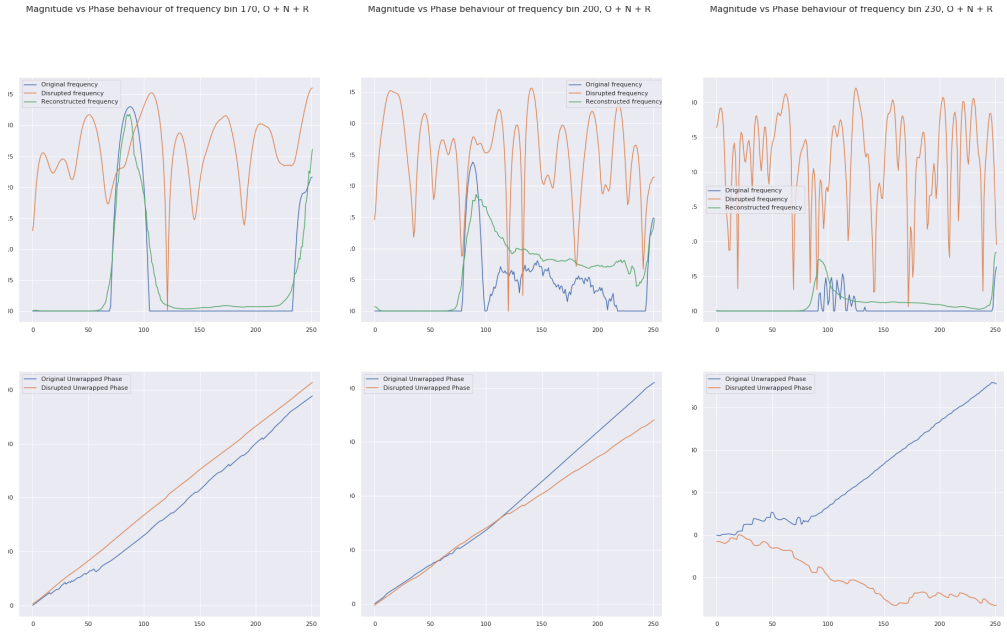
So far we have been analysing the results under a frame based point of view: a temporal representation of the evolution of all the frequency bands.
It might be convenient to visualise the evolution of each frequency bin during time and, at the same time, observe how the phase of each frequency behaved for both original and noisy versions.
The audio frames taken into analysis are purely arbitrary, the aim is to generalise how certain clean/disrupted frequency bins tend to look like and what effects on the unwrapped phase of such bins might be.
We start by analysing the evolution in magnitude for low frequencies:




(a) Bin 10 (b) Bin 30 (c) Bin 80




(a) Bin 100 (b) Bin 130 (c) Bin 160




(a) Bin 170 (b) Bin 200 (c) Bin 230

This new approach to results analysis helps us pointing out two very important aspects of the DAE:

- How well the denoiser works for a specific frequency band
- The effects of added noise to the phase response of specific frequency bins

As we can see, the DAE tries to augment in a disruptive way the magnitude evolution of low frequency bins (i.e. bin 10 and bin 30), while a good approximation of the expected results is obtained with mid frequencies (bins 80 to 170). For high frequency bins, a poor generalisation of the expected magnitude amount is obtained due to the high levels of noise disruption in such areas.

A reassuring outcome is that, for most of the bins, a linear phase is obtained, therefore, up to around bin 130, the same imaginary part of the

noisy CQT can be used for reconstructing the time domain result by applying the ICQT of imag(noisy CQT) + real (reconstructed CQT). However, for high frequency bins, as the noise spectrum gets sensibly stronger than the high harmonics of the signal, the noisy phase response will start deviating the original phase response. Solutions for the listed issues will be discussed in chapter 6.

## 5.4 Psychoacoustical evaluation of time domain results

So far we discussed on how the denoised products can be analysed under a mere statistical point of view by using comparative methods such as MSE and PSNR between the original and reconstructed products. However, as we are dealing with audio, we can evaluate the denoising quality under an objective auditive approach.
At this point, two evaluation methodologies can be considered:

1. Poll a group of critical listening trained people and ask them to mark the reconstructed audio files compared to the original ones on a quality-marking scale ranging from 5.0 (imperceptible) to 1.0 (very annoying).

2. Retrieve an algorithm able to model the frequency response of the human hearing and predict the outcome of a listening test.

As expected, option number 2 comes into resque. In 1998, the ITU (International Telecomunication Union) published a standard for objective measurement of audio quality [27]. 17 years later, Martin Holters and Udo Zolzer [29] implemented an open source algorithm of such metric: Gstpeaq comes as a gstreamer based plugin, two evaluation methods are supported: basic and advanced. Where, the first one uses only an FFT-based ear model, while, the second methodology also uses a filter-bank ear model. Ideally, the advanced mode requires more computational power, however results given by such mode are supposed to be closer to listening tests. The following table shows the results obtained with the peaq plugin in advanced mode:

| - | Original vs Noisy | Original vs Denoised |
|---|---|---|
| Objective difference grade | -3.97 | -3.612 |
| Distortion Index | -6.04 | -2.342 |

In order to obtain consistent results, the minimum duration for the analysed audio versions must be of at least half a minute. For our test we concatenated random audio files from the training/testing dataset directories, next, noise is applied and, finally, denoised with the autoencoder cited in 5.2. For different audio file durations (longer than 30 seconds) we obtained the exact same ODG and DI results, meaning we reached a good generalisation level. Quality-wise, we still have an ODG rate of -3.612 (placed in the annoying range) for the denoised audio file, a little improvement from the ODG rate of 3.97 (almost in the "Very Annoying" scale) obtained by the noisy version. The following table shows the conversion table between ODG and ITU-R Grade followed by their Impairment description:

| Impairment description | ITU-R Grade | ODG |
|---|---|---|
| Imperceptible | 5.0 | 0.0 |
| Perceptible, but not annoying | 4.0 | -1.0 |
| Slightly annoying | 3.0 | -2.0 |
| Annoying | 2.0 | -3.0 |
| Very annoying | 1.0 | -4.0 |

# Chapter 6

# Conclusion

Noise is a very well known problem inside the audio related industry, the most popular solution to such issue is to prevent and avoid any kind of noise related source inside a specific signal chain. However, this might not always be the case as, very commonly, mistakes or oversights might happen during the recording process.
This is where the proposed project comes to help by trying to offer the user an effective tool that can minimise the noise-related disruptions in a convenient way.
At the current stage, the Denoiser plugin has been only tested on a single kind of musical instrument and its noise-removal qualities are not excellent, however, with this project, we also deployed the software foundation for a future of AI based real-time plugins, by blending the real time performance of C++ with the powerful Machine Learning tools provided by Python.

# Chapter 7

# Future Work

There will never be a definitive way to deploy an audio denoiser as we showed previously, this final section discuss different paths the current research might take.

## 7.1 Adaptive noise classifier

At this stage, the same DAE deals with different levels of noise disruption as well as different types. The first idea is to create an audio classifier able to discriminate between different levels and types of noise and, according to such information, the main denoising algorithm will call the most suitable model. One of the downsides of such approach is the requirement to have as many pre-trained denoising models as the amount of noise typologies, leading to an increased requirement of data storage (non optimal if the algorithm needs to run on portable/embedded devices).

## 7.2 Area-specialised concurrent DAEs

The proposed method implements a single denoising autoencoder for the purpose of denoising a 252x252 Constant-Q Transform of a fixed-length sampled audio frame. As we saw from our tests, the reconstruction loss for denoised frames never went below 0.4, even with a very populated training set. Reason for such high loss values might be given by the fact that the number of layers is not sufficient for the network to learn all the required features inside the given CQTs. For this reason, an alternative might be to use four (or any $2^n$) DAEs of the same kind as presented in 4.1.3 specialised in denoising specific areas of the CQT. Ideally, the aim of such implementation, would be to find a good trade off between the amount of used DAEs and the obtained loss values.
Another aspect to consider while developing such solution is to give relevance to the coherence between edges of neighbour sub CQTs, such approach can be achieved by implementing a siamese network of parallel DAEs.

## 7.3 Denoising RNNs applied to single frequency bins

This third and final method considers dealing with the data obtained in 5.3 . Similar to what is proposed in 7.2, a recurrent siamese neural network that takes as inputs both the noisy evolution of magnitude and the unwrapped phase can be used to retrieve the denoised magnitude and correct phase response of the disrupted signal. We would still use a spectrogram similar to human's perceived frequency resolution like constant-Q and mel as these transforms give the best results compared to a standard STFT based spectrogram.
Few aspects still need to be clarified before implementing such solution:

- Decide the amount of RNNs needed: we might get better results by

having specialised RNNs for the main frequency bands (i.e a RNN for frequency bins ranging between 0 and 100 specialised in low harmonics denoising).

- Evaluate how to implement such solution under a practical point of view: as we are dealing with the processing of simultaneous frequency bands, using a multi threaded solution that denoises the frequency bins at the same time would considerably speed up the whole denoising process.

Advantages of this final solution are mostly related to the reconstruction quality: by addressing each frequency bin at the time, a finer reconstruction should be achieved. Additionally, if a proper multi threaded solution is used, the proposed method would probably give decent performances under a real-time case scenario.

In conclusion, the listed possibilities do not necessarily need to be taken singularly, instead, if combined together, they might lead to a very robust multi-source based denoising architecture.

# Bibliography


[1] Margaret Rouse, *Definition of audio noise*, 2005, [Online]. Retrieved 20/10/2018.

[2] Wikipedia, *Noise*, 2019, [Online]. Retrieved 28/10/2018.

[3] Wikipedia, *Balanced Audio*, 2019, [Online]. Retrieved 15/11/2018.

[4] Adnan Quadri, *A Review of Noise Cancellation Techniques for Cognitive Radio*,Department of Electrical Engineering, University of North Dakota, 2018, [Online]. Retrieved 15/11/2018.

[5] Julius O. Smith III, *Mathematics Of The Discrete Fourier Transform (DFT) With Audio Applications*, Second Edition, Center for Computer Research in Music and Acoustics (CCRMA), 2019, [Online]. Retrieved 15/11/2018.

[6] Wikipedia, *Discrete Fourier transform*, 2019, [Online]. Retrieved 20/9/2018.

[7] Wikipedia, *Short-time Fourier transform*, 2019, [Online]. Retrieved 8/12/2018.

[8] Julius O. Smith III, *The Short-Time Fourier Transform*,Center for Computer Research in Music and Acoustics (CCRMA), 2019, [Online]. Retrieved 20/10/2018.

[9] Julius O. Smith III, *Mathematical Definition of the STFT*,Center for Computer Research in Music and Acoustics (CCRMA), 2019, [Online]. Retrieved 15/12/2018.

[10] Benjamin Blankertz, *The Constant Q Transform*, 1999, [Online]. Retrieved 3/5/2019.

[11] Wikipedia, *Spectral leakage*, 2019, [Online]. Retrieved 17/1/2019.

[12] C. Schörkhuber and A. Klapuri, *Constant-Q transform toolbox for music processing*, Proceedings of the 7th Sound and Music Computing Conference, Barcelona, Spain, 2010, [Online]. Retrieved 20/5/2019.

[13] The Editors of Encyclopaedia Britannica, *Inner ear*, 2019, [Online]. Retrieved 23/11/2018.

[14] Wikipedia, *Loss Function*, 2019, [Online]. Retrieved 14/2/2019.

[15] Ian Goodfellow and Yoshua Bengio and Aaron Courville, Deep Learning, 2016, MIT Press. Retrieved 7/3/2019.

[16] Pierre Baldi , *Autoencoders, Unsupervised Learning, and Deep Architectures*, Department of Computer Science University of California, 2012, [Online]. Retrieved 13/2/2019.

[17] Francois Chollet, *Building Autoencoders in Keras*, 2016, [Online]. Retrieved 23/1/2019.

[18] Wikipedia, *Autoencoder*, 2019, [Online]. Retrieved 12/1/2019.

[19] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*, Journal of Machine Learning Research 11 3371-3408, 2010, [Online]. Retrieved 20/1/2019.

[20] Guillaume Alain and Yoshua Bengio, *What Regularized Auto-Encoders Learn from the Data Generating Distribution*, Department of Computer Science and Operations Research University of Montreal, 2014, [Online]. Retrieved 13/2/2019.

[21] Steinberg Techonologies, *VST3: New Standard for Virtual Studio Technology*,2019, [Online]. Retrieved 12/3/2019.

[22] Spencer Russell, *Steinberg brings VST to Linux, and does other good things*, 2017, [Online]. Retrieved 8/7/2019.

[23] Shuhang Gu, Radu Timofte , *A brief review of image denoising algorithms and beyond*, 2019, [Online]. Retrieved 22/1/2019.

[24] Vivek Kumar, Pranay Yadav, Atul Samadhiya, Sandeep Jain, and Prayag Tiwari , *Comparative Performance Analysis of Image De-noising Techniques*, International Conference on Innovations in Engineering and Technology (ICIET'2013) Dec. 25-26, Bangkok (Thailand), 2013, [Online]. Retrieved 3/10/2019.

[25] Mourad Talbi, Chafik Barnoussi, Cherif Adnane , *Speech Compression based on Psychoacoustic Model and A General Approach for Filter Bank Design using Optimization*, The International Arab Conference on Information Technology (ACIT'2013) 2013, [Online]. Retrieved 6/10/2019.

[26] The MusicRadar team, *SampleRadar: 392 free bass guitar samples*, 2012, [Online]. Retrieved 2/9/2018.

[27] The ITU Radiocommunication Assembly, *Method for objective measurements of perceived audio quality* , Rec. ITU-R BS.1387-1, 1998, [Online]. Retrieved 20/11/2019.

[28] Wikipedia, *PEAQ*, 2019, [Online]. Retrieved 20/11/2019.

[29] Martin Holters, Udo Zölzer, *GSTPEAQ – An Open Source Implementation Of The PEAQ Algorithm*, Proc. of the 18th Int. Conference on Digital Audio Effects (DAFx-15), Trondheim, Norway, Nov 30 - Dec 3, 2015, [Online]. Retrieved 20/11/2019.

[30] Wikipedia, *Audio Plug-in*, 2019, [Online]. Retrieved 4/1/2019.

[31] Francois Chollet, *Building powerful image classification models using very little data*, 2016, [Online]. Retrieved 8/10/2018.

[32] Librosa Development Team, *LibROSA*, 2019, [Online]. retrieved 10/1/2019.

[33] Keras, *Keras: The Python Deep Learning library*, 2019, [Online]. retrieved 29/3/2019.

[34] Numpy developers, *Numpy*, 2019, [Online]. retrieved 20/11/2018.

[35] Wenzel Jakob, *pybind11 — Seamless operability between C++11 and Python*, 2019, [Online]. retrieved 16/5/2019.

[36] Anssi Klapuri, Chris Cannam, Emmanouil Benetos, Luis Figueira, *C++ Constant-Q*, 2019, [Online]. retrieved 3/6/2019.

[37] 2019 ROLI Ltd, *JUCE*, 2019, [Online]. retrieved 20/1/2019.

[38] François Chollet, Deep Learning with Python, Manning Publications, retrieved 2/1/2019.

[39] 2019 ROLI Ltd, *Linux, VST3, the universe and everything*, 2019, [Online]. retrieved 20/9/2019.

# Chapter 8

# Appendix

## 8.1 Required Python libraries

```
#This part contains all the used libraries during
#the prototyping process on Python
import librosa
import librosa.display
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import seaborn as sns
sns.set(rc={'figure.figsize':(10,10)})
import numpy as np
import scipy
from IPython.display import Audio
import sklearn
import matplotlib.image as mpimg
from PIL import Image
import pickle
from keras.models import Model
from keras.layers import Input, Dense, Conv2D,
```

```
MaxPooling2D, UpSampling2D, Cropping2D, ZeroPadding2D, Activation,
Dropout, Flatten
from keras.models import Sequential
from keras.callbacks import EarlyStopping,
ModelCheckpoint, ReduceLROnPlateau
import os
from keras.models import load_model
import sklearn.preprocessing as sk
import random
from absl import logging
import sys
import time
from IPython.display import display, clear_output
import math
```

## 8.2 Dataset Creation

```
#Training Data For The Denoiser
x_train_noisy = [] #Data Arrays for both noisy and original
x_train = []
noise_factor = 0.05 #Amount of noise to apply
hl=64   #Hop Length for the CQT function
bpo=36  #Bands Per Octave for the CQT
sr=44100 #General Sampling Rate (librosa uses 22050Hz by default)
buffer=(16100)/sr #Setting buffer duration (in s)
path = "Denoiser/AllAudio/"  #Audio files path
counter=0 #counter for each audio frame added to the training set
scans = 0 #counter for total amount of analysed frames
frames_m=70 #Amount of analysed frames for each .wav file
#Getting details about the audio files directory
#(useful for total amount of files to be analysed)
path, dirs, files = next(os.walk("Denoiser/AllAudio/"))
#Main scan inside the entire audio directory
for names in os.listdir(path):
    #Making sure we are dealing with audio files
    if names.endswith(".wav"):
        #Iterating through the same audio file m times
        for x in range(frames_m):
            scans = scans + 1
            #as the basslines last 3 seconds we create random
            #readpoints between 0 and 2.5 seconds
            rp=random.randint(0,2500)
            rp=rp/1000 #3 decimal positions accuracy
            #setting random noise factor for such frame
            #between 0.001 and 0.05
```

```
nf=random.randint(100,5000)
nf=nf/100000;
#Calling load function from librosa for the m
#audio file with the random offset and prefixed
#buffer length
audio, sr = librosa.load(os.path.join(path,names), ...
offset=rp,duration=buffer,sr=sr)
#Creating disrupted audio frame with random noise factor
audio_noisy = audio + nf * np.random.normal(loc=0.0, ...
scale=1.0, size=audio.shape)
#Getting average rms for the clean loaded frame
#This step cheks if the audio frame contains
#actual bass information
rms=np.mean(librosa.feature.rms(audio))
#If the audio frame contains data then move to the CQT part
if rms>0.005:
    #Get CQT of noisy frame with the initially set parameters
    C_n=librosa.cqt(audio_noisy,sr=sr,hop_length=hl,...
    n_bins=7*bpo,bins_per_octave=bpo)
    #Same for clean frames
    C=librosa.cqt(audio,sr=sr,hop_length=hl,...
    n_bins=7*bpo,bins_per_octave=bpo)
    #Separate between real and imaginary parts for both
    #clean and noisy frame versions
    C_mag, C_phase=librosa.magphase(C)
    C_mag_n, C_phase_n=librosa.magphase(C_n)
    #Move the magnitude information to logarithmic scale
    #for both clean and noisy
    C_log=librosa.amplitude_to_db(C_mag,ref=np.max)
    C_log_n=librosa.amplitude_to_db...
```

```
                (C_mag_n, ref=np.max)
                #Normalise between 0 and 1 for the Neural Network
                C_log_nor=(C_log-np.amin(C_log))/(np.amax(C_log)...
                -np.amin(C_log))
                C_log_n_nor=(C_log_n-np.amin(C_log_n))/...
                (np.amax(C_log_n)-np.amin(C_log_n))
                #Attach the normalised logarithmic magnitudes to
                #the training arrays
                x_train.append(C_log_nor)
                x_train_noisy.append(C_log_n_nor)
                #Update the counter
                counter=counter+1
                clear_output(wait=True)
                #Tell the user how much has passed by and how many frames
                #are left to analyse
                display(str(counter)+' training snaps have been stacked...
            '+str(scans)+' analysed out of '+str(frames_m*len(files))...
                +' loss: '+str(scans-counter))
                time.sleep(0.001)
#Changing array type and dimension in order
#to suite the NN requirements
x_train = np.array(x_train, dtype=np.float32)
x_train_noisy = np.array(x_train_noisy, dtype=np.float32)
x_train = np.reshape(x_train, (len(x_train), 252, 252, 1))
x_train_noisy = np.reshape(x_train_noisy, (len(x_train_noisy),...
252, 252, 1))
#Saving the training arrays into external files in case
#they are needed for future access
np.save('Denoiser/Datasets/denoiser_x_train_271119v4.npy', x_train)
np.save('Denoiser/Datasets/denoiser_x_train_noisy_271119v4.npy',...
```

```
x_train_noisy)

#Training Data for the Classifier
#Very similar to the code for the autoencoder
#has a label vector rather than a noisy vector
cl_y_train = []
#Inside the main for loop
#This if statement balances the ratio between
#empty and populated labels
#it waits for the empty frames to appear and catch
#up with the amount
#of populated frames
if(((empty/signal)<0.48 and rms<0.005)or(empty/signal)>=0.48):
#After the if statement we fill the training datasets accordingly
            if rms>0.005: #if populated
              signal = signal + 1 #increase populated counter
              cl_x_train.append([C_log_n_nor,1]) #append with label (1)
            else:
              empty = empty + 1 #increase empty counter
              cl_x_train.append([C_log_n_nor,0]) #append with label (0)
```

## 8.3 Binary Classifier training

```
#Model creation
model = Sequential()
#convolutional layer for edge detection
model.add(Conv2D(32, (2, 2), input_shape=X.shape[1:]))
#relu activation node
model.add(Activation('relu'))
#maxpooling layer for dimensionality reduction
#picks the higher one from a set of 2x2
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (2, 2)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (2, 2)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
#changing array dimension
model.add(Dense(1))
#sigmoid activation for binary purposes
model.add(Activation('sigmoid'))
#copile the model
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
```

```
                metrics=['accuracy'])
#print details about model
model.summary()

#Callbacks section
#stops the trainin process according to the parameters
earlyStopping = EarlyStopping(monitor='loss', patience=3,...
verbose=1, mode='min')
#save best model
mcp_save = ModelCheckpoint('drive/Colab_Notebooks/...
Models/classifierv1.h5', save_best_only=True,...
#Reduce learning rate after loss stops improving
reduce_lr_loss = ReduceLROnPlateau(monitor='loss',...
factor=0.1, patience=3, verbose=1, epsilon=1e-4, mode='min')

#Classifier Training with callbacks
batch_size = 28
#actual training
model.fit(X,y,epochs=25,shuffle=True,...
callbacks=[earlyStopping,mcp_save, reduce_lr_loss],)
```

## 8.4 Autoencoder training

```
#Deeper autoencoder, increased size of conv to improve resolution
input_data = Input(shape=(252, 252, 1)) #Set input array size
# Conv layer
x = Conv2D(32, (4, 4), activation='relu', padding='same')(input_data)
#Maxpooling for dimensionality reduction
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (4, 4), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (4, 4), activation='relu', padding='same')(x)
x = MaxPooling2D((3, 3), padding='same')(x)
x = Conv2D(32, (4, 4), activation='relu', padding='same')(x)
encoded = MaxPooling2D((3, 3), padding='same')(x)

# at this point the representation is (7, 7, 32), bottleneck
#Reverse process begins
x = Conv2D(32, (4, 4), activation='relu', padding='same')(encoded)
x = UpSampling2D((3, 3))(x)
x = Conv2D(32, (4, 4), activation='relu', padding='same')(x)
x = UpSampling2D((3, 3))(x)
x = Conv2D(32, (4, 4), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (4, 4), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
#Getting decoded array, same size of the input array
decoded = Conv2D(1, (4, 4), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_data, decoded)
autoencoder.compile(optimizer='Adam', loss='binary_crossentropy')
```

```
autoencoder.summary()
```

## 8.5 Denoising algorithm

```
#we first create a crossfader in order to reduce audible
#discrepancies between neighbour audio frames
edge = 250 #sets the L and R slope sizes
inner = np.ones(16100-(edge*2)+1) #sets the middle area between slopes
window = np.blackman(edge*2)
fade = []
#concatenating window edges with inner parts
fade.extend(window[0:edge])
fade.extend(inner)
fade.extend(window[edge:-1])
#debugging options
#plt.plot(fade)
#print(np.size(fade))

#Moving to the actual denoising process
out = []
l=16100 #Frame size for the CQT
#going through the amount of frames that fit inside the audio file
#TODO implement method that denoises
#until the very end of a given file
for x in range (int(i)):
  #Reshape the processed audio file
  inp = np.reshape (Bass[x], (1,252,252,1))
  #Call the classifier for empty spotting
  #as we have a sigmoid activation we set 0.5 for the activation thrs
  if(model.predict(inp)[0]<0.5):
    #empty spotted, frame flagged as empty (1)
    flag=1
```

```
        #output for debugging
        print('empty_spotted')
        #skipping the denoising process
    else:
        #populated frame spotted, frame flagged as full (0)
        #calling the denoising autoencoder
        X=autoencoder.predict(inp)
        flag=0
    #reshape denoised frame for the ICQT
    X=X.reshape(252,252)
    #denormalise
    X_den=((80.0)*((X-np.amin(X))/(np.amax(X)-np.amin(X))))-80.0
    #move to linear scale from db
    X_amp=librosa.db_to_amplitude(X_den)
    #apply old phase (coming from noisy frame)
    X=X_amp*(Bass_phase[x])
    #If the frame has been denoised with the autoencoder do the following
    if flag==0:
        #iCQT with the initial settings
        Y=librosa.icqt(X,sr=sr,hop_length=hl,bins_per_octave=bpo,length=l
    else:
        #Empty noisy frame given which equals an ideal flat time domain o
        Y=np.zeros(16100)
    #Apply crossfade to the audio frame
    Y = Y * fade
    #Attach the modified frame to the final output
    out.extend(Y)
```

## 8.6 Results Analysis

```
#Statistical Evaluation
#Loading the datasets:
#original and noisy frames at different levels of noise
x_test = np.load('Denoiser/Datasets/denoiser_x_test_271119v2.npy',...
allow_pickle=True)
x_test_noisy = np.load('Denoiser/Datasets/...
denoiser_x_test_noisy_271119v2.npy',allow_pickle=True)
#Predict the denoised frames
predictions = autoencoder.predict(x_test_noisy)
#handy variables for storing temp spectrograms
A = []
B = []
mses=[] #Store MSEs between the iterated A and B (clean vs noisy)
psnrs=[] #Store the PSNRs difference
dim=(np.shape(x_test_noisy)[0]) #count how many frames there are
for i in range(0,dim): #iterate through each frame
  A = x_test[i].reshape(252, 252) #stores the ith clean frame
  B = predictions[i].reshape(252,252) #stores the ith denoised frame
  # Calculate the MSE between A and B
  mse = np.square(np.subtract(A, B)).mean()
  #Stores the current mse to the mses vector
  mses = np.append(mses, mse)
  #same process for the psnr value
  psnr = 20*math.log10(1)-10*math.log10(mse)
  psnrs = np.append(psnrs,psnr)

#plotting the mse distribution with seaborn
mse_distirbution=sns.distplot(mses)
```

```
fig = mse_distirbution.get_figure()
#saving the figure
fig.savefig('Denoiser/mse_distirbution18k.png')

#Same steps for the PSNRs distribution
psnr_distribution=sns.distplot(psnrs)
fig = psnr_distribution.get_figure()
fig.savefig('Denoiser/psnr_distirbution18k.png')
```

## 8.7 Plugin Python Wrapping

```
//Call to the Python interpreter in
//PluginProcessor cpp file, only Python related code,
//Full program attached in the submission files
#include <embed.h> //embeds the interpreter
#include <pybind11.h> //ports the python libraries
#include <stl.h> //compulsory for casting
//between python and c++ types and viceversa
#include "ConstantQ.h" //direct CQT
#include "CQInverse.h" //Inverse CQT

void Denoiserv2AudioProcessor::prepareToPlay
(double sampleRate, int samplesPerBlock)
{
    //std::cout<<"hmt: "<<hmt<<"\n";
    int bpo = 36; //CQT parameters
    int maxFreq = sampleRate / 3;
    int minFreq = 100;
    //counter for how many times the prepareToPlay function
    //is called (problem encountered with the plugin host)
    hmt++;
    if(hmt==3){ //Calling python interpreter at the third instance
        try{
            //Initialise the interpreter
            py::initialize_interpreter();
            //Import the custom python module test
            moduletest = py::module::import("moduletest");
            //increase the counter
            hmt++;
```

```
        }
        //Catching the case where the interpreter is already
        //running (not working at this stage)
        catch (py::error_already_set const &pythonErr)
        {  std::cout << pythonErr.what(); }
    }
    //instanting CQparameters object
    CQParameters params = CQParameters(sampleRate,
    minFreq, maxFreq, bpo);
    //Instantiating the direct CQT and inverse CQT objects with
    //params as settings
    ConstantQ cq(params);
    CQInverse cqi(params);
}

void Denoiserv2AudioProcessor::processBlock
(AudioBuffer<float>& buffer, MidiBuffer& midiMessages)
{
 for (int channel = 0; channel < totalNumInputChannels; channel++)
    {
        //TODO: use less vectors, work directly with the indexes
        float* channelData = buffer.getWritePointer (channel);
        std::vector<float> draft; //draft vector
        float* start = buffer.getWritePointer(0); // get the
        //pointer to the first sample
        int size = buffer.getNumSamples();
        std::vector<float> tmp (start, start + size);
        draft=testModule(tmp, channel); //saving the testmodule
                                        //output onto draft vector
        for (int i=0;i<numSamples;i++){
```

```
            channelData[i]=draft[i]; //feeding draft to op channel
        }
    }
}

//function that takes an input audio vector, sends it to a python
//module that boosts its gain and sends it back to C++.
//Implemented to prove that there is an actual exchange of data
//between C++ and Python at this stage
//TODO: implement keras based denoising rather than gain boost
std::vector<float> Denoiserv2AudioProcessor::testModule
(std::vector<float> tmp, int channel){
    try{
        std::vector<float> vect; //vector that receives the
                                 //module output
        //sending the incoming c++ vector (tmp) to the
        //python module containing the editarray function
        // we receive a generic auto array from such
        //python function
        auto resultobj = moduletest.attr("editarray")(tmp);
        //casting the python returned object to a c++ float
        //vector
        vect=resultobj.cast<std::vector<float>>();
        //Returning the edited on python vector
        return vect;
    }
    catch(py::error_already_set const &pythonErr){
        //Catching any possible interpreter error
        std::cout << pythonErr.what();
    }
```

```
}

//PluginProcessor header file
#include <embed.h>
#include <pybind11.h>
#include <stl.h>
#include "ConstantQ.h"
#include "CQInverse.h"
class Denoiserv2AudioProcessor  : public AudioProcessor
{
public:

    std::vector<float> testModule(std::vector<float> tmp,int channel);
    py::object scope;
    py::module moduletest;
    int hmt=0;
    }


#Python based module that changes the gain for an incoming
#audio frame, to be placed in the module folder found by os
import numpy as np
import essentia.standard
from keras.models import Model
 UpSampling2D, Cropping2D, ZeroPadding2D
import sys
print(sys.path) #print module path for debug

def editarray (arraytest): #receive, modify  and resend
                           #the c++ array
    arraytest*0.5;         #change gain
```

```
    return (arraytest)
```