Final report

ToneCrafter

Pol BODET
Pierre CHOUTEAU
Mamadou DIA
Adrien DUWAT
Louis PRADINES
Hector RICHARD
Quentin WACONGNE

Supervised by Sylvain "Syd" Reynal

Project realized during the second year of studies at ENSEA • 2020-2021

Contents

Chapter 1

Introduction

POL BODET, PIERRE CHOUTEAU, MAMADOU DIA

Have you ever wanted to start an instrument and, beyond the difficulty of getting started, it sounded very bad? You don't know how to tune it because there are too many buttons? And, even after several months, when you want to sound like your favorite artist you can't do it? You still don't understand how to set up your instrument, which pot to move for which effect or even which button to press on your synth ?
All of this undermines your motivation and the efforts of new musicians. And taking it a step further, it wastes a lot of the musician's time to get all their pedals and amps set up properly. With the ToneCrafter team, we are convinced that if you sound good, you'll be passionate and spend a lot more time on your instruments, that's why we wanted to offer you this little wonder called ToneCrafter.

Chapter 2

Concept

POL BODET, PIERRE CHOUTEAU, MAMADOU DIA

The ToneCrafter is above all a software that can be found on computer or on smartphone, easily downloadable and easy to use. Thanks to it, all you have to do is connect your instrument to your computer or your phone, search for the artist or the music you want to play, and that's it. You don't have to do anything else. The ToneCrafter will take care of setting all the effects pots itself so you can get the sound you were hoping for.

All you have to do is play. What's really cool about the ToneCrafter is that it allows you to :

- Make the sound of the greatest accessible to all
- Available to beginners and professionals alike
- Simple to use
- Possibility to create a community of users

Chapter 3

Searching for ideas

POL BODET, PIERRE CHOUTEAU, MAMADOU DIA

3.1 Design Thinking

In order to find this incredible and unique idea, we started with a design thinking approach. This approach is used a lot by designers and other creators, to conceive an object from the needs defined by a preliminary study. It also allows everyone to express themselves freely and without judgment through words written on post-its, about a slogan, a sentence, or a social fact. For our study, we worked on the slogan: "Vintage is Great Age". In the design thinking process, we analyzed this slogan and noted all the words that were evoked by the slogan, whether they were more or less close to the theme. We then tried to categorize them thematically in order to see more of the interactions that can occur between each of them.

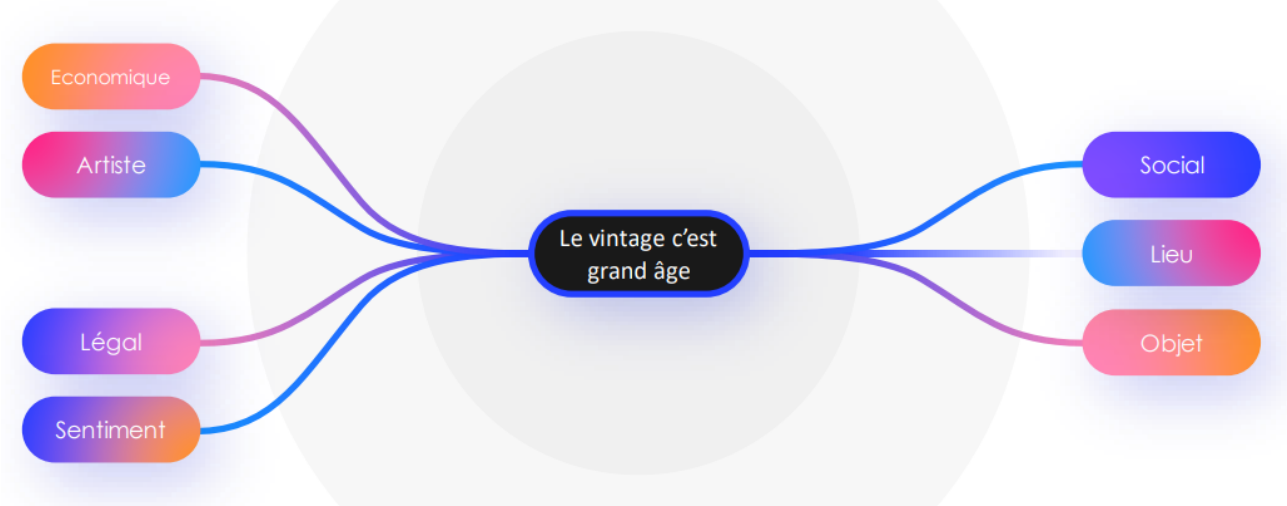

Figure 3.1: Concepts evocated by the slogan

3.2 Setting up the paradoxes

It is from all these words, but especially from the themes that were identified that we were able to put the paradoxes that gave birth to our project:

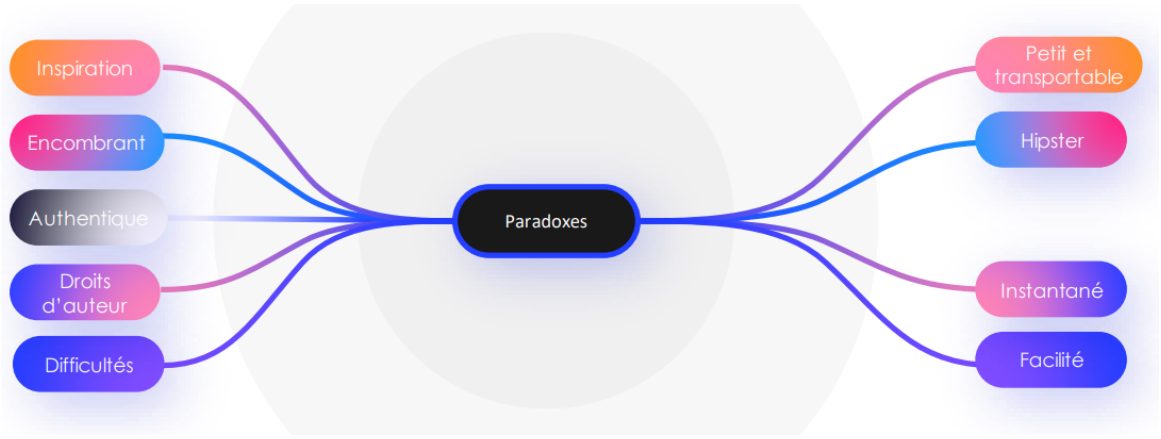

Figure 3.2: Paradoxes raised by the slogan

The opposition of the various paradoxes raised allowed us to define a problematic of the type:

“How can an object be authentic while pushing the creativity of the user without infringing copyright laws?”

The ToneCrafter is, according to us, the answer to this problem because of the novelty of the platforms on which it will work, while preserving this authentic and rough side that is the sacrosanct “Tone” of the guitarists, it brings to the masses a sound that was until now inaccessible which allows new creations to emerge. After posing our problem, we had to define in detail what the ToneCrafter was. To do this, we wrote a user guide as well as a comic strip explaining the heart of the project in a pictorial way. This one explained how to replicate the "Tone" of an instrument artist, in only a few clicks. The procedure is very simple: you choose an artist in the database or you upload a song from which you want to extract the parameters, the application then generates a filter allowing to reproduce the expected effects.

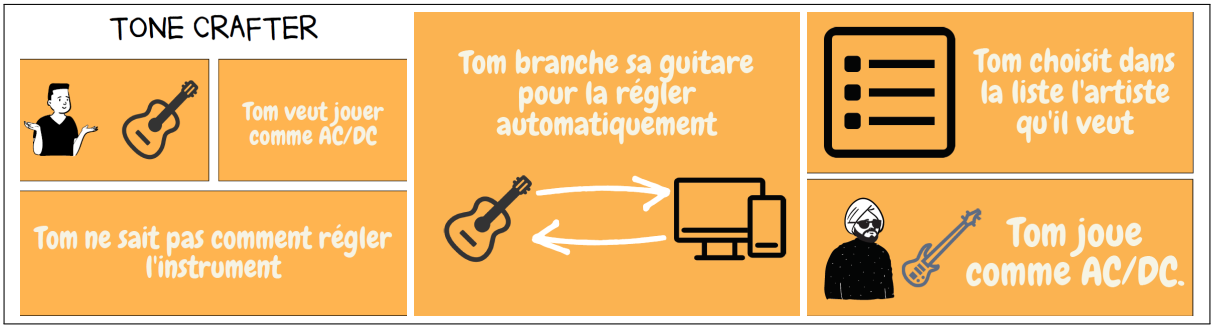

Figure 3.3: Cartoon ToneCrafter

Now that we have imagined how we want the product to be used, we need to determine how the product should be installed and how to configure it. So, we have written a user guide explaining how the first configuration should be done. The process must be simple and fast. That’s why we designed

something simple. A power supply and a connection to a terminal where the ToneCrafter software is installed should allow the system to be configured. You can see the shape we had imagined for our system. A kind of amplifier. This rather simplistic shape reduces the number of connections and allows to have only one system.

3.3 Reflection on the use of the product

In order to be able to project how the user can appropriate the product and how to create the best user experience, we thus realized a comic strip and a user guide. These 2 documents served as a reference on what we wanted to give as a deliverable at the end of this project.

3.4 State of the Art

In order to have a more global vision of the different technologies used in this domain or even of existing projects that are close to what we want to do, we started by doing a state of the art. While doing this research we came across a project that particularly caught our attention:

The Magenta Project

But what is the Magenta Project?

Simply put, Magenta is an open source research project that was launched by researchers and engineers from the Google Brain team. Their objectives are to develop DeepLearning and Reinforcement Learning algorithms to do a large number of things, ranging from music generation to signal transformation (timbre transfer, music transformation...). Two of these research works were of particular interest to us because they allowed us to observe and test different algorithms that could help us for this project:

3.4.1 NSynth: Neural Audio Synthesis

NSynth, as its name indicates, is a neural synthesizer: unlike a classical synthesizer that generates sounds from a VCO, NSynth uses a neural network trained with a dataset created for the occasion.
You may wonder why we were interested in a project so far from the one we were preparing ?
The answer: the NSynth Super
This is a hardware controller created specifically to use the algorithm of the software project. We were intrigued by this object whose form factor is reminiscent of that of a guitarist's effect pedal. effect pedal for guitarists and whose implementation is simply exquisite (see figure 3.4).
Just like the software side of the project, the NSynth Super is an Open-Source project that each user can modify according to his needs. We notice that the controls are not realized by a touch screen in the version proposed to the general public, surely in order to simplify the manufacturing of the controller and to make it accessible to a larger public. We will keep this objective of accessibility of the project during all of the development phase.

3.4.2 ToneTransfer | DDSP : Differentiable Digital Signal Processing

Who hasn't dreamed of being able to play any instrument, without even knowing how to use it? Or to transform everyday sounds into instrument sounds? Today, with this research project, it is done. If you want to try it, it's by ***here***.

Figure 3.4: Prototype of the NSynth Super

This is the main reason why this project caught our attention, especially for the software part. It is very similar to what we want to do, which is to reproduce the sound of a certain type of effect from a base signal. The only difference is that here, instead of approaching an effect, such as a distortion, a reverb or even a chorus, their machine learning algorithm allows to modify the original signal in order to obtain the sound of the instrument you want. For example, it is possible to obtain a violin sound from a voice

Play : Chant

Play : Violon

The principle is exactly the same, the algorithm modifies the input signal in order to obtain a different sound, as we can see with the two spectrograms of the figure below:

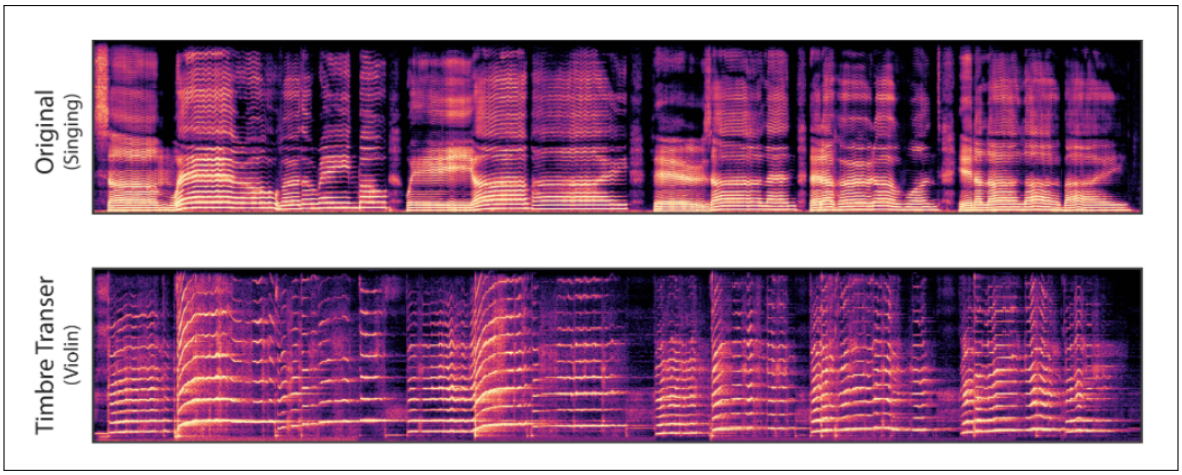

Figure 3.5: Timbre Transfer : Violin -> Flute

The ideal case for us would be to be able to use this algorithm in real time, but especially to be able to train it on audio effects and not on instruments.

What is also very interesting with this project is that in addition to being able to load pre-trained models in order to approach several instruments, it is also possible to train a model with its own sounds.

3.4.3 Effects pedals

As we have seen, these two previous projects (NSynth and ToneTransfer) are very interesting and have a lot in common with what we want to do. However, today, the existing and commercialized object that most resembles what we want to do remains the classic effect pedal. There is a very large number of them, for each effect (Distortion, Overdrive, Octaver, Delay, Reverb...) and for everyone, and thanks to them, we can do exactly what we want, that is to say: get closer to the sound of our favorite artists. However, the problem with these classic effects pedals is that you have to buy a lot of them if you want to get exactly the same sound as a particular artist... Simply because any pedal has its own sound and if you change the brand or the model... Then the effect is different. After a lot of discussion with the team, we agreed that the design of these pedals would fit our project completely. We would like to keep the sturdiness, the solidity as well as the size of these, so that our final object is easily transportable and usable everywhere, whether it is in concert or at home, without being afraid to break it.

Figure 3.6: Pedalboard

Figure 3.7: Multiple-Effects Pedal

Chapter 4

Software development

POL BODET, PIERRE CHOUTEAU, MAMADOU DIA, ADRIEN DUWAT, LOUIS PRADINES

4.1 Objective

The main objective of this part is to create and train an AI able to apply effects and filters on an audio signal in real time.

4.2 Discovery of the DDSP library

After having discovered the research work of the Magenta project on Timbre Transfer, and in order to test what it was possible to do in audio processing, we started by testing the potential of the library at the heart of this project: DDSP (Differentiable Digital Signal Processing). This library not only makes it possible to generate audio signals using neural networks that have already been trained, but it also makes it possible to apply effects such as reverb, vibrato, or flanger. The real advantage of DDSP is that it allows you to mix signal processing algorithms and techniques with neural networks. Thus, we can use linear filters, which we program quite easily on DSPs, with neural networks. This makes it easier to use, and increases the number of possibilities. Moreover, contrary to other artificial intelligence modeling tools, it is possible to work with auto-encoders, which could have been a real plus in the continuation of the project. So we started by trying out the different effects available in DDSP, by recording guitar sounds and applying them to them:

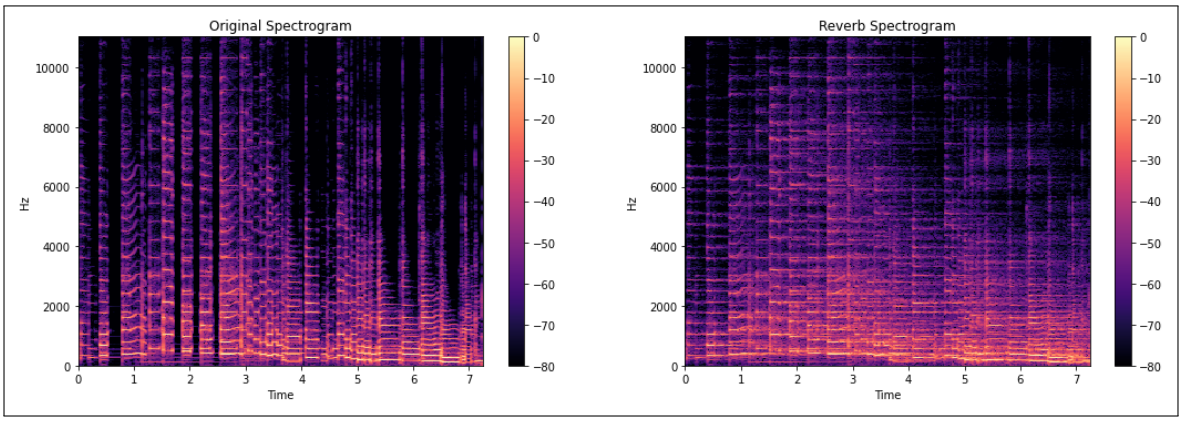

Figure 4.1: Reverb from the DDSP library

As we can see on these different spectrograms and by listening to the audios obtained, the results were very satisfactory for a beginning and the effects were well audible.
However, we were working here on guitar recordings, and the purpose of the project is to be able to modify the signal coming from the instrument in real time.

The next step is therefore obvious, after having succeeded in creating and modifying signals by applying effects to them, we started to look at the question of real time. To do this, we created a Python script allowing us to acquire a sound coming from the microphone of our computer in order to apply

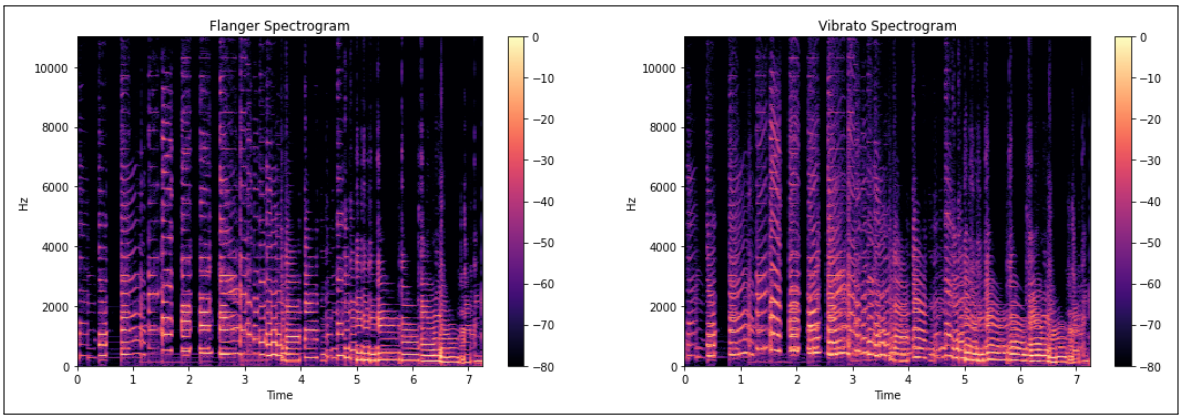

Figure 4.2: Flanger and Vibrato from the DDSP library

Play : Original Play : Reverb Play : Flanger Play : Vibrato

an effect to it and feed it back to the speakers. It worked perfectly, until we started to apply an effect from the DDSP library... That's when we realized the limitations of this library. With this library, we cannot modify the sound coming from the microphone in real time. Indeed, the management of samples and memory does not allow us to apply effects and manage a buffer in parallel. It would therefore be necessary to rewrite part of the library in another language in order to manage the low level layers, which would then quickly become tedious. In order to check that our real-time script was working well, we tried to apply a distortion, created by us, to the signal coming from the microphone. The distortion used is shown below:

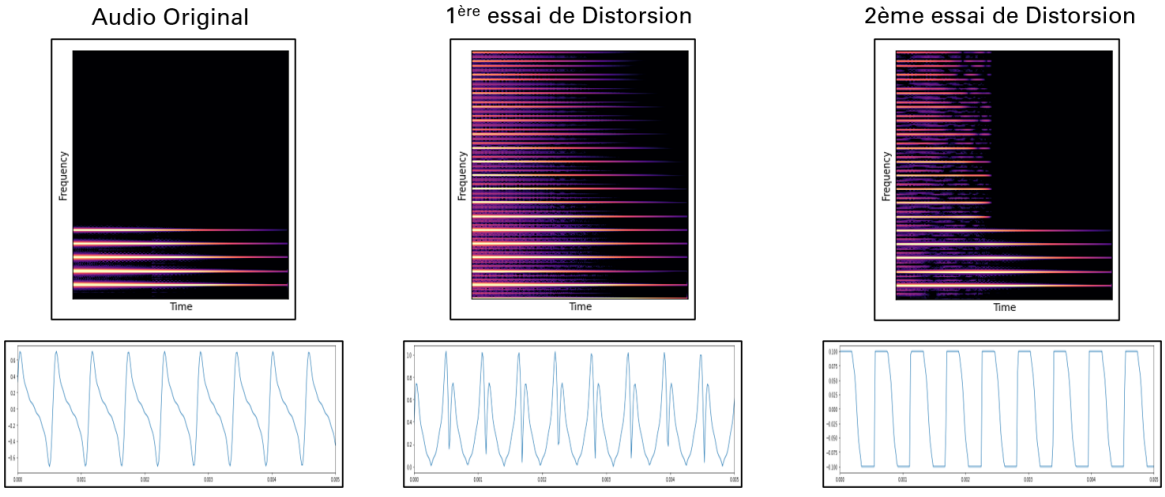

Figure 4.3: Test of our real-time distortion

After a few tests in real time, we were able to conclude that it worked well, and therefore that it was the definition of the DDSP library that was the problem. This is partly due to the types of variables used by DDSP (Tensors) which are not compatible with real-time signal processing.

Following this first problem, we decided to change our approach to achieve our goal.

The study is now focused on a real time application, which means that at the very moment the signal is acquired, it must be processed and restored. Since the DDSP library only allows us to work on signals of which we know all the samples, we decided to follow three different methods that we will study later.

4.3 Setting up our solutions

4.3.1 Variational Auto-Encoder

Initially, our wish was to have a neural network able to identify the characteristics of an effect in a sound in order to be able to apply this same effect on another sound. We were inspired by "Style Transfer" algorithms combined with Auto-Encoders. Indeed, auto-encoders are able to take an input signal, and apply filters or transforms to it. This corresponds completely with the requirements of our project. We therefore wanted to combine artificial intelligence tools with auto-encoders, hence our desire to use a Variational Auto-Encoder (VAE).

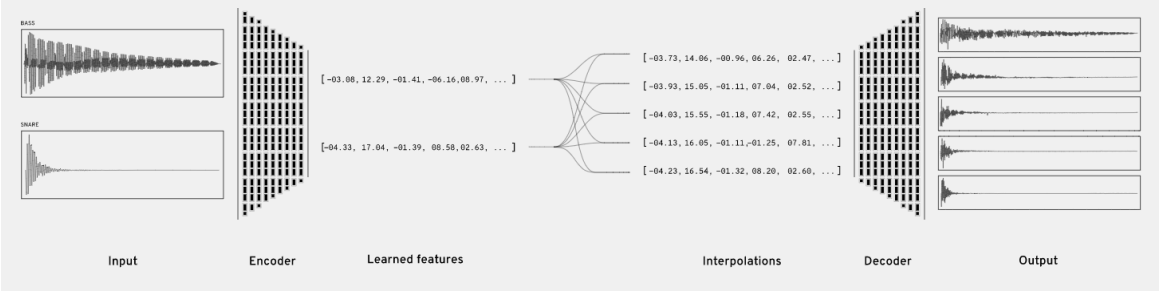

Figure 4.4: Structure of the VAE used by Magenta

Auto-Encoders are pairs of combined neural networks. Their objective is to find a way to encode data in a compressed form, which is called latent space, but unlike Auto-Encoders, Variational Auto Encoders (VAE) are not satisfied with simply being able to retrieve the original input from the latent space (compression). VAEs assume that the data is produced by a directed graphical model. The VAE will therefore learn an estimate of this graphical model. Thus the VAE can be used to generate data consistent with the input data. Indeed it is enough to put random numbers in input of the decoder to create a coherent data. But in order to be able to use a VAE to apply effects to a sound, we will study the latent space of a VAE.

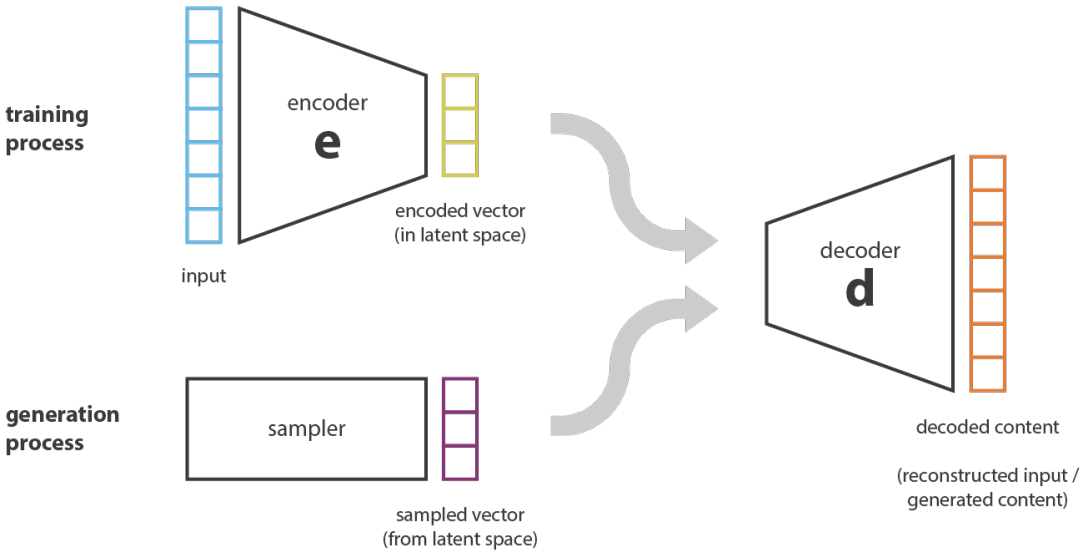

Figure 4.5: How a VAE works

4.3.1.1 Analysis of the latent space

Unfortunately for us, there are very few projects and research on music using a VAE. Fortunately, it is possible to work on audio signals using spectrograms from STFT (short time Fourier transform), which happen to be 2D signals. Moreover, working in frequency will have the advantage of avoiding that the VAE learns sinusoids, which it would constantly find if we used music in time. However no database of music in frequency representation exists. So we made the choice to start by studying a VAE using images, assuming that what would work on an image would work on a sound.
So we will use the MNIST database.The MNIST database for Modified or Mixed National Institute of Standards and Technology, is a database of handwritten numbers. It contains in our case, a training set of 60 000 images of handwritten digits and a test set of 10 000 images of digits. Each image is 28p*28p in size and each image is associated with the value of the written digit.

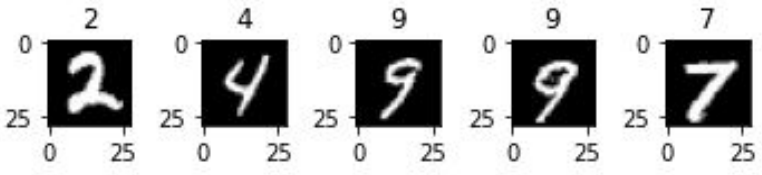

Figure 4.6: 5 images and their values randomly drawn from the training dataset

To analyze the latent space, we will start by analyzing the images that the decoder of our VAE associates to each point of the latent space. Thus we will use a 2 dimensional latent space to represent it easily on graphs. Then for each pair (x,y) of the set [-3,3]*[-3,3] we will generate the associated image. We then obtain a map of the data decoded by the decoder.

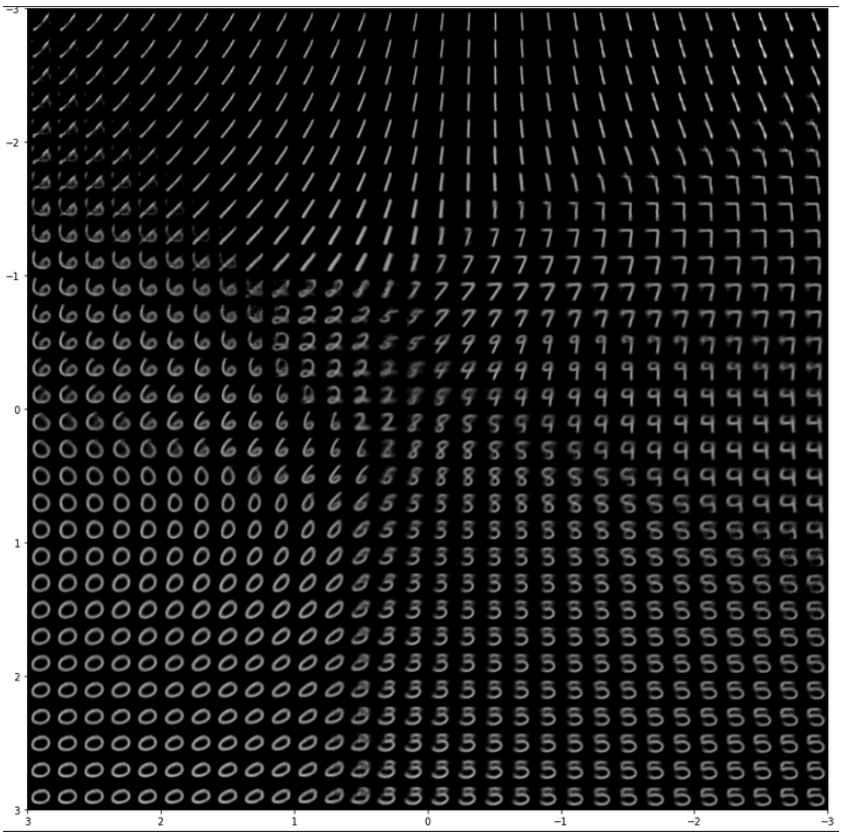

Figure 4.7: Map of the decoded data

In order to make a more accurate analysis we will now do the opposite. This time we will take all the images of the test set and display on a graph the 2D vector of the latent space with which this image is associated.

These two maps allow us to notice that for the VAE to work with very few errors, it must have a

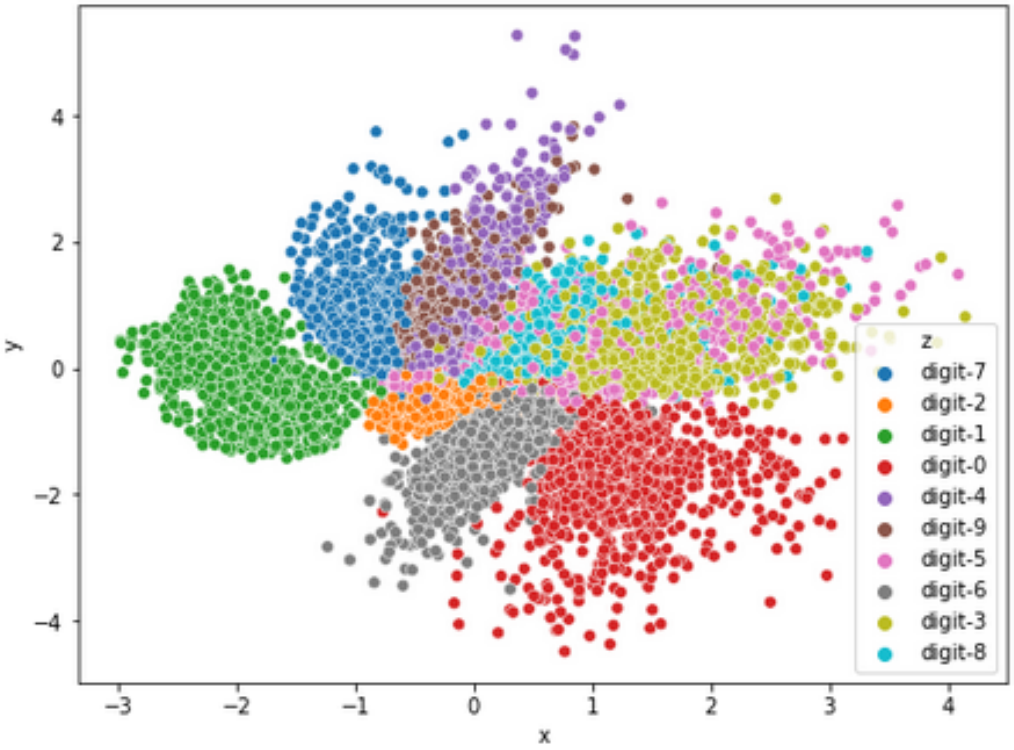

Figure 4.8: Map of the encoded data

latent space with a sufficiently large number of dimensions so that all the values have distinct spaces. Otherwise we will have errors of decoding for the numbers which do not have a distinct space. In our case we notice that the 4, the 9, the 7 and the 5 do not have a distinct space.

Indeed as we can see it on the figure below the VAE often produces errors during the decoding of these numbers.

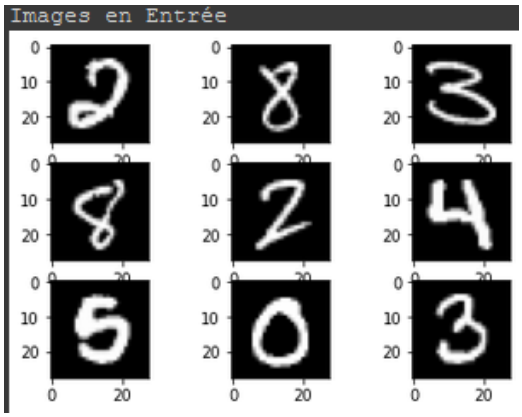

Figure 4.9: Input of the VAE

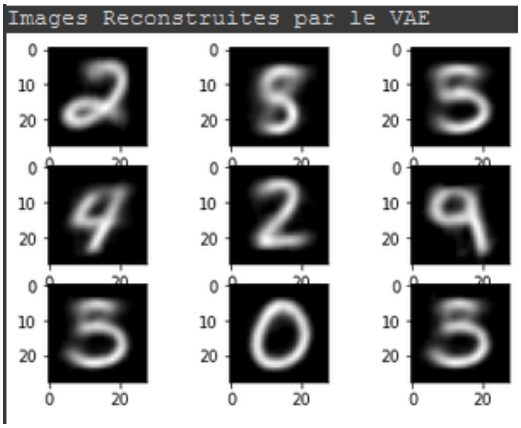

Figure 4.10: Output of the VAE

4.3.1.2 Transforming an image

In our project, the VAE would be driven by a database containing different sounds with and without effects and each of these sounds would be associated with the name of the effects applied. Thus if the VAE is well adapted to our database each effect would have a distinct space in the latent space. So if we go back to the MNIST database, for us applying an effect would be like transforming a number into another number. So since normally the numbers are all grouped in distinct spaces in the latent space. If we calculate the average value of each space, we could convert one number into another by making it follow the translation defined by the vector that goes from the average value of its space to the average value of the space of the number we want to transform it into. The advantage is that the number would keep its own characteristics, if it is on the left of the set of 1 it would remain on the left of the set of 0 after its translation.

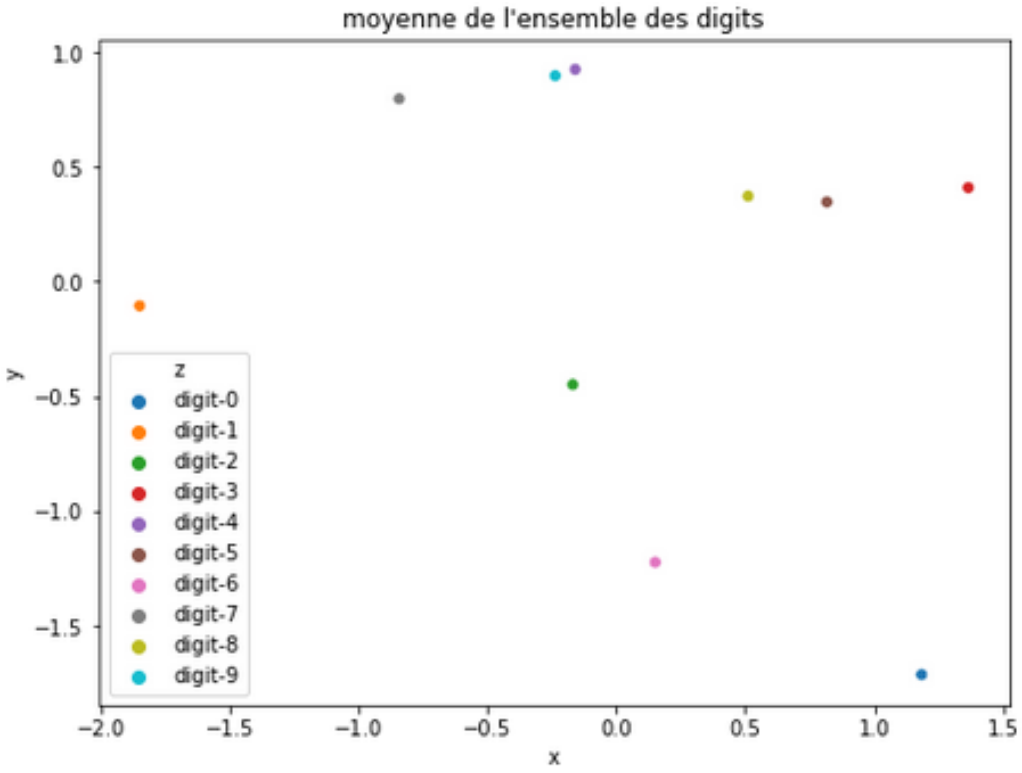

Figure 4.11: Mean value of each space

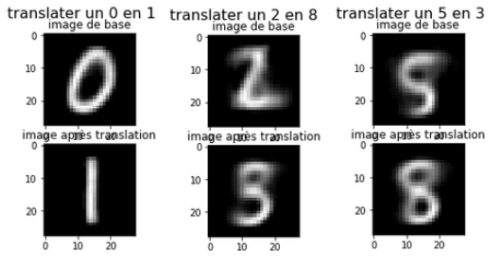

Figure 4.12: Translations for a 2D latent space

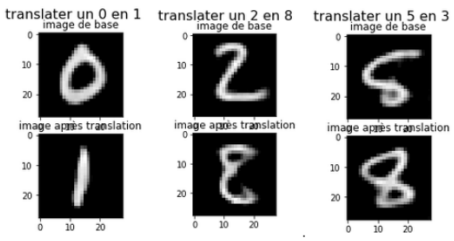

Figure 4.13: Translations for a 32D latent space

Translations in a 2-dimensional space are not satisfactory for numbers that do not have a distinct space. Thus it is essential to choose a sufficiently large number of dimensions. Indeed as we can

see on the figure below, this method works very well in a 32 dimensions latent space. Moreover this method allows us to easily implement adjustable effects, indeed it is enough to multiply the transition vector by a coefficient A to attenuate the effect. So if we test this for the translation of a 1 into a 0, we get a 0 which is more and more closed as a 1 the bigger A is.

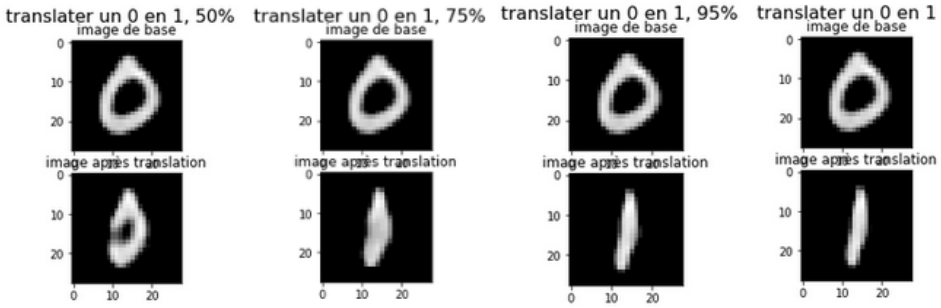

Figure 4.14: Attenuated translations of a 0 into a 1

So we are able to apply an effect on simple images of numbers. Now it remains to see if we can do the same thing on more complex images representing sounds with and without effects.

4.3.2 Convolutional Neural Networks

In parallel to the VAE study, we tried a second approach in order to model an effect and apply it in real time on an audio signal. For this one, we started with a simpler architecture than the VAE but just as efficient, and we reduced the number of effects to only one effect to reproduce. This allowed us to move forward, and especially to be able to do tests at the same time as the realization of the model.

So we decided to start with an architecture composed of convolutional neurons. As it has been previously mentioned, this architecture has several similarities with the VAE, but the advantage is that this one is more widespread, which will facilitate the research and its implementation. Convolutional neural networks are networks inspired by the visual cortexes found in humans and animals. These neurons work by stacking layers of perceptrons. This architecture aims to process limited amounts of information using convolution products. This architecture is mainly used to process images, videos or any other 2 dimensional signal. This can be an advantage for us because it is possible to perform audio processing by working on the short term Fourier transform (STFT) or by working on the MFC (Mel-frequency cepstrum).

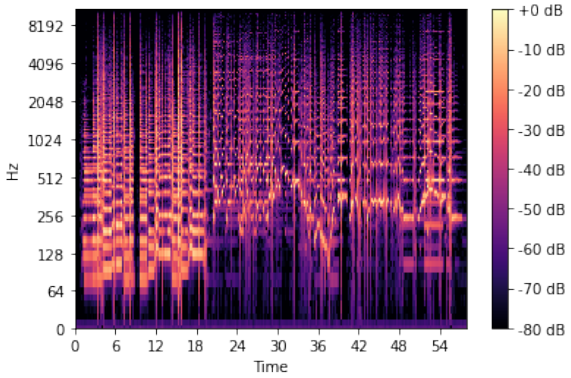

Figure 4.15: STFT of a guitar piece

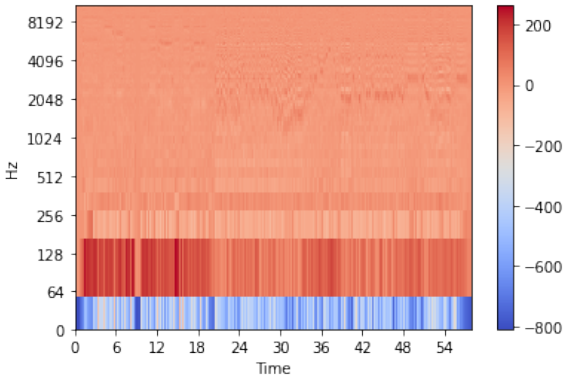

Figure 4.16: MFC of a guitar piece

These particular Fourier transforms allow to generate signals in 2 dimensions where the axes are the frequency for the ordinate and the time for the abscissa. Thus, we end up with a signal comparable to an image, and we can therefore use learning techniques from image processing.

However, before even starting to talk about STFT, MFC, or neural networks, we must first look at the dataset (database) that we are going to use.

4.3.2.1 Dataset - A crucial step

The choice of the dataset is one of the most important things in the implementation of a neural network. A network, even if perfectly configured, would be unable to correctly learn the task it is asked to do with poor quality data. The choice of the dataset used is therefore just as important as the choice of the network architecture.

It is important to know that there is a very large number of datasets available on the Internet, most of which are generally open-source and therefore completely free to access. For musical applications, we will quote for example the *Dataset of NSynth* or the one of *IDMT-SMT_Audio_Effects*, which respectively scan the whole of the notes of an incalculable number of instruments and many audio effects applied to a guitar and a bass.

Unfortunately, these datasets didn't really work for us, because we wanted to know exactly what effect we were going to try to approach. So we decided to create our own dataset.

To do this, we recorded a guitar for 1 minute and added different levels of distortion. Then, in order to increase the size of our database, we applied an EQ filter with different settings. We ended up with a lot of .wav files that you can find on our *GitHub*.

We then divided our files into 200 ms pieces, and organized them as follows :

◇ Training Data :

- X_train :
  - Clean
  - Clean_TrebbleBoost
  - Clean_BassBoost
  - Clean_BassCut
- y_train :
  - Disto
  - Disto_TrebbleBoost
  - Disto_BassBoost
  - Disto_BassCut

◇ Validation Data :

- Clean_TrebbleCut comme X_valid
- Disto_TrebbleCut comme y_valid

After the creation of this database, we could proceed to the configuration of our neural network.

4.3.2.2 Configuration of the model

To configure a neural network, apart from the architecture of the network itself, there are two important things to define to indicate to the network the problem it must solve, and how it must solve it. This is the role of the Loss function and the Optimizer.

In our case, the problem the neural network has to solve is a regression problem. We want the network to learn to reproduce an effect, more specifically a distortion. In other words, we want the network to learn a complex function, representing the distortion.

After understanding this, the choice of the Loss Function is relatively simple, since for problems of this type, there are mainly two:

- MAE - Mean Average Error
- MSE - Mean Square Error

These functions are used to measure the disparity between the actual target value and the value predicted by the model, as shown below:

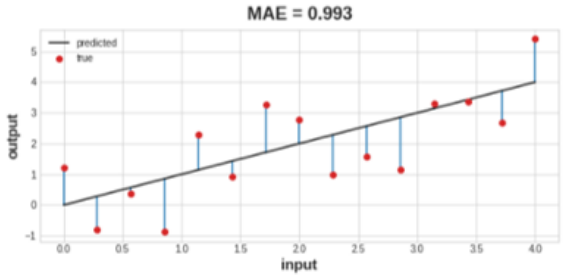

Figure 4.17: Visualization of an MAE

Then, as during the learning process, we try to minimize this gap between the prediction and the target, we have chosen for the Optimizer: a gradient descent (ADAM). For more details on how a gradient descent works, *we recommend this article*.

We also took the time to look at overfitting/underfitting problems, which are very common when working on machine learning. These problems are mainly due to the training of the neural network. It learns too well and specializes on the dataset data, which will prevent it from making good predictions on data that differ from it.
To limit this as much as possible, we have therefore taken care to set up an early-stopping system. This allows us to anticipate the phenomenon of over-learning by stopping the training of the neural network as soon as the average error (MAE) of the validation reaches its minimum. This can be imagined in the following way :

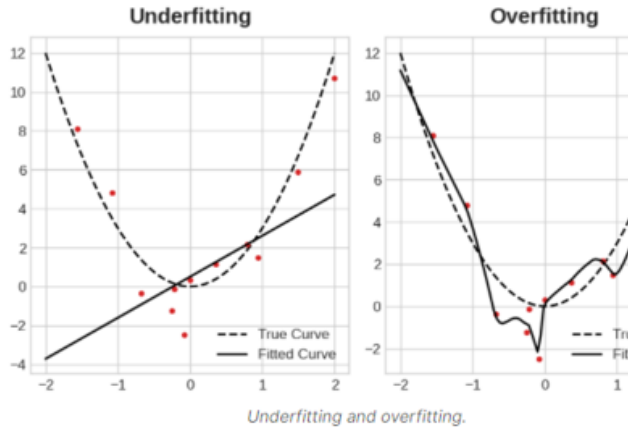

Figure 4.18: Underfitting/Overfitting

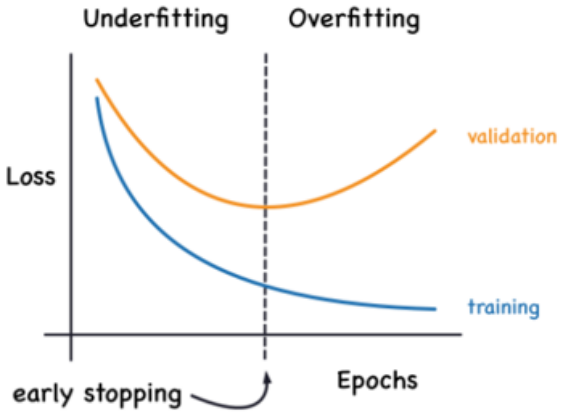

Figure 4.19: Early Stopping

4.3.2.3 First approach - MFCC

After learning how to configure a simple neural network [11], we started with a simple layout of three layers :

1. Batch Normalization : Normalize the input signal to avoid the network to train on too extreme values.

2. Conv1D : Performs a one-dimensional convolution on the temporal components of our MFCC

3. Dense : Connects all the neurons between them and allows to find an output having the same dimensions as the input, these having changed during the application of the convolution.

In this first approach, we decided to use the MFCCs (the coefficients of the MFC), because according to our research, they are widely used for AI speech recognition tasks, especially because they allow a faster learning than with data obtained for example with a STFT.

We have thus launched a first learning on the MFCC of 1160 audio samples and after a relatively short learning, linked to the rather simplistic architecture of our network and to the use of MFCC, we obtain the following result :

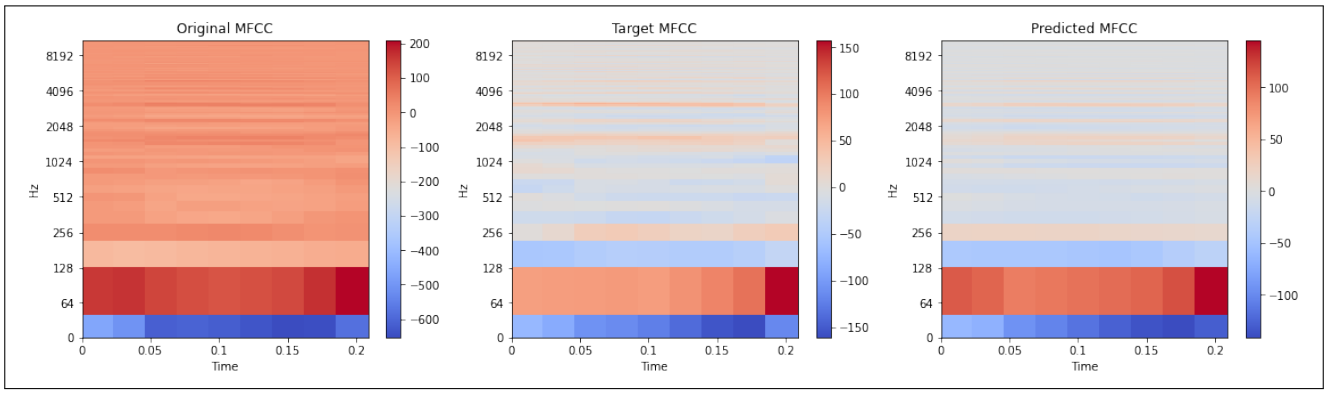

Figure 4.20: Observation of a sample following the first MFCC-based learning

We can see that the prediction is extremely close to the target, even if we can recognize a slight influence of the original sample (thus the undistorted sound) in the low frequencies during the first 50 milliseconds. Not bad for a network with only 360 parameters, is it? What we are interested in here is mainly the sound, so we can calculate the inverse MFC in order to listen to our predictions:

Play : Original Play : Predit

Listening to the prediction, one might think that the model was not able to generate the expected distortion and is only damaging the signal on which it predicts instead of applying the requested effect. After some research, we decided to calculate the MFC of our original audio and to calculate the inverse MFC to see the effect of these two conversions on the audio signal. We then realized how poorly this type of transformation works. This led us to concentrate our efforts on the STFT approach. After developing a more elaborate architecture, we decided to try using it with an MFC, since despite the sound deterioration it introduces, we should be able to hear if a distortion has been applied. In addition, since the models using MFC converge faster than those with STFT, they allowed us to test different architectures faster and with fewer resources.

4.3.2.4 2nd approach - STFT

During our research, in addition to MFC, we found many Machine Learning resources that used STFT. Not knowing which one to use, we decided to work on these two approaches in parallel.

So we started by implementing the same architecture as for MFC, i.e. a simple three-layer network consisting of: a Batch Normalization, a Conv1D, and a Dense. By running the same training, we obtained the following result

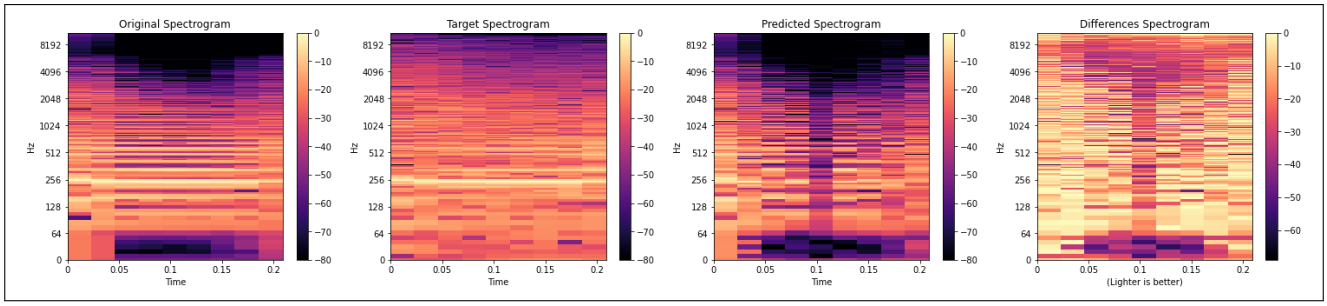

Figure 4.21: Observation of a sample following the first STFT-based learning

In order to visualize more quickly the differences that there can be between the target and the prediction, we made the choice to draw a spectrogram corresponding to the difference between these two. It is thus necessary to keep in mind that this one is not representative of any signal, it is just there to visualize quickly the differences between the target and the prediction. We can then notice that compared to the MFC where the result was very close to the target, for the STFT the result is much less convincing. The distortion is very little present, even almost non-existent. We notice however that the predicted signal is more faithful than with the MFC, which is more pleasant to listen to:
The results were better than with the MFC, but we were still far from what we expected. We then

Play : Cible Play : Predit

continued our research, and it is by looking at Mr Martinez's article [5] that we understood how we could greatly improve our network architecture.

We then modified our architecture by adding three layers of Conv1D, each followed by a layer of MaxPool1D. This structure is not unlike that of LeNet.[1].Here, our MaxPool1D layers take two samples in a row and keep only the largest value, thus dividing the parameter size by two. We then launched a training to see if our architecture brings an improvement in the learning of the neural network on the distortion effect. We then obtained the following results:

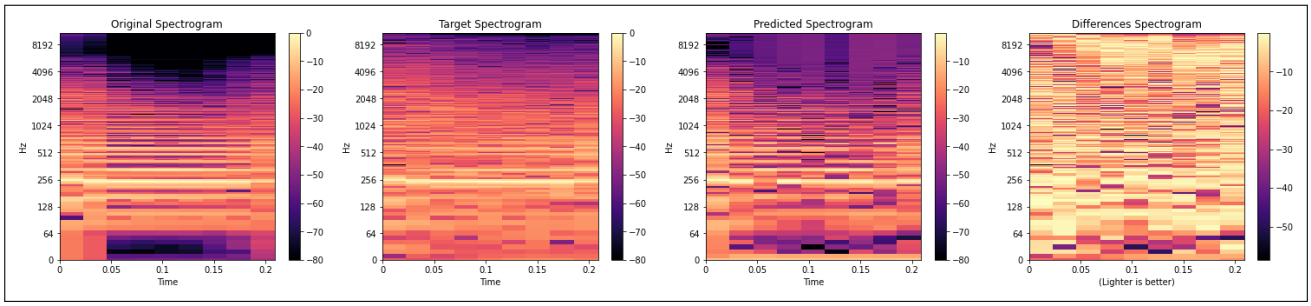

Figure 4.22: Observation of a sample obtained with our 2^{nd} architecture

These results are more convincing than those obtained with the first architecture, even if there is still some way to go before we can perfectly replicate our distortion. We can notice a clear improvement in listening:

Play : Cible Play : Predit

By observing the evolution of the Loss Functions over the years, we can already notice that the neural network is learning something. This is already a good thing. Moreover, we can notice that the implementation of our Early Stopping seems to work, because there is no appearance of over-learning.

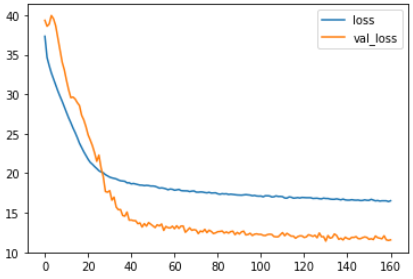

Figure 4.23: Loss Functions of our 2^{nd} architecture

We then continued in this approach by performing convolutions on the spectral and temporal dimensions of our data, generating much larger models with more than ten million parameters. These allow us to get closer and closer to the expected distortion:

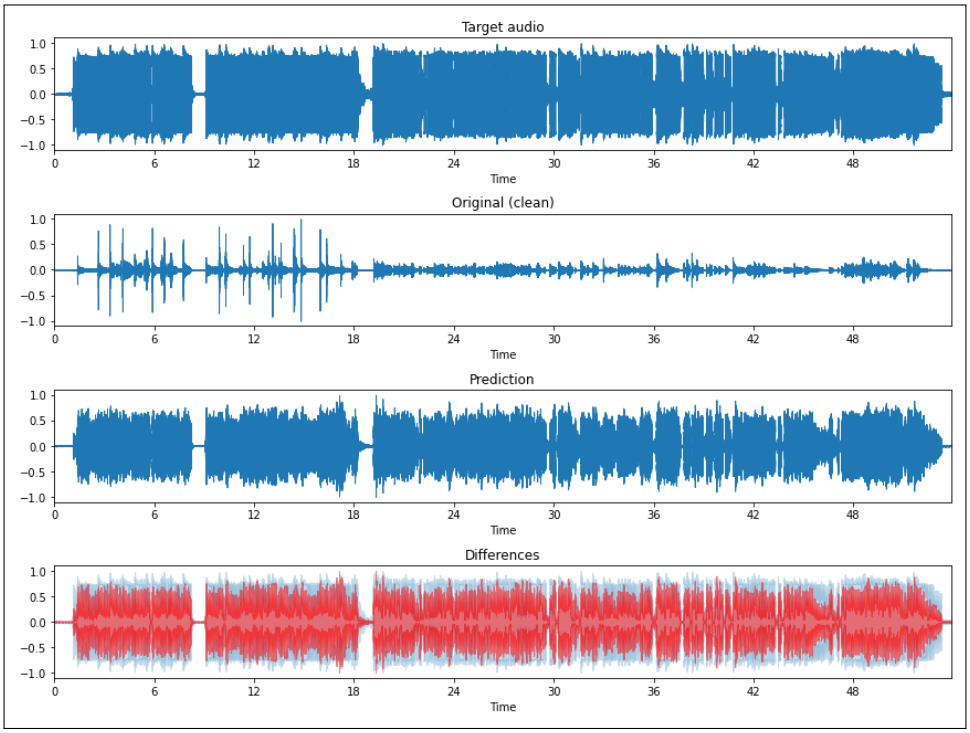

Figure 4.24: Temporal signals of our new results

Even if the temporal signal does not completely match the expected one, the spectral components are well present: the spectrograms of the prediction and the target are almost identical.

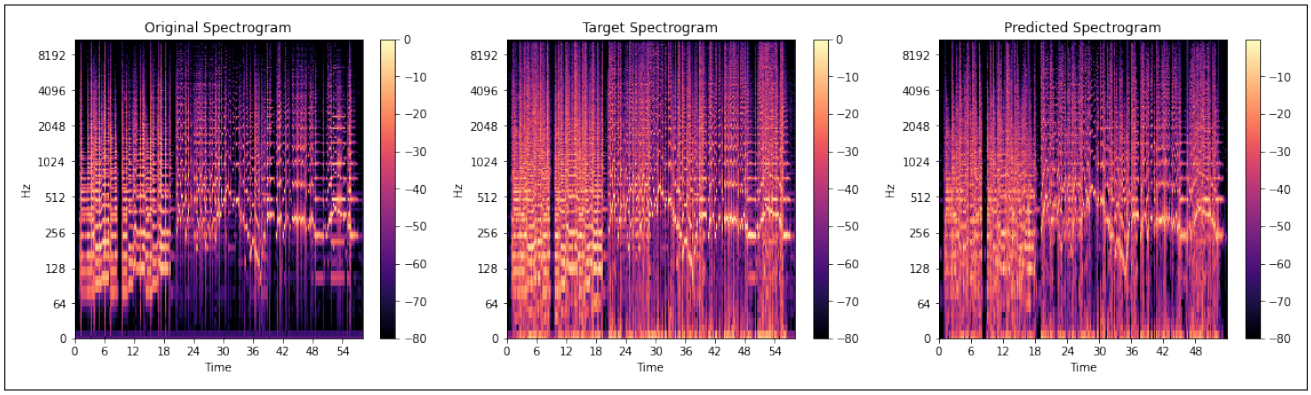

Figure 4.25: Spectrogram of our new results

We can finally listen to these new predictions which are the best we could get with this convolutional neural network approach :

Play : Cible

Play : Predit

4.3.3 Searching for parameters and audio effects

A third approach was parametric research. Indeed, by refocusing on the original idea of the project, and facing the complexity of the other methods studied, we opted for a more direct approach. The idea was to give the ToneCrafter a song by a certain artist and it would automatically adjust the effects to approximate the sound in the most faithful way. To do this, we first had to figure out how to implement the different effect pedals on Python, but more importantly how to implement the search for the parameters of each effect.

There are different types of effects used by guitarists. First of all, there is distortion, an effect that has already been discussed earlier. But also *Reverb*, *Delay*, *Chorus*, *Tremolo*, and the various filters. Each one can have its importance, especially if the guitarist you want to imitate uses them. That's why it's important to implement a myriad of different effects if you want to get close to the desired guitar sounds.

The parameter search method is based on the use of spectrograms, so we transform a sound of a few hundred milliseconds into an image. The interest of this is that we can observe more easily the sound we are listening to to better understand the appearance of some harmonics for example. But especially, to be able to compare different sounds between them more easily.

4.3.3.1 Implementation of effects

Approaching effects and analogical phenomena in Python is not always easy. Indeed, it is necessary to approach them with mathematical functions, or with a more specific signal processing. In particular for distortion, which is a very singular effect. Indeed, distortion or its cousin overdrive are characterized by the loss of linearity in the sound signal. The question then arises of how to recreate this loss of linearity. We then found formulas that will allow us to approach this so singular sound

The function *Distortion* is defined by :

$$f(x) = \frac{x}{|x|}(1 - e^{\frac{a \cdot x^2}{|x|}})$$

We now represent it for an input level ranging from -1 to 1 :

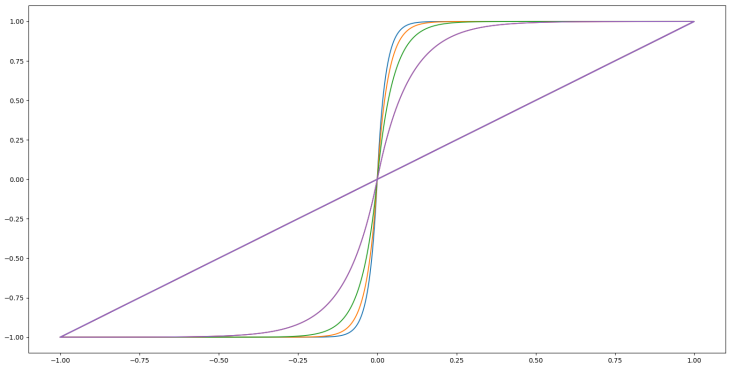

Figure 4.26: Graphical representation of the distortion for different gain values

ONote that this is hard clipping, i.e. a rather violent distortion of the signal. Moreover, for a zero gain, the function returns the initial signal.

The function *overdrive* is defined by :

$$f(x) = \begin{cases} 1 & \text{pour } \frac{2}{3} \leq |x| < 1 \\ \dfrac{3-(2-3x)^2}{3} & \text{pour } \frac{1}{3} \leq |x| < \frac{2}{3} \\ 2x & \text{pour } -\frac{1}{3} \leq |x| \leq \frac{1}{3} \end{cases}$$

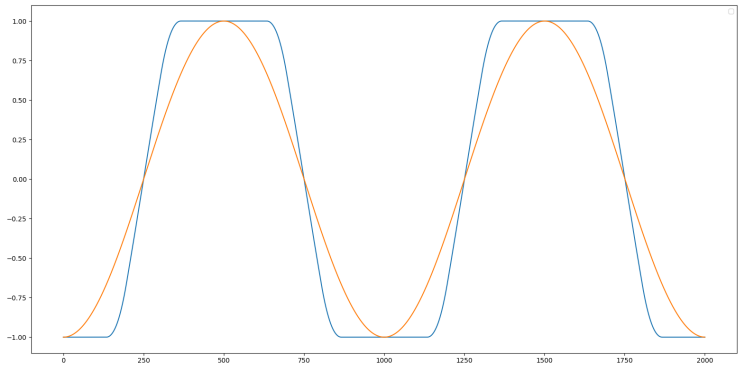

Figure 4.27: Graphical representation of the overdrive

By comparing with the yellow curve we can see the loss of linearity due to the function. Note that to simplify the study we have proceeded with a symmetrical distortion of the signal, but in many setups, it is advisable to have a more important clipping for positive signals than negative ones.

But distortion and overdrive are not the only effects used by guitarists. Some are fond of chorus and reverb like David Gilmour, the guitarist of Pink Floyd. Delay, tremolo and wah-wah pedals are also used. Finally, to give a certain color to their sound, guitarists also use equalizers, which are filters, to simulate them we will use high-pass, low-pass filters etc...

On Python, apart from the filters provided by certain libraries, such as the *Butterworth* filters, most effects use the decomposition of effects in the form of float lists. Indeed for the *Delay*, the output signal $y[t]$ is composed of the input signal $x[t]$ but also the input signal $x[t-d]$. Where d is variable, and indicates the waiting time between a sound and its echo. For the effect *Reverb*, it is more or less the same operation but with the help of a smaller parameter d.

As for the *Flanger* effect, we apply the same process, but this time the parameter d varies according to a very specific function. Indeed we have :

$$d = f(t) = A \cdot sin(2\pi \cdot t \cdot rate \cdot period)$$

Where A is the amplitude of the variation of d (of the order of 50) *rate* is the sampling frequency and *period* is the period of oscillation of *d* Finally as the *Chorus* is only in principle the use of several *Flangers* where we vary the parameters *A* and *period*, we will use 3 of them afterwards. After testing, and listening it is largely sufficient.

Finally, we also implemented a pink noise generator, more pleasant and more adapted than the white noise which was originally intended to imitate the breath created by some effect pedals or amps. Finally it will serve us more to observe the limits of our functions when faced with the addition of noise.

4.3.3.2 Finding the parameters of the effects

The second part of this method, and the most important, was the implementation of a search algorithm for each effect. For this, we had many questions. How to make a program recognize the proximity or

not of two guitar sounds? Do we need all the effects? In what order should we place the pedals? Do we have to play the melody perfectly so that the program can recognize the effects used?

First of all, we looked for the most efficient way to compare a single note to another. For this, the use of spectrograms seemed obvious but not sufficient. Indeed, for low frequencies, like the one of the low *Mi* of the guitar : 82 Hz, we notice the necessary precision (of the order of the half hertz) in order to represent well the sound of a guitar. This implies notably longer calculation times, because the precision of the spectrogram is increased.

Then came the problem of comparing sounds. To compare two sounds, it is necessary first of all that it is the same note, or at least that the note is very close. Taking this into account, if we took as an example the sound of Brian May, the guitarist of Queen, to want to imitate, would it be necessary to replay perfectly the guitar solo and then to compare it to the original? Of course we had to find another solution. So we had the idea to segment the audio file of the Bohemian Rhapsody solo so that each segment contains only one note. Then we will compare each segment, of a certain note, to the sound of the same note of a guitar without effect.

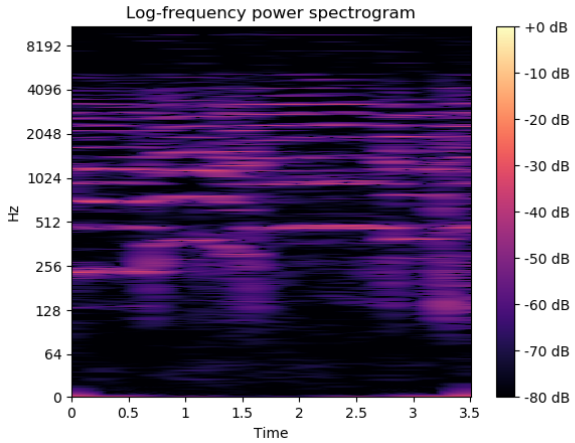

Figure 4.28: 3.5s of Brian May's solo

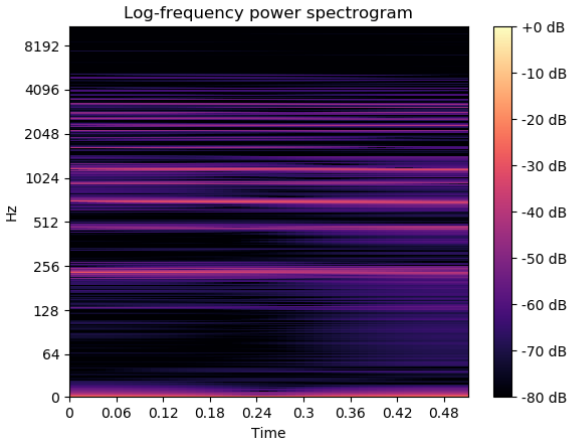

Figure 4.29: 0.5s of Brian May's solo

Note that these spectrograms, were realized after application of a series of notch filters. These filters attenuate all the frequencies between two notes. That already allows to remove noise, but also to better observe the spectrograms, these filters will be useful for the continuation. We notice that there is no variation of notes. Indeed, the lines of the spectrograms are well horizontal, one thus remains well on the same frequency. However a hash of 500ms allows sometimes that two appear, it is an axis of improvement of the program. However, as we will only use a few sequences of the 0.5s solo, this will be sufficient to continue.

Now that we have the singular notes to study, we still have to find something to compare them with. For that we needed a database containing all the notes of the guitar. This database was provided to us by the *Fraunhofer Institute for Digital Media Technology IDMT*, it includes recordings of several instruments, in order to make signal processing. We thus used the sound of a *Fender Stratocaster*, it is one of the most widespread model and its sound is relatively well declined with effects. This database is composed of a file for each possible note of the guitar. However a note on the guitar can be played on different places on the neck. There are sometimes up to three ways to play a note on a guitar.

Once our database is ready, we need to know how to compare the selected note to the right file in the database. To do this, we must already know which note it is. We have therefore created a program

which, for a given sound, returns the note and the files in the database corresponding to this note (the names of these files show the box to be pressed and the string to be plucked on the guitar in order to play a given note).
When we take the selected note from Brian May's solo:

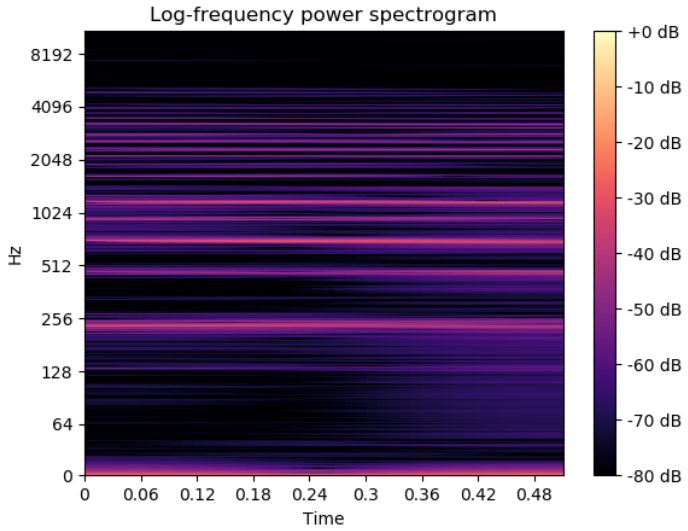

Figure 4.30: Graphical representation of the overdrive

We observe the appearance of many harmonics due in particular to the distortion of the guitar, it is what we will try to reproduce. However, a frequency seems to emerge around 250Hz.

The algorithm returns to us: *La#2*=233,08 Hz, which corresponds well to what we observe.

We classify the musical notes 12 by 12, this going from *Do* to *Si*, then according to the octave we add a number in 1 and 12. Here the first note of the solo of Bohemian Rhapsody is an *La#2*. To this note correspond three files of the database, we restrict ourselves to the second one (*G53-58308-1111-00035.wav*) closer to the note to imitate. The note detection algorithm being certainly very basic, it remains nevertheless very effective for our study. But we had the idea to improve it in order to avoid possible errors due to a too big distortion of the note, too important effects or a too important noise. We can now observe the two spectrograms of the two selected notes:

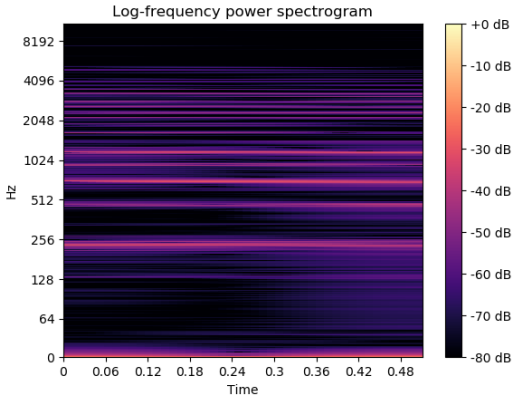

Figure 4.31: First note from Brian May

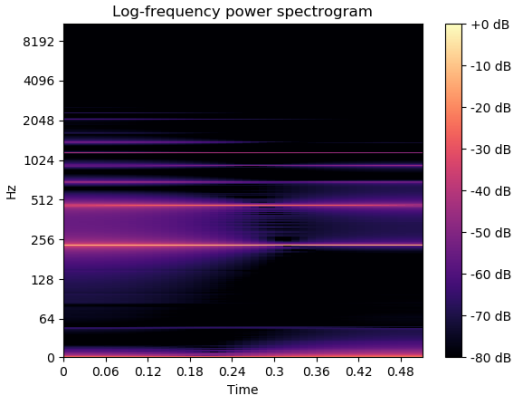

Figure 4.32: Corresponding clean note

Note that the two notes are not taken at their very beginning because it is what we call the attack,

it is the sound produced by the string as soon as we scratch it. It is a sound that depends on too many parameters to be used in our study. Indeed, Brian May, for example, does not use a pick to strum his guitar but an English coin. This gives a more metallic and aggressive sound to the attack. It is the same for each string and pick that a guitarist could use, their combination would always give the unique tone to the attack always. We observe the same intensity spikes at 233.08 Hz and 566.16 Hz.

Finally, now that we know which notes to use, we still have to compare them. For that we base ourselves on the spectrograms, being images, we will simply make function of inter-correlation, by parameterizing each iteration of the algorithm with different values. When we reach the image closest to the one from the Brian May solo, we return the parameters used to obtain this image.

In view of the precision of the spectrograms, the amount of calculation to be carried out is relatively high, that's why we restricted ourselves to search for the parameters of the functions of *Distortion* and *Overdrive* in a first time. Then to extend this search to each effect. Note that the order of the effects being important, we use the convention used by most guitarists which defines the order of the effects to apply: *Compression - Wah Wah - Overdrive/Distortion - Equalizer - Chorus/Flanger - Delay - Reverb*.

The results obtained are quite far from what we had hoped for at the beginning, except for distortion or overdrive, the possibilities were so immense that we had to find ways of converging more quickly towards a satisfactory result, notably by restricting the possible range of parameters.

However, by starting with the *Distortion/Overdrive* found and by adjusting some other pedals, we arrive at relatively coherent and pleasant results (the *Overdrive* gives the best result).

4.3.3.3 Realtime implementation

One of the goals of this project was also to play guitar with our effects in real time. That's why our Python algorithms were a good match for this feature. Indeed, each program, representing each effect, returns a list of floats that is directly handled in real time. Python being certainly not the language adapted to real time, for reasons of saving time, we were able to test our functions.moreover we had already made this program for the DDSP part that allowed us to save a little time.

To say that this algorithm works in real time is an abuse of language. In reality, it records 1024 samples, passes these 1024 through all the effects functions, and outputs the resulting sound. Moreover, while we are listening to these 1024 samples, the algorithm is recording another 1024 samples. Since this program is running in a continuous flow, we do not hear any interruptions.

However, several problems occurred when we used certain effects such as the *Delay*, the *reverb*. These effects use past samples, sometimes too far away (more than 1024 samples away from the present), so there is a loss of information and we hear cuts. To remedy this we added a buffer, which records all the samples that can be reused later by the different effect functions.

Another big problem was the quality of the signal sent by the guitar. To test the limits we added pink noise, and for a value of about a tenth of the guitar level the sound was already well impacted.

Chapter 5

Hardware development

QUENTIN WACONGNE, HECTOR RICHARD

5.1 Specifications

We started from the fact that it was very difficult, especially for novices, to correctly set up an effect pedal in order to obtain the desired sound. Moreover, an effect pedal is an object which must be transportable and robust in order to be used in conditions of strong mechanical constraints, like on stage during a concert.

Our goal is to create a robust, transportable and plug and play hardware support, housing electronic components allowing to perform audio processing in real time.

Our target would be musicians of all levels and horizons, current users or not of effect pedals and wishing to sound like one of their favorite artists without having to go through a tedious tuning stage.

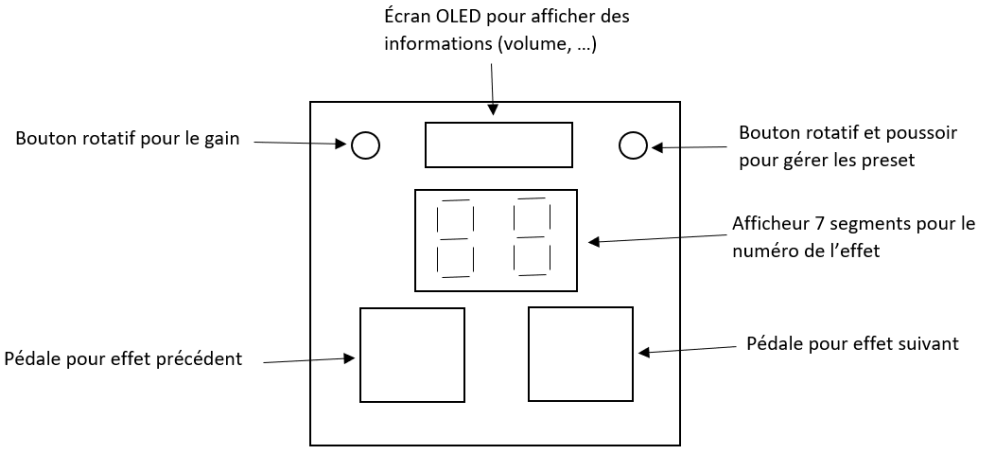

Figure 5.1: Prototype design of our pedal (dimensions 14x17cm)

5.2 Pedal design and functionality

First of all, our design idea for the pedal is strongly inspired by existing pedals. Indeed, we want our product to have the same robustness as those pedals so that it can be used in more demanding situations from a material point of view, for example on a stage during a concert.
As you can see on the diagram above, we want it to be quite simple with 2 pedals to switch from one set to another, 2 displays, 7 segments to show the current set, an OLED screen and 2 rotary knobs to adjust the gain or manage the presets more precisely. The case will be rectangular with the following dimensions: length 17cm, width 14cm, height 4cm. Inside the box, we can find the control part of the system. One finds there mainly a microprocessor, a codec and the various memories.
This configuration allows to have a simple, reliable and fast interface to use. Indeed, an input like a simple push button can have several functionalities. One can imagine an action with a single press,

another action with two short presses, and yet another action with a long press. This way, we can save on pins and ports, but above all facilitate the management of interrupts and events by the microprocessor.
All this logic part will be in charge of managing the inputs and outputs, including the buttons of the interface, and the audio acquisition, calculations in order to apply the effects on the input signal,

5.3 Choosing Components

The processor:
For the microprocessor, we chose an STM32H743ZITx. This microprocessor has 2 Mb of flash, 1 Mb of RAM and a clock frequency of 480 MHz. This is a high performance microprocessor because for real time sound processing, a large amount of RAM and a high clock rate are preferable. Another interesting feature with this microprocessor is that it is compatible with the IDE compiler "CubeAI". This utility allows us, from a file containing the model and the algorithms of our artificial intelligence, to make an implementation in C language. Thus, we have an optimized architecture capable of executing low-level tasks. On this microprocessor, there are also circuits for hardware acceleration like ASICs. This means that if the right data structure is applied, it is possible to mobilize several dedicated circuits, and thus to be faster. One of the circuits we are most interested in is the 'X-CUBE-AUDIO-F7'. The "X-CUBE-AUDIO-F7" is an audio processing component for the STM32F7 series and consists of a complete set of efficient and high quality software libraries dedicated to hardware acceleration, ready to be integrated into many types of audio devices. All modules come with 32-bit internal processing and support 16- or 32-bit I/O buffers. This makes it possible to have access to a "Bass manager", an intelligent volume control and a "gain manager". We find ourselves not only with software bricks that we do not have to realize, but especially with software bricks guaranteed optimized by the manufacturer.

The codec:
We have chosen the TLV320AIC23B. This audio codec has a sampling frequency of 96 kHz. We chose this one because we wanted a codec that was fast and efficient enough because, like the microprocessor, we need components with sufficient computing capacity and memory to be able to carry out signal processing in real time. This codec also has several serial ports. So we can give instructions to the codec.

The power supply:
In order to be able to propose the most flexible solution, and the easiest to use, it was chosen to use a Jack power cable. This cable has a round plug, which makes the connection very simple. Indeed, there is no need to worry about the direction in which the cable is plugged in. The jack cable delivers 5V that we regulate in 3,3V as soon as it enters the circuit. This solution allows us to supply components with a 5V voltage, but also other components with a 3.3V supply voltage.

5.4 PCB realization

5.4.1 Drawing the schematic

After having chosen our components according to our needs and chosen the design of our pedal, we made the PCB that will contain them on the Eagle software. In a first step, it is necessary to make the "schematic" of our project, this one will give us a global and schematic visual rendering of the whole. Then with the help of the software, we will be able to generate the "board". On it, we will put our different components like resistors, capacitors, etc. necessary for our project. Once we have

put all our components, we need to link them together either by wires or by labels (two wires with the same name will be automatically linked). Labels greatly increase readability. To further increase readability and make the whole thing easier to understand, it is useful to intelligently group different components into "modules" that we can name. For example, we have divided our project into several modules, as explained below.

The main part includes our processor and several components allowing its good functioning such as: the quartz oscillator and its assembly serving as a clock, the reset button and its anti-bounce circuit, LEDs that will allow us to perform tests and various terminals for data flow and control (respectively I2S and SPI which we will specify in the communication between components):

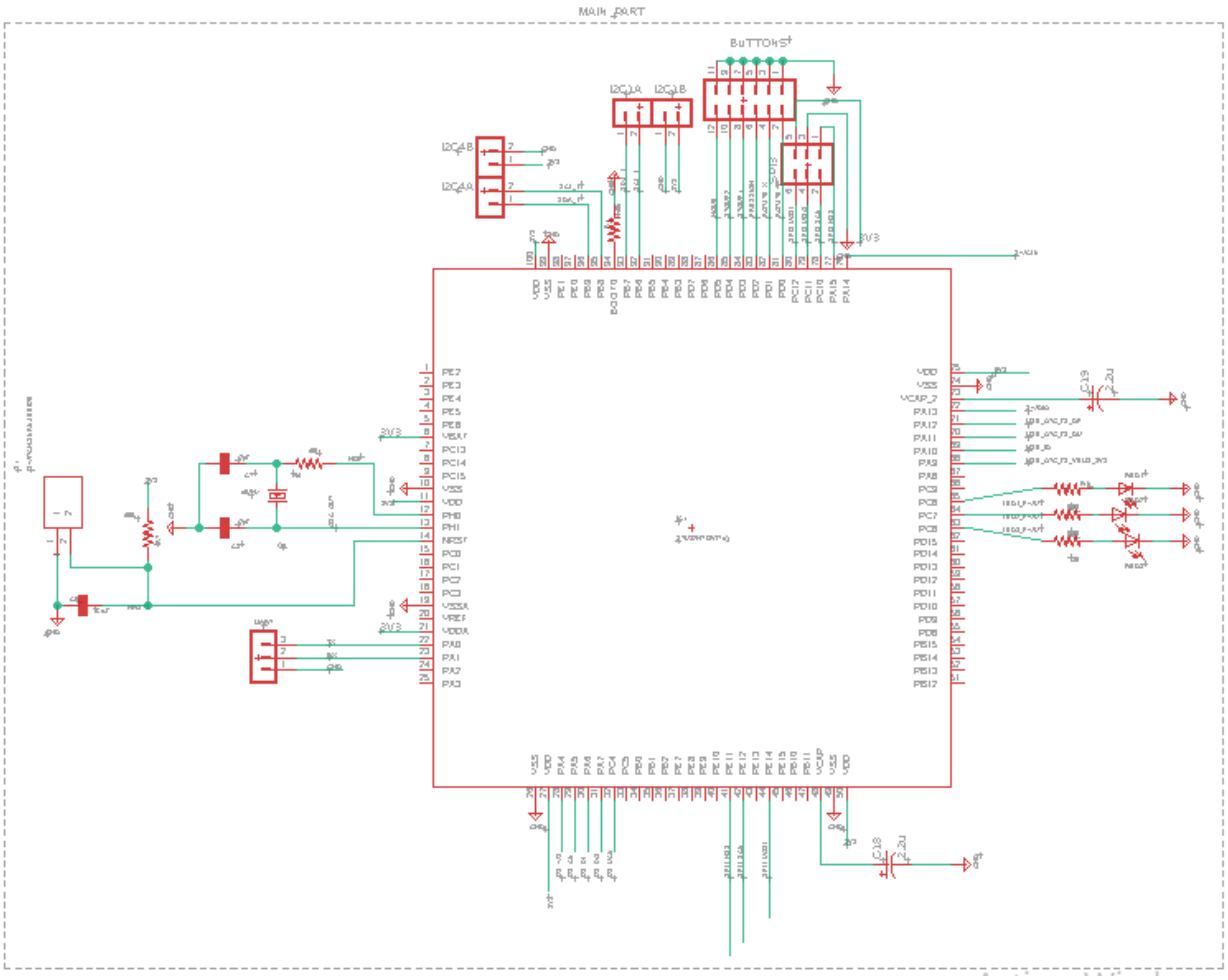

Figure 5.2: Central part of our EAGLE schematic

The second biggest part of our circuit board will be our audio codec and the elements necessary for its operation. On this block, we also have our audio input and output via two jack ports that will allow us to acquire our raw audio signal and then play it back after applying the desired effects.

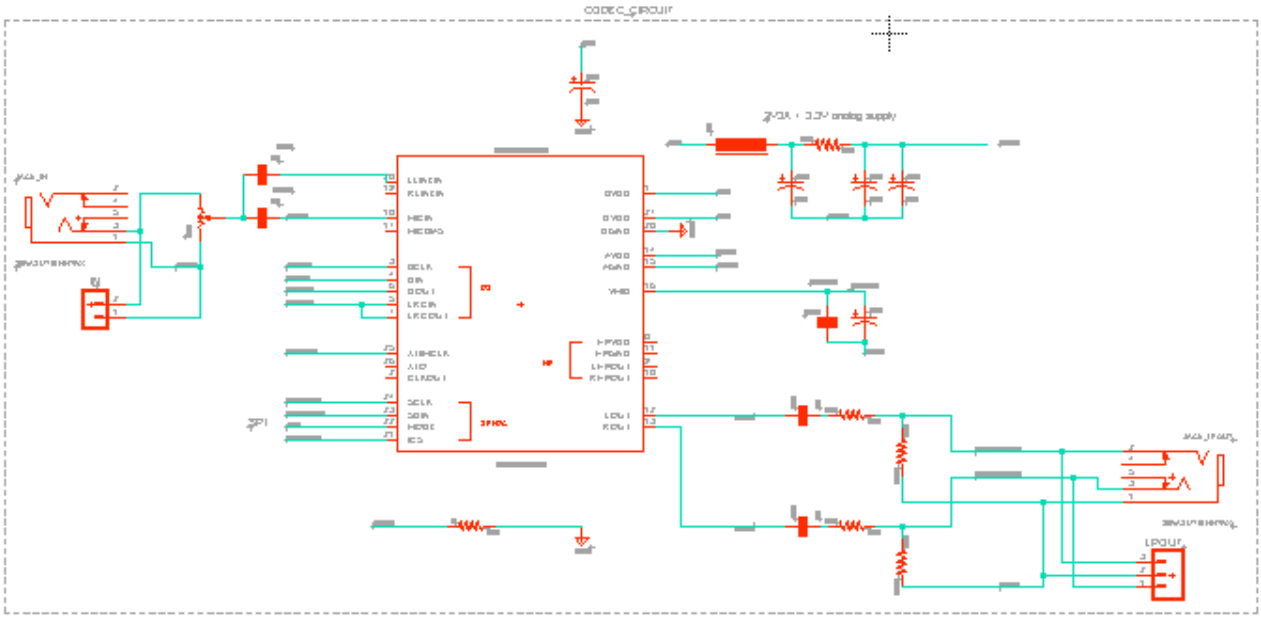

Figure 5.3: Audio codec bloc

Then, we have the power supply part including our power supply at 5V lowered in 3.3 using the voltage regulator TLV7553PDBVR as well as decoupling capacitors and a Bulck capacitor whose utility will be explained later (routing and EMC part)

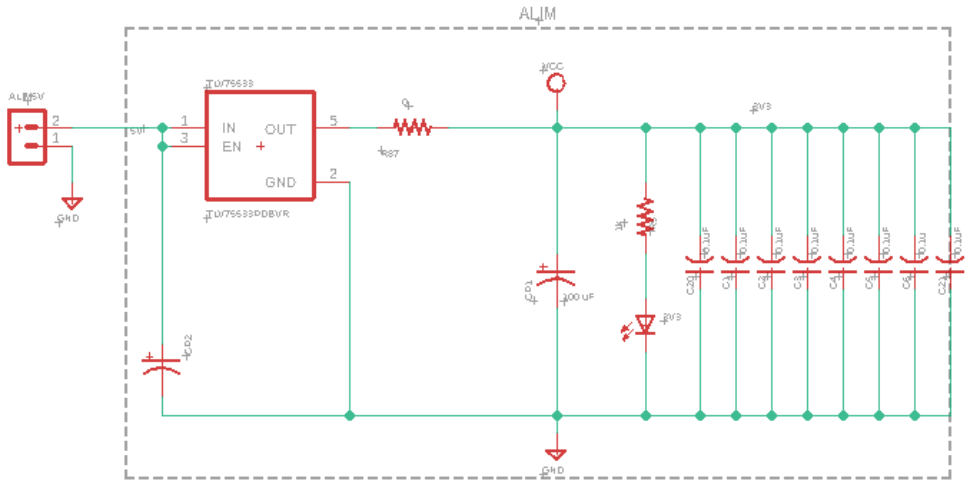

Figure 5.4: Alimentation 5V to 3.3V

Then, we have the USB connector, which will allow us to connect the ToneCrafter to a computer in order to update it or to load new presets.

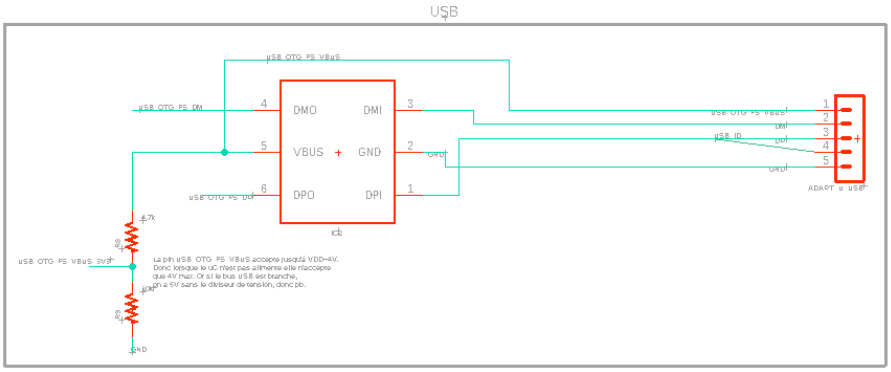

Figure 5.5: USB connectics

Finally, we have a small module "STlink debug" that will allow us to communicate with our processor, reprogram it, and perform tests.

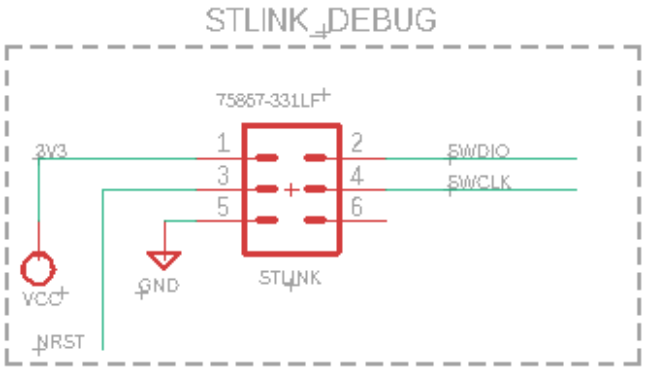

Figure 5.6: STlink debug

5.4.2 Routing and electromagnetic compatibility

First of all, our PCB must be compact enough to be easily integrated into the case. So we decided to use a 6.8x6.2cm PCB for several reasons. Simply, because of the number of components and the complexity of the routing, we opted for this technology. There will be a 3.3V power supply layer, a ground layer and two layers (Top and Bottom) for components and routing. This greatly simplifies the routing by allowing a connection to the ground, or to the direct power supply thanks to via (these are "wells" allowing to connect two different layers between them). This avoids excessively long tracks to get the ground or the power supply. It also avoids potential crossings between different tracks by passing from the top layer to the bottom layer, i.e. by passing one track under another. Moreover, having a ground plane also allows us to limit interference. Indeed, we work at high frequency because we want to carry out audio processing in real time. At high frequencies, it is very important that the return signals take the most direct path possible to ground in order to limit interference as much as possible. Moreover, having a ground plane allows us to have a lower impedance than a simple track and therefore offers a better capacity of absorption of noise peaks that can appear on the ground during the changes of state of the digital circuits.

Now we need to place our components before starting the routing. To do this, we will use our previously defined blocks and some tips. First of all, we place in the center the heart of our pcb, that is to say the main block of the processor because it will be connected to the different blocks and must therefore be placed in the center (note that on figure 5.7, the name of the processor is not the same, but their footprints are the same. We therefore use the one described in the component selection section). It is necessary to think of placing the quartz oscillator as close as possible to our processor because too much length of track can cause a distortion of the clock signal very high frequency.
Then, we place the codec block below the processor. For the input and output jacks, we will place them side by side on the lower left edge of the board so that we can easily plug our instrument in and our amps out.
The power supply will be placed on the right side of the pcb. There is a subtlety here because this block contains the decoupling capacitors. These must be placed appropriately. Indeed, these capacitors are an important element for the EMC insofar as their purpose is to serve as a local energy reservoir that will reduce fluctuations in supply and ground potential and reduce noise, as we can see on the graph below.

We will take capacitors with a capacity of 0.1 uF to decouple because we work in high frequency. They are to be placed as close as possible to the components to maximize their efficiency. We will also have a "Bulck" capacitor which will be placed downstream of our voltage down converter. Its role will be to maintain the DC supply voltage constant especially when there will be strong current calls (for example when many circuits switch at the same time). We will take here a 100uF capacitor because it will not intervene in high frequency.

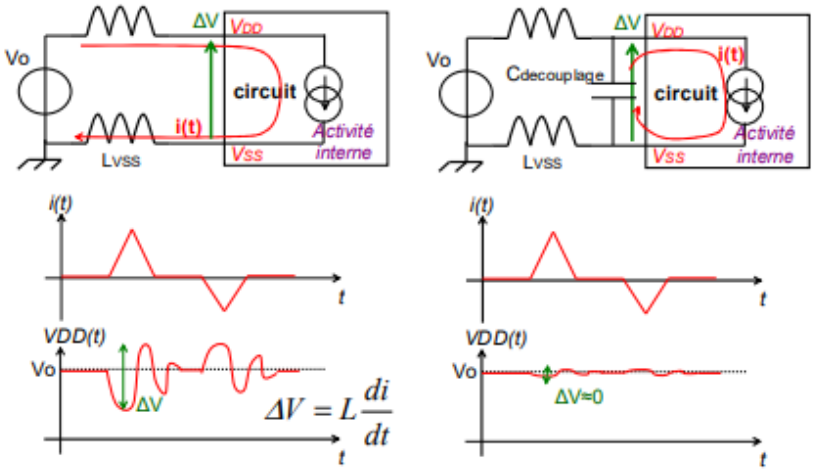

Figure 5.7: Effect of a decoupling capacitor: undecoupled circuit (left) and ideally decoupled (right)

The USB connection block will be placed on the upper right part of our pcb, just below the st-link block which will be in the upper right corner.

Once the different components have been properly placed, we must carry out the routing, i.e. the drawing of the different tracks that will connect our components. First of all, we must avoid as much as possible angles of more than 45°, tracks that are too long (strong parasitic impedance), tracks that are too close together (crosstalk), or else favour wider tracks for high current tracks to limit parasitic resistances and inductances, all of which can lead to EMC problems. Then, it will be necessary to use via to connect our components directly to the power and ground planes. It will be useful to use the bottom layer in order to be able to "cross" tracks by passing one under the other to avoid long bypasses and to simplify the whole.

Once all our tracks are done, the rendering of our "board" on EAGLE is as follows:

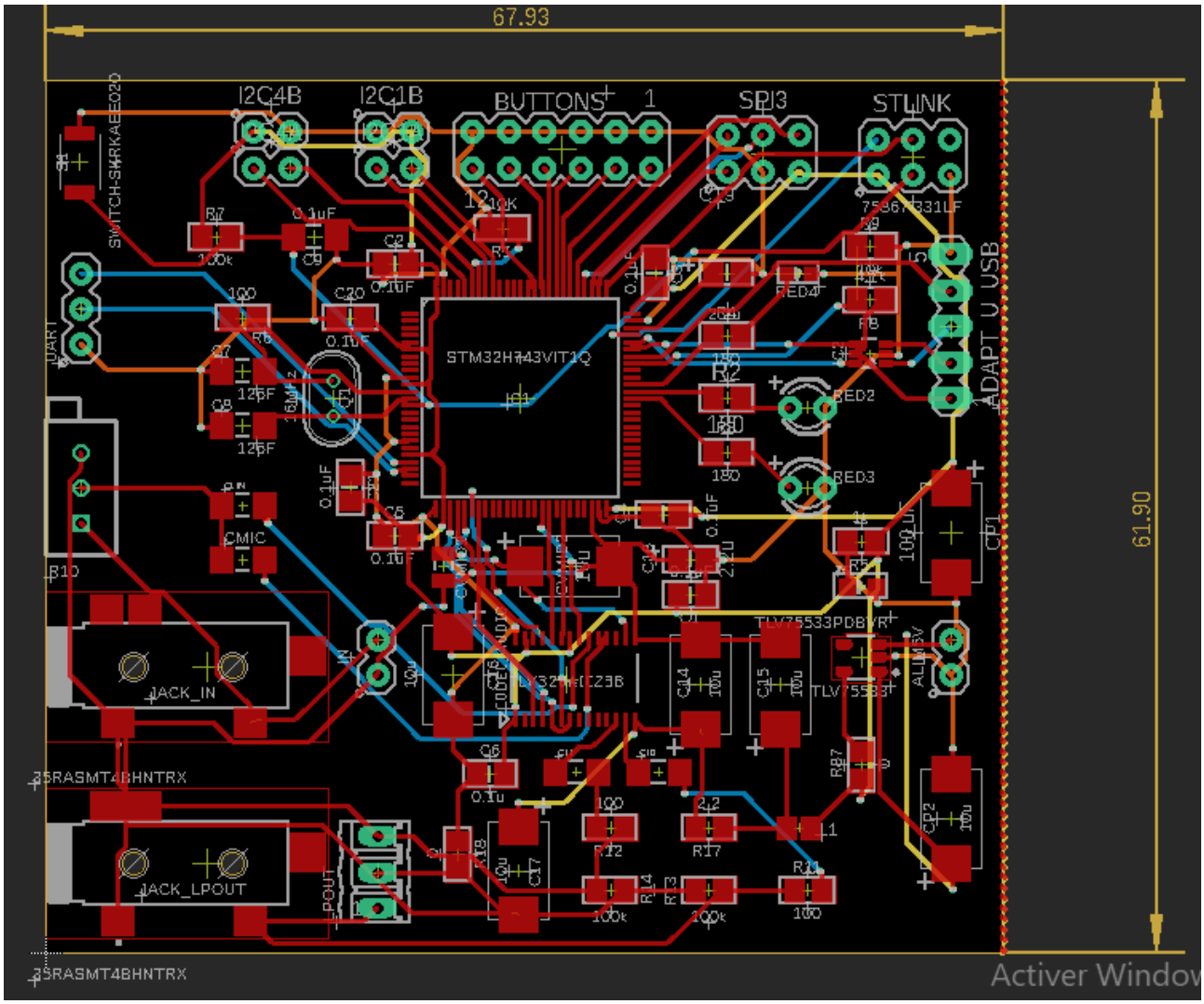

Figure 5.8: Final schematic of our PCB

5.4.3 Communication methods

The DMA:
Meaning Direct Access Memory, DMA is a feature that allows a direct, bidirectional sending of data between a device and the memory. It can also be between 2 memory blocks. The microprocessor does not necessarily intervene, and the data transfer is thus accelerated. The main purpose of DMA is to take care of large data transfers, thus freeing the processor's load by a diverted channel. The processor is thus freed from the transfer tasks and can devote itself to other tasks, thus gaining in capacity and speed of calculation.

THE I2S:
The I2S bus is a standard bus for the transfer of data between digital audio equipment. It is composed of a 'bit' clock signal, a 'word' clock signal (to indicate the beginning and the end of the word) and a data bus (SD: Serial Data) containing the word. One uses a bus I2S between the processor and the codec to transmit the data. The codec which is slave sends by the bus I2S the numerical data which it has just translated to the processor, which is master, so that then the processor returns these data to him with the desired effects applied.

the SPI:
An SPI transmission is a simultaneous communication between a master and a slave. The master generates the clock and gives requests while the slave responds to the master's requests at the clock rate. The advantages of this bus are that the SPI bus has a higher throughput than the I2C bus and also that the slaves use the clock of the master and therefore do not need their own oscillator. Here, our processor will send requests to our codec in order to parameterize the various converters of the codec.

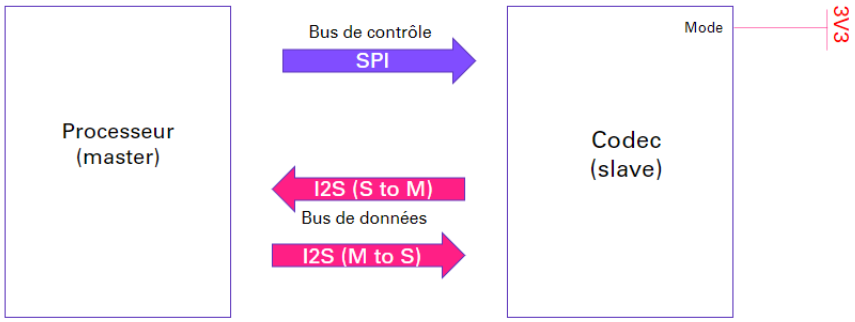

Figure 5.9: Communication mode between the processor and the codec

5.4.4 Power consumption and cost

In order to give a relatively precise idea of the financial cost to realize the hardware support, we realized at first a calculation of the cost of realization of the printed circuit and its components. This calculation is rounded down and the prices of the components and the shipping costs are those of the company Farnell. For the price of realization of the PCB, it is about a Chinese company what justifies the low cost. We thus obtain the following calculation:

	A	B
1	composant	prix en euro
2	stm32H743ZIT (Farnell)	13,3
3	codec TLV320AIC23B	7,13
4	régulateur tension	1
5	oscillateur quartz 16MHz	1
6	connecteur usb	2
7	port jack (X2)	5,2
8	borniers	10
9	résistances/condensateurs	2
10	fraise de port	8,9
11	TVA	8,326
12	réalisation du pcb 4 couches	10
13	frais de port	15
14	total pcb	83,856

Figure 5.10: Price of the components necessary for the realization of the printed circuit

he realization of the printed circuit thus costs about 80 euros.

Then you have to take into account the price of the whole pedal box containing the printed circuit. This can be done in a fablab using a 3D printer to reduce costs as much as possible. The difficulty will be to manage to make pressure pedals robust enough for foot pressure. It will otherwise be possible to make this box with companies that manufacture custom effect pedals but the cost may be relatively high.

Conclusion

Although this project is far from what we initially imagined, our research is conclusive and in a way it is close to a product that could be delivered.

Of course, there are still tracks to be deepened and points to be improved as well on the hardware part as on the software part. Indeed, the software architecture still doesn't allow us to do real-time processing, as we initially wished. It will surely be necessary to find a library or APIs written in C or C++ language that allow the management of low-level hardware. This could also lead to new strategies concerning the learning phases, either for the convolutional neural network, or the VAE.

For the hardware part, a large battery of tests and development is still necessary. All the components are not yet integrated or linked by the software to the microprocessor. The fundamental software bases are still not implemented, the control of the audio codec is still not done, and the integration of the software bricks dedicated and optimized to audio processing have not been implemented. So, if the project is to continue, these crucial points must be resolved.

If you want to continue the project or simply try it out at home, we strongly advise you to get the resources on GitHub. Most of the code runs on Google Colab, which means that it runs on a server and a remote session. This allows you to take a look and get results no matter what configuration you have, but it ensures consistency in the results. However, if you want to work locally, it will be impossible to move forward without installing the different libraries. This can take a lot of time. It is therefore advisable to analyze the various documentations carefully.

Finally, we would like to thank our supervisor, Dr. Sylvain Reynal, who accompanied us throughout this year, and who allowed us to advance in the best, and to go to the end of our thoughts.
We would also like to thank the lecturers of the option "Music and signal processing" of the ENSEA Mr. Romain Hennequin and Thomas Hézard. Thanks to their advice and their teachings, we were able to progress in our project, and produce the results that we present to you.
We also thank the professors of ENSEA, Mr. Christophe Barès and Mr. Nicolas Papazoglou, who provided us with hardware and software resources, but also their knowledge on entrepreneurship and project management. Finally, we would like to warmly thank all our fellow students of ENSEA, whatever their promotion. The contacts and exchanges during the different breaks, as well as the different opinions helped us to create our own idea of our project.

Bibliography

[1] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. eprint: https://direct.mit.edu/neco/article-pdf/1/4/541/811941/neco.1989.1.4.541.pdf. URL: https://doi.org/10.1162/neco.1989.1.4.541.

[2] Rafael Paiva. “Circuit modeling studies related to guitars and audio processing”. PhD thesis. Nov. 2013.

[3] Stefano D’Angelo. “Virtual Analog Modeling of Nonlinear Musical Circuits”. PhD thesis. Nov. 2014.

[4] Sandro Skansi. *Introduction to Deep Learning*. Springer International Publishing, 2018.

[5] Marco A. Martínez Ramírez and Joshua D. Reiss. “Modeling Nonlinear Audio Effects with End-to-end Deep Neural Networks”. In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 171–175. DOI: 10.1109/ICASSP.2019.8683529.

[6] Jesse Engel et al. “DDSP: Differentiable Digital Signal Processing”. In: 2020. URL: https://openreview.net/forum?id=B1x1ma4tDr.

[7] Alec Wright et al. “Real-Time Guitar Amplifier Emulation with Deep Learning”. In: *Applied Sciences* 10.3 (2020). ISSN: 2076-3417. DOI: 10.3390/app10030766. URL: https://www.mdpi.com/2076-3417/10/3/766.

[8] URL: https://www.ti.com/lit/ds/symlink/tlv320aic23b.pdf?ts=1616463392868.

[9] URL: https://www.st.com/resource/en/user_manual/dm00244518-stm32-nucleo144-boards-mb1137-stmicroelectronics.pdf.

[10] Alexandre Boyer. *Règles de conception faible émission rayonnée pour les circuits imprimés*. URL: http://www.alexandre-boyer.fr/alex/enseignement/Boyer_regles_CEM_PCB_v3.pdf.

[11] Alexis Cook. *Kaggle learn*. URL: https://www.kaggle.com/learn.