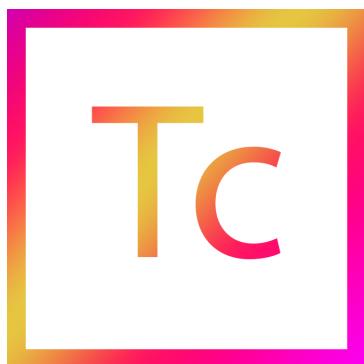


# Rapport final

## ToneCrafter



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

Encadré par Sylvain "Syd" Reynal

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Concept</b>	<b>4</b>
<b>3</b>	<b>Recherche d'idée</b>	<b>5</b>
3.1	Design Thinking . . . . .	5
3.2	Mise en place des paradoxes . . . . .	6
3.3	Réflexion sur l'utilisation du produit . . . . .	7
3.4	État de l'Art . . . . .	7
3.4.1	NSynth : Neural Audio Synthesis . . . . .	7
3.4.2	ToneTransfer   DDSP : Differentiable Digital Signal Processing . . . . .	8
3.4.3	Pédales d'effets . . . . .	9
<b>4</b>	<b>Réalisation logicielle</b>	<b>10</b>
4.1	Objectif . . . . .	10
4.2	Découverte de la bibliothèque DDSP . . . . .	10
4.3	Mise en place de nos solutions . . . . .	12
4.3.1	Variational Auto-Encoder . . . . .	12
4.3.1.1	Analyse de l'espace latent . . . . .	13
4.3.1.2	Transformer une image . . . . .	16
4.3.2	Réseau de neurones convolutifs . . . . .	19
4.3.2.1	Dataset - Une étape cruciale à ne pas négliger . . . . .	19
4.3.2.2	Paramétrage du modèle . . . . .	20
4.3.2.3	1ère approche - MFCC . . . . .	21
4.3.2.4	2ème approche - STFT . . . . .	22
4.3.3	Recherche des paramètres et effets audios . . . . .	26
4.3.3.1	Implémentation des effets . . . . .	26
4.3.3.2	Recherche des paramètres des effets . . . . .	28
4.3.3.3	Implémentation en temps réel . . . . .	31
<b>5</b>	<b>Réalisation matérielle</b>	<b>32</b>
5.1	Cahier des charges . . . . .	32
5.2	Design de la pédale et fonctionnalité . . . . .	32
5.3	Choix des composants . . . . .	33
5.4	Réalisation du PCB . . . . .	34
5.4.1	Réalisation du schematic . . . . .	34
5.4.2	Routage et compatibilité électromagnétique . . . . .	36
5.4.3	Les méthodes de communication . . . . .	38
5.4.4	Consommation énergétique et coût . . . . .	39
	<b>Conclusion</b>	<b>41</b>

# Chapitre 1

## Introduction

POL BODET, PIERRE CHOUTEAU, MAMADOU DIA

Il vous est déjà arrivé de vouloir commencer un instrument et que, au-delà de la difficulté de prise en main, vous sonniez très mal ? Vous ne savez pas comment régler celui-ci car il y a trop de boutons ? Et, même après plusieurs mois, lorsque vous voulez sonner comme votre artiste préféré vous n'y arrivez pas ? Vous ne comprenez toujours pas comment régler votre instrument, quel potard bouger pour quel effet ou même sur quel bouton de votre synthé appuyer ?

Tout cela met à mal votre motivation et les efforts des nouveaux musiciens. Et en allant plus loin, cela fait perdre pas mal de temps au musicien de régler toutes leurs pédales et amplis comme il faut. Avec l'équipe du ToneCrafter, nous sommes convaincus que si vous sonnez bien, vous serez passionné et passerez beaucoup plus de temps sur vos instruments, c'est pourquoi je vous propose une petite merveille qui porte le nom de ToneCrafter.

# Chapitre 2

## Concept

POL BODET, PIERRE CHOUTEAU, MAMADOU DIA

Le Tone Crafter c'est avant tout un logiciel trouvable sur ordinateur ou sur smartphone, facilement téléchargeable et simple d'utilisation. Grâce à lui, vous n'aurez plus qu'à brancher votre instrument à votre ordinateur ou votre téléphone, vous cherchez l'artiste ou la musique que vous voulez jouer, et c'est FINI. Vous n'avez plus rien à faire. Le ToneCrafter s'occupera de régler lui-même tous les potards d'effets afin que vous puissiez obtenir le même son que vous espériez.

Il ne vous reste plus qu'à jouer.

Ce qui est vraiment cool avec le ToneCrafter c'est qu'il permet de :

- Rendre le son des plus grands accessible à tous
- À disposition des débutants comme des professionnels
- Simple d'utilisation
- Possibilité de créer une communauté d'utilisateurs

# Chapitre 3

## Recherche d'idée

POL BODET, PIERRE CHOUTEAU, MAMADOU DIA

### 3.1 Design Thinking

Dans le but de trouver cette idée incroyable et surtout unique en son genre, nous avons commencé par une démarche de design thinking.

Celle-ci est beaucoup utilisée par les designers et autres créateurs, pour concevoir un objet à partir des besoins définis par une étude préalable. Elle permet également à chacun de s'exprimer librement et sans jugement au travers de mots que l'on vient écrire sur des post-its, au sujet d'un slogan, d'une phrase, ou d'un fait de société.

Pour notre étude, nous avons travaillé sur le slogan : “**le Vintage c'est Grand Âge**”

Dans la démarche de design thinking, on a donc analysé ce slogan et noté tous les mots qui nous étaient évoqués par le slogan, qu'ils soient plus ou moins proches du thème. On a ensuite essayé de les catégoriser de manière thématique afin de voir plus les interactions qu'ils peuvent y avoir entre chacun d'entre eux.



FIGURE 3.1 – Concepts évoqués par le slogan

## 3.2 Mise en place des paradoxes

C'est à partir de tous ces mots, mais surtout des thèmes dégagés que nous avons pu mettre les paradoxes qui ont donné la naissance de notre projet :

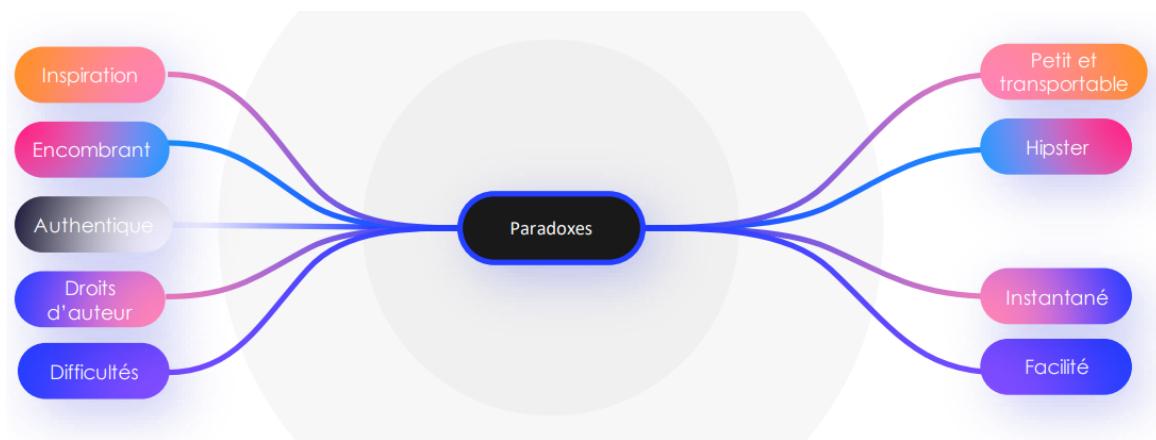


FIGURE 3.2 – Paradoxes soulevés par le slogan

La mise en opposition des différents paradoxes soulevés nous a permis de définir une problématique du type :

**“Comment un objet peut-il être authentique tout en étant dans l’air du temps tout en poussant la créativité de l’utilisateur sans enfreindre les droits d’auteur ?”**

Le ToneCrafter est selon nous la réponse à cette problématique de par la nouveauté des plateformes sur lesquelles il fonctionnera, tout en conservant ce côté authentique et rugueux qu'est le sacro-saint "Tone" des guitaristes, il apporte aux masses un son jusqu'ici inaccessible qui permet à de nouvelles créations d'émerger.

Après avoir posé notre problématique, nous avons dû définir en détail ce qu'était le ToneCrafter. Pour ce faire, nous avons écrit un guide d'utilisation ainsi qu'une bande dessinée expliquant le cœur du projet de manière imagée. Celle-ci expliquait comment répliquer le "Tone" d'un artiste instrument, en seulement quelques clics. La procédure est très simple : on choisit un artiste dans la base de données ou on upload un morceau dont on veut extraire les paramètres, l'application génère alors un filtre permettant de reproduire les effets attendus.



FIGURE 3.3 – Bande dessinée ToneCrafter

Maintenant que nous avons imaginé comment nous voulons que le produit soit utilisé, il faut déterminer comment le produit doit être installé et comment le configurer. Ainsi, nous avons rédigé un

guide de l'utilisateur expliquant comment la première configuration doit être opérée. Le processus doit être simple et rapide. C'est pourquoi nous avons imaginé quelque chose de simple. Une alimentation électrique et une connexion à un terminal où le logiciel ToneCrafter est installé doivent permettre à la configuration du système. On peut d'ailleurs y voir la forme que nous avions imaginée pour notre système. Une sorte d'amplificateur. Cette forme assez simpliste réduit le nombre de connexions et permet à terme de n'avoir qu'un seul système.

### 3.3 Réflexion sur l'utilisation du produit

Afin de pouvoir se projeter sur comment l'utilisateur peut s'approprier le produit et comment créer la meilleure expérience utilisateur, Nous avons donc réalisé une bande dessinée et un guide d'utilisateur. Ces 2 documents nous ont servi de référence sur ce que nous voulions donner comme livrable à la fin de ce projet.

### 3.4 État de l'Art

Afin de pouvoir avoir une vision plus globale des différentes technologies utilisées dans ce domaine ou même de projets existants qui s'approchent de ce nous voulons faire, nous avons commencé par effectuer un état de l'art. En faisant ces recherches nous sommes tombés sur un projet qui a particulièrement retenu notre attention :

#### *Le Projet Magenta*

Mais qu'est-ce que le projet Magenta ?

Pour faire simple, Magenta, c'est un projet de recherche open source qui a été lancée par des chercheurs et des ingénieurs de l'équipe Google Brain. Leurs objectifs sont de développer des algorithmes d'apprentissages profonds (DeepLearning) et d'apprentissage par renforcement (Reinforcement Learning) pour faire un grand nombre de choses, allant de la génération musicale à la transformation de signaux (timbre transfert, Music transformer...).

Deux de ces travaux de recherche nous ont particulièrement intéressés car ils nous ont permis d'observer et de tester différents algorithmes qui pouvaient nous aider pour ce projet :

#### 3.4.1 NSynth : Neural Audio Synthesis

NSynth, comme son nom l'indique est un synthétiseur neuronal : à la différence d'un synthétiseur classique qui génère des sons à partir d'un VCO, NSynth utilise un réseau de neurones entraîné avec un dataset créé pour l'occasion. Vous vous demandez peut-être pourquoi nous nous sommes intéressés à un projet si loin de celui que nous préparions ?

La réponse : le NSynth Super

Il s'agit d'un contrôleur hardware créé spécifiquement pour utiliser l'algorithme du projet logiciel. Nous avons été intrigués par cet objet dont le facteur de forme n'est pas sans rappeler celui d'une pédales d'effet pour guitariste et dont l'implémentation est tout simplement exquise (visible figure 3.4).

Tout comme la facette logiciel du projet, le NSynth est un projet Open-Source que chaque utilisateur peut modifier selon ses besoins. On remarque d'ailleurs que les contrôles ne sont pas réalisés par un écran tactile dans la version proposée au grand public, sûrement dans un souci de simplification



FIGURE 3.4 – Prototype du NSynth Super

de la fabrication du contrôleur afin de le rendre accessible à un public plus grand nombre. Nous conserverons cet objectif d'accessibilité du projet pendant toute la phase de développement.

### 3.4.2 ToneTransfer | DDSP : Differentiable Digital Signal Processing

Qui n'a pas déjà rêvé de pouvoir jouer de tous les instruments, sans même savoir comment s'en servir ? Ou de transformer des sons du quotidien en sons d'instruments ? Aujourd'hui, avec ce projet de recherche, c'est chose faite. Si vous voulez essayer, c'est par *ici*.

C'est en grande partie pour cela que ce projet a retenu toute notre attention, surtout pour la partie logicielle. Il s'apparente beaucoup à ce que nous voulons faire, c'est-à-dire reproduire le son d'un certain type d'effet à partir d'un signal de base. La seule différence, c'est qu'ici, à la place de s'approcher d'un effet, comme une distorsion, une reverb ou même un chorus, leur algorithme de machine learning permet de modifier le signal d'origine afin d'obtenir la sonorité de l'instrument que l'on souhaite. Par exemple, il est possible d'obtenir un son de violon à partir d'une voix :

Play : Chant

Play : Violon

Le principe est donc exactement le même, l'algorithme modifie le signal d'entrée afin d'obtenir un son différent, comme on peut le voir avec les deux spectrogrammes de la figure ci-dessous :

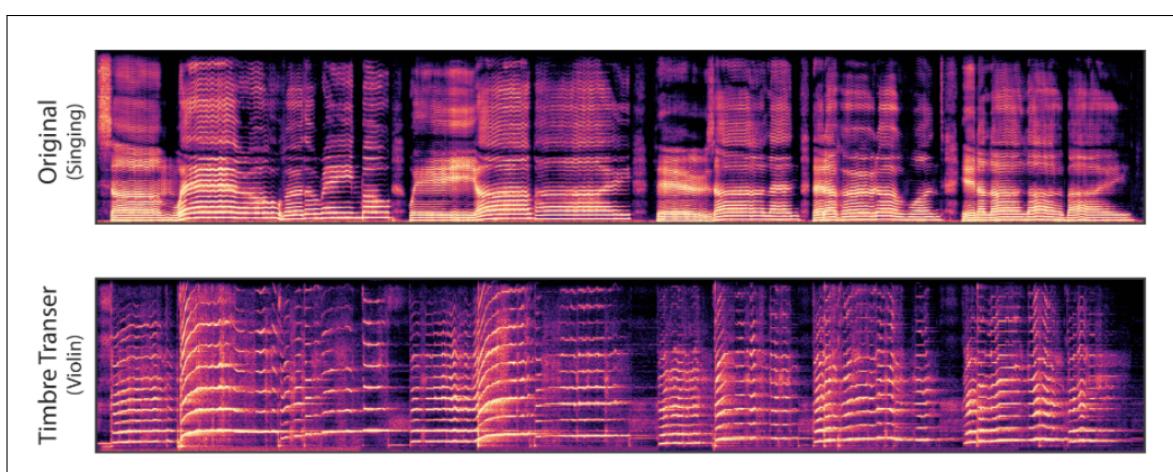


FIGURE 3.5 – Timbre Transfert : Violon -> Flûte

Le cas idéal pour nous serait donc de pouvoir utiliser cet algorithme pour l'utiliser en temps réel, mais surtout de pouvoir l'entraîner sur des effets audio et non des instruments.

Ce qui est également très intéressant avec ce projet c'est qu'en plus de pouvoir charger des modèles pré-entraînés afin de pouvoir s'approcher de plusieurs instruments, il est également possible d'entraîner un modèle avec ses propres sons.

### 3.4.3 Pédales d'effets

Comme on a pu le remarquer, ces deux précédents projets (le NSynth et le ToneTransfer) sont très intéressants et ils ont énormément de point en commun avec ce que nous voulons faire. Cependant, aujourd'hui, l'objet existant et commercialisé qui ressemble le plus à ce que l'on souhaite faire reste la classique pédale d'effet.

Il en existe un très grand nombre, pour chaque effet (Distortion, Overdrive, Octaver, Delay, Reverb...) et pour tout le monde, et grâce à celles-ci, on peut faire exactement ce que l'on souhaite, c'est-à-dire : s'approcher de la sonorité de nos artistes préférés.

Cependant, le problème qui subsiste avec ces classiques pédales d'effets c'est qu'il faut s'en procurer énormément si l'on veut obtenir exactement le même son qu'un artiste en particulier... Tout simplement car toute pédale quelle soit, a sa propre sonorité et si on change de marque ou de modèle... Alors l'effet est différent.

Après beaucoup de discussion avec l'équipe, on est tombé d'accord sur le fait que le design de ces pédales pourrait complètement convenir avec notre projet. On aimerait garder la robustesse, la solidité ainsi que la taille de celles-ci, afin que notre objet final soit transportable facilement et utilisable partout, que ce soit en concert ou chez soi, sans avoir peur de le casser.



FIGURE 3.6 – Pedalboard



FIGURE 3.7 – Multiple-Effects Pedal

# Chapitre 4

## Réalisation logicielle

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

### 4.1 Objectif

L'objectif principal de cette partie est de créer, et entraîner une IA capable d'appliquer des effets et filtres sur un signal audio en temps réel.

### 4.2 Découverte de la bibliothèque DDSP

Après avoir découvert le travail de recherche du projet Magenta sur le Timbre Transfert, et dans le but de tester ce qu'il était possible de faire en traitement audio, nous avons commencé par tester le potentiel de la bibliothèque au cœur de ce projet : DDSP (Differentiable Digital Signal Processing). Cette bibliothèque permet non seulement de générer des signaux audio à l'aide de réseaux de neurones déjà entraînés, mais elle permet également d'y appliquer des effets tels que de la réverbération, un vibrato, ou un flanger.

Le réel avantage de DDSP c'est qu'elle permet de pouvoir mêler les algorithmes et techniques de traitement du signal avec les réseaux de neurones. Ainsi, on peut utiliser des filtres linéaires, que l'on programme assez facilement sur des DSP, avec des réseaux de neurones. Cela rend la prise en main plus facile, et augmente le nombre de possibilités. De plus, contrairement aux autres outils de modélisation par intelligence artificielle, il est possible de travailler avec des auto-encodeurs, ce qui aurait pu être un vrai plus dans la suite du projet.

Nous avons donc commencé par essayer les différents effets disponibles dans DDSP, en enregistrant des sons de guitares et en les appliquant dessus :

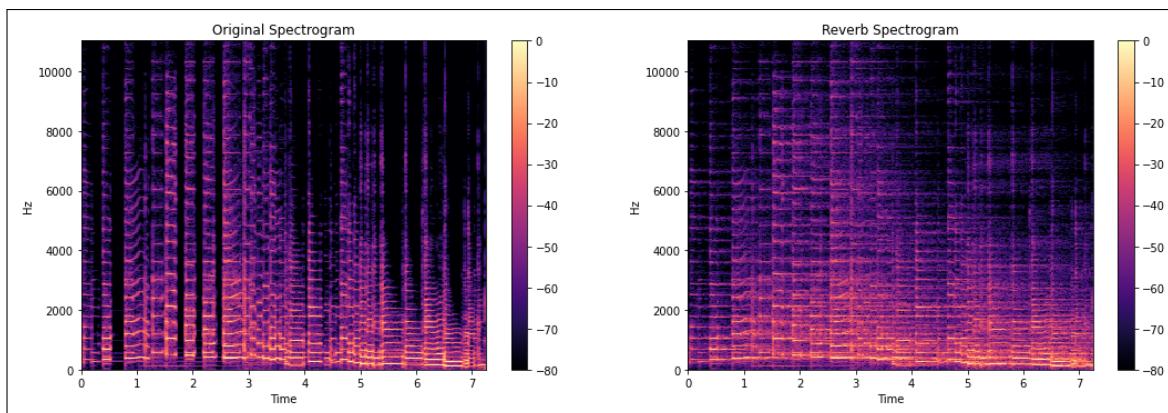


FIGURE 4.1 – Test de la reverb de la bibliothèque DDSP

Comme on peut le voir sur ces différents spectrogrammes et en écoutant les audios obtenus, les résultats étaient très satisfaisants pour un début et les effets étaient bien audibles.

Cependant, on travaillait ici sur des enregistrements de guitares, or la finalité du projet est bien de pouvoir modifier le signal provenant de l'instrument en temps réel.

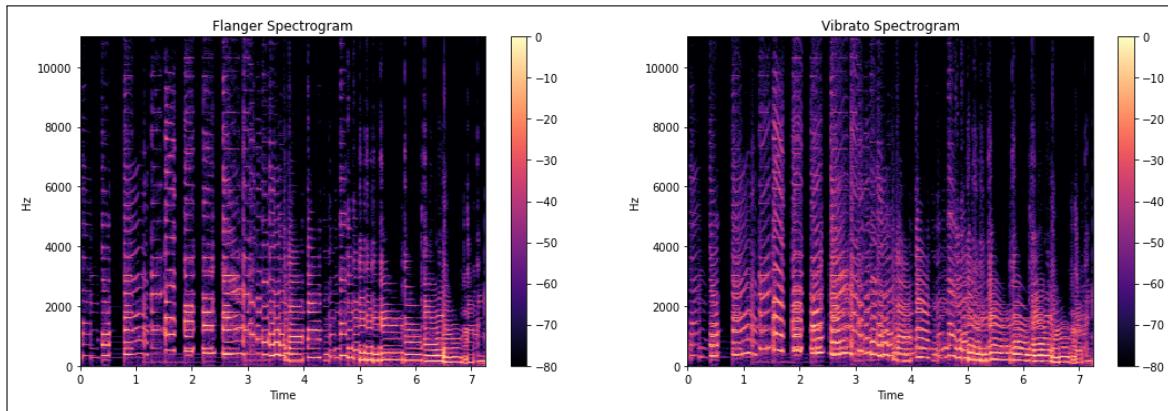


FIGURE 4.2 – Test du flanger et du vibrato de la bibliothèque DDSP

[Play : Original](#)

[Play : Reverb](#)

[Play : Flanger](#)

[Play : Vibrato](#)

La suite est donc évidente, après avoir réussi à créer et modifier des signaux en leur appliquant des effets, nous avons commencé à nous intéresser à la question du temps réel.

Pour cela, nous avons créé un script Python permettant d’acquérir un son provenant du microphone de notre ordinateur afin d’y appliquer un effet et de le restituer aux haut-parleurs.

Cela fonctionnait parfaitement, jusqu’à ce que l’on commence à vouloir y appliquer un effet de la bibliothèque DDSP... C’est à ce moment que l’on a pris conscience des limites de cette bibliothèque. Avec celle-ci, nous ne pouvons pas modifier le son provenant du microphone en temps réel. En effet, la gestion des échantillons et de la mémoire ne permet pas d’appliquer des effets et gérer un buffer en parallèle. Il faudrait donc pour cela réécrire une partie de la bibliothèque dans un autre langage pour gérer les couches bas niveaux, ce qui deviendrait alors vite pénible.

Dans le but de vérifier que notre script en temps réel fonctionnait bien, nous avons essayé d’appliquer une distorsion, créée par nos soins, sur le signal arrivant du microphone. La distorsion utilisée est visible ci-dessous :

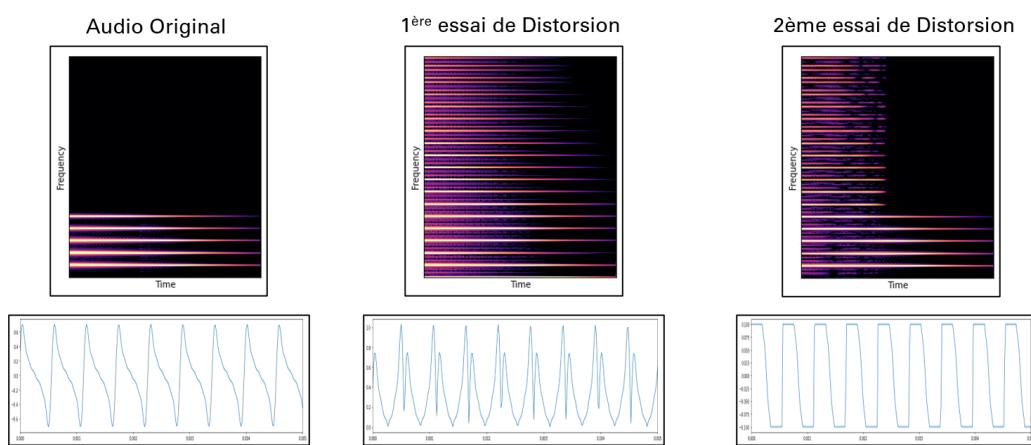


FIGURE 4.3 – Essai de distorsion en temps réel

Après quelques tests en temps réel, nous avons pu en conclure que celui-ci fonctionnait bien, et donc que c'était bien la définition de la bibliothèque DDSP qui posait problème. Cela est en partie due aux types des variables utilisées par DDSP (Torseurs) qui ne sont pas compatibles avec du traitement de signal en temps réel.

Suite à ce premier problème, nous avons donc décidé de changer un peu notre approche pour réussir à atteindre notre objectif.

L'étude se porte alors vers une application en temps réel, ce qui signifie qu'au moment même où le signal est acquis, il doit être traité et restitué. Étant donné que la librairie DDSP ne nous permet de travailler que sur des signaux dont on connaît tous les échantillons, nous avons décidé de suivre trois méthodes différentes que nous étudierons par la suite.

## 4.3 Mise en place de nos solutions

### 4.3.1 Variational Auto-Encoder

Initialement, notre souhait était d'avoir un réseau de neurone capable d'identifier les caractéristiques d'un effet dans un son afin de pouvoir appliquer ce même effet sur un autre son. Nous nous sommes inspirés des algorithmes de "Transfert de Style" combinés aux Auto-Encoders. En effet, les auto-encoder sont capables de prendre un signal en entré, et d'y appliquer des filtres ou des transformés. Ceci correspond totalement avec les exigences de notre projet. Nous voulions donc combiner les outils d'intelligence artificielle avec les auto-encoder, d'où note volonté d'utiliser un Variational Auto-Encoder (VAE).

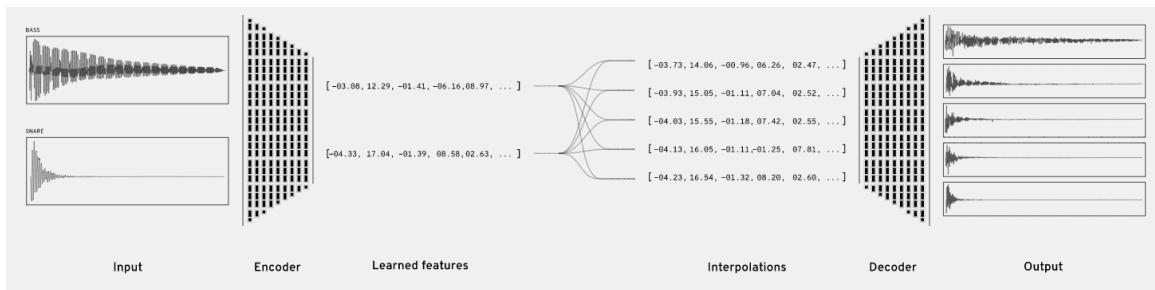


FIGURE 4.4 – Structure du VAE utilisé par Magenta

Les Auto-Encoder sont des paires de réseaux neuronaux combinés. Leurs objectifs est de trouver un moyen d'encoder les données sous une forme compressée, que l'on appelle l'espace latent, mais contrairement aux Auto-Encoder, les Variational Auto Encoder (VAE), eux ne se contentent pas de simplement être capable de retrouver l'entrée d'origine à partir de l'espace latent (compression). En effet les VAE supposent que les données sont produites par un modèle graphique orienté. Le VAE va donc apprendre une estimation de ce modèle graphique. Ainsi le VAE peut être utilisé pour générer des données cohérentes avec les données en entrée. En effet il suffit de mettre des nombres aléatoires en entrée du décodeur pour créer une donnée cohérente. Mais afin d'être capable d'utiliser un VAE pour appliquer des effets à un son, nous allons étudier l'espace latent d'un VAE.

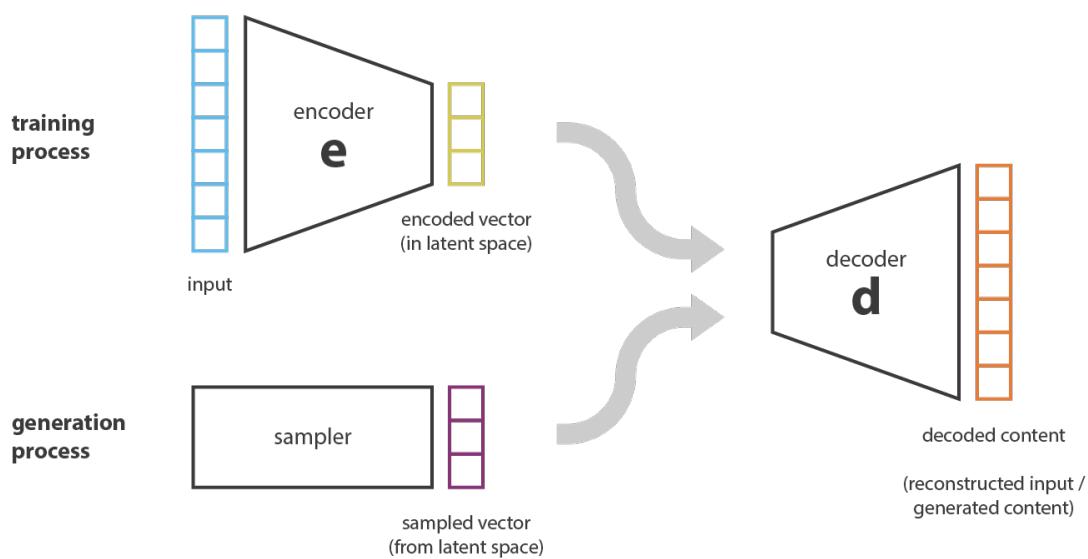


FIGURE 4.5 – Le fonctionnement d'un VAE

#### 4.3.1.1 Analyse de l'espace latent

Malheureusement pour nous, il n'existe que très peu de projet et de recherches sur la musique à l'aide d'un VAE. Fort heureusement, il est possible de travailler sur des signaux audio à l'aide des spectrogrammes issus des STFT (transformée de Fourier à temps court), qui se trouvent être des signaux 2D. En plus travailler en fréquentiel aura pour avantage d'éviter que le VAE apprenne des sinusoïdes, qu'il retrouverai constamment si on utilisait la musique en temporel. Cependant aucune base de donnée de musique en représentation fréquentielle n'existe. Ainsi nous avons fait le choix de commencer par étudier un VAE utilisant des images, en supposant que ce qui marcherai sur une image marcherai sur un son.

Donc nous allons utiliser la base de donnée MNIST. La base de données MNIST pour Modified ou Mixed National Institute of Standards and Technology, est une base de données de chiffres écrits à la main. Elle contient dans notre cas, un set d'entraînement de 60 000 images de chiffres écrits à la main et d'un set de test de 10 000 images de chiffres. Chaque image est de taille 28p\*28p et chaque image est associée à la valeur du chiffre écrit.

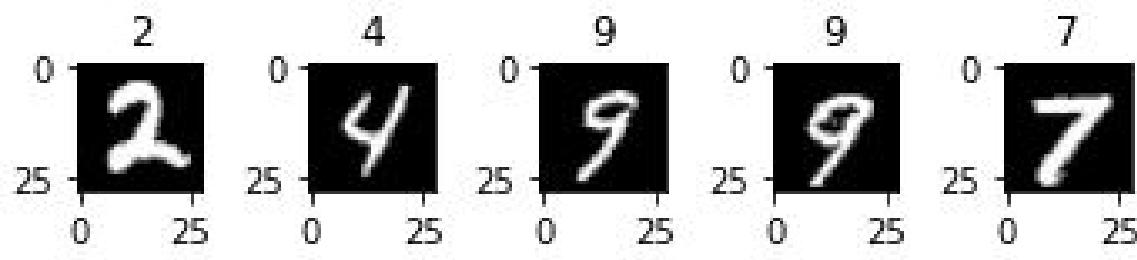


FIGURE 4.6 – 5 images et leur valeurs tirées aléatoirement du dataset d'entraînement

Pour analyser l'espace latent, nous allons commencer par analyser les images qu'associe le décodeur de notre VAE à chaque point de l'espace latent. Ainsi on va utiliser par la suite un espace latent en 2 dimensions pour le représenter facilement sur des graphes. Puis pour chaque couple (x,y) de l'ensemble  $[-3,3]*[-3,3]$  on va générer l'image associée. On obtient alors une carte des données décodées par le décodeur.

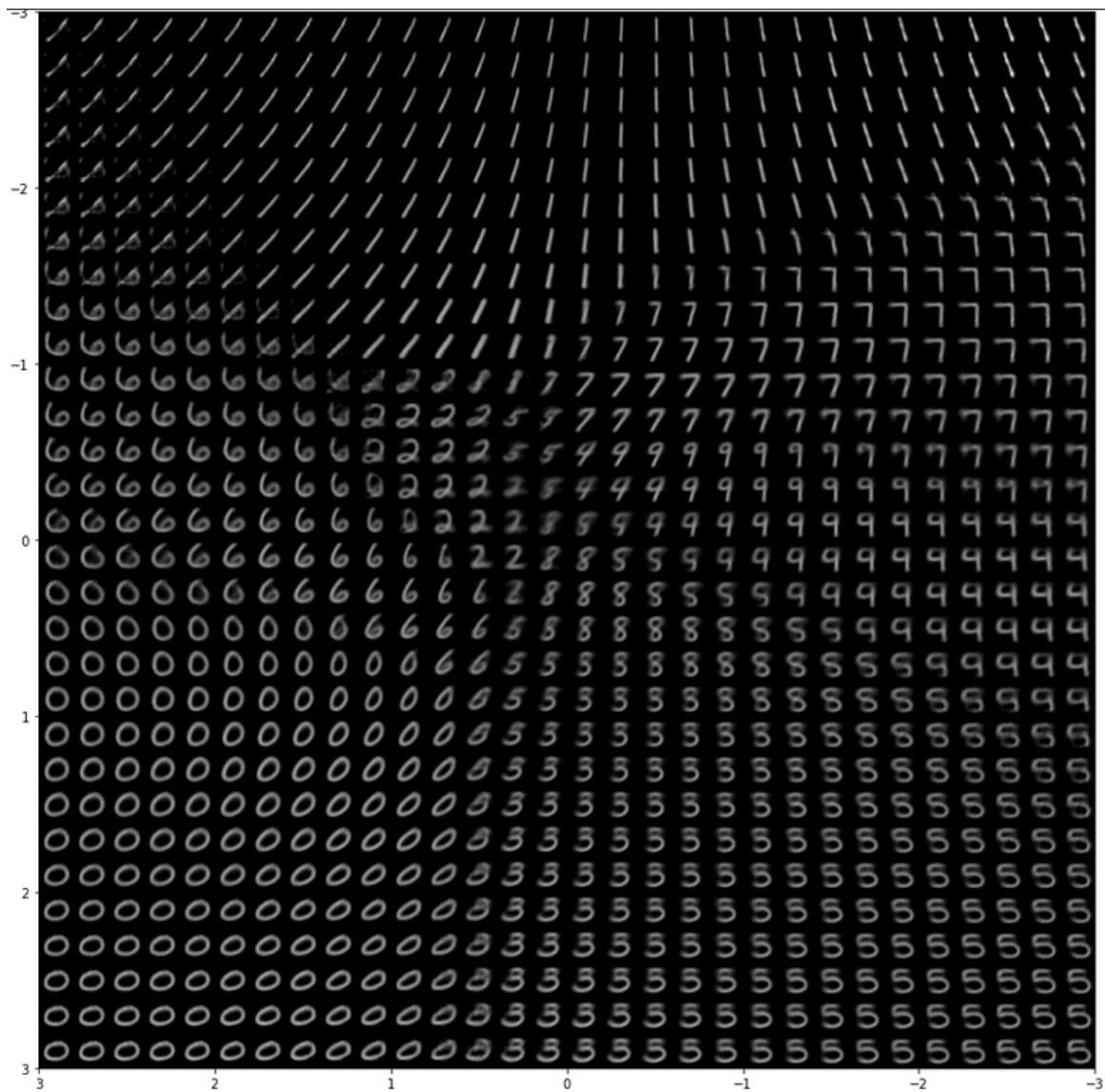


FIGURE 4.7 – carte des données décodées par le décodeur

Afin de faire une analyse plus précise nous allons maintenant faire l'inverse. On va cette fois prendre toutes les images du set de test et afficher sur un graphique le vecteur 2D de l'espace latent auquel est associé cette image.

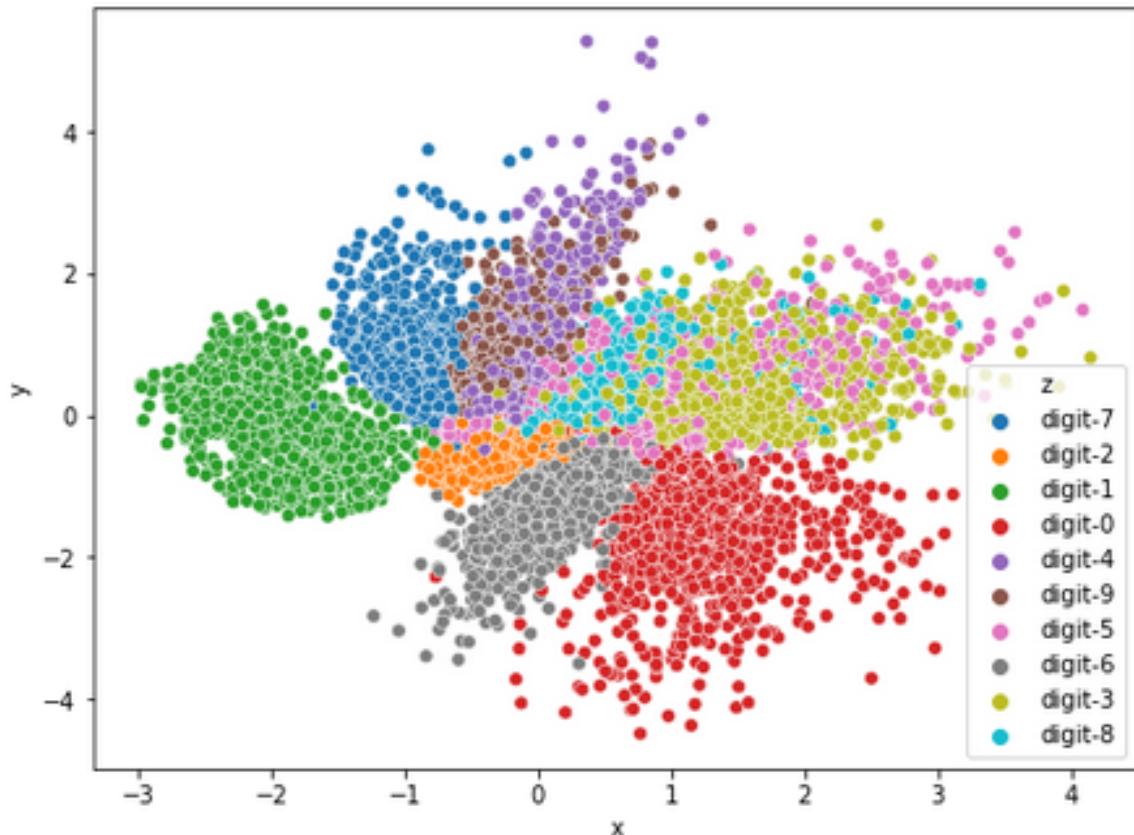


FIGURE 4.8 – carte des données encodées par l'encodeur

Ces deux cartes nous permettent de remarquer que pour que le VAE marche avec très peu d'erreurs, il faut qu'il ait un espace latent avec un nombre de dimension suffisamment grand pour que toutes les valeurs aient des espaces distincts. Sinon on va avoir des erreurs de décodage pour les nombres qui n'ont pas un espace distinct. Dans notre cas on remarque bien que les 4, les 9, les 7 et les 5 n'ont pas d'espace distinct.

En effet comme on peut le voir sur la figure ci dessous le VAE produit souvent des erreurs lors des décodages de ces chiffres.

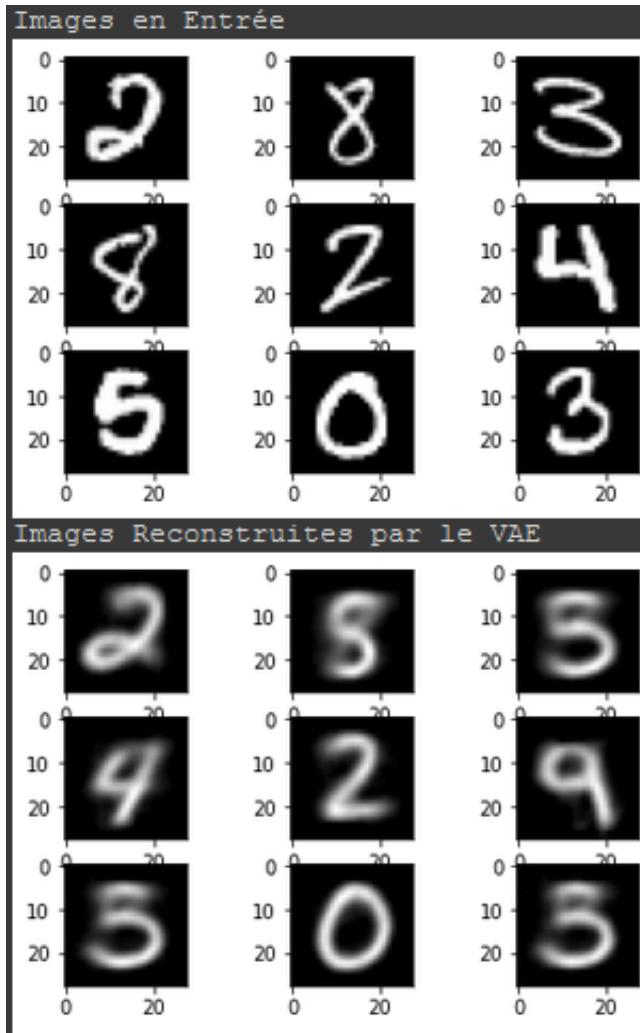


FIGURE 4.9 – Chiffres encodés puis décodés par le VAE

#### 4.3.1.2 Transformer une image

Dans le cadre de notre projet, le VAE serait entraînée par une base de donnée contenant différents sons avec et sans effets et à chacun de ces sons on associerait le nom des effets appliqués. Ainsi si le VAE est bien adapté à notre base de donnée chaque effet aurait un espace distinct dans l'espace latent. Ainsi si on revient à la base de donnée MNIST, pour nous appliquer un effet reviendrait à transformer un chiffre en un autre chiffre. Ainsi puisque normalement les chiffres sont tous regroupés dans des espaces distincts de l'espace latent. Si on calcule la valeur moyenne de chaque espace, on pourrait convertir un chiffre en un autre en lui faisant suivre la translation définie par le vecteur qui va de la valeur moyenne de son espace jusqu'à la valeur moyenne de l'espace du chiffre en lequel on souhaite le transformer. L'avantage c'est que le chiffre gardera ainsi ses caractéristiques propres, si il est sur la gauche de l'ensemble des 1 il resterait sur la gauche de l'ensemble de 0 après sa translation.

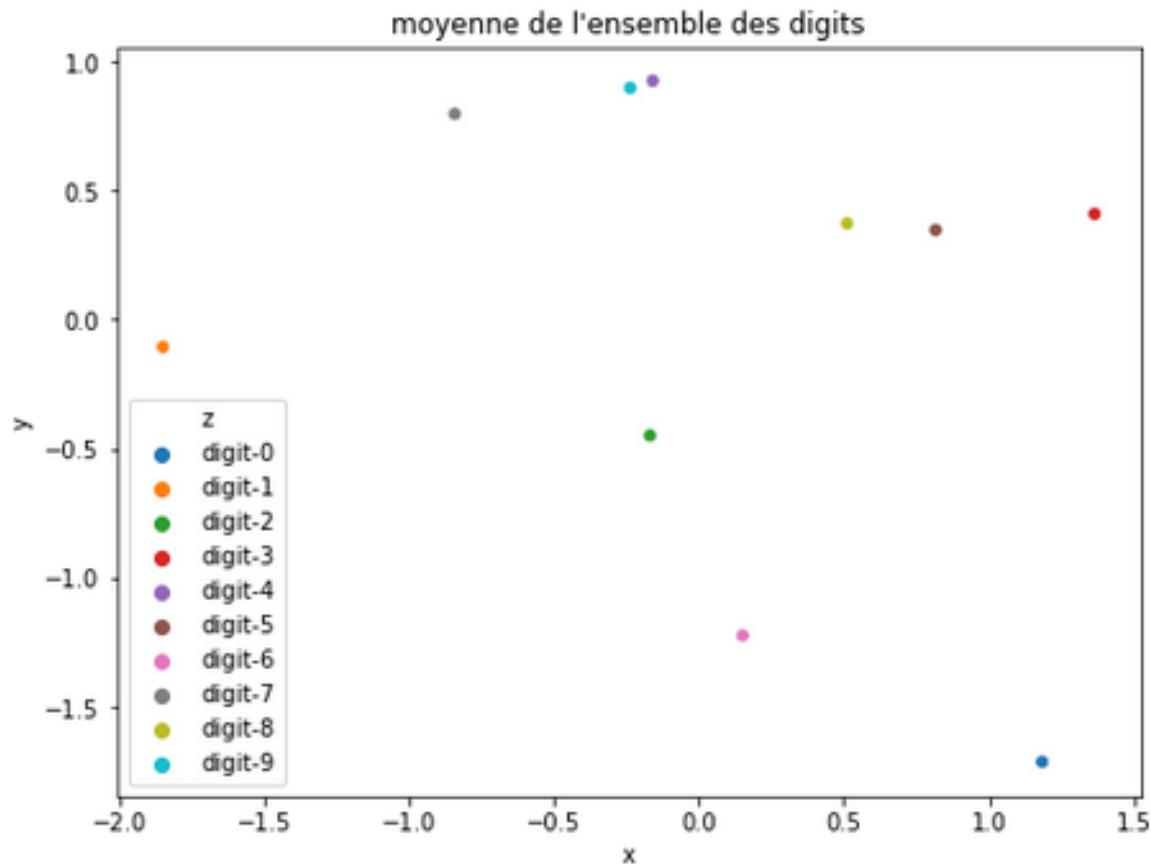


FIGURE 4.10 – Les valeurs moyennes de chaque espace

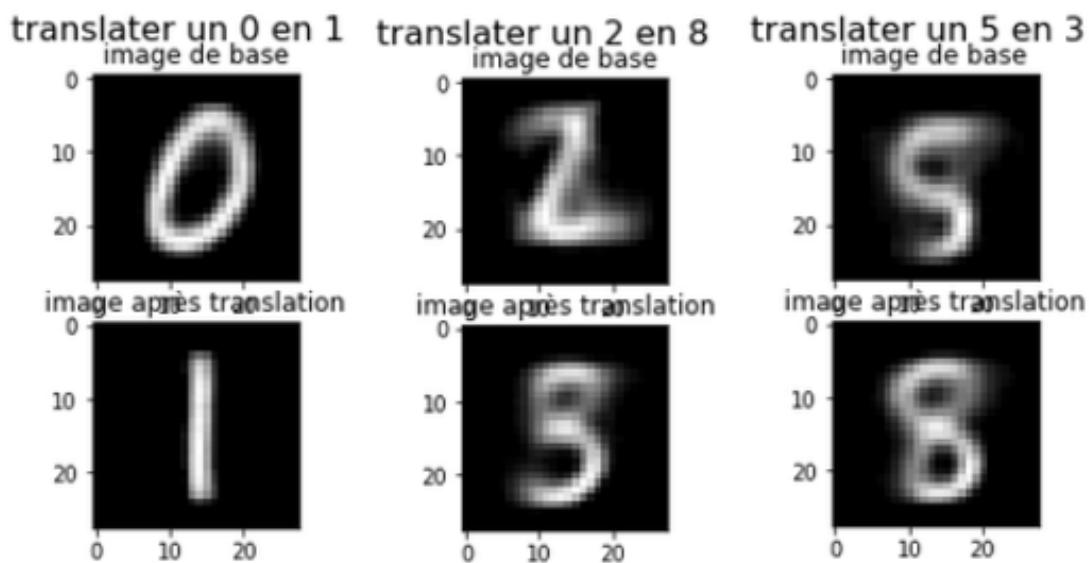


FIGURE 4.11 – Translations pour un espace latent de 2 dimensions

Les translations dans un espace en 2 dimensions ne sont pas satisfaisantes pour les chiffres qui n'ont pas un espace distinct. Ainsi il est primordial de choisir un nombre de dimensions suffisamment

grand. En effet comme on peut le voir sur la figure ci dessous, cette méthode marche très bien dans un espace latent de 32 dimensions.

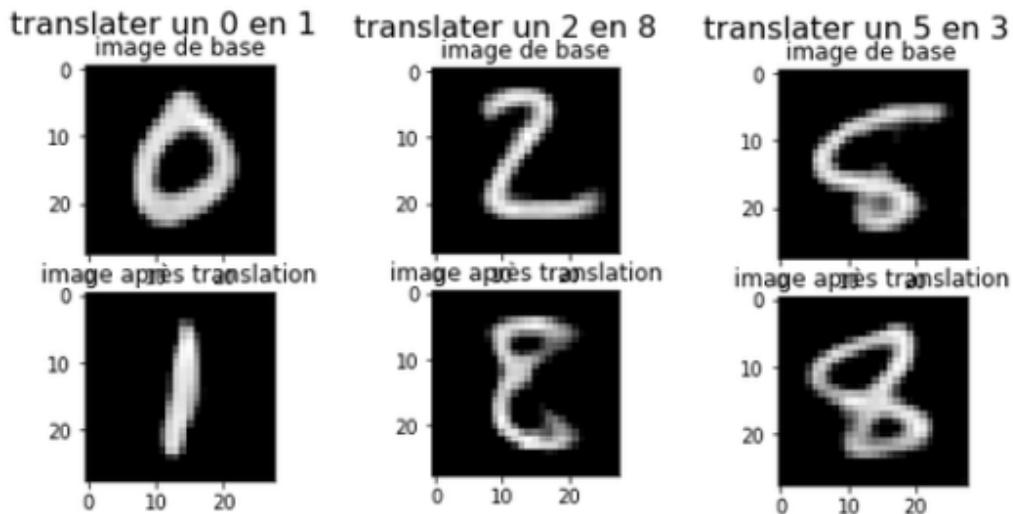


FIGURE 4.12 – Translations pour un espace latent de 32 dimensions

De plus cette méthode nous permet de facilement implémenter des effets réglables, en effet il suffit de multiplier le vecteur de transition par un coefficient A pour atténuer l'effet. Ainsi si on teste ça pour la translation d'un 1 en 0, on obtient un 0 qui est de plus en plus fermé comme un 1 plus A est grand

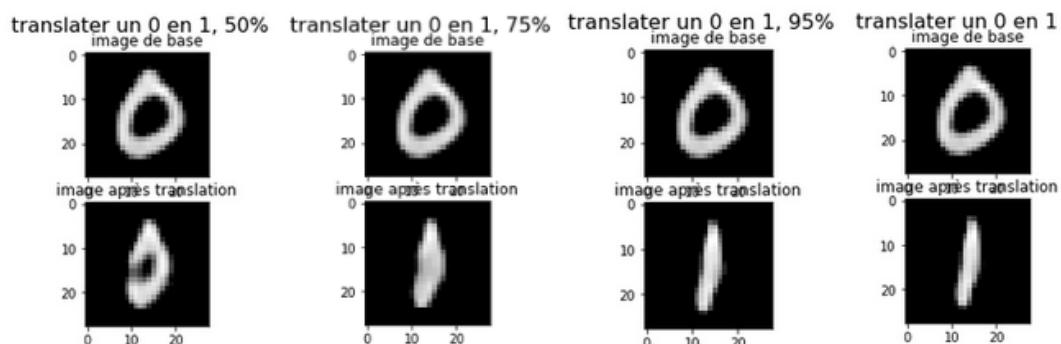


FIGURE 4.13 – Translations d'un 0 en 1 atténuées

Nous sommes donc capable d'appliquer un effet sur des images de chiffres simples. Maintenant il reste à voir si on peut faire la même chose sur des images plus complexes représentant des sons avec et sans effets.

### 4.3.2 Réseau de neurones convolutifs

En parallèle de l'étude sur le VAE, nous avons essayé une deuxième approche afin de pouvoir modéliser un effet et l'appliquer en temps réel sur un signal audio. Pour celle-ci, nous sommes partis sur une architecture plus simple que le VAE mais tout aussi efficace, et nous avons réduit le nombre d'effet à un seul effet à reproduire. Cela nous a permis d'avancer, et surtout de pouvoir faire des essais en même temps que la réalisation de la maquette.

Nous avons donc décidé de partir sur une architecture composée de neurones convolutifs. Comme il l'a été évoqué précédemment, cette architecture possède plusieurs similitudes avec les VAE, mais l'avantage c'est que celle-ci est plus répandue, ce qui facilitera les recherches et sa mise en place. Les réseaux de neurones convolutifs sont des réseaux inspirés des cortex visuels que l'on peut trouver chez l'homme ou chez les animaux. Ces neurones fonctionnent par un empilage de couches constituées de perceptrons. Cette architecture a pour but de traiter des quantités d'informations limitées à l'aide de produits de convolutions. On retrouve principalement cette architecture pour traiter des images, des vidéos ou n'importe quel autre signal à 2 dimensions. Ceci peut être un avantage pour nous car il est possible d'effectuer du traitement audio en travaillant sur la transformée de Fourier à court terme (STFT chez les anglophones) ou encore en travaillant sur la MFC (Mel-frequency cepstrum).

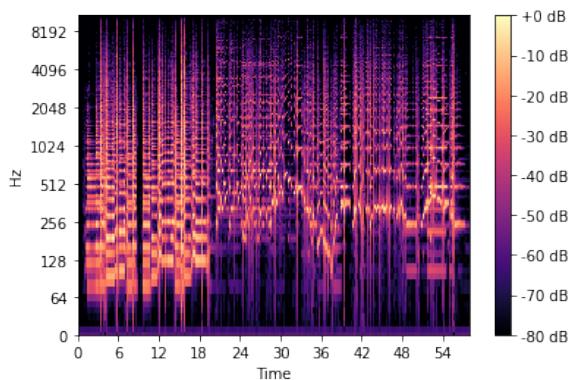


FIGURE 4.14 – STFT d'un morceau de guitare

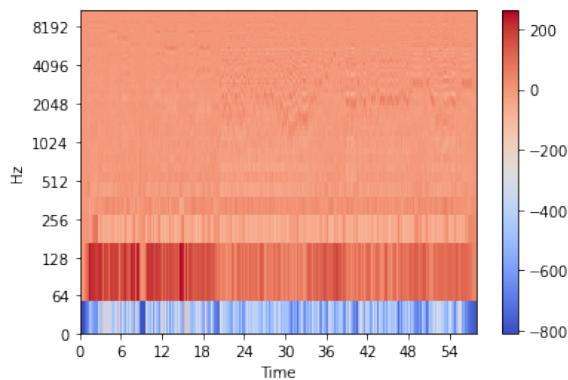


FIGURE 4.15 – MFC d'un morceau de guitare

Ces transformées de Fourier un peu particulière permettent de générer des signaux en 2 dimensions où les axes sont la fréquence pour l'ordonnée et le temps pour l'abscisse. Ainsi, on se retrouve avec un signal comparable à une image, et on peut donc utiliser les techniques d'apprentissage issues du traitement d'images.

Cependant, avant même de commencer à parler de STFT, MFC, ou de réseaux de neurones, il faut tout d'abord s'intéresser au dataset (base de données) que l'on va utiliser.

#### 4.3.2.1 Dataset - Une étape cruciale à ne pas négliger

Le choix du dataset est l'une des choses les plus importantes dans la mise en place d'un réseau de neurones. Un réseau, même parfaitement configuré serait incapable d'apprendre correctement la tâche qui lui est demandée avec des données de mauvaise qualité. Le choix du dataset utilisé est donc tout aussi important que celui de l'architecture du réseau.

Il faut savoir qu'il existe un très grand nombre de datasets disponibles sur internet, la plupart sont généralement open-sources et donc complètement libres d'accès.

Pour les applications musicales, on citera par exemple le *Dataset de NSynth* ou encore celui de *IDMT-SMT\_Audio\_Effects*, qui balayent respectivement l'ensemble des notes d'un nombre incalculable d'instruments et de nombreux effets audio appliqués à une guitare et une basse.

Malheureusement, ces datasets ne nous convenaient pas vraiment, car nous voulions connaître exactement l'effet que nous allions essayer d'approcher. Nous avons donc décidé de créer notre propre base de données.

Pour cela, nous avons enregistré une guitare pendant 1 minute et ajouté différents niveaux de distorsion. Ensuite, afin d'augmenter la taille de notre base, nous avons appliqué un filtre EQ avec différents réglages. Nous nous sommes retrouvés avec un grand nombre de fichiers .wav que vous pouvez retrouver sur notre *GitHub*.

Nous avons ensuite divisé nos fichiers en morceaux de 200 ms, et nous les avons organisés de la manière suivante :

◊ Données d'apprentissage :

- X\_train :
  - Clean
  - Clean\_TrebleBoost
  - Clean\_BassBoost
  - Clean\_BassCut
- y\_train :
  - Disto
  - Disto\_TrebleBoost
  - Disto\_BassBoost
  - Disto\_BassCut

◊ Données de validation :

- Clean\_TrebleCut comme X\_valid
- Disto\_TrebleCut comme y\_valid

Après la création de cette base de données, nous avons pu passer au paramétrage de notre réseau de neurones.

#### 4.3.2.2 Paramétrage du modèle

Pour paramétrier un réseau de neurones, en dehors de l'architecture du réseau en lui-même, il faut savoir qu'il y a deux choses importantes à définir pour indiquer au réseau le problème qu'il doit résoudre, et comment il doit le résoudre. C'est le rôle de la Loss function et de l'Optimizer.

Dans notre cas, le problème que doit résoudre le réseau de neurones est un problème de régression. Nous voulons que le réseau apprenne à reproduire un effet, plus particulièrement une distorsion. En d'autres termes, ce que l'on veut, c'est que le réseau apprenne une fonction complexe, représentant la distorsion.

Après avoir compris cela, le choix de la Loss Function est relativement simple, puisque pour des problèmes de ce type-là, il en existe principalement deux :

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

Ces fonctions servent à mesurer la disparité entre la valeur réelle de la cible et la valeur prédictée par le modèle, comme imaginé ci-dessous :

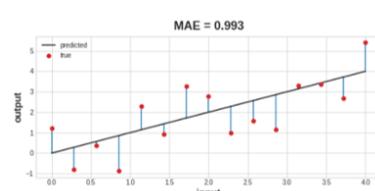


FIGURE 4.16 – Visualisation du fonctionnement d'une MAE

Ensuite, comme au cours de l'apprentissage, nous cherchons à minimiser cet écart entre la prédiction et la cible, nous avons choisi pour l'Optimizer : une descente de gradient (ADAM). Pour plus de détails sur le fonctionnement d'une descente de gradient, *ça se passe ici*.

Nous avons également pris le temps de s'intéresser aux problèmes de sur-apprentissage/sous-apprentissage, qui sont très fréquents lorsque l'on travaille sur du machine learning. Ces problèmes sont principalement dus à l'entraînement du réseau de neurones. Celui-ci apprend trop bien et se spécialise sur les données du dataset, ce qui l'empêchera de faire de bonnes prédictions sur des données qui diffèrent de celui-ci.

Pour limiter cela au maximum, nous avons donc pris soin de mettre en place un early-stopping. Celui-ci permet d'anticiper le phénomène de sur-apprentissage en arrêtant l'entraînement du réseau de neurones dès que l'erreur moyenne (MAE) de la validation atteint son minimum.

Cela peut être imaginé de la manière suivante :

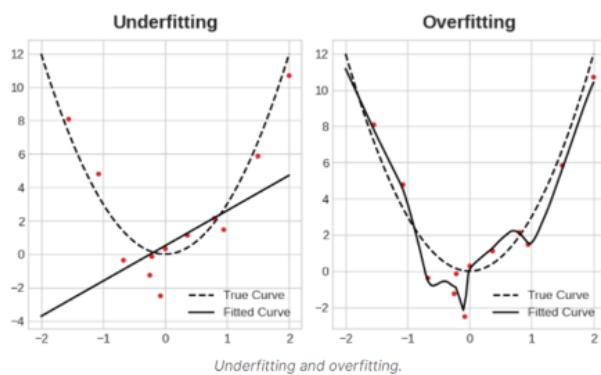


FIGURE 4.17 – Underfitting/Overfitting

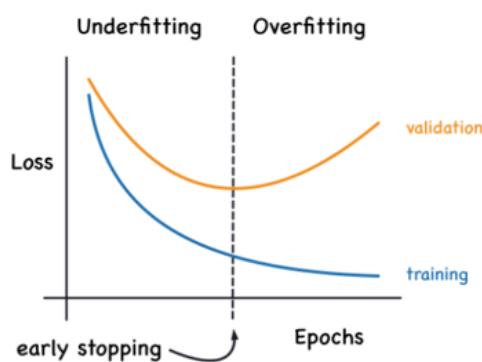


FIGURE 4.18 – Early Stopping

#### 4.3.2.3 1ère approche - MFCC

Après avoir compris comment paramétrier un réseau de neurones simple [11], nous avons commencé par une architecture relativement simple, constituée de 3 couches :

1. Batch Normalization : Normalise le signal en entrée afin d'éviter que le réseau ne s'entraîne sur des valeurs trop extrêmes.
2. Conv1D : Réalise une convolution à une dimension sur les composantes temporelles de nos MFCC
3. Dense : Connecte tous les neurones entre eux et permet de retrouver une sortie ayant les mêmes dimensions que l'entrée, celles-ci ayant changé lors de l'application de la convolution.

Dans cette première approche, nous avions décidé d'utiliser les MFCC (les coefficients de la MFC), car d'après nos recherches, ceux-ci sont très utilisés pour les tâches de speech recognition par IA, notamment car ils permettent un apprentissage plus rapide qu'avec des données obtenues par exemple avec une STFT.

Nous avons donc lancé un premier apprentissage sur les MFCC de 1160 échantillons audio et après un apprentissage relativement court, lié à l'architecture assez simpliste de notre réseau et à l'utilisation des MFCC, nous obtenons le résultat suivant :

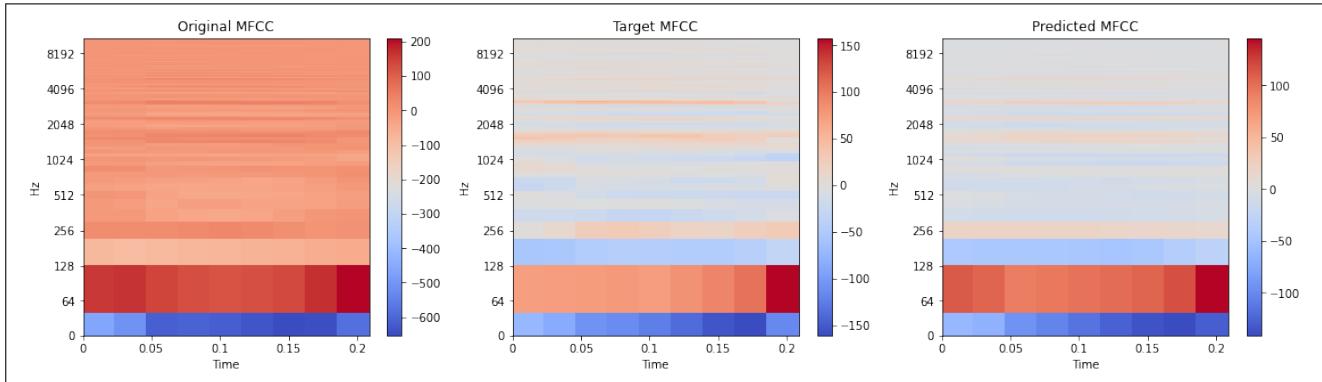


FIGURE 4.19 – Observation d'un échantillon suite au premier apprentissage

On peut voir que la prédiction est extrêmement proche de la cible, même si on peut reconnaître une légère influence de l'échantillon original (donc le son non distordu) dans les basses fréquences lors de 50 premières millisecondes. Plutôt pas mal pour un réseau avec seulement 360 paramètres n'est-ce pas ?

Ce qui nous intéresse ici, c'est principalement le son, on peut donc calculer la MFC inverse afin de pouvoir écouter nos prédictions :

[Play : Original](#)

[Play : Predit](#)

En écoutant la prédiction on pourrait croire que le modèle n'as pas été capable de générer la distorsion attendue et abîme le signal sur lequel il prédit au lieu de lui appliquer l'effet demandé. Après quelques recherches, nous avons décidé de calculer la MFC de notre audio original et d'en calculer la MFC inverse afin de voir l'effet de ces deux conversions sur le signal audio. Nous nous sommes alors rendu compte de la pauvre fidélité de ce type de transformation. Cela nous a conduit à concentrer nos efforts sur l'approche STFT.

Après y avoir développé une architecture plus étroite, nous avons décidé d'essayer de l'utiliser avec une MFC, puisque en dépit de la détérioration du son qu'elle introduit, nous devrions être en mesure d'entendre si une distorsion a bien été appliquée. De plus, puisque les modèles utilisant une MFC convergent plus rapidement que ceux avec une STFT, ceux-ci nous ont permis de tester différentes architectures plus rapidement et en utilisant moins de ressources.

#### 4.3.2.4 2ème approche - STFT

Lors de nos recherches, en plus de la MFC, nous avons trouvé beaucoup de ressources de Machine Learning qui utilisaient la STFT. Ne sachant pas laquelle utiliser, nous avons décidé de travailler sur ces deux approches en parallèle.

Nous avons donc commencé par implémenter la même architecture que pour la MFC, c'est-à-dire un simple réseau à trois couches constitué : d'une Batch Normalization, d'une Conv1D, et d'un Dense.

En lançant le même apprentissage, on obtenait le résultat suivant :

Afin de visualiser plus rapidement les différences qu'il peut y avoir entre la cible et la prédiction, nous avons fait le choix de tracer un spectrogramme correspondant à la différence de ces deux derniers. Il faut donc garder à l'esprit que celui-ci n'est en aucun représentatif d'un quelconque signal, il

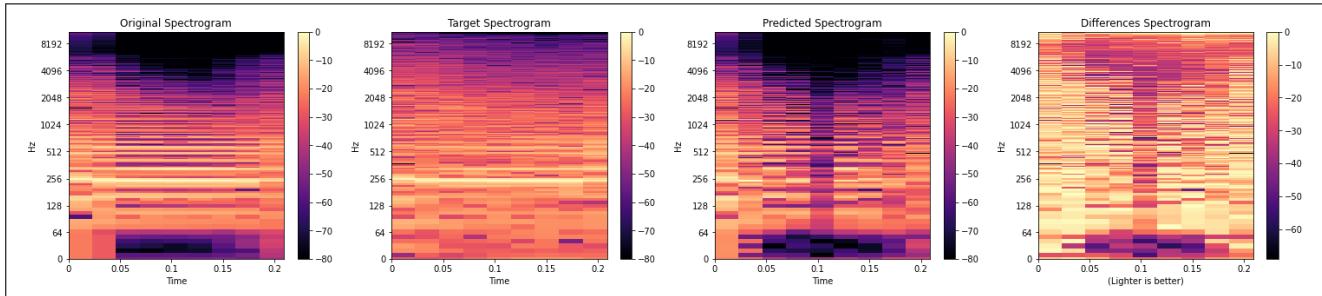


FIGURE 4.20 – Observation d'un échantillon suite au premier apprentissage avec la STFT

est juste là pour visualiser rapidement les différences entre la cible et la prédiction.

On peut alors remarquer que comparé à la MFC où le résultat était très proche de la cible, pour la STFT le résultat est beaucoup moins probant. La distorsion est très peu présente, voir quasiment inexistante.

On remarque cependant que le signal prédict est plus fidèle qu'avec la MFC, ce qui est plus agréable à l'écoute :

**Play : Cible**

**Play : Predit**

Les résultats étaient mieux qu'avec la MFC, mais nous étions encore loin de ce que nous espérons. Nous avons alors continué nos recherches, et c'est en s'intéressant à l'article de Mr Martinez [5], que nous avons compris comment nous pouvions améliorer grandement l'architecture de notre réseau.

Nous avons alors modifié notre architecture en y ajoutant trois couches de Conv1D, chacune suivie d'une couche de MaxPool1D. Cette structure n'est pas sans rappeler celle de LeNet [1].

Ici, nos couches de MaxPool1D prennent deux échantillons à la suite et ne gardent que la valeur la plus importante, divisant ainsi la taille des paramètres par deux.

Nous avons ensuite lancé un apprentissage pour voir si notre architecture apportait une amélioration quant à l'apprentissage du réseau de neurones sur l'effet de distorsion. Nous avons alors obtenu les résultats suivants :

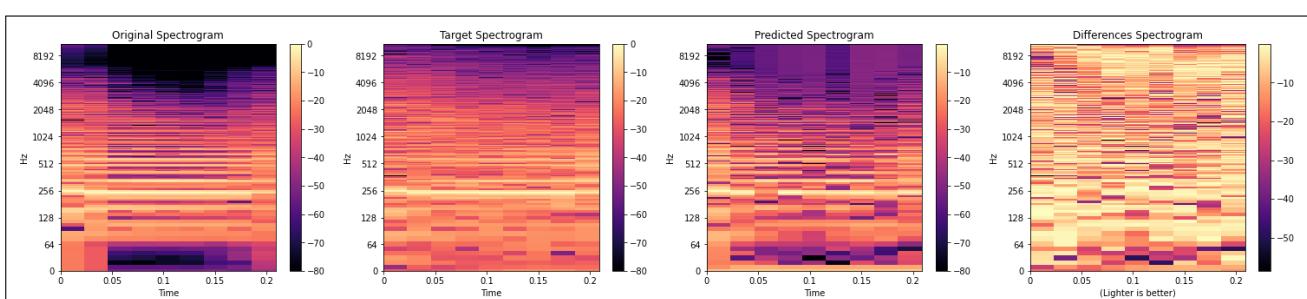


FIGURE 4.21 – Observation d'un échantillon pour la 2ème architecture de notre réseau

Ces résultats sont plus probants que ceux obtenus avec la première architecture, même s'il reste encore du chemin à parcourir avant de répliquer parfaitement notre distorsion. On notera une nette amélioration à l'écoute :

Play : Cible

Play : Predit

En observant l'évolution des Loss Functions au cours des epochs, nous pouvons déjà remarquer que le réseau de neurones apprend bien quelque chose. Ce qui est déjà une bonne chose. De plus, nous pouvons remarquer que la mise en place de notre Early Stopping semble fonctionner, car il n'y a aucune apparition de sur-apprentissage.

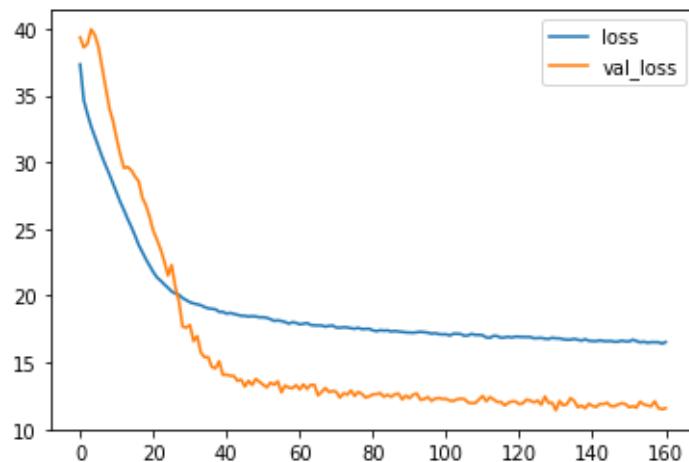


FIGURE 4.22 – Évolution des Loss Functions pour la 2ème architecture

Nous avons ensuite continué dans cette approche en réalisant des convolutions sur les dimensions spectrales et temporelles de nos données, engendrant des modèles beaucoup plus conséquents avec plus d'une dizaine de millions de paramètres. Ceux-ci nous permettent de nous approcher toujours plus de la distorsion attendue :

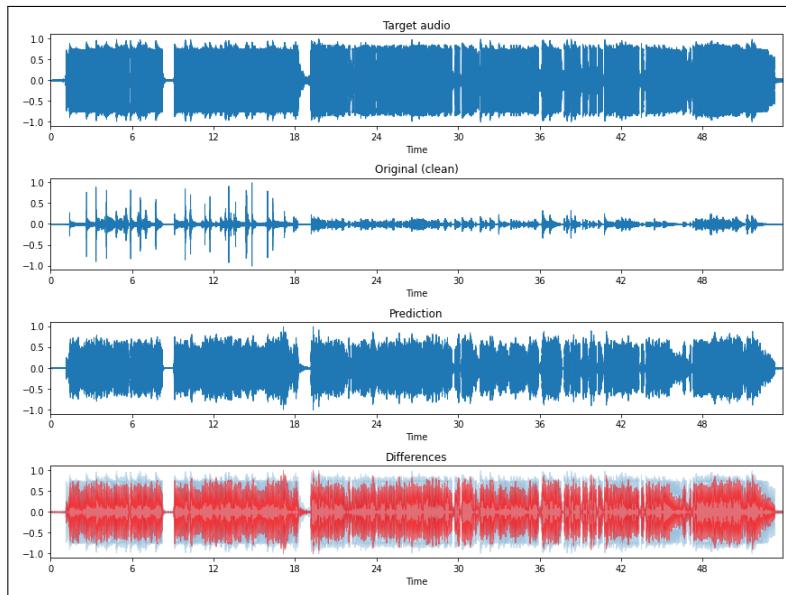


FIGURE 4.23 – Signal temporel des résultats obtenus

Même si le signal temporel ne colle pas complètement avec celui attendu, les composantes spectrales sont bien présentes : les spectrogrammes de la prédiction et de la cible sont quasiment identiques.

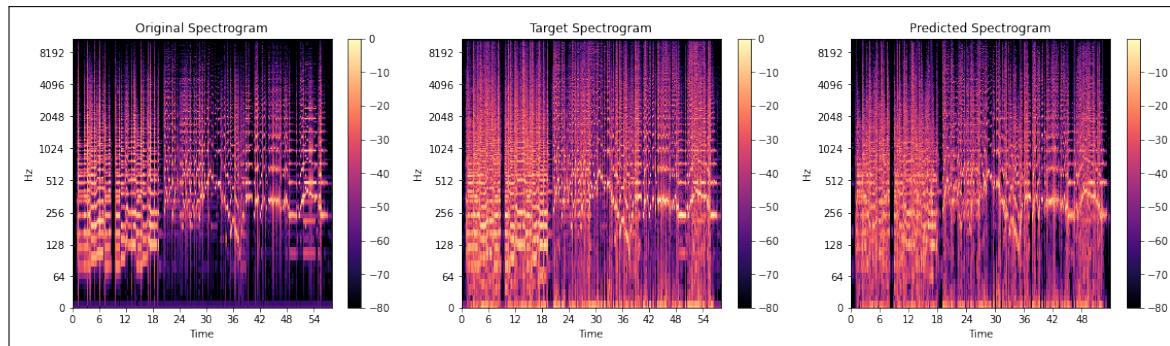


FIGURE 4.24 – Spectrogramme des résultats obtenus

On peut enfin écouter ces nouvelles prédictions qui sont les meilleures que nous avons pu obtenir par cette approche des réseaux de neurones convolutifs :

Play : Cible

Play : Predit

### 4.3.3 Recherche des paramètres et effets audios

Une troisième approche a été la recherche paramétrique. En effet, en se recentrant sur l'idée originelle du projet, et face à la complexité des autres méthodes étudiées, nous avons opté pour une approche plus directe. L'idée était de communiquer au ToneCrafter une chanson d'un certain artiste et il règle automatiquement les effets pour s'approcher du son de la manière la plus fidèle. Pour ce faire, il fallait tout d'abord trouver comment implémenter les différentes pédales d'effet sur Python, mais surtout comment implémenter la recherche des paramètres de chaque effet.

Il existe différents types d'effets utilisés par les guitaristes. Il y a tout d'abord la distorsion, un effet qui a déjà été abordé plus tôt. Mais aussi la *Reverb*, le *Delay*, le *Chorus*, le *Trémolo*, ou encore les différents filtres. Chacun peut avoir son importance, en particulier si le guitariste que l'on veut imiter les utilise. C'est pour cela qu'il sera important d'implémenter une myriade d'effets différents si l'on veut se rapprocher des sons de guitare souhaités.

La méthode de recherche paramètre s'appuie sur l'utilisation des spectrogrammes, ainsi on transforme un son de quelques centaines de millisecondes en une image. L'intérêt de cette est que l'on va pouvoir observer plus aisément le son qu'on écoute pour mieux comprendre l'apparitions de certains harmoniques par exemple. Mais surtout, de pouvoir comparer différents sons entre eux plus aisément.

#### 4.3.3.1 Implémentation des effets

Approcher des effets et des phénomènes analogiques sur Python n'est pas toujours chose aisée. En effet, il convient de s'en approcher à l'aide de fonction mathématiques, ou d'un traitement du signal un peu plus spécifique. En particulier pour la distorsion, qui est un effet très singulier. En effet la distorsion ou sa cousine l'overdrive se caractérisent par la perte de linéarité dans le signal sonore. Nous vient alors la question de comment recréer cette perte de linéarité.

Nous avons alors trouvé des formules qui permettront d'approcher ce son si singulier :

La fonction *Distortion* est définie par :

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

On la représente maintenant pour un niveau d'entrée allant de -1 à 1 :

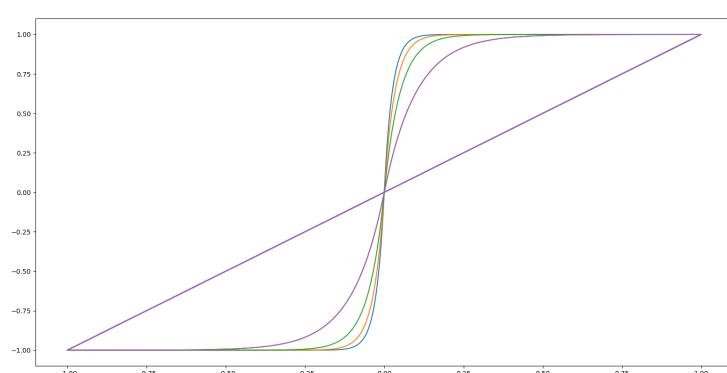


FIGURE 4.25 – Représentation graphique de la distorsion pour différentes valeurs de gain

On remarque qu'il s'agit d'un hard clipping, c'est à dire d'une distorsion plutôt violente du signal. De plus, pour un gain nul, la fonction renvoie bien le signal initial.

La fonction *overdrive* est définie par :

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

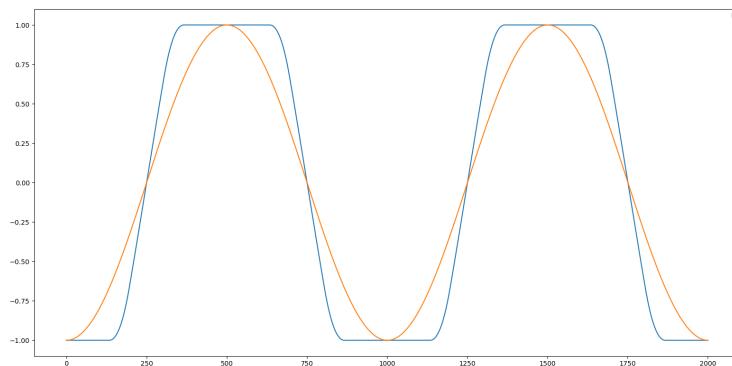


FIGURE 4.26 – Représentation graphique de l'overdrive

En comparant avec la courbe jaune on observe bien la perte de linéarité due à la fonction. Notons que pour simplifier l'étude nous avons procédé à une distorsion symétrique du signal. Mais dans bien des montages, il convient d'avoir un écrêtage plus important pour les signaux positifs que négatifs.

Mais la distorsion et l'overdrive ne sont pas les seuls effets utilisés par les guitaristes. Certains sont friands de chorus et reverb comme le David Gilmour, le guitariste de Pink Floyd. Sont utilisées aussi les pédales delay, tremolo, Wah-Wah. Enfin pour donner une certaine couleur à leur son les guitaristes utilisent aussi des égaliseurs, qui sont des filtres, pour les simuler nous utiliseront des filtres passe-haut, passe-bas etc..

Sur Python, mise à part les filtres fournies par certaines bibliothèques, comme les filtres *Butterworth*, la plupart des effets utilisent la décomposition des effets sous forme de listes de flottants. En effet pour la *Delay*, signal de sortie  $y[t]$  se compose du signal d'entrée  $x[t]$  mais aussi du signal d'entrée  $x[t - d]$ . Où  $d$  est variable, et indique le temps d'attente entre un son et son écho. Pour l'effet *Reverb*, c'est plus ou moins le même fonctionnement mais à l'aide d'un paramètre  $d$  plus petit.

Pour ce qui est de l'effet *Flanger*, on vient appliquer le même procédé, mais cette fois ci le paramètre  $d$  varie selon une fonction bien spécifique. En effet on a :

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

Où  $A$  est l'amplitude de la variation de  $d$  (de l'ordre de 50)  $rate$  est la fréquence d'échantillonage et  $period$  la période d'oscillation de  $d$ . Enfin comme la *Chorus* n'est qu'en principe l'utilisation de

plusieurs *Flangers* où nous faisons varier les paramètres *A* et *period*, nous en utiliseront 3 par la suite. Après test, et écoute c'est largement suffisant.

Enfin, Nous avons aussi implémenté un générateur de bruit rose, plus agréable et plus adapté que le bruit blanc qui devait à la base, servir à imiter le souffle créé par certaines pédales d'effet ou amplis. finalement cela nous servira plus à observer les limites de nos fonctions face à l'ajout de bruit.

#### 4.3.3.2 Recherche des paramètres des effets

La seconde partie de cette méthode, et la plus importante, fut l'implémentation d'un algorithme de recherche de chaque effet. Pour cela, nous sommes venues plein de questions. Comment faire reconnaître à un programme la proximité ou non de deux sons de guitare ? A-t-on besoin de tous les effets ? Dans quel ordre devons-nous placer les pédales ? Doit-on jouer parfaitement la mélodie afin que le programme puisse reconnaître les effets utilisés ?

Nous avons tout d'abord chercher le moyen le plus efficace de comparer une simple note à une autre. Pour cela, l'utilisation des spectrogrammes semblait évidente mais pas pour autant suffisante. En effet pour de basses fréquences, comme celle du *Mi* grave de la guitare : 82 Hz, on remarque bien la précision nécessaire (de l'ordre du demi hertz) afin de bien représenter le son d'une guitare. Cela implique notamment des temps de calcul plus long, car on augmente la précision du spectrogramme.

Puis nous sommes venus le problème de la comparaison des sons. Pour comparer deux sons, il faut tout d'abord qu'il s'agit de la même note, ou au moins que la note soit très proche. en prenant cela en compte, il l'on prenait comme exemple le son de Brian May, le guitariste de Queen, à vouloir imiter, faudrait-il rejouer parfaitement le solo de guitare et le comparer ensuite à l'original ? Bien évidemment nous dûment trouver une autre solution. Nous avons donc eu l'idée de segmenter le fichier audio du solo de Bohemian Rhapsody afin que chaque segment ne contienne qu'une note. Puis nous allons comparer chaque segment, d'une certaine note, au son de la même note d'une guitare sans effet.

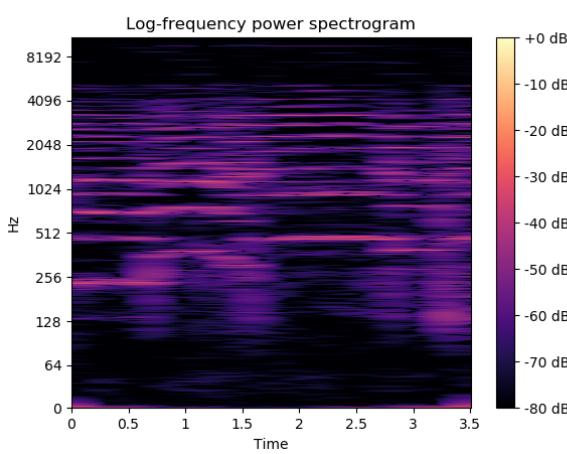


FIGURE 4.27 – 3.5s du solo de Brian May

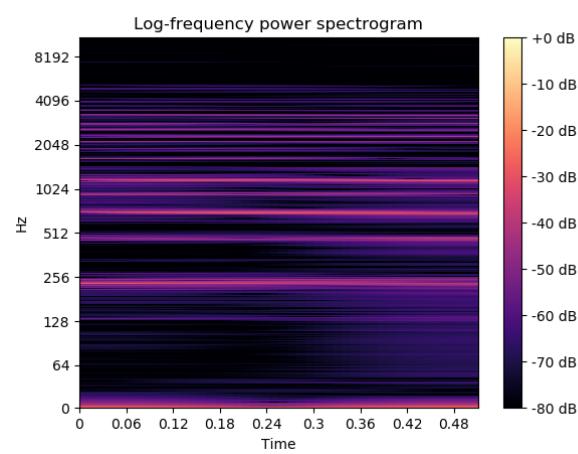


FIGURE 4.28 – 0.5s du solo de Brian May

Notons que ces spectrogrammes, ont été réalisé après application d'une série de filtres coupe bande. Ces filtres atténuent toutes les fréquences entre deux notes. Cela permet déjà de retirer du bruit, mais aussi de mieux observer les spectrogrammes, ces filtres seront utiles pour la suite. On remarque qu'il n'y a pas de variation de notes. En effet, les lignes du spectrogrammes sont bien horizontale, on reste donc bien sur la même fréquence. Cependant un hachage de 500ms permet

quelques fois que deux apparaissent, c'est un axe d'amélioration du programme. Cependant, comme nous n'utiliserons que quelques séquences du solo de 0.5s, cela sera suffisant pour continuer.

Maintenant que nous avons les notes singulières à étudier, il reste encore à trouver à quoi les comparer. Il fallait pour ça une base de données comprenant toutes les notes de la guitare. Cette base de données nous a été fournie par la *Fraunhofer Institute for Digital Media Technology IDMT*, elle comprends des enregistrements de plusieurs instruments, afin d'y faire du traitement de signal. Nous avons donc utiliser le son d'une *Fender Stratocaster*, c'est l'un des modèle les plus répandu et son son se décline relativement bien avec des effets. Cette base de données est composée d'un fichier pour chaque note possible de la guitare. Cependant une note à la guitare peut se jouer sur différents endroits sur le manche. Il existe parfois jusqu'à trois manière de jouer une note sur une guitare.

Un fois notre base de données prête, reste à savoir comment comparer la note sélectionnée au bon fichier de la base de données. Pour cela il faut déjà savoir de quel note il s'agit. Nous avons donc réalisé un programme qui, à un son donnée nous renvoie de quelle note il s'agit et les fichiers de la base de données correspondants à cette note (les noms de ces fichiers font apparaître la case à presser et la corde à pincer de la guitare pour reproduire une note donnée).

Lorsqu'on reprend la note sélectionnée du solo de Brian May :

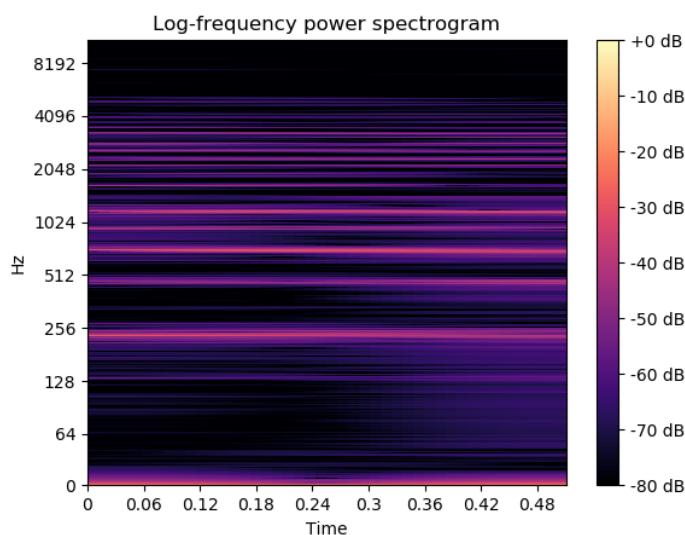


FIGURE 4.29 – Représentation graphique de l'overdrive

On observe bien l'apparition de beaucoup d'harmoniques dues notamment à la distorsion de la guitare, c'est ce que l'on va essayer de reproduire. Cependant, une fréquence semble quand même ressortir au alentours de 250Hz

L'algorithme nous renvoie :  $La\#2=233,08$  Hz, ce qui correspond bien à ce que l'on observe

On classe les notes de musique 12 par 12, ceci allant du *Do* au *Si*, puis selon l'octave on ajoute un chiffre en 1 et 12. Ici la première note du solo de Bohemian Rhapsody est un *La#2*. À cette note correspondent trois fichiers de la base de données, nous nous restreignons au deuxième (*G53-58308-1111-00035.wav*) plus proche de la note à imiter. L'algorithme de détection de note étant certes très basique, il reste néanmoins très efficace pour notre étude. Mais nous avions l'idée de l'améliorer afin

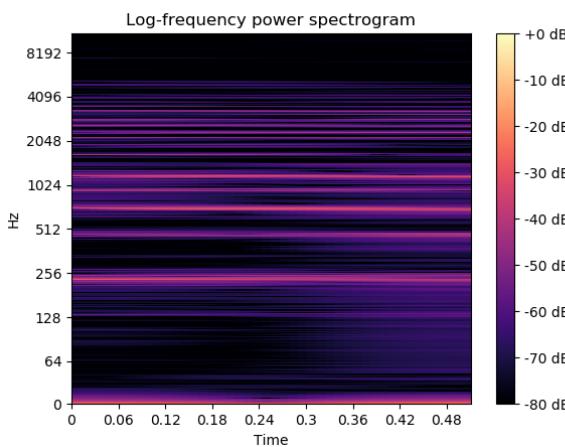


FIGURE 4.30 – Première note Brian May

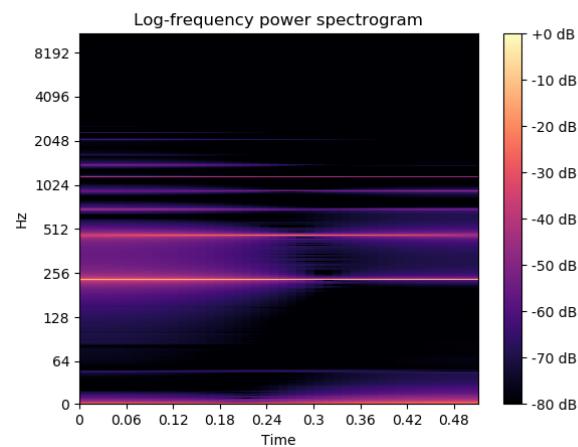


FIGURE 4.31 – Note en son clair correspondante

d'éviter les erreurs possibles dues à une trop grosse distorsion de la note, des effets trop importants ou encore un bruit trop important.

On peut maintenant observer les deux spectrogrammes des deux notes sélectionnées :

Notons que les deux notes ne sont pas prises à leur tout débuts car c'est ce qu'on appelle l'attaque, c'est le son produit par la corde dès qu'on la gratte. C'est un son qui dépend de trop de paramètres pour qu'on puisse l'utiliser dans notre étude. En effet, Brian May, par exemple, utilise non pas un médiator pour gratter sa guitare mais une pièce de monnaie anglaise. Cela donne un son plus métallique et agressif à l'attaque. Il en va de même pour chaque corde et médiator que pourrait utiliser un guitariste, leur combinaison donnerait toujours au timbre unique à l'attaque toujours.

On observe les mêmes piques d'intensité à 233.08 Hz et 566,16 Hz

Enfin, maintenant que nous savons quelle notes utiliser il reste encore à les comparer. Pour cela on se base sur les spectrogrammes, étant des images, on va simplement faire fonctionner une fonction d'inter-corrélation, en paramétrant chaque itération de l'algorithme avec des valeurs différentes. Lors que l'on atteindra l'image la plus proche de celle issue du solo de Brian May, on renvoie les paramètres utilisés pour obtenir cette image.

En vue de la précision des spectrogramme, la quantité de calcul à effectuer est relativement élevée c'est pour cela que nous nous sommes restreints pour commencer, à ne chercher les paramètres que des fonctions de *Distortion* et d'*Overdrive* dans un premier temps. Puis d'étendre cette recherche à chaque effet. Notons que l'ordre des effets ayant son importance, nous utilisons la conventions utilisée par la plupart des guitaristes qui définit l'ordre des effets à appliquer : *Compression - Wah Wah - Overdrive/Distortion - Equalizer - Chorus/Flanger - Delay - Reverb*.

Les résultats obtenus sont assez loin de ce que l'on espérait au départ, sauf pour la distorsion ou l'*Overdrive*, les possibilités étaient tellement immenses qu'il faut trouver des moyens de converger plus rapidement vers un résultat satisfaisant, notamment en restreignant la plage possible des paramètres.

Cependant en partant de la *Distorsion/Overdrive* trouvé et en ajustant certaines autres pédales,

on arrive à des résultats relativement cohérents et agréable à écouter (l'*Overdrive* donne le meilleur résultat).

#### 4.3.3.3 Implémentation en temps réel

L'un des objectifs de ce projet était aussi de jouer de la guitare avec nos effets en temps réel. C'est pour cela que nos algorithmes sur Python se mariaient bien avec cette fonctionnalité. En effet, chaque programme, représentation chaque effet, nous renvoie une liste de flottants qui est directement pris en charge en temps réel. Python n'étant certes pas le langage adapté au temps réel, pour des raisons de gain de temps, nous avons pu tester nos fonctions. De plus nous avions déjà fait ce programme pour la partie DDSP cela nous a permis de gagner un peu de temps.

Dire que cet algorithme fonctionne en temps réel est un abus de langage. En réalité, il enregistre 1024 échantillons, fait passer ces 1024 à travers toutes les fonctions d'effets, et nous ressort le son obtenu. De plus pendant que nous écoutons ces 1024 échantillons, l'algorithme réenregistre encore 1024 échantillons. Ce programme fonctionnant en flux tendu tout le temps, nous n'entendons pas d'interruption.

Cependant plusieurs problèmes sont arrivés lorsque que nous avons utilisé certains effets comme le *Delay*, la *reverb*. Ces effets utilisent des échantillons passés, parfois trop éloignés (plus de 1024 échantillons d'écart avec le présent), ainsi il y a perte d'information et on entend des coupures. Pour y remédier nous avons ajouter une mémoire tampon, qui enregistre tous les échantillons qui pourront être réutilisés par la suite par les différentes fonctions d'effet.

Un autre gros problème à été la qualité du signal envoyé par la guitare. En effet, si ce signal est bruité, le son renvoyé sera inaudible s'il passe par des distorsions. Pour tester les limites nous avons ajouter du bruit rose, et pour un valeur de l'ordre du dixième du niveau de la guitare le son était déjà bien impacté.

# Chapitre 5

## Réalisation matérielle

QUENTIN WACONGNE, HECTOR RICHARD

### 5.1 Cahier des charges

Nous sommes partis du constat qu'il était très difficile, plus particulièrement pour des novices de parvenir à correctement régler une pédale d'effet afin d'obtenir le son désiré. De plus, une pédale d'effet est un objet qui doit être transportable et robuste afin de pouvoir être utilisé dans des conditions de contraintes mécaniques assez fortes, comme sur scène lors d'un concert.

Notre objectif est donc de réaliser un support matériel, qui soit robuste, transportable et plug and play, abritant des composants électroniques permettant de réaliser du traitement audio en temps réel.

Notre cible serait donc des musiciens de tous niveaux et horizons, utilisateurs actuels ou non de pédale à effet et souhaitant sonner de la même manière qu'un de leurs artistes préférés sans avoir à passer par une étape de réglages fastidieuse.

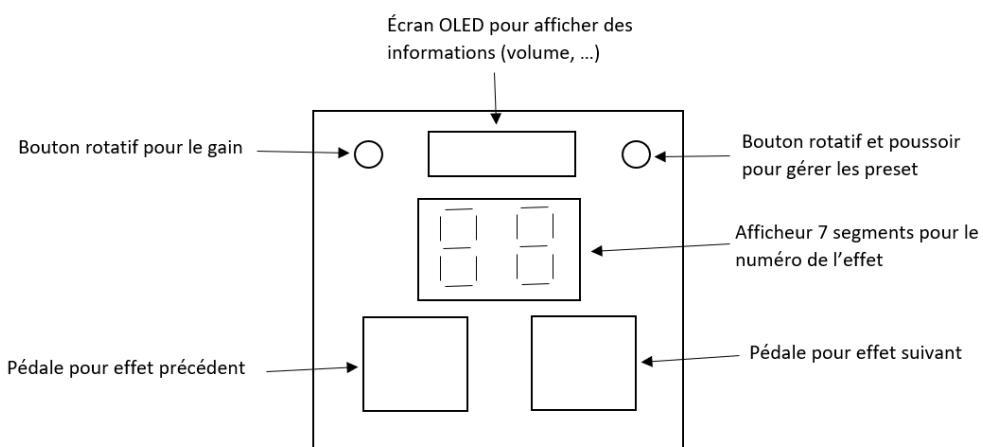


FIGURE 5.1 – Prototype de design de notre pédale (dimension 14x17cm)

### 5.2 Design de la pédale et fonctionnalité

Tout d'abord, notre idée de design pour la pédale est fortement inspirée de celui de pédales existantes. En effet, nous souhaitons que notre produit possède la même robustesse que ces dernières afin de pouvoir être utilisé dans des situations plus contraignantes d'un point de vue matériel, par exemple sur une scène durant un concert.

Comme vous pouvez le voir sur le schéma ci-dessus, nous le voulons assez simple avec 2 pédales permettant de passer d'un set à un autre, 2 afficheurs, 7 segments pour afficher le set actuel, un écran OLED et 2 boutons rotatifs pour régler le gain ou gérer les présets de manière plus précise. Le boîtier sera donc de forme rectangulaire avec les dimensions suivantes : longueur 17cm, largeur 14cm, hauteur 4cm. A l'intérieur du boîtier, on peut retrouver la partie commande du système. On y trouve

principalement un microprocesseur, un codec et les différentes mémoires.

Cette configuration permet d'avoir une interface simple, fiable et rapide à utiliser. En effet, une entrée comme un simple bouton poussoir peut avoir plusieurs fonctionnalités. On peut imaginer une action par une simple pression, une autre action avec deux pressions succinctes, encore une autre action avec une pression longue. Ainsi, on peut faire des économies sur les broches et les ports, mais surtout faciliter la gestion des interruptions et des événements par le microprocesseur.

Toute cette partie logique sera en charge de gérer les entrées sorties, dont notamment les boutons de l'interface, et l'acquisition audio, des calculs afin d'appliquer les effets sur le signal d'entrée,

## 5.3 Choix des composants

Le processeur :

Pour le microprocesseur, nous avons choisi une STM32H743ZITx. Ce microprocesseur possède 2 Mo de flash, 1 Mo de RAM et une fréquence d'horloge de 480 MHz. Il s'agit d'un microprocesseur de haute performance car pour des traitements en temps réel du son, une grande quantité de RAM ainsi qu'une fréquence élevée sont préférables. Une autre fonctionnalité intéressante avec ce microprocesseur est qu'il est compatible avec le compilateur de l'IDE "CubeAI". Cet utilitaire nous permet à partir d'un fichier contenant le modèle et les algorithmes de notre intelligence artificielle, de pouvoir faire une implémentation en langage C. Ainsi, nous possédons une architecture optimisée et capable d'exécuter des tâches à bas niveaux. Sur ce microprocesseur, on retrouve également des circuits pour de l'accélération matérielle comme des ASIC. Cela veut dire que si la bonne structure des données est appliquée, il est possible de mobiliser plusieurs circuits dédiés, et donc d'être plus rapide. Un des circuits qui nous intéresse le plus est le "X-CUBE-AUDIO-F7". Le "X-CUBE-AUDIO-F7" est un composant de traitement audio pour la série STM32F7 et consiste en un ensemble complet de bibliothèques logicielles efficaces et de haute qualité dédiées à l'accélération matérielle, prêtes à être intégrées dans de nombreux types d'appareils audio. Tous les modules sont livrés avec un traitement interne de 32 bits et supportent des tampons d'E/S de 16 ou 32 bits. Il devient donc possible entre autre d'avoir accès à un "Bass manager", un contrôle de volume intelligent et un "gain manager". On se retrouve donc non seulement avec des briques logicielles que nous n'avons pas à réaliser, mais surtout des briques logicielles garanties optimisées par le constructeur.

Le codec :

Nous avons choisi le TLV320AIC23B. Ce codec audio a une fréquence d'échantillonnage de 96 kHz. Nous avons choisi celui-ci car nous voulions un codec suffisamment rapide et efficace car à l'instar du microprocesseur nous avons besoin de composants ayant une capacité de calcul et de mémoire suffisante afin de pouvoir effectuer du traitement de signal en temps réel. Ce codec possède aussi plusieurs ports séries. On pourra donc donner des instructions au codec.

L'alimentation :

Afin de pouvoir proposer la solution la plus souple, et la plus facile à utiliser, il a été choisi d'utiliser un câble d'alimentation Jack. Ce câble possède une fiche ronde, ce qui rend le branchement très simple. En effet, il est inutile de se soucier du sens dans lequel le câble est branché. Le câble jack délivre du 5V que nous régurons en 3,3V dès l'entrée dans le circuit. Cette solution nous permet de pouvoir à la fois alimenter des composants avec une tension de 5V, mais aussi d'autres composants avec une tension d'alimentation de 3.3V.

## 5.4 Réalisation du PCB

### 5.4.1 Réalisation du schematic

Après avoir choisi nos composants en fonction de nos besoins et choisi le design de notre pédale, nous avons réalisé le circuit imprimé qui contiendra ces derniers sur le logiciel Eagle. Dans un premier temps, il convient de réaliser le "schématique" de notre projet, celui-ci nous donnera un rendu visuel global et schématique de l'ensemble. Puis à l'aide du logiciel, nous pourrons générer la "board". Sur celui-ci, nous allons mettre nos différents composants comme les résistances, condensateurs, etc. nécessaires à notre projet. Une fois que nous avons mis l'ensemble de nos composants, il convient de les relier entre eux soit par des fils soit par des labels (deux fils portant le même nom seront automatiquement reliés). Les labels permettent d'augmenter grandement la lisibilité. Pour augmenter encore la lisibilité et rendre plus facilement compréhensible l'ensemble, il est utile de regrouper intelligemment différents composants en "modules" que l'on peut nommer. Par exemple, nous avons découpé notre projet en plusieurs modules, comme expliqué ci-dessous.

La partie principale comprenant notre processeur et plusieurs composants permettant son bon fonctionnement tel que : l'oscillateur à quartz et son montage nous servant d'horloge, le bouton de reset et son circuit anti rebond, des DEL qui nous permettront de réaliser des tests et divers borniers pour les flux de données et de contrôle (respectivement I2S et SPI dont nous préciserons le fonctionnement dans la partie communication entre les composants) :

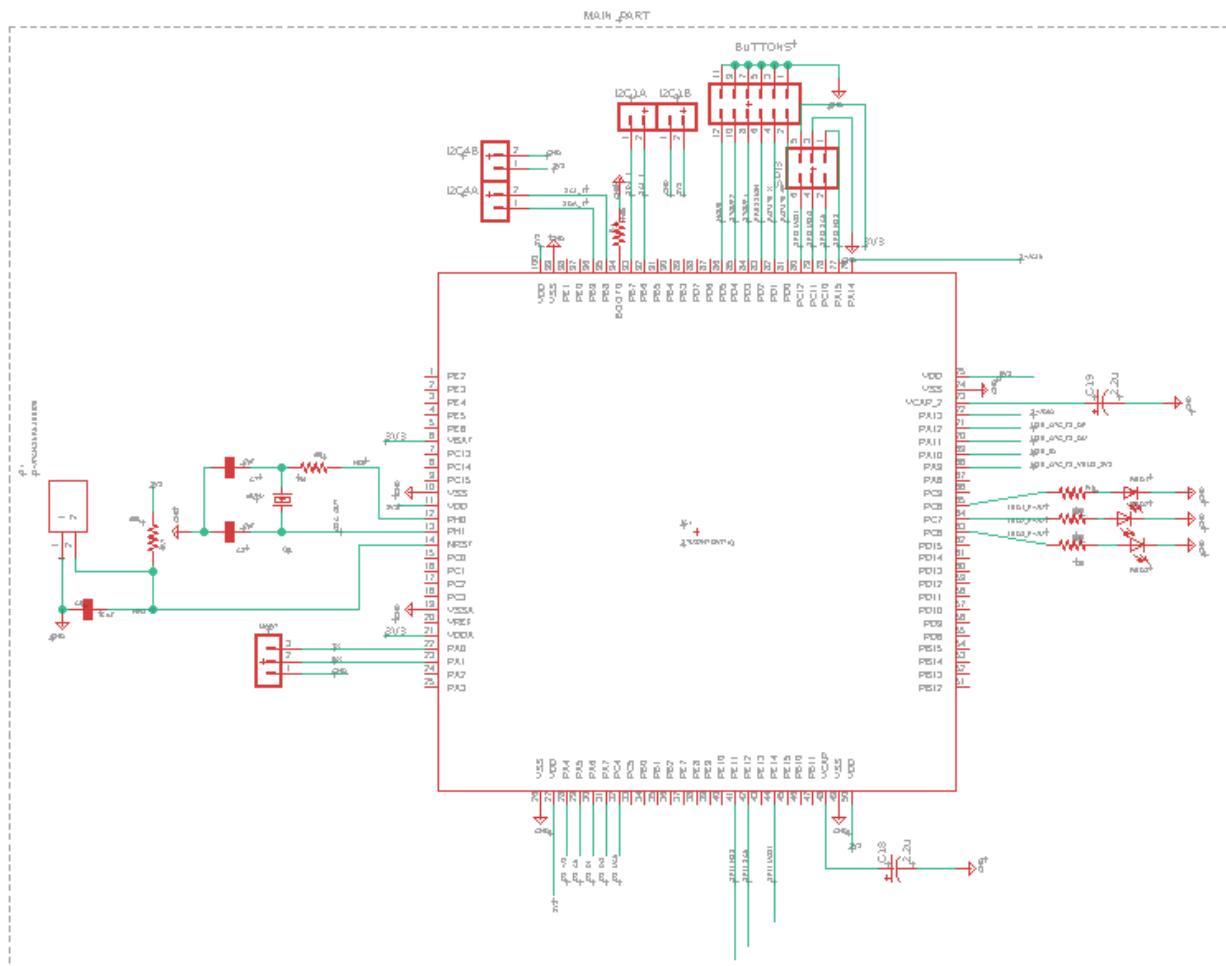


FIGURE 5.2 – Partie centrale du schéma EAGLE

La deuxième plus grosse partie de notre circuit imprimé sera donc notre codec audio et les éléments nécessaires à son fonctionnement. Sur ce bloc, nous avons aussi notre entrée et notre sortie audio via deux ports jack qui nous permettrons d'acquérir notre signal audio brut puis de le restituer après y avoir appliqué les effets désirés.

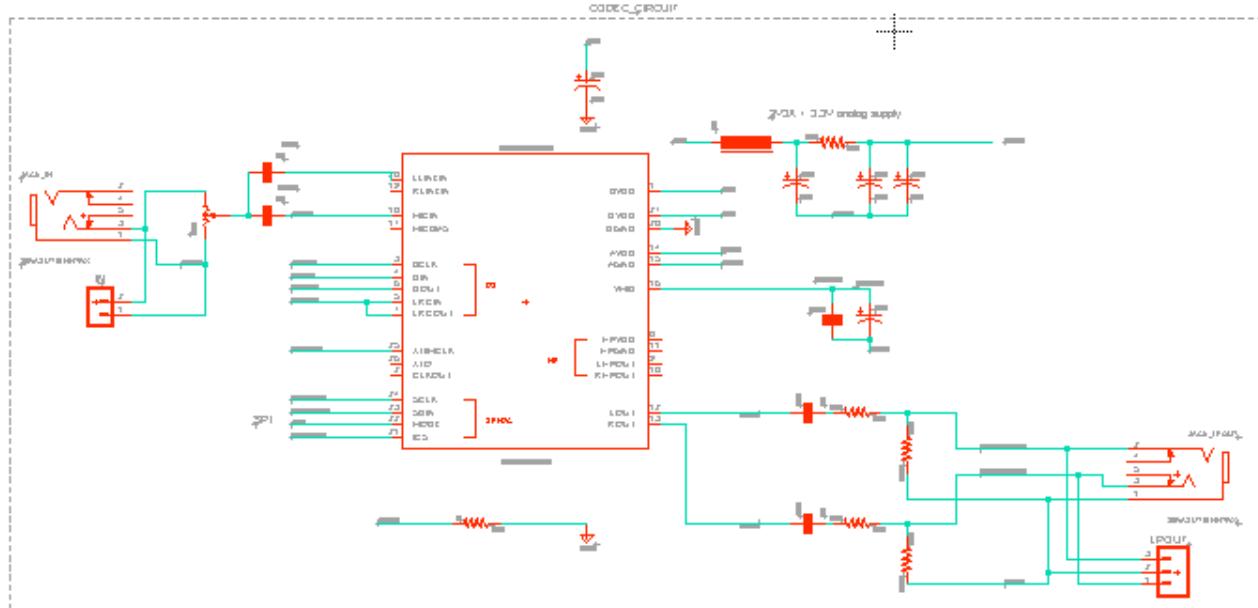


FIGURE 5.3 – Bloc codec audio

Ensuite, nous avons la partie alimentation comprenant notre alimentation à 5V abaissée en 3.3 à l'aide du régulateur de tension TLV7553PDBVR ainsi que des condensateurs de découplage et un condensateur Bulck dont l'utilité sera explicitée ultérieurement (partie routage et CEM).

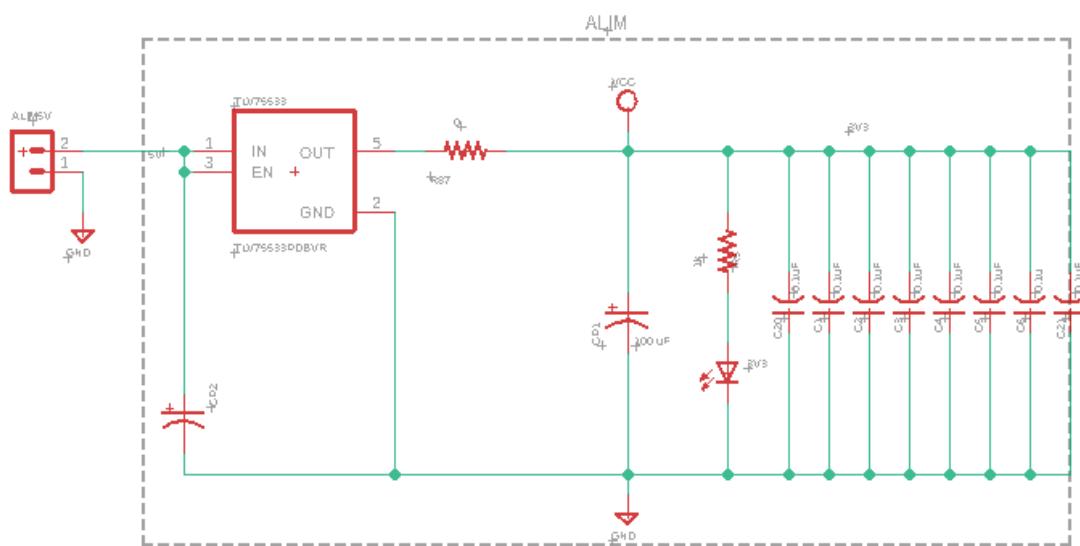


FIGURE 5.4 – Alimentation 5V to 3.3V

Ensuite, nous avons la partie connectique USB, qui nous permettra de relier le ToneCrafter à un ordinateur afin de pouvoir le mettre à jour ou bien de charger de nouveaux presets.

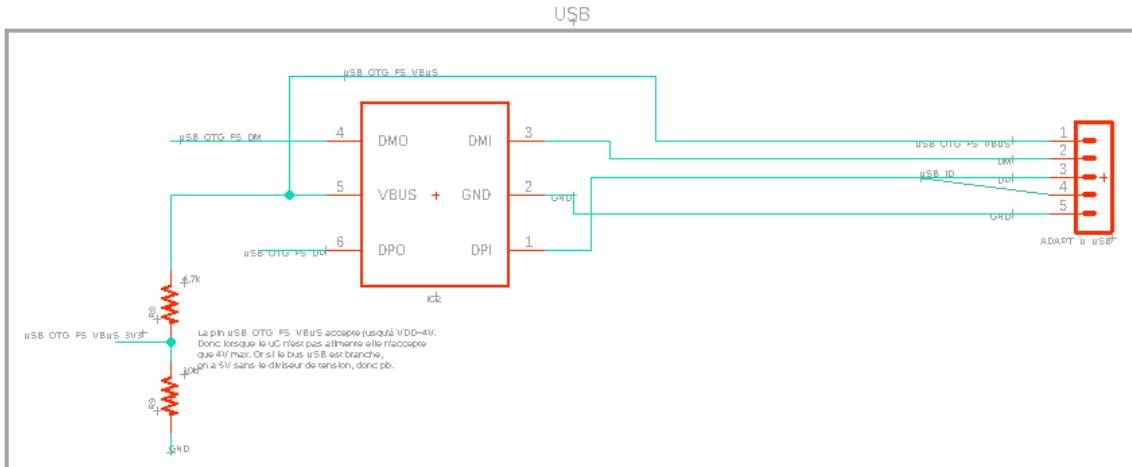


FIGURE 5.5 – Connectique USB

Enfin, nous avons un petit module "STlink debug" qui nous permettra de communiquer avec notre processeur, le reprogrammer, et effectuer des tests.

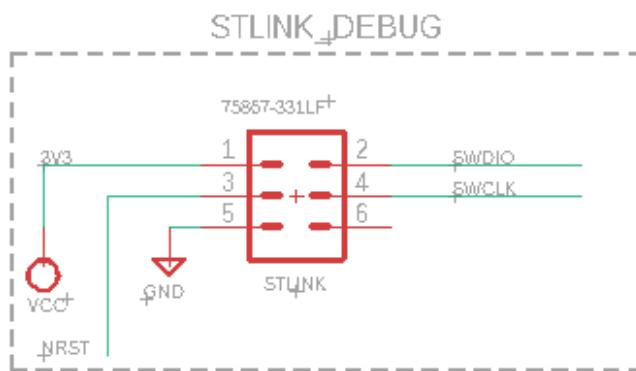


FIGURE 5.6 – STlink debug

#### 5.4.2 Routage et compatibilité électromagnétique

Tout d'abord, notre circuit imprimé doit être suffisamment compact pour être facilement intégré au boîtier. Nous sommes donc partis sur un circuit imprimé en 6.8x6.2cm . Nous avons choisi d'utiliser la technologie 4 couches pour notre PCB pour plusieurs raisons. Simplement, en raison du nombre de composants et de la complexité du routage, nous avons opté pour cette technologie. Il y aura donc une couche d'alimentation 3.3V, une couche de masse et deux couches (Top et Bottom) pour les composants et le routage. Cela permet de simplifier grandement le routage en permettant une mise à la masse, ou à l'alimentation directe grâce à des via (ce sont des "puits" permettant de relier deux couches différentes entre elles). On évite ainsi des pistes excessivement longues pour aller chercher la masse ou l'alimentation. Cela permet aussi d'éviter les croisements potentiels entre différentes pistes en passant de la couche top à la couche bottom, c'est-à-dire en faisant passer une piste sous une autre. De plus, le fait d'avoir un plan de masse permet aussi de limiter les interférences. En effet, nous travaillons à haute fréquence car nous voulons réaliser du traitement audio en temps réel. Or, à haute fréquence, il est très important que les signaux de retour empruntent le chemin le plus

direct possible vers la masse afin de limiter au maximum les interférences. De plus, avoir un plan de masse permet d'avoir une impédance plus faible qu'une simple piste et offre donc une meilleure capacité d'absorption des pics de bruit pouvant apparaître sur la masse lors des changements d'états des circuits numériques.

Il convient maintenant de placer nos composants avant de commencer le routage. Pour cela, nous allons utiliser nos blocs précédemment définis et quelques astuces. Tout d'abord, on place au centre le cœur de notre pcb, c'est-à-dire le bloc principal du processeur car il sera relié aux différents blocs et doit donc être placé au centre (à noter que sur la figure 5.7, le nom du processeur n'est pas le même, mais leur empreintes sont les mêmes). Nous utilisons donc bien celui décrit dans la partie choix des composants). Il faut penser à placer l'oscillateur à quartz au plus près possible de notre processeur car trop de longueur de piste peut entraîner une déformation du signal d'horloge très haute fréquence. Ensuite, nous plaçons le bloc du codec en-dessous du processeur. Pour les connectiques d'entrée et de sortie jack, nous les placerons côté à côté en bordure inférieure gauche de la carte afin de pouvoir facilement plucher notre instrument en entrée et nos amplis en sortie.

Le bloc d'alimentation sera placé sur la partie droite du pcb. Il y a ici une subtilité car ce bloc contient les condensateurs de découplage. Or, il convient de placer convenablement ces derniers. En effet, ces condensateurs sont un élément important pour la CEM dans la mesure où leur but est de servir de réservoir d'énergie local qui va permettre de réduire les fluctuations de potentiel d'alimentation et de masse et de réduire le bruit, comme nous pouvons le voir sur le graphique ci-dessous.

Nous prendrons des condensateurs d'une capacité de 0.1 uF pour découpler car on travaille en haute fréquence. Ils sont à placer au plus près possible des composants pour maximiser leur efficacité. Nous aurons aussi un condensateur "Bulck" qui lui sera à placer en aval de notre abaisseur de tension. Son rôle sera de maintenir la tension d'alimentation continue constante surtout au moment où il y aura de forts appels de courant (par exemple lorsque beaucoup de circuits commutent au même instant). Nous prendrons ici une capacité de 100uF car il n'interviendra pas en haute fréquence.

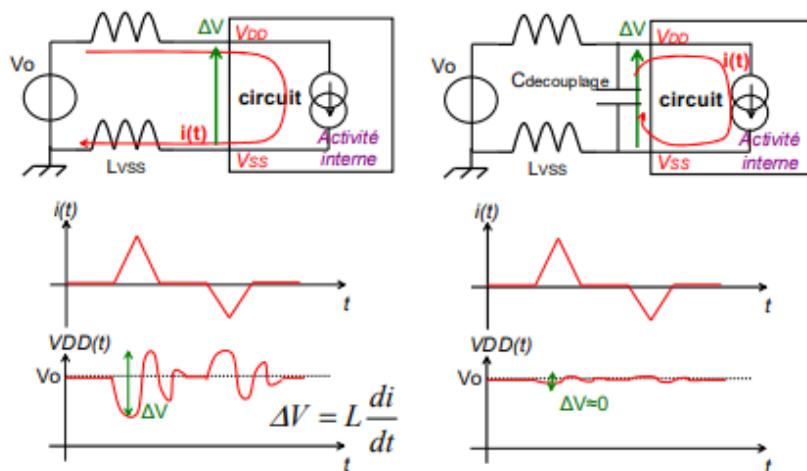


FIGURE 5.7 – Effet d'une capacité de découplage : circuit non découplé à gauche et découplé idéalement à droite

Le bloc de connection USB sera placé sur la partie supérieure droite de notre pcb, juste en dessous du bloc st-link qui sera dans le coin supérieur droit.

Une fois les différents composants convenablement placés, il faut réaliser le routage, c'est-à-dire le dessin des différentes pistes qui vont relier nos composants. Tout d'abord, il faut éviter au maximum les angles de plus de 45°, les pistes trop longues (impédance parasite forte), les pistes trop proches (diaphonie) ou bien privilégier des pistes plus larges pour les pistes à fort courant pour limiter les

résistances et inductances parasites, tout cela pouvant entraîner des problèmes de CEM. Ensuite, il conviendra d'utiliser des via afin de relier directement nos composants aux plans d'alimentation et de masse. Il sera utile d'utiliser la couche inférieure afin de pouvoir "croiser" des pistes en faisant passer l'une sous l'autre afin d'éviter de longs contournements et de simplifier l'ensemble.

Une fois l'ensemble de nos pistes réalisé, le rendu de notre "board" sur eagle est le suivant :

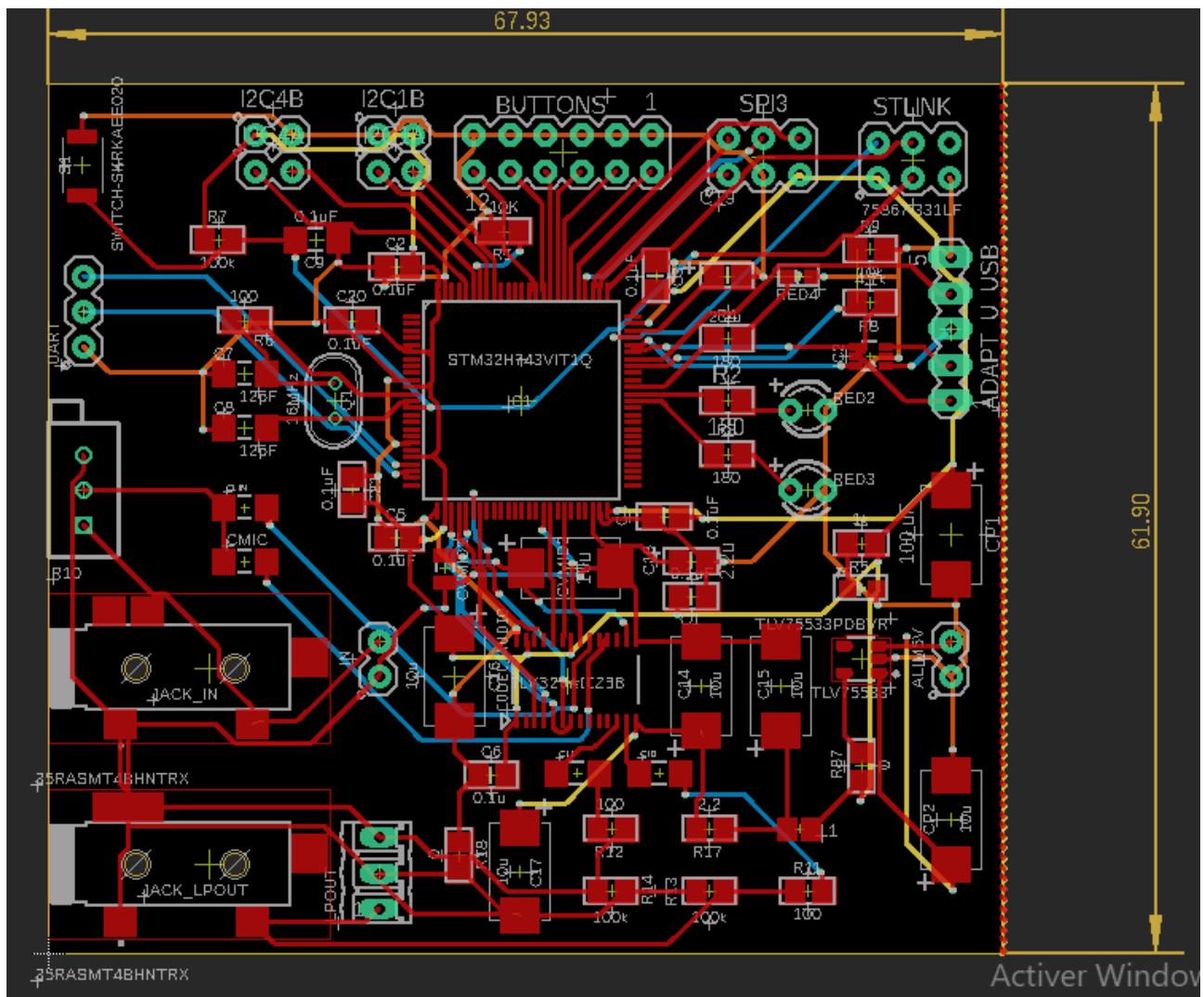


FIGURE 5.8 – Schéma final du routage de notre circuit imprimé

#### 5.4.3 Les méthodes de communication

Le DMA :

Signifiant Direct Access Memory, le DMA est une fonctionnalité permettant un envoi direct et bidirectionnel des données entre un périphérique et la mémoire. Cela peut aussi être entre 2 blocs de mémoire. Le microprocesseur n'intervient donc pas nécessairement, et le transfert des données est ainsi accéléré. Le but principal du DMA est de prendre en charge les gros transferts de données libérant ainsi la charge du processeur par un canal détourné. Le processeur ainsi libéré des tâches de transfert peut se consacrer à d'autres tâches et on peut donc gagner en capacité et rapidité de calcul.

L'I2S :

Le bus I2S est un bus standard pour le transfert de données entre matériels audio numériques. Il est composé d'un signal d'horloge 'bit', d'un signal d'horloge 'word' (pour indiquer le début et la fin du mot) ainsi qu'un bus de données (SD : Serial Data) contenant le mot. On utilise un bus I2S entre le processeur et le codec pour transmettre les données. Le codec qui est esclave envoie par le bus I2S les données numériques qu'il vient de traduire au processeur, qui est maître, pour qu'ensuite le processeur lui renvoie ces données avec les effets voulus appliqués.

le SPI :

Une transmission SPI est une communication simultanée entre un maître et un esclave. Le maître génère l'horloge et donne des requêtes tandis que l'esclave répond aux requêtes du maître à la fréquence de l'horloge. Les avantages de ce bus sont que le bus SPI possède un débit plus important que le bus I2C et aussi que les esclaves utilisent l'horloge du maître et n'ont donc pas besoin d'oscillateur propre. Ici, notre processeur va envoyer des requêtes à notre codec afin de paramétriser les différents convertisseurs du codec.

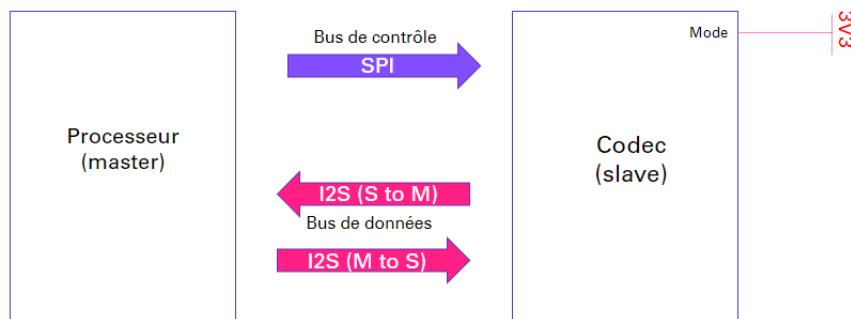


FIGURE 5.9 – Mode de communication entre le processeur et le codec

#### 5.4.4 Consommation énergétique et coût

Afin de donner une idée relativement précise du coût financier pour réaliser le support matériel, nous avons réalisé dans un premier temps un calcul du coût de réalisation du circuit imprimé et de ses composants. Ce calcul est arrondi à l'inférieur et les prix des composants et du frais de port sont ceux de la société Farnell. Pour le prix de réalisation du PCB, il s'agit d'une société chinoise ce qui justifie le faible coût. Nous obtenons donc le calcul suivant :

La réalisation du circuit imprimé revient donc à environ 80 euros.

Il faut ensuite prendre en compte le prix de la réalisation de l'ensemble du boîtier de la pédale contenant le circuit imprimé. Ce dernier peut être réalisé dans un fablab à l'aide d'une imprimante 3D afin de réduire au maximum les coûts. La difficulté sera d'arriver à fabriquer des pédales à pression suffisamment robustes pour des appuis avec le pied. Il sera sinon possible de faire ce boîtier avec des sociétés qui fabriquent des pédales à effet sur mesure mais le coût risque d'être relativement élevé.

	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 – Prix des composants nécessaires à la réalisation du circuit imprimé

# Conclusion

Bien que ce projet s'éloigne de ce que nous imaginions initialement, nos recherches sont concluantes et se rapprochent d'une certaine manière d'un produit que l'on pourrait délivrer.

Bien évidemment, il reste des pistes à approfondir et des points à améliorer tant sur la partie matérielle que sur la partie logicielle.

En effet, l'architecture logicielle ne permet toujours pas de faire du traitement en temps réel, comme nous le souhaitions initialement. Il faudra sûrement trouver une bibliothèque ou des API écrites en langage C ou C++ qui permettent la gestion du matériel bas niveau. Cela pourrait aussi mener à des nouvelles stratégies concernant les phases d'apprentissages, que ce soit pour le réseau neuronal convolutif, ou le VAE.

Pour la partie matérielle, une large batterie de tests et de développement est encore nécessaire. Tous les composants ne sont pas encore intégrés ou reliés par le logiciel au microprocesseur. Les bases fondamentales logicielles ne sont d'ailleurs toujours pas implémenté, le contrôle du codec audio n'est toujours pas fait, et l'intégration des briques logicielles dédiée et optimisée au traitement audio n'ont pas été implémentées. Ainsi, si le projet doit continuer, ces points primordiaux doivent être résolus.

Si vous souhaitez continuer le projet ou simplement l'essayer chez vous, nous vous conseillons fortement de récupérer les ressources sur le GitHub. La plupart des codes tournent sur Google Colab, ce qui signifie qu'ils tournent sur un serveur et une session distante. Cela permet donc de pouvoir jeter un coup d'oeil et d'avoir des résultats quelque soit la configuration dont vous disposez, mais cela assure une consistance dans les résultats.

Néanmoins, si vous souhaitez travailler en local, il sera impossible d'avancer sans installer les différentes bibliothèques. Cela peut prendre énormément de temps. Il est donc conseiller de bien analyser les différentes documentations.

Enfin, nous tenons à remercier notre encadrant, le docteur Sylvain "Syd" Reynal qui nous a accompagné tout au long de cette année, et qui nous a permis d'avancer dans les meilleures, et d'aller au bout de nos pensées.

Nous tenons aussi à remercier les intervenants de l'option "Musique et traitement du signal" de l'ENSEA monsieur Romain Hannequin et Thomas Hézard. Grâce à leur conseil et leurs enseignements, nous avons pu avancer dans notre projet, et produire les résultats que nous vous présentons.

Nous remercions aussi les professeurs de l'ENSEA messieurs Christophe Barès et Nicolas Papazoglou qui nous ont fourni des ressources matérielles et logicielles, mais aussi leurs connaissances sur l'entrepreneuriat et la gestion de projet. Pour finir, nous remercions chaleureusement tous nos camarades de l'ENSEA, quelle que soit leur promotion. Les contacts et les échanges aux différentes pauses, ainsi que les différents avis nous ont aidé à créer notre propre idée de notre projet.

# Bibliographie

- [1] Y. LECUN et al. « Backpropagation Applied to Handwritten Zip Code Recognition ». In : *Neural Computation* 1.4 (déc. 1989), p. 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 ». Thèse de doct. Nov. 2013.
- [3] Stefano D'ANGELO. « Virtual Analog Modeling of Nonlinear Musical Circuits ». Thèse de doct. Nov. 2014.
- [4] Sandro SKANSI. *Introduction to Deep Learning*. Springer International Publishing, 2018.
- [5] Marco A. MARTÍNEZ RAMÍREZ et 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, p. 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](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](http://www.alexandre-boyer.fr/alex/enseignement/Boyer-regles_CEM_PCB_v3.pdf).
- [11] Alexis COOK. *Kaggle learn*. URL : <https://www.kaggle.com/learn>.