Rapport de stage
Génération d'effets par Deep Learning
SCRIME, Université de Bordeaux

Gabriel Weil

Avril 2020 à Août 2020

Remerciements

Je tiens à remercier chaleureusement mes encadrants qui m'ont accompagné et m'ont conseillé tout au long de ce stage,

Myriam Desaintes-Catherine, Pierrick Legrand et Pierre Hanna.

Je tiens à remercier Annick Mersier qui m'a très bien accueilli au laboratoire, et Thibaud Keller pour m'avoir aidé à faire les enregistrements au studio du SCRIME.

Enfin, je tiens à remercier ma compagne ainsi que mes parents pour m'avoir aidé lors de la relecture de ce rapport.

Table des matières

1 Introduction

Dans le cadre du Master Informatique, spécialité Image et Son de l'Université de Bordeaux, j'ai été amené à intégrer le milieu professionnel de la recherche liée à l'informatique musicale au cours d'un stage qui s'est déroulé du 30 mars 2020 au 28 août 2020. J'ai été accueilli au sein de l'équipe de recherche du SCRIME (Studio de Création et de Recherche en Informatique et Musiques Expérimentales) de l'Université de Bordeaux. J'ai été encadré et conseillé pendant mon stage par Mme Desainte-Catherine, directrice du SCRIME et enseignante-chercheuse au Laboratoire Bordelais de Recherche en Informatique, M. Hanna, enseignant-chercheur au Laboratoire Bordelais de Recherche en Informatique, et M. Legrand, enseignant-chercheur à l'Institut de Mathématiques de Bordeaux.

Le sujet du stage est le fruit d'une réflexion avec monsieur Hanna autour des thématiques très étudiées dans l'informatique musicale ainsi que de mes centres d'intérêt. Ainsi mon choix s'est porté sur la synthèse et la transformation de sons numériques à l'aide de réseaux de neurones profonds. Ce champ d'étude ne m'est pas inconnu car j'ai commencé à m'y intéresser dès le début de la deuxième année du Master au travers de l'UE obligatoire *Lecture d'articles et documentation scientifique* où j'ai étudié l'article « Real-Time Black-Box Modelling with Recurrent Neural Network » d'Alec Wright [1], puis au cours de l'UE obligatoire *Projet de Fin d'Études* où nous avons implémenté avec d'autres étudiants du Master le réseau présenté dans cet article.

2 Contexte du stage

En raison de la pandémie de Covid-19 et du confinement imposé à la population, mon stage a débuté en télétravail et s'est poursuivi de la sorte jusqu'à la fin comme le préconisait M. Tunon de Lara, Président de l'Université de Bordeaux. Cela n'a pas empêché le bon déroulement du stage car un serveur Slack a été mis en place pour échanger rapidement et continuellement avec mes encadrants ainsi qu'un serveur BigBlueButton pour faire des audio-conférences hebdomadaires afin de faire un bilan sur le travail fait et sur les améliorations et pistes à explorer. Au delà de ces outils pour communiquer, aucune restriction ne m'a été donnée par mes encadrants sur la méthode de travail ou sur les outils de travail.

Je dispose d'un ordinateur fixe avec un dual boot elementary OS [1] (distribution Linux basée sur la dernière version 18.04 d'Ubuntu) et Windows 10. Ses caractéristiques principales sont un processeur Intel Core i5-6400 (génération Skylake), 8Go de mémoire vive, une carte graphique Nvidia GTX960 avec 4Go de mémoire. J'ai travaillé en majorité sous Linux avec des outils que j'utilisais déjà pendant le Master, à savoir :

- L'environnement de développement VSCodium [2], une version compilée de Visual Studio Code sans les trackers et la télémétrie de Microsoft.
- TensorFlow [3] (version 2) et Python3 pour la construction et l'entraînement des réseaux de neurones que j'ai développé pendant le stage. J'ai utilisé les versions 2.1, 2.2 et 2.3 de TensorFlow au cours du stage.
- JUCE [4], un framework développé par l'entreprise ROLI qui permet de faire des applications audio en C++ de manière simple et rapide.
- GitLab comme dépôt git du code source de mon stage [5].

En plus de ces outils, j'ai eu accès à un serveur GPU administré par Boris Mansencal, ce qui m'a permis d'entraîner les réseaux que j'ai développés pendant mon stage. J'ai aussi utilisé les logiciels Live d'Ableton [6] sous Win-

1. https://elementary.io/, last accessed : 28-08-2020
2. https://vscodium.com/, last accessed : 28-08-2020
3. https://www.tensorflow.org/, last accessed : 28-08-2020
4. https://juce.com/, last accessed : 28-08-2020
5. https://gitlab.com/gweil/black-box-emulation-of-non-linear-audio-effects/
6. https://www.ableton.com/, last accessed : 28-08-2020

dows et Audacity [7] sous Linux pour l'enregistrement et la mise en forme des données d'entraînement.

Je me suis aussi servi de mon espace web du CREMI pour stocker et partager mes résultats [8].

3 Travail de recherche effectué

3.1 Recherche bibliographique

Mon stage a débuté par une partie de recherche bibliographique où je devais faire un état de l'art de la recherche dans ce domaine. Pour ce faire, j'ai utilisé plusieurs outils comme Scinapse [9], Zotero [10], Sci-Hub [11] ou encore Google Scholar [12].

J'ai commencé par lire « Deep Learning Techniques for Music Generation – A Survey » de Jean-Pierre Briot [2], directeur de recherche au CNRS (Centre National de la Recherche Scientifique), qui a animé un séminaire [13] sur l'apprentissage profond et la génération de musique, le 16 Janvier 2020 à l'Université de Bordeaux. Cette étude vise à montrer les différentes utilisations de l'intelligence artificielle dans la génération de contenu musical. À partir de cette étude et des références données, j'ai lu d'autres articles de recherche qui me semblaient pertinents ou en rapport direct avec mon sujet d'étude, et qui pourraient fournir une bonne première voie d'exploration. J'ai lu par exemple [3], qui propose un réseau de neurones CAN (Creative Adversarial Networks) dont le but est d'apprendre un style de peinture de manière similaire à un GAN (Generative Adversarial Networks). Cependant, contrairement à un GAN, son objectif est de s'éloigner du style de départ pour satisfaire l'appréciation humaine et créer un sentiment de nouveauté,

7. https://www.audacityteam.org/, last accessed : 28-08-2020
8. https://gabriel-weil.emi.u-bordeaux.fr/, last accessed : 28-08-2020
9. https://scinapse.io/, last accessed : 28-08-2020
10. https://www.zotero.org/, last accessed : 28-08-2020
11. https://www.sci-hub.tw/, last accessed : 28-08-2020
12. https://scholar.google.fr/, last accessed : 28-08-2020
13. https://scrime.u-bordeaux.fr/Evenements/Saison-2019-2020/Article/Seminaire-Jean-Pierre-Briot, last accessed : 28-08-2020

mais aussi de rester assez proche du style de départ pour ne pas créer un sentiment de rejet et des réactions négatives de la part du spectateur.

Aussi, afin de voir quelles étaient les dernières études traitant de l'émulation d'effets audio par réseaux de neurones profonds, j'ai parcouru la liste des articles publiés lors de la dernière édition de la conférence Digital Audio Effects (DAFx-2019). Deux articles s'y intéressent à travers l'émulation d'effets audio non linéaires :

- *Real-Time Black-Box Modelling with Recurrent Neural Network*, Alec Wright [1]
- *Deep Learning for Black-Box Modeling of Audio Effects*, Marco A. Martínez Ramírez [4]

Les réseaux de neurones présentés dans les deux articles visent à émuler un canal de distorsion d'un ampli à lampe ou un pré-amplificateur. Dans les deux cas, il s'agit d'émuler un effet audio non-linéaire, ce qui signifie que l'effet appliqué sur un signal variera en fonction de la fréquence. Un exemple simple d'effet non-linéaire est un effet d'égalisation (Figure 1), car la modulation faite sur un son ne va pas être la même en fonction des fréquences qui composent ce son.

L'effet que je cherche à émuler est une distorsion d'amplificateur pour guitare électrique, qui est aussi un effet non-linéaire. Parmi ces deux papiers, mon point de départ fut le modèle CRAFx présenté dans [4] et que je présente dans la Section 3.4.

3.2 Effets reproduits

J'ai eu l'occasion d'implémenter le réseau présenté dans [1] au cours du Master, dans le cadre du PFE. De ce fait, j'avais déjà enregistré un ensemble de données pour l'entraînement du réseau. Dans un premier temps, j'ai repris les données que j'avais enregistrées avec un Blackstar HT-1 (Figure 2) pour l'entraînement des réseaux développés pendant le stage.

Par la suite, le laboratoire s'est procuré une loadbox Torpedo Reload (Figure 3), ce qui nous a permis de faire un ensemble de données d'entraînement de meilleure qualité avec une tête d'ampli Mesa Boogie Rectoverb 50

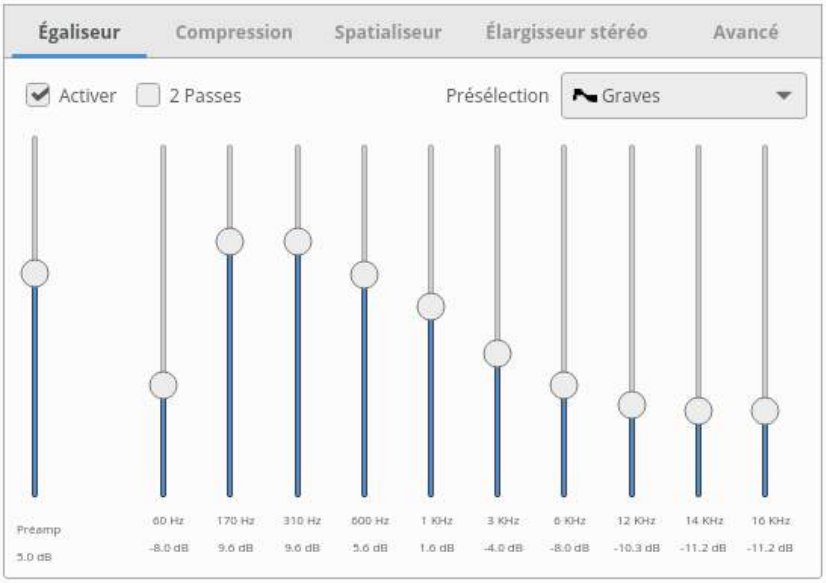


FIGURE 1 – Capture d'écran du panneau d'égalisation du logiciel libre VLC. Ici on voit clairement que les fréquences basses d'un signal audio seront plus audibles que les fréquences moyennes et hautes.

(Figure 4). Je présente plus en détail les méthodes d'acquisition des données qui m'ont servi pour l'entraînement des réseaux implémentés dans la Section 3.3.

Je voulais vérifier une intuition à l'aide de ces deux ensembles de données, en rapport avec les composants d'un amplificateur à lampe. Dans un ampli, l'effet de distorsion est produit à l'aide de composants électriques non-linéaires (transistors, lampes, ...). L'emploi de deux ensembles de données représentatifs de deux amplis de complexités différentes, m'a permis de vérifier si un réseau de neurones capable d'émuler fidèlement un ampli avec peu de composants non-linéaires était aussi capable d'émuler un ampli avec beaucoup plus de composants non-linéaires. En effet, le Blackstar HT-1 est un petit ampli 1W qui possède deux lampes, une pour la pré-amplification, et une autre pour l'amplification (Figure 5), alors que le Mesa Boogie Rectoverb 50 est un ampli à lampes de 50W qui possède 7 lampes, cinq pour la pré-amplification et deux pour l'amplification. En effet, si un ampli avec de nombreux composants non-linéaires requiert l'emploi d'un réseau de neu-

FIGURE 2 – Tête d'ampli à lampe Blackstar HT-1, première génération, 1 Watt

rones plus lourd, cela aura alors des impacts sur l'utilisation en temps réel et sur la puissance de calcul nécessaire. Je présente les résultats et les analyses dans la Section 3.8.

3.3 Enregistrement des échantillons audio

Afin de réaliser les entraînements de mon réseau de neurones, j'ai utilisé deux ensembles de données. Tous les enregistrements ont été réalisés à une fréquence d'échantillonnage de 44100Hz et encodés en 16 bits.

Le premier ensemble de données consiste en des enregistrements d'échantillons audio de guitare issus de la base de données du Fraunhofer Institute for Digital Media Technology [14]. Cette base de données est utilisée dans [1]. Je n'ai pas utilisé toute la base de données car elle possède plusieurs dossiers ayant chacun un objectif particulier. J'ai utilisé le Dataset 3 ainsi que le Dataset 4 dans lequel j'ai sélectionné les enregistrements blues-rock réalisés avec une Ibanez 2820. Cela permet d'obtenir à la fois des notes et des accords, joués lentement et rapidement. L'ensemble des fichiers audio utilisés forme une base de données de 7 minutes et 48 secondes. Une fois les enregistre-

14. https://www.idmt.fraunhofer.de/en/business_units/m2d/smt/guitar.html, last accessed : 28-08-2020

FIGURE 3 – Loadbox Torpedo Reload, vue de face en haut, vue de derrière en dessous.

FIGURE 4 – Tête d'ampli guitare à lampe Mesa Boogie Rectoverb 50, puissance 50W.

FIGURE 5 – Circuit interne du Blackstar HT-1. Les deux lampes sont entourées en rouge.

ments sélectionnés, je les ai passés en signal d'entrée dans une tête d'ampli Blackstar HT-1 (sortie de la carte son vers entrée de l'amplificateur), puis j'ai récupéré le signal de sortie à l'aide de la sortie ligne de l'amplificateur. Il est important de noter que le signal obtenu en sortie n'est pas exactement le signal de la tête d'ampli. La sortie que j'utilise pour récupérer le signal de l'ampli est une sortie qui émule le son de haut-parleurs. L'enregistrement s'est fait à l'aide du logiciel Live d'Ableton et d'une carte son Focusrite 2i2 deuxième génération (voir Figure 6). À noter, le fait de brancher une sortie ligne qui a une impédance d'environ 10kΩ, dans une entrée instrument qui a une impédance de 1MΩ, ne pose aucun soucis. Le niveau de sortie ligne a été ajusté pour qu'il ne soit pas trop important pour le niveau attendu par l'ampli.

Le second ensemble de données s'est fait au mois de juillet, au studio d'enregistrement du SCRIME. L'enregistrement s'est fait dans une configuration différente que celle utilisée pour le Blackstar HT-1, car le laboratoire a acquis entre temps une loadbox Torpedo Reload (Figure 3), qui permet de

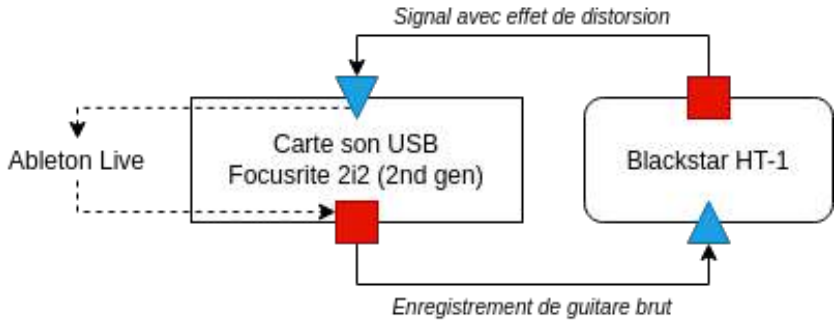


FIGURE 6 – Schéma du branchement du matériel pour l'enregistrement de la tête d'ampli Blackstar HT-1. Les triangles bleus correspondent aux entrées des appareils utilisés et les carrés rouges aux sorties. La sortie ligne émulée a été utilisée sur le Blackstar HT-1.

réaliser des enregistrements d'amplis à lampe ne possédant pas de sortie ligne (ce qui correspond à la grande majorité d'entre eux) et sans avoir à utiliser de microphone. L'emploi d'une loadbox rend le cheminement du signal pour enregistrer le Mesa Boogie Rectoverb 50 plus complexe :

1. La guitare est branchée dans l'entrée de la loadbox.
2. Le son de la guitare sort de la loadbox puis entre dans une table de mixage, ce qui permet d'enregistrer le signal sans modification (= la vérité terrain). Cette étape est entourée en vert dans la Figure 7.
3. Le son de guitare revient dans la loadbox pour ressortir vers l'entrée de l'amplificateur,
4. Le son de l'amplificateur est ensuite récupéré avec la sortie haut-parleur dans la loadbox,
5. Finalement, on récupère le signal modifié avec la sortie ligne de la loadbox. Cette étape est entourée en rose dans la Figure 7.

J'ai aussi réutilisé les enregistrements que j'avais utilisés avec le Blackstar HT-1 pour le premier ensemble de données. Le montage reste similaire, il suffit de remplacer le signal de la guitare au niveau du cercle vert par les enregistrements que l'on souhaite faire passer dans l'amplificateur, la suite ne changeant pas.

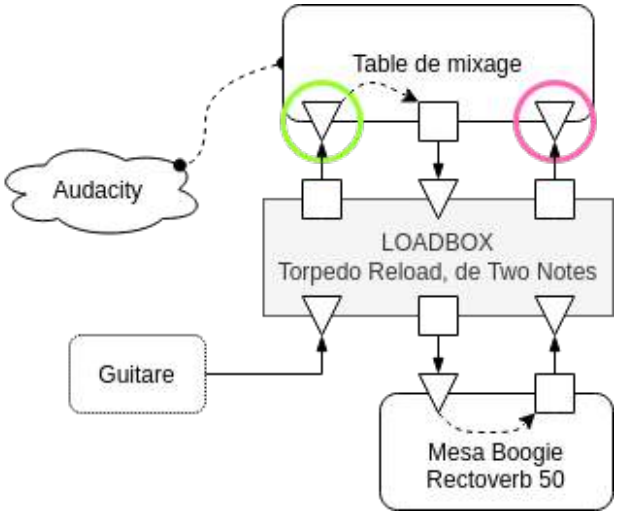


FIGURE 7 – Schéma du branchement du matériel pour l'enregistrement de la tête d'ampli Mesa Boogie Rectoverb 50. Les triangles correspondent aux entrées des appareils utilisés et les carrés aux sorties. Le signal au niveau du cercle vert correspond au son de la guitare. Le signal au niveau du cercle rose correspond au son de la guitare avec l'effet de distorsion de l'amplificateur.

3.4 Réseau de neurones implémenté

Les articles de recherche [1] et [4] présentent tous les deux des réseaux de neurones récurrents. Ces modèles sont particulièrement appréciés lors de traitement de séquences temporelles où l'information à un temps $t-1$ a un impact sur la prédiction au temps t. Comme nous travaillons sur des séquences audio avec un nombre d'échantillons fixe, nous risquons de ne pas prendre en compte des informations qui ne tiendraient pas dans cette fenêtre de taille fixe. Cette information correspond en général aux fréquences basses dont la période est grande.

J'ai implémenté le réseau *Convolutional Recurrent Audio Effects Modeling Network* (CRAFx), proposé par Martínez Ramírez dans [4]. J'ai codé le réseau en Python en utilisant la bibliothèque open source TensorFlow dans ses versions stables les plus à jour au moment où j'ai implémenté le réseau (v2.1, v2.2 et v2.3). Les entraînements ont été faits sur une carte graphique Nvidia Titan X qui repose sur une architecture Pascal et avec 12Go de mémoire. L'implémentation a été longue car l'auteur emploie beaucoup de mé-

thodes d'apprentissage profond qui n'avaient pas été abordées pendant ma formation comme la déconvolution de signaux 1D, la méthode *Squeeze-and-Excitation* introduite par Jie Hu dans [5] ou encore la fonction d'activation *Smooth Adaptive Activation Function* présentée par Le Hou dans [6].

À ces concepts nouveaux s'est ajoutée une difficulté technique pour implémenter le réseau. Dans le cas de la déconvolution, TensorFlow ne proposait pas de méthode capable de faire une déconvolution sur un signal en 1D avant la version 2.3. Je parle de ce problème plus en détail dans la Section 3.7.3 et de la solution que j'ai trouvée pour réaliser cette déconvolution avant la version 2.3.

L'auteur choisit de travailler sur des fenêtre de 4096 échantillons (chaque échantillon correspondant à une valeur comprises entre -1 et 1), en prenant en considération le contexte audio. Ainsi, pour une fenêtre de 4096 échantillons, le réseau recevra aussi 4 fenêtres audio précédant la fenêtre présente, ainsi que 4 fenêtres futures avec un espacement de 2048 échantillons entre chaque fenêtre (Figure 8). Nous passons au réseau des données sous la forme {9, 4096, 1}.

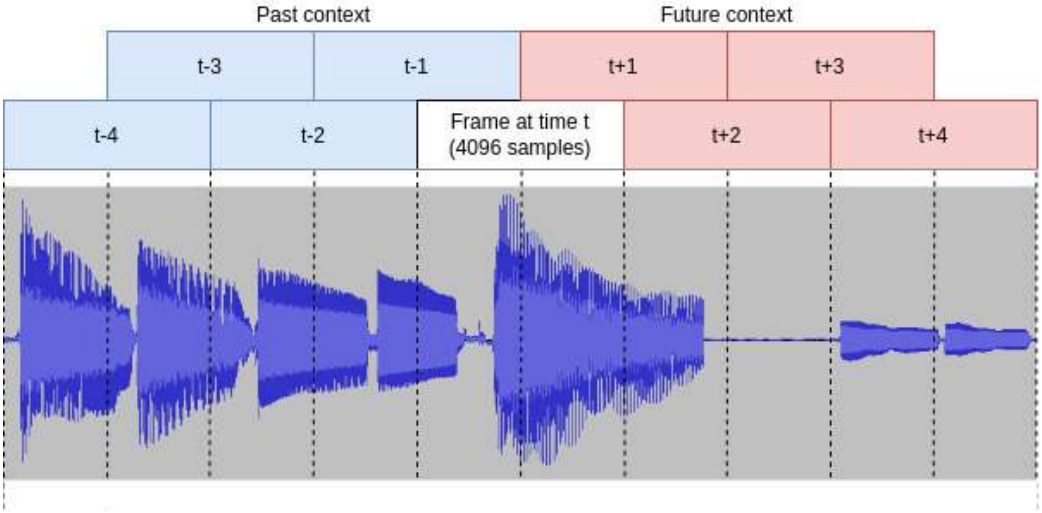


FIGURE 8 – Représentation graphique des données passées en entrée du réseau de neurones. *Le signal audio représenté sous les blocs n'est pas à l'échelle.*

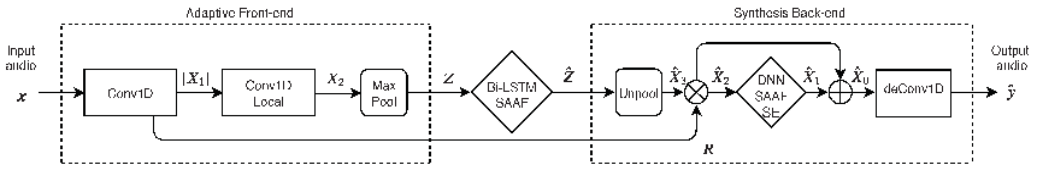


FIGURE 9 – Architecture du réseau de neurones CRAFx. Schéma repris depuis l'article d'origine.

Nous pouvons distinguer 3 groupes distincts dans l'architecture du modèle CRAFx (Figure 9) :

- Adaptive front-end qui vise à décomposer le signal en bandes de fréquences à l'aide de convolutions et à réduire la taille du signal passé en entrée.
- Latent-space qui cherche à reproduire l'effet de distorsion sur les échantillons qui lui parviennent depuis les couches précédentes.
- Synthesis back-end qui reconstruit le signal à partir des échantillons issus des couches latentes et des bandes de fréquences non-réduites obtenues au début du réseau.

La partie **Adaptive front-end** est composée de deux convolutions (Conv-1D et Conv1D Local), une couche de pooling (Max Pool) et une connexion résiduelle (R), qui servira dans la dernière partie, **Synthesis back-end**, pour reconstruire le signal.

La première couche de convolution utilise comme fonction d'activation non-linéaire la valeur absolue. Elle a pour but de décomposer le signal en bandes de fréquences à l'aide de convolutions, de manière similaire à [7] ou [8]. Après cette première couche de convolution, nous passons d'un signal en une dimension à un signal en n dimensions, que nous pouvons interpréter comme n signaux différents. Dans le réseau proposé, l'auteur choisit de découper le signal en $n = 32$ bandes.

La deuxième couche de convolution est localement connectée : chaque filtre de convolution est propre à une dimension du signal (n filtres de convolutions différents). Il existe une couche dans TensorFlow appelée *LocallyConnected1D* cependant elle ne correspond pas à la même chose. Dans l'implémentation de TensorFlow, une couche de convolution localement connectée est une couche

où tous les filtres sont indépendants les uns des autres, la Figure 10 représente ce principe. Il s'ensuit une couche de pooling où l'on garde la valeur maximale parmi 64 valeurs (Max Pool), ce qui nous donnera 64 valeurs par fenêtre en sortie de la première partie. Comme nous ne pouvons pas passer les données sous la forme {9, 64, 32} aux couches récurrentes de la partie suivante, nous devons d'abord les mettre bout à bout, ce qui fait que les données sont de la forme {64, 288} à la fin de la partie Adaptive front-end. Ces couches ont pour objectif de permettre au réseau d'apprendre une représentation proche de l'enveloppe du signal qu'on lui passe en entrée.

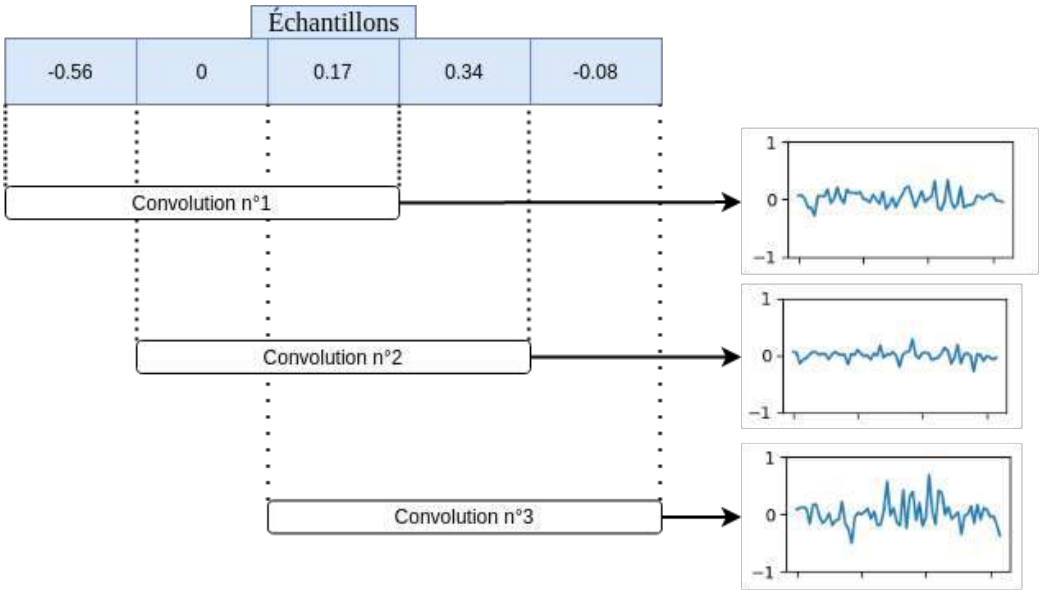


FIGURE 10 – Fonctionnement de la couche LocallyConnected1D (TensorFlow). Dans cet exemple, les filtres sont de taille 3, leur représentation n'est pas à l'échelle, elle sert uniquement à montrer que chaque convolution est différente de ses voisines.

La partie **Latent space**, que j'appellerai espace latent par la suite, est composée de trois couches bidirectionnelles LSTM avec 64, 32, et 16 unités respectivement. Les couches LSTM (Long Short-Term Memory) sont des couches récurrentes. Elles possèdent un état interne qui évolue en prenant en considération un état passé, cela les rend très performantes lors du traitement de séquences, temporelles ou non, où l'information passée a un impact sur l'information présente (texte, audio, etc.). Le fait que ces couches soient

bidirectionnelles fait qu'elles sont capables de lire l'information vers l'avant (du début d'une séquence à la fin) et vers l'arrière (de la fin d'une séquence au début). Le rôle de ces couches est de réduire le nombre de dimensions du signal $\mathbf{Z}$ et d'apprendre l'effet $\hat{\mathbf{Z}}$ (Figure 9) que l'on cherche à émuler, dans notre cas, la distorsion.
Les deux premières couches bi-LSTM utilisent la fonction *tanh* comme fonction d'activation non-linéaire et la dernière couche bi-LSTM utilise la fonction *SAAF* (Smooth Adaptive Activation Function), présentée pour la première fois dans [6]. Une SAAF est une fonction polynomiale du second degré qui peut approximer n'importe quelle fonction continue et qui est soumise à une constante de Lipschitz pour en assurer la régularité. Les paramètres de cette fonction sont appris au cours de l'entraînement par le réseau. Les données en sortie de l'espace latent sont de la forme {64, 32}.

La dernière partie du réseau est la partie **Synthesis back-end**. Elle sert à construire le signal transformé à l'aide de la sortie de l'espace latent et de la connexion résiduelle de la partie Adaptive front-end. Elle est composée d'une couche *Unpool* qui va passer les données de la forme {64, 32} vers la forme {4096, 32} à l'aide de la couche UpSampling1D (Figure 11). S'ensuit une couche de multiplication entre la connexion résiduelle et le signal latent augmenté, puis un bloc DNN-SAAF-SE, suivi d'une couche d'addition entre le résultat de la multiplication et le signal sorti du bloc DNN-SAAF-SE et enfin une couche de déconvolution qui fait l'exact opposé de la convolution en entrée du réseau avec le rassemblement des bandes de fréquences entre elles pour obtenir le signal final.

Le bloc DNN-SAAF-SE est une architecture de réseau de neurones profond où se succèdent des couches de convolution (DNN) dont la dernière aura pour fonction d'activation une fonction SAAF (DNN-SAAF). Ces couches sont suivies d'un bloc *Squeeze-and-Excitation* (DNN-SAAF-SE) introduit pour la première fois dans [5]. L'auteur reprend l'architecture du bloc SE présentée dans [9]. Il est composé d'une moyenne globale de chaque bande de fréquence suivie de deux couches densément connectées dont les fonctions d'activation sont respectivement ReLU et sigmoid. Une couche *valeur absolue* est placée avant le calcul de la moyenne comme nous travaillons avec un

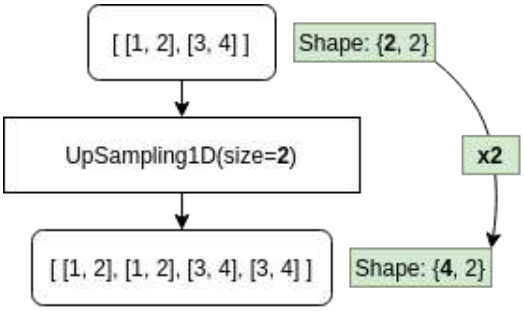


FIGURE 11 – Les données sont copiées de manière : $AB \rightarrow AABB$, où A et B représentent une donnée de dimension quelconque.

signal dans le domaine temporel (par opposition à un signal dans le domaine spectral, que l'on peut obtenir avec une transformée de Fourier).

3.5 Résultats obtenus

J'ai entraîné le réseau de neurones en suivant les recommandations données dans l'article [4]. L'apprentissage s'est fait en deux étapes :

- Étape de pré-apprentissage où l'on entraîne les couches des parties **Adaptive front-end** et **Synthesis back-end** uniquement. Les données de départ sont les mêmes que celles que le réseau doit reproduire, le but étant d'entraîner le réseau à décomposer le signal puis à le reconstruire à l'identique.
- Étape d'apprentissage où l'on entraîne toutes les couches en chargeant les poids de l'étape de pré-apprentissage.

Pour les deux étapes, la fonction de perte à minimiser (*loss function*) est la fonction *Mean Absolute Error* (MAE). On utilise l'optimiseur Adam (« L'optimisation Adam est une méthode de descente de gradient stochastique basée sur l'estimation adaptative des moments du premier et du second ordre. » [15]) avec un taux d'apprentissage (*learning rate*) initial fixé à 1×10^{-4}. L'auteur ajoute aussi une condition d'arrêt si la perte calculée sur l'ensemble de validation ne s'est pas améliorée au cours des 25 dernières époques. À la fin de la deuxième étape, l'entraînement du réseau est affiné en le refaisant

15. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam, last accessed : 18-08-2020

une nouvelle fois mais en divisant le taux d'apprentissage initial de l'optimiseur par 4. Je n'ai pas utilisé la même taille de batch que celle proposée par l'auteur car nous n'avons pas les mêmes données. La taille préconisée ne représentait rien dans mon cas de figure (dans l'article d'origine, l'auteur utilise des échantillons audio de deux secondes et prend une taille de batch équivalente à la durée des échantillons). J'ai pris une taille de batch de 124 échantillons, que j'ai augmentée par la suite à 500 car j'ai apporté des modifications au réseau d'origine et que cela m'a permis d'avoir un entraînement sans surapprentissage (« *overfitting* ») et assez général pour que le réseau apprenne correctement l'effet de distorsion (voir Section 3.7.2).

Ce processus doit permettre au réseau d'apprendre à décomposer le signal en bandes de fréquences dans un premier temps puis de s'en servir pour émuler correctement l'effet désiré. J'ai voulu vérifier très rapidement que le réseau découpait bien le signal en bandes de fréquences. Pour ce faire, j'ai affiché les filtres de convolution (Figure 12) et j'ai enregistré des échantillons audio après les avoir passés dans la couche de convolution [16].

Une fois le réseau implémenté et entraîné, j'ai écrit un script Python pour le tester et enregistrer le résultat. Le script charge deux fichiers audio qui correspondent à un extrait audio test ainsi qu'à sa référence. On transforme les données pour les rendre exploitables par le réseau puis on les parcourt à l'aide d'une boucle *for*. Comme nous travaillons sur des fenêtres de taille s et que nous avançons avec un pas de taille $\frac{s}{2}$, nous devons traiter les prédictions faites par le réseau avec une fenêtre de Hann afin de les assembler les unes avec les autres. Seuls le début et la fin du signal complet ne doivent pas être pondérés comme ils ne seront pas chevauchés par une autre fenêtre/prédiction. À part les $\frac{s}{2}$ premiers échantillons et $\frac{s}{2}$ derniers échantillons, tous sont pondérés avec une fenêtre de Hann, puis additionnés avec l'échantillon correspondant dans la fenêtre précédente (Figure 13). On affiche enfin le spectre des différents signaux et on les enregistre dans des fichiers séparés. Le but en faisant des prédictions qui se chevauchent est d'effacer les

16. Les échantillons audio sont disponibles à l'adresse https://gabriel.weil.emi.u-bordeaux.fr/filterbank_results/ sont les signaux récupérés après la première couche de convolution et avant l'application de la fonction d'activation non-linéaire, avec $n = 64$. Nous entendons très clairement que le réseau découpe le signal en bandes de fréquences.

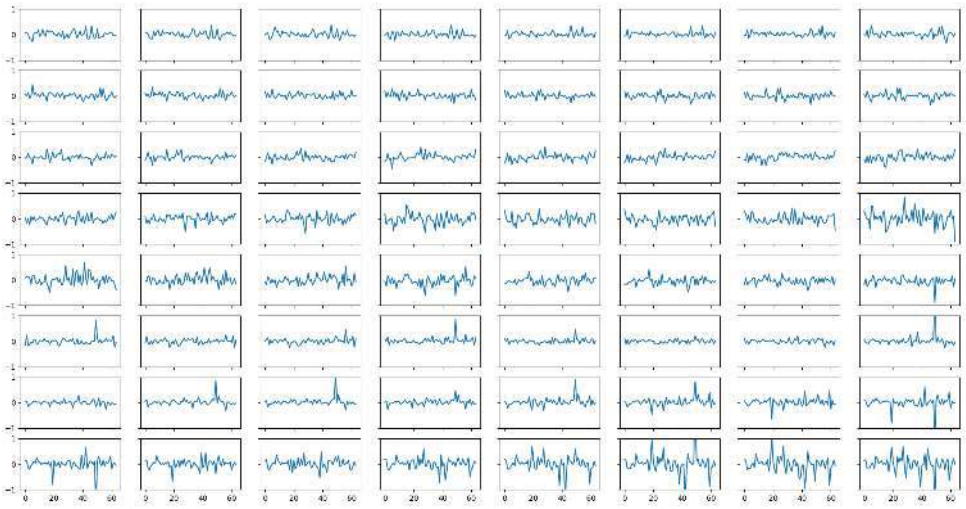


FIGURE 12 – Au cours de mes recherches j'ai augmenté le nombre de filtres à 64, d'où le nombre de filtres dans cette image.

discontinuités audio entre deux fenêtres. Sans cette méthode on entend un clic régulier entre chaque prédiction.

La première implémentation que j'ai faite ne contenait pas la fonction d'activation SAAF. Comme ni mes encadrants ni moi ne connaissions cette fonction d'activation, nous avons convenu de faire un premier test sans la prendre en compte dans le réseau de neurones. Le résultat que j'ai obtenu n'est pas très bon : on entend un bruit aigu constant et le son prédit ne reproduit pas très bien les fréquences les plus basses. De plus on entend clairement que les notes tenues ne sont pas reproduites justement et qu'un effet de modulation est appliqué par dessus à fréquence régulière (Figure 14).

Afin de constater l'impact que peut avoir la fonction d'activation SAAF sur le réseau de neurones, j'ai repris le même code en rajoutant uniquement ces couches d'activation aux endroits où elles manquaient. Pour implémenter la couche d'activation SAAF j'ai repris le code de l'auteur depuis son dépot GitHub [17]. On voit sur la Figure 15 que le résultat est bien meilleur et certains défauts ont été retirés à l'aide de la fonction d'activation SAAF. Toutefois, on arrive encore à distinguer le signal émulé du signal de référence parce

17. https://github.com/mchijmma/DL-AFx, l'auteur a retiré le 18 juin le code source du dépôt, mais il est toujours possible d'y accéder en faisant la demande par mail.

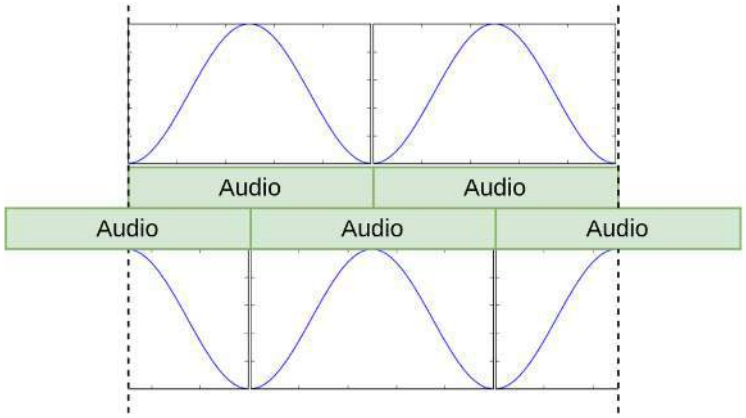


FIGURE 13 – Schéma de l'utilisation de la fenêtre de Hann dans le script de test. Les rectangles verts représentent les prédictions audio successives, avec la fenêtre de Hann qui leur correspond (en haut pour les rectangles du haut, en bas pour les rectangles du bas). Les premiers et derniers échantillons ne sont pas pondérés comme ils ne sont pas chevauchés par d'autres fenêtres.

que le réseau produit un bruit aigu qui n'est pas présent dans le signal de référence et le son émulé manque légèrement de basses par rapport au signal de référence. Or dans son papier, l'auteur montre que des professionnels de la musique, à qui on avait fait faire des tests à l'aveugle, n'arrivaient pas clairement à discerner le son émulé du son de référence.

En relisant l'article j'ai très rapidement compris d'où provenait la différence entre mes résultats et ceux de l'auteur. Les données que j'ai choisies (Section 3.3) cherchent à se rapprocher d'un musicien qui jouerait de la guitare alors que les données choisies par l'auteur pour entraîner son réseau sont issues d'un autre ensemble de données également produit par le Fraunhofer Institute for Digital Media Technology IDMT, *IDMT-SMT-Audio-Effects*[18]. Les extraits audio sont des notes monophoniques jouées à la guitare et à la basse et qui sont tenues 2 secondes seulement. Dans ce contexte, il est normal que le réseau ne donne pas des résultats aussi bons étant donné que mes extraits audio sont des notes jouées rapidement et lentement, mono- et polyphoniques, avec différents styles de jeu , donc des extraits plus « réalistes ».

18. https://www.idmt.fraunhofer.de/en/business_units/m2d/smt/audio_effects.html, last accessed : 19-08-2020

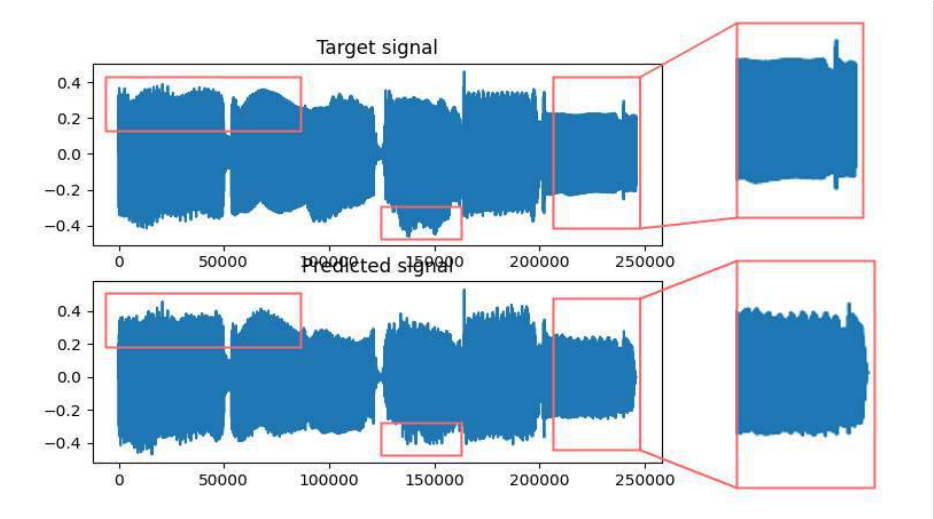


FIGURE 14 – Comparaison entre le signal de référence (en haut) et le signal émulé (en bas). On observe très clairement dans la forme de l'onde que le signal obtenu contient des artefacts qui permettent de le différencier du signal attendu. Les échantillons audio et l'image de comparaison sont disponibles à l'adresse https://gabriel-weil.emi.u-bordeaux.fr/first_result_without_SAAF.

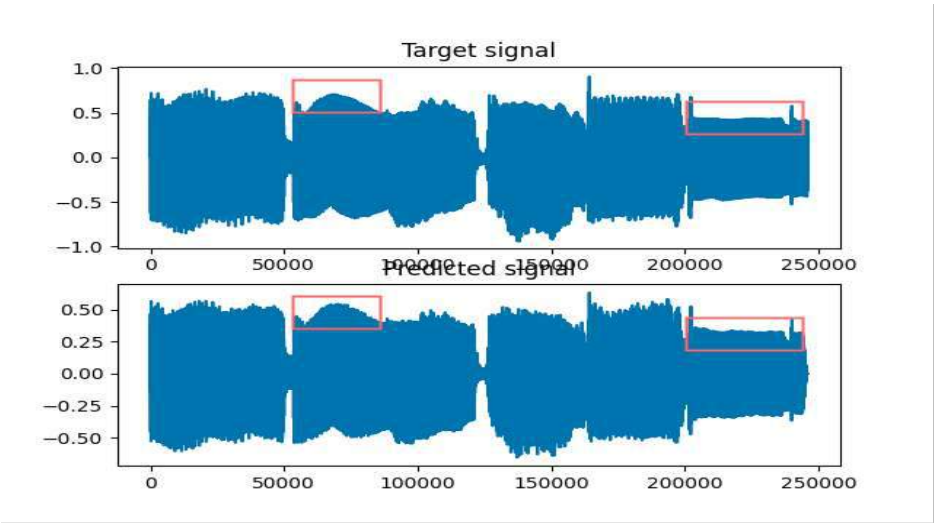


FIGURE 15 – Comparaison entre le signal de référence (en haut) et le signal émulé (en bas). Les modulations présentes sur les notes tenues ont quasiment disparu. De plus, on retrouve moins de pics aberrants et le signal est plus lisse. Les extraits audio et l'image sont disponibles à https://gabriel-weil.emi.u-bordeaux.fr/first_result_with_SAAF/

3.6 Améliorations du réseau de neurones

En me basant sur le réseau de neurones implémenté, l'objectif était de le rendre plus performant avec des données réalistes comme celles dont je dispose.

3.6.1 Error-to-Signal Ratio

La première modification que j'ai faite n'a pas été directement sur le réseau de neurones mais sur la fonction d'évaluation utilisée pour son entraînement. En effet la fonction *Mean Absolute Error* (MAE) présente le défaut de ne pas être représentative de la perception auditive humaine. Dans [1], Alec Wright présente une autre fonction, *Error-to-Signal Ratio with pre-emphasis* (ESR), qui cherche justement à se rapprocher de la perception humaine. Cette fonction est l'erreur au carré divisée par l'énergie du signal de référence :

$$\varepsilon_{ESR} = \frac{\sum_{n=0}^{N-1} |y_p[n] - \hat{y}_p[n]|^2}{\sum_{n=0}^{N-1} |y_p[n]|^2} \tag{1}$$

À l'ESR est ajouté le calcul du décalage DC. Il s'agit du décalage entre le signal de départ (son de guitare) et le signal récupéré après l'enregistrement (son de l'ampli). Si les signaux sont alignés à la main alors ce terme doit être proche de 0. Sa formule est :

$$\varepsilon_{DC} = \frac{\frac{1}{N}\sum_{n=0}^{N-1} |y[n] - \hat{y}[n]|^2}{\frac{1}{N}\sum_{n=0}^{N-1} |y[n]|^2} \tag{2}$$

La fonction de perte est donc l'addition des équations (1) et (2) :

$$\varepsilon = \varepsilon_{ESR} + \varepsilon_{DC} \tag{3}$$

L'intérêt d'appliquer un filtre de pré-emphase sur un signal audio est de permettre au réseau de ne pas traiter de manière équivalente toutes les bandes de fréquences et de minimiser plus rapidement l'erreur sur les plus importantes d'entre elles. Le filtre de pré-emphase utilisé est un passe-haut de premier ordre :

$$H(z) = 1 - 0.85z^{-1} \tag{4}$$

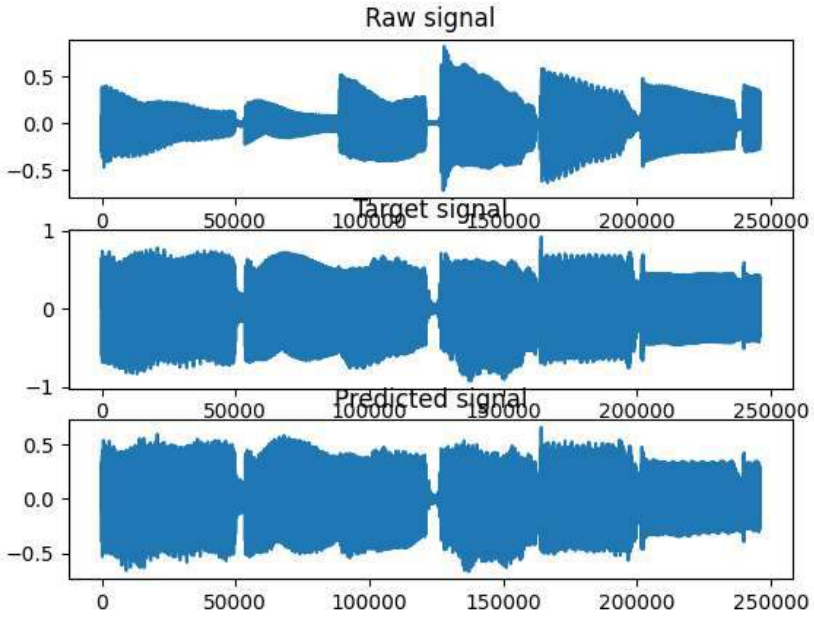


FIGURE 16 – Résultat avec la fonction de perte ESR (*Error-to-Signal Ratio*). Extraits audio et image disponibles à l'adresse https://gabriel-weil.emi.u-bordeaux.fr/esr/

L'entraînement s'est fait avec le même taux d'apprentissage initial, seule la fonction de perte a été modifiée. Le signal obtenu après l'entraînement avec l'ESR comme fonction de perte (Figure 16) est moins lisse que celui obtenu précédemment. On retrouve en effet le même phénomène que dans la Figure 14 où le réseau applique un effet de modulation sur les notes tenues. Le bon point est que l'objectif de départ, à savoir retirer le bruit aigu perceptible, a été atteint avec l'utilisation de l'ESR avec pré-emphase pour l'entraînement du réseau. Toutefois, d'autres modifications sont nécessaires pour obtenir un résultat satisfaisant.

3.6.2 Réduction de la taille des fenêtres et augmentation du nombre de filtres

Un autre paramètre sur lequel j'ai essayé de jouer afin d'obtenir de meilleurs résultats est la taille des fenêtres audio. À l'origine, les fenêtres audio sont composées de 4096 échantillons et le pas est de 2048 échantillons. J'ai choisi de réduire de moitié ces valeurs, en gardant le contexte futur et passé. Nous commencions à ce moment à nous interroger avec mes encadrants sur les voies que je pourrai explorer par la suite, et j'ai choisi celle de l'utilisation du réseau en temps réel. Nous en reparlerons dans la Section 3.7, je donne cette information pour justifier le choix de réduire le nombre d'échantillons par fenêtre plutôt que de l'augmenter. En réduisant le nombre d'échantillons par fenêtre, il y a un risque que nous perdions des informations primordiales pour faire une émulation fidèle. D'un autre côté, la bande de fréquence entre 20Hz et 50Hz n'est pas celle qui s'entend le plus sur un son de guitare électrique étant donné que sur une guitare à 6 cordes, accordée de manière standard (Mi-La-Ré-Sol-Si-Mi), la note la plus grave est un Mi2, dont la fréquence est 82Hz [19], et la note la plus aiguë est un Mi7 dont la fréquence est 2637Hz. Cette observation n'est valable que pour la guitare. Si j'avais utilisé des échantillons de guitare basse alors mes résultats auraient pu être moins bons en diminuant la taille des fenêtres.

Comme nous réduisons la taille des fenêtres, j'ai décidé d'augmenter le nombre de bande de fréquences dans le réseau à 64. Dans l'article d'origine, l'architecture CRAFx est mise face à d'autres réseaux de neurones et évaluée par rapport à eux. Tous ont une architecture similaire, à savoir une partie **Adaptive front-end**, une partie **Latent space** et une partie **Synthesis back-end**. La différence entre les réseaux est la composition de l'espace latent. Dans notre cas il s'agit de couches récurrentes bidirectionnelles mais dans l'architecture CAFx (*Convolutional Audio Effects Modeling Network*) il s'agit d'un sous-réseau de neurones profond composé de couches *Dense*. Contrairement aux couches de convolution, les couches récurrentes sont plus

19. Je précise que cette information est vraie tant que l'on reste sur une guitare électrique « classique ». La note la plus basse sur une guitare électrique à 8 cordes est un Si1 (23Hz).

longues à entraîner donc le nombre de filtres est réduit par rapport aux autres modèles (par exemple 128 filtres pour CAFx, 32 pour CRAFx).

Le résultat obtenu avec une fenêtre de 2048 échantillons et le pas de 1024 échantillons avec 64 bandes de fréquences est très bon. Le bruit aigu a presque complètement disparu et l'effet de distorsion est reproduit très fidèlement par rapport à la référence (Figure 17). Par ailleurs, les fréquences basses ne sont pas manquantes, elles sont clairement présentes et s'intègrent de façon harmonieuse aux autres fréquences, ce qui règle le soucis de leur sous-représentation dans le son émulé.

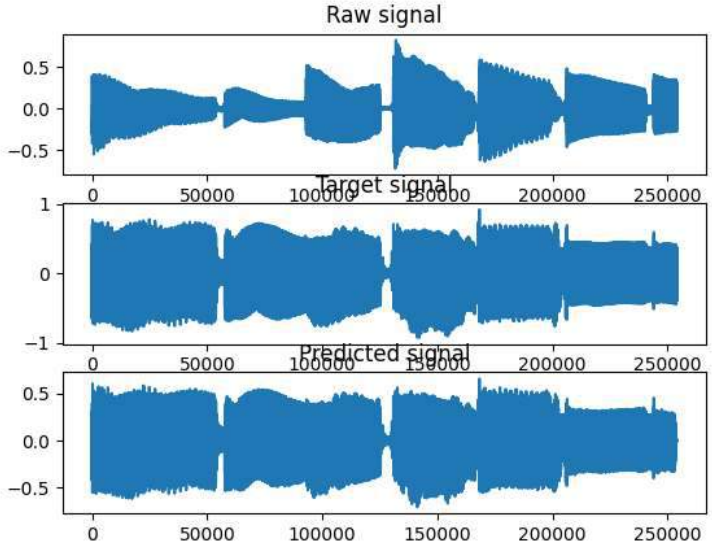


FIGURE 17 – Prédiction du réseau avec des fenêtres de 2048 échantillons, un pas de 1024 échantillons et 64 bandes de fréquences. Données accessibles à l'adresse https://gabriel-weil.emi.u-bordeaux.fr/esr_with_small_frames/

3.6.3 Fonctions d'activation du bloc DNN-SAAF-SE

Une autre modification que j'ai apportée au réseau a été sur la fonction d'activation des couches de convolution du bloc DNN-SAAF-SE de la partie **Synthesis back-end**. Dans le sous-bloc DNN, les 3 premières convolutions ont comme fonction d'activation la fonction softplus dont les valeurs vont

de 0 à 1 (Figure 18), or comme nous travaillons sur un signal audio dont les valeurs vont de -1 à 1, il m'a semblé judicieux de changer la fonction d'activation par la fonction *tanh* (Figure 18). En outre, nous nous étions rendus compte avec des camarades pendant notre cursus que la fonction tanh permet généralement d'obtenir de meilleurs résultats sur des signaux audio que la fonction softplus, c'était donc l'occasion de tester de nouveau cette intuition.

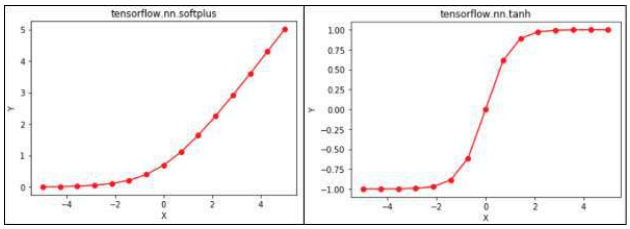


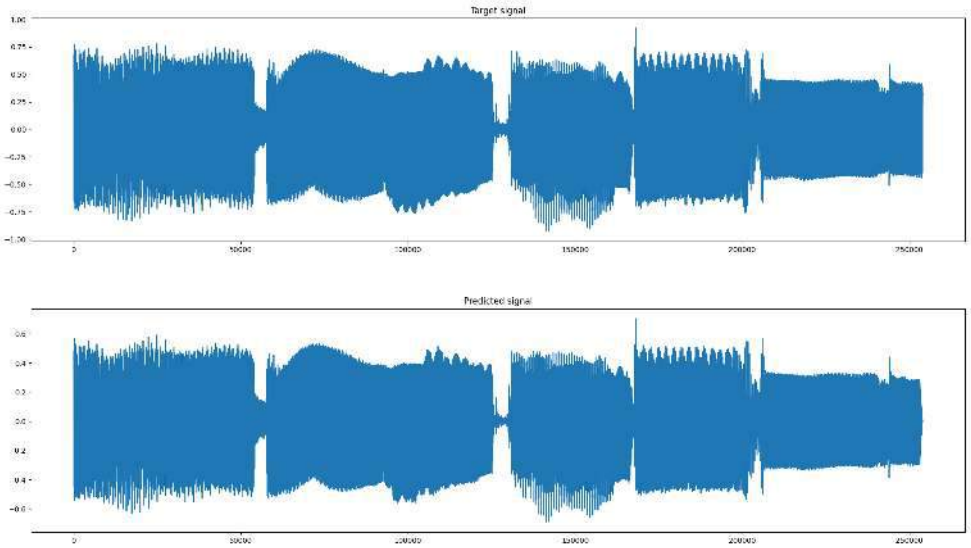
FIGURE 18 – Comparaison des fonctions d'activation softplus et tanh. *Source : https://www.geeksforgeeks.org*

Cette modification a permis de retirer les dernières irrégularités dans le spectre de la Figure 17. Le résultat audio est d'ailleurs le meilleur obtenu jusqu'à présent avec cette architecture et est extrêmement proche du signal de référence. La Figure 19 montre les spectres avec deux extraits audio. Le premier est celui que j'utilisais jusqu'alors (avec des notes tenues) et le deuxième est composé de notes courtes qui ne sont pas tenues.

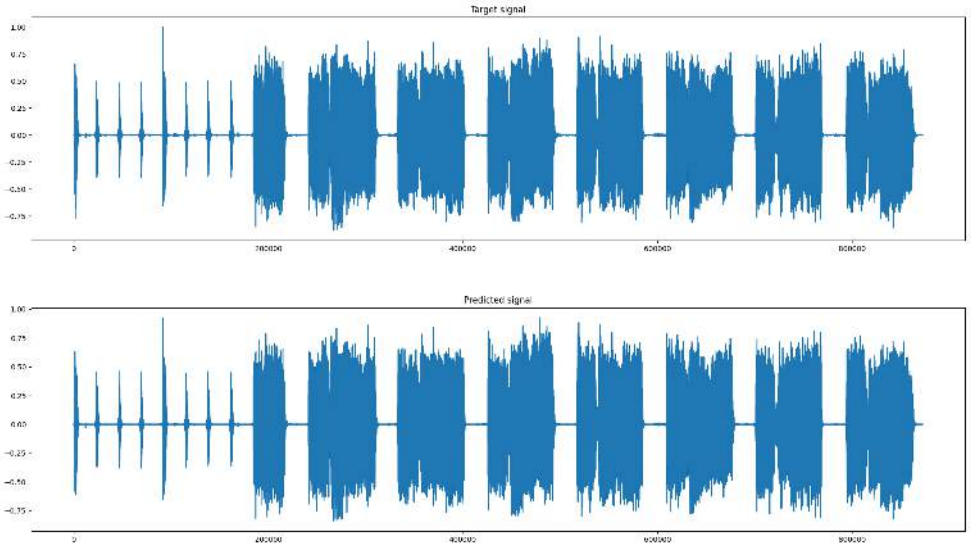
3.7 Notion de temps réel

J'ai abordé la question de l'utilisation en temps réel dès que les résultats que j'ai obtenus avec des améliorations semblaient convenables. Par ailleurs, j'ai pu échanger avec d'autres chercheurs et acteurs de l'industrie à ce propos et voir les limitations liées à l'utilisation de l'intelligence artificielle dans le contexte industriel musical actuellement. En tant que fabriquants d'effets numériques pour guitaristes, l'utilisation de l'intelligence artificielle les intrigue et ils éprouvent un réel intérêt pour les études qui se font dans ce champ de recherche.

Le problème majeur de l'utilisation de réseaux de neurones est la limita-



(a) Comparaison du signal prédit (en bas) et du signal de référence (en haut) avec des notes tenues.



(b) Comparaison du signal prédit (en bas) et du signal de référence (en haut) avec des notes courtes et jouées rapidement.

FIGURE 19 – Spectres des signaux prédits avec la fonction d'activation tanh. Les extraits audio et les images sont disponibles à https://gabriel-weil.emi.u-bordeaux.fr/tanh/

tion dans le temps de calcul due à la notion de temps réel. La notion de temps réel varie selon l'objet d'étude. Par exemple, un échange audio téléphonique est considéré tolérable si la latence reste en dessous des 200ms, la latence maximale pour un joueur professionnel de jeux vidéos devant être autour de 50ms, etc.

Dans notre cas, à savoir l'utilisation d'un effet audio par un musicien, le temps de latence maximum pour considérer l'effet comme utilisable en temps réel est nettement inférieur. Selon l'article [10] de Al Keltz, la latence adéquate se situe entre 10ms et 15ms selon l'individu.

> *« Latencies less than approximately 10 ms to 15 ms are not perceived as echoes with in-ear monitors. »*, Al Keltz, [10].

Au delà de ce seuil, le musicien est impacté dans son jeu. Si la latence est supérieure à 15ms mais reste relativement faible alors le musicien va inconsciemment ralentir. En effet, il aura l'impression de jouer en avance sur le son que produit son instrument. Lorsque la latence est beaucoup plus élevée le musicien ne peut tout simplement pas jouer comme le son qu'il entend ne correspond pas aux notes qu'il est en train de jouer.

Comme nous travaillons sur un signal numérique, nous devons distinguer deux périodes. La première est le temps de latence dû à la carte son car celle-ci transforme un signal continu en signal discret et remplit par la même occasion un buffer de taille paramétrable. Plus la taille du buffer est grande, plus le musicien entendra tard ce qu'il vient de jouer. La deuxième période est le temps de calcul que requiert le réseau une fois qu'il a récupéré un buffer depuis la carte son. Du point de vue de la recherche, nous pouvons employer un réseau sur une machine avec assez de puissance de calcul pour que cet aspect ne soit pas handicapant. Toutefois, dans un cadre industriel il n'est pas concevable de produire en série une machine de ce type dans un format assez compact pour qu'elle soit utilisable et transportable par des musiciens. Donc nous devons trouver un réseau suffisamment bon pour reproduire des effets complexes et optimal afin de fonctionner sur une machine dont les ressources sont limitées.

Pour dire qu'un effet est utilisable en temps réel, il faut donc qu'il soit

capable de prendre des échantillons audio en entrée et qu'il renvoie le son transformé en moins de 15ms. Supposons qu'un réseau attende 4096 échantillons audio en entrée (comme le réseau présenté dans [4]), si la fréquence d'échantillonnage est de 44100Hz, cela représente alors environ 93ms d'audio. Sans même prendre en considération le temps de calcul, nous ne sommes pas capables d'utiliser ce réseau en temps réel. Il faut donc que nous prenions un nombre d'échantillons plus petit. J'ai choisi de prendre 256 échantillons ($\sim 5,8ms$) en entrée du réseau de neurones, ce qui nous laisse un peu moins de 10ms pour le temps de calcul. Ce dilemme est une question d'équilibre entre les deux périodes. Si on augmente la taille du buffer, on aura plus d'informations sur les fréquences basses notamment mais la latence de la carte son sera plus importante et donc le temps de calcul devra être plus court. À l'inverse, si on prend de plus petits buffers, on devra augmenter le nombre de fenêtres qui constituent le contexte afin de ne pas perdre trop d'informations sur les fréquences basses. De même si le contexte est plus grand alors le temps de calcul sera plus long.

Comme je suis parti de l'état de l'art, la latence et le temps de calcul n'ont pas été des limitations au début de mon stage. J'ai commencé à me pencher sur cet aspect une fois que j'ai réussi à obtenir un réseau capable de reproduire convenablement un effet de distorsion d'un ampli à lampe comme le montrent les résultats que je présente dans la Section 3.6.

3.7.1 Suppression du contexte

Le premier problème qui s'est présenté à moi est l'utilisation du contexte futur dans le modèle CRAFx. En traitement d'images, il est possible de remplir les bords d'une image de différentes manières (miroir, zéros, hasard, ...). Malheureusement, ce n'est pas aussi vrai en traitement du son. Quand le musicien commence à jouer, on sait qu'avant il n'y avait que du silence, donc des zéros dans le buffer. Mais rien ne nous dit ce dont seront composés les futurs buffers que passera la carte son.

Mon premier test pour retirer le contexte a été de savoir s'il était possible d'entraîner le réseau avec un contexte passé et futur puis de supprimer les

poids des fenêtres correspondantes au contexte. TensorFlow permet une telle manipulation car on peut accéder aux poids avec l'attribut *.weights* d'une couche entraînée. Une couche bidirectionnelle LSTM est composée de 6 poids différents (dans cet exemple je prends la première couche bidirectionnelle de mon réseau) :

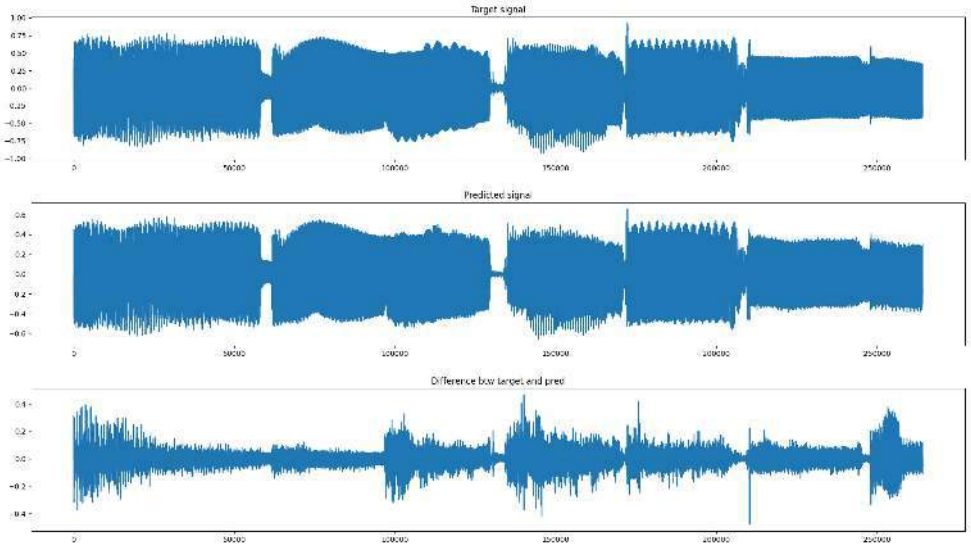
		Matrix size
forward/kernel	backward/_	(576, 512)
forward/rec_kernel	backward/_	(128, 512)
forward/bias	backward/_	(512,)

La taille de chaque matrice varie selon la couche en fonction du tenseur que reçoit la couche en entrée et du nombre de paramètres internes à cette couche. Il est très facile à partir de ce modèle d'en créer une version alternative où l'on ne chargerait que certaines parties des couches afin de ne garder que la partie relative à la fenêtre présente. Bien que l'implémentation soit faisable d'un point de vue technique, les résultats que j'ai obtenu avec cette idée sont très mauvais, à peine audibles. Je n'ai pas passé plus de temps là-dessus, me doutant dès le départ que cette voie ne serait probablement pas bonne, j'ai préféré m'en assurer avant d'en explorer d'autres.

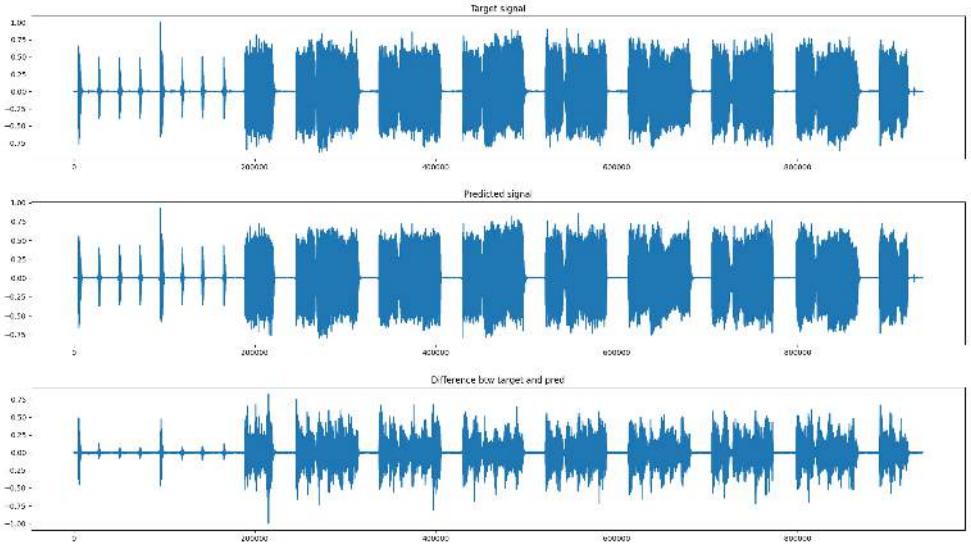
Mon deuxième essai a été de créer un nouveau modèle en retirant le contexte autour de la fenêtre que l'on souhaite émuler (2048 échantillons par fenêtre, saut de 1024 échantillons entre deux fenêtres). Je n'ai pas changé immédiatement la taille des buffers car je voulais me rendre compte de l'impact du contexte sur la qualité de l'émulation du réseau. La Figure 20 montre qu'on obtient une erreur assez importante entre le signal de référence et le signal prédit. Au niveau sonore, on obtient un son beaucoup plus sourd, étouffé, que l'on obtient généralement en appliquant la technique du *palm mute* sur la guitare. Les fréquences les plus hautes qui apportent de la clarté au son sont ainsi supprimées.

3.7.2 Réduction du nombre d'échantillons par fenêtre

À partir du résultat précédent, j'ai décidé de réduire la taille des fenêtres que je passe en entrée du réseau de neurones à 256, toujours afin de pouvoir



(a) Comparaison du signal prédit (au milieu) et du signal de référence (en haut) avec des notes tenues. Le spectre du bas est la différence entre les deux signaux.



(b) Comparaison du signal prédit (au milieu) et du signal de référence (en haut) avec des notes courtes et jouées rapidement. Le spectre du bas est la différence entre les deux signaux.

FIGURE 20 – Spectres des signaux prédits avec un réseau entraîné sans contexte et une fenêtre de 2048 échantillons. Les extraits audio et les images sont disponibles à https://gabriel-weil.emi.u-bordeaux.fr/no_context/

obtenir un réseau performant en temps réel. Comme dans la dernière expérience le contexte semblait jouer sur la clarté (= les fréquences aiguës) du signal, j'ai choisi de ne faire ce test qu'avec la présence du contexte. Cependant, le contexte faisant désormais référence uniquement au contexte passé, j'ai retiré le contexte futur qui n'a pas lieu d'être dans une application temps-réel. J'ai aussi augmenté la *batch size* de l'entraînement de 124 à 500, car l'entraînement se fait sur plus de fenêtres de plus petite taille.

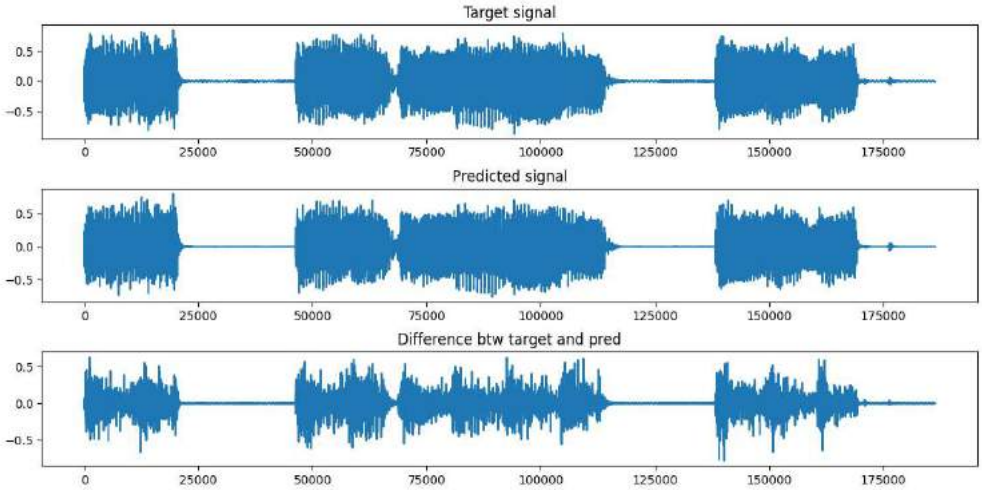
Le résultat obtenu est meilleur que celui que l'on obtient avec des fenêtres plus grandes et sans contexte. Toutefois, la différence entre le signal prédit et le signal de référence est encore grande (Figure 21). En faisant une analyse fréquentielle avec Audacity [20] (Figure 22) j'ai observé que le réseau peine à reproduire les fréquences les plus hautes (de 3000Hz à 7000Hz) et reproduit en moyenne 3dB moins fort les mediums (de 1100Hz à 3000Hz). Bien que les résultats soient différents, ils confirment néanmoins que le contexte joue sur la qualité des fréquences les plus aiguës et la taille de la fenêtre ne semble pas impacter les fréquences les plus graves.

En regardant de plus près les signaux de la Figure 21, il semblerait que le réseau n'arrive pas à reproduire les pics présents dans le signal de référence (Figure 23). Pour que le réseau apprenne à reproduire ces pics correctement j'ai donc augmenté le paramètre α du filtre de pré-emphase de premier ordre qui devrait permettre au réseau de distinguer plus facilement les fortes modulations dans le signal :

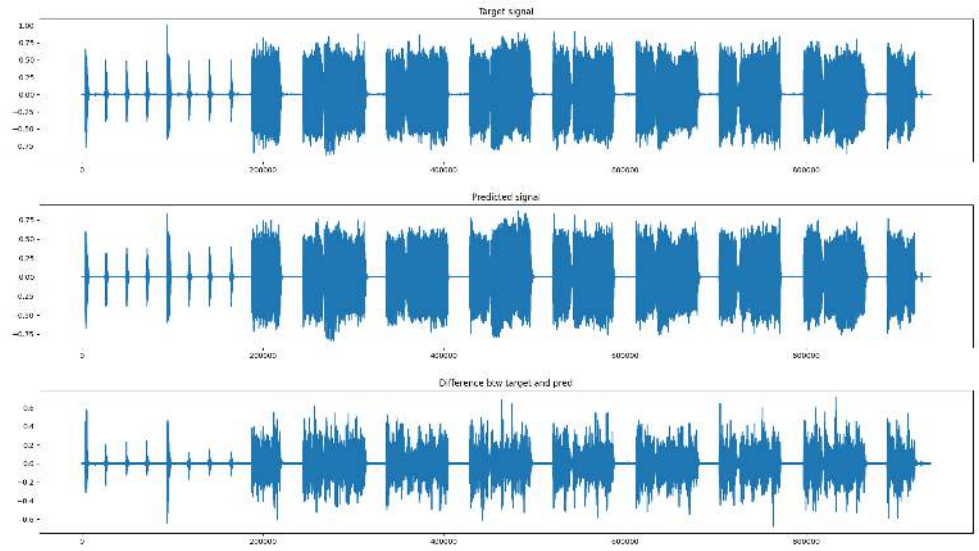
$$H(z) = 1 - \mathbf{0.95}z^{-1} \tag{5}$$

Dans le même temps, comme je travaillais sur la contrainte de temps réel et que j'étais aussi limité sur le temps de calcul, il m'a semblé judicieux de supprimer les opérations chronophages présentes dans le réseau et dont l'utilité n'était pas claire. Concrètement, j'ai supprimé les couches bidirectionnelles de l'espace latent, laissant donc 3 couches LSTM unidirectionnelles

20. https://manual.audacityteam.org/man/plot_spectrum.html, last accessed : 26-08-2020

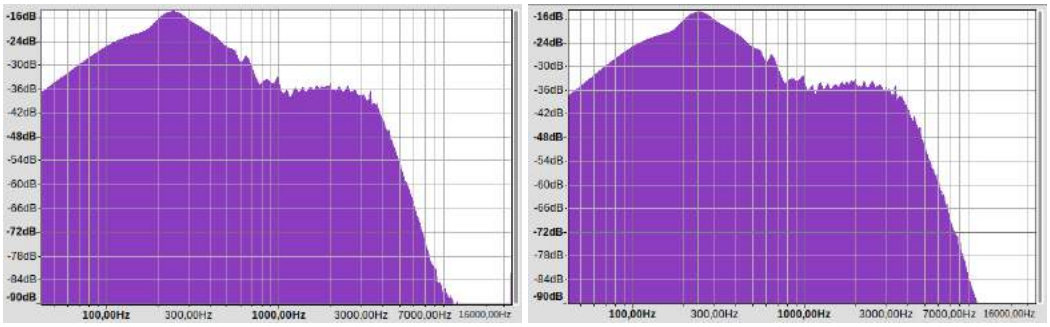


(a) Comparaison du signal prédit (au milieu) et du signal de référence (en haut) avec des notes tenues. Le spectre du bas est la différence entre les deux signaux.



(b) Comparaison du signal prédit (au milieu) et du signal de référence (en haut) avec des notes courtes et jouées rapidement. Le spectre du bas est la différence entre les deux signaux.

FIGURE 21 – Spectres des signaux prédits avec un réseau entraîné avec un contexte de 8 fenêtres, et 256 échantillons par fenêtres. Les extraits audio et les images sont disponibles à https://gabriel-weil.emi.u-bordeaux.fr/no_context_frames=256/



(a) (b)

FIGURE 22 – Analyse spectrale du signal prédit (a) et du signal de référence (b). Images en grande taille disponibles à https://gabriel-weil.emi.u-bordeaux.fr/no_context_frames=256/

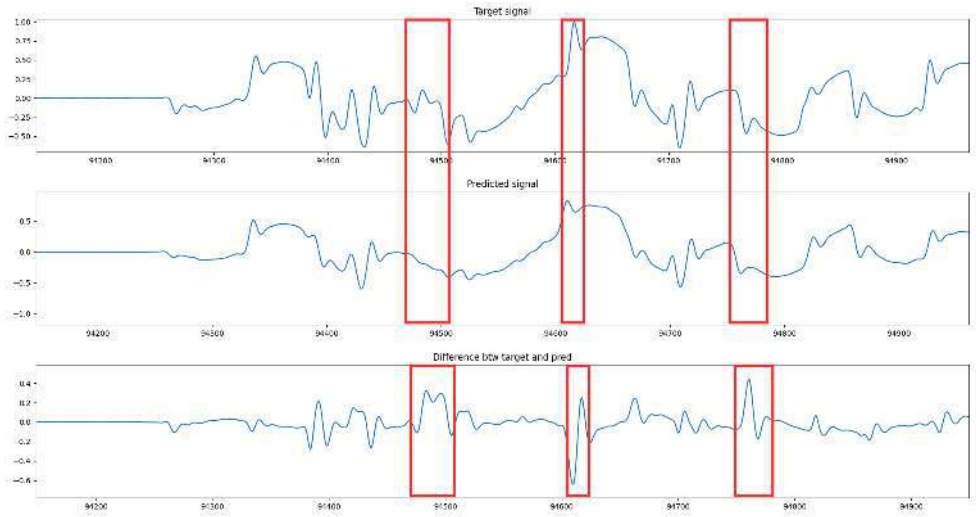


FIGURE 23 – Zoom sur les signaux (b) de la Figure 21. Les erreurs semblent être localisées autour des pics présents dans le signal.

dont j'ai doublé le nombre d'unités internes. La Figure 24 montre les résultats obtenus avec ce réseau. L'erreur est beaucoup moins importante sur les notes tenues. Toutefois, elle demeure assez importante sur les notes courtes. Au niveau de la perception, le signal prédit par le réseau est beaucoup plus proche du signal de référence bien qu'encore différenciable. Un bon aspect est que le signal prédit ne fait plus « numérique » ou « digital » comme certains anciens résultats et procure une sensation musicale plus réaliste.

3.7.3 Mise à jour de TensorFlow (v2.2 -> v2.3)

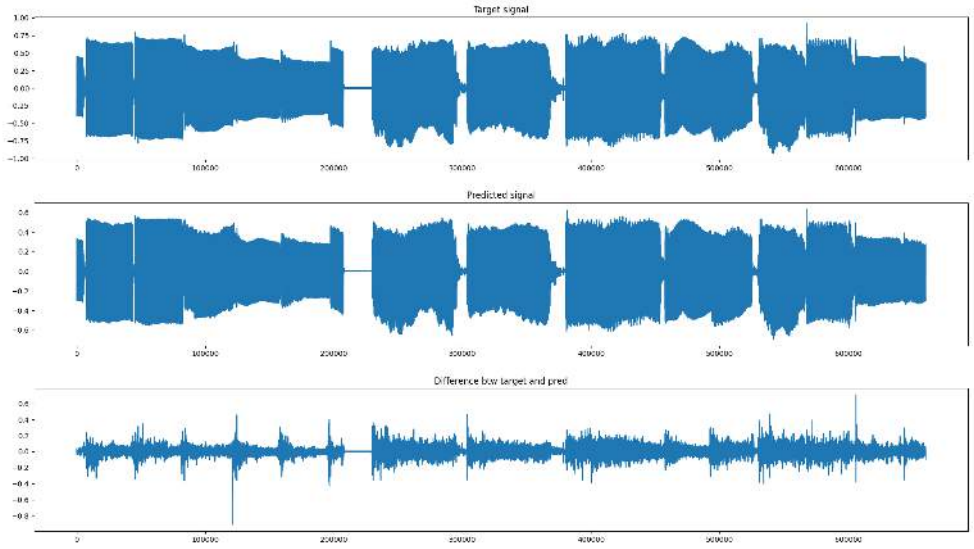
Dans cette mise à jour une nouvelle couche pour réaliser la déconvolution de données en 1D a été ajoutée : **Conv1DTranspose**. Jusqu'à présent pour faire la déconvolution en fin de réseau, j'augmentais la dimension du tenseur, j'appliquais la couche **Conv2DTranspose** puis je réduisais la dimension des données pour retrouver la dimension d'origine. Cette solution était celle proposée sur le GitHub de TensorFlow [21] ou sur Stack Overflow [22].

Lors de cette mise à jour j'ai donc remplacé ma fonction de déconvolution en 2D par **Conv1DTranspose** afin de voir si les résultats changeaient ou non. À ma grande surprise les résultats ont été bien meilleurs. Les signaux prédits sur différents extraits audio ne sont plus différenciables à l'oreille de leur référence et la différence entre les spectres est très faible (Figure 25).

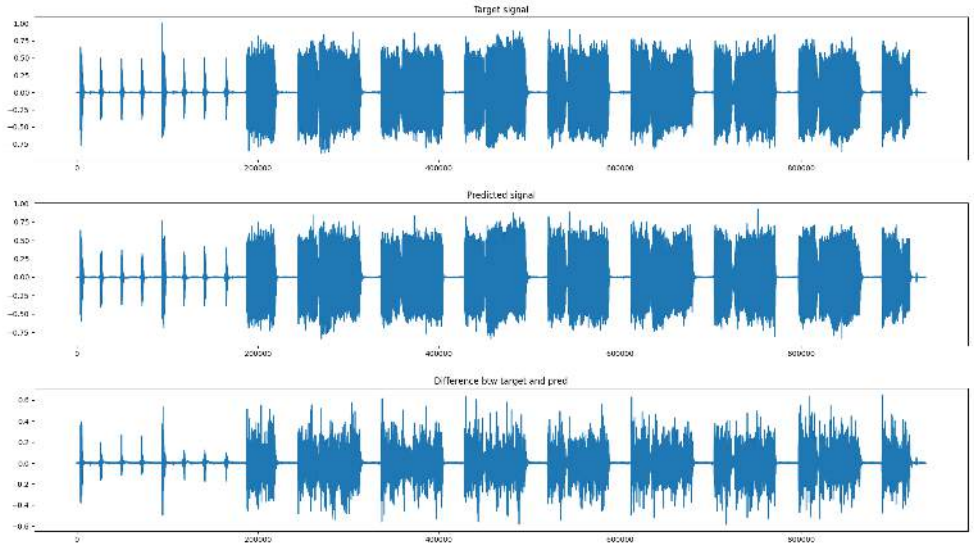
Ces résultats répondent donc à mon objectif de départ, à savoir adapter le réseau CRAFx pour le rendre performant avec des données réalistes qui imitent le jeu d'un guitariste et le rendre apte à une utilisation en temps réel en supprimant le contexte futur. De plus, en supprimant les couches bidirectionnelles j'ai commencé à alléger le réseau pour le rendre plus performant et ainsi réduire le temps de calcul (voir la Section 3.7.4).

L'architecture finale du réseau de neurones est écrite en annexe et une image de l'architecture est disponible à l'adresse https://gabriel-weil.emi.u-bordeaux.fr/realtime_ht1/final.png (à cause de sa taille l'image n'était pas suffisamment visible sur une page de ce rapport).

21. https://github.com/tensorflow/tensorflow/issues/30309, last accessed : 30-08-2020
22. https://stackoverflow.com/questions/44061208/how-to-implement-the-conv1dtranspose-in-keras/45788699#45788699, last accessed : 30-08-2020

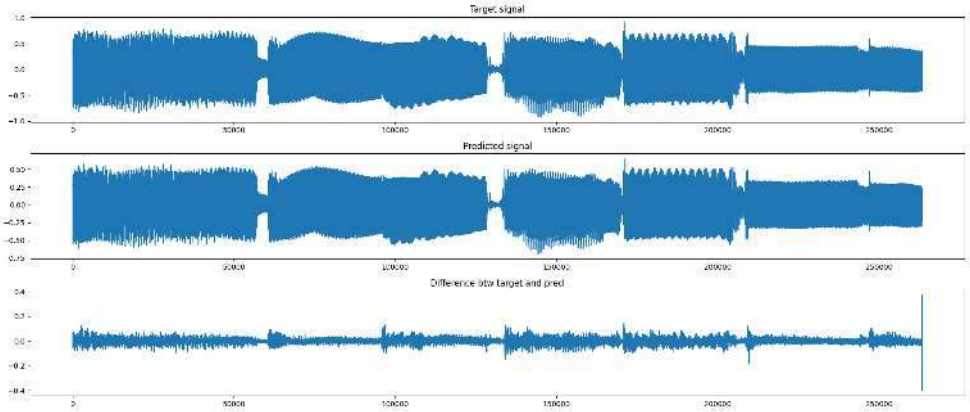


(a)

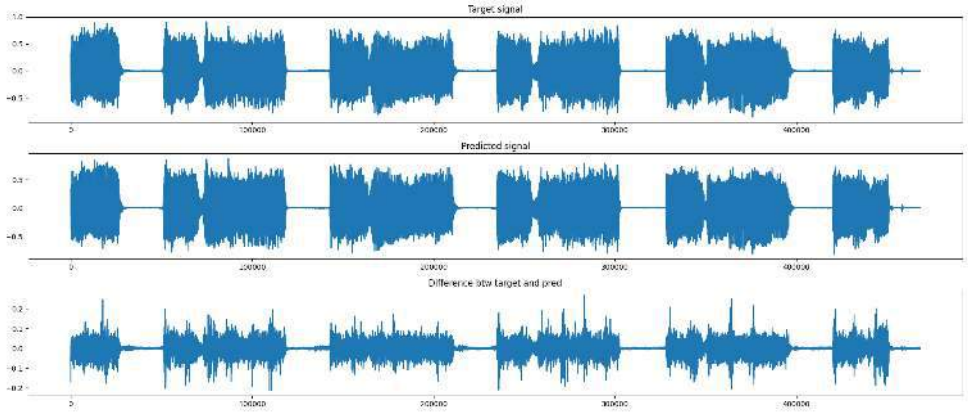


(b)

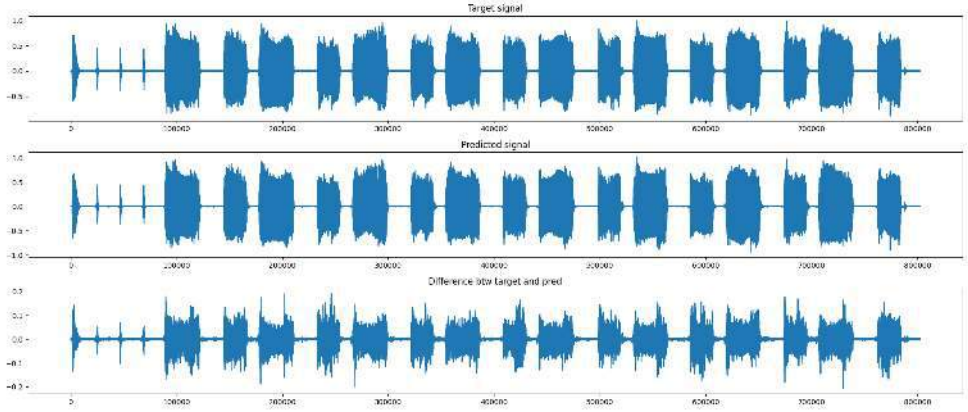
FIGURE 24 – Signaux prédits avec 3 couches LSTM unidirectionnelles dans la partie latente, avec des notes tenues (a) et avec des notes courtes (b). Extraits audio et images disponibles à https://gabriel-weil.emi.u-bordeaux.fr/3LSTMs/



(a)



(b)



(c)

FIGURE 25 – Signaux prédits avec la couche Conv1DTranspose en sortie du réseau, avec des notes tenues (a) et avec des notes courtes (b) et (c). Extraits audio et images disponibles à https://gabriel-weil.emi.u-bordeaux.fr/realtime_ht1/

3.7.4 Implémentation d'une application pour utiliser le réseau en temps réel

J'ai passé plusieurs semaines de ce stage à jongler entre optimisation, entraînements du réseau afin d'obtenir l'architecture finale précédente et implémentation d'une application qui permettrait de jouer directement en branchant une guitare dans une carte son en testant les capacités du modèle à fonctionner en temps réel.

J'ai fait ma première implémentation en Python, ce qui fut un mauvais choix de ma part comme Python est un langage interprété et qu'il est très lent pour effectuer des tâches en temps réel. De plus les bibliothèques en Python pour faire de la manipulation audio sont peu documentées ce qui m'a fait perdre beaucoup de temps à lire le code source des bibliothèques pour comprendre ce qui ne fonctionnait pas. Quand l'application a fonctionné, j'ai constaté que la latence était assez importante (100ms environ avec un réseau de neurones qui émule le signal passé en entrée de la carte son). Avant de poursuivre sur cette voie, j'ai préféré mesurer la latence induite par Python sans aucun réseau intermédiaire. Mes essais m'ont indiqué que l'application avait une latence qui varie entre 20ms et 50ms de base. J'ai donc cherché d'autres solutions pour implémenter l'application.

Mes recherches m'ont amené vers TensorFlow Lite [23], qui est une version de TensorFlow dédiée à l'utilisation de réseaux de neurones sur des systèmes embarqués ou l'Internet des Objets, donc avec des ressources de calcul limitées. Cette version est supportée dans plusieurs langages (Python, Swift, Java, C++, ...). Pour l'application je me suis tourné vers JUCE [24] qui est un framework partiellement open-source développé par ROLI qui permet de développer rapidement des plugins et applications audio en C++.

La première étape est de compiler le réseau entraîné à l'aide du compilateur **bazel** (compilateur propriétaire de Google). La compilation étant très lourde, j'ai dû la réaliser sur le serveur car la carte graphique de ma tour

23. https://www.tensorflow.org/lite/, last accessed : 27-08-2020
24. https://juce.com/discover, last accessed : 27-08-2020

d'ordinateur n'avait pas assez de mémoire (GTX960 avec 4Go de mémoire). La compilation doit permettre d'optimiser le réseau de neurones en combinant des couches ou en en remplaçant certaines couches par des calculs moins coûteux qui font la même chose. À l'issue de la compilation nous récupérons donc un fichier qui correspond au modèle optimisé et prêt à être chargé avec TensorFlow Lite.

La seconde étape est de charger le modèle compilé. TensorFlow Lite utilise deux entités, l'interpréteur et le convertisseur. Comme leur nom l'indique, le convertisseur a pour rôle de charger le modèle et de le convertir sous une forme adaptée en fonction de la plateforme et du langage et l'interpréteur se charge de faire les prédictions une fois que le convertisseur a fini son travail.

Je n'ai malheureusement pas eu le temps de compléter cette partie, le code de l'application est encore inachevé. Nonobstant le support de plusieurs langages par TensorFlow Lite, l'API au moment où je travaille dessus est très succincte et loin d'être claire. À plusieurs reprises, j'ai dû aller lire le code source sur le dépôt GitHub du projet pour comprendre comment implémenter différentes parties du programme. Par exemple, la structure de l'objet le plus commun, à savoir le tenseur, n'est pas présentée dans la documentation officielle et c'est donc en fouillant dans le code source que j'ai compris comment je devais mettre en forme les données que j'avais en C++ pour les passer en entrée à l'interpréteur, idem pour la sortie. J'ai aussi eu du mal à forcer la taille des buffers audio de la carte son à partir de l'application JUCE. Ce n'est écrit nulle part. Toutefois, cette manipulation est possible à l'aide de l'objet *deviceManager* qui permet d'accéder aux paramètres du driver audio, en l'occurrence JACK sous Linux. Cette information est disponible en fouillant quelques temps dans l'API de la classe.

De plus, une fois que j'ai pu testé l'application avec un réseau, je me suis rendu compte que le réseau importé par le convertisseur ne correspondait pas à celui que j'avais implémenté. Je m'en suis aperçu car le réseau renvoyait un tenseur avec une seule donnée à l'intérieur. En affichant les différentes couches je me suis rendu compte que les dernières couches qui me permettaient de faire la déconvolution n'apparaissaient pas totalement (j'ai fait ce test avec la version 2.2, quand je devais augmenter la dimension du tenseur

pour faire la déconvolution, puis revenir dans la dimension d'origine). La raison n'est pas clairement donnée sur les différents forums mais il semblerait que cela provienne de la non-compatibilité entre certains opérateurs dans l'API Python de TensorFlow et l'API TensorFlow Lite [25].

3.8 Émulation d'un Mesa Boogie Rectoverb 50

Une interrogation que j'ai eu au cours de mon stage fut à propos de la complexité d'une distorsion d'un amplificateur et celle d'un réseau de neurones. En effet, un amplificateur crée un effet de distorsion à l'aide de composants non-linéaires, et plus un amplificateur va avoir de composants non-linéaires, plus l'effet de distorsion sera complexe. Je me suis donc demandé si un réseau capable d'émuler un amplificateur simple comme le Blackstar HT-1 serait capable d'émuler aussi fidèlement un amplificateur plus gros comme le Mesa Boogie Rectoverb 50.

Pour tenter d'apporter un début de réponse à cette question, j'ai entraîné de nouveau le réseau obtenu à la Section 3.7.3 en remplaçant les données cibles par des échantillons enregistrés avec le Mesa Boogie Rectoverb 50 (voir Section 3.3 pour l'enregistrement des données d'entraînement). La *val_loss* du réseau le plus performant valait environ 0.09 pour le HT-1 et valait 0.07 pour le Rectoverb 50, avant même de faire des tests audibles, l'entraînement semble donc avoir légèrement mieux réussi pour le plus "gros" amplificateur.

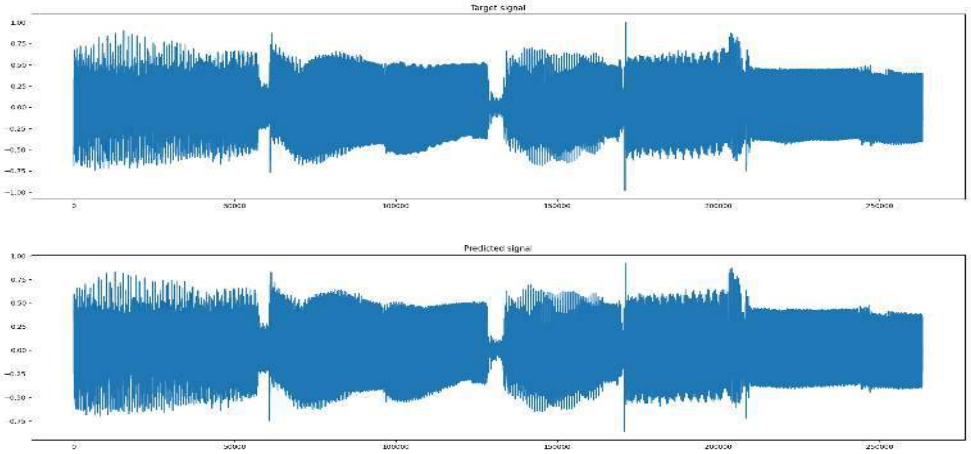
J'ai réalisé plusieurs tests sur le réseau obtenu afin de se rendre compte des limites de l'entraînement réalisé sur le réseau. J'ai réalisé des tests similaires à ceux que je présente précédemment dans ce rapport mais j'ai aussi réalisé des tests avec des données sur lesquelles le réseau n'a pas été entraîné que j'avais acquises lors de la session enregistrement du Mesa Boogie au studio du SCRIME.

Parmi les données "connues" (qui ont servies pendant l'entraînement), j'ai repris l'extrait par défaut avec des notes tenues et j'ai fait un autre test sur un extrait plus proche du style musical « *thrash metal* » pour lequel cet

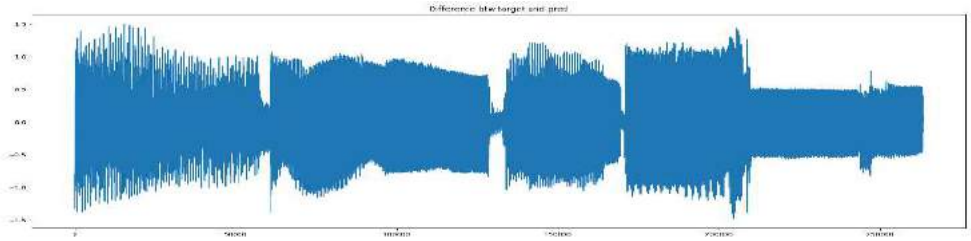
25. https://www.tensorflow.org/lite/guide/ops_compatibility, last accessed : 27-08-2020

amplificateur est particulièrement adapté (Figure 26). Le signal prédit est quasi identique à celui de référence du point de vue sonore. Au niveau du spectre, bien que les signaux prédits et de référence se ressemblent fortement, en regardant de plus près la phase semble inversée ce qui ne permet pas d'observer facilement la différence entre le signal prédit et celui de référence, comme le montre la Figure 26b.

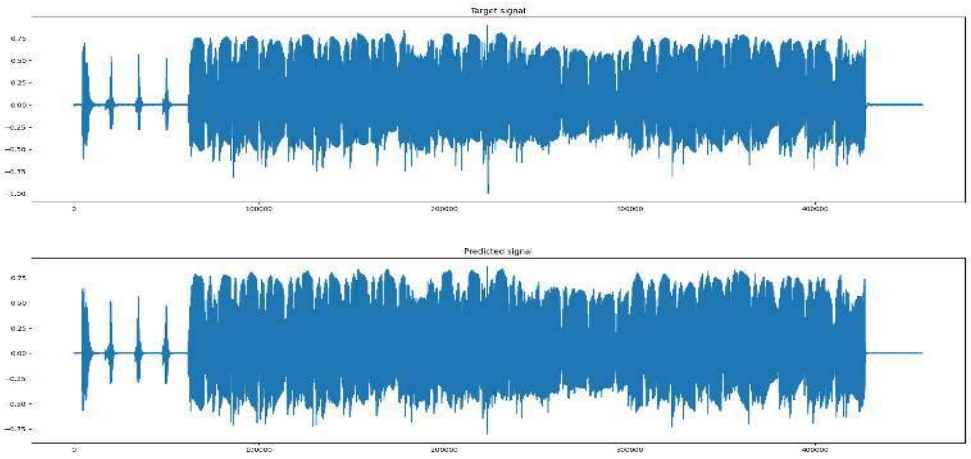
Plus intéressant, en utilisant des données "inconnues", qui n'ont pas été utilisées lors de la phase d'apprentissage, le réseau arrive à reproduire fidèlement le son de référence dans une certaine mesure. La Figure 27 montre trois extraits enregistrés avec une guitare différente de celle utilisée dans les données d'entraînement (une Gibson Flying V Faded), avec trois styles de jeu différents. Les extraits des Figures 27a et 27b sont joués en *Drop D*, ce qui signifie que la corde la plus basse de la guitare est accordée une note plus grave, soit D à la place de E (73Hz à la place de 82Hz). L'extrait de la Figure 27a est joué en partie en *palm mute* (la paume de la main droite vient étouffer les cordes en même temps qu'elles sont jouées), l'extrait de la Figure 27b est une succession d'accords ouverts tenus sans *palm mute* et l'extrait de la Figure 27c est une bossa nova jouée aux doigts. Ces résultats sont particulièrement intéressants car le réseau reproduit correctement la distorsion en *Drop D* et avec du *palm mute*, sans même avoir été entraîné avec des données qui prennent en compte ces paramètres. Par contre, le réseau reproduit mal la distorsion sur l'extrait de bossa nova, sûrement parce que les enregistrements pour l'entraînement sont joués au médiator et donc le réseau reproduit l'effet dynamique/agressif de ce style de jeu. De plus, sans même faire attention à la phase, nous voyons au niveau des pics dans la Figure 27b que le réseau ne semble pas reproduire exactement la distorsion du Rectoverb 50. Cependant, comme les fréquences basses se mélangent à l'octave sur certaines notes pendant l'extrait nous n'entendons pas des différences grossières ou trompeuses et au final l'ensemble sonne suffisamment proche du signal de référence pour ne pas sembler faux.



(a)

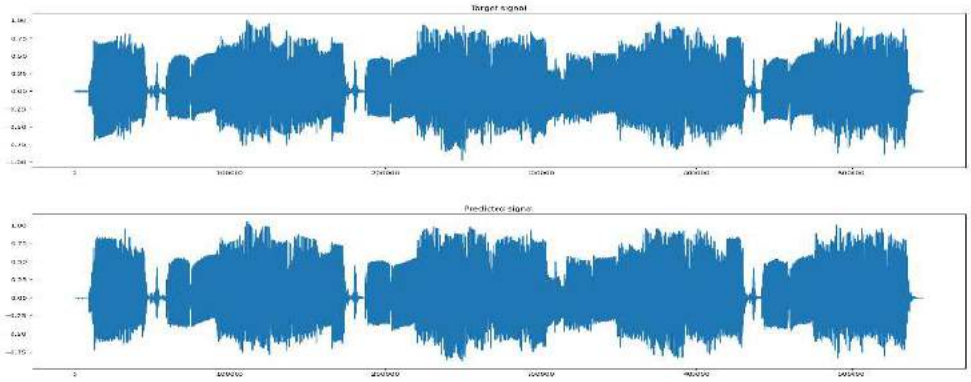


(b)

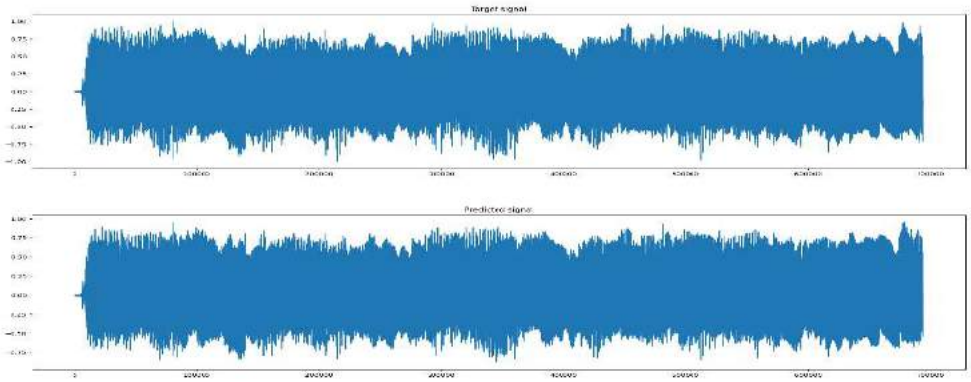


(c)

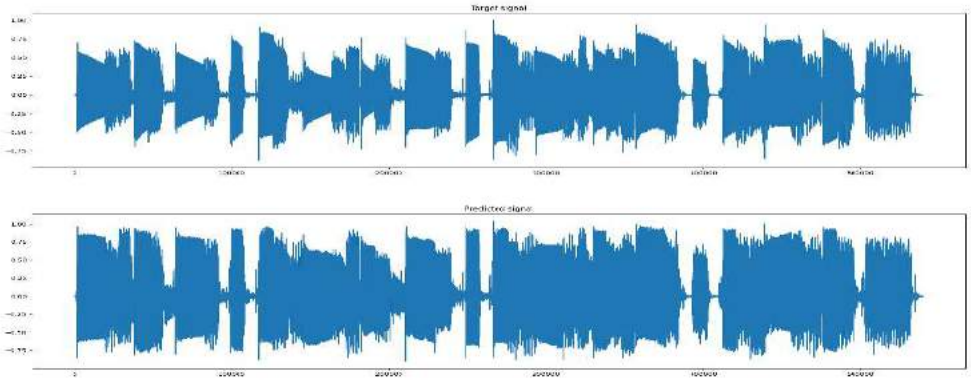
FIGURE 26 – Émulation du Mesa Boogie Rectoverb 50. Signal prédit (en haut) et signal de référence (en bas) avec des notes tenues (a) et avec un extrait dans un style *thrash metal*, avec des notes courtes et rapides (c). L'image (b) représente la différence entre les signaux de l'image (a). Extraits audio et images disponibles à https://gabriel-weil.emi.u-bordeaux.fr/mesa_boogie/



(a)



(b)



(c)

FIGURE 27 – Émulation du Mesa Boogie Rectoverb 50. Signal prédit (en haut) et signal de référence (en bas). Extraits audio et images disponibles à https://gabriel-weil.emi.u-bordeaux.fr/mesa_boogie/

4 Bilan

Mon stage m'a permis de travailler sur des technologies très performantes et largement adoptées dans la communauté scientifique et industrielle. Aussi, j'ai pu travailler sur un sujet passionnant dans lequel je me suis épanoui.

Mon passage au sein du laboratoire du SCRIME a permis la création d'un ensemble de données supplémentaire pour faire avancer la recherche dans le domaine de l'émulation d'effets audio avec des réseaux de neurones. Le réseau que j'ai construit peut permettre de faire avancer l'état de l'art car il propose une émulation fidèle dans un cadre d'utilisation réaliste et non théorique étant donné que peu de guitaristes se contentent de jouer des notes seules tenues plusieurs secondes. D'autre part, le réseau que je propose s'interroge sur le contexte d'utilisation en dehors du cadre expérimental, et de fait, prend en compte des limitations induites par la notion de temps-réel dans le milieu de la pratique musicale.

Au-delà de mon stage, mon travail a permis de dégager la voie pour un futur travail de thèse où l'émulation d'amplis et la notion de temps-réel seraient traitées plus en profondeur, notamment au travers des données utilisées et de la recherche sur les réseaux de neurones. En effet, une chose que j'ai apprise au cours de mon stage est que la valeur ajoutée d'un réseau de neurones est la base de données utilisée pour l'entraîner, et c'est aujourd'hui une limitation importante pour la recherche car il n'existe pas de base de données publique complète conçue dans le but d'être représentative de plusieurs guitaristes, avec plusieurs styles de jeu et techniques, plusieurs guitares différentes, etc. Il est possible que ce soit aussi la raison pour laquelle l'adoption de l'intelligence artificielle par l'industrie des effets audio numériques n'est pas encore importante.

Plus généralement, l'état de l'art et mes résultats m'amènent à penser que le domaine de l'émulation d'effets audio et d'instruments à l'aide de réseaux de neurones avec des méthodes black-box n'en est qu'à ses débuts. Aujourd'hui, le nombre d'applications et de plugins basés sur de l'intelligence artificielle commercialisés n'est pas très important dans le secteur musical. Toutefois avec la chute des prix des GPU à chaque nouvelle génération et

la tendance à fournir des logiciels plutôt que du matériel (par exemple les voitures Tesla, les plateformes de *cloud gaming*, ...), ce n'est qu'une question de temps avant que les éditeurs ne proposent massivement des solutions à base d'IA où l'émulation n'est plus produite avec une approche *white-box*, c'est-à-dire une approche où l'on cherche à remplacer chaque composant électronique par une équation mathématique.

5 Annexes

- Architecture du réseau de neurones final

```
TensorFlow version : 2.3.0
Keras version : 2.4.0
Model: "functional_1"
```

Layer (type)	Output Shape	Param #	Connected to
input (InputLayer)	[(None, 9, 256, 1)]	0	
conv1d (TimeDistributed)	(None, 9, 256, 64)	4160	input[0][0]
abs (TimeDistributed)	(None, 9, 256, 64)	0	conv1d[0][0]
conv1d_local (TimeDistributed)	(None, 9, 256, 64)	8256	abs[0][0]
softplus (TimeDistributed)	(None, 9, 256, 64)	0	conv1d_local[0][0]
maxpooling (TimeDistributed)	(None, 9, 64, 64)	0	softplus[0][0]
unstack_0 (Lambda)	[(None, 64, 64), (No	0	maxpooling[0][0]
concatenate (Concatenate)	(None, 64, 576)	0	unstack_0[0][0] unstack_0[0][1] unstack_0[0][2] unstack_0[0][3] unstack_0[0][4] unstack_0[0][5] unstack_0[0][6] unstack_0[0][7] unstack_0[0][8]
lstm_0 (LSTM)	(None, 64, 256)	852992	concatenate[0][0]
lstm_1 (LSTM)	(None, 64, 128)	197120	lstm_0[0][0]
lstm_2 (LSTM)	(None, 64, 64)	49408	lstm_1[0][0]
saaf_latent (SAAF)	(None, 64, 64)	832	lstm_2[0][0]
upsampling1d (UpSampling1D)	(None, 256, 64)	0	saaf_latent[0][0]
unstack_1 (Lambda)	[(None, 256, 64), (N	0	conv1d[0][0]
mult (Multiply)	(None, 256, 64)	0	upsampling1d[0][0] unstack_1[0][8]
dense_0 (Dense)	(None, 256, 64)	4160	mult[0][0]

```
--------------------------------------------------------------------------------------------
dense_1 (Dense)                 (None, 256, 32)      2080        dense_0[0][0]
--------------------------------------------------------------------------------------------
dense_2 (Dense)                 (None, 256, 32)      1056        dense_1[0][0]
--------------------------------------------------------------------------------------------
dense_3 (Dense)                 (None, 256, 64)      2112        dense_2[0][0]
--------------------------------------------------------------------------------------------
saaf_dnn (SAAF)                 (None, 256, 64)      832         dense_3[0][0]
--------------------------------------------------------------------------------------------
abs_se (Activation)             (None, 256, 64)      0           saaf_dnn[0][0]
--------------------------------------------------------------------------------------------
globalAverage_se (GlobalAverage (None, 64)           0           abs_se[0][0]
--------------------------------------------------------------------------------------------
reshape_se (Reshape)            (None, 1, 64)        0           globalAverage_se[0][0]
--------------------------------------------------------------------------------------------
dense_0_se (Dense)              (None, 1, 1024)      66560       reshape_se[0][0]
--------------------------------------------------------------------------------------------
dense_1_se (Dense)              (None, 1, 64)        65600       dense_0_se[0][0]
--------------------------------------------------------------------------------------------
mult_se (Multiply)              (None, 256, 64)      0           saaf_dnn[0][0]
                                                                 dense_1_se[0][0]
--------------------------------------------------------------------------------------------
add (Add)                       (None, 256, 64)      0           mult_se[0][0]
                                                                 mult[0][0]
--------------------------------------------------------------------------------------------
deconv (Conv1DTranspose)        (None, 256, 1)       4097        add[0][0]
============================================================================================
Total params: 1,259,265
Trainable params: 1,259,265
Non-trainable params: 0
--------------------------------------------------------------------------------------------
```

Références

[1] A. Wright, E.-P. Damskägg, V. Välimäki, *et al.*, "Real-time black-box modelling with recurrent neural networks," in *22nd International Conference on Digital Audio Effects (DAFx-19)*, 2019.

[2] J.-P. Briot, G. Hadjeres, and F.-D. Pachet, "Deep learning techniques for music generation–a survey," *arXiv preprint arXiv :1709.01620*, 2017.

[3] A. Elgammal, B. Liu, M. Elhoseiny, and M. Mazzone, "Can : Creative adversarial networks, generating" art" by learning about styles and deviating from style norms," *arXiv preprint arXiv :1706.07068*, 2017.

[4] M. A. Martínez Ramírez, E. Benetos, and J. D. Reiss, "Deep learning for black-box modeling of audio effects," *Applied Sciences*, vol. 10, no. 2, p. 638, 2020.

[5] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7132–7141, 2018.

[6] L. Hou, D. Samaras, T. M. Kurc, Y. Gao, and J. H. Saltz, "Neural networks with smooth adaptive activation functions for regression," *arXiv preprint arXiv :1608.06557*, 2016.

[7] A. I. Humayun, S. Ghaffarzadegan, Z. Feng, and T. Hasan, "Learning front-end filter-bank parameters using convolutional neural networks for abnormal heart sound detection," in *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 1408–1411, IEEE, 2018.

[8] H. B. Sailor, D. M. Agrawal, and H. A. Patil, "Unsupervised filterbank learning using convolutional restricted boltzmann machine for environmental sound classification.," in *INTERSPEECH*, pp. 3107–3111, 2017.

[9] T. Kim, J. Lee, and J. Nam, "Sample-level cnn architectures for music auto-tagging using raw waveforms," in *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pp. 366–370, IEEE, 2018.

[10] A. Keltz, "Opening pandora's box ?, the "l" word - latency and digital audio systems." http://whirlwindusa.com/support/tech-articles/opening-pandoras-box/. Accessed : 24-07-2020.