Projet de fin d'étude

Émulation d'un amplificateur à lampe par deep learning

Auteurs:
Camille Meyrignac
Jérémi Bernard
Maxence Lévêque
Gabriel Weil

Client:
Pierre Hanna

Mars 2020

Contents

1 Introduction

Dans le cadre du Projet de Fin d'Études (PFE), nous nous sommes intéressés à la synthèse sonore par apprentissage profond. Nous n'avons pas eu l'occasion d'aborder ce champ d'application de la synthèse sonore dans le cadre de nos études à cause de l'annulation de l'UE à choix Informatique Musicale. Ce projet fut une expérience pédagogique à la fois concernant le domaine d'application qui était nouveau pour nous, ainsi que sur le travail et l'organisation de celui-ci qui furent très différents de ce que nous avions appris au cours de l'UE Projet de Programmation (PdP) dispensée en première année du Master.

Le sujet du projet est le fruit d'une concertation entre les membres de notre groupe et notre client, M. Hanna. Le but était de faire une émulation d'effet analogique par apprentissage profond. Nous nous sommes basés sur le papier "*Real-time black-box modelling with recurrent neural networks*" d'Alec Wright[1] comme nous avions accès à l'amplificateur qu'il reproduit dans son papier. Nous expliquerons dans la section 3 les raisons qui nous ont conduits à émuler le canal de distorsion de l'ampli plutôt qu'un autre effet (comme l'égalisation du son, ou la réverbération de l'ampli). Notre projet se distingue donc dès le départ du fait qu'il se base exclusivement sur le papier donné[1]. Au-delà de la documentation technique liée aux technologies que nous avons utilisées, nous ne disposions d'aucune autre source d'information quant au réseau de neurones à implémenter.

Dans un premier temps, nous dépeindrons brièvement l'état actuel de la recherche dans la synthèse sonore par apprentissage profond ainsi que son application dans le milieu industriel, puis nous aborderons les spécificités du projet à travers les besoins de notre client. Nous présenterons ensuite l'acquisition des données, puis l'architecture du logiciel ainsi que son implémentation et enfin les tests réalisés.

Quelques termes importants...

- **Apprentissage profond**: Méthode d'apprentissage utilisée en intelligence artificielle où l'apprentissage d'un réseau de neurones est basé sur une base de données. Il existe plusieurs approches quant à la forme que doit prendre cet apprentissage. Dans notre cas l'approche choisie est une approche black-box, nous présenterons plus en détail ce que cela induit dans la Section 2.

- **Réseau de neurones récurrent**: Aussi appelé RNN (pour Recurrent Neural Network), est une classe de réseau neuronal où les connexions entre les neurones présentent un caractère récurrent dans le temps. Les états internes des neurones se basent sur les états précédents. Cela permet au réseau d'avoir un comportement cohérent dans le temps et cela explique son utilisation très prisée dans des problèmes de traitement de séquences temporelles.

- **LSTM** (Long Short-Term Memory): LSTM est un type d'unité récurrente. Sa particularité est que son état interne est composé de deux vecteurs. À chaque itération, chacun de ces vecteurs adapte ses poids en prenant en compte l'état actuel et l'état ultérieur de l'un et l'autre.

- **VST** (Virtual Studio Technology): interface logicielle de plug-in audio développée par Steinberg, utilisée comme un standard par la plupart des séquenceurs et outils de musique assistée par ordinateur. Il permet de créer des effets ou des instruments virtuels complets.

2 État de l'art

La recherche autour de la modélisation d'effets audio par des méthodes d'apprentissage profond est un champ de recherche très actif. Cela est dû à une forte demande de la part des musiciens qui désirent utiliser du matériel analogique auquel ils n'ont pas accès (instruments, amplis, effets, ...). En effet une partie de ce matériel n'est plus commercialisée ni fabriquée ce qui la rend de plus en plus rare à trouver dans un bon état et par conséquent, de plus en plus cher. L'émulation de matériel analogique est devenue aujourd'hui une part importante de l'industrie de la musique numérique, tout comme de la recherche dans la synthèse sonore.

On peut distinguer trois approches pour émuler un appareil. Une première approche est une approche dite *white box* où toutes les informations relatives au fonctionnement interne de l'appareil sont fournies au logiciel. Ainsi n'importe quel signal subit une transformation similaire à celle effectuée par l'appareil original.

Une seconde approche est l'approche *black box*. Cette approche est celle employée dans le papier que nous avons reproduit. Elle repose sur de l'apprentissage profond, en pratique cela signifie que contrairement à l'approche précédente, ici le logiciel ne dispose d'aucune connaissance sur le fonctionnement interne de l'appareil que l'on cherche à émuler. À la place, nous utilisons une base de données représentative du contexte final d'utilisation du logiciel puis à l'aide d'un réseau de neurones entraîné, le logiciel doit être capable de reproduire un comportement similaire à l'appareil d'origine.

La troisième approche existante est une approche *grey box* qui est une variante hybride des deux approches précédentes où le logiciel comble un manque d'information sur le fonctionnement interne d'un appareil avec une phase d'apprentissage.

Le réseau de neurones le plus plébiscité actuellement est le modèle**WaveNet** développé par les chercheurs du laboratoire **DeepMind** de Google en 2016[2]. L'émulation par apprentissage profond est une technique encore très nouvelle et très expérimentale. On peut toutefois noter que des émulations créées avec des réseaux de neurones commencent à être commercialisées, à l'image du Quad Cortex de Neural DPS[1].

L'amplificateur émulé dans notre papier (Blackstar HT-1, voir Figure 1) étant un amplificateur d'entrée de gamme et assez récent, il n'existe pas aujourd'hui d'émulation de celui-ci. Il s'agit d'une petite tête d'ampli à lampe dont la puissance est de 1W.

[1]Neural DPS (https://neuraldsp.com/quad-cortex)

Figure 1: Tête d'ampli HT-1 de Blackstar.

3 Besoins du client

Les besoins de notre client étaient assez succincts, comme il n'existe pour le moment aucun logiciel mettant en œuvre le papier de recherche étudié le but principal était de créer un précédent. Le logiciel devait pouvoir être utilisé en temps réel, dans un logiciel de musique assistée par ordinateur (MAO) ou de manière autonome, l'utilisateur devait avoir le contrôle sur le volume ainsi que sur l'effet produit, il devait aussi pouvoir l'activer et le désactiver.

Un autre point important de ce projet était de pouvoir dire si le réseau de neurones émule fidèlement l'effet de départ. Pour ce faire nous pouvons nous baser sur des valeurs quantitatives (calculer une valeur qui décrit si le signal émulé est proche ou loin du signal attendu), ou nous pouvons nous baser sur les retours de tests d'écoute. N'ayant pas le temps de mettre en place des tests d'écoute à l'aveugle, nous avons jugé le réseau sur des valeurs quantitatives (voir Section 3). Pour autant, il nous a été demandé de reproduire un effet facilement audible, afin d'avoir un exemple audio. Comme nous disposons d'un modèle du Blackstar HT-1 utilisé dans le papier de recherche nous avons décidé de reproduire son canal de distorsion.

4 Acquisition des données

L'étape la plus importante lors de l'élaboration d'un réseau de neurones est l'acquisition des données d'entraînement. En effet si les données choisies ne sont pas assez diversifiées (ici, cela signifierait l'emploi de différentes guitares, avec des notes seules et des accords, des notes courtes et longues, ...) ou représentatives de l'environnement d'exécution du réseau de neurones alors l'entraînement serait peu efficace car au final, le réseau ne serait pas apte à reproduire l'effet désiré correctement.

Dans notre cas nous avons récupéré une base de données fournie par le

Fraunhofer Institute for Digital Media Technology[2]. Cette base de données est celle utilisée par Alec Wright dans l'article sur lequel nous nous sommes basés. Nous n'avons pas utilisé toute la base de données car elle possède plusieurs dossiers ayant chacun un objectif particulier. Nous avons utilisé le Dataset 3 ainsi que le Dataset 4 dans lequel nous avons sélectionné les enregistrements blues-rock réalisés avec une Ibanez 2820. Cela nous permet d'obtenir à la fois des notes et des accords, joués lentement et rapidement. L'ensemble des fichiers audio utilisés forment une base de données de 7 minutes et 48 secondes. Nous avions aussi préparé une base de données de 3 minutes et 15 secondes avec des échantillons de basse, mais nous ne l'avons pas utilisée par manque de temps.

L'installation pour l'acquisition des données est assez simple, nous utilisons une carte son Focusrite Scarlett 2i2[3] (deuxième génération) et une tête d'ampli à lampe Blackstar HT-1[4] (première génération, voir Figure 2). Nous branchons une sortie de la carte son sur l'entrée de l'ampli, puis nous utilisons la sortie ligne émulée de l'amplificateur pour récupérer le signal distordu. Additionnellement, l'auteur enregistre le bruit généré par les entrées et sorties de la carte son en branchant directement la sortie de la carte son dans l'entrée. Nous avons réalisé cette étape mais la différence entre le signal brut et le signal "bruité" enregistré par la carte son ne présentaient pas de différence notable. Pour cette raison nous ne l'avons pas pris en compte dans notre implémentation.

Nous utilisons la sortie émulée de l'ampli par nécessité et non par choix. Le signal obtenu avec la sortie émulée ne correspond pas exactement au signal que nous obtiendrions avec la sortie haut-parleurs. Le problème vient de l'impédance du signal. Pour obtenir un signal de bonne qualité avec une entrée ligne de la carte son, nous devons récupérer un signal dont l'impédance correspond à celle attendue par l'entrée. L'impédance du signal attendu en entrée de la carte son doit être à environ 10kΩ(niveau ligne) alors que le niveau du signal de la sortie haut-parleur est situé entre 4 et 16Ω. C'est pour cette raison que nous avons utilisé la sortie ligne de l'ampli en dépit de la perte de fidélité par rapport à la modification sonore qu'il effectue.

5 Architecture

5.1 Modèle du réseau récurrent

L'objectif de ce projet étant de faire de la transformation audio, nous étions libre du type de réseau à utiliser et encouragé à faire des tests sur différentes technologies dans le cas où on aurait eu du temps. Pour ne pas s'éparpiller dès le début et pour avoir des résultats le plus rapidement possible, nous avons décidé d'implémenter le même réseau que celui présenté dans le papier de A. Wright[1].

Le réseau présenté dans le papier se compose d'une seule unité récurrente suivie d'une couche "fully connected". Dans son papier, A. Wright expérimente deux types d'unité récurrente différents : une LSTM (Long Short-Term Memory) et une GRU (Gated Recurrent Unit). Il est toutefois conseillé dans le papier d'utiliser une couche LSTM qui permet d'obtenir un meilleur résultat

[2] https://www.idmt.fraunhofer.de/en/business_units/m2d/smt/guitar.html
[3] https://focusrite.com/en/usb-audio-interface/scarlett/scarlett-2i2
[4] https://www.blackstaramps.com/fr/ranges/ht-1

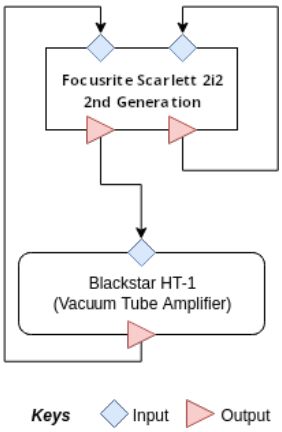

Figure 2: Schéma de l'installation pour l'acquisition des données d'entraînement du réseau de neurones. Les losanges bleus représentent les entrées et les triangles rouges les sorties des appareils utilisés.

comparé à l'utilisation d'une couche GRU. C'est pourquoi nous avons considéré uniquement ce type d'unité. Un schéma de l'architecture du réseau est présenté en Figure 3.
Une étude approfondie des performances de ce réseau a été publiée cette année par le même auteur[3] et montre que le réseau est compatible avec des contraintes de temps réel sous certaines conditions. La partie 7.3 de ce rapport est consacrée à l'analyse des performances de la dernière itération du projet.

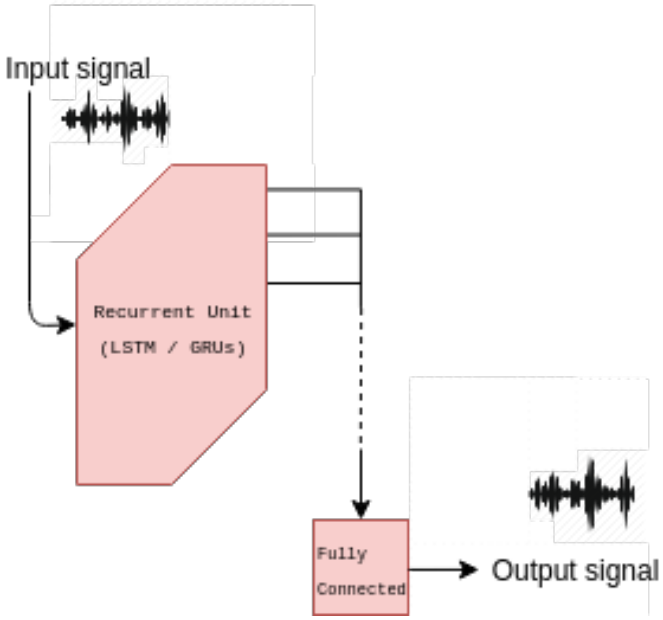

Figure 3: Schéma du modèle implémenté

5.2 Plugin VST

Afin d'utiliser le réseau, nous avons créé un logiciel à l'aide du framework **JUCE**. Grâce au framework nous pouvons produire un logiciel autonome ou un plugin (VST, AU, LV2, ...) avec une interface utilisateur permettant au musicien de varier les paramètres de contrôle sur l'effet. La Figure 4 montre l'interface utilisateur du logiciel.

Figure 4: Aperçu visuel du plugin.

Le plugin permet de récupérer l'entrée microphone de l'ordinateur, et d'y appliquer ou non l'effet désiré. Voici ce qui compose l'interface utilisateur, qui a été créée de manière à rester la plus simple et facile d'utilisation possible :

- Potentiomètre de gain : permet de changer le niveau de l'effet, plus la valeur de gain est élevée, plus le son passé en entrée sera saturé.
- Potentiomètre de volume : permet de changer le volume de la sortie audio globale, avec ou sans effet.
- Bouton d'activation : permet d'activer ou désactiver l'effet.

L'interface a été créée en suivant une architecture de la forme Modèle-Vue. La partie contrôleur est gérée directement dans les mises à jour de la partie modèle (appelée Processor).

6 Implémentation

Ce projet présente deux phases d'implémentation distinctes et indépendantes l'une de l'autre qui ont été faites en parallèle. L'implémentation et l'entraînement du réseau de neurones ont été faits en Python grâce aux bibliothèques **TensorFlow** et **Keras** et l'interface utilisateur a été développée en C++ grâce au framework **JUCE**.

6.1 Implémentation du réseau de neurones

L'implémentation du réseau de neurones a été faite en Python. L'objectif initial était de sauvegarder le réseau dans un format JSON et de le charger en C++ grâce à **Frugally-deep** pour l'utiliser dans le plugin. Dans cette section, nous verrons les détails d'implémentation de la dernière itération du code. Puis nous expliquerons les différents problèmes rencontrés durant le développement du réseau.

Suite au confinement national qui nous a obligé à travailler depuis chez nous, nous avons fourni avec le code python son homonyme Jupyter qui nous a permis de travailler sur Colab comme nous n'avions pas les ressources matérielles nécessaires à disposition pour entraîner nos réseaux. Cette décision a été prise après avoir obtenu l'aval de notre client pour qui cela ne posait aucun problème. Les deux codes sont identiques, c'est la raison pour laquelle les tests ne portent que sur le code Python.

Le code du réseau de neurones est très conventionnel. Nous avons construit une classe pour le définir et l'entraîner (*Model*), une classe pour mettre les données sous une forme compréhensible par le réseau (*Dataset*). La fonction de coût est une simple fonction (*custom_ loss(...)*).

Détails d'implémentation

La fonction de coût : Le réseau de neurones proposé dans le papier d'origine[1] utilise une fonction de coût qui n'est pas implémentée par défaut dans **Keras** ni **TensorFlow**. Le coût se calcule grâce aux équations présentées dans la Figure 5. Le calcul du coût est la somme de l'*ESR* et du *DC term*. L'*ESR* (Error-to-Signal Ratio) représente l'erreur entre le signal prédit par le réseau de neurones et le signal attendu. Le *DC term*, aussi appelé *0Hz term*, est la moyenne de toutes les valeurs dans une fenêtre.

Dans son papier[1], Wright effectue aussi une étape de pré-emphase de signaux. Cette étape a pour but d'améliorer l'apprentissage du réseau dans les hautes fréquences. Le calcul de cette pré-emphase n'a pas été intégré au code pour le moment à cause de problèmes d'implémentation.

$$\mathcal{E}_{\text{ESR}} = \frac{\sum_{n=0}^{N-1} |y_p[n] - \hat{y}_p[n]|^2}{\sum_{n=0}^{N-1} |y_p[n]|^2},$$

$$\mathcal{E}_{\text{DC}} = \frac{|\frac{1}{N}\sum_{n=0}^{N-1}(y[n] - \hat{y}[n])|^2}{\frac{1}{N}\sum_{n=0}^{N-1}|y[n]|^2}.$$

$$\mathcal{E} = \mathcal{E}_{\text{ESR}} + \mathcal{E}_{\text{DC}}.$$

Figure 5: Équation ε pour le calcul de la perte lors de l'entraînement du réseau.

Les deux fonctions calculant ε utilisent les opérations mathématiques internes de **TensorFlow** (*tensorflow.math*) comme elles font les calculs sur des tenseurs. Nous avons eu connaissance de ce fait que tard dans le processus de développement suite à des incohérences entre les valeurs obtenues et les valeurs

attendues. Nous avions à l'origine essayer de transformer ces tenseurs en tableau **numpy** puis d'utiliser les fonctions de cette bibliothèque sans résultats.

Comme **TensorFlow** n'intègre pas de fonction permettant d'appliquer simplement des filtres, nous n'avons pas intégré la pré-emphase. À l'origine nous utilisions une fonction de **scipy** nommée *lfilter* mais qui n'est malheureusement pas compatible avec **TensorFlow**.

Il aurait été envisageable d'implémenter une fonction de pré-emphase basée sur des convolutions en transformant les signaux dans le domaine de fourrier avec la fonction *tensorflow.signal.fft* de **TensorFlow**, faire une multiplication puis repasser le signal dans le domaine temporel. Il y a cependant une différence entre l'intuition et l'implémentation, il aurait fallu dédié un sprint à cette fonction sans assurance que cela ne prenne pas plus de temps en cas de complication. Sachant que nous avons eu beaucoup de mal à faire fonctionner notre réseau de neurones et à obtenir un résultat nous avons préféré laisser de côté cette fonction et la mentionner dans le rapport pour une implémentation future.

La dernière étape pour valider ces fonctions a été d'écrire des tests unitaires. Ils sont présentés plus en détails dans la section 7.

La classe *Dataset* : Cette classe gère le chargement des différentes ressources audio ainsi que la mise en forme dans un format compatible avec la couche d'entrée du réseau de neurone. La classe est actuellement limitée au chargement de fichiers audio au format wav, il n'a pas été jugé utile d'intégrer d'autres formats pour le moment. Les fichiers audio sont partagés en deux catégories globales :

- train (dans le code source : x_train, y_train) Données utilisées lors de l'entraînement du réseau. Le paramètre "validation_split" de la fonction *fit(...)* de **Keras** permet d'utiliser une partie de ces données comme ensemble de validation. En utilisant un ensemble de validation nous pouvons détecter le moment où le réseau fait de l'overfitting, c'est-à-dire le moment où le réseau a trop de connaissance sur l'ensemble d'entraînement au point où il n'est plus capable de les abstraire pour les utiliser sur des données autres que celles de l'ensemble d'entraînement.

- test (dans le code source : x_test, y_test) Données utilisées suite à l'entraînement du réseau pour s'assurer le réseau obtenu est correct et émule convenablement l'effet désiré.

Nous avons eu beaucoup de mal à entraîner le réseau de neurones. Le signal audio est encodé en 16 bits, ce qui fait 65536 valeurs possibles pour encoder un échantillon. De ce fait il était impossible pour nous d'appliquer une technique de *one-hot encoding* (créer un vecteur de 65536 valeurs représentant les choix possibles pour le réseau) car la mémoire vive des cartes graphiques est limitée et malheureusement celle des cartes graphiques à notre disposition n'était pas suffisante pour utiliser cette méthode.

Les valeurs des échantillons lorsque nous chargeons un fichier audio vont de -32768 à 32767. Si nous reprenons le modèle de notre réseau de neurones (Figure 3), nous pouvons déduire les valeurs que retournera notre réseau. L'unité récurrente, un LSTM, a comme fonction d'activation par défaut une fonction tangente hyperbolique (*tanh*) qui retourne une valeur comprise entre -1 et 1

inclus. La couche de sortie, Dense, ne possède pas de fonction d'activation et se contente de faire une transformation affine de la valeur renvoyée par l'unité récurrente. Il a donc fallu que nous transformons les valeurs des données au moment de les mettre en forme pour l'entraînement du réseau de neurones. Pour ce faire nous divisons simplement les valeurs par la valeur absolue maximale possible en 16 bits, soit 32768, pour que toutes les valeurs soient comprises entre -1 et 1 inclus. Une fois que nous avons réalisé cette modification nous avons réussi à entraîner correctement le réseau. La Figure 6 montre l'évolution de la loss et de la val_loss (loss calculée sur l'ensemble de validation) au cours d'un entraînement sur une période de 100 époques.

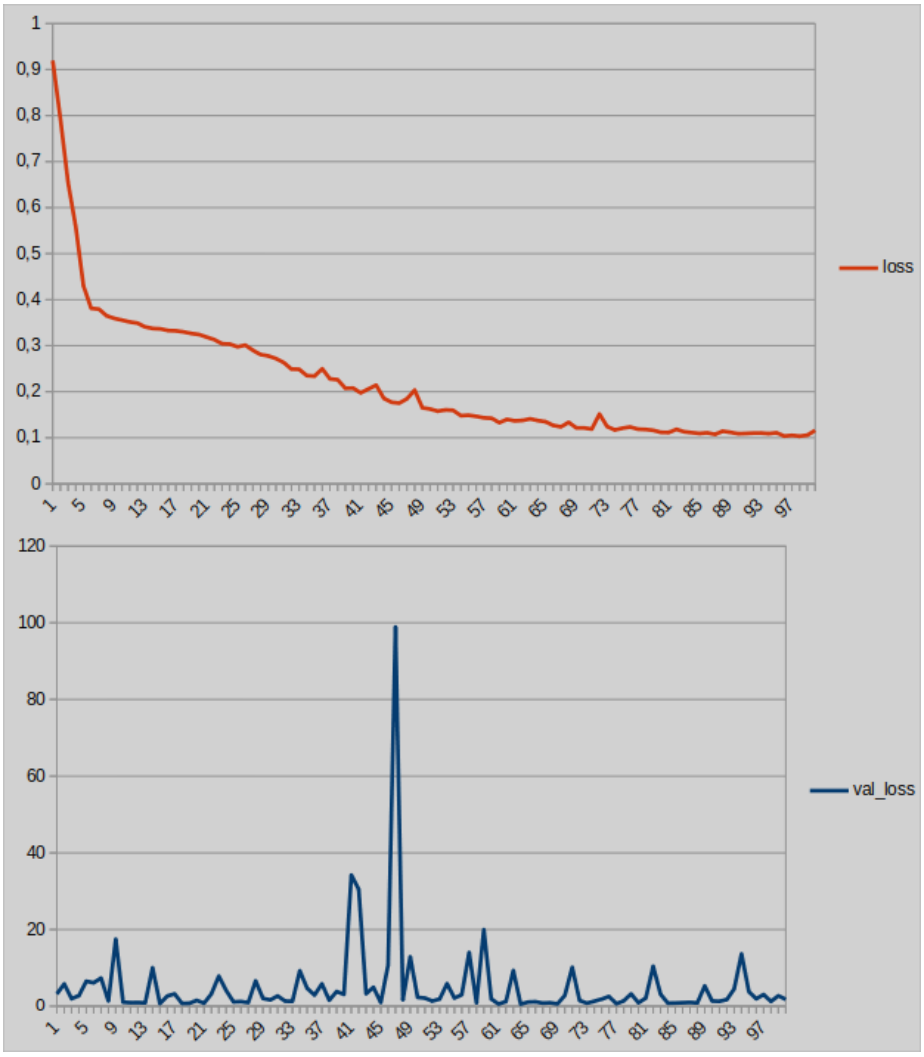

Figure 6: Évolution de la loss et de la val_loss sur une période de 100 époques. La couche récurrente LSTM dispose de 16 unités (hyperparamètre modifiable) et 90 secondes d'audio uniquement sont utilisées.

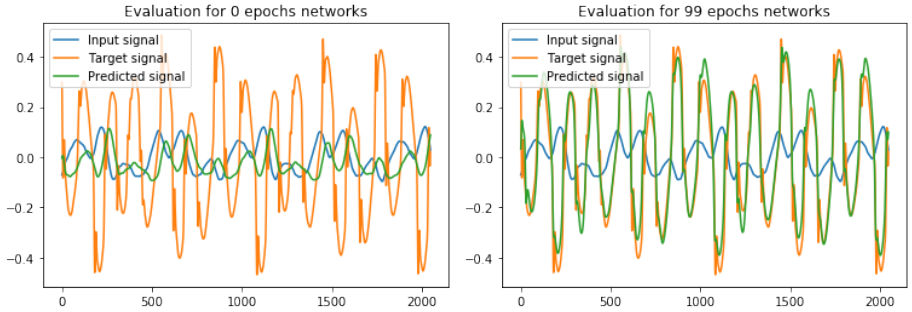

Figure 7: À gauche (1ère époque), nous pouvons voir que le signal prédit est très proche du signal passé en entrée alors que dans le graphique à droite (100ème époque), le signal prédit est beaucoup plus proche de celui attendu.

Boucle d'entraînement Dans nos premières itérations nous n'arrivions pas à entraîner le réseau de neurones à cause des erreurs mentionnées précédemment. Nous avons essayé alors d'implémenter notre propre boucle d'entraînement plutôt que d'utiliser la fonction *fit(...)* de **Keras**. Nos résultats ne se sont pas améliorés comme l'erreur ne provenait pas de cette partie du code cependant les résultats étaient similaires à ceux obtenus avec la méthode *fit(...)*. Afin d'améliorer la précision du réseau, il serait intéressant de reprendre cette partie du code, que nous avons laissé de côté par soucis de simplicité, afin d'avoir plus de contrôle sur la phase d'apprentissage, notamment lors de l'initialisation des poids de la couche récurrente au début d'une époque. Un clic est audible dans les résultats que nous obtenons, nous pensons que celui-ci est dû à un problème de phase qui serait corrigeable justement en ayant plus de contrôle sur les fonctionnements internes de la phase d'apprentissage. Actuellement, même avec un nombre assez réduit d'itération nous arrivons à voir que le réseau reproduit rapidement le comportement attendu de sa part (Figures 7).

6.2 Implémentation de l'interface utilisateur

Afin d'utiliser le modèle que devions créer, nous avons utilisé une bibliothèque C++ nommée *Frugally Deep*[4] qui permet d'utiliser un modèle créé avec *Keras* ou *TensorFlow* et sauvegardé au format .h5 au sein d'un programme en C++, et donc de notre plugin implémenté avec **JUCE**[5].

JUCE est un framework C++ open-source utilisé pour la création de plugins et d'interfaces utilisateur. Il permet de créer notamment des applications GUI, OpenGL, console, mais aussi des applications et des plugins audio. Cette dernière option justifie notre choix de nous orienter vers une application sous ce framework.

Le code est composé des trois classes suivantes :

- **PluginEditor** : Cette classe représente l'affichage de l'utilisateur. Elle a pour but de créer, placer et mettre à jour la fenêtre et les éléments qui s'y trouvent.
- **PluginProcessor** : Cette classe a pour but d'effectuer les calculs en

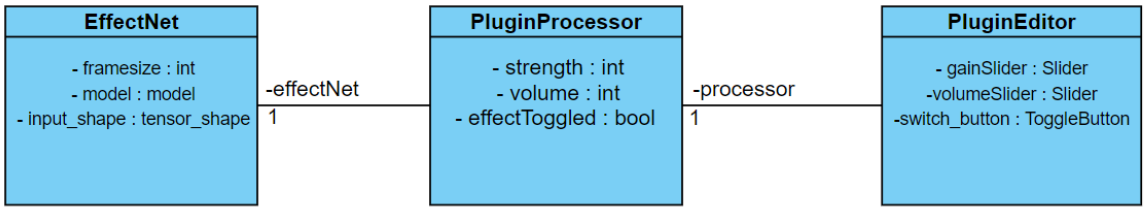

Figure 8: Diagramme de classe simplifié du plugin

fonction de l'interface utilisateur et de l'effet chargé par frugally-deep.

- **EffectNet** : Cette classe est la seule qui gère frugally-deep. Son but est de récupérer les données audio passées en entrée et d'y appliquer l'effet si le bouton de l'interface est activé.

Afin de gérer le volume de sortie de l'application, nous utilisons un potentiomètre de volume présenté dans la partie architecture. Le volume est calculé pendant la lecture des entrées audio, buffer par buffer.Un buffer est tout simplement un tableau de valeurs (flottantes dans notre cas) composé des valeurs lues dans l'entrée audio. Un buffer doit être assez petit (2,048 valeurs en général, ce qui fait 23.2ms pour un signal échantillonné à 44,100Hz) dans le cas d'une application en temps réel, car il faut être capable de traiter les valeurs et de les écrire dans la sortie dans un temps suffisant pour que l'oreille humaine ne sente pas le décalage (environ 20ms). La taille du buffer peut être changée nativement dans les options de l'application grâce aux options proposées par **JUCE**.

Comme précisés dans la section implémentation du modèle, nous avons passé une majeure partie du projet à essayer de créer un modèle du réseau de neurones satisfaisant. Cependant, ce temps perdu ne nous a pas permis de créer une interface fonctionnelle pour l'application d'un modèle. Ainsi, le potentiomètre de gain n'est pas utilisé dans notre programme. De plus, lorsque l'effet est activé, le code créé avec *frugally-deep* ne respecte pas la contrainte de temps-réel. Nous ne nous sommes pas penché en détail sur ce problème puisque, sans effet à appliquer, il est inutile de s'attarder sur une contrainte de temps-réel. Toutefois, notre code peut servir de base pour un projet futur. En effet, la classe EffectNet peut être appliquée à n'importe quel effet créé par un réseau neuronal. De plus, le potentiomètre de gain peut appliquer n'importe quelle transformation pour lequel le réseau serait entraîné.

7 Tests

Un point important de ce projet est de pouvoir mettre en place une méthode de travail SCRUM au sein de son équipe tout en respectant certains principes de développement. Un de ces principes consiste à tester le code quasi-perpétuellement afin de s'assurer du bon fonctionnement des différents parties du code au fur et à mesure que l'on ajoute de nouvelles fonctionnalités. Ainsi, une fonction écrite au début du projet sera testée à chaque nouvel ajout de code et sera ainsi assurée de fonctionner correctement.

Nous verrons dans cette partie les détails d'implémentation de la chaîne d'intégration continue ainsi que les différents tests unitaires.

7.1 Intégration continue

Nous avons cherché quels outils nous avions à notre disposition afin d'automatiser l'exécution des tests et nous avons décidé d'utiliser le système d'intégration continue fourni par GitLab (GitLab CI) où nous avons hébergé le dépôt de ce projet[5].

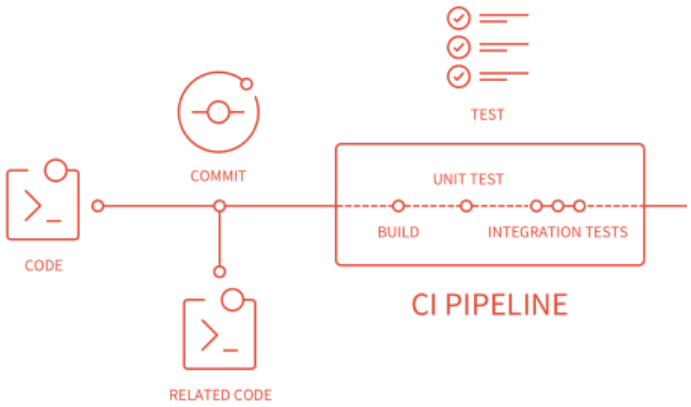

Figure 9: Intégration continue de GitLab.

Comme le montre la Figure 9, ce système fonctionne comme une chaîne où chaque modification apportée sur le serveur du dépôt est testée selon des règles données par les développeurs. Ces règles sont données dans un fichier **.gitlab-ci.yml**(voir l'exemple en annexe) qui doit se situer à la racine du dépôt. La présence de ce fichier permet au serveur d'exécuter les tests d'intégration continue de manière autonome. Voici quelques informations supplémentaires afin de comprendre plus en détail le fonctionnement de l'intégration continue dans notre projet :

- Les instructions sont structurées en tâches (terme GitLab: jobs) et étapes (terme GitLab: stages). Par défaut le pipeline dispose de trois étapes *build*, *test* et *deploy*. Il est possible de définir dans *stages:* des étapes supplémentaires de son choix. Dans notre projet, nous n'avons pas mis en place l'étape *deploy*, mais nous avons utilisé une étape préliminaire *.pre*. Les étapes s'exécutent dans l'ordre dans lequel elles sont définies dans *stages:* à l'exception des étapes *.pre* et *.post* qui s'exécutent respectivement avant et après les autres étapes définies.
- Toutes les tâches d'une étape s'exécutent en parallèle dans des images Docker isolées. Une tâche peut accéder aux ressources mises en cache par les tâches d'une étape antérieure à l'aide de la commande *artifacts: paths:*. Les caches accessibles sont définies dans *cache: paths:*.

[5] `https://gitlab.com/gweil/pfe`

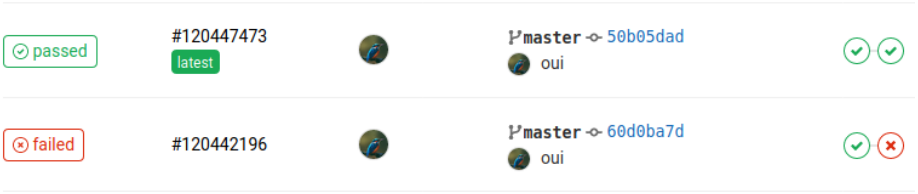

Figure 10: Exemple d'un pipeline d'intégration continue réussi (en haut) et d'un pipeline qui rate à cause d'une tâche (en bas).

- L'étape *.pre* nous permet de télécharger les bibliothèques python dont nous avons besoin lors des étapes suivantes et de les mettre en cache pour ne pas avoir à les télécharger de nouveau par la suite. Les bibliothèques sont téléchargées dans un cache du pipeline grâce à la variable globale PIP_CACHE_DIR définie dans *variables:*.
- Les tâches d'une même étape sont exécutées en parallèles et les tâches des étapes ultérieures sont mises en attente. Si une tâche de l'étape rate alors le pipeline s'arrête et les étapes suivantes ne sont pas déclenchées (voir Figure 10).
- Chaque tâche est exécutée dans un environnement Docker isolé. Il est toutefois possible d'utiliser un cache entre les tâches d'étapes différentes (deux tâches d'une même étape ne peuvent pas se partager leurs ressources).
- Au début de chaque tâche, le cache est ouvert et ses ressources sont mises à disposition puis il est mis à jour à la fin de la tâche si jamais la tâche doit mettre en cache des données supplémentaires.
- Les données mises en cache, appelées artefacts (terme GitLab: artifacts), sont téléchargeables une fois la tâche achevée. Cela nous permet de récupérer une trace dans des fichiers texte, csv, ou autre, sur les tests effectués (voir la sortie de pluginval en annexe).

7.2 Tests unitaires

7.2.1 Fonction de coût:

Cette partie de code teste la validité de la fonction de coût définie dans l'équation. Pour valider l'implémentation, il a fallu au préalable calculer à la main le coût pour des signaux prédéfinis. Trois tests simples sont donc mis en place :

Test 1 *test_loss_equal*: Le but de ce test est de vérifier que la fonction de coût renvoie bien $l_{th} = 0$ quand un signal est testé avec lui-même. En effet, cette fonction a pour but initial de quantifier la distance entre deux signaux. Pour ce faire, un signal sinusoïdal est créer grâce à l'équation $X = sin(2 * \pi * f)$ avec f la fréquence (440 Hz). Le résultat obtenu est $l_c = 0$, ce qui prouve que dans ce cas la fonction de coût fonctionne.

La fonction de coût passe donc les tests unitaires sur des signaux identiques conformément à un développement à la main sur papier de la formule.

Test 2 *test_loss_double*: Le but de ce test est de vérifier la validité de la fonction de coût pour deux signaux espacé d'un multiple de deux. Les deux signaux sont des sinus de fréquence fondamentale $f_0 = 440Hz$ et ça première harmonique $f_1 = 880Hz$. L'objectif est donc de comparer la valeur retournée par la fonction de coût avec la valeur théorique calculée à la main. À savoir, $l_{th} = 2$.

Le résultat obtenu est $l_c = 2$, ce qui valide la fonction de coût dans ce cas aussi.
La fonction de coût passe donc le test unitaire sur des signaux différents.

Test 3 *test_loss_rnd*: Ce test est similaire au test 1, nous testons deux signaux identiques mais avec des valeurs aléatoires. Le résultat attendu est le même que celui du test 1, c'est-à-dire $l_{th} = 0$.
Sur un signal aléatoire, la fonction de coût revoit bien $l_c = 0$ reste valide ce qui prouve ça validé dans ce cas précis aussi.

Dans ce cas, aussi, la fonction de coût passe le test unitaire pour deux signaux identiques mais aléatoire.

Conclusion et critiques : Les tests ont montré à travers des exemples simples que la fonction de coût est fonctionnelle. Il faut néanmoins rappeler que les trois tests effectués ont été fait sur des signaux très simples et surtout loin de la réalité. Il n'est cependant pas possible de vérifier sur des signaux plus complexes, car il est très difficile de faire les calcules théorique sur de tels signaux. Pour vérifier la validité de la fonction en condition réelle, une idée est de tester pour chaque signal d'entrer avec lui-même.

7.2.2 Test de la classe Dataset

La classe *Dataset* permet de charger et de formater les signaux pour le réseau. Cette partie est donc très importante et il est nécessaire de vérifier que les données sont bien formatées.

Test 4 *testCutDataEqualOne* : L'objectif de cette fonction est de tester le comportement de la fonction *cutData* dans le cas où uniquement un jeu de données d'entraînement est demandé. Cela se fait aux travers de deux variables *ratio_train* et *ratio_validation* (voir partie implémentation pour plus de détails).

Dans ce cas précis, la fonction revoie un tableau train avec les donnés est des tableaux validation et test vide. Le test est effectué au niveau des tableaux validation et test et est considéré comme réussit si et seulement si la longueur de ces deux tableaux est égale à zéro.

Les résultats montrent que la fonction *cutData* se comporte exactement comme prévue dans ce cas de figure. Le test est de donc validé.

Test 5 *testCutDataGreaterOne* : Le but de cette fonction est de tester le comportement de la fonction *cutData* dans le cas où les valeurs de *ratio_train* et *validation_ratio* sont aberrantes. Ce test à notamment été mis place à posteriori vérifier la correction d'un bug.

La fonction *cutData* test avec une série d'*assert* que les ratios sont correctes. C'est-à-dire re que les deux ratios sont plus petit que 1.0 et que leur somme est aussi plus petite que 1.0. Nous testons donc que les la fonction *cutData* lève bien une exception dans ce cas précis.

Les résultats sont corrects, une exception est bien levée et donc le test est validé.

Test 6 *testReshape* : Cette fonction vérifie la validité de la fonction *Reshape*. On créer un tableau 1D de taille 441000 (10 seconde) puis nous test la fonction sur ce tableau avec $frame\_size = 2048$.

Dans ce cas de figure,

$$\left\lfloor \frac{fs * duration}{frame\_size} \right\rfloor = \left\lfloor \frac{441000}{2048} \right\rfloor = 215 \tag{1}$$

On test donc les shapes du tableau précédemment créé avec $(215, 2048, 1)$. Et si elles sont identiques, le test est considéré comme réussit.

Les tests *testReshape* passe sans erreur ce qui montre que la fonction formate *Reshape* correctement les données.

Autres tests : D'autres tests moins formels ont aussi été faits de manière empirique tout au long du programme pour vérifier que les données ont le format attendu a chaque instant. C'est-à-dire l'affichage dans la console de la shape des données après chaque manipulation.

Conclusion et critiques : L'ouverture des fichiers n'a pas été testée. Nous utilisons une bibliothèque standard déjà validé donc nous n'avons jugé important de tester sa validité. De-plus le chargement des fichiers se fais qu'une seule fois au lancement du programme. Les seul motifs d'erreurs possibles sont l'absence de fichiers sur le disque ou des fichiers corrompu. Nous laissons la gestion de ces erreurs à la fonction *wavfile.read()* de la bibliothèque *scipy*.

7.3 Performance

Dans le but de tester les performances de *frugally-deep*, nous avons lancé un test sur le temps de traitement des données sur différentes tailles de buffers avec une comparaison effet activé/désactivé.
Ces tests ont été réalisé sur un CPU Intel(R) Core(TM) i5-4210M @ 2.60GHz.
Afin de calculer les meilleures approximations possibles, ces tests ont été calculés avec une moyenne faite sur mille buffers de chaque taille.
Dans ce graphique, nous pouvons remarquer qu'une fois l'effet activé, le temps nécessaire pour traiter un buffer est largement supérieur au temps de remplissage de ce dernier (représenté par la courbe jaune).

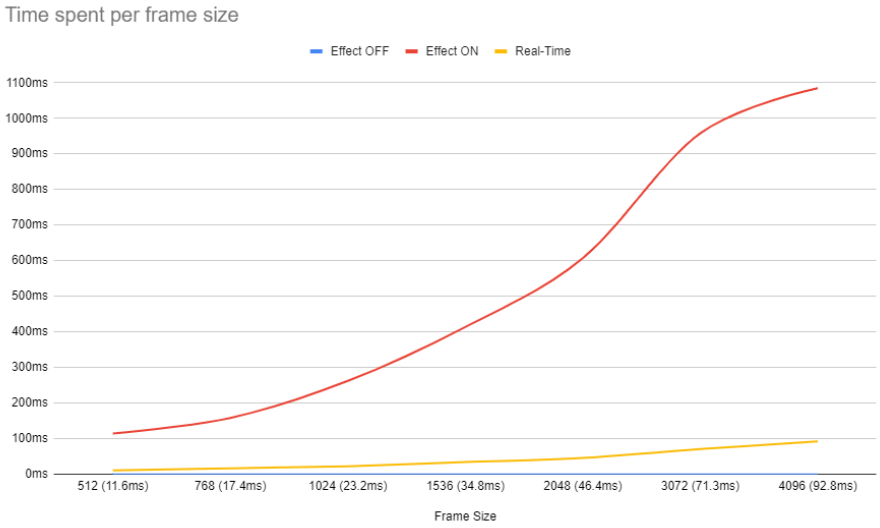

Figure 11: Graphique des performances obtenus sur l'utilisation du modèle dans le plugin.

Dans le papier de Wright [3], nous pouvons lire qu'il implémente son réseau en C++ en utilisant *WaveNet* au lieu de *Frugally-Deep*. Lorsque l'on compare ses performances, on peut observer qu'il arrive aisément à jouer sur ses paramètres en restant «assez rapide» pour du temps réel comme il l'explique dans son papier dans la partie *2.3 Real-time implementation*, et ce avec un CPU Intel(R) Core(TM) i5 @ 2.80GHz.

Comme expliqué dans la partie implémentation, nous ne nous sommes pas attardé sur les performances de l'interface utilisateur sans modèle efficace. De plus, suite à l'évolution de notre projet, notre choix initial d'implémenter le modèle avec frugally-deep devient discutable car nous utilisons *TensorFlow* et non plus *Keras*. Nous pourrions ainsi utiliser la bibliothèque de *TensorFlow* C/C++, qui nous donnerait certainement de meilleures performances.

8 Conclusion

L'idée de ce projet était de simuler un effet de pédale de guitare grâce à l'apprentissage profond.
Nous avons à l'origine fait des choix d'architecture qui semblaient être la bonne approche théorique afin de résoudre cette problématique :

- **Keras** comme bibliothèque pour l'apprentissage car, comme le projet est finalement assez court en temps, nous avons préféré utiliser la bibliothèque que nous connaissions déjà plutôt qu'une bibliothèque nouvelle comme PyTorch[6] par exemple.
- **Frugally-Deep** car cette bibliothèque possède une facilité d'utilisation pour le chargement des modèles Keras.

- **JUCE** comme framework de développement pour l'interface utilisateur puisqu'il permet nativement de gérer les entrées/sorties audio et dispose de facilités dans le code pour créer une interface graphique simple.

Nous avons donc pour ce projet fait plusieurs tâches permettant sa réalisation. Premièrement, nous avons créé un jeu de données à partir d'une base téléchargeable de jeu de guitare et de basse sans amplification. Nous y avons appliqué l'effet de distorsion à reproduire avec différents niveaux de gain afin d'obtenir nos données d'entraînement.

Nous avons ensuite implémenté le réseau de neurones en Python grâce à *Keras* et *TensorFlow* en suivant le papier de Wright [1].
Puis, nous avons créé le plugin C++ sous **JUCE**. Tout d'abord, nous avons créé une maquette possédant deux potentiomètres et un bouton. Cette interface utilisateur doit pouvoir appliquer l'effet créé à partir du modèle. Cependant, comme nous l'avons longuement expliqué plus tôt, nous n'avons pas eu le temps de nous pencher sur l'intégration du modèle dans le plugin, et donc de travailler sur son utilisation en temps réel.

Tout le travail prévu n'a cependant pas été fait, le réseau entraîné n'est actuellement pas intégré au plugin. Mais il est fonctionnel, dans le futur, le réglage des différents hyper-paramètres devra être fait afin d'avoir un résultat convaincant. Nous estimons à deux le nombre d'itérations qu'il aurait fallu en plus pour finir correctement le projet. Une pour l'intégrer au plugin, une autre pour régler les derniers problèmes et tuner les hyper-paramètres.

9 Annexes

Explication détaillée du modèle

```
def defineModel(self):
    self.model = keras.Sequential()                                          #1
    self.model.add(keras.layers.LSTM(self.lstm_unit, input_shape=(2048, 1),
                                     return_sequences=True))                 #2
    self.model.add(keras.layers.TimeDistributed(keras.layers.Dense(1)))      #3

    self.model.compile(loss=custom_loss,
                       optimizer=keras.optimizers.Adam(learning_rate=0.0005),
                       metrics=None)                                         #4
```

1 Cette couche correspond à couche récurrente du réseau, le premier paramètre correspond au nombre d'unité de récurrences. Le second paramètre permet de récupérer la séquences entière c'est à dire un tenseur de taille *frame_ size*.

2 Cette couche correspond à la couche *Dense*, Keras permet de distribuer dans le temps une couche avec la fonction *TimeDistributed*. Cela veux dire que même si le *Dense* a un seul neurone de sortie, il va retourner un tenseur de taille *frame_ size*.

3 Cette étape permet de créer le modèle.

4 Le modèle est compiler avec la fonction de coût définie plus haut. *custom_loss* correspond à la somme de ε_{ESR} et de ε_{DC}.

5 La méthode d'optimisation utiliser est *Adam*, avec un taux d'apprentissage fixé à $1e-5$ (comme dans [1]. Même si d'autre taux d'apprentissage ont pu être testé).

Structure simplifiée d'un fichier .gitlab-ci.yml:

```
default:
    image: < image docker >

cache: &global_cache
    paths:
        < chemins vers les caches >

# Variable globale pour que les bibliothèques
# installées avec pip se trouvent dans un cache
# réutilisable par les différentes tâches
variables:
    PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

stages:
    - .pre
    - build
    - test

< NOM DE LA TÂCHE >:
    image: < image docker >
    stage: < .pre / build / test >
    before-script:
        < instructions >
    script:
        < instructions >
    artifacts:
        paths:
            < fichiers à conserver dans le
            cache pour des tests utlérieurs >
```

Sortie de pluginval:

```
Random seed: 0x11ec91a
Validation started: 24 Feb 2020 9:38:23am

Strictness level: 5
-------------------------------------------------------------
Starting test: pluginval / Scan for known types: /builds/gweil/test/Plugin/Builds/LinuxMakefile/build/...
Num types found: 1

Testing plugin: VST-Plugin-ed794bec-50626379
yourcompany: Plugin v1.0.0.0
All tests completed successfully
-------------------------------------------------------------
Starting test: pluginval / Open plugin (cold)...

Time taken to open plugin (cold): 197 ms
All tests completed successfully
-------------------------------------------------------------
Starting test: pluginval / Open plugin (warm)...

Time taken to open plugin (warm): 15 ms
Running tests 1 times
All tests completed successfully
-------------------------------------------------------------
Starting test: pluginval / Plugin info...
```

```
Plugin name: Plugin
Alternative names: Plugin
SupportsDoublePrecision: no
Reported latency: 0
Reported taillength: 0

Time taken to run test: 0
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Editor...

Time taken to open editor (cold): 469 ms
Time taken to open editor (warm): 203 ms

Time taken to run test: 824 ms
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Open editor whilst processing...

Time taken to run test: 217 ms
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Audio processing...
Testing with sample rate [44100] and block size [64]
Testing with sample rate [44100] and block size [128]
Testing with sample rate [44100] and block size [256]
Testing with sample rate [44100] and block size [512]
Testing with sample rate [44100] and block size [1024]
Testing with sample rate [48000] and block size [64]
Testing with sample rate [48000] and block size [128]
Testing with sample rate [48000] and block size [256]
Testing with sample rate [48000] and block size [512]
Testing with sample rate [48000] and block size [1024]
Testing with sample rate [96000] and block size [64]
Testing with sample rate [96000] and block size [128]
Testing with sample rate [96000] and block size [256]
Testing with sample rate [96000] and block size [512]
Testing with sample rate [96000] and block size [1024]

Time taken to run test: 1 ms
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Plugin state...

Time taken to run test: 0
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Automation...
Testing with sample rate [44100] and block size [64] and sub-block size [32]
Testing with sample rate [44100] and block size [128] and sub-block size [32]
Testing with sample rate [44100] and block size [256] and sub-block size [32]
Testing with sample rate [44100] and block size [512] and sub-block size [32]
Testing with sample rate [44100] and block size [1024] and sub-block size [32]
Testing with sample rate [48000] and block size [64] and sub-block size [32]
Testing with sample rate [48000] and block size [128] and sub-block size [32]
Testing with sample rate [48000] and block size [256] and sub-block size [32]
Testing with sample rate [48000] and block size [512] and sub-block size [32]
Testing with sample rate [48000] and block size [1024] and sub-block size [32]
Testing with sample rate [96000] and block size [64] and sub-block size [32]
Testing with sample rate [96000] and block size [128] and sub-block size [32]
Testing with sample rate [96000] and block size [256] and sub-block size [32]
Testing with sample rate [96000] and block size [512] and sub-block size [32]
Testing with sample rate [96000] and block size [1024] and sub-block size [32]

Time taken to run test: 1 ms
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Editor Automation...

Time taken to run test: 1 sec
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Automatable Parameters...
```

```
Time taken to run test: 0
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Basic bus...
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Listing available buses...
Inputs:
Named layouts: Mono, Stereo
Discrete layouts: Discrete #1, Discrete #2
Outputs:
Named layouts: Mono, Stereo
Discrete layouts: Discrete #1, Discrete #2
Main bus num input channels: 2
Main bus num output channels: 2
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Enabling all buses...
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Disabling non-main busses...
All tests completed successfully
----------------------------------------------------------------
Starting test: pluginval / Restoring default layout...
Main bus num input channels: 2
Main bus num output channels: 2

Time taken to run test: 0

Time taken to run all tests: 2 secs
All tests completed successfully
```

References

[1] Alec Wright, Eero-Pekka Damskägg, Vesa Välimäki, et al. Real-time black-box modelling with recurrent neural networks. *accompanying webpage, available online at: http://research. spa. aalto. fi/publications/papers/dafx19-rnn/, Accessed*, pages 18–06, 2019.

[2] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[3] Alec Wright, Eero-Pekka Damskägg, Lauri Juvela, and Vesa Välimäki. Real-time guitar amplifier emulation with deep learning. *Applied Sciences*, 10(3):766, 2020.

[4] Tobias Hermann. Frugally-deep 0.12.1. `https://github.com/Dobiasd/frugally-deep`.

[5] ROLI. Juce 5.4.7. `https://juce.com/`.

[6] Pytorch. `https://pytorch.org/`.