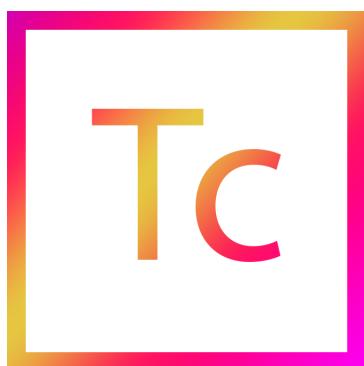


# Rapport Final

## Totalisateur



Pierre CHOUTEAU

Encadré par Sylvain "Syd" Reynal

# Contents

<b>1</b>	<b>Introduction - Concept</b>	<b>3</b>
1.1	Objectif . . . . .	3
<b>2</b>	<b>Etat de l'art</b>	<b>4</b>
2.1	NSynth : Neural Audio Synthesis . . . . .	4
2.2	ToneTransfer   DDSP . . . . .	4
2.3	Projet Neural-Pi . . . . .	5
<b>3</b>	<b>Réseau de Neurones</b>	<b>6</b>
3.1	Dataset . . . . .	6
3.2	Création du Modèle . . . . .	7
3.3	Résultats . . . . .	7
<b>4</b>	<b>Premier test sur une tâche simple (En collaboration avec Elisa DELHOMME)</b>	<b>9</b>
4.1	Modèle Test . . . . .	9
4.2	Implémentation sur STM32 - CubeAI x CubeIDE . . . . .	9
4.3	Implémentation du modèle LSTM . . . . .	11
	<b>Conclusion</b>	<b>12</b>

# Chapter 1

## Introduction - Concept

Il vous est déjà arrivé de vouloir commencer un instrument de musique et que, au-delà de la difficulté de prise en main, vous sonniez très mal ?

Vous ne savez pas comment régler celui-ci car il y a trop de boutons ? Et, même après plusieurs mois, lorsque vous voulez sonner comme votre artiste préféré vous n'y arrivez pas ? Vous ne comprenez toujours pas comment régler votre instrument, quel potentiomètre bouger pour quel effet ou même sur quel bouton de votre synthétiseur appuyer ?

Tout cela met à mal votre motivation et les efforts des nouveaux musiciens. Et en allant plus loin, cela fait perdre du temps aux musiciens de régler toutes leurs pédales et amplis comme il faut. Avec l'équipe du ToneCrafter, nous sommes convaincus que si vous sonnez bien, vous serez passionnés et passerez beaucoup plus de temps sur vos instruments. C'est pourquoi nous avons pensé à un tout nouvel objet prenant la forme d'une pédale, et permettant de reproduire n'importe quel son/effet, simplement en l'écoulant, le ToneCrafter.

Grâce à lui, vous pourrez jouer de votre instrument avec le même son que vos artistes préférés.

Après avoir essayé plusieurs approches pour la reproduction du son souhaité:

- L'implémentation d'un VAE
- La création d'un réseau de neurones (CNN)
- Et une approche plus mathématique consistant à faire une recherche paramétrique.

Nous nous sommes rendu compte que ce n'était pas suffisant pour avoir des résultats réellement concluants.

### 1.1 Objectif

Cette année, j'ai donc décidé de changer de direction. Je me suis fixé comme objectif :

- Comprendre un modèle pré-construit et fonctionnel [6][8] permettant la reproduction parfaite d'un effet.
- Implémentation de ce modèle d'intelligence artificiel sur une STM32 grâce à CubeAI. [9]

# Chapter 2

## Etat de l'art

Afin de pouvoir avoir une vision plus globale des différentes technologies utilisées dans ce domaine ou même de projets existants qui s'approchent du ToneCrafter, j'ai commencé par effectuer un état de l'art.

En faisant ces recherches je suis retombé sur les mêmes recherches que l'année dernière comme par exemple:

### *Le Projet Magenta*

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

Cela peut par exemple donné des projets comme ceux ci-dessous :

### 2.1 NSynth : Neural Audio Synthesis

Le NSynth est un synthétiseur neuronal. A la différence d'un synthétiseur classique qui génère des sons à partir d'un VCO, le NSynth utilise lui, utilise un réseau de neurones entraîné avec un dataset créé pour l'occasion.

Ce qui nous avait plu et qui m'a encore fasciné en retrouvant ce projet est qu'il s'agit d'un contrôleur hardware créé spécifiquement pour utiliser l'algorithme. Finalement ce projet n'est pas si éloigné de ce qu'est le ToneCrafter. De plus, la forme est vraiment jolie et c'est exactement ce à quoi nous voulions que le projet ressemble.



Figure 2.1: Prototype du NSynth Super

### 2.2 ToneTransfer | DDSP

Un autre projet que nous avions découvert avec le projet Magenta était un algorithme de ToneTransfer. L'idée est simple, pouvoir jouer de n'importe quel instrument sans même savoir s'en servir ou encore pouvoir transformer des sons du quotidien en sons d'instruments de musique.

Avec DDSP, tout ça est devenu possible. *Exemple*.

C'est en partie pour cela que ce projet avait retenu notre attention. Il s'apparente beaucoup à l'idée du ToneCrafter: reproduire le son d'un certain type d'effet à partir d'un signal de base. La seule différence, c'est qu'ici, à la place de s'approcher d'un effet, comme une distorsion, une reverb ou même un chorus, leur algorithme de machine learning permet de modifier le signal d'origine afin d'obtenir la sonorité de l'instrument que l'on souhaite.

Par exemple, il est possible d'obtenir un son de violon à partir d'une voix :

**Play : Chant**

**Play : Violon**

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

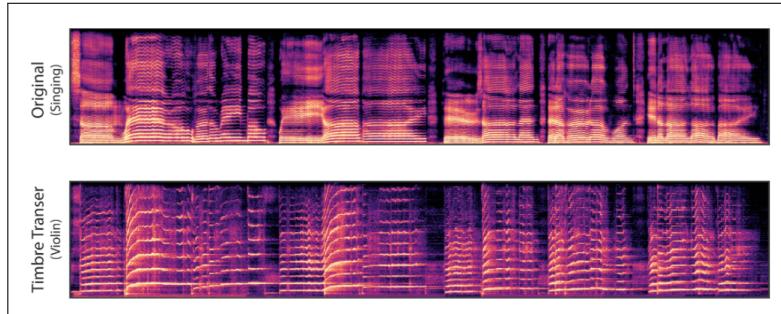


Figure 2.2: Timbre Transfert : Violon -> Flûte

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

## 2.3 Projet Neural-Pi

Et enfin, j'ai découvert un nouveau projet qui aujourd'hui est celui qui s'apparente le plus au ToneCrafter:

### *Le Projet Neural Pi [7]*

C'est une pédale de guitare utilisant des réseaux neuronaux pour émuler de vrais amplis et pédales d'effets sur une Raspberry Pi 4.

Le logiciel utilisé est un plugin VST3 construit avec JUCE, qui peut être exécuté comme un plugin audio normal ou compilé pour fonctionner sur Raspberry Pi 4 avec Elk Audio OS. Cette pédale comprend également des commandes de sélection de modèle, d'égalisation et de gain/volume.

La seule différence que l'on peut noter entre ce projet et le ToneCrafter, c'est qu'il utilise un logiciel pour faire fonctionner le réseau de neurones. Le réseau n'est pas directement implémenté dans la carte. Cette différence à l'air moindre au premier abord, mais si on réussit à implémenter le réseau de neurones directement sur la carte, on devrait pouvoir gagner en performances.



Figure 2.3: Neural Networks for Real-Time Audio: Raspberry-Pi Guitar Pedal

Pour en savoir plus sur ce projet, le créateur: Keith Bloemer, a écrit un article sur Toward Data Science expliquant tout le processus de création.

# Chapter 3

## Réseau de Neurones

Comme expliqué au début de ce rapport, le premier objectif était la création d'un réseau de neurones permettant de reproduire le son de n'importe quel effet.

Pour cela, je me suis basé sur le modèle venant de l'article de Keith Bloemer [8], qui lui-même s'appuie sur le papier d'Alec Wright : "Real-Time Guitar Amplifier Emulation with Deep Learning".[6]

Le modèle utilisé dans ces deux articles est à peu de chose près le même. Ils se basent principalement sur l'utilisation d'une architecture de réseau neuronal récurrent (RNN) appelée "*LSTM*".

### Mais qu'est-ce qu'un LSTM ?

Les modèles LSTM, pour "Long Short-Term Memory" ont été développés dans le milieu des années 90. Contrairement aux réseaux de neurones à anticipation standard (comme "*WaveNet*"), les LSTM possèdent des états récurrent qui sont mis à jour chaque fois que les données circulent dans le réseau ("*Stateful LSTM*"). Grâce à cela, le réseau peut effectuer la prédiction d'une donnée à l'instant T en ayant des informations sur son passé, c'est un réseau qui possède une "mémoire". C'est pour cela que les *LSTM* sont souvent applicables à des tâches telles que de la reconnaissance vocale. Ce sont également des réseaux qui sont très adaptés à la réalisation de prédictions basées sur des données de séries chronologiques, ce qui est exactement ce que l'on veut faire.

Dans l'article, Keith Bloemer utilise un "*StateLess LSTM*", ce qui signifie que la mémoire du réseau est réinitialisée à chaque batch. Et comme il utilise des batchs contenant un seul élément, le réseau devient essentiellement un réseau à anticipation, car il n'y a pas d'état récurrent. C'est ce qui fait que son modèle est très intéressant. De plus, il a été démontré qu'un modèle "*StateLess LSTM*" est efficace pour du traitement audio, car en réduisant les états internes à 0, la complexité du réseau est considérablement réduite, et la vitesse de traitement est donc améliorée. Ce qui n'est pas négligeable lorsque l'on veut obtenir du temps réel.

### 3.1 Dataset

Aujourd'hui les datasets qui existent pour de l'audio ne correspondent pas exactement au dataset souhaité pour ce genre de réseau de neurones. L'objectif étant de reproduire précisément un effet, il faut connaître exactement celui du dataset utilisé... Ce qui n'est pas encore le cas.

C'est pourquoi, comme cela est fait dans l'article, le dataset utilisé pour ce réseau de neurones est créé de toutes pièces. Il est constitué de deux audios :

- Un audio de 3min où l'on peut entendre une guitare fender telecaster jouée plusieurs riffs de musique connue, sans effet.  
C'est notre audio "**Original**" - **Clean**.
- Le même audio est dans un second temps envoyé dans un amplificateur à lampe de la marque Blackstar (Le Blackstar HT40), paramétré avec des EQs neutres et avec 25% d'overdrive. Cet audio est celui que l'on cherche à reproduire. Plus particulièrement, l'effet que l'on cherche à reproduire.  
C'est notre "**Target**" - **Effect**.

On peut écouter ces deux audios ci-dessous :

Play : Original

Play : Cible

Pour résumer, si le modèle apprend correctement celui-ci pourra alors reproduire l'overdrive d'un amplificateur blackstar, réglé à 25% et avec des EQs neutre.

## 3.2 Crédation du Modèle

Pour la création du modèle, j'ai commencé par reproduire la même architecture que dans l'article. C'est-à-dire :

- 2 couches de convolution 1D (*Conv1D*) permettant dans un premier temps d'extraire les caractéristiques importantes de l'audio. Ces 2 couches servent également à réduire la quantité de données transmises à la couche LSTM
- 1 couche *LSTM*, qui permet comme expliqué plus haut, de faire une prédiction grâce à la connaissance du passé.
- 1 couche *Dense (Fully Connected)*, qui permet la prédiction finale de notre échantillon

On obtient alors l'architecture ci-dessous :

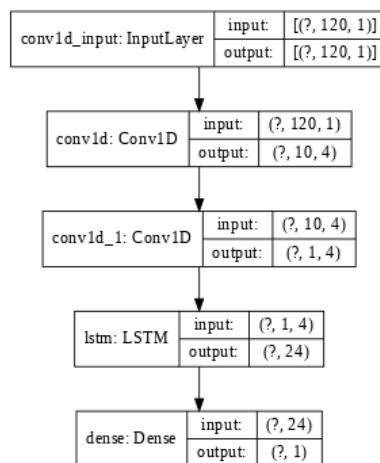


Figure 3.1: Stateless LSTM

Une fois l'architecture du réseau définie, il reste à définir la Loss Function et l'Optimizer, qui permettent d'indiquer au réseau de neurones le problème qu'il doit résoudre et comment il doit le résoudre. Dans notre cas, comme le problème à résoudre est un problème de régression linéaire, le choix a été vite effectué.

Comme dans l'article, la loss function utilisée est une **MSE - Mean Square Error**, qui permet de mesurer la disparité entre la valeur réelle de la target et la valeur prédictive par le modèle.

Et Pour l'Optimizer, c'est l'**Optimizer ADAM** qui a été choisi.

## 3.3 Résultats

Après avoir lancé l'apprentissage du réseau sur 50 époques, on vérifie dans un premier que le modèle apprend correctement et qu'il n'y a pas présence d'overfitting ou d'underfitting :

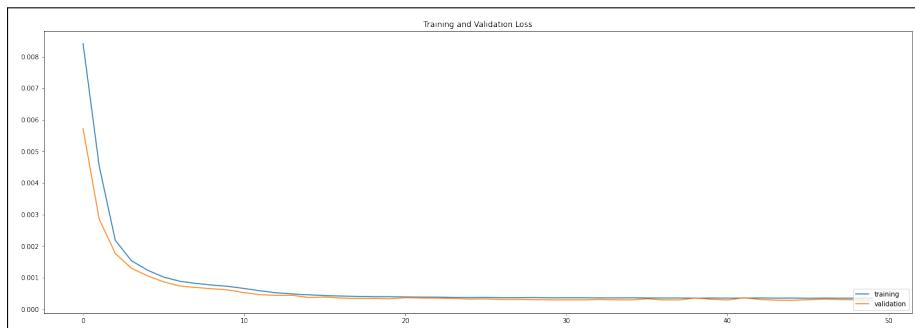


Figure 3.2: Loss de training et de validation correspondant à l'entraînement du réseau

On remarque grâce à ces courbes que :

- Le réseau de neurones apprend correctement (la loss décroît jusqu'à une valeur assez faible de convergence)
- Il n'y a ni overfitting, ni underfitting.

On peut alors s'intéresser aux résultats sur un signal audio.

## Résultats sur Signal Audio

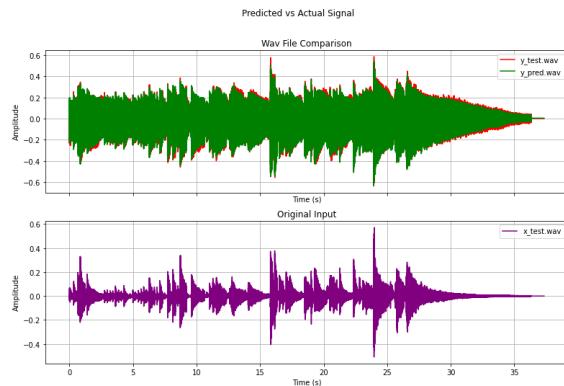


Figure 3.3: Signal Comparison | Target vs Predict

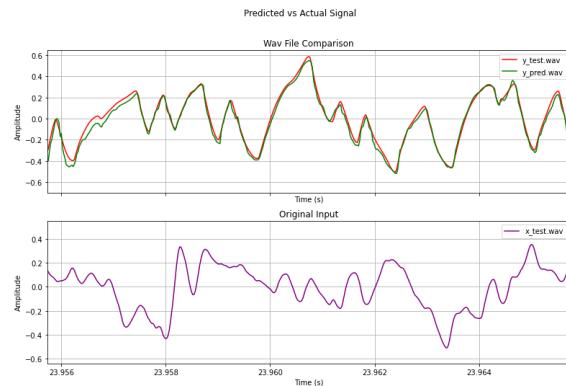


Figure 3.4: Zoom on Signal Comparison

[Play : Original](#)

[Play : Cible](#)

[Play : Predit](#)

On remarque qu'entre l'audio souhaité et l'audio prédit, il y a une très faible différence. L'audio prédit suit exactement la forme de l'audio original et à l'écoute on le ressent bien.

Le seul petit bémol que l'on peut remarquer à l'écoute, et qu'on pourrait certainement remarquer avec la STFT du Signal c'est que le timbre du signal prédit diffère un peu du signal souhaité. Il est légèrement plus aigu. Ceci pourrait certainement être amélioré en perfectionnant le modèle. Cependant dans un premier temps ce résultat est largement utilisable.

Dans la suite de ce projet on se basera sur ce modèle pour l'implémentation en C.

## Chapter 4

# Premier test sur une tâche simple (En collaboration avec Elisa DELHOMME)

Afin de mieux comprendre l'intégration d'un modèle de Deep Learning sur une carte électronique (STM32), la première étude porte sur le cas d'un modèle très simple qui nécessitera peu de traitements. Plus précisément, nous avons étudié la prédiction de la valeur d'un sinus en nous basant sur l'article et le tutoriel de Shawn Hymel.[9]

### 4.1 Modèle Test

Le modèle développé est extrêmement simple : il se compose de trois couches denses et prédit immédiatement la valeur estimée.

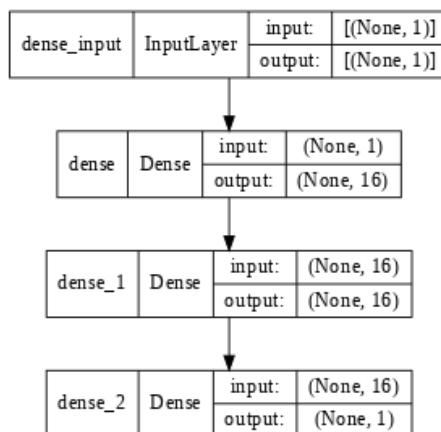


Figure 4.1: Modèle de prédiction d'un sinus

Les dataset de train, de test et de validation sont générés manuellement. L'entraînement du réseau est effectuée sur 500 époques et est assez efficace, comme on peut le voir ci-dessous :

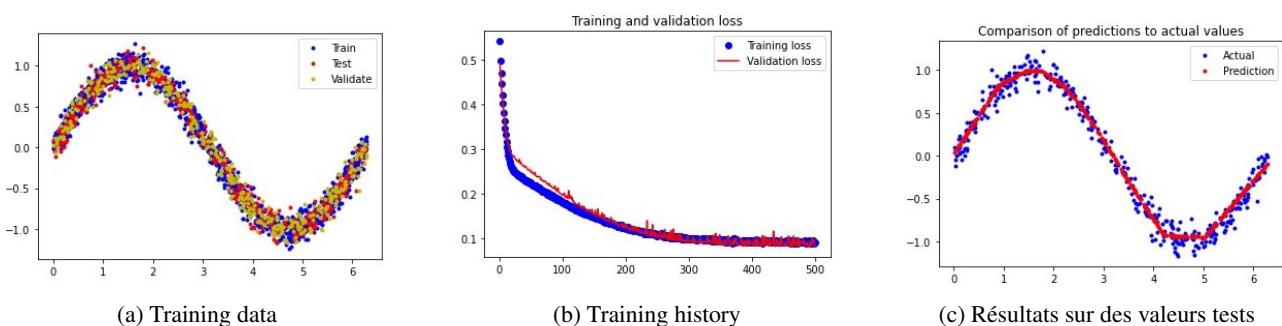


Figure 4.2: Informations sur le modèle

Quand l'entraînement du modèle est terminé, il faut ensuite l'exporter dans le format .h5 pour pouvoir l'intégrer au projet.

### 4.2 Implémentation sur STM32 - CubeAI x CubeIDE

X-Cube-AI est une extension du logiciel STM32CubeIDE, qui prend en charge la conversion automatique d'algorithme d'intelligence artificielle pré-entraînés, et l'intégration de la bibliothèque générée pour le projet. En d'autres termes, il permet de convertir en langage C un modèle Python, afin de pouvoir l'exécuter sur la carte.

## Génération du code C

En utilisant le fichier .h5 créé précédemment, le code C peut être généré simplement en sélectionnant l'extension X-Cube-AI (v.5.1.2) et en ajoutant le modèle correspondant dans l'interface CubeIDE.

**NB:** Un tutoriel plus détaillé est disponible sur le *Github* du projet.

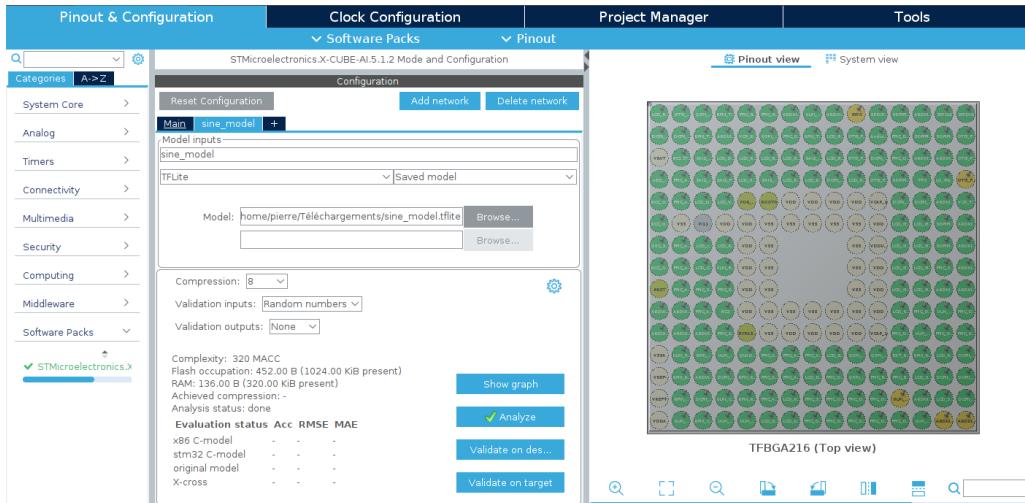
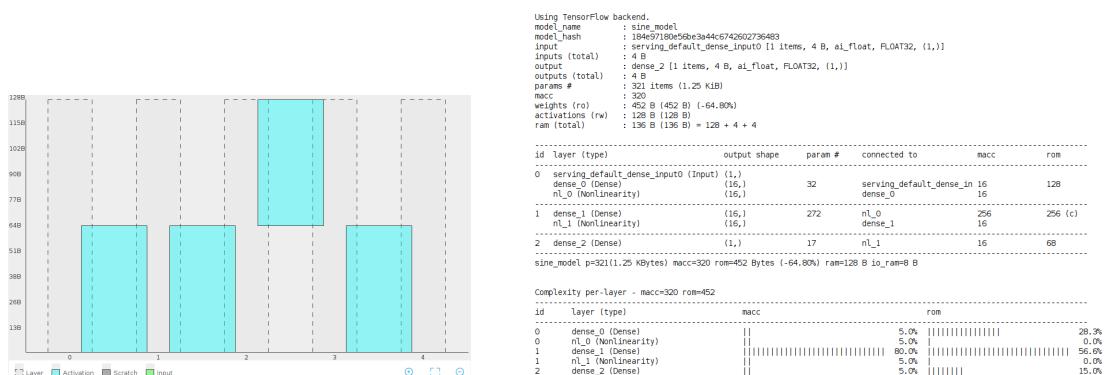


Figure 4.3: Interface CubeIDE : Génération de Code

Cette interface fournit également des informations sur le modèle, comme l'utilisation de la mémoire avec des détails par couches. Elle permet également d'effectuer une validation du modèle sur l'ordinateur ou sur la STM32, de manière à vérifier que celui-ci va fonctionner correctement, sans toucher au code.



(a) Distribution de la mémoire par couche

(b) Plus de détails sur la mémoire utilisée

## Résultas

Une fois le modèle généré en langage C par le logiciel, des ajustements sont souvent nécessaires (uniquement dans les parties USER CODE, qui permettent de conserver le code utilisateur si la génération doit être ré-effectuée), notamment pour réaliser des pré- et post-traitements sur les ensembles de données. Dans le cas de la prédiction du sinus, on entre une valeur et le modèle renvoie la valeur du sinus prédit, aucun autre ajustement n'est nécessaire. Voici ci-dessous quelques tests permettant de vérifier que le modèle fonctionne correctement :

test	float	0
y_val	float	0.0594997816

(a) Sine prediction of 0

test	float	1.57075
y_val	float	0.932305694

(b) Sine predictiton of  $\pi/2$

test	float	3.1415
y_val	float	-0.0227158722

(c) Sine predictiton of  $\pi$

test	float	4.71225023
y_val	float	-1.05025923

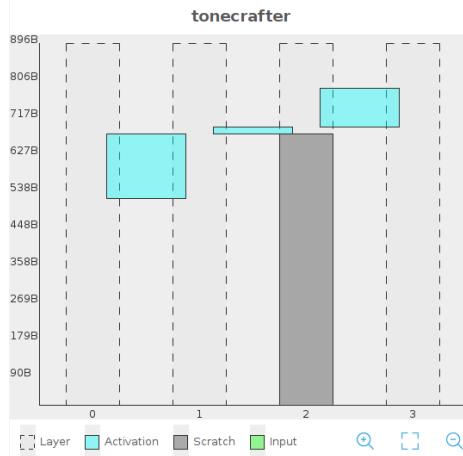
(d) Sine predictiton of  $3\pi/2$

Figure 4.5: Tests avec différentes valeurs afin de vérifier le bon fonctionnement du modèle

## 4.3 Implémentation du modèle LSTM

Après avoir pu tester l'extension CubeAI avec un modèle d'intelligence artificiel basique, il a fallu ensuite implémenter le modèle LSTM vu précédemment. L'implémentation se fait de la même manière que pour le modèle basique et ne requiert aucune étape supplémentaire. En effet, toutes les layers utilisées dans le modèle LSTM sont déjà présentes et fonctionnelles dans l'extension CubeAI, il n'y a donc pas besoin d'étape supplémentaire.

On obtient:



(a) Distribution de la mémoire par couche

ToneCrafter					
model_name	: ToneCrafter	model_hash	: 5dcf060e389259db865dc657cd9549	input	: input_0 [120 items, 480 B, ai_float, FLOAT32, (120, 1, 1)]
using	: TensorFlow backend	Inputs (total)	: 480 B	outputs (total)	: dense_1 [1 items, 4 B, ai_float, FLOAT32, (1, 1)]
Inputs (total)	: 480 B	parameters #	: 1,257 items (11.94 KiB)	macc	: 3,512
weights (rw)	: 12,516 B (12.22 KiB) (2.36%)	activations (rw)	: 784 B (784 B)	ram	: 1,288 B (1.24 KiB) = 784 + 480 + 4
ram (total)	: 1,288 B (1.24 KiB)				
Detailed memory usage					
id	layer (type)	output shape	param #	connected to	macc ram
0	input_0 (Input)	(120, 1, 1)	52	input_0	484 208
1	conv1d_2 (Conv2D)	(10, 1, 4)	196	conv1d_2	196 784
2	lstm_1 (LSTM)	(1, 1, 24)	2,784	lstm_1	2,784 11,424
3	dense_1 (Dense)	(1, 1, 1)	25	lstm_1	24 100
ToneCrafter p=0.057(11.94 KiBytes) macc=3512 ram=12.22 KiBytes (2.36%) ram=784 B 1o_ram=484 B					
Complexity per layer - macc=3,512 ram=12,516					
id	layer (type)	macc	ram		
0	conv1d_2 (Conv2D)	13.0%	1.7%		
1	conv1d_3 (Conv2D)	5.6%	6.3%		
2	lstm_1 (LSTM)	80.0%	91.3%		
3	dense_1 (Dense)	0.7%	0.8%		

(b) Plus de détails sur la mémoire utilisée

On remarque que ce modèle est approximativement 12 fois (12 kbits) plus gourmand en ressources que le modèle de prédiction du Sinus. Cependant, cela ne pose pas de problème car la carte est suffisamment puissante pour pouvoir l'utilisé.

La difficulté qu'il y a dans l'implémentation de ce modèle, est qu'en entrée, il prend 120 échantillons d'un signal audio, et en sortie, il n'y en a qu'un seul. Il faut donc faire attention aux indices utilisés pour la prédiction du réseau sur la STM.

La deuxième difficulté est de faire une prédiction sur un signal en temps réel, car pour faire cela, il y a toute l'initialisation des périphériques liés à l'audio à mettre en place. Ici, nous avions déjà effectué un code permettant la récupération (grâce aux micros présents sur la carte) et la restitution du son ambiant via un casque connecté au port jack présent sur la carte. Cela a grandement simplifié la tâche. Cependant, même avec ce code récupéré d'un autre projet, la fusion des deux n'a pas été aisée, car il fallait faire attention à toutes les librairies utilisées, et les rajouter pour la bonne compilation du code.

Finalement, après avoir réussi à tout compiler, le modèle fonctionne bien sur STM32. Il tourne correctement et est capable d'appliquer un effet sur un son en temps réel. Cependant, il reste tout de même un problème que je n'arrive pas à résoudre concernant la taille des buffers, car comme expliqué précédemment, le modèle prend 120 échantillons en entrée, il est donc en retard de 120 échantillons sur l'instant n. Pour l'instant, l'effet entendu ressemble plus à une voix de robot qu'à une distorsion, mais c'est est en bonne voie.

**NB:** Tout le code est sur *Github* et donc testable en clonant le projet.

# Conclusion

Dans la continuation de l'année dernière, le projet commence à sérieusement prendre forme. La mise en place d'un modèle d'apprentissage profond permettant la reproduction parfaite d'un effet a permis de faire avancer le projet rapidement et de le faire repartir sur de bonnes bases.

De plus, malgré que l'implémentation sur la carte STM32 ne soit pas complètement finie, celle-ci est bien avancée et en bonne voie pour obtenir un résultat rapidement. Le modèle tourne déjà correctement sur celle-ci. Il reste cependant à réussir à l'utiliser avec un signal en temps réel, comme avec le son ambiant récupéré des microphones et qui a déjà été commencé. Dans un second temps, il faudrait réussir à modifier la carte afin de pouvoir brancher une guitare directement dessus et de pouvoir jouer avec l'effet appris par le réseau de neurones en temps réel.

Malheureusement, la partie matérielle n'a pas du tout été avancée cette année... Il faudrait axer une partie dessus afin de pouvoir créer une carte dédiée au ToneCrafter, et de pouvoir par la suite affiner le design imaginé l'année dernière.

Si vous souhaitez continuer le projet ou simplement l'essayer chez vous, Je vous conseille de récupérer les ressources sur le GitHub, il y a toutes les recherches effectuées pendant 2ans.

La plupart des codes sur la création des réseaux de neurones tournent sur Google Colab, ce qui signifie qu'ils tournent sur un serveur et une session distante. Cela permet donc de pouvoir jeter un coup d'oeil et d'avoir des résultats quelque soit la configuration dont vous disposez.

Pour le reste des codes, en particulier ceux utilisant CubeAI, il faudra vous procurer une carte STM32 et installer les logiciels nécessaires (CubeAI et CubeIDE).

Enfin, je tiens à remercier Monsieur **Sylvain "Syd" Reynal** qui nous a accompagné pendant ces deux ans sur ce projet, et qui nous a permis d'avancer dans les meilleures conditions, et d'aller au bout de nos pensées.

Je tiens également à remercier **Elisa DELHOMME** qui a dû me supporter cette année, mais surtout qui m'a beaucoup aidé pour l'implémentation d'un réseau de neurones sur la STM32. Sans elle, ça n'aurait clairement pas été possible.

Merci également à **Pol Bodet**, qui a effectué en grande majorité la partie design de ce rapport, et sans qui ce projet n'aurait peut-être jamais vu le jour...

Et pour finir, Merci à toute l'équipe du ToneCrafter: **Mamadou DIA, Adrien DUWAT, Louis PRADINES, Hector RICHARD, Quentin WACONGNE, Pol BODET**, pour avoir commencé ce beau projet et qui sont toujours prêts à apporter leurs aides et une vision nouvelle sur celui-ci.

# Bibliography

- [1] Rafael Paiva. "Circuit modeling studies related to guitars and audio processing". PhD thesis. Nov. 2013.
- [2] Stefano D'Angelo. "Virtual Analog Modeling of Nonlinear Musical Circuits". PhD thesis. Nov. 2014.
- [3] Sandro Skansi. *Introduction to Deep Learning*. Springer International Publishing, 2018.
- [4] Marco A. Martínez Ramírez and Joshua D. Reiss. "Modeling Nonlinear Audio Effects with End-to-end Deep Neural Networks". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 171–175. DOI: 10.1109/ICASSP.2019.8683529.
- [5] Jesse Engel et al. "DDSP: Differentiable Digital Signal Processing". In: 2020. URL: <https://openreview.net/forum?id=B1x1ma4tDr>.
- [6] Alec Wright et al. "Real-Time Guitar Amplifier Emulation with Deep Learning". In: *Applied Sciences* 10.3 (2020). ISSN: 2076-3417. DOI: 10.3390/app10030766. URL: <https://www.mdpi.com/2076-3417/10/3/766>.
- [7] Keith Bloemer. *Neural Networks for Real-Time Audio: Raspberry-Pi Guitar Pedal*. URL: <https://towardsdatascience.com/neural-networks-for-real-time-audio-raspberry-pi-guitar-pedal-bded4b6b7f31>.
- [8] Keith Bloemer. *Neural Networks for Real-Time Audio: Stateless LSTM*. URL: <https://towardsdatascience.com/neural-networks-for-real-time-audio-stateless-lstm-97ecd1e590b8>.
- [9] Shawn Hymel. *Intro to TinyML Part 1: Training a Model for Arduino in TensorFlow*. URL: <https://www.digikey.com/en/maker/projects/intro-to-tinyml-part-1-training-a-model-for-arduino-in-tensorflow/8f1fc8c0b83d417ab521c48864d2a8ec> (visited on 11/2020).