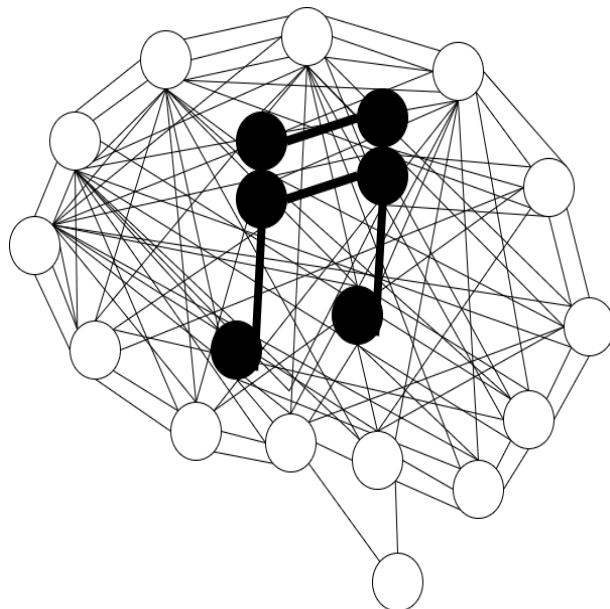


MODÈLES CONNEXIONNISTES POUR LA DÉTECTION DE NOTES ET ACCORDS DE GUITARE



Thèse de Bachelor présentée par

Jean-Daniel KÜENZI

pour l'obtention du titre Bachelor de Science HES-SO en

**Ingénierie des technologies de l'information avec orientation en
logiciels et systèmes complexes**

Septembre 2021

Professeur HES responsable

Guido BOLOGNA

Légende et source de l'illustration de couverture : Représentation d'une Intelligence Artificielle (IA) en forme de cerveau humain avec deux doubles croches reliées en son centre.
Réalisée par KÜENZI Jean-Daniel.

TABLE DES MATIÈRES

Remerciements	vi
Énoncé	vii
Résumé	viii
Liste des acronymes	ix
Liste des illustrations	xii
Liste des tableaux	xii
Liste des annexes	xiii
Introduction	1
1 Chapitre 1 : Analyse de l'existant	4
1.1 Chord AI	4
1.2 Uberchord	5
2 Chapitre 2 : Notions musicales	6
2.1 La gamme tempérée	6
2.2 Les notes de la gamme tempérée	7
a Explication des dièses (#) et bémols (b)	7
2.3 Construction d'un accord à trois sons	8
a Majeur	8
b Mineur	9
2.4 Attaque d'une note ou d'un accord	9
2.5 Transformée de Fourier Discrète (TFD)	10
2.6 Transformée de Fourier Rapide (TFR)	10
2.7 Fenêtre de Hamming	10
2.8 Partiel harmonique	12
a Partiels harmoniques d'une note	12
b Partiels harmoniques d'un accord	14
3 Chapitre 3 : Apprentissage machine	16
3.1 Perceptron	16
3.2 Fonction d'activation	18
3.3 Perceptron Multicouche (PMC)	18
3.4 Couche de convolution	19
3.5 Couche d'agrégation	20
3.6 Réseau Neuronai Convolutif (RNC)	21
3.7 Long Short-Term Memory (LSTM)	22
a Porte d'oubli	23
b Porte d'entrée	23

c	Cellule mémoire	24
d	Porte de sortie	24
4	Chapitre 4 : Ensemble de données	25
4.1	Regroupement des classes	25
4.2	Matériel utilisé	26
4.3	Enregistrement des accords à trois sons	26
a	Ensemble de données d'entraînement	26
b	Ensemble de données de validation	27
4.4	Enregistrement des notes	27
a	Ensemble de données d'entraînement	27
b	Ensemble de données de validation	27
4.5	Chargement de l'ensemble de données (prétraitement)	27
a	Calcul du coefficient du bruit	28
b	Chevauchement de signal	28
c	Saut de l'attaque	29
d	Taille de l'ensemble de données prétraité	30
5	Chapitre 5 : Conception	31
5.1	Architecture LSTMB avec TFR	31
5.2	Auto-encodeur débruiteur (AED)	31
a	Architecture	32
b	Bruit blanc gaussien	32
c	Normalisation	33
d	Espace latent	33
e	Perte de reconstruction	33
5.3	Transformée de Fourier rapide (TFR)	34
a	Composante du spectre continu	34
b	Normalisation	34
5.4	Compression des données	34
5.5	Normalisation par lots	35
5.6	Cellule LSTM Bidirectionnelle (LSTMB)	37
a	Architecture	37
5.7	Skip-Layer	38
5.8	Fonction de coût	38
6	Chapitre 6 : Expérimentation	39
6.1	PMC	39
a	Architecture avec TFR	39
b	Matrice de confusion sur l'ensemble de validation	39
c	Entraînements pour différentes tailles de fenêtres	41
6.2	RNC	43
a	Architecture avec TFR	43
b	Matrice de confusion sur l'ensemble de validation	43
c	Entraînements pour différentes tailles de fenêtres	46

7 Chapitre 7 : Résultats	48
7.1 AED	48
a Entrainements pour différentes tailles de fenêtres	48
b Reconstruction d'une donnée non bruitée	48
c Reconstruction d'une donnée avec du bruit blanc gaussien	49
d Reconstruction d'une donnée avec du gain	50
7.2 LSTMB	51
a Matrice de confusion sur l'ensemble de validation	51
b Entrainements pour différentes tailles de fenêtres	52
7.3 Ressenti d'utilisation en temps réel	53
Conclusion	56
Annexes	59
Références documentaires	68

REMERCIEMENTS

Je souhaite tout d'abord remercier le professeur HES-SO Guido BOLOGNA qui m'a guidé et soutenu tout au long de ce travail de Bachelor.

Je voudrais également remercier la professeur de français Dominique LAVELLE et ma sœur Judith KÜENZI pour leurs relectures attentives.

Pour finir, j'aimerais exprimer ma reconnaissance à mes camarades Baptiste COUDRAY, Julien DA SILVA, Ottavio BUONOMO et Ronaldo LOUREIRO PINTO pour le soutien apporté durant ces trois années de Bachelor.

MODÈLES CONNEXIONNISTES POUR LA DÉTECTION DE NOTES ET ACCORDS DE GUITARE

ORIENTATION : LOGICIELS ET SYSTÈMES COMPLEXES

Descriptif : Le but de ce projet est de pouvoir détecter des notes et des accords de guitare à l'aide de modèles connexionnistes profonds. L'étudiant construira un ensemble de données pour résoudre ce problème. Il sera possible de considérer le signal sonore dans le domaine temporel ou fréquentiel. L'étudiant se penchera aussi sur le problème des attaques et des transitions de notes/accords en général. Il sera aussi intéressant de déterminer si le prototype de détection est capable de fonctionner en temps réel.

Travail demandé :

- Construction d'un ensemble de données (notes/accords)
- Compréhension du domaine fréquentiel
- Compréhension/utilisation des modèles de réseaux de neurones artificiels convolutionnels et éventuellement LSTM
- Problématique des attaques et des transitions
- Comparaison des différents modèles utilisés
- Implantation sous forme d'application (facilement) utilisable
- Analyse des résultats
- Rédaction du rapport

Candidat :

KÜENZI JEAN-DANIEL

Filière d'études : ITI

Professeur responsable :

BOLOGNA GUIDO

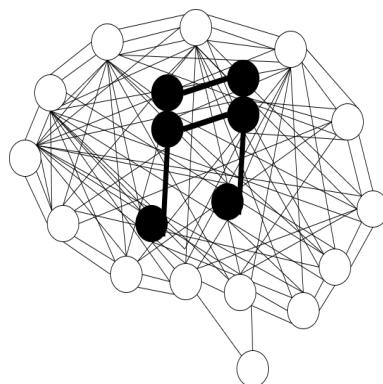
En collaboration avec : ---

Travail de bachelor soumis à une convention de stage en entreprise : **non**

Travail de bachelor soumis à un contrat de confidentialité : **non**

RÉSUMÉ

Durant ces dernières années, l'Intelligence Artificielle (IA) n'a fait que progresser et a révolutionné beaucoup de domaines comme le droit, le domaine médical, la biologie ou encore la musique. Ce travail est la continuité de mon travail précédent, dont le but était de détecter des notes de trompette sur une octave de Do. L'objectif du présent travail est d'utiliser différentes architectures de modèle connexionniste (réseau neuronal) et de voir s'il est possible de détecter des notes et accords de guitare pour des sons mono échantillonnés à 44.1[kHz]. De plus, il sera possible de considérer le signal dans le monde discret (échantillonné) et dans le monde fréquentiel. Ainsi, le but est de parvenir à une utilisation en temps réel avec le moins de latence possible (autant visuelle qu'auditive). Dans le monde de la musique, une latence supérieure à cinq millisecondes pour le traitement du signal (calculs) est indésirable et se fait fortement ressentir. Le challenge est donc que les architectures proposées prédisent le plus précisément possible l'accord ou la note jouée dans un temps inférieur à cinq millisecondes. Pour ce travail, j'ai dû également créer un ensemble de données en partant de zéro. Par rapport au temps et aux ressources mises à ma disposition, j'ai décidé de m'attaquer à la gamme tempérée de la musique occidentale (pas de musique microtonale) et de représenter les accords mineurs et majeurs à trois sons et les notes dans leurs différentes octaves pour l'accordage standard d'une guitare (Mi, La, Ré, Sol, Si, Mi). L'architecture m'ayant donné les meilleurs résultats, à savoir une précision moyenne de 95.51%, utilise une transformée de Fourier rapide pour passer dans le monde des fréquences et est composée d'une cellule LSTM bidirectionnelle.



Candidat-e :

KÜENZI JEAN-DANIEL

Filière d'études : ITI

Professeur-e(s) responsable(s) :

BOLOGNA GUIDO

En collaboration avec : —

Travail de bachelor soumis à une convention de stage en entreprise : **non**

Travail soumis à un contrat de confidentialité : **non**

LISTE DES ACRONYMES

AED Auto-encodeur débruiteur.

EPFL École Polytechnique Fédérale de Lausanne.

IA Intelligence Artificielle.

LSTM Long Short-Term Memory.

LSTMB LSTM Bidirectionnelle.

PMC Perceptron Multicouche.

RNC Réseau Neuronal Convolutif.

RNR Réseau de Neurones Récursifs.

ROC Reconnaissance optique de caractères.

TALN Traitement Automatique du Langage Naturel.

TF Transformée de Fourier.

TFCT Transformée de Fourier à court terme.

TFD Transformée de Fourier discrète.

TFR Transformée de Fourier rapide.

TQC Transformée à Q constant.

LISTE DES ILLUSTRATIONS

2.1	Gamme tempérée	6
2.2	Échelle chromatique	8
2.3	Spectre de Fourier d'un A à 220 Hertz (fuite spectrale)	11
2.4	Fenêtre de Hamming	12
2.5	Spectre de Fourier d'un A à 220 Hertz (entier)	13
2.6	Spectre de Fourier d'un A à 220 Hertz (spectre continu)	14
2.7	Spectre de Fourier d'un Amin à 220 Hertz	15
3.1	Perceptron	16
3.2	Rectified Linear Unit (ReLU)	18
3.3	Perceptron Multicouche (PMC)	19
3.4	Rétropropagation du gradient	19
3.5	Couche convulsive	20
3.6	Couche d'agrégation	21
3.7	Réseau Neuronal Convolutif (RNC)	21
3.8	Softmax	22
3.9	Entropie croisée	22
3.10	Long Short-Term Memory (LSTM)	23
4.1	Chevauchement du signal	29
5.1	Architecture LSTMB avec TFR	31
5.2	Architecture AED	32
5.3	Cellule LSTM Bidirectionnelle	37
6.1	Architecture PMC avec TFR	39
6.2	Matrice de confusion pour l'architecture PMC avec TFR	40
6.3	Architecture RNC avec TFR	43
6.4	Matrice de confusion pour l'architecture RNC avec TFR	44
7.1	Reconstruction d'un A à 110 Hertz non bruité	49

7.2	Reconstruction d'un A à 110 Hertz avec du bruit blanc gaussien	49
7.3	Reconstruction d'un Dmin avec du gain	50
7.4	Matrice de confusion pour l'architecture LSTMB avec TFR	51

Références des URL

- URL01 https://fr.wikipedia.org/wiki/Échelle_chromatique
- URL02 <https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/>
- URL03 https://fr.wikipedia.org/wiki/Fonction_d'activation
- URL04 <https://medium.com/@temi.ayo.babs/multi-layer-perceptron-for-beginners-6aee246c6a03>
- URL05 <https://medium.com/analytics-vidhya/from-convolutional-neural-network-to-variational-auto-encoder-97694e86bb51>
- URL06 <https://ai.plainenglish.io/pooling-layer-beginner-to-intermediate-fa0dbdce80eb>
- URL07 <https://www.kdnuggets.com/2016/11/intuitive-explanation-convolutional-neural-networks.html/3>
- URL08 <https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60>
- URL09 <http://apmonitor.com/do/index.php/Main/LSTMNetwork>
- URL10 <https://fr.mathworks.com/help/dsp/ref/dsp.stft.html>

LISTE DES TABLEAUX

2.1	Notes de musique occidentale	7
2.2	Construction de l'accord AMaj	9
2.3	Construction de l'accord Amin	9
4.1	Taille de l'ensemble de données pour différentes tailles de fenêtres et de sauts	30
5.1	Entrainement sur données discrètes sans la normalisation par lots	36
5.2	Entrainement sur données discrètes avec la normalisation par lots	36
6.1	Entrainement de l'architecture PMC avec TFR	41
6.2	Entrainement de l'architecture PMC sans TFR	42
6.3	Comparaison de l'accord C#Maj et Dmin	45
6.4	Comparaison de l'accord C#Maj et Dmin sous forme de fréquences	45
6.5	Entrainement de l'architecture RNC avec TFR	46
6.6	Entrainement de l'architecture RNC sans TFR	47
7.1	Entrainement de l'AED pour différentes tailles de fenêtres	48
7.2	Entrainement de l'architecture LSTMB avec TFR	52
7.3	Entrainement de l'architecture LSTMB sans TFR	53
7.4	Accord AMaj avec la quinte et la fondamentale sur plusieurs octaves	55

Références des URL

- URL11 https://fr.wikipedia.org/wiki>Note_de_musique

LISTE DES ANNEXES

Annexe A Annexe A : Architecture LSTM sans TFR	60
A.1 Architecture	60
A.2 Matrice de confusion sur l'ensemble de validation	61
Annexe B Annexe B : Architecture PMC sans TFR	62
B.1 Architecture	62
B.2 Matrice de confusion sur l'ensemble de validation	63
Annexe C Annexe C : Architecture RNC sans TFR	64
C.1 Architecture	64
C.2 Matrice de confusion sur l'ensemble de validation	65
Annexe D Annexe D : Prédictiton de l'architecture PMC sur le fichier FMaj2 de l'ensemble de validation	66
Annexe E Annexe E : Comparaison du spectre de Fourier de C#Maj et Dmin dans la même octave	67

INTRODUCTION

L'Intelligence Artificielle (IA) a révolutionné bien des domaines lors des décennies passées. Même en musique, l'IA est désormais omniprésente, allant du simple filtre à la composition de morceau¹. Généralement, ses applications sont définies pour le monde professionnel, mais avec l'explosion des technologies de l'information et l'accessibilité des données, beaucoup de personnes s'intéressent au monde de la musique de manière autodidacte. Étant moi-même de base autodidacte, j'ai beaucoup appris en écoutant des morceaux et en essayant de reproduire les accords. Toutefois, je me rendais compte que même si j'arrivais à reproduire les accords à l'oreille, je ne connaissais ni leurs noms ni leurs formes (majeure, mineure, etc.). En discutant avec mon prof de musique, il m'a fait part de la même constatation chez certains de ses élèves. C'est pourquoi j'ai eu l'idée de faire une IA capable de reconnaître des notes et accords de guitare, ainsi, il est possible pour les personnes autodidactes d'avoir plus de facilité à comprendre la structure d'un morceau ou d'une gamme.

Ce travail est la continuité de mon travail précédent, dont le but était de détecter des notes de trompette sur une octave de Do. L'objectif de cette nouvelle étude est d'utiliser différentes architectures de modèle connexionniste (réseau neuronal) et de voir s'il est possible de détecter des notes et accords de guitare pour des sons mono échantillonnés à 44.1[kHz]. De plus il sera possible de considérer le signal dans le monde discret (échantillonné) et dans le monde fréquentiel. L'objectif est ainsi d'arriver à une utilisation en temps réel avec le moins de latence possible (visuelle et auditive). Dans le monde de la musique, une latence supérieure à cinq millisecondes pour le traitement du signal (calculs) est indésirable et se fait fortement ressentir. Le challenge est donc que les architectures proposées prédisent le plus précisément possible l'accord ou la note jouée et tout ça dans un temps inférieur à cinq millisecondes. Bien sûr, comme il n'est pas possible de faire du temps réel en informatique, l'objectif sera d'utiliser un temps d'échantillonnage suffisamment court (moins d'un dixième de seconde) afin de ne pas ressentir le temps de latence.

Pour ce travail, j'ai dû également créer un ensemble de données en partant de zéro. Par rapport au temps et aux ressources mises à ma disposition, j'ai décidé de m'attaquer à la gamme tempérée de la musique occidentale (pas de musique microtonale) et de représenter les accords

1. <https://www.aiva.ai/>

mineurs et majeurs à trois sons dans l'accordage standard d'une guitare (Mi, La, Ré, Sol, Si, Mi). Ces accords seront joués jusqu'à la 12^{ème} frette (case de la guitare) non incluse et sur les deux cordes les plus graves, à savoir Mi et La. Quant aux notes, elles sont toutes représentées dans leurs différentes octaves et positions, toujours par rapport à l'accordage standard.

Les premières semaines de travail m'ont servi à récolter un maximum d'informations sur les solutions similaires existantes (traitement de la voix, détection de clés, etc.) et à comprendre les notions mathématiques nécessaires à la réalisation du travail. Pour les notions mathématiques je me suis basé sur un cours de 3^{ème} année à l'École Nationale Supérieure de l'Électronique et de ses Applications (ENSEA) sur le traitement de signal (1) présenté par le professeur Geofroy PEETERS. Il est chercheur à l'Institut de Recherche et Coordination Acoustique/Musique (IRCAM) où il supervise notamment les travaux de Master et de Doctorat.

Durant ces premières semaines, j'ai aussi profité de mettre en place un environnement virtuel Conda² afin de bénéficier d'un environnement de travail dans lequel je pourrais utiliser les API d'apprentissage profond.

Étant familier avec l'API d'apprentissage profond Keras³ (API de Google), j'ai commencé à développer de petits réseaux de neurones afin de bien comprendre comment fonctionnent les cellules LSTM que je n'avais jusqu'alors pas réellement explorées. Toutefois, Keras est une API assez haut niveau qui restreint un peu l'utilisateur, c'est pourquoi j'ai décidé de partir sur PyTorch⁴ (API de Facebook) qui est plus bas niveau, donc plus difficile à utiliser, mais qui me permettait d'être plus libre, surtout sur la création de couches personnalisées et les architectures de mes modèles. J'ai donc dû lire la documentation PyTorch et dû effectuer quelques expérimentations pour apprendre à utiliser correctement cette nouvelle API.

Durant mon apprentissage de l'API PyTorch, j'ai aussi pu examiner les modèles proposés par la communauté PyTorch et notamment ceux qui sont en lien avec le monde de la musique comme OPEN-UNMIX⁵. Ces modèles sont généralement bien documentés et m'ont permis d'acquérir et de comprendre des nouvelles bases comme les skip-layer, la normalisation par lots ou encore les cellules LSTM bidirectionnelles. Pour ces nouvelles connaissances, je me suis basé sur des études pour la plupart disponibles sur la plateforme arXiv⁶ (plateforme d'archives

-
2. <https://conda.io/projects/conda/en/latest/index.html>
 3. <https://keras.io/>
 4. <https://pytorch.org/>
 5. <https://github.com/sigsep/open-unmix-pytorch>
 6. <https://arxiv.org/>

de prépublication scientifique) qui m'ont permis de comprendre l'essentiel des informations. Elles ont aussi servi à renforcer mes bases déjà acquises et à me rendre plus confiant quant à mes choix.

Je suis ensuite passé à une phase d'expérimentation pendant laquelle j'ai testé plusieurs types d'architectures pour mes modèles connexionnistes, différentes tailles de fenêtres (temps d'échantillonnage) et différentes techniques de traitement du signal. J'ai notamment aussi enregistré deux ensembles de données, l'un avec une carte son interne à mon ordinateur et l'autre avec une carte son externe spécialisée dans le traitement audio.

Afin de sauvegarder mon travail, j'ai créé un répertoire Git⁷ sur lequel se trouve mon ensemble de données, ma thèse au format PDF ainsi qu'au format LaTeX, les images présentes dans cette thèse, le code ayant servi à entraîner et créer mes différentes architectures ainsi que mon environnement Conda au format YAML.

Au cours de cette thèse, nous verrons tout d'abord les solutions existantes que j'ai retenues et nous les détaillerons un peu.

Puis, nous verrons les notions théoriques nécessaires à la bonne compréhension du travail. Il y aura tout d'abord les notions musicales qui expliqueront ce que c'est qu'un son, comment est composé un accord et comment différencier un accord majeur d'un accord mineur.

Ensuite, il y aura une partie sur les notions mathématiques afin de bien comprendre comment on représente un son dans le monde fréquentiel et comment on passe du monde discret au monde fréquentiel ainsi que les différents problèmes qui peuvent y survenir.

Nous verrons également les notions nécessaires à la bonne compréhension de ce qu'est un réseau de neurones et comment il fonctionne depuis sa base qui est le Perceptron.

Je parlerai également de la manière dont j'ai créé mon ensemble de données, la structure des fichiers audio que j'ai enregistrés, le matériel utilisé et les différentes techniques que j'utilise dans le prétraitement de celui-ci.

Ensuite, j'expliquerai ma solution ainsi que les différentes parties qui la composent. Je vais notamment dans cette partie justifier mon choix concernant l'architecture de cette dernière.

Après cela, je présenterai sans rentrer dans le détail les architectures que j'ai testées durant mes expérimentations et que je n'ai pas retenues.

Finalement, je présenterai et débattraï des différents résultats obtenus avec ma solution et j'expliquerai également mon ressentiment d'utilisation en temps réel.

7. <https://gitedu.hesge.ch/jeandani.kuenzi/bachelor.git>

CHAPITRE 1 : ANALYSE DE L'EXISTANT

Lors de mes recherches de solutions similaires, j'ai notamment retenu deux travaux utilisant des réseaux de neurones afin de détecter en temps réel des accords de guitare, Chord AI⁸ et Überchord⁹. L'un deux a été développé en collaboration avec l'École Polytechnique Fédérale de Lausanne (EPFL).

1.1. CHORD AI

Chord AI est une application Android et iOS qui a été développée par Vivien SEGUY¹⁰, le CEO de Nomad AI¹¹, avec la collaboration de Guillaume BELLEC¹², un chercheur postdoctorant à l'EPFL. Lors de sa création, l'application était capable de reconnaître les accords majeurs et mineurs. À ce jour, elle a été améliorée et est maintenant capable de reconnaître plus de 500 accords sans compter les inversions. Elle propose aussi une reconnaissance de clés basée sur les accords détectés. Toutefois, elle n'est pas capable de reconnaître les notes de musique.

J'ai notamment pu tester l'application et lors de mon utilisation, j'ai constaté qu'elle enregistrait en temps réel ce que je jouais mais qu'il y avait un décalage de ~1/2[s] entre le moment où je jouais l'accord et la prédiction de celui-ci. Afin de comprendre pourquoi ce décalage avait lieu, j'ai contacté Vivien SEGUY et il a accepté de s'entretenir avec moi afin de discuter du fonctionnement de l'application. Il m'a expliqué qu'elle utilise un Réseau de Neurones Récursifs (RNR) pour la prédiction des accords et que la taille des fenêtres utilisée est de ~0.4[s] échantillonnée à 22.05[kHz]. Ce qui représente à peu près une fenêtre de 8820 échantillons. Les différentes fenêtres se chevauchent de ~0.2[s]. Une Transformée de Fourier à court terme (TFCT) va être appliquée sur la fenêtre lors du prétraitement ce qui va nous créer une image 2D, qu'on appelle un spectrogramme. Cette image traversera un petit Réseau Neuronal Convolutif (RNC) avant de traverser le RNR afin de faire la prédiction de l'accord.

8. <https://www.chordai.net/>

9. <https://www.uberchord.com/>

10. <https://vivienseguy.github.io/>

11. <https://www.nomadai.org/>

12. <http://guillaume.bellec.eu/>

1.2. UBERCHORD

Uberchord est une application iOS qui permet d'apprendre à jouer de la guitare de manière autodidacte. Elle propose notamment un système de suivi personnalisé en temps réel entièrement géré par une IA. Elle possède aussi une fonctionnalité de détection d'accords en temps réel. L'IA serait apparemment capable de prédire n'importe quels accords en temps réel et dans 19 accordages de guitare différents. Toutefois, il ne semble pas y avoir de fonctionnalité capable de reconnaître les notes de musique.

CHAPITRE 2 : NOTIONS MUSICALES

2.1. LA GAMME TEMPÉRÉE

La gamme tempérée ou à tempérament égal est en musique occidentale une gamme qui divise une octave en douze parties sur une échelle logarithmique avec un rapport égal à $\sqrt[12]{2}$. Pour passer à la note suivante, on multiplie la fréquence par $\sqrt[12]{2}$. Et donc inversement, pour passer à la note précédente on divise la fréquence par $\sqrt[12]{2}$.

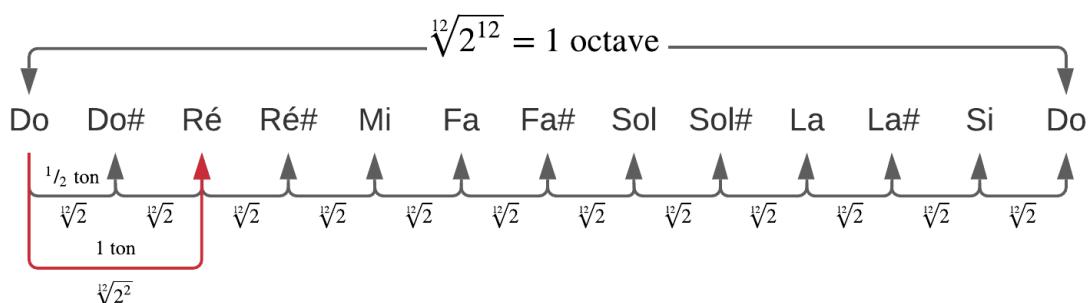


ILLUSTRATION 2.1 – Gamme tempérée. Source : Réalisé par KÜENZI Jean-Daniel

2.2. LES NOTES DE LA GAMME TEMPÉRÉE

Notation anglophone	Notation latine
C	Do
C♯ ou D♭	Do♯ ou Ré♭
D	Ré
D♯ ou E♭	Ré♯ ou Mi♭
E	Mi
F	Fa
F♯ ou G♭	Fa♯ ou Sol♭
G	Sol
G♯ ou A♭	Sol♯ ou La♭
A	La
A♯ ou B♭	La♯ ou Si♭
B	Si

TABLEAU 2.1 – Notes de musiques occidentale. Source : Réalisé par KÜENZI Jean-Daniel

Dans cette thèse, nous utiliserons la notation anglophone pour représenter les notes et accords de musique.

a. Explication des dièses (#) et bémols (♭)

Dans la musique occidentale, les # et ♭ permettent de représenter les altérations de la hauteur naturelle de la note visée. # la note est élevée d'un demi-ton, ♭ la note est abaissée d'un demi-ton.

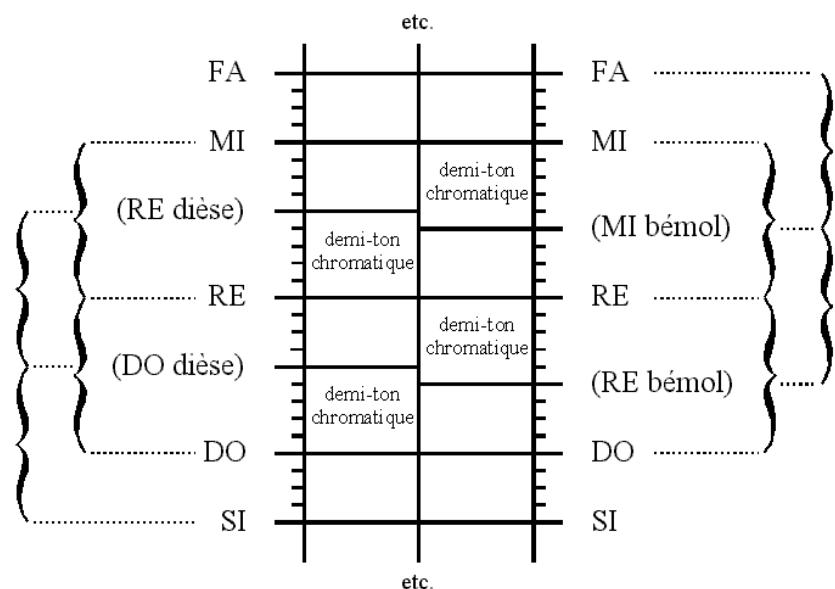


ILLUSTRATION 2.2 – Échelle chromatique. Source : tiré de *Wikipedia*, ref. URL01

2.3. CONSTRUCTION D'UN ACCORD À TROIS SONS

En harmonie tonale, un accord à trois sons est composé : d'une fondamentale, d'une tierce qui peut être mineure ou majeure et d'une quinte qui peut être juste, augmentée ou diminuée. Ce qui différencie un accord majeur d'un accord mineur est donc sa tierce.

Dans cette thèse nous verrons uniquement les accords à trois sons dit "parfaits". C'est-à-dire que leur quinte est juste.

a. Majeur

La construction d'un accord parfait majeur est très simple. Il y a tout d'abord la fondamentale qui va être notre note de base sur laquelle se construit notre accord. Puis la tierce qui est donc majeure, cela veut dire qu'elle se trouve à deux tons de la fondamentale. Puis la quinte qui est juste et se trouve donc à trois tons et demi de la fondamentale.

Prenons comme exemple l'accord AMaj (La Majeur). Il a comme fondamentale A. Puis la tierce majeure qui se trouve à deux tons de la fondamentale, c'est à dire C#. Pour finir la quinte juste qui se trouve à trois tons et demi de la fondamentale, c'est à dire E. Notre accord AMaj est donc construit de la manière suivante :

A Majeur		
	Notes	Ton(s)
Fondamentale	A	Ø
Tierce majeure	C♯	2
Quinte juste	E	3.5

TABLEAU 2.2 – Construction de l'accord AMaj. Source : Réalisé par KÜENZI Jean-Daniel

b. Mineur

La construction d'un accord parfait mineur est très simple. Il y a tout d'abord la fondamentale qui va être notre note de base sur laquelle se construit notre accord. Puis la tierce qui est donc mineure. Cela veut dire qu'elle se trouve à un ton et demi de la fondamentale. Puis la quinte qui est juste et se trouve donc à trois tons et demi de la fondamentale.

Prenons comme exemple l'accord Amin (La mineur). Il a comme fondamentale A. Puis la tierce qui doit être mineure et qui se trouve donc à un ton et demi de la fondamentale, c'est à dire C. Pour finir la quinte est juste, elle se trouve donc à trois tons et demi de la fondamentale, c'est à dire E (comme pour AMaj). Notre accord Amin est donc construit de la manière suivante :

A mineur		
	Notes	Ton(s)
Fondamentale	A	Ø
Tierge mineure	C	1.5
Quinte juste	E	3.5

TABLEAU 2.3 – Construction de l'accord Amin. Source : Réalisé par KÜENZI Jean-Daniel

2.4. ATTAQUE D'UNE NOTE OU D'UN ACCORD

En acoustique, l'attaque est la façon de commencer/jouer une note ou un accord sur un instrument. L'attaque va être un problème dans ce travail, car elle contient beaucoup d'informations (bruit) et se trouve au début de la note, elle va donc fausser l'apprentissage et les résultats prédictifs. Nous verrons dans le chapitre 4 ce que j'ai mis en place afin d'y pallier.

2.5. TRANSFORMÉE DE FOURIER DISCRÈTE (TFD)

La Transformée de Fourier discrète (TFD) sert essentiellement à transformer un signal discret (échantillonné) en une représentation spectrale discrète. L'échantillonnage est fait sur une fenêtre d'analyse bornée dans le temps. La TFD va décomposer notre signal en plusieurs superpositions d'ondes sinusoïdales de fréquences différentes. C'est donc un équivalent à la Transformée de Fourier (TF), mais dans le monde discret et pour les signaux non périodiques. Avec un signal discret y et un nombre d'échantillons N , la TFD peut être définie comme

$$y[k] = \sum_{n=0}^{N-1} y[n] * \exp\left(-\frac{2i\pi}{N}kn\right) \text{ avec } 0 \leq k < N$$

2.6. TRANSFORMÉE DE FOURIER RAPIDE (TFR)

La Transformée de Fourier rapide (TFR) est un algorithme de calculs utilisant la TFD. La TFR améliore la complexité de la TFD $O(N^2)$, où N est le nombre d'échantillons, en la réduisant à $O(N \log N)$ à l'aide de la factorisation matricielle et de divisions du problème en plusieurs parties (diviser pour régner).

2.7. FENÊTRE DE HAMMING

Lorsque nous voulons analyser un signal avec une TFD ou une TFR, on devrait théoriquement connaître la période du signal que l'on veut analyser. En pratique on ne connaît pas toujours cette période. La fenêtre va nous être utile pour éviter la fuite spectrale. La fuite spectrale est un phénomène qui survient lorsque l'on passe à la TFD un signal avec une période incomplète. Par exemple pour le A à 220[Hz], examinons son spectre de Fourier sans utiliser une fenêtre de Hamming.

Spectre de Fourier A220.wav

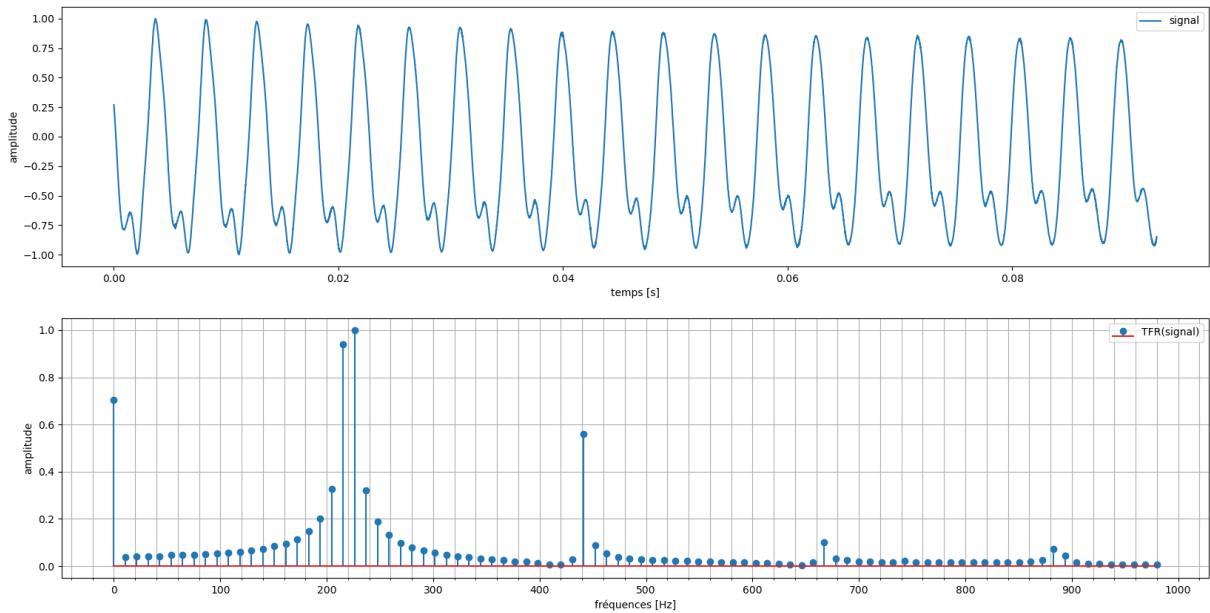


ILLUSTRATION 2.3 – Spectre de Fourier d'un A à 220[Hz] (Fuite spectrale). Source : Réalisé par KÜENZI Jean-Daniel

Comme on peut le voir, la TFR nous montre que notre signal est composé de plusieurs sinus autour de 220[Hz]. Or ce n'est pas très juste, même si en pratique il y a toujours du bruit et des interférences, on s'attendrait plutôt à obtenir un spectre plus précis, surtout autour de la fréquence fondamentale. C'est là que la fenêtre de Hamming va nous aider. En multipliant notre signal par cette fenêtre, nous allons réduire la fuite spectrale. Toutefois, on ne peut que la réduire et non l'annuler. Avec un signal x , un nombre d'échantillons N , la fenêtre de Hamming peut être définie comme

$$x[n] = 0.54 - 0.46 \cos\left(2\pi \frac{n}{N}\right)$$

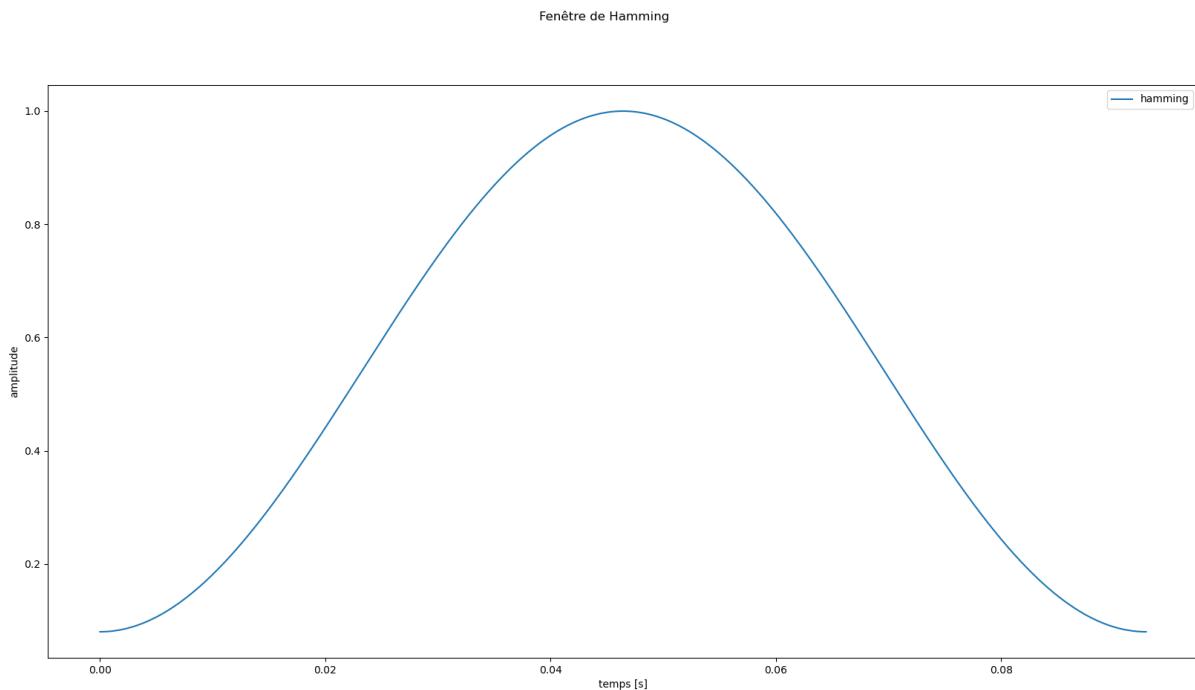


ILLUSTRATION 2.4 – Fenêtre de Hamming. Source : Réalisé par KÜENZI Jean-Daniel

2.8. PARTIEL HARMONIQUE

"En acoustique, un partiel harmonique est une composante d'un son périodique, dont la fréquence est un multiple entier d'une fréquence fondamentale." (2). On peut visualiser les partiels harmoniques d'un signal en affichant son spectre de Fourier. Mais comme la TFD se passe dans le monde des nombres complexes, ce que l'on va tracer sur notre spectre est le module de ses nombres. Avec un nombre complexe z , la partie réelle a et la partie imaginaire b , le module peut être décrit comme

$$|z| = \sqrt{a^2 + b^2}$$

a. Partiels harmoniques d'une note

Par exemple pour un A à 220 [Hz] qui est échantillonné à 44.1[kHz] sur une fenêtre de 4096 échantillons, ce qui correspond à une résolution temporelle de ~0.093[s], si l'on analyse son spectre de Fourier à l'aide d'une TFR, on obtient le spectre suivant

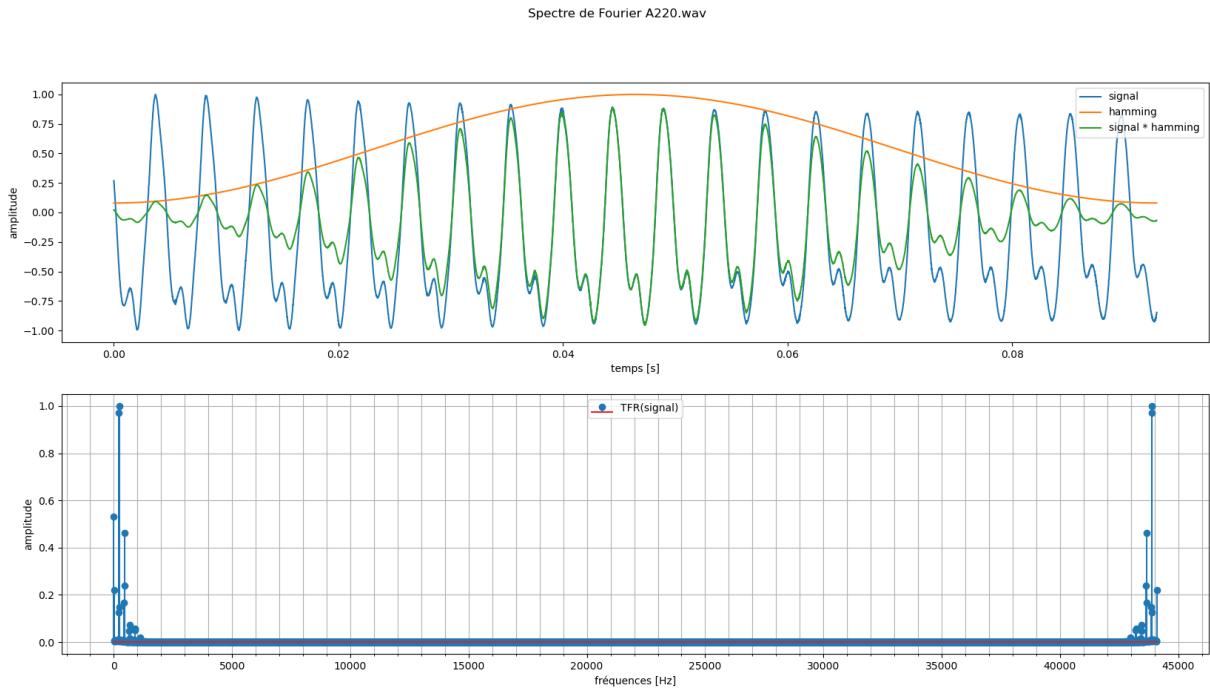


ILLUSTRATION 2.5 – Spectre de Fourier d'un A à 220[Hz] (entier). Source : Réalisé par KÜENZI Jean-Daniel

On ne voit pas grand-chose, mais on peut remarquer tout de même une sorte de symétrie. En fait les fréquences qui se trouvent à droite sont appelées les conjugués des coefficients de Fourier et elles se trouvent après la fréquence de Nyquist. La fréquence de Nyquist est égale à la moitié de la fréquence d'échantillonnage, donc dans notre cas 22.05[kHz]. Mais ce qui nous intéresse est le spectre continu du signal, c'est-à-dire la partie avec les fréquences à gauche. On va donc observer ce qui se passe entre la bande de fréquences [0;1'000] Hertz.

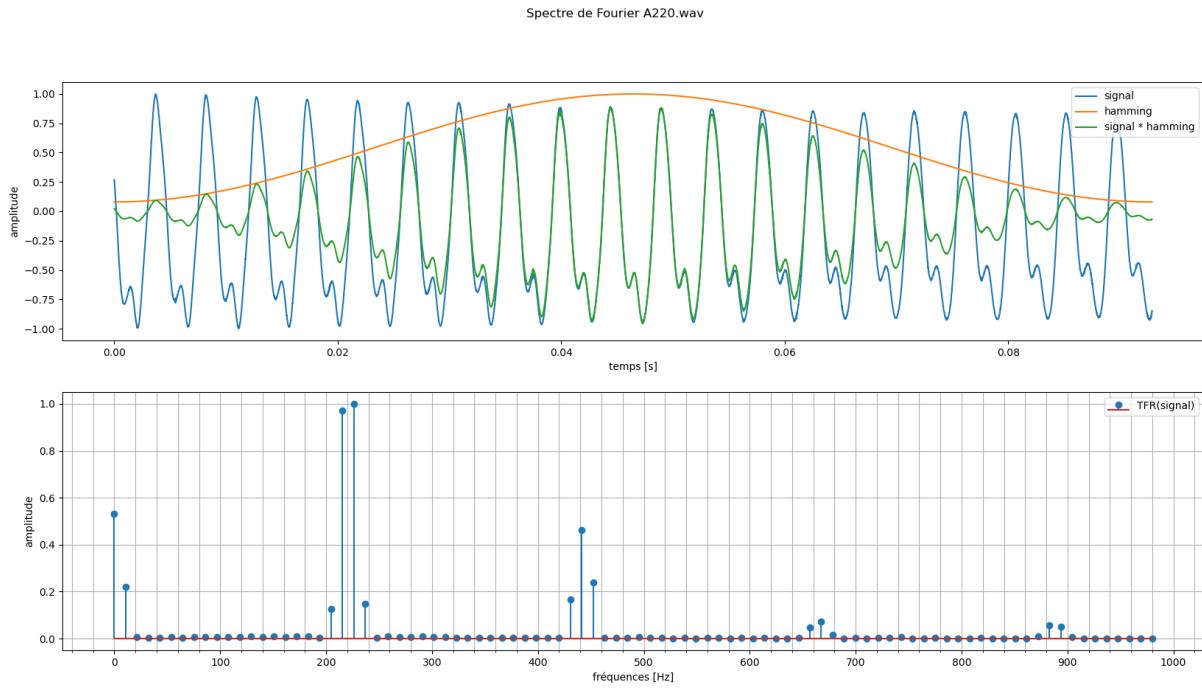


ILLUSTRATION 2.6 – Spectre de Fourier d'un A à 220[Hz] (spectre continu). Source : Réalisé par KÜENZI Jean-Daniel

On remarque tout d'abord un pic à zéro. Ce pic correspond à ce qu'on appelle la fréquence nulle. Ensuite, on voit qu'il y a deux pics autour de 220[Hz], ce qui correspond donc à notre fréquence fondamentale, puis un autre pic à 440[Hz] qui correspond donc au premier partielle harmonique $2f_1$, puis un troisième à $3f_1$, etc. Le spectre de Fourier n'est pas très précis, mais cela s'explique par le fait que nous avons une résolution fréquentielle (précision) de ~ 10.77 [Hz]. Avec f_e la fréquence d'échantillonnage et n la taille de notre fenêtre, la résolution fréquentielle peut être définie comme

$$r_f = \frac{f_e}{n}$$

b. Partiels harmoniques d'un accord

Prenons comme exemple un Amin avec comme fondamentale un A à 220[Hz]. Son spectre de Fourier est le suivant :

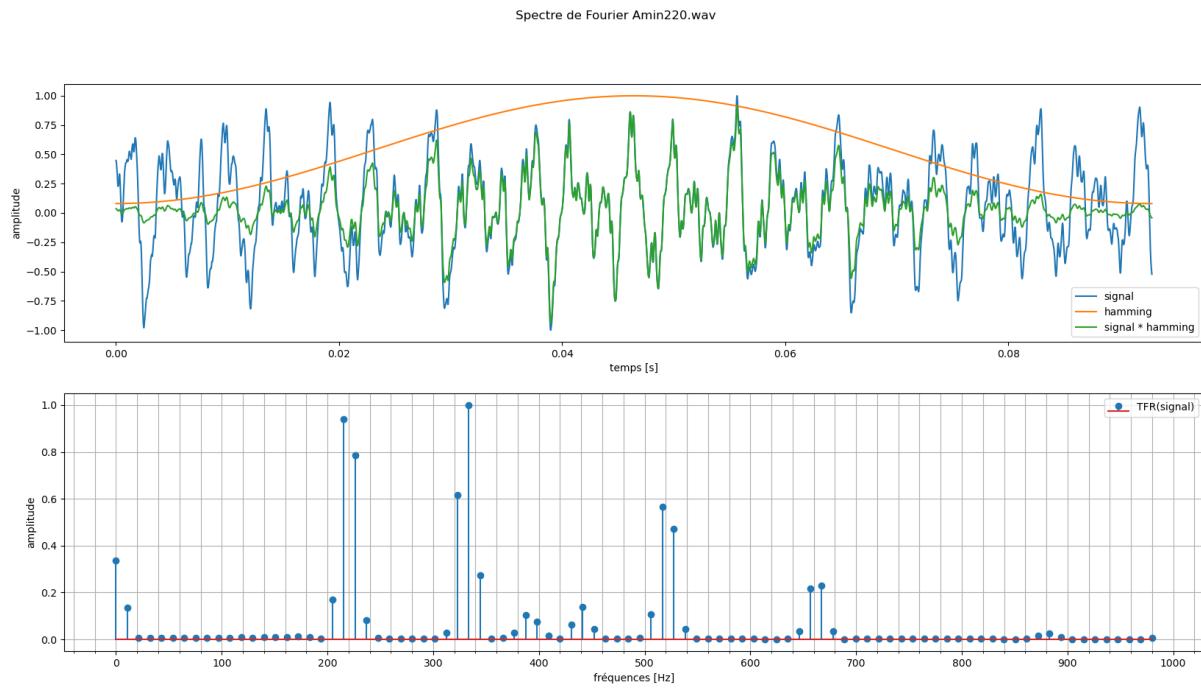


ILLUSTRATION 2.7 – Spectre de Fourier d'un Amin à 220[Hz]. Source : Réalisé par KÜENZI Jean-Daniel

On peut remarquer que le spectre de Fourier contient beaucoup plus de pics que le précédent, ce qui est normal étant donné que notre accord est une superposition de plusieurs notes et donc signaux différents. Mais on remarque bien que nous avons le pic de la fondamentale à ~220[Hz] ainsi que ses multiples. Ensuite, on constate un autre pic autour de 330[Hz]. Cela correspond à la quinte E, puis le pic de la tierce C autour de 520[Hz].

On peut se demander pourquoi la tierce a une fréquence plus élevée que la quinte, alors que normalement le C devrait être autour de 260[Hz]. C'est parce que généralement à la guitare on joue la tierce une octave plus haut.

CHAPITRE 3 : APPRENTISSAGE MACHINE

3.1. PERCEPTRON

Le perceptron est un algorithme d'apprentissage supervisé pour des problèmes de classification linéairement séparable (classifieur binaire). Le perceptron est la forme la plus simple de réseau de neurones possible. C'est un réseau qui a des entrées, un neurone comme couche intermédiaire et une sortie. On peut facilement le représenter sous forme de fonction qui mappe une entrée \vec{X} (un vecteur de valeurs réelles) à une valeur de sortie a (une valeur binaire) par un produit scalaire entre les poids \vec{W} et les entrées \vec{X} suivi d'une fonction d'activation.

$$z = \sum_{i=0}^{N-1} W_i X_i$$

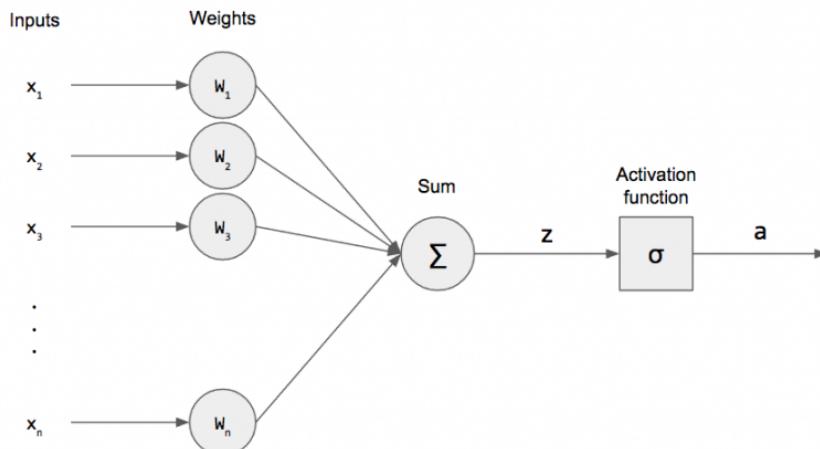


ILLUSTRATION 3.1 – Perceptron. Source : tiré de *pythonmachinelearning.pro*, ref. URL02 / réalisé par DESHPANDE Mohit

Les poids sont des valeurs réelles qui vont changer lorsque l'on va entraîner le perceptron. C'est-à-dire qu'on va lui faire classer une donnée dont on connaît déjà la classe correcte puis une fois le classement effectué, on utilise une fonction de coût pour savoir de combien le perceptron s'est trompé.

Par exemple l'erreur quadratique moyenne, où Y_i représente la sortie actuelle, \hat{Y}_i la sortie attendue et N la taille.

$$E[\vec{W}] = \frac{1}{N} \sum_{i=0}^{N-1} (Y_i - \hat{Y}_i)^2$$

Ensuite, grâce à la descente de gradient, nous pouvons trouver la valeur des poids qui minimisent la fonction de coût (l'erreur). Avec \vec{W} le vecteur de poids, la mise à jour des poids \vec{W}_t vers \vec{W}_{t+1} peut être définie comme :

$$\begin{aligned}\nabla E[\vec{W}] &= \left[\frac{\partial E}{\partial W_i} \right] \\ \Delta \vec{W} &= -\eta \nabla E[\vec{W}] \\ \vec{W}_{t+1} &= \vec{W}_t + \Delta \vec{W}_t\end{aligned}$$

η est le taux d'apprentissage, c'est un paramètre qui représente le pas que l'on va effectuer vers le minimum de la fonction de coût à chaque itération. C'est généralement un nombre assez petit, de sorte à ne pas sauter par-dessus le minimum de la fonction. Un taux d'apprentissage petit demande donc plus d'itérations pour converger et peut facilement se retrouver bloqué dans un minimum local.

Il est possible de rajouter une inertie (momentum) à la descente de gradient pour ne pas rester coincé dans un minimum local et de poursuivre la descente de gradient vers le minimum global de la fonction. Ce paramètre s'appelle généralement α (alpha) et il se situe entre zéro et un. Avec \vec{W} le vecteur de poids et α l'inertie, la mise à jour des poids \vec{W}_t vers \vec{W}_{t+1} peut être définie comme :

$$\begin{aligned}\nabla E[\vec{W}] &= \left[\frac{\partial E}{\partial W_i} \right] \\ \Delta \vec{W} &= -\eta \nabla E[\vec{W}] \\ \vec{W}_{t+1} &= \vec{W}_t + \alpha \Delta \vec{W}_t + (1 - \alpha) \Delta \vec{W}_{t-1}\end{aligned}$$

3.2. FONCTION D'ACTIVATION

Une fonction d'activation sert à définir la sortie d'un neurone par rapport à ses entrées, et donc de définir si ce neurone est activé ou non. Les fonctions d'activation sont généralement non linéaires mais il en existe plusieurs types.

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{sinon} \end{cases}$$

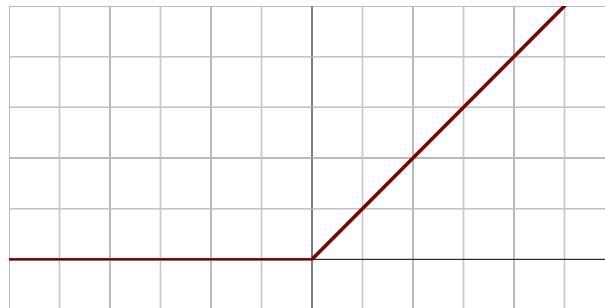


ILLUSTRATION 3.2 – Rectified Linear Unit (ReLU). Source : tiré de *Wikipedia*, ref. URL03

3.3. PERCEPTRON MULTICOUCHE (PMC)

Un Perceptron Multicouche (PMC) est une agrégation de couches cachées (couches entre les entrées et la sortie). Une couche est composée de perceptrons, ses entrées sont constituées de la sortie de la couche précédente (sauf pour la couche d'entrée) et sa sortie sera donc les entrées de la couche suivante.

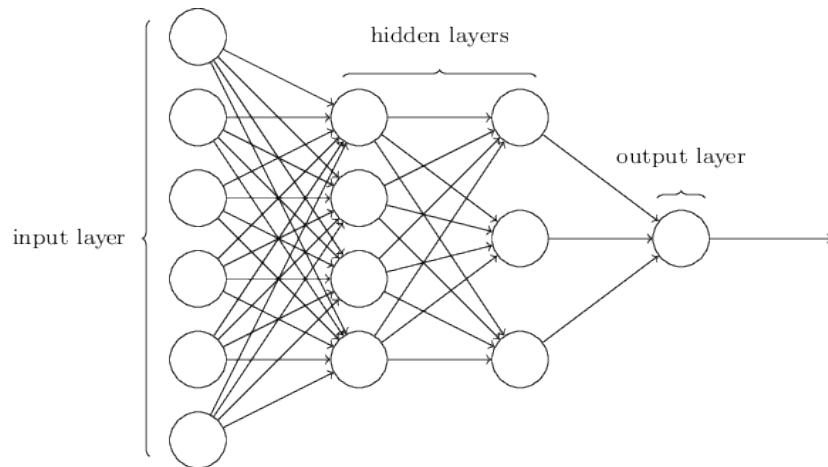


ILLUSTRATION 3.3 – Perceptron Multicouche (PMC). Source : tiré de *medium.com*, ref. URL04

Pour pouvoir corriger l’erreur sur les couches cachées, on utilise la technique de la rétropagation du gradient. Le principe est d’effectuer la descente de gradient sur tous les vecteurs de poids du réseau et de propager l’erreur aux couches précédentes. Dans le cas où la fonction de coût est l’erreur quadratique et la fonction d’activation sigmoïde, la rétropropagation peut être décrite comme :

$$\text{pour chaque unité cachée } h, \text{ calculer: } o_h = \sigma(\sum_i w_{hi} x_i)$$

$$\text{pour chaque unité de sortie } k, \text{ calculer: } o_k = \sigma(\sum_k w_{kh} x_h)$$

$$\text{pour chaque unité de sortie } k, \text{ calculer: } \delta_k = o_k(1 - o_k)(t_k - o_k)$$

$$\text{pour chaque unité cachée } h, \text{ calculer: } \delta_h = o_h(1 - o_h) \sum_k w_{hk} \delta_k$$

$$\text{mettre à jour chaque poids du réseau: } w_{ij} \leftarrow w_{ij} + \eta \delta_j x_{ij}$$

ILLUSTRATION 3.4 – Rétropropagation du gradient. Source : tiré de *l’Apprentissage Supervisé : Perceptrons Multicouche*, p. 18 / réalisé par BOLOGNA Guido

3.4. COUCHE DE CONVOLUTION

Dans le cas d’une couche de convolution, les entrées sont représentées différemment que pour le **PMC**. Pour le **PMC** nous avions un vecteur de valeurs réelles, la couche de convolution quant à elle prend un tenseur, c’est-à-dire un objet mathématique défini dans un espace vecto-

riel, et comme son nom l'indique, elle va appliquer un filtre dessus (convolution). Une couche convulsive est définie par plusieurs paramètres ; il y a les canaux (filtres) qui représentent la dimension de l'espace d'entrée et de sortie de la couche, le noyau convolutif défini par sa largeur et sa hauteur (généralement symétriques) et le stride qui représente le pas de la convolution sur les dimensions.

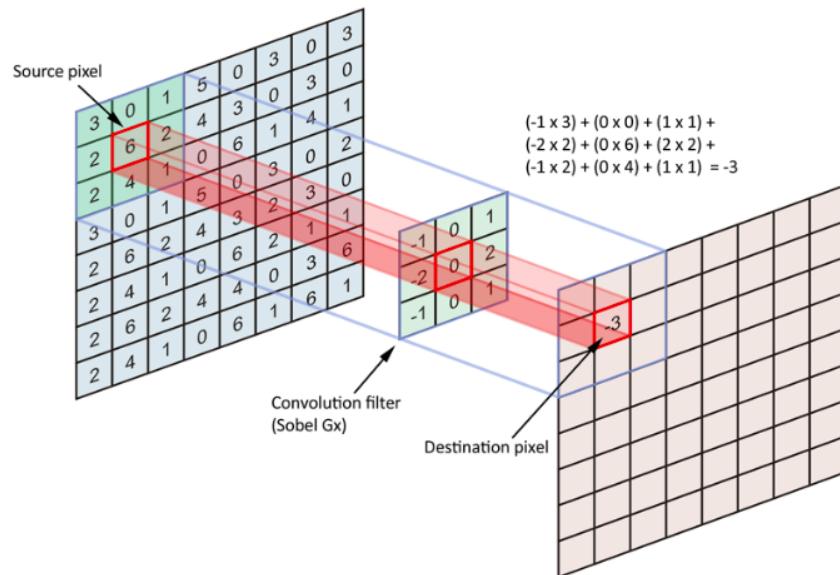


ILLUSTRATION 3.5 – Couche convulsive. Source : tiré de *medium.com*, ref. URL05 / réalisé par BUI Huy

3.5. COUCHE D'AGRÉGATION

Une couche d'agrégation vient généralement après la couche de convolution et une couche de rectification (ReLU). Elle sert à résumer l'information et à diminuer le nombre de dimensions. Elle permet notamment d'éviter le surentrainement, de réduire le nombre de paramètres et de réduire la quantité de calculs nécessaires au réseau, mais surtout, elle rend le réseau invariant aux légères distorsions des données (par exemple pour la Reconnaissance optique de caractères (ROC), si le chiffre n'est pas situé au centre de l'image). Elle est définie par un noyau et un stride (comme pour la couche convulsive) puis par une fonction non linéaire qui va appliquer la transformation, la plus connue étant Max Pooling.

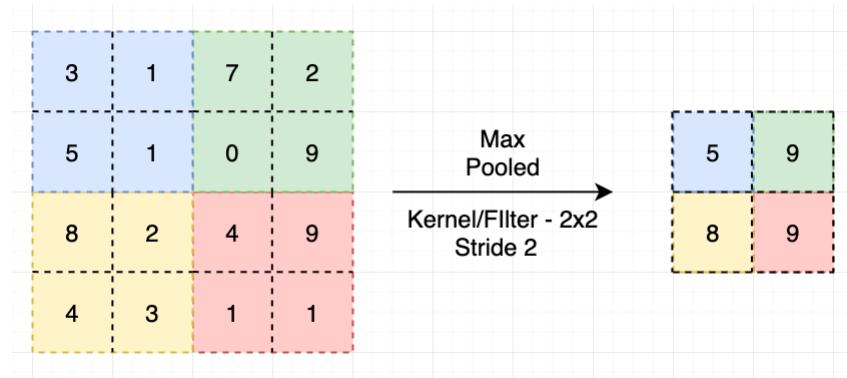


ILLUSTRATION 3.6 – Couche d’agrégation. Source : tiré de ai.plainenglish.io, ref. URL06 / réalisé par RANA Kartikeya

3.6. RÉSEAU NEURONAL CONVOLUTIF (RNC)

Un RNC est généralement utilisé dans les tâches avec des entrées fortement corrélées (la reconnaissance d’images, ROC, etc.). Dans notre cas, la détection de notes et accords de musique (séries chronologiques) présente des similarités suffisamment fortes pour qu’un RNC semble être un choix approprié. Un RNC possède une ou plusieurs couches convolutives entre les entrées et les couches inférieures du réseau.

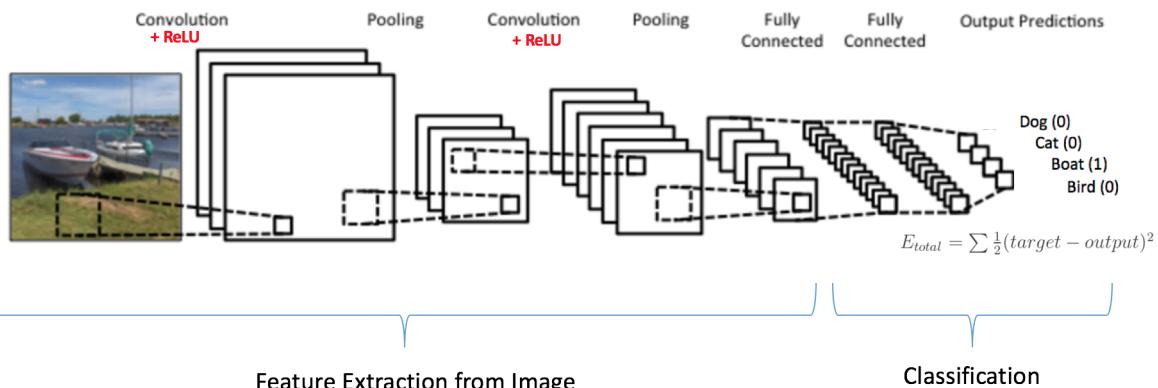


ILLUSTRATION 3.7 – Réseau Neuronal Convolutif (RNC). Source : tiré de [kdnuggets.com](https://www.kdnuggets.com), ref. URL07 / réalisé par UJIWAL Karn

Un RNC peut avoir plusieurs classes en sortie (classement non-binaire); il faut donc un moyen de connaître l’erreur pour chaque exemple. Pour cela, une fonction est appliquée à la couche de sortie, par exemple Softmax qui est une généralisation de la Sigmoïde et renvoie à un vecteur de probabilités sommant à un.

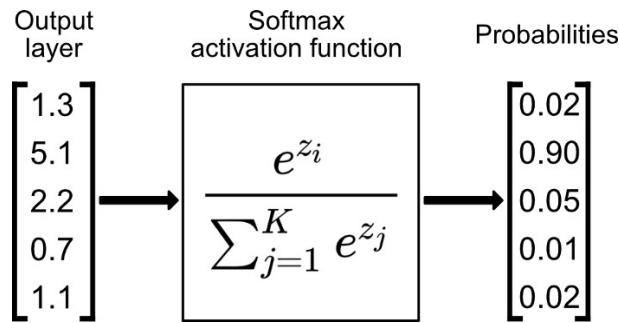


ILLUSTRATION 3.8 – Softmax. Source : tiré de [towardsdatascience.com](https://towardsdatascience.com/introduction-to-the-softmax-function-in-machine-learning-101-102-103-104-105-106-107-108), ref. URL08 / réalisé par RADECIC Dario

Dans ce cas, la fonction de coût devient l'entropie croisée (somme des log-vraisemblances pour chaque classe) et elle peut se décrire comme :

$$\sum_{i=1}^n \sum_{c=1}^C y_{i,c} \ln(\hat{y}_{i,c})$$

↑ ↑
 1 si i est dans la classe c, 0 sinon la proba prédictive d'être
 dans la classe c

ILLUSTRATION 3.9 – Entropie croisée. Source : tiré de *Les Réseaux Convolutifs*, p. 49 / réalisé par BOLOGNA Guido

3.7. LONG SHORT-TERM MEMORY (LSTM)

Un Long Short-Term Memory (LSTM) est un type de RNR. Une unité LSTM est généralement composée d'une cellule mémoire, d'une porte d'entrée, d'une porte de sortie et d'une porte d'oubli. Les réseaux LSTM sont de très bons réseaux lorsque l'on a des données chronologiques à traiter. Ils ont la possibilité d'oublier et de sauvegarder les données qui leur semblent utiles (extraction de caractéristiques ou *features extraction* en anglais). Ils proposent aussi une solution au problème de la disparition du gradient qui est bien connu dans le domaine de l'apprentissage profond.

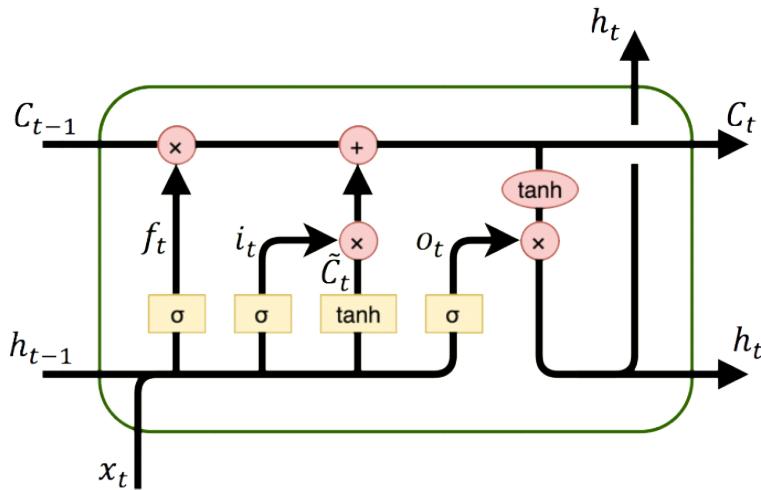


ILLUSTRATION 3.10 – Long Short-Term Memory (LSTM). Source : tiré de apmonitor.com, ref. URL09

Dans les calculs suivants, nous ne tiendrons pas compte du biais. Le biais est tout simplement un neurone fictif (souvent égal à un). Il sert à décaler la fonction d'activation en y ajoutant une constante, cela aide à contrôler la valeur à laquelle la fonction d'activation s'activera.

a. Porte d'oubli

Lorsque l'on observe l'illustration 3.10, on remarque que la première opération effectuée est celle de la porte d'oubli. Avec h_{t-1} étant l'état de la sortie à l'instant $t - 1$, X_t les entrées à l'instant t et W_f la matrice de poids, la porte d'oubli peut être décrite comme :

$$f_t = \sigma(W_f[\vec{h}_{t-1}, \vec{x}_t])$$

f_t représente le vecteur d'oubli, c'est donc un vecteur de valeurs réelles entre zéro et un que l'on va multiplier avec le vecteur C_{t-1} , qui lui représente le vecteur de mémoire à l'instant $t - 1$.

b. Porte d'entrée

La porte d'entrée quant à elle se décompose en deux parties, une première appliquant la Sigmoïde sur notre vecteur de valeurs et la deuxième qui applique Tanh.

La sortie i_t de la sigmoïde représente un vecteur de valeurs réelles entre zéro et un. Il représente la quantité (l'importance) d'informations que l'on va garder. Avec W_i une matrice de poids, cette partie peut être définie comme :

$$i_t = \sigma(W_i[\vec{h}_{t-1}, \vec{x}_t])$$

La sortie de Tanh va être un vecteur de valeurs réelles entre moins un et un. Cela correspond en quelque sorte à un vecteur de proposition, il représente les données que l'on va garder lors de la multiplication avec le vecteur i_t . Avec W_c une matrice de poids, cette partie peut être définie comme :

$$\tilde{C}_t = \text{Tanh}(W_c[\vec{h}_{t-1}, \vec{x}_t])$$

c. Cellule mémoire

La cellule mémoire est toute simple, il s'agit uniquement d'un calcul où l'on multiplie le vecteur mémoire de l'état précédent (C_{t-1}) au vecteur d'oubli (f_t) puis on lui ajoute les nouvelles données.

$$C_t = (C_{t-1} \times f_t) + (i_t \times \tilde{C}_t)$$

d. Porte de sortie

La porte de sortie se décompose également en deux étapes, une première où l'on applique la Sigmoïde sur notre vecteur de valeurs réelles, puis une deuxième où l'on multiplie la sortie de la Sigmoïde par le Tanh de notre mémoire à l'instant t .

Lors de la première partie, exactement comme pour la porte d'entrée, on va définir la quantité (l'importance) d'informations que l'on va garder pour notre sortie. Avec une matrice W_o , cette partie peut être décrite comme :

$$o_t = \sigma(W_o[\vec{h}_{t-1}, \vec{x}_t])$$

Pour finir, la sortie de notre unité à l'instant t (h_t) est tout simplement définie comme :

$$h_t = o_t \times \text{Tanh}(C_t)$$

CHAPITRE 4 : ENSEMBLE DE DONNÉES

L'ensemble de données est séparé en deux parties ; la première est destinée à l'entraînement des modèles, puis la seconde, qui est destinée à les évaluer.

Dans les deux parties, les données sont des fichiers audio mono au format wav, échantillonés à 44.1[kHz] et encodés sur 16 bits signés. Ainsi, on respecte le théorème d'échantillonnage de Nyquist-Shanon (3) et on évite le problème du repliement de spectre lors de l'analyse dans le monde des fréquences.

Nous allons tirer plusieurs extraits de ces fichiers audio qui seront nos exemples pour les différents modèles. Pour ce faire, nous allons définir une taille de fenêtre qui représentera la taille de notre exemple. Par exemple une fenêtre de 2048 échantillons, ce qui représente une résolution temporelle de ~0.046[s] pour un signal échantillonné à 44.1[kHz].

J'ai décidé de représenter les accords à trois sons mineurs et majeurs qui se jouent sur les deux cordes les plus graves, à savoir la corde de E et de A et jusqu'à la 12^{ème} frette (cas de la guitare) non incluse. Ce qui représente un total de 24 accords par corde. Les notes quant à elles sont toutes représentées.

4.1. REGROUPEMENT DES CLASSES

Afin de ne pas avoir trop de classes en sortie du modèle, j'ai décidé de regrouper les classes par notes et par accords, quelle que soit l'octave dans laquelle ils se trouvent. Par exemple la classe Emaj contient le Emaj avec la basse E à ~82[Hz] ainsi que l'accord Emaj avec la basse E à ~164[Hz]. Même si l'accord ne se trouve pas dans la même octave, il sera classifié comme un Emaj. Pareils pour les notes, même si elles ne sont pas dans la même octave, par exemple un A à 110[Hz] et un A à 220[Hz], elles seront classifiées comme A.

J'ai pris cette décision parce que si j'avais fait une classification par octave (Amaj1 = Amaj dans la première octave, Amaj2 = Amaj dans la deuxième octave, etc.) je me serais retrouvé avec beaucoup de classes en sortie du modèle. À savoir $8 * 2$ accords (mineur, majeur) dans la première octave plus $9 * 2$ accords dans la deuxième, ce qui correspond donc à 34 classes pour les accords mineurs et majeurs. À cela s'ajoutent les $4 * 12$ notes (chaque note est présente sur 4 octaves) pour un total de 82 classes. En regroupant de la sorte, j'ai réduit le nombre de classes à 36.

4.2. MATÉRIEL UTILISÉ

Pour enregistrer les fichiers, j'ai utilisé une carte son usb Roland Rubix24. Cette carte son est intéressante, car elle est entièrement blindée, ce qui réduit les interférences qui pourront se propager dans le signal et produit un bruit extrêmement faible de l'entrée à la sortie du signal. Elle m'a donc permis d'enregistrer des sons clairs et détaillés avec une excellente qualité sonore. Il est important pour le bon entraînement des modèles que l'ensemble de données soit le plus parfait et représentatif du problème que l'on veut résoudre.

Avant l'utilisation de cette carte son externe, j'avais enregistré un premier ensemble de données avec la carte son interne de mon ordinateur. Mais les signaux enregistrés à l'aide de cette carte son étaient fortement bruités, un bruit de souffle constant y était présent. On appelle ce phénomène "Bruit Blanc". J'ai constaté une nette amélioration de la précision des différents modèles après avoir enregistré le nouvel ensemble de données en utilisant la carte son externe (carte son spécialisée).

4.3. ENREGISTREMENT DES ACCORDS À TROIS SONS

Dans cette partie, nous allons détailler la structure des fichiers audio qui représentent les différents accords à trois sons. Par structure, j'entends l'organisation des accords dans les fichiers et les différentes attaques que j'ai effectuées afin d'avoir un ensemble de données le plus représentatif de la réalité. Un fichier audio contiendra uniquement l'accord qui le concerne (par exemple Amaj1.wav contient uniquement Amaj).

a. Ensemble de données d'entraînement

Le fichier va contenir huit fois le même accord, mais jouer avec des attaques et des manières différentes. Les deux premières fois, je vais jouer l'accord en pinçant les cordes avec les doigts, cela va me permettre de générer une attaque plutôt douce et longue dans la durée. Ensuite, les deux accords suivants sont joués en grattant les cordes avec la main, cela va produire une attaque légèrement plus brute et plus courte. Ensuite, deux accords sont joués avec le plectre (médiator) en grattant les cordes d'une manière sèche. Cela va produire une attaque très forte et très courte. Puis les deux derniers accords sont joués au plectre en arpège legato (toutes les notes de l'accord à la suite). Cette technique va décomposer l'accord en désynchronisant les différentes notes et donc signaux qui le composent.

b. Ensemble de données de validation

Le fichier va contenir quatre fois le même accord. La première fois, je vais jouer l'accord en pinçant les cordes avec les doigts. Ensuite, le deuxième accord est joué en grattant les cordes avec la main. Le troisième accord est joué avec le plectre en grattant les cordes de manière sèche. Puis le quatrième accord est joué au plectre en arpège.

4.4. ENREGISTREMENT DES NOTES

Dans cette partie, nous allons détailler la structure des fichiers audio qui représentent les différentes notes. Un fichier audio contiendra la note qui le concerne dans toutes ses octaves.

a. Ensemble de données d'entraînement

La note est jouée quatre fois dans ces différentes octaves et positions sur la guitare (pour un total de 24 fois), par exemple la corde à vide d'un A qui correspond à 110[Hz] et le A sur la corde de E qui correspond aussi à 110[Hz]. Deux fois où je vais la jouer avec le doigt, produisant ainsi une attaque douce et longue. Puis, deux fois où elle sera jouée au plectre, produisant ainsi une attaque brute et courte.

b. Ensemble de données de validation

La note est jouée deux fois dans ces différentes octaves et positions sur la guitare (pour un total de 12 fois). Une fois où je vais la jouer avec le doigt et une fois où elle sera jouée au plectre. Puis, en passant sur la corde suivante, j'alterne l'ordre donc je joue d'abord avec le plectre puis avec le doigt.

4.5. CHARGEMENT DE L'ENSEMBLE DE DONNÉES (PRÉTRAITEMENT)

Dans cette partie, nous allons détailler les différentes techniques que j'utilise lors du chargement de mon ensemble de données ainsi que de son prétraitement. Le prétraitement des données est une étape essentielle de l'apprentissage profond, c'est dans cette partie que nous allons nettoyer et sélectionner à partir des données brutes, les données qui vont servir à entraîner les différents modèles.

a. Calcul du coefficient du bruit

Lors du chargement d'un fichier audio, il nous faut savoir quand il y a un silence et quand il n'y en a pas. En effet, nous n'aimerions pas que le réseau apprenne du silence, cela fausserait énormément les résultats. Pour ce faire, nous allons donc calculer la valeur du bruit (silence) multiplié par un coefficient de poids. Nous allons donc récupérer notre premier exemple qui se trouve au début du fichier audio puis calculer sa valeur pour avoir une référence.

Avec un signal discret y , un nombre d'échantillons N et un coefficient de poids W , la valeur du bruit peut être définie comme :

$$P_{bruit} = \frac{W}{N} \sum_{n=0}^{N-1} |y[n]|$$

Ensuite pour chaque exemple, nous allons appliquer le même calcul sans le coefficient de poids et comparer le résultat à celui du bruit. Si la valeur est supérieure, cela veut dire qu'il ne s'agit pas d'un silence et que l'exemple est valide. Si la valeur est inférieure ou égale, il s'agit d'un silence et l'exemple n'est pas valide.

$$P_{exemple} = \frac{1}{N} \sum_{n=0}^{N-1} |y[n]|$$

b. Chevauchement de signal

Lorsque nous nous déplaçons dans le fichier audio (sur l'axe x) afin de récupérer nos exemples, nous allons appliquer la méthode du chevauchement de signal.

Cela consiste à définir une taille de saut qui va représenter le pas de déplacement sur l'axe x du signal. Généralement, cette taille de saut est deux ou quatre fois plus petite que la taille de la fenêtre. En effectuant un saut plus petit que la taille de la fenêtre, une partie de l'ancien exemple va se retrouver dans le nouveau.

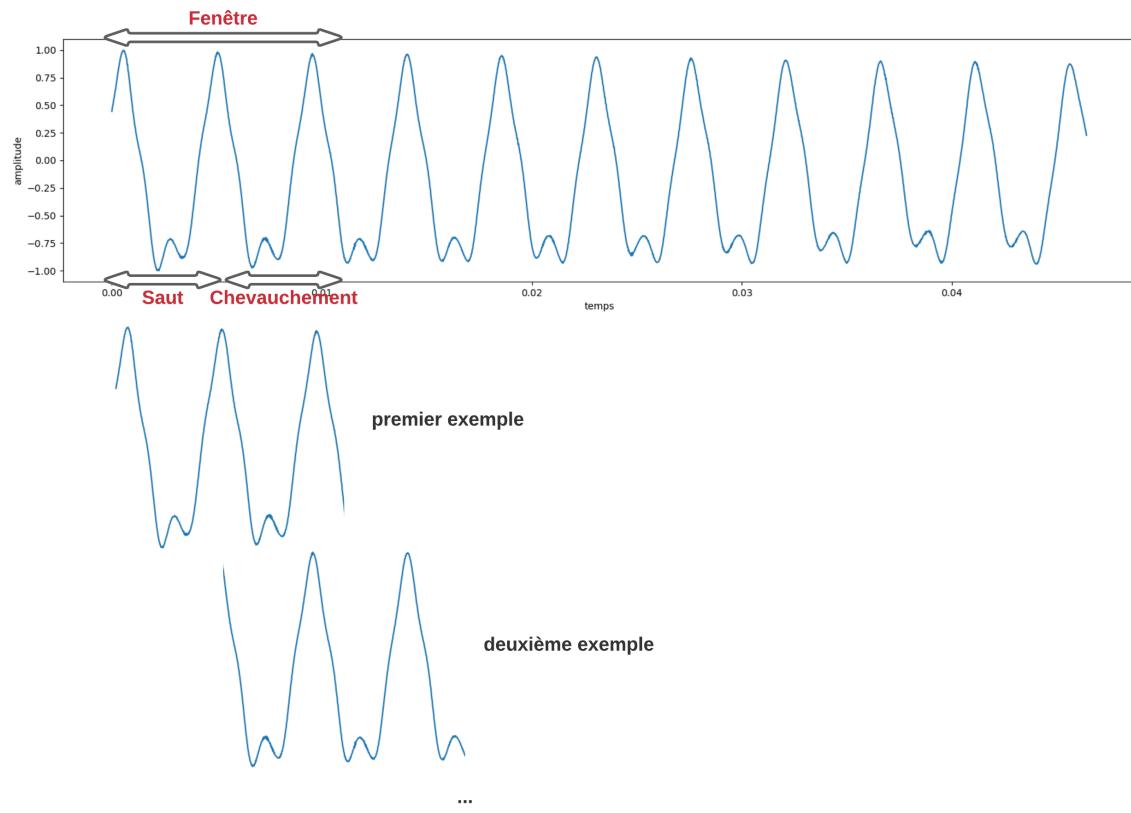


ILLUSTRATION 4.1 – Chevauchement du signal. Source : Réalisé par KÜENZI Jean-Daniel. A partir de *mathworks*, ref. URL10

C'est une convention connue dans les analyses et traitements de série chronologique. Cela permet d'augmenter le nombre d'exemples et aussi de rendre le passage d'un exemple à un suivant beaucoup plus lisse sur l'axe x. Cela va contribuer à renforcer les modèles sur les changements de phase dans les signaux.

c. Saut de l'attaque

Finalement comme notre but est une utilisation en temps réel, l'attaque ne pose pas réellement de problème lors de la prédiction. Une attaque dure rarement plus que ~0.02[s], donc lors de la prédiction en temps réel, elle sera à peine perceptive. En revanche elle va poser problème lors de l'entraînement du réseau. Comme elle contient beaucoup d'informations (bruit), si on essaye d'apprendre à classer l'attaque, cela va empêcher le réseau de converger correctement. Il est donc nécessaire de ne pas ajouter l'attaque dans nos exemples.

Pour ce faire, j'ai tout simplement décidé que le premier exemple valide de chaque si-

gnal était l'attaque et qu'il fallait l'ignorer. Ensuite pour ne pas retomber sur l'attaque lors du deuxième exemple, on se déplace directement sur l'axe x en utilisant la taille de la fenêtre et non la taille de saut.

d. Taille de l'ensemble de données prétraité

Le tableau ci-dessous, représente la taille de mon ensemble de données une fois qu'il est prétraité avec différentes tailles de fenêtres et donc de sauts. Une fois prétraité, l'ensemble de données sera réparti en lots (*batches* en anglais) de 4'000 exemples.

Nombre d'échantillons	Taille de saut	Données	Nombre d'exemples
1024	256	Entrainement	412'099
		Validation	254'564
2048	512	Entrainement	205'566
		Validation	127'019
3072	768	Entrainement	136'738
		Validation	84'516
4096	1024	Entrainement	102'347
		Validation	63'295

TABLEAU 4.1 – Taille de l'ensemble de données pour différentes tailles de fenêtres et de sauts.
Source : Réalisé par KÜENZI Jean-Daniel

CHAPITRE 5 : CONCEPTION

5.1. ARCHITECTURE LSTMB AVEC TFR

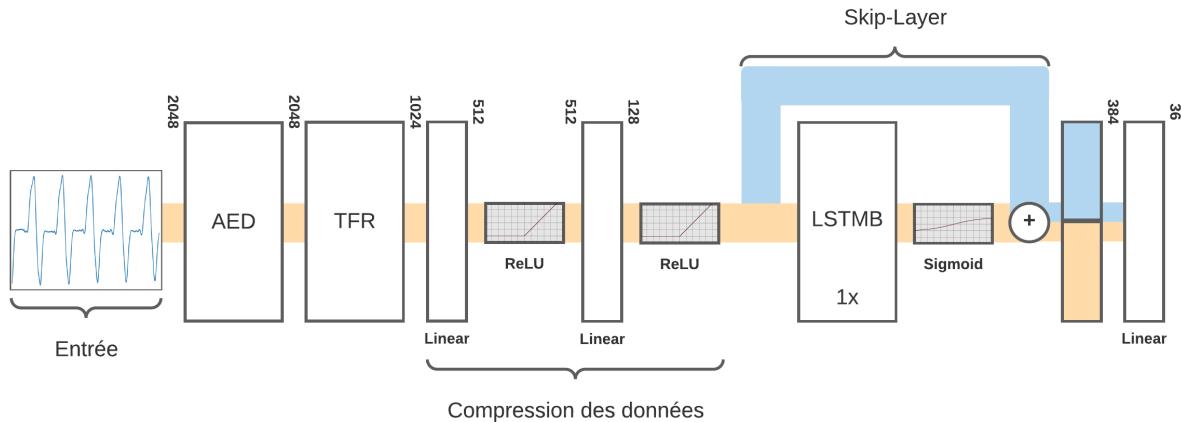


ILLUSTRATION 5.1 – Architecture LSTMB avec TFR. Source : Réalisé par KÜENZI Jean-Daniel

Dans cette thèse je vais uniquement présenter les architectures avec la **TFR** car ce sont celles qui ont fourni les meilleurs résultats. Les architectures que j'ai utilisées avec les données discrètes se trouvent en annexe. Il s'agit tout simplement de la même architecture sans la **TFR** et avec des couches de normalisation par lots (voir section 5.5) en plus.

5.2. AUTO-ENCODEUR DÉBRUISEUR (AED)

J'ai appris la connaissance de l'**Auto-encodeur débruiteur (AED)** en lisant le document "Autoencoders" (4). Je voulais trouver un moyen de retirer le bruit de mes données de manière rapide (nécessaire pour l'utilisation en temps réel) et sans avoir à passer par des filtres à convolution. Il était important de pouvoir réduire le bruit dans un signal dans le cas où nous n'utilisons pas du matériel spécialisé dans le traitement audio. De plus, je me suis dit que cela pourrait aider à réduire les différents effets, comme le gain ou la distorsion, que nous pouvons ajouter sur le signal. Cette réduction pourra donc aider le modèle à prédire plus facilement l'accord ou la note. En temps réel et dans le monde de l'audio, la latence que l'on est prêt à accepter est en dessous de cinq millisecondes pour le traitement du son (les calculs). D'où la nécessité d'avoir une méthode rapide et simple.

Le but d'un **AED** est de reconstruire les données originales non bruitées à partir de données

bruitées. Pour ce faire, on va volontairement brouter les données qui vont servir d'exemples lors de l'entraînement en rajoutant du bruit blanc gaussien $X \rightarrow \tilde{X}$ et en entraînant l'AED à reconstruire les données non bruitées en calculant la perte $L(\tilde{X}, X)$.

Il existe plusieurs types d'auto-encodeurs et différentes architectures, mais j'ai décidé d'opter pour une architecture profonde en bottle-neck afin de ne pas perdre trop de temps en traversant le réseau. Comme nous voulons prédire en temps réel, il doit y avoir le moins de temps possible passé à traverser le réseau.

Lorsque le signal va passer dans l'AED, il va être séparé en huit vecteurs de taille 256, puis il sera reconstruit en un vecteur de taille 2048 à la fin, juste avant d'être normalisé à nouveau. On effectue une séparation, car l'architecture dense utilisée est plus efficace pour réduire le bruit sur des petites tailles de vecteurs que sur des grandes.

a. Architecture

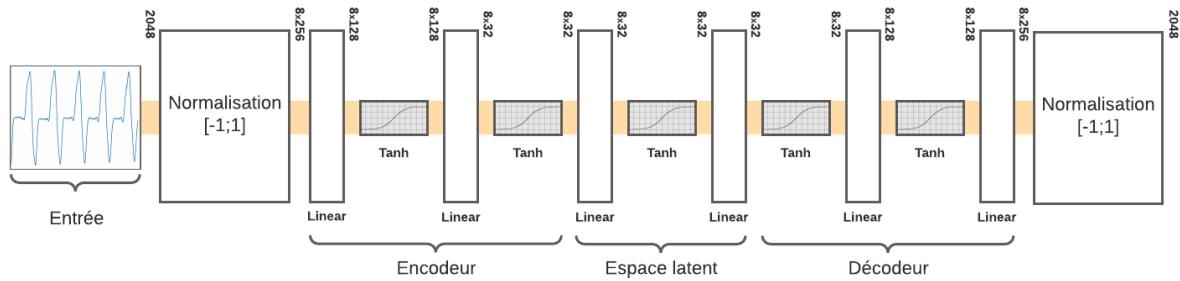


ILLUSTRATION 5.2 – Architecture AED. Source : Réalisé par KÜENZI Jean-Daniel

b. Bruit blanc gaussien

Le bruit blanc gaussien est un type de bruit qui est utilisé pour corrompre des données et qui est largement utilisé dans le monde des AED. Il est très intéressant, car sa puissance est uniforme sur toute la largeur de bande de fréquences et il suit une loi de distribution normale (gaussienne). Le bruit qui a été ajouté a une moyenne de zéro et un écart type de zéro virgule trois. Dans le monde des signaux, il peut être comparable à un bruit de souffle constant assez fort.

c. Normalisation

Lorsque le signal va passer à travers l’**AED**, il va être normalisé à son entrée puis à sa sortie. La normalisation des données est d’une importance capitale pour le réseau, cela va lui permettre de ne pas apprendre les amplitudes des signaux et va lui permettre de se concentrer sur d’autres caractéristiques. Aussi, la normalisation va aidée l’**AED** à se faire sa propre représentation des données dans son espace latent et à converger plus rapidement.

La normalisation effectuée est une Min-Max avec une plage entre moins un et un. Son avantage est qu’elle est rapide à faire et elle va conserver le rapport entre les différentes amplitudes du signal (la forme du signal reste inchangée). Avec un signal discret Y et le signal discret normalisé Y' , la normalisation appliquée peut être définie comme :

$$Y' = 2 \frac{Y - \min(Y)}{\max(Y) - \min(Y)} - 1$$

d. Espace latent

L’espace latent est une représentation compressée des données que le réseau va apprendre de lui-même. Lors de l’entraînement, il va apprendre à extraire les caractéristiques qui lui semblent les plus pertinentes et va ensuite les utiliser afin de reconstruire les données non bruitées.

e. Perte de reconstruction

Pour un auto-encodeur, on parle de perte de reconstruction pour signifier la différence entre les données attendues et les données reconstruites. La fonction que j’ai utilisée pour calculer ma perte de reconstruction est l’erreur quadratique moyenne. Avec Y_i la sortie actuelle, \hat{Y}_i la sortie attendue et N la taille. L’erreur quadratique moyenne peut être définie comme :

$$E[\overrightarrow{W}] = \frac{1}{N} \sum_{i=0}^{N-1} (Y_i - \hat{Y}_i)^2$$

5.3. TRANSFORMÉE DE FOURIER RAPIDE (TFR)

Une fois que le signal est passé par l'[AED](#), on va effectuer une [TFR](#) afin de récupérer son spectre. Mais avant de faire la [TFR](#), nous allons multiplier le signal par une fenêtre de Hamming afin de réduire la fuite spectrale et de faciliter ainsi l'apprentissage au réseau.

a. Composante du spectre continu

Comme vu dans la section 2.8, nous nous intéressons uniquement à la partie du spectre continu. Donc lorsque que nous effectuons une [TFR](#) avec une fenêtre de taille N nous allons uniquement prendre en compte les $N/2$ premiers coefficients de Fourier. Cette réduction de moitié permet au réseau d'être plus précis sur ses prédictions et plus rapide. Comme la [TFR](#) retourne des nombres complexes, c'est le module de ses nombres que l'on va passer au réseau (voir section 2.8).

b. Normalisation

Une fois la [TFR](#) faite, nous allons normaliser les données afin que la valeur des amplitudes ne soit pas un facteur que le réseau doivent apprendre. Finalement, tout ce qui nous intéresse c'est uniquement de connaître les fréquences et non leurs amplitudes. On va donc pouvoir normaliser le spectre obtenu afin de faciliter au réseau l'apprentissage et la prédiction.

La normalisation effectuée est une Min-Max avec une plage entre zéros et un. Son avantage est qu'elle est rapide à faire et elle va conserver le rapport entre les différentes amplitudes du spectre. Avec un signal discret Y et le signal discret normalisé Y' , la normalisation appliquée peut être définie comme :

$$Y' = \frac{Y - \min(Y)}{\max(Y) - \min(Y)}$$

5.4. COMPRESSION DES DONNÉES

Comme pour l'[AED](#), cette couche va permettre au réseau de se faire sa propre représentation des données qu'il reçoit de la [TFR](#). Lors de l'entraînement, il va apprendre à extraire les caractéristiques qui lui semblent importantes. Cette couche est ajoutée afin de permettre aux données de traverser plus rapidement le réseau sans perdre pour autant la précision.

5.5. NORMALISATION PAR LOTS

La normalisation par lots ou (*Batch Normalization* en anglais) est une technique décrite dans le document "Batch Normalization : Accelerating Deep Network [...]" (5). Le principe est de rajouter des couches qui vont aider à mieux coordonner la mise à jour des poids lors de la rétropropagation du gradient. Elle rend notamment le réseau moins dépendant du paramétrage initial comme le taux d'apprentissage et la génération aléatoire des poids. Elle permet aussi dans certains cas de ne pas avoir besoin de recourir à des couches de dropout.

En apprentissage profond, lors de la rétropropagation du gradient, on corrige les poids d'une couche à partir d'une estimation de l'erreur en supposant que les poids des couches précédentes sont fixes. Mais finalement comme nous mettons à jour toutes les couches du réseau simultanément cette supposition est fausse. Cela nous oblige à faire attention aux paramètres d'initialisations du réseau et ralentit l'entraînement en nous contraignant à utiliser un petit taux d'apprentissage. Dans le document (5), ce problème est nommé "Changement de Covariable Interne" (*Internal Covariate Shift* en anglais).

La normalisation par lots va nous aider à régler ce problème en normalisant les entrées des couches par mini-lots (*mini-batches* en anglais) pour avoir une moyenne de zéro et un écart type de un. Cette normalisation permet au réseau durant la rétropropagation du gradient de mieux régulariser l'erreur et de rendre la mise à jour des poids moins propice à varier de manière drastique.

Cette problématique de réduction du changement de covariable interne reste encore discutée. D'autres études comme "How Does Batch Normalization Help Optimization ?" (6) estiment que la normalisation par lots aide l'entraînement du réseau en rendant la descente de gradient plus lisse et simplifie donc la convergence du réseau.

Il est aussi possible de permettre à la couche de normalisation d'apprendre deux paramètres, β et γ . Généralement cela se fait beaucoup, car normaliser les entrées des couches peut modifier la représentation de la couche. Ces deux paramètres permettent de s'assurer de pouvoir retrouver la transformation identité (la réelle représentation).

Avec x les entrées, $E[x]$ l'erreur pour chaque entrée, $\text{Var}[x]$ l'écart type pour chaque entrée, ϵ une valeur ajoutée pour la stabilité numérique, β et γ les deux paramètres apprenables. PyTorch (7) définit la normalisation par lots comme :

$$y = \frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}} \times \gamma + \beta$$

J'ai utilisé les couches de normalisation par lots lors de la prédiction avec les données discrètes (sans TFR) car cela avait un impact non négligeable sur la précision des modèles. Mais lors de l'utilisation avec les données de la TFR j'ai constaté une perte de précision. Les couches de normalisation par lots sont ajoutées après les couches denses et les couches de convolution mais avant la fonction d'activation ReLU qui est non linéaire, comme précisé dans le document (5). Les tableaux ci-dessous représentent les différences d'entraînements de l'architecture LSTM sans TFR (voir annexe A) avec et sans couches de normalisation par lots.

Sans la normalisation par lots		
epochs = 50, $\eta = 0.005$, nombre d'entraînements = 10		
Données	Précision moyenne %	Ecart type %
Entrainement	98.19	0.43
Validation	88.36	0.51

TABLEAU 5.1 – Entraînement sur données discrètes sans la normalisation par lots. Source : Réalisé par KÜENZI Jean-Daniel

Avec la normalisation par lots		
epochs = 50, $\eta = 0.005$, nombre d'entraînements = 10		
Données	Précision moyenne %	Ecart type %
Entrainement	99.91	0.02
Validation	92.45	0.54

TABLEAU 5.2 – Entraînement sur données discrètes avec la normalisation par lots. Source : Réalisé par KÜENZI Jean-Daniel

En observant les résultats, on constate bien l'efficacité de ces couches de normalisation par lots. Avec les mêmes paramètres d'initialisation, le modèle a convergé beaucoup plus rapidement et l'on a ainsi gagné un peu plus de quatre pour cent de précision moyenne sur l'ensemble de validation.

5.6. CELLULE LSTM BIDIRECTIONNELLE (LSTMB)

Une cellule LSTM Bidirectionnelle (LSTMB) est en fait composé de deux cellules LSTM. Une qui va propager h_t vers l'avant et une qui va le propager vers l'arrière. On peut voir sa comme recevoir les informations du passé et du futur à la fois. Une cellule LSTM classique utilise h_{t-1} (passé) pour produire h_t . Une cellule LSTMB va à la fois utiliser h_{t-1} et h_{t+1} (futur) pour produire y_t qui est notre sortie. En fait y_t est tout simplement une concaténation (parfois accompagné d'une fonction d'activation) des sorties h_t des deux cellules (voir figure 5.3). Comme on récupère la sortie des deux cellules et qu'on concatène, on a plus de données à la sortie qu'à l'entrée.

Donc dans notre cas, on peut voir qu'on a un vecteur de taille 128 à l'entrée de la cellule et on aura donc un vecteur de taille 256 à sa sortie, on a augmenté la taille de nos données par un facteur deux.

a. Architecture

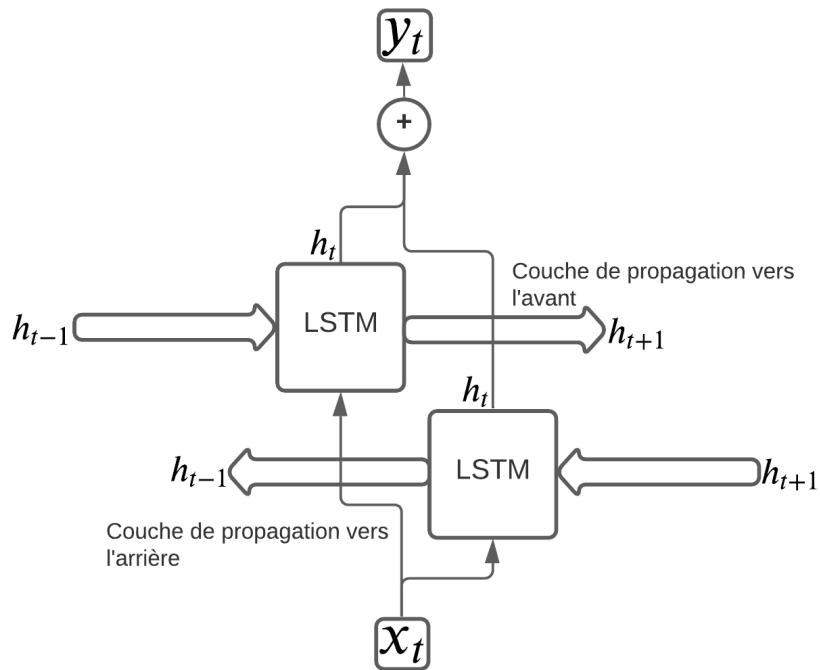


ILLUSTRATION 5.3 – Cellule LSTM Bidirectionnelle. Source : Réalisé par KÜENZI Jean-Daniel

5.7. SKIP-LAYER

La Skip-Layer est en fait une technique utilisée où, comme son nom l'indique, on va éviter des couches. Comme on peut le voir sur la figure 5.1, une copie des données vont être prises avant de traverser la cellule **LSTMB** et vont ensuite être concaténée à sa sortie. La skip-layer va permettre aux couches suivantes d'apprendre également des données qui se trouvent avant la cellule **LSTMB**. Elle aide aussi pour la rétropropagation du gradient, notamment en aidant à éviter le problème du gradient de fuite. La Skip-Layer est beaucoup utilisée dans les architectures modernes.

5.8. FONCTION DE COÛT

La fonction de coût que j'ai utilisé est l'entropie croisée (*cross-entropy* en anglais). PyTorch (8) utilise une combinaison de la fonction de perte logarithmique de vraisemblance négative (*Negative Log-Likelihood Loss* en anglais) et de la fonction Logarithmique Softmax (*Log-Softmax* en anglais), qui est tout simplement le logarithme de la fonction softmax. En pratique, on préfère l'utilisation de la fonction log-softmax plutôt que la fonction softmax. Elle est plus utilisée, car la fonction log-softmax aide le réseau en assurant une stabilité numérique de l'erreur. Elle ne va jamais devenir infiniment petite. Aussi, de par sa nature exponentielle elle inflige des pénalités beaucoup plus lourdes que la softmax (qui est linéaire) pour les classes incorrectes, et aide donc le réseau à converger plus vite. Pour finir, elle est optimisée pour effectuer moins de calculs que la softmax durant la descente de gradient. Avec x un tenseur de valeurs réelles. PyTorch définit la log-softmax comme :

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

et donc avec x un tenseur de valeurs réelles qui représente la sortie du réseau et y un tenseur correspondant aux classes correctes attendues, PyTorch (8) définit l'entropie croisée comme

$$L(x, y) = -x_y + \log\left(\sum_j \exp(x_j)\right)$$

CHAPITRE 6 : EXPÉRIMENTATION

6.1. PMC

a. Architecture avec TFR

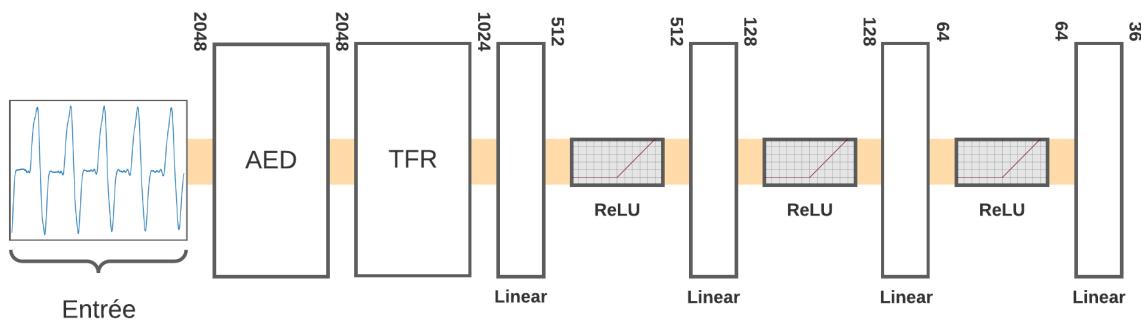


ILLUSTRATION 6.1 – Architecture PMC avec TFR. Source : Réalisé par KÜENZI Jean-Daniel

b. Matrice de confusion sur l'ensemble de validation

La matrice de confusion suivante a été réalisée avec le modèle utilisant une fenêtre de 2048 échantillons.

(voir annexe D) la prédiction du modèle sur le fichier FMaj2.wav de l'ensemble de validation.

Finalement les erreurs où, à la place d'un accord, le modèle trouve une note, à savoir soit la fondamentale, soit la tierce, soit la quinte, ne sont pas trop graves et généralement cela n'est pas réellement une erreur, mais relève d'une imperfection dans les enregistrements de l'ensemble de validation. Les erreurs les plus graves surviennent lorsque le modèle prédit une note comme étant un accord ou se trompe dans une note ou un accord de manière très éloignée de la classe correcte, par exemple F# prédit comme un F#min.

c. Entrainements pour différentes tailles de fenêtres

PMC avec TFR

epoch = 50, $\eta = 0.005$, nombre d'entrainements = 10

Nombre d'échantillons	Données	Précision moyenne %	Ecart type %
1024	Entrainement	99.44	0.07
	Validation	85.60	0.63
2048	Entrainement	99.86	0.03
	Validation	95.23	0.80
3072	Entrainement	99.89	0.02
	Validation	94.56	0.34
4096	Entrainement	99.90	0.01
	Validation	94.65	0.24

TABLEAU 6.1 – Entrainement de l'architecture PMC avec TFR. Source : Réalisé par KÜENZI Jean-Daniel

Lorsque l'on observe les résultats de l'architecture **PMC** avec **TFR**, le premier constat est qu'une fenêtre de 1024 échantillons, soit une résolution temporelle de ~0.023[s], donne de mauvais résultats avec la **TFR**. Cela peut s'expliquer par la résolution fréquentielle. Avec 1024 échantillons nous avons une résolution fréquentielle de ~43.06[Hz]. On peut donc facilement confondre des notes et des accords. Ensuite, le deuxième constat est qu'à partir d'une fenêtre de 2048 échantillons, augmenter le nombre d'échantillons (donc la taille de la fenêtre) n'améliore pas la précision du modèle. Elle la fait même légèrement chuter, mais cela peut être en partie dû à la réduction de la taille de l'ensemble de données ainsi que du problème évoqué dans la

sous-section b. Donc 2048 échantillons semblent une bonne de taille fenêtre, nous avons une résolution fréquentielle de ~21.53[Hz] et une résolution temporelle de ~0.046[s]. Comme sa résolution temporelle est faible, cette fenêtre semble être un bon choix pour le temps réel.

De plus, les résultats sont extrêmement bons, on voit bien que le modèle arrive correctement à différencier les différents accords et notes. Même s'ils sont légèrement faussés à cause du problème évoqué dans la sous-section b, ses résultats restent très prometteurs et encourageants.

PMC sans TFR

epochs = 50, η = 0.005, nombre d'entrainements = 10

Nombre d'échantillons	Données	Précision moyenne %	Ecart type %
1024	Entrainement	99.56	0.21
	Validation	91.21	0.27
2048	Entrainement	99.88	0.07
	Validation	92.34	0.37
3072	Entrainement	99.82	0.08
	Validation	92.22	0.41
4096	Entrainement	99.98	0.03
	Validation	91.29	0.11

TABLEAU 6.2 – Entrainement de l'architecture PMC sans TFR. Source : Réalisé par KÜENZI Jean-Daniel

Lorsque l'on observe les résultats de l'architecture **PMC sans TFR**, on constate tout d'abord que les résultats sont globalement plutôt bons, même s'ils sont moins bons que pour l'architecture avec **TFR**. Ensuite on constate qu'à partir de la fenêtre de 2048 échantillons, la précision du modèle chute légèrement. Cela peut être expliqué par le nombre d'epochs qui n'est peut-être plus assez suffisant pour laisser au modèle le temps de converger correctement. Les résultats sans **TFR** (avec les données discrètes) sont plutôt encourageants eux aussi. Bien qu'ils soient légèrement moins bons que les résultats avec **TFR**, cela reste une piste à ne pas écarter et à continuer de chercher de ce côté là. Il serait en effet intéressant pour le temps réel de ne pas avoir à passer par une **TFR**.

Les résultats du **PMC** m'ont assez surpris, je ne m'attendais pas à autant de performance de la part de cette architecture. Mais finalement cela peut s'expliquer par le fait que le **PMC**

prend en entrée la composante du spectre continu, donc les différentes raies de la TFR, et ses raies se trouveront toujours autour des mêmes neurones. De plus comme l'ensemble de données est relativement petit, le **PMC** arrive finalement à apprendre la classification de chaque note et accord.

6.2. RNC

a. Architecture avec TFR

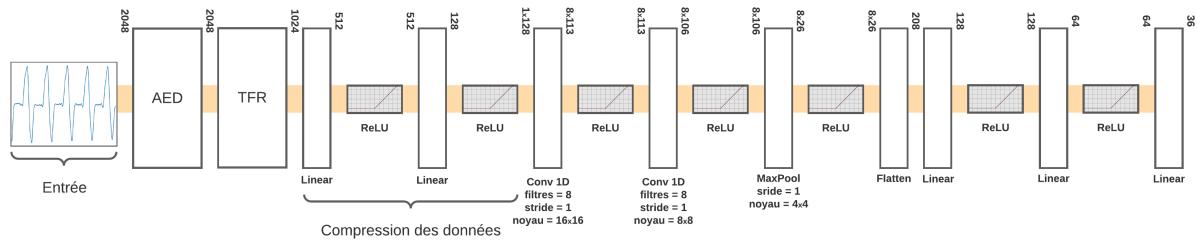


ILLUSTRATION 6.3 – Architecture RNC avec TFR. Source : Réalisé par KÜENZI Jean-Daniel

b. Matrice de confusion sur l'ensemble de validation

La matrice de confusion suivante a été réalisée avec le modèle utilisant une fenêtre de 2048 échantillons.

	C♯Maj	Dmin
Fondamentale	C♯	D
Tierce	F	F
Quinte juste	G♯	A

TABLEAU 6.3 – Comparaison de l'accord C♯Maj et Dmin. Source : Réalisé par KÜENZI Jean-Daniel

Dans mon ensemble de données, il se trouve que C♯Maj et Dmin sont dans la même octave, on peut donc en déduire qu'ils partagent la même tierce, à savoir F mais aussi que la fondamentale et la quinte de l'accord Dmin sont 1/2 ton plus élevé que celle de l'accord C♯Maj. On peut dès lors représenter ces notes par leurs valeurs de fréquences (en Hertz) plutôt que leurs noms afin d'observer les écarts entre les fréquences de ces deux accords.

	C♯Maj	Dmin	Écart en [Hz]
Fondamentale [Hz]	138.59	146.83	8.24
Tierce [Hz]	349.23	349.23	0.00
Quinte juste [Hz]	207.65	220	12.35

TABLEAU 6.4 – Comparaison de l'accord C♯Maj et Dmin sous forme de fréquences. Source : Réalisé par KÜENZI Jean-Daniel. A partir de *Wikipedia*, ref. URL11

Finalement cette erreur n'est pas si grave et plutôt intéressante, car on voit bien que les deux accords sont relativement proches et que peu de Hertz suffisent à les différencier. On comprend donc bien qu'avec une fenêtre de 2048 échantillons, soit une résolution fréquentielle de ~21.53[Hz], nous ne sommes pas du tout précis lorsque nous effectuons la TFR et que dans ce cas, le modèle a de la peine à bien différencier les deux accords à cause de leurs proximités. Vous pouvez retrouver en annexe (voir annexe E) la comparaison du spectre de Fourier entre C♯Maj et Dmin.

c. Entrainements pour différentes tailles de fenêtres

RNC avec TFR			
epochs = 50, η = 0.005, nombre d'entrainements = 10			
Nombre d'échantillons	Données	Précision moyenne %	Ecart type %
1024	Entrainement	99.00	0.35
	Validation	86.08	0.78
2048	Entrainement	99.81	0.01
	Validation	93.37	0.42
3072	Entrainement	99.82	0.06
	Validation	94.30	0.39
4096	Entrainement	99.91	0.02
	Validation	93.51	0.13

TABLEAU 6.5 – Entrainement de l'architecture RNC avec TFR. Source : Réalisé par KÜENZI Jean-Daniel

En observant ces résultats, on constate tout d'abord que l'architecture **RNC** avec **TFR** est légèrement moins précise que l'architecture **PMC** avec **TFR**. On remarque aussi qu'une fenêtre de 2048 échantillons n'est pas suffisante, l'architecture **RNC** a plutôt besoin d'une fenêtre de 3072 pour atteindre sa précision maximale. Même si les résultats sont légèrement moins bons que l'architecture **PMC** avec **TFR**, ils restent néanmoins plutôt corrects et intéressants. Donc 3072 échantillons semblent une bonne de taille fenêtre pour avoir une bonne précision, nous avons une résolution fréquentielle de ~14.35[Hz] et une résolution temporelle de ~0.069[s]. Toutefois, sa résolution temporelle est un peu élevée et la latence visuelle se fait un peu ressentir lors de l'utilisation en temps réel. Généralement, en dessous de ~0.025[s] l'œil ne perçoit pas la latence. Il vaut mieux donc utiliser une fenêtre de 2048 échantillons même si nous perdons un peu en précision afin de ne pas gêner l'utilisation en temps réel.

RNC sans TFRepochs = 50, η = 0.005, nombre d'entrainements = 10

Nombre d'échantillons	Données	Précision moyenne %	Ecart type %
1024	Entrainement	99.49	0.18
	Validation	90.28	0.41
2048	Entrainement	99.84	0.04
	Validation	91.51	0.42
3072	Entrainement	99.89	0.05
	Validation	90.78	0.30
4096	Entrainement	99.85	0.06
	Validation	89.54	0.58

TABLEAU 6.6 – Entrainement de l'architecture RNC sans TFR. Source : Réalisé par KÜENZI Jean-Daniel

En observant ces résultats, on constate tout d'abord que l'architecture **RNC sans TFR** est moins précise que l'architecture **PMC sans TFR**. On constate également que la fenêtre avec 2048 échantillons est celle qui a donné les meilleurs résultats. Finalement on peut observer qu'à partir de la fenêtre avec 2048 échantillons, l'augmentation du nombre d'échantillons fait chuter la précision du modèle. Cela est sûrement dû aux mêmes raisons que pour l'architecture **PMC sans TFR**.

Finalement, l'architecture convolutive est celle qui a donné les moins bons résultats, c'est pourquoi je ne l'ai pas retenue comme ma solution finale. Toutefois, ses résultats restent corrects et encourageants, cette architecture n'est donc pas à négliger.

CHAPITRE 7 : RÉSULTATS

7.1. AED

a. Entrainements pour différentes tailles de fenêtres

AED			
epochs = 50, η = 0.005, nombre d'entrainements = 10			
Nombre d'échantillons	Données	Valeur de perte moyenne	Ecart type
1024	Entrainement	0.0094	0.0038
	Validation	0.0096	0.0076
2048	Entrainement	0.0096	0.0018
	Validation	0.0083	0.0014
3072	Entrainement	0.0088	0.0025
	Validation	0.0084	0.0014
4096	Entrainement	0.0089	0.0010
	Validation	0.0085	0.0013

TABLEAU 7.1 – Entrainement de l’AED pour différentes tailles de fenêtres. Source : Réalisé par KÜENZI Jean-Daniel

Lorsqu'on observe ces résultats, on remarque que l’AED a eu plus de peine à reconstruire les données non bruitées quand on utilise une fenêtre de 1024 échantillons. Cela peut être dû au fait que le bruit blanc gaussien ajouté est trop fort pour la taille de cette fenêtre. Toutefois, la valeur de perte moyenne reste assez faible et les données bruitées sont plutôt bien reconstruites sans le bruit. On remarque aussi qu'à partir d'une fenêtre de 2048 échantillons, l'augmentation du nombre d'échantillons ne fait pas augmenter cette valeur de perte, ce qui est positif, une valeur de perte plus grande veut dire une reconstruction des données moins bonne.

b. Reconstruction d'une donnée non bruitée

Dans cette partie, nous allons voir comment l’AED se comporte lorsqu'il reçoit une donnée non bruitée en entrée. Pour cet exemple j'ai joué un A à 110 [Hz] avec le plectre.

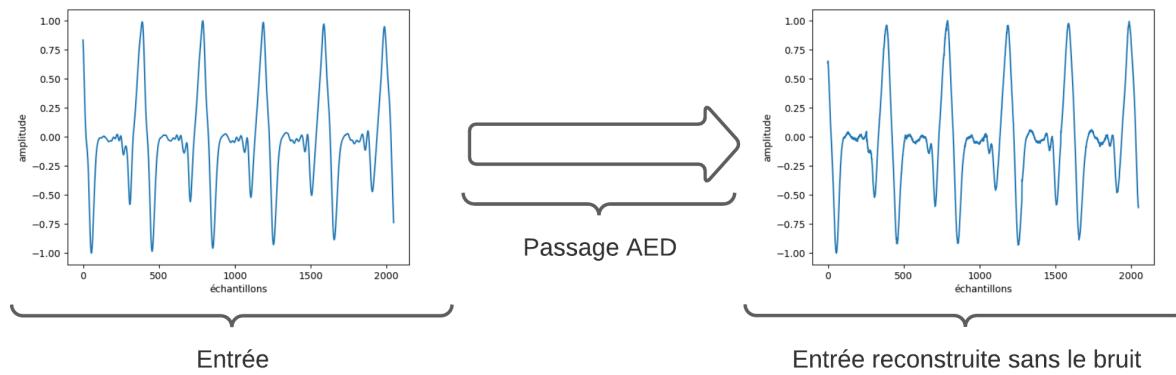


ILLUSTRATION 7.1 – Reconstruction d'un A à 110 [Hz] non bruité. Source : Réalisé par KÜENZI Jean-Daniel

On peut constater que le signal à la sortie de l'AED est quasiment le même qu'à l'entrée. Comme le signal d'entrée n'était pas bruité, l'AED s'est uniquement contenté de reconstruire le même signal. On peut retrouver les différents pics du signal aux bons endroits, et l'on remarque entre chaque pic qu'il y a une très légère différence entre l'entrée et la sortie.

c. Reconstruction d'une donnée avec du bruit blanc gaussien

Dans cette partie, nous allons voir comment l'AED se comporte lorsqu'il reçoit une donnée bruitée en entrée. Pour cet exemple, il s'agit du même A à 110 [Hz] de l'exemple précédent, mais avec du bruit blanc gaussien ajouté. Le bruit blanc gaussien qui a été ajouté a une moyenne de zéro et un écart type de zéro virgule trois. Ce bruit ajouté est comparable à un souffle constant assez fort.

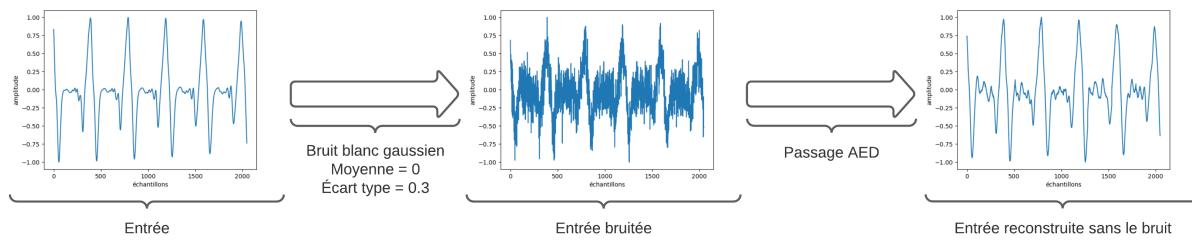


ILLUSTRATION 7.2 – Reconstruction d'un A à 110 [Hz] avec du bruit blanc gaussien. Source : Réalisé par KÜENZI Jean-Daniel

On peut constater que le signal à la sortie de l'AED est semblable à celui qui est à l'entrée, mais légèrement différent. Comme le signal d'entrée était bruité, l'AED a correctement extrait les caractéristiques nécessaires pour reconstruire le signal sans le bruit. On constate également que le signal reconstruit est quasiment le même que le signal qui se trouve en entrée. On peut

retrouver les différents pics du signal aux bons endroits, et on remarque entre chaque pic, qu'il y a une différence entre l'entrée et la sortie.

d. Reconstruction d'une donnée avec du gain

Dans cette partie nous allons voir comment l'**AED** se comporte lorsqu'il reçoit une donnée bruitée avec du gain en entrée. Pour cet exemple j'ai joué un Dmin avec la basse à ~146,83[Hz]. Pour ajouter le gain dans le signal, j'ai utilisé un amplificateur ASLIN DANE AG-15R. Toutefois, l'amplificateur étant assez vieux je n'ai pas réussi à trouver sa fiche technique et donc de ce fait, pas pu déterminer le gain ajouté en décibels.

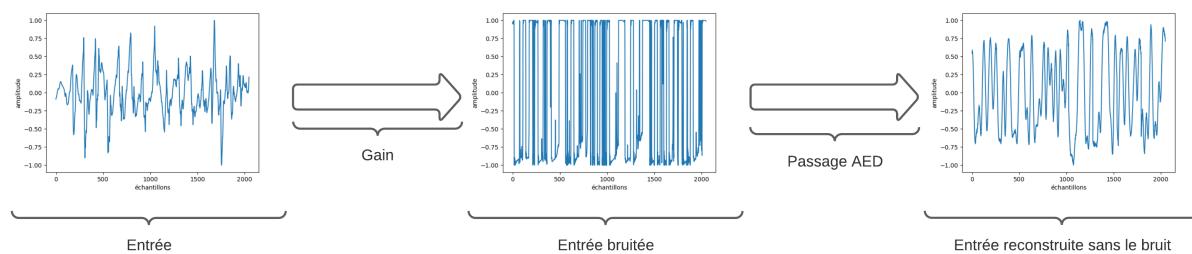


ILLUSTRATION 7.3 – Reconstruction d'un Dmin avec du gain. Source : Réalisé par KÜENZI Jean-Daniel

On peut constater que le signal à la sortie de l'**AED** est très différent de celui qui est en entrée. Cela est normal et compréhensible, l'**AED** a été entraîné à reconstruire les données sans du bruit blanc gaussien et non pas à reconstruire les données sans du gain. De plus, on observe que le gain ajouté est vraiment fort et qu'il déforme énormément le signal, il en devient presque carré. Toutefois, on remarque que la reconstruction des données est assez intéressante, l'**AED** a réduit les amplitudes et rendu le signal moins carré. J'ai également observé que l'utilisation de l'**AED** permettait au modèle de prédire beaucoup plus facilement l'accord avec du gain lors de l'utilisation en temps réel. Lorsque je n'utilisais pas l'**AED** et que le modèle prenait le signal brut avec le gain, le modèle prédisait surtout la quinte et la tierce mais rarement l'accord. Avec l'**AED**, il avait beaucoup plus de facilité à prédire l'accord.

7.2. LSTM-B

a. Matrice de confusion sur l'ensemble de validation

La matrice de confusion suivante a été réalisée avec le modèle utilisant une fenêtre de 2048 échantillons.

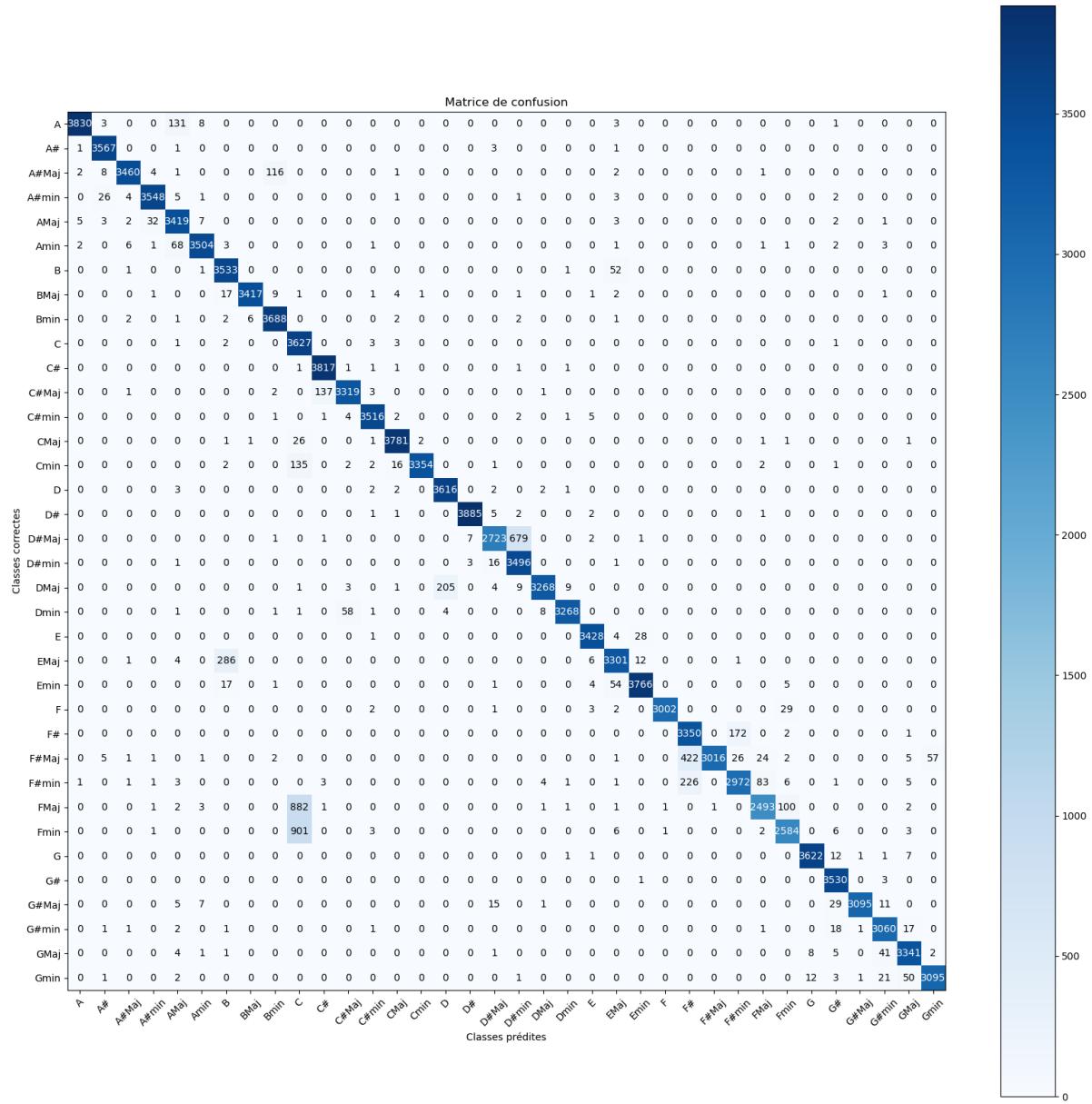


ILLUSTRATION 7.4 – Matrice de confusion pour l'architecture LSTM-B avec TFR. Source : Réalisé par KÜENZI Jean-Daniel

Tout d'abord, lorsque l'on observe la matrice de confusion, on peut constater le même problème qui est décrit dans la sous-section b de la section 6.1, c'est-à-dire que le modèle a prédit

certaine composante des accords, comme la fondamentale, la tierce ou la quinte. En fait cela n'est pas réellement une erreur et relève plutôt d'une imperfection dans l'ensemble de validation. Aussi on remarque que la grande majorité des données sont correctement classées (visible par la diagonale qui traverse la matrice). Toutefois, le modèle confond parfois les accords majeurs et mineurs.

Ainsi, comme décrit dans la sous-section b de la section 6.2, il confond les accords qui sont assez proches. C'est notamment le cas avec A#Maj que le modèle a confondu avec un Bmin. Finalement les résultats de l'architecture **LSTMB** avec **TFR** sont très similaires aux résultats obtenus avec l'architecture **PMC** avec **TFR** et sont dans leurs globalités excellents.

b. Entrainements pour différentes tailles de fenêtres

LSTMB avec TFR

epochs = 50, η = 0.005, nombre d'entrainements = 10

Nombre d'échantillons	Données	Précision moyenne %	Ecart type %
1024	Entrainement	99.53	0.11
	Validation	85.32	0.45
2048	Entrainement	99.81	0.01
	Validation	95.51	0.33
3072	Entrainement	99.88	0.02
	Validation	94.88	0.23
4096	Entrainement	99.91	0.01
	Validation	94.30	0.26

TABLEAU 7.2 – Entrainement de l'architecture LSTMB avec TFR. Source : Réalisé par KÜENZI Jean-Daniel

Lorsque l'on observe ces résultats, le premier constat est le même que pour l'architecture **PMC** et **RNC** avec **TFR**. Une fenêtre de 1024 échantillons n'est pas suffisante pour permettre une bonne prédiction des classes. Avec 1024 échantillons, nous avons une résolution fréquentielle de ~43.06[Hz]. Nous ne sommes donc pas du tout précis et cela va fortement impacter le modèle lors de la prédiction. Ensuite, on remarque que l'architecture **LSTMB** avec **TFR** nous a donné les meilleurs résultats avec une fenêtre de 2048 échantillons. Comme précisé dans la

section 3.7, les cellules LSTM sont très efficaces dans le traitement de données chronologiques. Finalement, on remarque qu'à partir de la fenêtre avec 2048 échantillons, augmenter le nombre d'échantillons fait chuter la précision. Cela peut être en partie dû à la réduction de la taille de l'ensemble de données ainsi qu'au problème évoqué dans la sous-section b.

LSTM sans TFR

epochs = 50, η = 0.005, nombre d'entrainements = 10

Nombre d'échantillons	Données	Précision moyenne %	Ecart type %
1024	Entrainement	99.69	0.15
	Validation	91.49	0.22
2048	Entrainement	99.91	0.02
	Validation	92.45	0.54
3072	Entrainement	99.94	0.05
	Validation	92.21	0.15
4096	Entrainement	99.94	0.03
	Validation	91.25	0.16

TABLEAU 7.3 – Entrainement de l'architecture LSTM sans TFR. Source : Réalisé par KÜENZI Jean-Daniel

En observant ces résultats, on constate tout d'abord qu'ils sont légèrement moins bons que ceux de l'architecture avec TFR, mais l'on remarque aussi que la fenêtre de 2048 échantillons donne le meilleur résultat entre les trois architectures sans TFR. Néanmoins, ces résultats sont très encourageants, on remarque que le modèle a une assez bonne précision avec les données discrètes. On remarque également qu'à partir de la fenêtre avec 2048 échantillons, l'augmentation du nombre d'échantillons fait légèrement chuter la précision du modèle. Cela peut être dû au nombre d'epochs qui n'est plus suffisant pour laisser au réseau le temps de bien converger ou à la réduction de la taille de l'ensemble de données ainsi qu'au problème évoqué dans la sous-section b.

7.3. RESSENTI D'UTILISATION EN TEMPS RÉEL

Lors de mon utilisation en temps réel, j'ai tout d'abord constaté que les accords et notes joués étaient correctement prédits. De plus, j'ai mesuré le temps passé à traverser le modèle, avec une utilisation en CPU, depuis le moment où nous avons nos 2048 échantillons jusqu'à

l'obtention de la prédiction de l'accord ou de la note. En moyenne, cela prend entre trois et quatre millisecondes, nous sommes donc bien en dessous des cinq millisecondes visées. De plus, lors de l'utilisation en temps réel le modèle prend en entrée une fenêtre de 2048 échantillons, ce qui représente une durée de $\sim 0.046[\text{s}]$. Mais j'utilise une taille de chevauchement de 1024 échantillons, soit $\sim 0.023[\text{s}]$, ce qui fait que la toutes premières fenêtre aura une latence de $\sim 0.046[\text{s}]$, mais les autres fenêtres auront une latence divisée par un facteur de deux grâce au chevauchement, soit une latence de $\sim 0.023[\text{s}]$. En somme, on utilise les 1024 derniers échantillons de la fenêtre à l'instant $t - 1$ pour construire notre fenêtre à l'instant t . La latence visuelle est donc presque imperceptible et l'impression du temps réel se fait bien ressentir.

Ensuite, concernant l'attaque, elle est légèrement visible au début de la prédiction, c'est-à-dire lors de la première fenêtre. Elle est donc uniquement visible pendant $\sim 0.046[\text{s}]$ ce qui n'est pas très gênant.

J'ai également constaté que le modèle ne semble pas avoir de problèmes avec les changements de phases dans les signaux. J'ai notamment testé différentes techniques de legato (hammer-on, pull-off, etc.), c'est-à-dire que j'ai lié les notes jouées sans laisser de silence entre elles et sans réattaquer la corde avec le plectre ou le doigt. Ce qui est intéressant avec ces techniques, c'est qu'elles vont créer un changement de phase brusque dans le signal, étant donné que l'on va par exemple passer d'un F à un F \sharp ou un G de manière instantanée.

J'ai aussi testé différends bends (technique à la guitare), c'est-à-dire que j'ai volontairement tiré latéralement la corde sur laquelle je jouais pour augmenter sa vibration et ainsi atteindre une note différente (par exemple pour atteindre un G depuis un F). Cette technique est très intéressante, car elle nous fait passer par toutes les fréquences entre la note de base et la note visée. Par exemple pour passer d'un F à un G dans la même octave, nous allons passer par toutes la bande de fréquence [174.61 ; 196.00]. C'est donc un bon moyen de déterminer si le modèle est sensible aux changements de phases.

J'ai constaté de plus que le modèle a plus de facilité à détecter les notes que les accords, ce qui est compréhensible étant donné que le spectre de Fourier d'une note, et donc d'un signal, est beaucoup moins complexe que celui d'un accord qui est une agrégation de plusieurs notes, et donc signaux différents.

Pour finir, j'ai rajouté la fondamentale et la quinte plusieurs fois à l'octave dans l'accord et donc de ne plus jouer un accord à trois sons, comme au départ, mais de jouer un accord avec

six sons. Par exemple l'accord EMaj est joué avec les notes dans l'ordre suivant :

A Majeur			
	Notes	Octave	Fréquence en [Hz]
Fondamentale	E	1	82.41
Quinte juste	B	1	123.47
Fondamentale	E	2	164.81
Tierce majeure	G♯	2	207.65
Quinte	B	2	246.94
Fondamentale	E	3	329.63

TABLEAU 7.4 – Accord AMaj avec la quinte et la fondamentale sur plusieurs octaves. Source : Réalisé par KÜENZI Jean-Daniel. A partir de *Wikipedia*, ref. URL11

Le fait de rajouter des composantes comme la fondamentale et la quinte à l'octave ne gêne pas les prédictions du modèle et cela peut s'expliquer par le fait que sur le spectre de Fourier, nous allons retrouver les trois composantes de l'accord à trois sons, à savoir la fondamentale, la tierce et la quinte juste. De plus, comme les composantes rajoutées sont des multiples des trois composantes de bases, cela correspond donc aux partiels harmoniques qui étaient déjà présents sur le spectre, mais cette fois elles seront plus fortes et vont à leurs tours rajouter leur propre partiels harmoniques.

CONCLUSION

Dans un premier temps, le but de ce travail était de pouvoir détecter automatiquement des accords et notes de guitare à l'aide de différents modèles connexionnistes. Je devais donc mettre en place différentes architectures afin de comparer leurs résultats et je devais également construire mon propre ensemble de données en partant de zéro. Pour ce projet, je devais également considérer les signaux dans le monde discret et dans le monde fréquentiel. Dans un deuxième temps, je devais déterminer si les modèles étaient capables de fonctionner en temps réel.

Nous avons vu au cours de cette thèse que j'ai décidé d'implémenter et tester trois architectures, à savoir l'architecture **PMC**, l'architecture **RNC** et l'architecture **LSTMB**. Nous avons pu constater et comparer les résultats de ses trois architectures qui sont très satisfaisants. Bien que l'architecture **RNC** soit celle qui ait fonctionné le moins bien, les résultats restent très corrects. Nous avons également vu que pour passer du monde discret au monde fréquentiel j'ai choisi d'utiliser la **TFR** pour sa simplicité et sa rapidité. Ce qui est un facteur essentiel pour l'utilisation en temps réel. Nous avons pu observer de plus que les modèles utilisant la **TFR** étaient plus efficace que ceux utilisant directement les données discrètes. Néanmoins, les résultats obtenus avec les données discrètes restent très encourageants et intéressants à explorer.

Nous avons vu aussi que j'ai dû mettre en place et construire mon propre ensemble de données. C'était très intéressant, car dans un premier temps j'avais fait mes enregistrements sans matériel spécialisé ce qui avait un impact néfaste sur mon ensemble de données, notamment au niveau du bruit dans le signal, et que cet impact se faisait ressentir sur les performances de mes modèles. J'ai ensuite eu accès à une carte son externe spécialisée dans le traitement audio, ce qui m'a permis, sur la base de mon ancien ensemble de données, d'en reconstruire un nouveau avec des données non bruitées et des structures de fichiers complètement différentes. Avec le nouvel ensemble de données, j'ai pu constater une nette amélioration dans l'entraînement des différents modèles.

Suite à ça, j'ai eu l'idée de construire un **AED** afin de réduire le bruit dans les signaux que je recevais en entrée. Tout le monde n'ayant pas forcément de matériel spécialisé, il était donc nécessaire de pouvoir réduire le bruit avant de faire passer les entrées dans les modèles prédictifs. Cet **AED** peut être aussi utile pour réduire les effets comme le gain, ce qui aide le modèle prédictif à prédire correctement.

Finalement, sur la base de mon ressenti d'utilisation en temps réel, nous avons pu constater que le modèle n'était pas sensible au problème du changement de phase dans les signaux et qu'il prédisait correctement les différents accords et notes joués. Nous avons également constaté que le chevauchement des fenêtres a réduit le temps de latence par deux ; à savoir, un temps de latence de ~0.023[s], ce qui rend l'utilisation en temps réel très fluide et agréable. De ce fait, le temps de latence visuelle n'est quant à lui presque pas perceptible. De plus, nous avons observé que la solution proposée, à savoir l'architecture **LSTMB** avec **TFR** était en moyenne traversée en trois à quatre millisecondes, ce qui est bien inférieur aux cinq millisecondes souhaitées.

En tant que passionné de musique et étant moi-même musicien c'était un réel plaisir de travailler sur ce projet. Les recherches effectuées durant ce travail m'ont apporté beaucoup de connaissances sur les notions mathématiques liées au traitement du signal. Notamment comme la fuite spectrale lors de la **TFR**, l'utilisation de fenêtre pour la réduire et le théorème d'échantillonnage de Nyquist-Shanon. J'ai aussi pu approfondir mes connaissances sur les réseaux de neurones en découvrant les cellules **LSTM**, la normalisation par lots, les auto-encodeurs et leurs variantes ainsi que les modèles transformer. J'ai de plus pu constater l'importance et la complexité d'avoir un ensemble de données étudiées et correctement construit et donc l'importance et l'implication des "Data Scientist" dans le domaine de l'**IA**. Mais ce travail m'a surtout permis de percevoir la musique dans un contexte purement physique et mathématique et de comprendre réellement ce qu'est la musique à la base.

Bien que la **TFR** nous ait donné de bons résultats sur notre ensemble de données, il pourrait y avoir un problème avec la résolution fréquentielle. En effet, plus on descend dans les basses fréquences et plus la hauteur des notes (c'est-à-dire leur fréquence) est rapprochée (1[Hz] peut suffire à différencier un C d'un C \sharp). Inversement, plus on monte dans les hautes fréquences, plus la hauteur des notes est éloignée (1[kHz] peut suffire à différencier un C d'un C \sharp). On remarque donc que dans les basses fréquences nous avons besoin d'une plus grande taille de fenêtre que dans les hautes fréquences. Mais la **TFR** utilise une taille fixe. Il serait donc intéressant de voir si l'on peut faire varier cette taille de fenêtre automatiquement ou utiliser d'autres fonctions comme la **TFCT**, la Transformée en Ondelette ou encore la Transformée à Q constant (**TQC**) et de voir si cela améliore la précision des différents modèles.

On peut voir que j'ai construit mon **AED** pour qu'il fonctionne dans le monde temporel (avec le signal discret). Il pourrait être intéressant de le considérer dans le monde des fréquences.

Ainsi, on pourrait voir si la reconstruction des données non bruitées est plus efficace et plus rapide.

Actuellement, l'ensemble de données créé se repose uniquement sur les accords à trois sons mineurs et majeurs ainsi que les notes sur l'accordage standard de la guitare. Dans un premier temps, augmenter l'ensemble de données en rajoutant des accords septièmes, diminués et augmentés pourrait permettre de voir si les architectures actuelles sont suffisantes pour différencier correctement les différents accords et notes. Ensuite, il serait intéressant de continuer l'ensemble de données dans d'autres types d'accordages afin de généraliser le problème et les modèles.

Un autre point important est que l'ensemble de données actuel, bien qu'il soit petit, m'a pris beaucoup de temps à faire. On imagine donc bien que pour un ensemble de données qui contiendrait plusieurs types d'accords et d'accordages cela deviendrait trop lourd à construire. On pourrait donc essayer d'utiliser un modèle génératif et de faire de l'Apprentissage par Transfert (*Transfer Learning* en anglais) pour pallier à ce problème. L'apprentissage par transfert consiste à utiliser un réseau préentraîné sur un ensemble de données sources qui contient des données similaires aux nôtres (par exemple un ensemble de données sur des accords et notes de piano) puis à faire du Fine Tuning avec notre ensemble de données (qu'on appelle ensemble de données de transfert) où l'on va uniquement modifier les couches denses du réseau. Ainsi, nous pouvons avoir un modèle qui va nous générer des données et de ce fait augmenter la taille de notre ensemble de données. On peut notamment penser à des modèles comme WaveNet (9) ou encore Wave2Vec (10).

Pour finir, nous allons un peu parler des modèles transformer. Ce sont des modèles qui ont changé la manière dont on faisait du Traitement Automatique du Langage Naturel (TALN) et qui commencent à changer d'autres domaines de l'intelligence artificielle. Notamment avec "AlphaFold 2", une intelligence artificielle qui prédit la structure 3D des protéines (11). Les modèles transformer ont été introduits en 2017 avec le document "Attention is all you need" (12). On pourrait donc essayer de voir si les modèles transformer peuvent apporter une amélioration dans le domaine du traitement chronologique et dans notre cas, la prédiction d'accords et de notes de musique.

Annexes

ANNEXE A : ARCHITECTURE LSTMB SANS TFR

A.1. ARCHITECTURE

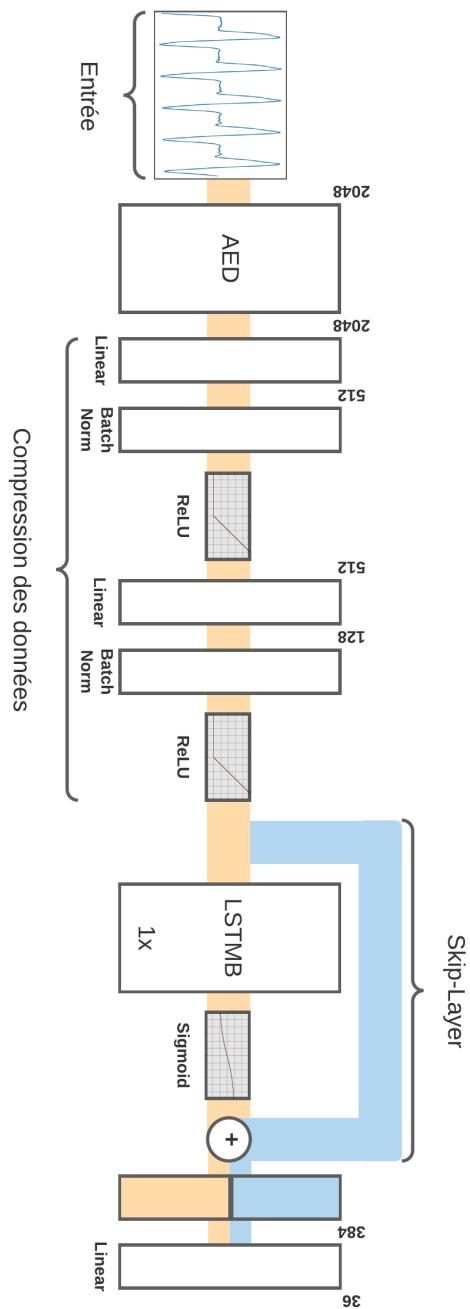


ILLUSTRATION A.1 – Architecture LSTMB sans TFR. Source : Réalisé par KÜENZI Jean-Daniel

A.2. MATRICE DE CONFUSION SUR L'ENSEMBLE DE VALIDATION

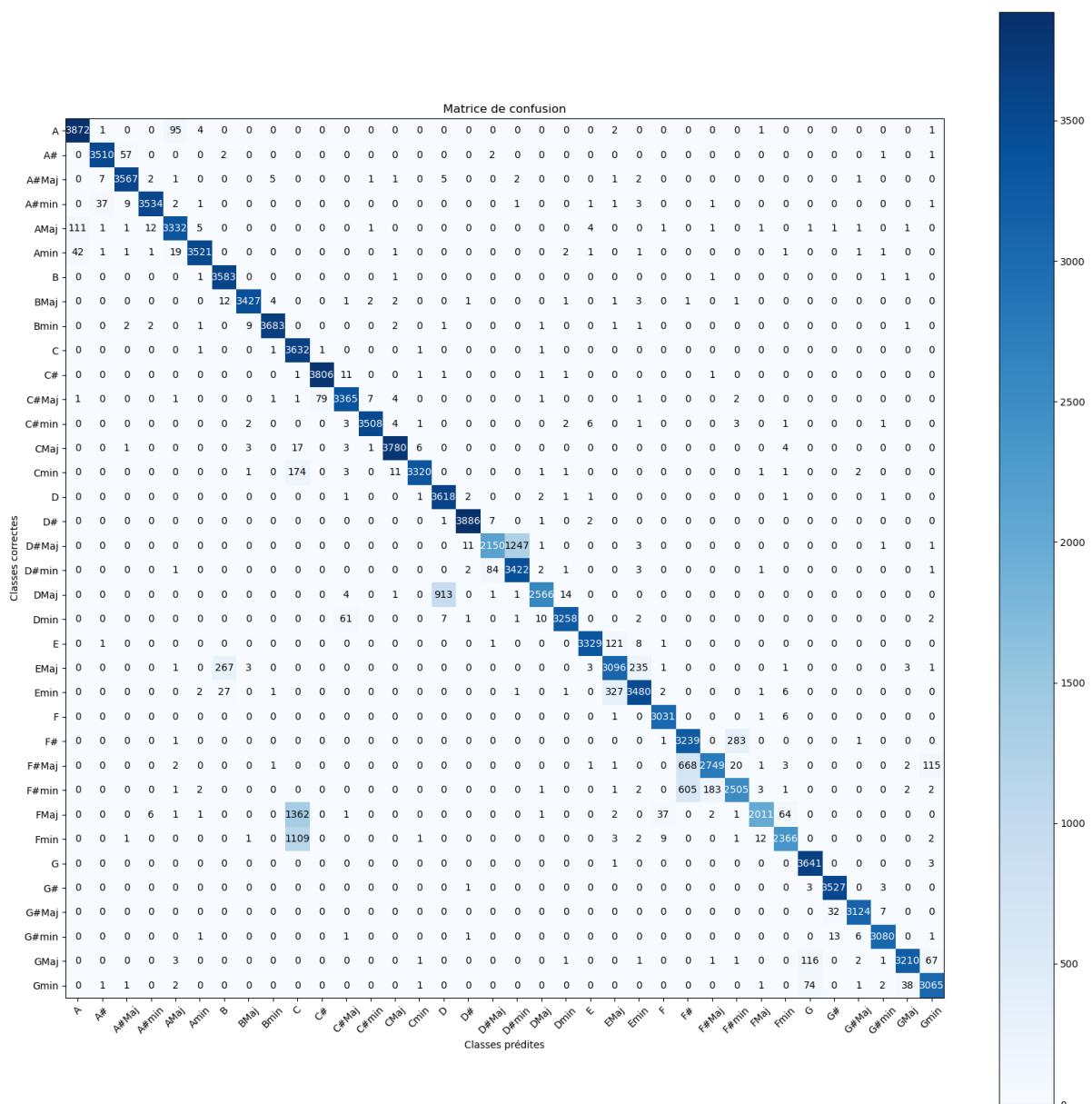


ILLUSTRATION A.2 – Matrice de confusion pour l'architecture LSTM sans TFR. Source : Réalisé par KÜENZI Jean-Daniel

ANNEXE B : ARCHITECTURE PMC SANS TFR

B.1. ARCHITECTURE

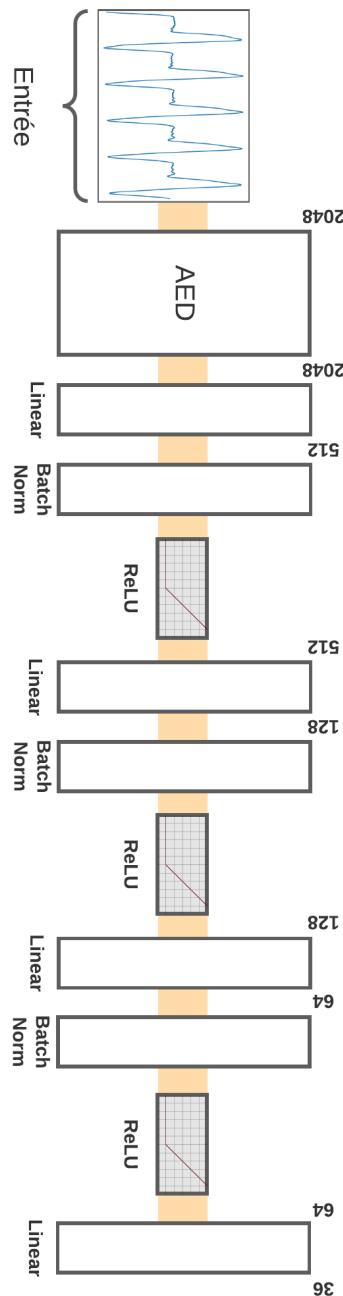


ILLUSTRATION B.1 – Architecture PMC sans TFR. Source : Réalisé par KÜENZI Jean-Daniel

B.2. MATRICE DE CONFUSION SUR L'ENSEMBLE DE VALIDATION

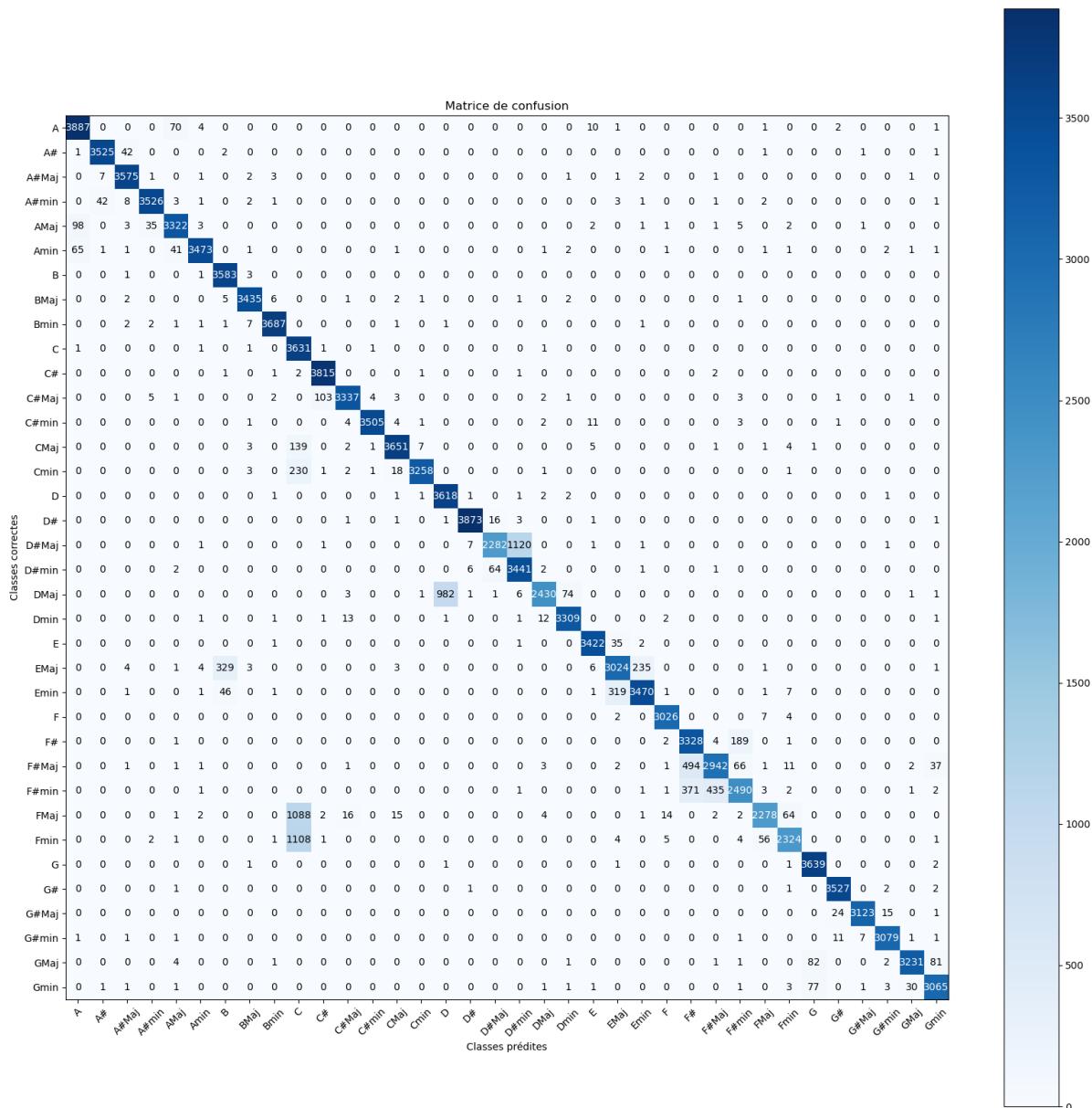


ILLUSTRATION B.2 – Matrice de confusion pour l'architecture PMC sans TFR. Source : Réalisé par KÜENZI Jean-Daniel

ANNEXE C : ARCHITECTURE RNC SANS TFR

C.1. ARCHITECTURE

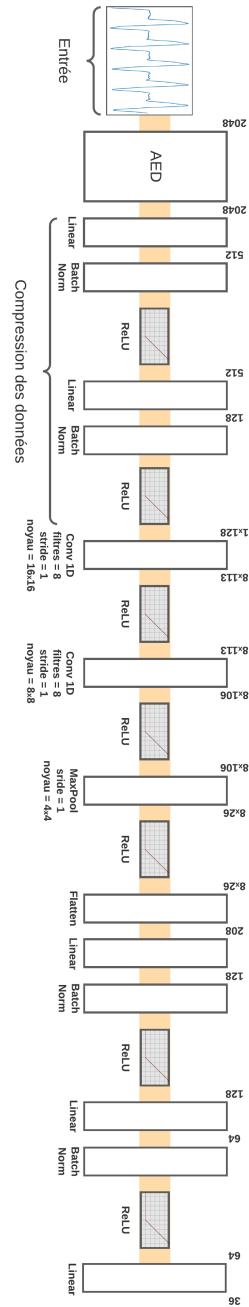


ILLUSTRATION C.1 – Architecture RNC sans TFR. Source : Réalisé par KÜENZI Jean-Daniel

ANNEXE D : PRÉDICTON DE L'ARCHITECTURE PMC SUR LE FICHIER FMAJ2 DE L'ENSEMBLE DE VALIDATION

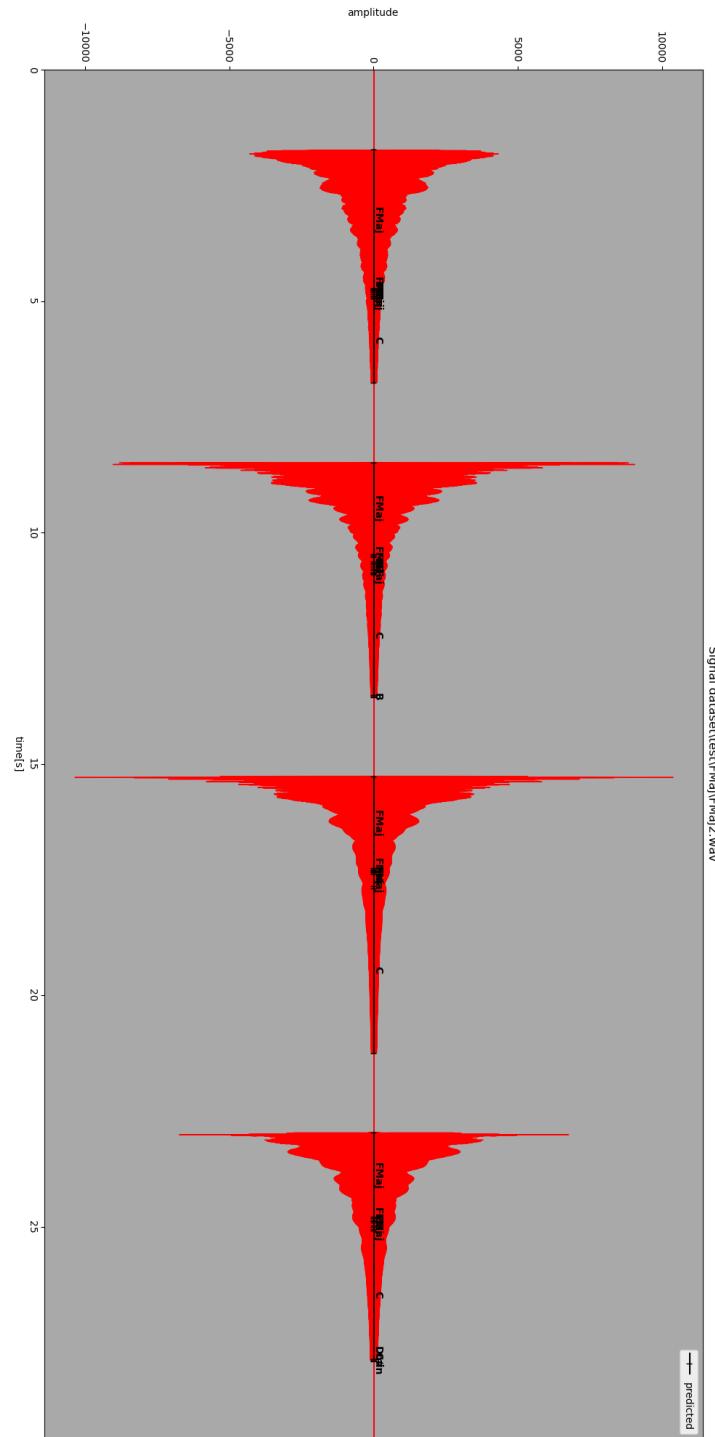


ILLUSTRATION D.1 – Prédiction de l'architecture PMC sur le fichier FMaj2.wav du dataset de test. Source : Réalisé par KÜENZI Jean-Daniel

ANNEXE E : COMPARAISON DU SPECTRE DE FOURIER DE C#MAJ ET DMIN DANS LA MÊME OCTAVE

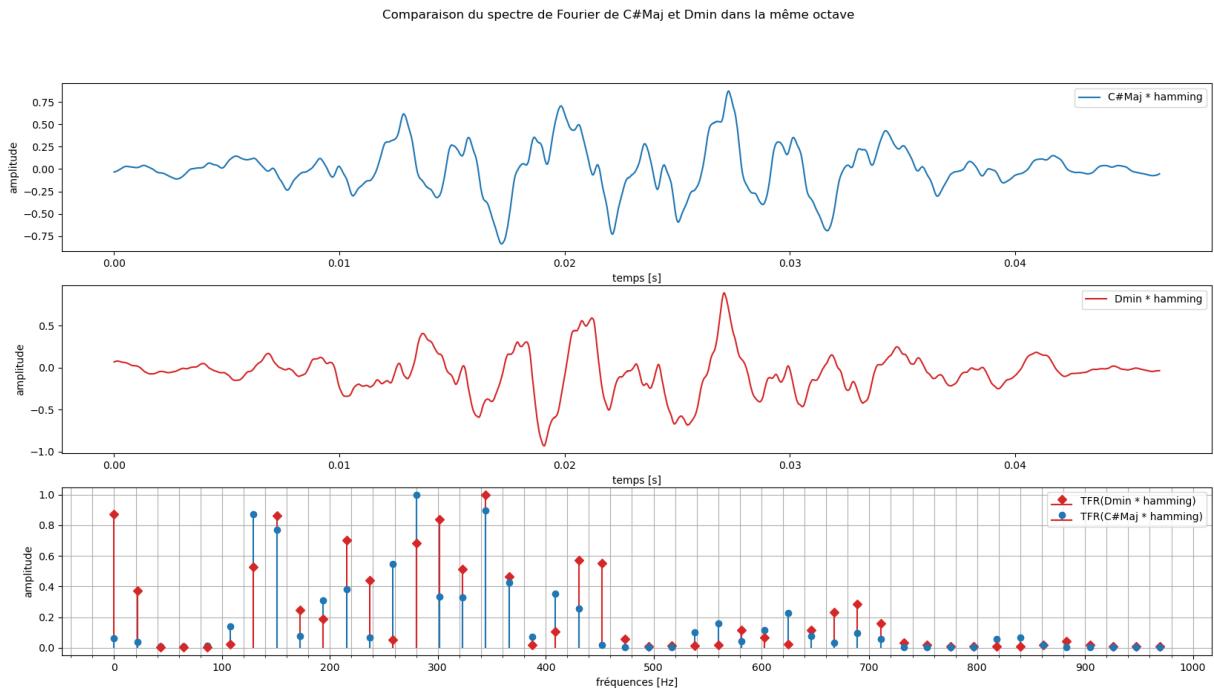


ILLUSTRATION E.1 – Comparaison du spectre de Fourier de C#Maj et Dmin dans la même octave. Source : Réalisé par KÜENZI Jean-Daniel

RÉFÉRENCES DOCUMENTAIRES

1. PEETERS, Geoffroy. ENSEA 3ème SyM (2017-2018) Traitement du signal audio musical Partie 1 : transformation et séparation du son [en ligne]. 2017, p. 79 [visité le 2021-05-17]. Disp. à l'adr. : http://recherche.ircam.fr/anasyneeters/pub/cours/20171010_Peeters_20172018_ENSEA_3SYM_Cours_Transformation.pdf.
2. Harmonique (musique). In : *Wikipédia* [en ligne]. 2021 [visité le 2021-05-15]. Disp. à l'adr. : [https://fr.wikipedia.org/w/index.php?title=Harmonique_\(musique\)&oldid=180763506](https://fr.wikipedia.org/w/index.php?title=Harmonique_(musique)&oldid=180763506). Page Version ID : 180763506.
3. Théorème d'échantillonnage. In : *Wikipédia* [en ligne]. 2021 [visité le 2021-05-14]. Disp. à l'adr. : https://fr.wikipedia.org/w/index.php?title=Th%C3%A9or%C3%A8me_d%C3%A9chantillonnage&oldid=183959686. Page Version ID : 183959686.
4. BANK, Dor; KOENIGSTEIN, Noam; GIRYES, Raja. Autoencoders. *arXiv:2003.05991 [cs, stat]* [en ligne]. 2021 [visité le 2021-06-15]. Disp. à l'adr. arXiv: 2003.05991.
5. IOFFE, Sergey; SZEGEDY, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs]* [en ligne]. 2015 [visité le 2021-07-08]. Disp. à l'adr. arXiv: 1502.03167.
6. SANTURKAR, Shibani; TSIPRAS, Dimitris; ILYAS, Andrew; MADRY, Aleksander. How Does Batch Normalization Help Optimization? *arXiv:1805.11604 [cs, stat]* [en ligne]. 2019 [visité le 2021-07-07]. Disp. à l'adr. arXiv: 1805.11604.
7. *BatchNorm1d — PyTorch 1.8.1 documentation* [en ligne] [visité le 2021-07-07]. Disp. à l'adr.: <https://pytorch.org/docs/1.8.1/generated/torch.nn.BatchNorm1d.html?highlight=batchnorm#torch.nn.BatchNorm1d>.
8. *CrossEntropyLoss — PyTorch 1.8.1 documentation* [en ligne] [visité le 2021-07-09]. Disp. à l'adr.: <https://pytorch.org/docs/1.8.1/generated/torch.nn.CrossEntropyLoss.html#torch.nn.CrossEntropyLoss>.
9. OORD, Aaron van den; DIELEMAN, Sander; ZEN, Heiga; [et al.] WaveNet: A Generative Model for Raw Audio. *arXiv:1609.03499 [cs]* [en ligne]. 2016 [visité le 2021-07-21]. Disp. à l'adr. arXiv: 1609.03499.

10. BAEVSKI, Alexei; ZHOU, Henry; MOHAMED, Abdelrahman; AULI, Michael. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. *arXiv:2006.11477 [cs, eess]* [en ligne]. 2020 [visité le 2021-07-21]. Disp. à l'adr. arXiv: 2006.11477.
11. *AlphaFold : a solution to a 50-year-old grand challenge in biology* [Deepmind] [en ligne] [visité le 2021-07-20]. Disp. à l'adr. : <https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology>.
12. VASWANI, Ashish; SHAZER, Noam; PARMAR, Niki; [et al.] Attention Is All You Need. *arXiv:1706.03762 [cs]* [en ligne]. 2017 [visité le 2021-07-20]. Disp. à l'adr. arXiv: 1706.03762.