



Université de Toulouse
Double Licence de Mathématiques et Informatique

La Synthèse Audio Inverse : Du Spectrogramme aux Paramètres

stage de recherche effectué par

PARODI Pierre

encadré par

MAUCLAIR Julie

réalisé le 12 juin 2026

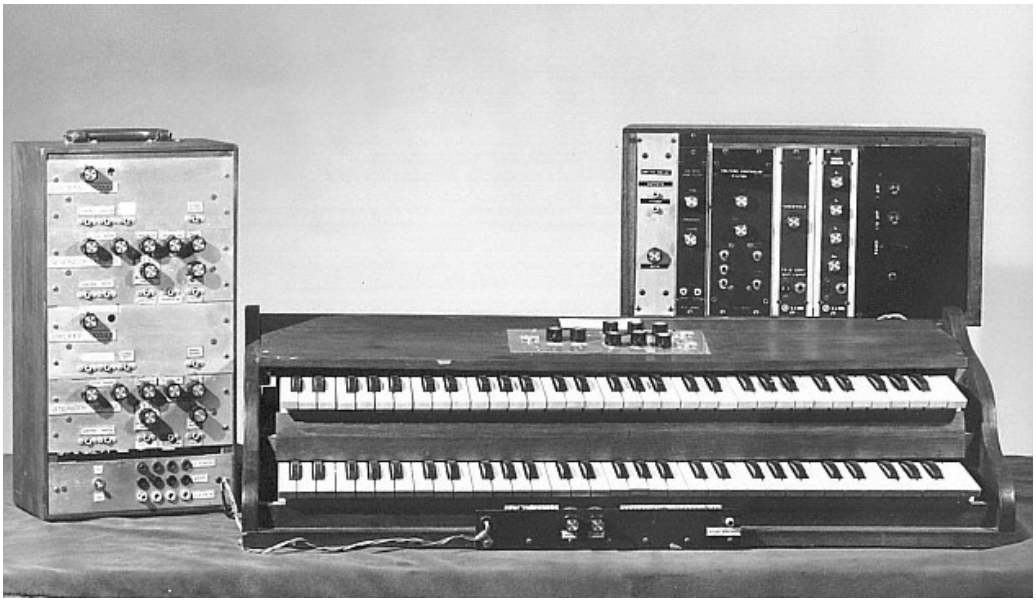


Photo du tout premier prototype modulaire développé par Robert Moog et Herbert Deutsch à l'été 1964 [1]

« La grande variété de sons qui peuvent être aisément produits par voie électronique [...] offre au compositeur une immense palette de nouveaux effets musicaux. [...] Le compositeur paie cependant le prix fort pour ces avantages supplémentaires. Avec une multitude de paramètres sonores sous son contrôle, le compositeur doit devenir expert dans la manipulation de son appareillage électronique pour qu'il produise les résultats désirés. La programmation et la manipulation de la plupart des équipements de musique électronique actuels tendent à devenir laborieuses et chronophages. »

— Robert A. Moog, *Voltage-Controlled Electronic Music Modules*, 1965. [2]

Table des matières

I Introduction et Définition du Projet **8**
I.1 Le problème de la synthèse inverse 8
I.2 Définition des objectifs 8
I.3 Le cadre institutionnel et l'équipe de recherche 9
I.4 Planification et démarche d'ingénierie 9

II Conception du Moteur de Synthèse **10**
II.1 Choix de l'environnement de développement : le pivot vers Python 10
II.2 L'écosystème Python : Pyo pour l'audio et PyTorch pour l'apprentissage 11
II.3 Architecture du synthétiseur soustractif sous Pyo 11

III Création du dataset **14**
III.1 Éviter les incohérences acoustiques et de données 14
III.2 Modélisation de la perception humaine : Les distributions non-linéaires 15
III.3 Règles de filtrage dynamiques 16
III.4 Validation automatique et *Sanity Checks* 18
Récapitulatif 19

IV Prétraitement et Choix de l'Architecture Neuronale **20**
IV.1 Du domaine temporel au Spectrogramme Log-Mel 20
IV.2 L'architecture & son Implémentation : Réseaux Convolutifs (CNN) pour l'audio 23

V Résultats, Évaluation et Analyse Critique **27**
V.1 Classification des Filtres 27
V.2 Classification des Oscillateurs 27
Conclusion 28

VI Bibliographie **29**

VII Annexes **30**

Remerciements

Je tiens tout d'abord à remercier avec sincérité ma tutrice de stage, Madame Julie Mauclair, pour sa confiance, sa disponibilité et ses précieux conseils tout au long de ce projet au sein de l'équipe SAMoVA. Son expertise technique et son regard critique m'ont permis de prendre le recul nécessaire face aux nombreux défis de l'apprentissage profond appliqué au traitement du signal. Plus encore, la grande liberté d'exploration qu'elle m'a accordée vis-à-vis du sujet a été un véritable moteur ; grâce à elle, ce stage a définitivement conforté mon ambition de poursuivre mon projet professionnel dans le domaine du traitement du signal et de l'image appliqués à l'audio.

Je souhaite également exprimer toute ma reconnaissance à Madame Armelle Bonenfant. Ayant été mon tout premier point de contact, elle a su m'orienter avec réactivité et bienveillance vers l'équipe SAMoVA et Madame Mauclair, rendant ainsi cette formidable opportunité de stage possible.

Mes remerciements s'adressent enfin à l'ensemble des membres de l'Institut de Recherche en Informatique de Toulouse (IRIT) pour m'avoir accueilli dans un environnement scientifique aussi stimulant.

I. Introduction et Définition du Projet

I.1 Le problème de la synthèse inverse

La conception sonore (*sound design*) est un art complexe qui requiert à la fois une oreille exercée et une maîtrise technique des instruments électroniques. Même pour quelqu'un d'aguerri, reproduire une sonorité spécifique à l'aide d'un synthétiseur nécessite de naviguer à taton sur l'instrument. Une combinaision de petits changements sur différents paramètres peut produire un tout nouveau son, que l'artiste peut découvrir par surprise. Ce processus appelé *preset matching* peut être long, parfois chaotique, et dépend de l'expérience que s'est fait l'utilisateur sur l'instrument.

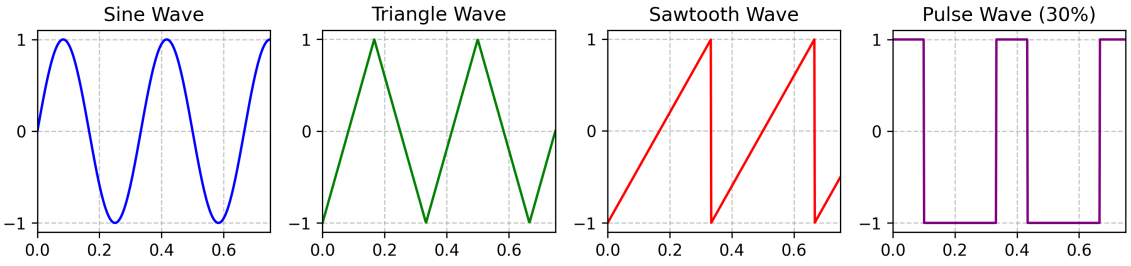
Avec l'avènement des architectures d'apprentissage profond, l'automatisation de ce processus est devenue un champ de recherche particulièrement actif. Le défi de la **synthèse audio inverse** consiste à développer un modèle capable d'écouter un fichier audio et d'en déduire instantanément les paramètres du synthétiseur qui l'a généré. Il s'agit d'un problème d'ingénierie inverse particulièrement ardu en raison de la non-linéarité des paramètres acoustiques et du problème dit *many-to-one*, où plusieurs configurations différentes peuvent produire des sons perceptuellement identiques.

I.2 Définition des objectifs

Afin de concevoir une architecture viable et de maîtriser correctement la synthèse ainsi que la prédiction, il a été nécessaire de délimiter le périmètre de cette étude. Plutôt que de tenter de prédire les centaines de paramètres d'un synthétiseur open source complexe (*Surge XT* [3] à titre d'exemple), ce projet se concentre sur un synthétiseur soustractif assez simpliste conçu sur mesure (oscillateur → filtre → enveloppe).

L'objectif premier est de construire un réseau de neurones capable d'inférer deux grandes familles de paramètres à partir du son généré par ce synthétiseur :

- **Les paramètres discrets :** Ils définissent la structure de base du son. Il s'agit notamment de la forme d'onde de l'oscillateur (*Sine Wave, Triangle, Saw Wave, Pulse, ...*) ou du type de filtre appliqué (*Low-Pass, High-Pass, Band-Pass*).
- **Les paramètres continus :** Ils sculptent le son dans le temps et l'espace fréquentiel. Ils agissent comme les "potentiomètres" du synthétiseur. Cela inclut la fréquence de coupure du filtre (*cutoff*), sa résonance, ou encore les temps d'enveloppe temporelle (*ADSR Envelope : Attack–Decay–Sustain–Release*).



I.3 Le cadre institutionnel et l'équipe de recherche

Le cadre du stage : L'IRIT et l'équipe SAMoVA

Ce stage de recherche s'est déroulé au sein de l'Institut de Recherche en Informatique de Toulouse (IRIT). Plus précisément, j'ai eu l'opportunité d'intégrer l'équipe SAMoVA (Structuration, Analyse et Modélisation de documents Vidéo et Audio), dont les travaux se concentrent sur l'analyse de données multimédias, le traitement du signal et l'intelligence artificielle.

Le choix de me tourner vers cette équipe s'est imposé comme une évidence vis-à-vis de mon sujet de recherche. Le défi de la synthèse audio inverse exige une double compétence très spécifique : une compréhension fine de la physique du son et une maîtrise des architectures d'apprentissage profond. L'équipe SAMoVA se trouvant à l'exacte intersection de ces deux disciplines, elle représentait l'environnement scientifique idéal pour mener à bien ce projet.

L'Encadrement

Durant cette période, j'ai eu l'honneur d'être encadré par Julie Mauclair, maître de conférences et enseignante-chercheuse au sein de l'équipe SAMoVA. Son expertise pointue a été un atout déterminant dans l'avancée de ce projet. Ses conseils m'ont particulièrement aidé à surmonter les obstacles liés à la modélisation de la perception humaine et à orienter mes choix algorithmiques, notamment sur l'implémentation d'un synthétisuer soustractif, lors du pivot vers l'utilisation des spectrogrammes pour l'entraînement du réseau de neurones mais aussi pour le choix de l'architecture de ce dernier.

I.4 Planification et démarche d'ingénierie

La réalisation de ce système prédictif ne se résume pas à l'entraînement d'un réseau de neurones. Elle nécessite la construction d'un procédé complet, allant de la génération acoustique à la prédiction neurale. Précisons d'emblée que le modèle prédictif n'aura bien sûr jamais accès aux paramètres de génération initiaux. La démarche adoptée pour ce projet, qui structure également ce rapport, s'articule autour des quatre grands axes suivants :

1. **L'implémentation du moteur de synthèse :** La première étape a consisté à concevoir et coder un synthétiseur en Python à l'aide de la librairie *Pyo* pour générer les données d'apprentissage.
2. **L'ingénierie des données et la création du dataset :** Pour entraîner un réseau de neurones sur ces données, il faut que celles-ci soit cohérentes. Ce chapitre détaillera alors les défis acoustiques rencontrés lors de la génération aléatoire de sons (gestion du bruit, des conflits entre oscillateurs et filtres) et les solutions mathématiques implémentées pour garantir une distribution spectrale équilibrée.
3. **Le prétraitement du signal :** Avant d'être fourni au réseau, le signal audio brut doit être transformé. je justifierai le passage dans le domaine fréquentiel via l'utilisation de spectrogrammes, permettant de transformer notre problématique audio en une tâche de vision par ordinateur.
4. **La conception et l'entraînement des réseaux de neurones :** Enfin, je présenterai l'architecture du Réseau de Neurones Convolutifs (CNN) choisie pour "observer" ces spectrogrammes, en détaillant l'état actuel de l'implémentation pour la prédiction des paramètres discrets et les pistes envisagées pour la régression des paramètres continus.

II. Conception du Moteur de Synthèse

L'entraînement d'un modèle d'apprentissage profond repose sur l'exploitation d'une quantité massive de données annotées. Pour conserver un contrôle absolu sur le dataset, j'ai fait le choix de développer un moteur de synthèse soustractive simple sur mesure, afin de générer ce corpus de manière standardisée. Utiliser d'emblée un synthétiseur commercial rendrait l'automatisation de la génération de données extrêmement complexe, tout comme la tâche d'apprentissage du réseau.

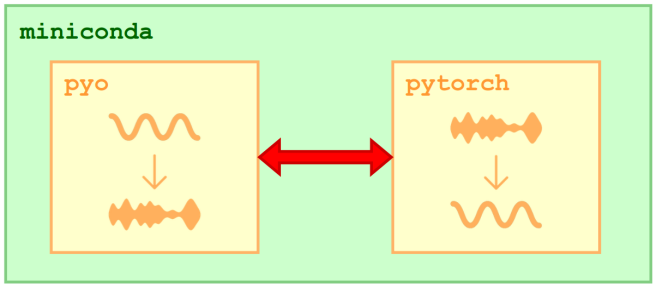
Ce chapitre détaille donc la conception de ce synthétiseur virtuel sur Python. Je justifierai dans un premier temps les choix technologiques adoptés pour son implémentation, avant de décrire en détail l'architecture acoustique du système mis en place.

II.1 Choix de l'environnement de développement : le pivot vers Python

Pour savoir par où commencer, il est tout d'abord nécessaire de se poser la question de quel language faut il utiliser pour avoir le plus de liberté concernant l'implémentation du synthétiseur ainsi que du réseau de neurones. Ce projet ayant pour objectif de s'étendre à la mise en place d'un *plugin* audio de prédiction et de synthèse audio, l'utilisation de C++ combiné à JUCE semblait être une bonne opportunité pour étudier à la fois le problème de la synthèse inverse audio et le développement logiciel.

Je me suis donc mis à la recherche de code open source pouvant implémenter un synthétiseur répondant à mes besoins : un synthétiseur soustractif simple comprenant un oscillateur avec un table d'ondes, un filtre modulable (*cutoff*, *resonance*), une enveloppe ADSR ainsi qu'un moyen de bruiter le signal. Cependant, bien qu'il soit tout à fait possible de trouver de tels projets en C++, leur configuration sur ma machine s'est très vite révélée fastidieuse. En effet, un grand nombre de ces dépôts souffrent d'un manque de maintenance, et la gestion des dépendances liées aux bibliothèques externes s'avère complexe. Ces conflits de versions entraînent fréquemment des erreurs de compilation ou font crasher le projet de manière inopinée. Ce processus de configuration logicielle, très chronophage, ralentissait considérablement la phase de recherche pure.

C'est face à cette lourdeur de mise en place que je me suis tourné vers Python qui s'est imposé comme une solution beaucoup plus robuste et complète pour le prototypage de ce genre de problème. En effet, Python offre la possibilité de scripter facilement le moteur de synthèse audio grâce à la librairie *Pyo*, tout en s'intégrant nativement avec *PyTorch* pour le développement et l'entraînement du réseau de neurones. De plus, grâce au gestionnaire de paquets *Miniconda*, l'installation des différentes bibliothèques utiles au projet se fait très facilement. Ce pivot technologique m'a ainsi permis de gagner un temps précieux sur la génération automatisée des données.



II.2 L'écosystème Python : Pyo pour l'audio et PyTorch pour l'apprentissage

Une fois le langage Python sélectionné comme environnement de travail principal, il a fallu identifier les bibliothèques les plus adaptées pour répondre aux deux grands défis du projet : la synthèse sonore automatisée et la construction du modèle prédictif.

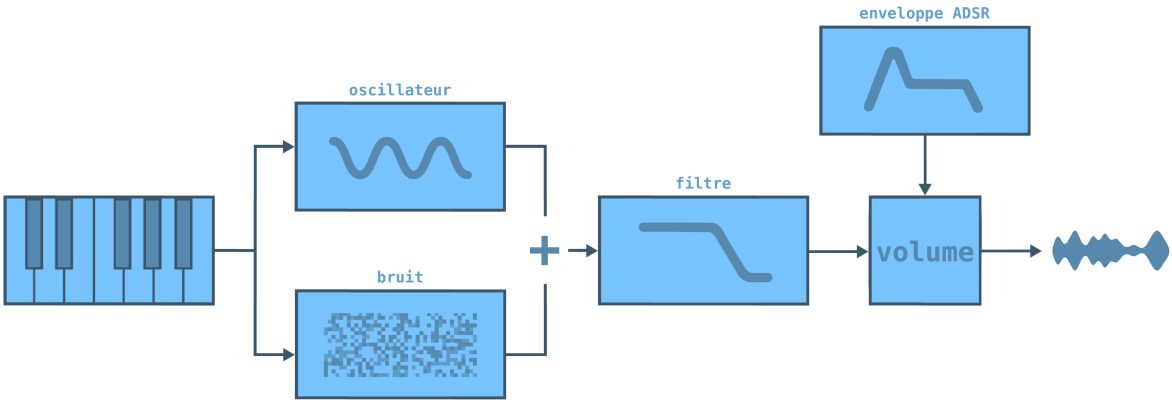
Pour la génération audio, mon choix s'est porté sur *Pyo* [4], un module Python écrit en C et dédié au traitement du signal numérique. L'utilisation de cette bibliothèque repose sur deux arguments techniques majeurs :

- **L'approche "boîte blanche" :** Les synthétiseurs commerciaux intègrent très souvent des traitements internes cachés (légère compression, saturation analogique simulée) destinés à rendre le son plus flatteur à l'oreille. Dans le cadre d'un apprentissage automatique, ces altérations sont néfastes car elles introduisent des biais dans les données. *Pyo*, à l'inverse, fournit des blocs mathématiques stricts (un oscillateur génère une onde pure, un filtre applique une équation différentielle précise). Cette transparence garantit que le réseau de neurones apprendra les véritables corrélations entre les paramètres et le timbre.
- **Le rendu hors-ligne :** L'entraînement d'un réseau de neurones requiert des dizaines de milliers d'échantillons. Si le synthétiseur générait le son en temps réel (comme un véritable instrument), la création du dataset prendrait des heures. *Pyo* permet de calculer le rendu audio directement en mémoire ou vers le disque dur beaucoup plus rapidement que le temps réel, ce qui est indispensable pour une génération de masse via des boucles de script.

En parallèle, *PyTorch* [5] a été retenu pour concevoir et entraîner l'architecture neuronale. Devenu un standard incontournable dans la recherche en apprentissage profond, cette librairie offre une grande flexibilité grâce à ses écosystème riche, notamment pour le traitement audio (via `torchaudio` [6]) ainsi que la vision par ordinateur.

II.3 Architecture du synthétiseur soustractif sous Pyo

La flexibilité offerte par la librairie *Pyo* m'a permis de concevoir l'architecture de l'instrument brique par brique, à la manière d'un synthétiseur modulaire physique. Le diagramme suivant illustre la topologie du moteur de synthèse souhaité, mettant en évidence l'interaction entre les différentes sources sonore (oscillateur et bruit), le traitement spectral (filtre) et la sculpture temporelle de l'amplitude (enveloppe ADSR).



L'implémentation de cette architecture repose sur l'instanciation et le chaînage des objets audio fournis par la librairie *Pyo*. Le script de génération a été conçu de manière modulaire afin de transformer un dictionnaire de paramètres (la configuration cible) en un fichier audio brut. Les éléments essentiels de ce moteur de synthèse seront définis dans la fonction ci-dessous :

```
def simple_synth(nom_fichier, params):
```

La variable `nom_fichier` désigne le nom du fichier dans lequel sera enregistré l'audio et `params` désigne le dictionnaire des paramètres utiles pour la génération d'un audio avec le synthétiseur.

Initialisation du serveur en mode *offline*

La première étape consiste à initialiser le moteur audio. Conformément à au besoin de génération de données massive, le serveur est configuré en mode *offline* (hors-ligne). Ainsi, le calcul ne sera pas envoyé vers la carte son de l'ordinateur, mais sera directement écrit dans un fichier `.wav`.

```
# Initialisation du serveur audio pour un rendu direct sur disque
s = Server(audio='offline')
s.boot()
duree_totale = params['duree'] + params['env_release']
s.recordOptions(dur=duree_totale, filename=nom_fichier)
```

Oscillateur et Bruit : L'étage de génération

La source sonore du synthétiseur est composée de deux éléments mixés : un oscillateur principal et un générateur de bruit blanc. Un choix d'implémentation notable ici est l'utilisation de la classe `LFO` (Low Frequency Oscillator) au lieu d'un oscillateur classique.

Bien que traditionnellement réservé aux modulations lentes, le `LFO` de *Pyo* possède l'avantage d'accepter un argument `type` sous forme d'entier qui me permet de choisir la forme d'onde souhaitée. Poussé à des fréquences audibles, il devient ainsi un oscillateur audio extrêmement pratique à piloter algorithmiquement depuis le dictionnaire de paramètres.

J'utilise ensuite `Noise` pour pouvoir générer un bruit blanc. Lorsque l'oscillateur et le bruit sont mis en place, je peux ensuite les superposer dans une variable de mixage `source_mix` représentée par le $+$ dans le diagramme précédent.

```
# Oscillateur principal
osc = LFO(freq=params['osc_freq'],
          sharp=params['osc_sharp'],
          type=OSC_CONVERSION[params['osc_type']],
          mul=params['osc_vol'])

# Generateur de bruit blanc
noise = Noise(mul=params['noise_vol'])

# Mixage des deux sources sonores
source_mix = osc + noise
```

L'étage de Filtrage

Le signal brut passe ensuite dans un filtre multimode implémenté via la classe `Biquadx`. Ce filtre biquadratique permet de moduler dynamiquement la fréquence de coupure (*cutoff*) et la résonance (Q), tout en pouvant basculer entre un comportement Passe-Bas, Passe-Haut ou Passe-Bande.

```
# Filtre biquadratique multimode
filt = Biquadx(source_mix,
               freq=params['filter_cutoff'],
               q=params['filter_res'],
               type=params['filter_type'])
```

Enveloppe temporelle et rendu final

Enfin, pour que le son ait une dynamique réaliste (une attaque, un maintien, et une extinction), j'utilise un générateur d'enveloppe `Adsr`. Dans *Pyo*, l'application d'une enveloppe sur un signal continu se fait par une simple multiplication mathématique entre le signal filtré et le signal de contrôle de l'enveloppe. Je me sers ensuite du `Pan` pour avoir le son dans les deux oreilles et ainsi avoir une écoute plus confortable.

```
# Generation du profil ADSR
env = Adsr(attack=params['env_attack'],
           decay=params['env_decay'],
           sustain=params['env_sustain'],
           release=params['env_release'],
           dur=params['duree'],
           mul=params['master_vol'])

# Application de l'enveloppe et spatialisation (Pan)
signal = filt * env
panner = Pan(signal, outs=2, pan=0.5)
panner.out()

# Declenchement de l'enveloppe et lancement du calcul
env.play()
s.start()
s.shutdown()
```

C'est cette architecture, pilotée de l'extérieur par des scripts de randomisation, qui va me permettre de générer la matière première nécessaire à l'apprentissage du réseau de neurones.

Finalement, comme j'ai le contrôle sur la totalité des paramètres du synthétiseur, il m'est tout à fait possible de générer un dataset de fichiers audio avec leurs paramètres associés dans un fichier `.csv`. De plus, comme le moteur de synthèse est en mode *offline*, je peux générer des milliers de fichiers en peu de temps. Il est maintenant question de générer un dataset audio/paramètres cohérent pour faciliter l'apprentissage au futur réseau de neurones.

III. Création du dataset

La performance d'un modèle d'apprentissage supervisé dépend directement de la qualité, de la quantité et de la diversité des données sur lesquelles il s'entraîne. Dans le cadre de ce projet, j'ai fait le choix de générer synthétiquement ma propre base de données, comprenant 10 000 fichiers audio annotés.

Cependant, la génération automatisée de données acoustiques pose un défi majeur : un tirage purement aléatoire et naïf des paramètres de synthèse produit en grande majorité des sons inexploitables ou physiquement incohérents (silences absolus, saturations extrêmes, fréquences inaudibles). Ce chapitre détaille les algorithmes et les règles heuristiques mis en place pour contraindre l'espace de génération, garantissant ainsi un dataset riche, équilibré et musicalement pertinent.

III.1 Éviter les incohérences acoustiques et de données

Le premier travail d'ingénierie a consisté à identifier et éliminer les "zones mortes" de l'espace des paramètres, c'est-à-dire les combinaisons de valeurs conduisant à des aberrations sonores ou compliquant inutilement la tâche d'apprentissage du réseau.

Gestion du volume global et du bruit

L'architecture du synthétiseur mixe un oscillateur principal avec un générateur de bruit blanc. Un tirage indépendant et uniforme des volumes de ces deux sources (`osc_vol` et `noise_vol`) introduit un risque non négligeable : obtenir un silence total si les deux valeurs s'approchent simultanément de zéro. Pour pallier cela, j'ai implémenté une boucle de sécurité forçant un nouveau tirage tant que la somme du volume des deux sources est nulle.

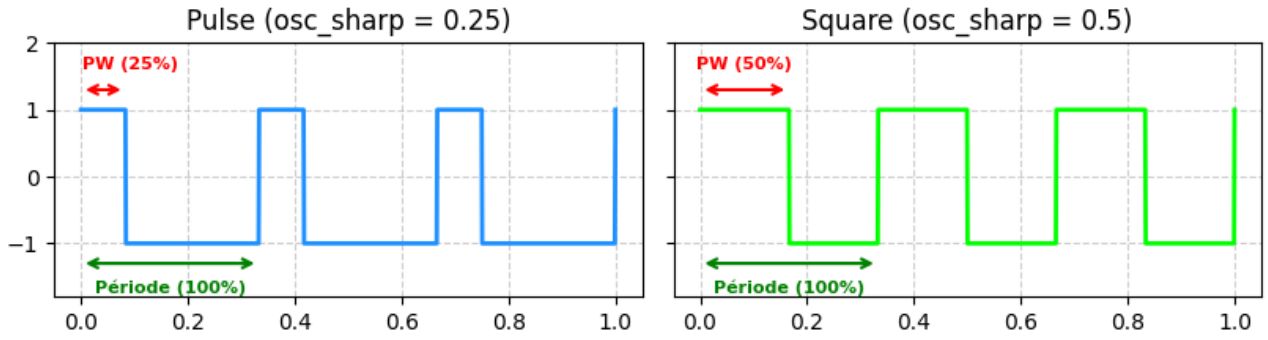
```
# Boucle de securite
no_volume = True
    while no_volume:
        p['osc_vol'] = random.uniform(*CONFIGS['osc_vol'])
        p['noise_vol'] = random_pow(CONFIGS['noise_vol'], 2)
        no_volume = p['osc_vol'] == 0.0 and p['noise_vol'] == 0.0
```

De plus, la proportion maximale de bruit a été empiriquement plafonnée à 40% car au-delà de ce seuil, le bruit blanc masque entièrement le contenu harmonique de l'oscillateur à l'oreille, rendant ainsi la prédiction de la forme d'onde absurde.

```
CONFIGS = {...
    'noise_vol': (0.0, 0.4), # Max 40% de bruit
...}
```

Le problème d'ambiguïté des formes d'ondes

Un défi plus subtil s'étant révélé lors de la phase d'apprentissage concerne la morphologie des ondes générées. Le synthétiseur propose des ondes de type *Pulse* (impulsion) dont la largeur d'impulsion (*Pulse Width*) est contrôlée par le paramètre `osc_sharp`. D'un point de vue acoustique et mathématique, une onde *Pulse* dont la largeur est exactement de 50% (`osc_sharp = 0.5`) est strictement identique à une onde *Square*.



Si l'algorithme génère aléatoirement des ondes *Pulse* avec une largeur proche de 0.5, le réseau de neurones sera pénalisé lors de l'entraînement : il recevra deux sons auditivement et spectralement identiques, mais portant des *labels* différents (l'un étiqueté *Pulse*, l'autre *Square*). Pour éviter cette confusion lors de l'apprentissage, j'ai forcé le paramètre `osc_sharp` des ondes *Pulse* à éviter la zone médiane.

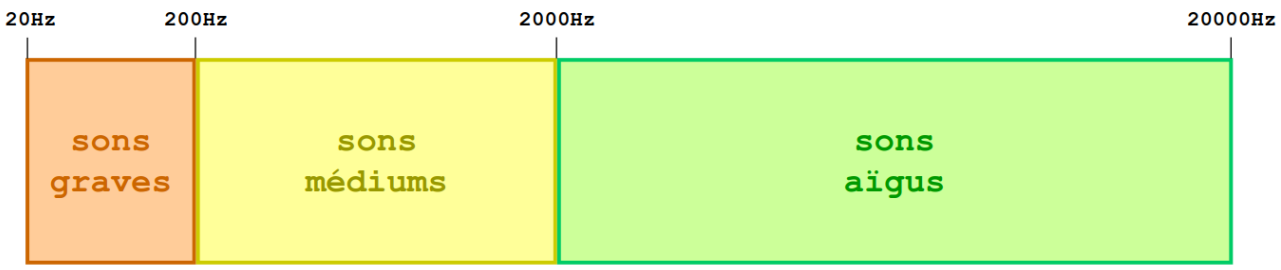
```
# Pulse (3) ou Bipolar Pulse (4) -> Sharp EXTREME (loin de 0.5)
if p['osc_type'] in [3, 4]:
    # On choisit soit tres fin (0.1-0.25) soit tres large (0.75-0.9)
    # On evite la zone 0.35-0.65 qui ressemble trop au Square
    if random.random() > 0.5:
        p['osc_sharp'] = random.uniform(0.1, 0.25)
    else:
        p['osc_sharp'] = random.uniform(0.75, 0.9)
else:
    p['osc_sharp'] = 0.5
```

Cette règle simple garantit que les ondes impulsionnelles du dataset possèdent systématiquement le timbre nasillard caractéristique qui les différencie clairement des ondes carrées pures.

III.2 Modélisation de la perception humaine : Les distributions non-linéaires

Lors des premières générations aléatoire de sons que j'ai pu faire, il s'est révélé un problème lié aux distributions linéaires pour tous les paramètres. Or, la perception humaine du son, notamment des fréquences, n'est pas linéaire mais logarithmique. Si cette réalité physique n'est pas prise en compte lors de la création du dataset, le réseau de neurones apprendra sur une base de données fortement biaisée avec peu de sons grave et une grande quantité de sons aïgus ; dataset qui serait ainsi peu représentatif de la musique réelle.

Je vais donc détailler la façon dont j'ai choisi et implémenté les différents tirages aléatoires, et pour quels paramètres je les appliquerai.



La distribution logarithmique des fréquences

Le synthétiseur génère des fréquences fondamentales comprises entre 40 Hz et 4000 Hz. Si j'utilise un simple tirage linéaire sur cet intervalle, la moitié des échantillons se trouverait au-dessus de 2000 Hz (soit dans les aïgus spectre). Le modèle serait alors surentraîné sur des sons extrêmement aigus et incapable de reconnaître des basses ou des bas-médiums.

Pour résoudre ce problème, j'ai implémenté une fonction de tirage aléatoire logarithmique `random_log`. Elle tire une valeur uniforme dans le domaine des logarithmes avant de la repasser dans le domaine exponentiel. Ainsi, j'ai la garantie que je tire autant de fréquences aïgues que graves.

```
def random_log(tuple_min_max):
    """Retourne une valeur aleatoire distribuee logarithmiquement."""
    log_min = math.log(tuple_min_max[0])
    log_max = math.log(tuple_min_max[1])
    log_val = random.uniform(log_min, log_max)
    return math.exp(log_val)
```

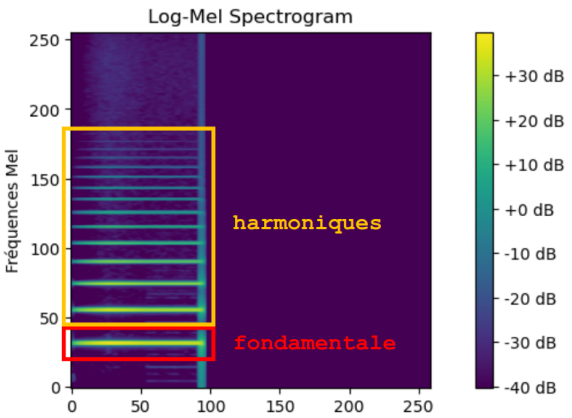
La distribution de puissance pour le volume du bruit

Le même principe de distribution non-linéaire a été appliqué au volume du générateur de bruit. Afin d'entraîner un modèle robuste, il est bénéfique que la plupart des sons contiennent une légère quantité de bruit, modélisant les imperfections analogiques ou les attaques de souffle.

Cependant, après une écoute des sons générés aléatoirement, un tirage linéaire conduit trop souvent à des niveaux de bruit élevés masquant le timbre de l'oscillateur. J'ai donc créé une fonction de distribution par puissance `random_pow` pour ce paramètre.

```
def random_pow(tuple_min_max, curve=3):
    """Retourne une valeur aleatoire distribuee exponentiellement."""
    raw = random.random()
    valeur_courbe = raw ** curve
    return valeur_courbe * tuple_min_max[1]
```

III.3 Règles de filtrage dynamiques



L'interaction entre la fréquence fondamentale générée par l'oscillateur (`osc_freq`) et la fréquence de coupure du filtre (`filter_cutoff`) constitue le point le plus délicat de la génération aléatoire. Si ces deux paramètres sont tirés de manière totalement indépendante, le risque d'obtenir des fichiers audio muet sur une dizaine de milliers de rendus devient statistiquement inévitable.

Pour comprendre ce phénomène, il faut rappeler le principe de la synthèse soustractive : l'oscillateur crée un signal, le plus souvent riche en harmoniques (situées au-dessus de la fréquence fondamentale), que le filtre vient ensuite atténuer. Si le filtre a été malencontreusement configuré pour supprimer la totalité des harmoniques (fondamentale comprise), l'audio sortant devient juste un silence.

J'ai donc conçu une fonction qui conditionne dynamiquement les bornes de tirage aléatoire de la fréquence de coupure en fonction du type de filtre choisi et de la note fondamentale générée. Tout le détail ci-dessous sera ainsi codé dans cette fonction :

```
def random_cutoff(osc_freq, filter_type):
    min_global = CONFIGS['filter_cutoff'][0]  # coupure minimale = 50Hz
    max_global = CONFIGS['filter_cutoff'][1]  # coupure maximale = 20000Hz
```

Le Filtre Passe-Bas (*Low-Pass*)

Le filtre passe-bas laisse passer les basses fréquences et coupe les aigus au-delà du *cutoff*. Si la fréquence de coupure est tirée au hasard en dessous de la fondamentale de l'oscillateur, la totalité du timbre de l'oscillateur est coupé, provoquant ainsi un silence.

La règle que j'ai implémenté garantit que la borne inférieure du tirage du *cutoff* soit toujours strictement supérieure ou égale à la fondamentale. Le réseau de neurones ne sera ainsi jamais confronté à un fichier "étouffé" où la hauteur de note est indétectable.

```
# LOW PASS (Type 0) : On garde les graves, on coupe AU-DESSUS de la note
if filter_type == 0:
    low_bound = max(min_global, osc_freq) # Au minimum la note elle-meme
    high_bound = max_global
```

Le Filtre Passe-Haut (*High-Pass*)

Le filtre passe-haut fonctionne à l'inverse : il coupe les graves et laisse passer les aigus. Si le *cutoff* est placé bien au-dessus de la fondamentale, nous perdons les premières harmoniques du signal qui caractérise le son en ne conservant que les harmoniques supérieures. Même si ce cas est moins impactant que le précédent, je dois faire en sorte que la fréquence fondamentale ne soit pas totalement muette pour pouvoir par la suite prédire la fréquence jouée par le signal.

L'algorithme limite donc sévèrement la borne supérieure de tirage : le *cutoff* ne peut pas dépasser la fondamentale de plus de 50% (`osc_freq * 1.5`). Cela permet d'obtenir des sons amincis, utiles pour la diversité du *dataset*, sans avoir des sons dépourvus de leurs identité.

```
# HIGH PASS (Type 1) : On garde les aigus, on coupe EN-DESSOUS de la note
elif filter_type == 1:
    low_bound = min_global
    # On permet d'aller un peu au-dessus de la fondamentale (ici x1.5)
    high_bound = min(max_global, osc_freq * 1.5)
```

Le Filtre Passe-Bande (*Band-Pass*)

Le filtre passe-bande isole une portion spécifique du spectre. Pour qu'une note reste identifiable musicalement, la bande passante doit impérativement englober la fondamentale. Les bornes du tirage aléatoire sont donc restreintes à une fenêtre relativement étroite encadrant la fréquence de l'oscillateur (ici de la moitié de sa valeur jusqu'au double).

```
# BAND PASS (Type 2) : On garde juste une bande autour de la note
else:
    low_bound = max(min_global, osc_freq * 0.5)
    high_bound = min(max_global, osc_freq * 2.0)
```

Une fois ces bornes dynamiques calculées, le tirage effectif du *cutoff* est réalisé de manière logarithmique entre ces deux limites de sécurité, assurant ainsi une distribution naturelle à l'oreille tout en respectant strictement les lois de l'acoustique.

III.4 Validation automatique et *Sanity Checks*

La génération d'un grand volume de données (10 000 fichiers audio) rend toute vérification manuelle ou auditive impossible. Il est pourtant crucial de s'assurer qu'aucune erreur logique ne s'est glissée dans les heuristiques de génération avant de lancer la longue et coûteuse phase d'entraînement du réseau de neurones.

Pour ce faire, le script de génération sauvegarde l'ensemble des paramètres associés à chaque fichier audio dans un fichier de journalisation global (`dataset_labels.csv`). J'ai ensuite développé un script d'analyse exploitant la librairie *Pandas* [7] pour effectuer un audit complet du corpus généré, une étape souvent qualifiée de *Sanity Check* (test de vérification de cohérence).

Équilibre des classes d'ondes

Le premier objectif de l'analyse est de s'assurer de l'équilibre du *dataset* vis-à-vis des paramètres discrets (classification). Un réseau de neurones entraîné sur des classes déséquilibrées aura tendance à favoriser la classe majoritaire au détriment des autres. Les résultats de l'analyse montrent une distribution quasi parfaite, dictée par nos tirages uniformes sur ces catégories :

- **Oscillateurs :** Environ 1660 échantillons pour chacun des 6 types d'oscillateurs.
- **Filtres :** Environ 3330 échantillons pour chacun des 3 types de filtres.

Aucune valeur manquante (*Null*) n'a été détectée dans l'ensemble des 10 000 lignes du registre.

Analyse de la distribution des fréquences

Comme théorisé lors de la conception de l'algorithme de génération (voir Section III.2), il était crucial de vérifier la bonne répartition spectrale de notre base de données. L'analyse du `.csv` confirme l'efficacité du tirage aléatoire logarithmique pour les fréquences fondamentales (`osc_freq`) :

- **Moyenne :** ≈ 840 Hz
- **Médiane :** ≈ 380 Hz

Le module de validation conclut automatiquement à la justesse de cette distribution puisque la médiane est strictement et largement inférieure à la moyenne. Cette validation mathématique prouve que la majeure partie des échantillons est correctement concentrée dans les graves et les bas-médiums, prévenant ainsi tout biais d'apprentissage vers les hautes fréquences, conformément à la perception sonore humaine.

Validation physique et détection de silence

Le test le plus critique concerne la validation des règles de filtrage dynamiques (détaillées dans la section précédente). Le script d'analyse parcourt le *dataset* à la recherche d'incohérences physiques (par exemple, un filtre passe-bas dont la fréquence de coupure serait inférieure à 90% de la fréquence fondamentale de l'oscillateur).

Voici l'extrait des résultats générés par le module de validation :

```
============================================================
 4. SANITY CHECK (DETECTION DE SILENCE)
============================================================
LowPass : Aucun risque de silence detecte.
HighPass : 692 fichiers ont la fondamentale coupee (Son fin/maigre).
   (C'est acceptable tant que ce n'est pas 100% du dataset)
```

Ce rapport d'exécution confirme l'efficacité de l'algorithme : **aucun fichier** ne souffre d'un passe-bas défectueux qui aurait entraîné un silence absolu. Concernant les filtres passe-haut, environ 7% des fichiers (692/10000) présentent une fondamentale fortement coupée. Ce comportement, voulu et limité par les bornes mathématiques fixées, est jugé acceptable et même désirable, car il introduit des timbres "fins" et riches en harmoniques supérieures qui accroissent la diversité du *dataset* sans pour autant le rendre inexploitable.

Récapitulatif

À l'issue de cette phase d'ingénierie, nous disposons d'un corpus de 10 000 fichiers audio générés avec un synthétiseur soustractif "boîte blanche" sous *Pyo* avec leurs paramètres associés dans un `.csv` parfaitement équilibrées, validées acoustiquement et prêtes à être ingérées. L'étape suivante consistera à transformer ces fichiers audio bruts en une représentation mathématique exploitable par un réseau de neurones convolutif.

IV. Prétraitement et Choix de l'Architecture Neuronale

Maintenant que le corpus de données est généré et validé, il convient de définir la manière dont le réseau de neurones va l'assimiler. C'est ainsi que ce chapitre détaillera la transition du domaine de l'acoustique vers celui de l'apprentissage profond, en justifiant le prétraitement appliqué aux fichiers audio, puis en détaillant l'architecture neuronale choisie pour résoudre au mieux notre problème de synthèse inverse.

IV.1 Du domaine temporel au Spectrogramme Log-Mel

La première étape du problème d'apprentissage consiste à transformer le signal audio brut en une représentation optimisée pour un réseau de neurones :

Les limites de l'onde temporelle brute

Dans l'état actuel de l'implémentation, la tâche principale du réseau de neurones consiste à prédire les caractéristiques de l'oscillateur (type d'onde, fréquence) et du filtre (type, fréquence de coupure). Ces paramètres définissent l'empreinte spectrale du son, c'est-à-dire son contenu en fréquences. Il est donc logique de fournir d'emblée au réseau de neurones une représentation qui isole et met en évidence ces données : le spectrogramme Log-Mel.

À l'inverse, utiliser le fichier audio à l'état brut (la forme d'onde temporelle ou *waveform*) poserait un problème majeur d'efficacité, principalement à cause de la phase.

Pour faire simple, la phase correspond au point de départ temporel de l'onde. Si l'on décale un son d'une fraction de milliseconde, l'oreille humaine entendra exactement le même timbre. Pourtant, sur un graphique 1D, ce micro-décalage modifie complètement le tracé visuel de la courbe. Si le modèle était entraîné directement sur cette onde brute, il serait constamment trompé par ces décalages invisibles à l'oreille. Il perdrait une puissance de calcul considérable à analyser des tracés visuellement différents pour des sons auditivement identiques. Le passage au spectrogramme permet donc de s'affranchir de cette phase instable pour ne conserver que l'énergie fréquentielle utile à la prédiction.

Le Spectrogramme Log-Mel : Une image du son

Pour extraire les fréquences qui composent notre audio, j'applique une **Transformée de Fourier à Court Terme (STFT)**. Cette opération découpe le signal en courtes fenêtres temporelles et calcule le spectre de fréquences pour chacune d'elles, créant ainsi une matrice 2D où l'axe X représente le temps, l'axe Y les fréquences, et la valeur des pixels représente l'amplitude.

Pour affiner cette matrice et la rendre la plus pertinente possible (et en cohérence avec le travail de génération au Chapitre 3), j'applique deux transformations psychoacoustiques :

- **L'échelle Mel :** Cette échelle resserre les hautes fréquences et étire les basses fréquences, reproduisant la perception logarithmique de l'oreille humaine. Le réseau concentrera ainsi son attention sur les bas-médiums, là où se situe l'essentiel de l'information tonale.

- **L'échelle Logarithmique (Décibels) :** L'amplitude du signal (l'énergie) est passée à l'échelle logarithmique. Cela permet de faire ressortir visuellement les harmoniques subtiles qui, bien que de faible volume, sont responsables de la "couleur" (le timbre) du filtre et de l'oscillateur.

À l'issue de ce prétraitement, un fichier audio de quelques secondes est transformé en une image 2D (le spectrogramme Log-Mel) de taille fixe. Cette transformation me permet de requalifier le problème de traitement du signal en un problème de vision par ordinateur.

Implémentation avec PyTorch : Standardisation et Extraction

Pour que le réseau de neurones puisse traiter ces données, chaque fichier audio doit être converti en un tenseur de dimensions strictement identiques (une spectrogramme de taille fixe). J'ai implémenté ce prétraitement via la librairie *Torchaudio*.

La première étape de la fonction d'extraction `get_spectrogram` consiste à standardiser le signal brut :

- **Fréquence d'échantillonnage :** Forcée à 22 050 Hz pour limiter la quantité de données tout en conservant les fréquences utiles (théorème de Shannon).
- **Monophonie :** Réduction des éventuels signaux stéréo en faisant la moyenne des canaux.
- **Dimension temporelle :** Troncature ou ajout de zéros (*zero-padding*) pour garantir une durée exacte de $6,0$ secondes pour tous les échantillons.

```
def get_spectrogram(audio_path, target_sr, target_sample,
                    mel_transform=None, db_transform=None):

    waveform, sr = torchaudio.load(audio_path)

    # Standardisation de la frequence
    if sr != target_sr:
        resampler = torchaudio.transforms.Resample(sr, target_sr)
        waveform = resampler(waveform)

    # Conversion en Mono : on fait la moyenne des canaux si audio en Stereo
    if waveform.shape[0] > 1:
        waveform = torch.mean(waveform, dim=0, keepdim=True)

    # Standardisation de la duree (Padding ou Troncature)
    num_samples = waveform.shape[1]
    if num_samples > target_sample: # si audio trop long : TRONCATURE
        waveform = waveform[:, :target_sample]

    elif num_samples < target_sample: # si audio trop court : PADDING
        num_missing = target_sample - num_samples
        waveform = torch.nn.functional.pad(waveform, (0, num_missing))
```

Une fois le signal standardisé, j'applique la Transformée de Fourier et le passage à l'échelle logarithmique. Le paramètre `n_fft = 4096` a été choisi pour offrir une très haute résolution fréquentielle, indispensable pour isoler la fréquence fondamentale de l'oscillateur. Le paramètre `n_mels = 256` définit la hauteur de notre "image" finale.

```
    # Application des transformations passees en parametre
    mel_spec = mel_transform(waveform)
    mel_spec_db = db_transform(mel_spec)
    return mel_spec_db
```

Après exécution, cette fonction transforme un fichier `.wav` en un tenseur PyTorch de forme `[1, 256, 259]` (1 canal de couleur, 256 pixels de hauteur pour les fréquences Mel, et 259 pixels de largeur pour le temps).

La classe SynthDataset

Afin d'alimenter efficacement le réseau de neurones lors de la phase d'entraînement, j'ai encapsulé cette logique d'extraction au sein d'une classe héritant de `Dataset` de *Pytorch* (`torch.utils.data.Dataset`). En effet, l'héritage de cette classe est une convention indispensable sous PyTorch. Il permet de rendre la classe nativement compatible avec le `DataLoader` de la librairie. C'est ce `DataLoader` qui s'appuiera sur les méthodes `__len__` et `__getitem__` pour automatiser des tâches critiques de l'apprentissage comme la création des lots d'entraînement ou encore le mélange aléatoire des données à chaque époque.

Un choix d'ingénierie important ici concerne l'optimisation des performances : les objets de transformation (`MelSpectrogram` et `AmplitudeToDB`) sont instanciés une seule fois dans le constructeur `__init__`, puis passés en argument à la fonction `get_spectrogram`. Cette approche évite de recréer ces lourds objets mathématiques à la volée pour chacun des 10 000 fichiers du *dataset*, accélérant considérablement le chargement des données.

Conformément à l'objectif actuel, la méthode `__getitem__` est configurée pour ne renvoyer, en plus du spectrogramme, que les *labels* liés au filtrage et à l'oscillateur.

```
class SynthDataset(torch.utils.data.Dataset):
    def __init__(self, csv_file, audio_dir, sr=22050, duration=6.0):
        self.df = pd.read_csv(csv_file)      # self.df <- dataset_labels.csv
        self.audio_dir = audio_dir           # self.audio_dir <- \wavs
        self.sr = sr                         # self.sr <- 22050 Hz
        self.samples = int(sr * duration)

        # Instanciation unique pour optimisation CPU/GPU
        self.mel_transform = torchaudio.transforms.MelSpectrogram(
            sample_rate=sr, n_mels=256, n_fft=4096, hop_length=512
        )
        self.db_transform = torchaudio.transforms.AmplitudeToDB(top_db=80)

    def __len__(self):
        return len(self.df)

    def __getitem__(self, index):
        # Localisation du fichier audio en fonction de son index
        audio_filename = self.df.iloc[index, 0]
        audio_path = os.path.join(self.audio_dir, audio_filename)

        # Extraction des labels cibles (Classification)
        osc_type = int(self.df.iloc[index, 2])
        filter_type = int(self.df.iloc[index, 7])

        # Calcul du tenseur avec get_spectrogram
        spectrogram = get_spectrogram(
            audio_path, self.sr, self.samples,
            self.mel_transform, self.db_transform
        )

        return spectrogram, osc_type, filter_type
```

IV.2 L'architecture & son Implémentation : Réseaux Convolutifs (CNN) pour l'audio

Les travaux de Barkan et Tsiris [8] sur l'estimation de paramètres de synthétiseurs ainsi que ceux de Keunwoo Choi sur l'utilisation des CNN pour la classification audio [9], démontre la grande pertinence des architectures d'apprentissage profond pour la tâche de synthèse inverse. Mon approche s'inspire directement de leur méthodologie, qui consiste à traiter la représentation fréquentielle du son (le spectrogramme STFT) comme une image d'entrée pour un réseau de neurones.

La supériorité des CNN face aux réseaux denses

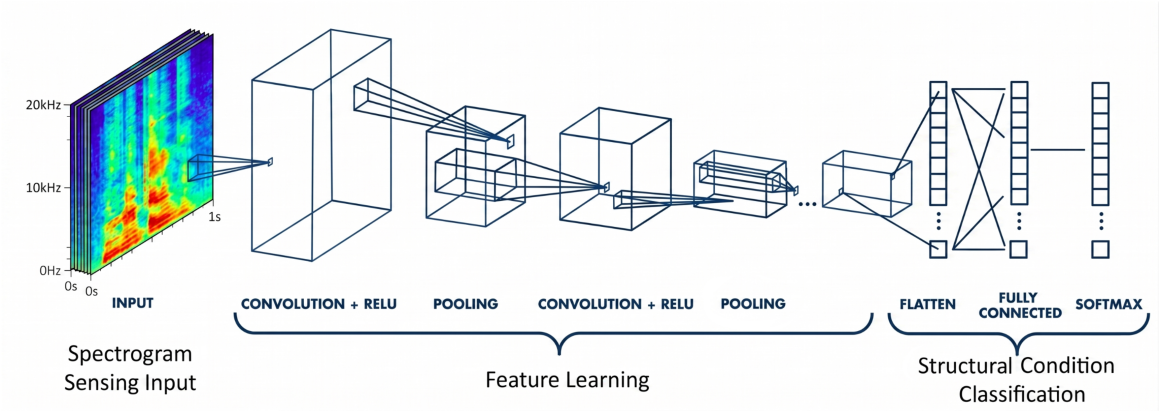
Historiquement, les algorithmes d'analyse audio utilisaient des réseaux de neurones denses (*Fully Connected*). Cependant, les réseaux denses se révèlent peu performants face à des données de grande dimensionnalité comme les signaux audio bruts ou les matrices de spectrogrammes, notamment en raison de leur manque d'invariance temporelle.

Pour bien comprendre cette limitation, il faut s'intéresser au fonctionnement d'un réseau dense classique. Dans ce type d'architecture, chaque pixel d'une image (ou chaque point d'un spectrogramme) est connecté à un neurone spécifique, avec un poids qui lui est propre. Le réseau apprend donc "par cœur" la position exacte d'une information.

Si j'entraîne un réseau dense à reconnaître la note "La" jouée à la première seconde de l'enregistrement, il l'apprendra parfaitement. Mais si, lors d'un test, cette même note "La" est jouée à la deuxième seconde, le motif visuel sur le spectrogramme sera décalé vers la droite. Le réseau dense, incapable de généraliser cette information spatiale, sera totalement désorienté et échouera à la reconnaître. C'est ce que l'on appelle le manque "d'invariance temporelle" (ou d'invariance par translation).

De plus, un spectrogramme est une matrice contenant souvent des dizaines de milliers de valeurs (par exemple 256×259 pixels). Un réseau dense nécessiterait un nombre astronomique de paramètres pour analyser chaque pixel individuellement, ce qui entraînerait une explosion des temps de calcul et un surapprentissage (*overfitting*) presque inévitable.

À l'inverse, les réseaux de neurones convolutifs (CNN) possèdent une propriété mathématique fondamentale : le partage de poids. Cette caractéristique leur permet d'appliquer de manière itérative les mêmes filtres spatiaux sur l'ensemble du signal, ce qui correspond naturellement au comportement des signaux auditifs et visuels. Le réseau devient ainsi capable de repérer une structure harmonique spécifique (le "timbre") indépendamment de l'instant exact où la note a été jouée. De plus, il a été démontré que la profondeur du réseau est un facteur déterminant contribuant à la précision des prédictions.



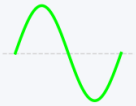
La signature spectrale : Distinguer les formes d'ondes par leurs harmoniques

Pour comprendre comment le réseau convolutif parvient à classifier les différents oscillateurs, il est crucial de rappeler comment ces formes d'ondes se traduisent visuellement sur un spectrogramme. Chaque type d'oscillateur possède une "signature spectrale" unique, définie par la répartition et l'intensité de ses harmoniques (les fréquences multiples de la note fondamentale).

Sur les spectrogrammes, ces harmoniques apparaissent sous la forme de lignes horizontales superposées. Le réseau de neurones est chargé d'identifier trois grands types de motifs :

- **La densité (Lignes pleines ou espacées) :** Une onde en dents de scie (*Sawtooth*) contient absolument toutes les harmoniques (paires et impaires), formant un empilement très dense. À l'inverse, une onde Carrée (*Square*) ne contient que des harmoniques impaires, ce qui laisse un espace visuel "vide" (des pixels sombres) entre chaque ligne brillante.
- **La décroissance (L'intensité lumineuse) :** La distinction est particulièrement complexe entre une onde Carrée et une onde Triangle, car toutes deux ne possèdent que des harmoniques impaires (leurs lignes sont situées exactement aux mêmes endroits). Le réseau doit alors s'appuyer sur la luminosité des pixels : les harmoniques du Triangle s'atténuent beaucoup plus rapidement que celles du Carré.
- **Les "trous" harmoniques :** Une onde impulsionnelle (*Pulse*), selon sa largeur, va faire disparaître périodiquement certaines harmoniques, créant des ruptures verticales caractéristiques dans le spectre.

Signatures Spectrales et Morphologie des Formes d'Ondes

Oscillateur	Densité Harmonique	Décroissance Harmonique	Trous Harmoniques
Sine (Sinusoïde) 	Uniquement la fondamentale Aucune harmonique	Aucune	Aucun (Spectre pur)
Sawtooth (Dents de scie) 	Toutes (Paires & Impaires)	Lente (1/n)	Aucun (Spectre continu)
Square (Carrée) 	Uniquement Impaires	Lente (1/n)	Constants (Absence des paires)
Triangle 	Uniquement Impaires	Rapide ($1/n^2$)	Constants (Absence des paires)
Pulse (Impulsion) 	Toutes (Paires & Impaires)	Lente (1/n)	Périodiques (Dépend du sharp)

Pour réussir sa tâche de classification, le réseau de neurone ne doit donc pas regarder l'image pixel par pixel, mais être capable de mesurer l'espacement vertical entre ces lignes et de comparer leur luminosité relative. C'est cette observation physique fondamentale qui m'a mené à l'architecture du CNN détaillée ci-après.

Notre architecture : Kernels asymétriques et champ récepteur en entonnoir

Bien que j'utilise un CNN, son implémentation diffère de l'architecture classique utilisée en vision par ordinateur (qui privilégie des noyaux carrés de type 3×3). Sur un spectrogramme, l'axe vertical représente les fréquences, tandis que l'axe horizontal représente le temps. C'est cet axe vertical qui contient les informations nécessaires au problème.
J'ai alors opté pour des noyaux de convolution strictement verticaux (de largeur 1) pour privilégier le contenu harmonique. Cela oblige le réseau à scanner les fréquences du spectre à un instant T précis, sans "flouter" ou mélanger l'information avec le contexte temporel adjacent.

J'ai donc conçu une architecture en "entonnoir" (*Funnel Architecture*), caractérisée par un noyau de très grande taille sur la première couche, se réduisant progressivement :

```
self.features = nn.Sequential(
    # Bloc 1 : Vue globale (Kernel geant pour capturer les harmoniques)
    nn.Conv2d(1, 16, kernel_size=(31, 1), padding=(15, 0)),
    nn.BatchNorm2d(16), nn.ReLU(), nn.MaxPool2d(2),

    # Bloc 2 : Vue moyenne
    nn.Conv2d(16, 32, kernel_size=(15, 1), padding=(7, 0)),
    nn.BatchNorm2d(32), nn.ReLU(), nn.MaxPool2d(2),

    # Bloc 3 & 4 : Analyse des details spectraux fins
    nn.Conv2d(32, 64, kernel_size=(7, 1), padding=(3, 0)),
    nn.BatchNorm2d(64), nn.ReLU(), nn.MaxPool2d(2),

    nn.Conv2d(64, 128, kernel_size=(7, 1), padding=(3, 0)),
    nn.BatchNorm2d(128), nn.ReLU(), nn.MaxPool2d(2),
)
```

L'utilisation d'un noyau `kernel_size=(31, 1)` offre un très large champ récepteur dès l'entrée du modèle. Dans les basses fréquences (dilatées par l'échelle Log-Mel), ce noyau géant permet au réseau d'englober trois à quatre harmoniques simultanément.

Maintien de la résolution spatiale : Le calcul du Padding

Afin de maîtriser parfaitement la réduction de dimensionnalité à travers le réseau, j'ai appliqué la technique du *Same Padding*. L'objectif est de s'assurer que seules les couches de réduction (*Max Pooling*) soient responsables du rétrécissement de l'image, et non les filtres de convolution eux-mêmes, afin d'éviter une perte d'information sur les bords du spectrogramme (les fréquences extrêmes).

La dimension de sortie en pixels d'une opération de convolution sur un axe donné est régie par la formule standard :

$$Sortie = Entrée + 2 \times Padding - Kernel + 1$$

Pour que l'image conserve sa taille exacte après son passage dans le filtre (c'est-à-dire $Sortie = Entrée$), l'équation se simplifie ainsi :

$$0 = 2 \times Padding - Kernel + 1 \implies Padding = \frac{Kernel - 1}{2}$$

J'ai appliqué cette règle mathématique de manière stricte et asymétrique sur les deux axes du spectrogramme :

- **Sur l'axe temporel (Largeur X) :** Tous les noyaux ont une largeur de 1 pixel. Le calcul donne donc systématiquement une marge nulle :

$$Padding_X = \frac{1-1}{2} = 0$$

- **Sur l'axe fréquentiel (Hauteur Y) :** Pour le premier bloc de l'entonnoir, le noyau possède une hauteur géante de 31 pixels. Le calcul impose une marge importante :

$$Padding_Y = \frac{31-1}{2} = 15$$

  De la même manière, pour les noyaux de taille 15 et 7, les marges nécessaires s'établissent respectivement à 7 et 3 pixels.

Grâce à cette justification mathématique, la dimension du tenseur reste parfaitement stable lors des phases d'extraction. Seules les quatre couches de `MaxPool2d(2)` divisent successivement la taille spatiale de l'image par deux. Cette approche garantit la stabilité du vecteur plat final (*flattened vector*) à 32 768 variables, évitant ainsi toute erreur de dimensionnalité lors du passage vers les couches denses de prédiction.

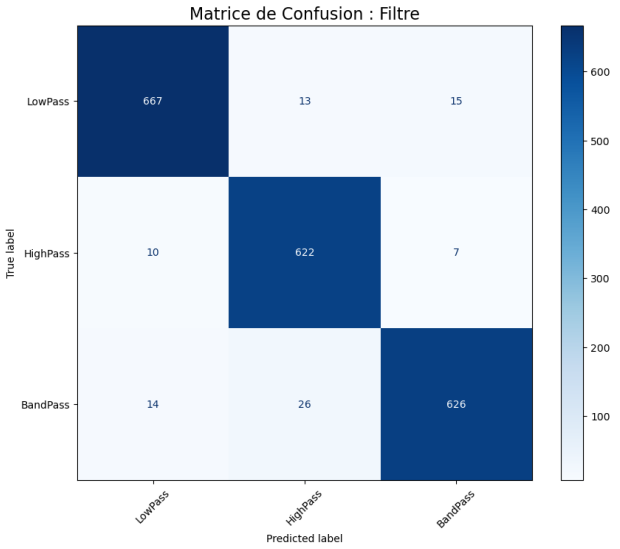
V. Résultats, Évaluation et Analyse Critique

Après avoir défini l'architecture neuronale en "entonnoir" (*Funnel Architecture*) et fiabilisé le jeu de données (notamment en fusionnant les classes *Saw Up* et *Saw Down* qui présentaient une symétrie spectrale indissociable), j'ai procédé à l'entraînement du modèle sur 25 époques. Ce chapitre présente les performances obtenues sur l'ensemble de test (20% du *dataset* non vus lors de l'entraînement) et analyse de manière critique les matrices de confusion.

V.1 Classification des Filtres

L'évaluation de la tâche de prédiction des filtres a initialement révélé un comportement aberrant : le réseau prédisait systématiquement la classe *LowPass* (Passe-Bas) pour 100% des échantillons. L'origine de ce biais n'était en fait qu'une erreur d'indexation lors de la lecture du fichier CSV des *labels* (`iloc[index, 6]` au lieu de `7`), qui fournissait au modèle la valeur continue du volume du bruit au lieu de la classe discrète du filtre.

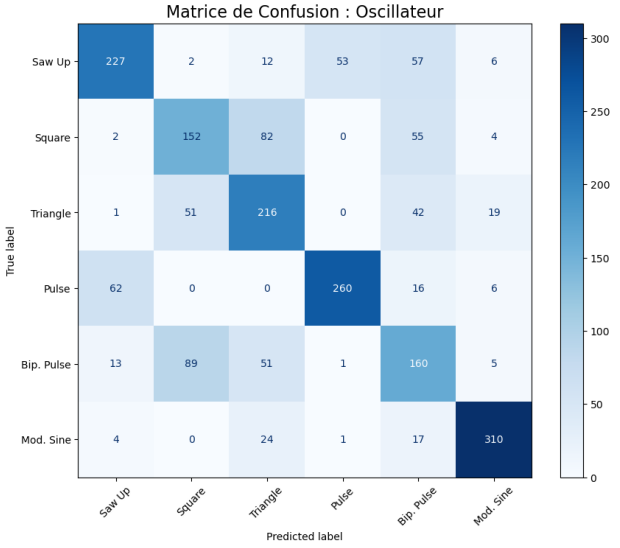
Une fois cette correction implémentée, la matrice de confusion démontre une excellente capacité de généralisation du modèle. Les trois classes (*LowPass*, *HighPass*, *BandPass*) présentent une diagonale très dense, avec des taux de bonne classification avoisinant les 95%. Le réseau parvient donc très efficacement à repérer les zones de coupure d'énergie spectrale caractéristiques de chaque type de filtre, validant ainsi la pertinence de l'approche par spectrogramme Log-Mel.



V.2 Classification des Oscillateurs

La prédiction du type d'oscillateur est un défi nettement plus complexe, car elle nécessite d'analyser l'espacement et la décroissance des harmoniques, souvent altérés par l'enveloppe ou le filtre susmentionné.

L'analyse de la matrice de confusion met en évidence des disparités intéressantes, directement corrélées à la nature physique des signaux :



Les succès de classification

Le modèle excelle dans l'identification de la sinusoïde modulée (*Mod. Sine*), avec 310 prédictions correctes. Sa signature spectrale, très éloignée des ondes géométriques classiques, est facilement repérée. De même, la *Pulse* et la *Saw Up* affichent de très bons scores. L'architecture de notre CNN a rempli son objectif pour ces différentes ondes.

La zone de confusion : Square, Triangle et Bipolar Pulse

La matrice révèle une difficulté à séparer clairement les ondes *Square*, *Triangle* et *Bipolar Pulse*. Cette confusion n'est pas un artefact d'apprentissage, mais une réalité physique :

- **L'ADN commun (Square vs Triangle) :** Ces deux ondes ne possèdent que des harmoniques impaires (fréquences $f, 3f, 5f...$). Leurs lignes spectrales sont donc positionnées exactement aux mêmes coordonnées. La seule différence réside dans la pente de décroissance de l'énergie des harmoniques.
- **L'impact du filtrage :** Le jeu de données applique aléatoirement des filtres *LowPass*. Lorsqu'une onde *Square* (riche en aigus) subit un filtrage passe-bas agressif, ses hautes fréquences sont coupées. Son profil énergétique (décroissance rapide) devient alors spectralement similaire à celui d'un *Triangle* pur.
- **La Bipolar Pulse :** Bien que générée avec un paramètre de largeur (*sharp*) volontairement asymétrique pour éviter de recréer un *Square* parfait (largeur de 50%), cette onde conserve une empreinte rectangulaire. Dans les basses fréquences, il devient difficile pour le réseau de distinguer avec certitude la morphologie exacte de l'impulsion.

Malgré ces subtilités acoustiques, nous pouvons distinguer clairement une diagonale marquée sur la matrice de confusion. L'architecture globale démontre alors une capacité robuste à identifier les familles spectrales fondamentales, validant l'hypothèse d'assimilation de l'analyse audio à un problème de vision par ordinateur.

Conclusion

Ce projet de synthèse inverse a permis de démontrer avec succès la viabilité des réseaux de neurones convolutifs pour analyser et rétro-ingénierer des signaux audio complexes. En construisant une démarche de résolution complète allant de la génération procédurale des données avec *Pyo* jusqu'à l'élaboration d'un réseau de neurones convolutif sous *PyTorch*, j'ai pu confronter la théorie de l'apprentissage profond à la réalité physique de l'acoustique.

Le principal enseignement de ce travail réside dans l'adaptation indispensable de l'outil informatique à la nature du son. La traduction du domaine temporel vers le domaine fréquentiel (Spectrogramme Log-Mel) a permis de contourner le problème d'invariance de phase, tandis que la conceptualisation d'une architecture CNN atypique avec des noyaux verticaux asymétriques s'est avéré cruciale pour capter les subtilités des signatures harmoniques.

Si quelques ambiguïtés spectrales subsistent, ces excellents résultats valident la démarche et prouvent que l'alliance entre traitement du signal et vision par ordinateur peut automatiser efficacement le design sonore. Ces bases solides ouvrent naturellement la voie à de futures avancées : le passage à la **régression continue** pour prédire précisément les fréquences et enveloppes ADSR, ainsi que l'évolution vers une architecture **multi-oscillatoire** (sommation, synthèse FM) pour appréhender des environnements de synthèse encore plus complexes.

VI. Bibliographie

[1] LUTHER R. (Moog Archives). *The First Moog Synthesizer (1964)*. URL : `http://www.moogarchives.com/moog1.htm`

[2] MOOG R. A. *Voltage-Controlled Electronic Music Modules*, Journal of the Audio Engineering Society, Vol. 13, No. 3, pp. 200–206 (1965).

[3] SURGE SYNTH TEAM. *Surge XT : Open Source Hybrid Synthesizer*. URL : `https://github.com/surge-synthesizer/surge`

[4] BÉLANGER O. *Pyo : Python DSP module*. URL : `https://belangeo.github.io/pyo`

[5] PASZKE A., GROSS S., MASSA F., et al. *PyTorch : An Imperative Style, High-Performance Deep Learning Library*. Advances in Neural Information Processing Systems 32 (2019). URL : `https://pytorch.org`

[6] TORCHAUDIO CONTRIBUTORS. *Torchaudio : An audio library for PyTorch*. URL : `https://pytorch.org/audio`

[7] MCKINNEY W. *Data Structures for Statistical Computing in Python*. Proceedings of the 9th Python in Science Conference, pp. 51-56 (2010).

[8] BARKAN O., TSIRIS D. *InverSynth : Deep Estimation of Synthesizer Parameter Configurations from Audio Signals*. IEEE International Conference on Acoustics, Speech, and Signal Processing (2018).

[9] CHOI K., FAZEKAS G., SANDLER M., CHO K. *Convolutional Neural Networks for Automatic Musical Chord Recognition*. 17th International Society for Music Information Retrieval Conference (2016).

VII. Annexes

Cette section recense de manière synthétique les travaux théoriques et les projets technologiques majeurs qui ont orienté ma réflexion et justifié mes choix d'ingénierie tout au long de ce projet. L'intégralité du code source développé dans le cadre de ce stage est disponible en libre accès.

URL : https://github.com/ctrlpierre/synth-param-inference



Le paradigme *Analysis-by-Synthesis* (M. Antunes)

Théorisé par J.-C. Risset et récemment exploré par M. Antunes (2024) pour l'étude des textures musicales (les masses sonores de G. Ligeti), ce cadre conceptuel valide l'utilisation de la synthèse sonore itérative pour mieux comprendre les paramètres de la perception auditive.

URL : https://cloud.prism.cnrs.fr/index.php/s/CXYJzERte7x273Q



SynPlant 2 par Sonic Charge et *Genopatch*

Ce synthétiseur logiciel commercial intègre une technologie d'IA capable d'écouter un échantillon audio pour en déduire instantanément les paramètres de synthèse. Il incarne l'aboutissement applicatif industriel de la "synthèse inverse" abordée dans ce rapport.

URL : https://soniccharge.com/synplant



Google Magenta et l'architecture DDSP

Les travaux de Google sur le *Differentiable Digital Signal Processing* démontrent la pertinence de coupler des réseaux de neurones avec des moteurs DSP (traitement du signal) stricts, confortant directement mon choix d'interfacer PyTorch avec Pyo.

URL : https://magenta.withgoogle.com/ddsp-vst



Le projet *Inver-Synth* (C. Rodriguez)

Ce dépôt open-source propose une implémentation pratique de la synthèse inverse par apprentissage automatique. L'étude de son code source et de son approche pour l'estimation des paramètres de synthétiseurs a constitué une référence technique pertinente pour confronter et structurer ma propre architecture prédictive.

URL : https://github.com/crodriguez1a/inver-synth

