Rapport de projet

Classification et génération de sons

Image générée par l'intelligence artificielle Bing

Clément Souvignet

Tristan Miotto

IMDS4A, 2023/2024

Sommaire

Clement Souvignet, Tristan Miotto
IMDS 4A, 2023/2024

Abstract (English)

Currently, neural networks can be used in all fields. Once you have a relatively important amount of data, you can use Deep Learning models to extract valuable information from it. Moreover, neural networks can process any type of data: numbers, text, pictures and even sounds.

Sounds are recorded by a voice recorder and transformed into images called spectrograms. From these spectrograms, it is possible to build powerful neural networks that will perform many tasks: speech recognition, music classification, music generation, deepfake voice...

The objective of this report is to present sound processing on Python (using *Librosa*), as well as associated Deep Learning models (using *Keras*).
More specifically, we first provide theoretical information. Secondly, we apply all this knowledge to create models capable of:

- classify numbers pronounced in English.
- find the gender of a person.
- predict a person's age.
- generate virtual voices that pronounce a number in English.

Résumé (français)

Aujourd'hui, les réseaux de neurones trouvent un usage dans tous les domaines. À partir du moment où on possède un nombre de donnés conséquent, on peut utiliser des modèles de Deep Learning pour en tirer des informations précieuses. De plus, les réseaux de neurones sont capables de traiter des données de tout type : nombre, texte, images, voire des sons.

Les sons sont capturés par des enregistreurs vocaux et sont transformés en images appelées spectrogrammes. À partir de ceux-ci, il est possible de construire des réseaux de neurones très puissants qui vont réaliser plusieurs tâches : reconnaissance vocale, classification de musique, génération de musique, imitation de la voix d'une personne...

Le rapport a pour but de présenter des techniques de traitement de son sur Python (avec la bibliothèque *Librosa*), ainsi que des modèles de Deep Learning associés (avec *Keras*).
Plus précisément, nous donnons dans un premier temps des informations théoriques. Dans un second temps, nous appliquons toutes ces connaissances pour créer des modèles capables de :

- classifier des chiffres prononcés en anglais;
- trouver le genre d'une personne;
- prédire l'âge d'une personne;
- générer des voix virtuelles qui prononcent un chiffre en anglais.

Keywords (Mots-clés)

English	**Français**
Python	Python
Keras	Keras
Librosa	Librosa
Artificial Intelligence	Intelligence Artificielle
Machine Learning	Machine Learning
Deep learning	Apprentissage profond
Neural network	Réseau de neurons
Sound	Son
Audio	Audio
Spectrogram	Spectrogramme
Classification	Classification
Auto-encoder	Auto-encodeur
Variational auto-encoder	Auto-encodeur variationnel
Generative model	Modèle génératif

Remerciements

Nous tenons fortement à remercier notre tuteur de projet, M. Cédric Chauvière. Les réunions hebdomadaires nous ont permis de guider notre travail tout en nous laissant la liberté de créer de nouvelles choses.

Nous souhaitons également remercier toutes les personnes qui ont enregistré leur voix afin de tester nos programmes.

1 Traitement du son – Théorie

Cette partie a plusieurs objectifs :

1. Comprendre ce qu'est un son, comment il est enregistré et reproduit.
2. Savoir comment le représenter et connaître ses principales caractéristiques.
3. Comprendre la transformée de Fourier et les algorithmes associés.

Après la compréhension de ces différentes notions théoriques, nous verrons comment réaliser du traitement de sons sur Python.

1.1 Définition physique du son

Pour obtenir un son, on doit se placer dans un espace dans lequel des différences de pression entre les particules ont la possibilité de se propager.

Exemples : l'air, l'eau. Le vide est un contre-exemple.

Maintenant, il nous faut deux éléments.

- Une **source**, capable de vibrer, c'est-à-dire d'osciller autour de sa position d'équilibre avec une faible amplitude et une grande rapidité. Lors du mouvement, les particules situées autour de la source vont se retrouver plus ou moins comprimées. On a alors des différences de pression qui vont se propager dans l'espace : c'est ce qu'on appelle une **onde sonore**.

Exemples de source : les cordes vocales d'un humain, la corde d'une guitare.

- Un **récepteur**, capable de percevoir le son.

Exemples de récepteurs : le système auditif d'un humain, un microphone.

Pour comprendre les prochains concepts, l'espace étudié sera naturellement l'air.

1.2 Enregistrement du son

Considérons une source sonore et un enregistreur vocal, composé d'un diaphragme. Les ondes sonores vont frapper le diaphragme qui va se mouvoir. Le mouvement ainsi créé génère un courant électrique (généralement en déplaçant un fil électrique autour d'un aimant).

La quantité de compressions et raréfactions détermine les **fréquences** du son et la force appliquée correspond à **l'amplitude**.
Ainsi, on obtient un signal audio, qui encode toutes les informations dont on a besoin pour reproduire le son.

Toutefois, ce signal audio n'est, en pratique, pas continu. En effet, on ne peut pas mesurer la totalité des amplitudes d'un son car celui-ci est continu.
On mesure donc des amplitudes à intervalles réguliers. Le **taux d'échantillonnage** correspond au nombre d'échantillons mesurés pendant une seconde. Ainsi, si cette valeur vaut 44100 Hz, une mesure est réalisée toutes les $\frac{1}{44100} \times 10^6 \approx 22{,}7$ µs. On comprend alors que plus le taux d'échantillonnage est élevé, meilleure est la qualité audio.

Pour reproduire un son (haut-parleur par exemple), on décode le signal audio pour le transformer en onde sonore.

1.3 Domaine temporel du son

1.3.1 Représentation temporelle d'un son

Pour représenter une onde sonore, on utilise un graphique avec les composantes suivantes.

- En ordonnée, la pression de l'onde sonore mesurée par le récepteur, ou amplitude.
- En abscisse, le temps.

Imaginons une source qui oscille à intervalle régulier et à une même distance de son point d'équilibre. Il s'agit d'un **son pur**, qui est simple à étudier.

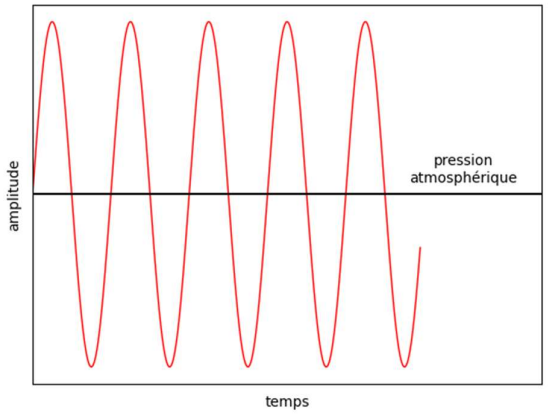


Figure 1.1 Représentation d'un son pur

Intuitivement, on aurait pu s'attendre à une telle courbe.

- Le fait que la source oscille à même distance du point d'équilibre donne les mêmes maximums locaux d'amplitudes.

- Le fait qu'il oscille à intervalles réguliers donne les mêmes périodes.

On a donc une fonction sinusoïdale f définie de la façon suivante :

$$f(t) = A\,sin\left(\frac{2\pi t}{T} + \varphi\right)$$

avec A l'amplitude maximale, T la période et φ le déphasage.

Exemples de sons purs : un diapason, une note de musique.

Cependant, la plupart des sons ne sont pas des sons purs, mais des **sons complexes**.

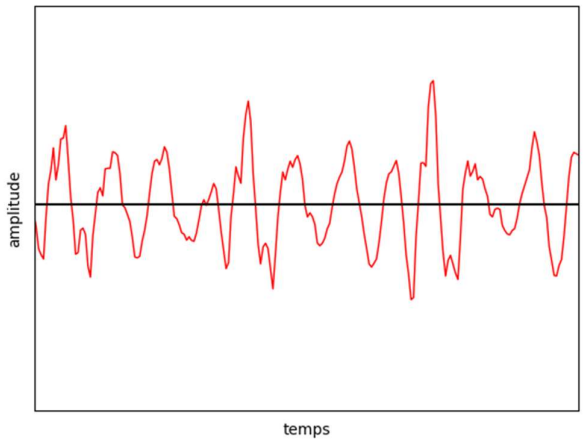


Figure 1.2 Représentation d'un son complexe

1.3.2 Principales caractéristiques temporelles d'un son

À partir de la représentation temporelle d'un son, on peut déterminer différentes caractéristiques telles que sa puissance, son intensité, son volume…

La puissance d'un son

Énergie par unité de temps émise par une source, mesurée en watt (W).

L'intensité d'un son

- Puissance du son par unité d'aire (W/m²).
- L'intensité est proportionnelle à l'amplitude au carré. En effet, on a :

$$I = \alpha A^2$$

avec α constante qui dépend notamment de la densité du milieu extérieur.

- Seuil d'entendement noté $I_{TOH} = 10^{-12}$ W/m².
- Seuil de douleur $I_{TOP} = 10$ W/m².

Le niveau d'intensité d'un son

- Ratio entre deux intensités exprimées en décibels (dB). En général, on choisit I_{TOH} comme intensité de référence.
- Cette quantité se mesure sur une échelle logarithmique, du fait de l'ordre de grandeur important entre I_{TOH} et I_{TOP} (10^{13}).

- Soit I l'intensité d'un son mesurée. Alors son niveau d'intensité est donné par la formule suivante :

$$\boldsymbol{DB(I) = 10\, log_{10}\left(\frac{I}{I_{TOH}}\right)}$$

- On a $DB(I_{TOH}) = 0$. En effet :

$$DB(I_{TOH}) = 10\, log_{10}\left(\frac{I_{TOH}}{I_{TOH}}\right) = 10\, log_{10}(1) = 0$$

- Tous les 3 dB environ, l'intensité est doublée. En effet, soit I une intensité.

$$DB(2I) = 10\, log_{10}\left(\frac{2I}{I_{TOH}}\right) = 10log_{10}(2) + 10log_{10}\left(\frac{I}{I_{TOH}}\right) = 3{,}01 + DB(I)$$

Le volume d'un son

- Le volume est une perception subjective de l'intensité d'un son.
- Il dépend évidemment de l'intensité du son, mais aussi de l'âge de la personne qui écoute le son, et de la fréquence de l'onde sonore (on parlera des fréquences plus tard).
- Il dépend aussi de la durée du son. Par exemple, un son de 3 dB qui dure 500 ms sera perçu plus fort qu'un son avec les mêmes caractéristiques mais qui dure 100 ms.

1.4 Domaine fréquentiel du son

1.4.1 Fréquence(s) d'un son

Fréquence d'un son pur

La fréquence d'un son pur est l'inverse de la période T. Elle correspond au nombre de cycles par seconde, mesurée en Hz. Les humains perçoivent des fréquences situées entre 20 et 20 000 Hz.

Fréquence d'un son complexe

D'après la décomposition en séries de Fourier, on sait que tous les sons complexes peuvent se décomposer en somme de sons purs. Dans ce cas, on ne parle pas d'une unique fréquence, mais d'une distribution de fréquences.

1.4.2 Partiels et hauteur

On appelle **partiels** toutes les sinusoïdes qui composent un son complexe. On trie les partiels par ordre croissant des distributions de fréquences.

- Le plus petit partiel est appelé **fréquence fondamentale**.
- Tous les partiels qui correspondent à une fréquence multiple de la fréquence fondamentale sont appelés **partiels harmoniques**.

- Un **son harmonique** est un son qui n'est composé que d'une fréquence fondamentale et de partiels harmoniques.

La distribution des fréquences d'un son complexe n'est pas aléatoire. En effet, chaque son présente une certaine harmonicité. Par exemple, certains instruments de musique, comme le violon, tendent à être parfaitement harmoniques.

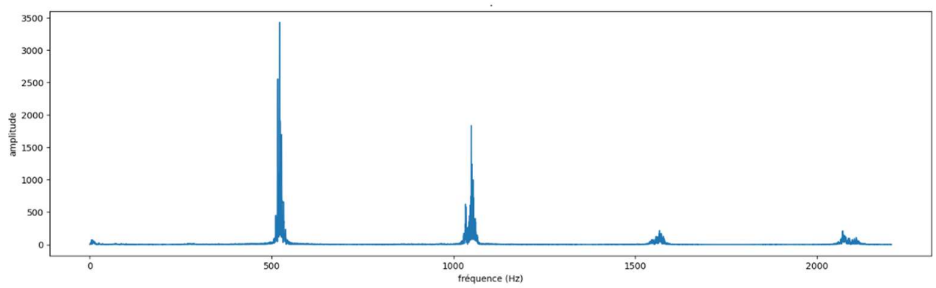
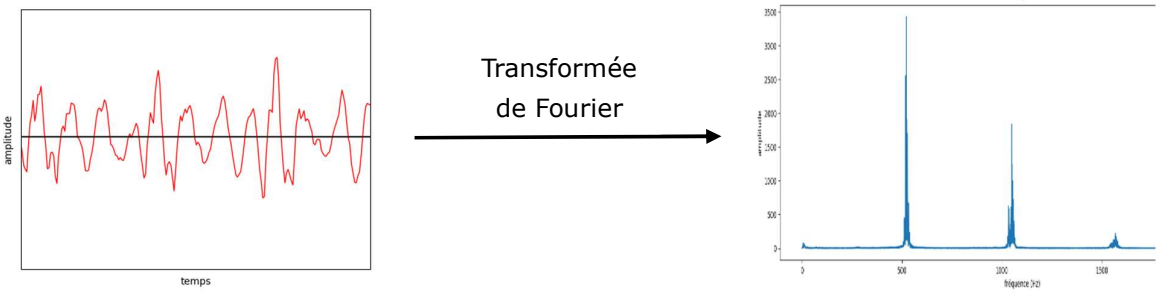


Figure 1.3 Distribution des fréquences d'un violon jouant la note C4

L'**inharmonicité** d'un son correspond aux variations des partiels par rapport aux multiples de la fréquence fondamentale.
La **hauteur** désigne notre perception des fréquences.
Deux fréquences sont perçues plus ou moins de la même l'une est un multiple de l'autre. Par exemple, on aura une perception similaire d'une fréquence de 440 Hz et d'une autre de 880 Hz. La plus haute fréquence sera toutefois perçue comme étant légèrement plus aigüe.

1.5 La transformée de Fourier

La transformée de Fourier est énormément utilisée pour les traitements du son car elle permet d'accéder à de précieuses informations. Il s'agit d'une transformation qui permet de passer d'une représentation temporelle (amplitude en fonction du temps) d'un son à une représentation fréquentielle (amplitude en fonction des fréquences).



1.5.1 Définition

- Dans le cas où f est une fonction continue, sa transformée de Fourier est la fonction $\hat{f}$ définie par :

$$\hat{f}(t) = \int_{-\infty}^{+\infty} f(x) e^{-2i\pi xt}\, dx$$

On trouve parfois des variantes de cette formule, à une constante près.

- Dans le cas discret où $(F_0, F_1, \dots, F_{n-1})^T$ est un vecteur qui contient un échantillon de taille n, sa **transformée de Fourier discrète** est le vecteur $(\hat{F}_0, \hat{F}_1, \dots, \hat{F}_{n-1})^T$ défini de façon analogue par :

$$\hat{F}_j = \sum_{k=0}^{n-1} F_k e^{-2i\pi j\frac{k}{n}} \qquad \forall j \in [\![0; n-1]\!]$$

1.5.2 Fast Fourier Transform (FFT)

Comme un signal sonore contient des données discrètes (à l'opposé de continu), on va s'intéresser à la transformée de Fourier discrète. Si on devait créer un premier algorithme naïf, on aurait une complexité $o(n^2)$: une boucle de taille n qui calcule chaque $\hat{F}_j$ et, à l'intérieur, une autre boucle de taille n qui exécute la somme.

Pour calculer la transformée de Fourier discrète plus rapidement et passer d'une complexité $o(n^2)$ à $\boldsymbol{o(n \log n)}$, on utilise généralement l'algorithme récursif FFT.
Pour simplifier les calculs, on prend un vecteur $(F_0, F_1, \dots, F_{n-1})^T$ de dimension $n = 2^a, a \in \mathbb{N}$, même si la formule se généralise quel que soit $n \in \mathbb{N}^*$.

- On fixe j dans $[\![0\,; n-1]\!]$ et on cherche donc $\hat{F}_j = \sum_{k=0}^{n-1} F_k e^{-2i\pi j\frac{k}{n}}$.
- On pose $\omega = e^{-\frac{2i\pi}{n}}$ et P un polynôme tel que $P = \sum_{k=0}^{n-1} F_k X^k$.
- On a donc $\hat{F}_j = \sum_{k=0}^{n-1} F_k \omega^{kj} = P(\omega^j)$.

L'algorithme est basé sur deux points clefs :

- Si $n = 1$, $\hat{F}_0 = F_0$
- Sinon, on peut décomposer P en deux polynômes P_{pair} et P_{impair} tel que :

$P(X) = P_{pair}(X^2) + X\, P_{impair}(X^2)$, où $P_{pair}(X) = \sum_{k=0}^{\frac{n-1}{2}} F_{2k}\, X^k$ et $P_{impair}(X) = \sum_{k=0}^{\frac{n-2}{2}} F_{2k+1}\, X^k$

Preuve

- Si $n = 1$, $\hat{F}_0 = P(\omega^0) = \sum_{k=0}^{0} F_0 (\omega^0)^0 = F_0$
- $P_{pair}(X^2) + X\, P_{impair}(X^2) = \sum_{k=0}^{\frac{n-1}{2}} F_{2k}\, (X^2)^k + X \sum_{k=0}^{\frac{n-2}{2}} F_{2k+1}\, (X^2)^k$
  $= \sum_{k=0}^{\frac{n-1}{2}} F_{2k}\, X^{2k} + \sum_{k=0}^{\frac{n-2}{2}} F_{2k+1}\, X^{2k+1}$
  $= F_0 X^0 + F_2 X^2 + \cdots + F_{n-2} X^{n-2} + F_1 X^1 + F_3 X^3 + \cdots + F_{n-1} X^{n-1}$

$= \sum_{k=0}^{n-1} F_k X^k = P(X)$

Algorithme FFT (Cooley-Tukey)

Entrée : $P = \sum_{k=0}^{n-1} F_k X^k$ et $\omega = e^{-2i\pi/n}$
Sortie : $\left(P(\omega^0);\ P(\omega^1);\ \dots;\ P(\omega^{n-1})\right) = (\hat{F}_0;\ \hat{F}_1;\ \dots;\ \hat{F}_{n-1})$

$\boldsymbol{FFT(P, \omega)}$:

Si $n = 1$: (P est constant)

Retourner $P(0)$

Sinon :

Calculer P_{pair} et P_{impair}

$\left(P_{pair}(\omega^0);\ P_{pair}(\omega^2);\ \dots;\ P_{pair}(\omega^{n-2})\right) = FFT(P_{pair}, \omega^2)$

$\left(P_{impair}(\omega^0);\ P_{impair}(\omega^2);\ \dots;\ P_{impair}(\omega^{n-2})\right) = FFT(P_{impair}, \omega^2)$

Pour j allant de 0 à $n-1$:

$P(\omega^j) = P_{pair}(\omega^{2j}) + \omega^j\, P_{impair}(\omega^{2j})$

Retourner $(P(\omega^0);\ P(\omega^1);\ \dots;\ P(\omega^{n-1}))$

1.5.3 Short Time Fourier Transform (STFT)

Toutefois, la FFT présente un inconvénient : la notion de temps est perdue. Pour régler ce problème, l'idée de la STFT est de découper notre signal audio en plusieurs fenêtres de temps, et on applique la FFT sur chaque extrait.

Imaginons que nous découpons notre signal en M fenêtres. On obtient alors une représentation fréquentielle pour chaque extrait. On rappelle qu'une représentation fréquentielle contient les amplitudes des différentes gammes de fréquences.
On stocke toutes les données dans un **spectrogramme**. Il s'agit d'une matrice composée de :

- M colonnes. La m-ème colonne correspond à la m-ème fenêtre de temps ;
- J lignes, où j est le nombre de gammes de fréquences considéré.

Ainsi, la valeur du coefficient ligne m colonne j correspond à l'amplitude de la m-ème gamme de fréquence de la j-ème fenêtre.

Pour obtenir la formule de la STFT, on part de la transformée de Fourier discrète avec les mêmes notations que précédemment :

$$\hat{F}_j = \sum_{k=0}^{n-1} F_k\, e^{-2i\pi j \frac{k}{n}} \qquad \forall j \in [\![0; n-1]\!]$$

Ensuite, il suffit d'utiliser une fenêtre temporelle qu'on note w. w est une fonction qui est appliquée au signal audio afin de le diviser en segments de temps courts et de les rendre

appropriés pour une analyse fréquentielle. Elle permet aussi de sélectionner des portions spécifiques du signal temporel F_k à un moment donné m.
On obtient ainsi un spectrogramme S où $S_{m,j}$ correspond à l'amplitude à une gamme de fréquence donnée j d'une fenêtre temporelle donnée m :

$$\hat{F}_{m,j} = \sum_{k=0}^{n-1} F_k \, w(k-m) \, e^{-2i\pi j \frac{k}{n}} \qquad \forall m [\![1; M]\!], \forall j \in [\![0; n-1]\!]$$

2 Traitement de son – sur Python

Dans cette partie, nous allons utiliser ***Librosa***, la bibliothèque de référence pour réaliser du traitement sonore sur Python. *Librosa* est un outil très puissant qui permet d'effectuer les algorithmes de la transformée de Fourier, mais aussi d'autres opérations très utiles que nous détaillons ici.
On va voir ici les deux éléments liés à des audios les plus utilisés en entrée d'un réseau de neurones.

2.1 Les spectrogrammes

2.1.1 Spectrogrammes utilisés en Deep Learning

Généralement, quand on travaille sur un réseau de neurones qui apprend sur des sons, on lui fait passer en entrées les spectrogrammes associés aux STFT des audios. Le réseau interprète le spectrogramme comme une image.

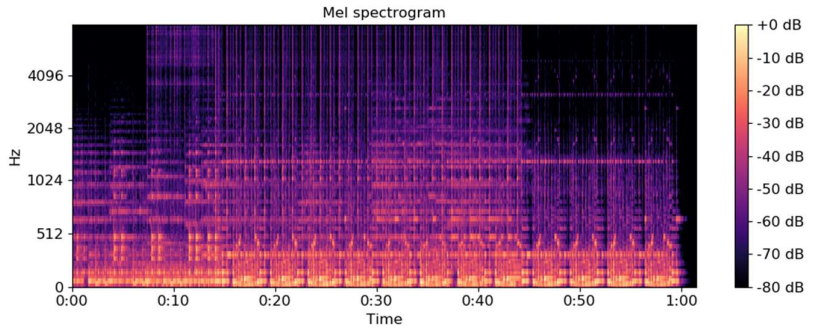


Figure 2.1 Exemple de spectrogramme envoyé dans un réseau de neurones

Comme on peut le voir sur l'image ci-dessus, on utilise des échelles non linéaires (sauf pour le temps).

- Une échelle logarithmique pour les fréquences, appelée **échelle Mel.**
- Une échelle logarithmique pour l'intensité : les **décibels**.

Ceci se justifie du fait qu'on perçoit les amplitudes et aussi les fréquences de façon logarithmique.

Pour obtenir un tel spectrogramme à partir d'un son, on utilise les fonctions suivantes.

```
import librosa as lb

signalAudio, tauxEchantillonage = lb.load("./sound.mp3") # chargement du son
S = lb.feature.melspectrogram(y=signalAudio, sr=tauxEchantillonage) # amplitude au carré / échelle mel
S = lb.power_to_db(S) # Db / échelle Mel
```

S est une matrice de type *numpy.ndarray.*

- D'abord, la fonction *feature.melspectrogram* applique la STFT au signal audio. Les transformées de Fourier renvoyant des nombres complexes, la fonction renvoie finalement les carrés des modules (ou amplitudes) des complexes.
- Ensuite, on veut généralement passer notre spectrogramme en décibels, qui est une échelle plus naturelle pour les humains (tout comme l'échelle Mel). On utilise donc la fonction *power_to_db*.

2.1.2 Influence des paramètres

Il est indispensable de savoir gérer les dimensions de nos spectrogrammes, qui vont passer en entrée de nos réseaux de neurones. La dimension d'un spectrogramme est déterminée à partir des principaux paramètres de la fonction *librosa.feature.melspectrogram*, qui effectue une STFT.

- ***hop_length*** : distance entre les positions successives de la fenêtre.

- ***n_fft*** : nombre d'échantillons utilisés dans chaque fenêtre de la STFT. On doit avoir *n_fft* ≥ *hop_length*. Ce paramètre permet d'obtenir une certaine régularité entre les fenêtres qui se suivent.

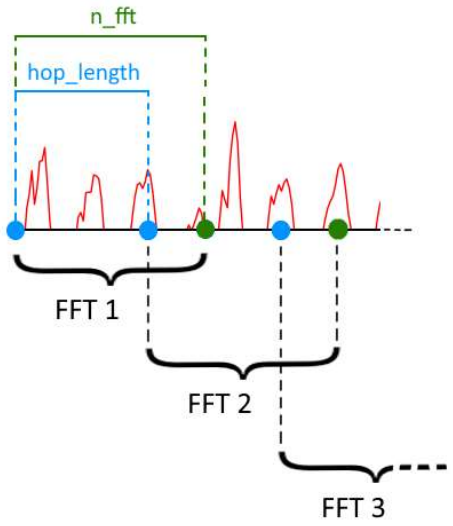


Figure 2.2 Illustration du fonctionnement des paramètres *hop_length* et *n_fft*

- ***n_mels*** : taille du maillage des fréquences de la STFT. Généralement, cette valeur se situe entre 20 et 128.

Dimension du spectrogramme

- Nombre de lignes : *n_mels*
- Nombre de colonnes : 1 + (L // *hop_length*)

où L correspond à la longueur du signal audio (taux d'échantillonnage multiplié par la durée du son en secondes), et // désigne la division entière.
Ainsi, si on souhaite créer un spectrogramme de taille (x, y), alors on doit avoir :

- ***n_mels*** $= x$;
- $\frac{L}{y-1} \leq$ *hop_length* $\leq \frac{L+1}{y}$. On pourra donc prendre ***hop_length*** $= \left\lfloor \frac{L+1}{y} \right\rfloor$.

Qualité du spectrogramme

Plus le spectrogramme est grand, plus il contient d'informations et préserve donc la qualité audio. Toutefois, les calculs seront naturellement plus longs.
Ainsi, si on augmente *n_mels*, on augmente la qualité. Au contraire, si on augmente *hop_length*, on perd de l'information et donc de la qualité (ce qui est logique puisqu'on crée moins de fenêtres de temps).

2.2 Mel Frequency Cepstral Coefficients

Les *Mel Frequency Cepstral Coefficients* (MFCC) sont une autre représentation de caractéristiques extraites d'un signal audio. Tout comme les spectrogrammes vus précédemment, les MFCC sont largement utilisés dans les entrées de modèles de Deep Learning. Ils sont utilisés dans diverses applications.

- La reconnaissance de la parole.
- La détection d'événements sonores.
- D'autres tâches de traitement du son.

Ils sont particulièrement utiles pour capturer des informations importantes du son tout en réduisant la taille du vecteur de caractéristiques, ce qui diminue les temps de calculs.

Les MFCC sont extraits en effectuant les étapes suivantes.

1. Calcul du spectrogramme (avec les paramètres *n_fft* et *hop_length*)
2. Transformation de l'échelle des fréquences en échelle Mel.
3. Application de la fonction log à toutes les amplitudes du spectrogramme.
4. Transformée de Fourier inverse sur le log-spectrogramme.

On obtient alors une matrice de type *numpy.ndarray* de dimension analogue au spectrogramme classique. La seule différence est le paramètre *n_mels* qui est remplacé par ***n_mfcc***, qui lui vaut généralement entre 12 et 20.

Pour obtenir les MFCC d'un son sur *Python*, on utilise les fonctions suivantes.

```
import librosa as lb

signalAudio, tauxEchantillonage = lb.load("./sound.mp3") # chargement du son
MFCC = lb.feature.mfcc(y=signalAudio, sr=sr, n_fft=nFFT, hop_length=hopLength, n_mfcc=nMFCC)
```

3 Deep Learning – Théorie

Dans cette partie, nous donnons des explications générales autour des réseaux de neurones. Sans rentrer dans les détails, nous donnons des éléments qui permettront une meilleure compréhension de notre travail. À la fin de ce rapport, nous mettons à disposition des sources qui présentent en profondeur les concepts introduits.

3.1 Qu'est-ce que le Deep Learning ?

Le Deep Learning peut être traduit par « apprentissage profond ». On lui associe aussi le terme « réseau de neurones ». Un réseau de neurones est un modèle inspiré du fonctionnement du cerveau humain, et conçu pour effectuer des **prédictions**, de la **compression** de données, de la **génération** de données…

Pour faire fonctionner un réseau de neurones, on doit lui fournir une base de données $(x^{(i)}, y^{(i)})_{1 \leq i \leq n}$ qui contient n données d'entrées $x^{(i)} = (x_1^{(i)}, \dots, x_p^{(i)})$ et les n sorties correspondantes $y^{(i)}$ qu'on appelle parfois les **données cibles**. On parle donc **d'apprentissage supervisé**.

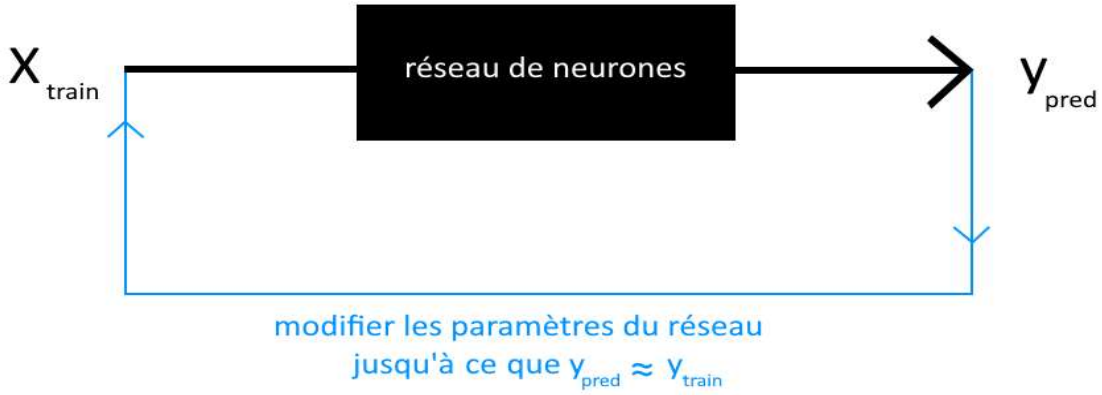


Figure 3.1 Phase d'apprentissage d'un réseau de neurones

1. Au début, le réseau initialise ses paramètres aléatoirement.
2. Ensuite, il va prendre en entrée les données X_{train} puis effectuer des calculs avec ses paramètres pour donner une valeur en sortie qu'on appelle y_{pred}.
3. Le modèle va répéter l'étape 2 et mettre ses paramètres à jour jusqu'à ce que la valeur y_{pred} qu'il prédit soit proche des vraies données de sortie y_{train}. C'est la **phase d'apprentissage**.

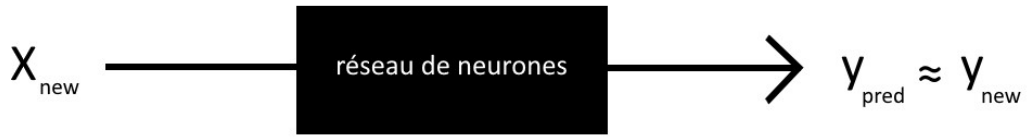


Figure 3.2 Prédiction d'un réseau de neurones

Ainsi, lorsqu'on entre de nouvelles données X_{new} à notre réseau, il sera en mesure de prédire une valeur y_{pred} proche de la vraie valeur y_{new} qu'on cherche à estimer.

Comment sait-on si $y_{pred} \approx y$?

Le réseau de neurones utilise une **fonction d'erreur**, ou fonction coût pour évaluer la distance entre y_{pred} et y. Il existe plusieurs fonctions d'erreurs. Les plus connues sont la ***mean square error*** (problème de régression) et la ***binary crossentropy*** (problème de classification binaire).

La fonction d'erreur renvoie toujours un nombre réel positif. Elle dépend des paramètres du modèles et des données d'entrée et de sortie. L'objectif du réseau de neurones est de minimiser cette fonction en faisant varier les paramètres du modèle (les données sont fixes). En effet, plus la valeur est proche de 0, plus y_{pred} est proche de y.

Mise à jour des paramètres lors de la phase d'apprentissage

Afin de minimiser la fonction d'erreur, le réseau va, lors de sa phase d'apprentissage, effectuer l'algorithme de ***backpropagation***. L'idée derrière cet algorithme est de calculer le gradient $\vec{g}$ de la fonction d'erreur et de faire en sorte qu'il se rapproche de 0. Ainsi, en déplaçant les paramètres du modèle de $-\alpha\vec{g}, \alpha \in \mathbb{R}^+$, les paramètres convergeront vers le minimum local de la fonction coût. α est appelé le taux d'apprentissage.

Le modèle va donc parcourir plusieurs fois les données X_{train} et ajuster ses paramètres en suivant le principe de *backpropagation*. Lorsque le réseau parcourt une fois toutes les données d'entrée X_{train}, on dit qu'il a réalisé une **époque**.

Pour améliorer l'algorithme de *backpropagation*, on a notamment deux solutions.

- Utiliser un **optimiseur** qui permet d'accélérer la convergence du modèle vers un minimum de la fonction coût. Généralement, on choisit *Adam* ou *rmsprop* car ils fonctionnent très bien en pratique.
- Diviser nos données d'entrées en plusieurs lots pour éviter qu'une itération de l'algorithme ne soit trop coûteuse en mémoire. On verra souvent le terme ***batch size*** qui correspond à la taille des lots. Ainsi, on effectuera une mise à jour du modèle pour chaque lot.
  Si, par exemple, on a 1 000 données, un *batch size* valant 10 et un nombre d'époques fixé à 5, le modèle mettra à jour ses paramètres $5 \times \frac{1\,000}{10} = 500$ fois ($\frac{1\,000}{10}$ représente le nombre de lots).

3.2 Les couches de neurones

Plus précisément, un réseau de neurones est composé de neurones artificiels interconnectés, organisés en couches. Ces neurones sont en fait des fonctions mathématiques. On peut couper un réseau en trois parties :

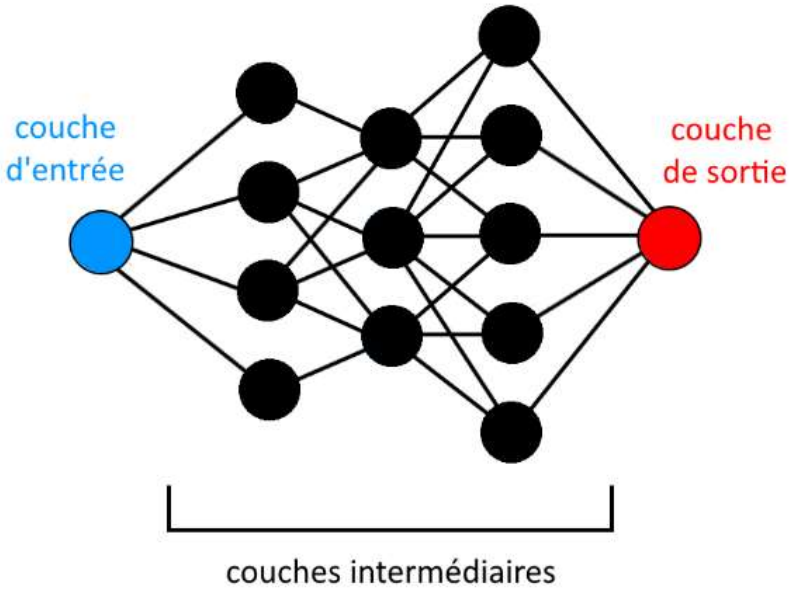


Figure 3.3 Représentation schématique d'un réseau de neurones

1. Une **première couche** qui contient les données X_{train}.
2. Les **couches intermédiaires** du réseau, qui contiennent les paramètres que le modèle va entrainer lors de la phase d'apprentissage. Dans l'exemple ci-dessus, le modèle est composé de trois couches intermédiaires.
3. La **dernière couche** du réseau, qui donne la sortie prédite par le modèle.

On rappelle que les données y_{train} sont bien connues par le modèle et utilisées pour minimiser l'erreur.
Les couches intermédiaires sont composées de couches dont le fonctionnement et la structure varie. On retrouve, entre autres, les couches denses et les couches convolutionnelles.

Les couches denses

Une couche dense est une couche entièrement connectée : chaque neurone est connecté à chaque neurone de la couche précédente, formant ainsi une connexion dense entre les couches.
Concrètement, une couche dense est représentée par une matrice $\boldsymbol{W \epsilon M_{(m,n)}}$, un vecteur $\boldsymbol{b} \in \boldsymbol{R^m}$ et une fonction d'activation (voir page suivante). W est appelée la matrice des poids et

b le biais. La couche de neurones renvoie le vecteur $a_i = (W \times a_{i-1} + b) \in \mathbb{R}^m$ où $a_{i-1} \epsilon \mathbb{R}^n$ contient les valeurs de la couche précédente. Ce vecteur a_i sera ensuite utilisé par la couche suivante. Les coefficients de W et de b sont les paramètres que le modèle va entrainer tout au long de la phase d'apprentissage. Dans ce cas, n représente le nombre de neurones de la couche actuelle et m le nombre de neurones de la couche précédente.
On comprend donc que plus un réseau possède de neurones, plus il possède de paramètres à entrainer.

Les couches convolutionnelles

Une couche convolutionnelle (ou convolutive) se compose d'une petite matrice appelée **filtre** (ou noyau). Le filtre va appliquer des opérations de convolutions à la sortie de la couche précédente. Le filtre se place d'abord sur une extrémité des données, puis "glisse" sur l'ensemble des données. Une convolution est calculée à chaque position, produisant une nouvelle séquence de sortie.
Il est notamment possible de définir la taille du filtre, mais aussi la façon dont il parcourt les données. Le modèle va ajuster les coefficients du filtre lors de la phase d'entrainement.
Les couches convolutionnelles s'avèrent être particulièrement efficaces lorsqu'on manipule des **images**. En effet, les filtres sont capables de détecter des motifs locaux dans les données d'entrée.

Les fonctions d'activation

Les opérations mathématiques cachées derrière un réseau de neurones sont des fonctions linéaires. Pour introduire de la non-linéarité et garder un certain contrôle dans notre modèle, on associe à chaque couche une fonction d'activation. Il s'agit d'une fonction qui va s'appliquer à la sortie de la couche.
Considérons par exemple un réseau de couches denses. Une couche intermédiaire va alors renvoyer $\sigma(a_i) = \sigma(W \times a_{i-1} + b)$ avec σ la fonction d'activation considérée.
Voici les fonctions d'activation les plus connues, qui ont démontré leur efficacité en pratique :

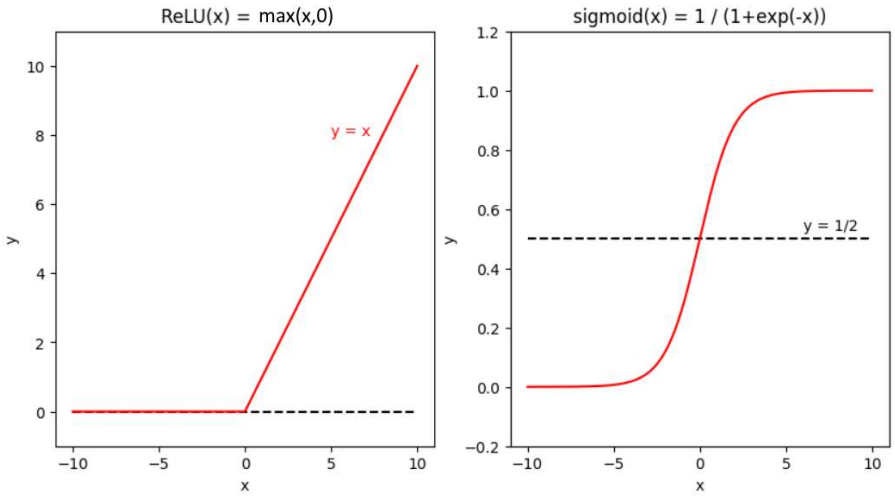


Figure 3.4 Visualisation de la fonction ReLU et sigmoid

- La fonction ***ReLU*** (*Rectified Linear Unit*) est à valeurs dans $\mathbb{R}^+$. Ainsi, chaque couche de neurones qui utilise cette fonction retournera des valeurs positives. Elle est très utilisée dans les couches intermédiaires. Elle peut tout aussi bien être utilisée en sortie finale du modèle si on travaille sur un problème de **régression** où les données de sorties sont forcément positives (le prix d'une maison, par exemple).

- La fonction ***sigmoid*** est à valeurs dans $[0,1]$. Elle est généralement utilisée pour la dernière couche du réseau lorsqu'on traite un problème de **classification binaire**. La sortie sera donc un nombre entre 0 et 1, qui représentera en fait une probabilité.

- Dans le cadre d'une **classification multiple** (y est à valeurs dans $[\![0,n]\!]$ où n est un entier strictement plus grand que 1), on utilise la fonction ***softmax***. Il s'agit d'une généralisation de la fonction *sigmoid*.

Il est fondamental d'identifier la bonne fonction d'activation en sortie. Le choix de celle-ci dépendra du problème étudié. Pour les couches intermédiaires, on peut se permettre de tester plusieurs fonctions d'activations, bien que *ReLU* fonctionne généralement très bien.

3.3 Classification – la procédure à suivre

Ici, nous détaillons les bonnes pratiques pour résoudre un problème de classification. En effet, beaucoup d'étapes sont essentielles pour arriver à un bon modèle.
Rappelons qu'on distingue deux cas de classification.

- La classification binaire, avec $y \in \{0,1\}$. On utilisera donc la *binary crossentropy* comme fonction d'erreur. La fonction d'activation de la dernière couche sera nécessairement la fonction *sigmoid*.
- La classification multiple, avec $y \in [\![0,n]\!]$, $n > 1$. Ici, on utilisera la fonction *categorical crossentropy* pour évaluer l'erreur, et la fonction *softmax* pour la dernière couche du modèle.

Étapes pour obtenir un modèle performant

1. Récupération (et traitement éventuel) de la base de données $\left(x^{(i)}, y^{(i)}\right)_{1 \leq i \leq n}$.

2. **Normalisation des données** d'entrée. On peut normaliser entre 0 et 1 (avec le minimum et le maximum des vecteurs X) ou obtenir une distribution normale (avec la moyenne et l'écart-type des vecteurs X).

3. **Séparation des données** en trois paquets : données d'entrainement, données de validation et données test. Généralement, on les répartit comme ceci : 60% entrainement, 20% validation, 20% test. Les données d'entrainement et de validation seront utilisées pour ajuster les paramètres du modèle, et les données test ne seront utilisées qu'à la fin pour évaluer le modèle final.

4. **Création du modèle**, avec des couches denses et/ou convolutionnelles.

5. **Phase d'apprentissage** : Le modèle utilise les données d'entrainement pour mettre à jour ses paramètres. Pour évaluer son efficacité, on testera le modèle sur les données de validation qu'il n'aura jamais vu auparavant.

6. **Méthode des K-folds**, ou validation croisée : on réunit nos données d'entrainement et de validation, puis on applique la méthode des K-folds. On se retrouve finalement avec K sous-modèles.

7. **Évaluation du modèle** : ici, on utilise uniquement nos données test, que le modèle ne connait pas encore. On aura donc pour chaque donnée K prédictions puisqu'on à K sous-modèles. Pour une donnée, on lui associe la prédiction qui revient le plus souvent. On obtient finalement un pourcentage de précision qui correspond aux performances du modèle final.

Zoom sur la phase d'apprentissage

Cette phase est probablement la plus longue. Si on souhaite trouver le meilleur modèle, il faudrait tester plein de configurations possibles. On effectue donc une ***grid search*** : on va tester différentes valeurs d'**hyperparamètres** et voir quelles sont les valeurs optimales.

Les hyperparamètres sont des paramètres dont les valeurs sont définies avant le processus d'apprentissage et qui influent sur le comportement global du modèle pendant l'entraînement. Contrairement aux paramètres à l'intérieur des couches de neurones, qui sont appris à partir des données, les hyperparamètres ne sont pas ajustés pendant l'apprentissage. Voici une liste des principaux hyperparamètres :

- Le **nombre et le type de couches de neurones**, et le **nombre de neurones dans chaque couche**.
- La **taille des filtres** (pour les couches convolutionnelles).
- Les **fonctions d'activations** pour les couches intermédiaires.
- Le nombre d'**époques**.
- Le ***batch size***.
- L'**optimiseur**.
- Le ***dropout***, ou taux de drop. Dans le cas de surapprentissage (le modèle fonctionne bien sur les données d'entrainement mais sous-performe sur les données de validation), on peut imposer du *dropout*. Celui-ci permet d'éviter que le réseau ne devienne trop dépendant de certains neurones particuliers et favorise une meilleure généralisation. Si *dropout* $= 0.3$, alors à chaque itération de l'algorithme de *backpropagation*, on choisit au hasard 30% des paramètres du modèle et on les désactive (on leur affecte la valeur 0).

Après avoir trouvé les meilleurs hyperparamètres, on les utilise pour réaliser la méthode des K-folds.

3.4 Les auto-encodeurs

3.4.1 Structure d'un auto-encodeur

Un auto-encodeur est un réseau de neurones avec une structure particulière.

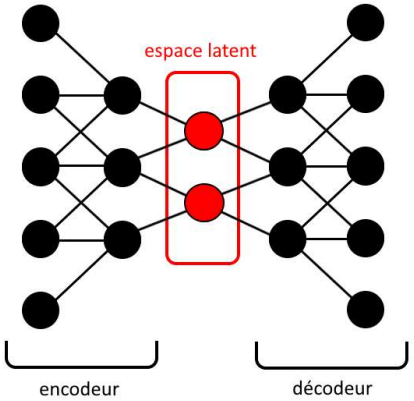


Figure 3.5 Structure d'un auto-encodeur

On peut diviser un auto-encodeur en trois parties :

- L'**encodeur** : l'idée est de compresser les données d'entrées. Pour cela, chaque couche de l'encodeur est plus petite que la précédente.
- L'**espace latent**. C'est la couche centrale du réseau, où les données d'entrées sont compressées. L'espace latent manipule ce que l'encodeur lui renvoie. La dimension de l'espace latent doit être inférieure à la taille des données X.
- Le **décodeur**. À partir des données compressées dans l'espace latent, le décodeur les décompresse pour retomber sur les données d'entrée X. Pour cela, nos données de sorties y sont en fait les données X.

Comme, dans ce cas, nous n'avons pas réellement de données cibles y (ce sont les données d'entrées X qui jouent ce rôle), on parle d'**apprentissage non supervisé**.
Pour évaluer le modèle, on utilise la *mean square error*. Si la *mean square error* est faible, cela signifie que le modèle est performant et que, par conséquent, la décompression des données n'engendrera pas une grande perte de qualité.

3.4.2 Génération de données

Si on fixe la dimension de l'espace latent à $d = \{1, 2, 3\}$, on peut représenter visuellement toutes les données compressées. On pourra s'apercevoir que généralement, les données compressées se situent dans une zone précise de $\mathbb{R}^d$. En effet, les coordonnées n'explosent pas à l'infini et semblent suivre une distribution particulière. Il en est de même pour $d > 3$. Pour générer de nouvelles données, on peut choisir un point aléatoire de $\mathbb{R}^d$, puis réaliser la phase de décompression. Ainsi, en sortie, on aura une nouvelle donnée générée par le modèle. Toutefois, si le point aléatoire de $\mathbb{R}^d$ est "loin" des points que le décodeur a l'habitude de manipuler, alors les données obtenues en sortie risquent de ne pas être cohérentes.

Pour régler ce problème, on a recours a deux solutions.

1. On impose une taille de l'espace latent de $d = \{1, 2, 3\}$. On peut alors visualiser l'espace latent, et voir les zones que le décodeur a l'habitude de traiter. Ensuite, on génère des points au centre de ces zones, et à la sortie du décodeur on obtient une nouvelle sortie mais similaire aux données connues.

2. On force l'espace latent à suivre une certaine distribution (normale, par exemple). Ainsi, en générant des points aléatoires qui suivent la distribution en question, le décodeur saura les traiter car il aura déjà manipulé des données similaires. Un tel auto-encodeur est appelé **auto-encodeur variationnel**.

4 Deep Learning – Sur Keras

Aujourd'hui, il existe principalement deux bibliothèques dans Python utilisées pour le Deep Learning, à savoir *Keras* et *Pytorch*. Nous allons ici apprendre à utiliser *Keras* pour être en mesure de créer différents réseaux de neurones.

Avant de créer un réseau de neurones, on doit réaliser plusieurs étapes.

1. Importer les fonctions de *Keras* avec la commande *import tensorflow.keras as kr*.
2. Avoir une base de données $(x^{(i)}, y^{(i)})_{1 \leq i \leq n}$. On appellera $X \in \mathcal{M}_{n,p}$ la matrice qui contient les données $x^{(i)}$ et $y \in \mathbb{R}^n$ le vecteur qui contient les données $y^{(i)}$. Pour un individu, on a donc p caractéristiques.
3. Séparer X en trois matrices X_{train}, $X_{validation}$, X_{test}. Faire de même pour y.

4.1 Régression

Voici un canevas de code Python pour construire un réseau de neurones qui résout des problèmes de régression.

```python
import numpy as np
import tensorflow.keras as kr
```

Les données

```python
p = 200

X_train, X_val, X_test = np.random.randn(600, p), np.random.randn(200, p), np.random.randn(200, p)
y_train, y_val, y_test = np.random.rand(600), np.random.rand(200), np.random.rand(200)
```

Création du modèle

```python
entree = kr.Input(shape=p) # l'entrée est de taille p
couche1 = kr.layers.Dense(100, activation="relu")(entree) # la couche 1 est de taille 100
couche2 = kr.layers.Dense(10, activation="relu")(couche1) # la couche 2 est de taille 10
#...
sortie = kr.layers.Dense(1, activation="relu")(couche2) # la sortie est de taille 1 (comme y)

modele = kr.models.Model(inputs=entree, outputs=sortie)
```

Compilation du modèle

```python
modele.compile(loss="mse", optimizer="adam", metrics="mse")
```

Phase d'apprentissage

```python
modele.fit(X_train, y_train, epochs=2, batch_size=50, validation_data=(X_val, y_val))
```

```
Epoch 1/2
12/12 [==============================] - 1s 34ms/step - loss: 0.2986 - val_loss: 0.2574
Epoch 2/2
12/12 [==============================] - 0s 6ms/step - loss: 0.2270 - val_loss: 0.2478
```

- On peut changer les fonctions d'activations, le nombre d'époques, le *batch_size* ou encore l'optimiseur comme on le souhaite.
- On peut ajouter ou supprimer des couches denses, et changer la taille de celles-ci.

- Cependant, la couche de sortie doit impérativement être de dimension **1**. Les valeurs y_{pred} seront donc de la même dimension que celles de y.
- La fonction coût doit être la ***mse***.
- L'ensemble d'arrivée de la fonction d'activation de la sortie doit correspondre à l'ensemble des valeurs que peut prendre y.

- Keras nous permet de voir l'erreur sur les données d'entrainement et de validation lors de chaque époque de la phase d'apprentissage.
- On peut utiliser la fonction *kr.utils.plot_model* pour obtenir une représentation visuelle de notre modèle.
- Pour prédire les sorties d'une ou plusieurs nouvelle(s) donnée(s), on peut utiliser la méthode *predict* de notre modèle.

4.2 Classification

Dans ce cas, la façon de coder est identique. Toutefois, il faudra faire quelques changements qui sont essentiels pour le bon fonctionnement du programme.

4.2.1 Classification binaire, $y \in \{0, 1\}$

Création du modèle

```
entree = kr.Input(shape=p)
couche1 = kr.layers.Dense(50, activation="selu")(entree) # la couche 1 est de taille 50
couche2 = kr.layers.Dense(8, activation="selu")(couche1) # la couche 2 est de taille 8
#...
sortie = kr.layers.Dense(1, activation="sigmoid")(couche2)

modele = kr.models.Model(inputs=entree, outputs=sortie)
```

Compilation du modèle

```
modele.compile(loss="binary_crossentropy", optimizer="adam", metrics="accuracy")
```

- La couche de sortie doit être de dimension 1.
- La fonction d'activation de la couche de sortie doit forcément être ***sigmoid***.
- La fonction d'erreur est la ***binary_crossentropy***.
- La métrique doit être ***accuracy***. Cette métrique compte le nombre de bonnes prédictions et le divise par le nombre total de prédictions.

Lorsqu'on prédit la valeur de sortie d'une nouvelle donnée, on obtient un nombre compris entre 0 et 1. Si ce nombre est inférieur à 0.5, on lui attribue la valeur 0. Sinon, on lui donne la valeur 1.

4.2.2 Classification multiple, $y \in [\![0, m]\!]$, $m > 1$

Les données

```
#...
y_train_hot = to_categorical(y_train)
y_val_hot = to_categorical(y_val)
y_test_hot = to_categorical(y_test)
```

Création du modèle

```
entree = kr.Input(shape=p)
couche1 = kr.layers.Dense(200, activation="selu")(entree) # la couche 1 est de taille 200
couche2 = kr.layers.Dense(23, activation="selu")(couche1) # la couche 2 est de taille 23
#...
sortie = kr.layers.Dense(m+1, activation="softmax")(couche2)

modele = kr.models.Model(inputs=entree, outputs=sortie)
```

Compilation du modèle

```
modele.compile(loss="categorical_crossentropy", optimizer="rmsprop", metrics="accuracy")
```

```
modele.fit(X_train, y_train_hot, epochs=50, batch_size=100, validation_data=(X_val, y_val_hot))
```

- La couche de sortie doit être de dimension $\boldsymbol{m+1}$.
- La fonction d'activation de la couche de sortie doit forcément être ***softmax***.
- La fonction d'erreur est la ***categorical_crossentropy***.
- La métrique doit être ***accuracy***.

Avant de créer le modèle, il faudra transformer les données y en un vecteur $y_{hot} \in \mathbb{R}^{m+1}$. Supposons que $y = i$, $i \in [\![0, m]\!]$. Alors chaque composante vaudra 0, sauf la $i^{\text{ème}}$ qui vaudra 1. Pour cela, on utilise la fonction *kr.utils.to_categorical*.

Par exemple, si $m = 3$ et $y = 2$, alors $y_{hot} = (0,0,1,0)^t$.

Lorsqu'on prédit la valeur de sortie d'une nouvelle donnée, on obtient un vecteur $y_{pred} \in \mathbb{R}^{m+1}$ telle que la somme des composantes vaut 1. Ainsi, la prédiction sera l'argument maximum de y_{pred} (en supposant que la première composante est indicée par 0).

Par exemple, si $y_{pred} = (0.1, 0.1, 0.5, 0.3)^t$, alors la prédiction sera 2 car 0.5 est le plus grand des $m + 1$ nombres.

4.3 Couches convolutionnelles

Lors de la création du modèle, on peut ajouter des couches convolutionnelles. Celles-ci sont énormément utilisées lorsqu'on traite des images. Supposons que nos données d'entrée sont des images de dimension (p_1, p_2).

```
X_train = X_train.reshape((p1, p2, 1))
X_val = X_val.reshape((p1, p2, 1))
X_test = X_test.reshape((p1, p2, 1))

entree = kr.Input(shape=(p1, p2, 1))
couche1 = kr.layers.Conv2D(32, (3, 3), strides=1, activation="relu")(entree)
couche2 = kr.layers.MaxPooling2D((2, 2))(couche1)
couche3 = kr.layers.Flatten()(couche2)
#...
sortie = kr.layers.Dense(40)(couche3) # la sortie est de taille 40
#...
```

- Dans un premier temps, les données doivent être redimensionnées $(\boldsymbol{p_1}, \boldsymbol{p_2}, \mathbf{1})$.
- La couche convolutionnelle est la fonction ***Conv2D***. Le premier argument désigne le **nombre de filtres** (ici 32) et le deuxième argument fait référence à la **taille des filtres** (dans ce cas 3×3). L'argument *strides* correspond au **décalage** des filtres.

Ainsi, la dimension des données obtenues à la sortie de *couche1* sera de taille $(\boldsymbol{p_1 - 2}, \boldsymbol{p_2 - 2}, \mathbf{32})$**.** On doit enlever 2 sur les deux premières composantes car chaque filtre va se déplacer $(p_1 - 2) \times (p_2 - 2)$ fois pour parcourir l'ensemble des données.
Plus généralement, avec q filtres de taille $m \times m$, on obtient en sortie des données de taille $(\boldsymbol{p_1 - m + 1}, \boldsymbol{p_2} - \mathbf{m} + \mathbf{1}, \boldsymbol{q})$**.**

- Une couche convolutionnelle (*Conv2D*) est toujours suivie d'une couche telle que ***MaxPooling2D*** ou ***AveragePooling2D***. Ces fonctions permettent de rétrécir la taille les données tout en préservant un maximum d'information. Ces fonctions prennent en argument un tuple de deux valeurs (a, b). Ainsi, la fonction réduit d'un facteur a (resp. b) la première (resp. la deuxième) dimension des données. Voici une image qui illustre le fonctionnement de *MaxPooling2D*.

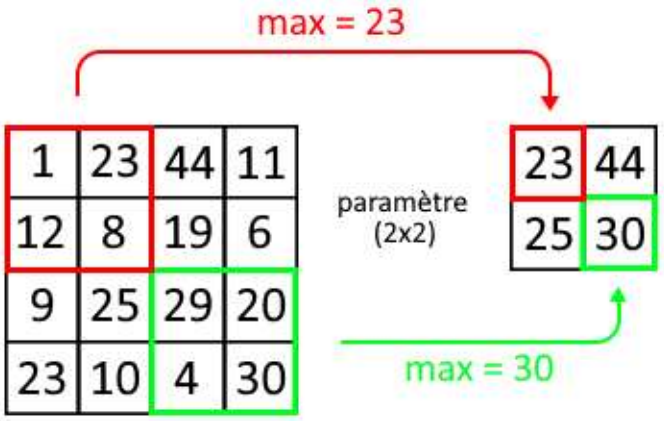


Figure 4.1 Fonctionnement de la fonction MaxPooling2D

- La fonction ***Flatten*** permet de redimensionner les données dans un vecteur. Ainsi, on peut utiliser des couches denses à la suite.

4.4 Auto-encodeurs variationnels

Sur *Keras*, les auto-encodeurs sont plus complexes à coder. De plus, on peut les créer de plusieurs façons différentes. Ici, nous allons voir dans cette partie les auto-encodeurs variationnels avec la **fonction coût** KL_{loss}.

```
def sampling(moyenne_logVariance): # moyenne_logVariance contient deux éléments

    moyenne, logVariance = moyenne_logVariance
    return moyenne + tf.random.normal(tf.shape(moyenne)) * tf.math.exp(logVariance/2)
```

Création du modèle encodeur

```
dimEspaceLatent = 3

entree_encodeur = kr.Input(shape=(p1, p2, 1))
# couche1, ..., couche i
sortie_moyenne = kr.layers.Dense(dimEspaceLatent)(couchei)
sortie_logVariance = kr.layers.Dense(dimEspaceLatent)(couchei)
sortie_Z = kr.layers.Lambda(sampling)([sortie_moyenne, sortie_logVariance])
modele_encodeur = kr.models.Model(inputs=entree_encodeur, outputs=[sortie_moyenne, sortie_logVariance, sortie_Z])
```

Création du modèle décodeur

```
entree_decodeur = kr.Input(shape=dimEspaceLatent)
#couche1, ..., sortie_decodeur
modele_decodeur = kr.models.Model(inputs=entree_decodeur, outputs=sortie_decodeur)
```

Création de l'auto-encodeur

```
entree_autoencodeur = kr.Input(shape=(p1, p2, 1))
espace_latent = modele_encodeur(entree_autoencodeur)
sortie_autoencodeur = modele_decodeur(espace_latent[2]) # on part de la couche Z

modele_autoencodeur = kr.models.Model(inputs=entree_autoencodeur, outputs=sortie_autoencodeur)
```

Pour ce faire, on doit créer trois modèles (encodeur, décodeur et auto-encodeur).

- L'**encodeur** prend en entrée les données initiales. L'avant dernière couche du modèle est particulière puisqu'elle est composée de deux parties. La première contient des éléments $\hat{\mu}$ qui vont se rapprocher de $\mu = 0$. La seconde possède des éléments $\widehat{log(\sigma^2)}$ qui doivent converger vers $log(\sigma^2) = 0 \ (\Leftrightarrow \sigma^2 = 1)$. À partir de ces deux parties, on crée notre **espace latent** *sortie_Z* dont sa distribution approchera une loi normale centrée réduite $\mathcal{N}(\mu = 0, \sigma^2 = 1)$.

- Le **décodeur** prend en entrée l'espace latent *sortie_Z* et renvoie en sortie des données prédites qui doivent être de taille $(\boldsymbol{p_1}, \boldsymbol{p_2}, \mathbf{1})$. On utilisera des couches ***Conv2DTranspose*** et ***UpSampling2D*** dans le décodeur. Pour plus d'informations à propos de ces couches, se référer à la documentation de *Keras*.
  Conv2DTranspose layer (keras.io)
  UpSampling2D layer (keras.io)

- L'**auto-encodeur** prend la même entrée que l'encodeur et la même sortie que le décodeur.

Ensuite, on doit définir la **fonction coût** $\boldsymbol{KL_{loss}}$. Celle-ci ressemble à la *MSE*, mais avec des termes supplémentaires. Elle permet d'obtenir des prédictions « proches » des données initiales tout en obtenant un espace latent qui suit plus ou moins une distribution normale.

Fonction coût

```
lambda_ = 100

MSE = kr.losses.mse(entree_autoencodeur, sortie_autoencodeur)

KL = tf.reduce_mean(tf.math.square(espace_latent[0])) # moyenne au carré
KL += tf.reduce_mean(tf.math.exp(espace_latent[1])) # variance
KL -= tf.reduce_mean(espace_latent[1]) # log variance
KL -= 1
KL = KL / 2

fonction_cout = lambda_ * MSE + KL

modele_autoencodeur.add_loss(fonction_cout)
```

Phase d'entrainement

```
modele_autoencodeur.fit(X_train, X_train, epochs=20, batch_size=100, validation_data=(X_val, X_val))
```

$$\boldsymbol{KL_{loss}} = \boldsymbol{\lambda} \times \boldsymbol{MSE} + \boldsymbol{\mathcal{D}_{KL}} \qquad (\lambda \in \mathbb{R}^+)$$

$$\mathcal{D}_{KL} = \frac{1}{n}\sum_{j=1}^{n}\frac{1}{2}\sum_{i=1}^{d}\left(\hat{\mu}_i\hat{\mu}_i + \widehat{\sigma_i^2} - \widehat{log(\sigma_i^2)} - 1\right)$$

- Le terme $\mathcal{D}_{KL}$ est appelé ***divergence de Kullback-Leibler***. Il permet de forcer les termes $\hat{\mu}_i$ (resp. $\widehat{\sigma_i^2}$) à valoir 0 (resp. 1).
- La valeur $\boldsymbol{n}$ correspond au nombre de données d'entrainement. La valeur $\boldsymbol{d}$ est la dimension de l'espace latent. Les $(\hat{\boldsymbol{\mu}})_{\mathbf{1}\leq \boldsymbol{j}\leq \boldsymbol{n}}$ et les $(\widehat{\boldsymbol{\sigma^2}})_{\mathbf{1}\leq \boldsymbol{j}\leq \boldsymbol{n}}$ sont les éléments de l'avant dernière couche de l'encodeur qui permettent de concevoir l'espace latent.

- Plus on diminue la valeur de $\boldsymbol{\lambda}$, plus on force l'espace latent à suivre une loi normale ($\mathcal{D}_{KL}$ diminue). Toutefois, ceci impliquera une augmentation de la *MSE*. Le réel positif λ est donc un hyperparamètre qu'on peut traiter dans une *grid search*.

Remarque

Il est possible d'utiliser d'autres fonctions coût. Par exemple, une fonction coût naturelle est $f_{cout} = MSE + \frac{1}{n}\sum_n \|\hat{\mu}\| + \frac{1}{n}\sum_n \left\|\widehat{\sigma^2} - 1\right\|$. Ainsi, en minimisant cette fonction, $\hat{\mu}$ se rapprochera de 0 et $\widehat{\sigma^2}$ de 1. La KL_{loss} présente l'avantage d'avoir une justification statistique. En effet, le terme $\mathcal{D}_{KL}$ mesure la divergence théorique entre deux distributions de probabilité (dans ce cas la distribution des $\left(\hat{\mu}, \widehat{\sigma^2}\right)_{1 \leq j \leq n}$ et une loi normale centrée réduite).

5 Classification de sons

Maintenant que nous connaissons les bases du traitement de sons et du Deep Learning, nous pouvons nous lancer dans la création de réseaux de neurones manipulant des documents audios. Tous nos modèles sont créés sur Python, à l'aide de la bibliothèque *Keras*. Ce qui nous a semblé le plus pertinent et accessible à réaliser dans un premier temps, c'est de la classification.

Nous utilisons ici des données audios publiques provenant du site *Kaggle*. *Kaggle* est un site Internet qui contient des bases de données de tout genre et facilement utilisables en Deep Learning. Nous ne nous attarderons pas sur le chargement des données car de nombreux codes Python se trouvent sur le site.

Nous appliquons la même procédure pour chaque modèle pour chacune de nos problématiques de classification.

1. **Mélange des données**.
2. **Normalisation des données** avec la moyenne et l'écart type.
3. **Séparation des données** : 60% entrainement, 20% validation, 20% test.
4. Réalisation d'une ***grid search*** pour trouver de bons paramètres.
5. Méthode des K**-folds** (on a toujours pris $K = 5$).
6. Application de notre modèle sur des **données test**.

Nous préciserons à chaque fois :

- Le traitement effectué sur les données (spectrogrammes et/ou MFCC).
- La taille de notre matrice passée en entrée du réseau qu'on appelle X.
- La structure et les hyperparamètres principaux du modèle créé.
- Le résultat final du modèle (sur les données test) et des commentaires.

5.1 MNIST

Pour un premier réseau de neurones, on souhaite classifier des chiffres, entre 0 et 9, prononcés en anglais.

Données et traitements

- Nous utilisons la base de données « Audio MNIST » : Audio MNIST (kaggle.com)
- Il s'agit de 60 personnes qui disent un chiffre entre 0 et 9. Chaque personne prononce en fait chaque chiffre plusieurs fois, ce qui nous fait 30 000 données au total.
- Nous transformons chaque son en spectrogramme de taille (50×50).
- Ceci nous donne X de taille $(30\,000,\ 2500) = 75$ millions.

- Pour la séparation des données, nous avons fait en sorte de ne pas avoir une personne dans deux groupes différents. Ainsi, on pourra tester notre modèle sur de nouvelles voix.

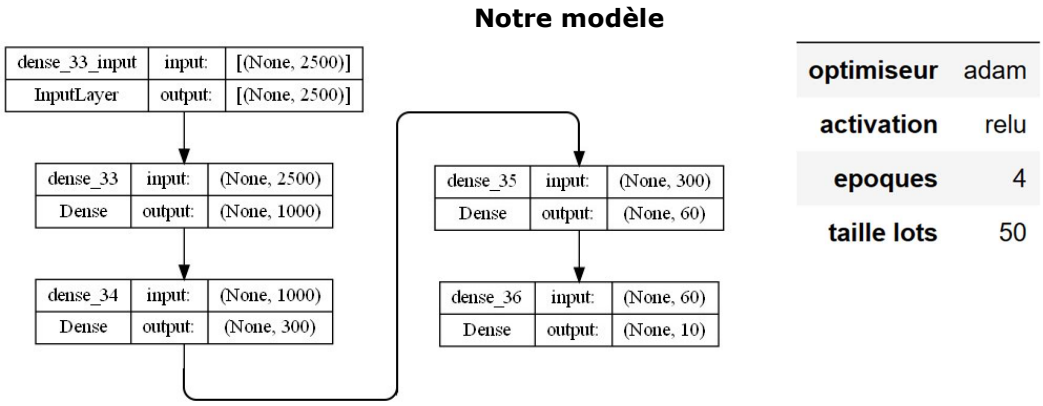


Figure 5.1 Description du modèle utilisé pour la classification MNIST

Résultat et commentaires

- Nous obtenons finalement un pourcentage de précision de **98 %**, ce qui est très convaincant.
- Nous nous attendions toutefois à un tel résultat car on peut déjà apercevoir à l'œil nu des motifs similaires pour chaque chiffre. Voici par exemple les spectrogrammes de cinq personnes de la base de données qui prononcent les cinq premiers chiffres.

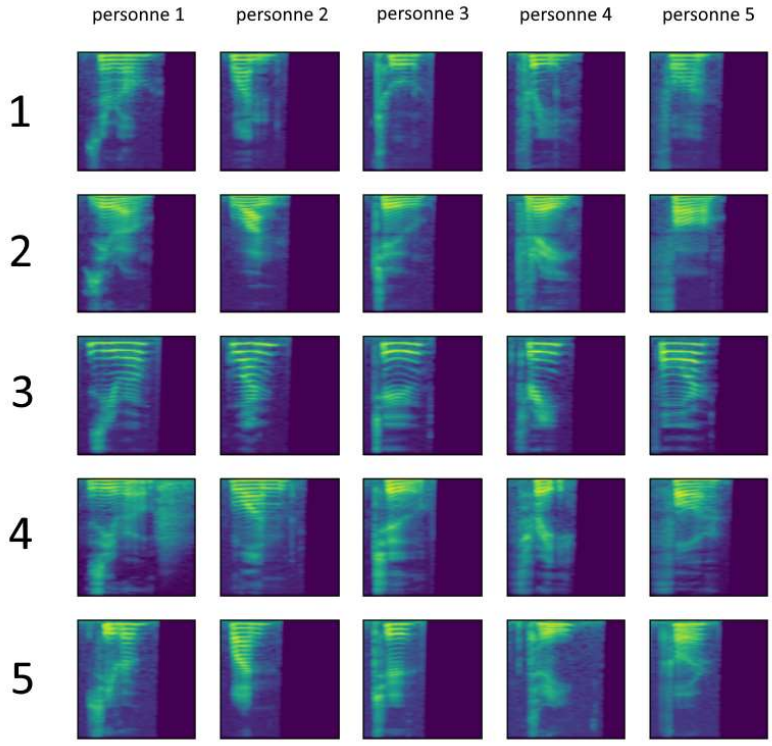


Figure 5.2 Quelques spectrogrammes obtenus après traitement audio

- Nous n'avons pas eu besoin d'une grande *grid search* pour atteindre une telle précision. Il est donc certainement possible de trouver un modèle encore plus performant.
- Cependant, le modèle ne fonctionne pas avec nos données personnelles. En effet, après l'enregistrement de nos voix, le modèle ne prédisait pas les bonnes valeurs. Nous avons testé une normalisation entre 0 et 1 et différentes méthodes d'enregistrement, sans succès. Nous pensons que le problème vient de l'enregistrement, car nos spectrogrammes obtenus ne ressemblent pas vraiment aux autres.

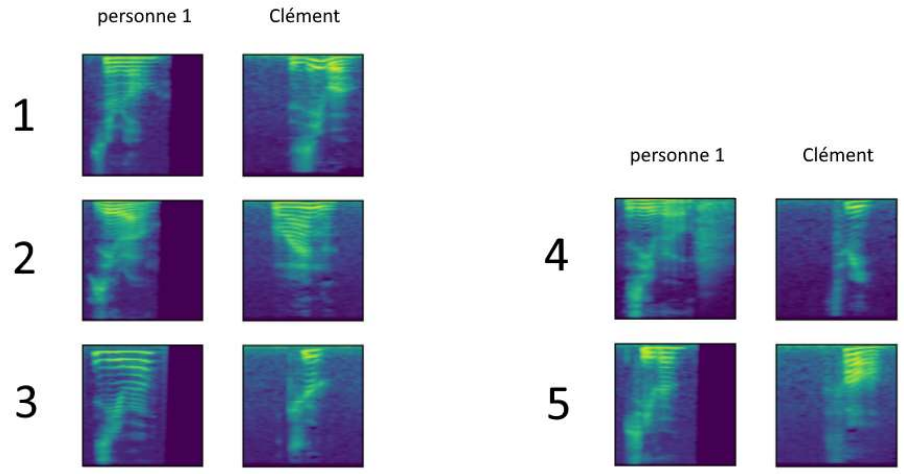


Figure 5.3 Spectrogrammes d'une personne de la base de données et de Clément

5.2 Classification par genre

On reste ici sur de la **classification binaire**. Plus intéressant, on cherche à prédire le genre d'une personne qui parle.

Données et traitements

- Nous utilisons la base de données « Common Voice » : Common Voice (kaggle.com)
- Cette base de données comporte de nombreux sons. Ce sont des personnes qui parlent, et généralement il est indiqué le genre et l'âge de l'individu.
- Nous transformons chaque son en spectrogramme de taille (50×50).
- Ceci nous donne X de taille $(27284, 2500) \approx 68$ millions.
- On a autant de femmes que d'hommes, soit $\frac{27284}{2} = 13642$.

Notre modèle

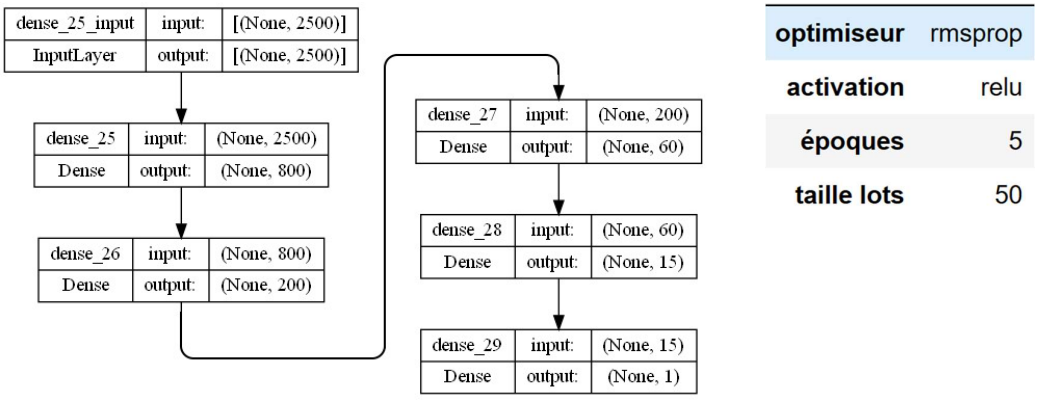


Figure 5.4 Description du modèle utilisé pour la classification du genre

Résultat et commentaires

- Nous obtenons un pourcentage de précision de **93 %**.
- Nous avons toujours des modèles très performants bien que nous n'utilisons toujours pas des couches convolutives.
- Nous avons testé notre modèle sur des données personnelles. Pour ce faire, nous avons enregistré la voix d'une dizaine de personnes (hommes et femmes). Cette fois-ci, les résultats étaient cohérents puisque tous les individus ont bien été classés.

5.3 Classification par âge

Nous nous attaquons maintenant à un problème de classification multiple. Nous allons essayer de prédire dans quelle tranche d'âge se situe une personne à partir de sa voix. Nous listons ci-dessous les tranches d'âge et le nombre d'individu dans chaque classe.

- 10 à 19 ans : 3577 individus.
- 20 à 29 ans : 5000 individus.
- 30 à 39 ans : 5000 individus.
- 40 à 49 ans : 5000 individus.
- 50 à 59 ans : 5000 individus.
- 60 à 69 ans : 3829 individus.
- 70 à 89 ans : 1549 individus.

Pour ce problème, nous avons choisi de comparer l'efficacité des spectrogrammes et des MFCC. Ensuite, nous avons combiné les deux pour voir si on augmente les performances de notre modèle.

5.3.1 Avec les spectrogrammes

Données et traitements

- Nous utilisons la base de données « Common Voice » : [Common Voice (kaggle.com)]
- Nous transformons chaque son en spectrogramme de taille (50×50).
- Ceci nous donne X de taille $(28955, 2500) \approx 72$ millions.

Notre modèle

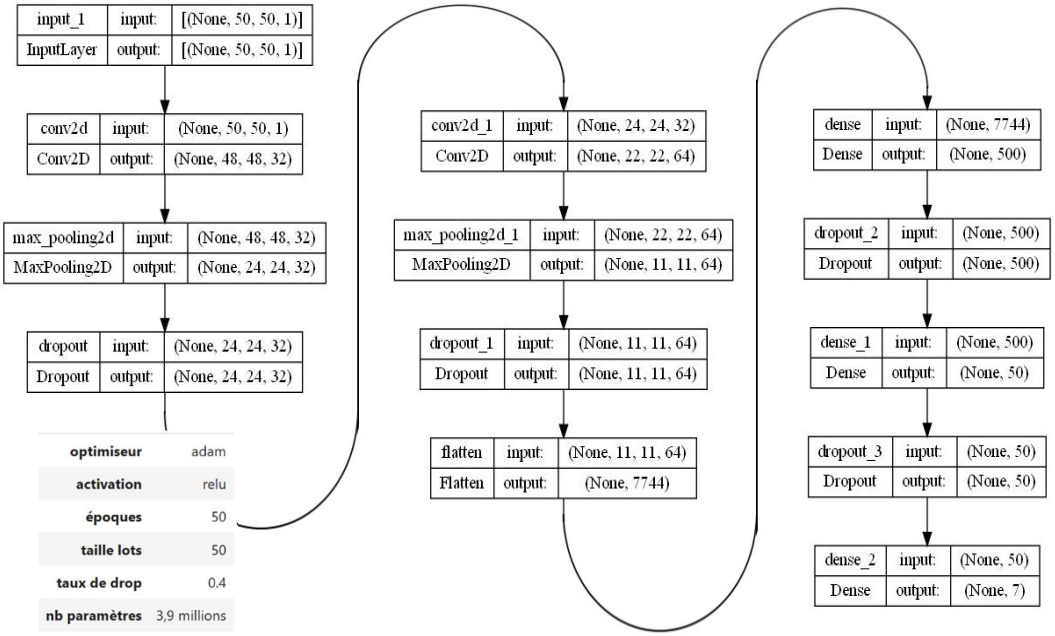


Figure 5.5 Description du modèle utilisé pour la classification de l'âge

Résultat et commentaires

- Nous obtenons un pourcentage de précision de **74 %**.
- Nous avons dû intégrer deux couches convolutives à notre modèle et un taux de drop (avec la fonction *kr.layers.Dropout*) pour augmenter les performances du modèle. En effet, sans *dropout*, le modèle réalisait du sur-apprentissage : l'erreur sur les données d'entrainement converge vers 0 mais l'erreur sur les données de validation reste très élevée.
- Nous trouvons que le résultat est satisfaisant. En effet, même avec un excellent modèle, nous pensons que nous aurons du mal à faire significativement mieux. Dans nos données, il y a probablement des personnes qui se situent à la limite entre deux classes. Le modèle peut donc mal classer un individu en se trompant de seulement une année par exemple. On va donc regarder les erreurs que notre modèle commet, en s'intéressant à la matrice de confusion associée à nos données test.

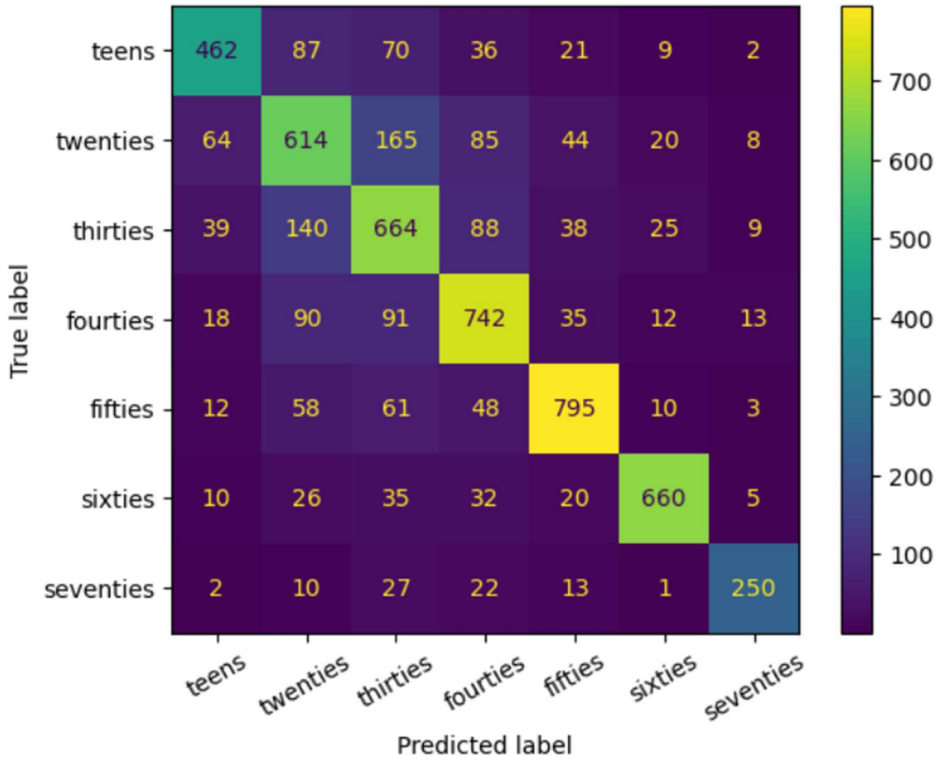


Figure 5.6 Matrice de confusion du modèle sur les données test

- À première vue, on remarque que quand notre modèle se trompe, il se trompe d'une seule tranche d'âge. En effet, plus on s'éloigne de la diagonale, plus les chiffres sont bas. La majorité des erreurs se font dans les tranches d'âge 20/30 et 30/40 ans, ce qui nous parait plutôt logique.

	1	2	3	4	5	6
%	49.6	27.7	13.0	6.8	2.5	0.5

Table 5.1 Répartition de la distance des erreurs parmi toutes les mauvaises prédictions

- Le tableau ci-dessus montre que lorsque le modèle se trompe, il prédit une fois sur deux une classe voisine. Tous ces résultats nous amènent à penser que le modèle fonctionne globalement très bien.
- Nos résultats se confirment lorsqu'on teste le modèle avec des données personnelles. En enregistrant les voix de plusieurs personnes de notre entourage, on constate soit des bonnes prédictions, soit des erreurs d'une seule classe.

5.3.2 Avec les MFCC

Données et traitements

- Nous utilisons la base de données « Common Voice » : Common Voice (kaggle.com)

- Nous transformons chaque son en matrice de MFCC de taille (13×192). Nous avons choisi ces paramètres de sorte qu'on ait le même nombre de données qu'avec les spectrogrammes ($50 \times 50 \approx 13 \times 192$).
- Ceci nous donne X de taille $(28955, 2496) \approx$ 72 millions.

Résultat et commentaires

- En travaillant sur notre *grid search*, nous ne sommes pas parvenus à trouver un pourcentage de précision supérieur à **50%** sur des données de validation.
- **Dans ce contexte**, les MFCC semblent donc moins performants que les spectrogrammes.

5.3.3 Avec les spectrogrammes et les MFCC

- Nous avons créé ici un modèle qui prend en entrée à la fois les données de la partie 5.3.1 et de la partie 5.3.2.
- Nous nous attendions à ce que les MFCC apportent des informations supplémentaires, et donc qu'on obtiendra une précision supérieure à 74 %. Toutefois, nos résultats sur notre *grid search* ne dépassent pas **64 %**.
- Nous en concluons que dans cet exercice, les MFCC ne sont pas utiles, même si on les utilise avec ou sans les spectrogrammes.

5.3.4 Avec des modèles pré-entraînés

Finalement, nous avons utilisé des modèles pré-entrainés de Keras, en s'attendant à de meilleurs résultats. Pour cela, nous avons testé plusieurs modèles, tels que **DenseNet201**, **ResNet50** et **EfficientNetV2M**. Les caractéristiques des modèles sont disponibles sur le site suivant : Keras Applications

- On représente nos données en spectrogrammes de taille (50×50), soit les mêmes que dans la partie 5.3.1.
- Pour chaque modèle, on ajoute quelques couches denses à la fin pour avoir en sortie une taille de 7, et pour spécialiser le modèle à nos données.
- Le modèle va seulement apprendre sur les paramètres de nos couches denses, et non sur le modèle en entier.
- On réalise une *grid search* pour tester différents hyperparamètres.

Toutefois, nous n'avons pas obtenu des résultats convaincants. Au mieux, nos modèles avaient une précision de 45 % sur nos données de validation.
Notre meilleur modèle est donc clairement celui présenté dans la partie 5.3.1.

6 Génération de sons

Après nous être confrontés à des problèmes de classification, nous souhaitions finalement nous lancer dans la génération de nouvelles données audio. L'objectif de cette partie est de créer des réseaux de neurones capables de générer des sons compréhensibles par un humain, et par la même occasion de découvrir les auto-encodeurs variationnels.

6.1 Données et fonction utilisées

Données

Nous utilisons ici toutes la base de données **MNIST**. Nous transformons toutes les données en **spectrogrammes de taille** $(\mathbf{50} \times \mathbf{50})$. Ensuite, nous **normalisons les données entre 0 et 1**. Nous avons aussi testé la normalisation avec la moyenne et l'écart-type, mais les modèles étaient beaucoup moins performants dans ce cas. Les données sont séparées en trois groupes : entrainement, validation et test.

Fonction *feature.inverse.mel_to_audio*

- Pour générer des sons, l'idée est de générer des spectrogrammes qui ressemblent aux données connues. Nous devrons donc être en mesure de passer d'un spectrogramme à un son. Pour cela, nous avons recours à la fonction ***feature.inverse.mel_to_audio*** de *librosa*. Cette fonction prend notamment en argument le spectrogramme qu'on veut transformer en signal audio, mais aussi *hop_length*. La valeur de ce paramètre doit être la même que le *hop_length* utilisé dans la fonction *feature.melspectrogram.*

- La fonction *feature.inverse.mel_to_audio* applique l'**algorithme de Griffin-Lim**, qui donne un signal audio plus ou moins proche du signal de départ. En effet, le résultat n'est pas exact puisqu'on rappelle qu'un spectrogramme contient les carrés des modules de nombres complexes. Par conséquent, il est impossible de retrouver les valeurs des parties réelles et imaginaires de chaque complexe. L'algorithme de **Griffin-Lim** estime ces valeurs en s'appuyant sur des corrélations entre les données (voir sources).

```
import librosa as lb

S = lb.db_to_power(S) # on passe de l'échelle Db aux carrés des modules des nombres complexes
signalAudio = lb.feature.inverse.mel_to_audio(S, hop_length=hopLength)

Audio(data=signalAudio, rate=sr) # pour écouter le son directement
```

- Lorsque nous appliquons cette fonction sur un spectrogramme de taille (50×50), le son obtenu est reconnaissable, mais on constate une perte de qualité conséquente. La qualité est nettement supérieure avec des spectrogrammes de taille (200×30). On en conclut que plus le paramètre *n_mels* augmente, meilleure est la qualité sonore. Nous avons toutefois été contraint de garder des spectrogrammes de taille (50×50) car sinon le nombre de données serait trop important par rapport à notre matériel informatique.

- Nous avons tenté de créer un modèle de Deep Learning capable de supprimer le bruit créé par la fonction *feature.inverse.mel_to_audio*. Si un tel modèle fonctionnait, on pourrait lui faire rentrer nos signaux sonores prédits et on aurait en sortie un signal sans dégradation. Le modèle parvenait à obtenir de bonnes performances sur les données d'entrainement mais il ne savait pas traiter des données inconnues. Nous avons donc abandonné cette problématique qui nous semblait potentiellement très complexe à résoudre. Selon nous, le meilleur moyen d'éviter les pertes de qualité est d'augmenter la valeur du paramètre *n_mels*.

6.2 Auto-encodeurs « simples »

6.2.1 Performances des modèles

Dans un premier temps, nous avons créé plusieurs auto-encodeurs non variationnels. Chaque modèle prend en entrée les spectrogrammes et en sortie la même chose. On doit donc utiliser la *mean square error* (MSE) pour mesurer les performances des modèles. Nous utilisons nos données d'entrainement et de validation pour réaliser une *grid search.* Nous remarquons que :

- Les auto-encodeurs composés de couches convolutionnelles sont plus performants, mais les modèles qui n'ont que des couches denses fonctionnent très bien aussi.
- Même avec un espace latent de dimension 3, la MSE des modèles reste très faible sur nos données test.

Nous avons donc retenu trois modèles.

- Le premier possède des **couches convolutives** dans l'encodeur. L'espace latent est de taille **3** [modèle n°1]. La *MSE* sur les données test est de **0,0059**.

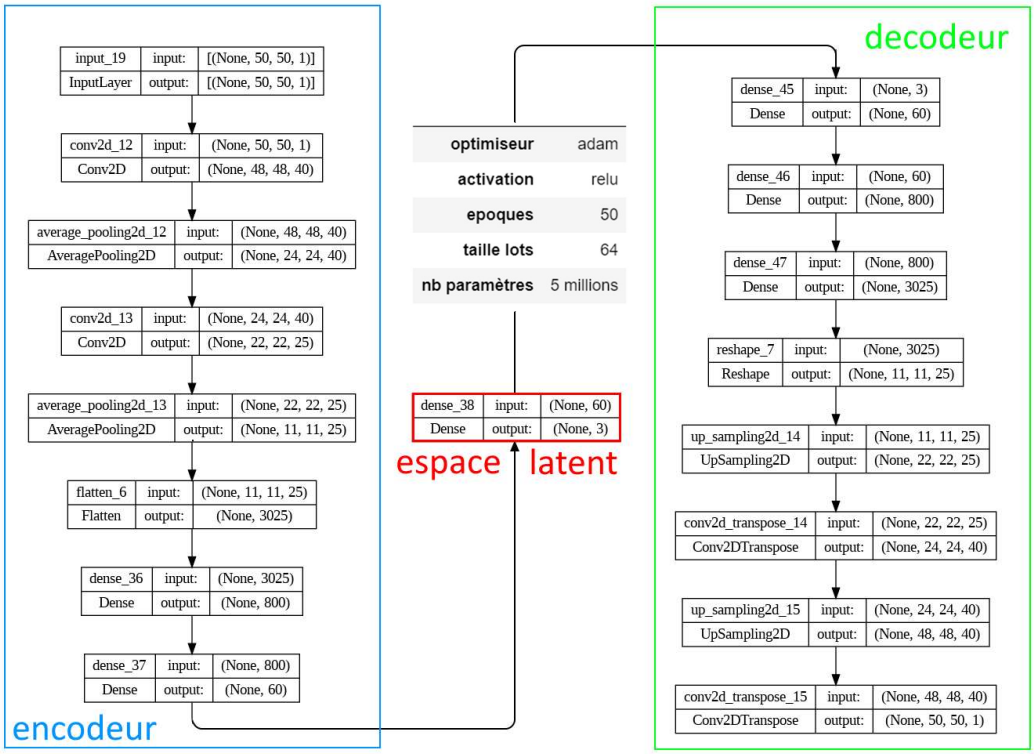


Figure 6.1 Description du modèle n°1

- Le second auto-encodeur contient des **couches convolutives** et l'espace latent est de dimension **10** [modèle n°2]. La *MSE* vaut **0,0038**.
- Le troisième n'a que des **couches denses** et son espace latent est de taille **3** [modèle n°3]. Pour ce modèle, on obtient une *MSE* de **0.0060** sur les données test.

6.2.2 Qualité des sons après passage dans les auto-encodeurs

Pour chacun de nos trois modèles, on s'attend à ce que la prédiction soit très proche des données d'entrées puisque la MSE est très faible. Mais la prédiction est un spectrogramme. Les spectrogrammes prédits seront très semblables aux spectrogrammes donnés en entrée. L'objectif étant d'obtenir des signaux sonores similaires, nous réalisons le test suivant.

1. Tirage aléatoire d'un spectrogramme S dans nos données test.
2. Passage du spectrogramme S dans les auto-encodeurs, qui renvoie un spectrogramme $\hat{S}$.
3. Application des fonctions *db_to_power* et *feature.inverse.mel_to_audio* sur le spectrogramme S et le spectrogramme prédit $\hat{S}$.
4. Écoute et comparaisons des sons obtenus.
5. Répétitions des étapes 1 à 4 pour confirmer les résultats.

Nous rappelons que le son obtenu à partir S ne sera pas parfait, puisque la fonction *feature.inverse.mel_to_audio* engendre une perte de qualité. Nous espérons donc que les deux sons seront identiques à l'oreille, et donc que le son obtenu à partir de $\hat{S}$ ne soit pas encore plus dégradé.

Résultats

Quelque soit le modèle utilisé, les deux sons semblent identiques : nous ne parvenons pas à identifier de différence à l'oreille. Nous pensons donc que la génération de sons pourrait fonctionner.

6.2.3 Génération de données

À présent, nous allons utiliser nos trois modèles et générer des points dans l'espace latent. Ensuite, nous ferons passer ces points dans le décodeur pour obtenir un nouveau spectrogramme. Finalement, on transforme le spectrogramme en signal sonore.

Génération de points aléatoires

On note d la dimension de l'espace latent.
Dans ce contexte, la fonction d'activation de la couche précédant l'espace latent est la fonction *sigmoïde*. Ainsi, tous les points de l'espace latent formés par les données d'entrainement sont dans $[0,1]^d$.
Nous générons donc des points dans $[0,1]^d$. Pour les modèles n°1 et n°3 ($d = 3$), nous obtenons souvent des sons inaudibles, et quelque fois nous reconnaissons un chiffre. Pour le modèle n°2 ($d = 10$), tous les sons sont quasiment incompréhensibles.

Ces résultats nous semblent cohérents.

- Le fait de générer un point de façon totalement aléatoire ne nous assure pas de tomber sur un point que le décodeur a l'habitude de traiter.
- Plus d augmente, plus on a de chance de générer un point dans le « vide ».

Génération avec les centroïdes

Pour générer des points dans l'espace latent de façon astucieuse, une idée naturelle serait de générer des points proches des centroïdes de chaque classe (une classe correspondant à un chiffre). Avant de faire cela, il est intéressant de visualiser l'espace latent du modèle n°1, puisque c'est le modèle le plus performant avec $d = 3$.

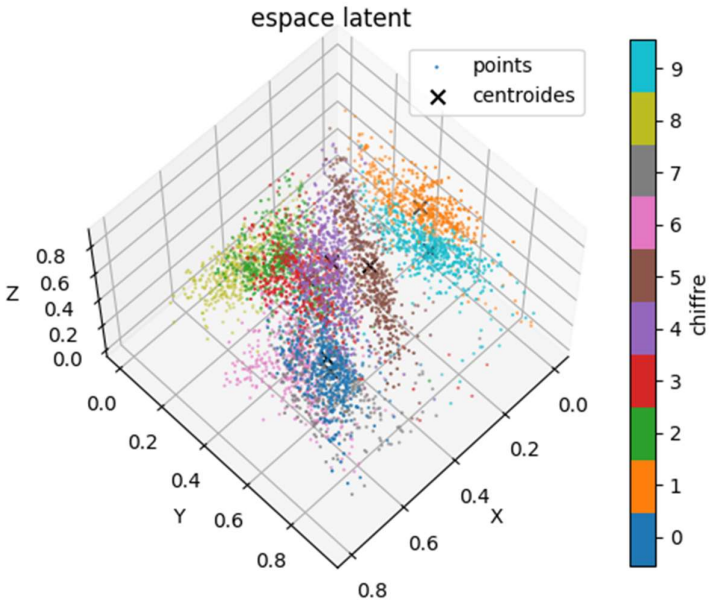


Figure 6.2 Représentation de l'espace latent du modèle n°1 en trois dimensions

Il semble que les points de chaque classe ne sont pas distribués de façon aléatoire. Pour plus de visibilité, nous proposons une autre représentation de l'espace latent, avec les centroïdes.

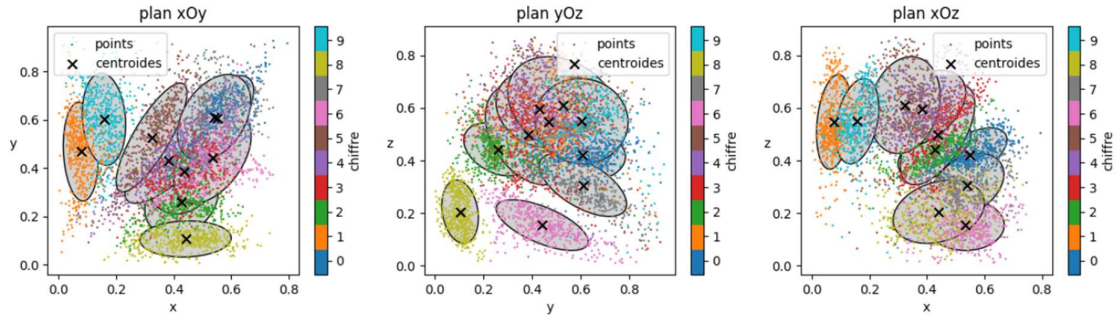


Figure 6.3 Projection des points de l'espace latent du modèle n°1 sur les plans canoniques

Cette représentation confirme la remarque ci-dessus. De plus, on voit que les nuages de points de certaines classes sont plus isolés que d'autres, comme le chiffre 1 (orange), 8 (jaune) et 9 (bleu clair).

Maintenant, nous prenons comme points de l'espace les dix centroïdes.

- Le modèles n°1 et n°2 donnent pour tous les chiffres un son compréhensible et correspondant au bon chiffre.
- Pour le modèle n°3, nous arrivons à reconnaitre 7 chiffres sur 10.

On observe aussi que plus on s'écarte des centroïdes, moins le son est compréhensible (ou se rapproche d'un autre chiffre).

6.3 Auto-encodeurs variationnels

La génération de données à partir des centroïdes est très convaincante, mais elle présente un inconvénient : pour un chiffre donné, nous sommes capables de générer seulement une seule « voix » qui prononce ce chiffre. Même si on choisit un point différent mais proche d'un centroïde, le résultat sera quasiment identique.
Les auto-encodeurs variationnels constituent une solution à ce problème. Nous présentons donc par la suite l'auto-encodeur variationnel que nous avons créé, ainsi que les résultats obtenus.

6.3.1 Description du modèle

Nous avons utilisé la fonction coût $\boldsymbol{KL_{loss}}$ pour notre modèle. L'espace latent est de dimension $\boldsymbol{d = 10}$.

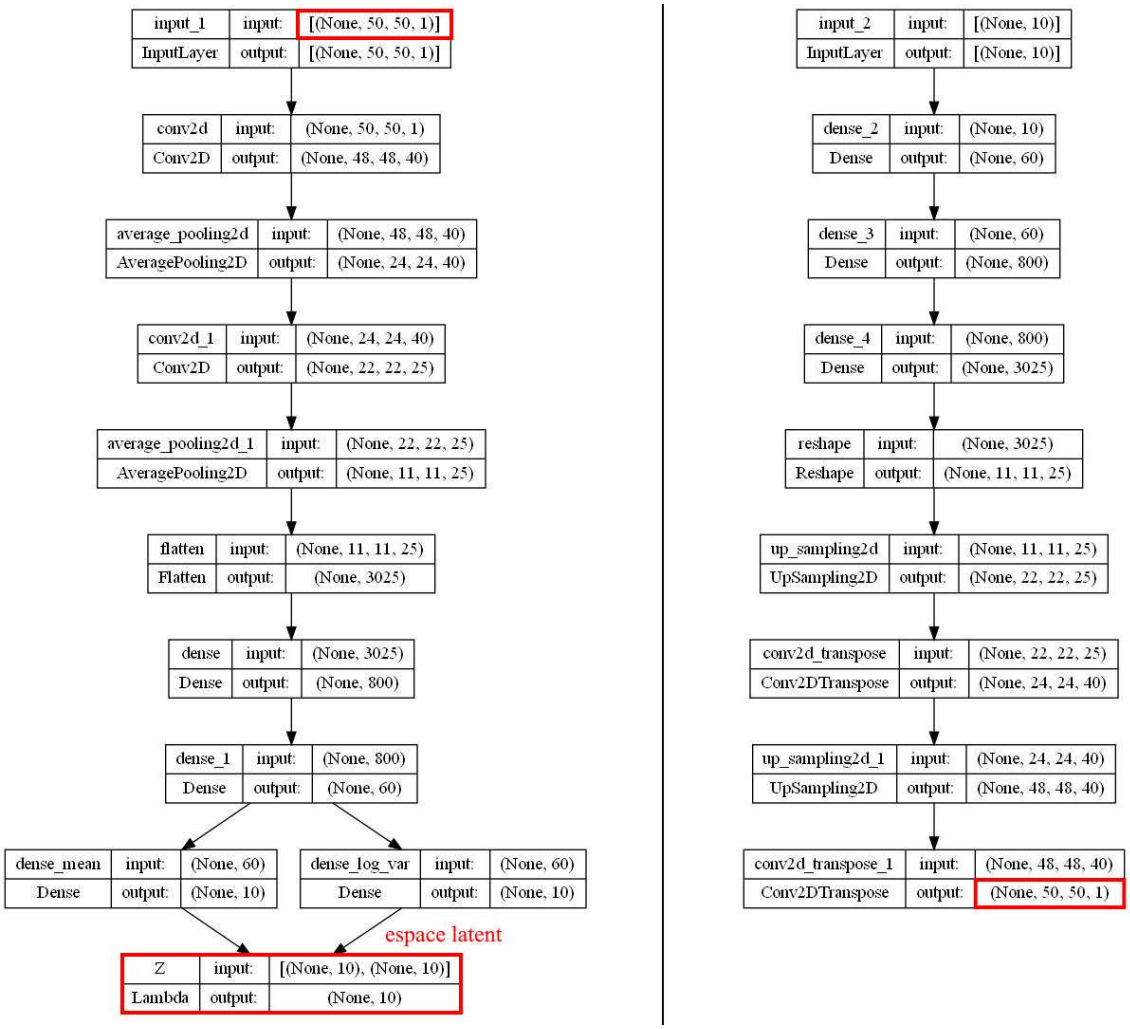


Figure 6.4 Description de l'encodeur (à gauche) et du décodeur (à droite)

Notre auto-encodeur est donc la concaténation de l'encodeur et du décodeur. Nous avons utilisé les hyperparamètres suivants.

optimiseur	adam
activation	relu
epoques	50
taille lots	64
λ	500
nb paramètres	5 millions

6.3.2 Erreur et distribution de l'espace latent

Erreur

Nous rappelons que nous utilisons la fonction coût $\boldsymbol{KL_{loss}}$. La *MSE* obtenue sur nos données test est **0.0054**.

Distribution de l'espace latent

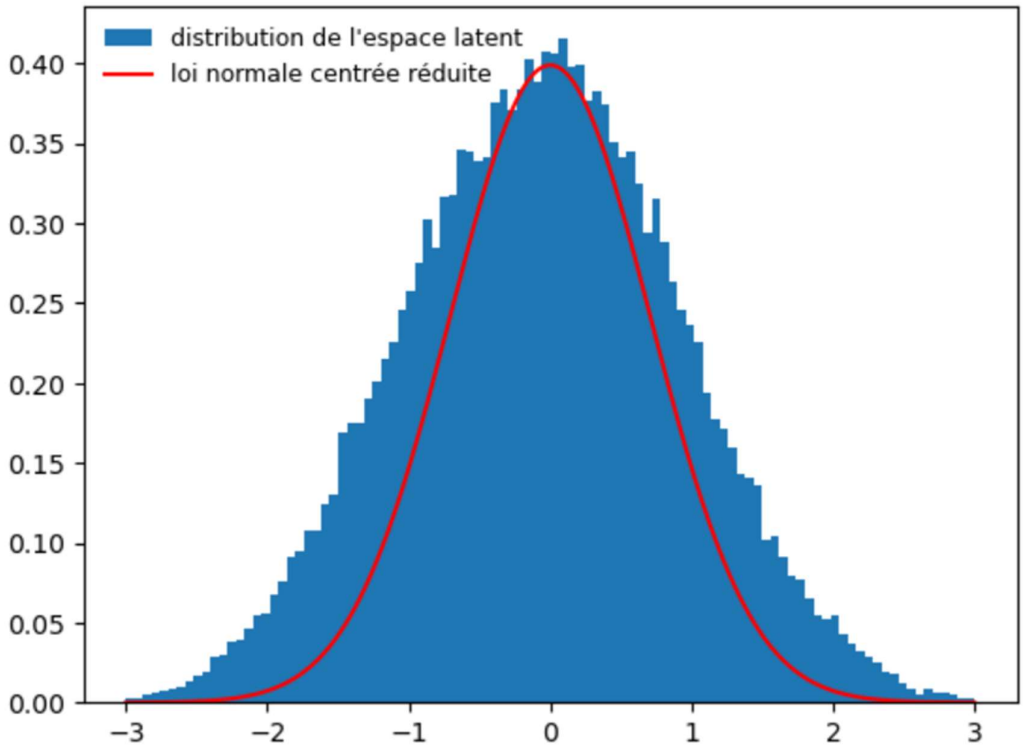


Figure 6.5 Distribution de l'espace latent

- En réalité, nous devrions obtenir un histogramme de dimension $d = 10$. Par soucis de représentation, nous avons mis toutes les données de l'espace latent dans un même vecteur de dimension $d \times n = 10n$. Ainsi, nous visualisons le vecteur sous la forme d'un histogramme de dimension 1.
- Nous avons remarqué que plus on diminue λ, plus l'histogramme converge vers la densité d'une loi normale centrée réduite, ce qui est cohérent.

6.3.3 Résultats

Notre modèle étant prêt, nous pouvons maintenant générer des échantillons d'une loi normale centrée et réduite de dimension $d = 10$. Ces données sont générées dans l'espace latent et nous utilisons par la suite le décodeur pour obtenir un nouveau spectrogramme. Enfin, nous transformons ce spectrogramme en son.
Après vingt tests, on s'aperçoit que pour **80%** des sons générés, on peut reconnaître le chiffre prononcé par la voix virtuelle.

Remarques

- Nous avons toujours une perte de qualité audio, due certainement à la fonction
- Il n'est pas possible de choisir à l'avance le chiffre qu'on va générer.
- Nous avons finalement réalisé le même auto-encodeur variationnel, mais avec la fonction coût $\boldsymbol{f_{cout}} = \boldsymbol{MSE} + \frac{\mathbf{1}}{\boldsymbol{n}} \sum_{\boldsymbol{n}} \|\widehat{\boldsymbol{\mu}}\| + \frac{\mathbf{1}}{\boldsymbol{n}} \sum_{\boldsymbol{n}} \|\widehat{\boldsymbol{\sigma^2}} - \mathbf{1}\|$. Dans ce cas, **25%** des vingt sons générés dans l'espace latent sont reconnaissables. La *MSE* vaut **0.0075** sur les données test. La fonction coût KL_{loss} semble donc être plus optimisée dans ce contexte.

7 Conclusion

Le matériel informatique qui était à notre disposition nous a permis de créer des modèles qui nous semblent très pertinents. En effet, nous avons notamment quatre modèles performants.

- Le premier permet d'identifier un chiffre prononcé en anglais. Il réalise 98% de bonnes prédictions.
- Le second détermine le genre d'une personne à partir d'un son de seulement 3 secondes. 93% des sons sont bien classifiés.
- Le troisième prédit la tranche d'âge d'une personne qui parle pendant 3 secondes, avec 74% de réussite.
- Le dernier est un auto-encodeur variationnel. Il génère des nouvelles voix virtuelles qui prononcent un chiffre an anglais. La majorité (80%) des sons générés sont reconnaissables.

Cependant, certains modèles présentent des inconvénients.

- Si on teste notre premier modèle avec des données personnelles, les résultats ne conviennent plus. Nous avons enregistré nos voix de plusieurs façons différentes, sans succès. Nous pensons toutefois que la source du problème est l'enregistrement de nos voix.
- Lorsqu'on génère des sons avec l'auto-encodeur variationnel, la qualité audio est toujours dégradée. Cette perte vient en partie d'une fonction de *Librosa* : *feature.inverse.mel_to_audio*. Pour contrer cela, nous avons tenté de manipuler des spectogrammes plus grands (de taille 150×78)). Cependant, la dégradation du son est toujours présente. Nous pensons qu'il faut créer des modèles encore plus profonds et performants pour résoudre ce problème.

Sources

- Valerio Velardo, ingénieur en IA et audio et consultant avec un doctorat en musique & IA. Il a publié une série de vidéos très détaillées à propos du son et de son traitement pour des modèles de Deep Learning.
  https://www.youtube.com/playlist?list=PL-wATfeyAMNqIee7cH3q1bh4QJFAaeNv0

- The Science of Sound: How Audio is Recorded and Reproduced
  https://youtu.be/wAhrej9PsYo?si=EYpTkfTjrf3cXfKL

- Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques
  Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques | by Ketan Doshi | Towards Data Science

- Audio Deep Learning Made Simple (Part 2): Why **Mel Spectrograms** perform better
  Audio Deep Learning Made Simple (Part 2): Why Mel Spectrograms perform better | by Ketan Doshi | Towards Data Science

- What is **backpropagation** really doing?
  https://youtu.be/Ilg3gGewQ5U?si=tVAVA3PuE2ANafJv

- A Complete Understanding of **Dense Layers** in Neural Networks
  https://analyticsindiamag.com/a-complete-understanding-of-dense-layers-in-neural-networks/

- **Convolutional** Neural Network : Tout ce qu'il y a à savoir
  Convolutional Neural Network : Tout ce qu'il y a à savoir (datascientest.com)

- Understanding **Variational Autoencoders** (VAEs)
  https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

- **Transpose Convolutions** – Andrew Ng
  https://youtu.be/qb4nRoEAASA?si=SDj8PjZK3Kw6sAY6

- The **Griffin-Lim algorithm**: Signal estimation from modified STFT
  https://speechprocessingbook.aalto.fi/Modelling/griffinlim.html

Commentaires

Rédaction du rapport

Nous sommes conscients qu'aujourd'hui, les intelligences artificielles (notamment *ChatGPT*) sont énormément utilisées dans le cadre de nos études. Nous-même avons régulièrement recours à *ChatGPT* pour résoudre certaines tâches. Toutefois, nous souhaitions créer un rapport qui soit unique, et propre à nous. Par conséquent, nous avons décidé de ne pas utiliser d'intelligence artificielle pour nous aider à rédiger le rapport.
De plus, toutes les illustrations nous appartiennent (sauf la page de garde). Nous les avons créées soit directement sur Python, soit sur le logiciel de dessin *Paint.net*.

Optimisation des modèles

Il est clair que nous aurions pu créer des modèles encore plus performants pour chaque problème étudié. Par exemple, nous aurions pu :

- étendre notre *grid search*, en intégrant notamment les paramètres *hop_length* et *n_fft* de *librosa*,
- augmenter la valeur de K pour la méthode des K-folds,
- tester d'autres traîtements de sons (MFCC, autres spectrogrammes…),
- faire plus de recherche autour des modèles pré-entrainés,
- augmenter la taille de nos bases de données…

Lorsque nous estimions que les modèles fonctionnaient plutôt bien, nous décidions de passer à un nouveau problème. Cette stratégie nous a permis de travailler sur différents modèles, et notamment de découvrir les auto-encodeurs variationnels.