Université de Tunis El Manar
École Nationale d'Ingénieurs de Tunis

Rapport de Projet de Fin d'année 2

Débruitage de fichiers audio par une architecture Deep-Complex U-Network

Elaboré par :
TELMINI Mohamed Fyras

Encadré par :
BEN HAJ HMIDA Moez
HADDAD Hatem

Année universitaire : 2020/2021
Filière : MIndS

Table des matières

Table des figures

Résumé

L'objectif de ce projet est d'employer les techniques du l'apprentissage approfondi, dans un contexte de débruitage de fichiers audio. Les modèles employés sont des modèles de réseau de neurones convolutif et plus précisément basés sur une architecture de type Deep-Complex-U-Network (DC-UNET).
Ce rapport résume les travaux faits dans ce cadre, de la théorique vers la pratique.
mots-clefs : apprentissage approfondi, réseaux de neurones artificiels, DC-UNET, débruitage, réseaux de neurones convolutif, amélioration de la parole

Introduction Générale

Le débruitage des fichiers audio est un problème à haute complexité. Le bruit étant de nature stochastique et imprévisible, pose des difficultés vis à vis sa localisation. Les techniques de débruitage étaient des approches statistiques[17], et permettaient un débruitage restreint à certains types de bruits qui sont faciles à modéliser. Le problème de débruitage est donc un problème de compréhension de ce qui constitue la parole humaine.

Afin de résoudre ce problème à haute complexité, des techniques d'apprentissage approfondi ont été développés. La problématique dont on souhaite résoudre, est le débruitage de fichiers audio contenant de la parole humaine. Ce débruitage devrait se faire sans savoir au-préalable le profil ou type de bruit contenu dans le fichier. Le fichier doit aussi garder une bonne qualité de parole, qui ne se détériore pas par le débruitage.

Dans ce projet, nous allons voir une implémentation d'un modèle de débruitage par réseau de neurones convolutif. La première partie introduit les technologies employées. Par la suite nous discuterons l'implémention du modèle. La dernière partie concerne les expériences faites et leurs résultats et interprétations.

Chapitre 1

Etat de l'art

Introduction

Dans cette partie du rapport, nous allons introduire les technologies utilisées dans le cadre de ce projet. Nous commencerons par la problématique du débruitage des fichiers audio et au contexte spécifique dans lequel nous cherchons à l'utiliser. Nous parlerons par la suite des divers modèles possibles afin de conclure sur lequel nous allons utiliser. Nous décortiquerons par la suite le modèle en question, et des métriques d'évaluation des performances utilisées.

1.1 Notions générales des réseaux de neurones artificiels

L'apprentissage profond, ou deep-learning, est un sous-domaine de l'apprentissage automatique qui utilise des réseaux de neurones artificiels afin de résoudre des problèmes à haute complexité[8]. L'ensemble de ces méthodes sont adaptées à des problèmes de haute abstraction telles que la reconnaissance faciale ou la reconnaissance de paroles. Les architectures des réseaux de neurones permettent de modéliser les données avec une haute abstraction.
Nous introduisons dans cette partie les notions de base d'un réseau de neurones profond, afin de pouvoir expliquer d'autres architectures plus complexes dans les parties suivantes.

1.1.1 Le Perceptron

Un réseau de neurones artificiel est composé de plusieurs couches de Perceptrons. Chaque Perceptron[13] représente un neurone individuel. Cette unité simule le fonctionnement d'un neurone réel.
Le Perceptron peut recevoir une ou plusieurs entrées simultanées, et moyennant des transformations mathématiques, renvoie une sortie. L'opération qui a lieu est simple, le Perceptron reçoit une ou plusieurs entrées x_i, et multiplie chaque x_i par w_i qui est son poids associé. Par la suite, il calcule la somme de ces termes avec le terme w_0 qui est le biais. Finalement, le résultat de ces opérations est donné à une fonction appelée fonction d'activation, et le résultat final de cette fonction est la sortie du neurone.
Le Perceptron est représenté dans la figure 1.1 Un réseau de neurones est enfaîte un agencement parallèle de plusieurs couches de neurones qui peuvent être des Perceptrons ou

autre.

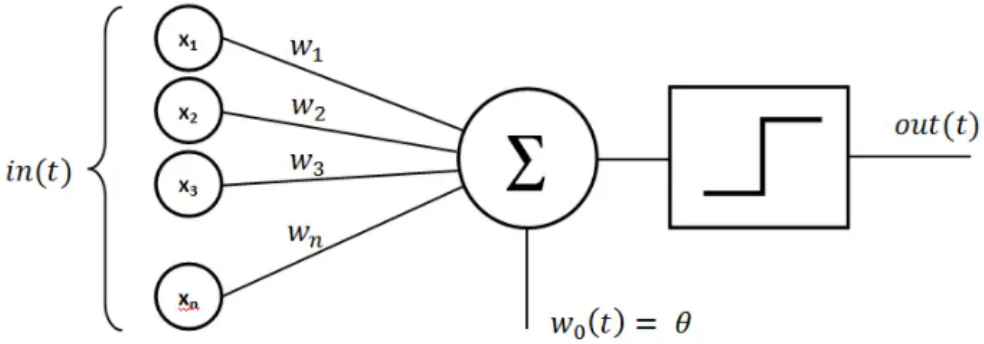

FIGURE 1.1 – Un perceptron simple

1.1.2 Apprentissage et retropropagation

L'utilité du modèle du Perceptron est qu'il est capable de simuler un apprentissage pour une machine. En effet, les poids w_i sont des paramètres que le réseau de neurones essaye d'optimiser afin de résoudre une tache bien particulière.
Le Perceptron est utilisé dans des problèmes de classification binaire, mais les notions qu'il introduit existent dans les autres architectures de réseaux de neurones. Un réseau de neurones est initialement incapable d'exécuter la tâche qu'il est conçu pour faire. Il est nécessaire qu'il apprenne ce qu'il doit faire.
L'apprentissage se fait moyennant des données dont on connaît au préalable le résultat que le réseau doit renvoyer. Chaque donnée passe à travers le réseau, et on récupère sa sortie. Cette sortie est par la suite comparée à la sortie voulue moyennant une fonction de perte. Cette fonction caractérise une distance entre la sortie prédite et la sortie visée. Dans le cas où la sortie est un vecteur, la fonction de perte calcule la moyenne des distances entre chaque composante du vecteur obtenu et le vecteur visé. Par la suite une retropropagation a lieu, le réseau de neurones est parcouru depuis sa dernière couche vers sa première couche en mettant à jour à chaque n couche les poids w_i^n associés. La mise à jour des poids se fait par des algorithmes d'optimisation comme par exemple, l'algorithme de descente du gradient[4] ou l'algorithme de descente du gradient stochastique[10]. Le but de cette opération est d'optimiser les paramètres du réseau afin de minimiser la fonction de perte, et ainsi d'adapter le réseau au problème visé.
Après l'apprentissage, le réseau garde les derniers poids calculés et devient capable de donner des sorties correctes pour des données qu'il n'avait pas vu auparavant.

1.2 Réseau de neurones convolutif

Les réseaux de neurones convolutifs[2] sont des architectures de réseaux de neurones adaptées au traitement d'images. Cette architecture utilise des couches de convolution afin de simuler la perception et le champ visuel de l'oeil humain. Ces architectures sont utilisées pour divers problèmes de vision par ordinateur[16], tels que la classification d'images[18], segmentation d'image[5]...

1.2.1 Les couches de convolution

La couche de convolution dans un réseau de neurones convolutif diffère de l'opération de convolution au sens mathématique. C'est en effet plutôt un produit de corrélation croisée qui est exécuté[15].
L'utilité de cette couche de convolution est qu'elle permet d'extraire, à partir d'une matrice en deux dimensions (voir trois si on travaille avec des images en couleurs) une carte de descripteurs qui contient des informations sur un aspect particulier de l'entrée. L'intuition dans l'usage de cette opération est que les images numériques sont en effet des matrices en deux ou trois dimensions. Et que moyennant des filtres bien construits nous pouvons détecter certains descripteurs dans ces images, tels que les contours par exemples, afin de résoudre des problèmes de traitement d'images ou de classification.
L'opération de convolution (appelée ainsi dans le domaine de l'apprentissage approfondi même si elle n'est pas exactement une convolution) nécessite une entrée une matrice en deux dimensions **X** qui représente l'image et une autre matrice en deux dimensions **M** qui est appelée le filtre de convolution. Le filtre est de taille inférieure à la taille de l'image. L'opération qui a lieu se fait entre le filtre **M** et les segments de **X** qui ont la même taille que **M**. Le filtre fait un parcours de **X** allant de gauche à droite, du haut en bas. à chaque nouvelle superposition entre le filtre et les cases de **X** s'exécute un produit salaire, le résultat de ce produit scalaire est la valeur de la case de sortie d'une nouvelle matrice **M'** qui correspond à cette superposition. La matrice résultante **X'** est appelée la carte des descripteurs de l'image et est nécessairement de dimension inférieure à l'entrée **X**.
En effet si le filtre est établi de manière à détecter une propriété particulière de l'image, le parcours de celle-ci par ce filtre en permet la détection moyennant une réduction de dimension. Cette réduction est importante puisque le traitement d'une image Pixel par Pixel par un réseau de neurones classique nécessite une puissance de calcul énorme. L'idée de réduire la complexité du problème en raisonnant sur des descripteurs particuliers a révolutionné le domaine du traitement d'images par l'apprentissage approfondi. L'information perdue par la convolution est une information spatiale, nous ne savons plus où le descripteur existe, mais nous savons s'il existe ou non.
La figure 1.2 illustre cette opération. L'image en entrée est une image colorée donc l'opération se fait sur chacun de ses couches (une image colorée est une superposition de 3 matrices en deux dimensions où chacune correspond aux valeurs de l'images par rapport à la couleur rouge, vert ou bleu). La multiplication se fait case par case entre chaque case du segment de taille 3x3 de la couche i (i=1,2,3) de la matrice d'entrée et du filtre w_i correspondant. La somme de chaque multiplication est calculée. Puis la somme des 3 multiplications et du terme b0 qui représente le bias.
Le résultat final de l'opération est le scalaire 9 qui est stocké dans la case correspondant à cette opération dans la matrice de sortie.

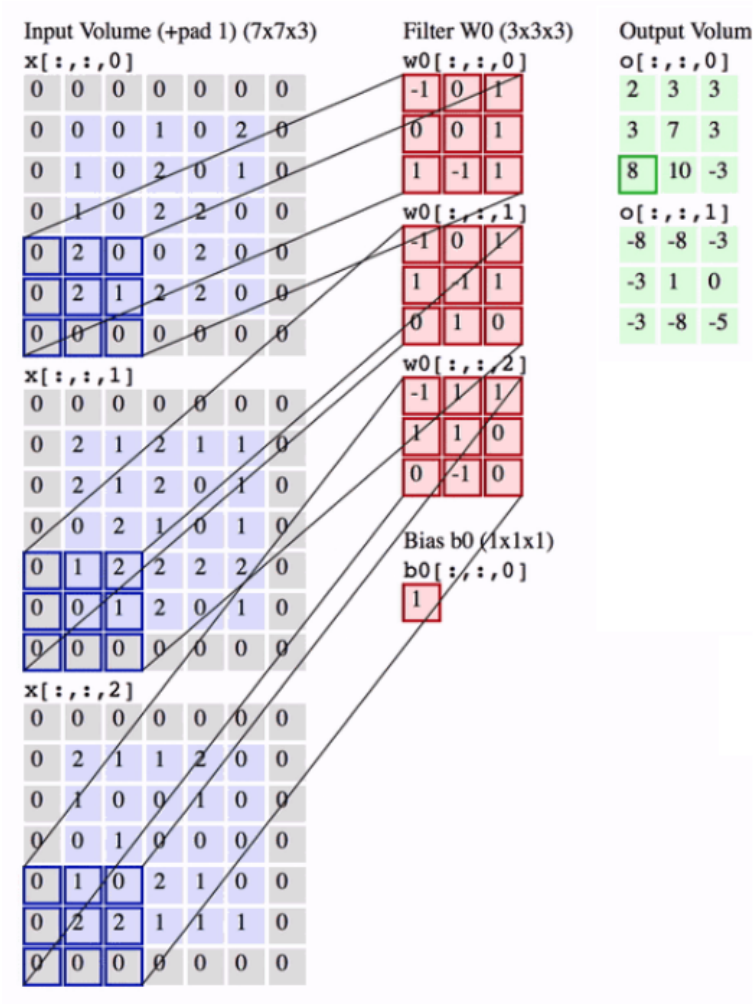

FIGURE 1.2 – L'opération de convolution dans une couche de convolution

Comme nous l'avons dit, il existe deux entrées pour une convolution. Les filtres en particulier ont été au début conçus manuellement par les chercheurs en vision par ordinateur pour détecter des divers descripteurs particuliers tels que les lignes horizontales ou verticales.
L'innovation dans l'introduction de ces filtres dans un réseau ne neurones est que leurs valeurs deviennent des paramètres qui interviennent dans l'apprentissage et qui y sont optimisés au fur et à mesure afin de détecter les descripteurs recherchés sans avoir à les retrouver manuellement. En effet, ces filtres sont multiples dans un même réseau de neurones et tendent à avoir des valeurs très différentes. Ces différences permettent aux réseau de s'adapter aux données sur lesquels il a fait son apprentissage.
Très souvent ces descripteurs sont trop abstraits pour être interprétés, et au fur et à mesure des convolutions l'abstraction de ces descripteurs augmente. C'est pour cela que les réseaux de neurones sont considérés comme des boites noires, puisqu'il sont des opérations dont les entrées eux-mêmes sont des opérations à complexité croissante suivant le nombre de couches employés.

1.2.2 Le redresseur linéaire

Le redresseur linéaire (RELU) est une fonction d'activation utilisée dans les réseaux de neurones artificiels afin d'y introduire une non-linéarité. C'est une fonction dont le calcul est simple. Son résultat est zéro si l'entrée x est négative, ou x si x est positif. Cette linéarité facilite aussi le calcul du gradient lors de l'apprentissage. Un problème du redressement linéaire est le fait que le gradient s'annule pour les sorties négatives. Une variation de redressement linéaire appelée Leaky-RELU[7] permet de contourner ce problème. Elle est définie comme suit :

$$Leaky_{RELU}(x) = \begin{cases} \text{x}, & \text{si } x > 0 \\ \text{ax}, & \text{Sinon} \end{cases}$$

La pente a est très petite, et permet de garder un gradient non nul lors de la retropropagation.

1.2.3 Couches de Batch normalization

Un problème qui se pose lors de l'apprentissage du réseau de neurones, est que les données d'entrée soient très différentes les unes des autres. Cela peut causer un problème au processus d'optimisation de la fonction de perte puisque celle-ci va fluctuer d'une manière importante à chaque étape de l'apprentissage. Et l'apprentissage devient lent et inconsistant.
La batch-normalization est une technique utilisée pour optimiser l'apprentissage d'un réseau de neurones[12]. Une batch est un ensemble d'entrées qui est groupé et qui entre simultanément au réseau. Au lieu de faire l'apprentissage sur chaque fichier d'entrée, on le fait sur plusieurs entrées simultanément et la fonction de perte est optimisées pour toutes ces entrées. Cela résulte en un apprentissage plus consistant puisque le données deviennent en un certain sens normalisées.

1.2.4 Max pool

La couche Max pool effectue une réduction de la carte de descripteurs de l'image afin de réduire le nombre de paramètres du réseau de neurones[3]. Chaque segment de la carte des descripteurs est réduit en un scalaire qui correspond à la valeur maximale qui y est présente. Cette réduction de dimension permet de garder l'information sur ce qui semble être important et d'éliminer ce qui ne l'est pas. Après l'opération du max pooling, la carte de descripteurs qui en résulte contient presque la même quantité d'informations que celle du départ mais cette information est dans un certain sens plus importante.
L'application d'autres couches de convolution par la suite permet au réseau de s'approfondir de plus en plus par rapport à l'image d'entrée en ayant une vision du contexte large de l'image plutôt que ses détails. La figure 1.3 illustre l'opération pour une donnée d'entrée de dimension 4x4 moyennant un filtre 2x2 avec un pas égal à 2.

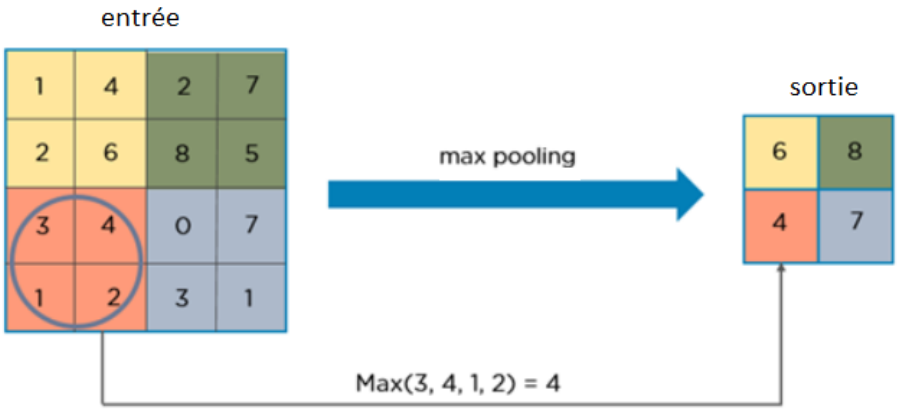

FIGURE 1.3 – Exemple d'une opération de max pooling

1.3 Amélioration de la parole avec les réseaux de neurones convolutifs

1.3.1 La transformée de fourrier locale

La transformée de fourrier locale, aussi appelée transformée de fourrier à fenêtre glissante, est une variation de la transformée de fourrier qui permet d'observer le contenu à la fois fréquentiel et temporel du signal. C'est une transformée dans l'espace des fréquences qui préserve la localité de ceux-ci dans le temps.
Cette transformée est une séquence de transformées de fourrier classiques mais localisées par une fenêtre glissante dans le temps qui effectue le parcours du signal. Comme la transformée de fourrier classique engendre une perte d'information temporelle par rapport aux fréquences présentes dans le signal, il suffit de choisir une fenêtre assez réduite pour l'appliquer à des segments du signal afin qu'on obtienne une restitution presque exacte de l'évolution temporelle des fréquences au long du signal d'entrée.
La transformée de fourrier locale est inversible et est utilisée dans le domaine de l'acoustique pour pouvoir identifier les différentes fréquences présents dans un signal au cours du temps. à partir du résultat de cette transformée nous pouvons construire un spectre (figure 1.4) dans un repère temps-fréquence.
L'amplitude des fréquences présentes est caractérisée par l'intensité des couleurs, allant des moins intense au plus intense pour caractériser les amplitudes des plus faibles aux plus fortes respectivement. Notons que puisque le résultat de cette transformée peut être interprété visuellement, qu'il est possible d'y appliquer les techniques de traitement d'image et plus particulièrement de débruitage d'images pour pouvoir éliminer les fréquences qui portent le bruit.

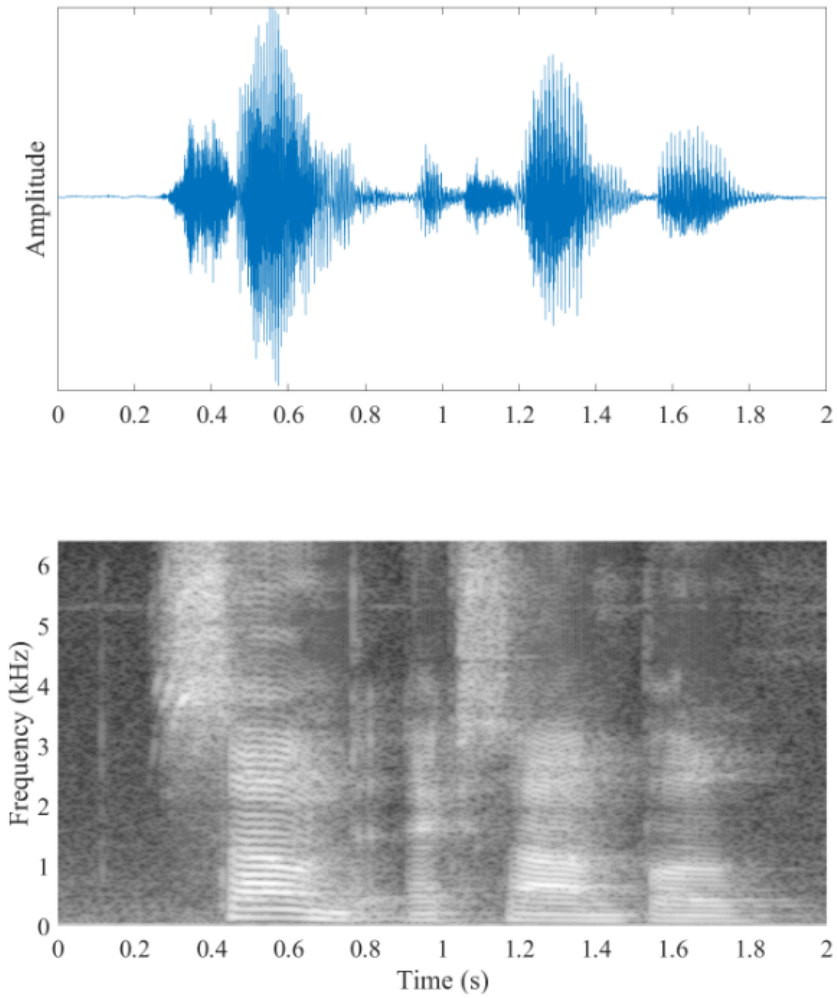

FIGURE 1.4 – Exemple d'un spectre de transformée de fourrier discrète

1.3.2 Auto-encodeur

L'auto-encodeur[9] est l'exemple le plus pertinent de réseau de neurones artificiel convolutif qui est employé pour des fins de reconstruction d'images, et plus précisément pour des fins de débruitage. Il constitué par un encodeur et un décodeur qui sont, en un certain sens, symétriques.
L'encodeur est une unité qui fait une réduction de dimension de l'image en la projetant sur un espace latent. Cette représentation compressée de l'image peut par la suite être donnée au décodeur qui l'utilise pour reconstruire l'image d'entrée. La figure 1.5 illustre clairement l'architecture d'un auto-encodeur.
L'idée derrière cette architecture est qu'il est possible pour l'encodeur d'apprendre à garder les descripteurs d'une image bruitée et d'y créer une représentation latente qui ignore le bruit. Cette représentation compressée est par la suite utilisée par le décodeur qui reconstruit une image débruitée puisque l'information qui caractérise le bruit a été éliminée par l'encodeur.
En effet, l'encodeur garde les caractéristiques de l'image qui sont statistiquement communs. Le bruit est un processus stochastique et n'est donc pas gardé. La représentation latente qui en résulte caractérise le contenu de l'image sans le bruit. Ces deux entités sont très souvent couplées. En effet, le décodeur partage l'architecture et les poids et même

les convolutions de l'encodeur à une transposition prés. Les couches de convolution dans l'encodeur deviennent des couches de convolution transposées dans le décodeur. Pareil pour les autres opérations.

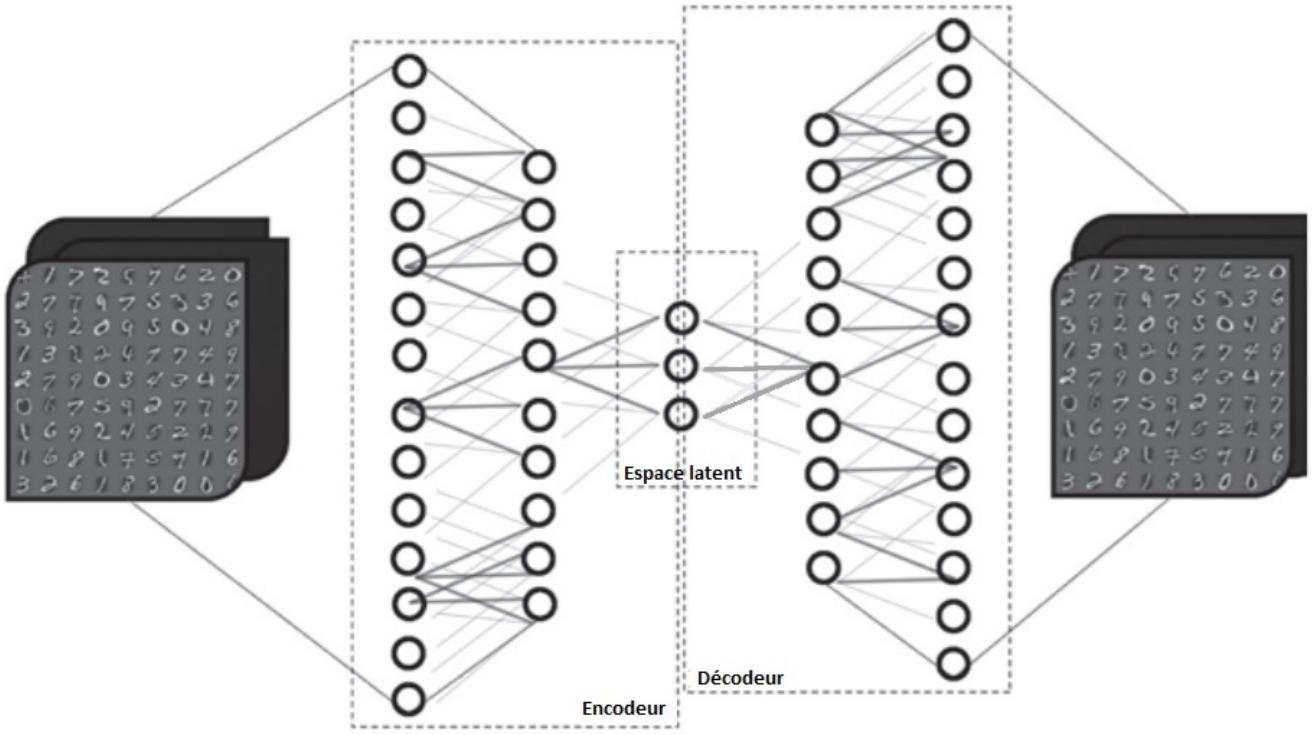

FIGURE 1.5 – Architecture d'un auto-encodeur[15]

Nous avons donc une entrée **x** que l'encodeur *E* transforme en une représentation compressée **c** : $E(\boldsymbol{x})=\boldsymbol{c}$. Cette représentation **c** est transformée par le décodeur *D* en une sortie $\hat{\mathbf{x}}$: $E(\boldsymbol{c})=\hat{\boldsymbol{x}}$. Les paramètres de *E* et *D* sont liés. Il suffit donc de faire l'apprentissage sur l'un d'eux et de transférer les paramètres acquis à l'autre.
L'apprentissage se fait moyennant une fonction de perte *L* qui calcule la distance entre **x** et $\hat{\mathbf{x}}$. Pour un problème de débruitage, l'entrée **x** est une entrée bruitée. Lors de l'apprentissage, la fonction de perte calcule la distance entre **x** et une version débruitée au-préalable de **x** qui est $\tilde{\mathbf{x}}$. La logique impose donc que les jeux de données d'un problème de débruitage soient des fichiers bruitée artificiellement pour qu'on puisse garder leurs versions débruitées afin de les utiliser pour l'apprentissage.
Le bruitage artificiel des fichiers audio est tout simplement une superposition d'un son clair avec un son qui contient purement du bruit. La création du jeu de données bruité va être explicitée clairement durant la seconde partie du rapport.

1.4 Le modèle U-network

La U-network[11] est un réseau de neurones convolutif développé pour des problèmes d'imagerie médicale. Ces problèmes nécessitaient deux critères auxquels une architecture de réseau de neurones convolutif simple ne permettait pas de répondre :

1. La sortie du réseau de neurones devait contenir plusieurs segments, c'est à dire que chaque région de l'image d'entrée devrait être segmentée indépendemment

(cellule cancérigène et cellule non-cancérigène par exemple). Les architectures de réseaux de neurones convolutifs devraient donc parcourir l'image Pixel par Pixel et caractériser chacun. Cela nécessitait une puissance de calcul très importante.

2. Même si un réseau de neurones convolutif était capable de segmenter différemment plusieurs régions d'une image, cela nécessitait que le jeu de données sur lequel il fait son apprentissage soit segmenté en plusieurs régions, cette segmentation augmentait fortement le temps de calcul nécessaire et posait aussi un problème de mémoire par rapport à la taille du jeu de données.

L'architecture U-network permettait de contourner ces deux problèmes. Elle pouvait donner des résultats plus précis en apprenant sur un jeu de données de taille réduite. L'idée derrière cette architecture est qu'elle cherchait à avoir une segmentation précise de l'image sans risquer une perte de contexte général de celle-ci. C'est à dire de garder à la fois l'information précise sur le contenu particulier de l'image, et du contexte général c'est à dire l'image dans son entier.

1.4.1 Architecture

L'architecture U-network est composée de deux parties, la figure 1.6 illustre son fonctionnement :

1. La partie contractante : Elle fonctionne comme un réseau de neurones convolutif simple. Une répétition de trois opérations fondamentales se fait le long de cette partie. La donnée d'entrée est soumise à des convolution par plusieurs filtres de taille 3x3. Le résultat de ces convolutions est une matrice de dimension réduite mais de profondeur augmentée. Remarquons à ce stade que chaque filtre de convolution permet d'extraire une carte de descripteurs de l'image, donc l'application de plusieurs filtres résulte en plusieurs cartes de descripteurs qui sont superposées et qui caractérisent chacun des différents aspects de l'image. Une non-linéarité est par la suite introduite via un redresseur linéaire. Et enfin une opération de max pooling par un filtre de dimension 2x2 avec un pas de 2 exécute un sous échantillonage de chaque carte de descripteurs obtenue. Après chaque opération de sous-échantillonage, la profondeur de la matrice est doublée via des convolutions par des filtres de dimension 3x3. Ces opérations sont répétées jusqu'à obtenir des cartes de descripteurs de taille très réduites mais de nombre très large, qui correspond à une grande segmentation de l'image avec une grande précision sur son contenu. L'information spatiale est cependant perdue le long de ces opérations.
2. La partie décontractante : Cette partie permet de récupérer la segmentation désirée de l'image à partir des cartes de descripteurs crées par la partie contractante. La première opération exécutée est une opération de sur-échantillonage par convolution transposée par des filtre de dimension 2x2. Le sur-échantillonage résulte en une profondeur réduite de la carte de descripteurs, la longueur et largeur de ceux-ci est augmentée. Chaque sortie de sur-échantillonage est concaténée avec la sortie de l'encodeur qui l'oppose, cette concaténation s'appelle une skip connection. Cela permet par la suite, à travers des convolutions par des filtres 3x3 d'obtenir une représentation de résolution supérieure. Ces opérations sont répétées jusqu'à avoir une sortie de résolution désirée. La dernière opération est une convolution par un filtre 1x1 qui projette toutes les cartes de descripteurs sur une matrice de profondeur égale au nombre de labels cherchés.

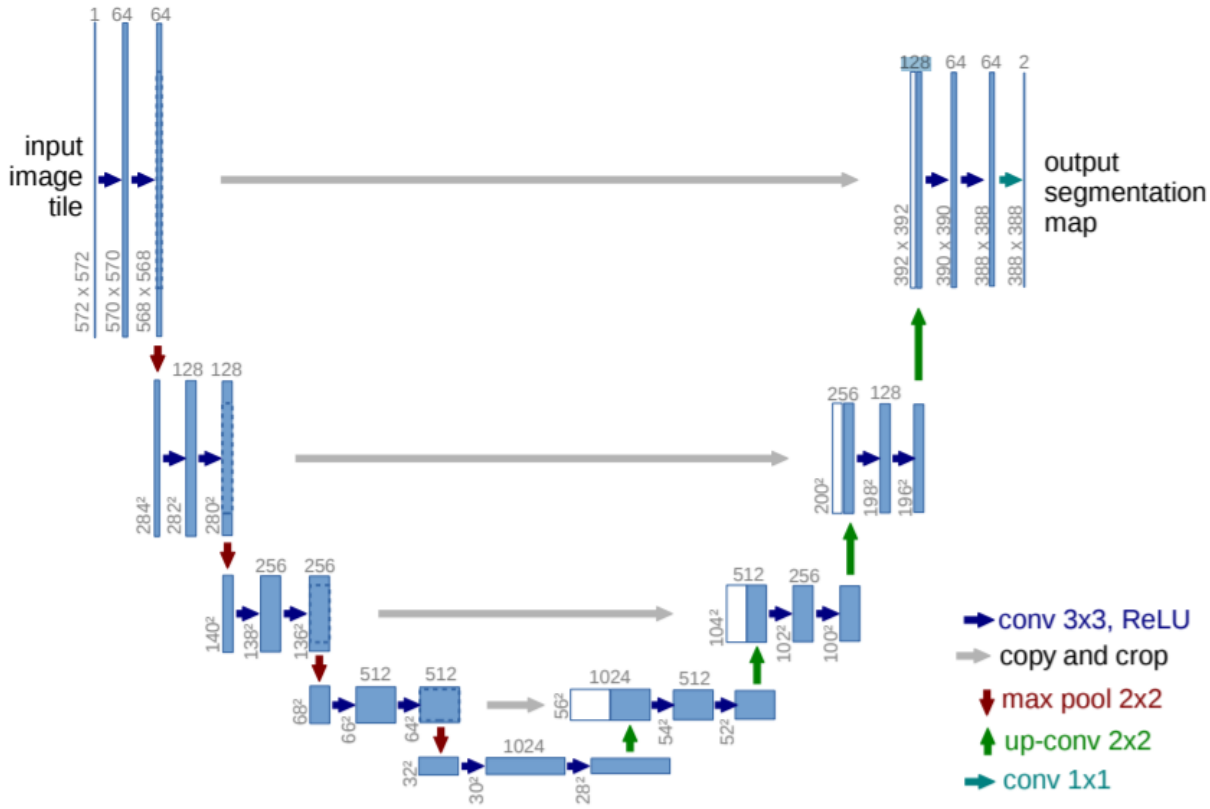

FIGURE 1.6 – Architecture de U-network[11]

Cette architecture permet donc de garder l'information des segments et du contexte général des convolutions à travers l'opération de concaténation, qui permet donc une restitution plus fidèle de l'entrée.
Cette architecture étant adaptée à des problèmes de segmentation du contenu d'une image, nous pouvons l'adapter au problème de débruitage en prenant en entrée la transformée de fourrier d'un signal. Cette représentation est analogue à une représentation visuelle, et nous pourrons essayer de la segmenter en deux labels : bruit et son clair. Cette segmentation permettrait par la suite la construction d'un filtre qui sépare ces deux. La représentation de fourrier étant complexe, il faudrait d'abord adapter les opérations de convolution au domaine complexe.

1.5 DCUNET

Une tendance dans le domaine du débruitage et de l'amélioration de la parole était de transformer le problème de séparation d'un signal audio en un problème de traitement du spectre de la transformée de fourrier locale moyennant des réseaux de neurones convolutifs. Les implémentations de ces techniques utilisaient des réseaux de neurones convolutifs réels, pour estimer l'amplitude du signal clair à partir du signal bruité.
Le problème avec ces estimations est qu'elles ignorent la composante complexe du signal et donc sa phase. La phase du signal clair étaient extraites directement depuis la phase du signal bruité, donc de la composante complexe du spectre. La figure 1.7 montre le résultat du cette estimation qui ignore la phase dans le cas des fichiers très

bruités. En effet, avec un rapport SNR (Speech Noise Ratio) de parole-bruit très petit (donc un bruit très important). La phase récupérée diffère beaucoup de la phase réelle recherchée. Cela montre donc l'intérêt par rapport à l'estimation correcte de cette phase. L'idée de l'architecture Deep-Complex-U-network est d'employer un réseau de neurones convolutif adapté aux calculs complexes, afin de pouvoir estimer le plus adéquatement possible à la fois l'information de l'amplitude et de la phase à partir du signal clair à partir du signal bruité.

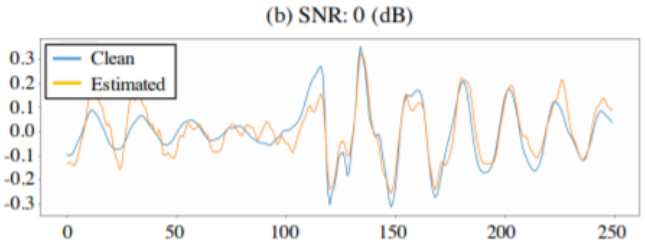

FIGURE 1.7 – Effet de la récupération de la phase d'un signal clair à partir d'un signal fortement bruité[6]

1.5.1 Deep complex CNN[14]

Pour pouvoir implémenter une architecture qui gère des entrées complexes, il faut tout d'abord définir les opérations des couches de base d'un réseau de neurones convolutif dans l'espace complexe.

La convolution complexe

Pour pouvoir définir une convolution complexe, nous commençons par définir le filtre complexe $\mathbf{W} = \mathbf{A} + i.\mathbf{B}$ avec $\mathbf{A}$ et $\mathbf{B}$ deux matrices à valeurs réelles. L'opération s'effectue sur une matrice $\mathbf{X} = \Re(\mathbf{X}) + i.\Im(\mathbf{X})$. Cette matrice peut être interprétée comme la concaténation de deux matrices, l'une étant $\Re(\mathbf{X})$ et l'autre $\Im(\mathbf{X})$. L'opération de convolution étant distributive, Le résultat de convolution est $\mathbf{M} = \mathbf{X} * \mathbf{W} = (\mathbf{A} * \Re(\mathbf{X}) - \mathbf{B} * \Im(\mathbf{X})) + i.(\mathbf{B} * \Re(\mathbf{X}) + \mathbf{A} * \Im(\mathbf{X}))$.
L'opération s'effectue donc comme deux convolutions réelles séparées, ayant les même paramètres $\mathbf{A}$ et $\mathbf{B}$. La figure 1.8 illustre cette opération.

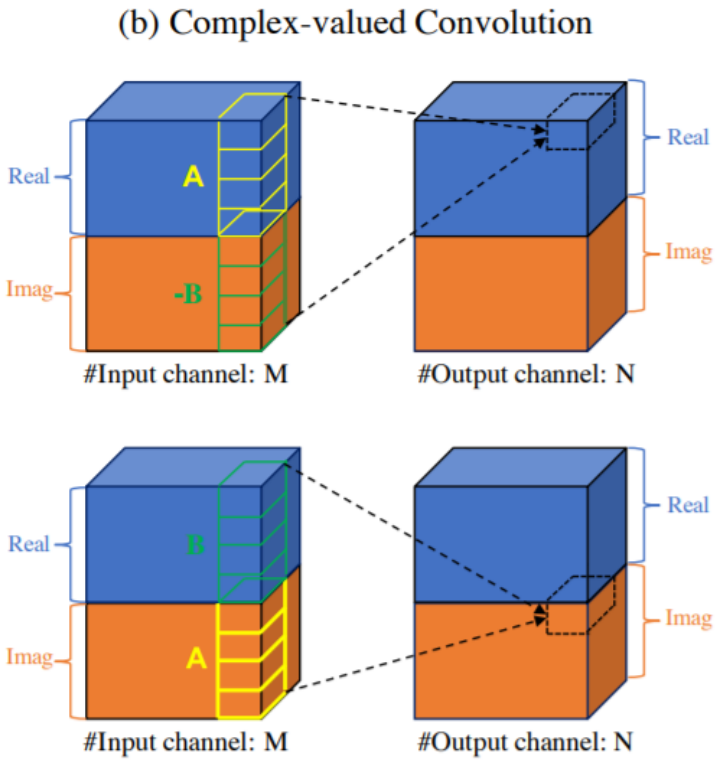

FIGURE 1.8 – La convolution complexe[6]

Le redressement linéaire complexe Le redresseur linéaire complexe est défini comme un couple de redresseurs linéaires réels, l'un est appliqué sur la partie réelle de l'entrée et l'autre sur sa partie imaginaire. L'équation est donc : $\mathbb{C}RELU(\mathbf{X}) = RELU(\Re(\mathbf{X})) + i.RELU(\Im(\mathbf{X}))$

La batch-nomalization complexe Il existe des techniques sophistiquées pour calculer adéquatement une batch-normalization complexe. Cependant, ces techniques sont assez difficiles à implémenter et à expliquer. De plus, l'implémentation de ce projet utilise une implémentation de batch normalization complexe dite "naive". L'observation des résultats de ce projet m'ont permis de conclure que la batch normalization complexe "naive" était plus adéquate puisqu'elle donnait des résultats admissibles tout en ayant une complexité de calcul inférieure aux autres implémentations. Même si elle ne les dépasse pas en terme de performance du modèle, elle a permis de minimiser le temps du training et c'est donc un bon compromis entre les performances et la rapidité.
La batch normalization complexe "naive" est une combinaison linéaire de deux batch normalizations réelles. L'une est appliqué sur la partie réelle de l'entrée et l'autre sur sa partie imaginaire.

La dé-convolution complexe L'opération de dé-convolution complexe est effectuée moyennant la transposée du filtre $\mathbf{W}^T = \mathbf{A}^T + i.\mathbf{B}^T$.

1.5.2 Architecture Deep-Complex-U-Network

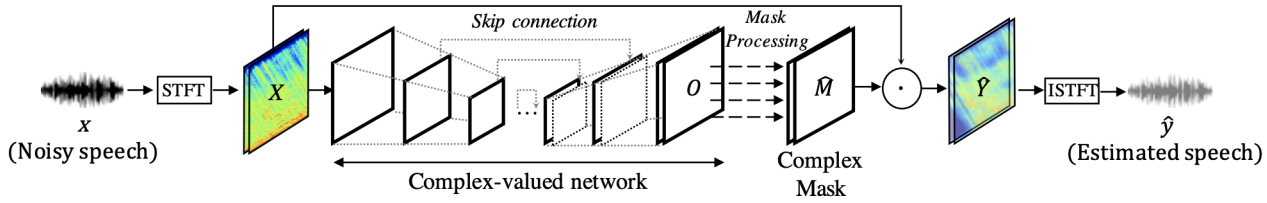

FIGURE 1.9 – L'architecture de la DCUNET[6]

La deep-complex-u-network est une architecture u-network modifiée. Une couche de transformée de fourrier locale permet de récupérer le spectre du signal. Ce spectre passe par la suite dans un auto-encodeur complexe inspiré de l'architecture U-net mais qui exécute des opérations complexes sur la partie réelle du spectre et sa partie imaginaire. Les opérations sont celles explicitées dans la partie précédentes.
Citons à ce stade que le réseau complexe diffère un peu de la U-network. Toutes les couches de convolution sont remplacées par des couches de convolution complexe. Des Couches de batch-normalization sont ajoutées après chaque convolution. Dans la partie de l'encodeur, les couches de max pooling sont remplacées par des couches de convolution de pas 2. Pour le décodeur, la transposée de ces convolutions est exécutée. Les skips-connexions sont gardées tels-quels. Les fonctions d'activations sont toutes des fonctions de redressement linéaire complexe. à la fin du décodeur s'exécute une convolution sans batch normalizations qui génère un filtre. Ce filtre est appliqué au spectre d'entrée pour pouvoir récupérer par la suite le spectre filtré qui est transformé, moyennant la transformée de fourrier locale inverse, en un fichier audio débruité. La figure 1.9 illustre l'architecture d'une deep-complex-u-network.

1.5.3 Le réseau complexe

Il existe 3 différentes implémentations de la deep-complex u-network, qui diffèrent par la profondeur de leur réseau complexe.
L'implémentation choisie est la naive-DCUNET16. Le terme naive caractérise le fait qu'elle utilise une batch normalization dite "naive" comme expliqué plus haut. L'auto-encodeur possède 16 couches en profondeur, 8 pour l'encodeur et 8 pour le décodeur. La figure 1.10 illustre l'architecture de ce réseau complexe.

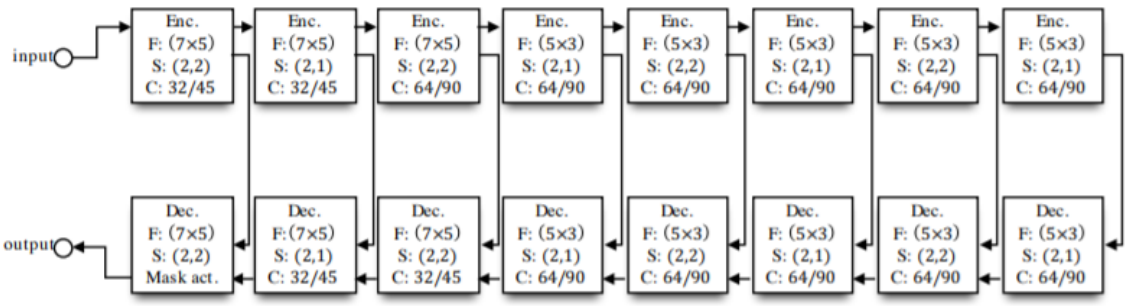

FIGURE 1.10 – L'architecture de l'auto-encodeur de la naive-DCUNET16[6]

1.5.4 La fonction de perte

La fonction de perte proposée pour ce modèle est le rapport pondéré entre la source (son clair) et la distortion (bruit) appelée weightedSDR[6] (sound distortion ratio). Soit y la donnée du son clair et y_2 le son clair prédit par le réseau de neurones. Le rapport entre la source est la distortion est défini comme suit :
$loss_{SDR}(y, y_2) = -\frac{\langle y, y_2 \rangle}{\|y\| \|y_2\|}$
Cette fonction est bornée par -1 et 1. Mais s'annule quand l'entrée est uniquement bruitée et pose donc un problème lors de la retro-propagation du gradient. Soit z le bruit présent dans l'entrée et $\hat{z}$= x$\hat{y}$ la quantité qui exprime le bruit prédit à partir de la donnée d'entrée x. Le terme $loss_{SDR}(z,\hat{z})$ caractérise la qualité de prédiction du bruit. L'idée de la fonction weightedSDR et de calculer ces deux rapports moyennant un facteur de pondération $\alpha = \frac{\|y\|^2}{(\|y\|^2 + \|z\|^2)}$ qui caractérise le rapport d'énergies entre le son et le bruit. Chaque terme est par la suite pondéré proportionnellement aux énergies afin d'équilibrer sa contribution.
L'écriture finale de cette fonction est :
$loss_{wSDR}(x, y, \hat{y}) = \alpha loss_{SDR}(y,\hat{y}) + (1 - \alpha) loss_{SDR}(z,\hat{z})$

Conclusion

L'architecture deep-complex-u-network permet donc d'effectuer un débruitage de fichiers audio en estimant une reconstruction de la parole claire depuis une parole bruitée. La performance de ce modèle doit donc être étudiée. Mais avant cela, il faudrait chercher comment l'implémenter. La partie qui va suivre va expliciter la construction du modèle sous python.

Chapitre 2

Implémentation

Introduction

Dans cette partie, nous allons parcourir le code de l'implémentation du modèle. Les modules de l'auto-encodeur seront explicités, puis l'architecture de l'ensemble du modèle. Par la suite, deux jeux de données distincts seront présentés. Enfin les détails de la boucle d'apprentissage ainsi que la boucle d'exécution du modèle vont être décrits.

2.1 Implémentation on avec Tensorflow

Le code est écris en python, moyennant les outils de Tensorflow[1]. Tensorflow est une bibliothèque open source gratuite développée par Google afin de faciliter l'implémentations des modèles de deep learning. C'est un outil adapté au calcul tensoriel. Il présente une interface de développement de haut niveau qui peut créer et gérer tous les aspects de l'apprentissage approfondi en quelques lignes de codes. Cette interface est personalisable, et permet l'accès à des couches plus bas niveau du code afin de personnaliser les fonctions et outils utilisés.
L'implémentation du code est récupérée depuis github.

2.1.1 Modules d'encodage et décodage

Cette partie du code définit les deux composantes principales de l'auto-encodeur :

1. L'encodeur qui fait une compression des données
2. Le décodeur qui génère des données à partir de leurs compressions

Les deux composants sont en un certain sens symétriques.
L'encodeur fonctionne comme l'une de deux couches, suivant le paramètre strides. Strides caractérise les pas avec lesquels le filtre de convolution parcours la matrice des données. Si le parcours est égal à deux pas dans la verticale et un pas dans l'horizontale, l'encodeur fonctionne comme une couche de convolution. Si les deux pas sont égales à deux, l'encodeur fonctionne comme une couche de max-pooling. Dans les deux cas, l'encodeur exécute trois opérations :

1. complex_conv2D() : Cette fonction exécute les opérations de convolutions, elle récupère en donnée les filtres et leurs paramètres. Kernel_size est la dimension du filtre, et strides son pas. L'encodeur renvoie aussi les parties réelles et imaginaires du résultat de convolution.

2. Cleaky_RELU() : Cette fonction applique un redressement linéaire complexe aux données.
3. complex_NaiveBatchNormalization() : Cette fonction exécute une batch normalization complexe aux données.

Le décodeur agit d'une manière symétrique, mais exécute une concaténation des données encodées avec leur équivalent de l'étape précédente de l'encodage. Il exécute des convolutions transposées et des sur-échantillonages par dé-convolution, qui sont les opérations inverses des convolutions et du max-pooling respectivement. La figure 2.1 montre ces deux fonctions.

```
def encoder_module (real, imag, filters, kernel_size, strides, training = True):

    conv_real, conv_imag = complex_Conv2D(filters = filters, kernel_size = kernel_size, strides = strides)(real, imag)
    out_real, out_imag   = CLeaky_ReLU(conv_real, conv_imag)
    out_real, out_imag   = complex_NaiveBatchNormalization()(conv_real, conv_imag, training = True)

    return out_real, out_imag, conv_real, conv_imag


def decoder_module (real, imag, concat_real, concat_imag, filters, kernel_size, strides, training = True):

    if concat_real == None and concat_imag == None:
        pass
    else:
        real = concatenate([real, concat_real], axis = 3)
        imag = concatenate([imag, concat_imag], axis = 3)
    deconv_real, deconv_imag = complex_Conv2DTranspose(filters = filters, kernel_size = kernel_size, strides = strides)(real, imag)
    deconv_real, deconv_imag = CLeaky_ReLU(deconv_real, deconv_imag)
    deconv_real, deconv_imag = complex_NaiveBatchNormalization()(deconv_real, deconv_imag, training = True)

    return deconv_real, deconv_imag
```

FIGURE 2.1 – Les modules de l'auto-encodeur

2.1.2 Définition du modèle

L'architecture utilisée est une architecture avec 8 encodeurs et 8 décodeurs. D'une manière itérative, chaque couche d'encodage reçoit les sorties de la couche qui la précèdent. Les couches de décodage reçoivent aussi les sorties des encodeurs qui leurs sont symétriques. Le modèle est défini comme un objet qui possède trois méthodes :

L'initialisation : cette méthode permet de récupérer les paramètres du modèle

model() : cette méthode exécute toutes les opérations du modèle et renvoie ses résultats

get_config() : cette méthode récupère la configuration du modèle pour permettre sa sauvegarde

La donnée d'entrée est un array qui contient le fichier audio, chaque case est la valeur d'un échantillon du fichier. Cet array est transformé par la suite en un spectre par transformée de fourrier locale. Le spectre passe par la suite à travers toutes les couches de l'auto-encodeur.

L'encodeur est construit de couches de convolution suivies de couches de max pooling, le décodeur est construit par des couches de déconvolution et de sur-échantillonage. Le résultat du dernier décodage est le masque obtenu. La spectre d'entrée est multiplié par le masque et donne un spectre débruité. Le spectre débruité est transformé en un array qui contient les échantillons du fichier audio débruité moyennant une transformée de fourrier locale inverse. Le modèle renvoie finalement le résultat débruité. La figure 2.2 montre la méthode model() qui exécute toutes ces opérations.

```
def model (self):

    noisy_speech = Input (shape = (self.input_size, 1), name = "noisy_speech")

    stft_real, stft_imag = STFT_network(**self.STFT_network_arguments)(noisy_speech)
    stft_real, stft_imag = tranposed_STFT(stft_real, stft_imag)

    real, imag, conv_real1, conv_imag1 = encoder_module(stft_real, stft_imag, 32, (7, 5), (2, 2), training = self.norm_trainig)
    real, imag, conv_real2, conv_imag2 = encoder_module(real, imag, 32, (7, 5), (2, 1), training = self.norm_trainig)
    real, imag, conv_real3, conv_imag3 = encoder_module(real, imag, 64, (7, 5), (2, 2), training = self.norm_trainig)
    real, imag, conv_real4, conv_imag4 = encoder_module(real, imag, 64, (5, 3), (2, 1), training = self.norm_trainig)
    real, imag, conv_real5, conv_imag5 = encoder_module(real, imag, 64, (5, 3), (2, 2), training = self.norm_trainig)
    real, imag, conv_real6, conv_imag6 = encoder_module(real, imag, 64, (5, 3), (2, 1), training = self.norm_trainig)
    real, imag, conv_real7, conv_imag7 = encoder_module(real, imag, 64, (5, 3), (2, 2), training = self.norm_trainig)
    real, imag, _, _                   = encoder_module(real, imag, 64, (5, 3), (2, 1), training = self.norm_trainig)
    center_real1, center_imag1 = decoder_module(real, imag, None, None, 64, (5, 3), (2, 1), training = self.norm_trainig)
    deconv_real1, deconv_imag1 = decoder_module(center_real1, center_imag1, conv_real7, conv_imag7, 64, (5, 3), (2, 2), training = self.norm_trainig)
    deconv_real1, deconv_imag1 = decoder_module(deconv_real1, deconv_imag1, conv_real6, conv_imag6, 64, (5, 3), (2, 1), training = self.norm_trainig)
    deconv_real1, deconv_imag1 = decoder_module(deconv_real1, deconv_imag1, conv_real5, conv_imag5, 64, (5, 3), (2, 2), training = self.norm_trainig)
    deconv_real1, deconv_imag1 = decoder_module(deconv_real1, deconv_imag1, conv_real4, conv_imag4, 64, (5, 3), (2, 1), training = self.norm_trainig)
    deconv_real1, deconv_imag1 = decoder_module(deconv_real1, deconv_imag1, conv_real3, conv_imag3, 32, (5, 3), (2, 2), training = self.norm_trainig)
    deconv_real1, deconv_imag1 = decoder_module(deconv_real1, deconv_imag1, conv_real2, conv_imag2, 32, (5, 3), (2, 1), training = self.norm_trainig)
    deconv_real1, deconv_imag1 = decoder_module(deconv_real1, deconv_imag1, conv_real1, conv_imag1, 1, (5, 3), (2, 2), training = self.norm_trainig)

    enhancement_stft_real, enhancement_stft_imag = mask_processing(deconv_real1, deconv_imag1, stft_real, stft_imag)
    enhancement_stft_real, enhancement_stft_imag = transpoed_ISTFT(enhancement_stft_real, enhancement_stft_imag)

    enhancement_speech = ISTFT_network(**self.STFT_network_arguments)(enhancement_stft_real, enhancement_stft_imag)
    enhancement_speech = tf.reshape(enhancement_speech, (-1, self.input_size, 1))

    return Model(inputs = [noisy_speech], outputs = [enhancement_speech])
```

FIGURE 2.2 – La méthode model()

2.2 Jeux de données

Pour pouvoir exécuter l'apprentissage, il fallait obtenir un jeu de données de fichiers audio qui ont deux versions. L'une est une voix claire, l'autre est artificiellement bruitée. Dans le cadre de ce projet, deux jeux de données ont été utilisés :

Un jeu de données de 2500 fichiers en anglais, chaque fichier a une durée égale à une seconde. Ce jeu de données a été récupéré depuis internet, il contient plusieurs types de bruits tels que le bruit statique, des imputions, paroles de fond, bruit de circulation..

Un jeu de données de 1758 fichiers en dialecte tunisien, chaque fichier a une durée égale à une seconde. Ce jeu de données a été créé manuellement à partir d'un enregistrement sur Youtube d'une session du parlement tunisien.
Le second jeu de données a été crée en récupérant environs une demi-heure de parole claire à partir de l'enregistrement de la session du parlement. Cette parole a été par la suite superposée avec un segment de même longueur qui contenait divers bruits isolés récupérés depuis le même enregistrement (paroles de fond, toux..). Notons que le second jeu de données contient moins de types de bruits que le premier.
Une fonction python a été utilisée pour fragmenter les deux fichiers (bruité et clair) en 1758 segments de même longueur. La taille des segments devait être uniforme pour qu'ils puissent passer dans le réseau de neurones.

2.3 Apprentissage

L'apprentissage se fait sur le jeu de données dont on dispose. Ce jeu de données est séparé en un ensemble d'apprentissage et un ensemble de validation. Dans les deux ensembles, Le parcours se fait par batch. C'est à dire pour une taille de batch égale à n, n fichiers sont donnés simultanément au modèle. Cela permet un apprentissage plus rapide et plus consistant. Chaque parcours du jeu de données s'appelle une epoch, Le modèle subit un apprentissage sur plusieurs epoch et à chaque epoch la fonction de perte est calculée pour chacun des ensembles. Seulement la valeur de fonction de

perte de l'ensemble d'apprentissage intervient dans la retropropagation du gradient. C'est à dire que le modèle optimise ses performances sur l'ensemble d'apprentissage uniquement.
L'ensemble de validation n'intervient pas dans l'apprentissage. Son rôle est de caractériser la généralité du modèle. Si les performances du modèle s'améliorent pour l'ensemble de validation, c'est qu'il devient de plus en plus général et qu'il apprend mieux. Si par contre la fonction de perte de l'ensemble de validation commence à diverger, c'est que le modèle fait un overfitting. C'est à dire que sa performance sur les fichiers qu'il n'a jamais vu devient médiocre.
L'idéal est de récupérer un modèle avec une valeur de perte minimale à la fois pour l'ensemble d'apprentissage et pour l'ensemble de validation.

2.3.1 Boucle d'apprentissage

L'apprentissage a lieu comme suit :

1. Un parcours du jeu des données commence. les deux ensembles (apprentissage et validation) sont divisés au-préalable.
2. L'optimisateur est initialisé avec le taux d'apprentissage qui correspondent à l'epoch courante. le taux d'apprentissage est déterminé tels que les mises à jours des paramètres du modèle deviennent de plus en plus fines le long de l'apprentissage. La fonction qui le calcule est donnée dans la figure 2.3.
3. La boucle d'apprentissage commence avec les données d'apprentissage. Leurs parcours se fait par batch. Pour chaque batch, la sortie du modèle est calculée. La fonction de perte est calculée par la suite et sa valeur est donnée à l'optimisateur afin qu'il modifie les paramètres du modèle pour la minimiser. La figure 2.4 présente la boucle d'apprentissage.
4. La boucle de validation commence par la suite. Le parcours se fait comme dans la boucle d'apprentissage, mais sans modification des paramètres du modèle.
5. Le modèle sauvegarde ses nouveaux paramètres, ou son état (selon ce qu'on cherche à récupérer) à chaque nombre d'epoch spécifié.

La figure 2.5 montre le code de d'apprentissage pour chaque epoch.

```
def learning_rate_scheduler (epoch, learning_rate):
      if (epoch+1) <= int(0.5*epoch):
            return 1.00 * learning_rate
      elif (epoch+1) > int(0.5*epoch) and (epoch+1) <= int(0.75*epoch):
            return 0.20 * learning_rate
      else:
            return 0.05 * learning_rate
```

FIGURE 2.3 – Le calcul du taux d'apprentissage

```
def loop_train (model, optimizer, train_noisy_speech, train_clean_speech):
      tf.config.experimental_run_functions_eagerly(True)
      with tf.GradientTape() as tape:
            train_predict_speech = model(train_noisy_speech)
            if loss_function == "SDR":
                  train_loss = modified_SDR_loss(train_predict_speech, train_clean_speech)
            elif loss_function == "wSDR":
                  train_loss = weighted_SDR_loss(train_noisy_speech, train_predict_speech, train_clean_speech)

      gradients = tape.gradient(train_loss, model.trainable_variables)
      optimizer.apply_gradients(zip(gradients, model.trainable_variables))
      return train_loss
```

FIGURE 2.4 – La boucle d'apprentissage

```
for epoch in tqdm(range (total_epochs)):
      train_batch_losses = 0
      test_batch_losses  = 0
      optimizer          = tf.keras.optimizers.Adam(learning_rate = learning_rate_scheduler(epoch, learning_rate), beta_1 = 0.9)

      'Training Loop'
      for index, (train_noisy_speech, train_clean_speech) in tqdm(enumerate(train_generator)):
            loss = loop_train (model, optimizer, train_noisy_speech, train_clean_speech)
            train_batch_losses = train_batch_losses + loss

      'Test Loop'
      for index, (test_noisy_speech, test_clean_speech) in tqdm(enumerate(test_generator)):
            loss  = loop_test (model, test_noisy_speech, test_clean_speech)
            test_batch_losses  = test_batch_losses + loss

      'Calculate loss per batch data'
      train_loss = train_batch_losses / train_step
      test_loss  = test_batch_losses / test_step

      templet = "Epoch : {:3d},     TRAIN LOSS : {:.5f},     TEST LOSS  :  {:.5f}"
      print(templet.format(epoch+1, train_loss.numpy(), test_loss.numpy()))

      if ((epoch+1) % 10) == 0: #save frequency was 10
            model.save_weights("./model_save/" + save_file_name + str(epoch+1) + ".h5")
```

FIGURE 2.5 – La boucle du parcours du jeu de données lors de l'apprentissage

La sauvegarde du modèle est possible moyennant deux approches :

Si on souhaite sauvegarder l'architecture de modèle en son entier, la méthode model.save() est utilisée. Tous les paramètres du modèle et de l'optimisateur sont sauvegardés, et le chargement du fichier de sauvegarde permet de reprendre l'apprentissage. Il permet aussi de récupérer directement l'architecture du modèle sans avoir à la redéfinir par la méthode model() de la figure 2.2.

Si on souhaite sauvegarder seulement les paramètres du modèle, la méthode model.save_weights() est utilisée.

2.4 Exécution du modèle

Une fois l'apprentissage terminé, l'exécution du modèle se fait ainsi

1. L'ensemble des fichiers qu'on souhaite débruiter doit être récupéré. Un fichier texte contient les chemins d'accès aux fichiers est ainsi parcouru.
2. Une boucle qui parcours l'ensemble des chemins récupérés commence. Chaque fichier est chargé comme un tableau des valeurs de ses échantillons.
3. Un pré-traitement est nécessaire, certains fichiers son sont encodés sur des tableaux dont le type de données n'est pas le même que ceux du jeu de données sur lequel le modèle a fait son apprentissage. Une conversion est exécutée.
4. Le modèle est initialisé par la méthode model() puis ses paramètres sauvegardés après l'apprentissage sont récupérés par la méthode load_weights().

5. Le fichier passe par le modèle. Puisque le modèle est adapté à des fichiers d'une longueur précise, le fichier d'entrée est segmenté par la fonction inference() dans la figure 2.6. Chaque segment est par la suite donné au modèle. Le résultat final est la concaténation de tous les segments prédits.
6. Le fichier débruité est enfin sauvegardé.

La figure 2.7 montre le code de l'exécution.

```
def inference(sampling_rate,unseen_noisy_speech):
    restore=[]
    for index2 in range (int(len(unseen_noisy_speech) / speech_length)):
        split_speech = unseen_noisy_speech[speech_length * index2 : speech_length * (index2 + 1)]
        split_speech = np.reshape(split_speech, (1, speech_length, 1))
        enhancement_speech = model.predict([split_speech])
        predict = np.reshape(enhancement_speech, (speech_length, 1))
        restore.extend(predict)
    restore = np.array(restore)
    restore = np.int16(restore)
    return(restore)
```

FIGURE 2.6 – la fonction inference()

```
for index,input_path in tqdm(enumerate(inputs)):
    print("\ninput path is ",input_path)
    fs, data = wavfile.read(input_path)

    if data.dtype==np.float32:
        data=data*32767
        data=data.astype(np.int16)

    if not(((len(data)-1)%speech_length)==0):
        dist=np.zeros(speech_length-(len(data)-1)%speech_length)
        dist=dist.astype(data.dtype)
        data=np.append(data,dist)

    outputch=outputs[inputs.index(input_path)]
    model=Naive_DCUnet16().model()
    model.load_weights(load_model_path)
    print("output path is ",outputch)
    prediction=inference(fs,data)
    wavfile.write(outputch, rate = fs, data = prediction)
```

FIGURE 2.7 – L'exécution du modèle

Conclusion

Cette implémentation s'est donc faite en plusieurs étapes, chacune traitant une partie bien particulière du fonctionnement du modèle. Une fois implémenté, il faudrait que l'apprentissage aie lieu. Pour que enfin on puisse le tester sur des fichiers bruités afin d'en conclure des résultats et des potentielles améliorations à faire. La partie qui va suivre montre comment l'apprentissage est exécuté. Par la suite l'analyse les divers résultats par rapport aux deux jeux de données utilisés sera faite, afin de conclure sur la totalité de ce projet.

Chapitre 3

Expérimentations

Introduction

Dans cette partie, on va commencer par présenter le service Google colaboratory sur lequel l'apprentissage a été fait. Par la suite, l'analyse du déroulement de l'apprentissage sur les deux jeux de données sera faite. Chaque analyse de l'évolution de la fonction de perte du modèle par jeu de données est suivie par l'observation des résultats du débruitage pour cette même version du modèle. Enfin, une conclusion sera faite sur les améliorations possibles à appliquer afin d'améliorer les performances de ce réseau de neurones.

3.1 Préliminaires

3.1.1 Training sous Google Colaboratory

La plate-forme Google colaboratory est développée par Google afin de permettre aux chercheurs de développer d'exécuter des codes python en ligne sur des appareils de haute puissance. Cette plate-forme donne accès à des unités de calcul à haute puissance et permet donc d'accélérer les tâches de calculs lourdes, principalement ceux qui concernent les réseaux de neurones et l'intelligence artificielle. Cette plate-forme est gratuite pour des sessions d'usage limités. Elle permet aussi d'accéder à des données depuis Google drive. La figure 3.1 montre l'interface d'utilisation de Google colaboratory.

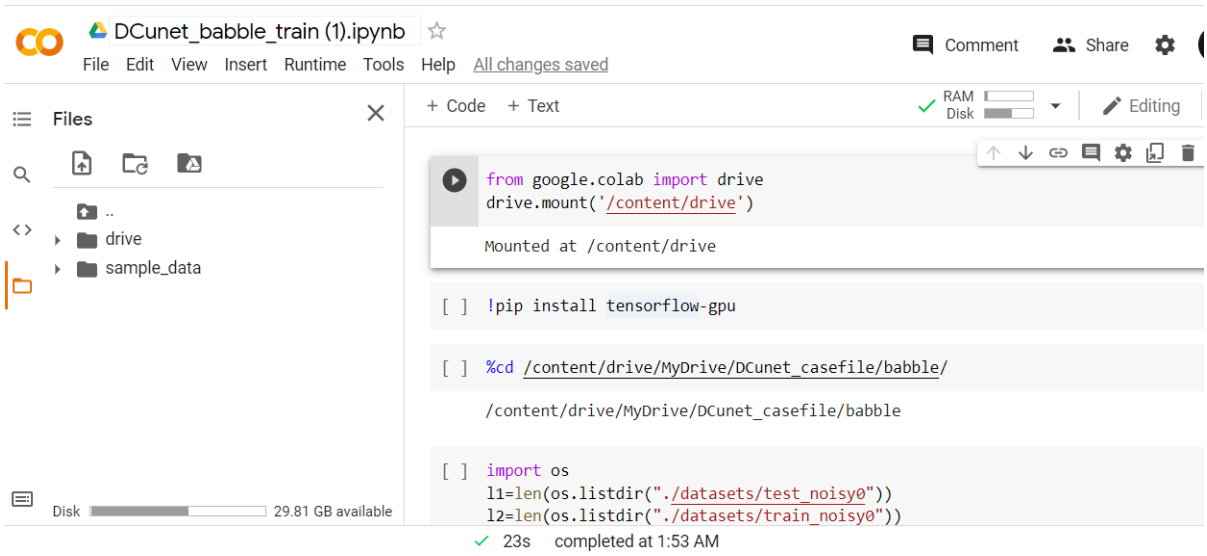

FIGURE 3.1 – l'interface de Google colaboratory

Une carte graphique modèle Tesla-T4 a été utilisée durant l'apprentissage, possédant une mémoire graphique de 15GB. Le nombre de calculs effectués dépasse les 10Î2 opérations par seconde, la figure 3.2 montre les caractéristiques de cette carte graphique. La taille de batch était de 11 fichiers. L'apprentissage prenait 8 heures, qui est la limite d'usage de Google colaboratory.

```
[ ] !nvidia-smi

Sun Jun  6 11:54:24 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 465.27       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   55C    P8    10W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

FIGURE 3.2 – Les caractéristiques de la carte graphique Tesla T4

3.2 Training avec le premier jeu de données

Le premier apprentissage s'est fait avec le jeu de données en anglais. Ce jeu de données comprenait plusieurs types de bruits, donc le modèle créé devrait être robuste et général.

La courbe représentée dans la figure 3.3 montre l'évolution de la fonction de perte lors de l'apprentissage. Le modèle a pu exécuter 380 epoch en total durant la durée de l'apprentissage.

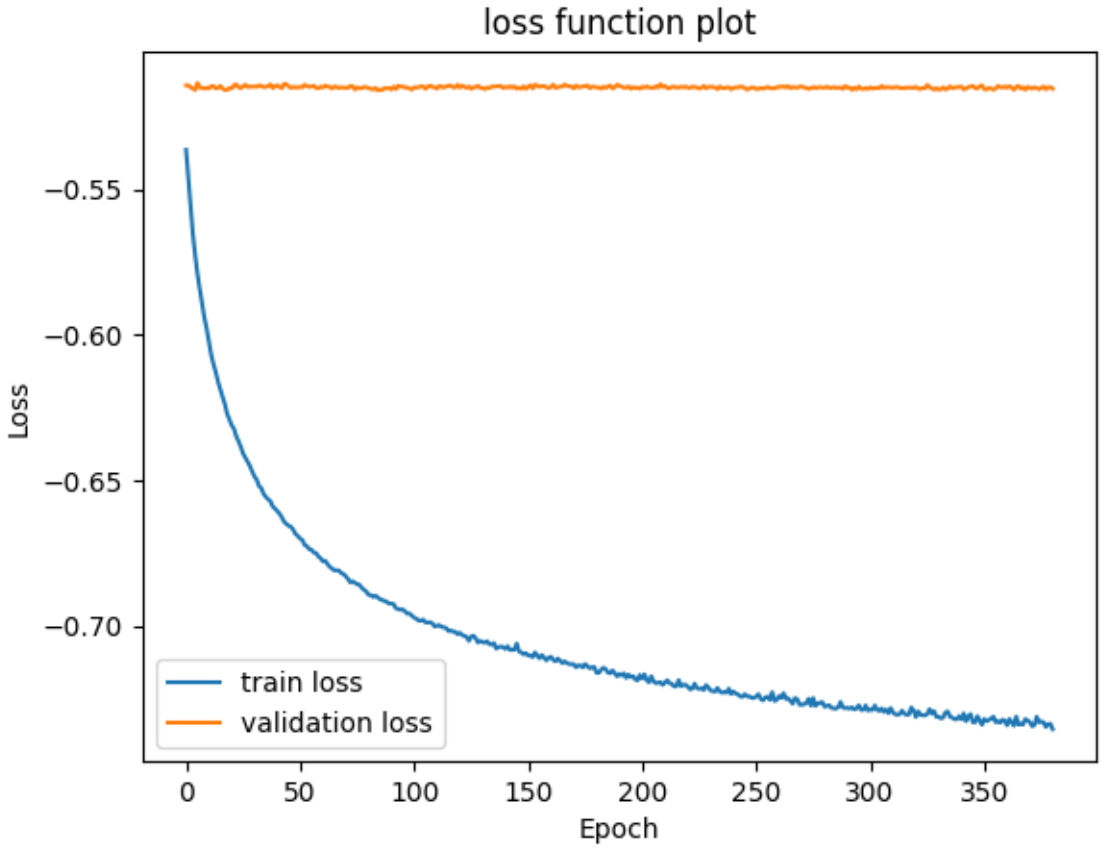

FIGURE 3.3 – La perte pour l'apprentissage sur le premier jeu de données

Le minimum théorique de cette fonction étant égal à -1, on observe que l'ensemble d'apprentissage atteint une perte de -0.73. L'ensemble de validation oscille légèrement autour de -0.51. On peut conclure donc que le modèle n'arrive pas a se généraliser, cela peut provenir de plusieurs raisons :

1. La complexité du modèle n'est pas adaptée au problème qu'on veut résoudre
2. Il existe un problème dans les données d'apprentissage
3. Il existe un problème dans les hyper-paramètres de l'apprentissage, tels que le taux d'apprentissage, la taille de chaque batch..

D'autre part, nous pouvons observer l'évolution de la performance du modèle selon les epoch en visualisant le spectre d'un même fichier audio après le débruitage pour différentes sauvegardes du modèle. La figure 3.4 nous montre cette évolution, le fichier bruité étant la "Baseline", notons que ce fichier contient de la parole en arabe.

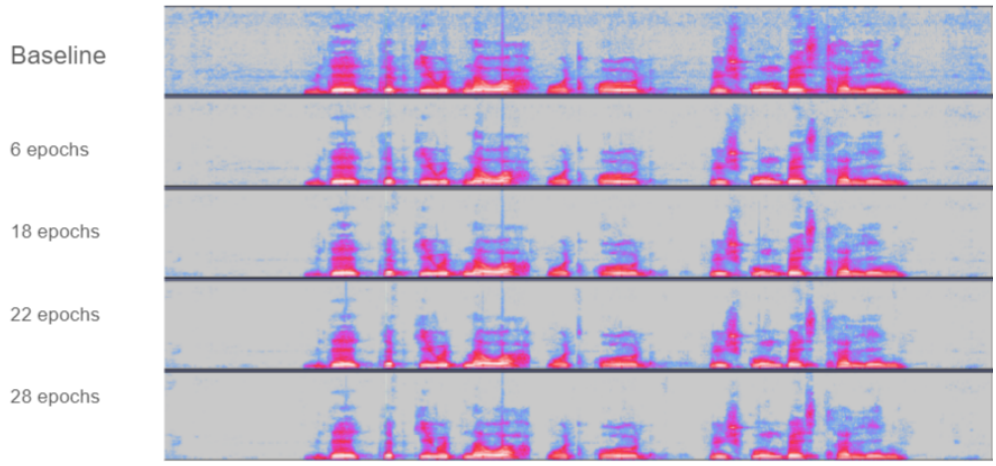

FIGURE 3.4 – Spectres d'un fichier audio bruité traité pour différentes sauvegardes du modèle

On remarque que le modèle apprend rapidement à éliminer les fréquences d'amplitude faible colorées en bleu. Selon l'évolution des epoch, le modèle semble apprendre à garder les fréquences qui entourent la parole (caractérisée en rouge et blanc puisque son amplitude est élevée). On observe aussi qu'il tolère de moins en moins les hautes fréquences.
Pour mieux comprendre ce que le modèle élimine, nous pouvons aussi visualiser la différence entre le fichier d'entrée et celui débruité. La figure 3.5 nous montre cette différence.

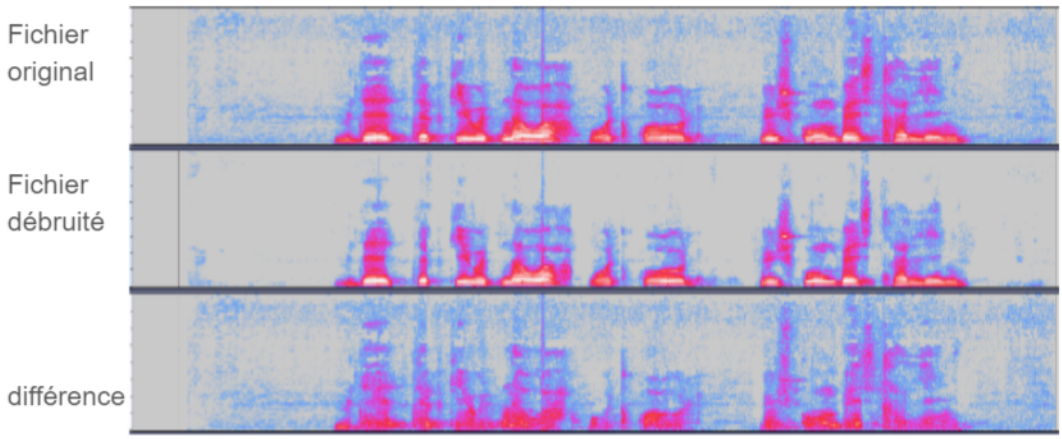

FIGURE 3.5 – Observation de ce que le modèle élimine

Le modèle semble donc éliminer une partie de la parole avec le bruit, cela peut s'expliquer par le fait que les données d'apprentissage contiennent une voix anglaise, donc que le système n'est pas adapté pour une parole en une autre langue. En entendant les fichiers débruités, on observe aussi que la qualité de la parole est détériorée.
Pour améliorer les performances du système, plusieurs approches sont possibles. L'une est de faire l'apprentissage sur un jeu de données restreint. C'est à dire, qui contient un unique type de bruit. En effet, il est possible que le modèle n'arrive pas a se généraliser dû à la différence entre les types de bruits qu'il essaye d'éliminer. Il devient moins précis et finis par détériorer de plus en plus la parole, ce qui résulte en une fonction de perte

qui oscille sans diminuer.

3.3 Training avec le second jeu de données

Le second jeu de données contient des fichiers bruités par un nombre plus restreint de bruits, la plus-part étant des bruits de parole de fond. La figure 3.6 montre l'évolution de la fonction de perte lors de l'apprentissage.

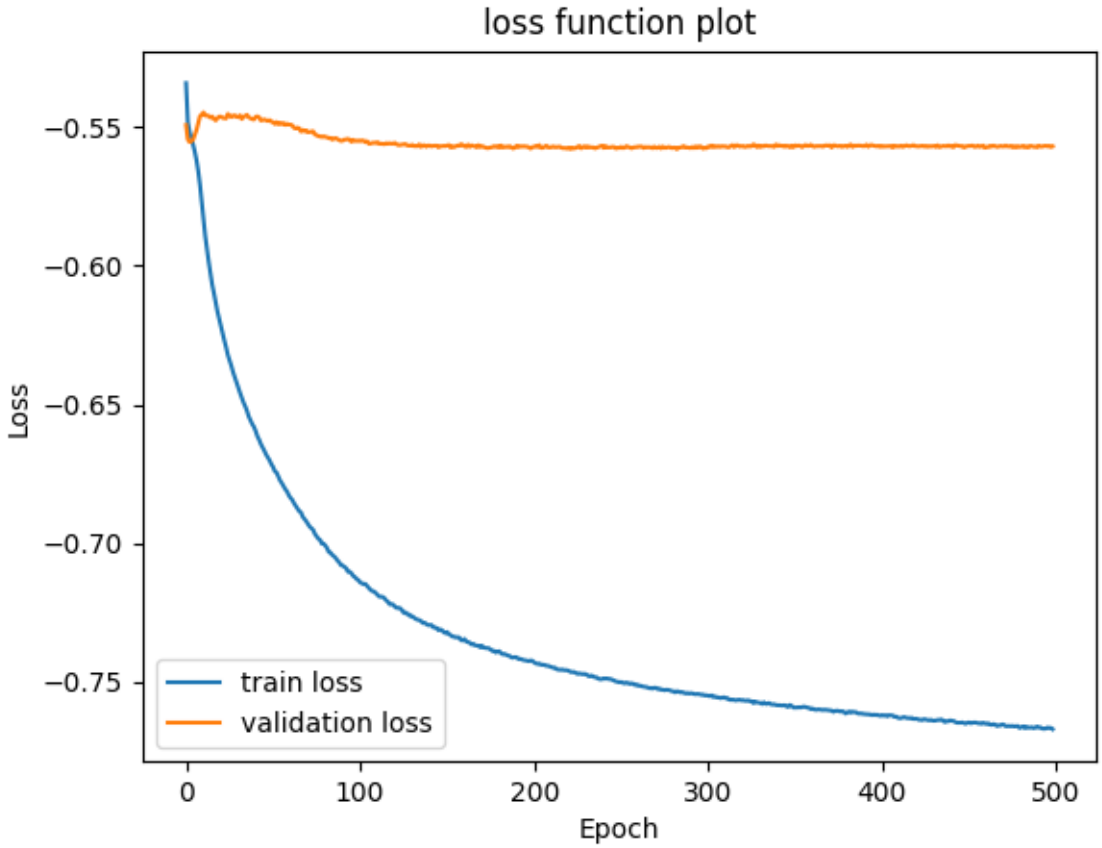

FIGURE 3.6 – La perte pour l'apprentissage sur le second jeu de données

On observe que l'ensemble d'apprentissage atteint une perte de -0,76. L'ensemble de validation diminue légèrement pour atteindre une perte de -0,55. Ces résultats sont légèrement meilleurs que ceux du premier jeu de données, mais cela peut s'expliquer par le fait que l'apprentissage a atteint plus d'epoch puisque le jeu de données contenait moins de fichiers. La figure 3.7 montre l'effet du débruitage sur un fichier.

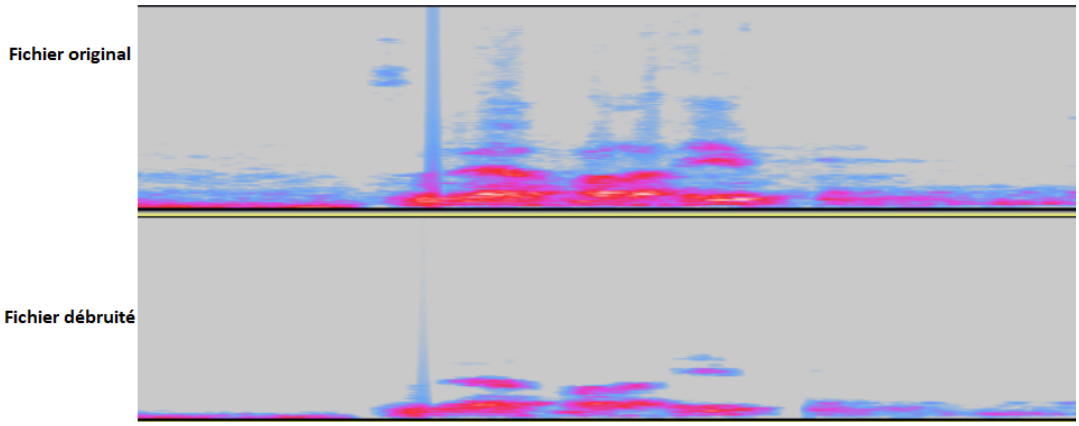

FIGURE 3.7 – Spectres d'un fichier audio bruité traité par le modèle dont l'apprentissage a été exécuté sur le second jeu de données

Si on observe la différence entre un fichier bruité et sa sortie du modèle, on remarque que la réduction de bruit est plus ciblée. Les fréquences contenant le son sont bien isolées mais certaines parties d'elles, notamment les fréquences les plus hautes, ont été éliminées. Cela résulte en une voix détériorée, mais la détérioration est moins forte que celle faite par le modèle dont l'apprentissage a été exécuté sur le jeu de données en anglais. La langue affecte donc la quailté du débruitage.

Conclusion et améliorations

Il est clair que la performance du modèle est loin d'être parfaite, le fait que la parole se détériore pose un problème pour plusieurs applications de débruitage. Mais il capable de débruiter un fichier quelconque sans savoir au-préalable le profil du bruit qu'il contient. Les mauvaises performances semblent provenir de l'apprentissage. En effet, le fait que l'apprentissage eu lieu sur Google colaboratory veut dire qu'il était limité par le temps des sessions. Les fonctions de perte n'ont donc pas atteint leurs minimum. La taille des batch était aussi limitée par la mémoire de la carte graphique. Il serait possible d'avoir une taille de batch plus importante en faisant l'apprentissage sur le processeur au lieu d'une carte graphique, mais cela résulterait en un temps d'apprentissage plus long. Il faudrait donc que l'apprentissage se fait sur une machine assez puissante sans avoir à faire recours à Google colaboratory. Il est aussi possible que une autre architecture de DC-U-NET résulte en des meilleurs performances, notamment une architecture avec plus de couches dans l'auto-encodeur comme la DC-U-NET20. Le jeu de données pourrait aussi être amélioré en ajoutant d'autres voix et d'autres amplitudes de bruits.

Conclusion Générale

Pour conclure sur ce projet de fin d'année. Nous avons étudié les concepts généraux de l'apprentissage automatique, afin de nous focaliser sur le cas du modèle deep-complex-u-network. La compréhension de ce modèle nécessitait la compréhension de différentes architectures de réseaux de neurones convolutifs notamment les auto-encodeurs et la u-network.

Nous avons aussi fait le lien entre le domaine de reconnaissance et amélioration de la parole avec la domaine de traitement d'images en observant que les spectres de la transformée de fourrier discrète d'un fichier audio peut être traitée comme une image. Et que donc le problème de débruitage se transformait en un problème de segmentation d'image. Nous avons aussi observé que ce problème nécessitait une architecture adaptée aux données complexes afin de bien estimer la parole depuis sa version bruitée, et notamment bien estimer sa phase.

L'implémentation de ce projet nous a permis de comprendre l'aspect informatique de l'apprentissage automatique. Cette compréhension nous a aussi permis de bien pouvoir interpréter les résultats de ce modèle et de l'opération de l'apprentissage, et de découvrir les différentes approches qu'on peut prendre afin d'améliorer ses résultats.

Le domaine de l'apprentissage étant vaste, ce projet représente une simple introduction sur un cas bien particulier de cette technologie. Mais cette introduction a suffit pour voir l'intérêt de cette technologie, et sa polyvalence.

Bibliographie

[1] Martın ABADI et al. « Tensorflow : A system for large-scale machine learning ». In : *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, p. 265-283.

[2] Saad ALBAWI, Tareq Abed MOHAMMED et Saad AL-ZAWI. « Understanding of a convolutional neural network ». In : *2017 International Conference on Engineering and Technology (ICET)*. Ieee. 2017, p. 1-6.

[3] Saad ALBAWI, Tareq Abed MOHAMMED et Saad AL-ZAWI. « Understanding of a convolutional neural network ». In : *2017 International Conference on Engineering and Technology (ICET)*. Ieee. 2017, p. 1-6.

[4] Marcin ANDRYCHOWICZ et al. « Learning to learn by gradient descent by gradient descent ». In : *arXiv preprint arXiv :1606.04474* (2016).

[5] Liang CHEN et al. « DRINet for medical image segmentation ». In : *IEEE transactions on medical imaging* 37.11 (2018), p. 2453-2462.

[6] Hyeong-Seok CHOI et al. « Phase-aware speech enhancement with deep complex u-net ». In : *International Conference on Learning Representations*. 2018.

[7] Arun Kumar DUBEY et Vanita JAIN. « Comparative study of convolution neural network's relu and leaky-relu activation functions ». In : *Applications of Computing, Automation and Wireless Systems in Electrical Engineering*. Springer, 2019, p. 873-880.

[8] Ian GOODFELLOW et al. *Deep learning*. T. 1. 2. MIT press Cambridge, 2016.

[9] Chuanmin JIA et al. « Layered image compression using scalable auto-encoder ». In : *2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*. IEEE. 2019, p. 431-436.

[10] Nikhil KETKAR. « Stochastic gradient descent ». In : *Deep learning with Python*. Springer, 2017, p. 113-132.

[11] Olaf RONNEBERGER, Philipp FISCHER et Thomas BROX. « U-net : Convolutional networks for biomedical image segmentation ». In : *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, p. 234-241.

[12] Shibani SANTURKAR et al. « How does batch normalization help optimization ? » In : *arXiv preprint arXiv :1805.11604* (2018).

[13] I STEPHEN. « Perceptron-based learning algorithms ». In : *IEEE Transactions on neural networks* 50.2 (1990), p. 179.

[14] C TRABELSI et al. « Deep complex networks. arxiv 2018 ». In : *arXiv preprint arXiv :1705.09792* ().

[15] Jianxin WU. « Introduction to convolutional neural networks ». In : *National Key Lab for Novel Software Technology. Nanjing University. China* 5 (2017), p. 23.

[16] Ruoyu YANG et al. « CNN-LSTM deep learning architecture for computer vision-based modal frequency detection ». In : *Mechanical Systems and Signal Processing* 144 (2020), p. 106885.

[17] Guoshen YU, Stéphane MALLAT et Emmanuel BACRY. « Audio denoising by time-frequency block thresholding ». In : *IEEE Transactions on Signal processing* 56.5 (2008), p. 1830-1839.

[18] Mengmeng ZHANG, Wei LI et Qian DU. « Diverse region-based CNN for hyperspectral image classification ». In : *IEEE Transactions on Image Processing* 27.6 (2018), p. 2623-2634.