PROJET - Deep Learning-based Speech Enhancement

Le but de ce projet est le débruitage de signal audio par deep learning.
Nous avons choisi de nous intéresser seulement aux spectrogrammes, et ainsi prendre le problème comme un problème de traitement d'image.

1 - Méthodes et conventions

Les réseaux de neurones que nous utiliserons seront donc des convolutional neural network (CNN), et plus particulièrement des CNN autoencoder. L'avantage de ce type de CNN est l'extraction des caractéristiques du signal, essentiel dans notre problème, ou il faudra différencier le signal du bruit.
Pour essayer de résoudre le problème, nous avons eu recours à 2 méthodes:

- **La méthode par masque (méthode dite *'mask'*)**:
  Cette méthode va consister à prédire un masque en sortie nous permettant ensuite de filtrer notre signal. Nous travaillerons avec 2 types de masque: Binaire (à valeur 0 ou 1) et Soft (à valeur comprises entre 0 et 1)
- **La méthode de prédiction du spectrogramme débruité en sortie (dite *'ori'*)**

Au cours de ce projet, nous avons utilisé diverses conventions pour nommer nos fichiers. On parlera de fichiers *'ori'* quand on parlera de fichiers originaux non bruités, de fichiers 'no' (noise only) quand on parlera de modèle de bruits utilisé pour le bruitage et de 'n' (noisy) pour les fichiers bruités.
Enfin, nos modèles seront définis et entraînés avec **tensorflow et keras**.

2 - Structuration du projet

Le projet est contenu dans un dossier appelé *SE_Deep.*
À la racine du dossier on retrouve :

- **le dossier notebooks**

Celui-ci contient différents notebooks montrant comment entraîner les modèles, comment les évaluer (evaluate.ipynb), comment les tester (test.ipynb) ou encore l'affichage de spectrogrammes avant/après.
Remarque: lorsqu'on parle d'évaluer des modèles on parle en termes de metrics, alors que tester a pour but l'écoute des résultats avant/après débruitage.

- **le dossier models**

Ce dossier contient les modèles entraînés et les courbes d'entraînement au format CSV.

- **le dossier train**

Il contient le dossier *ori* qui contenant les fichiers non bruités, et plusieurs autres dossiers contenant les fichiers bruité avec différents SNR.

- **le dossier test**

Il a le même contenu que le dossier train, mais pour des fichiers différents et en moins grande quantité (cf partie *3 - préparation des données*)

- **le dossier utils**

Il contient toutes nos fonctions et objets contenues dans 7 fichiers .py. On peut notamment citer data_transfo.py qui va contenir nos fonctions de passage audio à spectro et inversement, ou encore models.py qui va contenir nos trois différents modèles.

- **les fichiers gen_noise.py, babble_sub.wav et requirements.txt**

Le fichier gen_noise.py va nous permettre de bruiter nos signaux., requirements.txt contient les modules utiles pour faire fonctionner notre code.
Pour vérifier que vous les possédez, il suffit de lancer la commande *: pip install -r requirements.txt*

3 - Préparation des données

a) Répartition des fichiers

Parmi les 3720 fichiers audio au total, 3420 ont été utilisés pour l'entraînement et 300 pour les tests. Sur les fichiers de la base d'entraînement, on en prendra 75% pour “fit” notre modèle, et les 25% restant pour sa validation (vérification à la fin de chaque epoch des performances du modèles). Ce choix a été fait pour des raisons que nous développerons dans *4 - Modèles et générateurs*, mais sont bien sûr lié à la taille de nos modèles.
Ces fichiers ont ensuite été coupés à N = 49152 ech, étant la moyenne de longueur des fichiers audios. Ceci a été fait de manière à ce que le modèle ne soit pas entraîné sur du silence.

b) Bruitage des fichiers

Nous avons ensuite bruité les fichiers. Pour cela, nous avons utilisé le fichier babble_sub.wav, version sous echantillonée à 16kHz de babble.wav, afin d’avoir la même freq d’ech que les fichiers audio. Pour bruiter un son nous avons pris deux parties sélectionnées aléatoirement de babble_sub.wav, que nous avons recombiner pour former un bruit unique à chaque son .wav. (cf fonction random_part_noise dans ‘utils/noise.py’).
Ce bruit est ensuite normalisé en puissance selon le SNR voulu et additionné au signal non bruité, afin de créer un fichier bruité (cf fonction normalisation_and_sum_noise dans ‘utils/noise.py’).
A l’aide du gen_noise.py, nous avons réalisé ces deux étapes sur tous les fichiers dans les dossiers train et test.
Pour par exemple bruiter nos fichiers avec un SNR de -3dB, 0dB et +3dB, on fera appel au script comme ceci: *python3 gen_noise.py -3 0 3*.
Ce dernier créera dans test des dossiers 0dB, -3dB et +3dB contenant eux même les sons bruités (sous dossiers ‘noisy’) et le modèles de bruit (‘noise_only’) pour le SNR donné.

Remarque: Nous avons utilisé des conventions précises dans le nom de nos fichiers. Prenons par exemple le fichier original non bruité 1560.wav. La version bruité de ce son est appelée 1560_n.wav et est contenue dans ‘noisy’. Le modèle de bruit est appelé 1560_no.wav et est contenu dans ‘noise_only’.

c) Format et limitations de scipy.io.wavfile.write

Pour charger nos fichiers, nous avons utilisé la fonction read de scipy. Ce faisant, nous avons remarqué que la fonction chargeait les fichiers ori en vecteur de int16, avec des valeurs comprises entre -32768 et +32767. Cependant, la normalisation du bruit nous donne un vecteurs au format float32, puisqu’on prend une certaine proportion de la combinaison de babble_sub.wav (en fonction du SNR).
Cependant, la fonction write de scipy n’écrit des float32 qu’avec des valeurs comprises entre -1.0 et 1.0. Or si on ne fait qu’ajouter le fichier *ori* au *no*, on obtient un vecteur en float32 dépassant très largement ces bornes. Par conséquent, il nous faudra normaliser les vecteur *no* et *n* par *n*.max()-*n*.min(). Le SNR étant sensible au facteurs de normalisation, si on souhaite obtenir un fichier ori, on ne le chargera pas dans le dossier ori, mais on prendra les fichier *n* et *no* et on en fera la soustraction.

d) Spectrogrammes et normalisation

Nos spectrogrammes seront chargés au fur et à mesure (cf partie 4 - générateur) avec la fonction sound_to_images dans 'utils/data_transfo.py'.

Les paramètres utilisés sont: n_fft = 256 (longueur de fft); nperseg=256 (longueur de chaque segment);noverlap = 256/2=128 (dépassement de chaque fenêtre). Ces paramètres nous donneront des spectrogrammes de tailles 129x385 que nous donnerons ensuite en entrée du réseau.

Ces paramètres étant très classique en traitement de la parole, nous les avons conservé tout au long du projet, et nous avons privilégié la variation d'autres paramètres: le type de méthodes ('ori' ou 'mask') , le SNR ou encore la taille du réseaux de neurones (en nombre de paramètres).

Afin d'entraîner nos réseaux au mieux, nous chargeons nos spectrogrammes en dB, puis nous les normaliserons par norm = max-min, où max et min sont:

max = 73.94335 et min = -49.70791

Ces valeurs ont été obtenues en parcourant tous les spectrogrammes en dB de notre base d'entraînement et de test afin d'en retirer le max et min global.

Ce choix de normalisation a été choisi empiriquement, après plusieurs essais, c'est ce qui semble fonctionner le mieux.

4 - Modèles et générateurs

a) Modèles

Pour ce projet, nous avons construit 3 modèles: *model1*, *model2* et *unet*. Ces modèles sont des CNN autoencoders avec une *"skip architecture"* .Ce type d'architecture, comme en *fig1* a l'avantage de fournir un autre chemin pour la descente de gradient, et ainsi de permettre de converger plus facilement. Ce type d'architecture a fait ses preuves et est maintenant très utilisé (dans ResNet par exemple). Nos modèles contiennent beaucoup de paramètres, comme nous pouvons le voir *tab1*. Par conséquent, pour pouvoir les entraîner au mieux, nous devons disposer de beaucoup de fichiers audio.

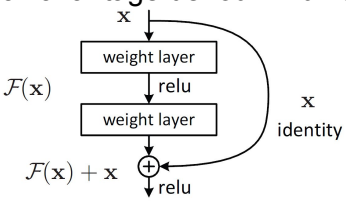

fig - skip architecture

	model1	model2	unet
Nb paramètres	641 185	7 037 313	31 072 913

Tab1 - Nombres de paramètres pour les différents réseaux

Au niveau des réseaux, on a choisi une taille de kernel de 3x3 pour toutes les différentes couches, et des fonctions d'activation 'relu' sur toutes les couches sauf la dernière.

Pour la dernière couche, la fonction d'activation dépend de la méthode utilisée. En effet, si on choisit la méthode 'mask', la sortie devra être comprise entre 0 et 1. Par conséquent on utilisera la fonction d'activation sigmoid qui fournie une sortie entre 0 et 1. Si la méthode est 'ori', on utilisera la fonction d'activation tanh qui fournit des valeurs comprise entre -1.0 et 1.0 et c'est ce qu'on cherche à avoir (*cf 3)c)*).

On remarquera que les spectrogrammes d'entrée ont une taille impaire: 129*385.

Par conséquent, lors des operations de MaxPool classiques dans un CNN autoencoder,

De plus, nous avons choisi d'utiliser des BatchNormalization, qui sont des couches qui réalisent des normalisation par batch à l'intérieur du réseau afin de pouvoir converger plus facilement.

Nous avons aussi utilisé des Dropouts qui sont des couches permettant de mettre à 0 une certaine proportion des neurones afin d'éviter tout surajustement.
Enfin, nous avons utilisé des “callbacks” qui sont des fonctions que l’ont donne à model.fit (méthode de tensorflow permettant d’entrainer le modèle ‘model’) pendant l’entrainement. Nous avons utilisé le callback appelé ModelCheckpoint qui nous permettra de sauvegarder le modèle entraîné que lorsque l’erreur de validation diminuera. Par exemple, si on entraîne un modèle donné pendant 100 epoch, mais que passé l’epoch 5 l’erreur d'entraînement diminue, mais celle de validation augmente (cas de surajustement), alors à la fin des 100 epochs, le modèle enregistré sera celui obtenu à l’epoch 5. C’est donc une sécurité supplémentaire pour éviter tout surajustement.
Comme nous l’avons dit en 3)a), en entraînement, nous disposons de 3420 fichiers. Nous n’allons bien sûr par charger une matrice de 3420 spectrogrammes (ce qui ferait 3420*129*385). Nous utilisons un générateur.

b) Générateur

Un générateur est un objet qui nous permettra de fournir au réseau des données au fur et à mesure par batch. On définit batch=5, le générateur donnera donc au fur et à mesure au réseau des tableaux de 5*129*385*1 ce qui est beaucoup plus supportable.
Notre générateur sera donné à notre model.fit (utilisé pour entraîner nos modèles) mais doit respecter des contraintes très précises.
D’après la documentation de keras, model.fit accepte: *“A generator or keras.utils.Sequence returning (inputs, targets) or (inputs, targets, sample_weights)”*
Le “generator” dont parle la documentation est un générateur python, une sorte de fonction qui se termine par yield et qui fournit au fur et à mesure des données. Nous avons essayé une implémentation de ce type de generator, mais sans succès. Par conséquent, nous avons implémenté la deuxième solution, un objet *keras.utils.Sequence*
Cet objet doit contenir 5 méthodes afin de pouvoir être utilisé par notre modèle.fit:

- **Une méthode __init__**
  Classique pour tout objet en POO, pour l’initialisation des variables.
- **Une méthode __len__**
  Qui nous fournira le paramètre steps_per_epoch qui donnera le nombre de total de batch par epoch à model.fit. Classiquement, on choisira (nb_total_spectro/taille_de_la_batch) afin que le model voit tout le dataset à chaque epoch
- **Une méthode __gen_data**
  Qui donnera par batch des tuple (X,Y), où X est le spectro bruité et Y la sortie (le spectro débruité si la méthode est ‘ori’ ou le masque si la méthode est ‘mask’)
- **une méthode on_epoch_end**
  Qui sera appelée à chaque fin d’epoch, et qui aura pour but de mélanger les échantillons, afin de ne jamais avoir les mêmes epoch d’une à la suivante.
- **Une méthode __getitem__**
  Qui appellera __gen_data pour obtenir les tuple (X,Y) et qui les fourniera à model.fit

Cet implémentation a été réalisé à l’aide de 3 sources: deux documentations et un tutoriel:
https://www.tensorflow.org/api_docs/python/tf/keras/utils/Sequence
https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit
https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly
Notre implémentation de notre générateur est contenue dans *utils/generators.py*.

c) Metrics

Afin de procéder à l'entraînement de nos modèles, il nous faut choisir des metrics.
Nous avons choisis ces trois fonction:

- Mean Square Error (MSE)

```
MSE = square(y_true - y_pred)
```

- Mean Absolute error (MAE)

```
MAE = abs(y_true - y_pred)
```

- Signal-to-Noise ratio (SNR)
- Binary Accuracy (BA)

MSE va être utilisé comme fonction de coût pour l'entraînement de tous nos modèles (peu importe la méthode).
MAE va être utilisé comme metric lors de l'entraînement de tous les modèles, pour les méthodes ‘ori’ et les masques soft.
BA va être utilisé pour l'entraînement des tous les modèles seulement pour les masques binaires
Enfin, on notera que dans la partie 5 - Résultat, MSE, MAE et SNR seront utilisé dans l'évaluation des performances de notre modèles entre le spectrogramme bruité d’entrée et celui obtenu après débruitage (même pour les méthodes de masques, on multipliera le masque par le spectrogramme d’entrée, puis on comparera le résultat à celui en entrée).

d) Entrainement

Pour entraîner notre modèle, nous avons choisi un learning rate (vitesse de descente de gradient) de 1e-3 qui est très classique et flexible pour ce genre d’application de traitement d’image. Au niveau du nombre d’epochs, nous n’avons pas trouvé de nombre idéal qui fonctionne pour chaque SNR ou méthode. Par conséquent, il sera variable au cas par cas. Cependant, une technique semble avoir fait ses preuves empiriquement pour faire converger nos modèles: commencer par entraîner avec des SNR bas (0dB) pendant une dizaine-vingtaine d’epochs puis baisser le SNR progressivement jusqu’à -8dB par pallier de -3dB en entraînant à chaque fois pendant 5 ou 7 epochs.
Voici par exemple, les résultats d'entraînement de unet pour la méthode “ori” :

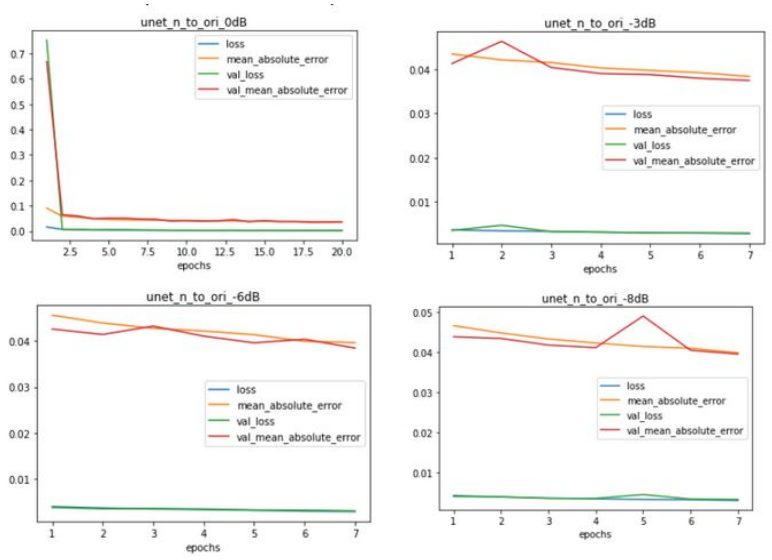

fig - Résultat entraînement d’unet avec la méthode ori

Les modèles présents dans le rendus seront bien sûr les plus aboutis et par conséquent les modèles entraînés sur du -8dB. Cependant, libre à l'utilisateur de réentraîner les modèles avec le SNR de son choix à l'aide du notebook *'notebooks/train.ipynb'* et l'évaluer ou le tester avec les notebooks *evaluate.ipynb* et *test.ipyn*b. Les notebooks ont été pensés pour.

5 - Résultats

Au vu des nombreux modèles et différentes variantes possibles en fonction de la méthode utilisée, nous montrerons dans cette partie le modèle ayant les meilleures performances: unet. Les résultats des autres réseaux sont visible dans le notebook *"notebooks/evaluate.ipynb"*

a) Méthode **'ori'**

	3dB	0dB	-3dB	-6dB	-8dB
SNR (Gain)	4.0	6.7	8.4	12.2	14.7
MSE	0.03067	0.07424	0.01937	0.01354	0.00953
MAE	0.36424	11.14397	0.09527	0.0383	0.00032

Tab - Résultats pour unet et avec la méthode ori sur l'ensemble de test

Voici deux exemples de spectrogramme débruité avec cette méthode et pour un SNR d'entrée de -6dB:

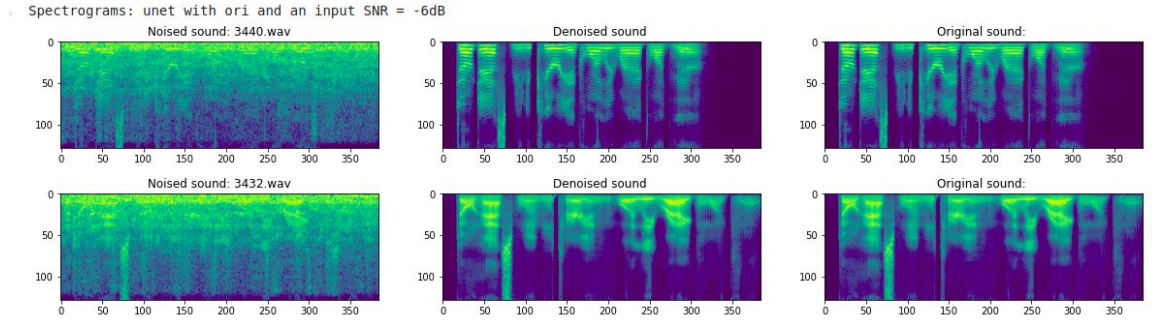

fig - Résultat de débruitage de spectro avec d'unet avec la méthode ori

De plus, voici un exemple de formes d'onde débruité avec cette méthode et pour un SNR d'entrée de -6dB:

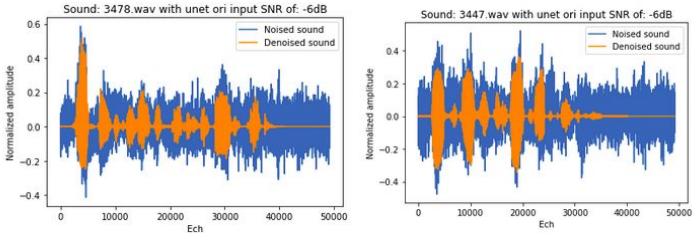

fig - Résultat des formes d'onde pour unet, la méthode ori et un SNR d'entrée de -6dB

Enfin, on retrouvera également dans le dossier “audio_test” à la racine de SE_Deep plusieurs fichiers débruité avec un SNR d’entrée de -6dB.
Néanmoins, il est parfaitement possible d’écouter le résultat du débruitage dans le notebook *notebooks/test.ipynb*.

b) Méthode masque binaire

Nous allons maintenant observer les résultats avec la méthodes “masque binaire”

	3dB	0dB	-3dB	-6dB	-8dB
SNR (Gain)	8.4	10.1	11.1	12.6	13.5
MSE	0.00909	0.00922	0.01131	0.01139	0.01136
MAE	0.00023	0.00027	0.00038	0.00042	0.00041

Tab - Résultats pour unet et avec la méthode masque binaire sur l’ensemble de test

Voici un exemple de spectrogramme débruité avec cette méthode et pour un SNR d’entrée de -6dB:

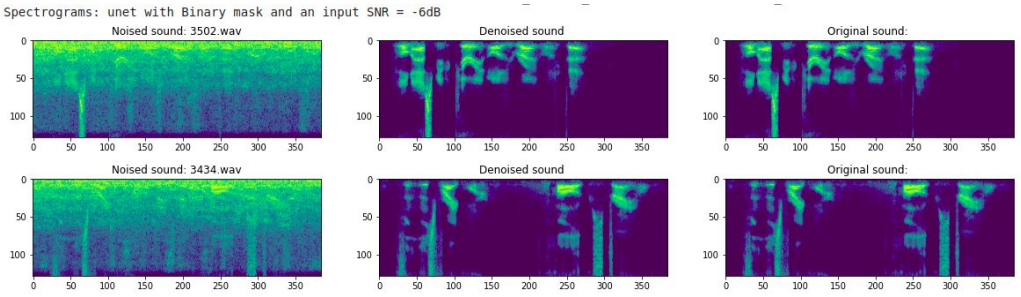

fig - Résultat de débruitage de spectro avec d’unet avec la méthode masque binaire (snr=-6dB)

De plus, voici un exemple de forme d’ondes débruités avec cette méthode et pour un SNR d’entrée de -6dB:

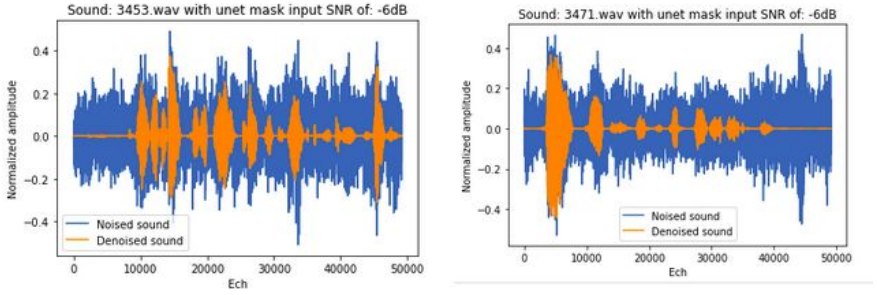

fig 5 - Résultat des formes d’onde pour unet, la méthode masque binaire et un SNR d’entrée de -6dB

c) Méthode masque Soft

	3dB	0dB	-3dB	-6dB	-8dB
SNR (Gain)	9.4	10.5	12.1	13.0	14.3
MSE	0.00795	0.00913	0.00999	0.01093	0.01046
MAE	0.00018	0.00025	0.00031	0.00036	0.00036

Tab5 - Résultats pour unet et avec la méthode masque soft sur l'ensemble de test

Voici un exemple de spectrogramme débruité avec cette méthode et pour un SNR d'entrée de -6dB:

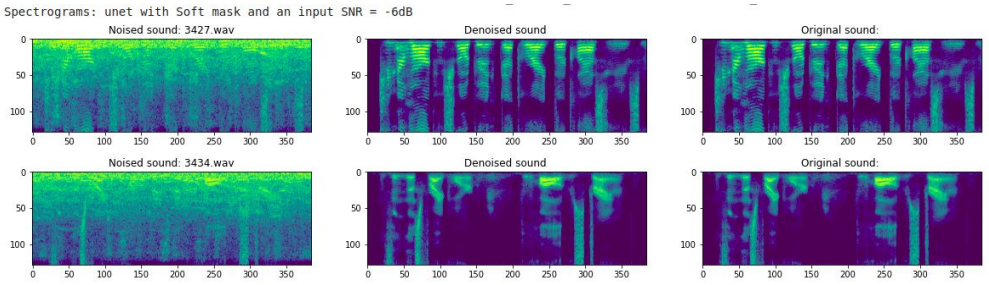

fig - Résultat de débruitage de spectro avec d'unet avec la méthode masque soft (snr=-6dB)

De plus, voici un exemple de forme d'ondes débruités avec cette méthode et pour un SNR d'entrée de -6dB:

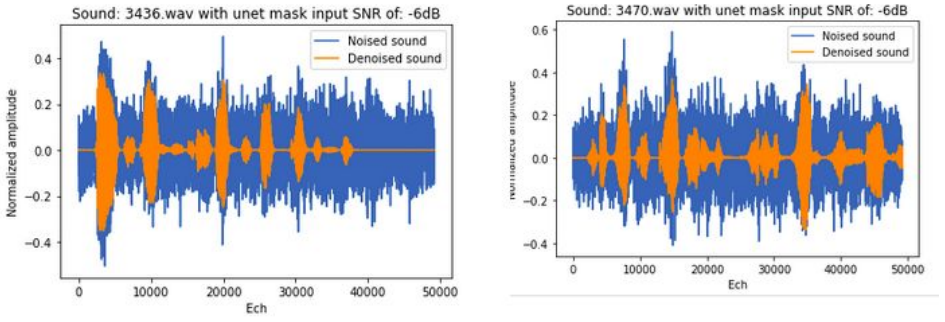

fig 5 - Résultat des formes d'onde pour unet, la méthode masque soft et un SNR d'entrée de -6dB

6 - Conclusion:

Au vu de nos résultats, on peut en conclure que, pour des SNR haut, c'est la méthode "ori" qui semble fonctionner le mieux: nous offrir le gain en SNR le plus important.
Cependant, pour des SNR bas, il serait préférable de privilégier la méthode par masque, et notamment celle du masque Soft.