Faculté des Sciences
Aix*Marseille Université

Compte rendu du mini projet d'image et IA

Saïda GUEZOUI
M2 MAS DS

Année universitaire
2021 / 2022

Encadrant : Frederic RICHARD

Mail universitaire : saida.guezoui@etu.univ-amu.fr

Contents

1 Introduction

Dans ce projet, nous allons utiliser un ensemble de données d'électrocardiogramme (ECG) afin de classifier trois groupes de personnes : celles souffrant d'arythmie cardiaque (ARR), d'insuffisance cardiaque congestive (CHF) et de rythme sinusal normal (NSR).

Pour celà, nous allons utiliser des méthodes de classification telles les réseaux de neurones.
En revanche, comme nous avons un signal en 1D, une approche consistant à utiliser le scaleogramme qui permet de représenter en 2D des caractéristiques extraites en 1D sera mise en oeuvre pour avoir une entrée de notre réseau de neurones, autrement dit une entrée sous forme d'une image.

Au cours de ce projet, deux types de réseaux de neurones ont été testés pour ce problème de classification : un réseau de neurones simple composé des couches denses et un réseau de neurones à convolution (CNN) qu'on a va définir plus en détail prochainement.

2 Présentation et chargement des données

Nous rappelons qu'un ECG est une représentation graphique de l'activité électrique du cœur,autrement dit, c'est un signal électrique qui varie dans le temps. Il fournit des informations très vitales sur un large éventail de troubles cardiaques en fonction des déviations du signal normal de l'ECG.
Notre jeu de données est composé de 162 électrocardiogrammes (observations) répartis ainsi :

- ARR : 96 observations correspondant aux patients atteints d'arythmies cardiaques
- CHR : 36 observations correspondant aux patients touchés par des insuffisances cardiaques congestives
- NSR : 36 observations correspondants aux patients représentant des rythmes sinusoidaux normaux.

Le jeu de données peut être téléchargé sur github sous le nom du fichier EVGData.zip. Les données se trouvent dans le fichier matlab ECGData.mat. Pour les importer, on utilise le package "mat4py" de python. Les ECG ont été échantillonnés à 128 Hz (128 échantillons par seconde) et comportent 65536 échantillons (soit 512 secondes d'enregistrement).

Avant de commencer, nous importons les bibliothèques pour charger notre jeu de données, pour le calcul scientifique (scalogramme, transformée de Fourier...) et pour le deep learning (réseau de neurones simple et convolutif).

```
import numpy as np
from mat4py import loadmat
from matplotlib import pyplot as plt

import scipy
import scipy.fft
from scipy.fftpack import fft, rfft, fftshift, fftfreq, ifft, irfft

import pywt
import scaleogram as scg

from sklearn import preprocessing
from sklearn.model_selection import train_test_split

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D
```

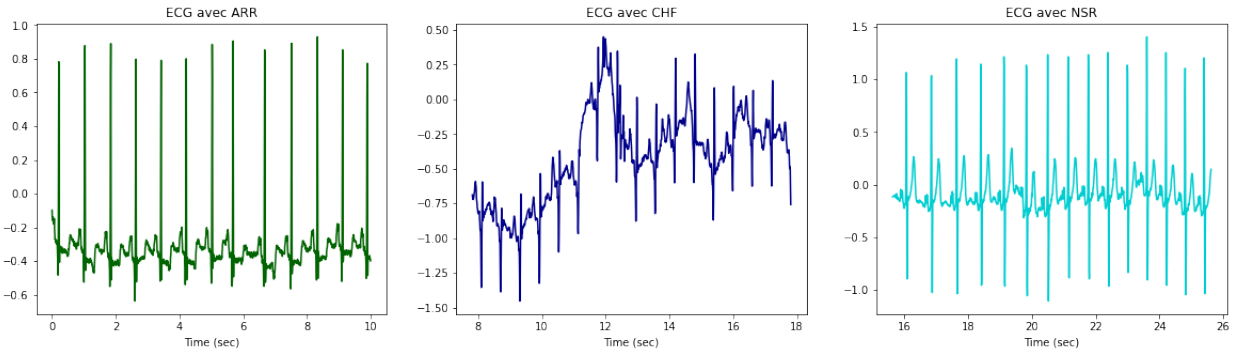
On commence par importer nos données. Pour ce faire, on utilise la fonction *loadmat* de la bibliothèque *mat4py* qui nous renvoi le jeu de données des électrocardiogrammes des patients. On décide ensuite de séparer nos données en deux ensembles : l'ensemble des électrocardiogrammes (dans la liste ecg) et l'ensemble des étiquettes associées (dans la liste cl).

```
data = loadmat('ECGData/ECGData.mat')

M = len(data['ECGData']['Data'])

ecg = [] ; cl = []
for i in range(0, M):
    ecg.append(data['ECGData']['Data'][i])
    cl.append(data['ECGData']['Labels'][i])
```

3 Visualisation des données

Dans cette partie, nous allons visualiser l'allure de l'électrocardiogramme pour les trois catégories de patients (ARR, CHF et NSR).

```
nbr_freq = len(ecg[0])
sampling_freq = 128
time = np.arange(nbr_freq) / sampling_freq

plt.figure(figsize=(20,5))

plt.subplot(1,3,1)
plt.plot(time[0:1280], ecg[0][0:1280], 'darkgreen')
plt.title(f'ECG avec {cl[0][0]}')
plt.xlabel('Time (sec)')

plt.subplot(1,3,2)
plt.plot(time[1000:2280], ecg[100][1000:2280], 'darkblue')
plt.title(f'ECG avec {cl[100][0]}')
plt.xlabel('Time (sec)')

plt.subplot(1,3,3)
plt.plot(time[2000:3280], ecg[150][2000:3280], 'darkturquoise')
plt.title(f'ECG avec {cl[150][0]}')
plt.xlabel('Time (sec)')

plt.show()
```



4 Détection des complexes QRS

Le signal ECG est caractérisé par cinq pics et vallées marqués par les lettres P, Q, R, S, T. Dans certains cas, on utilise également un autre pic appelé U. Le complexe QRS est la forme d'onde la plus importante du signal

électrocardiographique (ECG), avec une durée normale de 0,06 seconde à 0,1 seconde. Les performances du système d'analyse de l'ECG dépendent de la détection précise et fiable du complexe QRS, ainsi que des ondes T et P.
Cependant, comme les complexes QRS ont une morphologie variable dans le temps, ils ne sont pas toujours la composante la plus forte du signal ECG. Il existe de nombreuses sources de bruit dans un environnement clinique comme l'interférence des lignes électriques, le bruit des contractions musculaires, le mauvais contact des électrodes, les mouvements du patient dus à la respiration qui peuvent dégrader le signal ECG.

Les algorithmes de détection des QRS peuvent être classés en trois catégories :

- Les algorithmes non syntaxiques, qui prennent beaucoup de temps

- Les algorithmes syntaxiques qui sont largement utilisés

- Les algorithmes hybrides.

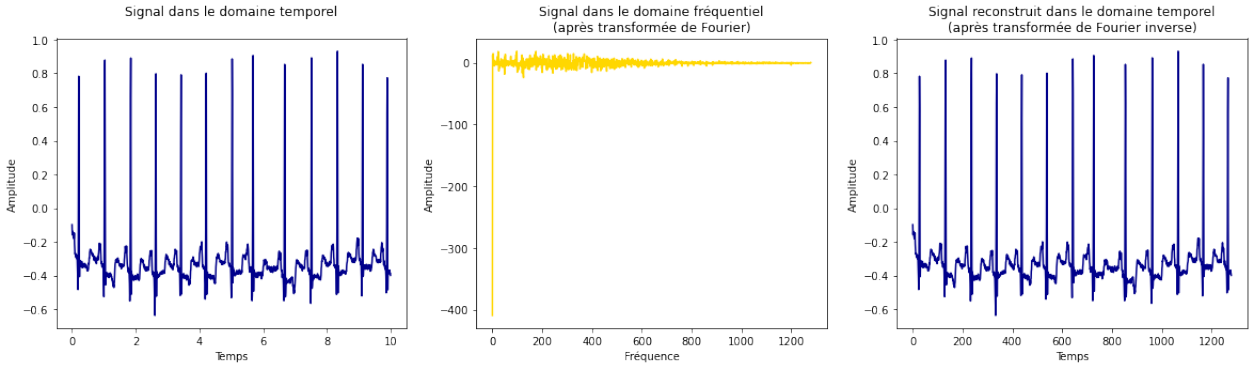
Les algorithmes précédemment appliqués utilisent généralement un filtrage non linéaire pour détecter les complexes QRS en utilisant le seuillage, ou de l'intelligence artificielle en utilisant les modèles de Markov cachés ou bien les techniques de prédiction récursive dans le temps.

Concernant l'analyse en ondelettes, c'est est un outil mathématique très prometteur, un " microscope mathématique " qui donne une bonne estimation de la localisation temporelle et fréquentielle.

On s'accorde à dire que l'essentiel du complexe QRS réside dans une bande de fréquence qui varie selon les auteurs. Aussi pour mettre en évidence le complexe QRS et atténuer les bruits et composantes de nuisance, on réalise souvent un prétraitement qui consiste à filtrer le signal ECG avec un filtre passe-bande dans la bande de fréquence de l'ECG.

La transformée de Fourier permet d'analyser un signal dans son domaine fréquentiel, autrement dit, assure le passage du temporel au fréquentiel. Il existe aussi une transformée de Fourier inverse qui permet de reconstruire notre signal dans son domaine temporel. Le code suivant permet d'illustrer le résultat de ces 2 transformations.

```
# On test la transformation de Fourier et son inverse
signal_time     = np.array(ecg[0][0:1280])
signal_freq     = rfft(signal_time)
signal_time_new = irfft(signal_freq)

plt.figure(figsize=(20,5))

plt.subplot(1,3,1) ; plt.plot(time[0:1280], ecg[0][0:1280], "darkblue")
plt.xlabel("Temps"); plt.ylabel("Amplitude"); plt.title("Signal initial")

plt.subplot(1,3,2) ; plt.plot(signal_freq,color = "gold")
plt.xlabel("Fréquence"); plt.ylabel("Amplitude"); plt.title("Signal dans le domaine fré
    quentiel ")

plt.subplot(1,3,3) ; plt.plot(signal_time_new, "darkblue")
plt.xlabel("Temps"); plt.ylabel("Amplitude"); plt.title("Signal retrouvé avec la ifft")
```

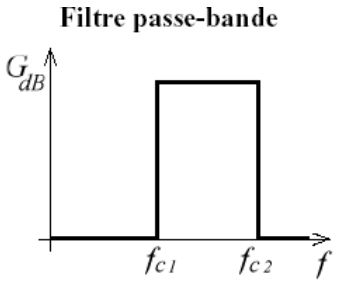


Le résultat montre que la transformée de Fourier inverse arrive à bien reconstruire le signal dans le domaine temporel.

4.1 Réalisation d'un filtrage passe-bande

On peut considérer un signal comme l'évolution d'une grandeur - l'amplitude de l'impulsion électrique du coeur dans notre cas - au cours du temps. Comme tout signal capté par un instrument de mesure, de nombreux bruits peuvent biaiser l'analyse. En effet, l'instrument peut aussi capter des signaux provenant d'une activité musculaire autre que celle du coeur ou des interférences avec d'autres appareils électriques.
On peut néanmoins limiter les bruits en filtrant le signal selon une certaines plage de fréquences. Comme nous l'avons vu dans la partie précédente, on utilise la transformée de Fourier qui permet une analyse fréquentielle du signal.

Dans cette partie, nous allons construire un filtrage passe-bande. Il est appliqué sur le signal dans le domaine fréquentiel. L'objectif est de garder seulement une plage fréquentielle. Comme le montre l'illustration ci-dessus, seulement les fréquences allant de f_{c1} à f_{c_2} seront gardées, les autres seront coupées.



Revenons à l'étude de notre signal. Par le biais des ECG f de la base de données, l'activité électrique du coeur g est mesurée sur une période de temps T (en seconde) à une fréquence f_0 de 128 Hz (128 échantillons par seconde). Soit $N = f_0 T$ le nombre total d'échantillons. Pour $n = 1, \cdots, N$, on a

$$f[n] = g\left(\frac{n}{f_0}\right).$$

Par ailleurs, en supposant que g est de carré intégrable sur $[0, T]$, on a

$$g(x) = \sum_{n\in\mathbb{Z}} \hat{g}_n e^{i\frac{2\pi}{T}nx} \quad \text{avec} \quad \hat{g}_n = \int_{[0,T]} g(y) e^{-i\frac{2\pi}{T}ny} dy.$$

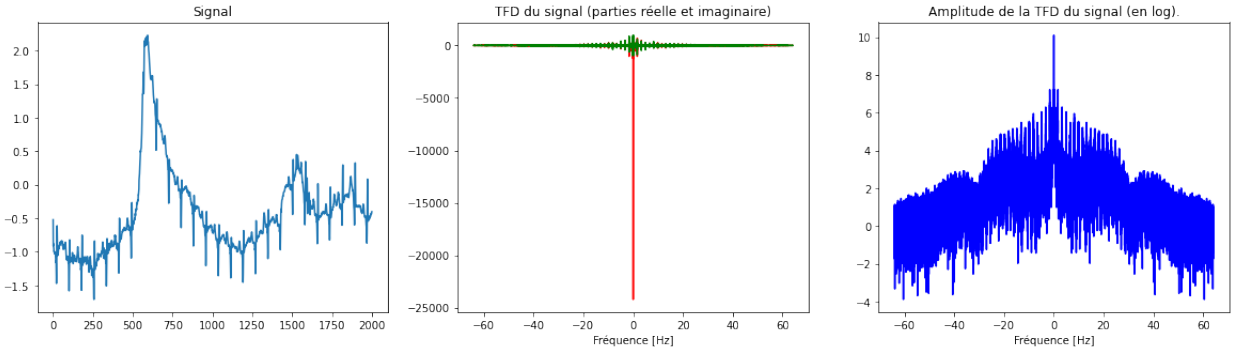
Le coefficient $\hat{g}_n$ de Fourier de g peut s'approcher par

$$\hat{g}_n \simeq \frac{T}{N}\sum_{k=0}^{N-1} g\left(\frac{kT}{N}\right) e^{-i\frac{2\pi}{T}\frac{kT}{N}n} = \frac{T}{N}\sum_{k=0}^{N-1} f[k]e^{-i\frac{2\pi kn}{N}} = T\hat{f}[n],$$

où $\hat{f}$ est la transformée de Fourier discrète (TFD) de f. Autrement dit, à un facteur près, $\hat{f}[n]$ approche le coefficient de Fourier de g correspondant à la fréquence $w_n = \frac{n}{T} = \frac{n}{N} f_0$.

La TFD d'un signal peut se calculer en Python à l'aide de la méthode fft du package scipy.

```
f0= 128
f = np.array(ecg[100])  # Signal ECG sous forme de ndarray.
fh = fft(f)  # TFD du signal f.
freq = fftfreq(f.size, d=1/f0)  # Fréquences associées aux coefficients de la TFD.

plt.figure(figsize=(20,5))
# Visualisation d'un extrait du signal.
plt.subplot(1,3,1)
plt.plot(f[0:2000])
fig1 = plt.title('Signal')

# Visualisation de la TFD en centrant sur la fréquence 0.
plt.subplot(1,3,2)
plt.plot(fftshift(freq), fftshift(fh.real), 'r', fftshift(freq), fftshift(fh.imag), 'g')
fig1 = plt.title('TFD du signal (parties réelle et imaginaire)')
plt.xlabel('Fréquence [Hz]')

plt.subplot(1,3,3)
plt.plot(fftshift(freq), fftshift(np.log(np.absolute(fh))), 'b')
fig2 = plt.title('Amplitude de la TFD du signal (en log).')
plt.xlabel('Fréquence [Hz]')


# Reconstruction à l'identique du signal d'origine f en appliquant la TFD inverse.
fs = ifft(fh)
```



Pour appliquer un filtre de passe-bande, on doit se fixer un seuil bas et un seuil haut. Dans le signal ECG, on élimine les composantes du spectre dont les fréquences sont, en valeur absolue, au-dessus du seuil bas ou en dessous du seuil haut. Le code suivant est un exemple d'application du filtre passe-bande sur un signal avec la transformée de Fourier et une seuil entre 5Hz et 15Hz.

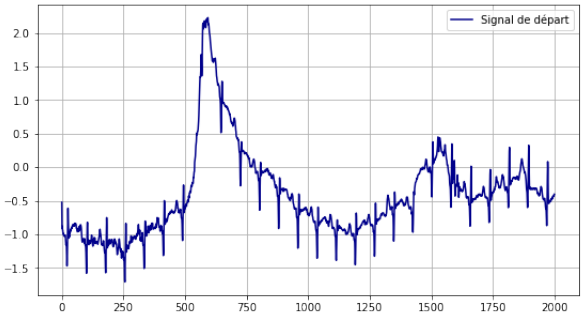
La fonction de transfert utilisée pour un filtre passe bande d'un second ordre est définie par :

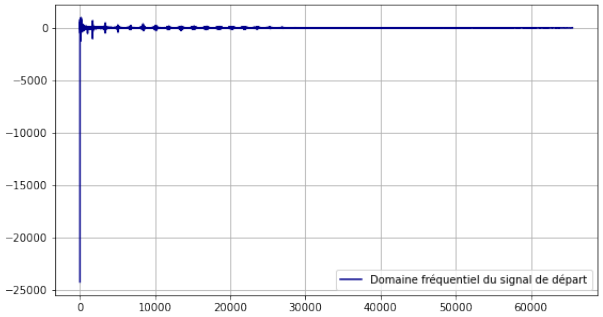
$$h(jw) = \frac{A_0}{1 + jQ(x - \frac{1}{x})}$$

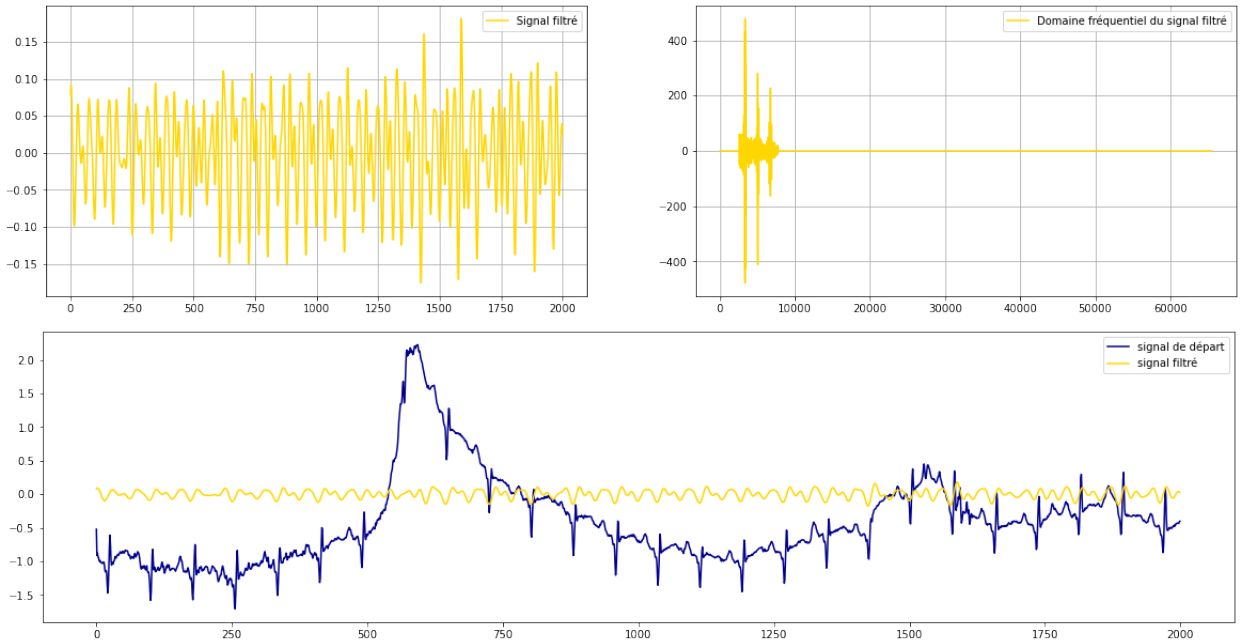
Avec A_0 est le coefficient de gain, $x = \frac{w}{w_0}$ correspondant à une variable réduite et Q est le facteur de qualité (d'après wikipédia : https://fr.wikipedia.org/wiki/Filtre_passe-bande). Le code suivant montre l'application de notre filtre.

```
# Elimination des fréquences
sb = 5 ; sh = 15
signal = np.array(ecg[100])

freq_signal = rfft(signal)
freq = fftfreq(signal.size, d = 1/128)

cut_freq_signal = freq_signal.copy()
cut_freq_signal[freq <=  5] = 0
cut_freq_signal[freq >= 15] = 0

cut_signal = irfft(cut_freq_signal)

# Affichage des résultats
plt.figure(figsize = (20,5))
plt.subplot(1,2,1) ; plt.plot(signal[:2000])
plt.subplot(1,2,2) ; plt.plot(freq_signal)
plt.grid()
plt.show()

plt.figure(figsize = (20,5))
plt.subplot(1,2,1) ; plt.plot(cut_signal[:2000])
plt.subplot(1,2,2) ; plt.plot(cut_freq_signal)
plt.grid()
plt.show()

plt.figure(figsize = (20,5))
plt.plot(signal[:2000])
plt.plot(cut_signal[:2000])
plt.grid()
plt.show()
```

Les graphiques suivants sont les résultats du code. La première ligne représente le signal de départ dans son domaine temporel (à gauche) et dans son domaine fréquentiel (à droite). Dans la seconde ligne, on observe le signal après l'application du filtre dans son domaine temporel et dans son domaine fréquentiel une nouvelle fois. Enfin, le dernier graphique permet de comparer entre le signal de départ (en bleu) et le signal filtré (en orange).



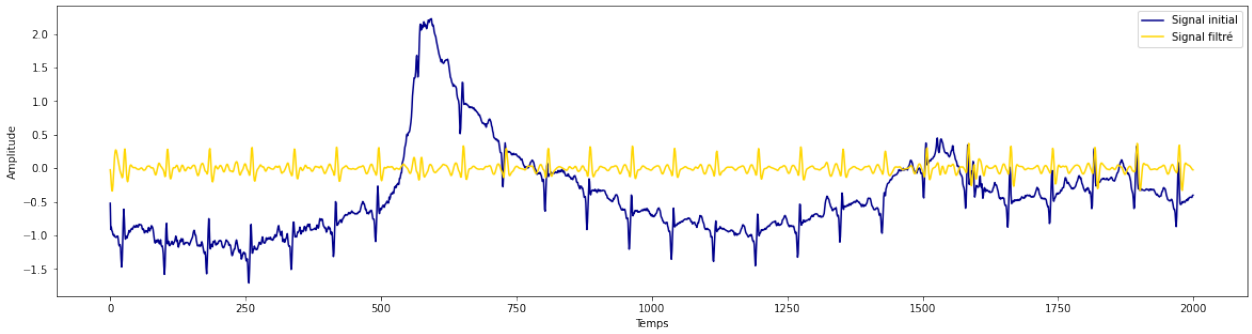




4.2 Détection avec un filtre de Butterworth

On peut améliorer le qualité du filtrage avec le filtre de Butterworth qui est aussi un filtre linéaire, mais avec un gain aussi constant que possible dans sa bande passante. Cela a pour effet d'éviter un décalage entre le signal de départ et le signal filtré.

```
signal_ = np.array(ecg[100])

sos = signal.butter(2, [5, 15], 'bp', fs = 128, output='sos')
filtered = signal.sosfilt(sos, signal_[:2000])


plt.figure(figsize = (20,5))
plt.plot(signal_[:2000], "darkblue", label="Signal initial")
plt.plot(filtered, "gold",label = "Signal filtré")
plt.xlabel("Temps")
plt.ylabel("Amplitude")
plt.legend()
plt.show()
```



5 Classification des ECG

Nous rappelons que les données contiennent des ECG de trois groupes de personnes, ARR, CHF, NSR, les deux premiers correspondent à des maladies et le troisième groupe sont des personnes en bonne santé. Pour classer ces groupes, nous proposons dans un premier temps,la transformée en ondelettes qui permet d'avoir une représentation des informations fréquentielle et temporelle également, contrairement à la transformée de Fourier qui donne uniquement la représentation fréquentielle.

5.1 Scalogramme

On commence par définir la transformée en ondelettes d'un point de vue mathématique :
Etant donnés une fonction ϕ, appelée ondelette, et un signal f, tous deux dans $L^2(\mathbb{R})$, on définit, pour $\alpha \in \mathbb{R}^*$ et $u \in \mathbb{R}$, la transformée en ondelette continue par

$$\tilde{f}_\psi(t,\alpha) = \int_{\mathbb{R}} f(u)|\alpha|^{\frac{1}{2}}\overline{\psi}(\alpha(u-t))du = \langle f, |\alpha|^{\frac{1}{2}}\psi(\alpha(\cdot - t))\rangle.$$

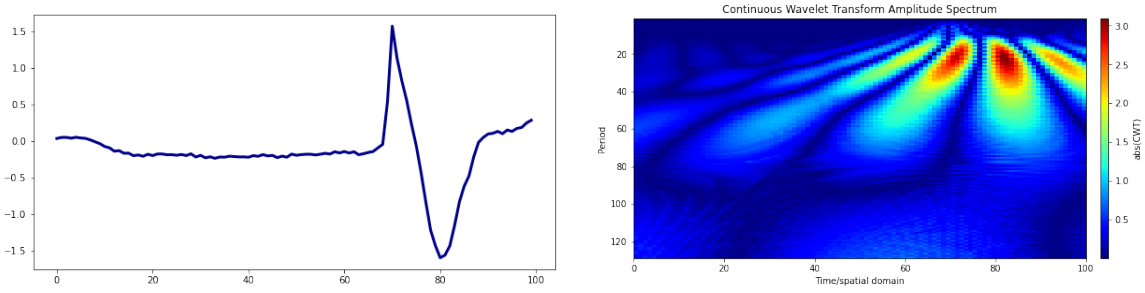
Cette transformée mesure une similarité de f avec le motif ψ dilaté de $|\alpha|$ et positionnée dans le voisinage de t.
La représentation du graphe $\{(t,\alpha), (|\tilde{f}_\psi(u,\alpha)|^2, u, \alpha)\}$ s'appelle un scalogramme. L'information véhiculée par le scalogramme peut s'analyser pour chaque position t (abscisse) et chaque échelle α (axe des ordonnées).

On peut calculer le scalogramme d'un signal en utilisant l'une des méthodes suivantes en Python :

- la méthode cwt du package scipy.signal,
- le package pywt avec sa documentation,
- le package scaleogram basé sur pywt.

Dans notre cas, on utilise le module signal de la bibliothèque scipy. Le code suivant montre comment construire un scalogramme à partir d'un signal.

```
scg.set_default_wavelet('morl')
scales = scg.periods2scales(np.arange(1, 129))

x_values_wvt_arr = range(0,len(ecg[30][0:100]),1)

plt.figure(figsize=(10,5))
plt.plot(x_values_wvt_arr, ecg[30][0:100], linewidth=3, color='darkblue')
plt.show()

scg.cws(ecg[30][0:100], scales=scales, figsize=(10, 5), coi = False)
plt.show()
```



L'image de droite est le scalogramme qui est le résultat de la transformée en ondelettes. On choisit d'appliquer une ondelette de type Morlet, car elle est souvent utilisée dans le cadre d'étude sur les électrocardiogramme.

5.2 Classification par réseau de neurones simple

Dans cette partie, nous allons construire un réseau de neurones pour classer les ECG, qui prend en entrée le scalogramme des ECG. L'avantage des scalogrammes est qu'elles représentent à la fois des données temporelles sur le signal et aussi des données fréquentielles. Cela permet d'augmenter les chances de mieux discriminer les différentes classes d'électrocardiogramme.
On commence donc par transformer l'ensemble de nos données et créer la structure de notre réseau de neurones.

5.2.1 Préparation des données

En s'inspirant de l'article donné, nous avons pu faire une phase de préparation de données afin de faire une classification des ECG avec un réseau de neurones simple dans un premier temps.

```
arr_set = ecg[0:95]
chf_set = ecg[96:131]
ncr_set = ecg[131:167]


arr_subset_256 = []
for i in range(len(arr_set)):
    arr_subset_256.append(np.array_split(arr_set[i], 256))

chf_subset_256 = []
for i in range(len(chf_set)):
    chf_subset_256.append(np.array_split(chf_set[i], 256))

ncr_subset_256 = []
for i in range(len(ncr_set)):
    ncr_subset_256.append(np.array_split(ncr_set[i], 256))

arr_vect = []
for sublist in arr_subset_256:
    for item in sublist:
        arr_vect.append(item)

chf_vect = []
for sublist in chf_subset_256:
    for item in sublist:
        chf_vect.append(item)

ncr_vect = []
for sublist in ncr_subset_256:
    for item in sublist:
        ncr_vect.append(item)

resize = 500
data = (arr_vect[0:resize] + chf_vect[0:resize] + ncr_vect[0:resize])
M = len(data[0])
```

1. Création du jeu de données Design

Ensuite, on crée le jeu de données Design qui contient les données sous forme d'image scalogramme. Concernant les étiquettes (ARR, CHF et NSR), nous allons utiliser la classe "processing_LabelEncoder" de Sklearn pour encoder ces trois étiquettes en 0, 1 et 2 respectivement.

```
design = np.ndarray(shape=(len(data), M-1, M-1, 3))

for i in range(0, len(data)):
    for j in range(3):
        signal = data[i]
        coeff, freq = cwt(signal, range(1, M), "morl", 1)
        design[i, :, :, j] = coeff[:,:M - 1]
```

2. Création des étiquettes

```
labels = (["ARR"]*resize + ["CHF"]*resize + ['NSR']*resize)

encode = preprocessing.LabelEncoder()
target = encode.fit_transform(labels)
```

3. Séparation de notre jeu de données avec la fonction split

Pour répartitionner les données en un ensemble d'entrainement et de test, on se base sur la fonction "train_test_split" qui prend en paramètre la matrice de Design (X), les étiquettes (target) et un ratio entre la taille de l'ensemble des données pour la phase d'entraînement et la phase de test. On choisit de mettre un ration de 0.25 pour avoir 75% des données de jeu Design comme données d'apprentissage et 25% pour le test.

```
X_train, X_test, y_train, y_test = train_test_split(design, target, test_size = 0.25,
    random_state = 10)
```

5.2.2 Construction de notre réseau de neurones dense

Maintenant, nous allons créer notre réseau de neurone dense composé d'une couche de Flatten permettant de mettre à plat (mettre sous forme d'un vecteur) les images de taille 255 x 255 pixels en couleurs de nos scalogramme à l'entrée du réseau de neurones, et d'une succession de trois couches denses telles que la dernière couche renvoie en sortie un vecteur de taille 3 correspondant aux étiquettes codées (0, 1 et 2).

Le nombre de paramètre de chaque couche se calcule avec "model.summary". On peut également le calculer à la main de la manière suivante :

Rappel : pour une couche dense, le nombre de paramètres est égal à output * (input + 1)
Le flatten n'a pas de parmètre, concernant la dimension de sortie, elle est sous forme d'un vecteur de taille 195075. Donc pour la couche dense 1, le nombre de paramètre est égal à : 300(195075+1)= 58522800

De la même façon, on retrouve le nombre de paramètre des autres couches denses.

```
model = Sequential()
scalogram_shape = [M-1, M-1, 3]

model.add(Flatten(input_shape = scalogram_shape))
model.add(Dense(units = 300 ,activation = 'relu', name = 'dense'))
model.add(Dense(units = 100 ,activation = 'relu', name = 'dense_1'))
model.add(Dense(units = 3 ,activation = 'softmax', name = 'dense_2'))

model.summary()
```

Voici la sortie de la fonction summary() :

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 195075)            0
_________________________________________________________________
dense (Dense)                (None, 300)               58522800
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 303
=================================================================
Total params: 58,553,203
Trainable params: 58,553,203
Non-trainable params: 0
_________________________________________________________________
```

5. Entrainement de notre modèle

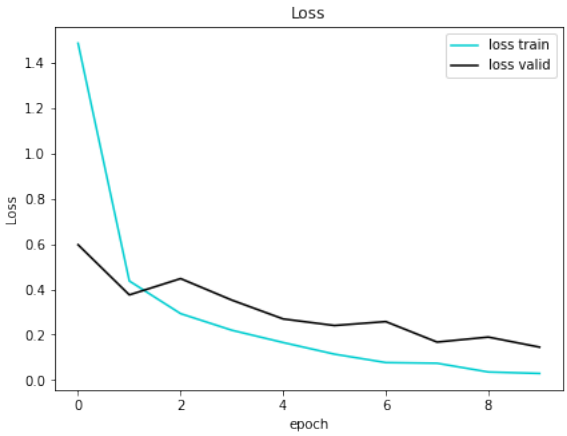
Après avoir crée notre modèle, nous allons l'entraîner avec l'ensemble d'apprentissage et ensuite l'évaluer avec les données de test.

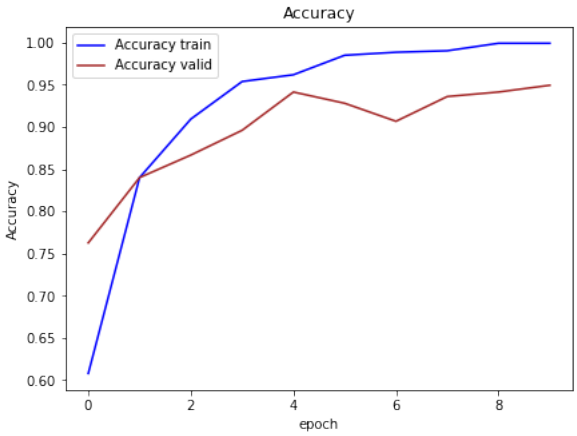
```
model.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'sgd', metrics=["
    accuracy"])

hist = model.fit(X_train, y_train, epochs = 10,validation_data = (X_test, y_test))
model.evaluate(X_test, y_test)

prediction = model.predict(X_test)
print(prediction)
```

6. Courbes d'apprentissage du réseau de neurones

Le code ci-dessous permet de représenter l'évolution de la loss et de l'accuracy en fonction des itérations.

```
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(hist.history['loss'], color='darkturquoise', label="loss train")
plt.plot(hist.history['val_loss'], color='black', label = "loss valid")
plt.title("Loss")
plt.xlabel("epoch")
plt.ylabel("Loss")
plt.legend()

plt.subplot(1,2,2)
plt.plot(hist.history['accuracy'], color='blue', label = "Accuracy train")
plt.plot(hist.history['val_accuracy'], color='brown', label = "Accuracy valid")

plt.title("Accuracy")
plt.xlabel("epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```





Les résultats montrent que la loss diminue avec les itérations pour les deux ensemble d'entraînement et de validation avec une légère différence : loss_valid est moins élevé que celui de l'entraînement au bout de la première itération, mais plus on fait des itérations, loss_valid reste un peu plus élevé que loss_train.

Concernant la précision du modèle, elle augmente avec les epochs juqu'à atteindre 100% à partir de 8 itérations pour l'ensemble d'apprentissage. L'accuracy de validation est moins élevée et n'atteint pas 100% au bout de 8 itération mais un peu moins.

Nous avons donc des résultats assez satisfaisants et le modèle reste performant. Essayons donc de prédire les étiquettes de l'ensemble de test.

```
model.evaluate(X_test, y_test)
y_hat_predict_0 = model.predict(X_test)
print(y_hat_predict_0)

Y_pred_test_0 = np.argmax(y_hat_predict_0, axis=1)
```

La fonction ci-dessous permet d'afficher les images dont les étiquettes ont été mal prédites par le NN :

```
def affichage(X, y, y_pred, nbr_image, nbr_columns = 3, height = 5):
    nbr_errors = np.sum(y != y_pred)

    nbr_lines = int(nbr_errors/3)+1 if nbr_errors > 2 else 1

    error_index = np.where(y != y_pred)[0]

    plt.figure(figsize=(20,height))
    for i in range(error_index.shape[0]):
        plt.subplot(nbr_lines, nbr_columns, i+1)
        plt.imshow(X[error_index[i]])
        plt.title(f"Classe réelle : {y[error_index[i]]}\nClasse prédite : {y_pred[
    error_index[i]]}")

    plt.show()
```

Les résultats obtenus montre que le réseau de neurone basique fait des erreurs sur 19 étiquettes, ce qui est équivalent à un taux d'erreur de (nombre d'images dont les étiquettes ont été mal prédites / nombre total d'image dans l'ensemble de test) = 5Pour chaque image ci_dessous, on a présenté la classe réelle et le classe prédite. Nous remarquons que le NN a fait plus d'erreur sur la classe 2 correspondant à l'étiquette "CHF" (9 erreurs sur 23 au total) contre 6 erreurs pour la classe 0 ("ARR") et 8 pour la classe 1 ("NSR").

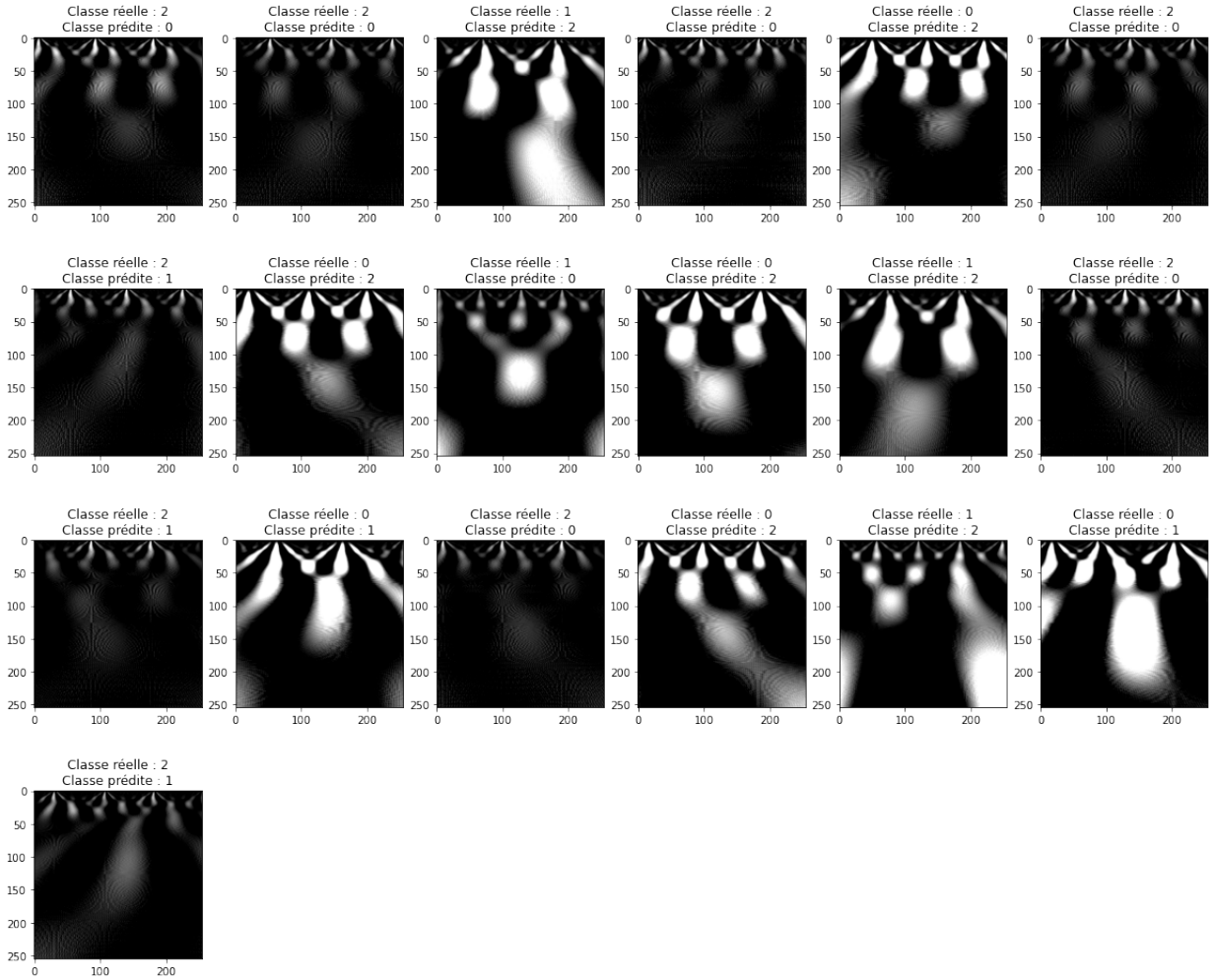


Figure : Résultats de la classification avec un réseau de neurones simple

Une autre manière d'afficher les erreurs est de calculer la matrice de confusion :

```
from sklearn.metrics import confusion_matrix
import pylab as pl

labels=["ARR","CHF","NSR"]

confu_matrix = confusion_matrix(y_test, Y_pred_test_0,normalize = "all")

pl.matshow(confu_matrix)
```

Le résultat ci-dessous, montre que 95% des étiquettes ARR sont correctement prédites, uniquement 1% est prédit en CHF et 4% en NSR. En général le résultat est assez satisfaisant.

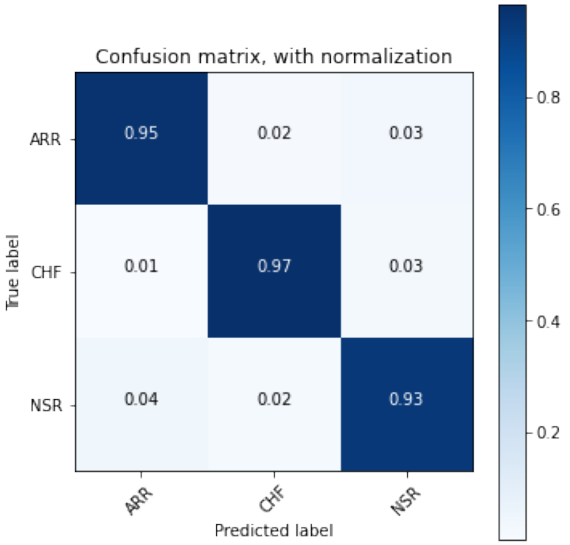
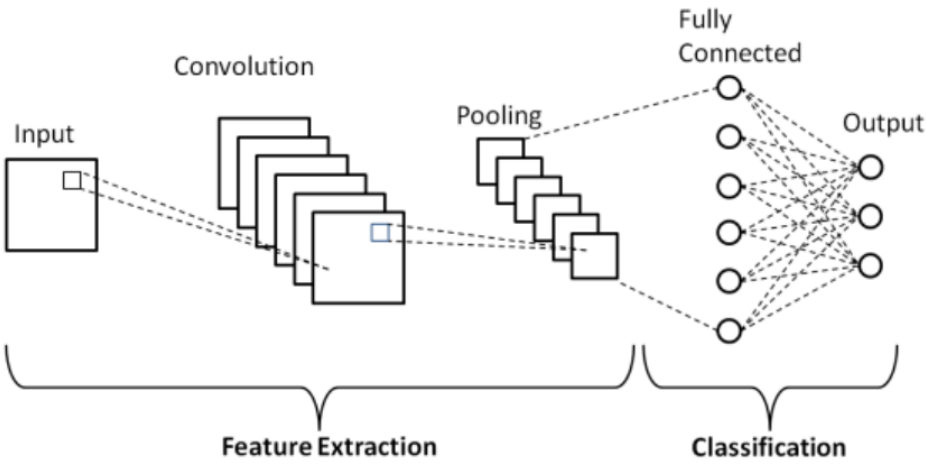


Figure : Résultat de la matrice de confusion de la classification avec NN

5.3 Classification avec un CNN



Le CNN est caractérisé par une architecture composée de plusieurs couches : une couche de convolution permettant d'utiliser des filtres pour repérer des features(les éléments remarquables de l'image). Ensuite, on retrouve une couche de MaxPoling qui extrait les valeurs maximales de certains pixels de l'image à l'aide d'une fenêtre glissante, autrement dit c'est une réduction de la taille de l'image. On peut avoir une succession de ces deux couches autant qu'on veut (dans notre cas, il y a 2 successions). Cette succession est suivie d'un flatten qui sert à applatir la sortie du réseau de convolution (la mettre sous forme d'un vecteur). Ce vecteur sera envoyé à l'entré d'une couche dense qui joue le role d'un classifieur qui renvoie la classe de l'image.

1. Création de notre CNN

```
M = 256
model_1 = Sequential()
scalogram_shape = (M-1, M-1, 3)

model_1.add(Conv2D(filters = 32, kernel_size = (5,5), padding = 'same', input_shape =
    scalogram_shape))
```

```
model_1.add(MaxPooling2D(pool_size = (2, 2),strides = (2, 2), padding='valid'))

model_1.add(Conv2D(filters = 16, kernel_size = (5,5), padding = 'same'))
model_1.add(MaxPooling2D(pool_size = (2, 2),strides = (4, 4), padding='valid'))

model_1.add(Flatten())
model_1.add(Dense(128, activation='relu'))
model_1.add(Dense(3, activation='softmax'))
model_1.summary()
```

L'affichage de la fonction summary est le suivant :

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 255, 255, 32)      2432
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 127, 127, 32)      0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 127, 127, 16)      12816
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 32, 32, 16)        0
_________________________________________________________________
flatten_4 (Flatten)          (None, 16384)             0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               2097280
_________________________________________________________________
dense_5 (Dense)              (None, 3)                 387
=================================================================
Total params: 2,112,915
Trainable params: 2,112,915
Non-trainable params: 0
_________________________________________________________________
```

Le nombre de paramètre est calculé de la manière suivante :

Nous rappelons qu'on a choisi d'appliquer un noyau de taille 5x5 et 32 filtres pour la première couche de convolution. la sortie de cette couche est de 32 images de taille 255 x 255 pixels.

Rappel : Pour une couche de convolution, le nombre de paramètres est égal à $(input \times k_x \times k_y + 1) \times output$ où $k_x \times k_y$ correspond à taille du kernel et le +1 est dû au fait qu'on prend en compte un biais.
La première convolution a donc $(3 * 5 * 5 + 1) * 32 = 2432$ paramètres.
La couche de Max Pooling a un stride de (2,2), donc la sortie devient 32 images de taille 127 x 127 (255/2 et prendre la partie entière), concernant le nombre de paramètre, il est nul.

Comme on applique 16 filtres dans la deuxième convolution, nous avons donc un output de 16 images de taille 127 x 127. Le nombre de paramètre est de $(32 * 5 * 5 + 1) * 16 = 12816$.

La sortie de la couche suivante (Max pooling) est de 16 images de taille 32 x 32 (stride = (4,4). Le flatten transforme cette sortie à un vecteur de taille 32*32*16 = 16384.

La première couche dense a (16384+1)*128 = 2097280 paramètres et la deuxième a 387 paramètres. Nous avons donc un total de 2112915 paramètres.

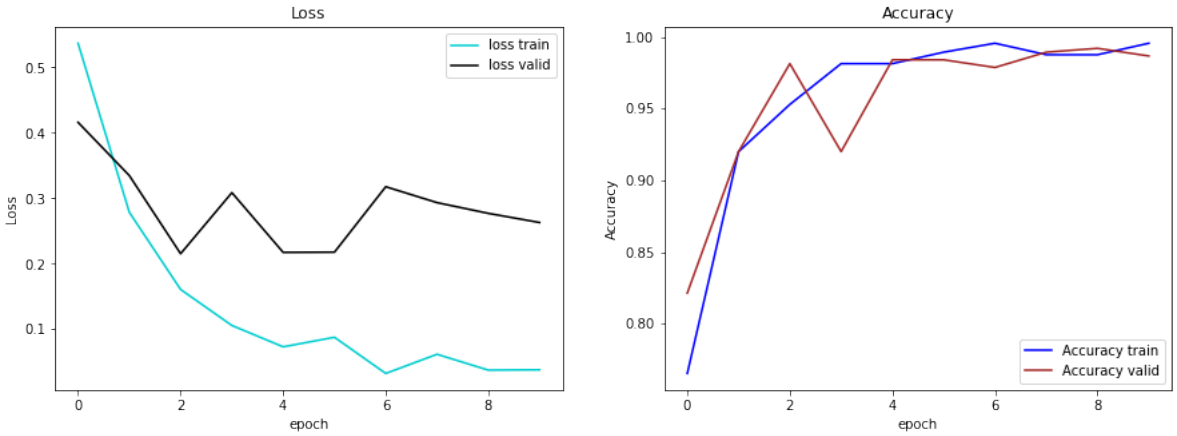
2. Entrainement de notre CNN

```
model_1.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'sgd', metrics=["
    accuracy"])

hist_1 = model_1.fit(X_train, y_train, epochs = 10, validation_data = (X_test, y_test))
```

Le code ci-dessous permet d'afficher les courbes d'apprentissage de notre CNN :

```
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(hist_1.history['loss'], color='darkturquoise', label="loss train")
plt.plot(hist_1.history['val_loss'], color='black', label = "loss valid")
plt.title("Loss")
plt.xlabel("epoch")
plt.ylabel("Loss")
plt.legend()

plt.subplot(1,2,2)
plt.plot(hist_1.history['accuracy'], color='blue', label = "Accuracy train")
plt.plot(hist_1.history['val_accuracy'], color='brown', label = "Accuracy valid")
plt.title("Accuracy")
plt.xlabel("epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Après avoir calculé la perte avec une entropie croisée catégorielle et une métrique de précision, les résultats montrent de très bonnes performances : une perte qui décroît avec l'augmentation du nombre d'itérations pour les deux ensembles d'entraînement et de validation. Nous remarquons aussi une métrique de précision qui augmente jusqu'à atteindre 100% à partir de 8 itérations. Le modèle a donc de très bonnes performances.



Maintenant, en utilisant la même fonction que précédemment, nous affichons les images dont les étiquettes ont été mal prédites par le CNN.

```
model_1.evaluate(X_test, y_test)
y_hat_predict = model_1.predict(X_test)
print(y_hat_predict)

Y_pred_test = np.argmax(y_hat_predict, axis=1)

affichage(X_test, y_test, Y_pred_test, nbr_image = 10, nbr_columns = 6, height = 30)
```

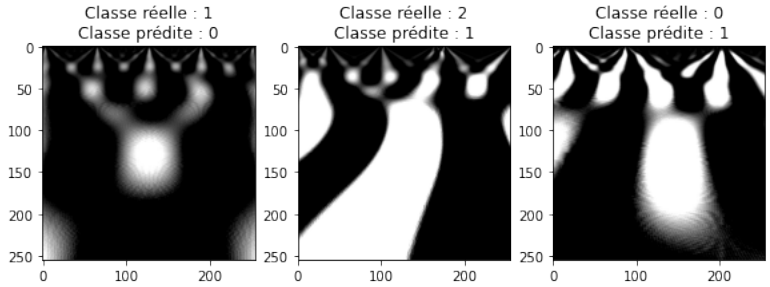


Figure : Résultats de la classification avec CNN

Le résultat montre que le CNN se trompe uniquement sur la prédiction des étiquettes de 3 images. Nous avons donc un résultat meilleur par rapport à la classification avec un réseau de neurones simple.

La matrice de confusion est donc la suivante :

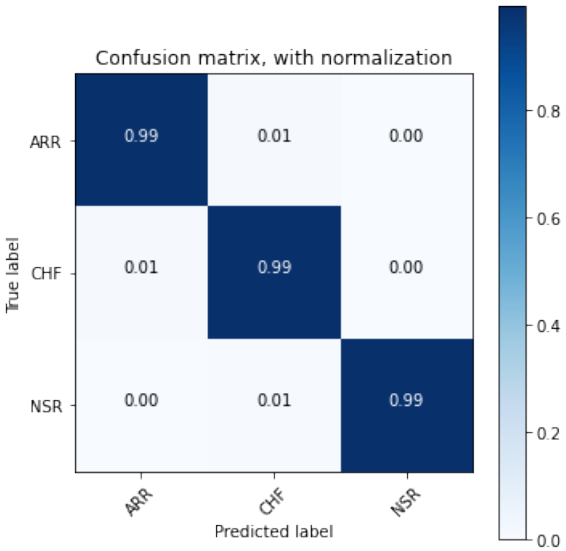


Figure : Résultat de la matrice de confusion de la classification avec CNN

Le résultat est donc meilleur comparé à la classification avec un réseau de neurones simple. Le CNN arrive à prédire les étiquettes des 3 classes correctement avec un taux de 99

6 Conclusion

Dans cet projet, nous avons présenté une analyse d'un signal ECG dans le domaine fréquentiel et temporel et on a utilisé la transformée de Fourier et son inverse pour le passage d'un domaine à un autre. Nous avons découvert aussi la transformée en ondelette permettant d'avoir à la fois l'information sur la fréquence et le temps, cela représente un avantage pour la classification en utilisant soit un réseau de neurones simple ou un réseau de neurones à convolution. En effet, les résultats obtenues sont assez satisfaisants pour le NN et ils sont meilleurs pour le CNN.
Les études actuelles peuvent éventuellement éclairer la voie vers le développement d'algorithmes très lucides et efficaces en termes de temps pour identifier et représenter les complexes QRS, qui peuvent être utilisés dans le cadre d'un programme de recherche. Il est également possible de révéler les caractéristiques d'autres formes d'onde de l'ECG comme les formes d'onde P et T, on pourra dans ce cas apporter des informations importantes sur les conditions physiologiques du patient souffrant d'une maladie cardiaque.

7 Bibliographie

https://www.hindawi.com/journals/mpe/2018/7354081/
https://blog.octo.com/en/time-series-features-extraction-using-fourier-and-wavelet-transforms-on-ecg-data/
https://github.com/alsauve/scaleogram/blob/master/doc/tests.ipynb
https://github.com/mnf2014/article_fft_wavelet_ecg/blob/develop/wavelet_article_octo.ipynb
https://fr.wikipedia.org/wiki/Filtre_passe-bande
https://www.researchgate.net/figure/Schematic-diagram-of-a-basic-convolutional-neural-network-CNN-architecture-26_fig1_336805909