IPSA PARIS

PROTOTYPAGE ET CONCEPTION DES SYSTEMES

Etude de la gouverne de direction et prédiction de la défaillance

Sidney BAKHOUCHE
Romain DENOIX
Aéro 5 CDI

Table des matières

1 Introduction

Le but de ce mini-projet est, après une rapide approche du fonctionnement de la gouverne de direction d'un A320, de mettre en place un algorithme de Machine Learning afin de prédire un défaillance et si possible de la localiser via les données collectées de débit et pression. Nous aurons à notre disposition des données réelles issues d'un banc hydraulique et d'un modèle Amesim. Nous avons choisi de mettre en place un Autoencodeur parce que cette approche en Machine Learning est particulièrement compétente dans la détection d'anomalies d'un système.

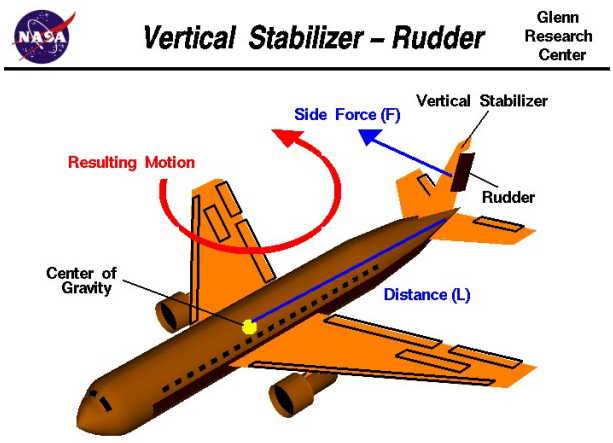
2 La gouverne de direction et le stabilisateur vertical

A la fin du fuselage d'un avion, se trouve deux stabilisateurs : horizontal et verticale. Dans ce projet, nous nous intéresserons à la stabilisation verticale, nous parlerons donc de la gouverne de direction.

FIGURE 1 – A320 Air France

Le stabilisateur horizontal sert à apporter une navigation fiable et droite. Notamment, cette partie va empêcher le nez de l'avion d'être perturbé par des contraintes sur l'axe de lacets. Plus précisément, la gouverne de direction est une partie mobile rattachée au stabilisateur via des charnières. C'est le mouvement de cette pièce qui va dégager une force orthogonale au système pour résulter en un mouvement autour du centre de gravité de l'avion selon l'axe de lacet : le nez de l'avion tourne.

Mais, la gouverne de direction n'est pas utilisée pour faire tourner l'avion. En effet, c'est la combinaison des efforts engendrés par les ailerons et les aérofreins qui provoque le mouvement. La gouverne de ce fait assure que l'avion soit bien aligné avec la courbe de rotation durant la manœuvre. Sans cette contrepartie, l'avion serait victime d'une trainée additionnelle ainsi que d'un mouvement inverse sur l'axe de lacets.



Le couple généré par la gouverne est basé sur le même procédé d'écoulement d'air visible sur les ailes de l'avion. La force F est appliquée sur le centre de poussée du stabilisateur avec une distance L du centre de gravité de l'avion. Ainsi, :

$$T = F \times L \tag{1}$$

Intéressons-nous maintenant à la partie hydraulique qui permet le mouvement de la gouverne. Les ordres de mouvement du pilote partant du cockpit sont acheminés au système par les cables électriques. En effet, ces ordres anciennement mécaniques sont dorénavant electriques, commandant des servo-valvles. Le système est le suivant :

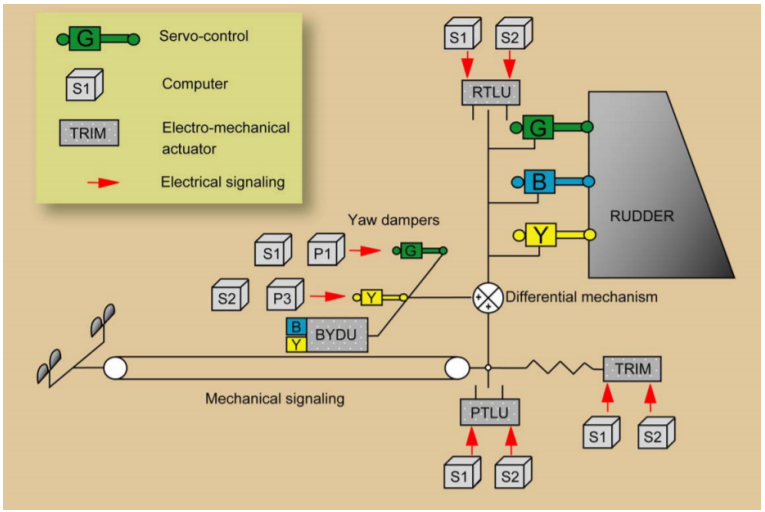


FIGURE 2 – Aperçu du fonctionnement des commandes de la gouverne de direction

Ici, nous notons la présence d'un Damper sur l'axe delacet. Sur l'A320, il remplis ses fonctions en cas de vol manuel et sert à :

- Amortir le "Dutch roll"
- Coordination des virages
- Contrôle latéral

3 La modélisation Amesim

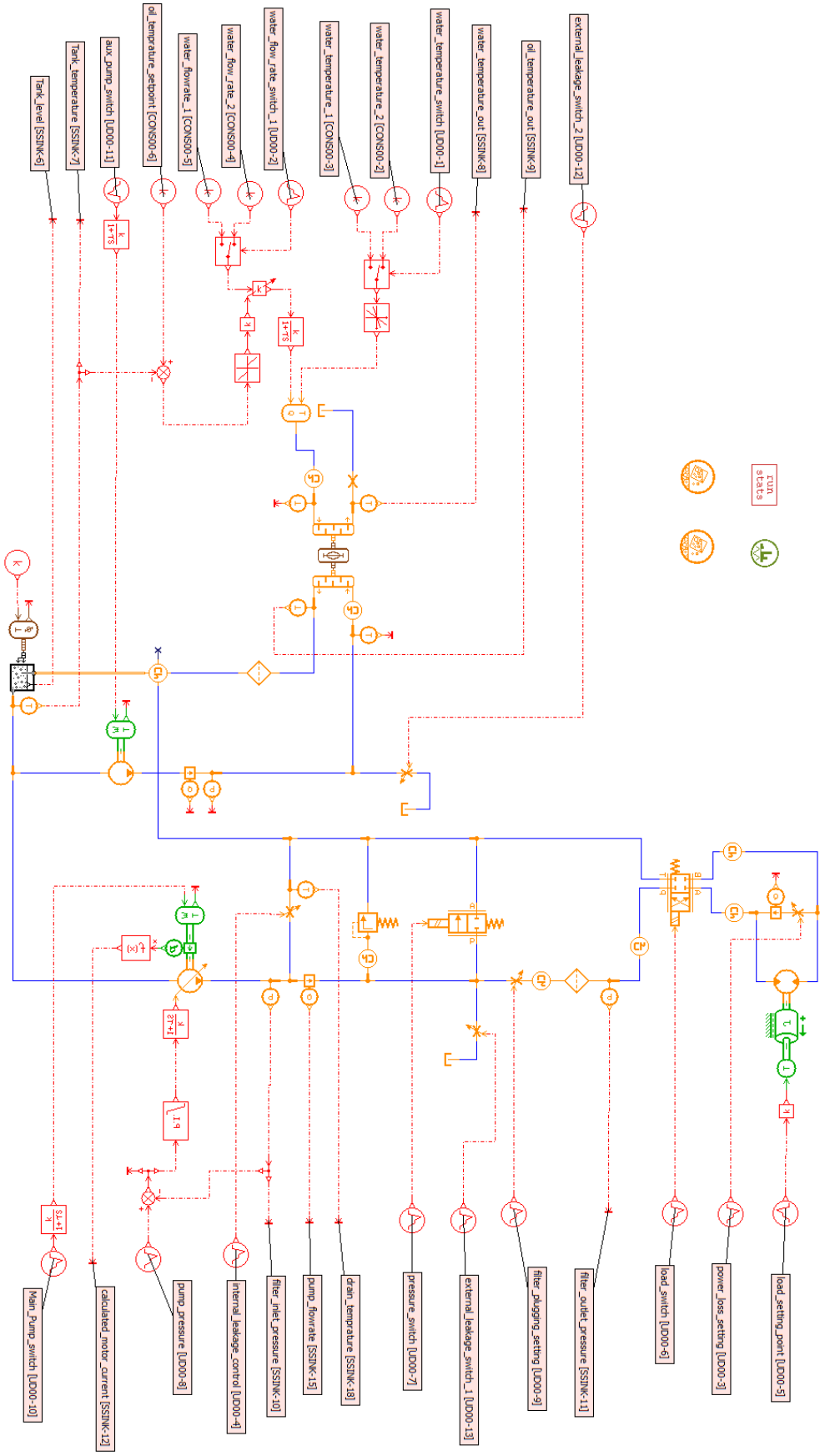


FIGURE 3 – Modèle Amesim du banc d'essai hydrolique

4 La prédiction de la défaillance

Dans cette partie, nous verrons ce qu'est un auto-encodeur pour ensuite l'appliquer à nos données.

Un auto-encodeur est un réseau de neurones capables d'apprendre des représentations efficaces des données d'entrées sans être supervisé, c'est à dire de trouver des structures sous-jacentes à partir de données non étiquetées. Ils sont utilisés pour des modèles génératifs comme avec des images ou alors pour de la détection d'anomalies comme dans le cas présent.

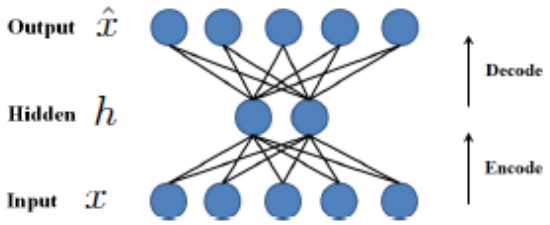


FIGURE 4 – Modélisation de l'auto encoder

Ce système possède une fonction qui encode et une autre qui décode. La première va "tracer" la donnée x sur une couche cachée h via une fonction de retracement :

$$f_\theta(x) = \sigma(W_f^T x + b_f) \tag{2}$$

Avec :

- f_θ le "code"
- σ la fonction d'activation Relu, sigmoïd etc
- W la matrice poid
- b_f le vecteur de biais

Ensuite, h est retracé grâce au décodeur en une reconstruction $\hat{x}$ de la même dimension que x :

$$g_\theta(h) = \sigma(W_g^T h + b_g) \tag{3}$$

Avec :

- g_θ la reconstruction de f_θ
- σ la fonction d'activation Relu, sigmoïd etc
- W la matrice poid
- b_g le vecteur de biais

A noter que la fonction d'activation, les poids et le biais peuvent ou nous différer des paramètres de f_θ selon la construction de l'algorithme.

C'est la reconstruction qui va permettre de détecter l'anomalie. L'entrainement ici va permettre au réseau de reconstruire la donnée d'entrée le plus précisément possible. L'objectif est alors de mesurer l'erreur dans cette reconstruction.En effet, comme l'encodeur est calibré sur le fonctionnement normal, la reconstruction d'anomalies va résulter en des incertitudes en sortie.

Nous évoluerons en Python via l'API Keras. Il s'agit de la librairie la plus utilisée sur Python à des fins de Machine Learning. Nous ponvons représenter notre programme par le diagramme suivant :

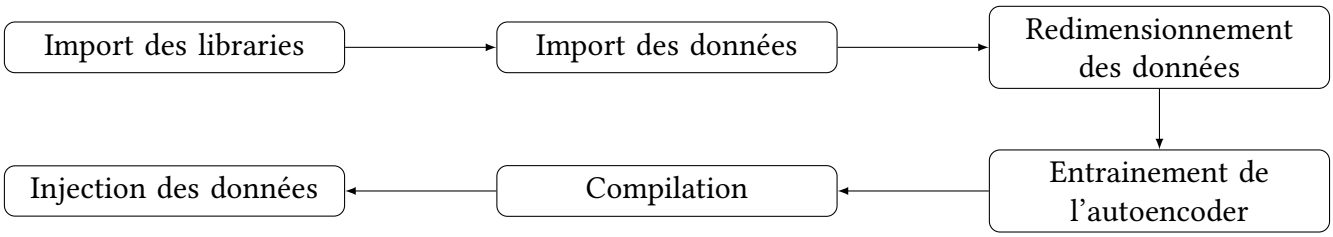


FIGURE 5 – Schéma block de l'algorithme

Une fois importées, les données du banc d'essai sont représentées selon les graphiques suivants

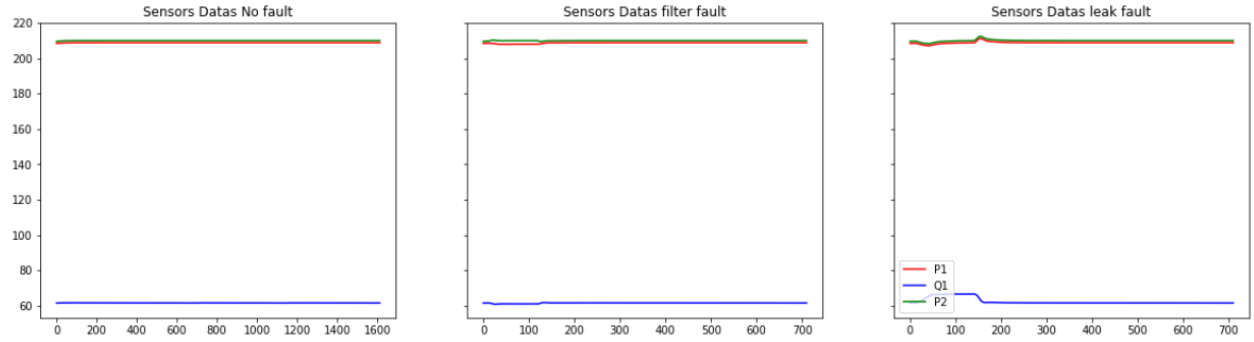


FIGURE 6 – Représentation des données sans faute (gauche), avec une faute de filtre (milieu) et une faute type fuite (droite)

Ensuite, nous créons notre Auto-encodeur. C'est ici que nous allons pouvoir découvrir son architecture. (voir Annexe C) Notre autoencoder se présente sous 10 couches dont 5 d'encodage et 5 de décodage.

Chaque couche est représentée par $LSTM(x)$ signifiant Long short-term memory. Ce type de couche est particulièrement adaptée pour détections d'anomalies grâce à ses trois cellules :

- Input gate
- Output gate
- Forget gate

Communément, la fonction d'activation pour les LSTM est toujours $relu$ ou "logistic sigmoid". Cette fonction d'activation permet de normaliser les données en sortie de neurone. Voici un aperçu partiel du réseau :

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 1, 3)              0
_________________________________________________________________
lstm_31 (LSTM)               (None, 1, 27)             3348
_________________________________________________________________
lstm_32 (LSTM)               (None, 1, 18)             3312
_________________________________________________________________
lstm_33 (LSTM)               (None, 1, 9)              1008
_________________________________________________________________
lstm_34 (LSTM)               (None, 1, 6)              384
_________________________________________________________________
lstm_35 (LSTM)               (None, 3)                 120
_________________________________________________________________
repeat_vector_4 (RepeatVecto (None, 1, 3)              0
_________________________________________________________________
lstm_36 (LSTM)               (None, 1, 3)              84
```

FIGURE 7 – Représentation partielle du réseau avec *Shape* pour le nombre de neurones et *Param* indiquant le nombre de paramètres d'entrée

Après avoir déterminé l'architecture de notre réseau, le réseau est entrainé sur une partie des données puis compilé. Cette étape nous permet de tracer la fonction de coût appelée $loss$. Cette fonction nous permet d'avoir une vue sur les performances de notre modèle selon le graphique tracé suivant :

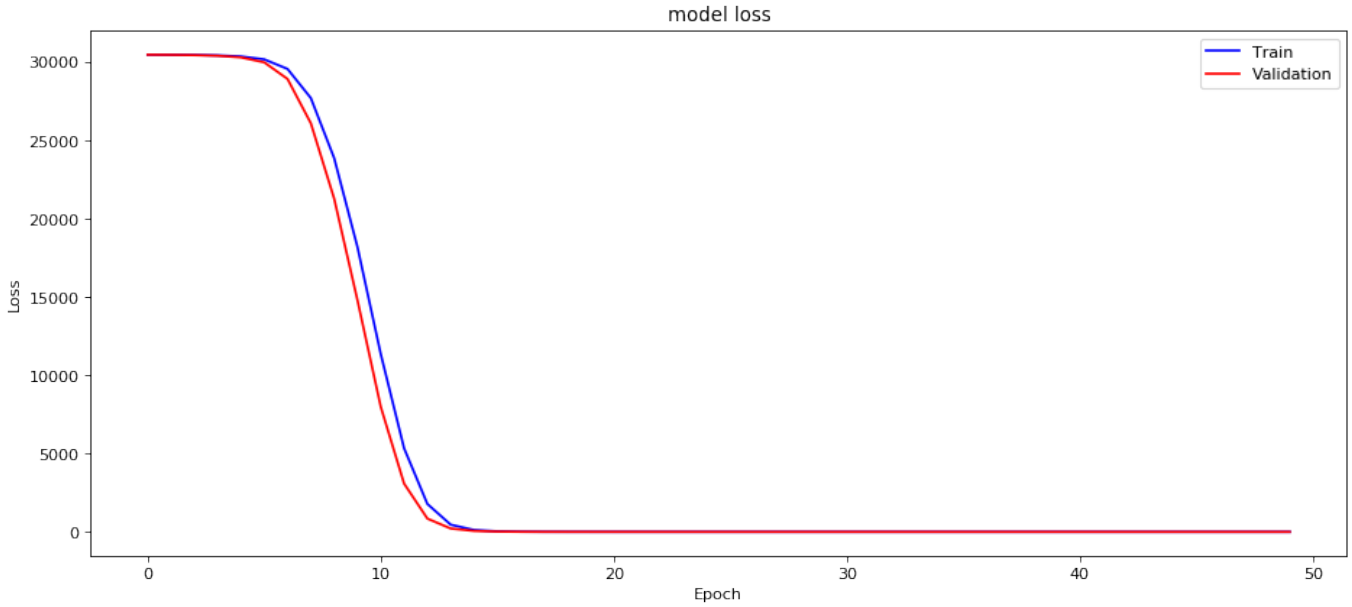
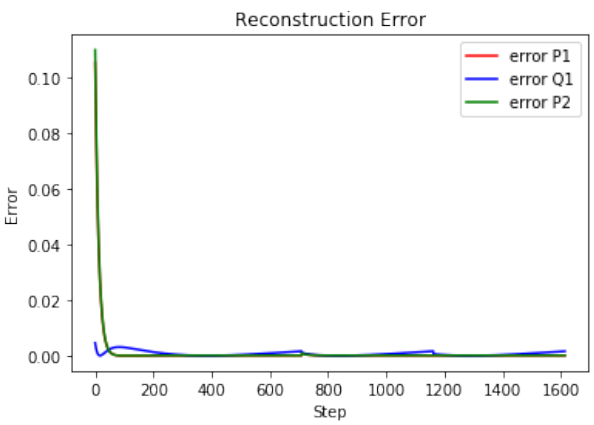


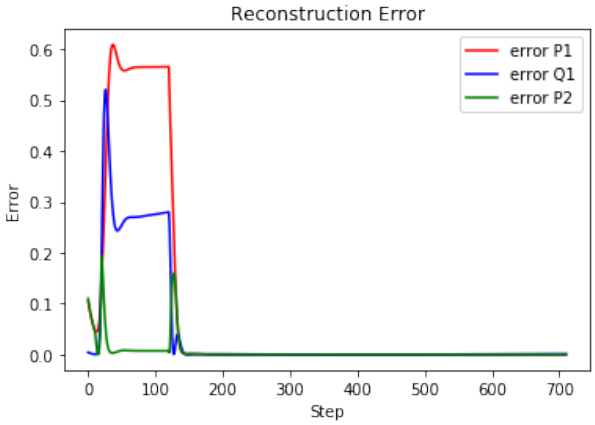
FIGURE 8 – Représentation de la fonction $loss$

Le but de tout réseau de neurones est de minimiser la fonction $loss$, indicateur de sa fiabilité. Sur notre modèle, nous nous stabilisons à partir du 24e "epoch" signifiant la 24e complète présentation des données à apprendre dans le réseau. On voit très bien qu'ensuite la fonction se stabilise donc le modèle est adapté aux données d'entrée. Les deux courbes sont liées aux deux "tests" du réseau. En effet, afin d'apporter un degré supplémentaire de fiabilité au réseau, nous avons effectué une validation croisée ou *cross validation* en divisant trois fois le jeu de données : train, validation et test.

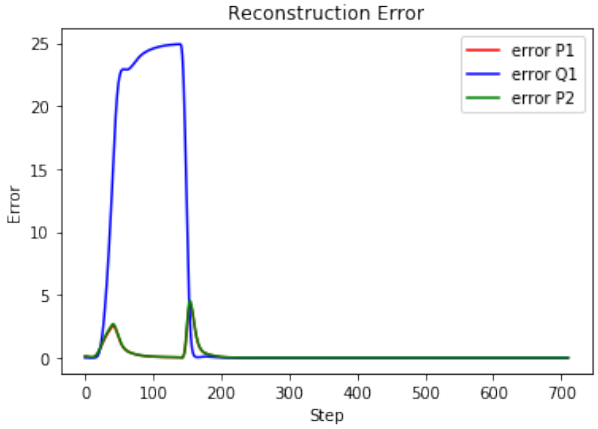
Maintenant que le réseau est entrainé et compilé nous allons pouvoir tracer ses performances. Autrement dit, a-t-il detecté l'anomalie ou non ?



(a) Sans faute



(b) Faute de filtre



(c) Faute type fuite

FIGURE 9 – Représentation des erreurs selon les trois conditions de test

On remarque facilement les erreurs par des pics, expliquer ++

Il y a plusieurs pistes d'amélioration et d'optimisation du réseau. En effet, nous avons ici pris en compte les données issues des fichiers qu'à partir de la 290e step.Ces 290 premières données correspondent au démarage de la machine. Comme le fonctionnement est différent d'un fonctionnement normal dans cet intervale, nous avons décidé de ne pas les prendre en compte afin d'éviter toute polution. Ainsi, en appliquant un 2e réseau sur les 290 première steps, nous aurions eu un résultat plus global. Cependant, cet ajout est assez coûteux en calculs pour peu de résultats significatifs, nous avons donc décidé d'écarter l'idée de l'optimisation par un 2e réseau. Finalement, en négligeant les 290 premières données nous optimisons l'impact du bruit et autres pertubateurs.

Pour aller plus loin et chercher aussi la localisation de la défaillance, nous aurions besoin de plus de données en entrée pour ensuite traiter chaque erreur dans son entièretée afin de diagnostiquer sa provenance. Ainsi, pour l'entrainement du réseau une nouvelle variable verrait le jour impliquant la localisation. Par rapport aux erreurs de P1,P2 et Q1 nous pourrions hypothétiquement déterminer d'où provient la fuite.

5 Tutoriels Python

Cette partie est consacrée à fournir une aide à toute utilisation du réseau de neurones pour d'autres applications.

Afin d'accéder au script, nous devons tout d'abord lancer *Anaconda Navigator*. Pour ce faire, il suffit de le rechercher dans la barre windows ou simplement utiliser un raccourci.

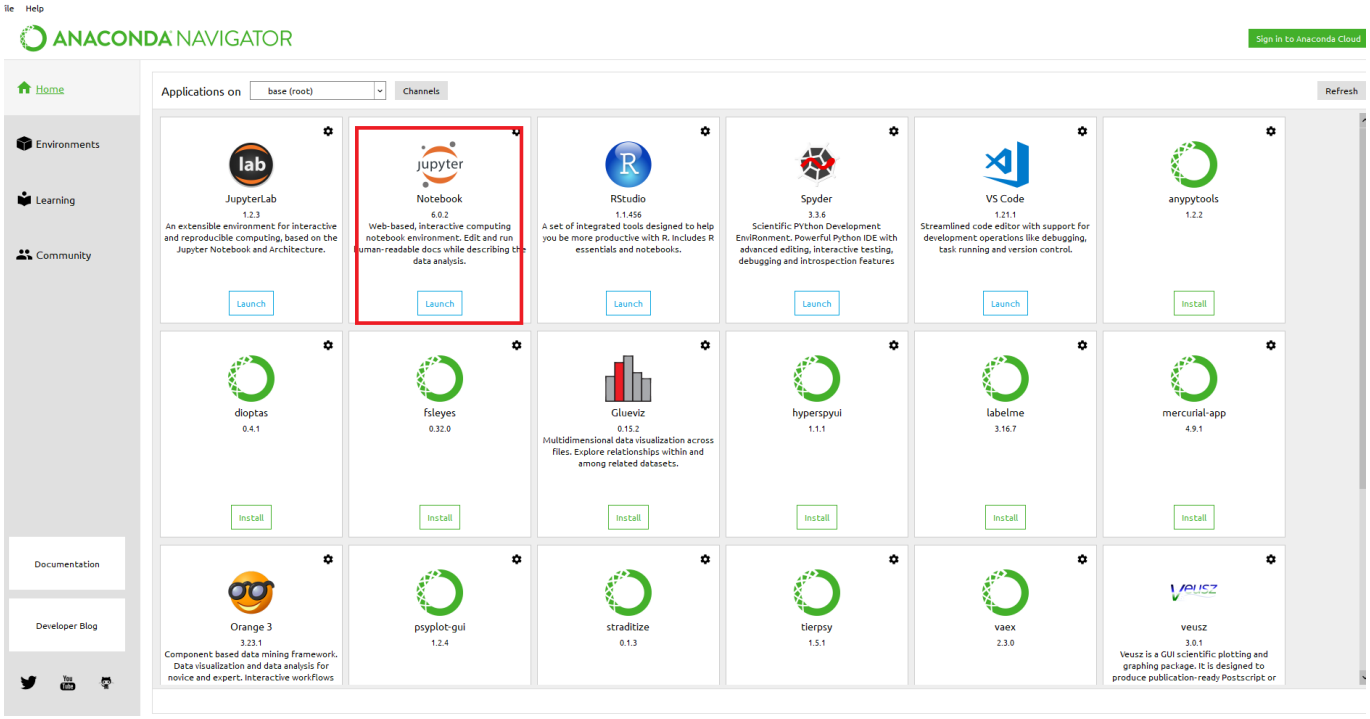


FIGURE 10 – Aperçu de la fenêtre d'acceuil *Anaconda Navigator* et *Jupyter* (encadré)

Après avoir lancé *Jupyter*, des fenêtres s'ouvrent dont une console et un navigateur. Il ne faut surtout pas fermer la console parce qu'elle permet d'établir le lien avec le kernel Python. Une fois dans le navigateur, nous devons ouvrir le fichier Python. Pour ce faire, il suffit de le localiser via l'arborescence. Nous pouvons alors obtenir le résultat suivant :

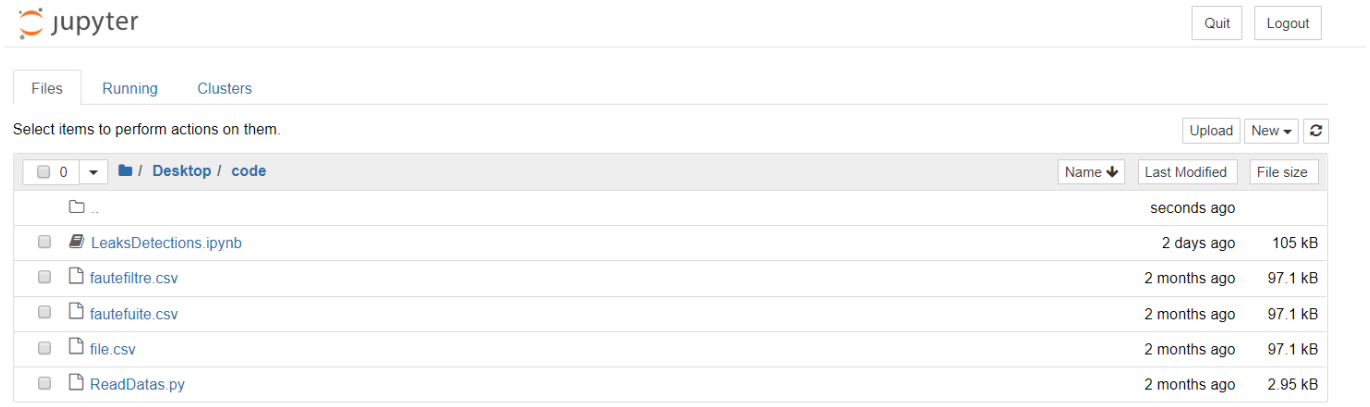


FIGURE 11 – Aperçu du dossier où se trouve notre Python ici *LeaksDetections.ipynb*

Il est très important que les csv contenant les données de test soient dans le même répertoire que le fichier Python sinon l'import ne pourra s'effectuer et une erreur surgira.

Maintenant que vous savez lancer un Python avec *Jupyter*, nous allons maintanant préparer l'environnement pour pouvoir lancer notre algorithme :

1. Ouvrir Anaconda Prompt, accessible via la barre de recherche windows
2. Se placer dans le répertoire du projet avec la commande suivante *cd "C :/[chemin d'accès]/Final"*
3. Créer l'environnement avec *conda env create -f environment.yml*
4. Activer l'environnement grâce à *conda activate my_env*

Enfin, lancer *Jupyter* pour pouvoir accéder au programme pour tester de nouvelles donneés :

1. Ouvrir *LeaksDetections.ipynb*
2. Changer les noms à l'endroit indiqué dans le programme
3. Charger des nouvelles données en suivant le même formatage que dans les anciens fichier grâce à la cellule correspondante.

In [8]:
```
#Creation des tableaux
training=create_set('NoFaultDatas.csv') #Changer ici le nom du fichier

print('\nShape Training set:')
print('Shape X_train:'+ str(training.shape))


test1=create_set('fautefiltre.csv') #Changer ici le nom du fichier

print('\nShape Test1 set:')
print('Shape Test:'+ str(test1.shape))

test2=create_set('fautefuite.csv') #Changer ici le nom du fichier

print('\nShape Test2 set:')
print('Shape Test:'+ str(test2.shape))
```

FIGURE 12 – Cellule d'import des .csv

4. Les 2 dernières cellules correspondent respectivement 1° à la fonction permettant de tester le model 2° d'appeler la fonction avec les données souhaitées.
5. Si des problèmes surgissent, lancer la première cellule pour s'assurer que les librairies soient bien importées. Sinon, relancer la cellule où le nom des fichiers a été changé.

A Import des librairies

```
import csv
import matplotlib.pyplot as plt
get_ipython().magic('matplotlib inline')
import numpy as np
from keras.layers import Input, Dropout, Dense, LSTM, TimeDistributed, Repe
from keras.models import Model
from keras import regularizers
import tensorflow as tf
import pandas as pd
import seaborn as sns
from keras.optimizers import Adam,SGD
from sklearn.preprocessing import MinMaxScaler
```

B Import des données et redimensionnement

```
X_train=training
X_test1=test1
X_test2=test2

### Reshape for LSTM ###
X_train=X_train.reshape(int(X_train.shape[0]),1,X_train.shape[1])
print('\nShape Xtrain LSTM shape:')
print('Shape Train:'+ str(X_train.shape))

X_test1=X_test1.reshape(int(X_test1.shape[0]),1,X_test1.shape[1])
print('\nShape Xtest LSTM shape:')
print('Shape Test1:'+ str(X_test1.shape))

X_test2=X_test2.reshape(int(X_test2.shape[0]),1,X_test2.shape[1])
print('\nShape Xtest LSTM shape:')
print('Shape Test2:'+ str(X_test2.shape))

model = autoencoder(X_train)
history, model=compile_auto(model)
```

C Autoencoder

```
def autoencoder(X):

    inputs = Input(shape=(X.shape[1],X.shape[2]))

    encoder = LSTM(27,activation='relu', return_sequences=True,
                   kernel_regularizer=regularizers.l2(0.003))(inputs)
    encoder = LSTM(18,activation='relu', return_sequences=True)(encoder)
    encoder = LSTM(9,activation='relu', return_sequences=True)(encoder)
    encoder = LSTM(6,activation='relu', return_sequences=True)(encoder)
    encoder = LSTM(3,activation='relu', return_sequences=False)(encoder)
    encoder = RepeatVector(X.shape[1])(encoder)
    decoder = LSTM(3,activation='relu', return_sequences=True)(encoder)
    decoder = LSTM(6,activation='relu', return_sequences=True)(decoder)
    decoder = LSTM(9,activation='relu', return_sequences=True)(decoder)
    decoder = LSTM(18,activation='relu', return_sequences=True)(decoder)
    decoder = LSTM(27,activation='relu',return_sequences=True)(decoder)

    decoder = TimeDistributed(Dense(X.shape[2]))(decoder)
    model = Model(inputs=inputs,outputs=decoder)

    return model
```

D Compiler

```
def compile_auto(model):

    model.compile(optimizer=Adam(lr=0.0003),loss='mse')
    model.summary()

    epochs=50
    bs=16
    history=model.fit(X_train, X_train, epochs=epochs,batch_size=bs,
                      validation_split=0.05)

    fig,ax=plt.subplots(figsize=(14,6), dpi=80)
    ax.plot(history.history['loss'],'b',label='Train')
    ax.plot(history.history['val_loss'],'r',label='Validation')
    ax.set_title('model␣loss')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')
    ax.legend(loc='upper␣right')
    plt.show()
    return history, model
```

E Tracé des erreurs

```
def plot_recons_error(model,X_train):
    pred_train = model.predict(X_train)
    mse = np.mean(np.power(X_train - pred_train, 2), axis=1)
    error_df = pd.DataFrame({'reconstruction_error_P1': mse[:,0],
                             'reconstruction_error_Q1': mse[:,1],
                             'reconstruction_error_P2': mse[:,2]})
    error_df.describe()
    plt.plot(error_df['reconstruction_error_P1'].index,
             error_df['reconstruction_error_P1'],label='error P1',c='r')
    plt.plot(error_df['reconstruction_error_Q1'].index,
             error_df['reconstruction_error_Q1'],label='error Q1',c='b')
    plt.plot(error_df['reconstruction_error_P2'].index,
             error_df['reconstruction_error_P2'],label='error P2',c='g')
    plt.title('Recons Error No fault')
    plt.xlabel('Step')
    plt.ylabel('Error')
    plt.legend(loc='upper right')
    plt.show()
    return pred_train,mse,error_df
```