Rapport TP réseaux de neurones

5 décembre, 2021

1 Problème IV : Time series analysis

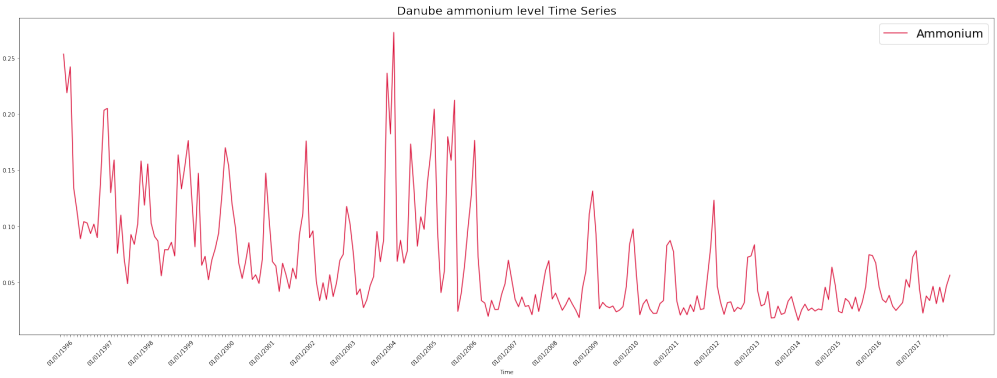
Dans ce problème nous allons utiliser des réseaux de neurones pour la prédiction des séries temporelles. En effet, les réseaux de neurones récurrents (RNN pour Recurrent Neural Network) permettent de traiter des séquences temporelles (langage, vidéos, données numériques). Ils conservent la mémoire des données du passé pour prédire des séquences de données dans le futur proche.

L'objectif est d'implémenter un réseau de neurones de type RNN ou LSTM de haute performance pour prédire le taux d'ammonium mensuel dans l'eau du Danube à un point géographique bien défini. Ainsi, on considére en premier lieu l'échantillon de données « Danube ammonium level Time Series ». Cet échantillon représente le taux d'ammonium dans l'eau du Danube à un point géographique bien défini au début de chaque mois de 01/01/1996 jusqu'à 01/12/2017.

La construction du modèle s'effectue selon les étapes suivantes: data preprocessing, model construction et estimation de la performance.

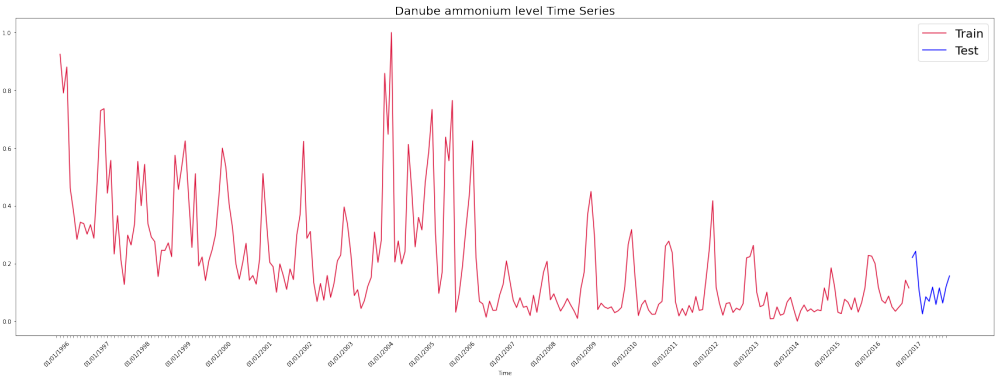
1.1 Data pre-processing:

On considère la base de données « Danube ammonium level Time Series » avec les mois d'observation et les valeurs d'ammonium. On obtient le graphe suivant:



Ensuite, on normalise les données selon la formule suivante en utilisant *sklearn.preprocessing.MinMaxScaler*:
$X_{std} = (X - X_{min})/(X.max - X.min)$

Puis on considére les observations de 01/01/1996 jusqu'à 01/12/2016 pour l'apprentissage et de 01/01/2017 jusqu'à 01/12/2017 pour le test.



On représente les données sous deux colonnes : la première pour le taux d'ammonium à un instant t et la deuxième pour le taux à l'instant t + 1. (Cette représentation sera faite pour les données d'apprentissage et celles de test). Pour cela on met en place la fonction de décalage *createdataset* qui retourne la liste des données et la liste des données décalées

```
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
```

```
look_back = 1
X_train, y_train = create_dataset(train, look_back)
X_test, y_test = create_dataset(test, look_back)
```

L'étape suivante du *Data pre-processing* consiste à préparer les données pour le réseau de neurones on remodele les donnees ainsi: L'entrée de chaque couche LSTM/RNN doit être tridimensionnelle. Les trois dimensions de cette entrée sont :

```
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
print(X_train.shape)
print(X_test.shape)
```

```
(250, 1, 1)
(10, 1, 1)
```

- Échantillons: Une séquence est un échantillon. Un lot est composé d'échantillons
- Pas de temps : Un pas de temps est un point d'observation dans l'échantillon
- Features: Une caractéristique est une observation à un pas de temps.

1.2 Neural Network:

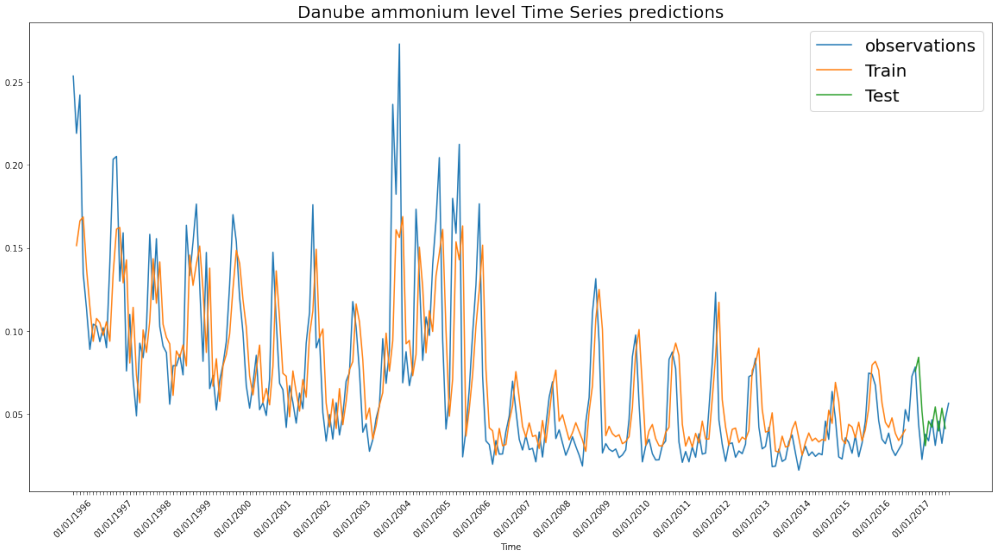
A l'issue du data preprocessing on a des données d'entrée tri-dimensionnelles qu'on introduit au réseau de neurones. Dans ce cas nous allons utiliser un modèle *LSTM*:

- **Sequential model:** On adopte un modele sequentiel oû on peut empiler plusieurs couches
- **LSTM:** On ajoute le LSTM. La cellule permet de maintenir un état aussi longtemps que nécessaire. Cette cellule consiste en une valeur numérique que le réseau peut piloter en fonction des situations. En guise de precision stateful==True, signifie l'état de la dernière formation sera utilisé comme état initial, input shape correspond aux dimensions de l'entrée et units la dimension des cellules internes.
- **Dense:** On ajoute une couche dense ou le nombre d'unité correspond au nombre de features.
- **Compile and fit:** On precise la fonction a minimiser de ce cas le MSE et l'optimizer de ce cas *l'algorithme de Adam* enfin on fit les données au modèle qu'on a mis en place.

```
batch_size = 1
model = Sequential()
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(X_train, y_train, epochs=100, batch_size=batch_size, verbose=2, shuffle=True)
```

1.3 Prédictions et RMSE:

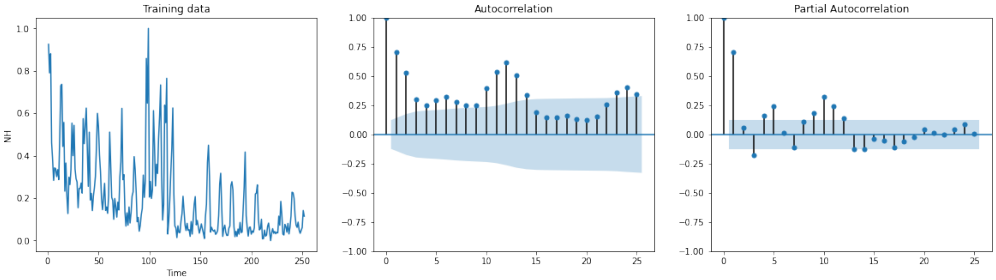
Après avoir compilé le réseau de neurones on examine les performance des paramètres de ce modèle en récupérant les prédictions dur les données d'apprentissage et les données test. Il est important d'inverser la normalisation (on utilise scaler.inverse_transform) puis on calcule le rmse. **On obtient un rmse de 0.03 sur les données d'apprentissage et un rmse de 0.02 sur les données test**



1.4 Modèle série temporelles classique et comparaison:

En guise de comparaison on effectue la prédiction des niveaux d'ammonium a l'aide d'un modèle de série temporelle classique afin d'estimer la valeur ajouté des réseaux de neurones dans la prédiction des séries temporelles.

On commence par une analyse de l'acf et le pacf de la série temporelle afin d'extraire les modèles les plus adéquats. Ainsi on obtient:

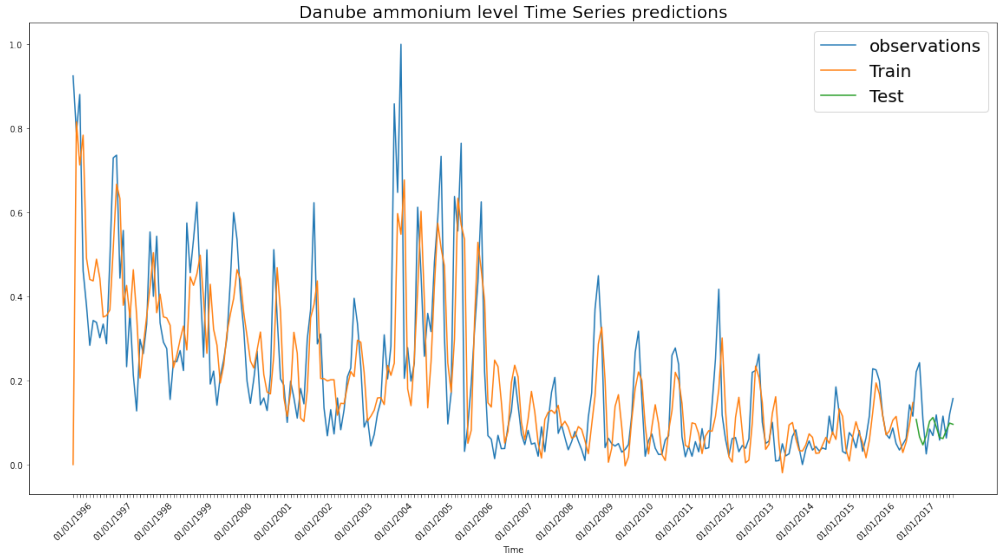


Pour simplifier la tache on utilise la fonction de python *pmàuto_ arima* aui renvoit le modele avec les paramètre optimaux:

ARIMA(4,0,3)(0,0,0)[0]

```
#find optimal model
model = pm.auto_arima(train, start_p=1, start_q=1, test='adf',
                      max_p=5, max_q=5, m=12,
                      d=0, seasonal=False, start_P=0,
                      D=0, trace=True, error_action='ignore',
                      suppress_warnings=True,stepwise=True)
```

On exécute le modèle suggéré ensuite on effectue les prédictions à l'aide du modèle et on calcule le rmse. **On obtient un rmse de 0.13 sur les données d'apprentissage et un rmse de 0.07 sur les données test**

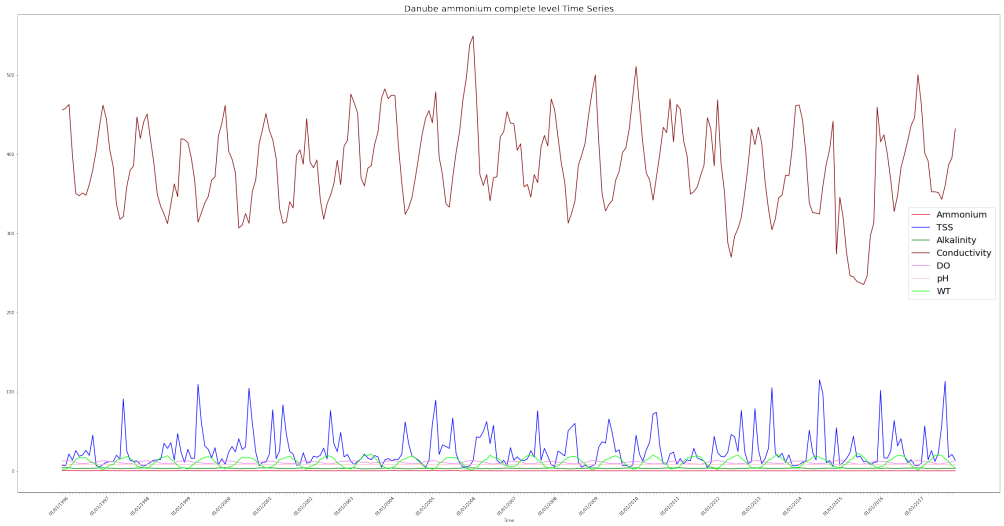


On compare les performances du LSTM et ARIMA a partir des resultats obtenus. On remarque que le rmse du de neurones est proche de celui du mod de temporelles classique, cela se justifie par l'utilisation d'un seul feature ce qui ne traduit pas totalement l'aptitude du LSTM. Bien que les résultats ne soient pas superbes, et inférieurs au modèle ARIMA, il sert de point de départ et il y a certainement place à des améliorations. L'un des avantages du LSTM par rapport aux autres modèles autorégressifs simples est qu'il peut utiliser autant de fonctionnalités d'entrée que nous le souhaitons. On va exploiter cette particularité dans le paragraphe suivant.

1.4.1 LSTM avec plusieurs features:

Dans cette partie, on va utiliser la base de données 'Danube ammonium level complete data Time Series' qui contient les données mensuelles d'Ammonium, TSS, Alkalinity, Conductivity, DO, pH et WT. On trace les données:

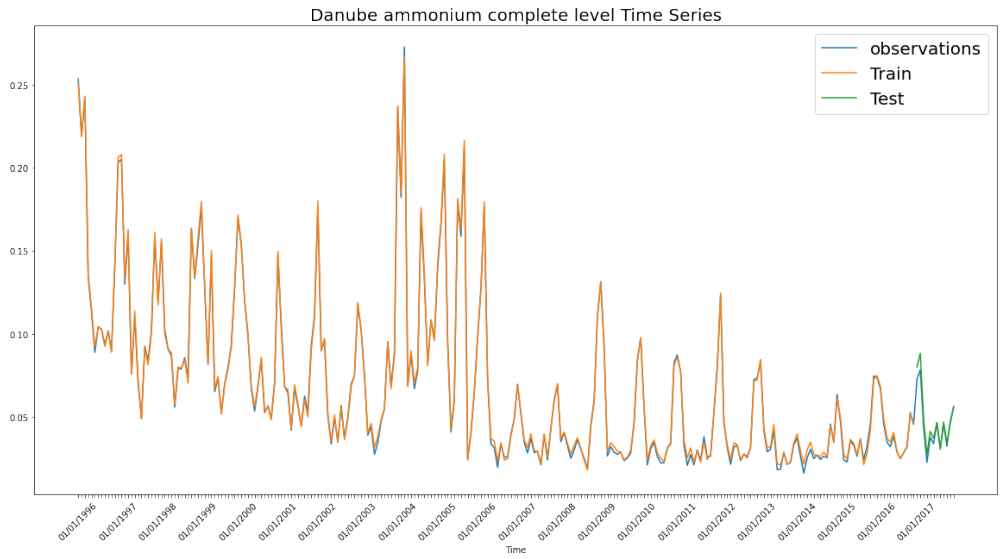
	Months	Ammonium	TSS	Alkalinity	Conductivity	DO	pH	WT
0	01/01/1996	0.253333	7.016667	3.836667	455.916667	12.383333	8.056667	2.075000
1	01/02/1996	0.218889	6.666667	3.968889	458.000000	12.877778	8.068889	1.911111
2	01/03/1996	0.242000	21.420000	3.798000	462.900000	12.990000	8.083000	3.930000
3	01/04/1996	0.134545	13.400000	3.337273	399.727273	11.404545	8.180000	8.863636
4	01/05/1996	0.113333	25.666667	3.032222	350.444444	11.055556	8.393333	14.455556



Ainsi on construit un réseau de neurones avec plusieurs caractéristiques suivant les étapes suivantes: normaliser les données, split les données en apprentissage et test, reshape les données pour afin d'être compatible avec l'entrée du LSTM, ensuite modifier les paramètres du modèle et enfin calculer le Rmse.

```
batch_size = 1
model = Sequential()
model.add(LSTM(5, batch_input_shape=(batch_size, 1, 7), stateful=True))
model.add(Dense(units=7))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(X_train_c,y_train_c,  epochs=100, batch_size=batch_size, verbose=2, shuffle=True)
```

Enfin, on effectue les prédictions selon le modèle et **on obtient un rmse de l'ordre de** 10^{-7} **sur les données d'apprentissage et sur les données test**



En guise de conclusion, on remarque une amélioration significative de la fonction de perte, dans ce cas le rmse, en ajoutant des caractéristiques supplémentaires au modèle. Précédemment l'utilisation d'une seule caractéristique a donné des résultats presque équivalents aux modèle de série temporelles optimales des données. Ainsi, les réseaux de neurones apportent une amélioration à la prédiction des séries temporelles en permettant l'implémentation de plusieurs features qui donnent une meilleure prédiction.