
MAI 2019

# Compte-Rendu
# 3M101

YOUSFI Youcef
PAVLOFF Ulysse
BERREBI Nathane
SERGE Arthur

# Table des matières

# Introduction

Dans de nombreux domaines, pouvoir prédire le déroulement d'un phénomène est très utile et très intéressant. Dans cette optique, l'étude de données au cours du temps est l'objet de nombreuses recherches.
Dans le cadre de l'UE 3M101, il nous est demandé de prévoir la consommation électrique de l'île d'Ouessant sur une durée d'une semaine à partir d'une base de données contenant l'historique d'une année de données à la maille horaire. De nombreuses méthodes existent, et nous utilisereons un modèle mathématique appelé Modèle ARMA, puis un réseau de neurones. L'objectif, à la fin de ce rapport, est de comparer les méthodes utilisées et les optimiser afin d'avoir la meilleure prédiction possible.

# 1 Modèle ARMA et applications

Le modèle ARMA est basé sur la théorie des probabilités et statistiques, et requiert donc un certain niveau d'apprentissage pour en comprendre le fonctionnement. Ainsi, nous allons commencer par définir et étudier le modèle de manière théorique, avant de s'intéresser à son application dans le cadre de notre dataset.

## 1.1 Bases de probabilités

$(\Omega, F, P)$ désigne un espace de probabilité, tel que $\Omega$ soit au plus dénombrable.

### 1.1.1 Variables aléatoires et loi

On appelle variable aléatoire toute fonction $X : (\Omega, F) \to (E, T)$.

On appelle sa loi la mesure de probabilité $P_X$ sur $(E, T)$ donnée par :

$$P_X(A) = \mathbb{P}(X^{-1}(A)) = \mathbb{P}(X \in A) = \mathbb{P}(\{\omega \in \Omega : X(\omega) \in A\})$$

On note la probabilité de l'évenement A pour cette loi plus souvent comme $\mathbb{P}(X \in A)$.

Des variables aléatoires indépendantes et identiquement distribuées (i.i.d) sont des variables aléatoires qui suivent toutes la même loi de probabilité et sont indépendantes.

On définit sa fonction de densité $f_X$ telle qu'elle soit continue et positive, et $F_X$ la fonction de répartition associée à $X$ telles qu'elles soient définies par :

$$F_X(b) = P(X \leq b) = \int_{-\infty}^{b} f_x(x)dx$$

La densité jointe de deux *v.a.r.* continues X et Y, notée $f_{X,Y}(x, y) \geqq 0$, satsifait les proriétés suivantes :

$$\forall (a, b, c, d) \in \mathbb{R}, P(a \leqq X \leqq b, c \leqq Y \leqq d) = \int_a^b \int_c^d f_{X,Y}(x, y)dydx$$

### 1.1.2 Moments

Pour une variable aléatoire réelle continue X, de densité $f_X$, la population des moments associée à tout ordre $k \in \mathbb{N}$, est définie par :

$$E[X^k] = \int_{\mathbb{R}} x^k f_X(x) dx$$

La population des moments centrés associée à tout ordre $k \in \mathbb{N}$, qu'on note $\mu_k$ est définie par :

$$\mu_k = E[(X - \mu)^k] = \int_{\mathbb{R}} (x - \mu)^k f_X(x) dx$$

On utilisera principalement le premier et deuxième moments pour obtenir l'espérance dont on aura besoin pour la stationnarité d'une série temporelle et la variance qui intervient également.

### 1.1.3 Théroèmes fondamentaux

Nous allons maintenant introduire et démontrer des résultats fondamentaux de probabilités qui sont essentiels dans l'étude de série stochastique.

**Loi faible des grands nombres :** soit $(X_n)$ une suite de variables aléatoires iid,tel que $\forall n \in \mathbb{N}$, $X_n$ admet un moment d'ordre 1, alors $\sum_{i=0}^{n} X_i$ converge en probabilité vers $E[|X|]$.

**Démonstration :** Soit $\phi_{\frac{Sn}{n}}$ la fonction caractéristique de $\frac{1}{n}\sum_{i=0}^{n} X_i$, par indépendance des $X_i$ et par propriété de la fonction caractéristique on a :
$\phi_{\frac{Sn}{n}}(t) = \phi_X(\frac{t}{n})^n$
$\Rightarrow ln(\phi_{\frac{Sn}{n}}(t)) = n * ln(\phi_X(\frac{t}{n})) = n * ln(1 - (1 - \phi_X(\frac{t}{n})))$
$= n * (1 - \phi_X(0) + \frac{t}{n}\phi'_X(0) + o(\frac{t^2}{n^2}))$
$= itE[X] + o(\frac{t^2}{n^2}) \Rightarrow \phi_{\frac{Sn}{n}} \rightarrow \exp^{itE[X]}$ qui est la fonction caractéristique de la constante E[X]. Donc $\sum_{i=0}^{n} X_i$ converge en loi vers $E[|X|]$, or $E[X]$ est une constante donc $\sum_{i=0}^{n} X_i$ converge aussi en probabilité.
On peut étendre ce resultat de convergence a une convergence presque sûrement grâce à la loi forte des grands nombres.

**Loi forte des grands nombres :** soit $(X_n)$ une suite de variables aléatoires iid,tel que $\forall n \in \mathbb{N}$ $X_n$ admet un moment d'ordre 1, alors $\sum_{i=0}^{n} X_i$ converge presque sûrement vers $E[|X|]$.

**Démonstration :** Démontrons ce résultat dans le cas où les $X_n$ admettent un moment d'ordre 2. Soit $S_n = (\sum_{i=0}^{n} X_i)$ et $\mu = E[X]$, alors d'après l'inégalité de Tchebychev on a :
$P(|\frac{S_{n^4}}{n^4} - \mu| \geq \epsilon) \leq \frac{var(\frac{S_{n^4}}{n^4} - \mu)}{\epsilon}$
or $var(\frac{S_{n^4}}{n^4} - \mu) = var(\frac{S_{n^4}}{n^4}) = var(S_{n^4})/n^8 = var(X)/n^4$, on a alors :
$P(|\frac{S_{n^4}}{n^4} - \mu| \geq \epsilon) \leq \frac{var(X)}{n^4}$.
Or $\sum_{i=0}^{\infty} \frac{var(X)}{n^4} < \infty \Rightarrow \sum_{i=0}^{\infty} P(|\frac{S_{n^4}}{n^4} - \mu| \geq \epsilon) < \infty \; \forall \epsilon \in \mathbb{R}$
Donc d'après le lemme de Borel-Cantelli, $P(limsup(|\frac{S_{n^4}}{n^4} - \mu| \geq \epsilon)) = 1 \; \forall \epsilon \in \mathbb{R}$,d'où $\frac{S_{n^4}}{n^4} \Rightarrow_{p.s} \mu$
De plus, $\forall k \in [n^4, (n+1)^4]$, on a :
$\frac{S_k}{k} \leq \frac{S_{(n+1)^4}}{n^4} = (\frac{n+1}{n})^4 \frac{S_{(n+1)^4}}{n^4} \Rightarrow_{p.s} \mu$
Donc $limsup \frac{S_k}{k} = \mu$.
$\frac{S_k}{k} \geq \frac{S_{n^4}}{(n+1)^4} = (\frac{n}{n+1})^4 \frac{S_{n^4}}{(n+1)^4} \Rightarrow_{p.s} \mu$
Donc $liminf \frac{S_k}{k} = \mu$.
D'où $\frac{S_k}{k} \Rightarrow_{p.s} \mu$

**Théorème Central limite :** soit $(X_n)$ une suite de variables aléatoires iid tel que $\forall n \in \mathbb{N}$ $X_n$ admet un moment d'ordre 2. alors, $\frac{1}{\sqrt{n}}(\sum_{i=0}^{n} X_i - \mu)$ converge en loi vers une loi normale centrée de variance $\sigma^2$, avec $\sigma^2 = E[|X|^2]$ et $\mu = E[|X|]$.

**Démonstration :** Soit $(X_n)$ une suite de variables aléatoires iid tel que $\forall n \in \mathbb{N}$ $X_n$ admet un moment d'ordre 2. On suppose que $[|X|^2] = \sigma^2$ et $E[|X|] = 0$ (on peut pour ce faire considérer les v.a $X = X - E[X]$). Soit $\phi_{\frac{Sn}{\sqrt{n}}}$ la fonction caractéristique de $\frac{1}{\sqrt{n}}(\sum_{i=0}^{n} X_i$, par indépendance des $X_i$ et par propriété de la fonction caractéristique on a :
$\phi_{\frac{Sn}{\sqrt{n}}}(t) = \phi_X(\frac{t}{\sqrt{n}})^n$
$\Rightarrow ln(\phi_{\frac{Sn}{\sqrt{n}}}(t)) = n * ln(\phi_X(\frac{t}{\sqrt{n}})) = n * ln(1 - (1 - \phi_X(\frac{t}{\sqrt{n}})))$
$= n * (1 - \phi_X(0) + \frac{t}{\sqrt{n}}\phi'_X(0) + (\frac{t}{\sqrt{n}})^2 \phi''_X(0) + o(\frac{t^2}{n^2}))$
$= n * ((\frac{t}{\sqrt{n}})^2 \sigma^2 + o(\frac{t^2}{n^2}))$ car $\phi_X(0) = 1$, $\phi'_X(0) = iE[|X|] = 0$ et $\phi''_X(0) = -E[|X|^2] = \sigma^2$
Donc on $lim_{n\to\infty} \phi_{\frac{Sn}{\sqrt{n}}}(t) = \exp^{\sigma^2 t^2}$ qui est la fonction caractéristique d'une loi normale de variance $\sigma^2$. D'où la convergence en loi souhaitée.

## 1.2 Séries Temporelles

Afin d'étudier le modèle ARMA, il faut d'abord étudier l'objet mathématique que l'on utilisera : les séries temporelles.

### 1.2.1 Définition

Une série temporelle est une suite d'observations d'une même grandeur, chacune d'entre elles correspondant à une date.
Concrètement, il s'agit d'une manière structurée de représenter les données et visuellement, on la voit comme une simple courbe en fonction du temps. En pratique, on utilise les séries temporelles dans le cadre des études statistiques d'un phénomène au cours du temps.

### 1.2.2 Stationnarité d'une série temporelle

Un processus $(x_t, t \in \mathbb{Z})$ est dit stationnaire au second ordre, ou stationnaire au sens faible, ou stationnaire d'ordre deux si les trois conditions suivantes sont satisfaites :

- $\forall t \in \mathbb{Z}, E[x_t^2] < \infty$
- $\forall t \in \mathbb{Z}, E[x_t] = m$, indépendant de $t$
- $\forall (t,h) \in \mathbb{Z}^2, \gamma(h)$ est indépendant de $t$

On définit alors une série temporelle $(X_t)_{t\in\mathbb{N}}$ de variables indépendantes et identiquement distribuées (qui suivent toutes la même loi de probabilité) comme stationnaire, c'est-à-dire si ses propriétés probabilistes sont les mêmes que celles de la série $X_{t+h}$ pour tout entier $h$. $(X_t)$ est stationnaire au sens strict si $(X_1, X_2, ..., X_k)$ a même loi que $(X_{1+h}, X_{2+h}, ..., X_{k+h})$ pour tout entiers $h$ et $k > 0$.

On définit également la fonction d'autocovariance d'une série de variables aléatoires i.i.d par :

$$\gamma_t(h) = cov(Y_t, Y_{t-h}) = E[(Y_t - E[Y_t])(Y_{t-h} - E[Y_{t-h}])]$$

## 1.3 Théorème de Décomposition de Wold

### 1.3.1 Bruit Blanc

Un bruit blanc est un processus stationnaire à accroissements indépendants (c'est-à-dire $X_{t_i}$-$X_{t_{i-1}}$ sont indépendantes). Un processus $(x_t, t \in \mathbb{Z})$ est un bruit blanc (qu'on note $(\epsilon_t) \sim BB(0, \sigma^2)$) si $\forall$t $\in \mathbb{Z}$ :

- $E[\epsilon_t] = 0$
- $Var(\epsilon_t) = \sigma^2$
- $cov(\epsilon_t, \epsilon_s) = 0, t \neq s$

### 1.3.2 Théorème de Décomposition de Wold

Tout processus stationnaire d'ordre deux $(x_t, t \in \mathbb{Z})$ peut être représenté sous la forme :

$$x_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j} + \kappa_t$$

- où $\psi_j \in \mathbb{R}$ ,$\forall j \in \mathbb{N}^*$ et vérifie $\psi_0 = 1$
- et $\sum_{j=0}^{\infty} \psi_j^2 < \infty$
- et $\epsilon_t$ est un bruit blanc et $\kappa_t$ et une composante linéaire déterministe telle que $cov(\kappa_t$ ,$\epsilon_{t-j}) = 0$

Ce modèle est théoriquement bon mais la modélisation de somme infini en informatique n'est pas optimal, on ne va donc pas implémenter cette méthode pour représenter une série stationnaire.

### 1.3.3 Application de Wold

Après un bref calcul qui utilise la linéarité de l'espérance et la formule de Bayes on obtient:

$$X_{t+1} = \sum_{j=1}^{\infty} \psi_j \epsilon_{T-j+1}$$

On a donc une erreur d'approximation égale à $\epsilon_t$.

Pour un processus stationnaire, l'erreur de prévision optimale à un horizon d'une période correspond à l'innovation fondamentale (au choc) associé à cette même prédiction. En particulier, cette erreur est indépendante des erreurs précédentes (pas d'accumulation d'erreurs).

En appliquant la même méthode, on peut alors approximer les valeurs de $X_{t+h}$ $\forall h \in \mathbb{N}$.

On a ainsi :

$$X_{t+h} = \sum_{j=1}^{\infty} \psi_j \epsilon_{T+h-j}$$

avec une erreur d'approximation égale à $\sum_{j=1}^{h-1} \psi_j \epsilon_{T+h-j}$

## 1.4 Modèle ARMA

Nous avons à présent presque tous les outils mathématiques pour définir le modèle ARMA. Il ne reste qu'à définir l'opérateur retard.

### 1.4.1 Polynôme retard

Un retard est un opérateur tel que $Lx_t = x_{t-1}$. Cet opérateur permet donc de définir tous les $x_t$ à partir de $x_0$ car $L^t x_t = x_0 \ \forall t \in \mathbb{N}$
Un polynôme retard est un polynôme évalué en un opérateur retard. On peut donc exprimer la décomposition de Wold en posant :

$$\Phi(X) = \sum_{t=0}^{t=\infty} \psi_t X^t$$

On retrouve donc d'après la décomposition de Wold:

$$\Phi(L)\epsilon_t = \kappa_t - x_t$$

On pourra exprimer d'autres formules à l'aide du polynôme retard, notamment celle d'un modèle AR, MA ou encore ARMA.

### 1.4.2 Moyenne Mobile d'ordre $q$ : MA(q)

Soit $X_t$ un processus, on dit que $X_t$ représente une moyenne mobile si :

$$X_t = m + \sum_{n=1}^{q} \theta_i \epsilon_{t-i} \quad \text{, avec } \epsilon_t \sim BB(0, \sigma^2), \text{ m, } \theta \in \mathbb{R}.$$

Un processus MA $X_t$ est stationnaire car $E[X_t] = m$, et $E[X_t^2] = m^2 + \sigma^2(1 + \sigma^2) < +\infty$.
On peut aussi voir un MA(q) comme une décomposition de Wold tronquée au rang q.

### 1.4.3 Autoregression d'ordre $p$: AR(p)

Soit $X_t$ un processus, on dit que $X_t$ représente une auto-régression d'ordre p notée AR(p) si :

$$X_t = c + \sum_{j=1}^{p} \phi_j X_{t-j} + \epsilon_t$$

avec $\phi \in \mathbb{R}$ et $\epsilon_t \sim BB(0, \sigma^2)$ (un bruit blanc)

On peut également définir le AR avec le polynôme retard de la manière suivante :

$$\Phi : \Phi(L)X_t = c + \epsilon t$$

Un modèle autoregressif est toujours inversible mais en général AR(p) ($\forall p \in \mathbb{N}$) n'est pas stationnaire sauf lorsque toutes les racines du polynôme retard sont de module strictement supérieur à l'unité.

### 1.4.4 Modèle ARMA(p,q)

Un processus ARMA(p,q) est une concaténation d'un AR(p) et un MA(q) :

$$\sum_{i=0}^{p} -\phi_i X_{t-i} = \sum_{j=0}^{q} -\psi_j \epsilon_{t-j}$$

avec $\phi_0 = \psi_0 = -1$, $\phi_p \neq 0$, et $\psi_q \neq 0$. $\epsilon_t$ un bruit blanc faible.

### 1.4.5 Saisonnalité

On peut introduire un caractère saisonnier à notre série pour modéliser une variation liée à la saison en rajoutant une fonction périodique $s_t$ appelée saisonnalité. On a ainsi:

$$X_t = s_t + Y_t$$

avec $s_t = s_{t-d}$ car $s$ est de période $d$.

Si $Y_t$ est un modèle ARMA alors on dit que $X_t$ est un SARMA.

## 1.5 Principe d'utilisation d'un modèle ARMA

Maintenant que le modèle ARMA est définie, nous nous intéressons à comment il est utilisé en pratique. De nombreuses hypothèses doivent être vérifiées afin de pouvoir modéliser une série temporelle par un modèle ARMA.

### 1.5.1 Inversibilité et stationnarité

On dit qu'un modèle AR est inversible si on peut le réécrire comme un modèle MA. De même, un modèle MA est inversible si on peut le réécrire comme un modèle AR. Ainsi, un processus ARMA est inversible si ces composantes AR et MA sont toutes les deux inversibles.
Comme on la vu précédemment, un processus AR est toujours inversible et un processus MA est toujours stationnaire. Donc un processus est inversible si et seulement si sa composante MA est inversible et il est stationnaire si et seulement si sa composante AR est stationnaire.

De manière générale, le critère principale d'inversibilité (respectivement stationnarité) d'un processus MA (respectivement AR) est que les racines du polynôme retard associé sont de module inférieur à 1. De plus, inverser un processus revient à trouver le polynôme inverse du polynôme retard.

Afin de déterminer si le processus ARMA est stationnaire, on doit donc chercher les racines du polynôme retard associé à sa composante AR en cherchant par exemple les valeurs propres de la matrice compagnon du polynôme retard. Puis, pour déterminer si le processus ARMA est inversible, on cherche les racines de sa composante MA.

### 1.5.2 Méthode d'inversion de polynômes

Pour inverser un processus AR ou MA on doit inverser son polynôme retard. Pour cela, il existe deux méthodes:

- Identification des coefficients : on résout un système linéaire à $n$ équations pour déterminer les $n$ premiers coefficients du polynôme inverse.
  En effet, soit $\Phi$ un polynôme et $\Phi^{-1}$ son inverse, alors $\Phi(\Phi^{-1}(x)) = 1 \ \forall x \in \mathbb{R}$, donc tous les coefficients du polynômes produit sont nul sauf le premier qui vaut 1. De plus, on peut calculer le $n^{ime}$ coefficient du polynômes produit à partir des $(n-1)$ coefficients de $\Phi^{-1}$ et $\Phi$. On peut donc retrouver tous les coefficients de $\Phi^{-1}$ par itération.

- La deuxième méthode consiste à calculer la décomposition en facteurs premiers de la fonction $f(x) = \frac{1}{\Phi(x)}$. On inverse ensuite chaque facteur de la décomposition pour obtenir $\Phi^{-1}$

Même si on peut vérifier que notre modèle est inversible et stationnaire, il reste que les données initiales peuvent présenter des caractéristiques qui empêchent cette inversibilité et stationnarité. Il faut donc traiter ces données et il existe différents moyens de se débarasser de ces problèmes, selon les cas.

### 1.5.3 Elimination de la tendance par différenciation

Soit une série temporelle $(Y_t)$ tel que : $\forall\ t \in \mathbb{Z} : Y_t = T_t + X_t$ où $T_t$ est la tendance déterministe de la série et $X_t$ est une série stationnaire.
Pour stationnariser la série, on doit éliminer la tendance. Pour cela il existe plusieurs méthodes.

Supposons dans un premier temps que la tendance est une fonction affine de la forme $T_t = at + b$ avec $(a, b) \in \mathbb{R}^2$.
On pose $Z_t = Y_t - Y_{t-1}\ \forall\ t \in \mathbb{Z}$. On a donc :

$$Z_t = at + b + X_t - (a(t-1) + b + X_{t-1}) = at + b + X_t - (at - a + b + X_{t-1}) = a + X_t - X_{t-1}$$

$(X_t)$ est une série stationnaire donc $(X_t - X_{t-1})$ est aussi stationnaire, d'où $Z_t$ est aussi stationnaire.

Supposons maintenant que $T_t$ est un polynôme de degrés $n$. Montrons par récurrence que l'on peut supprimer la tendance par différenciation.

- Initialisation : Un polynôme de degré 1 est une fonction affine donc on peut procéder comme vu précédemment.
- Hérédité : Soit $T_t = \sum_{i=0}^{n} a_i t^i$.

$$\begin{aligned}
Z_{t+1} &= \sum_{i=0}^{n} a_i (t+1)^i + X_{t+1} - (\sum_{i=0}^{n} a_i t^i + X_t) \\
&= \sum_{i=0}^{n} a_i \sum_{j=0}^{i} \binom{i}{j} t^j - \sum_{i=0}^{n} a_i t^i + X_{t+1} - X_t \text{ (D'après la formule de Newton)} \\
&= \sum_{i=0}^{n} a_i (\sum_{j=0}^{i} \binom{i}{j} t^j - t^i) + X_{t+1} - X_t \\
&= \sum_{i=0}^{n} a_i \sum_{j=0}^{i-1} \binom{i}{j} t^j + X_{t+1} - X_t \\
&= \sum_{i=0}^{n-1} b_i t^i + X_{t+1} - X_t = P(t) + X_{t+1} - X_t \text{ ,avec P un polynôme de degré } n-1.
\end{aligned}$$

- Conclusion : D'après la propriété de récurrence on peut éliminer la tendance en effectuant $n$ différenciations successives.

Une fois qu'on à éliminer la tendance par la méthode de différenciation, on obtient une série stationnaire (dans le cas où $Y_t = T_t + X_t$ ). On peut donc faire nos prévisions avec le modèle ARMA. Une fois les prévisions faites, on se ramène à la série initiale en effectuant une différentiation inverse grâce a une méthode itérative.

On a $Z_t = X_t - X_{t-1} \Rightarrow X_t = X_{t-1} + Z_t \ \forall t \neq 0$.
On commence donc avec $X_0 = Z_0$, puis $X_i = X_{i-1} + Zi \ \ \forall i \in \{1, ..., n\}$

### 1.5.4 Elimination de la tendance par moyenne mobile

Une autre méthode pour éliminer la tendance consiste à calculer la moyenne mobile :

$$Z_t = \frac{1}{2l+1} \sum_{i=t-l}^{t+2} Y_t$$

Ainsi, on remplace chaque valeur par la moyenne des $2l+1$ valeurs qui l'entourent. On peut ainsi éliminer toutes les anomalies ainsi que le bruit. La moyenne mobile correspond donc à la tendance de la série des $Y_t$, donc $Z_t = T_t \Rightarrow X_t = Y_t - Z_t$.

On peut donc extraire la partie stationnaire de la série avec laquelle on fait les prédictions. On réinjecte ensuite la tendance en interpolant la moyenne mobile par un polynôme puis on ajoute à tous les termes prédits l'évaluation au temps correspondant.

### 1.5.5 Elimination de la saisonnalité

Soit $(Y_t)$ une série temporelle telle que $Y_t = s_t + X_t$ où $s_t$ est une composante déterministe périodique de période $p$ et $X_t$ est une série stationnaire.
Pour éliminer la saisonnalité, on peut comme pour la tendance faire une différenciation mais cette fois-ci de degré $p$ *i.e.* on pose $Z_t = Y_t - Y_{t-p}\ \forall t \geqslant p$.
En effet, on a:

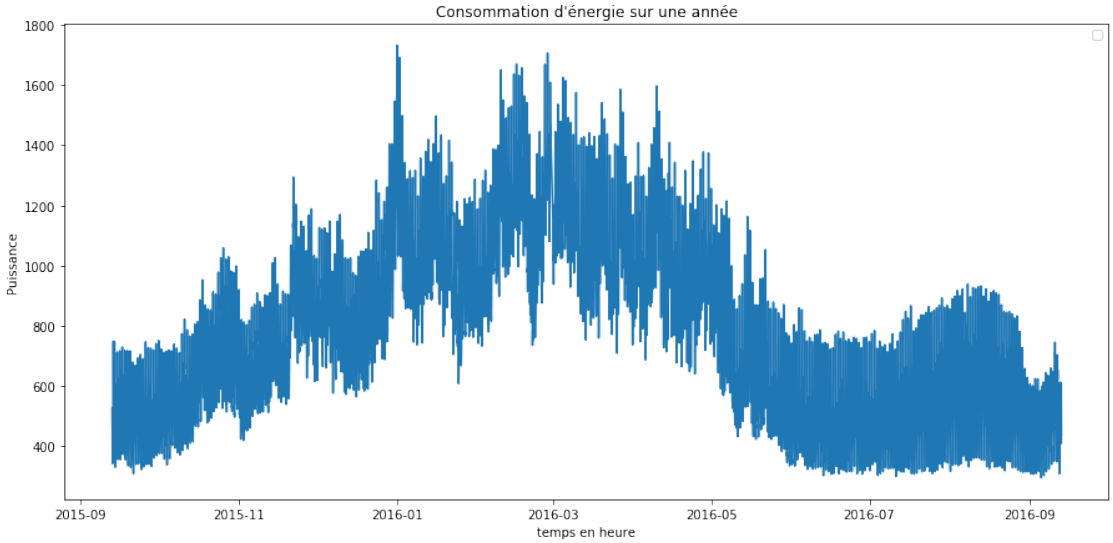
$Z_t = Y_t - Y_{t-p} = s_t - s_{t-p} + X_t - X_{t-p} = s_t - s_t + X_t - X_{t-p}$(car $s_t$ est périodique de période $p$).
$= X_t - X_{t-p}$, or $(X_t)$ est stationnaire donc $(Z_t)$ est aussi stationnaire

Une fois les prédictions effectuées, on peut retrouver la série initiale en effectuant une différentiation inverse itérative comme décrit précédemment avec les p premiers éléments initialisés.

## 1.6 Applications d'ARMA aux données d'électricité

Nous connaissons à présent le modèle ARMA ainsi que les différentes applications qu'on peut en faire. Il s'agit maintenant de réaliser une prédiction par le modèle sur le dataset des consommations électriques sur une semaine.

Dans un premier temps, il faut observer le dataset afin de décider quelles opérations faire. Celui-ci est composé de deux variables. La **date** sous le format AAAA-MM-JJ hh:mm:ss , et la **puissance**, c'est à dire l'expression de la consommation eélectrique de la population de l'Ile. La consommation électrique sur l'année 2015-2016 est un ensemble de points indexés par ordre chronologique. On l'exprimera donc sous forme de série temporelle.




On se rend compte immédiatement que la série temporelle n'est pas stationnaire, et qu'il sera donc nécessaire de la stationnariser avant de pouvoir appliquer le modèle ARMA. Ensuite, il faut trouver les bons ordres $p$ et $q$ du modèle ARMA afin d'avoir une prédiction optimale. Finalement, il faut réaliser la prédiction. On va donc procéder en 3 étapes :

- Choisir une méthode inversible pour avoir une série temporelle stationnaire
- Trouver l'ordre $(p, q)$ optimal de ARMA$(p, q)$
- Réaliser la prédiction

Avant toute chose, nous implémentons cette prédiction sur Python 3.6 à l'aide des bibliothèques Numpy et Pandas qui sont particulièrement utiles pour les dataset. On importe donc les bibliothèques suivantes :

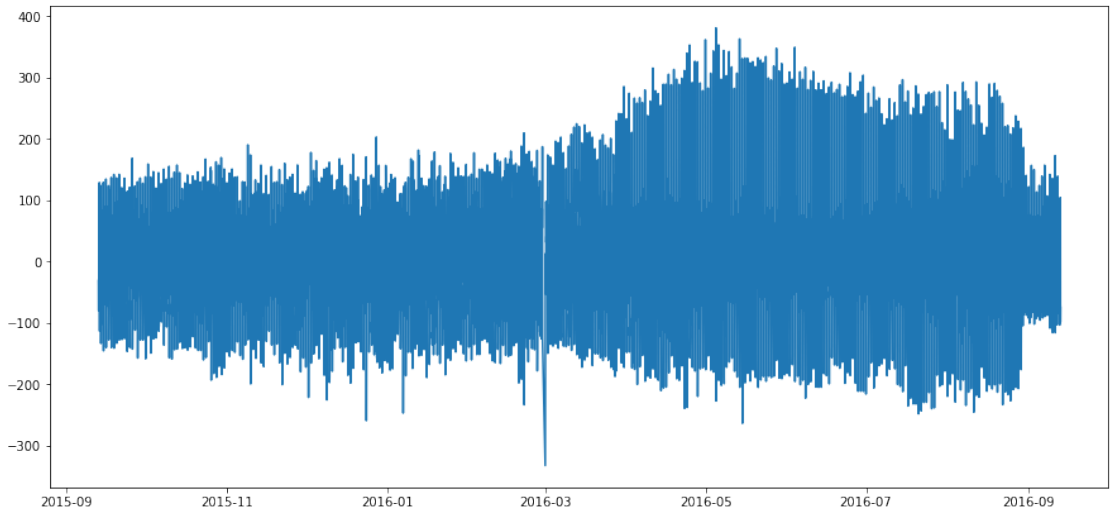
```
In [2]: import pandas as pd
        import numpy as np
        from matplotlib import pyplot as plt
        from statsmodels import api as sm
        import statsmodels
```

### 1.6.1 Première approche par différenciation

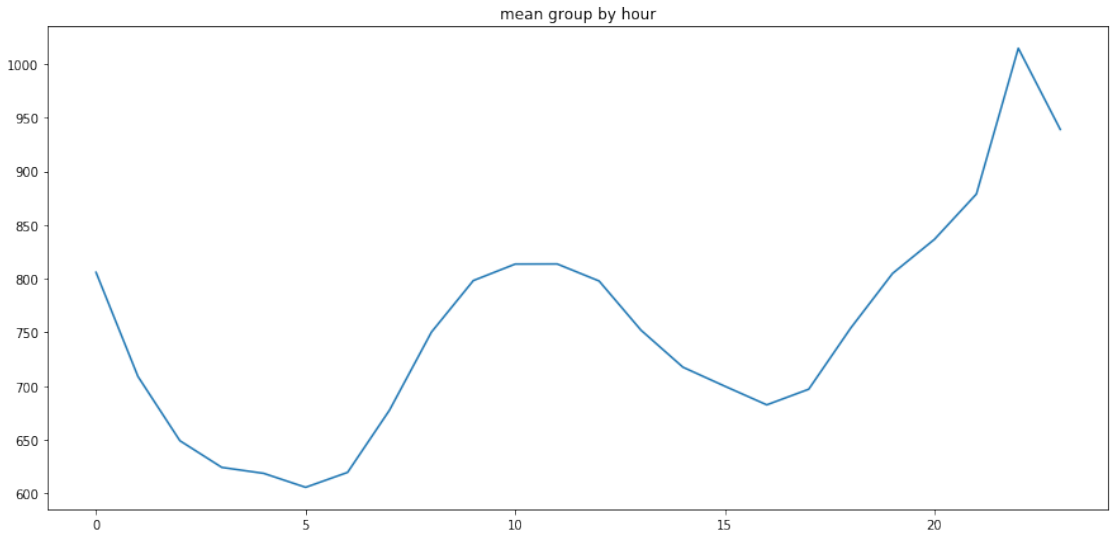
On réalise dans un premier temps une différenciation, pour les raisons ci-dessous. On se rendra compte à la fin qu'un problème appraraît.

**1.6.1.1 Stationnarisation du dataset** Nous supposons que notre série $Df_t$ suit un modèle additif, c'est donc la somme de sa tendance $T_t$, de sa saisonnalité $S_t$ et d'une erreur $E_t$.

Nous allons tout d'abord retirer la tendance en faisant une différenciation. Nous effectuons l'opération de différenciation $D_t - D_{t-1}$.




Puis nous traitons la saisonnalité. On suppose que la série initiale avait une composante saisonnière journalière, ce qui a un sens dans le quotidien des consommateurs dont la consommation électrique la nuit est relativement basse par rapport à celle de la journée. La consommation décroit de minuit à 6h du matin, puis croit jusqu'à 11h et subit un léger creux jusqu'à 16h pour enfin remonter encore plus.




Pour traiter la saisonnalité, nous avons calculé les coefficients de saisonnalité en prenant la moyenne de chaque heure sur l'année. Si ces coefficients sont égaux, alors il n'y a pas de saisonnalité, sinon on retire à chaque heure de la série différenciée son coefficient respectif.

Enfin nous traitons le pic qui arrive au mois de mars 2016 en remplaçant cette valeur par $-200$ .

Finalement on procède au test de stationnarité et on utilise l'indicateur des **pvalues** qui est bien inferieur à 0.05 et qui nous permet donc de continuer avec cette série stationnaire.

```
ADF Statistic: -17.867720
p-value: 0.000000
Critical Values:
        1%: -3.431
        5%: -2.862
        10%: -2.567
```

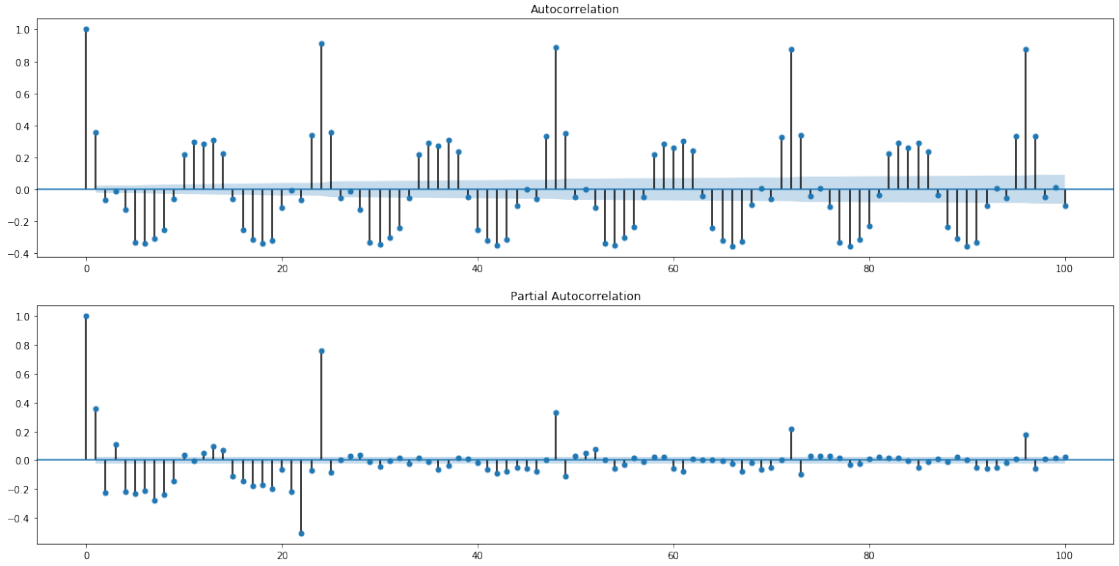
**1.6.1.2 Choix de l'ordre (p,q) optimal** Les ordres $p$ et $q$ des modèles $AR$ et $MA$ peuvent être obtenus en étudiant les autocorrélations et autocorrélations partielles. Bien souvent, ces ordres n'apparaissent pas de manière évidente. On peut dans ce cas obtenir des bornes supérieures pour $p$ et $q$ puis sélectionner un modèle en minimisant des critère de type $AIC$ ou $BIC$. Nous allons ici utiliser le critère $AIC$.

L'$AIC$, Akaike information criterion, se calcule ainsi :

$$AIC = 2(p+q) - 2ln(L)$$

où $p$ et $q$ sont les paramètres du modèle et $L$ est le maximum de la fonction de vraisemblance du modèle.

Le modèle ayant l'$AIC$ le plus faible sera donc celui que nous utiliserons.



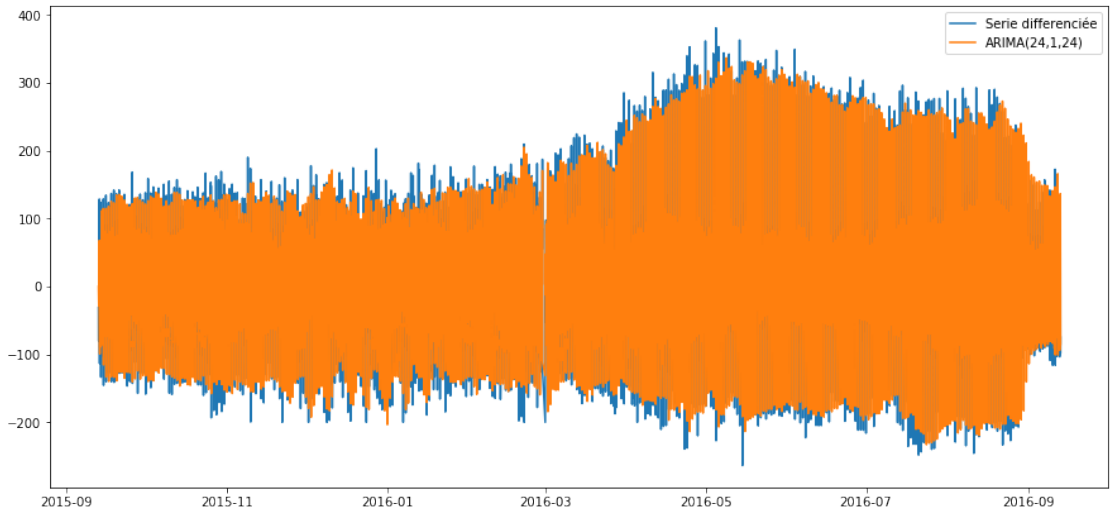

Les auto-corrélations partielles nous indiquent que la partie $AR$ du processus $ARMA$ aura sûrement un paramètre $p$ proche de 24.
Les auto-corrélations nous indiquent que la partie $MA$ aura un paramètre $q$ proche lui aussi de 24.
Pour déterminer ensuite précisément les paramètres du modèle $ARMA(p,q)$, on utlise les fonctions :

```
model = ARIMA(df_diff, order=(p,0,q))
results_ARMA = model.fit(disp=1)
current_aic = results_ARMA.aic
```

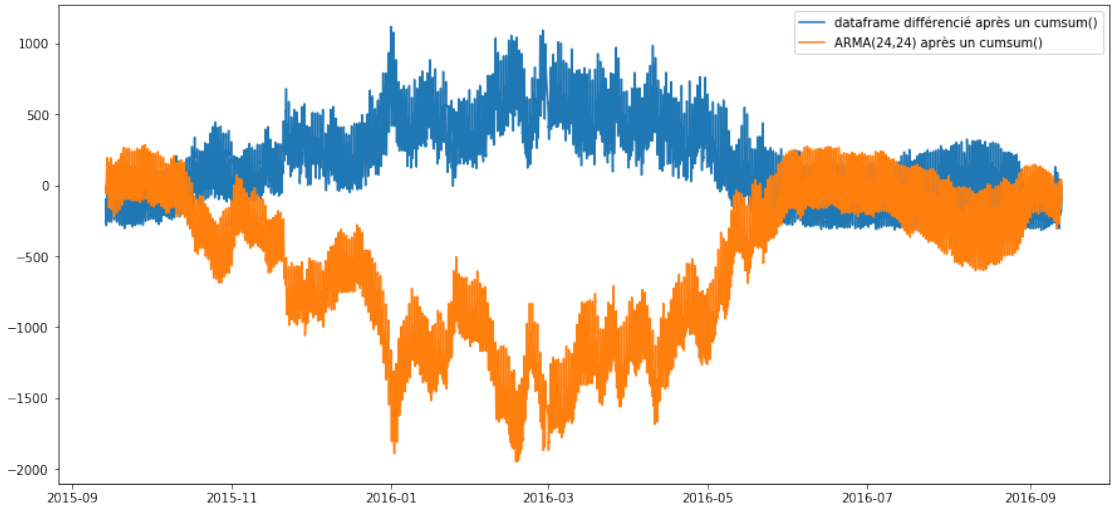
On trouve alors que le meilleur modèle est $ARMA(24, 24)$.

Comme nous savons que la série a une composante saisonnière journalière (de 24h), cela semble logique que les paramètres de notre modèle soient $(24, 24)$. En effet la consommation à l'heure $h$ est lié à $h - 24$.

**1.6.1.3 Utilisation et résultats du modèle ARMA** On utilise le modèle $ARIMA(24, 1, 24)$ sur la série initiale qui n'est autre que le modèle $ARMA(24, 24)$ sur la série différenciée.




On peut voir que le modèle $ARMA(24, 24)$ s'accorde bien avec les valeurs de la série différenciée.
Cependant, en essayant de faire l'opération inverse de la différentiation sur le modèle $ARMA(24, 24)$ , on s'aperçoit que le résultat est l'inverse de celui qu'on espérait. On affiche ci-contre $ARMA(24, 24)$ après l'opération inverse de la différentiation ainsi que le dataframe différencié ayant subit la même opération.



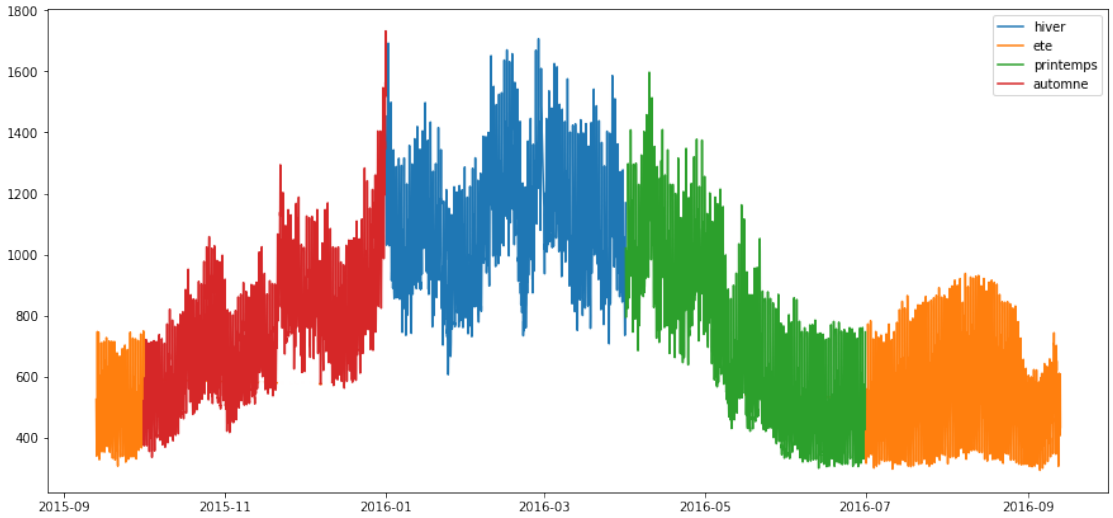

Le dataframe différencié après l'opération **cumsum()** est égal au dataframe de départ à une constante près. Pour revenir aux vraies valeurs, il suffirait de rajouter partout la valeur initiale du dataframe.
Bien que la différenciation semble la bonne manière de rendre la série stationnaire, cela ne permet pas d'avoir une bonne approximation des vraies valeurs. Nous allons donc nous tourner vers une autre méthode pour rendre cette série temporelle stationnaire.

### 1.6.2 Seconde approche

Nous allons cette fois-ci chercher une approximation de la tendance ou de la saisonnalité pour ensuite la retirer afin de rendre la série stationnaire. Tout comme précédemment, on raisonne en 3 étapes :

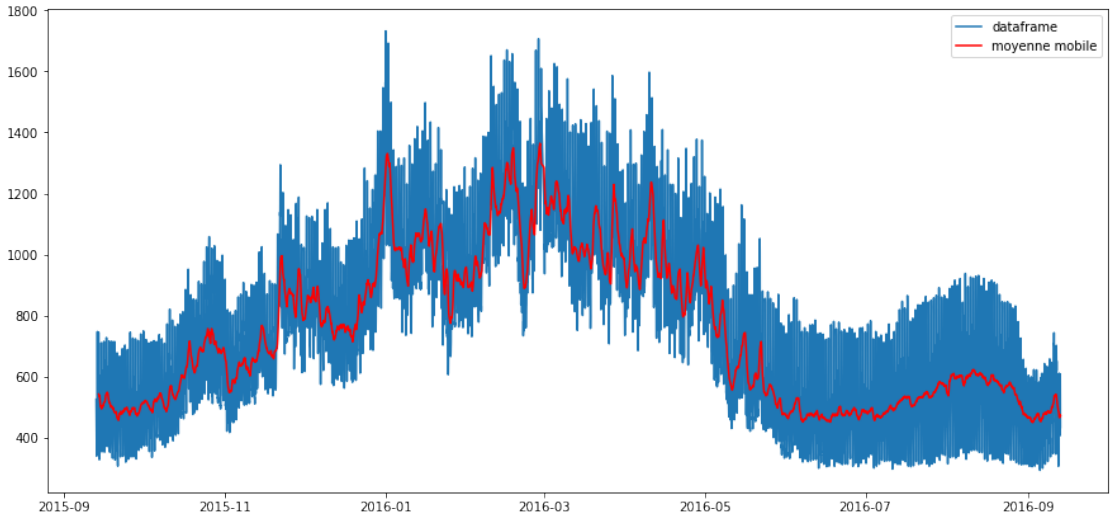
- Choisir une méthode inversible pour avoir une série temporelle stationnaire
- Trouver l'ordre $(p, q)$ optimal de $ARMA(p, q)$
- Réaliser la prédiction

**1.6.2.1 Stationnarisation** À première vue, la consommation d'électricité est plus élevée en Hiver qu'en Été. De plus, durant le Printemps la consommation diminue, et augmente pendant l'Automne.
Cela devient très clair lorsqu'on découpe la série en suivant les 4 saisons :




On reconnaît une forme sinusoïdale pour la tendance annuelle. Notre but est alors de trouver une tendance qui confirme notre intuition.

Pour se rapprocher de la tendance, nous allons utiliser la Moyenne Mobile qui vaut en chaque point la moyenne des valeurs aux alentours. C'est donc ce qui se rapproche de la tendance et voici son aspect :




Maintenant essayons d'approximer la tendance avec une méthode d'interpolation.

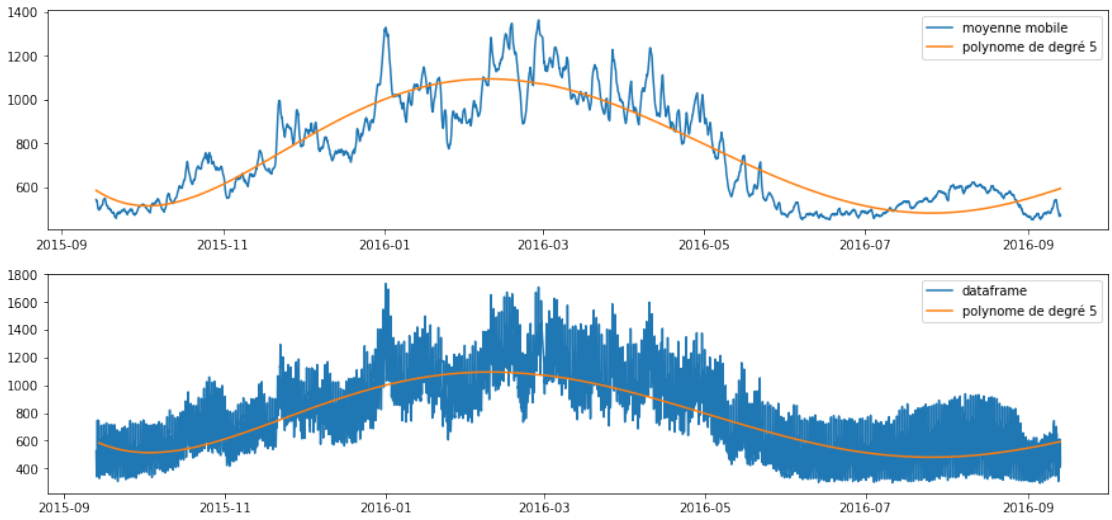
```
In [67]:
        taille = len(moving_avg.axes[0])
        x = np.linspace(0,taille,taille)
```

```
y = np.array([moving_avg['puissance'][i] for i in range(taille)])
approx_ma = np.poly1d(np.polyfit(x,y,5))
```

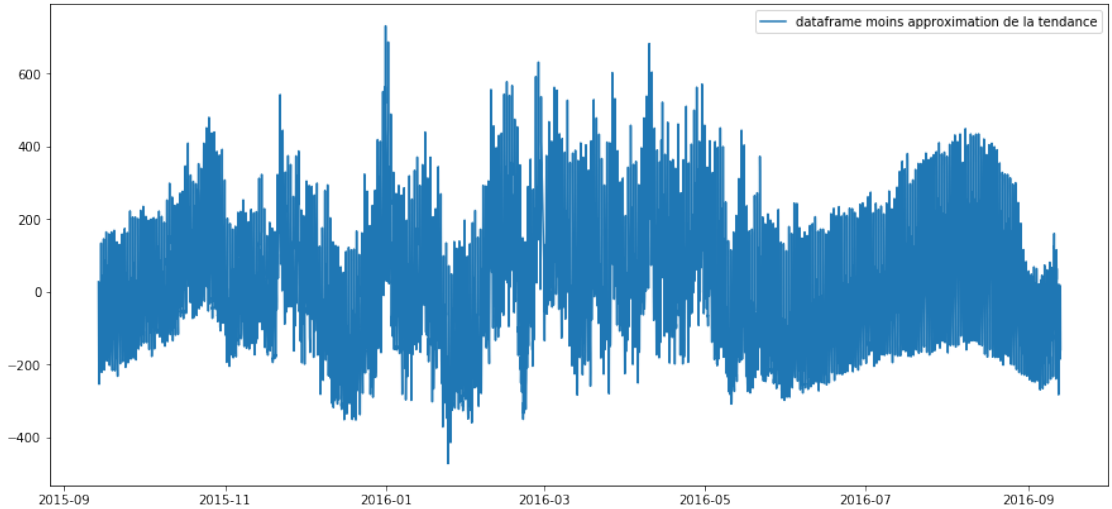
Nous utilisons un polynôme de degré 5, il nous semble que la saisonnalité anuelle ressemble à un sinus dont le développement limité est :

$$\sin(x) \underset{x\to 0}{=} x - \frac{x^3}{3!} + \frac{x^5}{5!} + o(x^6)$$

Ce polynôme correspond à nos attentes :




Ensuite, l'étape la plus importante est de rendre la série stationnaire. On travail à présent sur le Dataframe auquel nous avons retiré la tendance :

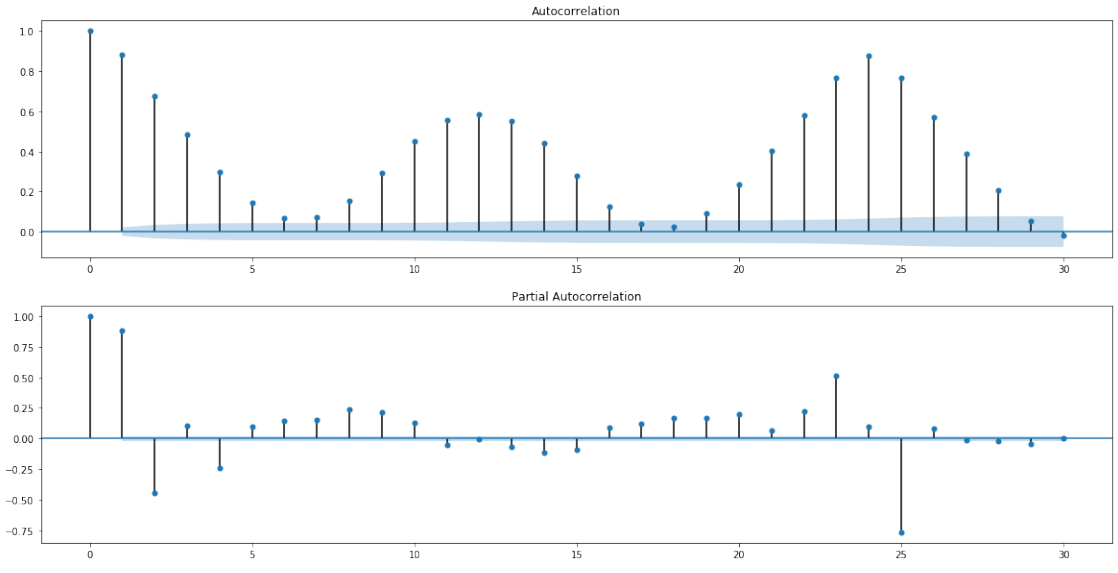


Voyons à l'aide du test de Dicker-Fuller augmenté, si le Dataframe auquel nous avons retiré la tendance est stationnaire :

```
ADF Statistic: -6.275326
p-value: 0.000000
Critical Values:
        1%: -3.431
        5%: -2.862
        10%: -2.567
```

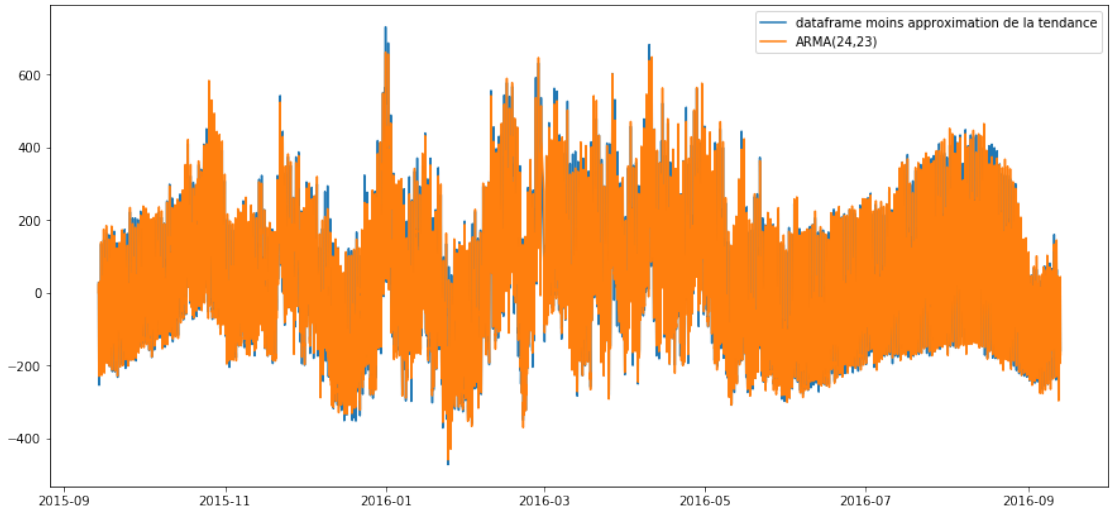
Les $pvalues < 0.05$ nous indiquent que la série est stationnaire. Il nous faut maintenant déterminer quel modèle concorde avec celle-ci.

**1.6.2.2 Choix de l'ordre (p,q) optimal** Comme nous l'avons vu précèdemment la détermination des paramètres du modèle se fait à l'aide du graphe des auto-corrélations et celui des auto-corrélations partielles, ainsi que l'$AIC$ de chaque modèle.

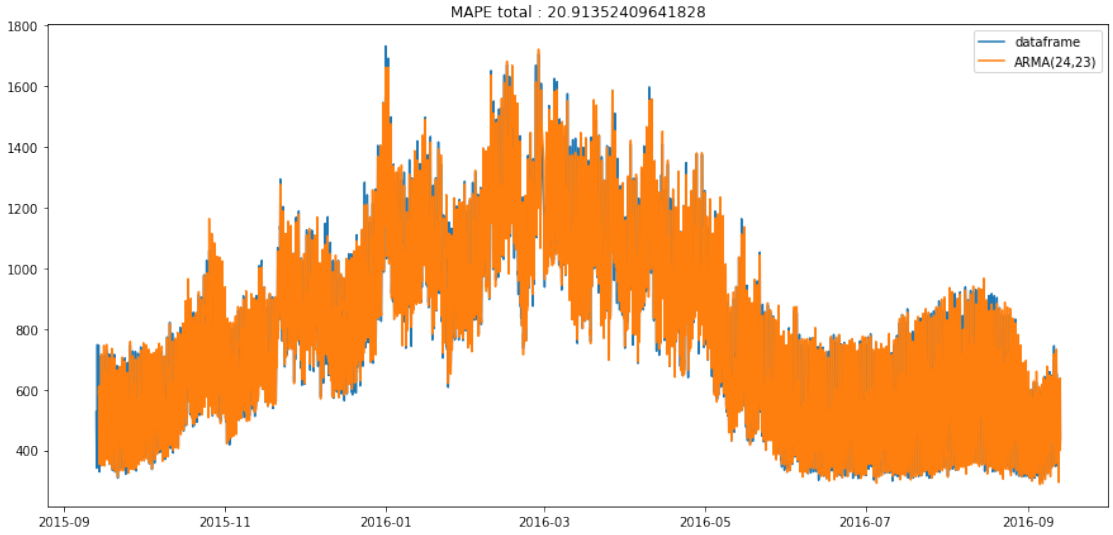



On déduit de ces graphes que l'ordre maximal de $AR(p)$ sera au maximum 25 et que l'ordre de $MA(q)$ sera environ 24.
Après une utilisation du critère d'Akaike, on trouve que le modèle le plus adapté est $ARMA(24, 23)$. On retrouve encore la saisonnalité journalière (24h).

**1.6.2.3 Modèle ARMA et prédiction** Cette fois-ci l'approximation du modèle semble être meilleure que l'approximation pour la différentiation :

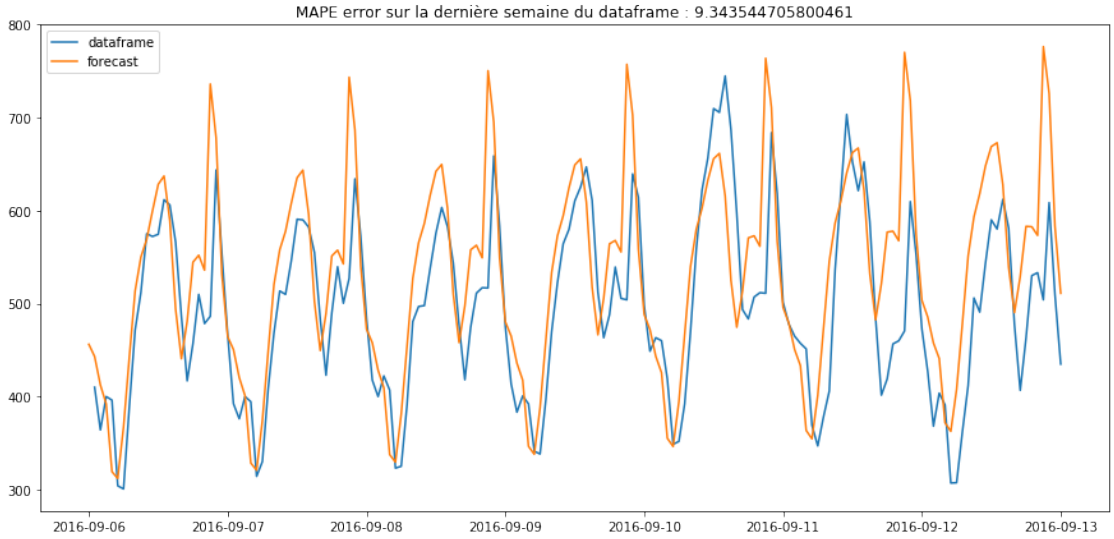



Voyons comment se comporte notre modèle lorsqu'on lui rajoute l'approximation de la tendance :



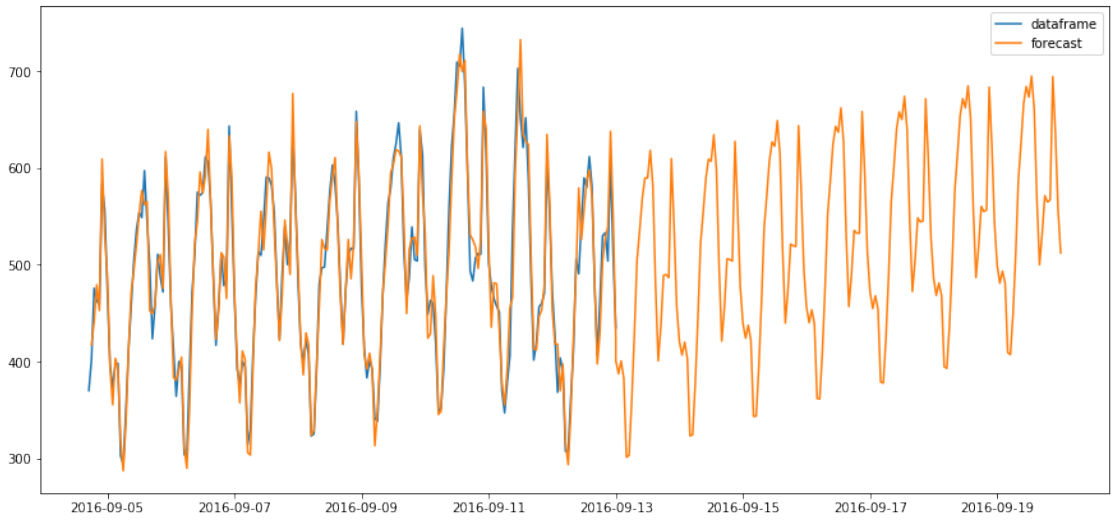

Pour notre plus grand bonheur, le modèle répond à nos attentes et ressemble énormément au Dataframe initial. On peut noter que le titre de ce graphe est la $MAPE$ (Mean Absolute Percentage Error) qui nous indique à quel point le modèle est proche de la série initiale.
On s'intéresse maintenant aux prédictions. Une dernière étape de vérification pour juger

de la valeur de notre modèle est de l'entraîner sur tout le Dataframe auquel on retire la dernière semaine.
Ensuite, on fait une prédiction d'une semaine, et on compare les résultats de la prédiction avec les vrais données pour estimer dans quelles mesures nous pouvons faire confiance à nos resultats.




On remarque que les résultats sont proches des vraies données et la MAPE nous indique une erreur de l'ordre de 9% ce qui est tout à fait raisonnable. On conclut donc que le modèle est adéquat et que les prédictions sont assez bonnes.

Affichons maintenant les valeurs prédites pour la semaine suivante :



# 2 Machine Learning et applications

Aujourd'hui, le *Machine learning* est un centre d'intérêts car on peut l'utiliser pour résoudre des problèmes jusqu'ici insolvables, notamment grâce aux réseaux de neurones.

## 2.1 Première approche en Machine Learning : Support Vector Regression

En même temps que l'on comprenait les réseaux de neurones, nous avons étudié en parallèle une autre méthode de Machine learning appelée Support Vector Regression.

### 2.1.1 Principe

Le principe du SVR *(support Vector Regression)* est d'effectuer des prédictions grâce à une regression linéaire entre nos *features* (attribut d'entrées) et nos *labels* (prédiction).
On cherche donc une fonction qui associe à chaque *feature* son *label*.
Pour ce faire on cherche une relation linéaire qui prend en entrée nos *features* et dont l'image est la plus proche possible de nos *labels*. On peut par exemple utiliser la méthode des Moindres Carrés qui mimimise la somme des erreurs au carré ou encore la méthode de Laplace-Boscovich qui minimise la somme de la valeur absolue des erreurs. Si on considère que le modèle est effectivement linéaire et que les données ont bien été collecté, alors les erreurs sont des bruits gaussiens, et donc des variables aléatoires de densité :

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}.$$

Donc la somme des erreurs est aussi une variable aléatoire qui suit une loi gaussienne (en tant que somme de gaussienne) de densité:

$$f(x_1, .., x_n) = \frac{1}{\sqrt{2\pi}\sigma}^{n} e^{-\frac{\sum x_i^2}{2\sigma^2}}.$$

Dans ce cas, minimiser l'erreur revient à minimiser la somme des erreurs au carré donc la méthode des carrés semble la plus optimale. Cependant, si on omet l'hypothèse que toutes les erreurs sont de simples bruits (par exemple si certaines valeurs sont aberrantes) , le modèle de Laplace-Boscovich peut être plus robuste.

Cependant, si le modèle n'est pas linéaire quelque soit la méthode utilisée nous ne pourrons pas approximer optimalement la fonction recherchée. Pour palier à ce problème nous utilisons le *Kernel tricks*.
La méthode du *Kernel tricks* consiste à augmenter la dimension de l'espace de départ en rajoutant par des applications non-linéaires l'image des actuelles *features*, jusqu'à obtenir un espace de Hilbert (espace vectorielle muni d'une norme) où le modele est linéaire.

Ainsi, si on pose $< .,. >$ le produit scalaire de l'espace de Hilbert et $f$ la composée de toutes les applications linéaires que nous avons utilisé pour obtenir cet espace, on obtient : $K =< f(x), f(x') >$.On appelle $K$ une fonction noyau. Cette applcation est non linéaire dans notre espace de départ mais elle l'est dans l'espace du produit scalaire. On peut donc avec cette méthode faire des prédictions sur tout type de modèle.
En pratique les fonctions noyaux les plus utilisés sont :

*RBF* (Radial Basis Function) : $K(x, x') = e^{\frac{||x-x'||^2}{2\sigma}}$ avec $\sigma$ un paramètre qui dépend du modele et $||.||$ la norme euclidienne.

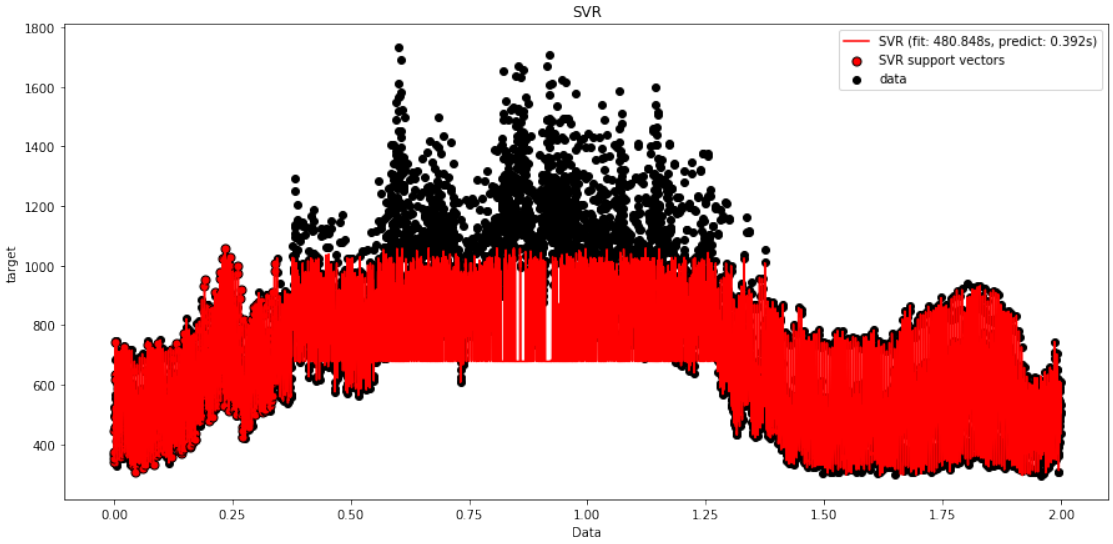
*Noyau polynomiale* : $K(x, x') = (< x, x' > +c)^d$ où $d$ est le degré du polynome et $c$ une constante associée au modèle.

### 2.1.2 Résultats

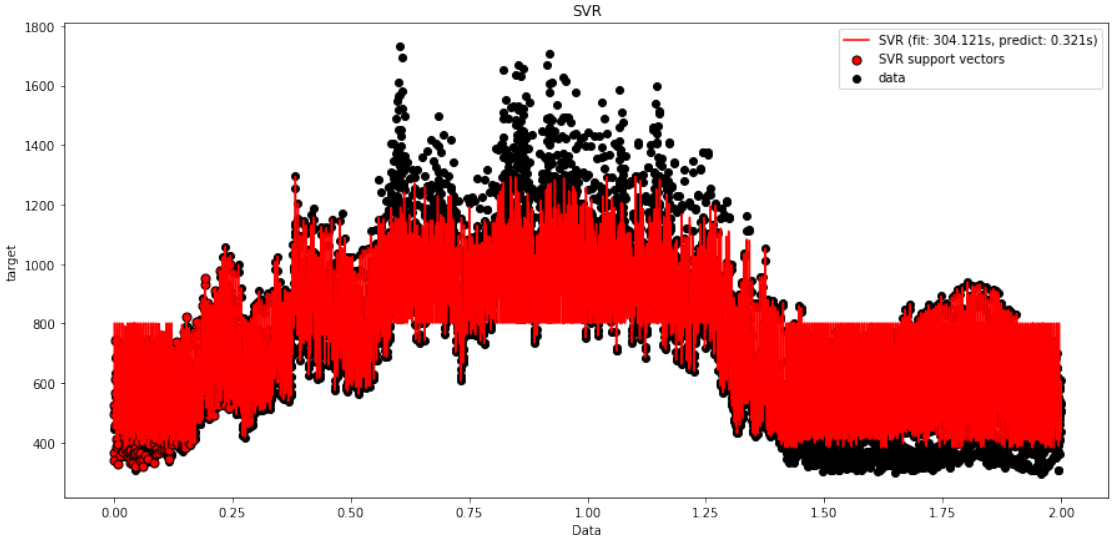
Nous avons donc effectué des prédictions avec le procédé de machine learning SVR en utilisant le Kernel *RBF*. Ce Kernel Gaussien prend différents paramètres qui sont optimisés grâce à la fonction *GridSearchCV* de la bibliothèque Model *Selection* pour choisir le paramètre $C$ par exemple qui lorsqu'il est élevé pénalise fortement les erreurs.

```
train_size = 3000
svr = GridSearchCV(SVR(kernel='rbf', gamma=0.9),cv=12,param_grid={"C": [1e0, 1e1, 1e2, 1e5],"gamma": np.logspace(-2, 2, 5)})
```

Nous avons entrainé notre modèle sur les 1500 valeurs de la série en commencant à differents endroits.
En commencant l'entrainement au début de la série nous remarquons que la prédiction de la derniere semaine est bonne mais que durant la periode de Janvier à Juin la prediction est très mauvaise.
On comprends aisément que la dernière semaine à prédire (en septembre 2016) est très similaire à la première (Septembre 2015) et donc que les resultats sont bons.




En décallant la zone d'entrainement on obtient une meilleure prédiction au long de l'année mais une précision moins bonne pour la dernière semaine, ce qui est compréhensible en suivant l'argument précédent.




Finalement, nous aurions pu entrainer notre modèle sur plus de valeurs, mais il nous a semblé inefficace de le faire car pour avoir des résultats cohérents au long de l'année il fallait entrainer notre modèle sur la moitié de celle-ci. Nous avons donc préféré nous orienter vers les réseaux de neurones qui nous semblaient plus appropriés.

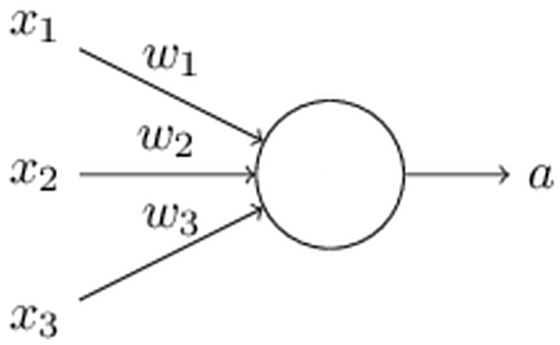
## 2.2 Réseau de neurones

Les réseaux de neurones sont une branche majeur du *Machine learning* dont le champs d'application est très vaste. Dans l'objectif de prédire notre série temporelle à l'aide de ce procédé, nous allons tout d'abord définir et comprendre ce qu'est un réseau de neurones et l'appliquer à notre série temporelle.

L'étude des reseaux de neurones constitue un vaste domaine scientifique.. Il existe de manière générale deux types de réseaux de neurones : les réseaux de classification et les réseaux de prédiction. Dans les deux cas, l'architecture du réseau reste la même et nous allons donc étudier celle-ci.

### 2.2.1 Perceptron

Un perceptron est une fonction de $n$ arguments binaires qui retourne un résultat. On peut le voir simplement par cette image :




Le résultat obtenu par le perceptron est définit par les poids attribués à chaque argument en entrée. Par exemple, ici, l'entrée $x_1$ à un poids $w_1$, et de même pour $x_2$ et $x_3$ avec $w_2$ et $w_3$.

Le poids représente l'importance de la variable auquel il est associé. Par exemple, si $x_1$ a un poids plus elevé que $x_2$ et $x_3$, cela signifie que cette variable impact de façon plus importante le résultat final que les autres variables.
Ensuite, le résultat est calculé selon un seuil de la manière suivante :

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ seuil} \\ 1 & \text{if } \sum_j w_j x_j > \text{ seuil} \end{cases}$$

Ici, si le résultat est 0, alors :

$$w_1 x_1 + w_2 x_2 + w_3 x_3 \leq seuil$$

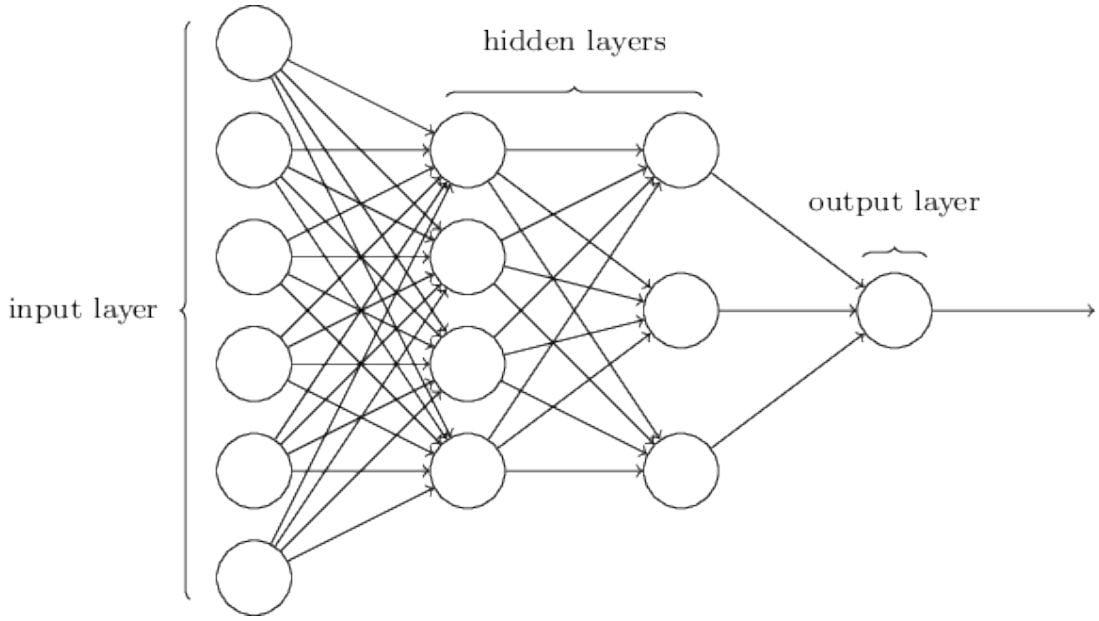
De manière générale, on choisit d'utiliser un biais $b = -seuil$ et on réecrit l'équation de la manière suivante :

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j + b \leq 0 \\ 1 & \text{if } \sum_j w_j x_j + b > 0 \end{cases}$$

Bien qu'un perceptron prenne en entrée des variables binaires et retourne un résultat binaire. Il peut permettre, si on utilise plusieurs couches (sous forme de réseau), de résoudre un grand nombre de problemes complexes.

### 2.2.2 Réseau de perceptrons

On définit un réseau de perceptrons comme des perceptrons reliés entre eux par couches. On donne en entrée toujours des variables binaires, et la dernière couche peut également être composée de plusieurs perceptrons.




L'exemple ci-dessus est un réseau simple composée d'une *input layer* (couche d'entrée) et deux *hidden layers* (couches cachées) et une *output layer* (couche de sortie).

### 2.2.3 Multi-layer perceptron (MLP)

On appelle *Multi-Layer Perceptron* (MLP) un réseau de perceptrons définit comme précédemment avec pour seule règle que l'information circule de la couche d'entrée vers la couche de sortie. On appelle ce procédé de circulation de l'information *feedforward* (propagation directe).

### 2.2.4 Neurones

Bien que la notion de perceptron paraisse complète, elle ne permet pas de résoudre un grand nombre de problèmes par un simple apprentissage. En effet, les résultats sont binaires alors que de nombreux problèmes ne se résolvent pas par un simple oui ou non. Ainsi, il est nécessaire de définir une variation des perceptrons.

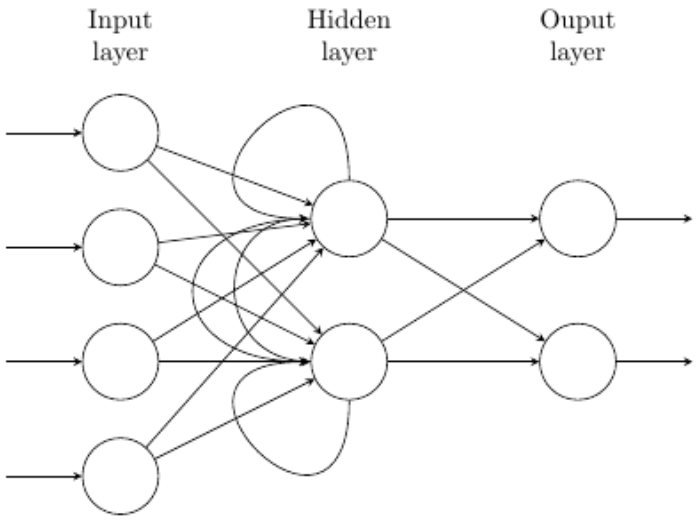
Passer aux réels est très utile pour l'entraînement d'un réseau de neurones. En effet, lorsqu'on cherche à avoir un résultat qui vaut 1, on peut quantifier la précision de notre réseau selon la distance du résultat obtenu avec le résultat espéré. Ceci permet d'introduire trois points essentielles à notre réseau de neurones :

- Permet de quantifier un résultat qui peut être complexe.
- Permet de traiter des problèmes non-linéaires et non binaires.
- Permet à une légère modification de poids d'une variable de modifier seulement un petit peu le résultat (contrairement au saut binaire dans le cas des perceptrons).

On appelle ainsi *neurone* un perceptron de $\mathbb{R}^n$ dans $\mathbb{R}$. Plus simplement, un neurone est un perceptron qui prend en argument des réels, et retourne un réel. On définit également un réseau de neurones de la même manière qu'un réseau de perceptrons.

### 2.2.5 Recurrent neural networks (RNN)

**Recurrent neural network** ou réseau de neurones récurrent sont des réseaux de neurones qui ont la particularité de tirer profit des relations qui peuvent exister entre les événements successifs d'un flux donné. Contrairement aux *MLP*, ce n'est pas un réseau de neurones feedforward. Par exemple, le comportement d'un neurone n'est pas uniquement déterminé par l'activation des valeurs des neurones dans les couches précédentes mais aussi par l'activation de neurones d'autres couches à d'autres temps. Voici un exemple de RNN:




Donc pour un neurone dans un *RNN*, si $f$ est la fonction d'activation associé au neurone, l'output du neurone sera :

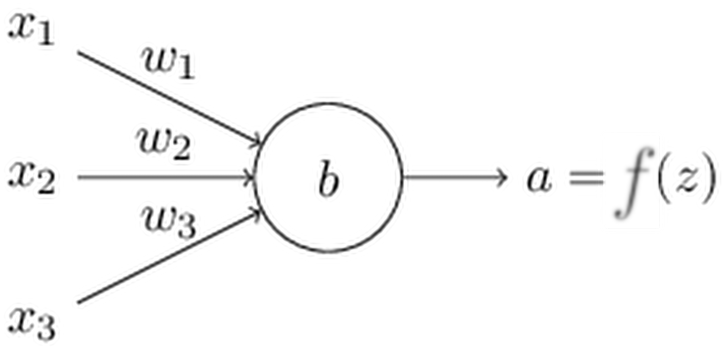
$$f(\sum_j w_j x_j^{(t-1)} + \sum_i w_i x_i^{(t)})$$

Où $x_k^{(t)}$ est la valeur de l'output du neurone $x_k$ au temps $t$. Cela permet donc à un neurone d'être influencé par son output à un temps précédent.

Ce système ne présente aucun problème en apparence pour être utilisé sur des longues séquences de données, cependant, en pratique, on se rend compte que sans ajustement les résultats ne sont pas trés bons. On préfére donc utiliser un type particulier de RNN qui sont les **LTSM** (*Long Short-Term Memory*). Ils sont parfaitement adaptés à la prédictions de données pour les séries temporelles. Nous ne les étudirons pas par manque de temps.

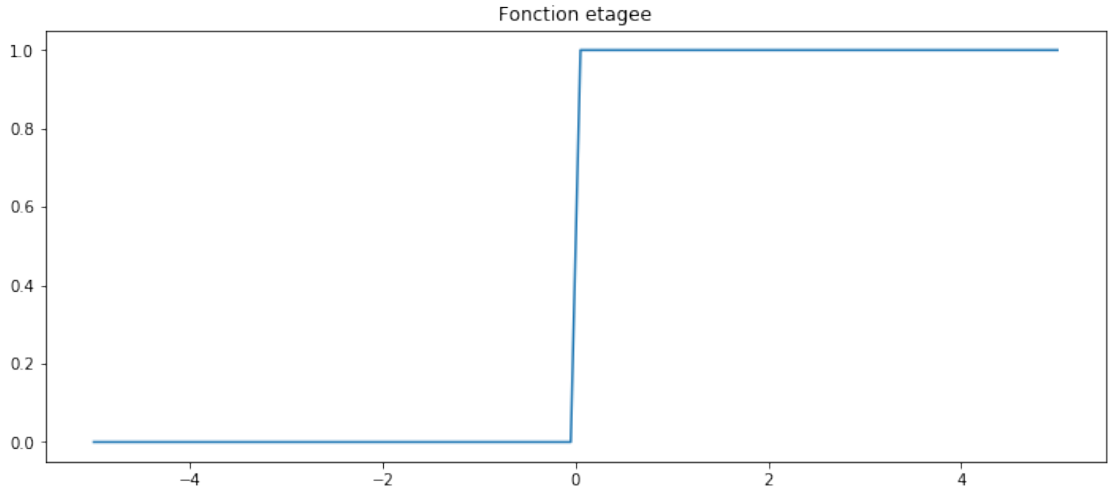
### 2.2.6 Activation function

La fonction d'activation (*activation function*) est la fonction qui définit le comportement du signal renvoyé par un neurone. C'est une fonction de $\mathbb{R}^n$ dans $\mathbb{R}$ :




Où $z = \sum_j w_j x_j + b$, et $f$ est donc la fonction d'activation.

Dans le cas des perceptrons, la fonction $f$ suit la courbe suivante :

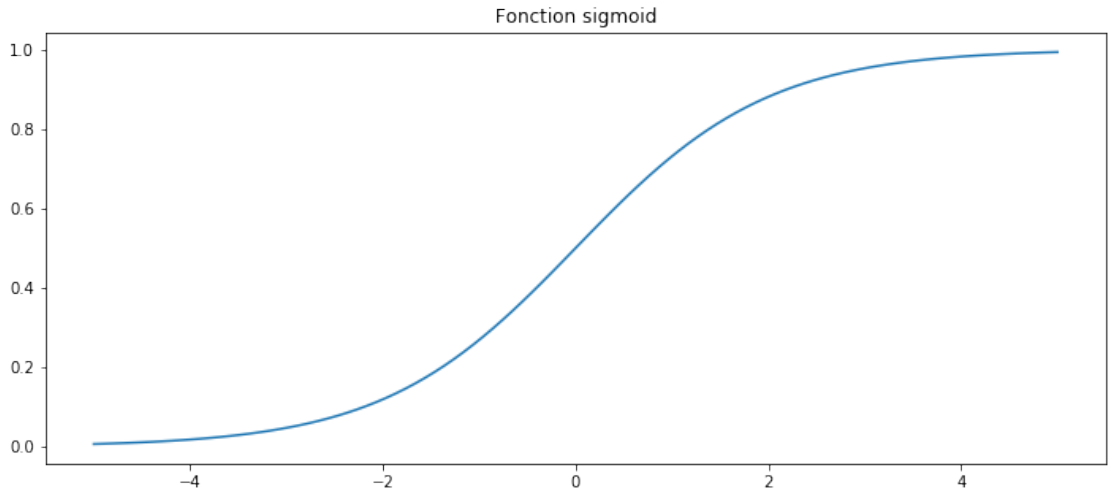



### 2.2.7 Activation functions usuelles

Il existe de nombreuses *activation functions* et nous allons voir quelques exemples connus et très utilisés.

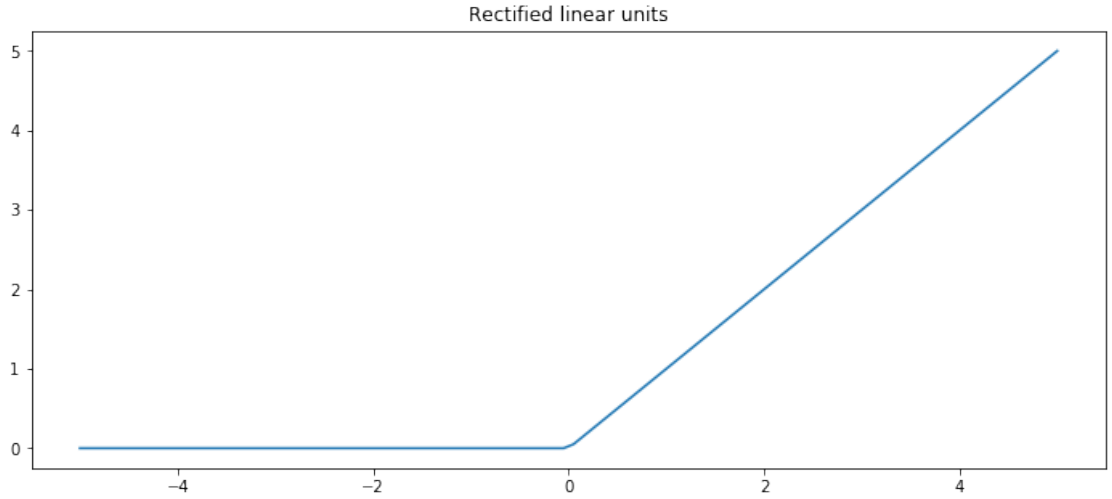
**2.1.7.1 Sigmoid function** La *sigmoid function* est une *activation function* définie de $\mathbb{R}$ dans $[0, 1]$ de la manière suivante :

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$



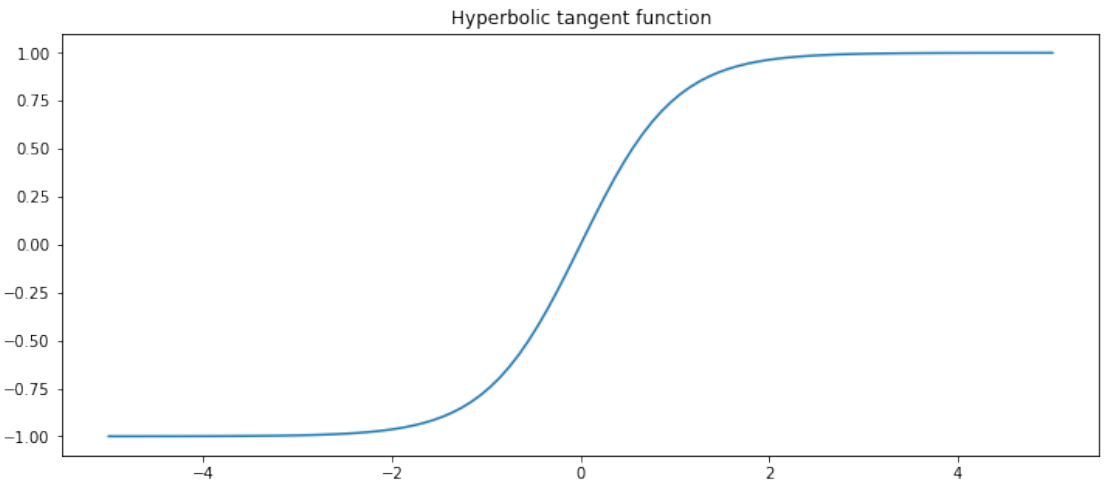

**2.1.7.2 - Rectified linear units function (ReLu)** La *Rectifid linear units function (ReLu)* est une *activation function* très utilisée dans la reconnaissance vocale ou encore reconnaissance d'images. Elle est définie de $\mathbb{R}$ dans $\mathbb{R}^+$ par :

$$f(z) = max(0, z).$$




**2.1.7.3 - Hyperbolic tangent function** La *Hyperbolic tangent function* est une *activation function* définie de $\mathbb{R}$ dans $[-1, 1]$ par :

$$f(z) = tanh(z) = \frac{1 - e^{-2z}}{1 + e^{-2z}}.$$




### 2.2.8 Cost function

La **Cost function** aussi connue comme **Loss function**, en français, fonction de coût, est très importante car c'est elle qui permet de juger d'un résultat fourni par notre réseau de neurones. La fonction de coût est toujours positive et sa valeure est petite lorsque le résultat donnée par notre réseau de neurones est proche de la vraie solution . Voici des exemples de cost functions :

- MSE :

$$C = \frac{1}{2n} \sum_x (y_x - a_x)^2.$$

- Cross-entropy function :

$$C = -\frac{1}{n} \sum_x \left[ y_x \ln a + (1 - y_x) \ln(1 - a_x) \right],$$

Où $n$ est le nombre de données d'entrainements, les $y_x$ et $a_x$ sont respectivements les outputs et les valeurs attendus pour chaque $x$.

### 2.2.9 Backpropagation

Maintenant que l'on a un moyen de mesurer notre erreur, on aimerait pouvoir la diminuer. C'est ici qu'intervient le processus de **backpropagation** (rétro-propagtion).

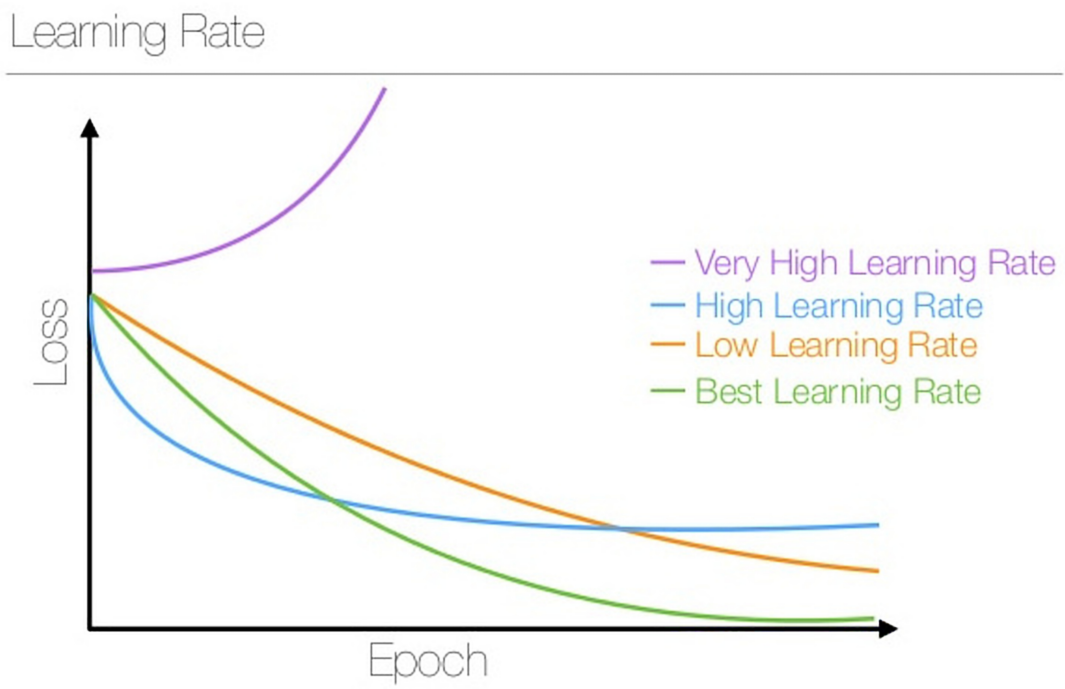
Au coeur de ce processus se situent deux dérivées partielles $\partial C/\partial w$ et $\partial C/\partial b$, où $C$ est la cost function, $w$ représente l'ensemble des poids et $b$ l'ensemble des biais. Ces

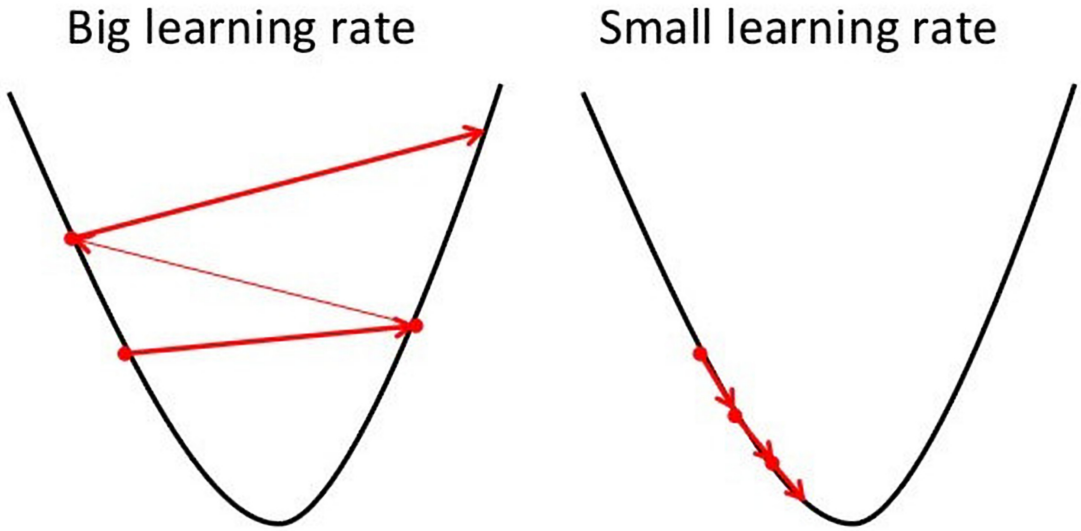
dérivées partielles nous permettent de savoir comment et à quelle vitesse la valeur de la cost function change lorsque l'on fait varier les poids et les biais.

Donc pour diminuer la valeur de la fonction de coût $C$ on fait changer les poids et les biais comme suit:

$$\begin{cases} w \leftarrow w - \eta \frac{\partial C}{\partial w} \\ b \leftarrow b - \eta \frac{\partial C}{\partial b} \end{cases}$$

Où $\eta$ est le learning rate. Le learning rate (taux d'apprentissage) est introduit comme une constante (d'habitude petite) de manière à ce que les poids se mettent à jour lentement pour ne pas rater de minimum local.






### 2.2.10 Partie Calculatoire

Pour comprendre plus en détails comment calculer les dérivées partielles de la fonction de coût prenons un exemple spécifique. Imagininons un réseau de neurones avec seulement trois neurones.

Nous allons nommer la troisième couche (celle qui donne l'output du réseau de neurones) $y^{(L)}$. La couche précedente sera donc $y^{(L-1)}$ et ainsi de suite. On note $b^L$ le biais associé à $y^{(L)}$ et $w^{(L-1)}$ est le poids qui relie $y^{(L-1)}$ à $y^{(L)}$. Notons :

$$z^{(L)} = w^{(L-1)} y^{(L-1)} + b^L \tag{1}$$

D'où :

$$y^{(L)} = \sigma(z^{(L)}) \tag{2}$$

Soit $C_0 = \frac{1}{2}(y^{(L)} - a)^2$ le coût pour l'input $x_0$ où $a$ est l'output désiré. On remarque alors que :

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial y^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial y^{(L)}}$$

Or on voit simplement d'après (1) que :

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = y^{(L-1)}$$

D'après (2) que :

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

Et d'après la définition de $C_0$ :

$$\frac{\partial C_0}{\partial y^{(L)}} = (y^{(L)} - a)$$

Donc dans ce cas spécifique voici l'effet qu'aura la backpropagation sur $w^{(L)}$ :

$$w^{(L)} \leftarrow w^{(L)} - \eta[y^{(L-1)}\sigma'(z^{(L)})(y^{(L)} - a)]$$

On n'a volontairement pas détaillé $y^{(L-1)}$ en fonction de la couche qui le précède de manière à simplifier la compréhension. Mais lorsque l'on se trouve dans réseau de neurones avec beaucoup de couches possédant beaucoup de neurones les équations ne sont en fait pas beaucoup plus compliquée. $C_0$ peut alors s'écrire :

$$C_0 = \sum_{j=0}^{n_L-1} \frac{1}{2}(y_j^{(L)} - a_j)^2 \text{ où } n_L \text{ est le nombre de neurones sur la couche } L$$

Le poids reliant le $k^{ieme}$ neurone de la couche $(L-1)$ au $j^{ieme}$ neurone de la couche $(L)$ sera noté $w_{jk}^{(L)}$ et la dérivée partielle de la fonction de coût par rapport à ce poids devient simplement :

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial y_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial y_j^{(L)}}$$

Pour arriver à faire sur l'ensemble de réseau il manque juste à savoir que :

$$z_j^{(L)} = w_{j0}^{(L-1)} y_0^{(L-1)} + w_{j1}^{(L-1)} y_1^{(L-1)} + ... + b_j^L = \sum_{m=0}^{n_L-1} w_{jm}^{(L-1)} y_m^{(L-1)} + b_j^L \tag{3}$$

L'ensemble des résultats trouvés précédement permettent de faire tous les calculs nécessaires pour trouver comment faire varier chaque poids d'un réseau. La méthode pour trouver comment faire varier les biais est similaire.

### 2.2.11 Hyperparmeters

Les **hyper-paramètres** (*hyperparameters*) sont ce qui définit le réseau de neuronnes. Ils déterminent le nombre de couches cachées, la vitesse d'apprentissage ($\eta$), taille des mini-batches, fonction de coût choisi, initialisation des poids, etc.

Toute la recherche d'optimisation d'un réseau de neurone, pour améliorer ses résultats se fait dans la recherche des choix optimaux pour chaque hyper-paramètre. Nous allons utilser la méthode du *grid search* pour trouver les hyper paramètres. C'est la méthode usuelle qui consiste à faire une recherche quasi-exhaustive des combinaisons entre les paramètres possible.

Un des objectif du choix des hyperparamètres est d'éviter l'$overfitting$. L'overfitting ou surentraînement, se produit quand le réseau de neurones commence à s'adapter aux spécificités des données d'entraînement. Quand cela se produit, on a de très bons résultats sur les données d'entraînement, mais de très mauvais sur des données que le réseau de neurones n'a pas encore vu. Nous verrons quelle méthode nous avons choisis pour éviter l'overfitting dans la partie pratique.

## 2.3 Nos résultats par la méthode des réseaux de neurones

Maintenant que nous savons ce qu'est un réseau de neurones ainsi que le fonctionnement de celui-ci, nous pouvons appliquer ces connaissances afin de prédire notre série temporelle.

### 2.3.1 Application simple du procédé

Dans un premier temps, nous avons réalisé des apprentissages sans données météos qui nous ont été fournies plus tard. Ainsi, voyons dans un premier temps ces résultats là. On profite également pour expliquer le fonctionnement d'un réseau de neurones en pratique.

Tout d'abord, on pourrait coder nous-mêmes un réseau de neurones sur Python à partir de rien, mais ça demanderait bien plus de travail, alors qu'il existe des bibliothèques spécialisées dans ce domaine et très optimisées afin de se préoccuper uniquement de la recherche de nos hyper-paramètres optimaux.

Nous avons choisi de faire un réseau de neurones qui fonctionne de la manière suivante : le réseau de neurones prendra en entrée une suite de valeurs des jours précédents et possiblement des informations sur le jour actuel (ex: heure du week-end, données météorologique ou autre). Ensuite nous choisirons tous les hyper paramètres en cherchant à minimiser l'erreur MAPE sur la dernière semaine de notre dataframe.

On commence par faire une fonction qui prépare les données pour entraîner notre réseau, car comme dit précèdemment, pour prédire chaque donnée, il faut les informations nécéssaires à chaque neurone de la première couche. Donc, voici comment se

présente la fonction permettant d'avoir en entrée le réseau de neurones les *step* valeurs précédentes pour chaque valeur de $data_y$:

```
In [11]: def prepare_data(data_y, step):
             x = []
             y = []
             for i in range(step,len(data_y)):
                 x.append(data_y[i-step:i])
                 y.append(data_y[i])
             return np.array(x),np.array(y)
```

Ensuite, on importe les bibliothèques qui permettent de gérer notre réseau de neurones et on choisit les données sur lesquelles on entraine le réseau :

```
In [37]: from keras.models import Sequential, Model
         from keras.layers import Dense, Flatten, Input
         from keras import optimizers
         from keras.callbacks import EarlyStopping, ModelCheckpoint

         np.random.seed(13)

         begin_train_data = 40*24*7
         end_train_data = int(len(df)-24*7)

         trainer = df['puissance'].values[begin_train_data:end_train_data]
         step = 24*3


         train_x, train_y = prepare_data(trainer, step)
```

On peut remarquer l'utilisation de l'*Early Stopping*. C'est une méthode pour éviter l'overfitting. On arrête de s'entraîner lorsque l'erreur de la loss function commence à stagner car c'est à partir de ce moment que l'overfitting se produit. Sinon, le réseau de neurones commence à s'adapter aux spécificités des données d'entraînement pour améliorer les résultats.

Un détail crucial et qui semble pourtant anodin est le np.random.seed(13). Cela assure que relancer le même modèle donnera les même poids et biais initiaux. Il s'agit d'utiliser un aléatoire contrôlé. C'est crucial car selon les poids et les biais initiaux, nous allons nous retrouver près ou non de certains minimums locaux. Or, selon les minimums locaux les plus proches, nous allons obtnir une erreur petite ou grande. Donc bien que l'architecture soit importante, si on a une très bonne architecture mais un mauvais départ sur les biais et poids aléatoires, on peut avoir de très mauvais résultats.

On initalise finalement le réseau de neurones puis on ajoute chaque couche. On choisit les paramètres suivants :

- Activation function : ReLu
- Loss function : MAPE
- Input layer : 72 neurones
- Hidden layer 1 : 10 neurones
- Hidden layer 2 : 5 neurones
- Output layer : 1 neurone (la valeur prédite)
- Backpropagation : algorithme ADAM (une version optimisée de la descente de gradient stochastique que l'on a décrit précédemment)

Les 72 neruones d'entrées sont pour les 24*3 (=72) valeurs électriques précedentes.

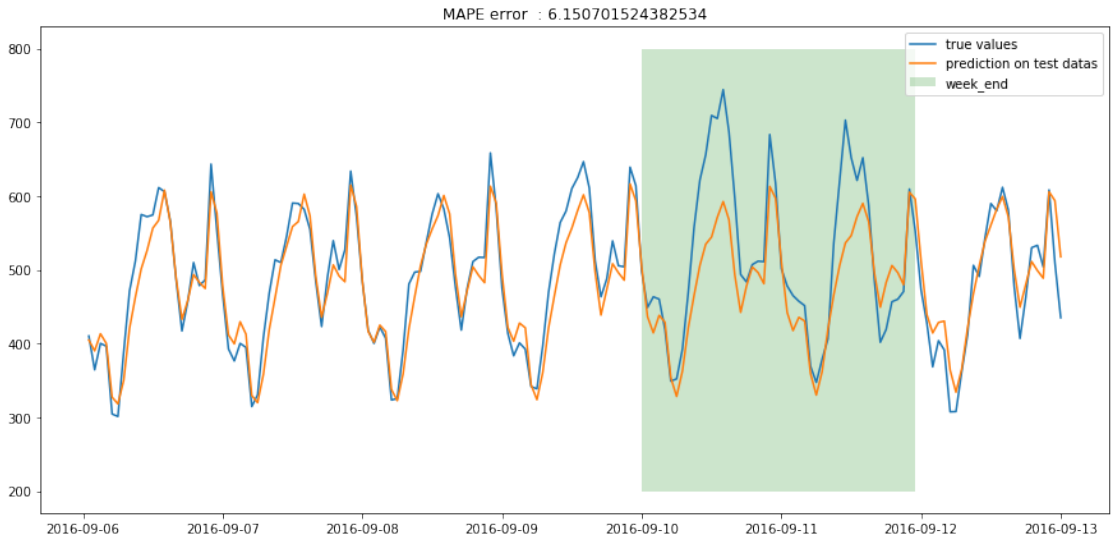
```
model = Sequential()
model.add(Dense(10, activation='relu', input_dim=step))
model.add(Dense(5, activation='relu'))
model.add(Dense(1))
ADAM = optimizers.Adam(lr=0.001)
model.compile(optimizer=ADAM, loss='mape')


early_stop = EarlyStopping(monitor='loss', patience=10, verbose=1)
checkpointer = ModelCheckpoint(filepath="weights0.hdf5", , monitor='loss',
save_best_only=True)
```

Finalement on entraine le réseau avec des tailles d'échantillons de 10, et au maximum 300 epochs (même si en réalité le early-stop arrête l'entrainement avant afin d'empêcher un surentrainement).

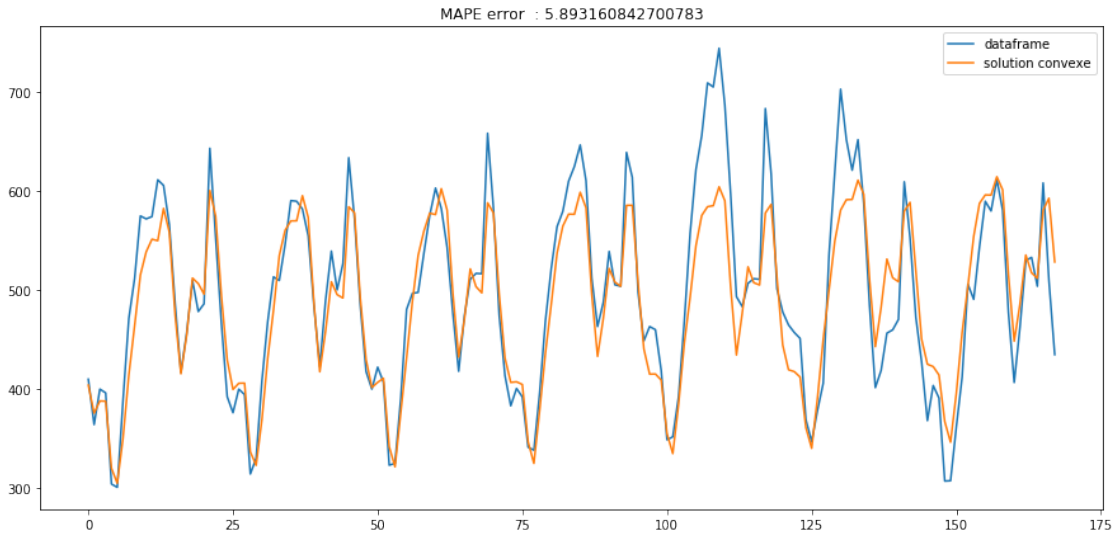
```
history = model.fit(train_x, train_y, batch_size = 10, epochs=300,callbacks = [early_sto
#model.load_weights('weights0.hdf5')
```

Voici le premier résultat que l'on obient :




On remarque que l'erreur entre la prédiction et les vrais valeurs est plus grande le week-end. Cependant en rajoutant des neurones ayant des valeurs d'entrées différentes selon si on se trouve en période week-end ou non ne permet de régler le problème.

Mais après un grid search différent on arrive à trouver un meilleur résultat :




On obtient un résultat déjà assez bon comparé au modèle ARMA, sans avoir ajouter les données météos. Nous allons maintenant faire cela afin de comparer et potentiellement d'obtenir de meilleurs résultats.

### 2.3.2 Utilisation des données météos

Nous allons maintenant utiliser les données météos, qui devraient améliorer les prédictions. En réalité, il suffit simplement de comprendre comment les utiliser, et modifier l'architecture du réseau de neurones.

On utilise les données météos car elles peuvent avoir un impact majeur sur la consommation électrique. En effet, lorsque la température est faible, il est logique que les habitants de l'île allume le chauffage.

Les données météos que nous avons ne sont pas à la maille horaire comme les données éléctrique, mais à la maille tri-horaire (1 donnée toute les trois heure). Voilà à quoi ressemble les données :

Out[109]:

| | date | T¬ (C) | P (hPa) | HR (%) | P.rosée (¬C) | Visi (km) | \ |
|---|---|---|---|---|---|---|---|
| 0 | 2015-09-13 00:00:00 | 12.5 | 1008.7 | 81.0 | 9.3 | 40.0 | |
| 1 | 2015-09-13 03:00:00 | 12.3 | 1006.4 | 83.0 | 9.5 | 40.0 | |
| 2 | 2015-09-13 06:00:00 | 12.3 | 1004.7 | 82.0 | 9.3 | 40.0 | |
| 3 | 2015-09-13 09:00:00 | 14.2 | 1002.9 | 80.0 | 10.8 | 40.0 | |
| 4 | 2015-09-13 12:00:00 | 13.3 | 1000.8 | 93.0 | 12.2 | 4.0 | |

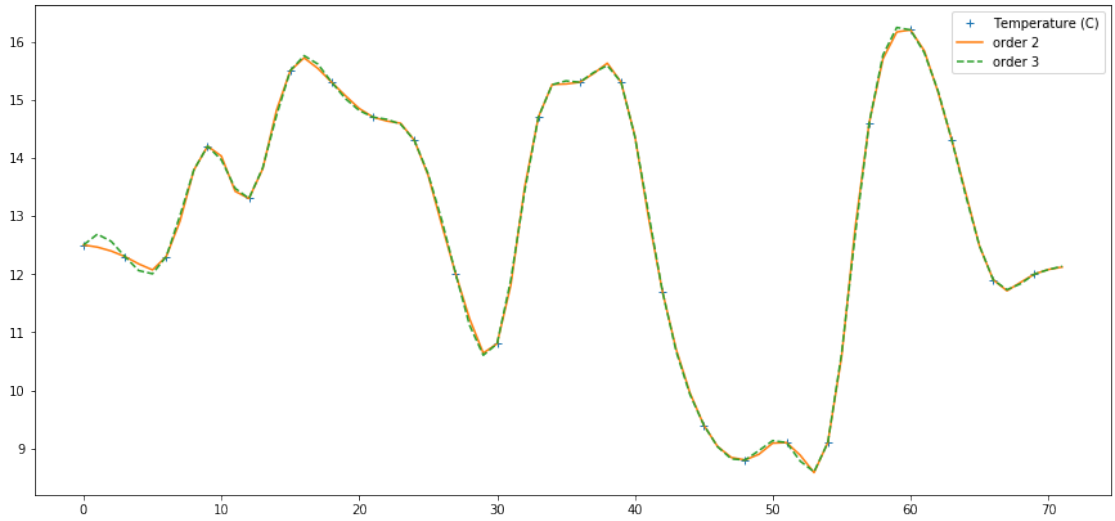
| | Vt. moy. (km/h) | Vt. raf. (km/h) | Vt. dir (¬) | RR 3h (mm) | Neige (cm) | \ |
|---|---|---|---|---|---|---|
| 0 | 9.260 | 18.520 | 140.0 | 0.0 | NaN | |
| 1 | 11.112 | 16.668 | 120.0 | 0.0 | NaN | |
| 2 | 14.816 | 22.224 | 130.0 | NaN | NaN | |
| 3 | 18.520 | 31.484 | 140.0 | NaN | NaN | |
| 4 | 18.520 | 38.892 | 140.0 | 4.0 | NaN | |

| | Nebul. (octats) |
|---|---|
| 0 | 8.0 |
| 1 | 8.0 |
| 2 | 7.0 |
| 3 | 7.0 |
| 4 | 7.0 |

Nous n'allons pas utiliser toutes les données mais seulement celles qui nous semblent utiles. Nous choisissons d'utiliser la température et la vitesse du vent. Cepandant pour les utiliser il nous faut des informations toutes les heures. Pour cela on utilise la fonction de pandas : *interpolate*().

```
df['Vt. moy. (km/h)'] = df['Vt. moy. (km/h)'].interpolate(method='polynomial', order=2)
df['Vt. raf. (km/h)'] = df['Vt. raf. (km/h)'].interpolate(method='polynomial', order=2)
df = df.round(3)
```

Nous avons choisi l'ordre de l'interpolation après avoir vu cette courbe :

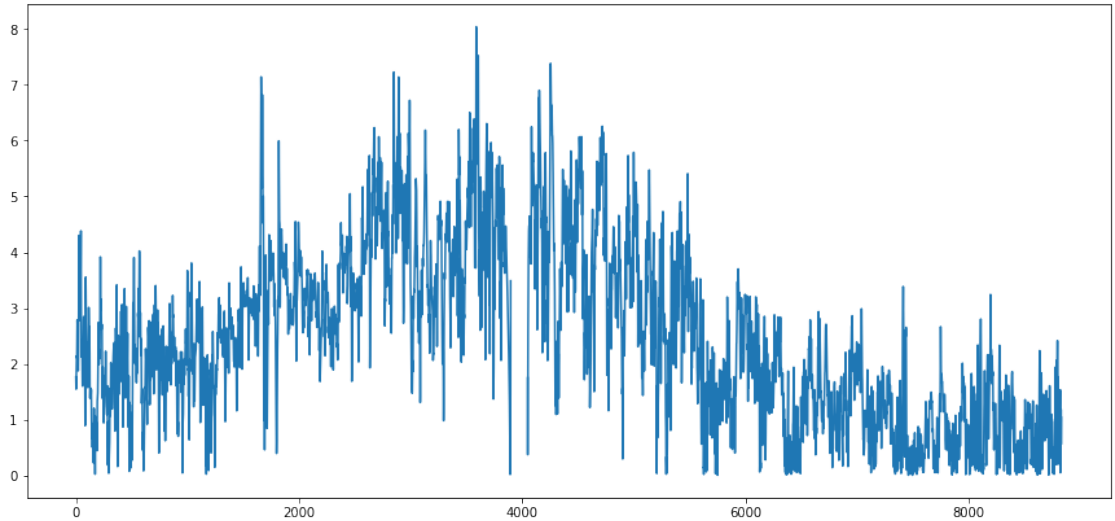



On remarque sur les premières valeurs que l'interpolation d'ordre deux rejoints les points d'une manière plus plausible selon nous.

De plus pensé que l'utilisation du chauffage dépendait plus de la température ressentie que de la vraie température (qui sont assez différents en raison des vents forts dans l'île). On calcule donc la température ressentie et on l'utilise pour améliorer les prédictions. Pour calculer la température ressentie on utilise la formule trouvée sur *wikipedia* :

```
df['Temperature ressentie'] = round(13.2+0.6215*df['Temperature (C)']+
(0.3965*df['Temperature (C)']-11.37)*df['Vt. raf. (km/h)']**(0.16),3)
```

Voici les différences entre les températures ressenties et les vrais températures sur l'île durant l'année.




On remarque que la différence entre la température et la température ressentie est plus

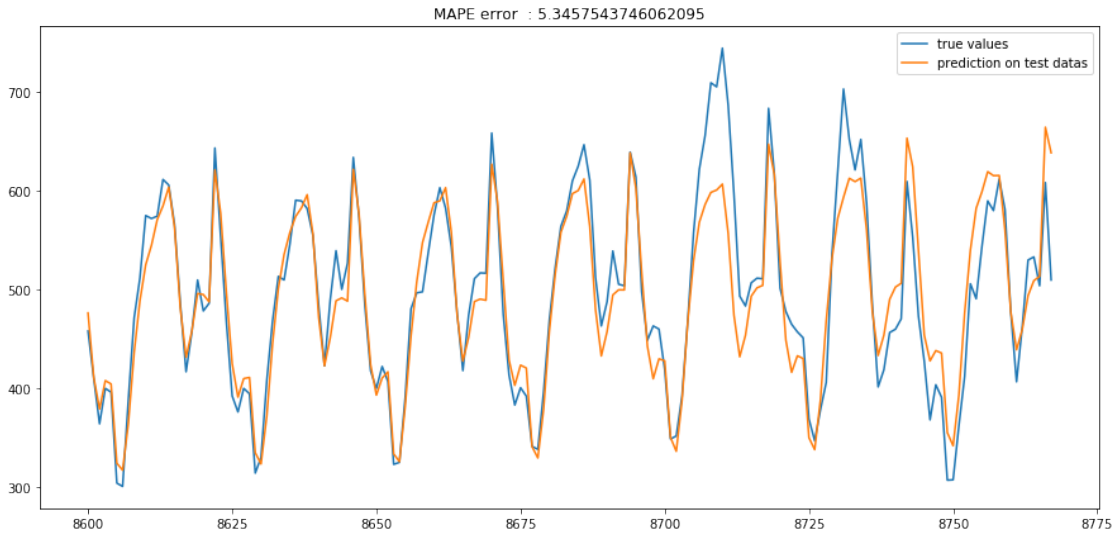
grande en hiver quand il fait froid.

On modifie légèrement la préparation des données :

```
def prepare_data2(data_y, step,meteo_y):
                x = []
                y = []
                for i in range(step,len(data_y)):
                    x.append(np.concatenate( (data_y[i-step:i] , meteo_y[i-1:i+1],  [meteo
                    y.append(data_y[i])
                return np.array(x),np.array(y)
```

Concernant l'architecture, après un grid search très long, voici les paramètres qui semblent optimaux :

- Activation function : ReLu
- Loss function : MAPE
- Input layer : 171 neurones
- Hidden layer 1 : 48 neurones
- Hidden layer 2 : 7 neurones
- Output layer : 1 neurone (car nous avons uniquement besoin du résultat)
- Backpropagation : algorithme ADAM




Les 171 nerones d'entraînes correspondent aux 24*7 (=168) valeurs précedente d'éléctricité, puis 3 neurones utilisant les données météos de l'heure à prédire, de l'heure précédente, et de la météo du jour précédent à la même heure.

On obtient un résultat très satisfaisant qui montre l'impact de la météo sur la consommation électrique. Cependant, on peut encore améliorer la prédiction si on parvient à récupérer le meilleur des prédictions ARMA et le meilleur des prédictions par ce réseau de neurones.

# 3 Concaténation convexe des deux méthodes

Après avoir étudié les deux méthodes (ARMA et réseaux de neurones), nous allons maintenant les concaténer de manière à obtenir un résultat meilleur.

## 3.1 Principe

Le principe est simple. On crée un grand nombre de valeurs entre 0 et 1 (qu'on peut nommer $\alpha_i$). Pour chaque $\alpha_i$, en notant le tableau des prédictions ARMA $p_A$ et le tableau des prédictions par réseau de neurones $p_N$, et le résultat final $p_i$, on réalise l'opération suivante :

$$p_i = \alpha_i * p_N + (1 - \alpha_i) * p_A$$

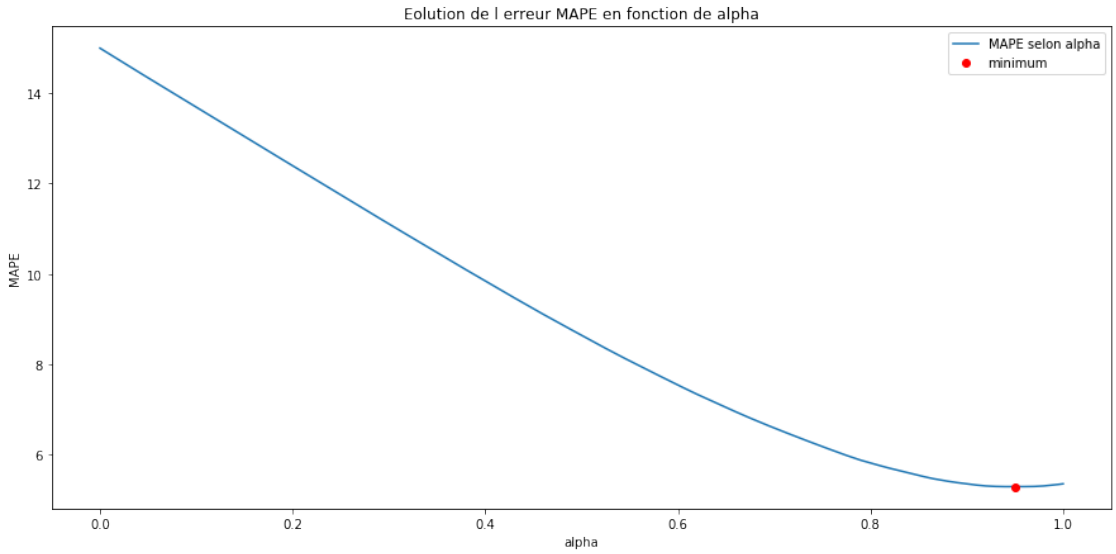
On cherche ensuite à prendre la valeur $\alpha_i$ qui minimise l'erreur MAPE de la prédiction $p_i$ obtenue. Finalement, cette prédiction sera au pire des cas aussi bonne que la meilleure prédiction entre ARMA et le réseau de neurones, car il suffit d'avoir $\alpha_i = 0$ ou $\alpha_i = 1$, et dans le meilleur des cas une valeur intermédiaire qui concatène les deux méthodes par une équation convexe.

## 3.2 Application et résultats

Pour appliquer ce principe, on commence par générer nos $\alpha_i$. Ensuite, pour chaque $\alpha_i$, on calcule la prédiction sur nos données et on compare avec les données initiales en calculant le MAPE également.

```
In [91]: alpha_tab = np.linspace(0,1,201)
         alpha_mape = []
```

On trace la courbe des erreurs MAPE selon la valeur de alpha et on cherche à minimiser cette erreur :

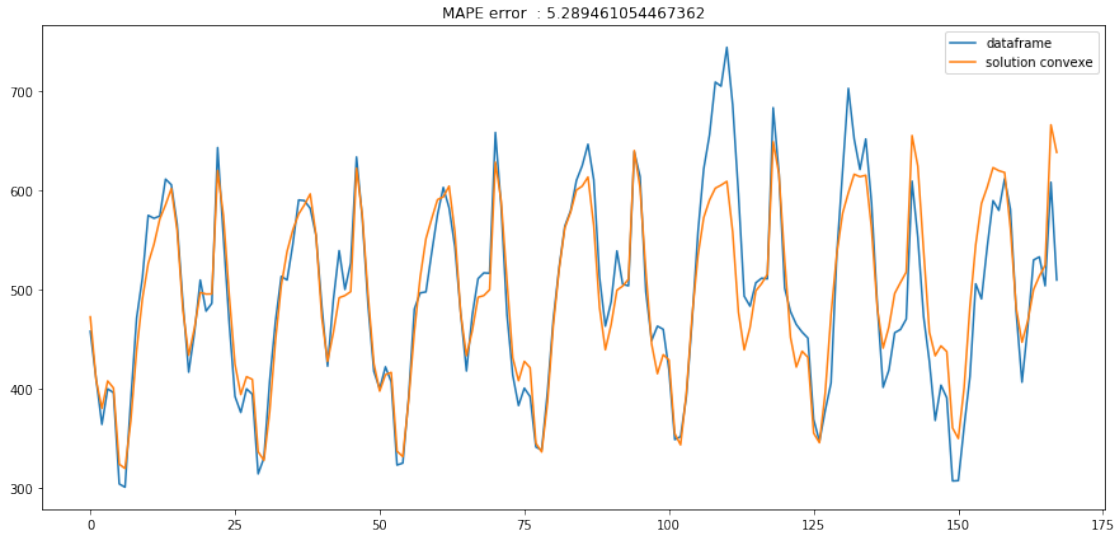



Comme on peut le voir, on n'a pas choisi notre meilleur solution ARMA car elle prenait trop de temps à compiler (celle-ci à un MAPE deux fois plus elevé), cependant, le résultat est quand même meilleur qu'avec le réseau de neurones tout seul, lorsque l'on récupère le alpha qui minimise l'erreur :

```
print('Le mape est ',alpha_mape[np.argmin(alpha_mape)],' pour alpha = ',best_alpha)
```

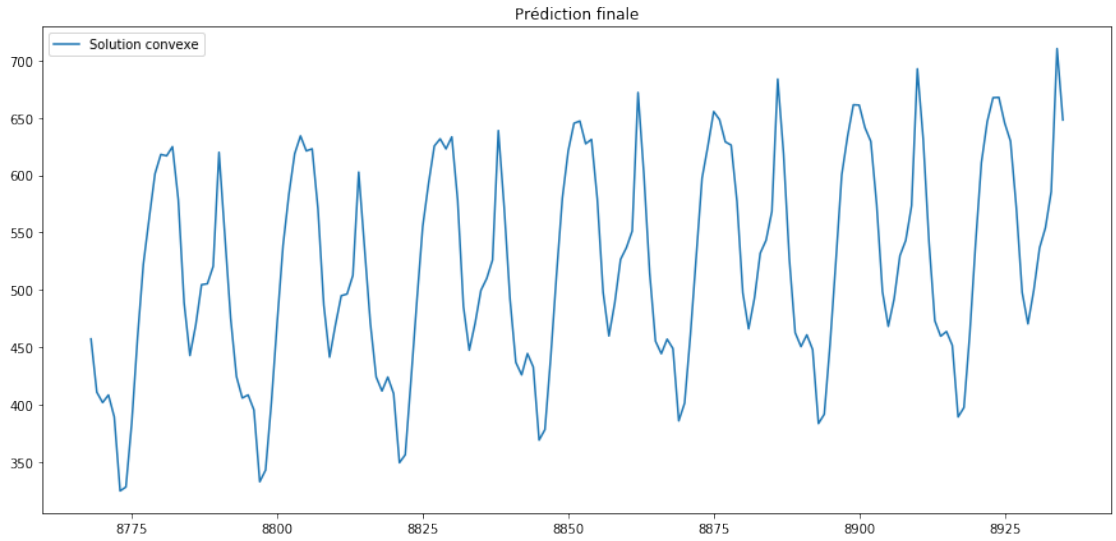
```
Le mape est  5.282864365154211  pour alpha =  0.9500000000000001
```

Finalement, la solution convexe est meilleure que ARMA tout seul, ou le réseau de neurones tout seul, donc on choisit cette dernière afin de prédire la semaine suivante.



## 3.3 Prédiction finale de la semaine suivante

Maintenant que nous avons un modèle de prévision optimal, nous pouvons enfin prédire la semaine que l'on cherchait à prédire au départ. Après avoir essayé d'obtenir la meilleure solution ARMA, la meilleure solution par réseau de neurones, nous avons pu former une solution convexe des deux prédictions qui soit encore un peu mieux. En appliquant ce résultat au calcul de prédiction de la semaine suivante, on obtient la courbe suivante :




On joint par mail le tableau des valeurs afin de pouvoir comparer avec les vraies données dont nous n'avons pas l'accès.

# 4 Conclusion

En conclusion, nous avons compris ce qu'est un modèle $ARMA$, et de quelle manière il faut l'utiliser pour réaliser une prédiction à partir d'une série temporelle. Nous avons vu dans nos recherches de nombreuses applications de tels models sur des données concrètes. Cela nous a aidé à réaliser à quel point ce projet peut être proche de ce qui est fait dans le monde du travail, comment analyser les données, les interpréter et enfin choisir le modèle adapté.

Nous avons également étudié des principes du Machine Learning, et particulièrement les réseaux de neurones. Nous avons été impressionné par la flexibilité de cet outil qui semble pouvoir s'adapter à un grand nombre de problèmes. Cela nous a permis de comprendre pourquoi ils sont si souvent utilisés et donnent de si bons résultats.

Finalement, on a vu qu'une solution convexe mélangeant ARMA et réseau de neurones était meilleure et avons pu ainsi affiner notre prédiction. Nous avons donc étoffé nos connaissances mathématiques en probabilité ainsi qu'en statistique, mais également pu les appliquer sur des modèles concrets comme ARMA, ou le Machine Learning.

# 5 Bibliographie

## 5.1 Probabilités

https://www.ljll.math.upmc.fr/~ayi/files/cours_proba_l3.pdf
https://mail.google.com/mail/u/0?ui=2&ik=45b8861290&attid=0.1&permmsgid=
msg-f:1625299167019578249&th=168e38fb2ff0b389&view=att&disp=safe
http://www.wikipedia.org

## 5.2 ARMA

https://mail.google.com/mail/u/0?ui=2&ik=45b8861290&attid=0.1&permmsgid=
msg-f:1625299167019578249&th=168e38fb2ff0b389&view=att&disp=safe
https://www.digitalocean.com/community/tutorials/a-guide-to-time-series-forecasting-with
https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/
https://www.math.u-psud.fr/~goude/Materials/ProjetMLF/time_series.html
http://zoonek2.free.fr/UNIX/48_R_2004/20.html
http://www.wikipedia.org
https://www.analyticsvidhya.com/blog/2015/12/complete-tutorial-time-series-modeling/
https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/
https://medium.com/@josemarcialportilla/using-python-and-auto-arima-to-forecast-seasonal
https://www.quantstart.com/articles/Autoregressive-Moving-Average-ARMA-p-q-Models-for-T
https://www.kaggle.com/poiupoiu/how-to-use-sarimax
http://people.duke.edu/~rnau/411arim2.htm
https://halshs.archives-ouvertes.fr/cel-01261174/document
https://didierdelignieresblog.files.wordpress.com/2016/03/arimacomplet.pdf

## 5.3 SVR

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html
https://optunity.readthedocs.io/en/latest/notebooks/notebooks/sklearn-svr.
html#determining-the-kernel-family-during-tuning
https://zestedesavoir.com/tutoriels/1760/un-peu-de-machine-learning-avec-les-svm/

## 5.4 Réseaux de Neurones

http://neuralnetworksanddeeplearning.com/
https://keras.io/