

Perceptron multi-couche : régression

1 Perceptron multicouches

1.1 Les données

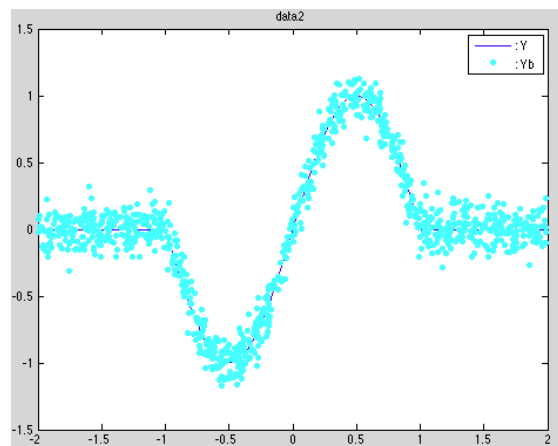
1. Ecrire (ou faire copier-coller) les instructions ci-dessous dans un fichier [donnee2.m](#).

```
n=1000;
X=4*(rand(n,1)-.5);
X=sort(X);
Y=zeros(size(X));
Y=Y+sin(pi*X).*((X>-1) & (X<1));
Yb=Y+.1*randn(size(Y,1),1);
figure
plot(X,Y,'b-','X',Yb,'c.')
legend(': Y',': Yb')
title('data2')
save data2 X Y Yb
```

2. Placez-vous dans la fenêtre de commande Matlab et tapez

`>> donnee2` → c'est pour générer les données [data2.mat](#)

On obtient la figure suivante :



1.2 Base d'apprentissage et base de test

1. Ecrire (copier-coller) la fonction suivante dans un fichier [GetSample.m](#)

```
function [Xa,Ya]=GetSamples(N,X,Y)
fprintf('\n Cliquez %d points dans la figure \n',N*2)
[xval, yval]=ginput(2*N);
for i=1:2*N
    [val(i) x(i)]=min(abs(xval(i)-X));
end
x(2*N+1)=length(X);
Xa=[];Ya=[];
k=1;
for i=1:N
    Xa=[Xa;X((x(k)+1):x(k+1))];
    Ya=[Ya;Y((x(k)+1):x(k+1))];
    k=k+2;
end
drawnow
```

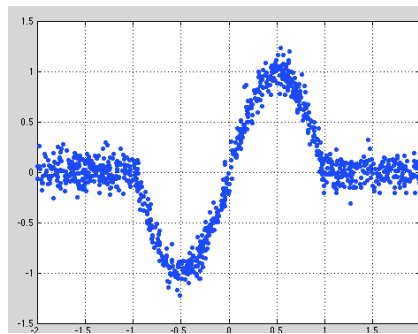
2. Ecrire (copier-coller) les instructions suivantes dans un fichier **GereneBases.m**

```
load data2;
figure
plot(X,Yb,'b.')
hold on
grid on
NbreblocsApp=4;
[Xapp,Yapp]=GetSamples(NbreblocsApp,X,Yb);
plot(Xapp,Yapp,'ro')
NbreblocsTest=3;
[Xtest,Ytest]=GetSamples(NbreblocsTest,X,Yb);
plot(Xtest,Ytest,'go')
legend(': donnees de depart', 'donnees apprentissage',':donnees test')
save base1 X Y Yb Xapp Yapp Xtest Ytest
```

3. Placez-vous dans la fenêtre de commande Matlab et tapez la commande suivante :

>> GenereBases

une fenêtre matlab s'ouvre.

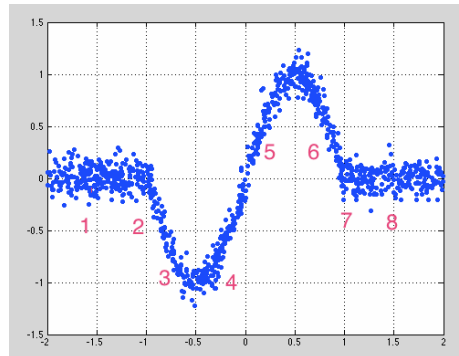


→ Il faut cliquer sur cette figure comme expliqué ci-dessous.

- (a) D'abord il faut cliquer 8 fois sur la figure → pour choisir les données d'apprentissage (4 blocs)

Exemple : cliquez là où il y a des numéros sur la figure suivante

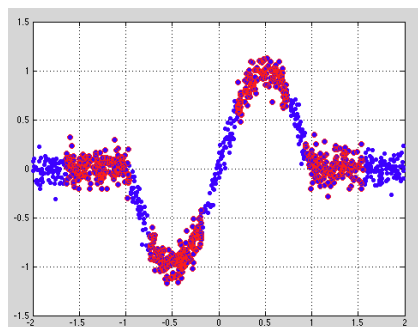
Cliquer d'abord sur 1, puis sur 2, puis sur 3 ... (c'est à dire dans l'ordre croissant des numéros)



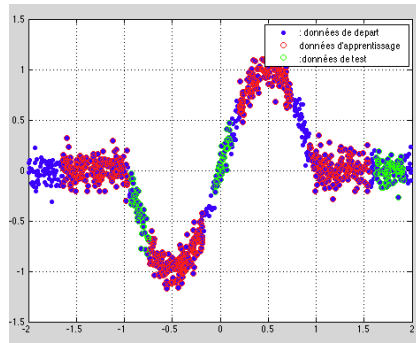
Explications : dans le fichier **GereneBases.m** on a choisi **NbreblocsApp=4** (4 blocs de données).
2 cliques consécutifs ⇒ un bloc de données

→ Donc il faut cliquer : $2 * \text{NbreblocsApp} = 2 * 4 = 8$ fois

Par exemple, la figure suivante montre les données choisies pour l'apprentissage : les 4 blocs rouges



- (b) Ensuite il faut **cliquer 6 fois sur la figure** pour choisir les données test (**3 blocs**).
 Dans le fichier **GereneBases.m** on a choisi **NbreblocksTest=3** $\Rightarrow$ 3 blocs de données
 $\rightarrow$ Donc il faut cliquer : **2 * NbreblocksTest = 2 * 3 = 6 fois**
 Par exemple la figure suivante montre les données choisies pour l'**apprentissage**
 (les **4 blocs rouges**) et les données choisies pour le **test** (les **3 blocs verts**).



1.3 Apprentissage

Utiliser les instructions suivantes pour l'apprentissage

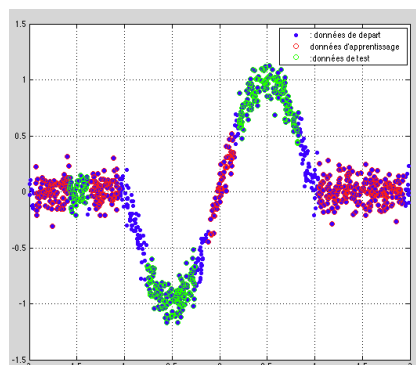
```
global options
addpath Netlab
figure
load base1
NbreNC = 10; % Nombre de neurones dans la couche cachée
Net = mlp(size(Xapp,2), NbreNC, size(Yapp,2), 'linear');
options = foptions;
options(1) = 1;
options(14) = 100;
options(18) = 0.001;
[Net options errlog] = netopt(Net, options, Xapp, Yapp, 'graddesc');
figure
plot(errlog)
Yc=mlpfwd(Net,X);
figure
plot(X,Y,'b-',X,Yc,'r-')
legend(': Y',': Yc');
Etest=mlperr(Net,Xtest,Ytest);
```

Il vaut mieux écrire ces instructions (vous pouvez ignorer les commentaires) dans un fichier texte **.m** (par exemple dans un fichier : **ReseauxNeurones1.m**). Ensuite, vous tapez le nom du fichier (sans le .m) dans la fenêtre de commande matlab

```
>> ReseauxNeurones1
```

Ensuite, vous pouvez faire les tests suivants :

- Faire varier le **nombre de cycles d'apprentissage** : donc faire varier **options(14)**
- Générer de **nouveaux ensembles d'apprentissage**. Par exemple, **cacher les parties importantes de la fonction qu'on veut apprendre** $\rightarrow$ comme sur la figure ci-dessous (**les points rouges** ne représentent pas la fonction $\Rightarrow$ il va apprendre une droite ou presque)



2 Fonctions importantes de Netlab pour un perceptron multi-couche

- **mlp.m** : permet de définir une architecture et d'initialiser les poids.

```
Net = mlp(ne, nc, ns, fsortie);
```

- . **ne** : nombre de neurones en entrée (c'est la dimension des entrées).
- . **nc** : nombre de neurones cachés.
- . **ns** : nombre de neurones en sortie (c'est la dimension des sorties).
- . **fsortie** : fonction d'activation pour la couche de sortie
→ **fsortie** = 'linear', 'logistic' ou 'softmax'
→ Faire **help mlp** pour plus de détails.

- **netopt.m** : permet d'effectuer l'apprentissage en utilisant l'algorithme de rétropropagation du gradient.

```
[Net, options] = netopt(Net, options, Xapp, Yapp, alg);
```

- . **Net** : architecture du réseau (avec les poids)
- . **(Xapp,Yapp)** : base d'apprentissage (couples entrée-sortie)
- . **options(1)** : permet d'activer (ou désactiver) l'affichage des erreurs.
- . **options(14)** : nombre de cycles d'apprentissage
- . **options(18)** : le pas d'apprentissage
- . **alg** : algorithme d'apprentissage (d'optimisation)
→ **alg** = 'graddesc', 'scg', 'conjgrad' ou 'quasineu'
→ Faire **help netopt** pour plus de détails.

- **mlpfwd** : permet de calculer la sortie d'un réseau pour une entrée donnée

```
Ytestcal = mlpfwd(Net, Xtest);
```

- . **Net** : architecture du réseau (avec les poids)
- . **Ytestcal** : sortie calculée par le réseau **Net** pour l'entrée **Xtest**.

Ensuite il faut comparer **Ytest** avec **Ytestcal**.

→ **Ytest** : la sortie désirée pour l'entrée **Xtest**

Remarque

- L'apprentissage avec l'algorithme 'graddesc' dépend fortement de la valeur donnée au pas d'apprentissage → **options(18)**.
- Les algorithmes du deuxième ordre 'scg', 'conjgrad' et 'quasineu' utilisent les dérivées secondes de la fonction de coût pour pouvoir calculer les pas d'apprentissage optimaux (donc 'scg', 'conjgrad' et 'quasineu' n'utilisent pas **options(18)**).
- Dorénavant vous pouvez utiliser l'algorithme 'scg' à la place de 'graddesc'

3 Fonction d'activation des neurones de sortie : 'linear', 'logistic' ou 'softmax'

Dans Netlab l'architecture est limitée à une seule couche cachée.

- La fonction d'activation des neurones cachés est une tangente hyperbolique.
- Pour les neurones de la couche de sortie, la fonction d'activation peut être :
 - linéaire ('linear'), logistique ('logistic') ou softmax ('softmax')
 - voir, par exemple, la fonction `mlpfwd.m` de Netlab (voir encadré ci-dessous)

Un extrait de `mlpfwd.m`

```
ndata = size(x, 1); fonction d'activation : couche cachée
z = tanh(x*net.w1 + ones(ndata, 1)*net.b1);
a = z*net.w2 + ones(ndata, 1)*net.b2;

switch net.outfn fonction d'activation : couche de sortie
case 'linear' % Linear outputs
    y = a;
case 'logistic' % Logistic outputs
    % Prevent overflow and underflow: use same bounds as mlperr
    % Ensure that log(1-y) is computable: need exp(a) > eps
    maxcut = -log(eps);
    % Ensure that log(y) is computable
    mincut = -log(1/realmin - 1);
    a = min(a, maxcut);
    a = max(a, mincut);
    y = 1./(1 + exp(-a));
case 'softmax' % Softmax outputs
    % Prevent overflow and underflow: use same bounds as glmerr
    % Ensure that sum(exp(a), 2) does not overflow
    maxcut = log(realmax) - log(net.nout);
    % Ensure that exp(a) > 0
    mincut = log(realmin);
    a = min(a, maxcut);
    a = max(a, mincut);
    temp = exp(a);
    y = temp./(sum(temp, 2)*ones(1, net.nout));
otherwise
    error(['Unknown activation function ', net.outfn]);
end
```

voir
`mlpfwd.m`

Choix d'une fonction d'activation à la couche de sortie

- Pour un problème de régression : il vaut mieux utiliser une fonction d'activation linéaire à la couche de sortie
- Pour un problème de classification : il vaut mieux utiliser des fonctions d'activation non linéaires ('logistic' ou 'softmax') à la couche de sortie.

Quand on utilise la fonction d'activation 'softmax' la somme des sorties est égale à 1.

☞ On peut ainsi interpréter les sorties du réseau comme des probabilités que l'entrée appartienne à chacune des classes.

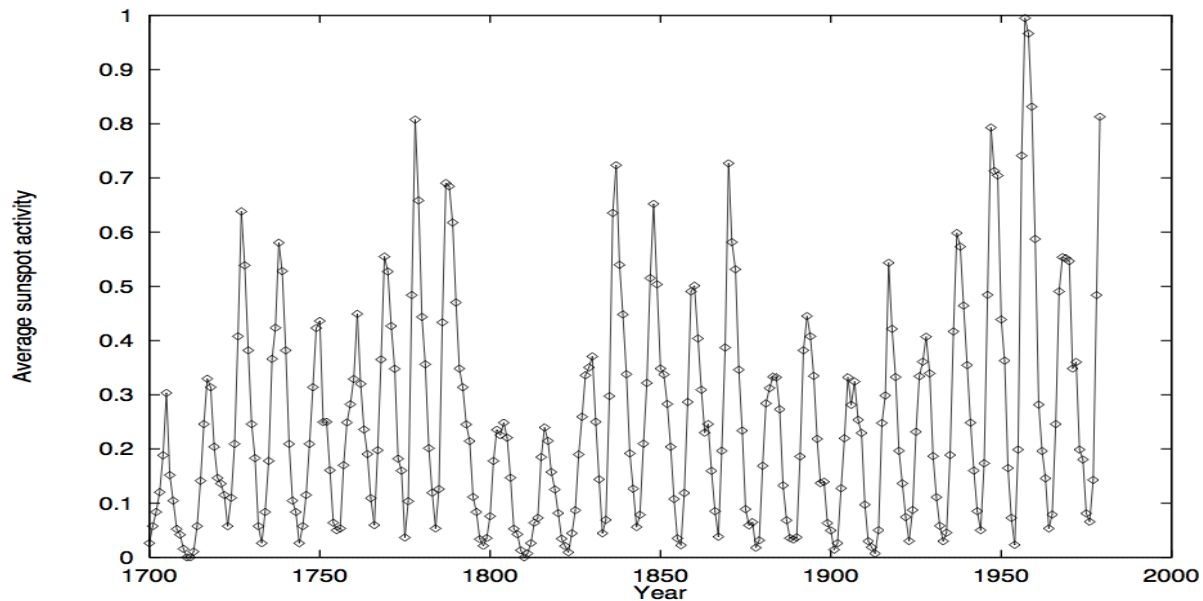
Codage des sorties désirées pour la classification

Le codage classique des sorties désirées pour la classification utilise un neurone de sortie par classe, avec une valeur désirée haute pour le neurone de la classe correcte, et une valeur désirée faible pour les autres classes. Par exemple, pour les Iris de Fisher, on peut utiliser le codage suivant :

classe 1 → (1 0 0) classe 2 → (0 1 0) classe 3 → (0 0 1)

4 Un problème réel de régression : le problème "sunspot"

Il s'agit du nombre moyen annuel de taches noires observées sur le soleil entre 1700 et 1979 (**280 points**). La série des taches solaires est très courte et célèbre dont l'équation sous jacente est inconnue. La figure ci-dessous montre l'évolution annuelle de cette série.



Série sunspot normalisée de 1700 à 1979.

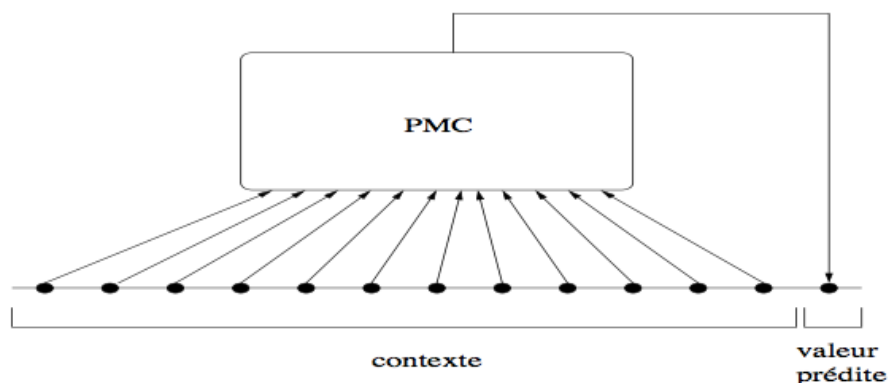
Pour ce problème, on essaie de prédire une valeur en utilisant les 12 valeurs précédentes.

Télécharger le fichier **DonneesSunspots** (**DonneesSunspots.zip**)

<http://deptinfo.cnam.fr/new/spip.php?article851>

Indications pour traiter ce problème avec un perceptron multi-couche

- Comme on essaie de prédire une valeur en utilisant les 12 valeurs précédentes, nous allons donc utiliser un réseau avec une couche d'entrée de 12 neurones ($n_e = 12$) et une couche de sortie de 1 neurone ($n_s = 1$), comme le montre la figure suivante.



- Pour ce problème, on utilise (dans les compétitions internationales)
 - les données de **1712 à 1920 pour l'apprentissage** : un ensemble D^{App} de 209 exemples
☞ **Ensemble d'apprentissage** : il sert à calculer les poids du réseau.
 - les données de **1921 à 1955 pour la validation** : un ensemble D^{Val} de 35 exemples.
☞ **Ensemble de validation** : il sert à éviter le sur-apprentissage
 - les données de **1956 à 1979 pour le test** : un ensemble D^{Test} de 24 exemples.
☞ **Ensemble de test** : il sert à évaluer les performances du réseau obtenu.
→ Cet ensemble ne doit pas intervenir ni pour la détermination des poids ni pour l'arrêt de l'apprentissage.

- Pour constituer les ensembles D^{App} , D^{Val} et D^{Test} , vous pouvez utiliser les instructions suivantes (**voir les détails à la fin de ce document**)

```
data = load('Sunspots');
dataSunspots=data(:,2);
% Data input
Entree(:,1)=dataSunspots(1:268);
Entree(:,2)=dataSunspots(2:269);
Entree(:,3)=dataSunspots(3:270);
Entree(:,4)=dataSunspots(4:271);
Entree(:,5)=dataSunspots(5:272);
Entree(:,6)=dataSunspots(6:273);
Entree(:,7)=dataSunspots(7:274);
Entree(:,8)=dataSunspots(8:275);
Entree(:,9)=dataSunspots(9:276);
Entree(:,10)=dataSunspots(10:277);
Entree(:,11)=dataSunspots(11:278);
Entree(:,12)=dataSunspots(12:279);
% Data output
Sortie=dataSunspots(13:280);
% Bases : Apprentissage, Validation et Test
DappInput=Entree(1:209,:); % Learning input
DappOutput=Sortie(1:209); % Learning output
DvalInput=Entree(210:244,:); % Validation input
DvalOutput=Sortie(210:244); % Validation output
DtestInput=Entree(245:268,:); % Test input
DtestOutput=Sortie(245:268); % Test output
```

- **Normaliser les entrées**

- Il est impératif que les entrées soient de moyenne pas trop loin de zéro et de variance pas trop loin de 1. Si on a une entrée très grande, elle fait saturer plusieurs neurones et bloque l'apprentissage pour cet exemple. Donc il faut toujours penser à la normalisation des entrées.
- Les données "sunspots" (que vous avez chargé) sont déjà normalisées entre 0 et 1. Mais comme la fonction d'activation des neurones cachés est une tangente hyperbolique, il vaut mieux essayer de les ramener entre -1 et 1. (**Si** X est entre 0 et 1 **alors** (2X-1) est entre -1 et 1).

- **Mesure de qualité** Les performances du modèle sont calculées en utilisant le critère ARV ("Average Relative Variance"), qui est le rapport entre l'erreur quadratique moyenne du modèle et la variance des données. La définition de l'ARV établit un rapport entre l'erreur du modèle et la variance des données calculée sur le même ensemble D (pris de l'ensemble entier S).

Cependant, la définition de l'ARV la plus utilisée utilise la variance totale des données (de la série).

$$arv(D) = \frac{\frac{1}{|D|} \sum_{i \in D} (y^i - f(x^i))^2}{\frac{1}{|S|} \sum_{i \in S} (y^i - \mu_S)^2} \quad (1)$$

où y^i est la valeur de la série à l'instant i , $f(x^i)$ est la sortie du réseau à l'instant i , et μ_S la moyenne de la valeur désirée dans S (la série entière) .

Afin de pouvoir comparer vos résultats avec ceux obtenus en utilisant d'autres méthodes, utilisez cette formule.

Quelques résultats sur le problème "sunspots"

Modèle/arv	$D^l(1712-1920)$	$D^v(1921-1955)$	$D^t(1956-1979)$
Yacoub & Bennani	0.089	0.076	0.331
Goutte[40]	0.082	0.082	0.357
Svar & al.[87]	0.090	0.082	0.35
Weigend & al.[96]	0.082	0.086	0.35
Linéaire	0.131	0.128	0.36

TAB. 2.4 – Comparaison des résultats obtenus en utilisant différentes méthodes. N

- **Lancer un premier apprentissage**

Je vais utiliser X pour les entrées et Y pour les sorties (c'est plus simple) .

C'est à dire (X_{app} , Y_{app} , X_{val} , Y_{val} , X_{test} et Y_{test}) :

X_{app} =DappInput;

Y_{app} =DappOutput; → X_{app} et Y_{app} : pour l'apprentissage (pour calculer les poids du réseau)

X_{val} =DvalInput;

Y_{val} =DvalOutput; → X_{val} et Y_{val} : pour la validation (pour déterminer quand arrêter l'apprentissage)

X_{test} =DtestInput;

Y_{test} =DtestOutput; → X_{test} et Y_{test} : pour le test (évaluer les performances du réseau obtenu)

→ Lancer un premier apprentissage (un premier essai) en utilisant les instructions suivantes :

```
nbre_neur_entree = size(Xapp,2);
nbre_neur_sortie = size(Yapp,2);
nbre_neur_cache = 10;
Net = mlp(nbre_neur_entree, nbre_neur_cache, nbre_neur_sortie, 'linear');
options      = foptions;
options(1)   = 1;
options(14)  = 1000;
algorithm    = 'scg';
[Net, options] = netopt(Net, options, Xapp, Yapp, algorithm);
```

→ Calculer les erreurs et ARV

ErreurApprentissage= mlperr(Net,Xapp,Yapp); % Erreur sur l'ensemble d'apprentissage

ErreurValidation= mlperr(Net,Xval,Yval); % Erreur sur l'ensemble de validation

ErreurTest= mlperr(Net, Xtest,Ytest); % Erreur sur l'ensemble de test

```
variance_data_sunspots = mean((Sortie - mean(Sortie)).^2);
YappCalculee = mlpfwd(Net,Xapp);
arv_apprentissage= mean((Yapp-YappCalculee).^2)/variance_data_sunspots
YValCalculee = mlpfwd(Net,Xval);
arv_validation=mean((Yval-YValCalculee).^2)/variance_data_sunspots
YTestCalculee = mlpfwd(Net, Xtest);
arv_test=mean((Ytest-YTestCalculee).^2)/variance_data_sunspots
```

→ Faire : [help mlpfwd](#)

→ Faire : [help mlperr](#)

→ Faire varier le **nombre de cycles d'apprentissage** → **options(14)**
et recalculer les différentes erreurs

Détails pour constituer les ensembles D^{App} , D^{Val} et D^{Test}

Remarque : on peut utiliser n'importe quel langage de programmation pour traiter cette question. L'utilisation de matlab (ou octave) n'est indispensable que lorsqu'on fait appel aux fonctions des toolbox [Netlab](#) ou [som-toolbox](#). Je vais utiliser **octave** (comme alternative gratuite de matlab)

Pour prédire une valeur on utilise les 12 valeurs précédentes. Donc

→ on peut prédire les valeurs des années 1712 à 1979

→ on ne peut pas prédire les valeurs des années 1700 à 1711 (il n'y a pas assez de valeurs).

Donc on va utiliser :

les 12 valeurs des années 1700 à 1711 pour prédire la valeur de l'année 1712
les 12 valeurs des années 1701 à 1712 pour prédire la valeur de l'année 1713
les 12 valeurs des années 1702 à 1713 pour prédire la valeur de l'année 1714
...
les 12 valeurs des années 1967 à 1978 pour prédire la valeur de l'année 1979

Si on note : x_1 la valeur de l'année 1700,

x_2 la valeur de l'année 1701

x_3 la valeur de l'année 1702

...

x_{280} la valeur de l'année 1979

le problème consiste alors à utiliser :

$x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}$ pour prédire x_{13}
$x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}$ pour prédire x_{14}
$x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}$ pour prédire x_{15}
...
$x_{268}, x_{269}, x_{270}, x_{271}, x_{272}, x_{273}, x_{274}, x_{275}, x_{276}, x_{277}, x_{278}, x_{279}$ pour prédire x_{280}

Donc on va constituer 2 matrices (matrice des entrées et matrice de sortie) comme suit :

Entrée	Sortie
$x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}$	x_{13}
$x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13}$	x_{14}
$x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14}$	x_{15}
...	...
$x_{268} x_{269} x_{270} x_{271} x_{272} x_{273} x_{274} x_{275} x_{276} x_{277} x_{278} x_{279}$	x_{280}

- Chargement des données (ici dans une matrice data)

```
octave-3.2.3:146> data = load('Sunspots');
```

→ data est une matrice 280 x 2 (**280 lignes et 2 colonnes**)

→ La première colonne contient les années. Voici, par exemple, comment afficher les 12 premières années

```
octave-3.2.3:153> data(1:12,1)
```

```
ans =
```

```
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
```

→ La deuxième colonne contient la valeur de chaque année (le nombre moyen annuel de taches noires observées sur le soleil). Voici, par exemple, comment afficher les 12 premières valeurs

```
octave-3.2.3:154> data(1:12,2)
ans =

0.02620
0.05750
0.08370
0.12030
0.18830
0.30330
0.15170
0.10460
0.05230
0.04180
0.01570
0.00000
```

On va mettre ces valeurs ([celles de la deuxième colonne](#)) dans dataSunspots

```
octave-3.2.3:157> dataSunspots=data(:,2);
```

D'après la table suivante (voir ci-dessus):

Entrée	Sortie
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12	x13
x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13	x14
x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14	x15
...	...
x268 x269 x270 x271 x272 x273 x274 x275 x276 x277 x278 x279	x280

la [première colonne](#) de la matrice **"Entrée"** contient les valeur **x1 à x268**

la [deuxième colonne](#) de la matrice **"Entrée"** contient les valeur **x2 à x269**

etc ...

la [matrice de sortie](#) contient les valeurs **x13 à x280**

On peut alors [construire](#) la matrice **"Entree"** et la matrice **"Sortie"** comme suit :

```
octave-3.2.3:159> Entree(:,1)=dataSunspots(1:268);
octave-3.2.3:160> Entree(:,2)=dataSunspots(2:269);
octave-3.2.3:161> Entree(:,3)=dataSunspots(3:270);
octave-3.2.3:162> Entree(:,4)=dataSunspots(4:271);
octave-3.2.3:163> Entree(:,5)=dataSunspots(5:272);
octave-3.2.3:164> Entree(:,6)=dataSunspots(6:273);
octave-3.2.3:165> Entree(:,7)=dataSunspots(7:274);
octave-3.2.3:166> Entree(:,8)=dataSunspots(8:275);
octave-3.2.3:167> Entree(:,9)=dataSunspots(9:276);
octave-3.2.3:168> Entree(:,10)=dataSunspots(10:277);
octave-3.2.3:169> Entree(:,11)=dataSunspots(11:278);
octave-3.2.3:170> Entree(:,12)=dataSunspots(12:279);
octave-3.2.3:171> %
octave-3.2.3:171> Sortie=dataSunspots(13:280);
```

→ Tapez [whos](#) pour voir les variables

```
octave-3.2.3:172> whos
Variables in the current scope:
```

Attr	Name	Size	Bytes	Class
====	====	====	=====	=====
	Entree	268x12	25728	double
	Sortie	268x1	2144	double
	ans	12x1	96	double
	data	280x2	4480	double
	dataSunspots	280x1	2240	double

On utilise :

- les données de 1712 à 1920 pour l'apprentissage (un ensemble D^{App} de 209 exemples)
- les données de 1921 à 1955 pour la validation (un ensemble D^{Val} de 35 exemples)
- les données de 1956 à 1979 pour le test (un ensemble D^{Test} de 24 exemples).

Donc

- Les 209 premiers exemples vont servir pour l'apprentissage
→ les exemples d'indices 1 à 209 pour l'apprentissage
- Les 35 exemples qui suivent pour la validation : de 209+1 à 209+35
→ les exemples d'indices 210 à 244 pour la validation
- Les 24 derniers exemples pour le test : de 244+1 à 244+24
→ les exemples d'indices 245 à 268 pour le test

On peut alors construire Dapp, Dval et Dtest comme suit :

```
octave-3.2.3:173> DappInput=Entree(1:209,:);
octave-3.2.3:174> DappOutput=Sortie(1:209);
octave-3.2.3:175> DvalInput=Entree(210:244,:);
octave-3.2.3:176> DvalOutput=Sortie(210:244);
octave-3.2.3:177> DtestInput=Entree(245:268,:);
octave-3.2.3:178> DtestOutput=Sortie(245:268);
octave-3.2.3:179>
```

Voici donc l'ensemble des instructions à exécuter sous octave ou matlab

```
data = load('Sunspots');
dataSunspots=data(:,2);
% Data input
Entree(:,1)=dataSunspots(1:268);
Entree(:,2)=dataSunspots(2:269);
Entree(:,3)=dataSunspots(3:270);
Entree(:,4)=dataSunspots(4:271);
Entree(:,5)=dataSunspots(5:272);
Entree(:,6)=dataSunspots(6:273);
Entree(:,7)=dataSunspots(7:274);
Entree(:,8)=dataSunspots(8:275);
Entree(:,9)=dataSunspots(9:276);
Entree(:,10)=dataSunspots(10:277);
Entree(:,11)=dataSunspots(11:278);
Entree(:,12)=dataSunspots(12:279);
% Data output
Sortie=dataSunspots(13:280);
% Bases : Apprentissage, Validation et Test
DappInput=Entree(1:209,:); % Learning input
DappOutput=Sortie(1:209); % Learning output
DvalInput=Entree(210:244,:); % Validation input
DvalOutput=Sortie(210:244); % Validation output
DtestInput=Entree(245:268,:); % Test input
DtestOutput=Sortie(245:268); % Test output
```