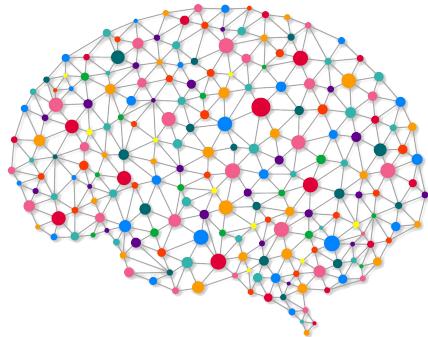


RAPPORT DE PROJET

PROJET - RÉSEAUX DE NEURONES ARTIFICIELLES



Margaux MORIN
ISUP - ISDS 3^{me} année

Professeur : Annick VALIBOUZE

Institut de Statistique de l'Université de Paris
Sorbonne Université - Pôle Science

Master 2 Ingénierie Mathématiques
Parcours Ingénierie Statistique et Data Science

15 avril 2020

Table des matières

1	Introduction	2
1.1	Histoire	2
1.2	Jeu de données pour Application	3
2	Perceptron Simple	6
2.1	Modélisation	6
2.2	Apprentissage	8
2.3	Application : avec le package nnet	12
3	Le Perceptron Multicouches (MLP)	15
3.1	Modélisation	15
3.2	Apprentissage	16
3.3	Application	17
3.3.1	Avec le package nnet	17
3.3.2	Avec le package neuralnet	23
3.3.3	Avec le package deepnet	27
4	Carte de Kohonen	29
4.1	Principe	29
4.2	Application : avec le package RCurl	29
5	La Machine de Boltzmann Restreinte - RBM	32
5.1	Principe	32
5.2	Application : avec le package deepnet	34
6	Les réseaux de croyances profondes - DBN	35
6.1	Principe	35
6.2	Application : avec le package deepnet	37
6.3	Conclusion	37
7	La machine de Boltzmann Profonde - DBM	38
7.1	Principe	38
8	Conclusion	40

1 Introduction

1.1 Histoire

Les réseaux de neurones artificiels (RNA) sont des modèles mathématiques inspirés du réseau neuronal du cerveau humain. La composante de base de ces réseaux, le neurone artificiel, a été développée à l'origine pour modéliser le fonctionnement d'un neurone biologique.

On peut décomposer le neurone en 3 grandes entités :

- un corps cellulaire, appelé péricaryon ;
- un ensemble de dendrites (de l'ordre de 7 000) : les capteurs du neurone. Elles transmettent l'influx nerveux (l'information) générée par des stimuli ;
- un axone.

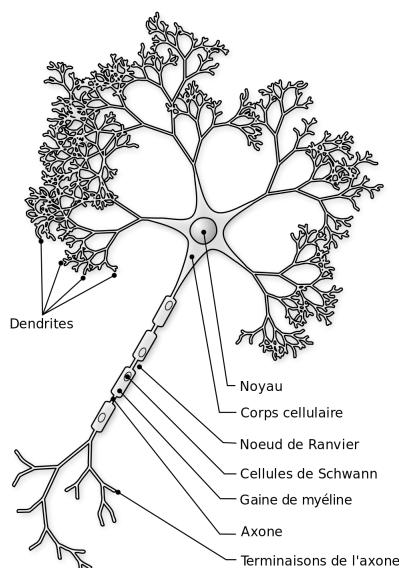


FIGURE 1 – Schéma neurone humain

L'objectif initial de l'approche des réseaux de neurones artificiels était de résoudre les problèmes de la même manière que le ferait un cerveau humain. Mais avec le temps, l'attention s'est déplacée vers l'exécution de tâches spécifiques, ce qui a conduit à des déviations par rapport à la biologie. Les RNA ont été utilisées pour diverses tâches, notamment la vision par ordinateur, la reconnaissance vocale, la traduction automatique, le filtrage des réseaux sociaux, les jeux de société et les jeux vidéo, le diagnostic médical.

Ces systèmes "apprennent" à exécuter des tâches en prenant des exemples, généralement sans être programmés avec des règles spécifiques à la tâche. Par exemple, en reconnaissance d'images, ils peuvent apprendre à identifier des images qui contiennent des chats en analysant des exemples d'images qui ont été manuellement étiquetées comme "chat" ou "pas de chat" et en utilisant les résultats pour identifier les chats dans d'autres images. Ils le font sans savoir au préalable que les chats, par exemple, ont une fourrure, une queue, des moustaches et un visage de chat. Au lieu de cela, ils génèrent automatiquement des caractéristiques d'identification à partir des exemples qu'ils traitent.

1.2 Jeu de données pour Application

L'ensemble des applications logicielles de ce projet seront réalisés sur R, avec différents packages.

Pour tester les différents types de réseaux qui seront étudiés au cours de ce projet, on télécharge un jeu de données portant sur les cellules cancéreuses du cancer du sein. Ces données proviennent du centre des sciences cliniques de l'université du Wisconsin aux États-Unis.

Lien : <https://archive.ics.uci.edu/ml/datasets.php?fbclid=IwAR0Ve0RT79hwKWbGqFgvLZLOqqR309Hz60OSQWzRQ16kw>

Ce jeu de données comporte 569 observations (correspondant à des patients) et 32 variables. On possède :

- L'identifiant du patient
- Le diagnostic du cancer : malin ou bénin

Dix caractéristiques sont calculées à partir d'une image numérisée d'une masse mammaire. Ils décrivent les caractéristiques des noyaux cellulaires présents dans l'image. Le jeu de données nous fournit la moyenne, l'écart-type et la "pire" (moyenne des trois plus grandes valeurs) pour ses caractéristiques :

- Le rayon (moyenne des distances du centre aux points du périmètre)
- La texture (écart-type des valeurs de l'échelle de gris)
- Le périmètre
- La zone
- La régularité (variation locale des longueurs de rayon)
- La compacité ($perimetre^2 / surface - 1$)
- La concavité (gravité des parties concaves du contour)
- Les points concaves (nombre de parties concaves du contour)
- La symétrie
- La dimension fractale

Notre variable à expliquer est le diagnostic, comme il s'agit d'une variable de nature qualitative, il nous faut utiliser un modèle de discrimination.

Ensuite on sépare notre jeu de données en deux pour avoir une partie pour entraîner les réseaux de neurones et une partie pour les tester. On prend 2/3 du jeu aléatoirement pour la partie apprentissage et le tiers restant pour la partie test et validation.

```
1 library(dplyr)
2
3 #charger le fichier
4 data.all <- read.table(paste(getwd(),"/wdbc.txt", sep=""), header = F, sep=",", dec=".")
5 #noms des colonnes
6 names(data.all) <- c("ID.number", "Diagnosis", "radius_mean", "texture_mean", "perimeter_mean", "area_mean", "smoothness_mean", "compactness_mean", "concavity_mean", "concave.points_mean", "symmetry_mean", "fractal.dimension_mean", "radius_se", "texture_se", "perimeter_se", "area_se", "smoothness_se", "compactness_se", "concavity_se", "concave.points_se", "symmetry_se", "fractal.dimension_se", "radius_largest", "texture_largest", "perimeter_largest", "area_largest", "smoothness_largest", "compactness_largest", "concavity_largest", "concave.points_largest", "symmetry_largest", "fractal.dimension_largest")
7
8 #taille de la base
9 print(str(data.all))
10
11 #sommaire de la base
```

```

12 print(summary(data.all))
13
14 #création des jeux d'entraînement et de test
15 prop.train = 0.7
16 train <- sample(1:nrow(data.all), round(2*nrow(data.all)/3), replace = FALSE)
17 test <- setdiff(1:nrow(data.all),train)
18
19 data.train <- data.all[train,-1]
20 data.test <- data.all[test,-1]

```

Initialisation des données

Diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean
B:357	Min. : 6.981	Min. : 9.71	Min. : 43.79	Min. : 143.5	Min. :0.05263
M:212	1st Qu.:11.700	1st Qu.:16.17	1st Qu.: 75.17	1st Qu.: 420.3	1st Qu.:0.08637
NA	Median :13.370	Median :18.84	Median : 86.24	Median : 551.1	Median :0.09587
NA	Mean :14.127	Mean :19.29	Mean : 91.97	Mean : 654.9	Mean :0.09636
NA	3rd Qu.:15.780	3rd Qu.:21.80	3rd Qu.:104.10	3rd Qu.: 782.7	3rd Qu.:0.10530
NA	Max. :28.110	Max. :39.28	Max. :188.50	Max. :2501.0	Max. :0.16340

FIGURE 2 – Sommaire des cinq premières variables

Les variables étant exprimées dans des unités différentes, il faut les standardiser. A ce stade, seuls les paramètres calculés sur l'échantillon d'apprentissage doivent intervenir dans la transformation des données. De fait, on utilise exclusivement les moyennes et écarts-type calculés sur l'échantillon d'entraînement pour normaliser les deux échantillons.

```

1 #calcul des moyennes sur l'échantillon d'apprentissage
2 mean.train <- sapply(data.train[,-1],mean)
3 #idem pour l'écart-type
4 sd.train <- sapply(data.train[,-1],sd)
5
6 #centrage-réduction de l'échantillon d'apprentissage
7 data.train.cr <- data.frame(Diagnosis = data.train[,1], scale(data.train[,-1],center=mean.train , scale
     =sd.train))
8 #vérification
9 print(colMeans(data.train.cr[,-1]))
10
11 #centrage-réduction de l'échantillon de test
12 #avec les paramètres (moyenne, écart-type) calculés sur l'échantillon d'apprentissage
13 data.test.cr <- data.frame(Diagnosis = data.test[,1], scale(data.test[,-1],center=mean.train , scale=sd
     .train))
14 #moyenne pas forcément nulle <- logique
15 print(colMeans(data.test.cr[,-1]))

```

Standardisation des données

	mean_train	mean_test
radius_mean	0	-0.0555226
texture_mean	0	0.1289337
perimeter_mean	0	-0.0564350
area_mean	0	-0.0559723
smoothness_mean	0	-0.0139043
compactness_mean	0	-0.0204380
concavity_mean	0	-0.0287846
concave.points_mean	0	-0.0214868
symmetry_mean	0	0.1033931
fractal.dimension_mean	0	0.0363009
radius_se	0	-0.0516086
texture_se	0	0.1898560
perimeter_se	0	-0.0476297
area_se	0	-0.0756037
smoothness_se	0	-0.0006639
compactness_se	0	-0.0168975
concavity_se	0	-0.0385219
concave.points_se	0	0.0066105
symmetry_se	0	0.0384444
fractal.dimension_se	0	-0.0174919
radius_largest	0	-0.0481792
texture_largest	0	0.1233574
perimeter_largest	0	-0.0537411
area_largest	0	-0.0612360
smoothness_largest	0	0.0566957
compactness_largest	0	0.0016682
concavity_largest	0	-0.0219768
concave.points_largest	0	-0.0091084
symmetry_largest	0	0.1032310
fractal.dimension_largest	0	0.0377822

FIGURE 3 – Moyennes des variables des deux échantillons

Les moyennes des variables sont toutes nulles (aux erreurs de troncature près). On fait de même pour l'échantillon test, les moyennes des variables transformées de l'échantillon test ne sont pas forcément nulles puisque les paramètres de centrage (et de réduction) ont été calculés sur le premier échantillon.

Evaluation des performances :

Pour évaluer les performances prédictives des modèles, on rédige une fonction qui fournira tous les résultats souhaités. Elle prend en entrée la cible, la prédiction et une indication sur la modalité-cible ("M" par défaut pour malin). Elle affiche la matrice de confusion, le taux d'erreur, le rappel, la précision et le F1-Score (F-Mesure).

```

1 evaluation.prediction <- function(yobs, ypred, posLabel="M") {
2   #matrice de confusion
3   mc <- table(yobs, ypred)
4   print("Matrice de confusion")
5   print(mc)
6   #taux d'erreur
7   err <- 1-sum(diag(mc))/sum(mc)
8   print(paste("Taux d'erreur =", err))
9   #rappel
10  recall <- mc[posLabel, posLabel]/sum(mc[, posLabel])
11  print(paste("Rappel =", round(recall,3)))
12  #precision
13  precision <- mc[posLabel, posLabel]/sum(mc[, posLabel])
14  print(paste("Precision =", round(precision,3)))
15  #F1-Measure
16  f1 <- 2.0*(precision*recall)/(precision+recall)
17  print(paste("F1-Score =", round(f1,3)))
18 }
```

Fonction d'évaluation des performances

2 Perceptron Simple

2.1 Modélisation

Un réseau de neurones est un graphe orienté pondéré. Un noeud de ce réseau est appelé un neurone (formel).

L'architecture du réseau est définie par le nombre de neurones, leur type ainsi que leur connectivité.

Le modèle général de réseau de neurones artificiels est appelé le **perceptron**.

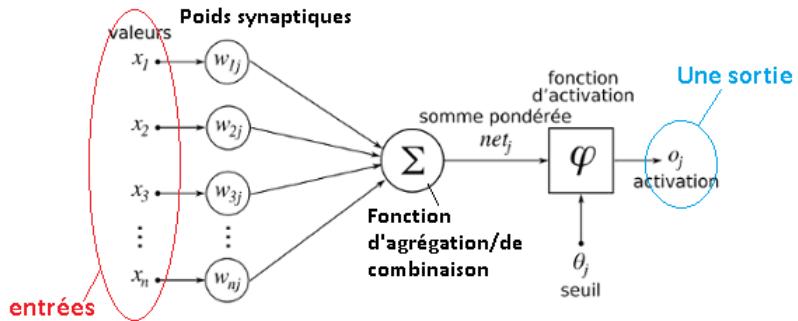


FIGURE 4 – Schéma perceptron

Il s'agit d'un réseau à seule couche, qui a une sortie unique.

On note que notre réseau possède $n > 1$ neurones.

Dans la figure ci-dessus, pour une seule observation j , x_0, x_1, \dots, x_n représente les différentes entrées (variables indépendantes) du réseau. Chacune de ces entrées est multipliée par un **poids synaptique**. Les poids sont représentés par $w_{0,j}, w_{1,j}, \dots, w_{n,j}$. Ces poids représentent les pondérations entre le neurone j et les neurones $(x_i)_{1 \leq i \leq n}$.

La matrice des poids synaptiques représente donc la connectivité du réseau. Elle est notée :

$$W = (w_{i,j})_{1 \leq i, j \leq n}$$

Parfois les neurones possèdent une valeur **seuil**, notée θ_j ici.

Chaque neurone possède également un état, appelé **activation**. Le vecteur $a = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}$. Souvent ces états appartiennent à un ensemble de booléen, ou numérique (entiers, réels,...) . (*Remarque : cette fonction est notée o_i sur le schéma de la Figure 3*)

On verra par la suite que cet état d'activation est fourni par la fonction d'activation.

On introduit tout d'abord la **fonction d'entrée** h_i , cette fonction sera identique pour tous les neurones du réseau ou ceux d'une même couche la plupart du temps, on la note donc h dans la suite du projet.

Cette fonction dépend du vecteur d'activation a et de la matrice des poids W . Cette fonction peut aussi dépendre d'un biais, noté ϕ_i pour le neurone i .

- Lorsque a est un booléen, la fonction h est booléenne et dépend du biais ϕ_i .
- Lorsque a est numérique, la fonction h est linéaire quand le biais est nul :

$$h(i) = h(i, a, W) := W_i \cdot a = \sum_{j=1}^n w_{i,j} a_j$$

Et est affine sinon :

$$h(i) = h(i, a, W, \phi_i) := \sum_{j=1}^n w_{i,j} a_j - \phi_i = \sum_{j=0}^n w_{i,j} a_j$$

avec $w_{i,0} = \phi_i$ et $a_0 = -1$. On remarque que les biais sont traités grâce à un neurone supplémentaire x_0 .

Cas particulier : Pour les réseaux RBF (Radial Basis Function), la fonction d'entrée h est une distance entre le vecteur d'activation a et la transposée de du vecteur W_i .

On introduit maintenant la notion **d'activation pondérée** (ou potentiel) d'un neurone à l'instant $t > 0$, noté $e_i(t)$. On a donc :

$$e_i(t) = h(i, a(t), W)$$

(Remarque : cette fonction est notée net_j sur le schéma de la Figure 3)

Chaque neurone possède une **fonction d'activation** f_i qui permet de convertir un signal d'entrée en un signal de sortie (qui sera ensuite utilisé comme signal d'entrée de la fonction suivante). (Remarque : cette fonction est notée ϕ sur le schéma de la Figure 3)

En général cette fonction est la même pour tous les neurones d'une couche, et peut être noté simplement f . Elle s'applique au résultat de la fonction d'entrée h_i , donc à l'activation pondérée e_i

A l'instant $t > 0$, on a donc :

$$a_i(t) = f_i(e_i(t-1), \theta_i)$$

Exemples de fonctions d'activation

- La **fonction linéaire** :

$$f(x) = \lambda x$$

- La **fonction de seuil** : Soit le seuil noté θ ,

$$f(x) = \begin{cases} 1 & \text{si } x \geq \theta \\ 0 & \text{sinon} \end{cases}$$

- La **fonction linéaire bornée** par θ^- et θ^+ :

$$f(x) = \begin{cases} M & \text{si } x > \theta^+ \\ kx & \text{si } \theta^- \leq x \leq \theta^+ \\ m & \text{si } x < \theta^- \end{cases}$$

- La **fonction sigmoïde exponentielle** (souvent utilisé avec le perceptron multicouches) :

$$f(x) = \frac{1}{1 + e^{-x}}$$

Et il en existe encore d'autres.

Equation générale :

Pour tenir compte de tous les types de fonctions d'activation nous pouvons écrire que l'activation $a_i(t+1)$ du neurone i à l'instant $t+1$ est donné par :

$$a_i(t+1) = a_i(t) + F(h(i)(t), \theta_i, I(t)) \quad (1)$$

où $f_i := a_i + F(x, \theta_i, I(t))$ est la fonction d'activation du neurone i , $h(i)(t)$ la fonction d'entrée à l'instant t , θ_i le seuil et $I(t)$ une valeur d'entrée à l'instant t . Ce qui dans la littérature s'écrira aussi sous la forme :

$$\frac{dx}{dt} = F(x, W, \theta, I) \quad (2)$$

permettant d'étudier l'évolution du réseau comme un système dynamique.

Quand on étudie un réseau de neurones, on regarde :

- Si le réseau comporte des boucles : s'il existe $i_1, \dots, i_p \in [1, n]$ tels que $i_1 = i_p$ et $w_{i_k, i_{k+1}} \leq 0$ pour $k \in [1, p-1]$
Si le réseau comporte des boucles on dit que c'est un réseau **récurent**. Sinon la matrice W est triangulaire.
- La structure des couches de neurones : s'il y a de la connectivité intra-couche.
- La connectivité inter-couches
- La symétrie des connections (exemple : Réseau de Hopfield)

En fonction de l'utilité du réseau de neurones, on doit fixer :

- Une architecture (nombre de neurones et connectivité)
- Une matrice des poids, en utilisant un poids égal à 0 quand les neurones ne sont pas connectés.
- Les fonctions d'entrée et d'activation de chaque neurone (en général identiques pour tous les neurones)

2.2 Apprentissage

Un réseau fonctionne selon deux modes différents :

- En mode **reconnaissance** : le réseau est utilisé pour calculer.
- En mode **apprentissage** : l'apprentissage des réseaux de neurones permet d'ajuster la matrice de poids synaptiques à une application. Pour cela on utilise un échantillon d'apprentissage.

L'entraînement d'un perceptron est un processus itératif. Après chaque observation, les poids de connexion sont ajustés pour réduire l'erreur de prédiction faite par le perceptron dans son état actuel.

Pour cela on utilise une **règle d'apprentissage**, qui permet d'adapter un réseau à une application. Cette règle porte le plus souvent sur la matrice des poids synaptiques. On distingue l'apprentissage supervisé de l'apprentissage non supervisé.

Pour modifier la matrice W des poids synaptiques on utilise un échantillon d'apprentissage, appelé **corpus** \mathcal{C} . On considère le réseau comme une "boîte noire" avec k neurones en entrée et m neurones en sortie.

- Pour **l'apprentissage supervisé**, le corpus \mathcal{C} est un ensemble de couples (x, d) où x est le vecteur des k neurones d'entrées, et d le vecteur des m valeurs désirées en sortie, que nous connaissons a priori.

On fournit le vecteur x à l'entrée du réseau et on récupère un vecteur de sortie qui sera noté s , qui comporte m valeurs. Il faut ensuite comparer s et d pour effectuer les corrections sur les poids synaptiques. Pour cela on calcule une fonction d'erreur en d et s qui sera minimisé en modifiant la matrice des poids W .

Les fonctions d'erreur les plus utilisées sont :

- La fonction **d'erreur quadratique** :

$$E = \frac{1}{2} \sum_{i=1}^m (d_i - s_i)^2 \quad (3)$$

- La fonction de **corrélation croisée** qui s'adapte aux cas la vitesse de convergence est plus grande :

$$E = - \sum_{i=1}^m (d_i \ln(s_i) + (1 - d_i) \ln(1 - s_i)) \quad (4)$$

Ce mode d'apprentissage est utilisé pour faire de la prédiction, de l'identification, de la classification...

L'apprentissage consiste en l'initialisation $W = W(0)$ de la matrice W des poids synaptiques et en l'exécution de plusieurs cycles d'apprentissage jusqu'à ce que la fonction d'erreur $E(s, d)$ atteigne un minimum.

Soit $W = W(t)$ l'état de la matrice des poids à l'instant t , un cycle d'apprentissage consiste à :

1. présenter le patron d'entrée x au réseau et calculer s en propageant l'activation,
2. calculer l'erreur $E(s, d)$ entre s et d ,
3. en déduire la nouvelle matrice des poids $W(t + 1)$.

La formule établissant $W(t + 1)$ est la **règle d'apprentissage** des poids du réseau.

- Pour **l'apprentissage non supévisé**, on ne connaît pas le vecteur de sortie d . Il s'agit donc de regrouper les $(x_i)_{1 \leq i \leq n}$ du corpus d'apprentissage. Le but est donc de modifier la matrice des poids W afin qu'ils soient suffisamment regroupés.

Ce mode d'apprentissage est utilisé pour faire de l'auto-classification par exemple.

Exemples de règles d'apprentissage :

- **Le principe de Hebb**

Ce principe permet de modifier les liens entre les neurones.

$$w_{i,j} = \lambda a_i \bar{a}_j$$

où $a_i \bar{a}_j$ est la corrélation entre a_i et a_j et $\lambda \in [0, 1]$ est le paramètre de l'intensité d'apprentissage.

L'idée est la suivante : si deux neurones connectés entre eux sont activés de la même manière (au même moment) alors la connexion qui les relie doit être renforcée. Sinon, elle doit rester inchangée ou être affaiblie.

- **La règle d'apprentissage DELTA :**

Le perceptron est utilisé pour la classification. La règle d'apprentissage du perceptron est la règle DELTA. Nous avons une couche d'entrée et une de sortie.

Soit la sortie s_i du neurone i de sortie et d_i sa sortie désirée. Alors la règle d'apprentissage du poids $w_{i,j}$ entre le neurone j (de la couche précédente) et le neurone i est la suivante :

$$\Delta w_{i,j} = \lambda \cdot a_i \cdot (d_i - s_i) \quad (5)$$

où $\lambda > 0$ est le pas d'apprentissage.

Théorème (Convergence de Rosenblatt, 1962). Quelles que soient les entrées et la classification désirée, avec la règle DELTA, la convergence s'établit en un temps fini vers une matrice des poids W correcte si elle existe.

Notons S l'ensemble des indices des neurones de sortie.

La règle DELTA d'apprentissage effectue une descente en gradient sur cette fonction d'erreur :

$$E = \frac{1}{2} \sum_{s \in S} (a_s - d_s)^2$$

Comme E est une somme de carrés, le minimum sera unique. La correction des poids sera donnée par l'équation

$$\Delta w_{i,j} = -\lambda \frac{dE}{dw_{i,j}}$$

où $\lambda > 0$ est le pas d'apprentissage.

On veut calculer $\frac{dE}{dw_{i,j}}$ où le neurone i est celui de la couche de sortie S . On a :

$$\frac{dE}{dw_{i,j}} = \frac{dE}{dh(i)} \cdot \frac{dh(i)}{dw_{i,j}}$$

Pour de réutiliser ces calculs ultérieurement pour le PMC, oublions pour le moment que la fonction f d'activation est l'identité et considérons l'identité plus générale $a_i = f(h(i))$. Nous avons alors :

$$\frac{dE}{dh(i)} = \frac{1}{2} \frac{d(a_i - d_i)^2}{dh(i)} = \frac{1}{2} \frac{d(a_i - d_i)^2}{da_i} \cdot \frac{df(h(i))}{dh(i)}$$

d'où :

$$\frac{dE}{dh(i)} = (a_i - d_i)^2 f'(h(i)) \quad (6)$$

et

$$\frac{dh(i)}{dw_{i,j}} = a_j \quad (7)$$

car $h(i) = \sum w_{i,k} a_k$

On établit donc que :

$$\frac{dE}{dw_{i,j}} = a_j (a_i - d_i)^2 f'(h(i)) \quad (8)$$

et donc la règle d'apprentissage pour toute fonction d'activation f est la suivante :

$$\Delta w_{i,j} = \lambda a_j (a_i - d_i)^2 \cdot f'(h(i)) \quad (9)$$

où i est un neurone de sortie. On trouve la règle DELTA en posant $f = id$.

Il existe plusieurs modèles classiques qui permettent de répondre à des problèmes particuliers. Voici un tableau qui présente de manière très simple la forme de différents réseaux de neurones :

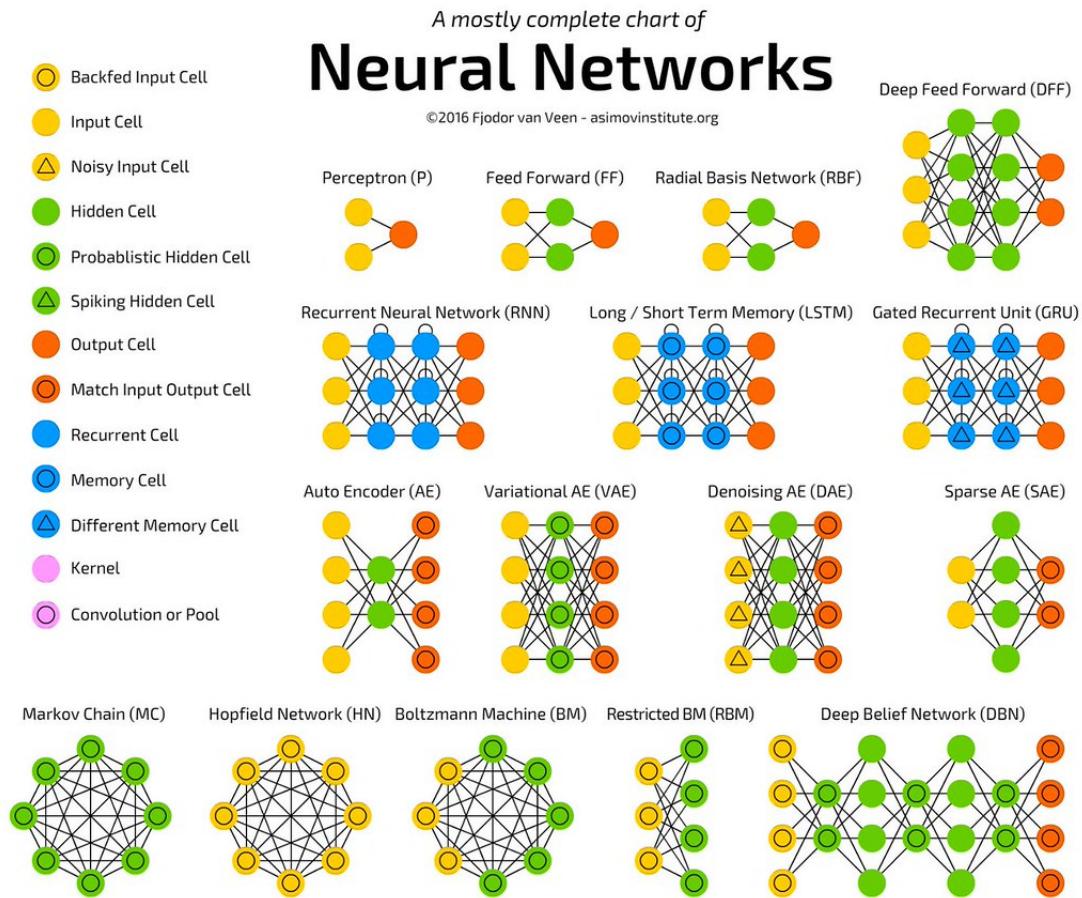


FIGURE 5 – Exemples de Réseaux de neurones connus

Dans la suite de ce projet on présentera :

- Le perceptron multicouches
- Les cartes de Kohonen
- La machine de Boltzmann Restreinte
- Les réseaux de croyances profondes

2.3 Application : avec le package nnet

Le package `nnet` est ancien mais très populaire, notamment parce qu'il est performant, robuste, facile à utiliser. La seule restriction est que son perceptron multicouche n'accepte qu'une seule couche cachée.

Dans cette section, nous commençons par un perceptron simple pour circonscrire les performances d'un classifieur linéaire sur nos données. L'intérêt aussi est de pouvoir identifier facilement, grâce à la valeur des coefficients, l'impact des variables dans la prédiction.

Les deux paramètres clés sont `skip = TRUE` (pas de couche cachée) et `size = 0` (par conséquent, pas de neurones dans la couche cachée). Pour le reste (`formule`, `data`), nous retrouvons les paramètres usuels utilisés dans les fonctions R pour la prédiction.

```
1 #chargement package
2 library(nnet)
3 #apprentissage perceptron simple
4 #set.seed() pour rendre l'expérimentation reproductible pour le lecteur
5 set.seed(100)
6 ps.nnet <- nnet(Diagnosis ~ ., data = data.train.cr, skip = TRUE, size = 0)
```

```
# weights:  31
initial value 248.274015
iter  10 value 19.651860
iter  20 value 13.870568
iter  30 value 11.176763
iter  40 value 6.627508
iter  50 value 0.021681
final value 0.000099
converged
```

On regarde les performances prédictives : on effectue la prédiction sur l'échantillon test, et on confronte les valeurs prédites et observées.

```
1 #prediction
2 pred.ps.nnet <- predict(ps.nnet, newdata=data.test.cr, type="class")
3 #évaluation
4 evaluation.prediction(data.test.cr$Diagnosis, pred.ps.nnet)
```

```
[1] "Matrice de confusion"
     ypred
yobs   B    M
      B 106    3
      M   5  57
[1] "Taux d'erreur = 0.0467836257309941"
[1] "Rappel = 0.919"
[1] "Precision = 0.95"
[1] "F1-Score = 0.934"
```

On a (5+3) 8 observations mal classés sur 171, cela équivaut à 4,7% des données. Ce résultat est plutôt bon.

`nnet` propose des outils supplémentaires pour mieux cerner la qualité du modèle. On dispose par exemple les probabilités conditionnelles calculées durant l'apprentissage. On affiche l'histogramme de fréquences.

```
1 #distribution des probas estimées en apprentissage
2 hist(ps.nnet$fitted.values)
```

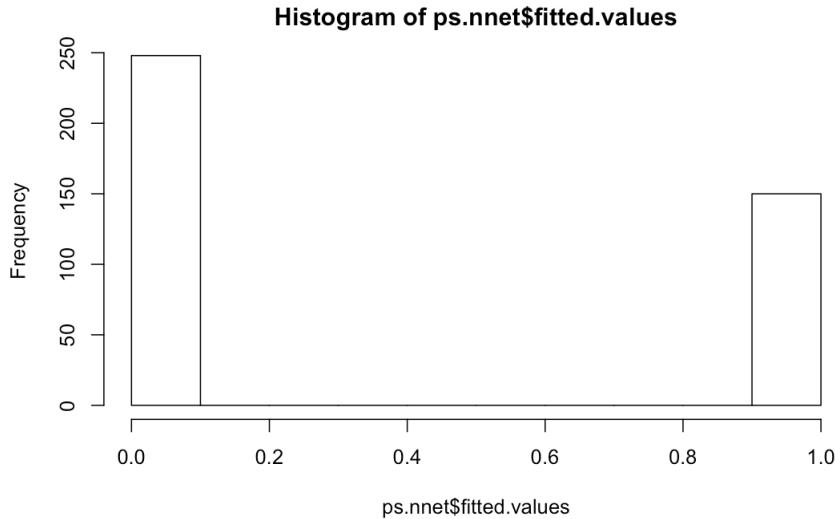


FIGURE 6 –

Les décisions sont tranchées sur la quasi-totalité des observations, les probabilités sont proches de 0 ou 1, ce qui explique la bonne tenue du modèle en classement.

Nous disposons d'un affichage détaillé du réseau :

```
1 #les poids
2 print(summary(ps.nnet))
```

```
a 30-0-1 network with 31 weights
options were - skip-layer connections entropy fitting
    b->o      i1->o      i2->o      i3->o      i4->o      i5->o      i6->o      i7->o      i8->o      i9->o
  307.33   -5744.26    570.35   -832.35   -399.67   1233.31   -2664.39    1.33   2701.95   -387.95
    i10->o     i11->o     i12->o     i13->o     i14->o     i15->o     i16->o     i17->o     i18->o     i19->o
   34.96    3536.78   -369.08  -2668.02   -811.31    247.63   -796.41   -1066.16   2580.22   -14.65
    i20->o     i21->o     i22->o     i23->o     i24->o     i25->o     i26->o     i27->o     i28->o     i29->o     i30->o
 -1676.74    872.90    854.56   6091.27   4166.43   -865.88    308.77   2457.18  -1539.39    532.60   1243.67
```

On observe l'essentiel de l'information : "b->o" représente la connexion entre le biais et la sortie, le coefficient associé est 4.01 ; "i1->o", la connexion entre la première variable et la sortie avec 0.06 ; etc.

On peut également récupérer explicitement les poids synaptiques. On les trie selon leur absolue et on affiche les plus grands.

```
1 #récupération des poids synaptiques
2 poids <- ps.nnet$wts[2:length(ps.nnet$wts)]
3 names(poids) <- ps.nnet$coefnames
4
5 #les variables avec les poids les plus élevés en valeur absolue
6 print(sort(abs(poids),decreasing=TRUE)[1:10])
```

perimeter_largest	6091.2683
radius_mean	5744.2557
area_largest	4166.4284
radius_se	3536.7806
concave.points_mean	2701.9541
perimeter_se	2668.0246
compactness_mean	2664.3870
concave.points_se	2580.2196
concavity_largest	2457.1764
fractal.dimension_se	1676.7441

3 Le Perceptron Multicouches (MLP)

3.1 Modélisation

Un perceptron multicouches (PMC) est une classe de réseau de neurones artificiel (RNA) à action anticipée.

Un PMC est constitué d'au moins trois couches de nœuds : une couche d'entrée, une couche cachée et une couche de sortie. À l'exception des nœuds d'entrée, chaque nœud est un neurone qui utilise une fonction d'activation non linéaire. Le PMC utilise une technique d'apprentissage supervisé appelée rétropropagation pour la formation.

Ses multiples couches et son activation non linéaire (sigmoïde la plupart du temps) distinguent le PMC d'un perceptron linéaire. Elle peut distinguer les données qui ne sont pas séparables linéairement.

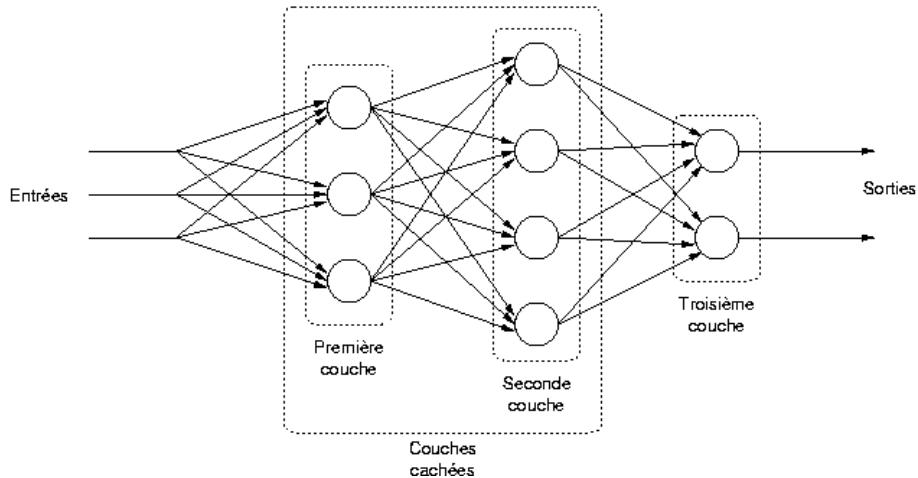


FIGURE 7 – Schéma Perceptron Multicouches

La fonction d'entrée du neurone i est linéaire pour la couche cachée et la couche de sortie :

$$h(i) = \sum_{j=1}^N w_{i,j} a_j$$

La fonction d'activation f_i pour la couche cachée est définie par une sigmoïde exponentielle, qui est facilement dérivable :

$$f_i(x) = \frac{1}{1 + e^{-x}}$$

Celle de la couche de sortie est linéaire la plupart du temps (ie $f_i = id$)

La rétropropagation consiste à corriger les poids synaptiques des neurones au cours de l'apprentissage. Cela consiste à modifier d'abord la matrice des poids qui relie les neurones de la couche de sortie aux neurones de la couche d'entrée.

On remarque que pour connaître l'erreur sur les couches cachées, il est nécessaire de connaître celles des neurones auxquels ils sont reliés.

3.2 Apprentissage

La règle de modification des poids est la suivante :

$$\Delta w_{i,j} = -\lambda \frac{dE}{dw_{i,j}}$$

où j est le neurone de la couche précédente à celle du neurone i .

La fonction d'erreur E des neurones de la couche de sortie S est aussi une fonction quadratique :

$$E = \frac{1}{2} \sum_{i \in S} (a_i - d_i)^2$$

Et donc la règle delta généralisée d'apprentissage pour une neurone i de la couche de sortie S est donnée par :

$$\Delta w_{i,j} = \lambda \cdot a_j \cdot (d_i - a_i) \cdot f'(h(i))$$

Comme préciser dans la partie précédente, il faut ensuite trouver la règle d'apprentissage des neurones d'une couche cachée, notée C .

On doit donc calculer $\frac{dE}{dw_{i,j}}$ où le neurone i est celui de la couche cachée C . On rappelle qu'on a :

$$\frac{dE}{dw_{i,j}} = \frac{dE}{dh(i)} \cdot \frac{dh(i)}{dw_{i,j}} = a_j \cdot \frac{dE}{dh(i)}$$

On utilise la même technique que pour la règle DELTA (cf partie précédente).

On calcule donc $\frac{dE}{dh(i)}$, on note k les neurones de la couche de sortie S , on a alors :

$$\frac{dE}{dh(i)} = \frac{1}{2} \sum_{k \in S} \frac{d(a_k - d_k)^2}{dh(i)} = \sum_{k \in S} \frac{dE}{dh(k)} \cdot \frac{dh(k)}{da_i} \cdot \frac{df(h(k))}{dh(i)}$$

car $a_i = f(h(i))$.

et donc :

$$\frac{dE}{dw_{i,j}} = a_j f'(h(i)) \sum_{k \in S} w_{k,j} (a_k - d_k) f'(h(k))$$

car $\frac{dE}{dh(k)} = (a_k - d_k) f'(h(k))$, $\forall k \in S$, et $h(k) = \sum_{j \in C} w_{k,j} a_j$

Et on a :

$$\Delta w_{i,j} = \lambda \cdot a_j \cdot f'(h(i)) \sum_{k \in S} w_{k,j} (d_k - a_k) \cdot f'(h(k))$$

Note : on peut généraliser ce résultat à plusieurs couches cachées, où on note j le neurone appartenant à la couche précédente de celle à laquelle appartient le neurone i . On a alors :

$$\Delta w_{i,j} = \lambda \cdot a_j \cdot (d_i - a_i) \cdot f'(h(i))$$

3.3 Application

3.3.1 Avec le package nnet

On ajuste notre modèle de réseau de neurones sur la base d'apprentissage en utilisant le package `nnet` :

Le pas d'apprentissage est initialisé avec le paramètre `decay`. On joue ensuite sur les paramètres `skip` et `size` pour spécifier un perceptron à 2 neurones dans l'unique couche cachée. On augmente le nombre maximal d'itérations pour aboutir à la convergence.

```
1 #PMC avec 1 couche cachée à 2 neurones
2 set.seed(100)
3 pm.nnet <- nnet(Diagnosis ~ ., data = data.train.cr, skip = FALSE, size = 2, maxit = 1000, decay =
   0.001)
```

```
# weights:  65
initial value 331.398173
iter  10 value 28.622108
iter  20 value 8.309987
iter  30 value 4.222551
iter  40 value 3.786220
iter  50 value 3.626460
iter  60 value 3.512411
iter  70 value 3.460880
iter  80 value 3.202861
iter  90 value 2.841932
iter 100 value 2.201706
iter 110 value 1.997418
iter 120 value 1.942199
iter 130 value 1.916326
iter 140 value 1.892471
iter 150 value 1.716280
iter 160 value 1.559256
iter 170 value 1.474349
iter 180 value 1.445190
iter 190 value 1.433499
iter 200 value 1.423871
iter 210 value 1.420442
iter 220 value 1.418410
iter 230 value 1.414488
iter 240 value 1.406182
iter 250 value 1.390398
iter 260 value 1.385397
iter 270 value 1.382892
iter 280 value 1.381948
iter 290 value 1.381776
iter 300 value 1.381715
iter 310 value 1.381700
iter 320 value 1.381695
iter 330 value 1.381693
final value 1.381692
converged
```

```
1 #prediction
2 pred.pm.nnet <- predict(pm.nnet, newdata=data.test.cr, type="class")
3 #evaluation
4 evaluation.prediction(data.test.cr$Diagnosis, pred.pm.nnet)
```

```
[1] "Matrice de confusion"
     ypred
yobs   B    M
      B 107    2
      M   3   59
[1] "Taux d'erreur = 0.0292397660818714"
[1] "Rappel = 0.952"
[1] "Precision = 0.967"
[1] "F1-Score = 0.959"
```

Le modèle a une meilleure tenue avec (3+2) 5 observations mal classées (vs. 8 pour le perceptron simple).

Comme pour le perceptron simple on peut afficher les poids synaptiques.

Pour notre première tentative, nous avons pris des paramètres par défauts. A présent, l'objectif va être de trouver le `size` et le `decay` optimal pour notre réseau de neurones.

En effet, toute méthode d'apprentissage a ses propres paramètres que l'utilisateur doit ajuster pour construire un bon modèle à la fois adapté aux données de la base d'apprentissage et performant du point de vue de la prédiction sur les nouvelles observations.

Les 2 principaux paramètres de notre modèle sont : - `size` : le nombre de neurones de la couche cachée - `decay` : le pas d'apprentissage (paramètre de décomposition)

En faisant varier ces deux paramètres, on s'aperçoit que plus le modèle est complexe (plus `size` est grand), mieux il apprend sur les données et plus il est performant du point de vue de la prédiction car chaque neurone supplémentaire permet de prendre en compte des profils spécifiques de neurones d'entrée. On peut également remarquer qu'au-delà d'un certain seuil, la performance n'augmente plus, voire diminue la capacité de généralisation du réseau, c'est le sur-apprentissage.

Il n'existe pas de règle générale mais des règles empiriques. La taille de la couche cachée doit être :

- soit égale à celle de la couche d'entrée (Wierenga et Kluytmans, 1994)
- soit égale à 75% de celle-ci (Venugopal et Baets, 1994)
- soit égale à la racine carrée du produit du nombre de neurones dans la couche d'entrée et de sortie (Shepard, 1990).

En faisant varier le `decay`, nous constatons que plus la décomposition de nos neurones est importante, plus l'est la performance et réciproquement, s'il y a trop de décompositions, le modèle redevient moins performant. La problématique consiste à trouver le bon paramétrage du modèle.

A cette fin, nous utilisons la fonction `tune` du package `e1071` pour essayer de répondre à cette question :

```
1 library(MASS)
2 library(e1071)
3
4 tune.model = tune.nnet(Diagnosis ~ ., data = data.train.cr, size = c(1,2,3,4,5), decay = c(0.1,
0.001,0.0001,0.00001,0.000001))
5
6 plot(tune.model)
7 tune.model
```

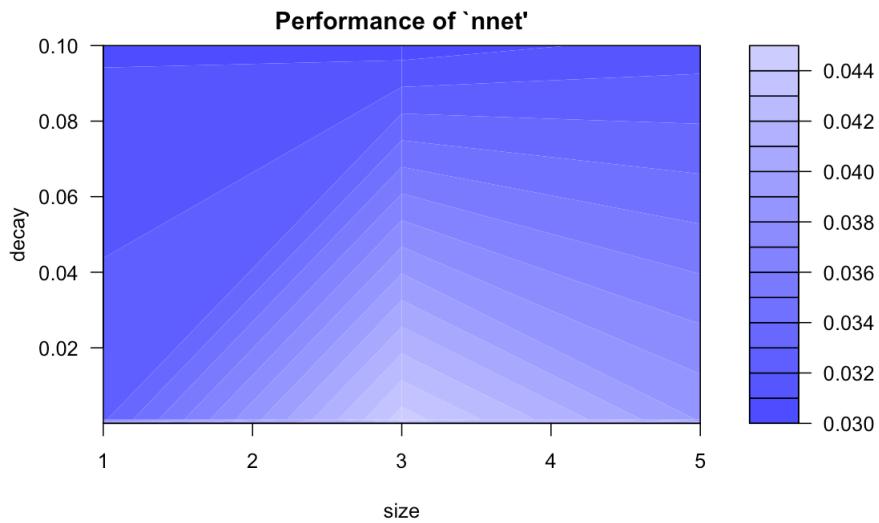


FIGURE 8 –

Parameter tuning of nnet :

- sampling method: 10-fold cross validation
 - best parameters:
 - size decay
 - 3 0.1
 - best performance: 0.03044872
-

On teste les nouveaux paramètres.

```

1 #PMC avec 1 couche cachée à 2 neurones
2 set.seed(100)
3 pm.nnet <- nnet(Diagnosis ~ ., data = data.train.cr, skip = FALSE, size = 3, maxit = 1000, decay =
  0.1)
```

```

# weights:  97
initial value 274.473756
iter  10 value 29.856111
iter  20 value 23.773751
iter  30 value 22.969975
iter  40 value 22.016077
iter  50 value 21.763836
iter  60 value 21.753742
final  value 21.753722
converged
```

```

1 #prediction
2 pred.pm.nnet <- predict(pm.nnet, newdata=data.test.cr, type="class")
3 #evaluation
4 evaluation.prediction(data.test.cr$Diagnosis, pred.pm.nnet)
```

[1] "Matrice de confusion"

```

ypred
yobs   B   M
      B 108   1
      M   3   59
[1] "Taux d'erreur = 0.0233918128654971"
[1] "Rappel = 0.952"
[1] "Precision = 0.983"
[1] "F1-Score = 0.967"

```

Il n'y a plus que 4 observations mal classées.

A présent, on peut réaliser une discrimination sur le même jeu de données. Nous allons utiliser le package `rpart`. On commence par réaliser le même découpage avec une partie pour l'apprentissage et une autre pour la validation. Puis on calcule notre modèle et on trace l'arbre de décision :

```

1 library(rpart)
2 model = rpart(Diagnosis~., data = data.train.cr, method = "class")
3 plot(model, uniform = TRUE, branch = 0.5, margin = 0.1, main = " Arbre de discrimination ")
4 text(model, all = FALSE, use.n = TRUE)

```

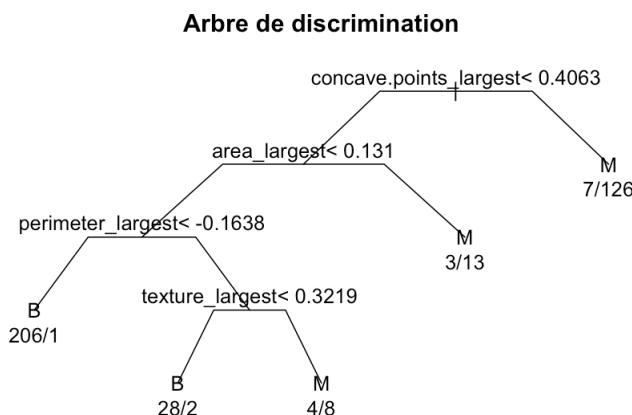


FIGURE 9 –

Ensuite, comme pour les réseaux de neurones, nous allons rechercher les paramètres optimaux pour la fonction `rpart`. Ici, nous avons à faire varier les paramètres suivants :

- `minsplit` : nombre minimal d'observations d'un noeud.
- `cp` : coefficient de complexité du modèle.
- `xval` : nombre de validations croisées
- `maxdepth` : profondeur maximale de l'arbre, sachant qu'à la racine, la profondeur vaut 0.

```

1 tune.model = tune.rpart (Diagnosis ~., data = data.train.cr, minsplit = c (15, 20, 25), cp = c
                           (0.00001,0.000001), maxcompete = 4, maxsurrogate = 5, usesurrogate = 2, xval = c (10, 15),
                           surrogatestyle = 0, maxdepth = c (25, 30))
2 tune.model

```

Parameter tuning of `rpart.wrapper` :

- sampling method: 10-fold cross validation

```

– best parameters:
  minsplit      15
  cp            1e-05
  maxcompete    4
  maxsurrogate   5
  usesurrogate   2
  xval          10
  surrogatestyle 0
  maxdepth     25
– best performance: 0.07775641

```

L'arbre qui nous intéresse est celui qui minimise l'erreur standard (`xerror + xstd` dans notre modèle). Si plusieurs arbres minimisent cette quantité, on privilégie l'arbre le plus petit. Ainsi il faut réaliser un élagage puis on peut afficher l'arbre de décision :

```

1 model = rpart (Diagnosis~., data = data.train , method = "class" , control = rpart.control (cp =
  0.00001, minsplit = 15, maxcompete = 4, maxsurrogate = 5,usesurrogate = 2, xval = 10,
  surrogatestyle = 0, maxdepth = 25))
2
3 cpTab = as.data.frame(model$cptable)
4 ind = cpTab$xerror + cpTab$xstd
5 cpTab = cbind.data.frame(cpTab, ind)
6 monCp = cpTab[which.min(cpTab$ind),"CP"]
7 elag = prune(model, cp = monCp)
8 plot(elag, uniform = TRUE, branch = 0.5, margin = 0.1, main = " Arbre de décision 2")
9 text(elag, all = FALSE, use.n = TRUE, cex = 0.6)

```

Arbre de décision 2

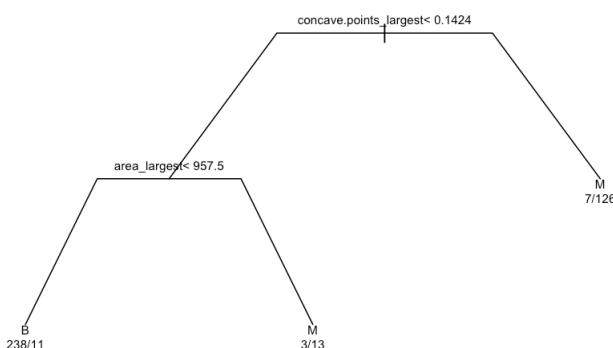


FIGURE 10 –

```

1 prev = predict(elag, newdata = data.test[, -1], type = "class")
2 mat = table(prev, data.test[, "Diagnosis"])
3 taux = sum(diag(mat))/sum(mat)
4 mat
5 taux

```

```

prev  B  M
B 99  6
M 10 56
[1] 0.9064327

```

Le taux d'erreur de prédiction est moins bon pour la discrimination (9%) que pour le réseau de neurones (2%). Cependant, la différence est minime et dépend de l'aléa du découpage de notre jeu de données. De plus, le critère de qualité de la prédiction (best performance) est meilleur pour le réseau de neurones que pour la discrimination avec un arbre de décision.

Ainsi, on peut affirmer que le Perceptron Multi-Couches est de qualité similaire à une technique de discrimination pour faire de la prévision.

3.3.2 Avec le package neuralnet

Le package `neuralnet` est également populaire parce qu'il propose une représentation graphique assez sympathique, et parce qu'il est possible d'exploiter les résultats intermédiaires.

Avec ce package, nous devons coder la variable cible catégorielle "Diagnosis" en une variable numérique. L'utilisation d'une indicatrice 0/1 semble s'imposer naturellement. Cependant c'est un piège, l'algorithme ne peut pas converger lors de l'apprentissage du modèle. Le problème vient de l'estimation des probabilités à l'aide de la fonction de transfert sigmoïde. Les valeurs sont très proches de 0 ou 1, à saturation, là où les dérivées (gradients) sont quasi-nulles. De fait, les corrections des coefficients se font mal durant le processus d'apprentissage. Il n'est pas possible de progresser efficacement vers la minimisation de la fonction de perte.

On décide de coder la cible en (0.8, 0.2) lorsqu'on utilise la fonction de transfert sigmoïde. Nous nous situerons ainsi dans la zone où sa pente reste importante.

```
1 #chargement – il faut installer au préalable
2 library(neuralnet)
3
4 #codage explicite de la cible – codage 0.8/0.2
5 y.Train <- ifelse(data.train$Diagnosis=="M", 0.8, 0.2)
6 print(table(y.Train))
```

```
y.Train
0.2 0.8
248 150
```

`neuralnet` ne sait pas gérer les expressions du type "cible ~ ." pour les formules. Nous devons spécifier explicitement les variables explicatives. Nous en avons $p = 30$.

```
1 #formule – liste des explicatives
2 formule <- paste(colnames(data.train[, -1]), collapse=" + ")
3
4 #suite formule
5 formule <- paste("y.Train ~ ", formule, sep="")
6 print(formule)
```

```
[1] "y.Train ~ radius_mean+texture_mean+perimeter_mean+area_mean+smoothness_mean+compactness_mean+
concavity_mean+concave.points_mean+symmetry_mean+fractal.dimension_mean+radius_se+texture_se+
perimeter_se+area_se+smoothness_se+compactness_se+concavity_se+concave.points_se+symmetry_se+
fractal.dimension_se+radius_largest+texture_largest+perimeter_largest+area_largest+smoothness_
largest+compactness_largest+concavity_largest+concave.points_largest+symmetry_largest+fractal.
dimension_largest"
```

On lance ensuite l'apprentissage avec 3 neurones sur la couche cachée, car c'est ce qui nous a fournir le meilleur résultat avec `nnet` :

```
1 #apprentissage
2 set.seed(100)
3 pm.neural <- neuralnet::neuralnet(as.formula(formule), data=data.train.cr, hidden = c(3), linear.
output = FALSE)
```

- `hidden` : permet de spécifier le nombre de neurones dans la couche cachée, s'il y a plusieurs couches, nous utilisons un vecteur ;
- `linear.output = FALSE` : introduit la fonction d'activation sigmoïde dans le neurone de sortie, elle est par défaut présente dans les couches intermédiaires.

Affichage du réseau avec les poids synaptiques :

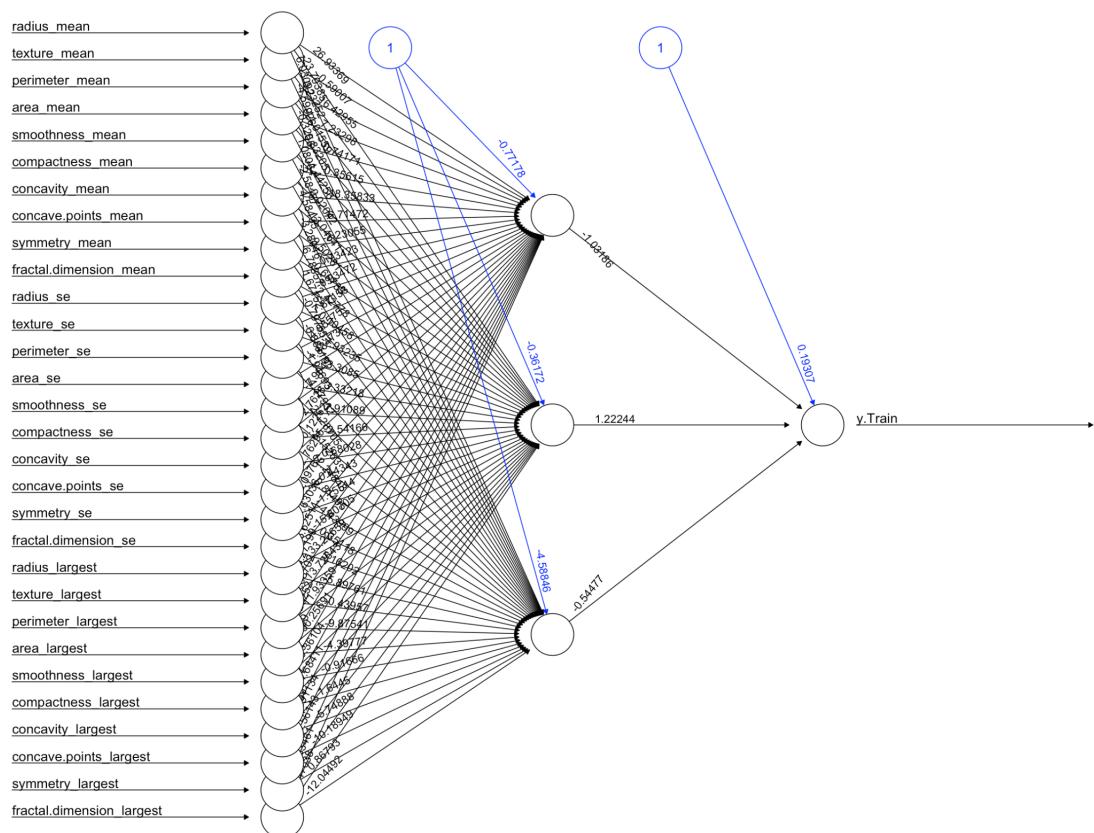


FIGURE 11 –

Le biais est bien présent sur chaque couche. Mais, lire sur ce graphique les poids entre les couches d'entrée et cachée n'est pas trop possible au regard du nombre de variables explicatives.

Nous utilisons la fonction `compute()` pour obtenir les probabilités d'affectation en prédition sur l'échantillon test. Voici les 10 premières valeurs.

```

1 #prédiction – proba d'affectation
2 proba.pred.pm.neural <- compute(pm.neural, covariate=data.test.cr[, -1])
3 print(proba.pred.pm.net.result[1:10])

```

```
[1] 0.8046332 0.8046332 0.3017882 0.8046332 0.8046332 0.2004870 0.2004360 0.8046332 0.8046332
0.8046332
```

Nous les comparons au seuil 0.5 pour obtenir les classes prédites. Il est dès lors possible de confronter les valeurs observées et prédites de la variable cible.

```

1 #traduire en "yes" "no" en comparant à 0.5
2 pred.pm.neural <- ifelse(proba.pred.pm.neural$net.result > 0.5, "M", "B")
3 #évaluation
4 evaluation.prediction(data.test.cr$Diagnosis, c(pred.pm.neural))

```

```

[1] "Matrice de confusion"
      ypred
yobs   B   M
      B 109   0
      M   5  57
[1] "Taux d'erreur = 0.0292397660818714"
[1] "Rappel = 0.919"
[1] "Precision = 1"
[1] "F1-Score = 0.958"

```

On trouve un résultat similaire à celui obtenu avec `nnet`. Cependant ici nous avons une valeur mal classée de plus. Les différences ne sont pas significatives.

Avec `neuralnet`, il est possible d'accéder aux coordonnées des individus dans les espaces intermédiaires définis par les couches cachées du réseau. Pour les 10 premiers individus de l'échantillon d'apprentissage dans la couche intermédiaire par exemple :

```

1 #coordonnées intermédiaires
2 hidden.neural <- proba.pred.pm.neural$neurons[[2]]
3 print(head(round(hidden.neural,10),10))

```

```

[,1]      [,2]      [,3]      [,4]
2     1 0.0000000000 1.00000e+00 0.0000000000
11    1 0.0000000000 1.00000e+00 0.0000004374
13    1 1.0000000000 0.00000e+00 0.0000000000
15    1 0.0000000004 1.00000e+00 0.0000000000
17    1 0.0000000000 1.00000e+00 0.0000000000
20    1 0.9997388918 3.60941e-05 0.9999907521
21    1 1.0000000000 0.00000e+00 1.0000000000
23    1 0.0000000000 1.00000e+00 0.0000000000
27    1 0.0000000000 1.00000e+00 0.0000000000
35    1 0.0000000000 1.00000e+00 0.0000000000

```

Ainsi, on peut regarder la disposition des individus dans un plan 3D car nous avons 3 neurones dans la couche cachée (la première colonne correspond au biais, toujours égale à 1) :

A partir de ces coefficients, on trace la droite de séparation dans l'espace de représentation intermédiaire défini par les neurones de la couche cachée.

```

1 #poids synaptiques entre couche cachée et sortie
2 poids.out <- pm.neural$weights[[1]][[2]]
3 print(poids.out)
4
5 library(scatterplot3d)
6 nuage3d <- scatterplot3d(hidden.neural[,2],hidden.neural[,3],hidden.neural[,4],color = "blue", main = "disposition des individus")

```

```
7 nuage3d$plane3d(-poids.out[1,1]/poids.out[4,1], -poids.out[2,1]/poids.out[4,1], -poids.out[3,1]/poids.out[4,1])
```

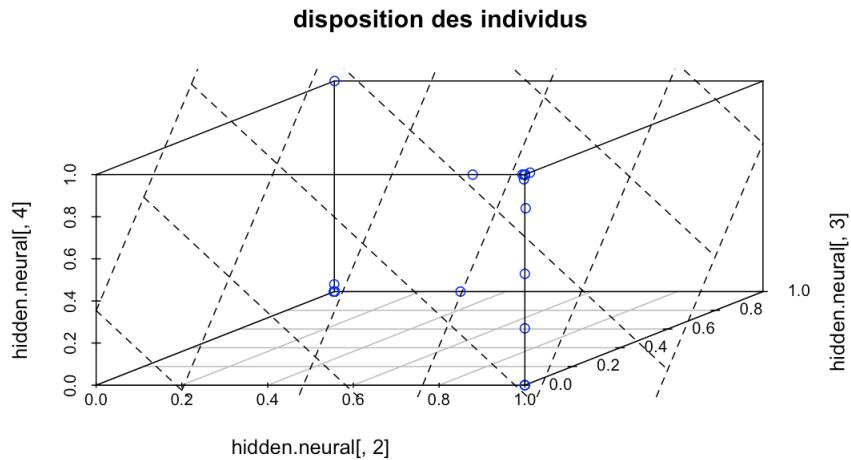


FIGURE 12 –

On semble remarqué deux groupes se former aux deux extrémités du cube, cela paraît logique car on a deux classes de cancer : "Malins" et "Benin"

3.3.3 Avec le package deepnet

Pour finir cette partie on teste le PMC avec un dernier package, deepnet, car c'est celui qui sera utilisé dans les autres applications logiciels de réseau.

deepnet est un package récent, avec un nombre de fonctions assez réduit (cf. la documentation). Après avoir installé et chargé le package, nous pouvons lancer directement la modélisation. La fonction activation par défaut est sigmoïde. Pour cette raison l'application est très rapide.

On prépare les données comme pour le package neuralnet, on code la cible en (0.8, 0.2).

```
1 #codage explicite de la cible – codage 0.8/0.2
2 Malin.train <- ifelse(data.train$Diagnosis=="M",0.8,0.2)
3 print(table(Malin.train))
4
5 #préparation pour l'apprentissage
6 data.num.train <- as.matrix(data.train[, -1]) # on supprime la colonne "Diagnosis"
7 print(head(data.num.train, 2))
8
9 #préparation pour prédiction
10 Malin.test <- ifelse(data.test$Diagnosis=="M",0.8,0.2)
11 data.num.test <- as.matrix(data.test[, -1])
```

```
1 #chargement
2 library(deepnet)
3 #apprentissage
4 set.seed(100)
5 pm.dpn <- nn.train(x = data.num.train, y = Malin.train, hidden = c(2), numepochs = 150)
6
7 #proba prediction
8 proba.pred.dpn <- nn.predict(pm.dpn, data.num.test)
9 summary(proba.pred.dpn)
10
11 #prédiction
12 pred.dpn <- ifelse(proba.pred.dpn > 0.5, "M", "B")
13
14 #évaluation
15 evaluation.prediction(data.test[, "Diagnosis"], pred.dpn)
```

```
[1] "Matrice de confusion"
     ypred
yobs   B   M
      B 109   0
      M   2  60
[1] "Taux d'erreur = 0.0116959064327485"
[1] "Rappel = 0.968"
[1] "Precision = 1"
[1] "F1-Score = 0.984"
```

Les résultats sont meilleurs par rapport aux autres packages. Ici nous n'avons plus que 2 individus mal placés.

On dispose des poids synaptiques en plus.

```
1 #poids synaptiques
2 print(pm.dpn$W)
```

```

[[1]]
  radius_mean texture_mean perimeter_mean area_mean smoothness_mean compactness_mean
[1,] -0.2086045 -0.2802986 -0.1048323 -0.2470292 -0.1341858 0.040228773
[2,] -0.3765456 -0.3817920 -0.3470434 -0.3300113 -0.1891856 -0.007268839
[3,] 0.1500724 0.1808082 0.1448572 0.2419764 0.1172994 -0.110480775

  concavity_mean concave.points_mean symmetry_mean fractal.dimension_mean radius_se texture_se
[1,] -0.2471164 -0.1981176 -0.04002863 0.2079649 -0.1872785 0.13409392
[2,] -0.3887650 -0.4507664 -0.10040256 0.2986830 -0.2836897 0.15628730
[3,] 0.2300209 0.2862320 0.04921899 -0.2030908 0.1870820 0.01298799

  perimeter_se area_se smoothness_se compactness_se concavity_se concave.points_se
[1,] -0.2003316 -0.2410961 -0.01699576 0.1401237 -0.06227338 -0.08673734
[2,] -0.2462375 -0.3782801 -0.13959187 0.3637348 -0.05486283 -0.09504653
[3,] 0.2490909 0.2639064 0.14810015 -0.1335658 -0.04589046 -0.04361803

  symmetry_se fractal.dimension_se radius_largest texture_largest perimeter_largest
[1,] 0.05152560 0.09063271 -0.2790040 -0.2903244 -0.2626641
[2,] 0.01179186 0.32603205 -0.5130674 -0.6751564 -0.5142625
[3,] -0.07207004 -0.23618556 0.3676358 0.3176071 0.2000906

  area_largest smoothness_largest compactness_largest concavity_largest concave.points_largest
[1,] -0.2256439 -0.2421591 -0.04112709 -0.1709674 -0.2624865
[2,] -0.5515233 -0.4413333 -0.01655187 -0.5450359 -0.4594767
[3,] 0.2450393 0.2857213 0.05938513 0.2215047 0.3668747

  symmetry_largest fractal.dimension_largest
[1,] -0.2703163 -0.08568981
[2,] -0.3451645 -0.07752763
[3,] 0.2328711 0.05499099

[[2]]
  [,1]      [,2]      [,3]
[1,] -0.5822358 -1.694453 0.9790122

```

[[1]] pour les connexions entre les couches d'entrées et cachées ; [[2]] entre l'intermédiaire et la sortie.

Les poids associés aux biais sont également disponibles.

```

1 #biais
2 print(pm.dpn$B)

```

```

[[1]]
[1] 0.14876175 0.35171400 -0.01428369

[[2]]
[1] 0.5941905

```

4 Carte de Kohonen

4.1 Principe

Les cartes de Kohonen, aussi appelé cartes auto-adaptives, appartiennent à une famille de réseau de neurone artificiel qui utilise l'apprentissage non supervisé pour produire une représentation en faible dimension (généralement bidimensionnelle) et discréteriser l'espace d'entrée des échantillons d'entraînement, appelée carte. C'est une méthode pour faire de la réduction de dimension. Les cartes de Kohonen diffèrent des autres réseaux neuronaux artificiels dans la mesure où elles appliquent un apprentissage compétitif par opposition à un apprentissage par correction d'erreurs (comme la rétropropagation avec descente de gradient), et dans le sens où elles utilisent une fonction de voisinage pour préserver les propriétés topologiques de l'espace d'entrée.

Les cartes de Kohonen sont donc utiles pour la visualisation grâce à la création de vues en basse dimension pour des données en grande dimension, un peu comme une mise à l'échelle multidimensionnelle.

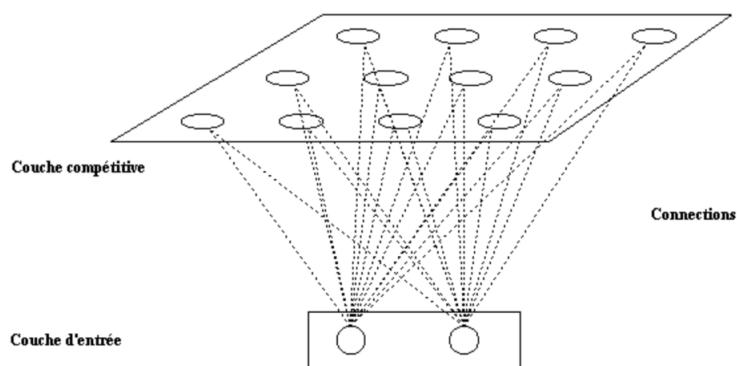


FIGURE 13 – Carte de Kohonen - Principe

4.2 Application : avec le package RCurl

Les cartes de Kohonen (SOM) sont un outil permettant de visualiser des motifs dans des données à haute dimension en produisant une représentation bidimensionnelle, qui affiche des motifs significatifs dans la structure à haute dimension. Les SOM sont "entraînées" avec les données de la manière suivante :

- La taille de la grille de la carte est définie.
- Chaque cellule de la grille se voit attribuer un vecteur d'initialisation dans l'espace de données.
- Les données sont introduites de manière répétée dans le modèle pour l'entraîner. Chaque fois qu'un vecteur d'entraînement est entré, le processus suivant est entrepris :
 - La cellule de grille avec le vecteur représentatif le plus proche du vecteur d'entraînement est identifiée.
 - Tous les vecteurs représentatifs des cellules de grille proches de celle identifiée sont légèrement ajustés vers le vecteur d'entraînement.
 - Plusieurs paramètres de convergence forcent les ajustements à devenir de plus en plus petits au fur et à mesure que les vecteurs d'entraînement sont alimentés à plusieurs reprises, ce qui entraîne la stabilisation de la carte en une représentation.

La caractéristique clé que cet algorithme est que les points qui étaient proches dans l'espace de données sont proches dans le SOM. Ainsi, les SOMs peuvent être un bon outil pour représenter des groupes spatiaux dans vos données.

Le package Kohonen permet de créer rapidement quelques cartes de Kohonen de base en R.

Les fonctions de Kohonen nécessitent l'utilisation de champs numériques ne comportant aucune entrée manquante. On vérifie donc que nos données sont bien complètes à l'aide de la fonction `summary()`.

Carte de Kohonen de base

On commence par affiché les cartes pour les variables qui correspondent à des moyennes, puis des écarts type et enfin les plus grandes valeurs :

On utilise les données qui ont été centrées-réduites, et qui sont numériques. La variable "Diagnosis" n'intervient pas car on est en apprentissage non supervisé.

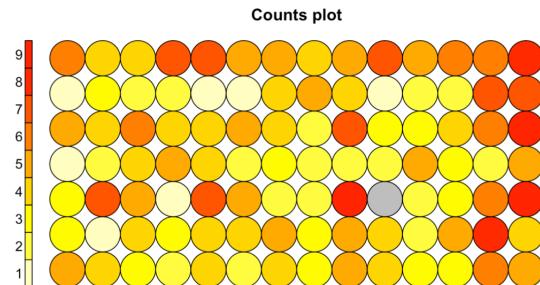
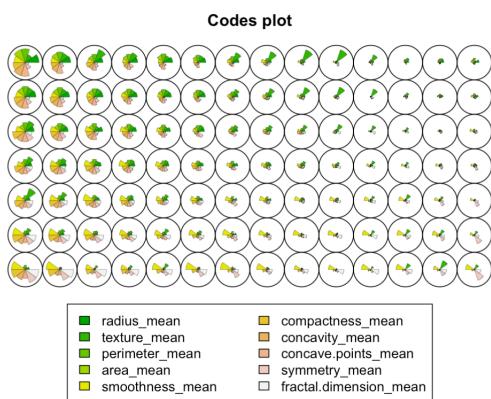
```
1 data.SOM1 <- som(data.num.train[,1:10], grid = somgrid(6, 4, "rectangular"))
2 plot(data.SOM1)
3
4 data.SOM2 <- som(data.num.train[,11:20], grid = somgrid(6, 4, "rectangular"))
5 plot(data.SOM2)
6
7 data.SOM3 <- som(data.num.train[,21:30], grid = somgrid(6, 4, "rectangular"))
8 plot(data.SOM3)
```

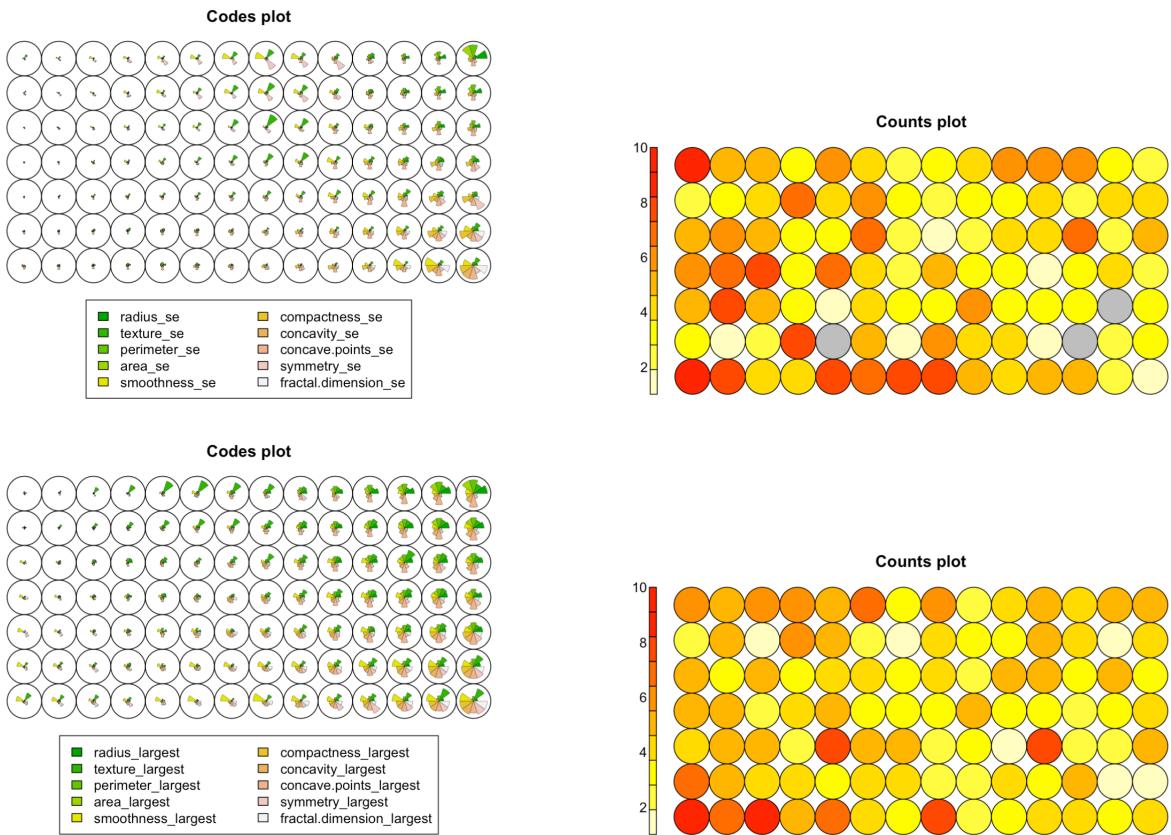
Carte thermique de Kohonen

On peut identifier les patients ayant des cellules sur la carte en assignant à chaque patient la cellule avec le vecteur représentatif le plus proche de la ligne de stat de ce joueur. C'est ce que fait la carte de Kohonen de type `count`, et elle crée une carte thermique basée sur le nombre de patient assignés à chaque cellule.

Dans les cartes ci dessous le rouge représente les cellules de la grille avec un nombre plus élevé de patients représentés.

```
1 # reverse color ramp
2 colors <- function(n, alpha = 1) {
3   rev(heat.colors(n, alpha))
4 }
5
6 plot(data.SOM1, type = "counts", palette.name = colors, heatkey = TRUE)
7 plot(data.SOM2, type = "counts", palette.name = colors, heatkey = TRUE)
8 plot(data.SOM3, type = "counts", palette.name = colors, heatkey = TRUE)
```





La taille et la disposition de la grille sont choisi arbitrairement. Le tracé de la carte standard de Kohonen crée ces représentations en camembert des vecteurs représentatifs des cellules de la grille, où le rayon d'un coin correspond à l'importance dans une dimension particulière. Certains schémas commencent à apparaître, on peut voir que deux groupes se forment selon les variables.

Dans notre cas on remarque qu'à certaines extrémités des cartes sont présents un nombre d'individus important.

Plus la distance moyenne entre les individus associés à un neurone et son poids est faible, plus la représentation des individus par leur neurone est de bonne qualité.

5 La Machine de Boltzmann Restreinte - RBM

5.1 Principe

Les machines Boltzmann restreintes (RBM) sont parmi les éléments de base les plus courants des réseaux profonds. Il s'agit d'un réseau neuronal stochastique génératif, destiné à apprendre une distribution de probabilité sur ses entrées.

Il contient une couche de variables observables et une seule couche de variables latentes. Les RBM peuvent être empilés (l'un sur l'autre) pour former des modèles plus profonds (cf Figure 14).

Comme leur nom l'indique, les machines de Boltzmann restreintes sont une variante des machines de Boltzmann, avec la restriction que leurs neurones ne doivent pas tous être inter-connectées.

Ils doivent former un graphe bipartite : une paire de neurones appartenant à chacune des couches ("visibles" et "cachées) (communément appelés respectivement unités "visibles" et "cachées") possède une connexion symétrique entre eux (la propagation est dirigée dans un sens ou dans l'autre avec le même poids) ; et il n'y a pas de connexion entre les noeuds d'un groupe.

En revanche, les machines Boltzmann "sans restriction" peuvent avoir des connexions entre les unités cachées. Cette restriction permet d'obtenir des algorithmes d'entraînement plus efficaces que ceux disponibles pour la classe générale des machines Boltzmann.

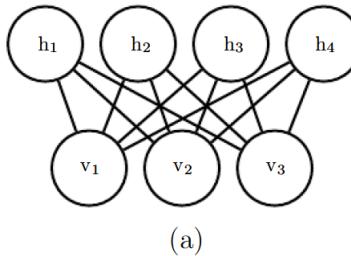


FIGURE 14 – Schéma réseau de machine de Boltzmann restreinte

Le réseau (a) représente un modèle de RBM, il s'agit d'un graphique bipartite, sans aucune connexion autorisée entre les neurones de la couche "visible" ou entre ceux de la couche "cachée".

Fonctionnement :

Nous commençons par la version binaire de la machine Boltzmann restreinte, mais comme nous le verrons plus tard, il existe des extensions à d'autres types d'unités visibles et cachées.

Plus formellement, la couche observée consiste en un ensemble de n_v variables aléatoires binaires auxquelles nous nous référons collectivement avec le vecteur v . Nous appelons h la couche cachée de n_h variables aléatoires binaires.

La machine de Boltzmann restreinte est un modèle basé sur l'énergie avec la distribution de probabilité commune spécifiée par sa fonction énergétique :

$$P(v = v, h = h) = \frac{1}{Z} \exp(-E(v, h)) \quad (10)$$

avec :

$$E(v, h) = -b^T v - c^T h - v^T W \quad (11)$$

Avec :

- W est la matrice des poids
- x est le vecteur d'état des neurones de la couche "visible", $x_i \in \{0, 1\}$
- h est le vecteur d'état des neurones de la couche "cachée"
- b et c sont respectivement les biais des neurones de la couche "visible" et de la couche "cachée".

et Z est la constante de normalisation connue sous le nom de fonction de partition :

$$Z = \sum_v \sum_h \exp(-E(v, h)) \quad (12)$$

La tâche consiste à trouver la matrice de pondération W qui explique le mieux les données visibles pour un nombre donné d'unités cachées.

Distribution conditionnelle

Bien que $P(v)$ soit insoluble, la structure graphique bipartite du RBM a la propriété très particulière que ses distributions conditionnelles $P(h|v)$ et $P(v|h)$ sont factorielles et relativement simples à calculer et à échantillonner.

5.2 Application : avec le package deepnet

L'apprentissage de la machine de Boltzmann est non supervisé. On utilise la fonction `rbm.train()` du package `deepnet` pour entraîner une machine de Boltzmann restreinte sur nos données. On supprime la variable à expliquer car on est en apprentissage non supervisé.

Puis on compare les classes obtenues à la variable à expliquer de l'échantillon.

```
1 #chargement
2 library(deepnet)
3 #apprentissage
4 set.seed(100)
5 pm.rbm <- rbm.train(x=data.num.train, hidden=c(1), numepochs=150)
6
7 #proba prediction
8 proba.pred.rbm <- rbm.up(pm.rbm, data.num.test)
9
10 #prédiction
11 pred.rbm <- ifelse(proba.pred.rbm>0.5, "B", "M")
12
13 #évaluation
14 evaluation.prediction(data.test.cr$Diagnosis, pred.rbm)
```

```
[1] "Matrice de confusion"
     ypred
yobs   B   M
      B 102   7
      M   5  57
[1] "Taux d'erreur = 0.0701754385964912"
[1] "Rappel = 0.919"
[1] "Precision = 0.891"
[1] "F1-Score = 0.905"
```

On a 12 individus mal classés, ce score est moins bon en comparaison des résultats du perceptron multi-couches.

On en déduit que nos données ne sont pas forcément adaptées pour ce type de réseau, cela paraît logique car ces données sont faites pour faire de la classification supervisée.

6 Les réseaux de croyances profondes - DBN

6.1 Principe

Les réseaux de croyances profondes sont des modèles générateurs comportant plusieurs couches de variables latentes. Les variables latentes sont généralement binaires, tandis que les unités visibles peuvent être binaires ou réelles. Il n'y a pas de connexions entre les couches. Habituellement, chaque unité de chaque couche est connectée à chaque unité de chaque couche voisine, bien qu'il soit possible de construire des DBN plus faiblement connectés. Les connexions entre les deux couches supérieures sont non dirigées. Les connexions entre toutes les autres couches sont dirigées, avec les flèches pointant vers la couche la plus proche des données.

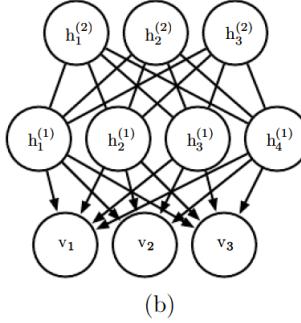


FIGURE 15 – Réseau à croyance profonde

Un DBN avec l couches cachées contient l matrices de poids : $W^{(1)}, \dots, W^{(l)}$. Il contient également $l + 1$ vecteurs de biais : $b^{(0)}, \dots, b^{(l)}$ avec $b^{(0)}$ fournissant les biais pour la couche visible. La distribution de probabilité représentée par le DBN est donnée par

$$P(h^{(l)}, h^{(l-1)}) \approx \exp(b^{(l)T}h^{(l)} + b^{(l-1)T}h^{(l-1)} + h^{(l-1)T}W^{(l)}h^{(l)}) \quad (13)$$

$$P(h_i^{(k)} = 1 | h^{(k+1)}) = \sigma(b_i^{(k)} + W_{:,i}^{(k+1)T}h^{(k+1)}), \quad \forall i \forall k \in 1, \dots, l-2 \quad (14)$$

$$P(v_i = 1 | h^{(1)}) = \sigma(b_i^{(0)} + W_{:,i}^{(1)T}h^{(1)}), \quad \forall i \quad (15)$$

Dans le cas d'unités visibles à valeur réelle, substituer :

$$v \sim \mathcal{N}(v; b^{(0)} + W^{(1)T}h^{(1)}, \beta^{-1}) \quad (16)$$

avec β en diagonale pour la tractabilité. Les généralisations aux autres unités visibles de la famille exponentielle sont simples, du moins en théorie.

Remarque : Un DBN avec une seule couche cachée est un RBM.

Pour générer un échantillon à partir d'un DBN, nous effectuons d'abord plusieurs étapes d'échantillonnage Gibbs sur les deux couches cachées supérieures. Cette étape consiste essentiellement à prélever un échantillon à partir du RBM défini par les deux couches cachées supérieures. Nous pouvons alors utiliser un seul passage de l'échantillonnage ancestral à travers le reste du modèle pour prélever un échantillon des unités visibles.

Pour former un réseau de croyances profondes, on commence par former un RBM pour maximiser $E_{v \sim p_{\text{data}}} \log p(v)$ en utilisant la divergence contrastive ou la probabilité maximale stochastique. Les paramètres de la RBM définissent ensuite les paramètres de la première couche du DBN. Ensuite, un second RBM est formé pour maximiser approximativement

$$E_{v \sim p_{\text{data}}} E_{h^{(1)} \sim p^{(1)}(h^{(1)}|v)} \log p^{(2)}(h^{(1)}) \quad (17)$$

où $p^{(1)}$ est la distribution de probabilité représentée par le premier RBM et $p^{(2)}$ est la distribution de probabilité représentée par le second RBM. En d'autres termes, le second RBM est formé pour modéliser la distribution définie par l'échantillonnage des unités cachées du premier RBM, lorsque le premier RBM est piloté par les données. Ce peut être répétée indéfiniment, pour ajouter autant de couches au DBN que souhaité, chaque nouveau RBM modélisant les échantillons du précédent. Chaque RBM définit une autre couche du DBN. Cette procédure peut être justifiée comme augmentant une limite inférieure variationnelle de la log-vraisemblance des données sous le DBN.

Le DBN formé peut être utilisé directement comme un modèle génératif, mais l'intérêt des DBN vient surtout de leur capacité à améliorer les modèles de classification. Nous pouvons prendre les poids du DBN et les utiliser pour définir un MLP :

$$h^{(1)} = \sigma(b^{(1)} + v^T W^{(1)}) \quad (18)$$

$$h^{(l)} = \sigma(b_i^{(l)} + h^{(l-1)T} W^{(l)}), \quad \forall l \in 2, \dots, m \quad (19)$$

Après avoir initialisé ce MLP avec les poids et les biais appris par l'entraînement génératif du DBN, nous pouvons entraîner le MLP à effectuer une tâche de classification. Cet entraînement supplémentaire du MLP est un exemple de réglage fin discriminatoire.

Le terme "réseau de croyances profondes" est couramment utilisé à tort pour désigner tout type de réseau neuronal profond, même les réseaux sans sémantique variable latente. Le terme "réseau de croyances profondes" devrait se référer spécifiquement aux modèles ayant des connexions non dirigées dans la couche la plus profonde et des connexions dirigées pointant vers le bas entre toutes les autres paires de couches consécutives.

6.2 Application : avec le package deepnet

On utilise encore une fois le package deepnet pour entraîner un réseau de croyance profonde sur nos données.

```
1 #chargement
2 library(deepnet)
3 #apprentissage
4 set.seed(100)
5 pm.dbn <- dbn.dnn.train(x=data.num.train,y=Malin.train,hidden=c(3,3),numepochs=150)
6
7 #proba prediction
8 proba.pred.dbn <- nn.predict(pm.dbn,data.num.test)
9 summary(proba.pred.dbn)
10
11 #prédiction
12 pred.dbn <- ifelse(proba.pred.dbn>0.43,"M","B")
13
14 #évaluation
15 evaluation.prediction(data.test.cr$Diagnosis,pred.dbn)
```

```
V1
Min.   :0.2383
1st Qu.:0.2383
Median :0.2383
Mean   :0.4184
3rd Qu.:0.7195
Max.   :0.7195
[1] "Matrice de confusion"
     ypred
yobs   B    M
      B 102    7
      M   5  57
[1] "Taux d'erreur = 0.0701754385964912"
[1] "Rappel = 0.919"
[1] "Precision = 0.891"
[1] "F1-Score = 0.905"
```

On a encore 12 individus mal classés, ce score est moins bon en comparaison des résultats du perceptron multicouches.

Cela paraît logique de retrouver le même nombre d'individus mal classés que pour le RBM car comme expliqué ci-dessus, le DBN utilise des RBM pour former chacune de ces couches.

6.3 Conclusion

Les machines de Boltzman Profondes ont été développées après les DBN. Par rapport aux DBN, la distribution postérieure $P(h|v)$ est plus simple pour les DBM. De manière quelque peu contre-intuitive, la simplicité de cette distribution postérieure permet des approximations plus riches de la partie postérieure.

On présente rapidement les machines de Boltzmann profondes dans la dernière partie car ce sont des modèles qui sont de plus en plus utilisés à la place des DBN.

7 La machine de Boltzmann Profonde - DBM

7.1 Principe

Une machine Boltzmann profonde (DBM) est un type de modèle profond et génératif. Il s'agit d'un modèle entièrement non dirigé. Contrairement à la machine de Boltzmann restreinte, le DBM comporte plusieurs couches de variables latentes (les RBM n'en ont qu'une). Mais comme les RBM, à l'intérieur de chaque couche, chacune des variables est mutuellement indépendante, conditionnée par les variables dans les couches voisines. Les machines de Boltzmann profondes ont été appliquées à une variété de tâches, y compris la modélisation de documents.

Les DBM ne contiennent généralement que des unités binaires - comme c'est supposé précédemment la simplicité de présentation du modèle - mais il est simple d'inclure des unités visibles à valeurs réelles.

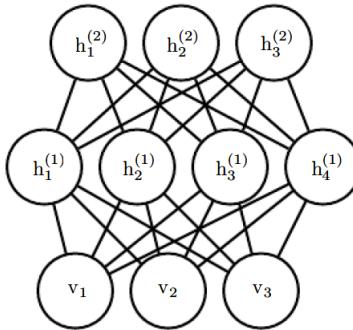


FIGURE 16 – Machine de Boltzmann profonde

Une machine de Boltzmann profonde est un modèle basé sur l'énergie, ce qui signifie que la distribution de probabilité conjointe des variables du modèle est paramétrée par une fonction énergétique E . Dans le cas d'une machine Boltzmann profonde avec une couche visible, v , et trois couches cachées, $h^{(1)}$, $h^{(2)}$ et $h^{(3)}$, la probabilité commune est donnée par :

$$P(v, h^{(1)}, h^{(2)}, h^{(3)}) = \frac{1}{Z(\theta)} \exp(-E(v, h^{(1)}, h^{(2)}, h^{(3)}; \theta)) \quad (20)$$

Pour simplifier la présentation, on omet les paramètres de biais ci-dessous. La fonction énergétique de la DBM est alors définie comme suit :

$$E(v, h^{(1)}, h^{(2)}, h^{(3)}; \theta) = -v^T W^{(1)} h^{(1)} - h^{(1)T} W^{(2)} h^{(2)} - h^{(2)T} W^{(3)} h^{(3)} \quad (21)$$

Par rapport à la fonction énergétique de la RBM (équation 11), la fonction énergétique de la DBM comprend des connexions entre les unités cachées (variables latentes) sous la forme de matrices de poids ($W^{(2)}$ et $W^{(3)}$). Comme nous le verrons, ces connexions ont des conséquences importantes tant sur le comportement du modèle que sur la manière dont nous procédons à l'inférence dans le modèle.

La DBM offre certains avantages qui sont similaires à ceux offerts par la RBM. Plus précisément, comme l'illustre la figure 16, les couches de la DBM peuvent être organisées en un graphique bipartite, avec des couches impaires d'un côté et des couches paires de l'autre. Cela implique immédiatement que lorsque nous conditionnons les variables de la couche paire, les variables des couches impaires deviennent conditionnellement indépendantes. Bien sûr, lorsque nous nous basons sur les variables des couches impaires, les variables des couches paires deviennent également indépendantes.

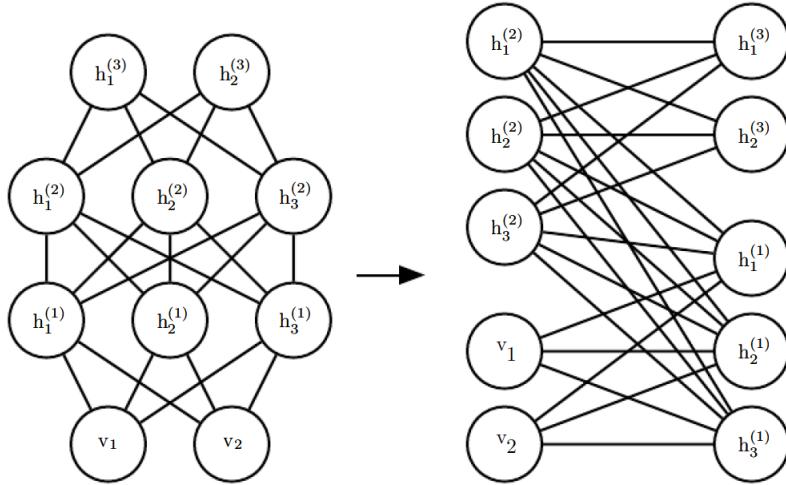


FIGURE 17 – Machine de Boltzmann profonde réarrangée en graphe bipartite

La structure bipartite de la DBM permet d'appliquer les mêmes équations que celles utilisées pour les distributions conditionnelles d'un RBM pour déterminer les distributions conditionnelles dans une DBM. Les unités d'une couche sont conditionnellement indépendants les uns des autres compte tenu des valeurs du voisin de sorte que les distributions sur les variables binaires puissent être entièrement décrites par les paramètres d'une Bernoulli donnant la probabilité que chaque unité soit active. Dans notre exemple avec deux couches cachées, les probabilités d'activation sont données par :

$$P(v_i = 1|h^{(1)}) = \sigma(W_{i,:}^{(1)}h^{(1)}) \quad (22)$$

$$P(h_i^{(1)} = 1|v, h^{(2)}) = \sigma(v^T W_{:,i}^{(1)} + W_{:,i}^{(2)}h^{(2)}) \quad (23)$$

$$P(h_k^{(2)} = 1|h^{(1)}) = \sigma(h^{(1)T} W_{:,k}^{(2)}) \quad (24)$$

Remarque : Les applications logicielles des DBM sont encore rares sur le web, sûrement car ce sont des réseaux récents. Cependant cela peut être réglé à l'aide de la librairie Tensor Flow sur Python. (Ce ne sera pas fait dans ce projet)

8 Conclusion

Au cours de ce projet, plusieurs réseaux de neurones ont été présentés. Ils diffèrent notamment par leur structure et leur utilité.

On remarque que pour le jeu de données choisi, la meilleure classification est obtenue avec le PMC. Alors que d'autres réseaux plus complet et complexe sont testés. Cela vient peut être du fait que ce jeu de données est de petite taille, car plus les réseaux sont complexes plus ils demandent un nombre de données importantes pour être efficace.

On a également pu observer que le résultat du PMC était meilleur que celui des arbres de décision ce qui indique que les réseaux neurones arrivent à égaler, voire surpasser les modèles mathématiques plus classiques.

Précisons quand même que l'ensemble des résultats trouvés étaient satisfaisants, avec moins de 8% d'erreurs, et donc que les écarts entre les différents réseaux sont plutôt minimes.

On aurait également pu présenter d'autres réseaux très connus, tels que les réseaux qui utilisent l'apprentissage par renforcement, les réseaux antagonistes génératifs ou encore les réseaux de neurones convolutionnels.

Références

- [AR89] ANDERSON et ROSEFIELD. "Neurocomputing, Foundations of research". In : *MIT Press Cambridge* (1989).
- [Jod94] JODOUIN. *Les Réseaux de neurones*. Hermès, 1994.
- [MRE96] MOZER, RUMELHARD et ERLBAUM. *Mathematical perspectives on neural network*. P. Smolensky Associates, 1996.
- [Tan17] Dan TANNER. "Introduction to Self Organizing Maps in R - the Kohonen package and NBA player statistics". In : *github* (2017).
- [GBC18] GOODFELLOW, BENGIO et COURVILLE. *L'apprentissage Profond*. Massot Edition, 2018.
- [Le18] James LE. "The 10 Neural Network Architectures Machine Learning Researchers Need To Learn". In : *Medium* (2018).
- [RT18] RAKOTOMALALA et TANAGRA. "Cours et TP de Machine Learning, Université Lyon 2". In : (2018).
- [TCM18] TAHERKHANI, COSMA et McGINNITY. "Deep-FS : A feature selection algorithm for Deep Boltzmann Machines". In : *Elsevier* (2018).
- [Cha19] Nagesh Singh CHAUHAN. "Introduction to Artificial Neural Networks(ANN)". In : *Towards Data Science* (2019).
- [Val19] Annick VALIBOUZE. "Réseau de Neurones Artificiels". In : (2019).
- [Inta] INTERNET. "Image Neurone". In : (). URL : <https://fr.wikipedia.org/wiki/Neurone#/media/Fichier:Neuron-figure-fr.svg>.
- [Intb] INTERNET. "Image Réseau de Neurones 2". In : (). URL : <https://fr.audiofanzine.com/techniques-du-son/forums/t.662434,ces-appareils-audio-etonnants,p.11.html>.
- [Intc] INTERNET. "Images Réseau de Neurones". In : (). URL : <http://www.becoz.org/these/memoirehtml/ch06s04.html>.
- [Pru] Lallich PRUDHOMME. "Validation statistique des cartes de Kohonen en apprentissage supervisé". In : *Laboratoire E.R.I.C, Univ. Lyon 2* ().