Deep Learning

Perceptrons simples et multicouches

Ricco Rakotomalala

Université Lumière Lyon 2

Plan

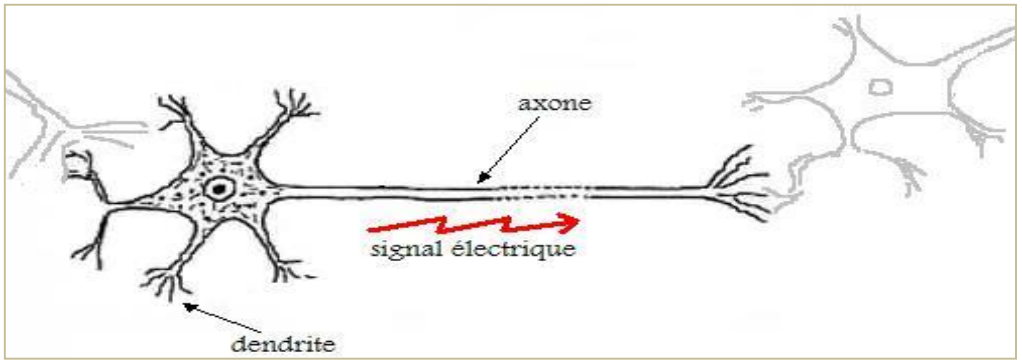
1. Perceptron simple
2. Plus loin avec le perceptron simple
3. Perceptron multicouche
4. Plus loin avec le perceptron multicouche
5. Pratique des perceptrons (sous R et Python)
6. Références
7. Conclusion

Métaphore biologique et transposition mathématique

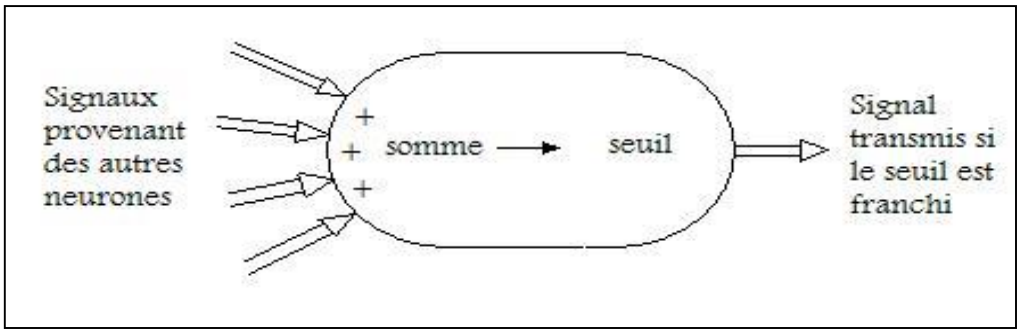
PERCEPTRON SIMPLE

Métaphore biologique

Fonctionnement du cerveau
Transmission de l'information
et apprentissage



Idées maîtresses à retenir



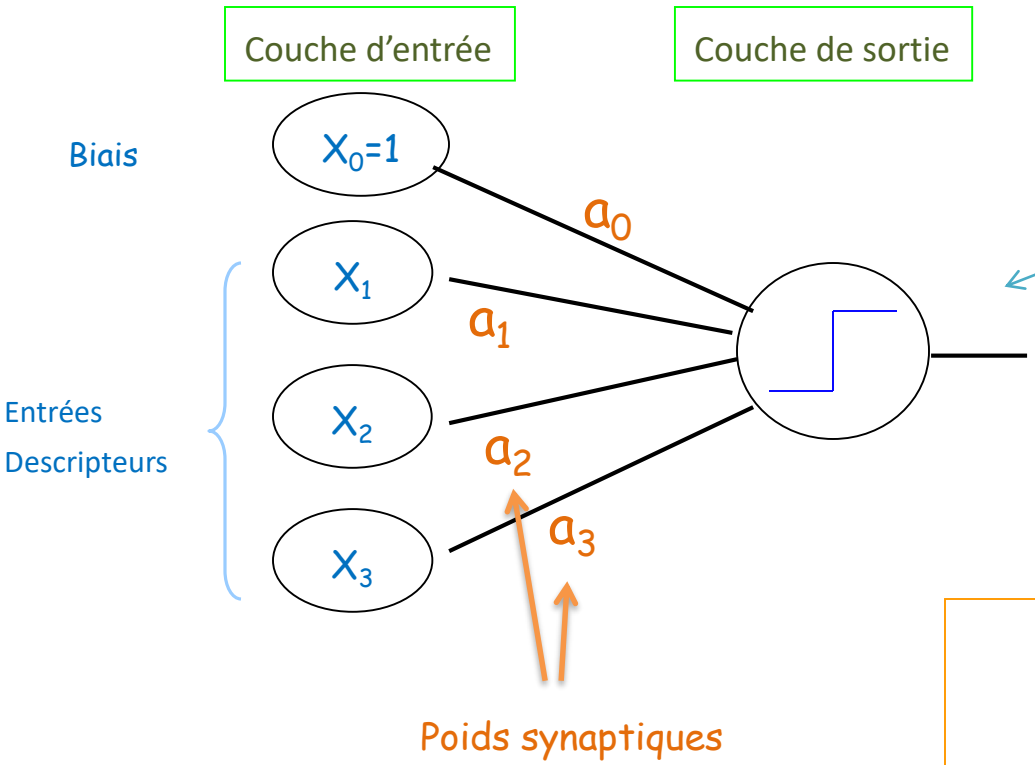
Etapes clés :

- Réception d'une information (signal)
- Activation + Traitement (simple) par un neurone
- Transmission aux autres neurones (si seuil franchi)
- A la longue : renforcement de certains liens → APPRENTISSAGE

Modèle de Mc Colluch et Pitts – Le Perceptron Simple

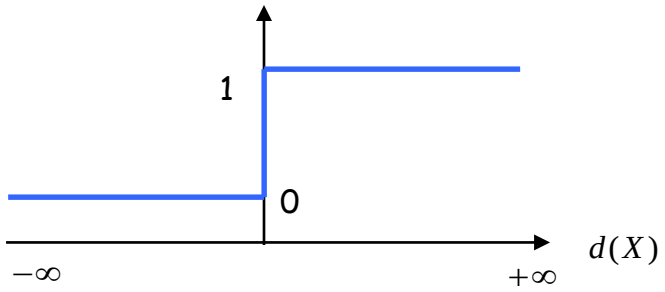
Problème à deux classes (positif et négatif)

$Y \in \{1(+), 0(-)\}$



Fonction de transfert

Fonction à seuil -- Fonction de Heaviside



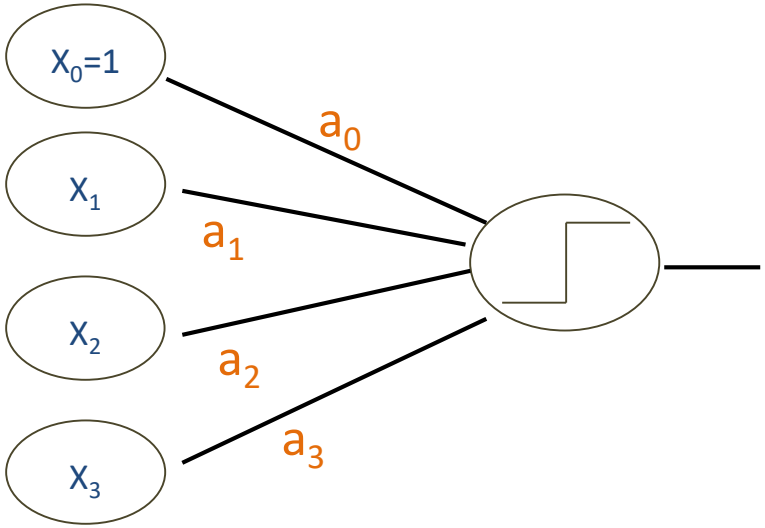
Modèle de prédiction et règle d'affectation

$$d(X) = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3$$

Si $d(X) > 0$ Alors $Y = 1$ Sinon $Y = 0$

Le perceptron simple est un modèle de prédiction (supervisé) linéaire !

Apprentissage à partir de données



Comment calculer les poids synaptiques à partir d'un fichier de données (Y ; X1, X2, X3)

Faire le parallèle avec la régression et les moindres carrés

Un réseau de neurones peut être utilisé pour la régression (fonction de transfert avec une sortie linéaire)

(1) Quel critère optimiser ?

(2) Comment procéder à l'optimisation ?

(1) Minimiser l'erreur de prédiction

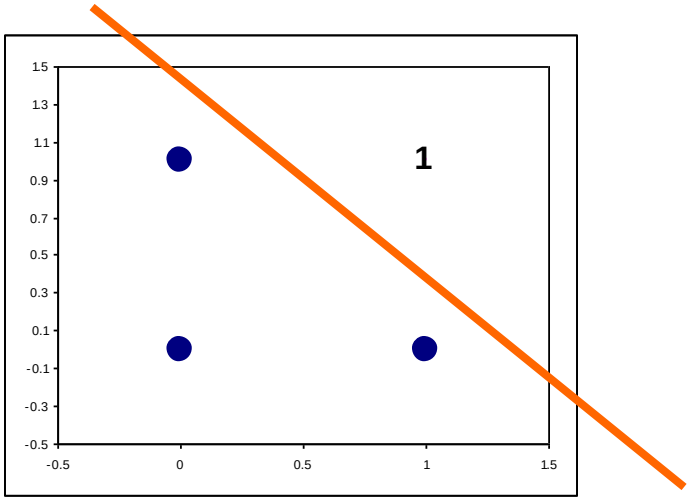
(2) Principe de l'incrémentalité (online)

Exemple – Apprentissage de la fonction AND (ET logique)

Cet exemple est révélateur – Les premières applications proviennent de l'informatique

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

Données



Représentation dans le plan

Principales étapes :

1. Mélanger aléatoirement les observations
2. Initialiser aléatoirement les poids synaptiques
3. Faire passer les observations unes à unes
   - Calculer l'erreur de prédiction pour l'observation
   - Mettre à jour les poids synaptiques
4. Jusqu'à convergence du processus

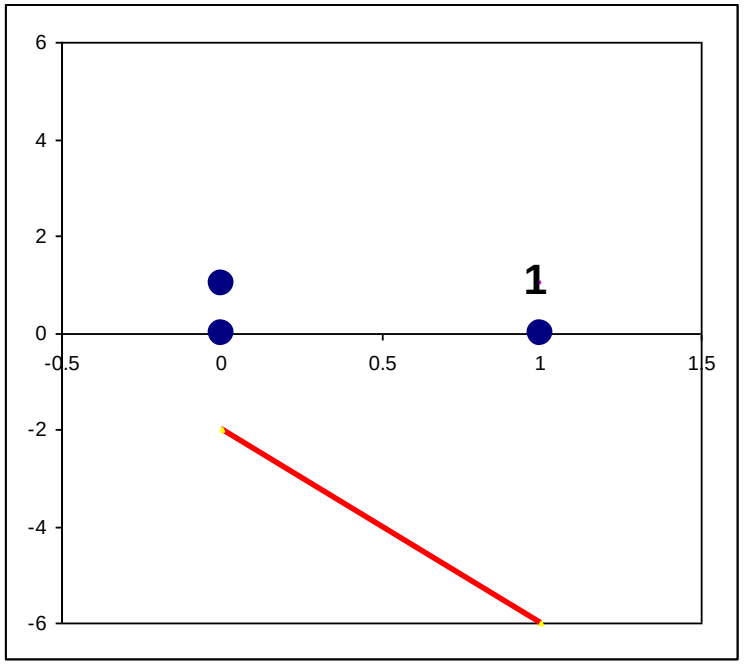
Une observation peut passer plusieurs fois !

Exemple AND (1)

Initialisation aléatoire des poids : $a_0 = 0.1; a_1 = 0.2; a_2 = 0.05$

Frontière :

$0.1 + 0.2x_1 + 0.05x_2 = 0 \Leftrightarrow x2 = -4.0x_1 - 2.0$



Règle de mise à jour des poids

Pour chaque individu que l'on fait passer

(Principe de l'incrémentalité)

$$a_j \leftarrow a_j + \Delta a_j$$

S'assurer que les variables sont sur la même échelle

avec

Force du signal

$$\Delta a_j = \eta\ (y - \hat{y})\ x_j$$

Taux d'apprentissage

Détermine l'amplitude de l'apprentissage

Quelle est la bonne valeur ?

Trop petit → lenteur de convergence

Trop grand → oscillation

En général autour de 0.05 ~ 0.15 (0.1 dans notre exemple)

Erreur

Détermine s'il faut réagir (corriger) ou non

Ces 3 éléments sont au cœur du mécanisme d'apprentissage

Exemple AND (2)

Observation à traiter

$$\begin{cases} x_0 = 1 \\ x_1 = 0 \\ x_2 = 0 \\ y = 0 \end{cases}$$

Appliquer le modèle

$$0.1 \times 1 + 0.2 \times 0 + 0.05 \times 0 = 0.1$$
$$\Rightarrow \hat{y} = 1$$

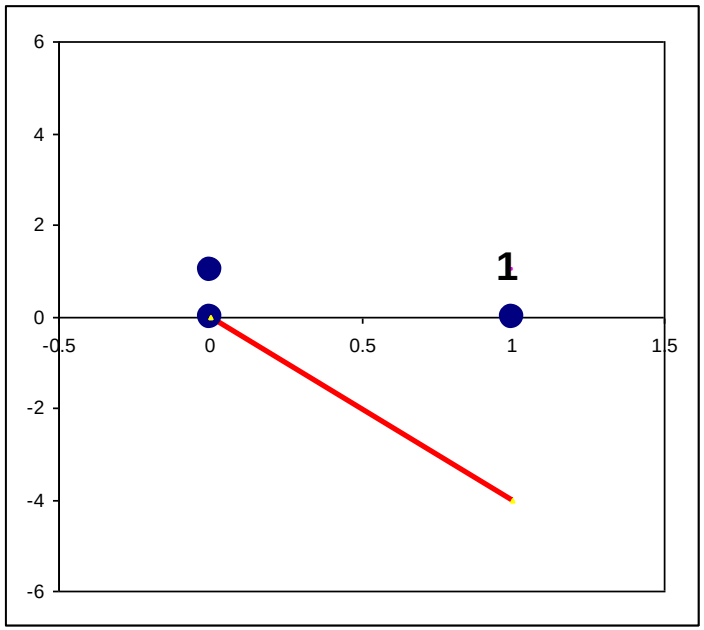
Màj des poids

$$\begin{cases} \Delta a_0 = 0.1 \times (-1) \times 1 = -0.1 \\ \Delta a_1 = 0.1 \times (-1) \times 0 = 0 \\ \Delta a_2 = 0.1 \times (-1) \times 0 = 0 \end{cases}$$

Valeur observée de Y et prédiction ne matchent pas, une correction des coefficients sera effectuée.

Nouvelle frontière :

$$0.0 + 0.2x_1 + 0.05x_2 = 0 \Leftrightarrow x_2 = -4.0x_1 + 0.0$$



Signal nul ($x_1 = 0$, $x_2 = 0$), seule la constante a_0 est corrigée.

Exemple AND (3)

Observation à traiter

$$\begin{cases} x_0 = 1 \\ x_1 = 1 \\ x_2 = 0 \\ y = 0 \end{cases}$$

Appliquer le modèle

$$0.0 \times 1 + 0.2 \times 1 + 0.05 \times 0 = 0.2$$

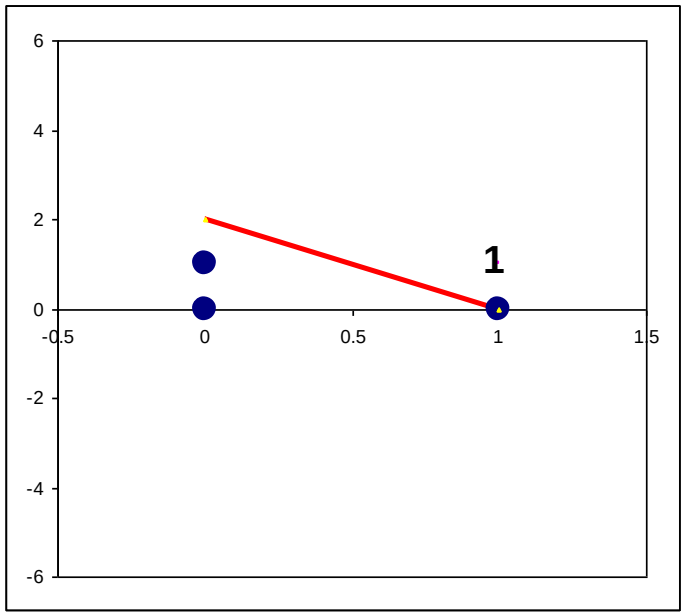
$$\Rightarrow \hat{y} = 1$$

Màj des poids

$$\begin{cases} \Delta a_0 = 0.1 \times (-1) \times 1 = -0.1 \\ \Delta a_1 = 0.1 \times (-1) \times 1 = -0.1 \\ \Delta a_2 = 0.1 \times (-1) \times 0 = 0 \end{cases}$$

Nouvelle frontière :

$$-0.1 + 0.1x_1 + 0.05x_2 = 0 \Leftrightarrow x_2 = -2.0x_1 + 2.0$$



Exemple AND (4) – Définir la convergence

Observation à traiter

$$\begin{cases} x_0 = 1 \\ x_1 = 0 \\ x_2 = 1 \\ y = 0 \end{cases}$$

Appliquer le modèle

$$-0.1\times 1 + 0.1\times 0 + 0.05\times 1 = -0.05$$

$$\Rightarrow \hat{y} = 0$$

Màj des poids

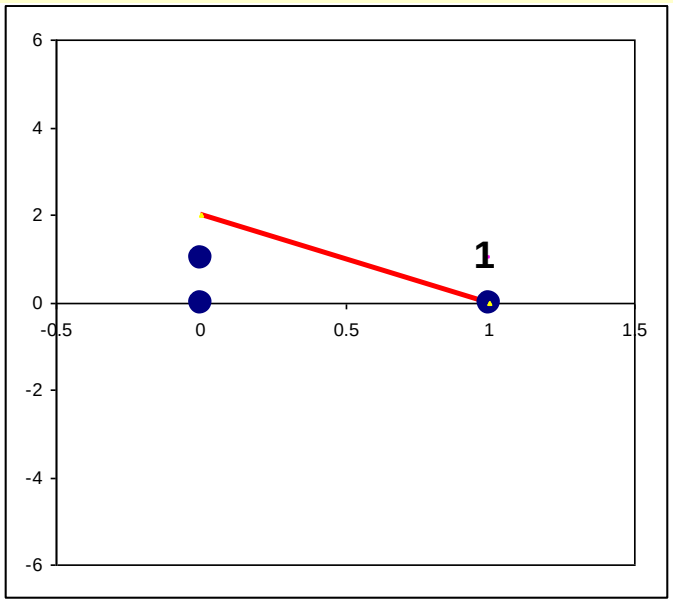
$$\begin{cases} \Delta a_0 = 0.1\times(0)\times 1 = 0 \\ \Delta a_1 = 0.1\times(0)\times 0 = 0 \\ \Delta a_2 = 0.1\times(0)\times 1 = 0 \end{cases}$$

Pas de correction ici ? Pourquoi ?

Voir aussi la position du point par rapport à la frontière dans le plan !

Frontière inchangée :

$$-0.1 + 0.1x_1 + 0.05x_2 = 0 \Leftrightarrow x_2 = -2.0x_1 + 2.0$$



Remarque : Que se passe-t-il si on repasse l'individu (x_1=1 ; x_2=0) ?

Convergence ?

(1) Plus aucune correction effectuée en passant tout le monde

(2) L'erreur globale ne diminue plus « significativement »

(3) Les poids sont stables

(4) On fixe un nombre maximum d'itérations

(5) On fixe une erreur minimale à atteindre

(2), (4) et (5) deviennent des « paramètres » de l'algorithme à considérer (*attention aux valeurs par défaut*) dans les logiciels !!! Il y en aura d'autres...

Estimation de P(Y/X), descente du gradient, problèmes multi-classes

PLUS LOIN AVEC LE PERCEPTRON SIMPLE

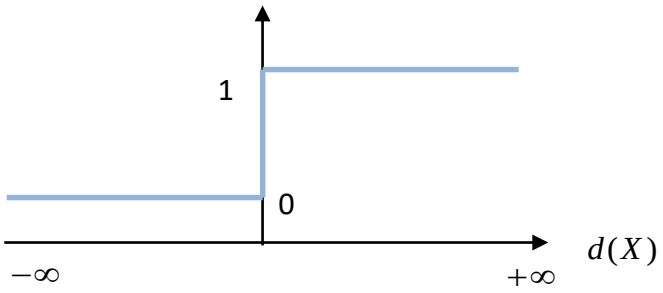
Evaluation de P(Y/X) – Fonction de transfert sigmoïde

Le Perceptron propose un classement Y/X

Dans certains cas, nous avons besoin de la probabilité P(Y/X) ex. Scoring

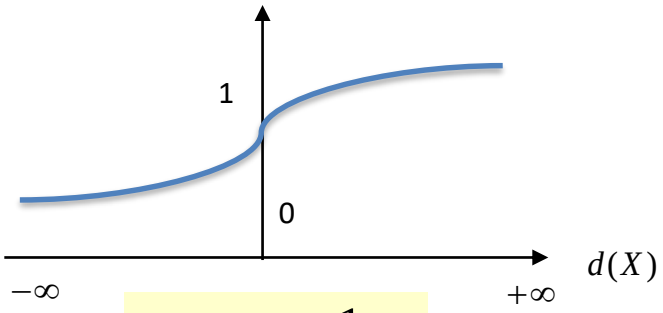
Fonction de transfert

Fonction à seuil -- Fonction de Heaviside



Fonction de transfert (fonction d'activation)

Fonction sigmoïde – Fonction logistique



$$g(v) = \frac{1}{1+e^{-v}}$$

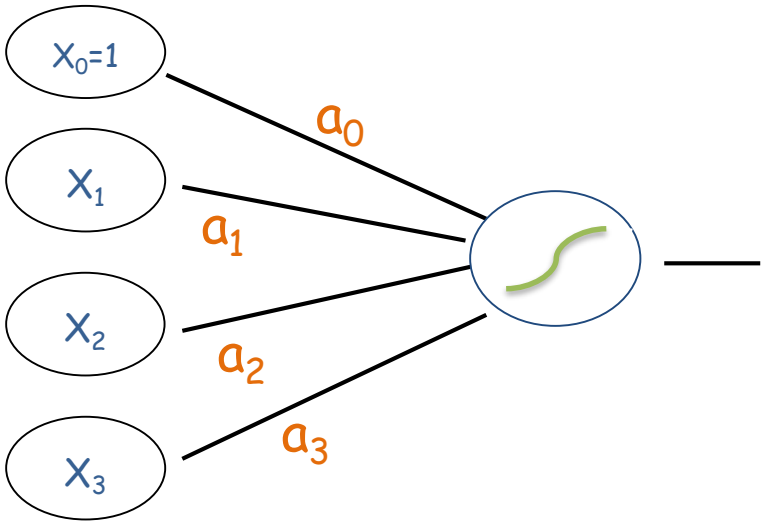
$$v = d(X)$$

Qui constitue à la fois une forme de lissage, mais aussi une modification du domaine de définition. Cf. la régression logistique.

g(v) est une estimation de P(Y/X), la règle de décision devient : Si g(v) > 0.5 Alors Y=1 Sinon Y=0

Modification du critère à optimiser

Conséquence d'une fonction de transfert continue et dérivable



Sortie du réseau

$$\hat{y} = g(v) = f[d(x)]$$

Critère à optimiser : critère des moindres carrés

$$E = \frac{1}{2}\sum_{\omega \in \Omega}\left(y(\omega) - \hat{y}(\omega)\right)^2$$

Mais toujours fidèle au principe d'incrémentalité. L'optimisation est basée sur la descente du gradient (gradient stochastique)!

Descente du gradient

Fonction de transfert sigmoïde dérivable

$$g'(v) = g(v)(1 - g(v))$$

Optimisation : dérivation de la fonction objectif par rapport aux coefficients

$$\frac{\partial E}{\partial a_j} = -\sum_i [y(\omega) - \hat{y}(\omega)] \times g'[v(\omega)] \times x_j(\omega)$$

Règle de mise à jour des coefficients pour un individu (Règle de Widrow-Hoff ou Règle Delta)

$$a_j \leftarrow a_j - \eta \underbrace{(y - \hat{y})\, g'(v)\, x_j}$$

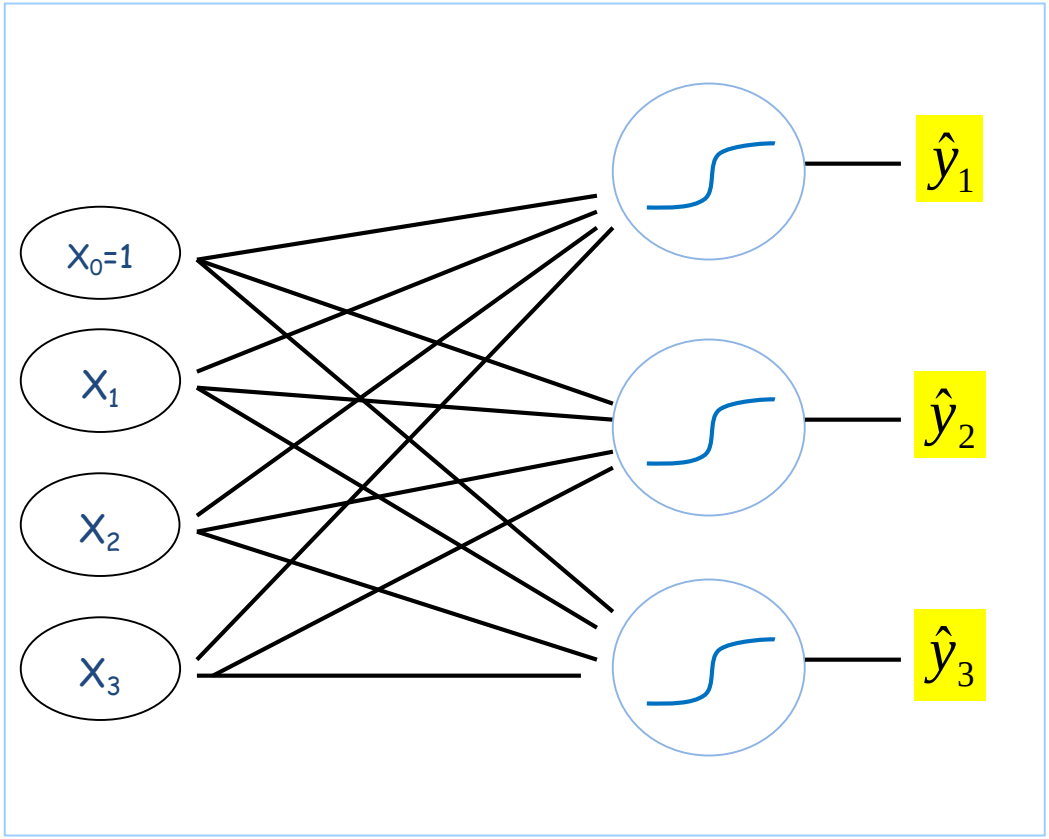
Gradient : màj des poids dans la direction qui minimise E

+

- La convergence vers le minimum est bonne dans la pratique
- Capacité à traiter des descripteurs corrélés (pas d'inversion de matrice)
- Capacité à traiter des problèmes à très grande dimension (lignes x colonnes)
- Mise à jour facile si ajout de nouveaux individus dans la base

Problèmes à (K > 2) classes (multi-classes)

Que faire lorsque K (modalités de Y) est supérieur à 2 ?



(1) Codage disjonctif complet de la sortie

(« *one hot encoding* »)

$$y_k = 1 \;\; ssi \;\; y = y_k$$

(2) « Output » pour chaque sortie

$$\hat{y}_k = g[v_k]$$

avec $v_k = a_{0,k} + a_{1,k}x_1 + \cdots + a_{J,k}x_J$

(3) P(Y/X) $P(Y = y_k / X) \propto g[v_k]$

(4) Règle de décision

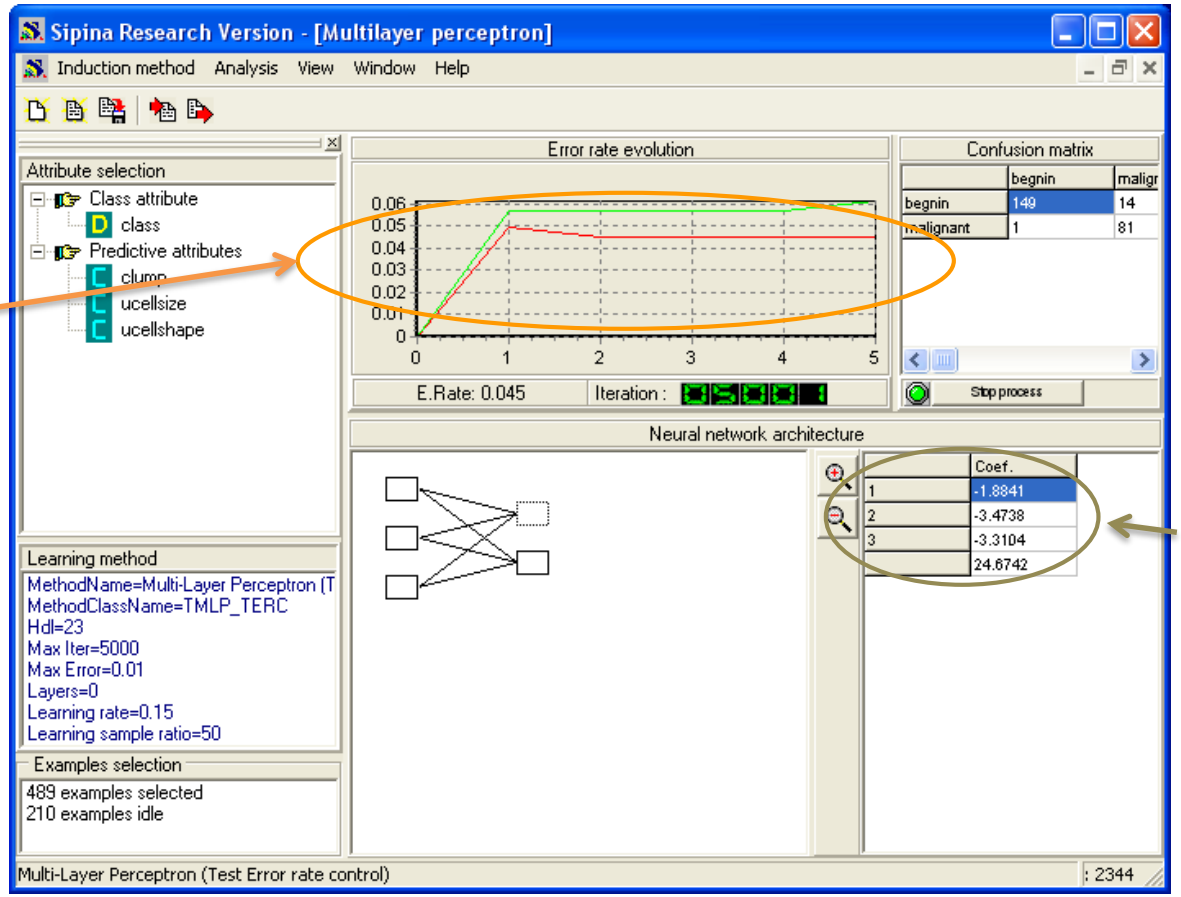
$$\hat{y} = y_{k^*} \;\; ssi \;\; k^* = \arg\max_k \hat{y}_k$$

Minimisation de l'erreur quadratique toujours, mais étendue aux K sorties

$$E = \frac{1}{2}\sum_{\omega}\sum_{k=1}^{K}\left(y_k(\omega) - \hat{y}_k(\omega)\right)^2$$

Un exemple sous SIPINA – ''Breast cancer'' dataset

Évolution de l'erreur



Poids synaptiques (y compris le biais)

Quelques conseils

Ramener les descripteurs sur la même échelle
Standardisation, Normalisation, etc.

(Éventuellement) Subdiviser les données en 3 parties :
Training + Validation + Test

Attention au paramétrage, notamment de la règle d'arrêt

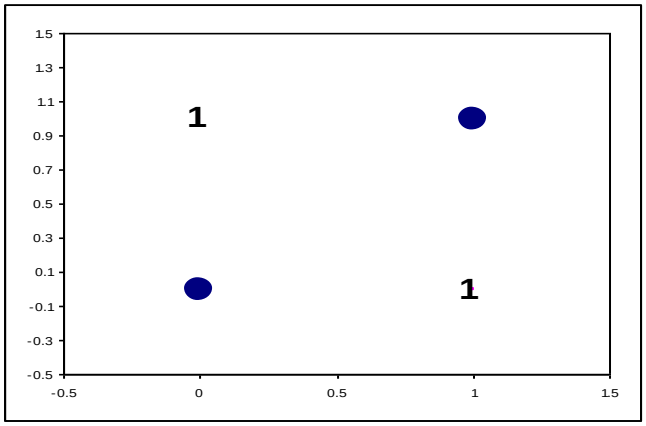
Mort et résurrection du perceptron

PERCEPTRON MULTICOUCHE

Problème du XOR – L'impossible séparation linéaire

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

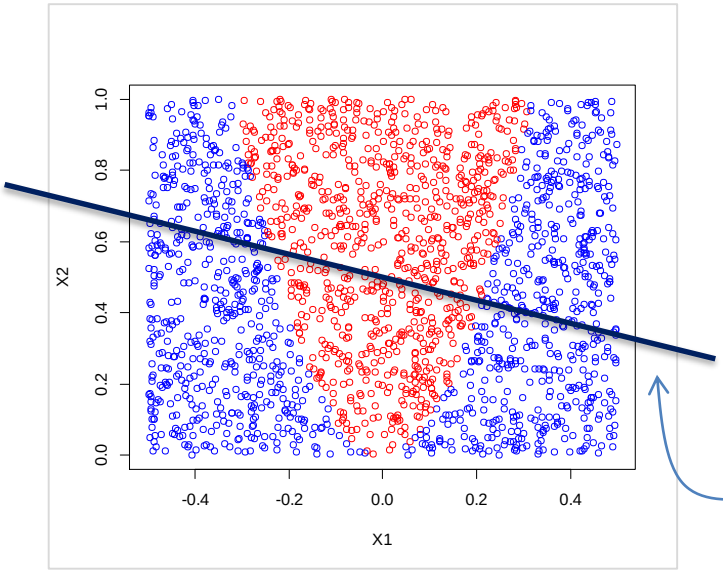
Données



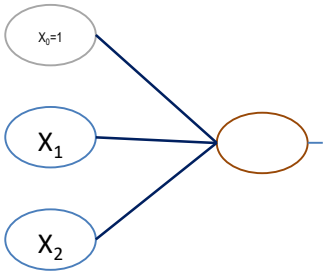
Non séparable linéairement

(Minsky & Papert, 1969)

Un perceptron simple ne sait traiter que les problèmes linéairement séparables.

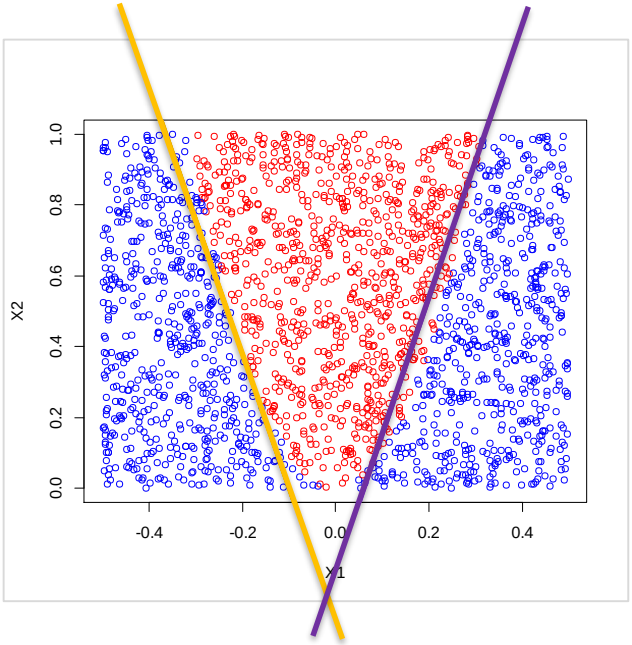


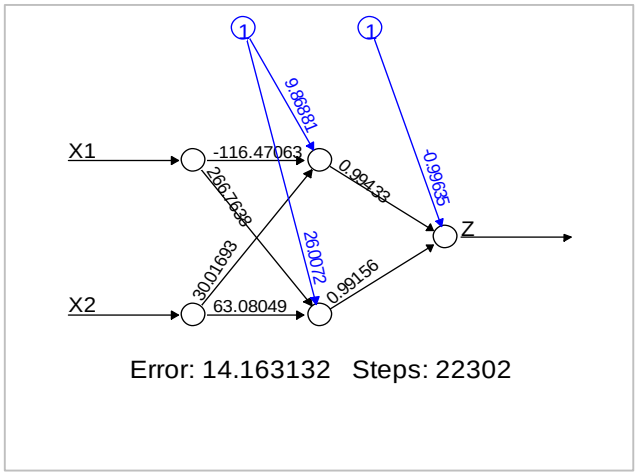
Trouver une droite de séparation efficace n'est pas possible.



Perceptron multicouche - Principe

$26.0 + 266.76 \times X1 + 63.08 \times X2 = 0$ $\quad 9.86 - 116.47 \times X1 + 30.01 \times X2 = 0$

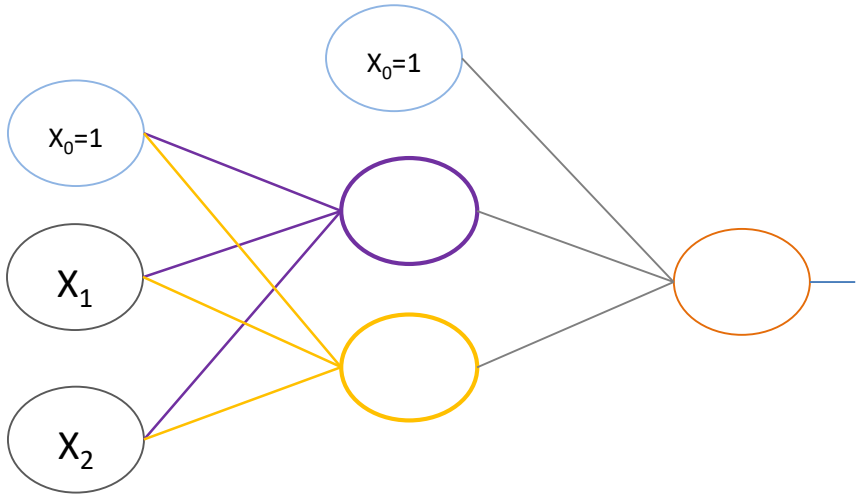




Perceptron Multicouche (PMC)

Une combinaison de séparateurs linéaires permet de produire un séparateur global non-linéaire (Rumelhart, 1986).

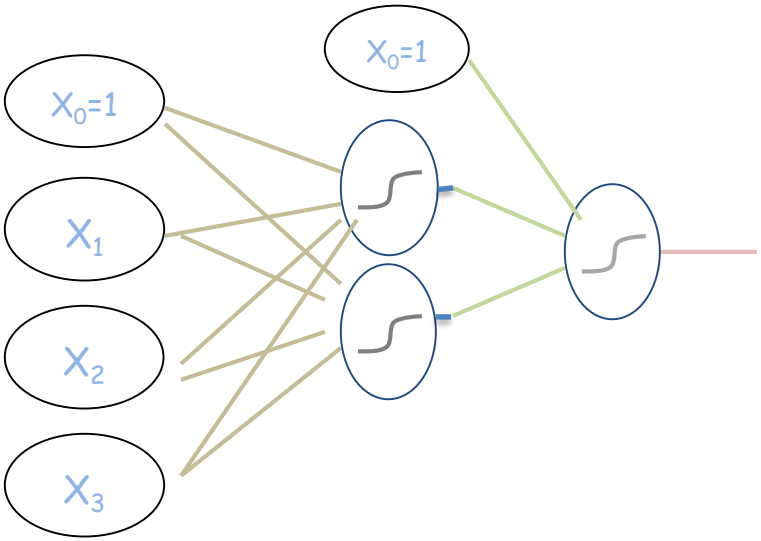
On peut avoir plusieurs couches cachées, cf. plus loin



Couche d'entrée | Couche cachée | Couche de sortie

Perceptron multicouche – Formules et propriétés

Fonction de transfert sigmoïde dans les couches cachées et de sortie (il peut en être autrement, cf. plus loin)



Passage C.Entrée → C.Cachée

$$v_1 = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3$$
$$v_2 = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3$$

Sortie de la C.Cachée

$$u_1 = g(v_1) = \frac{1}{1+e^{-v_1}}$$
$$u_2 = g(v_2) = \frac{1}{1+e^{-v_2}}$$

Passage C.Cachée → C.Sortie

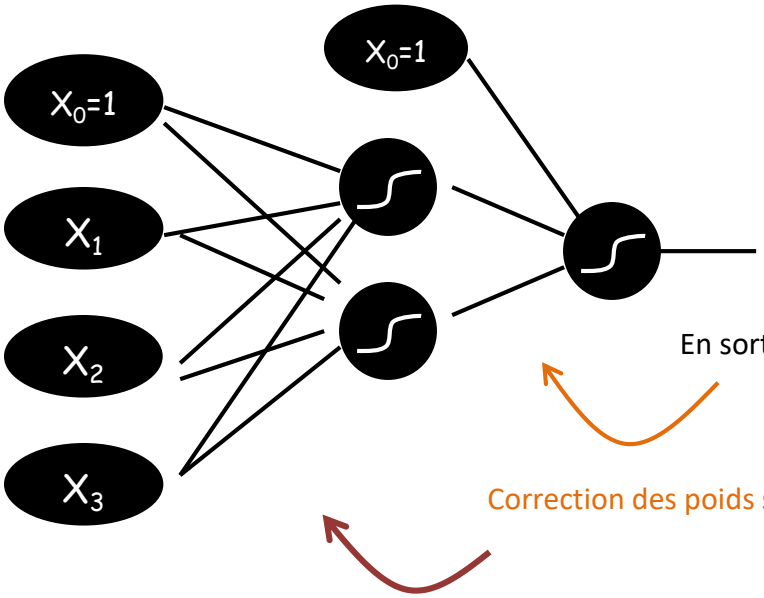
$$z = c_0 + c_1 u_1 + c_2 u_2$$

Sortie du réseau

$$\hat{y} = g(z) = \frac{1}{1+e^{-z}}$$

Propriété fondamentale : Le PMC est capable d'approximer toute fonction continue pourvu que l'on fixe convenablement le nombre de neurones dans la couche cachée.

Généraliser la règle de Widrow-Hoff – Rétropropagation



En sortie : Erreur mesurée (pour un individu)

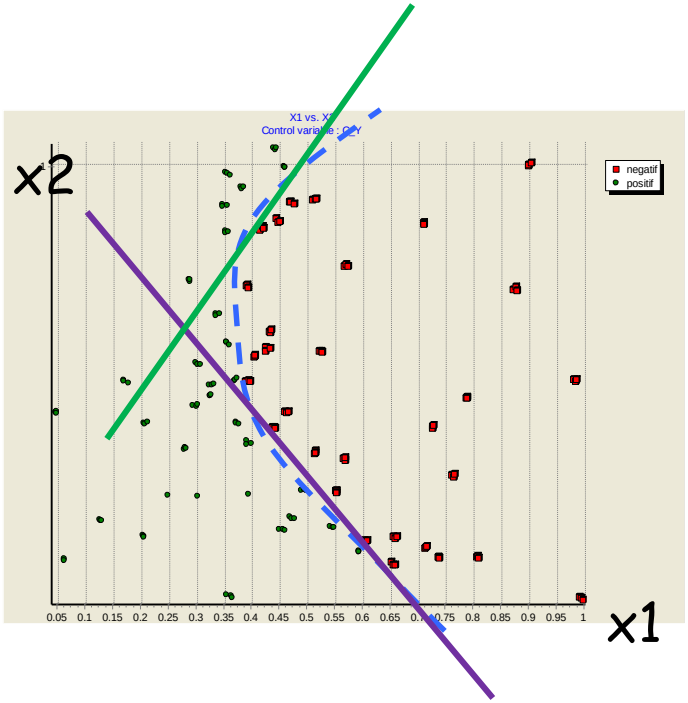
Correction des poids synaptiques menant à la couche de sortie

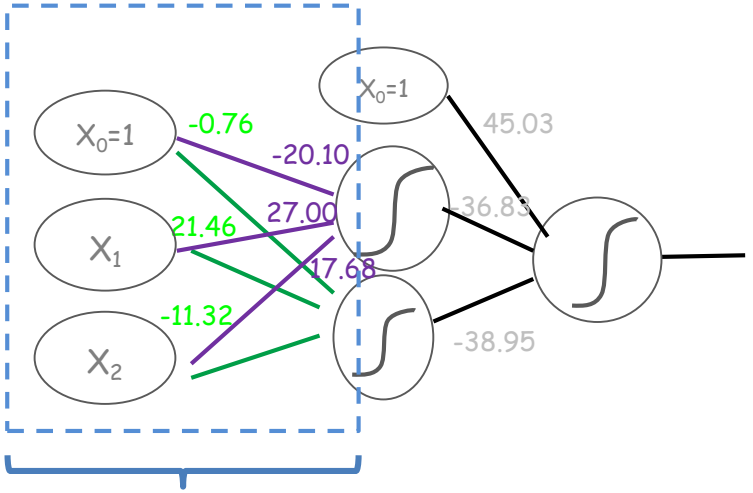
Propagation (en arrière) des corrections dans les couches intermédiaires

L'algorithme de la rétropropagation du gradient donne de bons résultats dans la pratique même si le risque de stagnation dans un optimum local n'est pas à négliger → normaliser ou standardiser impérativement les données et bien choisir le taux d'apprentissage

Perceptron multicouche – Espace initial de représentation

1ère vision du PMC (couche d'entrée vers couche cachée): nous disposons d'une série de droites frontières définies dans l'espace initial de représentation, imbriquées de manière à produire une séparation non-linéaire.



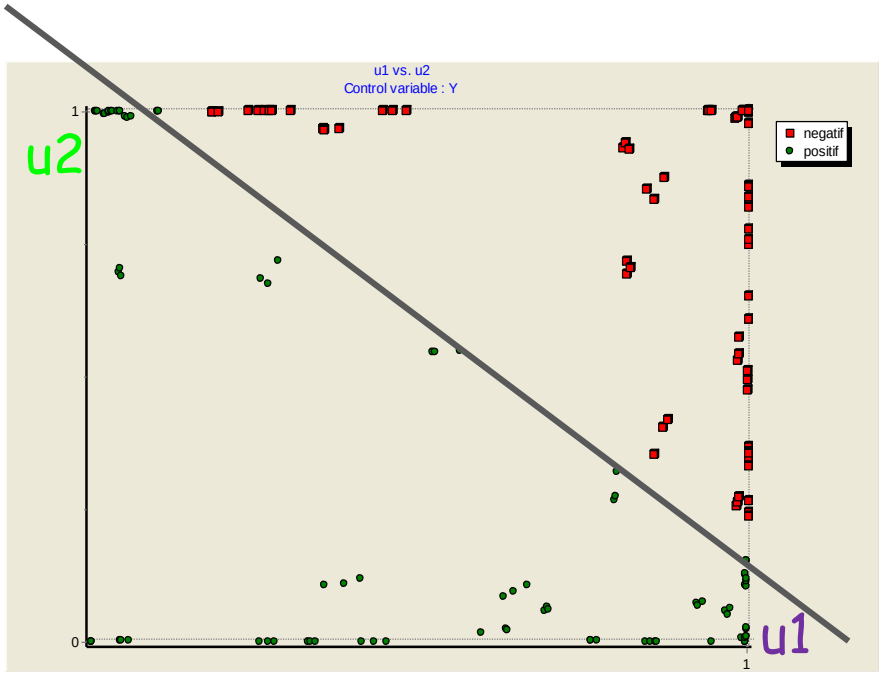


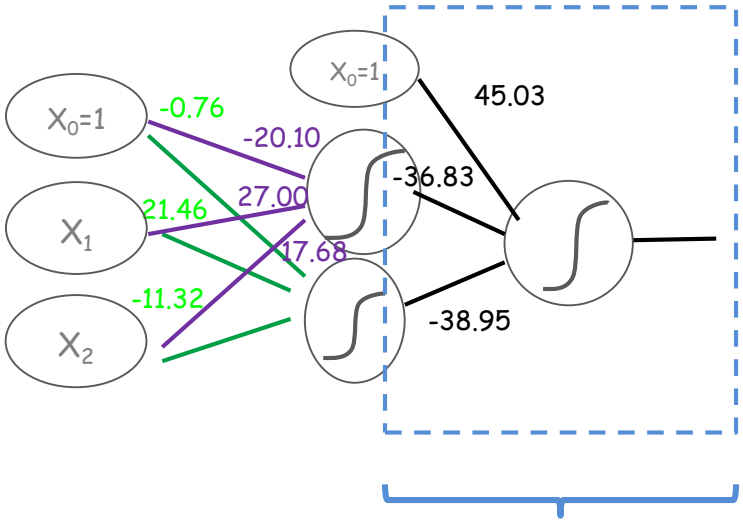
$$D1: -20.10 + 27.00\ X1 + 17.68\ X2 = 0$$

$$D2: -0.76 + 21.46\ X1 - 11.32\ X2 = 0$$

Perceptron multicouche – Espace intermédiaire de représentation

2ème vision du PMC (couche cachée vers couche de sortie): la couche cachée définit un espace intermédiaire de représentation (une sorte d'espace factoriel) où l'on définit un séparateur linéaire (en espérant que les points soient linéairement séparables, si ce n'est pas le cas on peut augmenter le nombre de neurones dans la couche cachée… avec le danger du surapprentissage…).



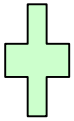


$$U1 = \frac{1}{1+e^{-(20.10+27.0\ X1+17.68\ X2)}}$$

$$U2 = \frac{1}{1+e^{-(-0.76+21.46\ X1-11.32\ X2)}}$$

$$D : 45.03 - 36.83\ U1 - 38.95\ U2 = 0$$

Perceptron multicouche – Avantages et inconvénients



Classifieur très précis (si bien paramétré)

Incrémentalité

Scalabilité (capacité à être mis en œuvre sur de grandes bases)

Modèle boîte noire (causalité descripteur – variable à prédire)

Difficulté de paramétrage (nombre de neurones dans la couche cachée)

Problème de convergence (optimum local)

Danger de sur-apprentissage (trop de neurones dans la couche cachée)

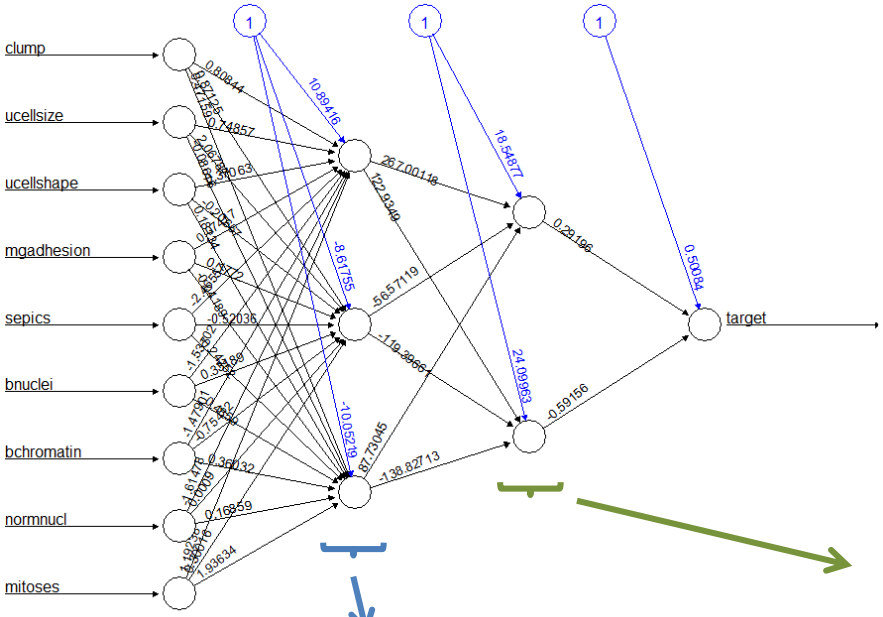
Plusieurs couches cachées, régularisation, (constante) d'apprentissage, etc.

PLUS LOIN AVEC LE PERCEPTRON MULTICOUCHE

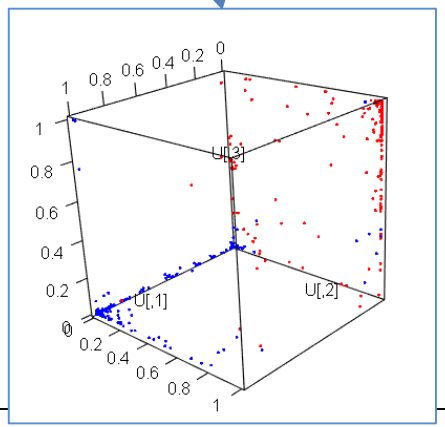
PMC avec plusieurs couches cachées

Pourquoi plusieurs couches cachées ? Démultiplie le pouvoir explicatif du modèle, mais : (1) le paramétrage est quasi inextricable (sauf à l'utiliser les couches intermédiaires comme filtres, cf. les réseaux de convolution) ; (2) la lecture des couches (des systèmes de représentation) intermédiaires n'est pas évidente.

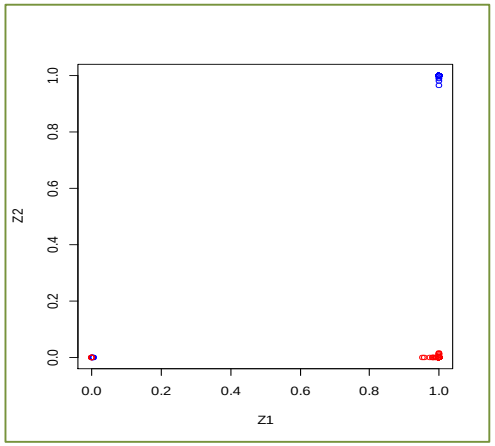
« Breast cancer » dataset



A noter : (A) Dès l'espace U, on avait une bonne discrimination ; (B) Z2 suffit à la discrimination ; (C) saturation des valeurs à 0 ou 1.



Espace de représentation à 3 dimensions.



Espace de représentation à 2 dimensions. Séparation linéaire possible.

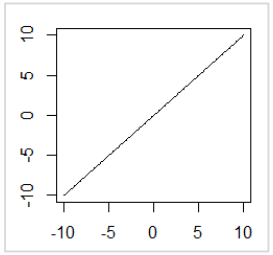
Fonctions d'activation

Différents types de fonctions d'activation sont utilisables. En fonction du problème à traiter, de la définition des espaces intermédiaires, du filtrage que l'on veut effectuer sur les données…. Il faut être attentif à ce que l'on veut obtenir.

Fonction linéaire

Pour la régression, pas de transformation

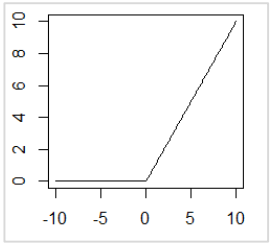
$$u = x$$



Fonction ReLu (Rectified Linear units)

Filtre les valeurs négatives

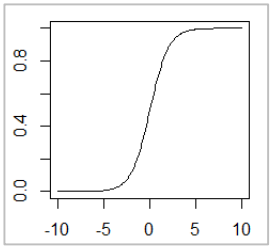
$$u = \max(0, x)$$



Fonction Sigmoïde

Ramène les valeurs entre [0, 1]. Utilisé pour le classement. Y codé {0,1}.

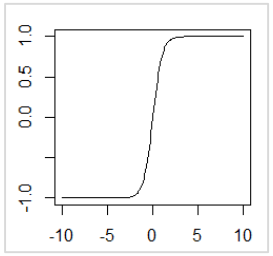
$$u = \frac{1}{1 + e^{-x}}$$



Fonction Tangente hyperbolique

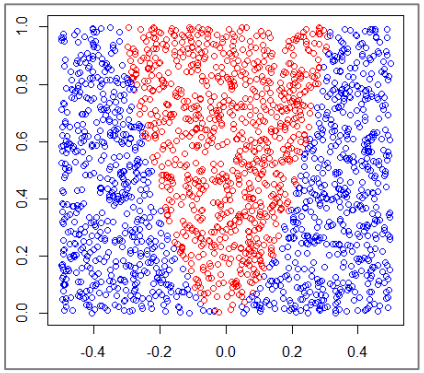
Ramène les valeurs entre [-1, 1]. Alternative à sigmoïde pour le classement. Y codé {-1,+1}.

$$u = \frac{e^{2x} - 1}{e^{2x} + 1}$$



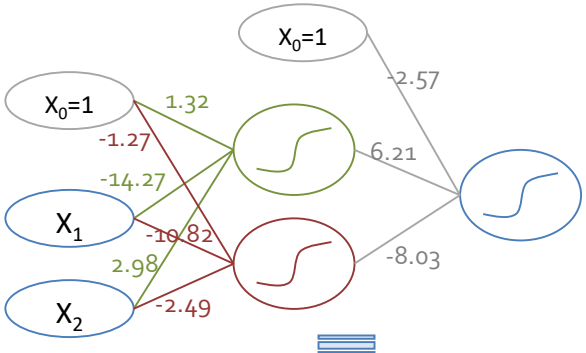
Fonctions d'activation

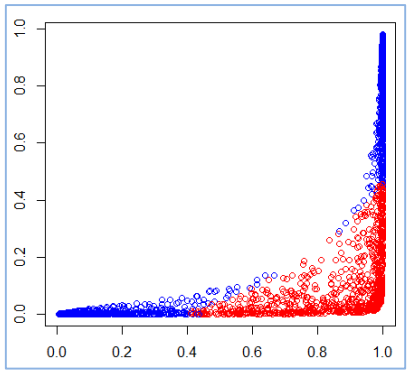
Mixer les fonctions dans un PMC



Problème à deux classes. A l'évidence une couche cachée avec 2 neurones suffit… mais cela est-il valable pour tous types de fonctions d'activations ?

Fonction sigmoïde dans toutes les couches.



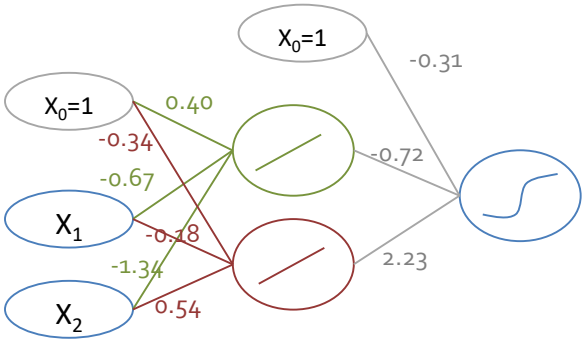


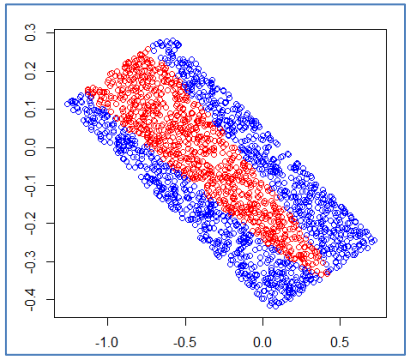
$$u1 = \frac{1}{1 + e^{-(1.32 - 14.27\,x1 + 2.98\,x2)}}$$

$$u2 = \frac{1}{1 + e^{-(-1.27 - 10.82\,x1 - 2.49\,x2)}}$$

Fonction linéaire dans la couche cachée.

Fonction sigmoïde dans la sortie.





$$z1 = 0.40 - 0.67\,x1 - 1.34\,x2$$

$$z2 = -0.34 - 0.18\,x1 + 0.54\,x2$$

Avec la fonction d'activation linéaire, l'espace intermédiaire n'est pas discriminant (linéairement s'entend).

Fonctions de perte pour le classement

Processus d'optimisation repose sur la descente du gradient (cf. page 15). E est la fonction de perte à optimiser.

$$a_j = a_j - \eta \frac{\partial E}{\partial a_j}$$

Différentes fonctions de perte sont possibles pour le classement. Il faut surtout choisir une fonction adaptée au problème à traiter (*il y en a d'autres… cf. Keras*)

- (Mean) squared error : $E = \frac{1}{2}\sum_{\omega}(y(\omega) - \hat{y}(\omega))^2$
- Binary cross-entropy : $E = \sum_{\omega} -y(\omega)\log\hat{y}(\omega) - (1 - y(\omega))\ \log(1 - \hat{y}(\omega))$
- Hinge loss (utilisé dans les SVM). Y codé {-1,+1} : $E = \sum_{\omega} \max(0,1 - y(\omega) \times \hat{y}(\omega))$

Gradient et gradient stochastique

Formule de mise à jour

$$a_j = a_j - \eta \frac{\partial E}{\partial a_j}$$

Calcul du gradient [Loss : squared error, Activation : sigmoïde g()]

$$\frac{\partial E}{\partial a_j} = \sum_{\omega} \left(y(\omega) - \hat{y}(\omega)\right) g\left(v(\omega)\right) \left[1 - g\left(v(\omega)\right)\right] x_j(\omega)$$

Descente du gradient (classique)

Mettre à jour les poids après avoir fait passer toutes les observations. Problème : lenteur parce que beaucoup de calculs sur les grandes bases.

Gradient stochastique (online)

Approche incrémentale (màj des poids au passage de chaque individu). Accélération du processus. On les mélange au hasard au départ. Problème : instabilité, oscillations.

Gradient stochastique (mini-batch)

Màj par blocs d'individus (size = paramètre). Cumuler les avantages de l'approche incrémentale (rapidité de convergence) et une forme de lissage (moins d'instabilité) dans le processus.

Faire évoluer le taux d'apprentissage

Taux d'apprentissage ($\eta > 0$)

$$a_j = a_j - \eta \frac{\partial E}{\partial a_j}$$

Comment concilier deux exigences antagoniques. Taux élevé au départ du processus pour aller rapidement vers la bonne zone de résultat ; taux faible à la fin pour plus de précision.

Réduire le taux d'apprentissage en fonction du nombre de passage sur l'ensemble des observations. Différentes solutions possibles.

Valeur de départ. Paramètre.

$$\eta(s) = \frac{\eta_0}{1 + s \times \eta_d}$$

Valeur de décroissance. Paramètre.

Itération. Ex. nombre de passage sur la base.

$$\eta(s) = \frac{\eta_0}{s^t}$$

Lissage de la décroissance. Paramètre.

Adaptatif. Laisser η constant tant que optimisation ok. Si stagnation, le réduire (ex. divisé par 5). Etc. Cf. Scikit-Learn.

Momentum

Les oscillations restent une hantise (surréaction par rapport au gradient calculé à une étape donnée). Pour **lisser** le cheminement vers la solution, on conserve une mémoire de la correction à l'étape précédente.

$$a_j^{(k)} = a_j^{(k-1)} + \Delta_j^{(k)}$$

Pour calculer $\Delta_j^{(k)}$, on utilise le gradient et la correction effectuée à l'étape précédente (k-1).

$$\Delta_j^{(k)} = -\eta \left(\frac{\partial E}{\partial a_j} \right)^{(k)} + \mu \, \Delta_j^{(k-1)}$$

$(0 \leq \mu < 1)$ est un paramètre supplémentaire à manipuler. $(\mu = 0)$, on a la descente du gradient usuelle. $(\mu = 0.9)$ par défaut dans Scikit-Learn par ex.

Régularisation des coefficients

Dans un contexte de colinéarité et de forte dimensionnalité, les coefficients (poids synaptiques) peuvent être très erratiques. Pour les stabiliser / harmoniser (et éviter le surapprentissage c.-à-d. la sur-dépendance aux données) on peut imposer des contraintes sur leurs valeurs. Cf. cours « Régression Ridge – Lasso – Elasticnet ».

Contrainte sur la norme L2 (Ridge)

Réécriture de la fonction de perte à optimiser.

$$E = \frac{1}{2}\sum_{\omega}(y(\omega) - \hat{y}(\omega))^2 + \frac{\alpha}{2}\sum_{j} a_j^2$$

Conséquence sur le gradient, intégration de la contrainte sans difficulté.

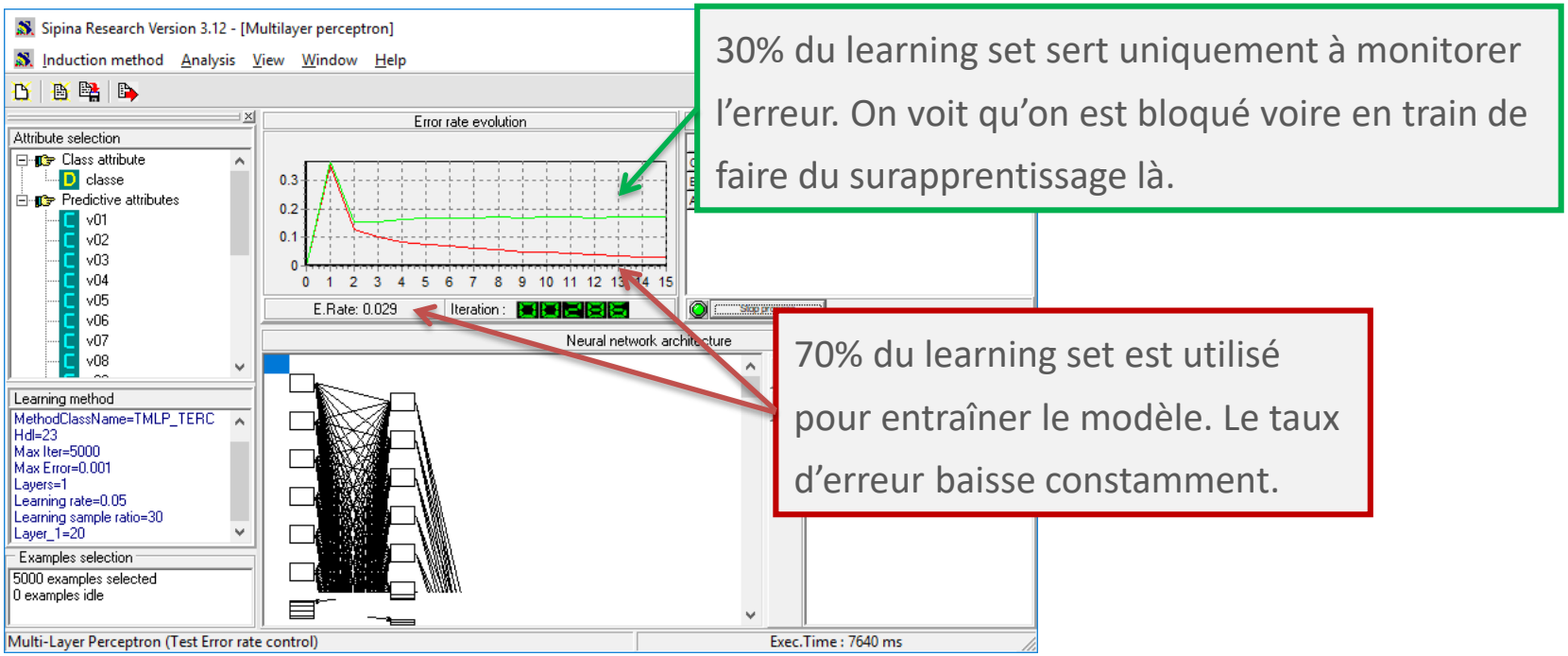
$$\frac{\partial E}{\partial a_j} = \sum_{\omega}(y - \hat{y})\, g(v)\,[1 - g(v)]\, x_j + \alpha \times a_j$$

Plus α est élevé, plus on impose des contraintes sur les coefficients (a_j faible), avec le danger de faire du sous-apprentissage (ne plus exploiter efficacement les informations dans les données !).

Nous pouvons également imposer une contrainte sur la norme L1 (Lasso) ou encore combiner Ridge et Lasso (Elasticnet). Cf. la documentation de Keras.

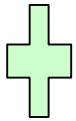
Echantillon de validation

S'appuyer sur l'évolution de la performance du MLP sur l'échantillon d'apprentissage n'a pas trop de sens parce qu'elle s'améliore constamment (presque). L'idée est de réserver une fraction des données (échantillon de validation, validation fraction) pour surveiller cette évolution et pourquoi pas définir une règle d'arrêt (ex. stagnation au bout de x itérations).



Cf. les paramètres EARLY_STOPPING et VALIDATION_FRACTION dans Scikit-learn.

Bilan



De nombreuses améliorations ces dernières années

Notamment avec des algorithmes d'optimisation (au-delà du gradient stochastique) performants

Librairies de calcul puissantes disponibles sous R et Python

Attention au paramétrage

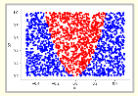
Bien lire la documentation des packages pour ne pas s'y perdre

Faire des tests en jouant sur les paramètres « simples » (ex. architecture du réseau) dans un premier temps, affiner par la suite.

Librairies sous Python et R

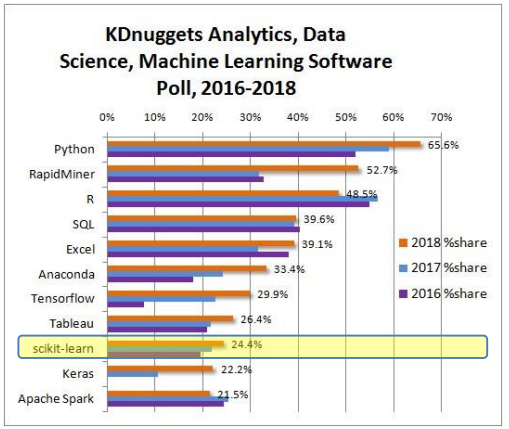
PRATIQUE DES PERCEPTRONS

```
#importation des données
import pandas
D = pandas.read_table("artificial2d_data2.txt",sep="\t",header=0)

#graphique
code_couleur = D['Y'].eq('pos').astype('int')
D.plot.scatter(x="X1",y="X2",c=pandas.Series(['blue','red'])[code_couleur])

#séparer cible et descripteurs
X = D.values[:,0:2]
Y = D.values[:,2]

#subdivision en apprentissage et test
from sklearn import model_selection
XTrain,XTest,YTrain,YTest = model_selection.train_test_split(X,Y,train_size=1000)

#initialisation du classifieur
from sklearn.neural_network import MLPClassifier
rna = MLPClassifier(hidden_layer_sizes=(2,),activation="logistic",solver="lbfgs")

#apprentissage
rna.fit(XTrain,YTrain)

#affichage des coefficients
print(rna.coefs_)
print(rna.intercepts_)

#prédiction sur l'échantillon test
pred = rna.predict(XTest)
print(pred)

#mesure des performances
from sklearn import metrics
print(metrics.confusion_matrix(YTest,pred))
print("Taux erreur = " + str(1-metrics.accuracy_score(YTest,pred)))
```



1000 TRAIN, 1000 TEST

Scikit_Learn est une librairie de machine learning puissante pour Python.



solver : *{'lbfgs', 'sgd', 'adam'}, default 'adam'*

The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

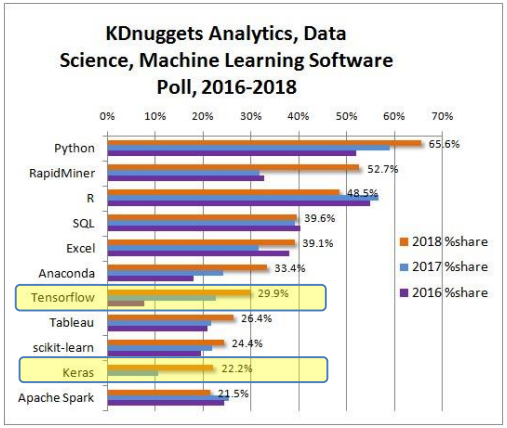
De l'importance du paramétrage. cf. LBFGS.

	neg	pos
neg	565	5
pos	2	428

$\varepsilon = 0.007$

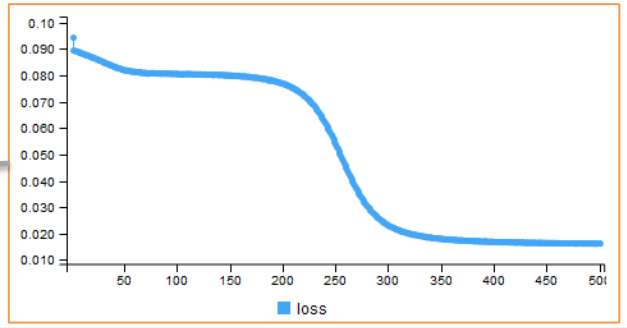
Keras sous R

Keras repose sur la technologie Tensorflow. Ces deux librairies de « deep learning » sont très populaires.




```r
#importer le package
library(keras)

#construire l'architecture du perceptron
rna <- keras_model_sequential()
rna %>%
  layer_dense(units = 2, input_shape = c(2), activation = "sigmoid") %>%
  layer_dense(units = 1, activation = "sigmoid")

#paramétrage de l'algorithme
rna %>% compile(
  loss="mean_squared_error",
  optimizer=optimizer_sgd(lr=0.15),
  metrics="mae"
)

#codage de la cible - éviter la saturation
yTrain <- ifelse(DTrain$Y=="pos",0.8,0.2)

#apprentissage avec son paramétrage
rna %>% fit(
  x = as.matrix(DTrain[,1:2]),
  y = yTrain,
  epochs = 500,
  batch_size = 10
)

#affichage des poids calculés
print(keras::get_weights(rna))

#prédiction sur l'échantillon test
pred <- rna %>% predict_classes(as.matrix(DTest[,1:2]))

#matrice de confusion
print(mc <- table(DTest$Y,pred))
print(paste("Taux erreur =", 1-sum(diag(mc))/sum(mc)))
```

Pour éviter que les valeurs des combinaisons linéaires soient élevées en valeurs absolues, et que la transformation sigmoïde sature à 0 ou 1 (dans la zone où la dérivée est nulle), on peut ruser en codant Y de manière à se situer plus vers la partie linéaire de la fonction d'activation.



L 'évolution de la perte est assez singulière quand-même. Si nous avions fixé (epochs = nombre de passages sur la base ≤ 200), nous n'aurions pas obtenu le bon résultat !

	neg	pos
neg	565	10
pos	1	424

$\varepsilon = 0.011$

RÉFÉRENCES

Références

- Blayo F., Verleysen M, « Les réseaux de neurones artificiels », Collection QSJ, PUF, 1996.
- Heudin J.C., « Comprendre le deep learning – Une introduction aux réseaux de neurones », Science eBook, 2016.
- Tutoriels Tanagra :
  - « Deep learning avec Tensorflow et Keras (Python) », avril 2018.
  - « Deep learning – Tensorflow et Keras sous R », avril 2018.
  - « Paramétrer le perceptron multicouche », avril 2013.