

APPRENTISSAGE SUPERVISÉ

SVM et NN (Neural Nets)

LES BOÎTES NOIRES

Nous parlerons dans ce cours de deux types de méthodes : les **SVM** et les **réseaux de neurones**. Leur point commun: ce sont des **boîtes noires** :

- Reposent sur des calculs complexes (leurs “moteurs” sont très difficiles à coder);
- Résultats sont difficilement interprétables;
- Nombreux paramètres devant être optimisés;
- Impliquent beaucoup de “tuning” manuel;
- Populaires car performantes, en particulier sur les données à structure complexe.
- Font le “Feature Engineering” à votre place

Support **V**ector **M**achines

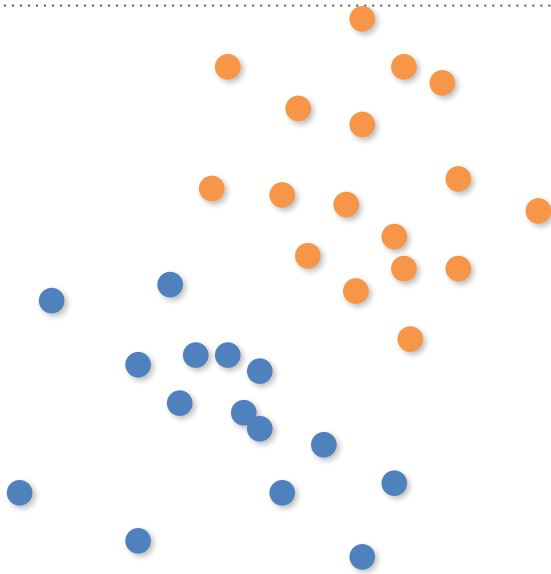
INTRODUCTION

L'arrivée des **SVMs** en **1992** marque un tournant dans l'histoire du **machine learning**.

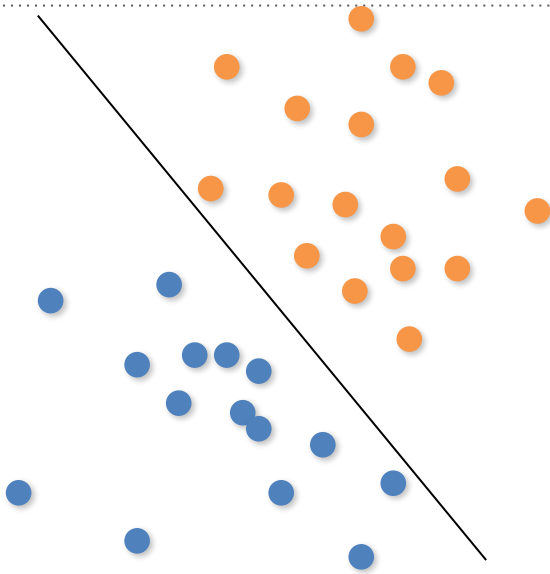
La théorie de laquelle ces méthodes sont issues est alors nouvelle, intuitive et permet de résoudre des problèmes complexes dans un **nouveau paradigme**.

Les notions mathématiques à la base des SVMs sont difficiles. Nous n'aborderons dans ce cours que leur surface.

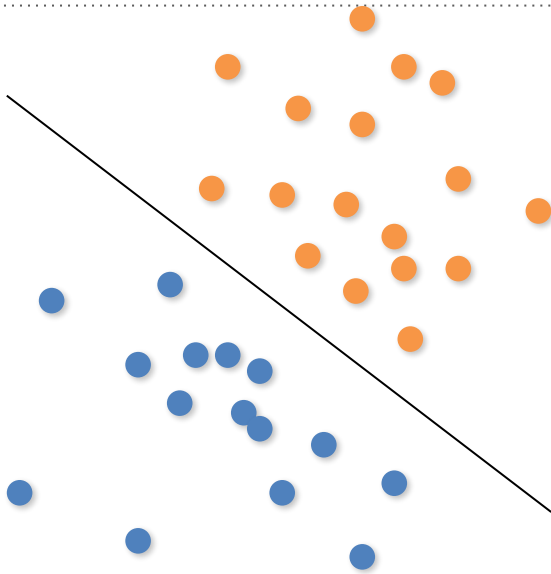
CLASSIFIEURS À VASTE MARGE: DONNÉES LINÉAIREMENT SÉPARABLES



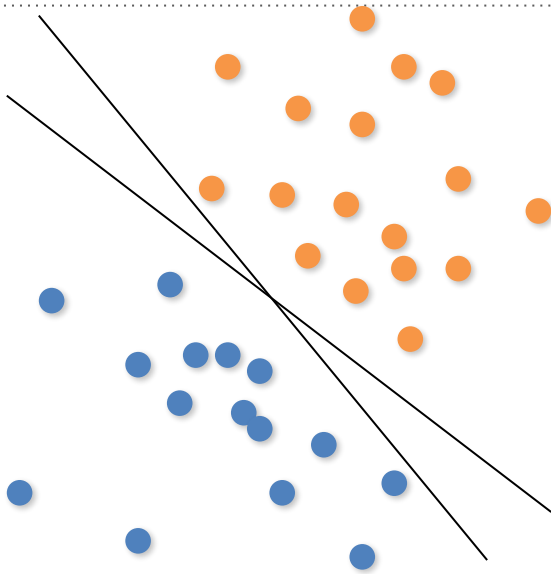
CLASSIFIEURS À VASTE MARGE: DONNÉES LINÉAIREMENT SÉPARABLES



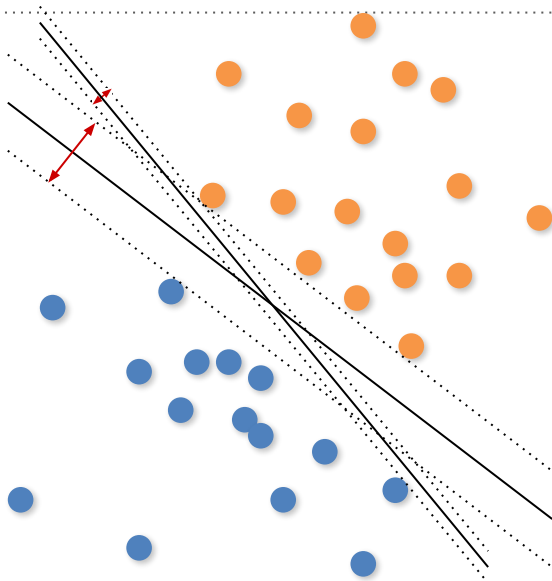
CLASSIFIEURS À VASTE MARGE: DONNÉES LINÉAIREMENT SÉPARABLES



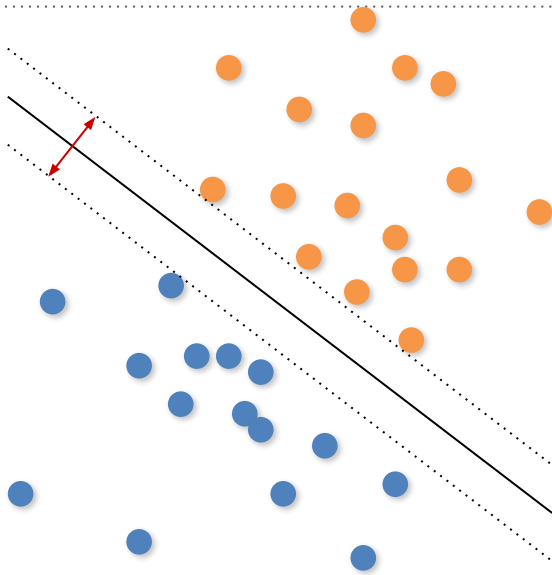
CLASSIFIEURS À VASTE MARGE: DONNÉES LINÉAIREMENT SÉPARABLES



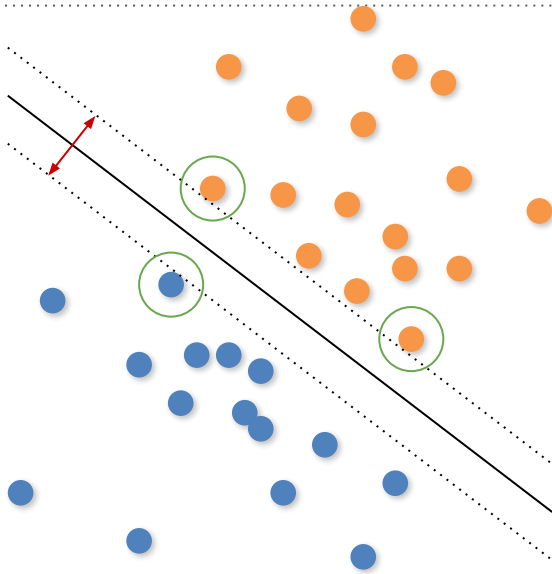
CLASSIFIEURS À VASTE MARGE: DONNÉES LINÉAIREMENT SÉPARABLES



CLASSIFIEURS À VASTE MARGE: DONNÉES LINÉAIREMENT SÉPARABLES

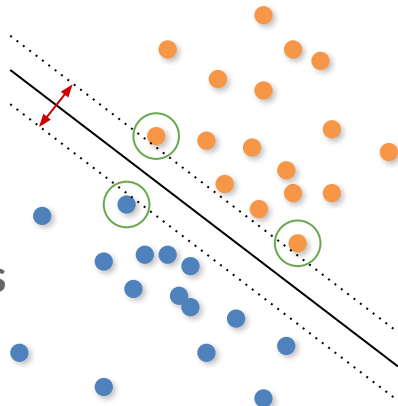


CLASSIFIEURS À VASTE MARGE: DONNÉES LINÉAIREMENT SÉPARABLES



CLASSIFIEURS À VASTE MARGE: DONNÉES LINÉAIREMENT SÉPARABLES

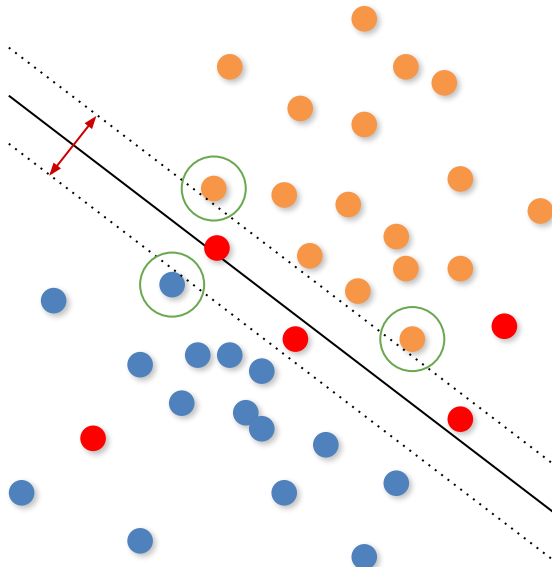
- Il existe une **infinité** de droites séparant les points.
- On cherche la **séparation linéaire** qui donne la plus grande **marge**.
- Les "bords" de cette marge s'appuient sur des **vecteurs supports**.
- Sous-jacent: problème d'**optimisation convexe**.



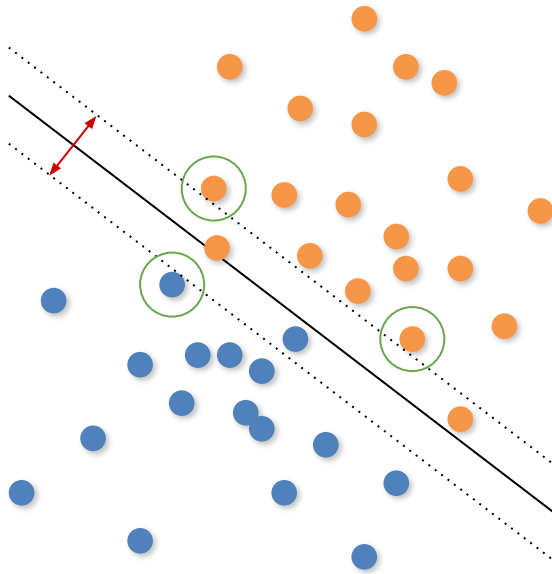
INTUITION

- Les **vecteurs supports** sont les points les plus **ambigus** et donc les plus **difficiles à classifier**.
- Ce sont ces points qui influencent le choix de la **meilleure droite** : si ces points changent, la droite change.
- En quelque sorte, on se met dans un "worst-case" scenario : puisque l'on sait classifier les points les plus ambigus, le système devrait être **robuste**.
- Plus la marge est grande, plus on est sûrs de nous.

PHASE DE TEST (ou PREDICTION)



PHASE DE TEST (ou PREDICTION)



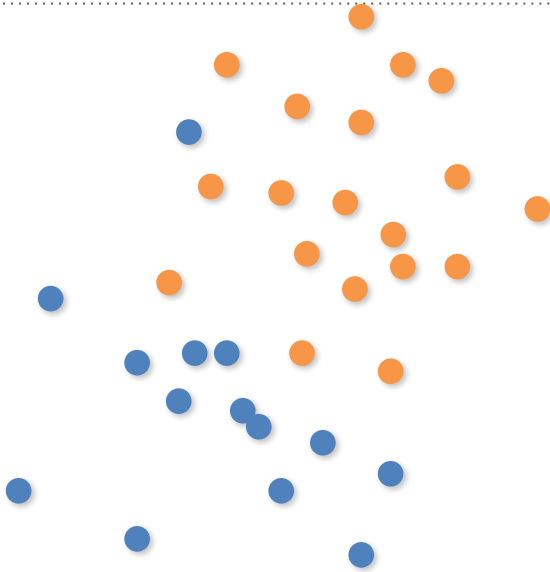
PROBLÈME

Les vraies données sont rarement
aussi simples !

PROBLÈME

Les vraies données ne sont ~~rarement~~
jamais aussi simples !

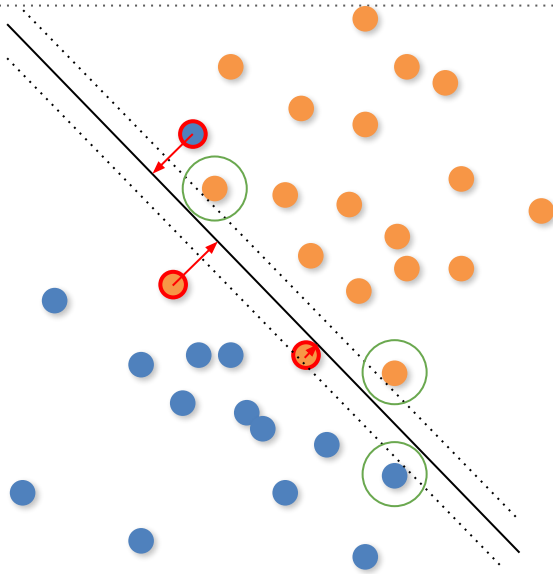
CLASSIFIEURS À VASTE MARGE: CAS **NON** LINÉAIREMENT SÉPARABLE



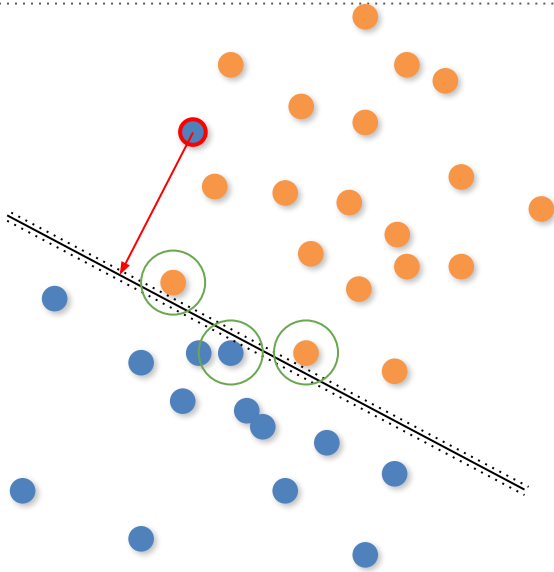
CLASSIFIEURS À VASTE MARGE: CAS NON LINÉAIREMENT SÉPARABLE

- On **autorise** des erreurs de classification sur l'**ensemble d'entraînement**. Il sera moins sensible aux **outliers** et donc plus robuste (moins de sur-apprentissage / overfitting)
- Mais jusqu'à quel point autoriser ces erreurs ?

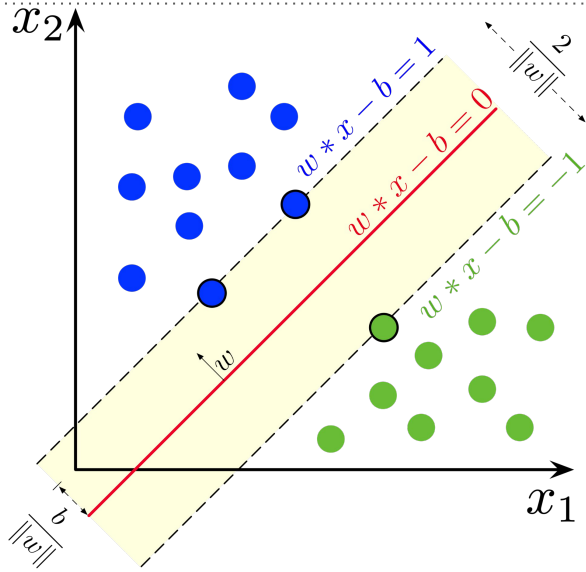
CLASSIFIEURS À VASTE MARGE: CAS NON LINÉAIREMENT SÉPARABLE



CLASSIFIEURS À VASTE MARGE: CAS NON LINÉAIREMENT SÉPARABLE



CLASSIFIEURS À VASTE MARGE: CAS NON LINÉAIREMENT SÉPARABLE



Source:
Wikipedia

CLASSIFIEURS À VASTE MARGE: CAS NON LINÉAIREMENT SÉPARABLE

.....
minimiser cette fonction (les y_i sont 1 ou -1):

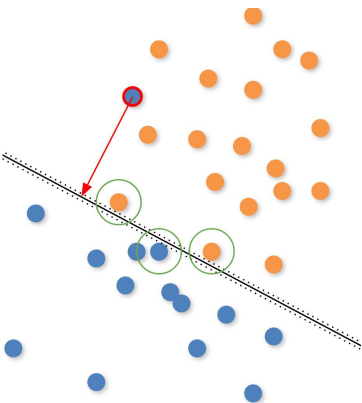
$$\left[\frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i (\vec{w} \cdot \vec{X}_i - b)) \right] + \lambda ||\vec{w}||^2$$

revient à trouver n coefficients $(\alpha_1, \dots, \alpha_n)$ tels que

$0 \leq \alpha_i \leq \lambda$ et: $\sum_{i=1}^n y_i \alpha_i = 0$ en minimisant:

$$\sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (\vec{X}_i \cdot \vec{X}_j)$$

CLASSIFIEURS À VASTE MARGE: CAS NON LINÉAIREMENT SÉPARABLE

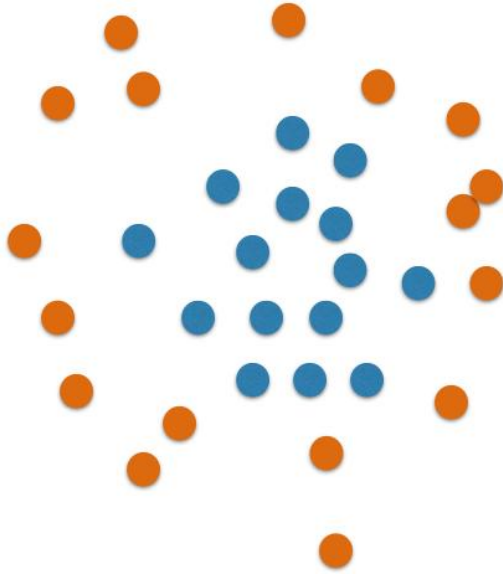


- On préfère des erreurs sur l'ensemble d'entraînement au **sur-apprentissage**.
- Le paramètre de **régularisation λ** fait le compromis entre marge large et erreurs.
- λ est optimisé, comme d'habitude, en **validation croisée**.

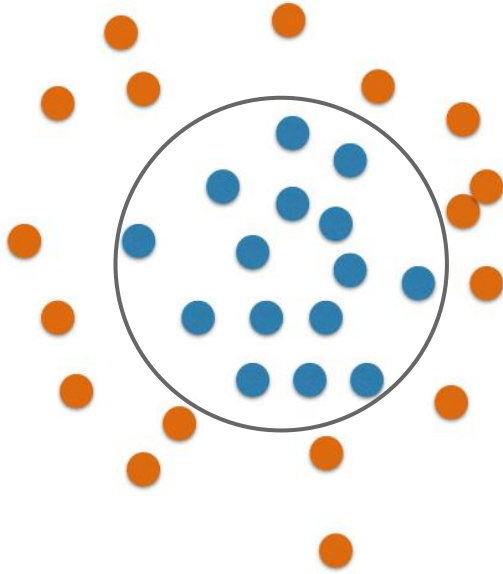
Démo

<http://cs.stanford.edu/people/karpathy/svmjs/demo/>

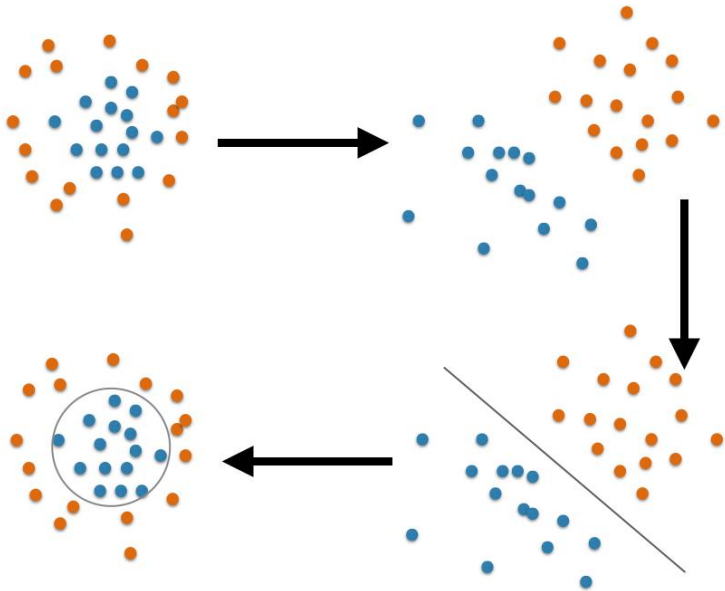
ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?



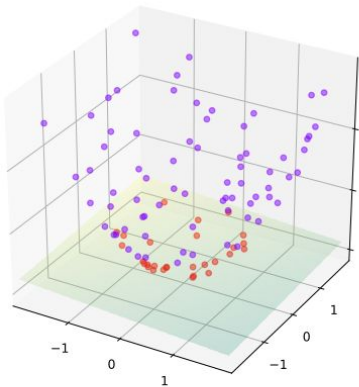
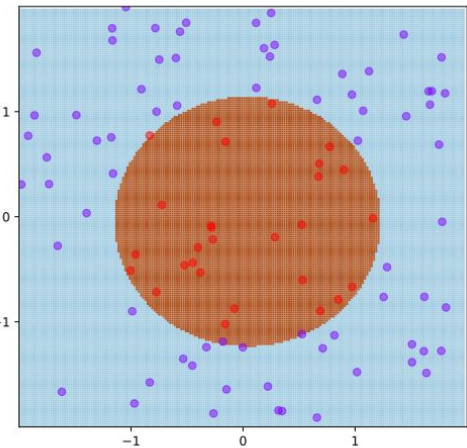
ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?



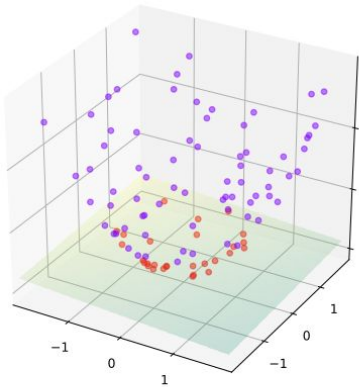
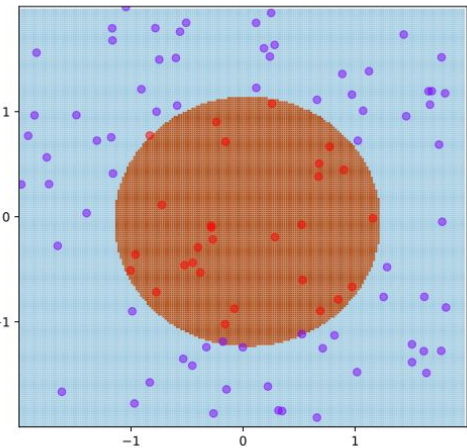
ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?



ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?

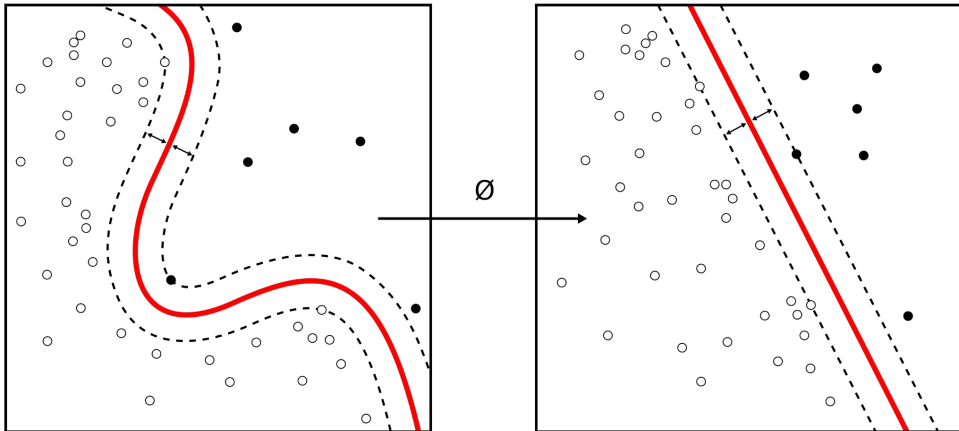


ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?



$$(x_1, x_2) \rightarrow (x_1, x_2, x_3) \text{ avec } x_3 = x_1^2 + x_2^2$$

ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?



Source:
Wikipedia

ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?

Idée: appliquer une **transformation** aux données pour que le problème redevienne linéaire: le classique "feature engineering".

Ici on utilise une fonction : $X \rightarrow \vec{\phi}(X)$

[RAPPEL SLIDE ANTÉRIEURE: IDÉE DES MATHS DERRIÈRE LES SVM]

.....
minimiser cette fonction (les y_i sont 1 ou -1):

$$\left[\frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i (\vec{w} \cdot \vec{X}_i - b)) \right] + \lambda ||\vec{w}||^2$$

revient à trouver n coefficients $(\alpha_1, \dots, \alpha_n)$ tels que

$0 \leq \alpha_i \leq \lambda$ et: $\sum_{i=1}^n y_i \alpha_i = 0$ en minimisant:

$$\sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (\vec{X}_i \cdot \vec{X}_j)$$

ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?

Idée: appliquer une **transformation** aux données pour que le problème redevienne linéaire: le classique "feature engineering".

Ici on utilise une fonction: $X \rightarrow \vec{\varphi}(X)$

$$\left[\frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i (\vec{w} \cdot \vec{\varphi}(X_i) - b)) \right] + \lambda ||\vec{w}||^2$$

devient, si on note $K(X, X') = \overrightarrow{\varphi(X)} \cdot \overrightarrow{\varphi(X')}$

$$\sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{K}(X_i, X_j)$$

[pour une présentation bien plus complète au niveau mathématique, voir [ces slides](#).]

ET SI LA SÉPARATION LINÉAIRE N'EST PAS DU TOUT ADAPTÉE ?

- On peut donc se **dispenser d'exprimer la transformation** $X \rightarrow \vec{\varphi}(X)$
On a juste besoin d'une fonction $K(X, X')$. Dans le cas linéaire, c'est le produit scalaire.
- On peut choisir une autre $K(X, X')$!
- On appelle K le **noyau** (kernel). Intuitivement c'est une **similarité** entre 2 points du dataset.
- Il existe différents types de noyaux. Il faut choisir le plus adapté aux données.

Démo

<http://cs.stanford.edu/people/karpathy/svmjs/demo/>

COMMENT DÉFINIR LE BON NOYAU ?

Il existe des noyaux “off-the shelf”, tous prêts à être utilisés. On peut en essayer différents pour voir ce qui fonctionne le mieux.

Pour entraîner un SVM, on a “juste” besoin du noyau $K(X, X')$. On n’a pas besoin de savoir représenter les objets eux-mêmes.

Donc on utilise un SVM en particulier quand on sait définir la similarité entre deux objets (similarité exprimée dans le noyau), mais qu’on n’a pas de “features” pour les objets.

QUELQUES NOYAUX

Linéaire: $K(X_i, X_j) = \vec{X}_i \cdot \vec{X}_j = \prod_{k=1}^d x_{i,k} x_{j,k}$

RBF: $K(X_i, X_j) = e^{-\gamma \sum_{k=1}^d (x_{i,k} - x_{j,k})^2}$

Polynomial:

$$K(X_i, X_j) = (c + \vec{X}_i \cdot \vec{X}_j)^p = \left(c + \prod_{k=1}^d x_{i,k} x_{j,k}\right)^p$$

Et surtout : les **noyaux ad-hoc** ! Sans décrire les objets (pas besoin de coordonnées x_1, x_2, \dots, x_p) : il suffit de décrire leur **similarité**, et de prendre cette fonction de similarité pour noyau.

Les noyaux **déforment** les distances entre les points et changent donc la “forme” du problème.

EXEMPLE AD-HOC

SVM pour classer des protéines. On peut comparer leurs 2 séquences de nucléotides comme on comparerait 2 chaînes de caractères:

Protéine 1: AGGGCTTACTA

Protéine 2: GGGCTACTAGGGGGCC

EXEMPLE AD-HOC

SVM pour classer des protéines. On peut comparer leurs 2 séquences de nucléotides comme on comparerait 2 chaînes de caractères:

Protéine 1: A**GGGCTT**ACTA

Protéine 2: GGGCT ACTAGGGGCC

Par exemple une **distance de Levenshtein**, avec trois opérations: insertion, deletion, remplacement.

EXEMPLE AD-HOC

SVM pour classer des protéines. On peut comparer leurs 2 séquences de nucléotides comme on comparerait 2 chaînes de caractères:

Protéine 1: A**GGGCTT**ACTA

Protéine 2: GGGCT ACTAGGGGCC

→ En calculant cette distance entre toutes les paires de protéines, vous pouvez entraîner un SVM avec ce noyau.

À aucun moment vous n'avez eu besoin de **features** pour représenter les protéines ! 🎉

EN PYTHON

```
.....  
from sklearn.svm import SVC  
model = SVC(C=1, kernel = 'linear')  
model.fit(Xtrain,Ytrain)  
predictions = model.predict(Xtest)
```

CONCLUSION SUR LES SVMS

Avantages :

- Performants.
- Ne nécessitent que la *similarité* des objets en entrée et non les objets en eux-mêmes. On peut traiter images, séquences ADN, vidéos...
- Efficaces en grande dimension.

Inconvénients :

- Il faut trouver un noyau adapté.
- Difficiles à interpréter.
- Dépendent de paramètres à optimiser.
- Peuvent être lents !

LES RÉSEAUX DE NEURONES

- Le perceptron multi-couches
- Le réseau convolutif

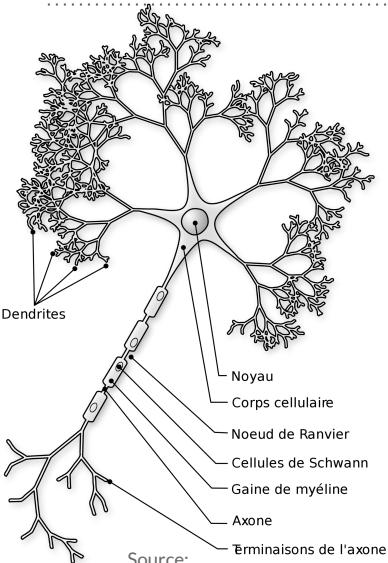
HISTOIRE

- Premières idées, années 50 : "copier" le fonctionnement du cerveau.
- 1986: **Perceptron multi-couches**.
- **Théorème (Cybenko, 1989)** : Toute fonction continue bornée est estimable, avec une précision arbitraire, par un réseau à 2 couches (avec "assez" de neurones).
- Aujourd'hui : extensions: **deep learning**, réseaux convolutifs, adversarial, generatif...
- **Boîte noire** par excellence : très compliqué de les interpréter, de les "tuner".

UN VRAI NEURONE

Dendrites = input. 7000 par neurone en moyenne.

Axone = output, unique. Des dendrites d'autres neurones s'y connectent (avg. 7000).



Source:
Wikipedia

En gros: l'axone s'active en "tout ou rien", quand le neurone reçoit assez de signaux par ses dendrites.

Activation = ϕ (combinaison linéaire des dendrites)

PRINCIPE

OUTPUT:
 $P(\in$
catégorie)



y_1



y_2



y_3



y_4

INPUT:
Pixels

x_1

x_2

x_3

x_4

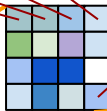
x_5

...

x_N

⋮

⋮



PRINCIPE

OUTPUT:
 $P(\in$
catégorie)



y_1



y_2



y_3



y_4



INPUT:
Pixels

x_1

x_2

x_3

x_4

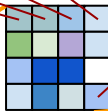
x_5

...

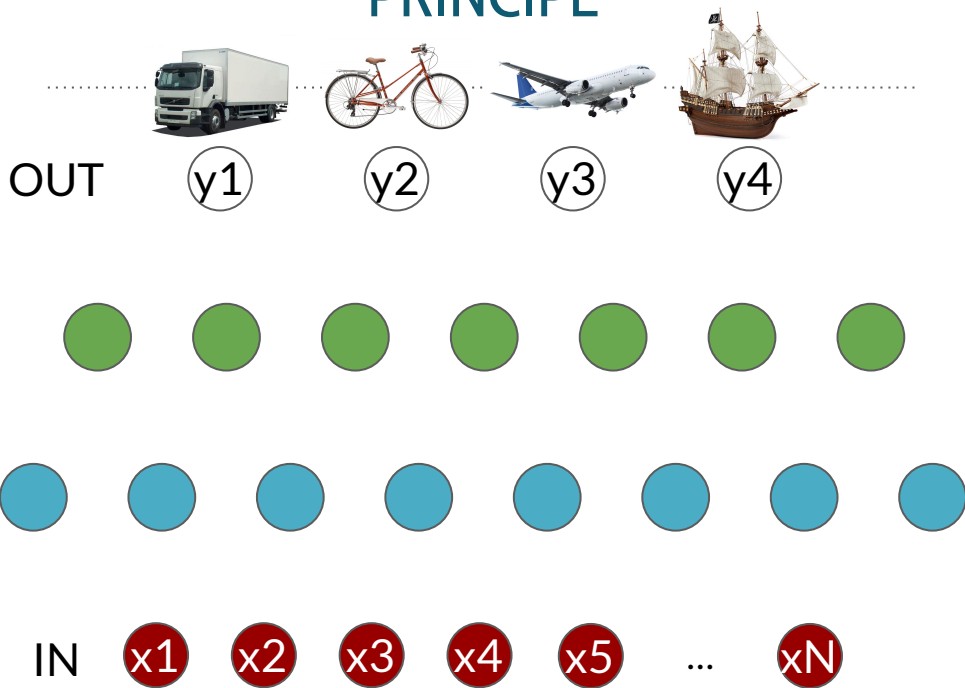
x_N

⋮

⋮



PRINCIPE



PRINCIPE



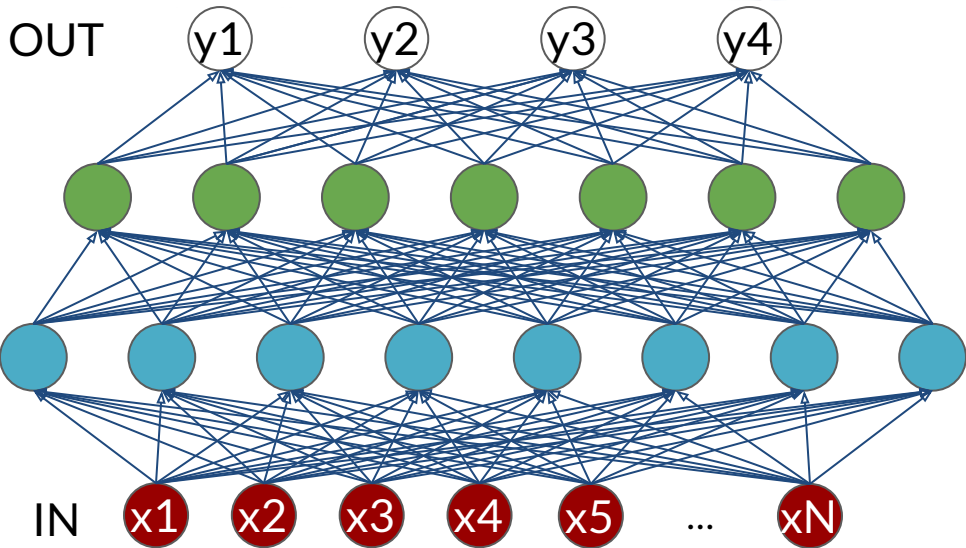
OUT

y1

y2

y3

y4



IN

x1

x2

x3

x4

x5

...

xN

PRINCIPE

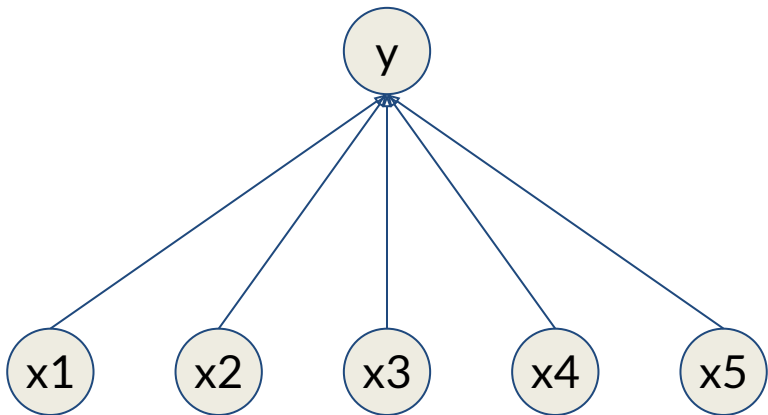
Le réseau est composé:

- d'une **entrée** (e.g. pixels) : N réels $\in [0, 1]$
- d'une couche de **sortie** (prédiction): K neurones
- de couches **intermédiaires** (de tailles arbitraires) appelées *couches cachées*.

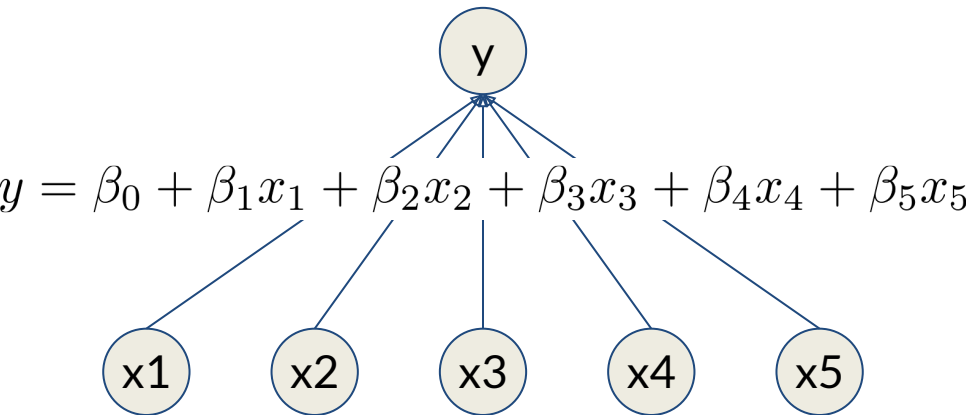
1^{ère} couche: chaque "neurone" est une fonction des N entrées. Puis à la $i+1$ ^{ème} couche, chaque neurone est une fonction de tous les neurones de la i ^{ème} couche précédente.

Chaque neurone de sortie (résultats) est donc une fonction (de fonctions de ...) de l'entrée.

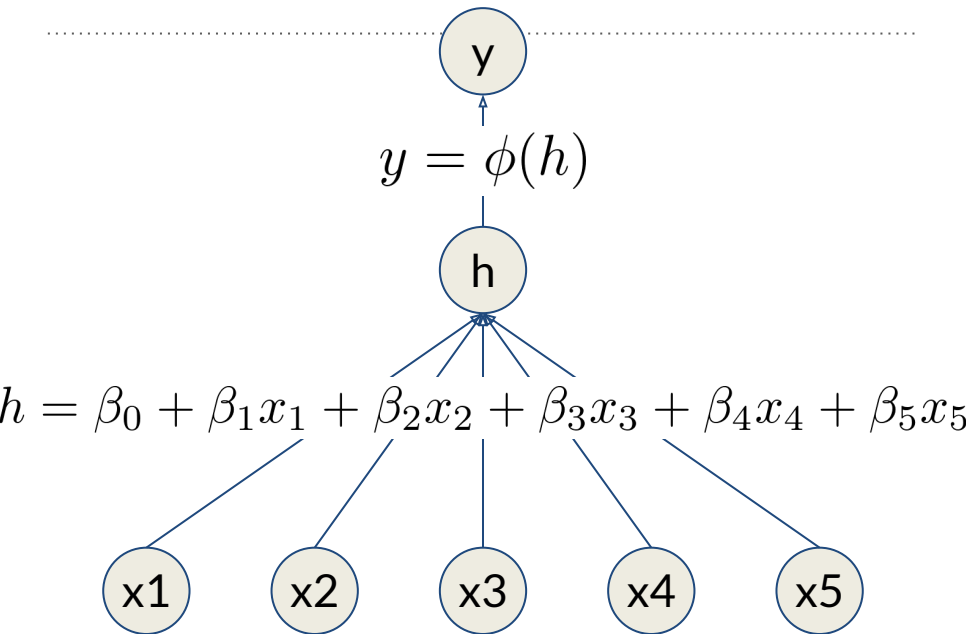
EXAMPLE



EXAMPLE



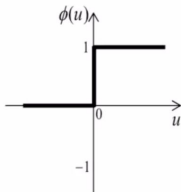
EXAMPLE



FONCTIONS D'ACTIVATION

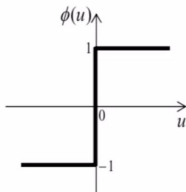
Source: slideshare,
présentation de
Sung-ju Kim

step function



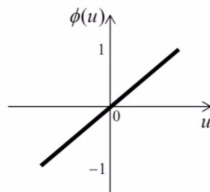
$$\phi_{step}(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

sign function

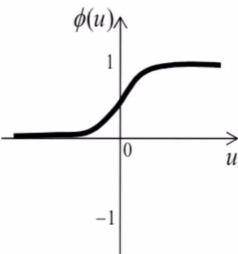


$$\phi_{sign}(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

identity function

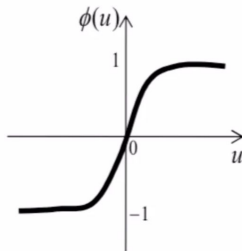


$$\phi_{id}(u) = u$$



$$\phi_{sig}(u) = \frac{1}{1 + e^{-u}}$$

sigmoid function



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

hyper tangent function

Friendship ended with SIGMOID

Now

RELU

is my
best friend



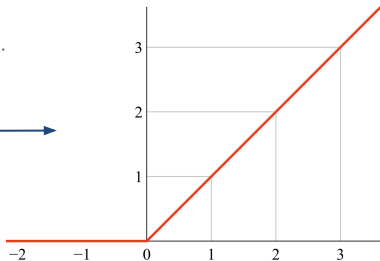
imgflip.com



FONCTIONS D'ACTIVATION

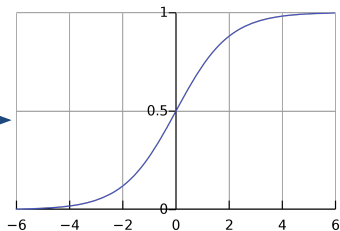
Couches **intermédiaires**:

ReLU le plus populaire,
parfois sigmoid.



Couche **finale**:

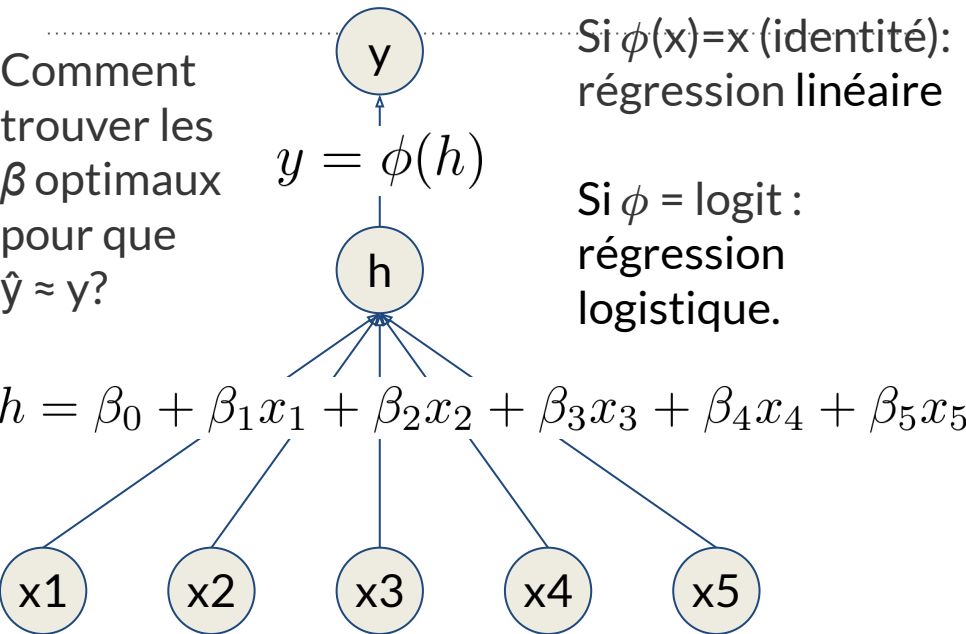
- Classification binaire
(1 neurone): Logit
- Multi-classification
(K neurones):



Softmax: $\phi(h_1, h_2, \dots, h_n)_i = \frac{e^{h_i}}{\sum_{j=1}^n e^{h_j}}$

EXEMPLE

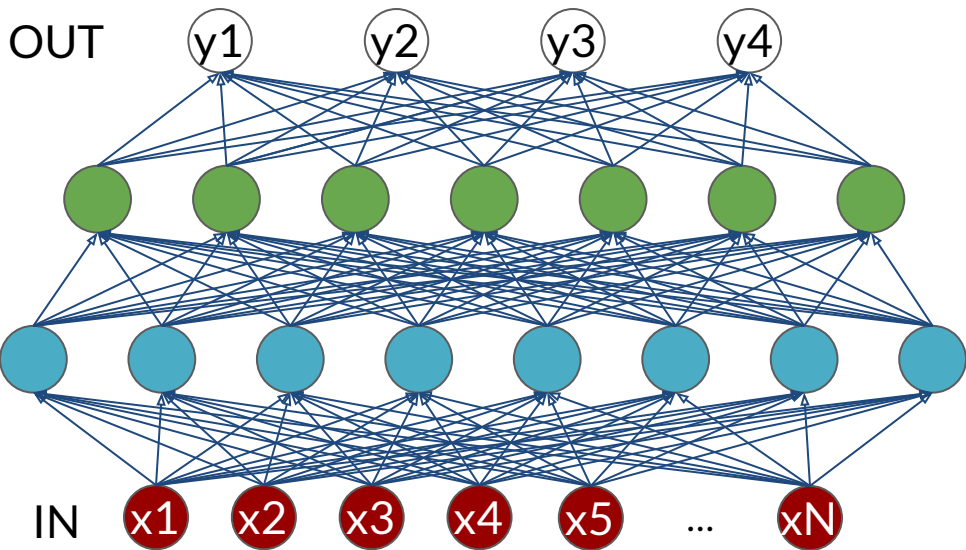
Comment
trouver les
 β optimaux
pour que
 $\hat{y} \approx y$?



Si $\phi(x)=x$ (identité):
régression linéaire

Si $\phi = \text{logit}$:
régression
logistique.

PLUS COMPLIQUÉ



PLUS COMPLIQUÉ

Le réseau est totalement connecté: chaque neurone est une fonction de tous les neurones de la couche précédente.

C'est une fonction linéaire des neurones précédents + une fonction d'activation.

Dans cette fonction, on a un paramètre par connection (+ un paramètre 'offset')

$$h_i^{(c)} = \phi \left(\beta_{c,i,0} + \beta_{c,i,1} h_1^{(c-1)} + \beta_{c,i,2} h_2^{(c-1)} + \dots \right)$$

COMBIEN DE PARAMÈTRES?

Votre réseau contient:

- 1000 neurones en couche d'entrée (les poids TF-IDF par exemple, ou des pixels)
- 100 neurones en couche cachée
- 4 neurones en couche de sortie (pour une classification à 4 classes possibles).

Combien de poids avez-vous en tout ?

COMBIEN DE PARAMÈTRES?

Votre réseau contient:

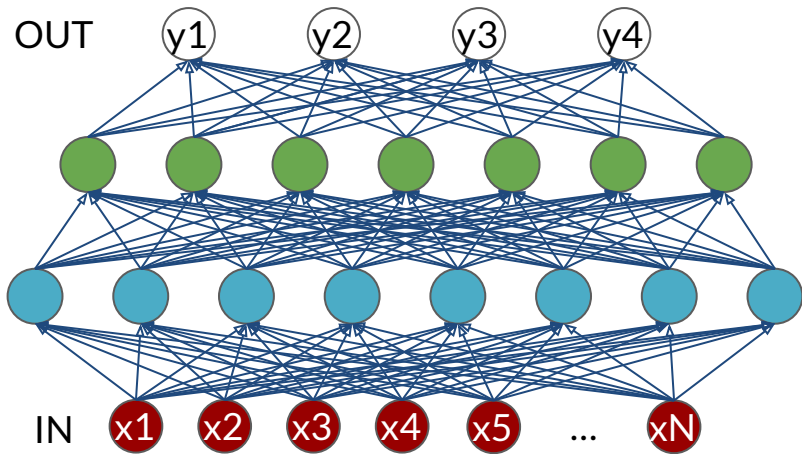
- 1000 neurones en couche d'entrée (les poids TF-IDF par exemple, ou des pixels)
- 100 neurones en couche cachée
- 4 neurones en couche de sortie (pour une classification à 4 classes possibles).

Combien de poids avez-vous en tout ?

Le réseau est totalement connecté. On a donc:
 $(1000 + 1) \times 100 + (100 + 1) \times 4 = \mathbf{100504}$ poids

100504: C'est un nombre **énorme** de paramètres (pour un assez "petit" réseau!).

COMMENT TROUVER LES POIDS OPTIMAUX ?



On ne peut pas! On va procéder de manière itérative avec la **rétro-propagation** (*Backpropagation*)

RÉTRO-PROPAGATION: 1 Neurone

$$y = 1$$

$$x_1 = 0.3$$

$$x_2 = 0.7$$

$$\beta_0 = 0.5$$

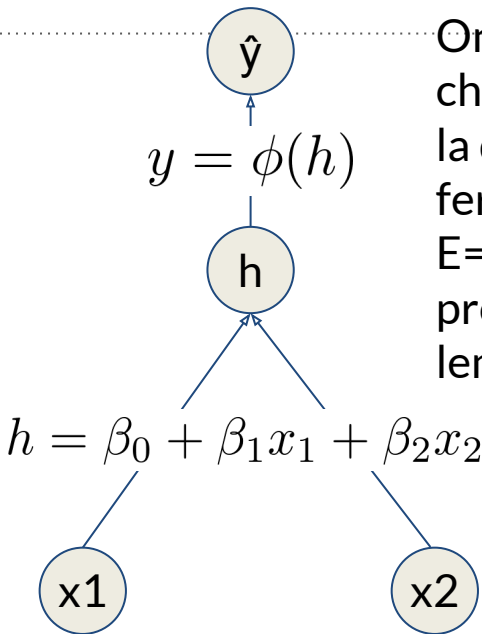
$$\beta_1 = 2$$

$$\beta_2 = -1$$

$$h = \dots$$

$$\hat{y} = \phi(h)$$

$$\phi = \text{logit}$$

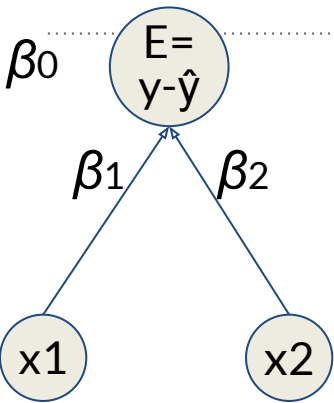


On modifie
chaque β_i dans
la direction qui
ferait bouger
 $E = y - \hat{y}$ vers 0,
proportionnel-
lement à E et à

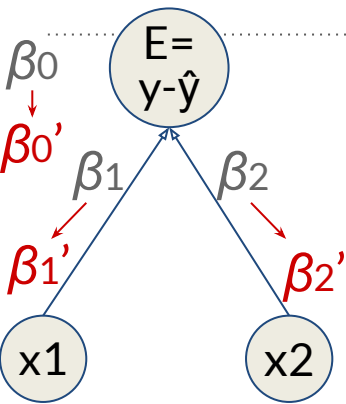
$$\frac{\partial E}{\partial \beta_i}$$

C'est une
**descente
de gradient**

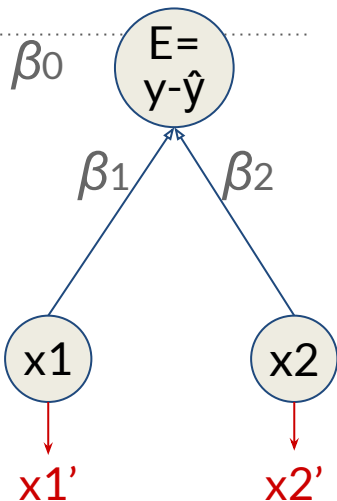
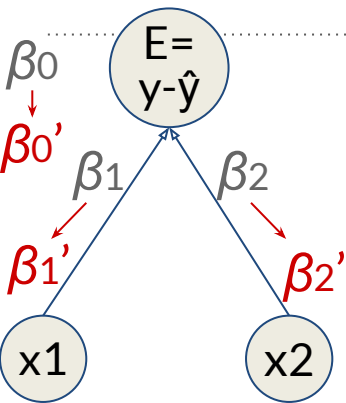
RÉTRO-PROPAGATION: Multi-Couche



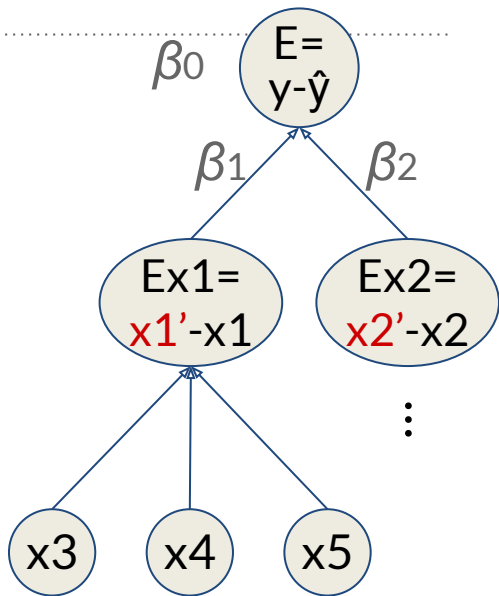
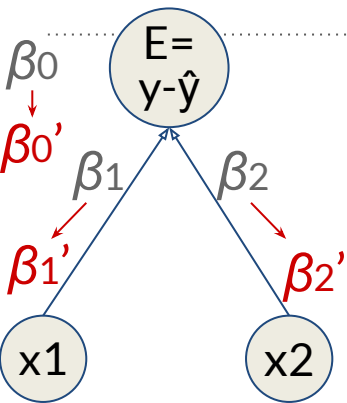
RÉTRO-PROPAGATION: Multi-Couche



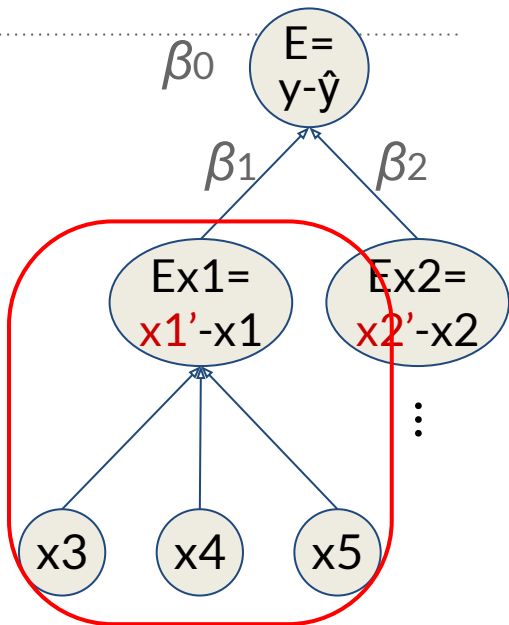
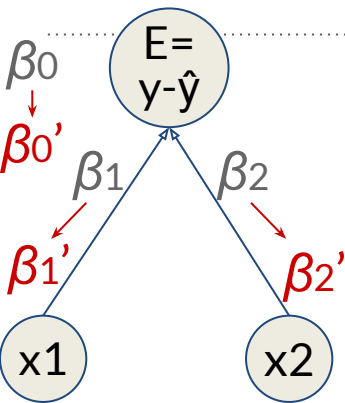
RÉTRO-PROPAGATION: Multi-Couche



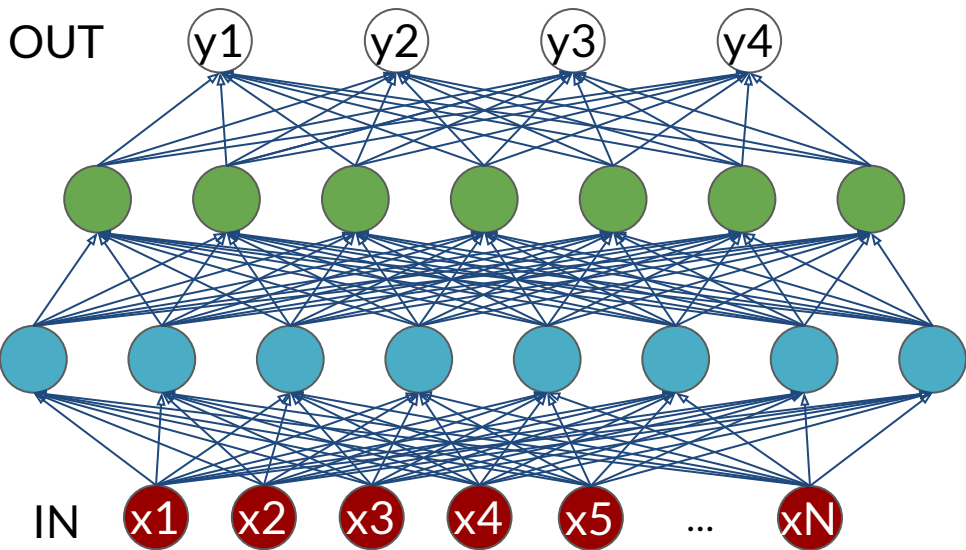
RÉTRO-PROPAGATION: Multi-Couche



RÉTRO-PROPAGATION: Multi-Couche

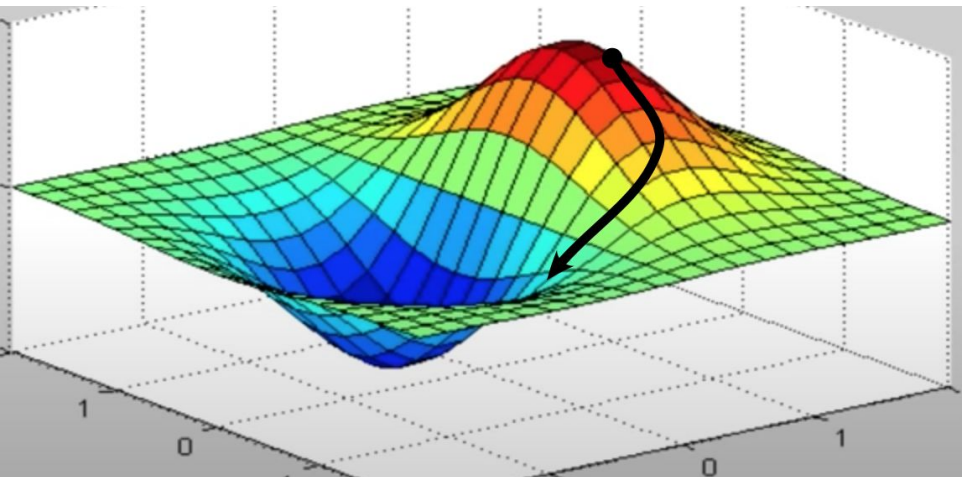


RÉTRO-PROPAGATION: Multi-couche



DESCENTE DE GRADIENT

Idée de la descente de gradient: faire des **petits pas** dans la direction de la pente jusqu'à atteindre le minimum.



DESCENTE DE GRADIENT

Idée de la descente de gradient: faire des petits pas dans la direction de la pente jusqu'à atteindre le minimum.

La taille des pas est ce qu'on appelle le taux d'apprentissage ou encore learning rate. Il nous dit à quel point ce que nous venons de voir doit influencer la modification des poids.

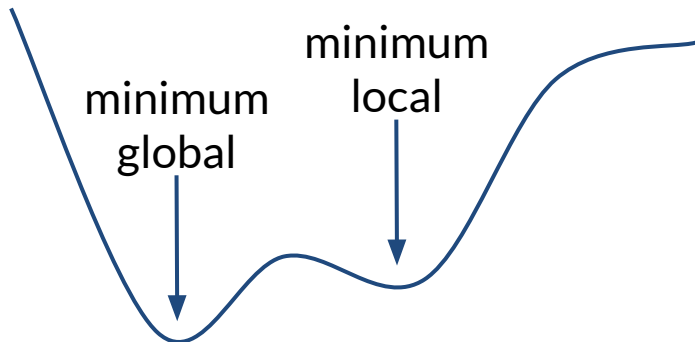
Il est généralement noté η .

DESCENTE DE GRADIENT

Rappel: fonction d'erreur pas (du tout!) convexe!

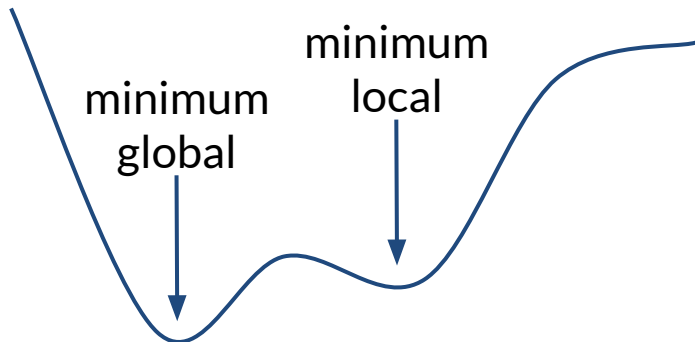
- Si le η est trop petit, on peut rester bloqué dans un minimum local.
- S'il est trop grand, on peut passer à côté du minimum global.

Aïe!



DESCENTE DE GRADIENT

“Solution”: ne pas choisir un learning rate fixe...
Par exemple, on peut choisir $\eta = 1/n$ où n est le nombre d'observations. La première observation modifiera beaucoup le réseau, la dernière l'affinera seulement.



SOMMAIRE


Je reçois une **nouvelle** image, je **prédis** grâce à mon réseau qu'il s'agit d'un chat.

Une fois au niveau de la **sortie**, je m'aperçois que c'est en fait un camion.

Je **rétro-propage** cette information pour **updater** la valeur de mes (millions de) paramètres.

Plus le réseau voit d'exemples labélisés, plus il

PARAMÈTRES GLOBAUX

- Complexité, forme:
 - Nombre de **couches** : + il y en a, + on peut prédire des choses complexes. Mais temps de calcul + long. Et  au **sur-apprentissage**!
 - Nombre de **neurones** : idem !
- **Fonctions** d'activation ϕ , pour couches intermédiaires et couche de sortie
- **Learning rate** η : taux d'apprentissage

BATCH LEARNING

En pratique, le plus souvent on ne fait pas la
rétro-propagation sur un exemple (X, Y) à la fois,
mais sur un batch (e.g. de 128 exemples).

Bénéfices:

- Éviter les zig-zags dans la descente de gradient (**très important!**)
- Grande optimisation des calculs :
 - Architecture vectorielle des CPUs
 - Parallelisation

Termes à connaître: **batch**, **epoch**.

RECONNAISSANCE D'IMAGES

Beaucoup trop de pixels.

Même si on réduit l'image (e.g. 100x100 pixels),
c'est encore trop!

→ ça ne marche pas bien.

Solution: prochain cours!

CONCLUSION SUR LE PERCEPTRON MULTI-COUCHES

Avantages :

- Potentiellement extrêmement performants, e.g. reconnaissance d'images, langage.
- Flexibles sur la complexité.

Inconvénients :

- Peu de résultats théoriques expliquant la perf.
- Impossibles à interpréter.
- Calculs **extrêmement** lourds.
- Complexité difficile à calibrer : gros risques de **sur-apprentissage**.