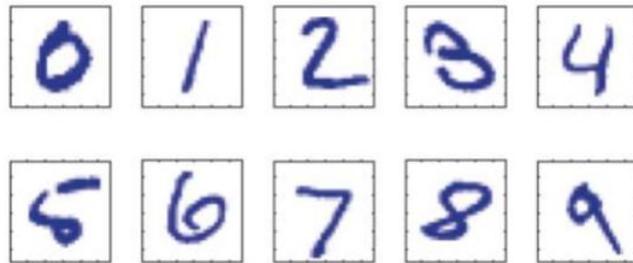


# JDIS Academy #5

Intro. au Machine Learning

# Apprentissage Automatique

**Question :** comment reconnaître des caractères manuscrits?

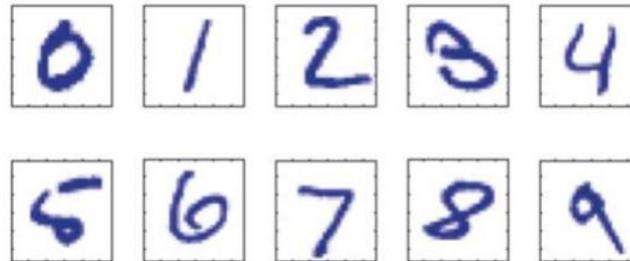


**Réponse :** Énumérer des règles?

- Une série de pixels alignés => ‘1’
- Une série de pixels en rond => ‘0’
- Etc.

# Apprentissage Automatique

**Question :** comment reconnaître des caractères manuscrits?



**Réponse :** Énumérer des règles? NON!

- Généralise mal à tous les cas



- Souvent fastidieux



Chien

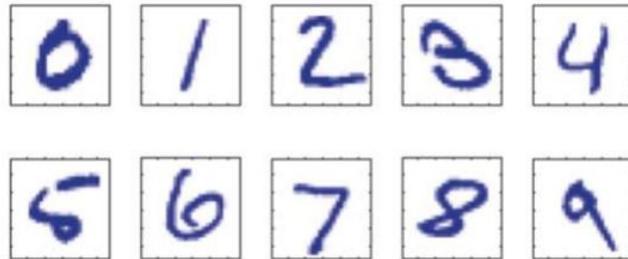


Oiseaux

Vs

# Apprentissage Automatique

**Question :** comment reconnaître des caractères manuscrits?



**Réponse :** Laisser l'ordinateur « **apprendre** » les règles

- Algorithmes d'apprentissage (*machine learning*)

# Objectif des algorithmes d'apprentissage

Partant d'un **ensemble d'entraînement:**  $D = \{(\vec{x}_1, t_1), (\vec{x}_2, t_2), \dots, (\vec{x}_N, t_N)\}$

$\vec{x}_i \in \Re^d$  donnée

$t_i$  cible associée à  $\vec{x}_i$

le but est **d'apprendre** une fonction qui sache prédire  $t_i$  partant de  $\vec{x}_i$

$$y_{\vec{w}}(\vec{x}_i) \rightarrow t_i$$

où  $\vec{w}$  sont les **paramètres** du modèle

# Apprentissage supervisé

Une fois le modèle  $y_{\vec{w}}(\vec{x})$  entraîné, on utilise un **ensemble de test**  $D_{test}$  pour mesurer la performance du modèle en **généralisation**.

○ 0 / 1 | 0 0 /  
‘0’ ‘0’ ‘1’ ‘1’ ‘1’ ‘0’ ‘0’ ‘1’

# Apprentissage supervisé

Deux sortes d'apprentissage supervisé

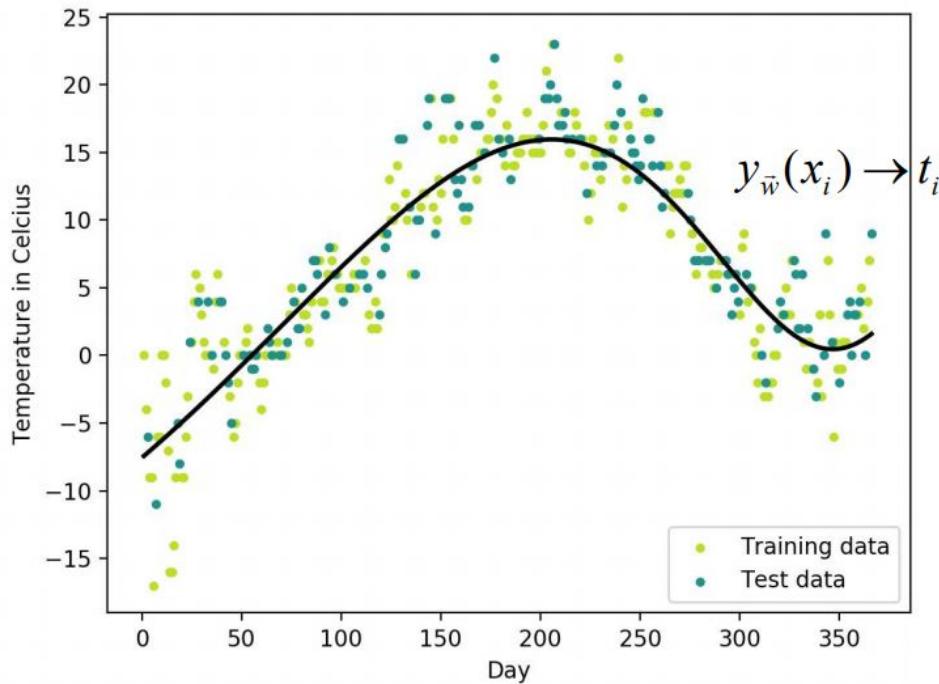
➤ **Classification** : la cible est un indice de classe  $t \in \{1, \dots, K\}$

- Exemple : reconnaissance de caractères
  - ✓  $\vec{x}$  : vecteur des intensités de tous les pixels de l'image
  - ✓  $t$  : identité du caractère

➤ **Régression** : la cible est un nombre réel  $t \in \mathbb{R}$

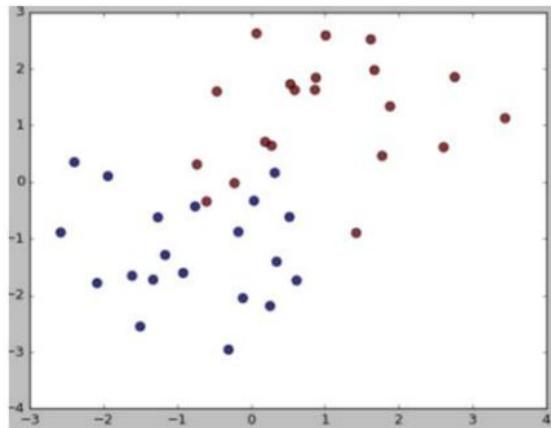
- Exemple : prédiction de la valeur d'une action à la bourse
  - ✓  $\vec{x}$  : vecteur contenant l'information sur l'activité économique de la journée
  - ✓  $t$  : valeur d'une action à la bourse le lendemain

# Exemple de régression

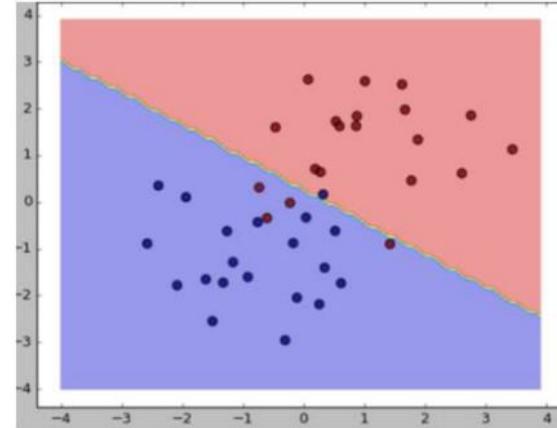


# Exemple de classification

Cas 2 classes



Soit des données issues de 2 classes ● et ● dans un espace à 2 dimensions



Une fois l'entraînement terminé

$$\begin{aligned}y(\bullet) &= \text{class 1} \\y(\bullet) &= \text{class 2}\end{aligned}$$

# Régression

# Régression linéaire

- Le modèle de **régression linéaire** est le suivant :

$$y_{\vec{w}}(\vec{x}) = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d$$

$$\text{où } \vec{x} = (x_1, x_2, \dots, x_d)^T$$

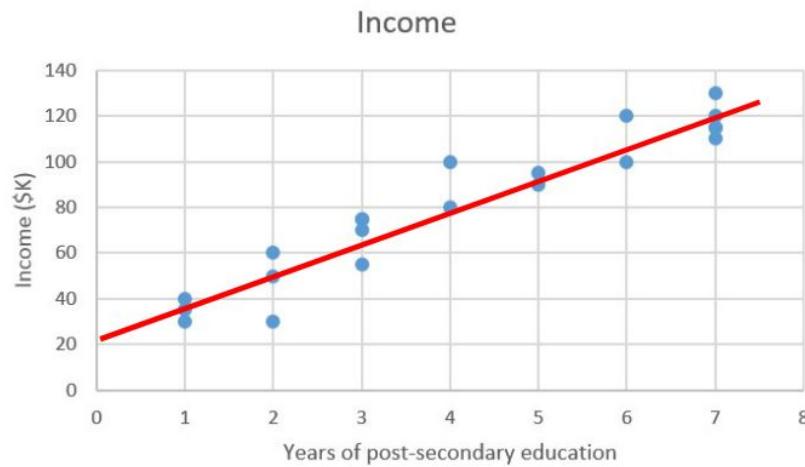
- La prédition correspond donc à

- Une **droite** pour  $d=1$
- Un **plan** pour  $d=2$
- Un **hyperplan** pour  $d>2$

# Régression linéaire 1D

Exemple

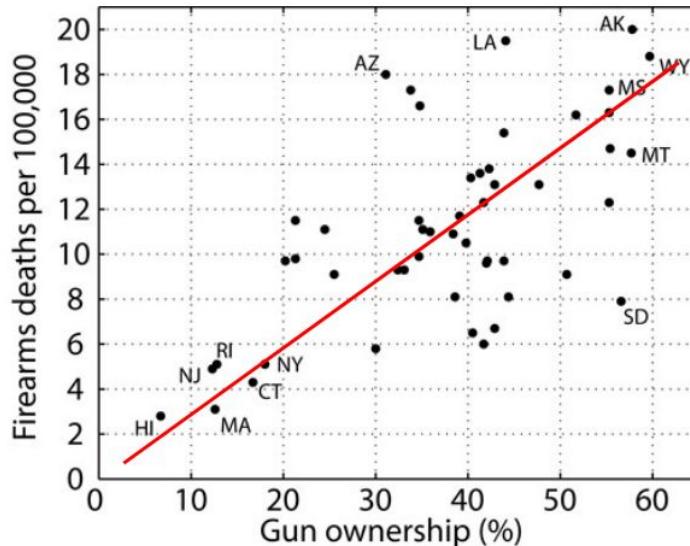
$$y_{\vec{w}}(x) = w_0 + w_1 x$$



# Régression linéaire 1D

Exemple

$$y_{\vec{w}}(x) = w_0 + w_1 x$$

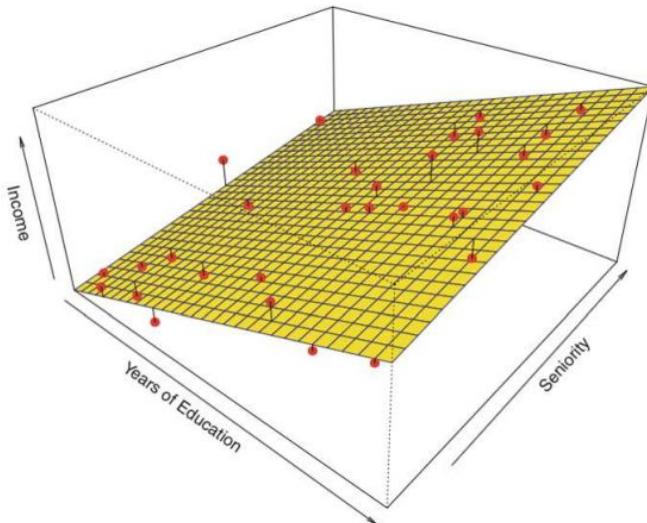


Source: <http://election.princeton.edu/2012/12/22/scientific-americans-gun-error/>

# Régression linéaire 2D

Exemple

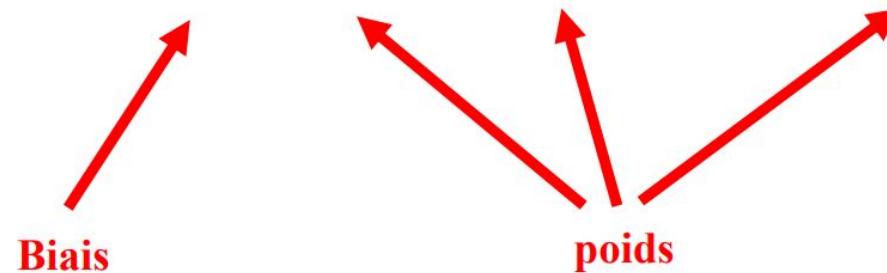
$$y_{\vec{w}}(\vec{x}) = w_0 + w_1 x_1 + w_2 x_2$$



Credit : [sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/R/R5\\_Correlation-Regression/R5\\_Correlation-Regression4.html](http://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/R/R5_Correlation-Regression/R5_Correlation-Regression4.html)

# Régression linéaire

$$y_{\vec{w}}(\vec{x}) = w_0 + w_1x_1 + w_2x_2 + \dots + w_dx_d$$



# Régression linéaire

Produit scalaire

Par simplicité, nous écrirons

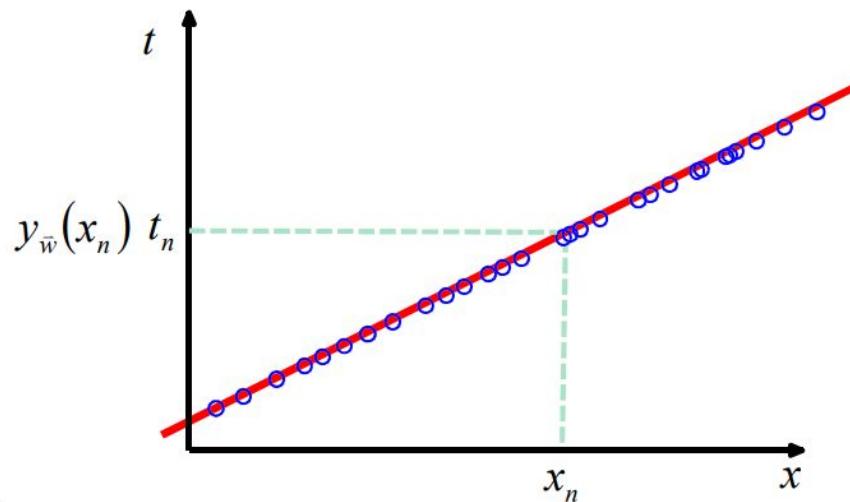
$$y_{\vec{w}}(\vec{x}) = \vec{w}^T \vec{x}$$

# Problème à résoudre

Soit un ensemble d'apprentissage :

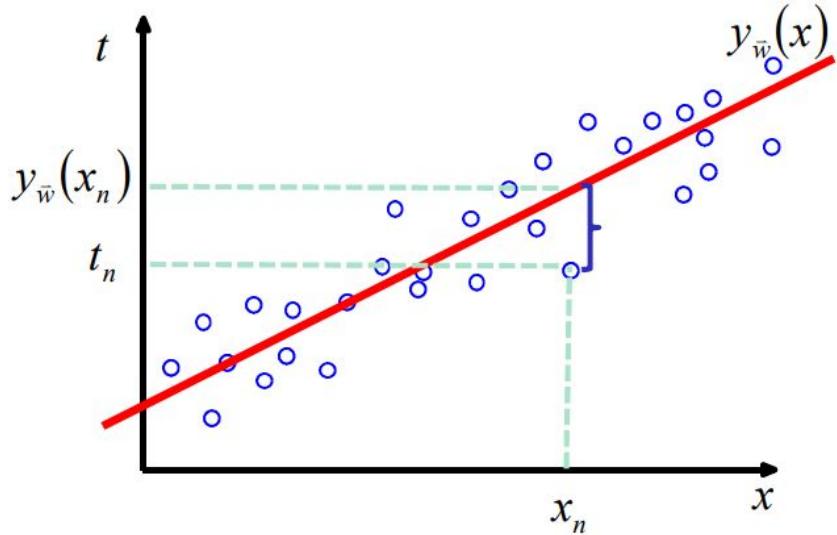
$$D = \{(x_1, t_1), (x_2, t_2), \dots, (x_N, t_N)\}$$

Idéalement, on souhaiterait trouver un modèle tel que  $y_{\bar{w}}(x_i) = t_i$



# Problème à résoudre

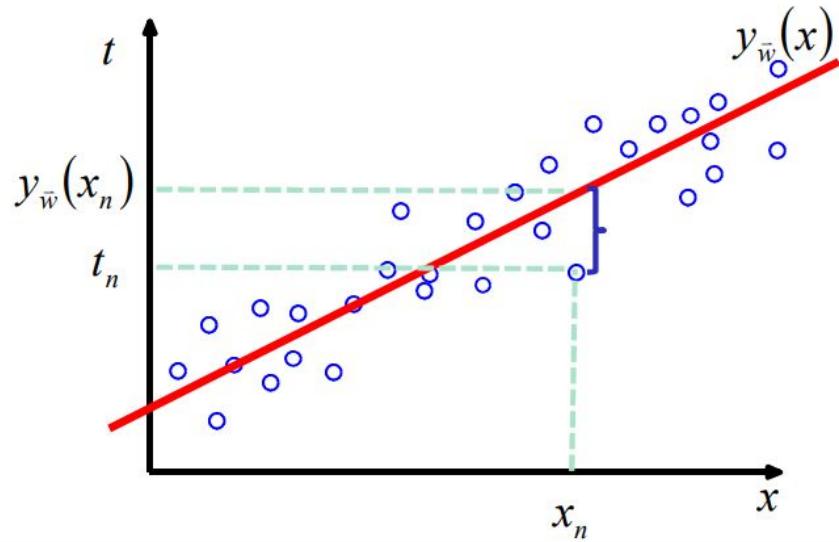
Malheureusement, dans la vraie vie, les données sont **bruitées**



Dans ce cas, le but est de trouver un modèle qui **fait le moins d'erreurs possible**.

# Problème à résoudre

$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (y_{\vec{w}}(x_n) - t_n)^2$$



## Problème à résoudre

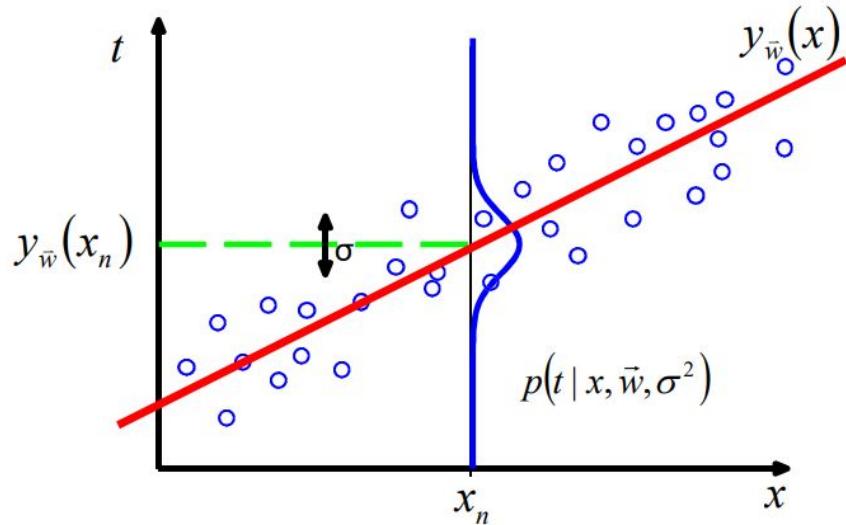
$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (y_{\vec{w}}(x_n) - t_n)^2$$

Il est très bien connu en technique d'apprentissage que cette solution est **optimale** lorsque le bruit est gaussien.



# Formulation probabiliste

Loi gaussienne conditionnelle



# Formulation probabiliste

Pour entraîner le modèle  $y_{\vec{w}}(x)$  nous passerons par une formulation probabiliste :

$$p(t | x, \vec{w}, \sigma^2) = N(t | y_{\vec{w}}(x), \sigma^2)$$

- Revient à supposer que les **cibles** sont des **versions bruitées** du vrai modèle

$$t_n = y_{\vec{w}}(x_n) + \varepsilon$$

↑  
Bruit gaussien de moyenne 0  
et de variance  $\sigma^2$

# Maximum de vraisemblance

Soit notre **ensemble d'entraînement**

$$D = (X, T)$$

où

$$X = \{\vec{x}_1, \dots, \vec{x}_N\} \text{ et } \vec{x}_i \in R^d$$

$$T = \{t_1, \dots, t_N\}$$

et la fonction de **probabilités dont les données sont issues**

$$p(T | X, \bar{w}, \sigma^2)$$

Le **maximum de vraisemblance** s'exprime comme

$$\bar{w} = \arg \max_{\bar{w}} p(T | X, \bar{w}, \sigma^2)$$

The equation  $\bar{w} = \arg \max_{\bar{w}} p(T | X, \bar{w}, \sigma^2)$  is shown. Two orange arrows point from the words 'Connue' and 'Inconnue' to the terms 'X' and ' $\bar{w}$ ' respectively in the equation.

Connue      Inconnue

# Maximum de vraisemblance

$$\begin{aligned}\vec{w} &= \arg \max_{\vec{w}} p(T | X, \vec{w}, \sigma^2) \\ &= \arg \max_{\vec{w}} p(t_1, \dots, t_N | \vec{x}_1, \dots, \vec{x}_N, \vec{w}, \sigma^2)\end{aligned}$$

En supposant que **les données sont i.i.d**

$$\begin{aligned}\vec{w} &= \arg \max_{\vec{w}} \prod_{n=1}^N p(t_n | \vec{x}_n, \vec{w}, \sigma^2) \\ &= \arg \max_{\vec{w}} \prod_{n=1}^N N(t_n | y_{\vec{w}}(\vec{x}_n), \sigma^2) \\ &= \arg \max_{\vec{w}} \prod_{n=1}^N \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_{\vec{w}}(\vec{x}_n)-t_n)^2}{2\sigma^2}}\end{aligned}$$

# Maximum de vraisemblance

$$\vec{w} = \arg \max_{\vec{w}} \ln \left( \prod_{n=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y_{\vec{w}}(\vec{x}_n) - t_n)^2}{2\sigma^2}} \right)$$

$$= \arg \max_{\vec{w}} \sum_{n=1}^N \ln \left( \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y_{\vec{w}}(\vec{x}_n) - t_n)^2}{2\sigma^2}} \right)$$

$$= \arg \max_{\vec{w}} N \ln \left( \cancel{\frac{1}{\sqrt{2\pi}\sigma}} \right) + \sum_{n=1}^N \ln \left( e^{-\frac{(y_{\vec{w}}(\vec{x}_n) - t_n)^2}{2\sigma^2}} \right)$$

Indépendant de  $\vec{w}$

$$= \arg \max_{\vec{w}} \sum_{n=1}^N -\frac{(y_{\vec{w}}(\vec{x}_n) - t_n)^2}{2\sigma^2}$$

# Maximum de vraisemblance

$$\vec{w} = \arg \max_{\vec{w}} \sum_{n=1}^N -\frac{(y_{\vec{w}}(\vec{x}_n) - t_n)^2}{2\sigma^2}$$

$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (y_{\vec{w}}(\vec{x}_n) - t_n)^2$$

Et puisque  $y_{\vec{w}}(\vec{x}) = \vec{w}^T \vec{x}$  (voir quelques pages précédentes)

$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (\vec{w}^T \vec{x}_n - t_n)^2$$

# Maximum de vraisemblance

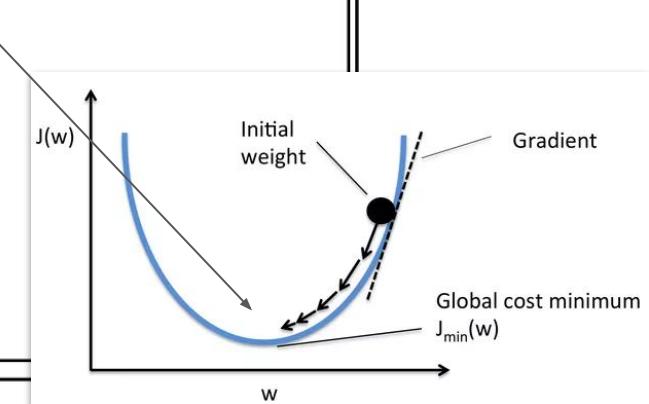
$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (\vec{w}^T \vec{x}_n - t_n)^2$$

$E_D(\vec{w})$

Le « meilleur »  $\vec{w}$  est celui pour lequel le **gradient est nul**

$$\nabla_{\vec{w}} E_D(\vec{w}) = \sum_{n=1}^N (\vec{w}^T \vec{x}_n - t_n) \vec{x}_n^T = 0$$

$$\vec{w}^T \sum_{n=1}^N \vec{x}_n \vec{x}_n^T - \sum_{n=1}^N t_n \vec{x}_n^T = 0$$



# Maximum de vraisemblance

$$\vec{w}^T \sum_{n=1}^N \vec{x}_n \vec{x}_n^T - \sum_{n=1}^N t_n \vec{x}_n^T = 0$$

En isolant  $\vec{w}$ , on obtient que

$$\vec{w}_{\text{MV}} = (X^T X)^{-1} X^T T$$

où

$$X = \begin{pmatrix} 1 & x_{1,1} & \cdots & x_{1,d} \\ 1 & x_{2,1} & \cdots & x_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N,1} & \cdots & x_{N,d} \end{pmatrix} \quad T = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{pmatrix}$$

# Maximum *a posteriori* (MAP)

Cherche les meilleurs paramètres  $\vec{w}$  en **maximisant la probabilité a posteriori**

Inconnue

Connues

$$\vec{w} = \arg \max_{\vec{w}} p(\vec{w} | X, T, \sigma^2)$$

$$= \arg \max_{\vec{w}} \frac{p(T | X, \vec{w}, \sigma^2) p(\vec{w})}{P(X, T, \sigma^2)}$$

=> Par le théorème de Bayes

Constante par rapport à

$$= \arg \max_{\vec{w}} p(T | X, \vec{w}, \sigma^2) p(\vec{w})$$

MODIFIED BAYES' THEOREM:

$$P(H|x) = P(H) \times \left( 1 + P(C) \times \left( \frac{P(x|H)}{P(x)} - 1 \right) \right)$$

H: HYPOTHESIS

x: OBSERVATION

P(H): PRIOR PROBABILITY THAT H IS TRUE

P(x): PRIOR PROBABILITY OF OBSERVING x

P(C): PROBABILITY THAT YOU'RE USING  
BAYESIAN STATISTICS CORRECTLY

# Maximum *a posteriori* (MAP)

On va émettre l'hypothèque que les données  $X, T$  ainsi que les paramètres  $\vec{w}$  sont iid de **distributions gaussiennes**

$$\begin{aligned}\vec{w} &= \arg \max_{\vec{w}} p(T | X, \vec{w}, \sigma^2) p(\vec{w}) \\ &= \arg \max_{\vec{w}} \prod_{n=1}^N N(t_n | y_{\vec{w}}(\vec{x}_n), \vec{w}, \sigma^2) N(\vec{w} | 0, \alpha^2)\end{aligned}$$

Moyenne nulle

$$N(t_n | y_{\vec{w}}(\vec{x}_n), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y_{\vec{w}}(\vec{x}_n) - t_n)^2}{2\sigma^2}}$$
$$N(\vec{w} | 0, \Sigma) = \frac{1}{(2\pi)^{1/d} |\Sigma|} e^{-\frac{\vec{w}^T \Sigma^{-1} \vec{w}}{2}}$$

# Maximum *a posteriori* (MAP)

Cherche les meilleurs paramètres  $\vec{w}$  en **maximisant la probabilité *a posteriori***

$$\begin{aligned}\vec{w} &= \arg \max_{\vec{w}} \ln \left[ \prod_{n=1}^N N(t_n | y_{\vec{w}}(x_n), \sigma^2) N(\vec{w} | 0, \Sigma) \right] \\ &= \arg \max_{\vec{w}} \sum_{n=1}^N \ln [N(t_n | y_{\vec{w}}(x_n), \sigma^2) N(\vec{w} | 0, \Sigma)] \\ &= \arg \max_{\vec{w}} \sum_{n=1}^N \ln \left[ \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n - y_{\vec{w}}(x_n))^2}{2\sigma^2}} \right] + \ln \left[ \frac{1}{(2\pi)^{1/M} |\Sigma|} e^{-\frac{\vec{w}^T \Sigma^{-1} \vec{w}}{2}} \right] \\ &= \arg \max_{\vec{w}} \sum_{n=1}^N -\frac{(t_n - y_{\vec{w}}(x_n))^2}{2\sigma^2} - \frac{\vec{w}^T \Sigma^{-1} \vec{w}}{2} + \ln \left[ \frac{1}{\sqrt{2\pi}\sigma} \right] + \ln \left[ \frac{1}{(2\pi)^{1/M} |\Sigma|} \right]\end{aligned}$$

# Maximum *a posteriori* (MAP)

Cherche les meilleurs paramètres  $\vec{w}$  en **maximisant la probabilité a posteriori**

$$\vec{w} = \arg \max_{\vec{w}} \sum_{n=1}^N -\frac{(t_n - y_{\vec{w}}(x_n))^2}{2\sigma^2} - \frac{\vec{w}^T \Sigma^{-1} \vec{w}}{2} + \ln \left[ \frac{1}{\sqrt{2\pi}\sigma} \right] + \ln \left[ \frac{1}{(2\pi)^{M/2} |\Sigma|} \right]$$

**Constante par rapport à  $\vec{w}$**

De plus, comme on ne connaît généralement pas  $\Sigma$ , on suppose qu'elle est isotropique

$$\Sigma = \begin{pmatrix} \alpha^2 & 0 & \dots & 0 \\ 0 & \alpha^2 & & \vdots \\ \vdots & & \ddots & \\ 0 & 0 & \dots & \alpha^2 \end{pmatrix} = \alpha^2 I$$

# Maximum *a posteriori* (MAP)

Cherche les meilleurs paramètres  $\vec{w}$  en **maximisant la probabilité *a posteriori***

$$\begin{aligned}\vec{w} &= \arg \max_{\vec{w}} \sum_{n=1}^N -\frac{(t_n - y_{\vec{w}}(x_n))^2}{\sigma^2} - \frac{\vec{w}^T \Sigma^{-1} \vec{w}}{\alpha^2} \\ &= \arg \max_{\vec{w}} \sum_{n=1}^N -\frac{(t_n - y_{\vec{w}}(x_n))^2}{\sigma^2} - \frac{\vec{w}^T \vec{w}}{\alpha^2} \\ &= \arg \min_{\vec{w}} \sum_{n=1}^N (t_n - y_{\vec{w}}(x_n))^2 + \lambda \vec{w}^T \vec{w}\end{aligned}$$

où  $\lambda = \frac{\sigma^2}{\alpha^2}$

# Maximum *a posteriori* (MAP)

Cherche les meilleurs paramètres  $\vec{w}$  en **maximisant la probabilité a posteriori**

$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (t_n - y_{\vec{w}}(x_n))^2 + \lambda \vec{w}^T \vec{w}$$

**NOTE**

Formule également connue sous le nom de  
« **régression de Ridge** »

Voir sklearn pour une implémentation simple

[scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.Ridge.html](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html)

## Maximum *a posteriori* (MAP)

$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (t_n - y_{\vec{w}}(x_n))^2 + \lambda \vec{w}^T \vec{w}$$

$E_D(\vec{w})$

Les meilleurs paramètres sont ceux qui correspondent au gradient nul

$$\nabla_{\vec{w}} E_D(\vec{w}) = 0$$

## Maximum *a posteriori* (MAP)

Puisque  $y_{\vec{w}}(\vec{x}) = \vec{w}^T \vec{x}$  (voir quelques pages précédentes)

$$E_D(\vec{w}) = \sum_{n=1}^N (t_n - \vec{w}^T \vec{x})^2 + \lambda \vec{w}^T \vec{w}$$

En forçant le **gradient à zéro**  $\nabla E_D(W) = 0$  on peut démontrer que

$$W_{\text{MAP}} = (X^T X + \lambda I)^{-1} X^T T$$

~~Cette preuve est sujette à devoir...~~

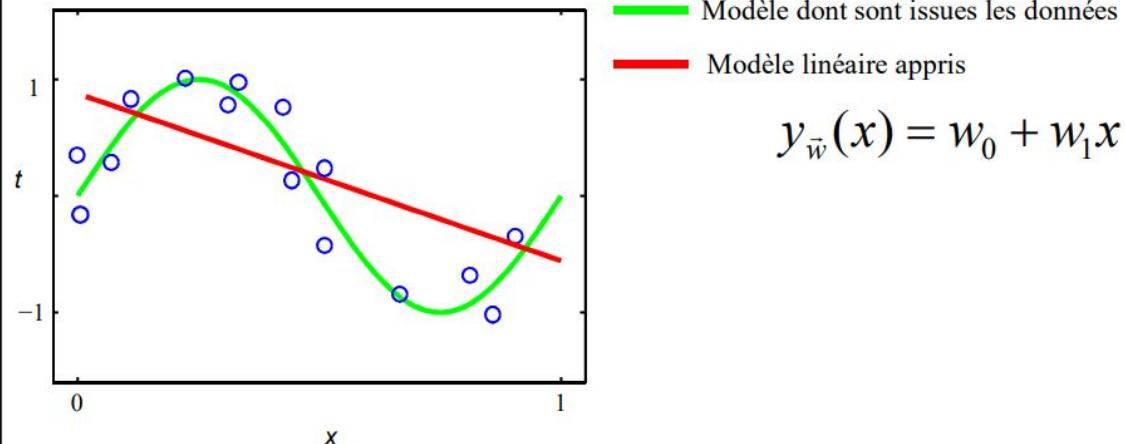
## Maximum *a posteriori* (MAP)

$$W_{\text{MAP}} = (X^T X + \lambda I)^{-1} X^T T$$

- Le terme de régularisation  $\lambda \frac{W^T W}{2}$  est souvent appelé ***weight decay***
- La régression avec un ***weight decay*** est souvent appelé ***régression de Ridge***
- On retrouve le **maximum de vraisemblance** lorsque  $\lambda = 0$
- Permet de réduire le **sur-apprentissage** lorsque  $\lambda > 0$

Allons coder !

# Exemple sous-apprentissage

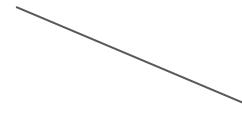


# Fonctions de base

**Solution:** on va projeter les données dans un **espace plus grand**, là où les données sont **distribuées linéairement**.

=> régression sur des données  $M$  dimensions au lieu de  $d$  dimensions ( $M > d$ )

$$\phi : R \rightarrow R^M$$



Méthodes à noyaux

# Fonctions de base

De façon plus générale

$$y_{\vec{w}}(\vec{x}) = w_0 + \sum_{i=1}^M w_i \phi_i(\vec{x})$$

où les  $\phi_i(\vec{x})$  sont des **fonctions de base** (*basis functions*)

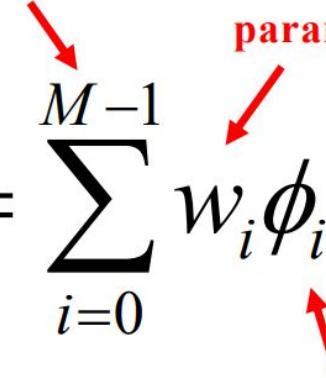
- Cas particulier :  $\phi_i(\vec{x}) = x_i$  et  $M = d + 1$

# Fonctions de base

Pour simplifier la notation, on va supposer que  $\phi_0(\vec{x})=1$  afin d'inclure le **biais** dans la sommation

$$y_{\vec{w}}(\vec{x}) = \sum_{i=0}^{M-1} w_i \phi_i(\vec{x})$$

**hyperparamètre**  
**paramètre**  
**Fonction de base**



# Fonctions de base

Une des fonctions de base les plus fréquentes est la **fonction polynomiale**

$$\phi_i(x) = x^i$$

=> **Régression polynomiale**

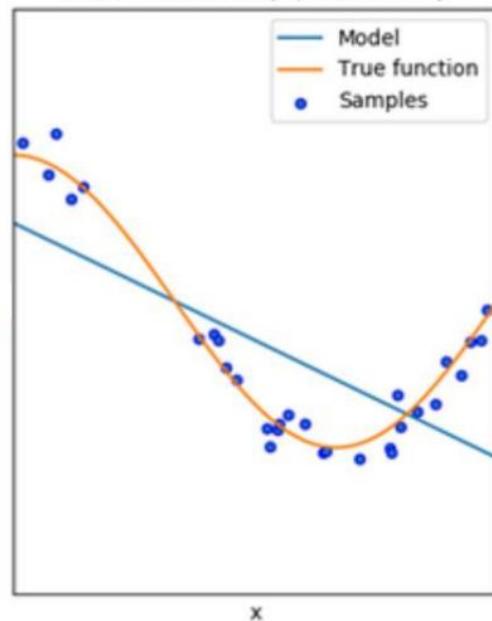
# Fonctions de base

**Exemple:** au lieu de faire une régression linéaire 1D,  
=> faire une régression linéaire en 4D

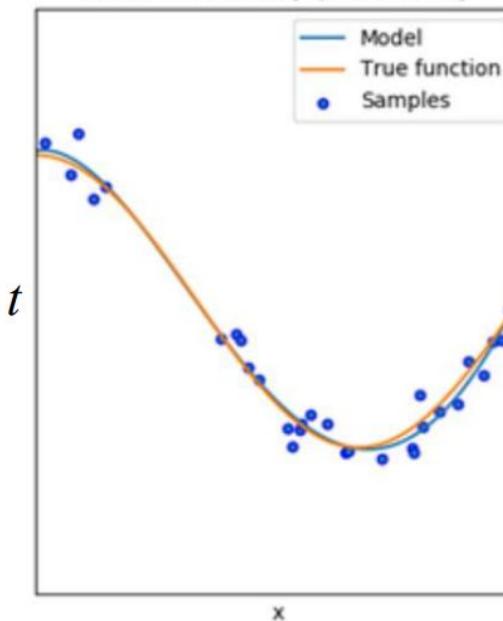
$$\phi(x) \rightarrow (x, x^2, x^3, x^4)$$

$$y_{\vec{w}}(x) = w_0 + w_1 x \quad \begin{array}{c} \text{orange arrow} \\ \rightarrow \end{array} \quad y_{\vec{w}}(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 \\ = w_0 + \sum_{i=1}^4 w_i \phi_i(x)$$

Degree 1  
MSE = 4.08e-01(+/- 4.25e-01)



Degree 4  
MSE = 4.32e-02(+/- 7.08e-02)



$$y_{\vec{w}}(x) = w_0 + w_1 x$$

$$y_{\vec{w}}(x) = w_0 + \sum_{i=1}^4 w_i \phi_i(x)$$

Source : [stats.stackexchange.com/questions/305619/artificial-neural-networks-equivalent-to-linear-regression-with-polynomial-featu](https://stats.stackexchange.com/questions/305619/artificial-neural-networks-equivalent-to-linear-regression-with-polynomial-featu)

# Maximum de vraisemblance

$$\vec{w}^T \sum_{n=1}^N \vec{\phi}(\vec{x}_n) \vec{\phi}(\vec{x}_n)^T - \sum_{n=1}^N t_n \vec{\phi}(\vec{x}_n)^T = 0$$

En **isolant**  $\vec{w}$ , on obtient que

$$\vec{w}_{\text{MV}} = (\Phi^T \Phi)^{-1} \Phi^T T$$

où

$$\Phi = \begin{pmatrix} \phi_0(\vec{x}_1) & \phi_1(\vec{x}_1) & \cdots & \phi_{M-1}(\vec{x}_1) \\ \phi_0(\vec{x}_2) & \phi_1(\vec{x}_2) & \cdots & \phi_{M-1}(\vec{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\vec{x}_N) & \phi_1(\vec{x}_N) & \cdots & \phi_{M-1}(\vec{x}_N) \end{pmatrix} \quad T = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{pmatrix}$$

# Maximum *a posteriori* (MAP)

Ici aussi on obtiens la solution optimale en forçant le **gradient à zéro**

$$\nabla E_D(\vec{w}) = 0$$

Et ainsi obtenir

$$\vec{w}_{\text{MAP}} = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T T$$

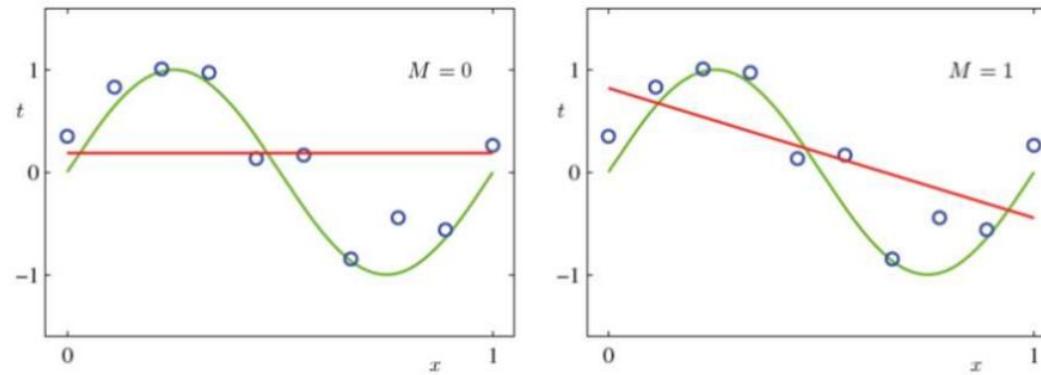
Cette preuve est sujette à devoir...

# Sur/Sous-apprentissage

# Sur- et sous-apprentissage

➤ Comment trouver le bon  $M$ ?

Un petit  $M$  donne un modèle trop simple qui cause du **sous-apprentissage**



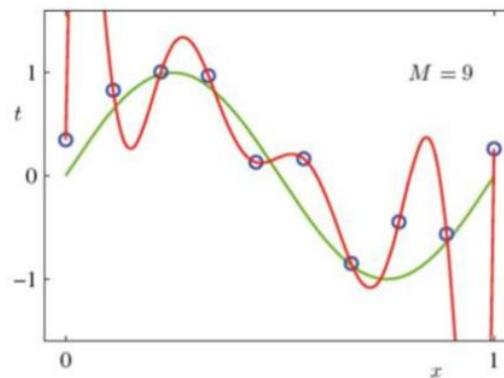
$E_D(\vec{w}) \Rightarrow$  élevée

$E_{D_{test}}(\vec{w}) \Rightarrow$  élevée

# Sur- et sous-apprentissage

- Comment trouver le bon  $M$ ?

Un grand  $M$  donne un modèle qui « apprend par cœur » les données d'apprentissage ce qui cause du **sur-apprentissage**



$E_D(\vec{w}) \Rightarrow$  TRÈS faible

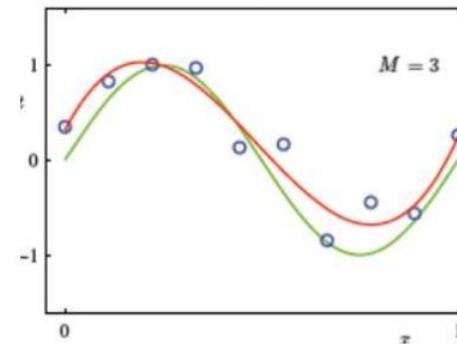
$E_{D_{test}}(\vec{w}) \Rightarrow$  élevée

# Sur- et sous-apprentissage

- Comment trouver le bon  $M$ ?

Idéalement, il faudrait une valeur intermédiaire de sorte que **l'erreur d'entraînement et de test soient faibles.**

Trouver cette meilleure valeur de  $M$  s'appelle de la **sélection de modèle**



$$E_D(\vec{w}) \Rightarrow \text{faible}$$

$$E_{D_{test}}(\vec{w}) \Rightarrow \text{faible}$$

# Sur- et sous-apprentissage

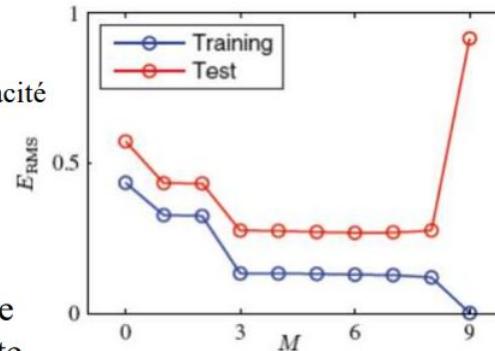
## Capacité d'un modèle

- ✓ aptitude d'un modèle à apprendre «par cœur»
- ✓ exemple : plus  $M$  est grand, plus le modèle a de capacité

Plus la capacité est grande, plus la différence entre l'erreur d'entraînement et l'erreur de test augmente

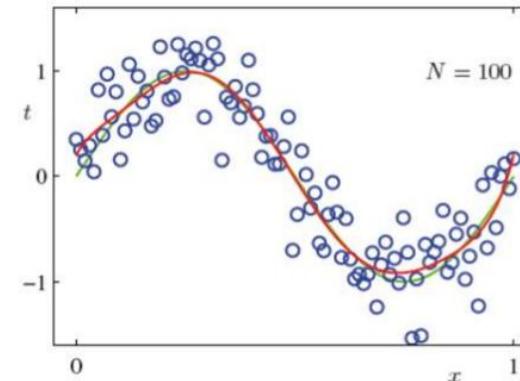
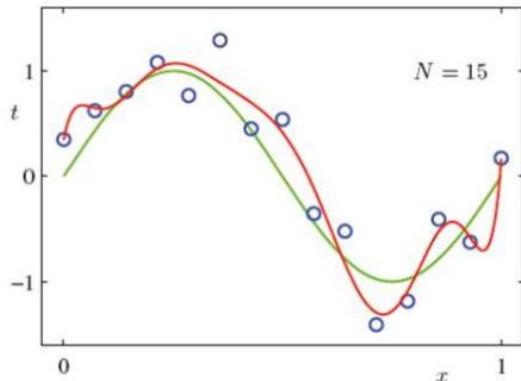
- ✓ en régression, l'erreur sur tout un ensemble est souvent mesurée par la racine de la moyenne des erreurs au carré (*root-mean-square error*)

$$E_{RMS} = \sqrt{\frac{2E(\vec{w})}{N}}$$



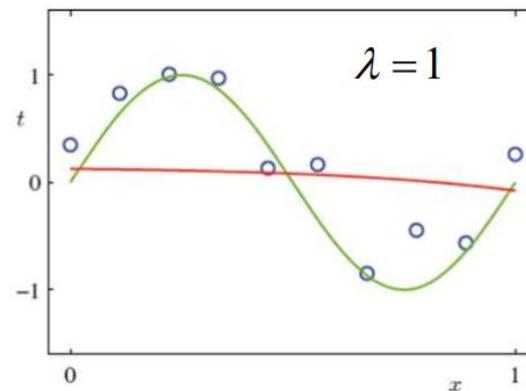
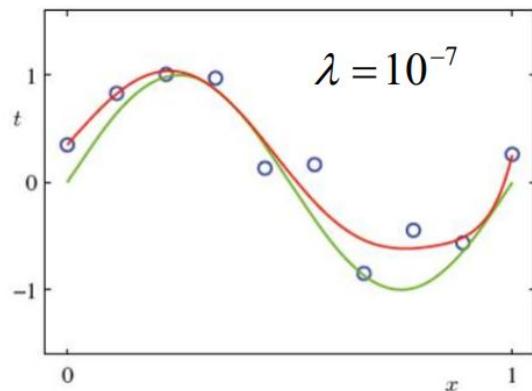
# Généralisation

Plus la quantité de données d'entraînement augmente,  
plus le modèle entraîné va bien généraliser



# Régularisation

Forte régularisation = modèle moins flexible

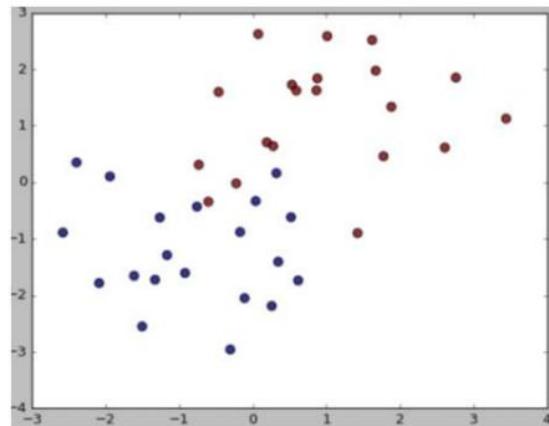


Allons coder !

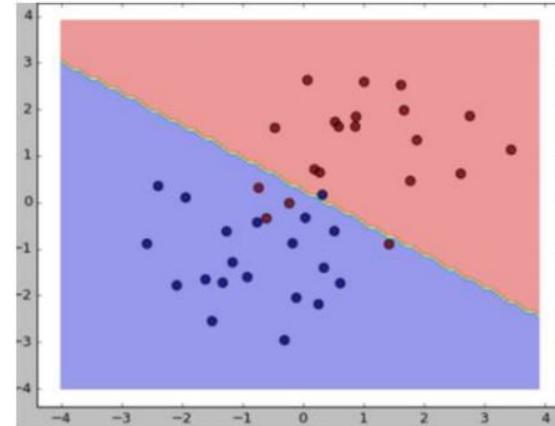
# Classification

# Classification supervisée (illustrée)

Entraînement



Soient des données de 2 classes ● et ●  
(ici dans un espace 2D)



Le but est de trouver une fonction  $y(\vec{x})$  telle que  
 $y(●) = \text{class 1}$   
 $y(●) = \text{class 2}$

# Notation

**Ensemble d'entraînement:**  $D = \{(\vec{x}_1, t_1), (\vec{x}_2, t_2), \dots, (\vec{x}_N, t_N)\}$

$\vec{x}_n \in \Re^d$  vecteur de données du n-ème élément

$t_n \in \{c_1, c_2, \dots, c_K\}$  étiquette de classe du i-ème élément

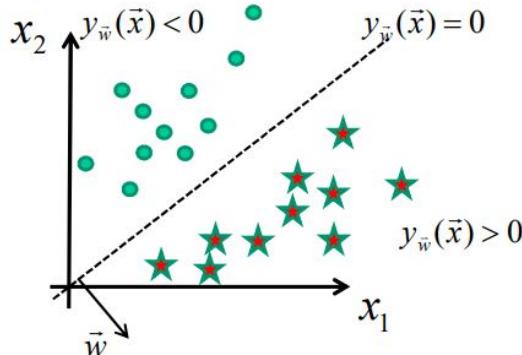
**Fonctions:** avec  $D$ , on doit *apprendre* une **fonction de classification**

$$y : \Re^d \rightarrow \{c_1, c_2, \dots, c_k\}$$

qui nous informe à quelle classe appartient le vecteur  $\vec{x}$ .

# Séparation linéaire

(2D et 2 classes)



$$\begin{aligned}y_{\vec{w}}(\vec{x}) &= w_0 + w_1 x_1 + w_2 x_2 \\&= w_0 + \vec{w}^T \vec{x} \\&= \vec{w}'^T \vec{x}'\end{aligned}$$

$$y_{\vec{w}}(\vec{x}) = \vec{w}^T \vec{x}$$

Par simplicité

2 grands **avantages**. Une fois l'entraînement terminé,

1. Plus besoin de données d'entraînement
2. Classification est très rapide (**produit scalaire** entre 2 vecteurs)

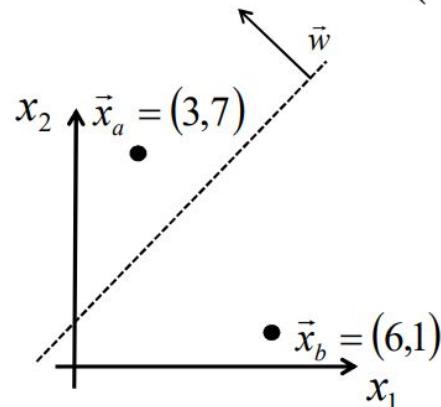
# Séparation linéaire

Exemple jouet

biais  
poids

$$\vec{w}^T = (2.2, -5.5, 4.4)$$

$\vec{x}_a$  est en FACE du plan



$$y_W(\vec{x}_a) = \vec{w}^T \vec{x}_a = (2.2, -5.5, 4.4) \begin{bmatrix} 1 \\ 3 \\ 7 \end{bmatrix} = 16.5$$

$$y_W(\vec{x}_b) = \vec{w}^T \vec{x}_b = (2.2, -5.5, 4.4) \begin{bmatrix} 1 \\ 6 \\ 1 \end{bmatrix} = -26.4$$

$\vec{x}_b$  est DERRIÈRE le plan

# Régression par les moindres carrés

Cas 2 classes

On peut **classifier des données** en utilisant une approche de **régression** comme celle vue au chapitre précédent.

- On pourrait **prédire directement** la valeur de la cible ( $t=1.0$  vs  $t=-1.0$ )
- Si  $y_{\vec{w}}(\vec{x}) \geq 0$  on classifie dans *Classe1* sinon dans *Classe2*

# Régression par les moindres carrés

Cas 2 classes

**RAPPEL**

Maximum de vraisemblance

$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (t_n - y_{\vec{w}}(\vec{x}_n))^2$$

$$\vec{w}_{\text{MV}} = (X^T X)^{-1} X^T T$$

Maximum *a posteriori*

$$\vec{w} = \arg \min_{\vec{w}} \sum_{n=1}^N (t_n - y_{\vec{w}}(\vec{x}_n))^2 + \lambda \vec{w}^T \vec{w}$$

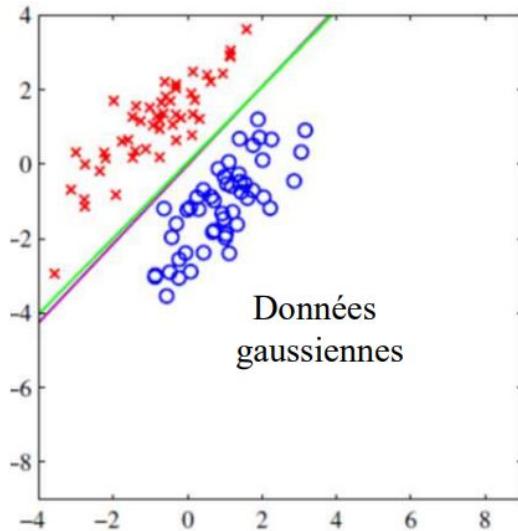
$$\vec{w}_{\text{MAP}} = (X^T X + \lambda I)^{-1} X^T T$$



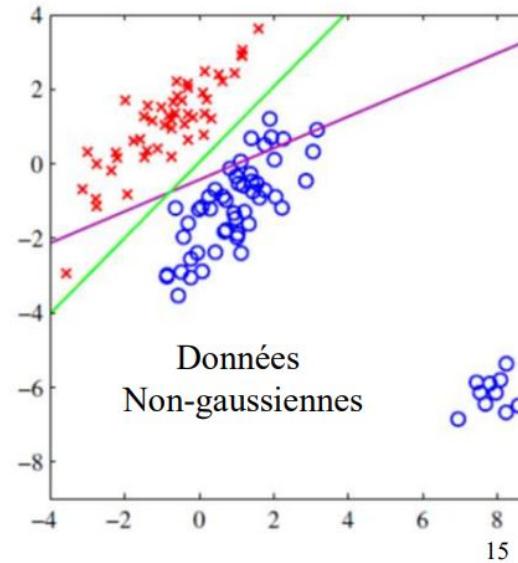
Ces fonctions de coût s'appuient sur  
l'hypothèse de données gaussiennes

# Régression par moindres carrés

- Régression logistique (à voir plus loin)
- Moindres carrés (maximum de vraisemblance)



Données  
gaussiennes



Données  
Non-gaussiennes

# Régression par les moindres carrés

Cas K>2 classes

On va traiter le cas K classes comme une **régression multiple**

- **Cible** : vecteur à K dim. indiquant à quelle classe appartient l'entrée
- **Exemple** : Pour K=5 classes et un entrée associée à la classe 2

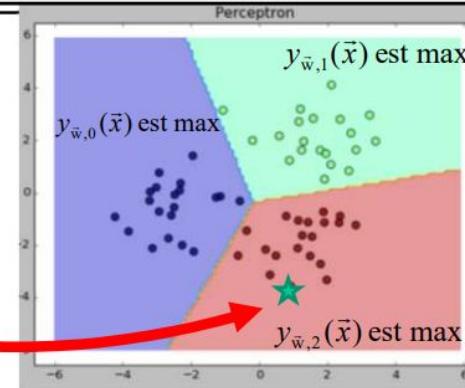
$$t_n = \begin{pmatrix} -1 & 1 & -1 & -1 & -1 \end{pmatrix}^T$$

- **Classification**: On classifie dans la classe k une donnée dont la valeur de  $y_{\vec{w},k}(\vec{x})$  est la plus élevée.

# Cas K=3 classes

Example

★ (1.1, -2.0)



$$y_{\vec{w}}(\vec{x}) = \vec{w}^T \vec{x} = \begin{bmatrix} -.2 & .05 & -.36 \\ -.4 & .41 & .24 \\ -.6 & -.49 & .40 \end{bmatrix} \begin{bmatrix} 1 \\ -2.0 \\ 1.1 \end{bmatrix} = \begin{bmatrix} -.7 \\ -1 \\ .8 \end{bmatrix} \begin{array}{l} \text{Classe 0} \\ \text{Classe 1} \\ \text{Classe 2} \end{array}$$

Allons coder !

# Perceptron (2 classes)

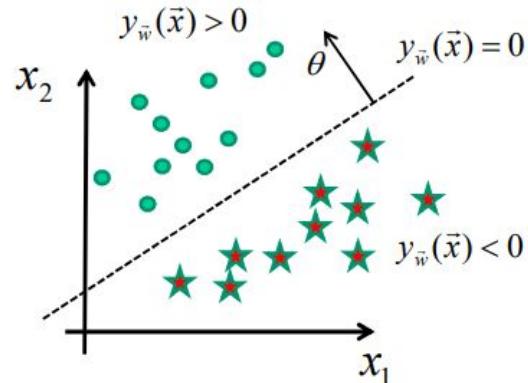
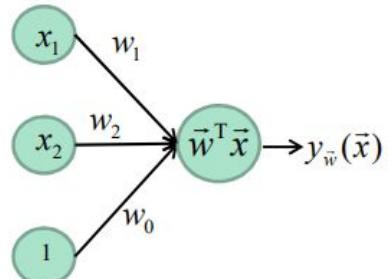
Contrairement aux approches précédentes, le Perceptron **n'émet pas** l'hypothèse que les données sont **gaussiennes**

Le Perceptron part de la définition brute de la classification binaire par **hyperplan**

$$\begin{aligned}y_{\vec{w}}(\vec{x}) &= \text{sign}(\vec{w}^T \vec{x}) \\&= \text{sign}(w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d)\end{aligned}$$


# Perceptron

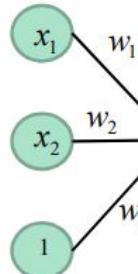
(2D et 2 classes)



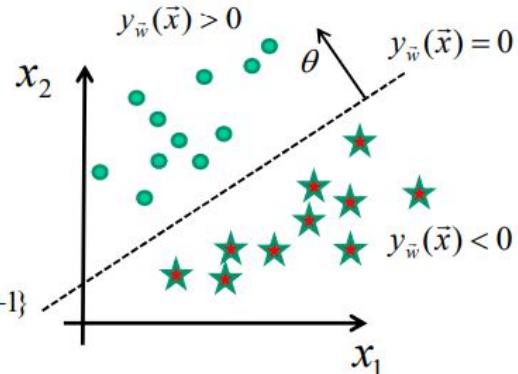
$$\begin{aligned}y_{\vec{w}}(\vec{x}) &= w_0 + w_1 x_1 + w_2 x_2 \\&= \vec{w}^T \vec{x} \\&= \vec{w}'^T \vec{x}' \\&\Rightarrow \vec{w}^T \vec{x}\end{aligned}$$

# Perceptron

(2D et 2 classes)



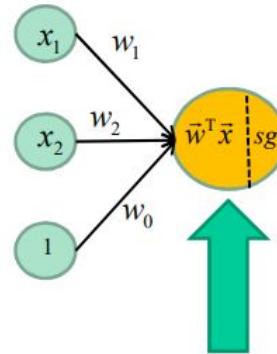
Fonction d'activation



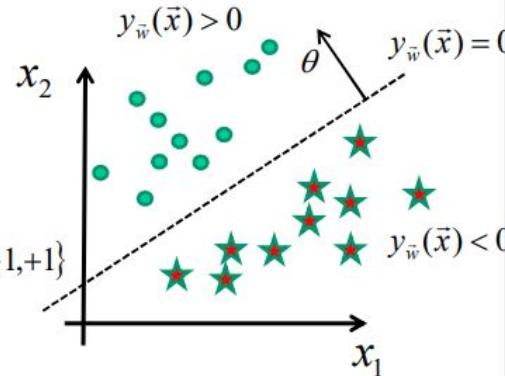
$$y_{\vec{w}}(\vec{x}) = \text{sign}(\vec{w}^T \vec{x})$$

# Perceptron

(2D et 2 classes)

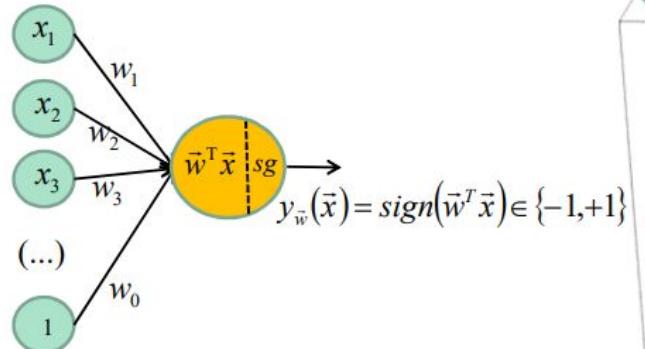


**Neurone**  
Produit scalaire + fonction d'activation

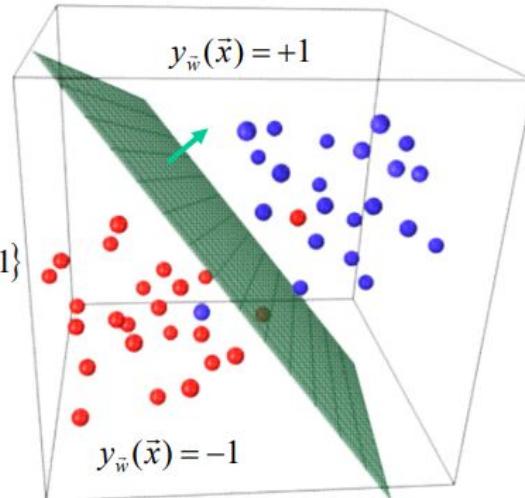


# Perceptron

(N-D and 2-class case)



Example 3D



## Nouvelle fonction de coût pour apprendre $W$

**Le but:** avec des données d'entraînement  $D = \{(\vec{x}_1, t_1), (\vec{x}_2, t_2), \dots, (\vec{x}_N, t_N)\}$ , estimer  $w$  afin que:

$$y_{\vec{w}}(\vec{x}_n) = t_n \quad \forall n$$

En d'autres mots, minimiser **l'erreur d'entraînement**

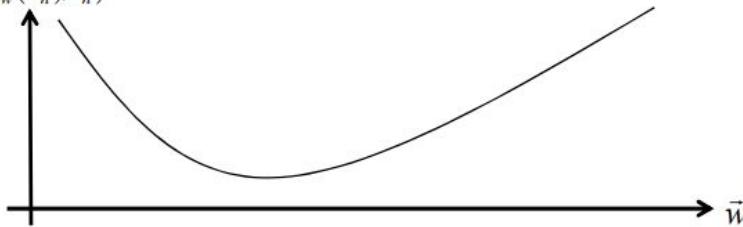
$$E_D(\vec{w}) = \sum_{n=1}^N l(y_{\vec{w}}(\vec{x}_n), t_n)$$

où  $l(.,.)$  est une **fonction de perte** (*loss function* en anglais).

Trouver la bonne fonction de perte et le bon algorithme **d'optimisation** et un sujet central en **apprentissage machine**.

## Nouvelle fonction de coût pour apprendre $W$

$$E_D(\vec{w}) = \sum_{n=1}^N l(y_{\vec{w}}(\vec{x}_n), t_n)$$



Comme nous l'avons vu auparavant, les algorithmes d'apprentissage sont des **problèmes d'optimisation** qu'on peut formuler ainsi:

$$\vec{w} = \arg \min_{\vec{w}} = \underbrace{\sum_{n=1}^N l(y_{\vec{w}}(\vec{x}_n), t_n)}_{E_D(\vec{w})}$$

En général, on cherche une fonction de coût :

- qui a **un seul minima**
- qui est *smooth* et qui est **dérivable en tout point**
- solution **optimale** à  $\frac{dE_D(\vec{w})}{d\vec{w}} = 0$

# Critère du Perceptron

## Observation

Une donnée est **mal classée** quand  $\vec{w}^T \vec{x}_n > 0$  et  $t_n = -1$  ou quand  $\vec{w}^T \vec{x}_n < 0$  et  $t_n = +1$ .

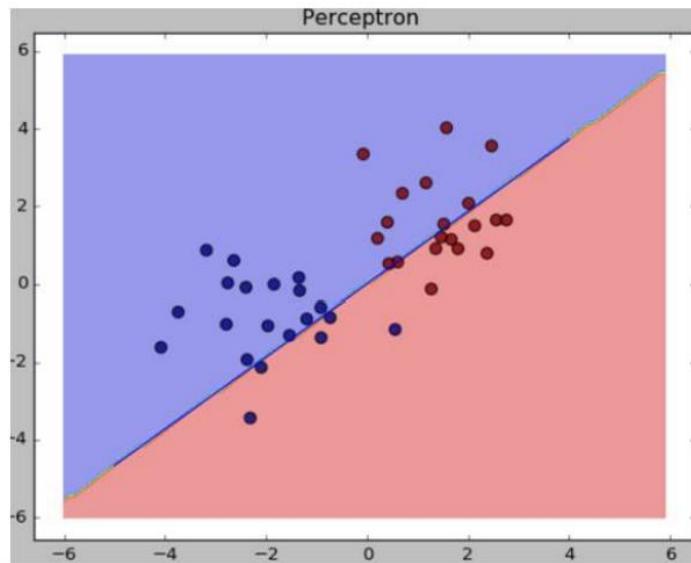
Par conséquent

$-\vec{w}^T \vec{x}_n t_n$  est **toujours positif pour les données mal classés**

# Critère du Perceptron

Le **critère du Perception** est une fonction qui pénalise les données mal classées

$$E_D(\vec{w}) = \sum_{\vec{x}_n \in M} -\vec{w}^T \vec{x}_n t_n \quad \text{où } M \text{ est l'ensemble des données mal classées}$$

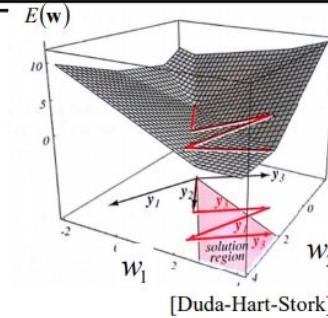


$$E_D(\vec{w}) = 464.15$$

# Perceptron

Pour le critère du Perceptron

$$\nabla E_D(\vec{w}) = \sum_{\vec{x}_n \in M} -t_n \vec{x}_n$$



[Duda-Hart-Stork]

*Batch optimization*

Initialiser  $\vec{w}$

k=0

DO k=k+1

$$\vec{w} = \vec{w} - \eta \left( \sum_{\vec{x}_n \in M} -t_n \vec{x}_n \right)$$

UNTIL toutes les données sont bien classées

NOTE importante sur le **taux d'apprentissage  $\eta$** :

- **Trop faible** => convergence lente
- **Trop grand** => peut ne pas converger (et même diverger)
- Peut **décroître** à chaque itération (e.g.  $\eta^{[k]} = cst/k$ )

# Perceptron

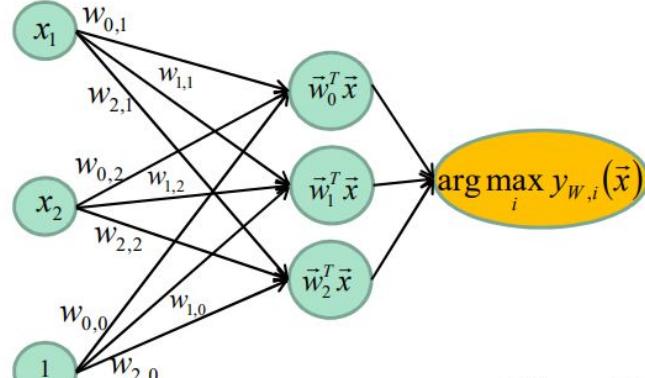
Une autre version de l'algorithme consiste à analyser une donnée par itération.

## Descente de gradient stochastique

```
Initialiser  $\vec{w}$ 
k=0
DO k=k+1
    FOR n = 1 to N
        IF  $\vec{w}^T \vec{x}_n t_n < 0$  THEN /* donnée mal classée */
             $\vec{w} = \vec{w} + \eta t_n \vec{x}_n$ 
UNTIL toutes les données sont bien classées.
```

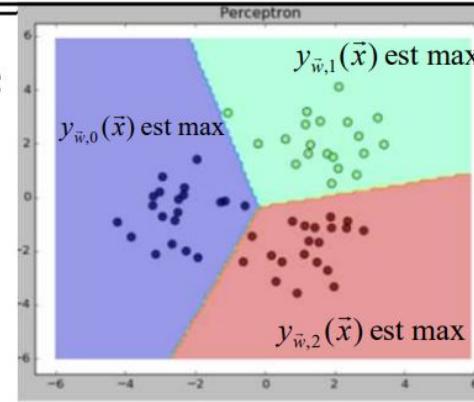
# Perceptron Multiclasse

(2D et 3 classes)



$$y_W(\vec{x}) = W^T \vec{x}$$

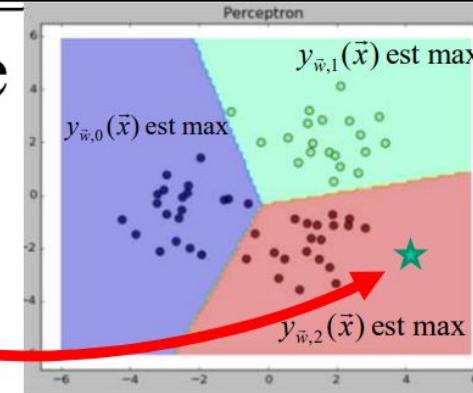
$$y_W(\vec{x}) = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$



# Perceptron Multiclasse

Exemple

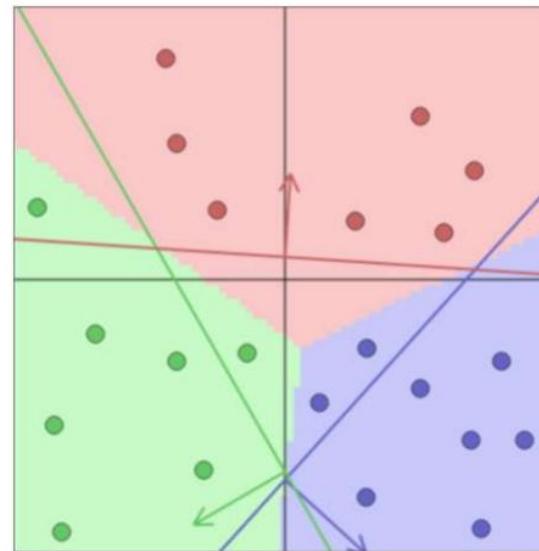
★ (1.1, -2.0)



$$y_w(\vec{x}) = \begin{bmatrix} -2 & -3.6 & 0.5 \\ -4 & 2.4 & 4.1 \\ -6 & 4 & -4.9 \end{bmatrix} \begin{bmatrix} 1 \\ 1.1 \\ -2 \end{bmatrix} = \begin{bmatrix} -6.9 \\ -9.6 \\ 8.2 \end{bmatrix} \begin{array}{l} \text{Classe 0} \\ \text{Classe 1} \\ \text{Classe 2} \end{array}$$

$$y_W(\vec{x}) = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

Tel qu'illustre ici, chaque **ligne de la matrice W** contient les paramètres (**normale + biais**) du **plan de séparation** linéaire de chaque classe.



# Perceptron

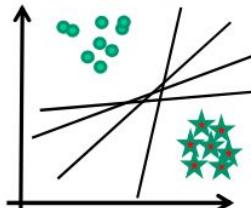
## Advantages:

- Très simple
- Ne suppose pas que les données sont **gaussiennes**.
- Si les données sont linéairement séparables, le Perceptron est **garantie(!) de converger** en un nombre fini d'iterations (see Duda-Hart-Stork pour la preuve)

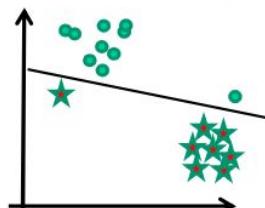
## Limitations:

- Gradient nul pour plusieurs solutions => plusieurs solutions “parfaites”
- Les données doivent être **linéairement séparables!**

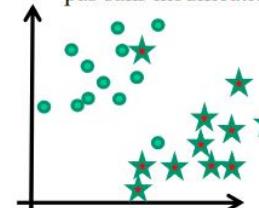
Plusieurs solutions  
“optimales”



Solution sous-optimale



Ne fonctionne  
pas sans modification

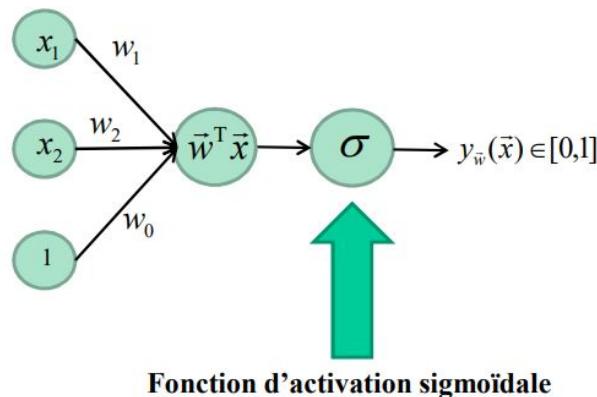


Allons coder !

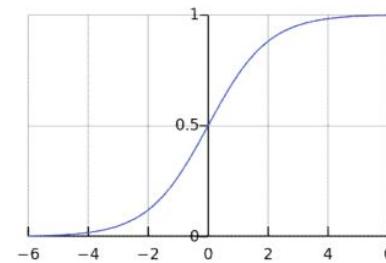
# Amélioration du Perceptron

(2D, 2 classes)

Nouvelle fonction d'activation : **sigmoïde logistique**



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

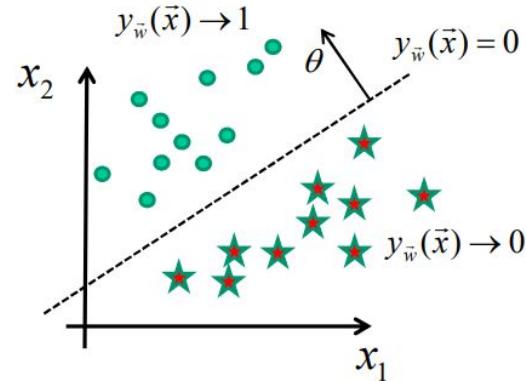
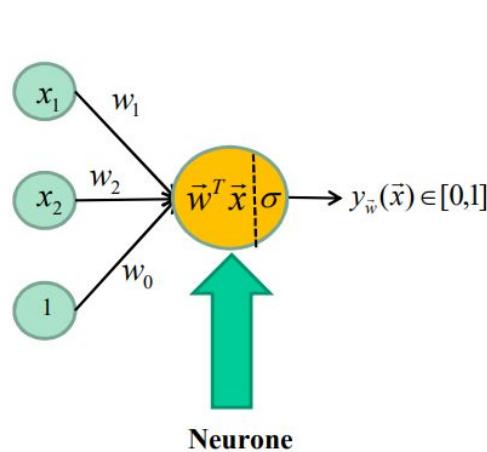


$$y_{\vec{w}}(\vec{x}) = \sigma(\vec{w}^T \vec{x}) = \frac{1}{1 + e^{-\vec{w}^T \vec{x}}}$$

# Amélioration du Perceptron

(2D, 2 classes)

Nouvelle fonction d'activation : **sigmoïde logistique**



$$y_{\vec{w}}(\vec{x}) = \sigma(\vec{w}^T \vec{x})$$

# Amélioration du Perceptron

(N-D, 2 classes)

Avec une sigmoïde, on peut **simuler une probabilité conditionnelle** sur  $c_1$  étant donné  $\vec{x}$

$$y_{\vec{w}}(\vec{x}) = \sigma(\vec{w}^T \vec{x}) \Rightarrow P(c_1 | \vec{x})$$

**Preuve:**

$$P(c_1 | \vec{x}) = \frac{P(\vec{x} | c_1)P(c_1)}{P(\vec{x} | c_0)P(c_0) + P(\vec{x} | c_1)P(c_1)} \quad (\text{Bayes})$$

$$= \frac{1}{1 + \frac{P(\vec{x} | c_0)P(c_0)}{P(\vec{x} | c_1)P(c_1)}}$$

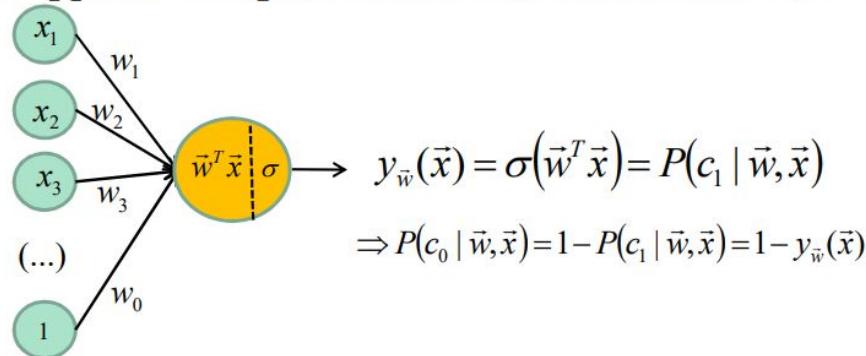
$$= \frac{1}{1 + e^{-a}} \quad \text{où } a = \ln \left[ \frac{P(\vec{x} | c_0)P(c_0)}{P(\vec{x} | c_1)P(c_1)} \right]$$

$$= \sigma(a)$$

# Amélioration du Perceptron

(N-D, 2 classes)

En d'autres mots, si on entraîne correctement un réseau logistique, on fini par apprendre la **probabilité conditionnelle de la classe  $c_1$** .



Quelle est la fonction de coût  
d'un réseau logistique?

# Fonction de coût d'un réseau logistique?

(2 classes)

Dans le cas d'un réseau logistique nous avons

Ensemble d'entraînement :  $D = \{(\vec{x}_1, t_1), (\vec{x}_2, t_2), \dots, (\vec{x}_n, t_n)\}$

Sortie du réseau:  $y_{\vec{w}}(\vec{x}) = \sigma(\vec{w}^T \vec{x}) = P(c_1 | \vec{w}, \vec{x})$

$$\begin{aligned} P(D | \vec{w}) &= \prod_{n=1}^N P(c_1 | \vec{w}, \vec{x}_n)^{t_n} (1 - P(c_1 | \vec{w}, \vec{x}_n))^{1-t_n} \\ &= \prod_{n=1}^N y_{\vec{w}}(\vec{x}_n)^{t_n} (1 - y_{\vec{w}}(\vec{x}_n))^{1-t_n} \end{aligned}$$

# Fonction de coût d'un réseau logistique?

(2 classes)

$$P(D | \vec{w}) = \prod_{n=1}^N y_{\vec{w}}(\vec{x}_n)^{t_n} (1 - y_{\vec{w}}(\vec{x}_n))^{1-t_n}$$

**Solution : Maximum de vraisemblance**

$$W = \arg \max_W P(D | W)$$

$$= \arg \max_W \prod_{n=1}^N y_W(\vec{x}_n)^{t_n} (1 - y_W(\vec{x}_n))^{1-t_n}$$

$$= \arg \min_W \sum_{n=1}^N -\ln \left[ y_W(\vec{x}_n)^{t_n} (1 - y_W(\vec{x}_n))^{1-t_n} \right]$$

$$= \arg \min_W - \sum_{n=1}^N t_n \underbrace{\ln(y_W(\vec{x}_n))}_{E_D(\vec{w})} + (1-t_n) \ln(1 - y_W(\vec{x}_n))$$

# Fonction de coût d'un réseau logistique?

(2 classes)

$$P(D | \vec{w}) = - \prod_{n=1}^N y_{\vec{w}}(\vec{x}_n)^{t_n} (1 - y_{\vec{w}}(\vec{x}_n))^{1-t_n}$$

La fonction de coût est **-ln de la vraisemblance**

$$\underline{E_D(\vec{w}) = - \sum_{n=1}^N t_n \ln(y_{\vec{w}}(\vec{x}_n)) + (1-t_n) \ln(1 - y_{\vec{w}}(\vec{x}_n))}$$

On peut également démontrer que

Entropie croisée  
(Cross entropy)

$$\underline{\frac{dE_D(\vec{w})}{d\vec{w}} = \sum_{n=1}^N (y_{\vec{w}}(\vec{x}_n) - t_n) \vec{x}_n}$$

Contrairement au Perceptron  
le gradient ne depend pas seulement  
des données mal classées

# Optimisation d'un réseau logistique

## Optimisation par batch

Initialiser  $\vec{w}$

k=0, i=0

DO k=k+1

$$\frac{dE_D(\vec{w})}{d\vec{w}} = \sum_{n=1}^N (y_{\vec{w}}(\vec{x}_n) - t_n) \vec{x}_n$$

$$\vec{w} = \vec{w} - \eta \frac{dE_D(\vec{w})}{d\vec{w}}$$

UNTIL K==K\_MAX.

## Descente de gradient stochastique

Initialiser  $\vec{w}$

k=0, i=0

DO k=k+1

FOR n = 1 to N

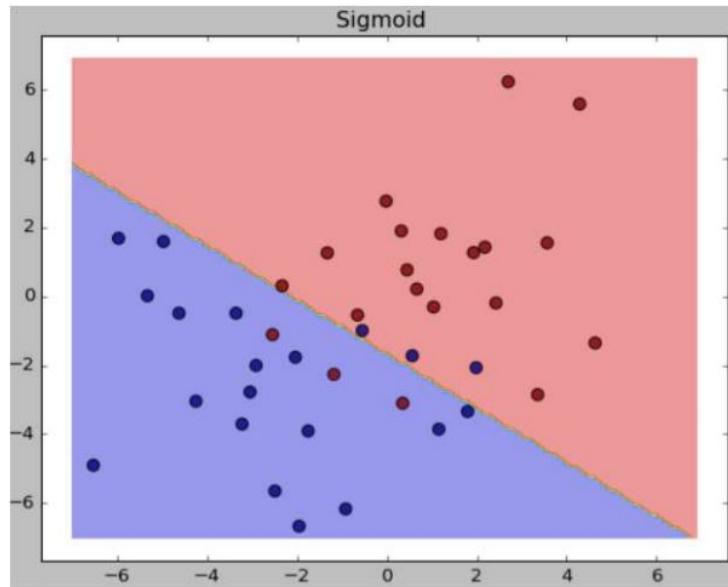
$$\vec{w} = \vec{w} - \eta (y_{\vec{w}}(\vec{x}_n) - t_n) \vec{x}_n$$

UNTIL K==K\_MAX.

# Réseau logistique

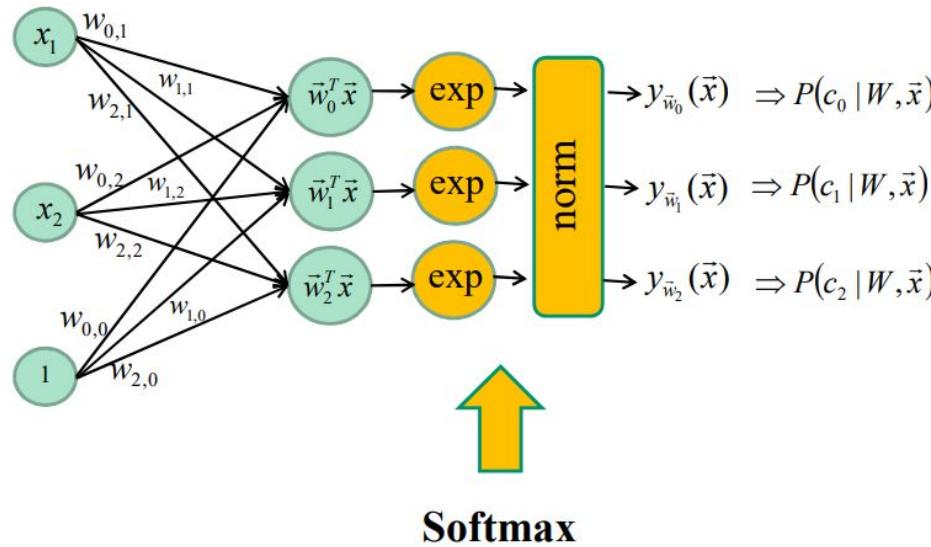
Avantages:

- Plus stable que le Perceptron
- Fonctionne mieux avec des données non séparables



# Et pour $K > 2$ classes?

Nouvelle fonction d'activation : **Softmax**

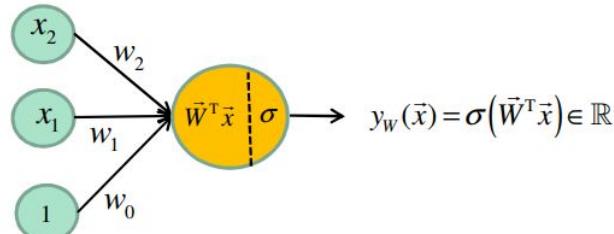


Allons coder !

Maintenant, rendons le réseau  
**profond**  
profond

Maintenant, rendons le réseau  
profond

## 2D, 2Classes, Régression logistique linéaire

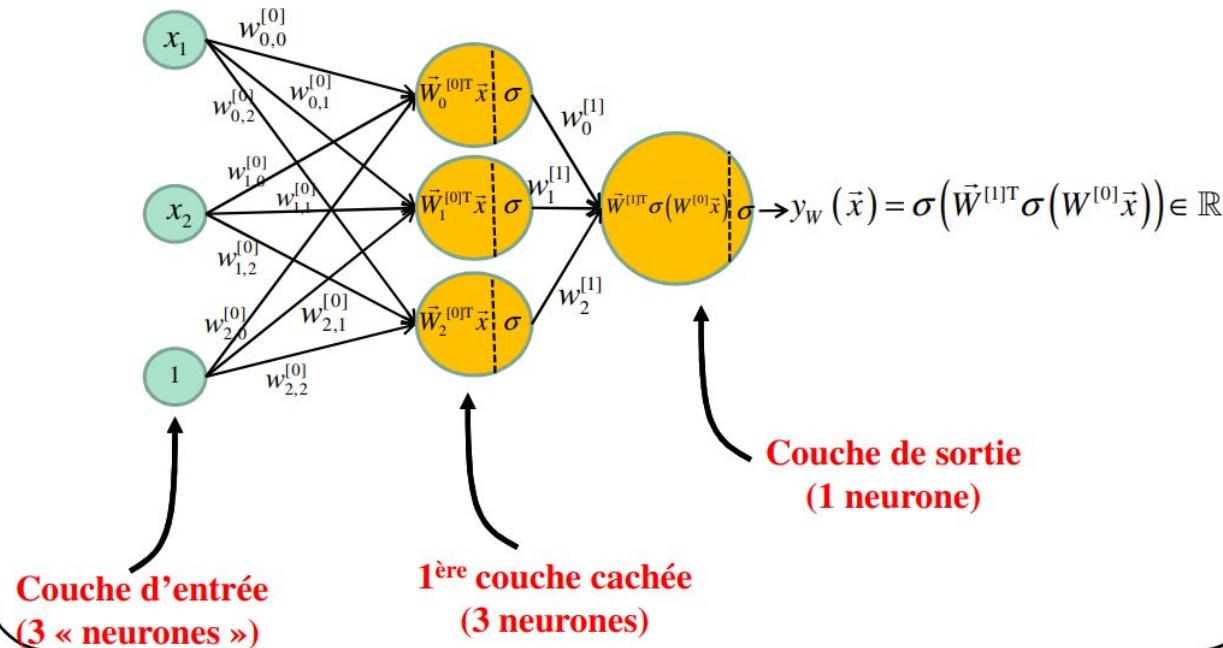


Couche d'entrée  
(3 « neurones »)

Couche de sortie  
(1 neurone avec sigmoïde)

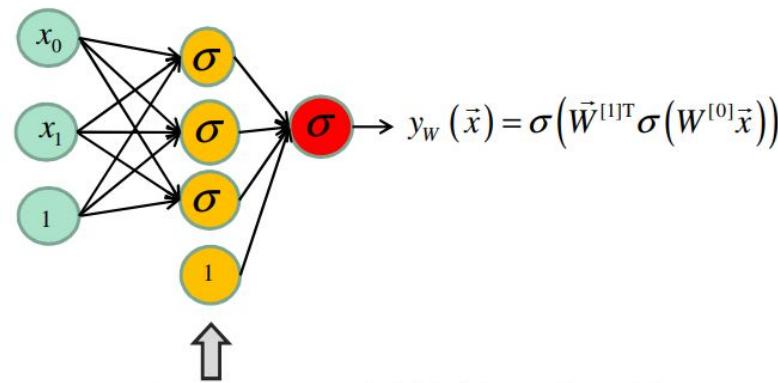
# 2D, 2Classes, Réseau à 1 couche cachée

Si on veut effectuer une **classification 2 classes** via une **régression logistique** (donc une fonction coût par « entropie croisée ») on doit ajouter **un neurone de sortie**.



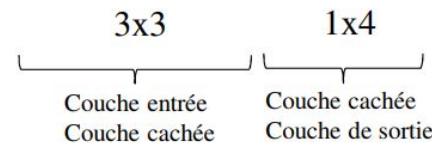
# 2D, 2Classes, Réseau à 1 couche cachée

Couche d'entrée      Couche cachée      Couche de sortie



Très souvent, on ajoute un neurone de biais à la couche cachée.

Ce réseau possède au total **13 paramètres**

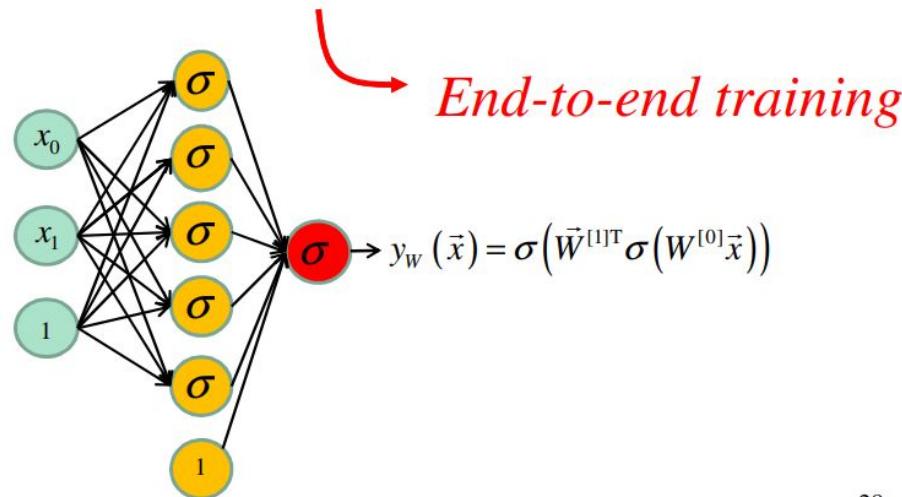


# NOTE Importante

Le but de la première couche est de **projeter les données d'entrée** (ici  $\vec{x} \in \mathbb{R}^2$ ) vers un espace dimensionnel plus grand (ici  $\sigma(W^{[0]}\vec{x}) \in \mathbb{R}^5$ ) là où les **classes sont linéairement séparables**.

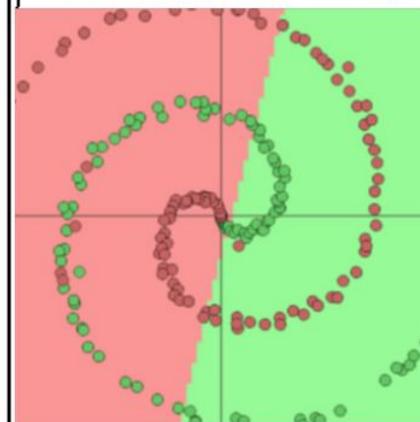
Car il ne faut pas oublier que la **couche de sortie** est une **régession logistique linéaire**.

Par conséquent, au lieu de fixer nous même la fonction de base, on laisse le **réseau l'apprendre**.



# Nombre de neurons VS Capacity

Aucun neurone caché

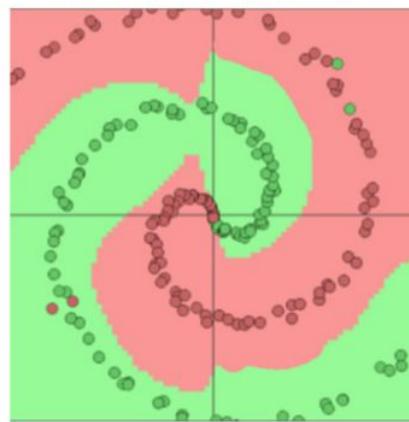


Classification linéaire

**Underfitting**

(pas assez de capacité)

12 neurones cachés

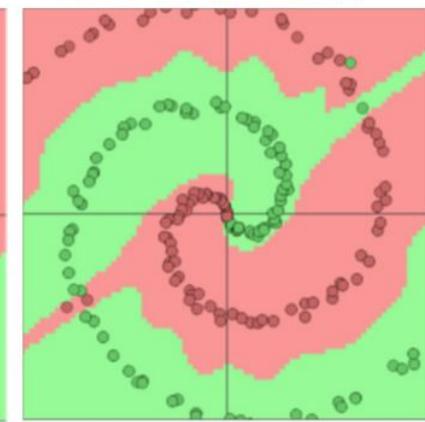


Classification non linéaire

**BON RÉSULTAT**

(bonne capacité)

60 neurones cachés

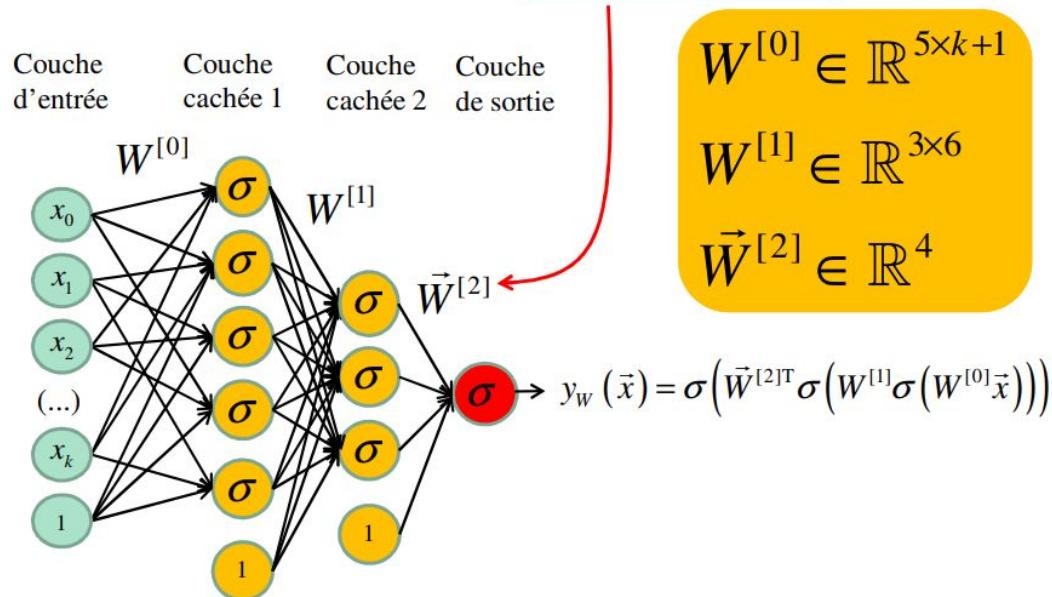


Classification non linéaire

**Over fitting**

(trop grande capacité)

## kD, 2Classes, Réseau à 2 couches cachées

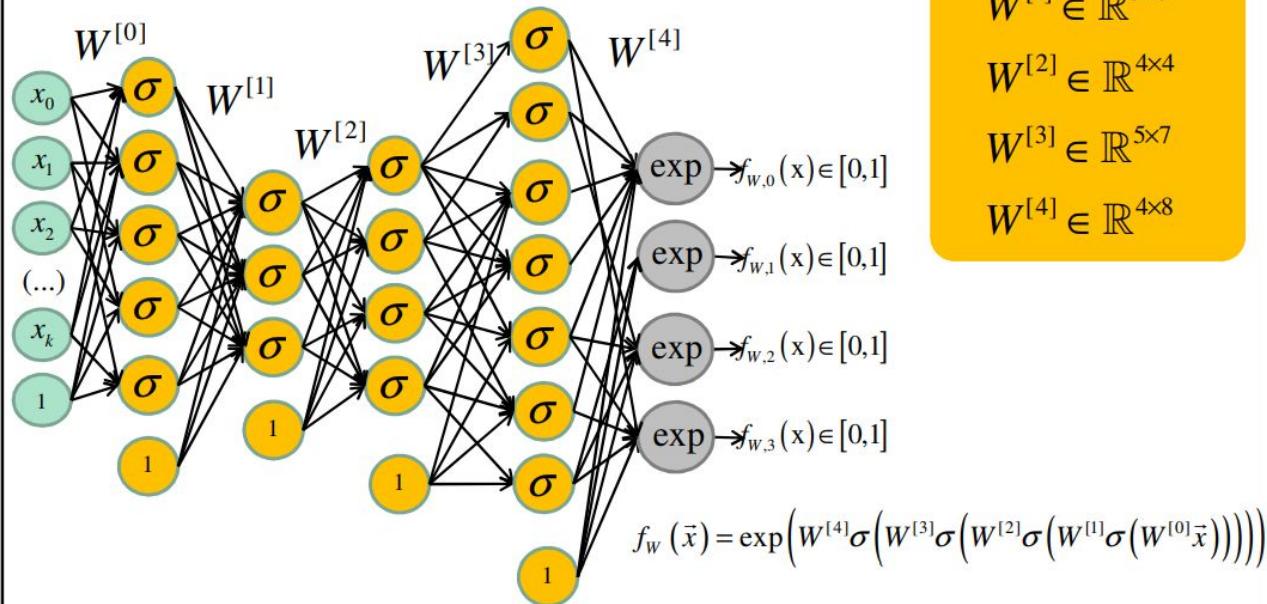


En ajoutant une couche cachée, on ajoute une multiplication matricielle

Ce réseau a  $5 \times (k+1) + 6 \times 3 + 1 \times 4$  **paramètres**

# kD, 4 Classes, Réseau à 3 couches cachées

Couche d'entrée   Couche cachée 1   Couche cachée 2   Couche cachée 3   Couche cachée 4   Couche de sortie



$$W^{[0]} \in \mathbb{R}^{5 \times k+1}$$

$$W^{[1]} \in \mathbb{R}^{3 \times 6}$$

$$W^{[2]} \in \mathbb{R}^{4 \times 4}$$

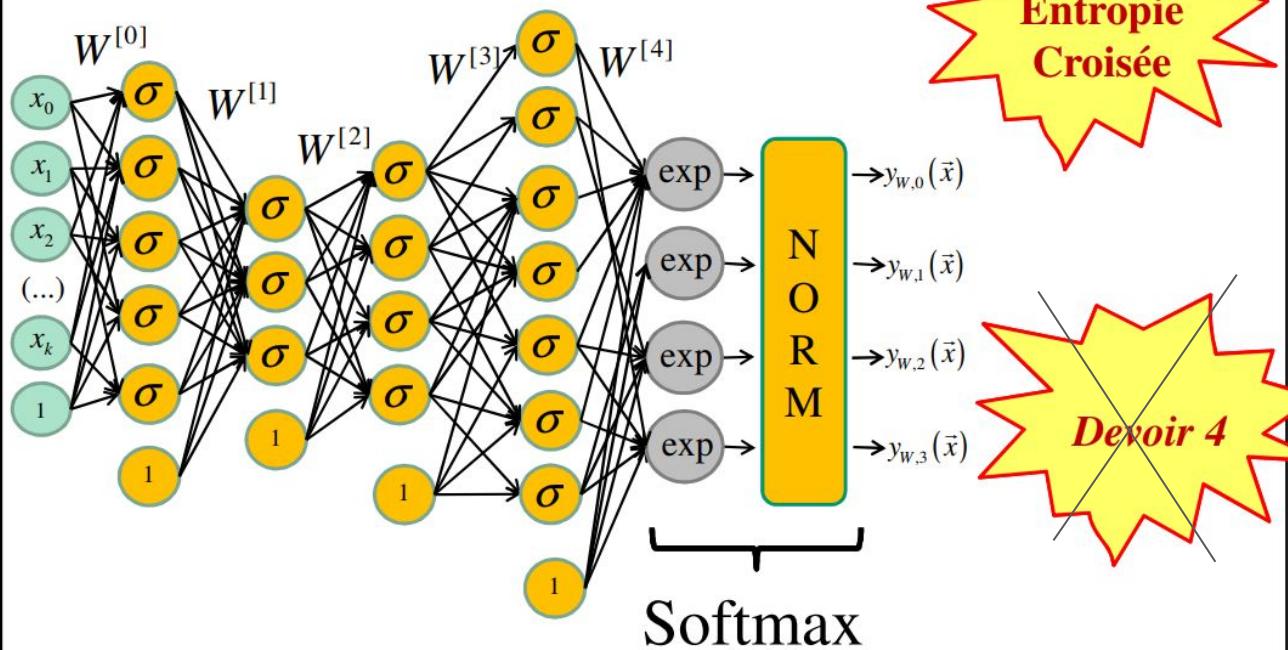
$$W^{[3]} \in \mathbb{R}^{5 \times 7}$$

$$W^{[4]} \in \mathbb{R}^{4 \times 8}$$

**Softmax**  $y_{W,i}(x) = \left( \frac{e^{f_i}}{\sum_j e^{f_j}} \right)_k$

# kD, 4 Classes, Réseau à 3 couches cachées

Couche d'entrée   Couche cachée 1   Couche cachée 2   Couche cachée 3   Couche cachée 4   Couche de sortie



$$y_W(\vec{x}) = \text{softmax}\left(W^{[4]}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)\right)$$

# Comment faire une prédiction?

Ex.: faire transiter un signal de **l'entrée à la sortie**  
d'un réseau à **3 couches cachées**

```
import numpy as np

def sigmoid(x):
    return 1.0 / (1.0+np.exp(-x))

x=np.insert(x,0,1) # Ajouter biais

H1 = sigmoid(np.dot(W0,x))
H1 = np.insert(H1,0,1) # Ajouter biais } Couche 1

H2 = sigmoid(np.dot(W1,H1))
H2 = np.insert(H2,0,1) # Ajouter biais } Couche 2

H2 = sigmoid(np.dot(W2,H1))
H2 = np.insert(H2,0,1) # Ajouter biais } Couche 3

y_pred = np.dot(W3,H2)
out = softmax(y_pred) # dans le cas de multiclassé } Couche sortie
```

*Forward pass*

# Comment optimiser les paramètres?

0- Partant de

$$W = \arg \min_W E_D(W) + \lambda R(W)$$

Trouver une fonction de régularisation. En général

$$R(W) = \|W\|_1 \text{ ou } \|W\|_2$$

# Comment optimiser les paramètres?

**1-** Trouver une loss  $E_D(W)$  comme par exemple

**Hinge loss**

**Entropie croisée (*cross entropy*)**



N'oubliez pas d'ajuster la sortie du réseau en fonction de la loss que vous aurez choisi.

*cross entropy => Softmax*

# Comment optimiser les paramètres?

**2-** Calculer le gradient de la loss par rapport à chaque paramètre

$$\frac{\partial E_D(W)}{\partial w_{a,b}^{[c]}} \longrightarrow \text{Rétropropagation/}\text{“Backpropagation”}$$

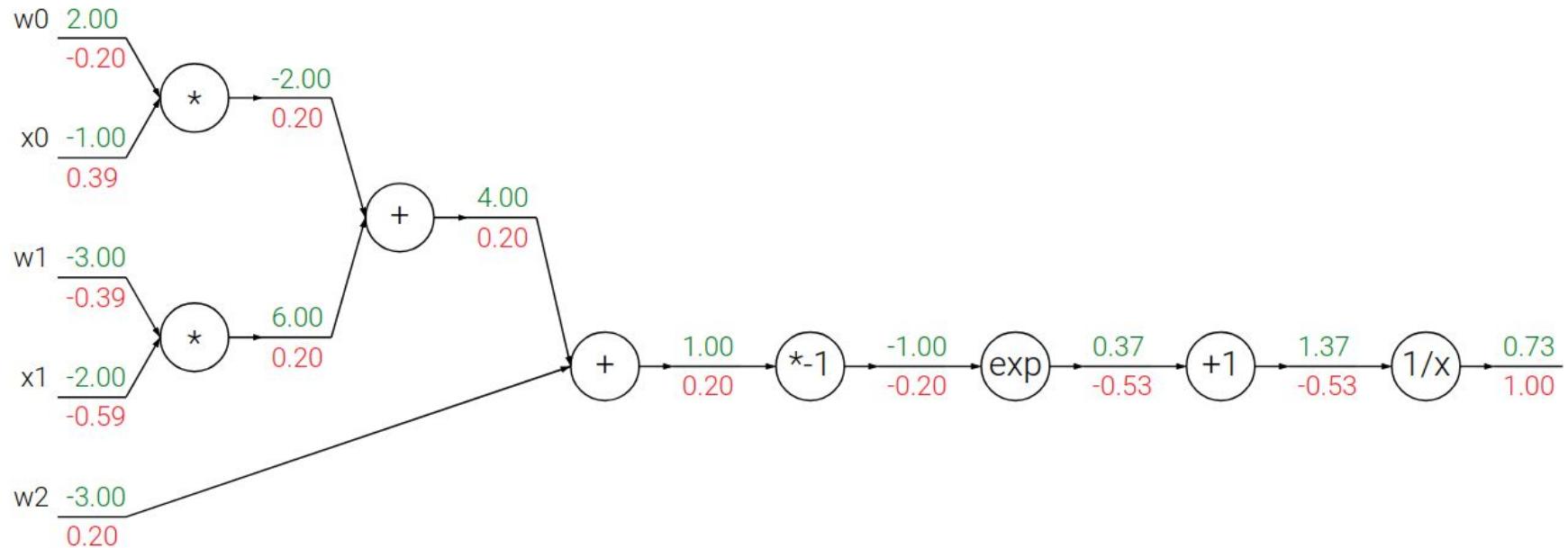
et lancer un algorithme de descente de gradient pour mettre à jour les paramètres.

$$w_{a,b}^{[c]} = w_{a,b}^{[c]} - \eta \frac{\partial E_D(W)}{\partial w_{a,b}^{[c]}}$$

Vert: Forward pass

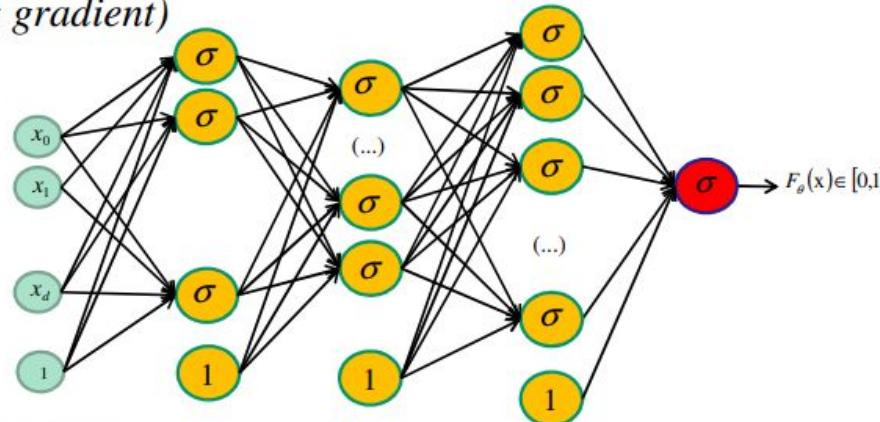
Rouge: Backward-pass

## Exemple de backpropagation



# Disparition du gradient

(vanishing gradient)



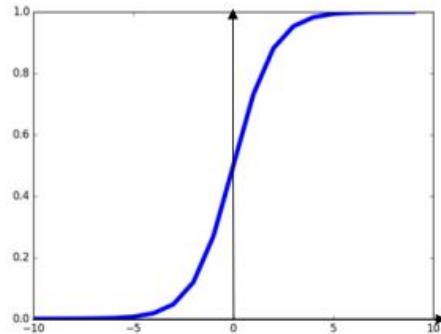
Malheureusement, l'entraînement d'un **réseau profond** avec **rétro-propagation** et des fonctions d'activations **sigmoïdales** entraîne des problèmes de

**disparition du gradient**

Démonstration au  
tableau

# Fonction d'activation

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



**Sigmoïde**

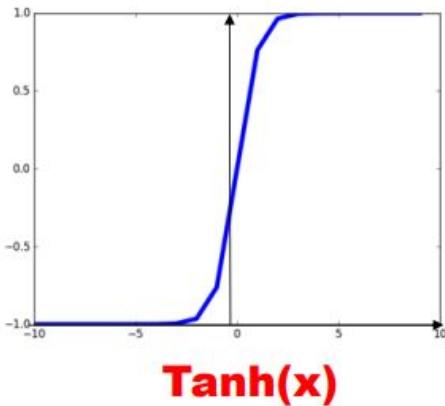
- Ramène les valeurs entre 0 et 1
- Historiquement populaire

## 3 Problèmes :

- Un neurone saturé a pour effet de « **tuer** » **les gradients**
- Sortie d'une sigmoïde n'est **pas centrée à zéro**.
- `exp()` est **coûteux** lorsque le nombre de neurones est élevé.

On résoud le problème de la  
disparition du gradient à l'aide  
**d'autres fonctions d'activations**

# Fonction d'activation

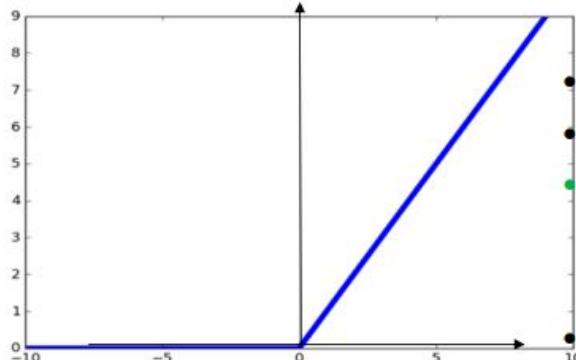


- Ramène les valeurs entre -1 et 1
- Sortie centrée à zéro 😊
- **Disparition du gradient** lorsque la fonction sature 😞

[LeCun et al., 1991]

# Fonction d'activation

$$\text{ReLU}(x) = \max(0, x)$$



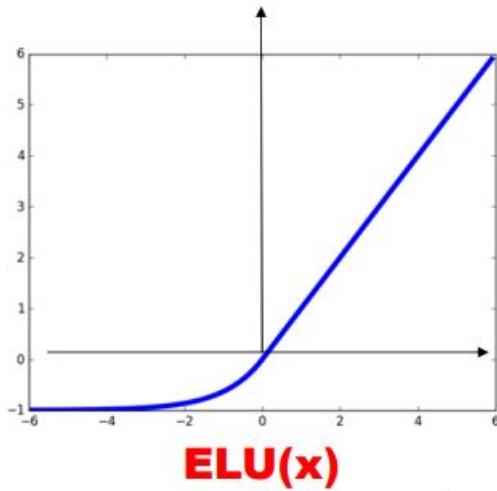
**ReLU(x)**  
(Rectified Linear Unit)

- Aucune **saturation** 😊
- Super **rapide** 😊
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) 😊
- Sortie **non centrée à zéro** 🤦
- **Un inconvénient** : qu'arrive-t-il au gradient lorsque  $x < 0$ ? 🤔

[Krizhevsky et al., 2012]

# Fonction d'activation

$$\text{ELU}(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha(e^x - 1) & \text{sinon} \end{cases}$$

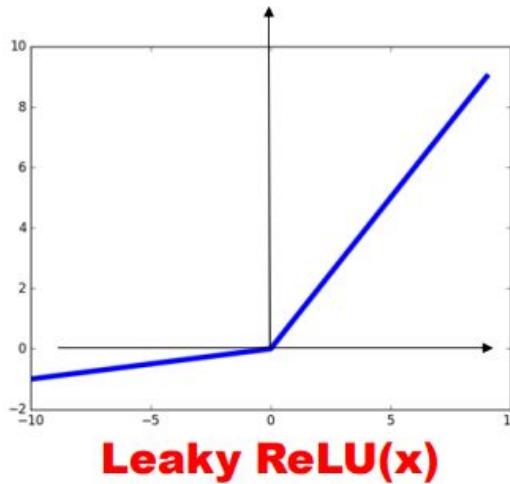


- Tous les avantages de **ReLU** 😊
- Sortie plus « **centrée à zéro** » 😊
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) 😊
- **Gradients meurent plus lentement** 😓
- $\exp()$  est **coûteux** 😟

[Clevert et al., 2015]

# Fonction d'activation

$$\text{LReLU}(x) = \max(0.01x, x)$$



- Aucune **saturation** ☺
- Super **rapide** ☺
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) ☺
- **Gradients ne meurent pas** ☺
- 0.01 est un **hyperparamètre** ☺

[Mass et al., 2013]  
[He et al., 2015]

Allons coder !

# Ce qu'il manque

- Bonnes pratiques d'optimisation
- Autres “trucs” pour les réseaux de neurones
- Convolutions, LSTMs, GANs, RL
- Et plus !

# Pour en apprendre plus

- IFT603 : Techniques d'optimisations
- CS231n: Convolutional Networks for Visual Recognition
- Cours d'Andrew Ng sur Coursera

Merci !