

*IA et Nous*  
*Deep Learning*  
*Réseaux de neurones*

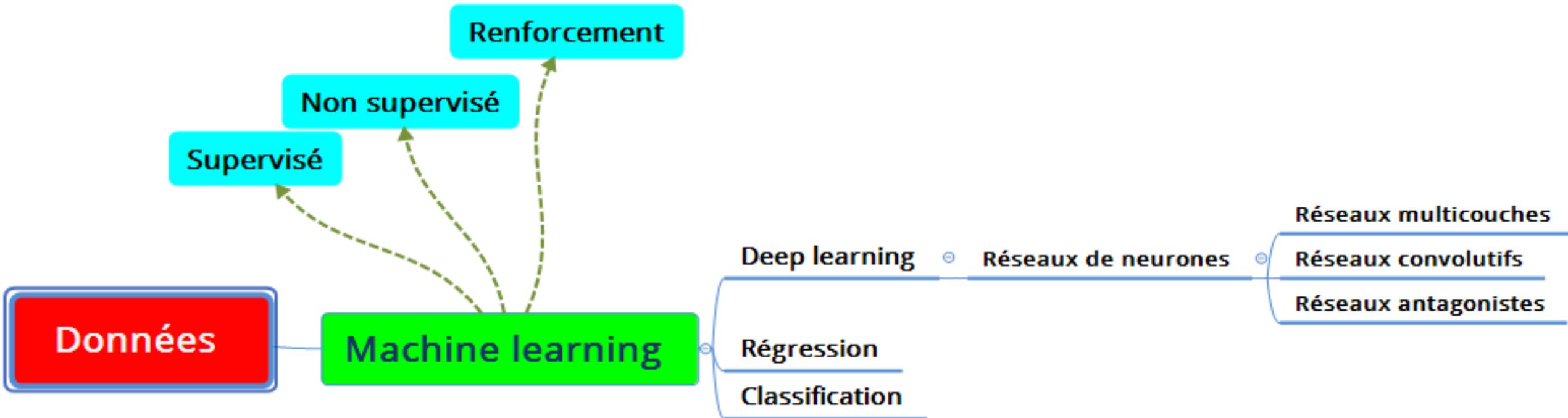
UATL 24/25

# Plan

- 1. Le Neurone formel de McCulloch et Pitts***
- 2. Le Perceptron de Rosenblatt***
- 3. Le Perceptron Multicouche***
- 4. Améliorations du perceptron***
  - Illustration*
- 5. Retour au multicouche***
  - Illustration*
- 6. Démonstrations – Exemples – Exercices***

# Apprentissage

## Élément essentiel de l'intelligence



*Les statistiques ou l'art d'extraire des connaissances des données*

# Le Cerveau

- $85 \times 10^9$  neurones - 85 Milliards
- $10^4$  Synapses/neurone  $\rightarrow 10^{15}$  synapses
- 1,4 kg - 1,7 litres
- Cortex  $2500 \text{ cm}^3$  - 2 mm épaisseur
- 180 000 km de « câbles »
- 250 millions de neurones par  $\text{mm}^3$
- 20 Watts

*L'apprentissage partie intégrante de l'intelligence*

*L'apprentissage modifie l'efficacité des synapses :*

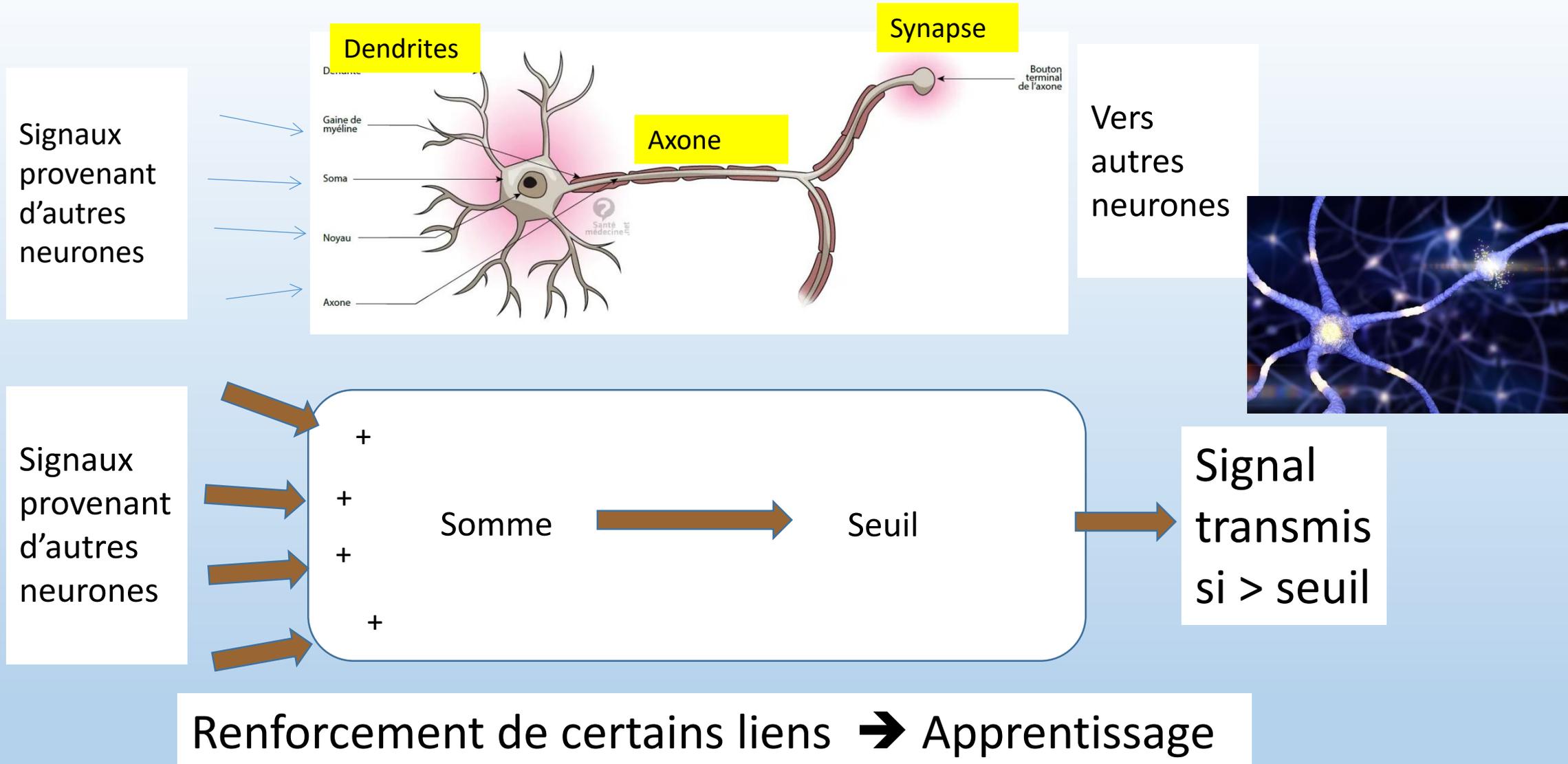
- *renforcement/affaiblissement*
- *apparition/disparition*



# *Le neurone formel*

*McCulloch et Pitts 1943*

# Modélisation neurone



# Le neurone formel

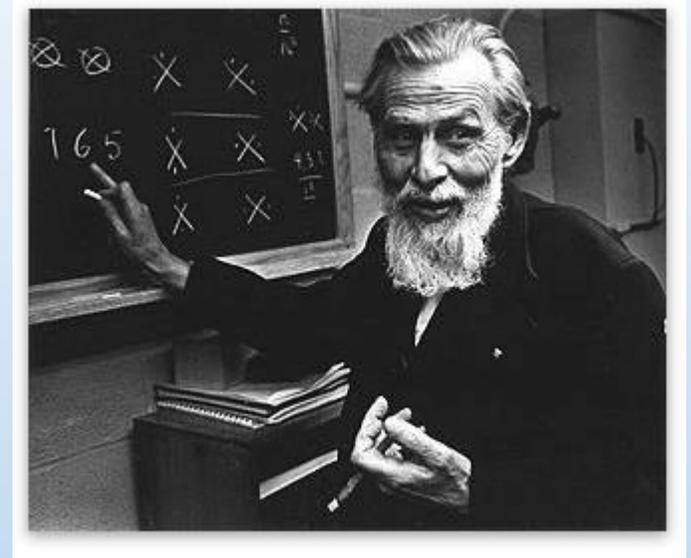
*McCulloch et Pitts 1943*



**Walter Pitts**  
Psychologue

Premier modèle mathématique et informatique du neurone biologique, proposé par Warren McCulloch et Walter Pitts en 1943. Il s'agit d'un neurone binaire, c'est-à-dire dont la sortie vaut 0 ou 1.

En étudiant l'analogie entre le cerveau humain et les machines informatiques universelles, ils montrèrent qu'un réseau (bouclé) constitué des neurones formels de leur invention a la même puissance de calcul qu'une machine de Turing.

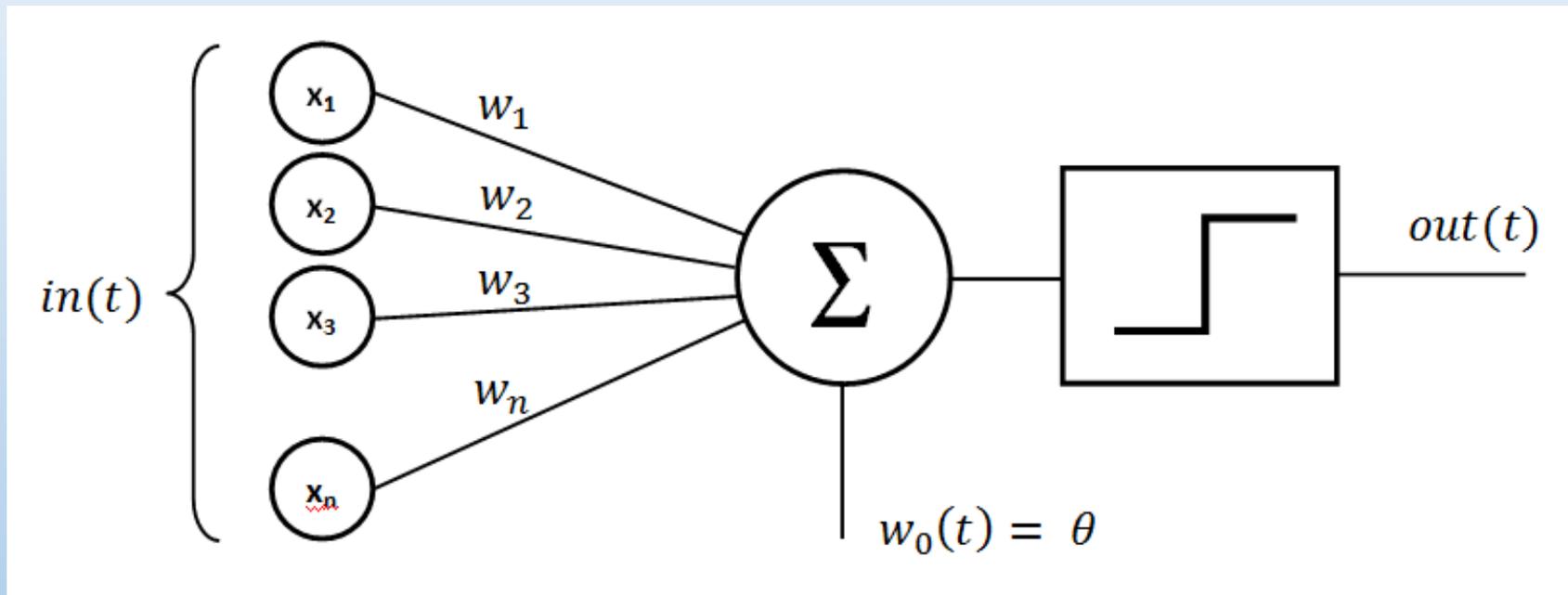


**Warren McCulloch**  
Chercheur en neurologie

# Le neurone formel

McCulloch et Pitts 1943

## Neurone binaire



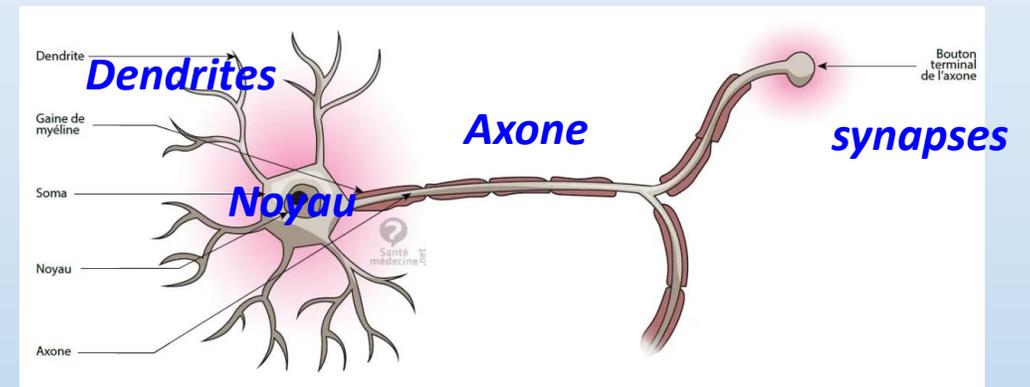
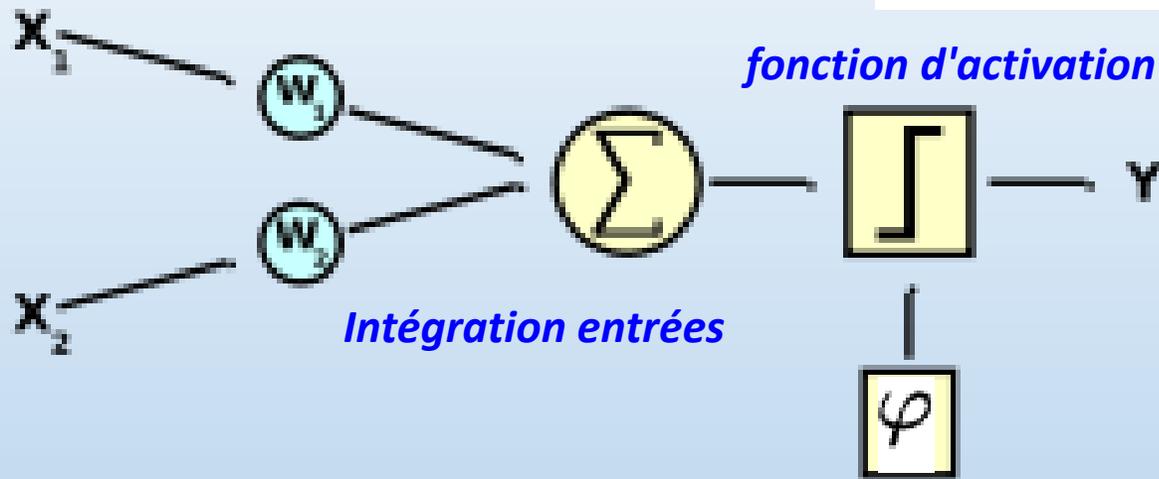
**Sortie**  
**0 ou 1**

# Le neurone formel

McCulloch et Pitts 1943

## Neurone binaire

Dans le modèle de McCulloch et Pitts, à chaque entrée est associé un poids synaptique, c'est-à-dire une valeur numérique notée  $w_1$  pour l'entrée 1 jusqu'à  $w_m$  pour l'entrée  $m$

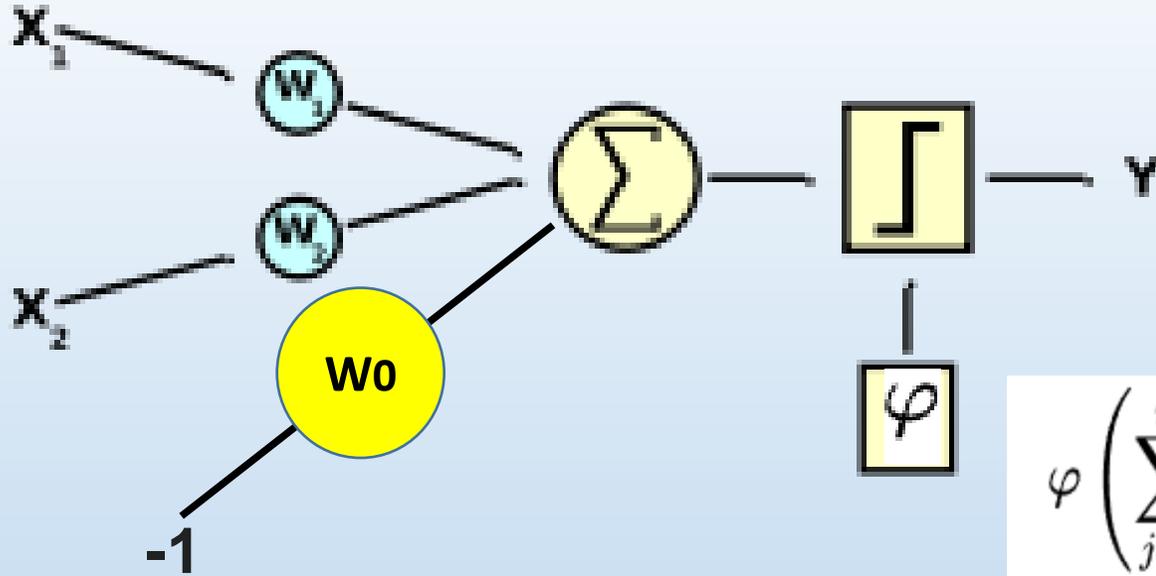


$$w_1 x_1 + \dots + w_m x_m = \sum_{j=1}^m w_j x_j.$$

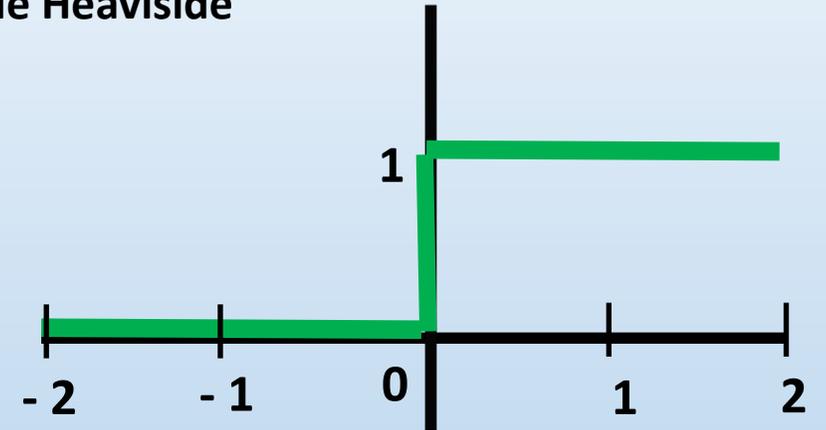
Neurone biologique	Neurone formel
Synapses	Poids des connexions
Axones	Signal de sortie
Dendrites	Signal d'entrée
Noyau ou Somme	Fonction d'activation

# Le neurone formel

McCulloch et Pitts 1943



Le résultat est transformé par une **fonction d'activation** non linéaire  $\varphi$  (parfois appelée fonction de sortie)  
**Fonction de Heaviside**



À cette grandeur s'ajoute un seuil  $w_0$

Si la somme  $\sum_{j=1}^m w_j x_j \geq w_0$   $Y = 1$ , sinon  $Y = 0$  :

$w_0$  est donc le seuil d'activation du neurone, d'où le nom de neurone binaire

# Le neurone formel

McCulloch et Pitts 1943

## Threshold Logic Unit

Au départ le modèle n'est conçu que pour traiter des entrées logiques donc à 0 ou à 1

Mc Culloch et Pitts ont montré que l'on pouvait reproduire les **fonctions booléennes** « ET » et « OU »

$H(x_1 + x_2 + w_0)$  avec  $H$  fonction de Heaviside

Selon la valeur de  $w_0$

$-1 \leq w_0 < 0$

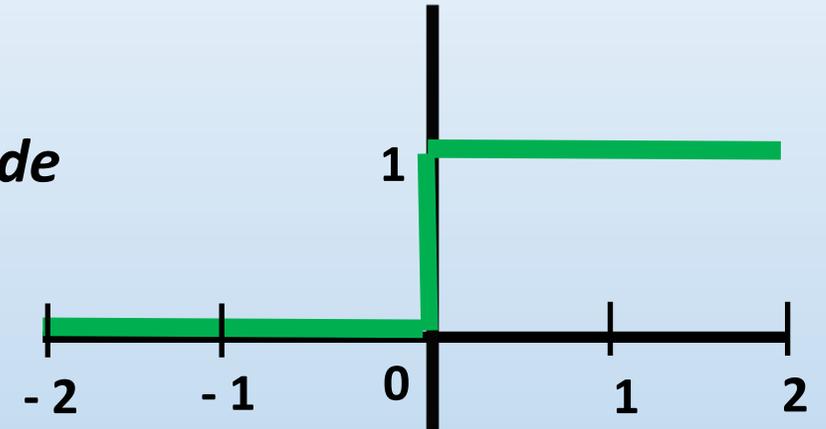
$x_1/x_2$	0	1
0	0	1
1	1	1

Porte OU

$-2 \leq w_0 < -1$

$x_1/x_2$	0	1
0	0	0
1	0	1

Porte ET

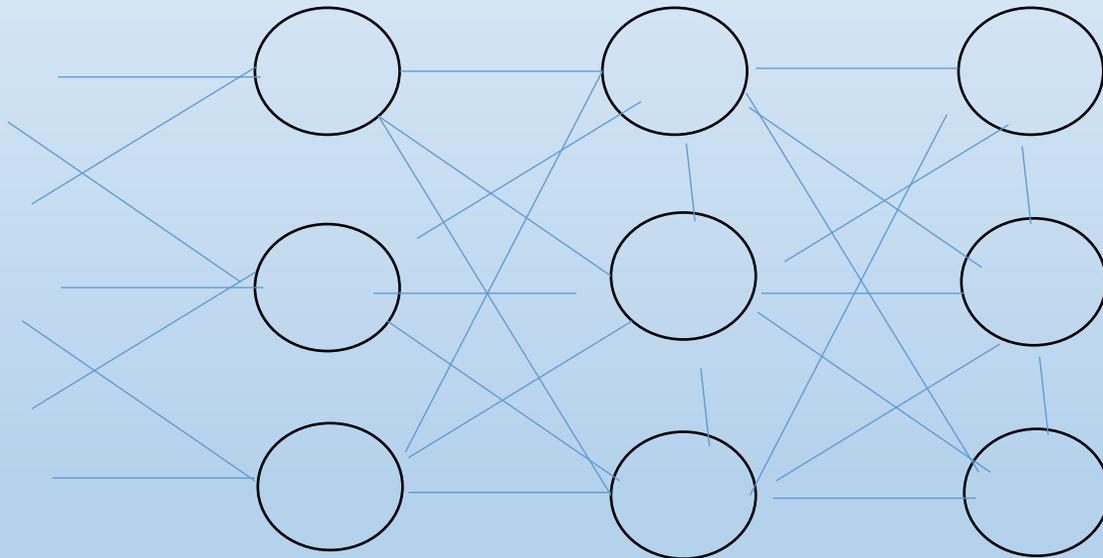


# Le neurone formel

McCulloch et Pitts 1943

*En combinant des neurones de McCulloch et Pitts,, on peut donc réaliser n'importe quelle fonction logique.*

*Quand on autorise en outre des connexions formant des boucles dans le réseau, on obtient un système avec la même puissance qu'une machine de Turing*



*L'engouement fut donc extraordinaire voire démesuré*

***Certains imaginent que ces machines allaient être capables de remplacer le cerveau humain !!!***

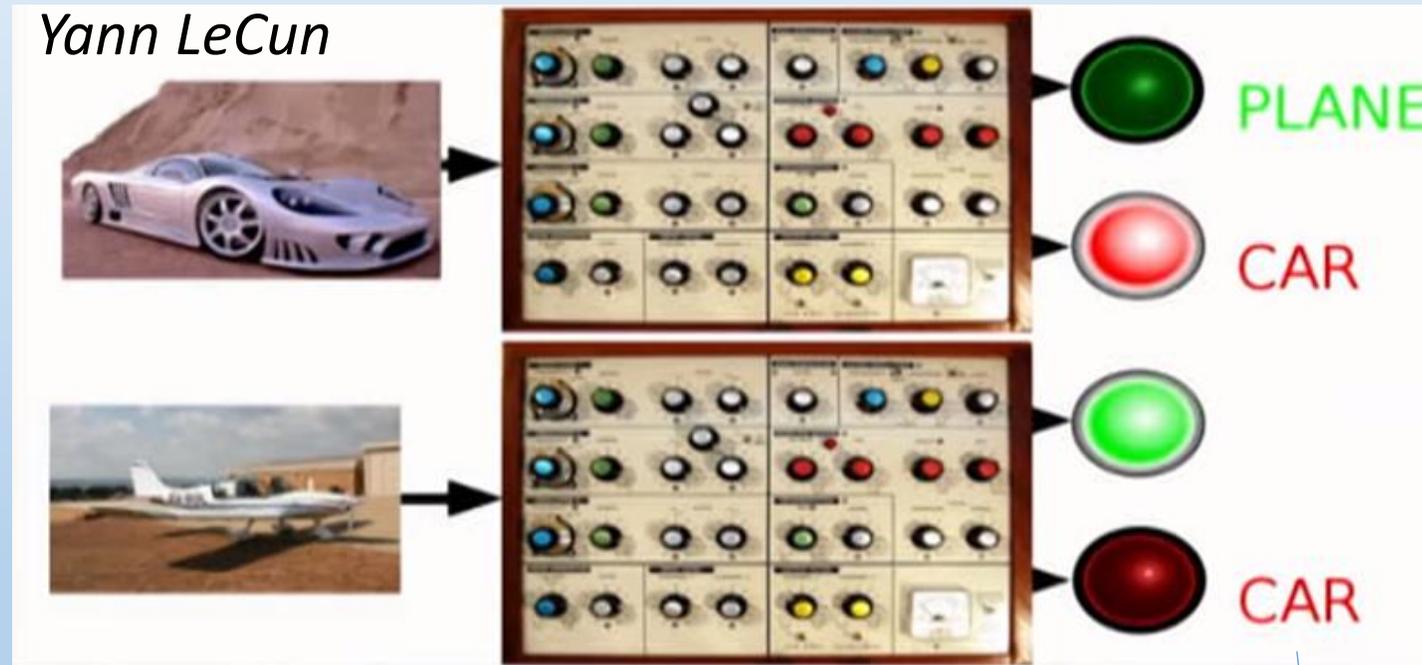


# Le neurone formel

McCulloch et Pitts 1943

Mais même si ce modèle pose les bases de qui est encore aujourd'hui le Deep Learning, il a plusieurs lacunes dont :

- il ne traite que des valeurs logiques
- il ne dispose pas *d'algorithme d'apprentissage*, ce qui nous oblige à trouver nous-mêmes les valeurs des « poids »  $W_i$  pour des applications du monde réel



# Perceptron \*

*Rosenblatt 1957*



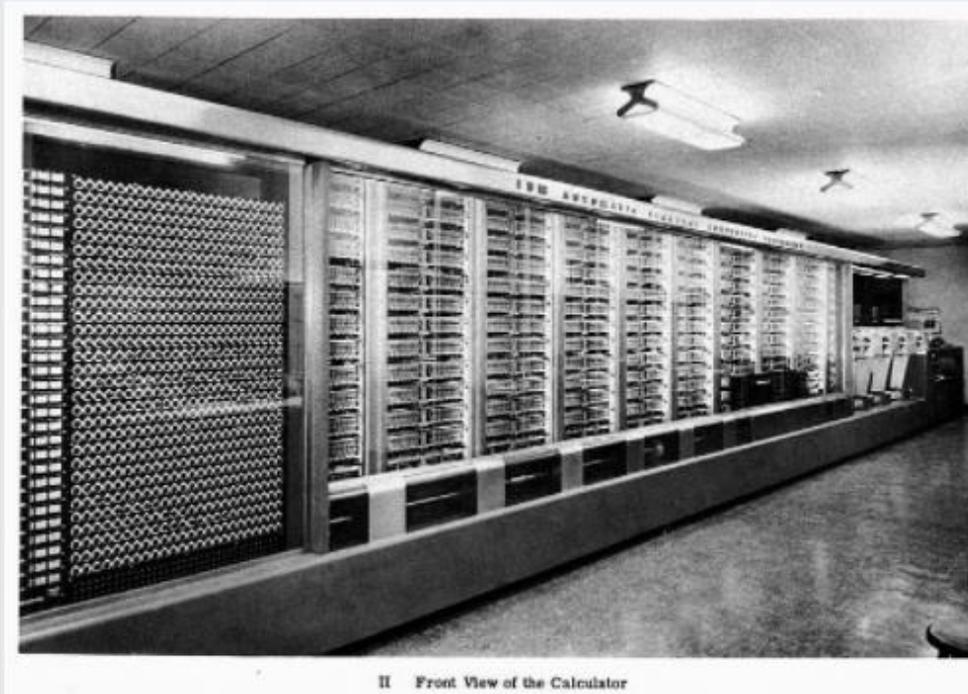
*Vient de Perception....*

# Les progrès du connexionnisme

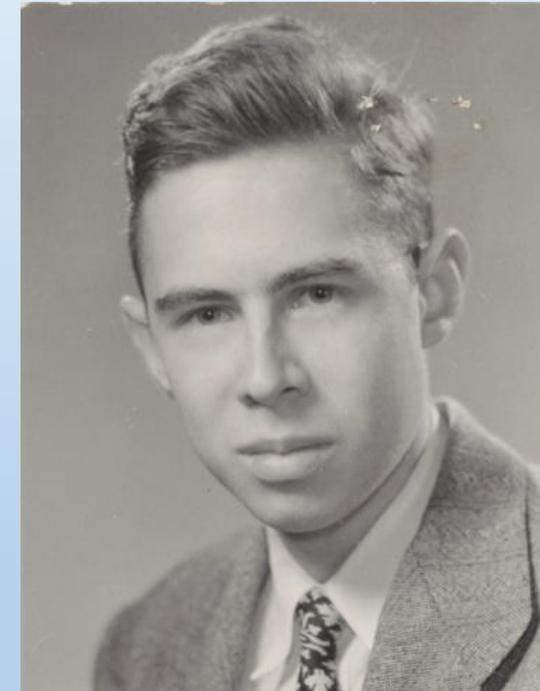
*Frank Rosenblatt invente en 1957 le premier réseau de neurones artificiels à une seule couche : le Perceptron*

*Il implémente sur un IBM 704 le Perceptron et parvient à simuler le processus de mémorisation en appliquant la règle de Hebb.*

Perceptron (1960)

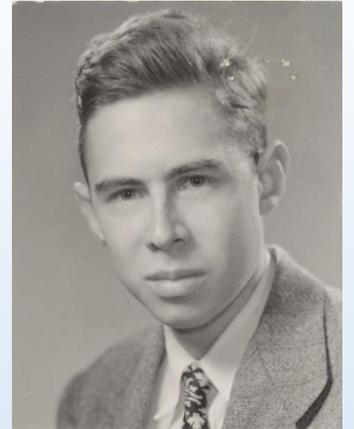


II Front View of the Calculator

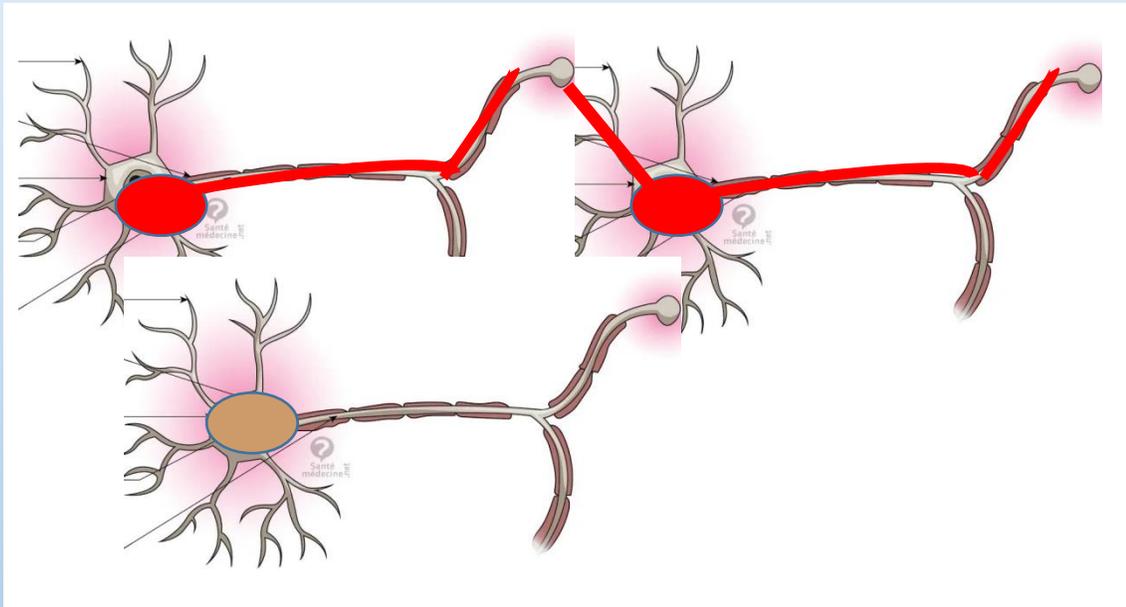


# Perceptron

Rosenblatt 1957



En 1957, un psychologue américain proposa le premier algorithme d'apprentissage **Franck Rosenblatt** avec le **Perceptron**



**Il utilise la règle de Hebb :**

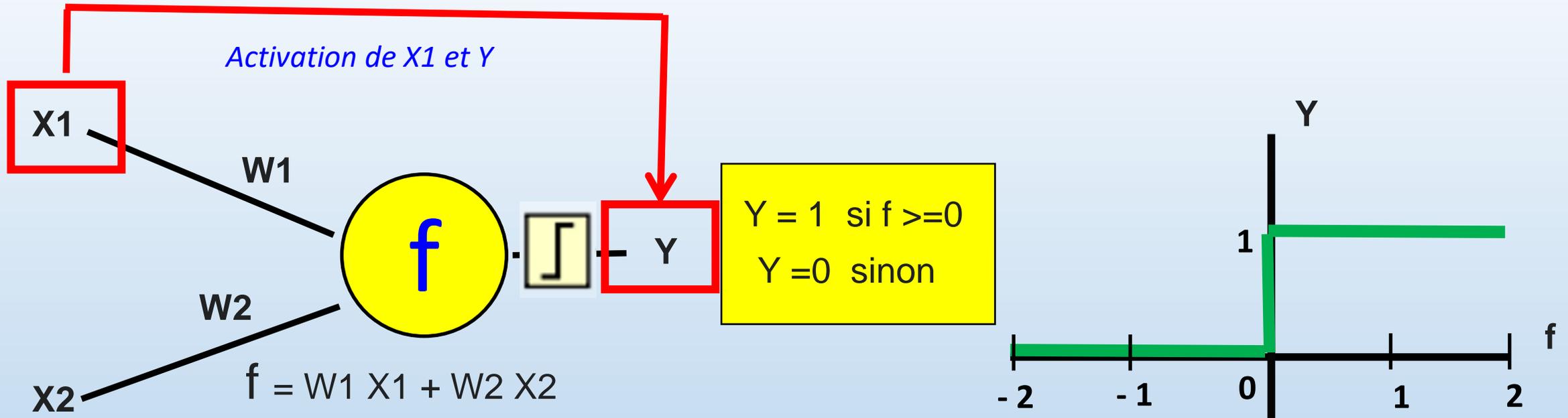
Résumée par la formule : « des neurones qui s'excitent ensemble se lient entre eux. »

( cells that fire together, wire together ).

Ils renforcent leurs connexions, leurs liens synaptiques : c'est ce que les neurosciences appellent la plasticité synaptique

# Perceptron

## Algorithme d'apprentissage

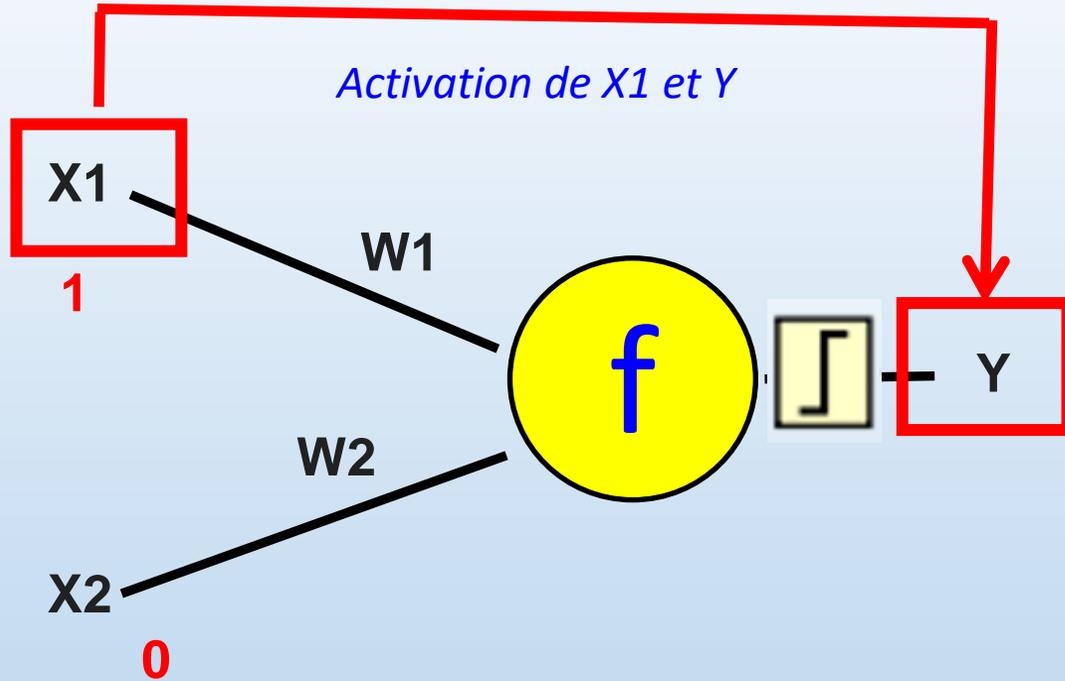


Formule de l'algorithme :  
 $W = W + a (Y_{true} - Y) X$

$Y_{true}$  sortie de référence , ou attendue  
 $Y$  Sortie produite  
 $X$  Entrée

# Perceptron

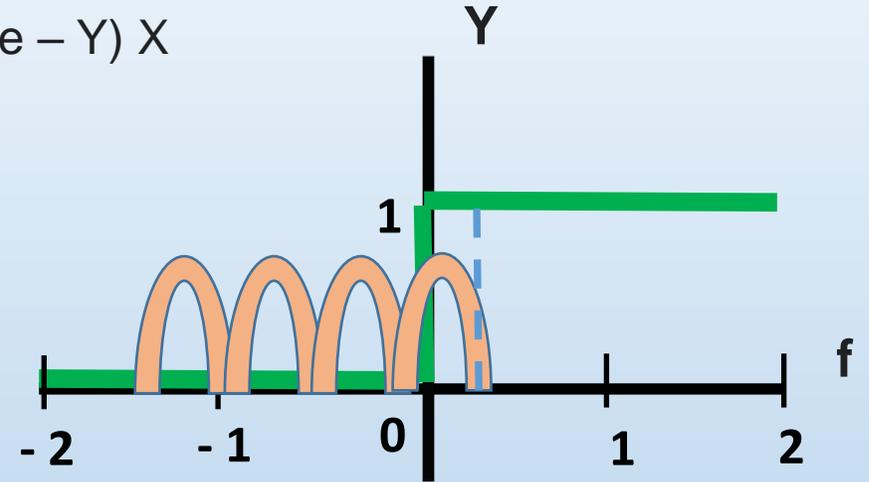
## Algorithme d'apprentissage



W1 et W2 ?

$$W = W + a (Y_{true} - Y) X$$

Y<sub>true</sub> = 1

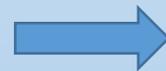


Si  $Y = 0$

$$W1 = W1 + a (1 - 0); \quad W1 = W1 + a$$

$$W2 = W2$$

Et ainsi de suite tant que  $Y = 0$

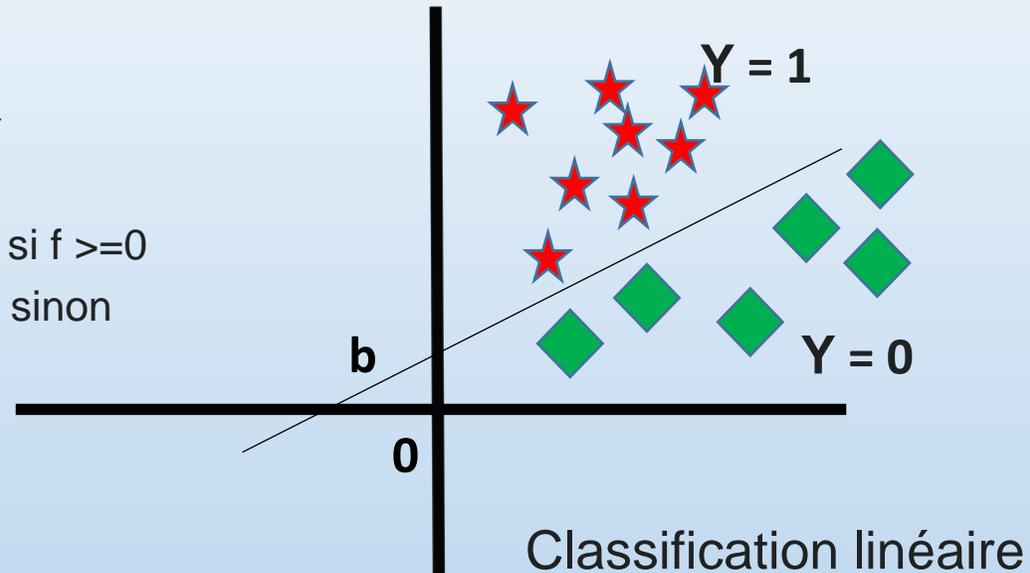
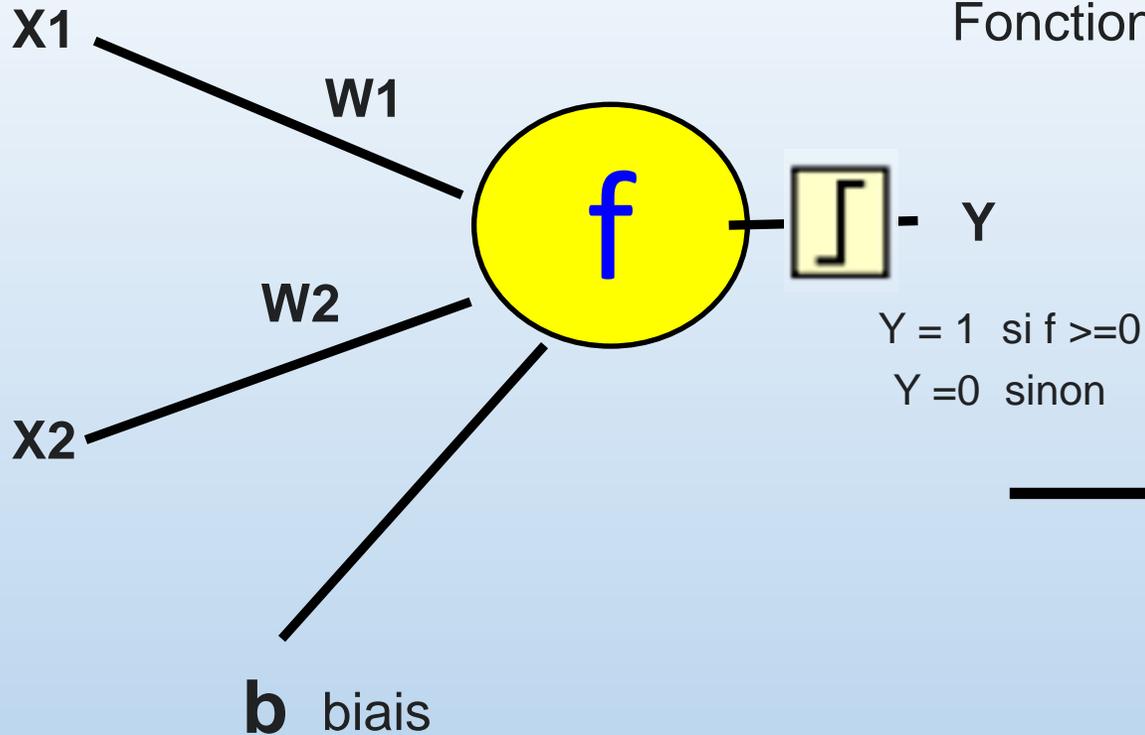


$$W1 = W1 + a$$

# Perceptron

## Modèle linéaire

$$\text{Fonction linéaire } f = W1 X1 + W2 X2 + b$$



On démontre que l'algorithme de Roseblatt ne **converge** vers une solution qui dépend des conditions initiales que si les données d'entraînement sont **séparables linéairement**

# Perceptron

*Là encore , l'engouement fut extraordinaire voire démesuré.*

*Ce fut un boom*

*On pensait que, grâce au Perceptron, on pourrait réaliser des machines capables d'écrire, de parler, de marcher, voire même d'avoir une conscience !!!*

*En 1969, Marvin Minsky et Seymour Papert montrent que les neurones formels ne peuvent enregistrer que des données linéairement séparables.*

*En particulier, un neurone formel ne peut simuler la fonction **OU exclusif (XOR)***

*Or, une grande partie des phénomènes de notre univers ne sont pas linéaires*

*→ le perceptron simple n'a donc pas trop d'intérêt*

*Les travaux sur les réseaux de neurones s'arrêtent quasi totalement pendant 10 ans*

*Premier Hiver de l'IA 1970 -1980 Quasiment plus d'investissements pendant 10 ans*

# *Le Perceptron Multicouche*

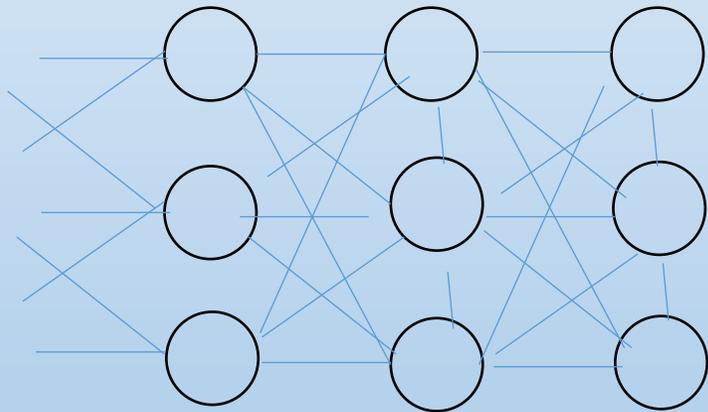
*Réseau neuronal  
« feed-forward »*

# La renaissance du connexionnisme

Au début des **années 80**, redémarrage de travaux sur les **réseaux de neurones**

En **1986**, **Geoffrey Hinton**, un des pères du **Deep Learning**, développe le **Perceptron multicouches**

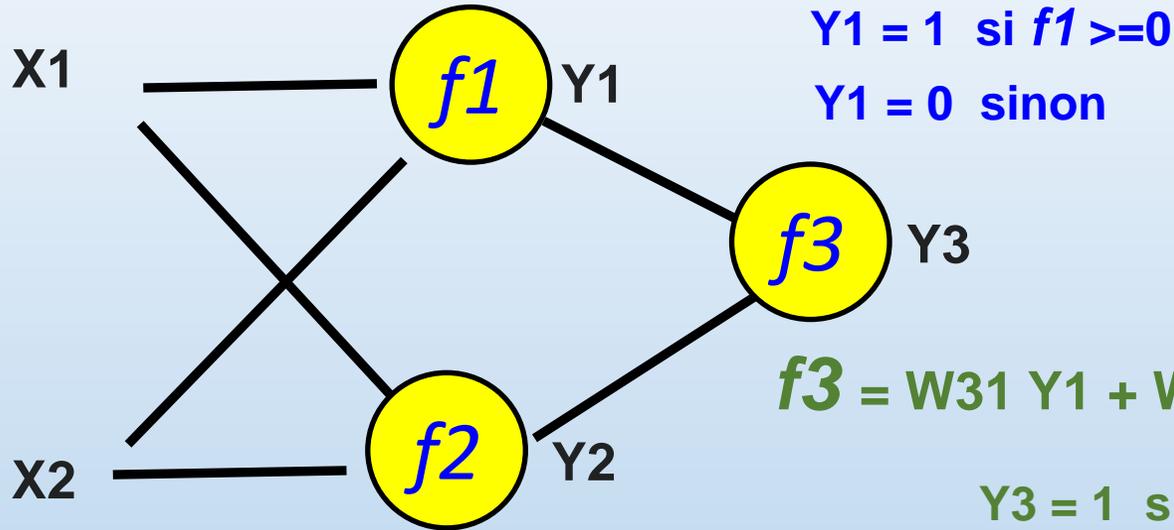
**Idée** : en connectant ensemble plusieurs neurones, on devrait pouvoir résoudre des problèmes plus complexes et représenter toute fonction



# Le Perceptron Multicouche

## Association de 3 neurones

$$f1 = W11 X1 + W12 X2 + b1$$



$$f3 = W31 Y1 + W32 Y2 + b3$$

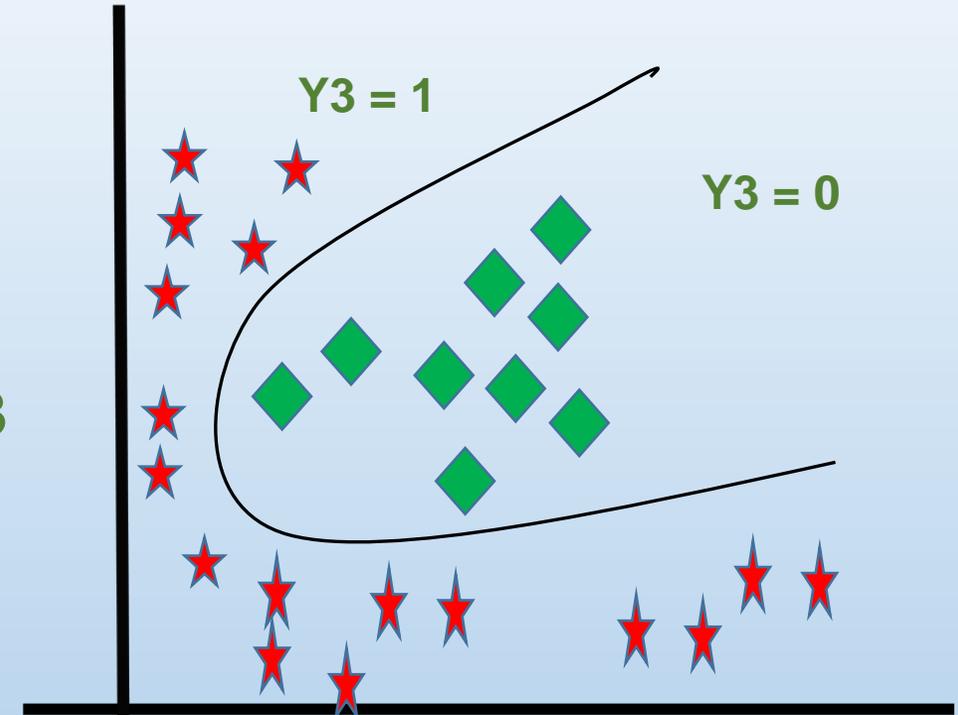
$$Y3 = 1 \text{ si } f3 \geq 0$$

$$Y3 = 0 \text{ sinon}$$

$$f2 = W21 X1 + W22 X2 + b2$$

$$Y2 = 1 \text{ si } f2 \geq 0$$

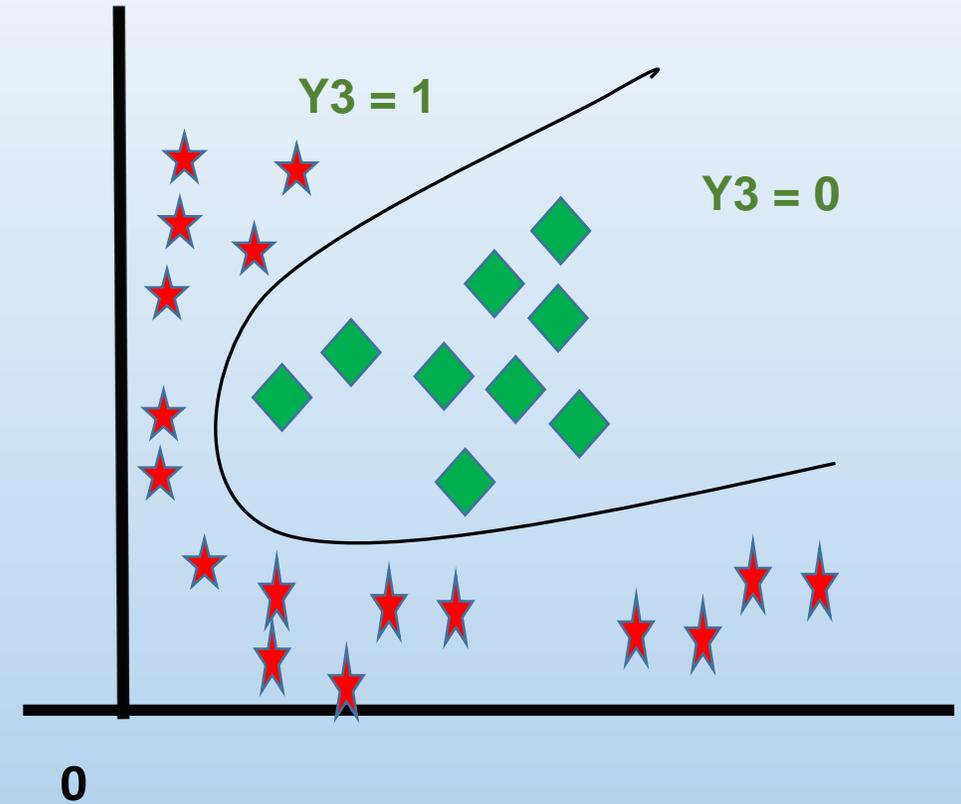
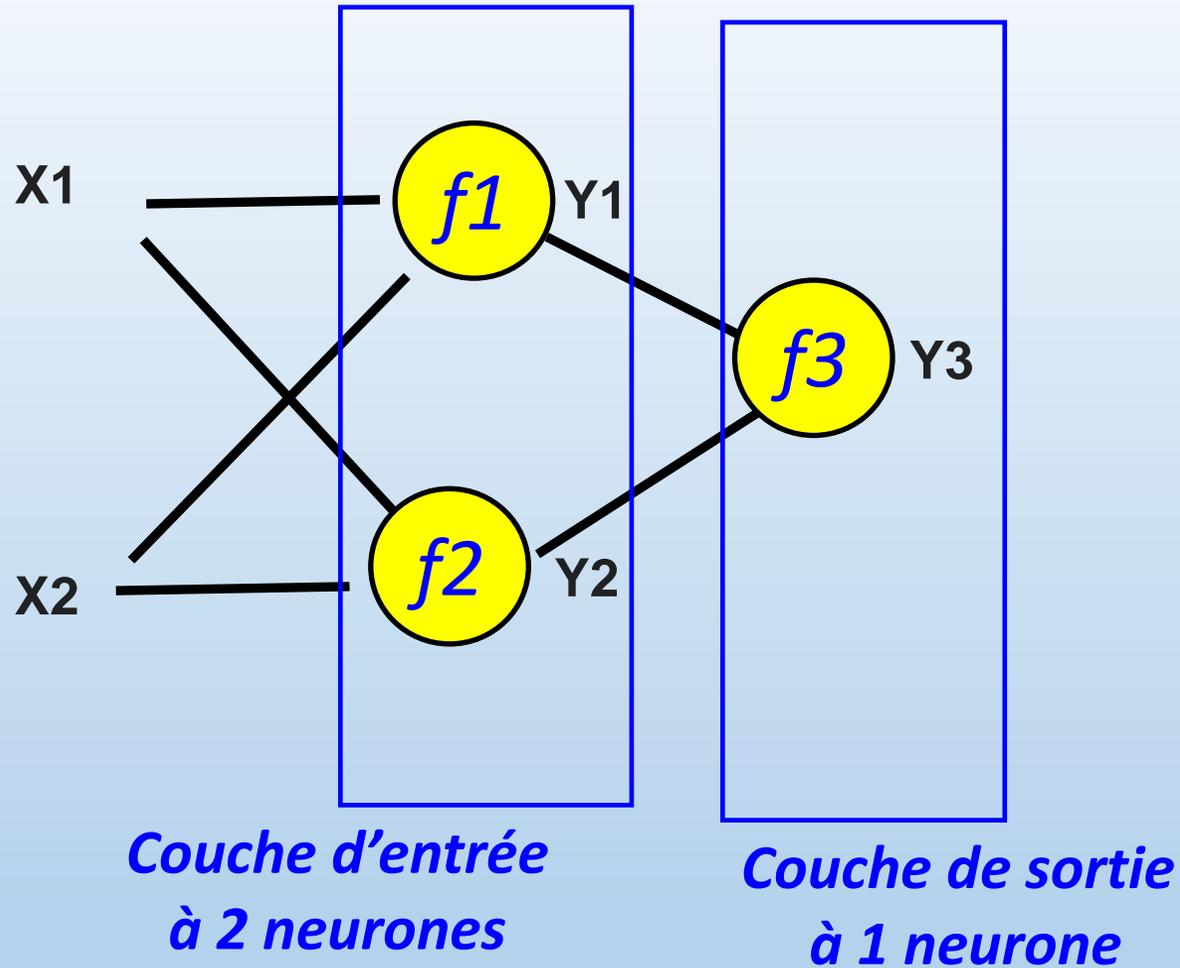
$$Y2 = 0 \text{ sinon}$$



*Modèle non linéaire*

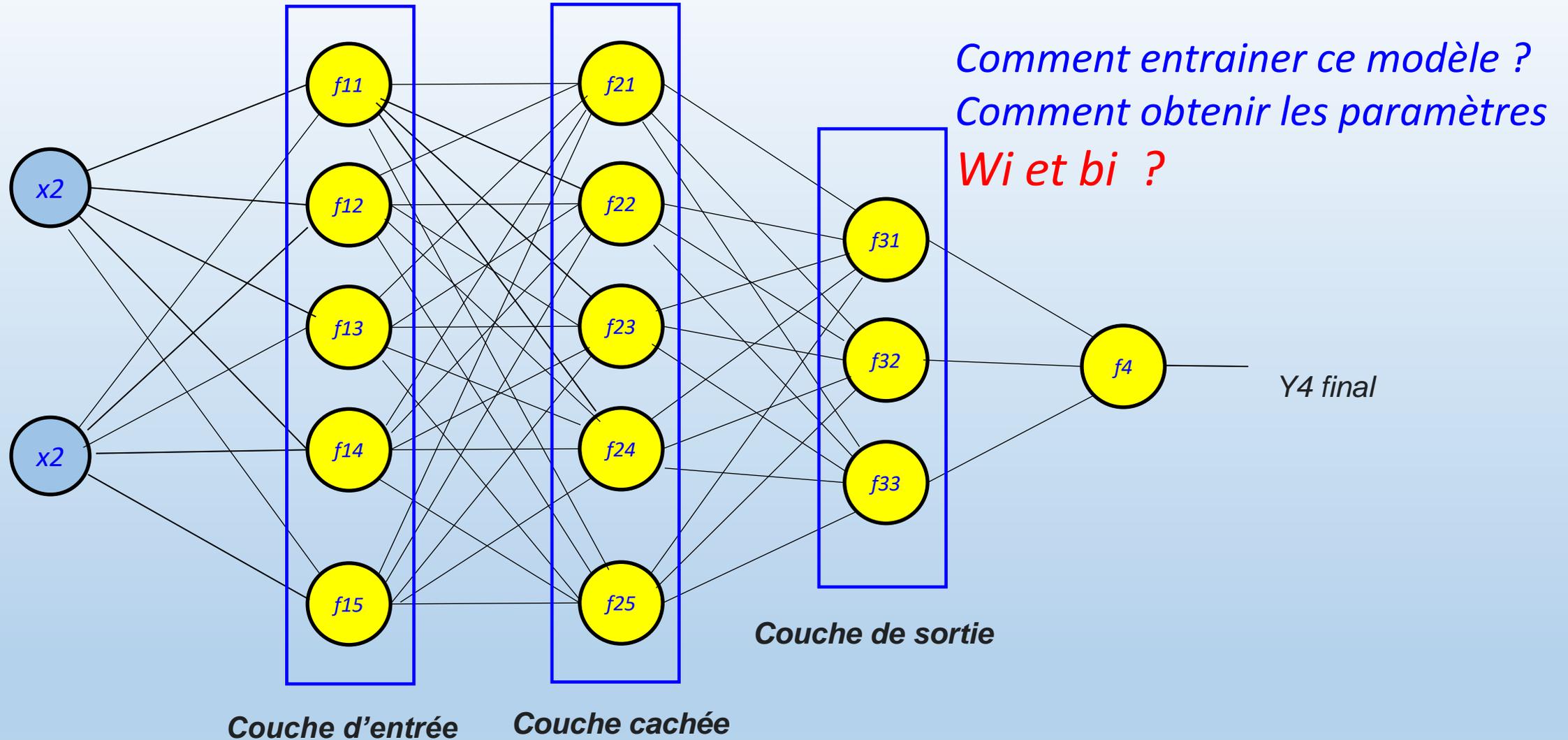
# Le Perceptron Multicouche

Association de 3 neurones



# Le Perceptron Multicouche

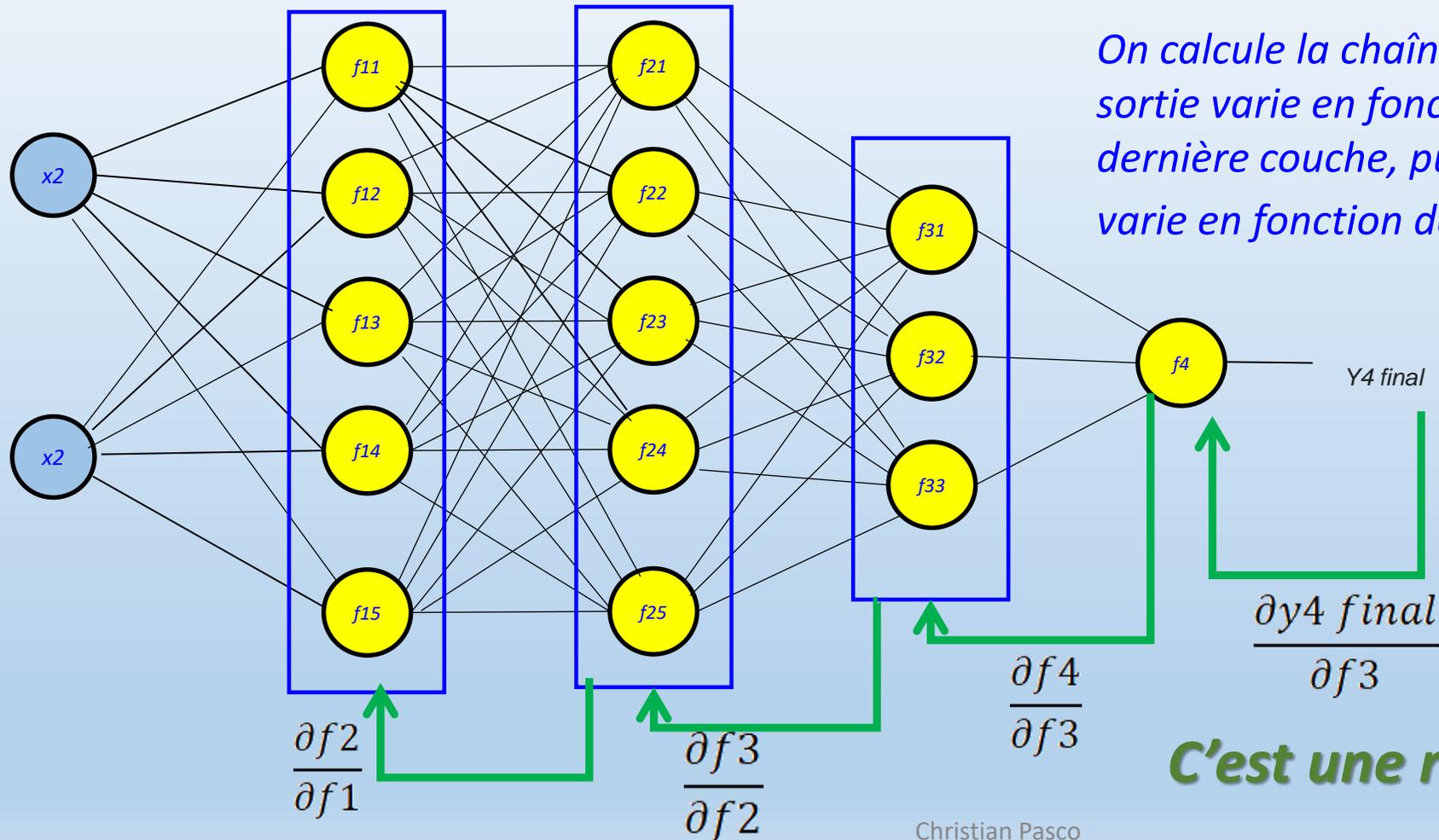
## Généralisation



# Le Perceptron Multicouche

## 1989 La rétropropagation (Backpropagation)

Déterminer comment la sortie varie en fonction des paramètres de chaque couche du modèle



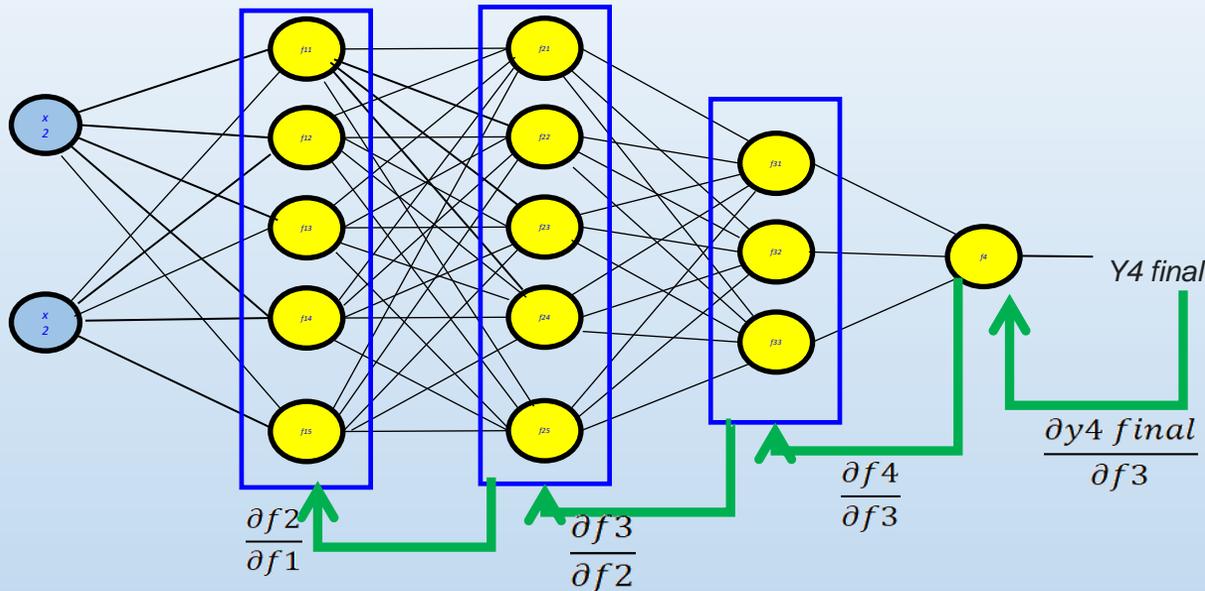
On calcule la chaîne de gradients, i.e. comment la sortie varie en fonction des paramètres de la dernière couche, puis comment la dernière couche varie en fonction de l'avant dernière etc.

**C'est une rétropropagation**

# Le Perceptron Multicouche

La descente de gradient

$$W = W - \alpha \frac{\partial \text{Erreur}}{\partial w}$$



Comment calculer ces erreurs ?

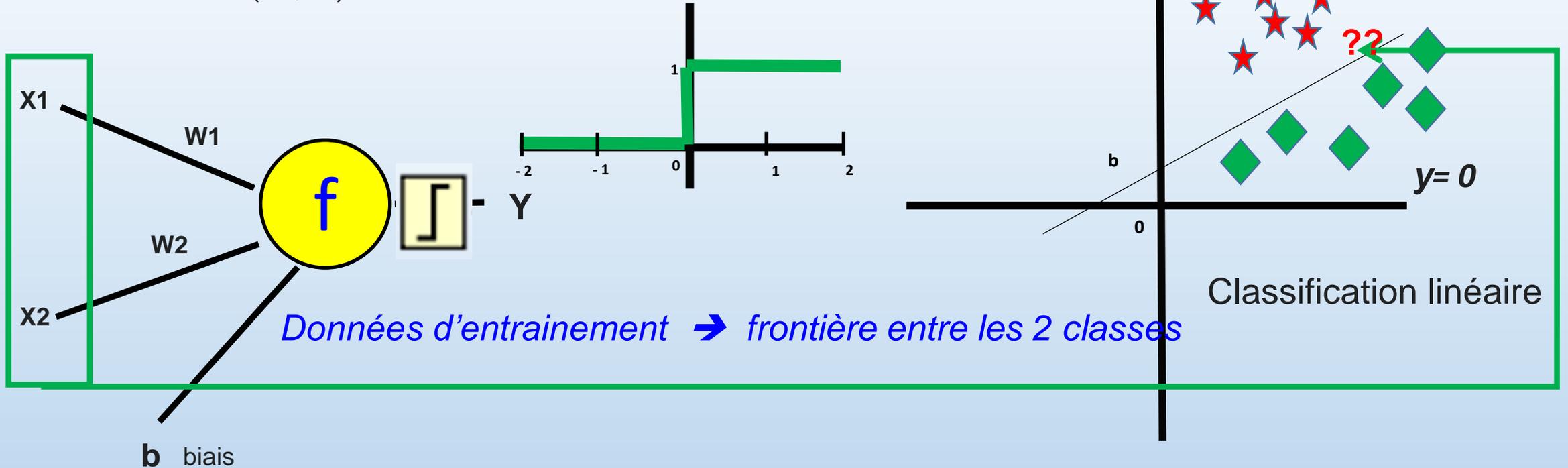
Comment obtenir les paramètres  $W_i$  et  $b_i$  ?

En **1989**, **Rumelhart** McClelland, psychologues, publient largement sur le **Traitement Parallèle Distribué** et la **rétropropagation de gradient**.

*Retour sur le perceptron*  
*Améliorations*

# Retour sur le perceptron

Fonction linéaire  $z(x_1, x_2) = w_1 x_1 + w_2 x_2 + b$



Données d'entraînement → frontière entre les 2 classes

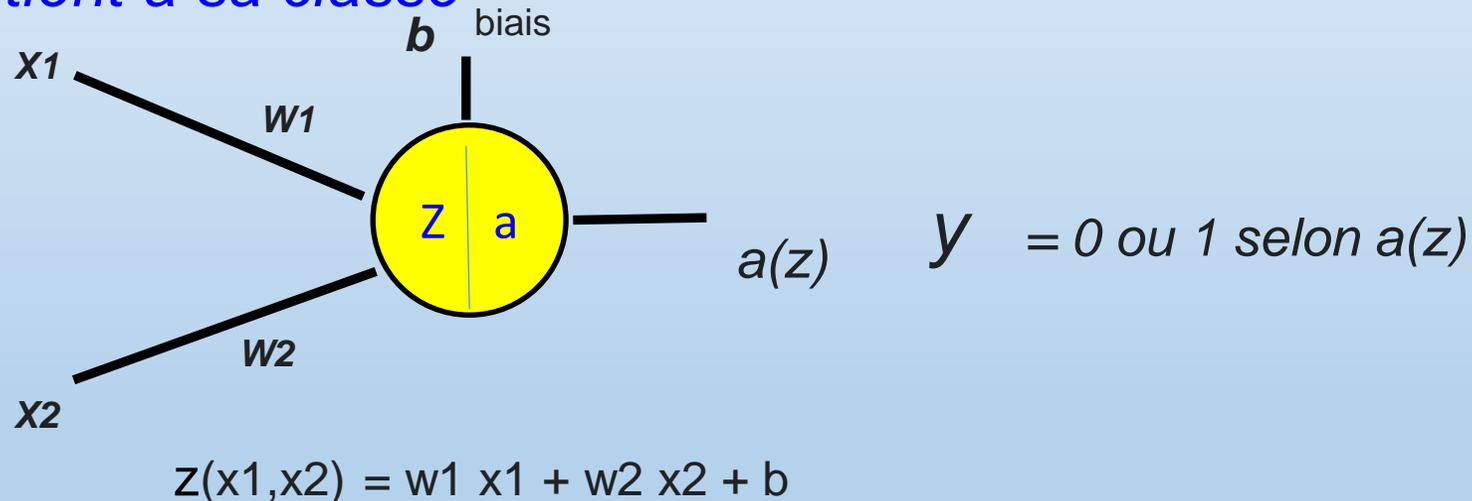
**Prévision : à quelle classe appartient un jeu de données  $x$  ?**  
**Dans  $y=0$  ou  $y=1$  ?**

# Retour sur le perceptron

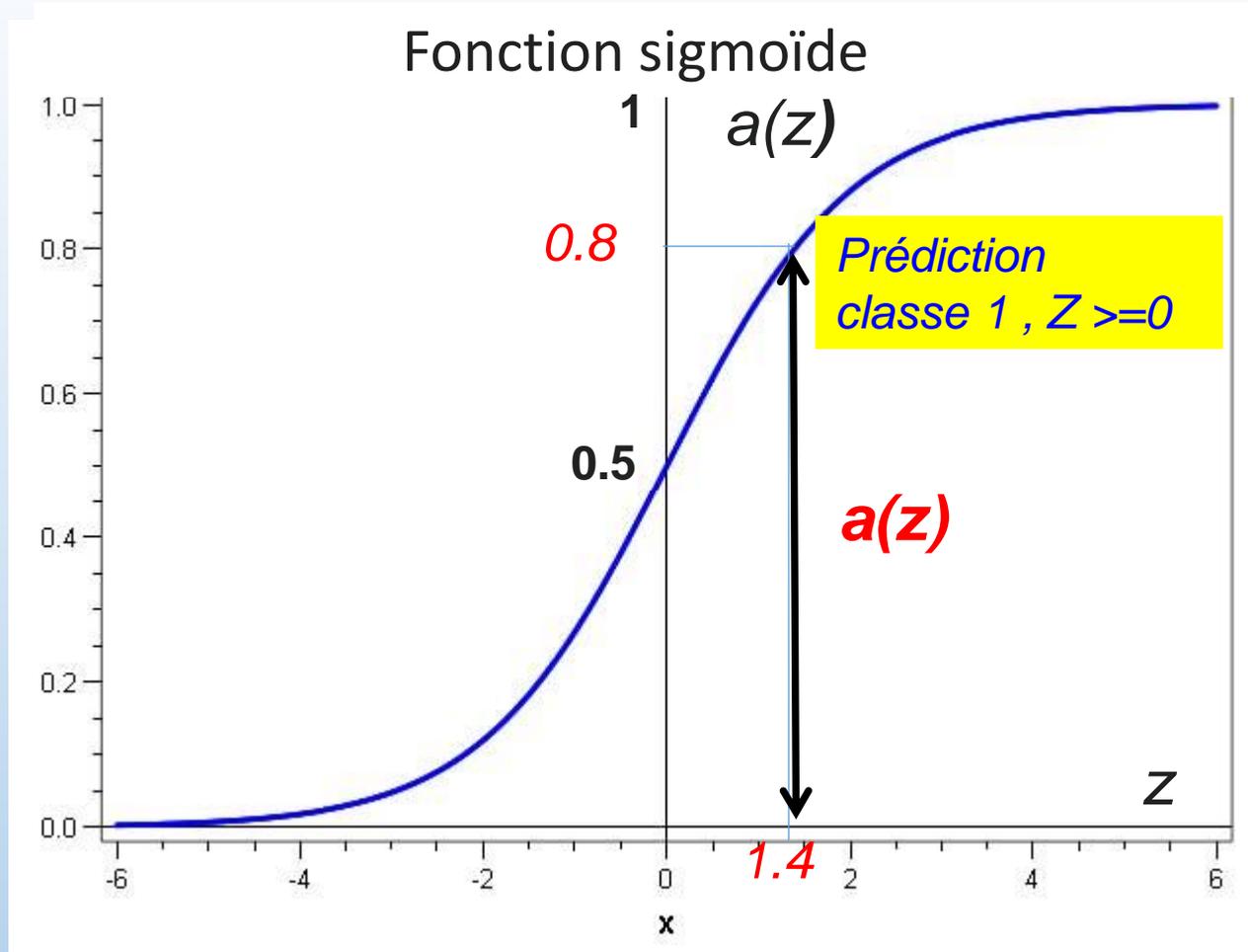
*Pourrait on améliorer le modèle pour qu'il nous fournisse une information plus précise ?*

*En effet, si une donnée se situe à la frontière de séparation, il y a une incertitude sur le résultat binaire 0 ou 1*

*Plus une donnée est éloignée de la frontière de décision, plus il est probable qu'elle appartient à sa classe*



# Amélioration du perceptron



$$z(x_1, x_2) = w_1 x_1 + w_2 x_2 + b$$

Fonction sigmoïde

$$a(z) = \frac{1}{1 + e^{-z}}$$

Pour la sortie  $z$ , la fonction sigmoïde renvoie **la probabilité  $a(z)$**  que la sortie  $z$  appartienne à la classe 1

**La probabilité** que la sortie  $z$  appartienne à la classe 0 est :  $1 - a(z)$

# Amélioration du perceptron

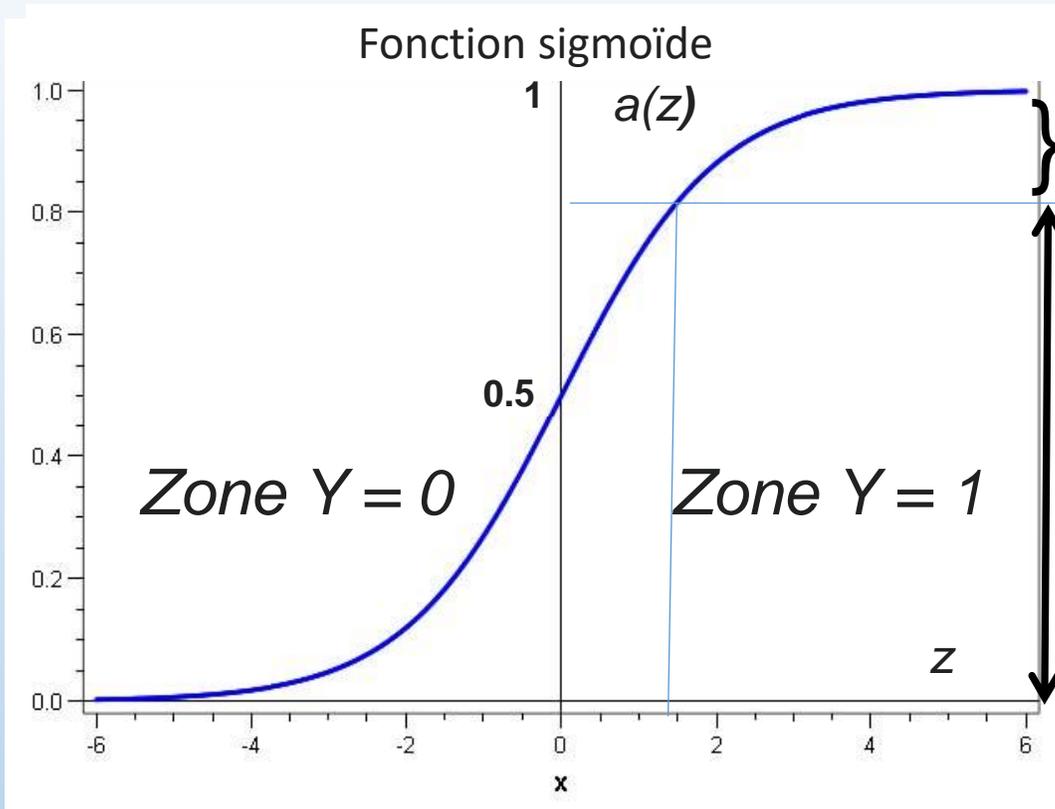
## Probabilités

$$P(Y = 1) = a(z)$$

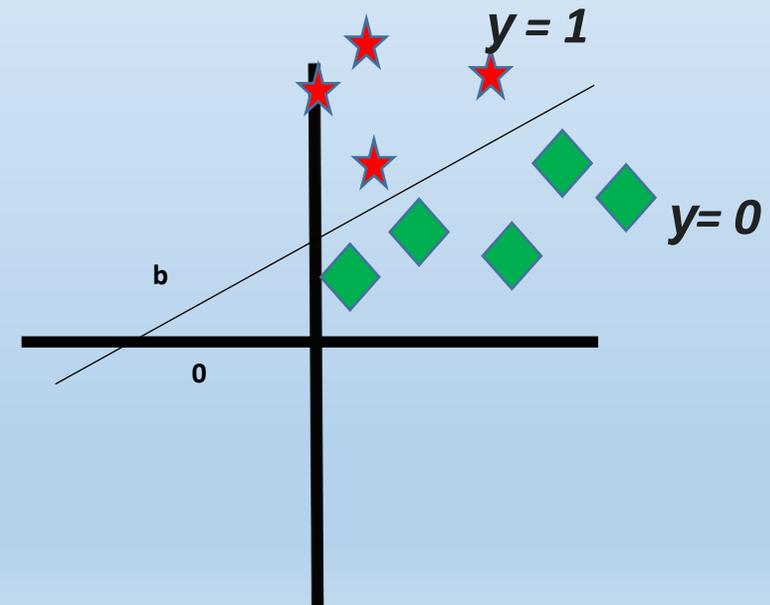
$$P(Y = 0) = 1 - a(z)$$

$$P(Y = y) = a(z)^y (1 - a(z))^{1-y}$$

*Bernoulli*

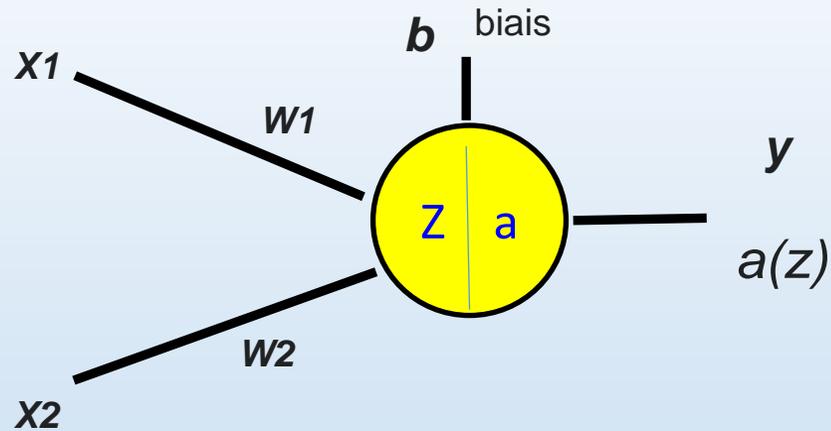


$Y = 1$  si  $a(z) \geq 0,5$   
 $Y = 0$  sinon



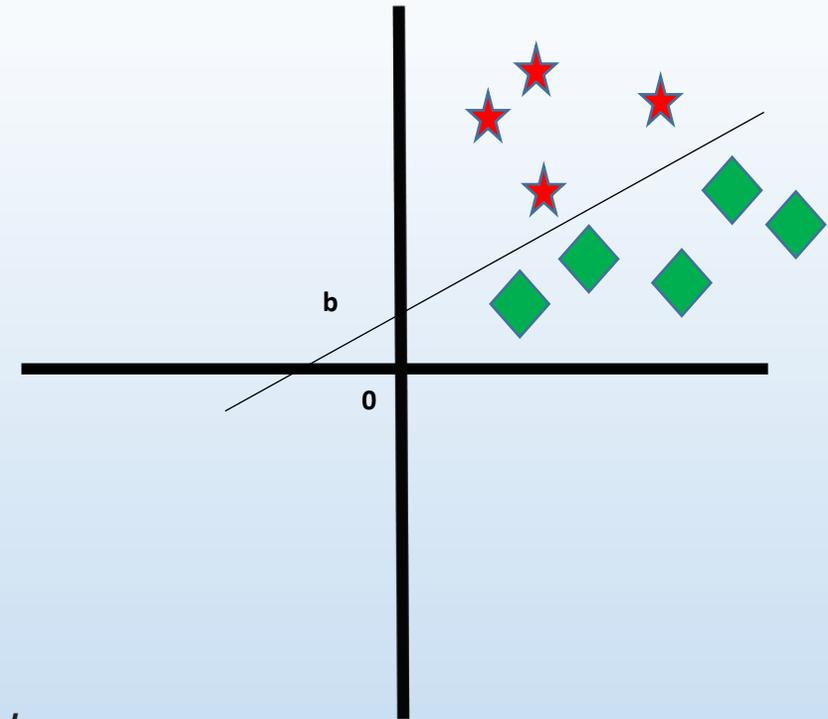
# Amélioration du perceptron

## Résumé



$$Y = 1 \text{ si } a(z) \geq 0,5$$

$$Y = 0 \text{ sinon}$$



Perceptron monocouche

Fonction linéaire  $z$   $z(x1,x2) = w1 x1 + w2 x2 + b$

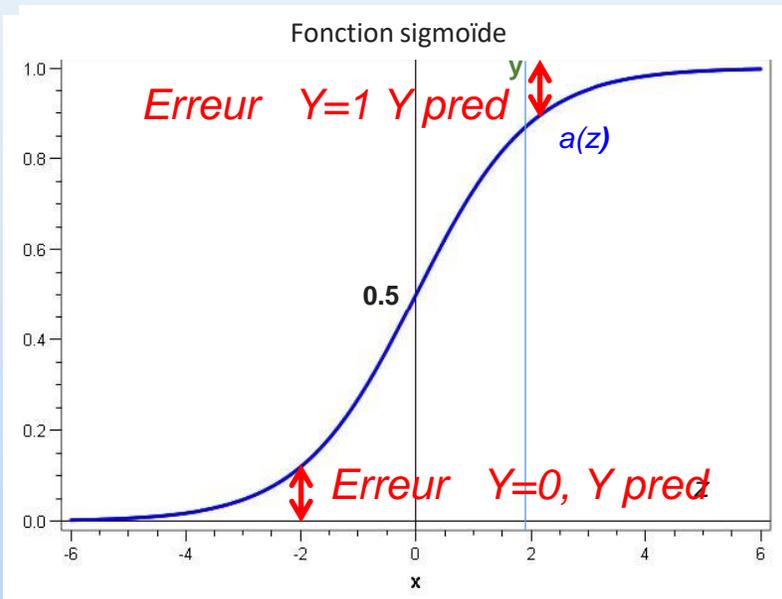
Fonction d'activation  $a$   $a(z) = \frac{1}{1 + e^{-z}}$  Sigmoide

Retourne une probabilité  $P$   $P(Y = y) = a(z)^y X (1 - a(z))^{1-y}$  Loi de Bernouilli

# Calcul de l'erreur du modèle

## Fonction de coût

Il s'agit maintenant de calculer l'erreur entre le résultat produit par la fonction d'activation  $a(z)$  et la valeur réelle  $Y$



En Machine Learning, c'est le rôle de la fonction de coût

On a

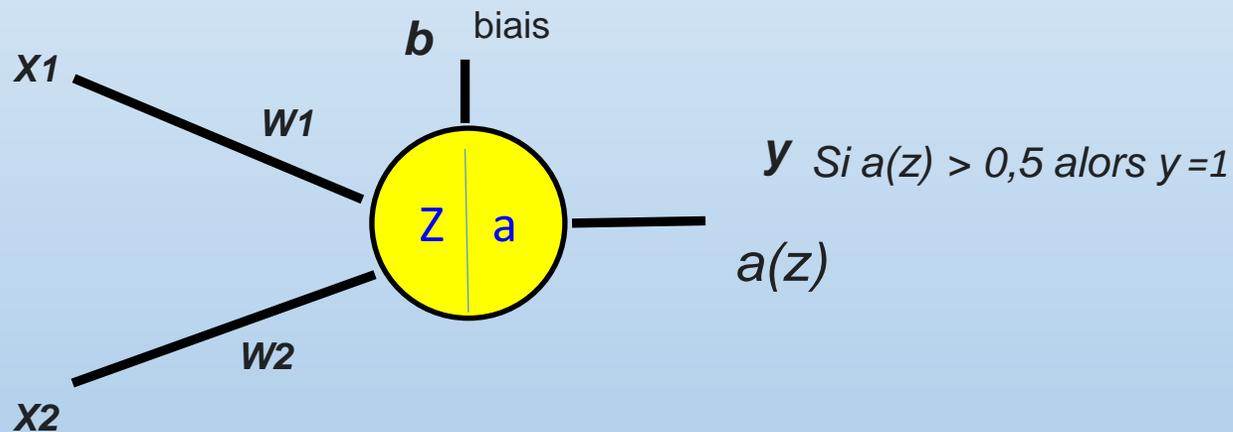
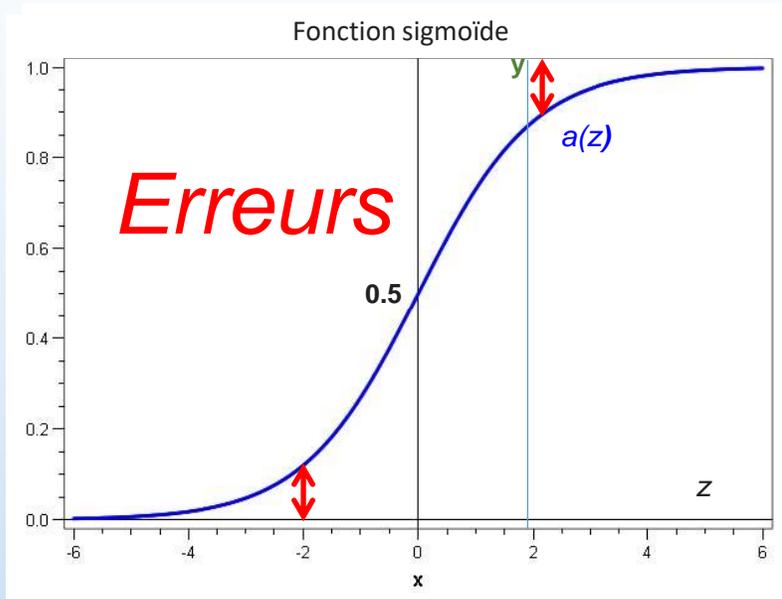
- un jeu de  $m$  données étiquetées  $Y=0$  ou  $Y=1$
- les sorties  $a(z)$
- les erreurs

# Calcul de l'erreur du modèle

## Fonction de coût

En machine Learning on utilise une fonction de coût

- $m$  = nombre de données
- $y_i$  = sortie attendue n°  $i$
- $a_i$  = sortie produite n°  $i$



# Fonction de coût

## Vraisemblance

La vraisemblance indique la plausibilité du modèle vis-à-vis de vraies données

Si pour une donnée étiquetée de classe 1, le modèle sort une probabilité de 80%, alors le modèle est **vraisemblable** à 80%

Pour qualifier la plausibilité d'un modèle, on **multiplie les probabilités** :

**Likelihood**  
Vraisemblance

$$L = \prod_{i=1}^m P(Y = y_i) \quad L = \prod_{i=1}^m a_i^{y_i} \times (1 - a_i)^{1-y_i}$$



*Loi de Bernouilli*

**Produit des probabilités que  $y_i$  soit la sortie attendue**

# Fonction de coût

## Vraisemblance et Log Loss

On va chercher à **maximiser la vraisemblance**

Pour les calculs il va être plus facile d'utiliser le log : additions au lieu de multiplications:

$$L = \prod_{i=1}^m a_i^{y_i} \times (1 - a_i)^{1-y_i}$$

### Log Likelihood

Log de la Vraisemblance

$$LL = \log \prod_{i=1}^m a_i^{y_i} \times (1 - a_i)^{1-y_i}$$

Log Likelihood  $LL = \sum_{i=1}^m y_i \log a_i + (1 - y_i) \log(1 - a_i)$  À maximiser

Log Loss  $L = -\frac{1}{m} \sum_{i=1}^m y_i \log(a_i) + (1 - y_i) \log(1 - a_i)$  À minimiser

*Algorithme  
de descente de gradient  
Minimiser les erreurs du modèle*

# Algorithme de descente de gradient

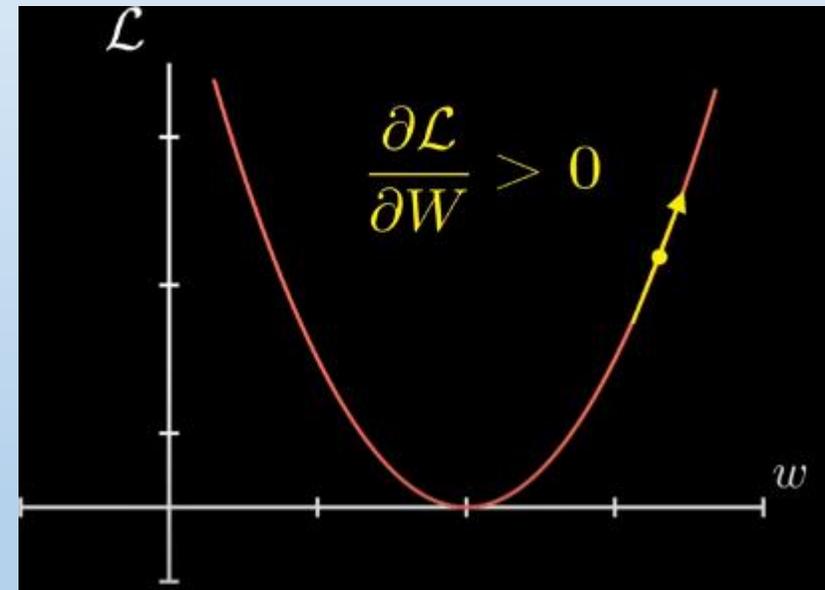
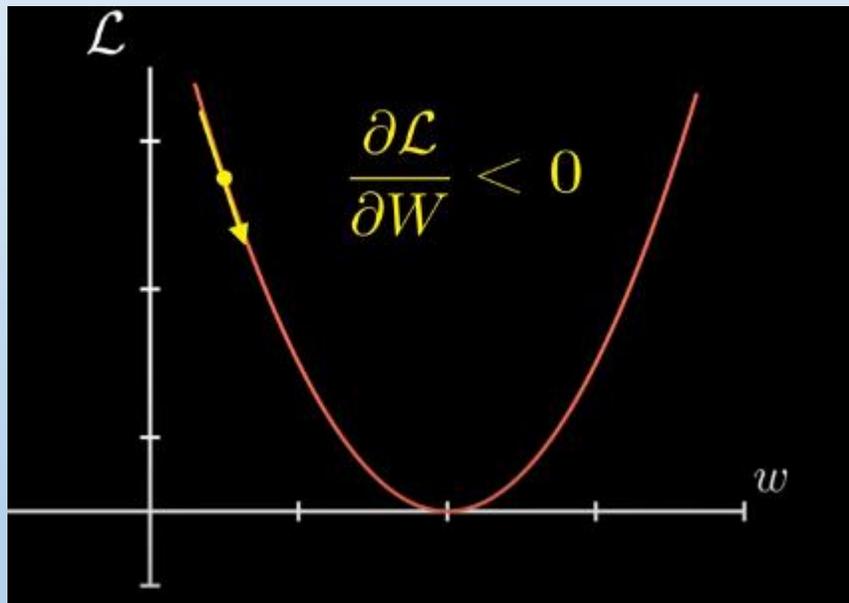
Minimiser les erreurs du modèle

On va donc chercher à minimiser les erreurs du modèle en minimisant la fonction de coût **Log Loss**

Il faut donc déterminer comment elle **varie** en fonction des différents paramètres

**w et b**

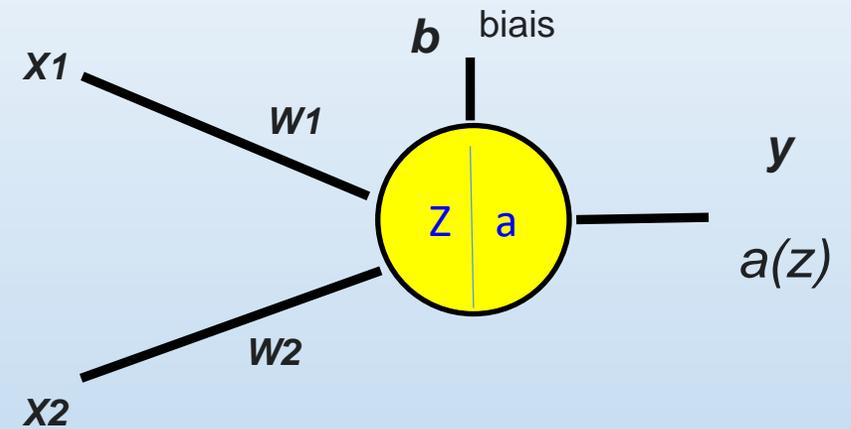
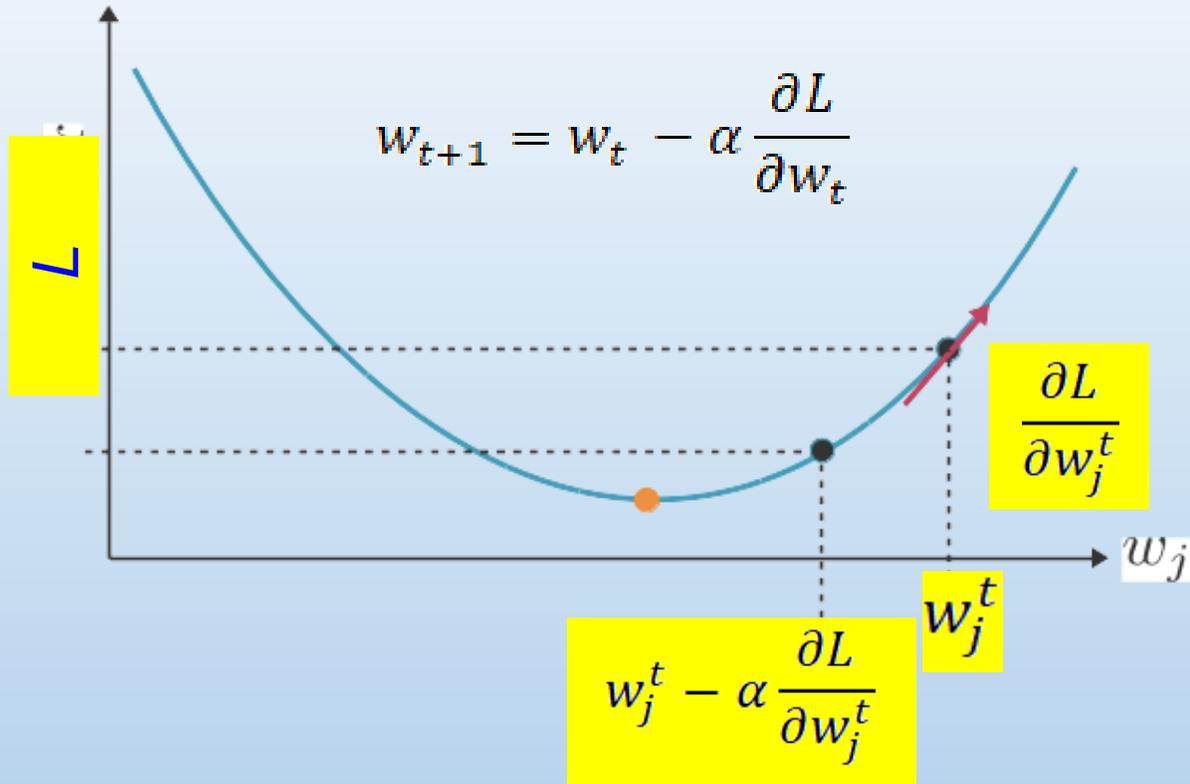
C'est pourquoi on calcule un gradient ou la dérivée de la fonction de coût Log Loss:



# Algorithme de descente de gradient

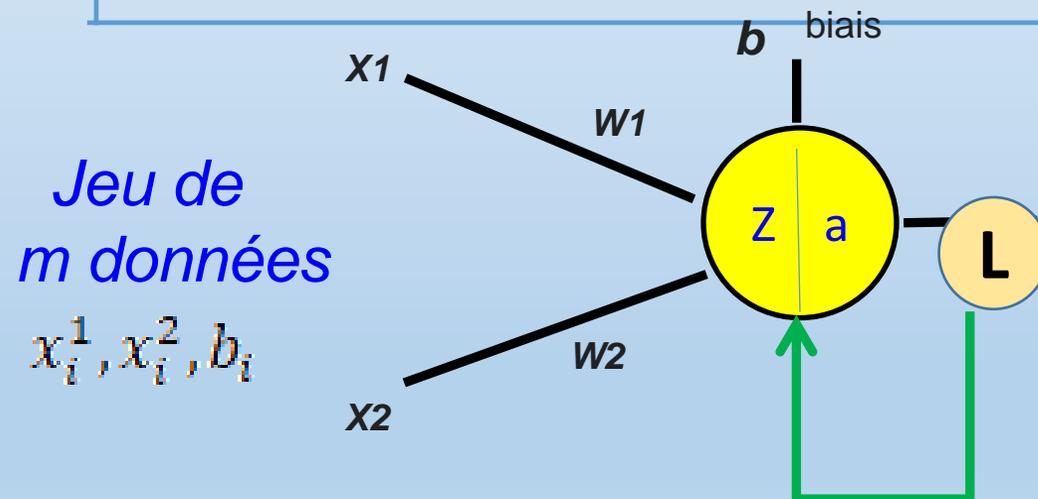
Minimiser les erreurs du modèle

Minimisation de la fonction de coût **Log Loss**



# Résumé

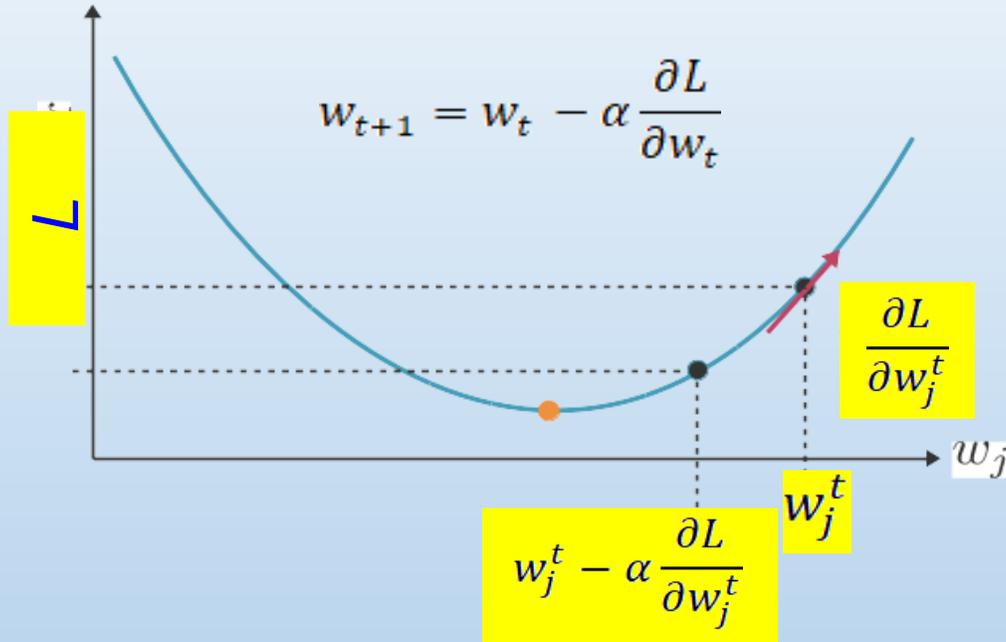
<i>Nombre de jeux d'entrée</i>	$m$ vecteurs $(x_1, x_2)$
<i>Intégration des entrées</i>	$z(x_1, x_2) = w_1 x_1 + w_2 x_2 + b$
<i>Fonction d'activation</i>	$a(z) = \frac{1}{1 + e^{-z}}$ Sa dérivée est $a(1-a)$
<i>Fonction de coût</i>	$L = -\frac{1}{m} \sum_{i=1}^m y_i \log(a_i) + (1 - y_i) \log(1 - a_i)$
<i>Descente de gradient</i>	$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}$



# Algorithme de descente de gradient

Minimiser les erreurs du modèle

## Calcul des gradients



$$\frac{\partial L}{\partial w_1} = \frac{1}{m} \sum_{i=1}^m (a_i - y_i) x_1$$

$$\frac{\partial L}{\partial w_2} = \frac{1}{m} \sum_{i=1}^m (a_i - y_i) x_2$$

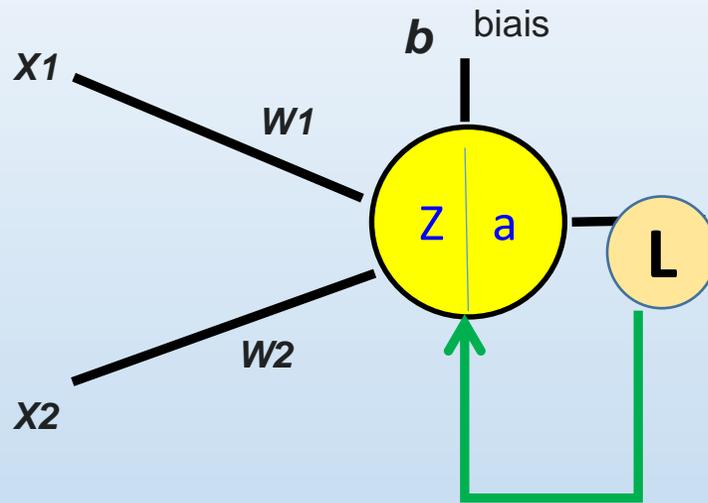
$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum_{i=1}^m (a_i - y_i)$$

Rappel de l'algorithme initial du Perceptron 1957 :  
 $W = W + a (Y_{true} - Y) X$

# En résumé

Jeu de  
 $m$  données

$x_i^1, x_i^2, b_i$



Ajustement des paramètres  $w1$ ,  $w2$  et  $b$

1. Forward Propagation
2. Cost Function
3. Backward Propagation
4. Gradient Descent

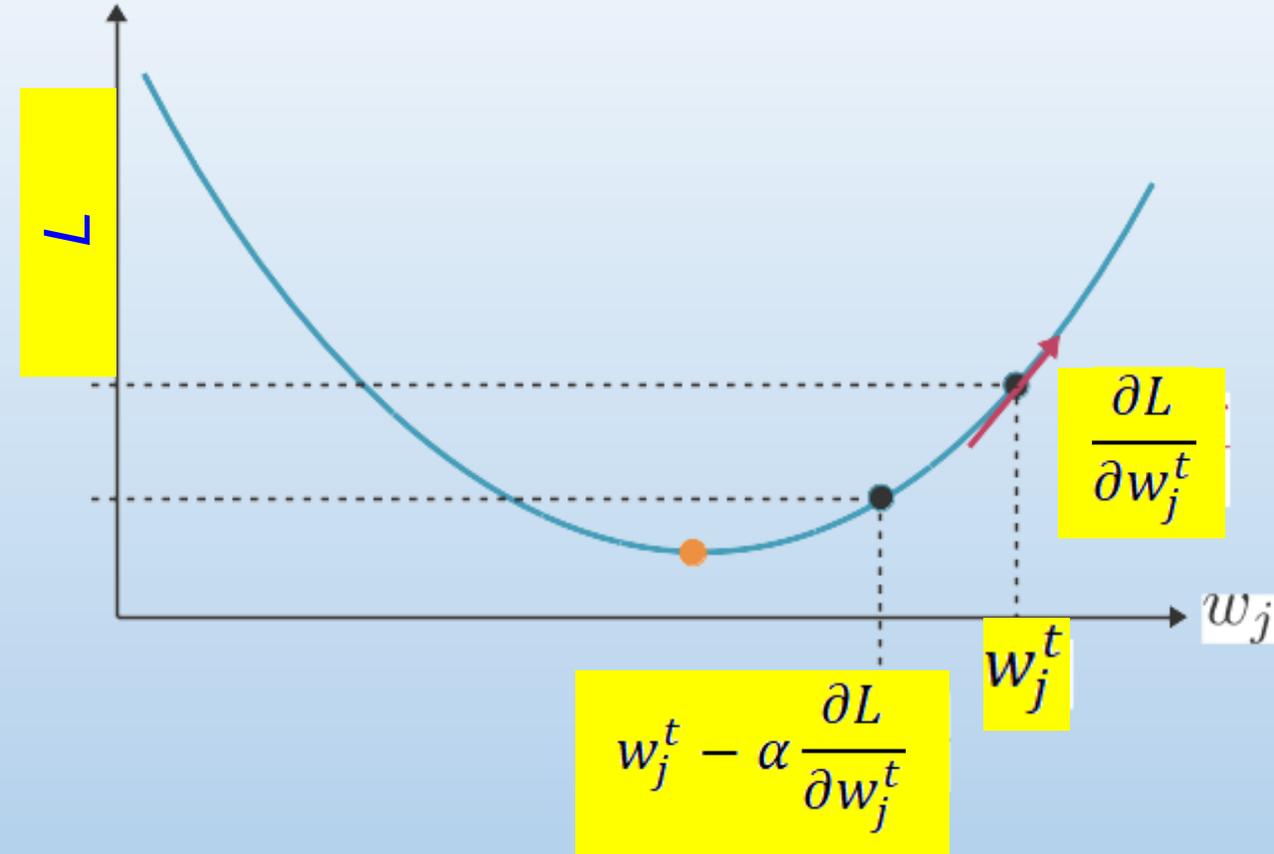
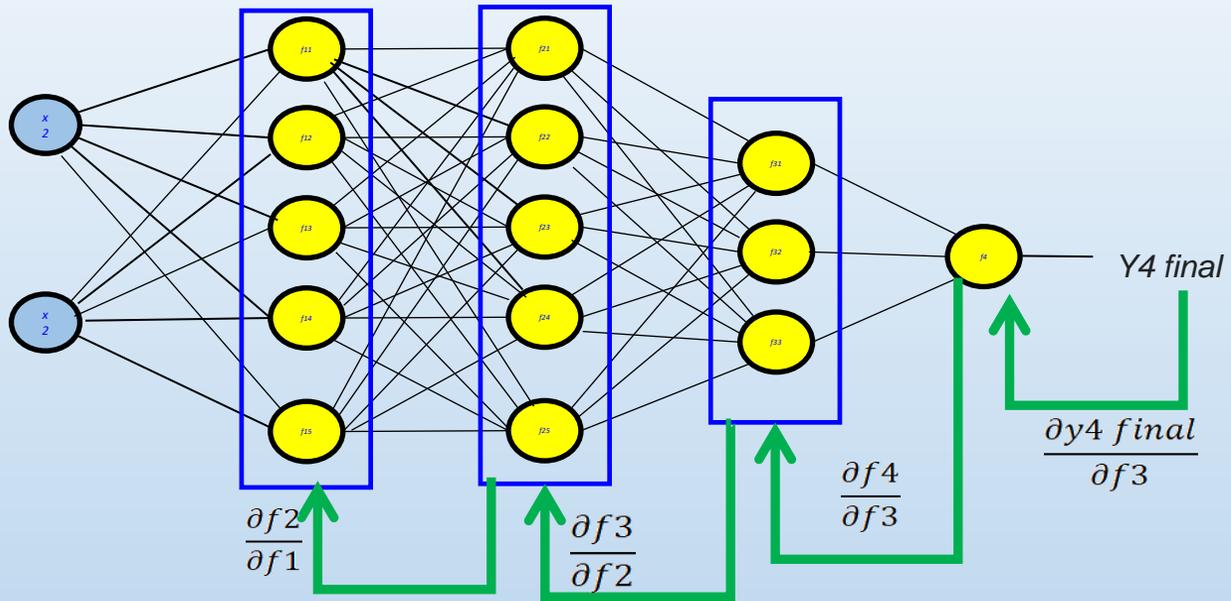


*Retour au multicouche*

# Extension au Perceptron Multicouche

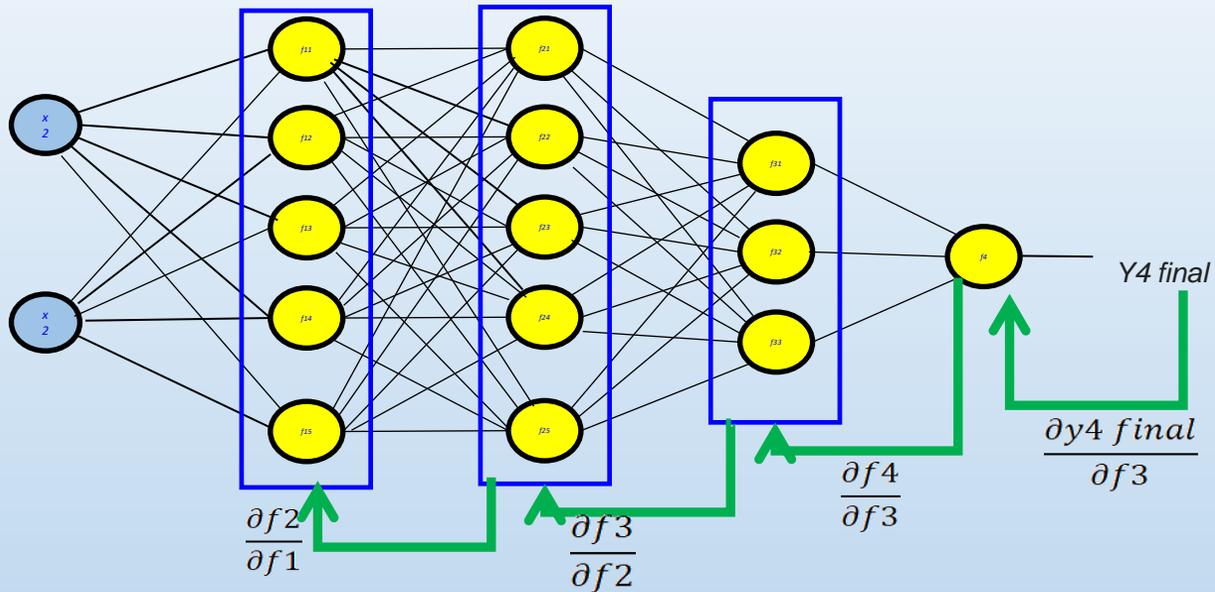
La descente de gradient

$$w = w - \alpha \frac{\partial \text{Erreur}}{\partial w} \quad \longrightarrow \quad w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}$$



# En résumé

Jeu de  
 $m$  données

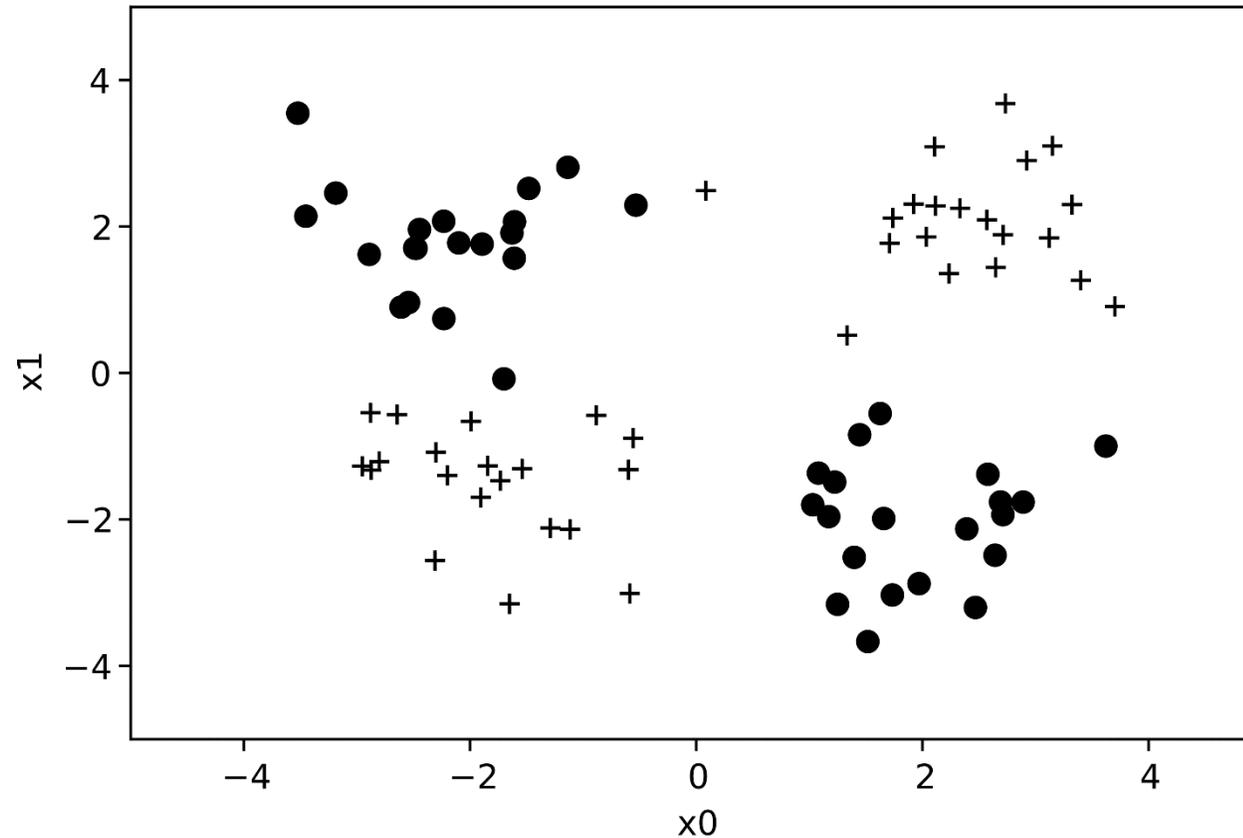


Ajustement des paramètres  $w$  et  $b$

1. Forward Propagation
2. Cost Function
3. Backward Propagation
4. Gradient Descent

# Illustration

*Données non séparables linéairement*



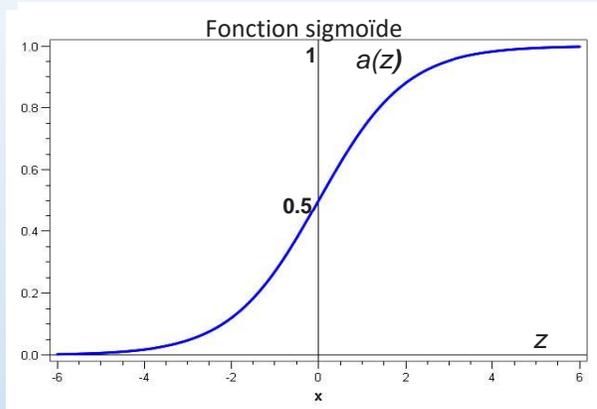
[https://s3-eu-west-1.amazonaws.com/course.oc-static.com/courses/5801891/xorgsep\\_anim.gif](https://s3-eu-west-1.amazonaws.com/course.oc-static.com/courses/5801891/xorgsep_anim.gif)

# *Evolutions*

*Au fil du temps le modèle continua d'évoluer*

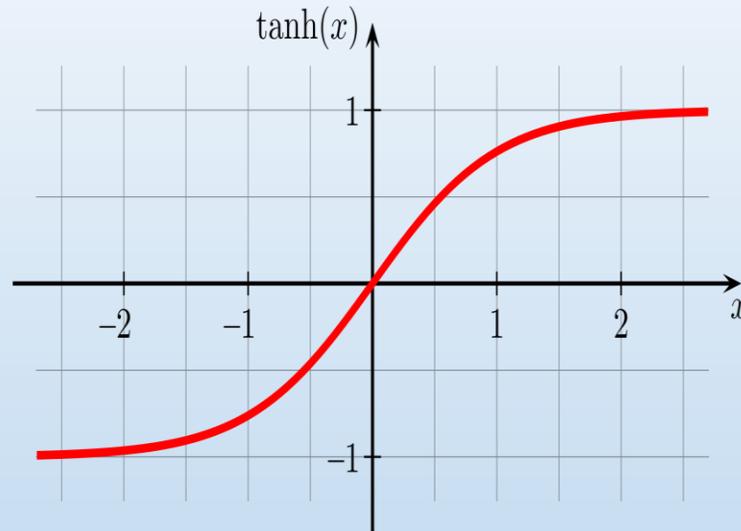
# Fonctions d'activation

## Utilisation d'autres fonctions d'activation



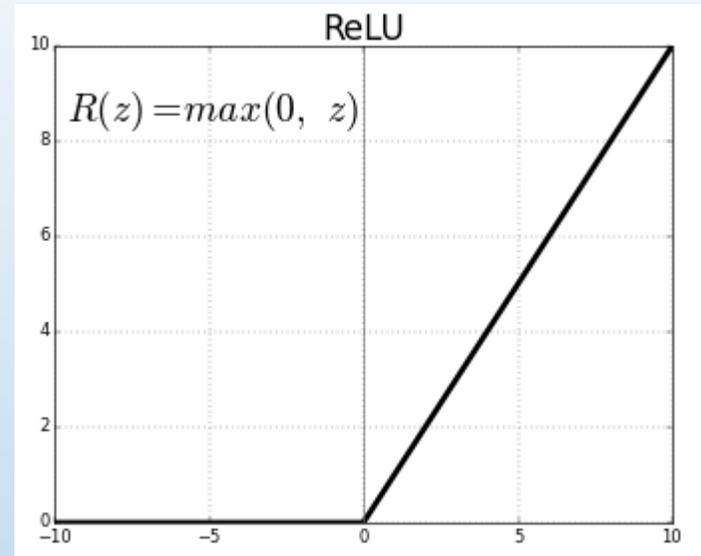
Fonction sigmoïde

$$a(z) = \frac{1}{1 + e^{-z}}$$



Fonction tangente hyperbolique

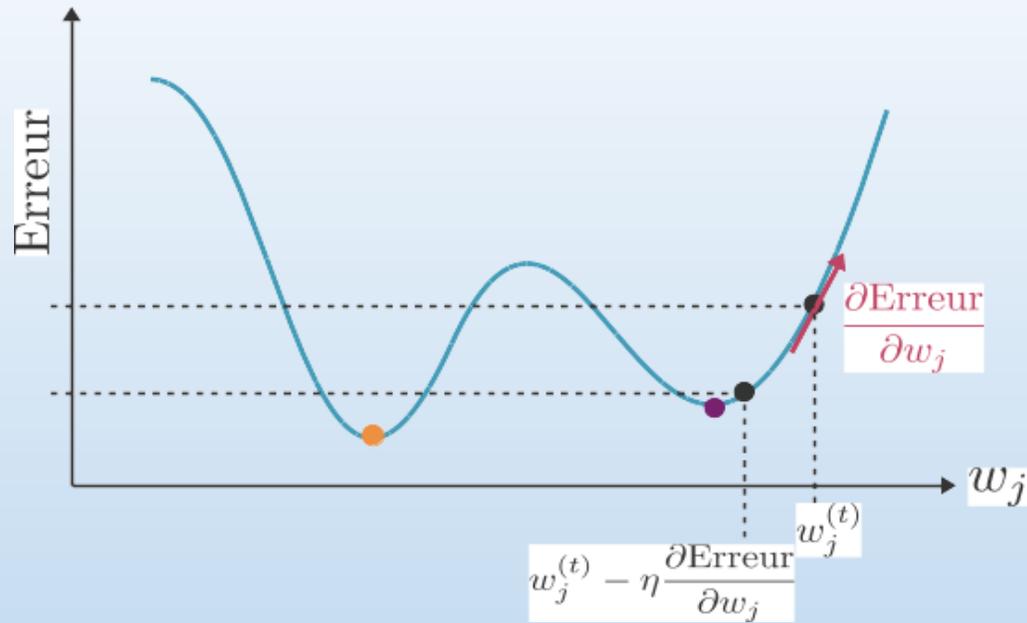
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



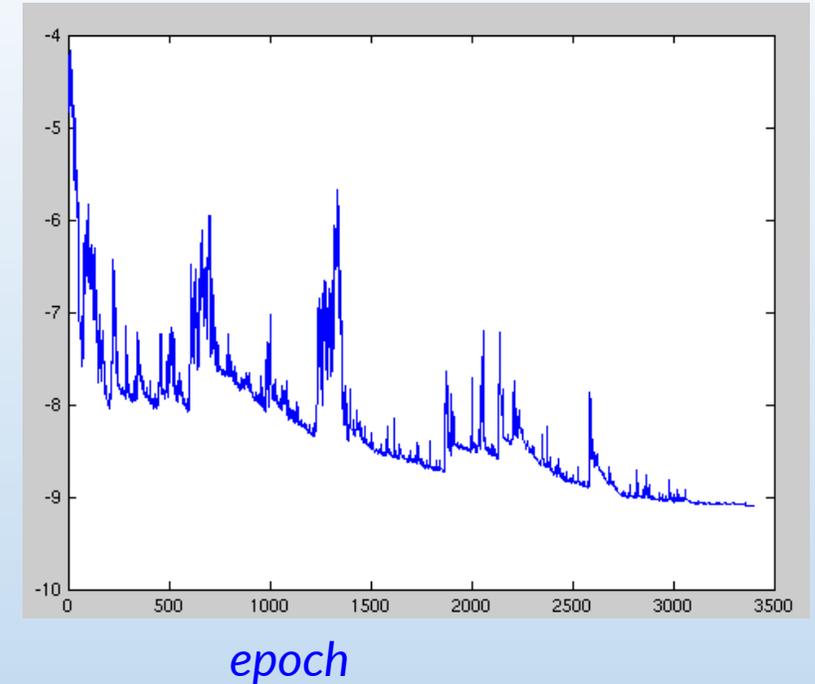
Fonction ReLu  
Rectified Linear Unit

# Descente de gradient stochastique

Stochastique : aléatoire, lié au hasard



Loss



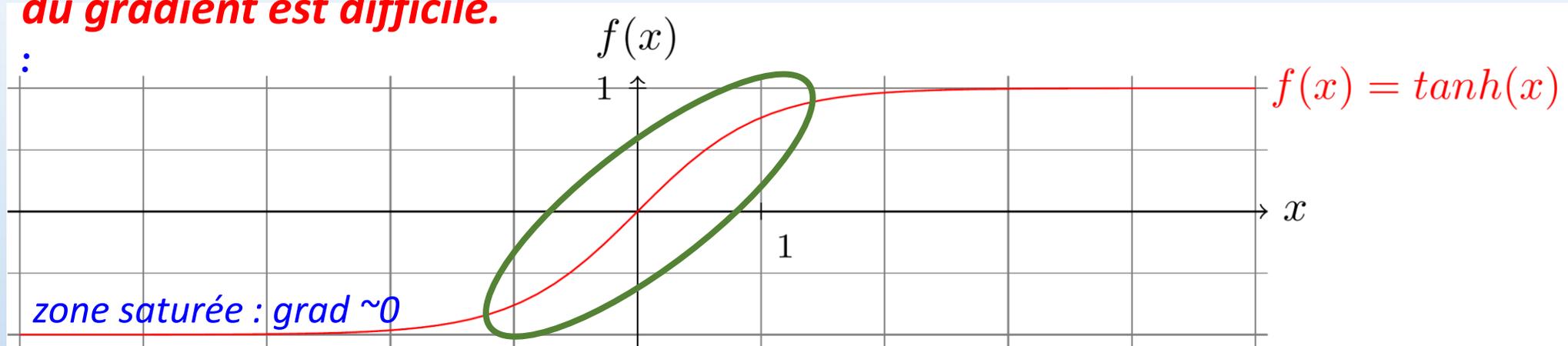
## Deux difficultés :

- Minimum local → tirage aléatoire et itérations
  - Quantité de données telle que le calcul) va devenir vite très coûteux en machine
- On travaille à chaque epoch sur des échantillons (lots), choisis de façon aléatoire,

...

# Gradient évanescent Normalisation par lots

Plus on rajoute de couches, plus l'apprentissage par rétropropagation du gradient est difficile.



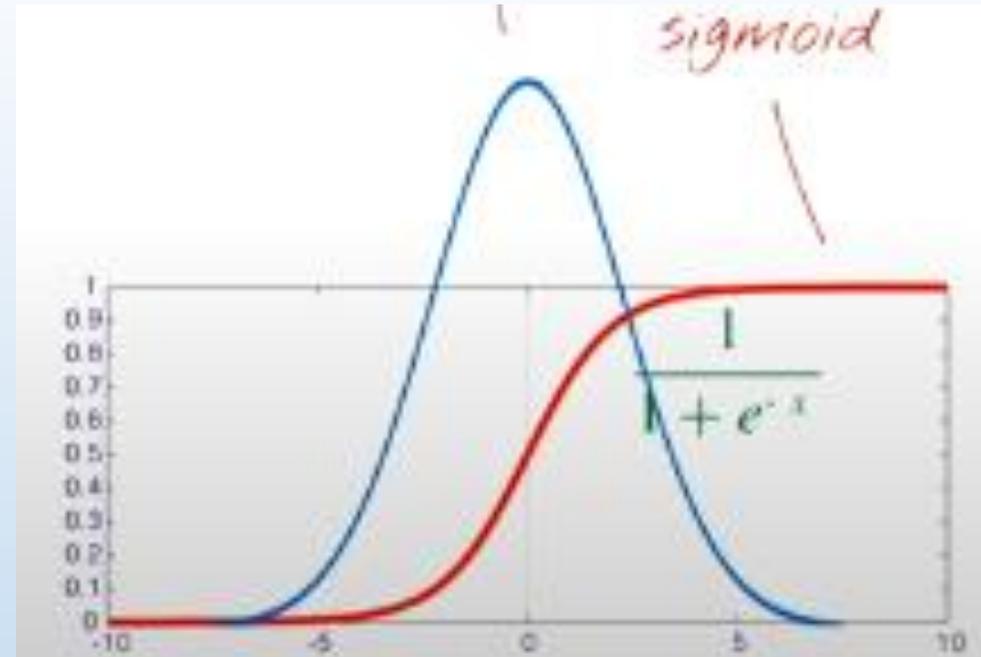
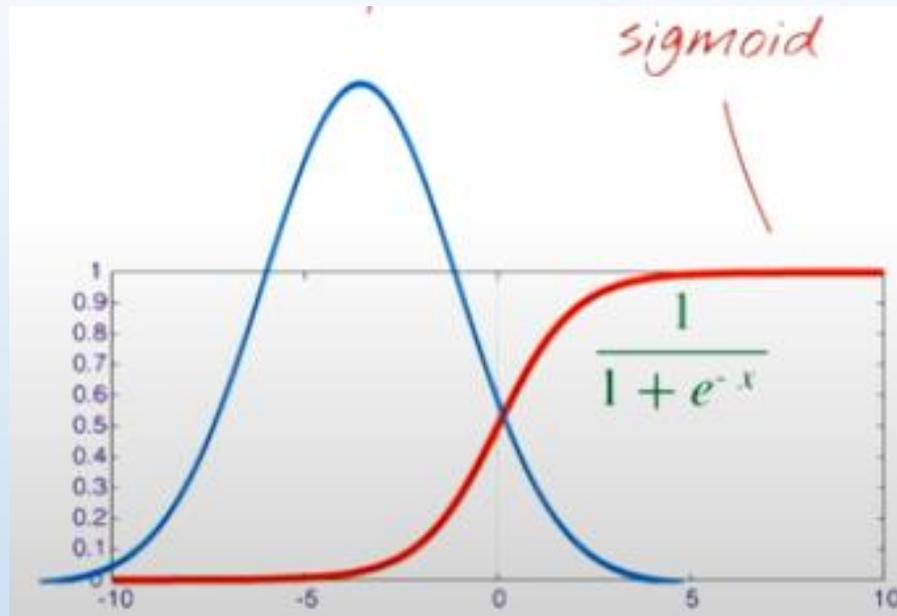
→ **Une réponse : Couche Batch Normalization**

Réduction du phénomène d'évanescence du gradient

Permet de maintenir les exemples d'un batch (lot) dans la zone non saturée d'une unité.

Par exemple, la zone proche de 0 dans la fonction tangente hyperbolique

# Normalisation par lots



## Couche Batch Normalization

*Pour les sorties de chaque couche de neurones :*

*On calcule la moyenne et les écarts-types d'un lot*

*On fait l'opération suivante pour chaque sortie :*

*Valeur sortie – valeur moyenne*

*Écart type*

*Applicable aux **sorties pondérées** avant la fonction d'activation*

# Démonstrations - Exemples

*Tensorflow : plate-forme Open Source de bout en bout*

.

# Démonstrations - Exemples

## *Utilisation de Tensorflow*

*En pratique, on arrive à trouver des paramètres satisfaisants, mais il n'y a pas vraiment de cadre théorique pour formaliser tout cela, c'est affaire d'expérience :*

- choisir le bon nombre de neurones, le bon nombre de couches de neurones,*
- quels calculs préliminaires ajouter en entrée ; par exemple multiplier les entrées pour augmenter les degrés de liberté permettant de faire le calcul*

*Ce genre de techniques permet d'obtenir des résultats impressionnants en pratique, comme en reconnaissance de la voix ou d'objets dans une image (voir les [vidéos](#) du cours de Yann Le Cun au Collège de France).*

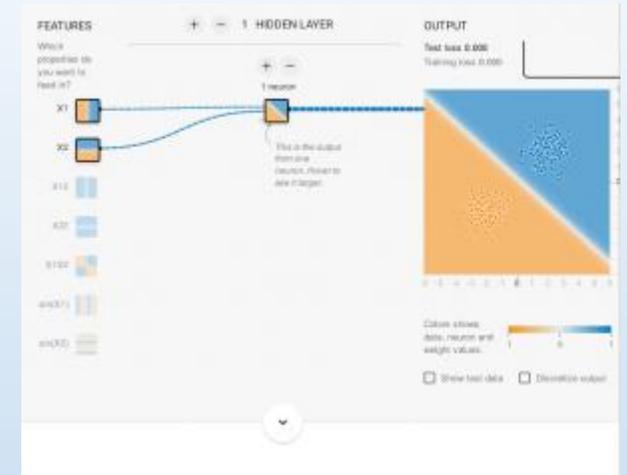
# Démonstrations - Exemples

## Tensorflow

*Classer deux populations, qui sont déjà groupées en deux blocs facilement séparés par un ligne, alors c'est très facile : il suffit de trois neurones.*

*Les deux premiers neurones de la couche d'entrée calculent pour la position horizontale et verticale et le neurone de sortie combine ses informations pour faire une sortie oblique.*

*On parle de problème linéaire pour décrire un tel problème de classification dont la solution est une simple séparation par une surface plate.*

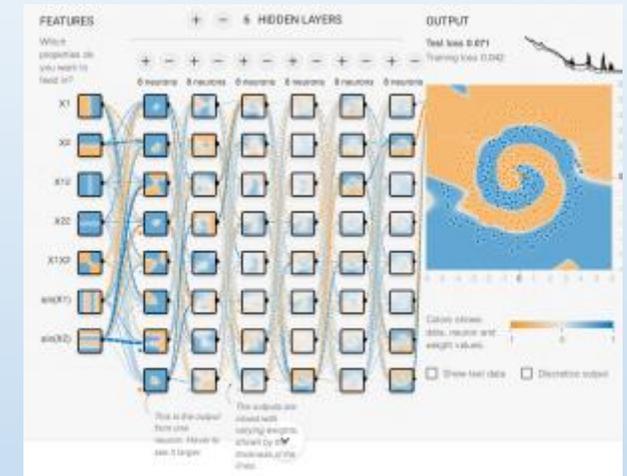


# Démonstrations - Exemples

## Tensorflow

*Mais si les données sont complètement intermêlées comme dans le cas de données en spirale alors il faut beaucoup plus de couches de calcul avec plus de neurones.*

*On voit alors comment au fil du calcul dans les couches de l'architecture, chaque neurone code pour un aspect de la forme à trouver, basique dans les couches basses, plus sophistiqué dans les couches plus hautes. Beaucoup de calculs pour une simple paire de spirales ! Mais ce qui est remarquable c'est que cette foule de calculs élémentaires a pu coder approximativement un objet non trivial, en s'adaptant aux données, sans avoir eu à donner des informations à priori sur ces formes spiralées.*

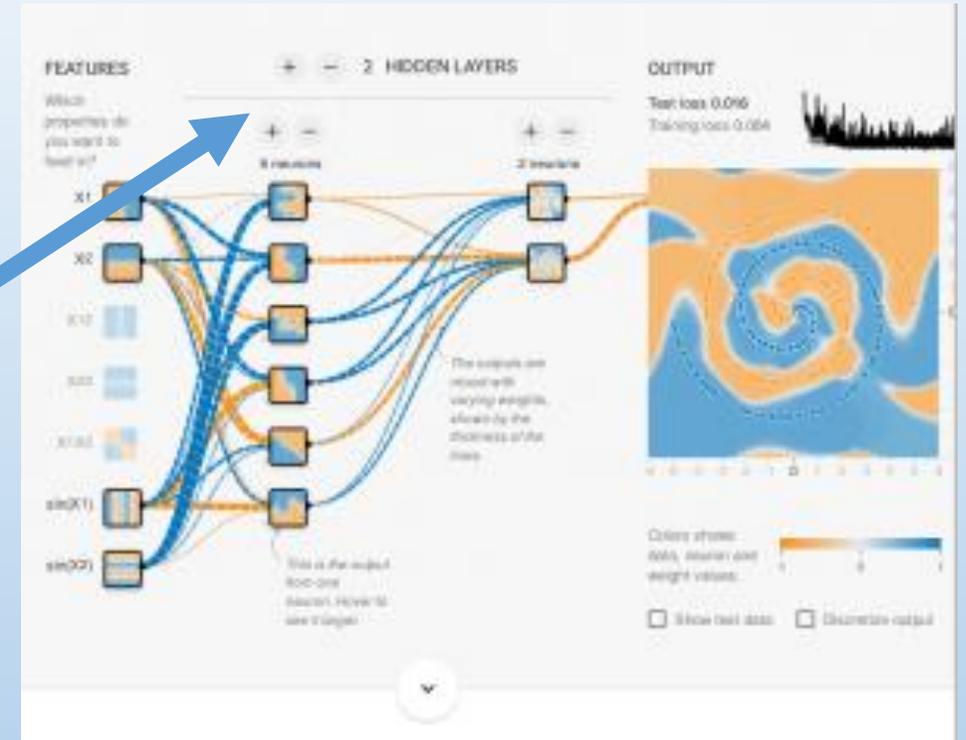


# Démonstrations - Exemples

Tensorflow

*Faut-il tous ces neurones pour résoudre ce problème de classification de données ?*

*Essayez et testez plusieurs configurations, à l'aide des boutons*



# *Jouons avec les neurones*

[Jouez avec les neurones de la machine — Pixees](#)

*Fin*