Réseaux de neurones

LIONEL PREVOST

HEAD OF LEARNING, DATA & ROBOTICS LAB – ESIEA

lionel.prevost@esiea.fr

Sommaire

Réseaux de neurones et apprentissage supervisé

- Du neurone naturel au neurone artificiel
- Le perceptron

- **Les réseaux de neurones monocouches**
- **L'Adaline**

- Les réseaux multicouches
- La rétro-propagation

Bilan et limites du perceptron

Neurone à **fonction seuil**, capable de traiter UNIQUEMENT :

- Des données **linéairement séparables**
- Des problèmes **binaires** (2 classes)

Q **intuitivement**, comment traiter un problème multiclasse ? (→ solution)

(…)

Paramètre à optimiser : vecteur des poids ω

Fonction objectif :

« nombre d'exemples mal classés = 0»

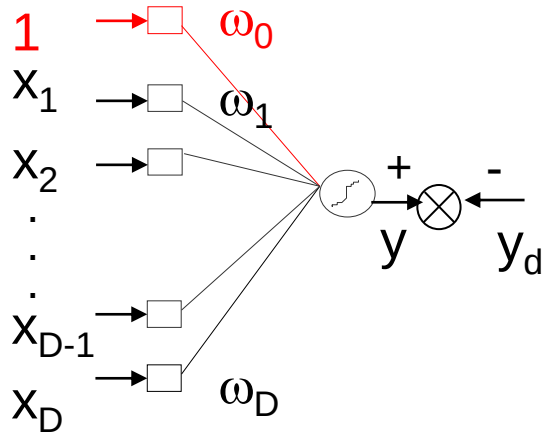
→ Minimum de la fonction F(X) où F est le nombre d'erreur de classification sur la base d'apprentissage

→ F peut s'exprimer en fonction de W mais n'est pas dérivable par rapport à W !!

1960 : Algorithme Adaline

⇔ ***Ada**ptive **Li**near **Ne**uron*

Modifications du perceptron



1) Neurones à fonction d'activation linéaire : $y = W^T.X$

$$y = \Sigma\, \omega_j\, x_j$$

2) Erreur (entre sortie désirée et sortie réelle/prédiction) :

$$y_d - y = y_d - \Sigma\, \omega_j\, x_j$$

Fonction objectif

2) **Fonction d'erreur :** Erreur quadratique pour un exemple **X** ayant pour valeur désirée $\mathbf{y_d}$

$$E = \frac{1}{2}(y_d - y)^2$$

Q **paramètres et intérêt** de cette fonction ?

3) **Fonction objectif * (à minimiser)** : erreur quadratique MOYENNE (*Mean Squared Error: MSE*) sur les données d'apprentissage

$$E = \sum_{1}^{Nb\_app} \frac{1}{2}(y_d - y)^2$$

(*) aussi appelé fonction de perte (*LOSS function*)

Descente de gradient

4) **Minimiser l'erreur par rapport aux poids** ⇔ modifier itérativement les poids jusqu'à atteindre un gradient nul (correspondant à un minimum de la fonction E – **voir cours optimisation**)

Gradient d'une fonction E(w_0 , w_2 ,… , w_D) de (D+1) variables :

$$\nabla E = \left(\frac{\partial E}{\partial \omega_0} , \frac{\partial E}{\partial \omega_1} , \dots , \frac{\partial E}{\partial \omega_D}\right)^T$$

mise à jour du vecteur : *W(t+1)* = *W(t)* – $\lambda \nabla E$

→ Modification des poids **dans la direction opposée au gradient de l'erreur**

Modification des poids

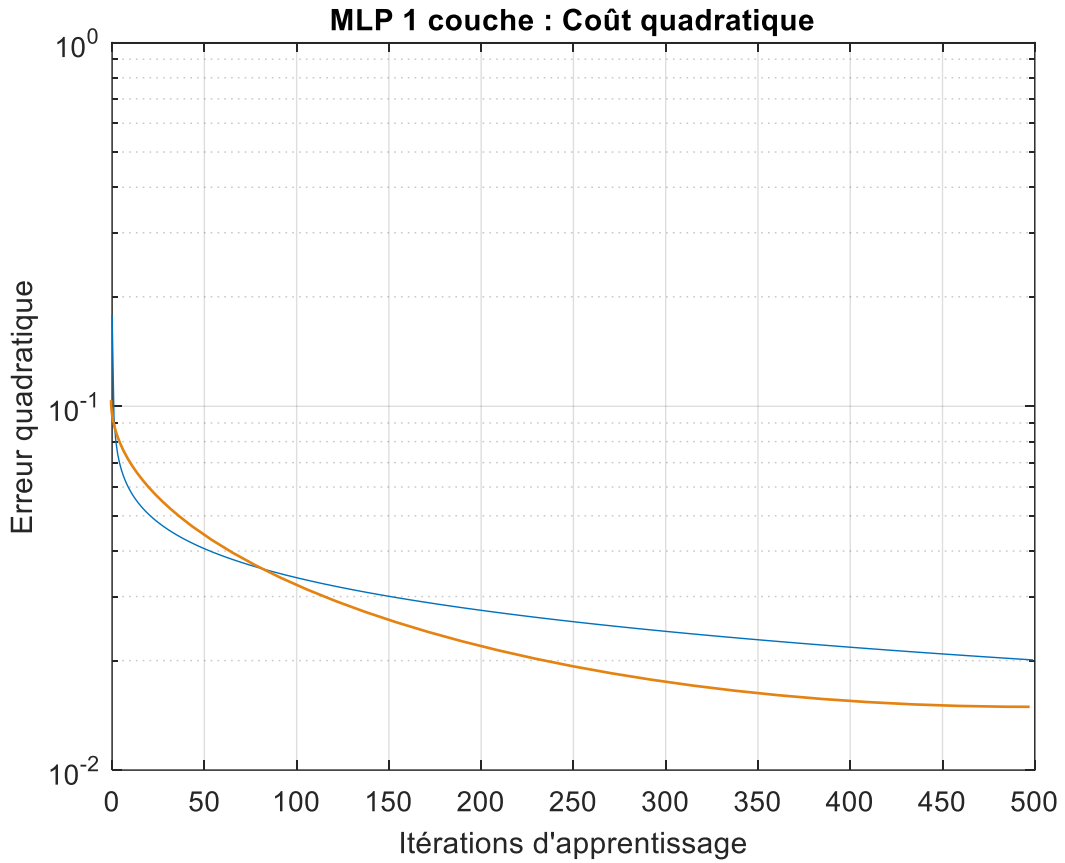
mise à jour de chaque poids : $\omega_j(t+1) = \omega_j(t) - \lambda \frac{\partial E}{\partial \omega_j}$

Calcul des dérivées partielles :

$$E = \frac{1}{2}(y_d - y)^2 = \frac{1}{2}(y_d - \sum \omega_j x_j)^2$$

$$\frac{\partial E}{\partial \omega_j} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial \omega_j} = -(y_d - y)x_j$$

Évolution de l'erreur et impact de l'initialisation aléatoire

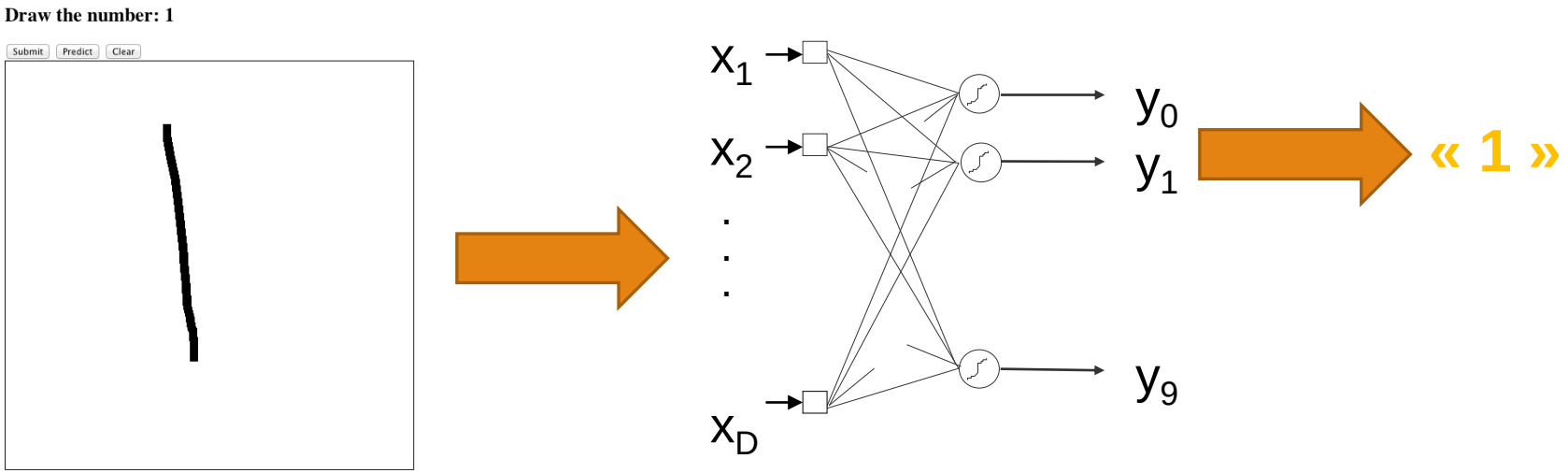


1960 : cas multiclasse (N classes)

Exemple : reconnaissance de CHIFFRES manuscrits → 10 neurones :

neurone du « 1 » : actif quand l'image d'un « 1 » est présentée au réseau

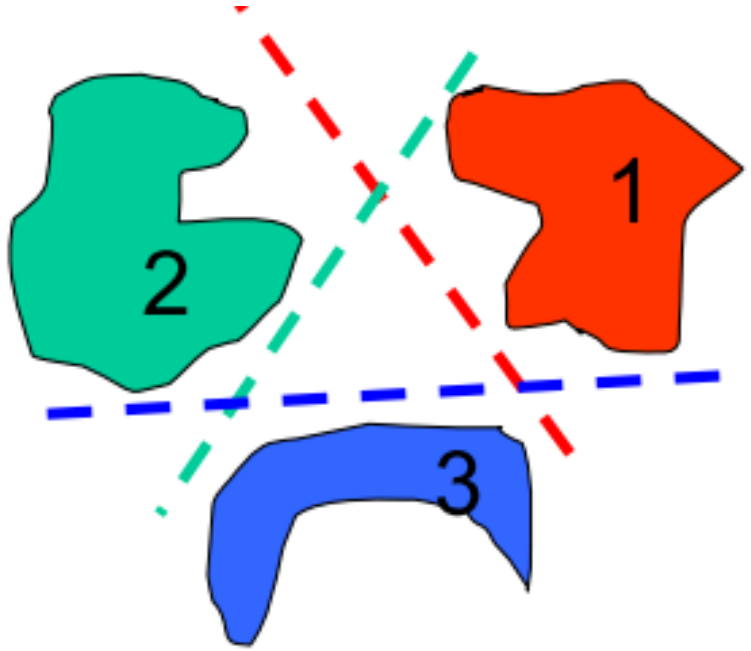
neurone du « 2 » : actif quand l'image d'un « 2 » est présentée au réseau

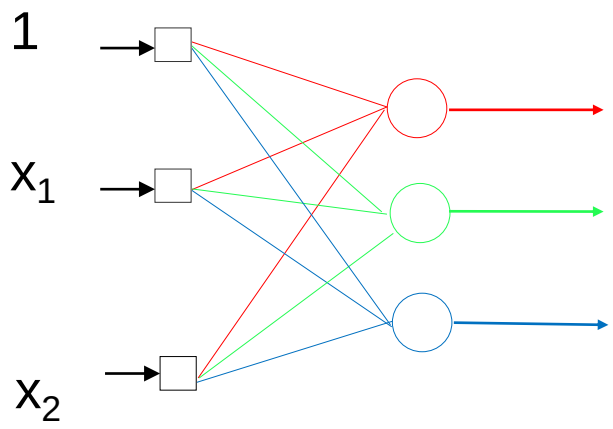


Réseau de neurones monocouche

Principe d'un réseau monocouche

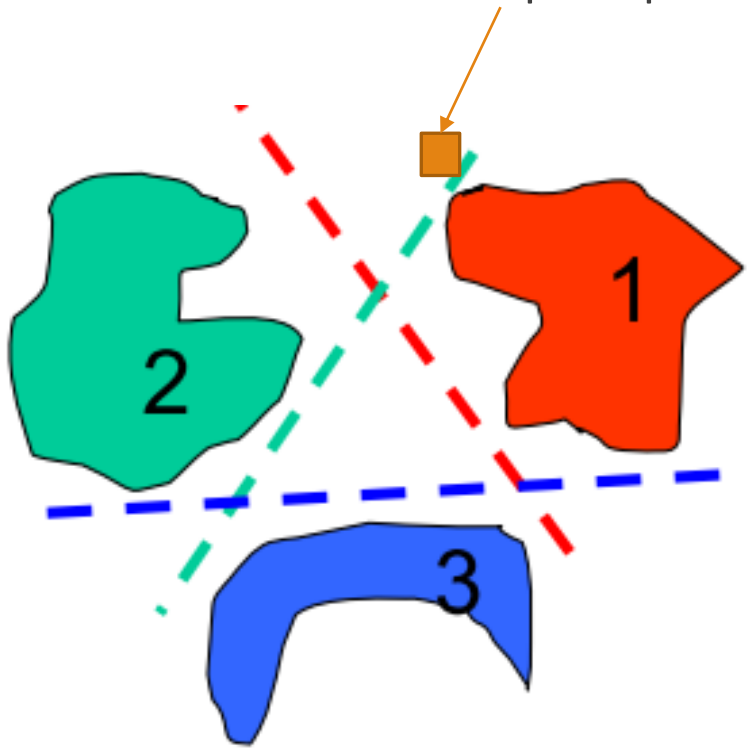
N neurones (1 par classe) : **le neurone i « sépare » la classe i des autres classes**

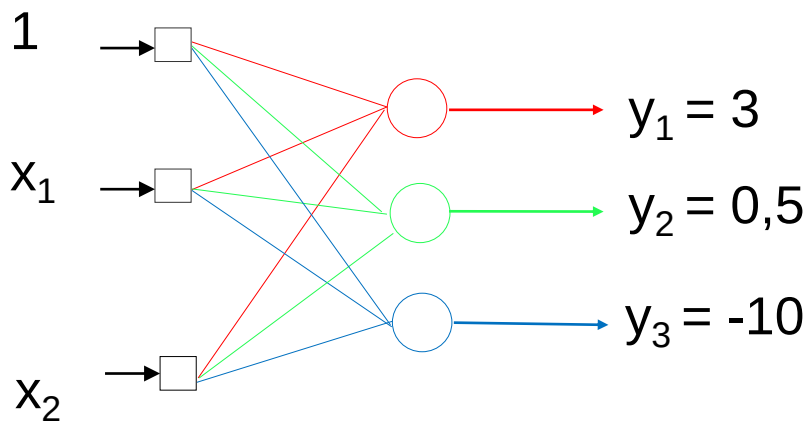




Fun time

Q1 Trouver la classe de l'exemple X produisant les sorties y_i

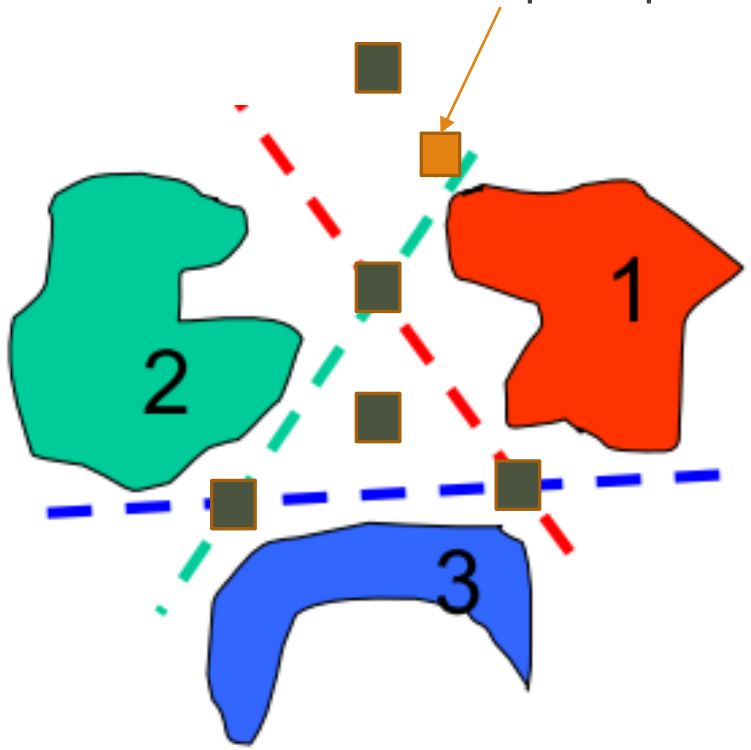


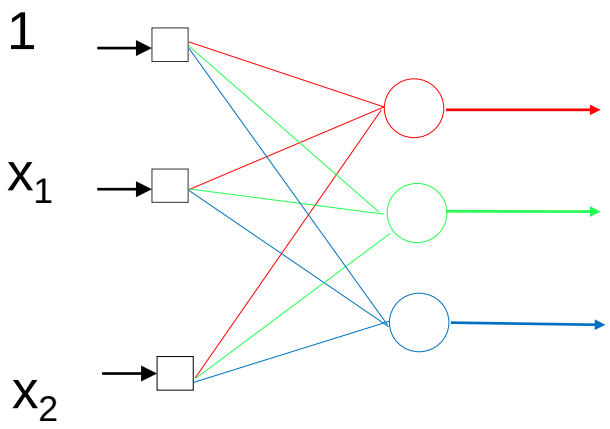


Q2 Déterminer les frontières de décision

Fun time

Q1 Trouver la classe de l'exemple X produisant les sorties y_i

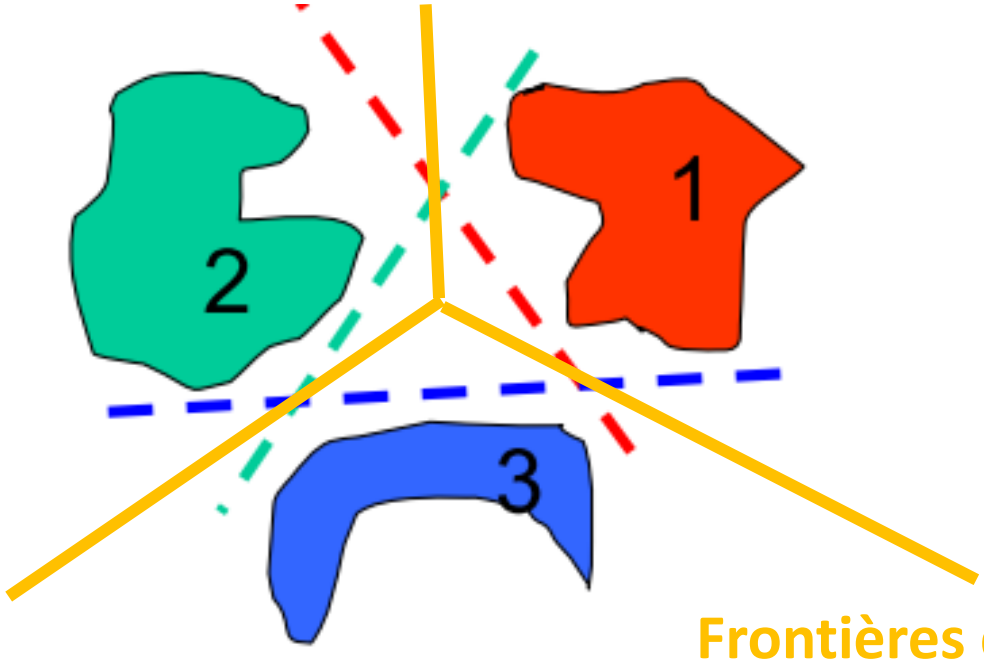


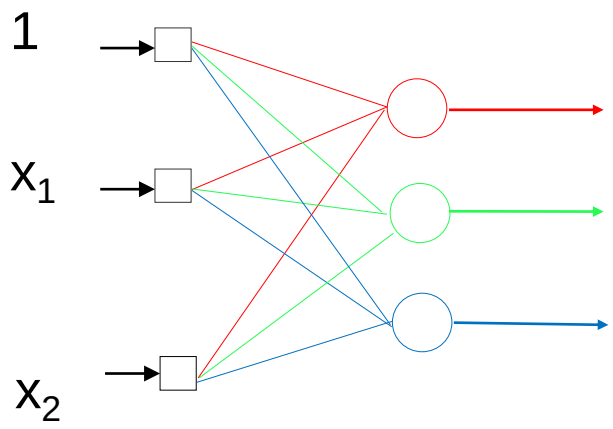


Q2 Déterminer les frontières de décision

Frontières de décision

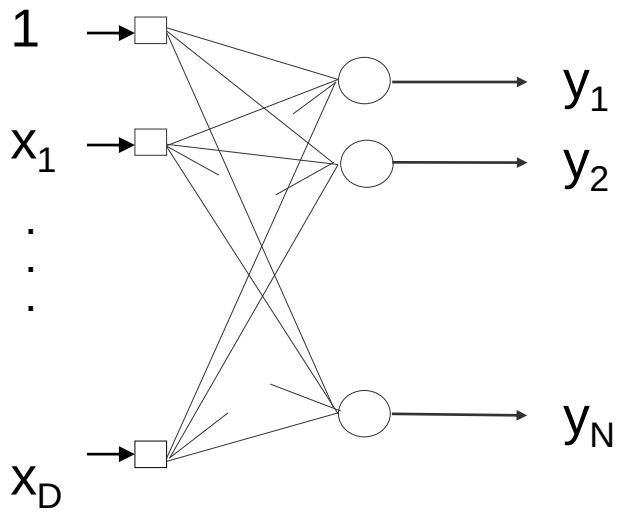
N neurones (1 par classe) : **le neurone i « sépare » la classe i des autres classes**





Architecture

N neurones (1 par classe) : **le neurone i « sépare » la classe i des autres classes**



Neurone « actif » : $y_i \rightarrow 1$

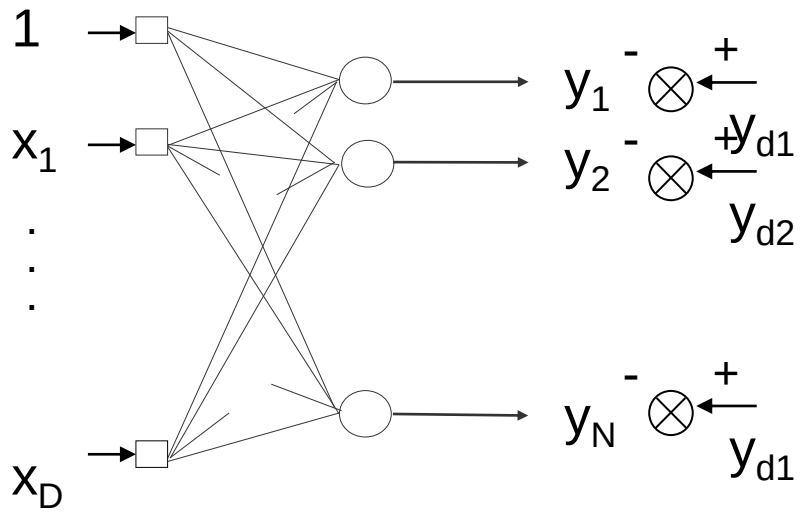
Neurone « inactif » : $y_i \rightarrow -1$

Paramètres

N neurones (1 par classe) : **le neurone i « sépare » la classe i des autres classes**

→ Matrice de poids W de dimensions (D,N) (nombre d'entrées, nombre de sorties)

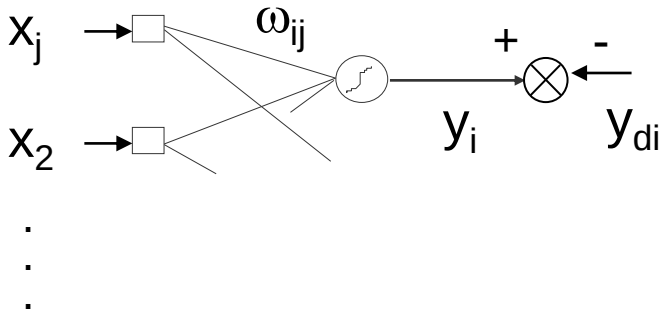
→ Vecteur des biais B de dimension N



→ vecteur des valeurs désirées : $y_d = (y_{d1}, y_{d2}, \ldots, y_{dN})$

Apprentissage

4) Trouver les poids optimaux :



mise à jour de chaque poids : $\omega_{ij}(t+1) = \omega_{ij}(t) - \lambda \frac{\partial E}{\partial \omega_{ij}}$

(Modification des poids **dans la direction opposée au gradient de l'erreur)**

Calcul des dérivées partielles : $\frac{\partial E}{\partial \omega_{ij}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial \omega_{ij}} = -(y_{di} - y_i)\, x_j$

$$\Rightarrow \Delta\omega_{ij} = -\lambda \frac{\partial E}{\partial \omega_{ij}} = -\lambda \delta_i x_j$$

Algorithme

Initialiser W(0) aléatoirement

- Apprentissage :
  - (1) Tirer au hasard un exemple X de la base d'apprentissage
  - (2) Modifier les poids ω suivant la relation :

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \lambda \Delta\omega_{ij} \text{ avec } \Delta\omega_{ij} = \delta_i x_j$$

λ : pas d'apprentissage

Incrémenter le compteur de mises à jour t=t+1

  - (3) critère d'arrêt :

Si MSE (*mean squarred error*) < ε OU t = t_{max} (nombre d'itération max)

Fin

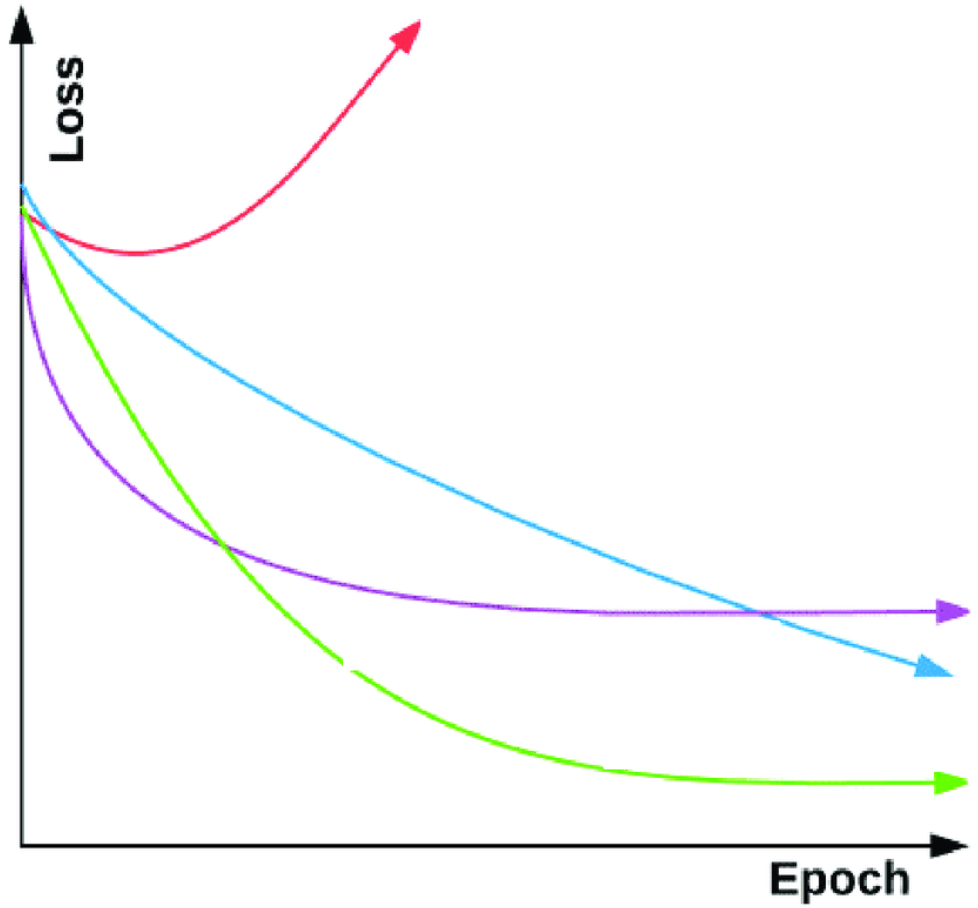
Sinon : retour en (1)

Fun time

1) Quel paramètre explique ces différences de fonctionnement ?

2) Que vaut-il dans ces 4 cas ?

3) Quel est la meilleure solution ?



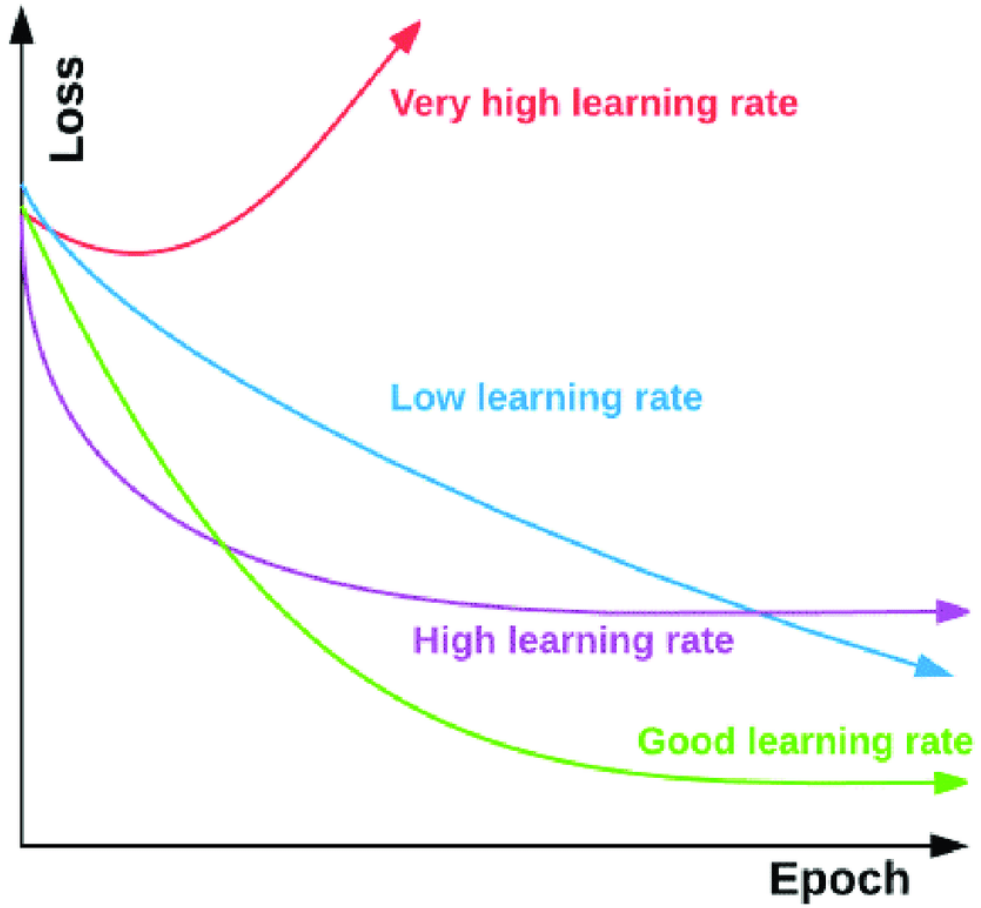
(*) itération = *epoch*

(…)

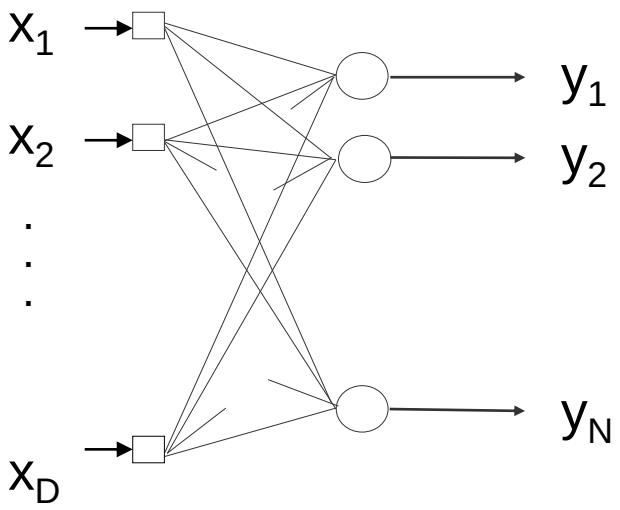
1) Quel paramètre explique ces différences de fonctionnement ?

2) Que vaut-il dans ces 4 cas ?

3) Quel est la meilleure solution ?



Décision : *Winner takes all*



→ **Décision directe : la cellule de sortie la plus active donne la classe «gagnante»**

→ Interprétation probabiliste (FAUSSE !) : chaque sortie évalue la probabilité *a posteriori* $P(C_i|X)$ que l'exemple appartienne à la classe C_i. D'après la règle de Bayes, la classe «gagnante» est celle qui maximise $P(C_i|X)$

Décision avec rejet –reconnaissance de lettres (26 classes)

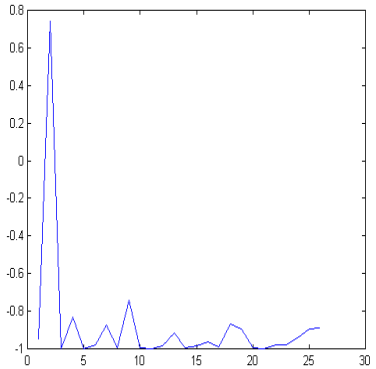
Dans certain cas le coût d'une erreur de classification peut être élevé

→ Mieux vaut « rejeter » l'exemple

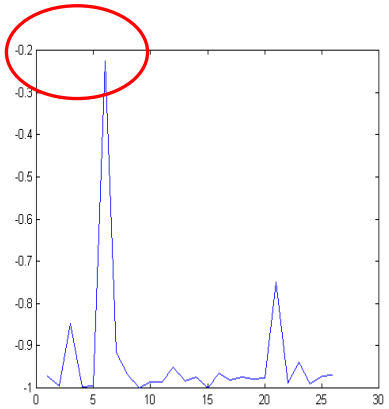
→**Rejet de distance**

sortie la plus active inférieure

à un seuil



OK

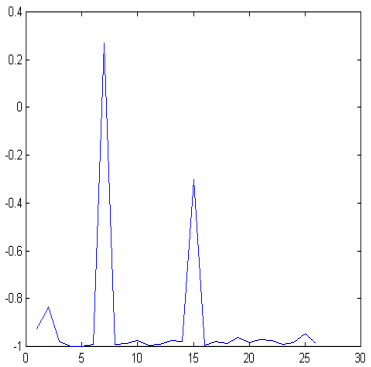


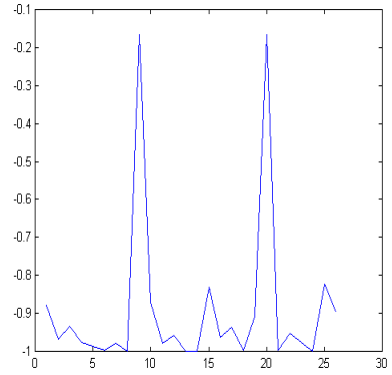
rejet

→**Rejet d'ambiguïté**

faible écart entre les deux

sorties les plus actives





Sommaire

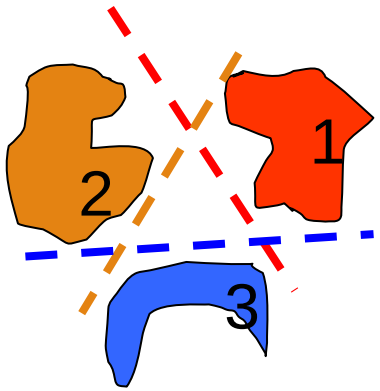
Réseaux de neurones et apprentissage supervisé

- Du neurone naturel au neurone artificiel
- Le perceptron

- Les réseaux de neurones monocouches
- L'Adaline

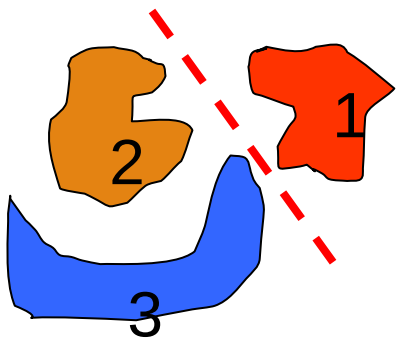
- **Les réseaux multicouches**
- **La rétro-propagation**

Limites



Les régions 1, 2 et 3 sont **linéairement séparables**

→ **l'algorithme converge ☺**



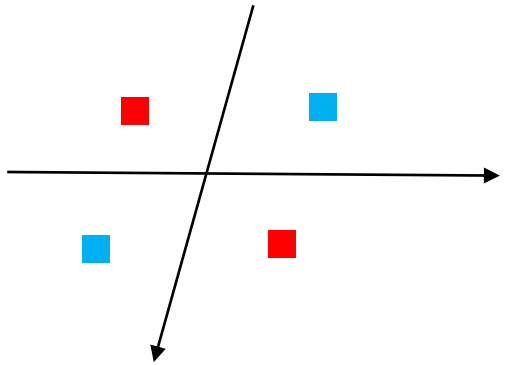
La région 1 est LS de la région 2 et de la région 3
Les régions 2 et 3 ne le sont pas

→ **L'algorithme converge vers une solution sub-optimale !!**

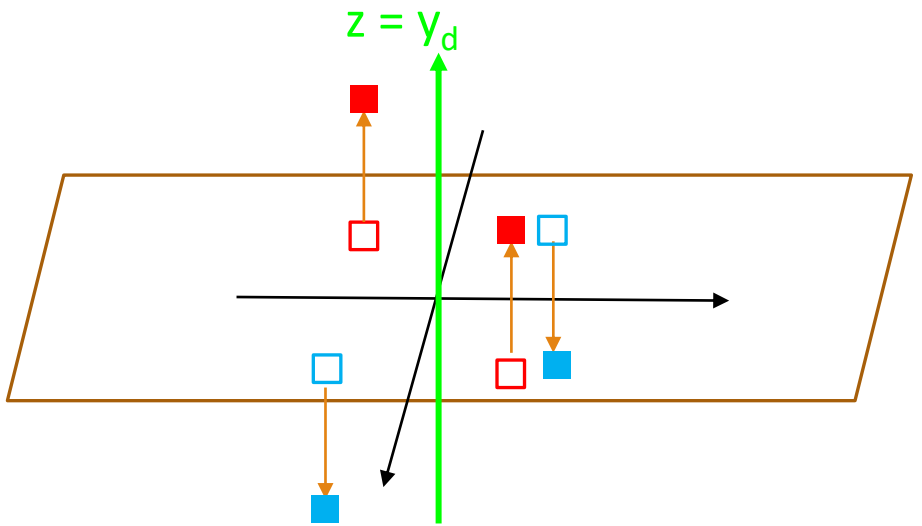
☹ incapacité à traiter les problèmes non linéairement séparables !!

Résoudre le “problème du XOR”

2D



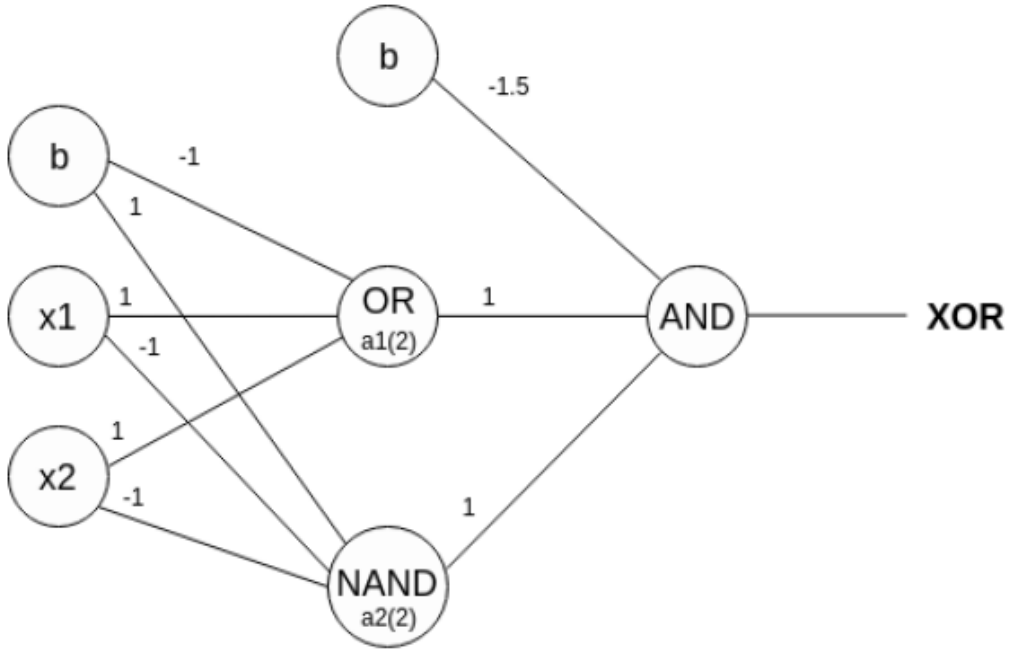
→ 3D



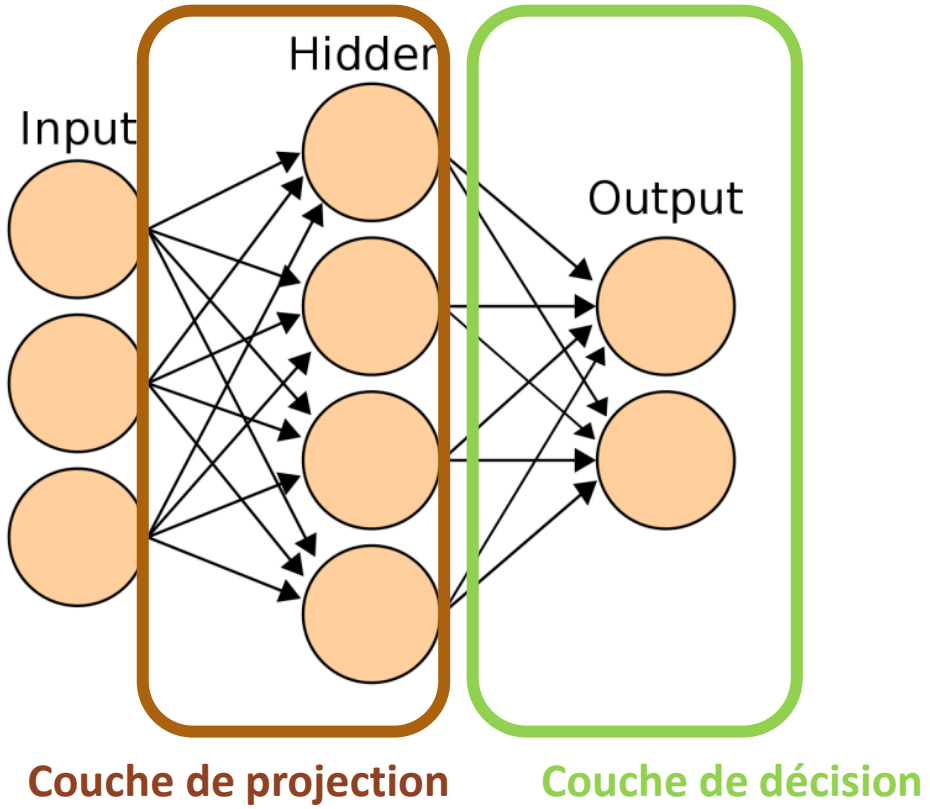
→ **Solution :** projeter les données dans un espace où elles sont linéairement séparables

Résoudre le “problème du XOR”

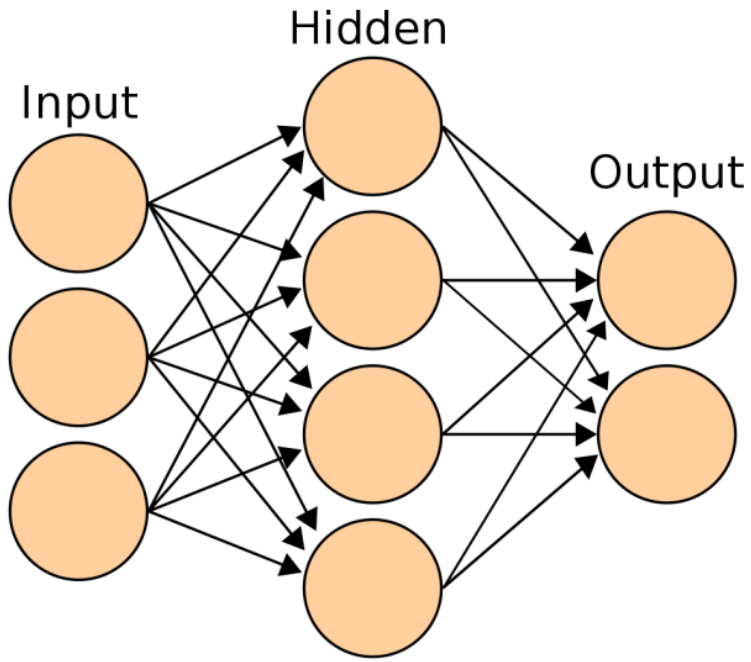
Une solution à base de portes logiques



Solution : réseau à deux couches



(…)



Pb : impossible d'apprendre les poids de la 1ère couche… **Premier « hiver » du connexionnisme**

→ **systèmes experts**

Bilan

Apprentissage supervisé : classe connue

→ **cas binaire, linéairement séparable : perceptron (1 neurone)**

→ **cas multiclasse (N classes), linéairement séparable : réseau monocouche (N neurones)**

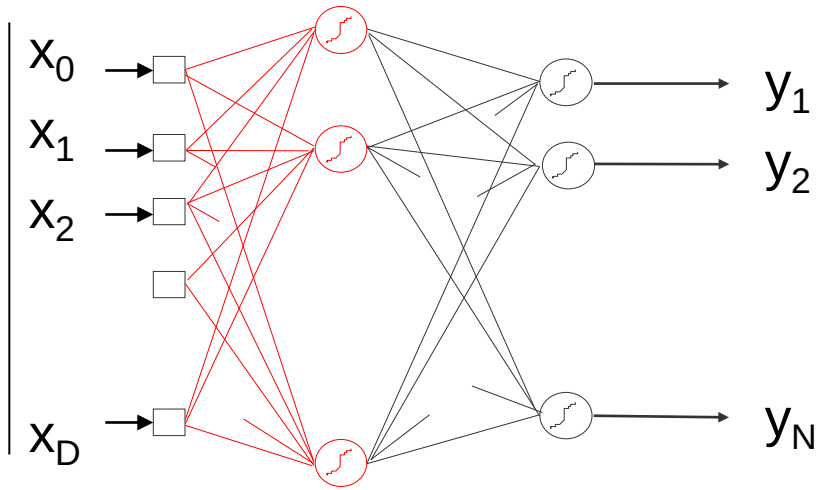
→ **cas non linéairement séparable : réseau multicouche**

→ **Comment apprendre ??**

Apprentissage d'un réseau multicouche

→ **MLP : *Multi Layer Perceptron***

exemple =



Un nouveau neurone …

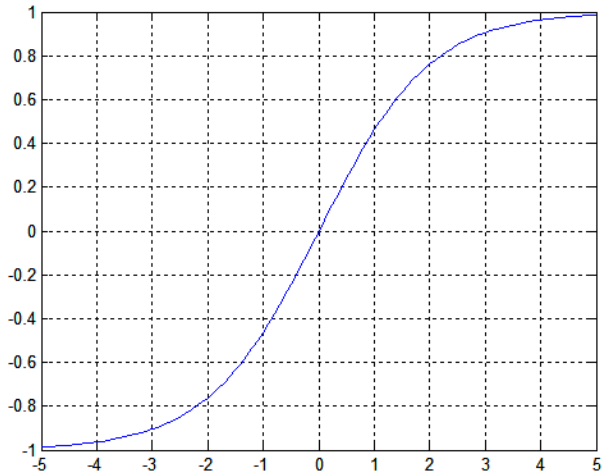
Neurone **produit scalaire → $\nu = W^T.X$**

Fonctions d'activation :
- à seuil (non linéaire, non dérivable)

- linéaire (linéaire, dérivable)

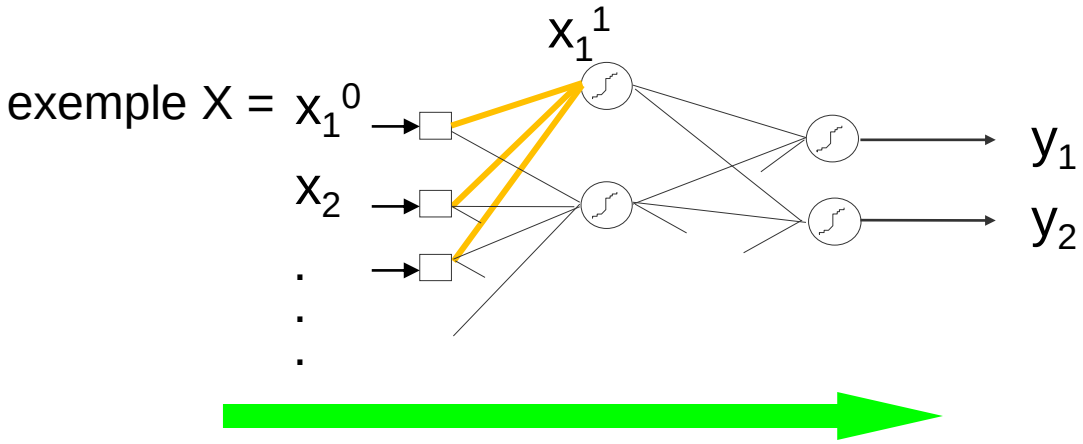
- linéaire saturée

- sigmoïde (non linéaire, dérivable)



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

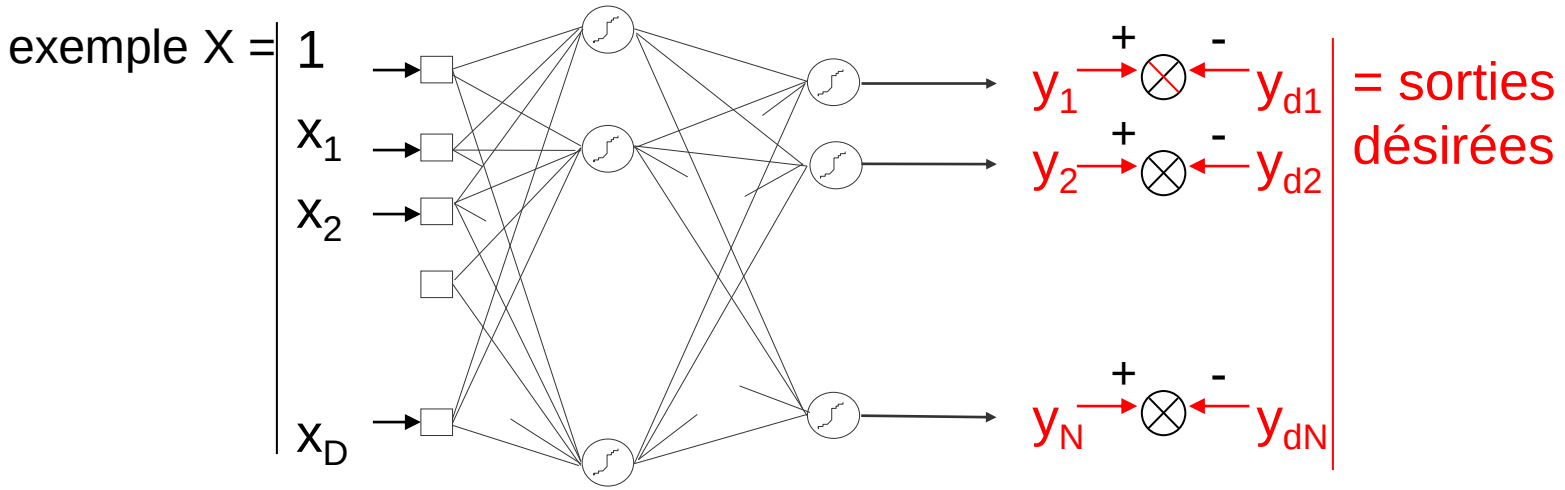
Phase 1 : propagation



- **Propager** un exemple dans le réseau :

$$x_i^{c} = f(\nu_i) = f\left[\sum_j \omega_{ij}\, x_j^{c-1}\right]$$ pour la couche **c**

Phase 2 : calcul de l'erreur

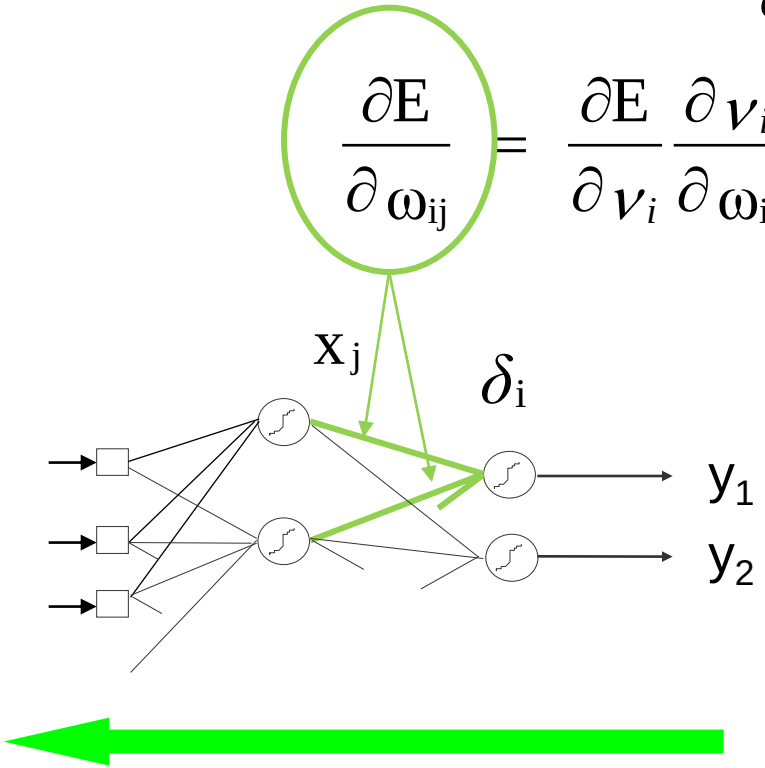


Erreur Quadratique : $$E = \frac{1}{2}\sum_{i=1}^{i=N}(y_{di} - y_i)^2$$

Phase 3 : retro-propagation

On souhaite minimiser l'erreur : $\frac{\partial E}{\partial \omega} = 0$

$$\frac{\partial E}{\partial \omega_{ij}} = \frac{\partial E}{\partial \nu_i}\frac{\partial \nu_i}{\partial \omega_{ij}} = \delta_i x_j$$



Neurones de sortie :

$$\delta_i = \frac{\partial E}{\partial \nu_i} = \frac{\partial}{\partial \nu_i}\left[\frac{1}{2}\left(y_{di} - f(\nu_i)\right)^2\right]$$

$$\delta_i = -f'(\nu_i)(y_{di} - f(\nu_i))$$

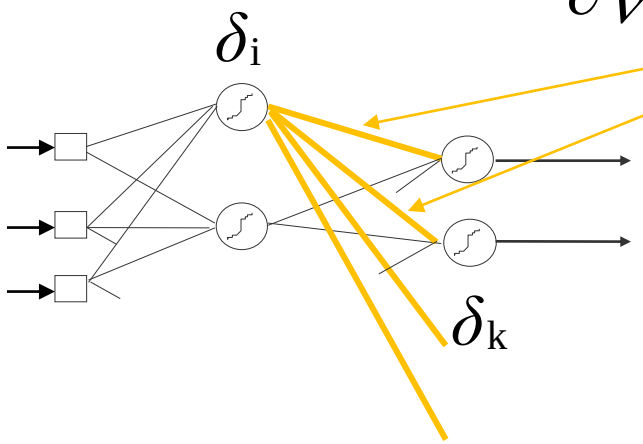
(…)

On souhaite minimiser l'erreur : $\frac{\partial E}{\partial \omega} = 0$

$$\frac{\partial E}{\partial \omega_{ij}} = \frac{\partial E}{\partial \nu_i}\frac{\partial \nu_i}{\partial \omega_{ij}} = \delta_i x_j$$

Neurones cachés : calcul de δ_i

$$\delta_i = \frac{\partial E}{\partial \nu_i} = \sum_k \frac{\partial E}{\partial \nu_k}\frac{\partial \nu_k}{\partial \nu_i} = \sum_k \delta_k \frac{\partial \nu_k}{\partial \nu_i}$$

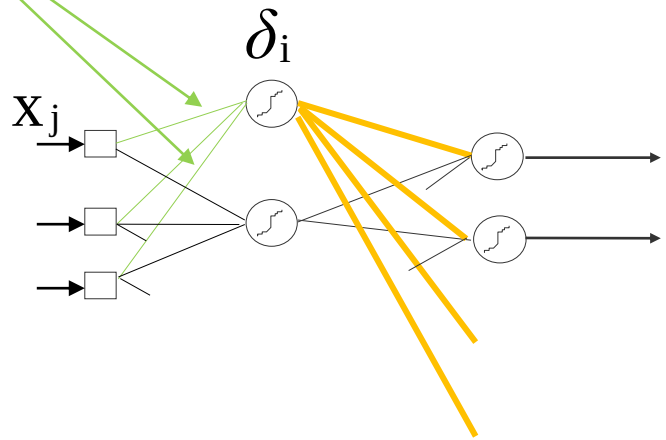


k : couche suivante

(…)

On souhaite minimiser l'erreur : $\frac{\partial E}{\partial \omega} = 0$

$$\frac{\partial E}{\partial \omega_{ij}} = \frac{\partial E}{\partial \nu_i}\frac{\partial \nu_i}{\partial \omega_{ij}} = \delta_i x_j$$



Neurones cachés :

$$\delta_i = \frac{\partial E}{\partial \nu_i} = \sum_k \frac{\partial E}{\partial \nu_k}\frac{\partial \nu_k}{\partial \nu_i} = \sum_k \delta_k \frac{\partial \nu_k}{\partial \nu_i}$$

or $\nu_k = \sum_i \omega_{ki} x_i = \sum_i \omega_{ki} f(\nu_i)$

donc $\frac{\partial \nu_k}{\partial \nu_i} = \omega_{ki} f'(\nu_i)$ $\quad \delta_i = f'(\nu_i)\sum_k \omega_{ki}\delta_k$

Bilan : Algorithme

1. Initialiser les ω_{ij} aléatoirement
2. Propager un exemple X dans tout le réseau : $x_i^k = f(\nu_i)$ avec $\nu_i = \sum_j \omega_{ij} x_j^{k-1}$
3. Calcul l'erreur quadratique en sortie
4. Rétro-propager :

   Couche de sortie (pour k de 1 à N) : $\delta_k = -f'(\nu_k)(y_{dk} - x_k)$

   Couche cachée (pour i de 1 à C) : $\delta_i = f'(\nu_i)\sum_k \omega_{ki}\delta_k$
5. Modifier les poids :

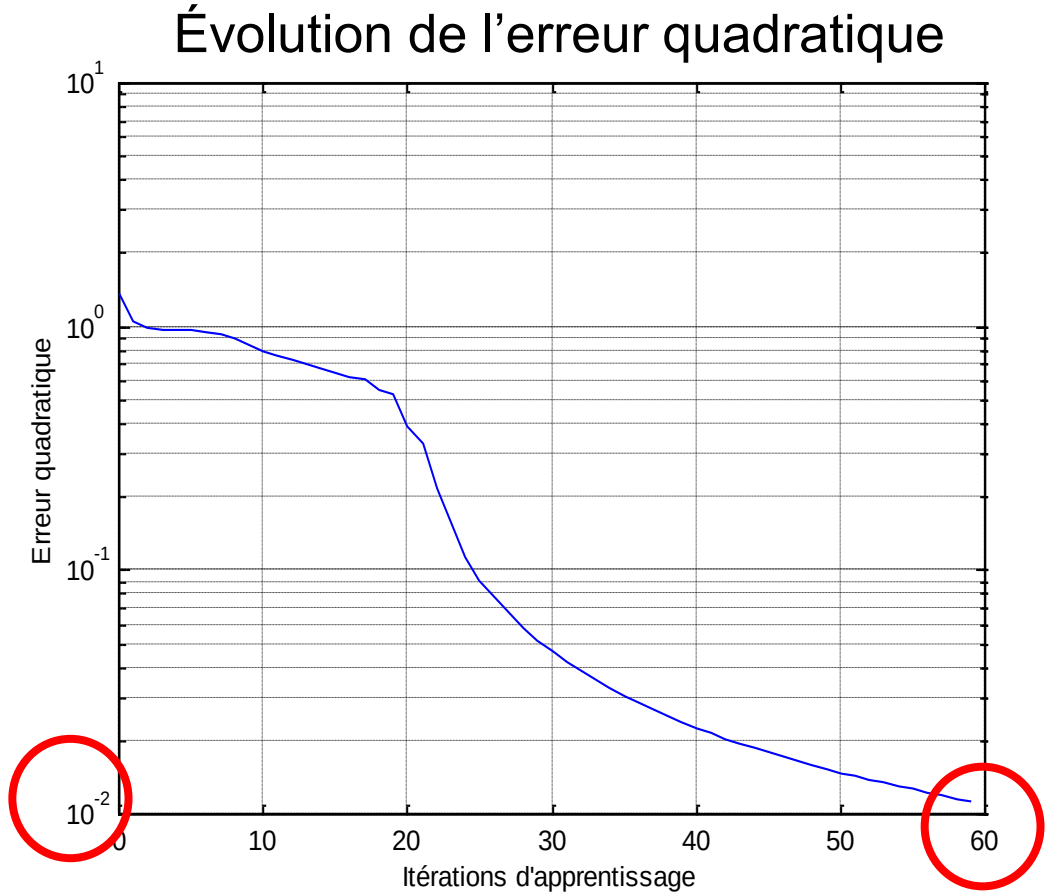
   $\omega_{ij}(t+1) = \omega_{ij}(t) - \lambda \frac{\partial E}{\partial \omega_{ij}}$ avec $\frac{\partial E}{\partial \omega_{ij}} = \delta_i x_j$
6. Si **critère d'arrêt** alors FIN sinon retour en 2 (nouvelle itération)

Critères d'arrêt

→ erreur admissible : $E < \varepsilon$

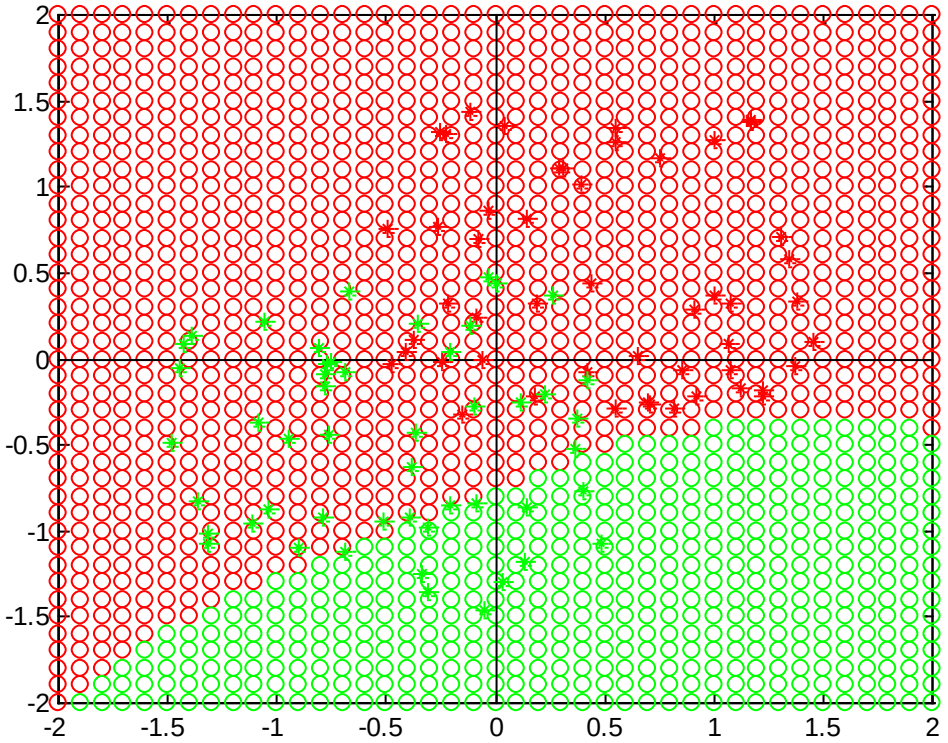
→ nombre d'itérations : $t = t_{max}$

→ ?



Exemple en 2D, à deux classes

MLP **2x8x2** : frontières initiales



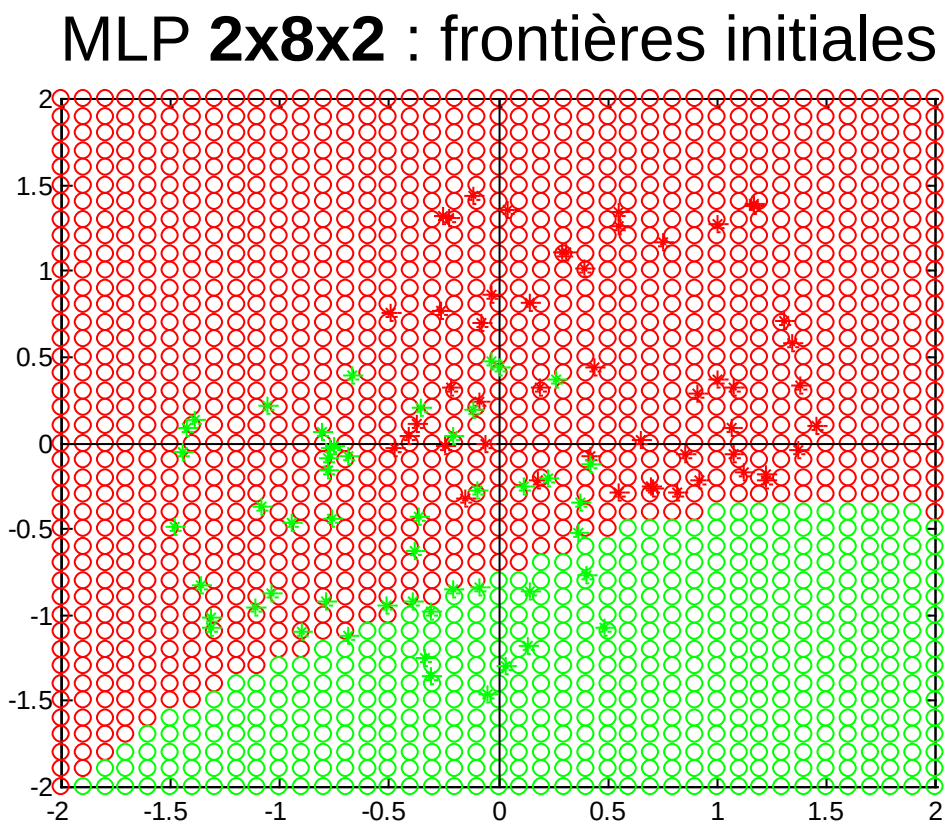
Q nombre de poids du réseau ?

Exemple en 2D, à deux classes

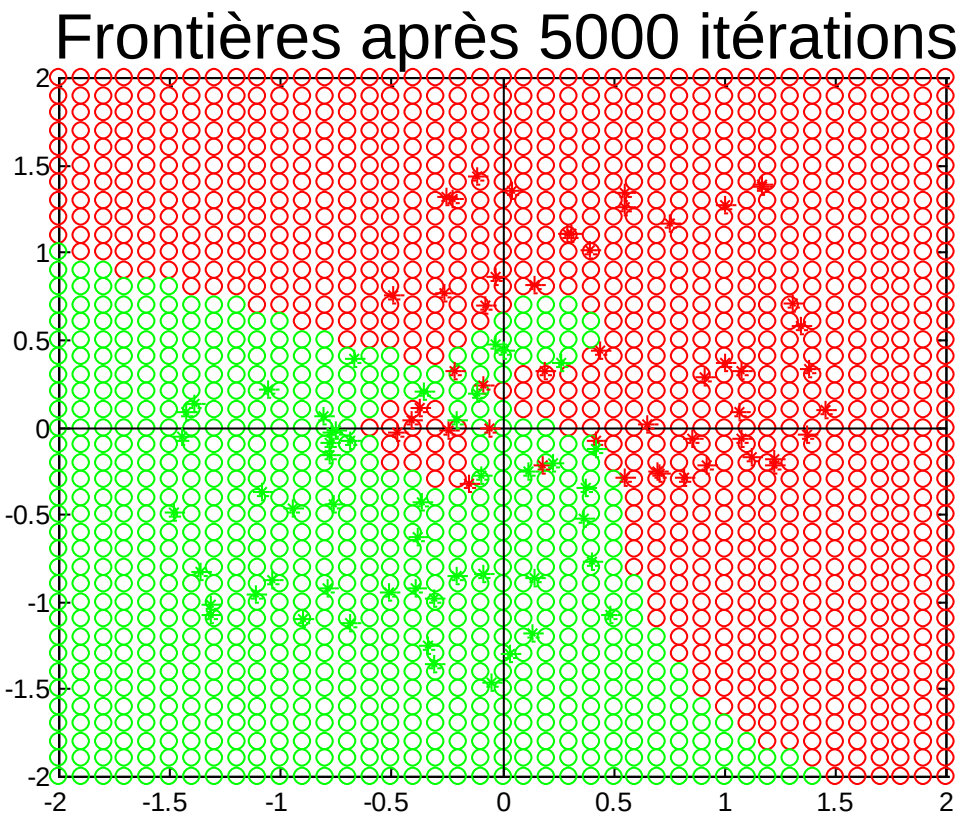
Entrée->cachée
(2+1)x8 = 24

Cachée->sortie
(8+1)x2 = 18

42 = 32 + 10 (biais)

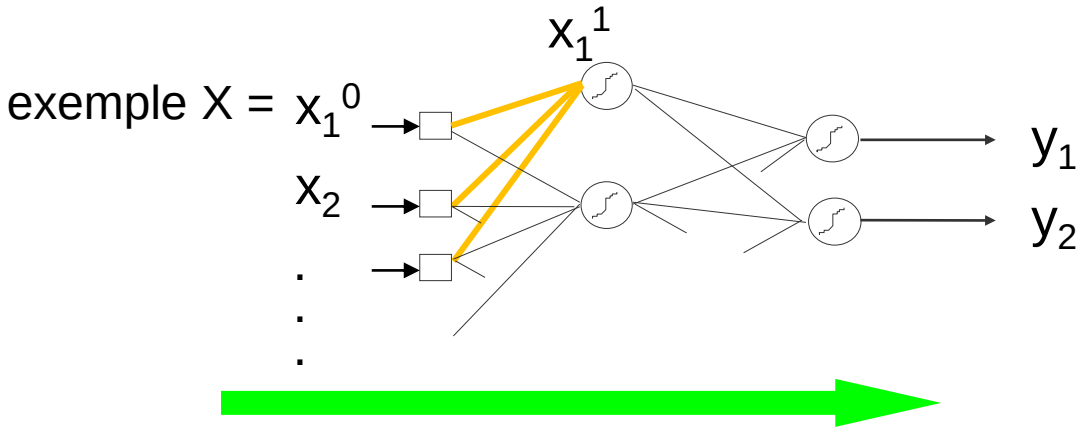


(...)



Q Comment obtenir cette décision ???

Réseaux de neurones : inférence



1) Propager un exemple dans le réseau :

$$x_i^{c} = f(\nu_i) = f\left[\sum_j \omega_{ij}\, x_j^{c-1}\right]$$ pour la couche **c**

2) Appliquer le critère WTA pour décider de la classe de l'exemple

Paramètres du réseau

Nombre de cellules d'entrée et de sortie : dépend du problème à résoudre

Nombre de couches et de cellules cachées déterminé par recherche exhaustive mais :

- Pour des problèmes de complexité « raisonnable », une ou deux couches suffisent
- Au-delà, phénomène de disparition du gradient (→ *deep learning*)

Le nombre de poids doit être inférieur au produit du nombre d'exemples d'apprentissage par le nombre de sorties

Initialisation des poids : aléatoire entre –1/M et 1/M où M est le nombre de connexions du neurone (afin d'éviter la saturation)

Bilan

→ cas binaire, linéairement séparable : perceptron (1 neurone)

→ cas multiclasse (N classes), linéairement séparable : réseau monocouche (N neurones)

→ **cas non linéairement séparable : réseau multicouche**
(C neurones cachés et N neurones de sorties)

→ **Préparation des données**

→ **Apprentissage par rétro-propagation**

→**Comment régler les paramètres ?**