

Réseaux de Neurones:

- Modèles & Architectures*
- Apprentissage*

Modèles & Architecture

- Neurone formel de McCulloch & Pitts
- Modèle non-linéaire d'un neurone
- Fonctions d'activation
- Architectures de réseaux
- Représentation des connaissances
- Apprentissage: Perceptron
- Apprentissage: Rétro-propagation

Quelques notes historiques

références les plus importantes:

- J.J. Hopfield

Neural Networks and physical systems with emergent collective computational abilities

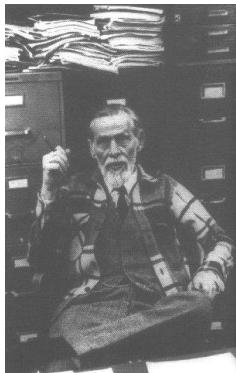
Proc. Natl. Acad. of Sciences, vol. 79, 1554-1558, 1982.

- D.E. Rumelhart & J.L. McClelland

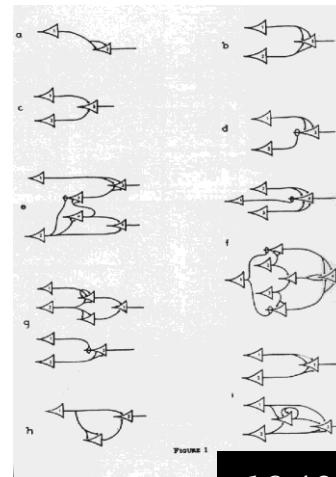
Parallel Distributed Processing: Exploration in the Microstructure of Cognition

MIT Press, Cambridge, 1986.

Neurone formel de McCulloch & Pitts



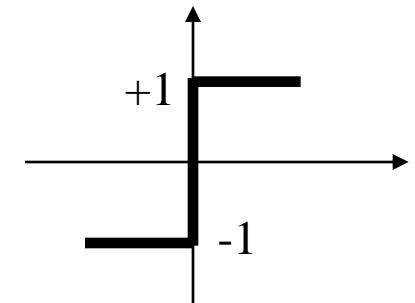
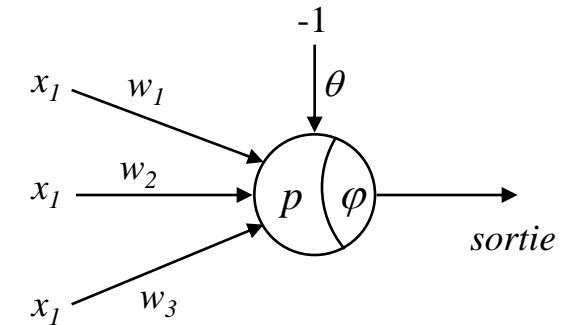
Warren S. McCulloch



William Pitts

$$p = \sum_i w_i x_i \quad \text{si } p > \theta \text{ then } \text{sortie} = +1$$

else $\text{sortie} = -1$



fonction Signum

Interprétation géométrique

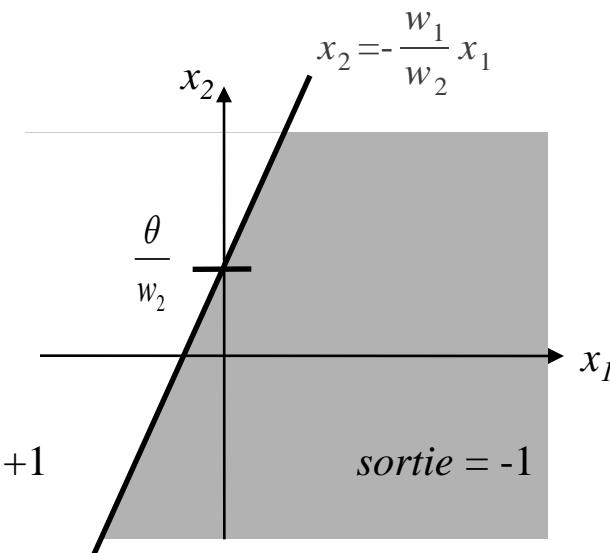
Soit un neurone formel de McCulloch&Pitts avec deux entrées x_1 et x_2

sa *sortie* est +1 si: $w_1x_1 + w_2x_2 > \theta$

sa *sortie* est -1 si: $w_1x_1 + w_2x_2 \leq \theta$

Géométriquement ces deux équations partitionnent le plan (x_1, x_2) par une droite définie par l'équation suivante:

$$x_2 = -\frac{w_1}{w_2}x_1 + \frac{\theta}{w_2}$$



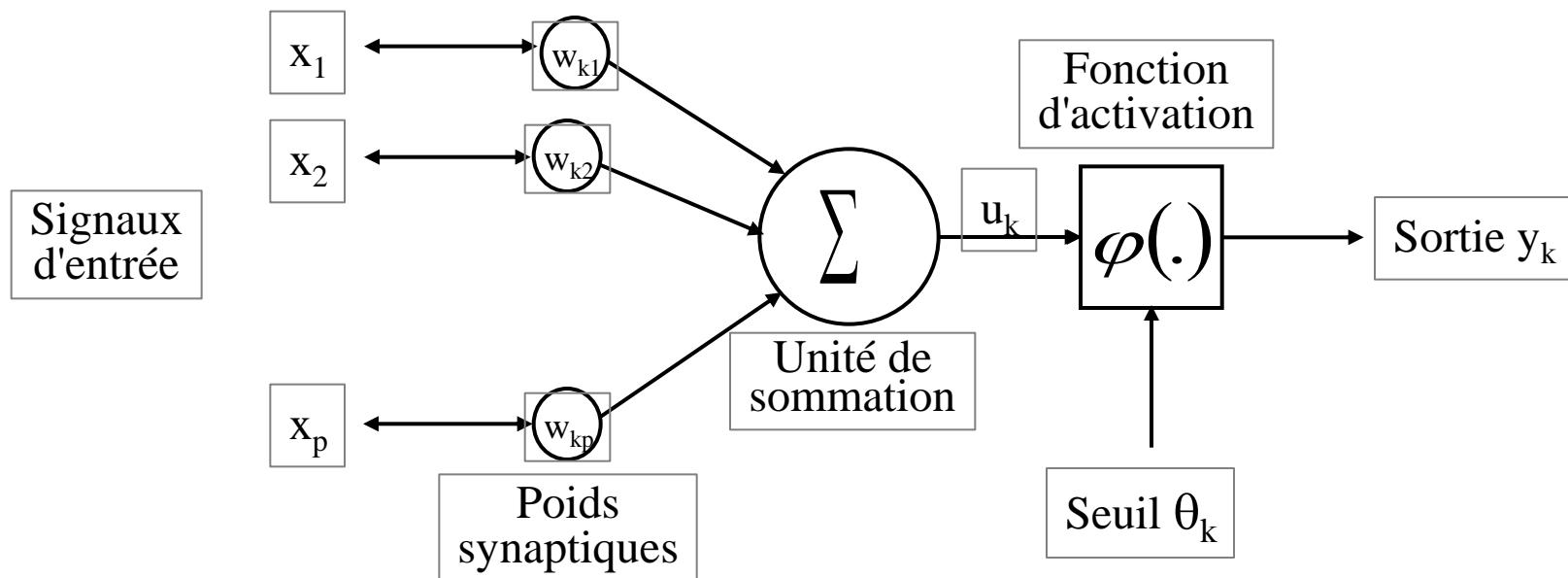
Modèle non-linéaire de neurone

3 éléments fondamentaux:

- Un ensemble de *synapses* caractérisées par un poids w_{kj}
 - $w_{kj} > 0 \Rightarrow$ synapse excitatrice,
 - $w_{kj} < 0 \Rightarrow$ synapse inhibitrice,
- un *additionneur* pour sommer les signaux d'entrée,
- une *fonction d'activation* pour limiter l'amplitude de la valeur de sortie.

Modèle non-linéaire de neurone (cont.)

(version moderne du modèle de McCulloch & Pitts)



Modèle non-linéaire de neurone (cont.)

Ce modèle est décrit mathématiquement par:

$$(1) \quad u_k = \sum_{j=1}^p w_{kj} x_j \quad \text{et} \quad y_k = \varphi(u_k - \theta_k)$$

où:

x_1, x_2, \dots, x_p sont les *entrées*,

$w_{k1}, w_{k2}, \dots, w_{kp}$ sont les *poids synaptiques* du neurone k,

u_k est la sortie de *l'additionneur*,

θ_k est le *seuil*,

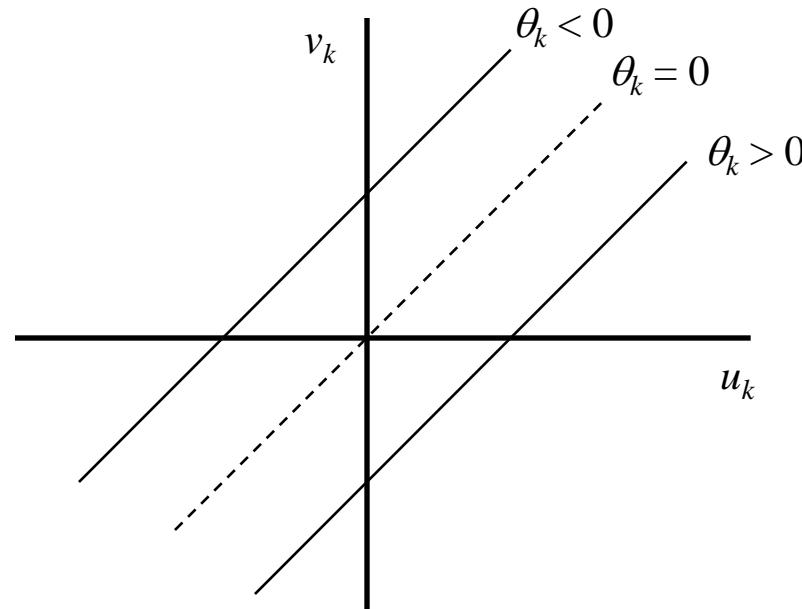
$\varphi(\cdot)$ est la *fonction d'activation*,

y_k est la valeur de *sortie* du neurone.

Effets de la valeur de seuil

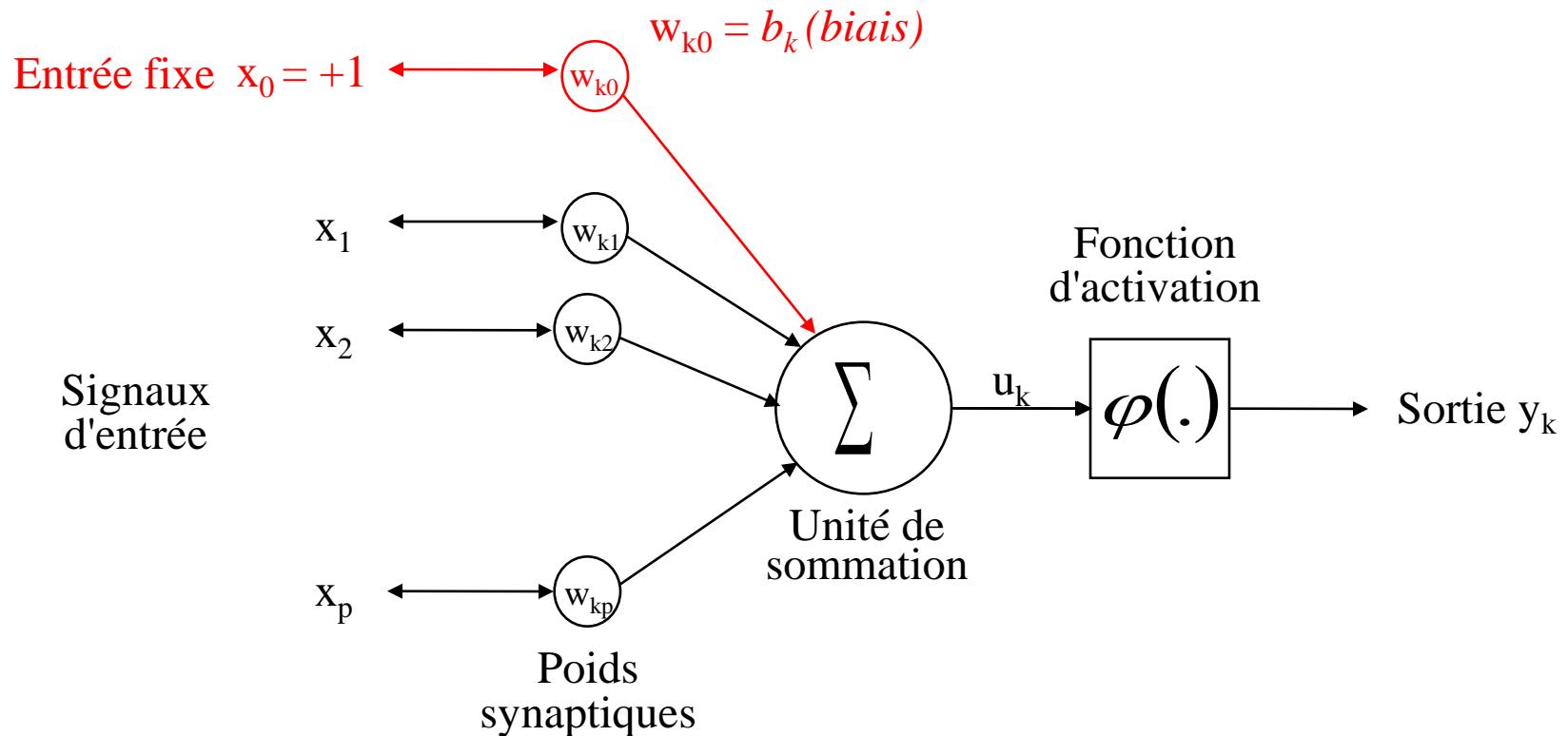
La valeur de seuil agit comme une *transformation affine* sur la valeur de sortie u_k :

v_k = potentiel d'activation du neurone k



$$v_k = u_k - \theta_k \quad \text{ou en utilisant l'équ. (1)} \quad v_k = \sum_{j=1}^p w_{kj} x_j - \theta_k$$

Modèle étendu (alternative)



Types de fonctions d'activation

La fonction d'activation définit la valeur de sortie d'un neurone en fonction des valeurs de ses entrées.

3 types de fonctions d'activation:

- *Fonction à seuil*

$$y_k = \varphi(v_k) = \begin{cases} 1 & \text{si } v_k \geq 0 \\ 0 & \text{si } v_k < 0 \end{cases}$$

- *Fonction 3 marches*

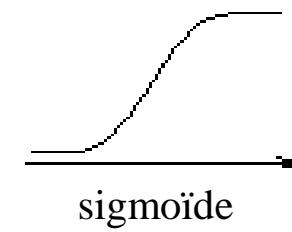
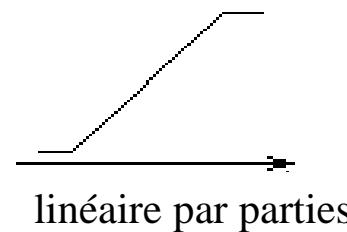
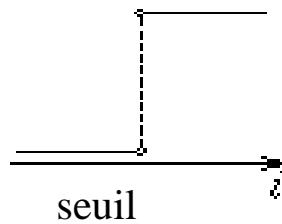
$$\varphi(v) = \begin{cases} 1 & v \geq \alpha \\ v & \alpha > v > \beta \\ 0 & v \leq \beta \end{cases}$$

Types de fonctions d'activation (cont.)

- Fonction sigmoïde

$$\varphi(v) = \frac{1}{1 + e^{-av}} \quad \text{ou} \quad \varphi(v) = \tanh\left(\frac{v}{2}\right) = \frac{1 - e^{-v}}{1 + e^{-v}}$$

- graphiquement

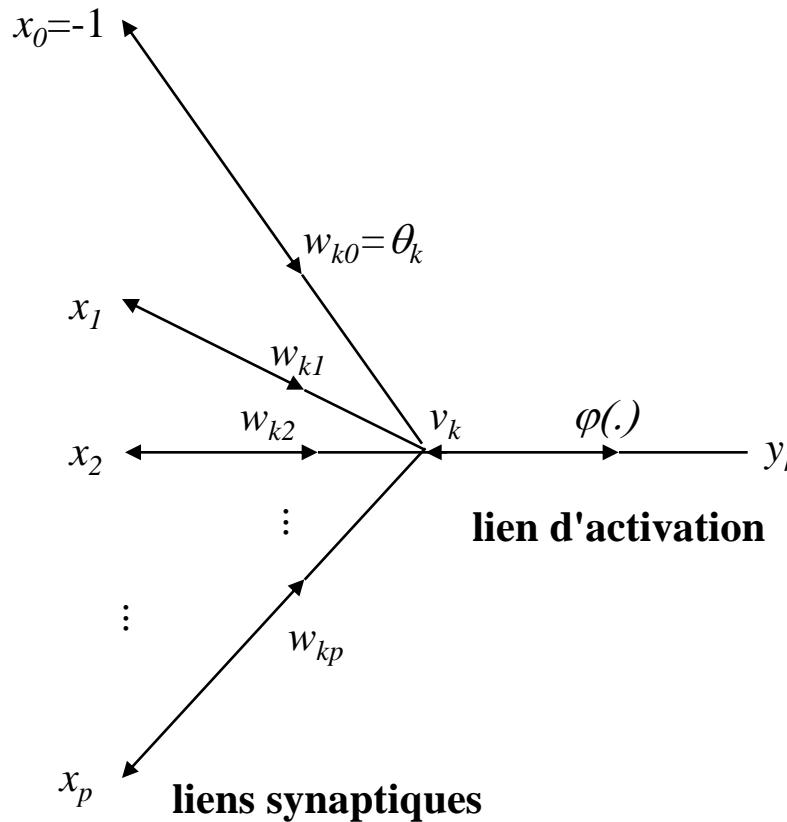


Définition mathématique

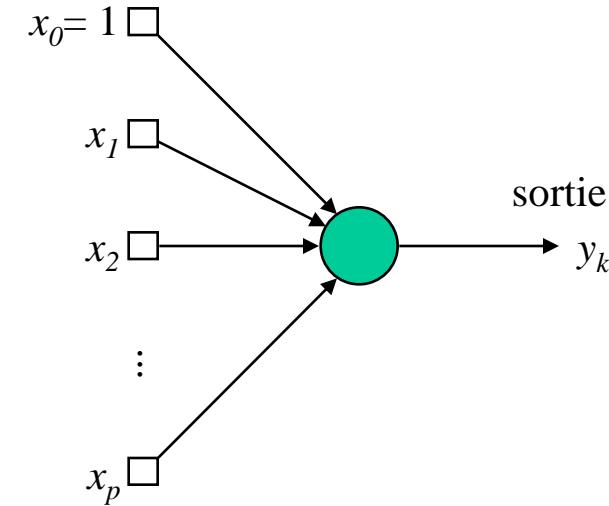
Un réseau neuronal est un *graphe orienté* constitué de *nœuds* liés par des *liens synaptiques* et *d'activation* caractérisé par les propriétés suivantes:

- un neurone est représenté par:
 - un ensemble de liens synaptiques linéaires,
 - un lien d'activation non-linéaire,
 - un seuil
- les liens synaptiques pondèrent leurs signaux d'entrée,
- la somme pondérée des signaux d'entrée = niveau d'activité interne du neurone,
- le lien d'activation transforme le niveau d'activité interne en valeur de sortie (= variable d'état du neurone).

Diagrammes "signal-flow" & architecturaux



Graphe "signal-flow"

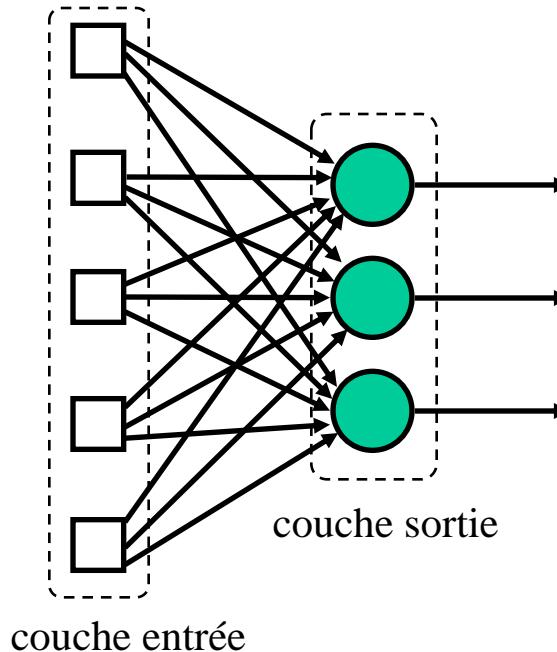


Graphe architectural

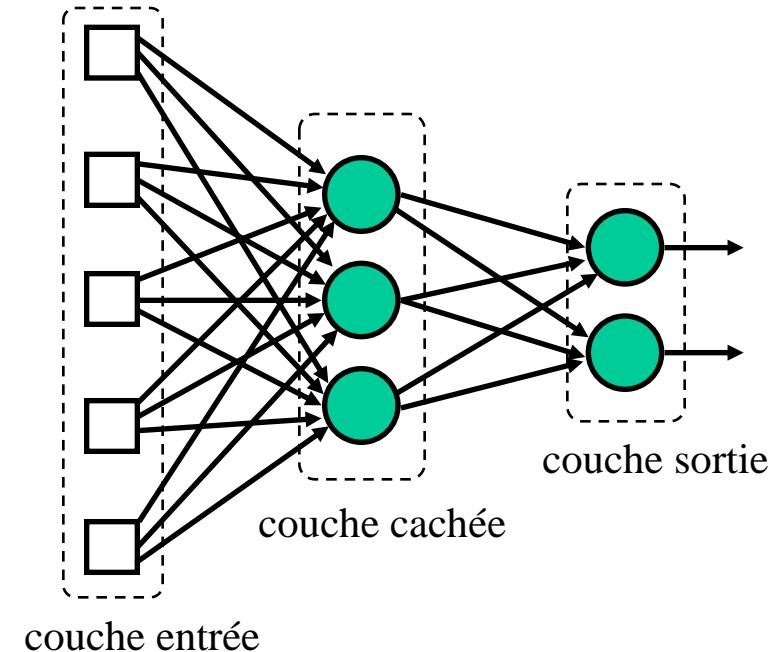
Architectures de réseaux

- Il y a 4 différentes classes d'*architectures*:
 - réseaux monocouche "feedforward",
 - une couche d'entrés de nœuds-source,
 - une couche de sortie,
 - "feedforward": entrée \Rightarrow sortie (pas vice versa)
 - réseaux multicouche "feedforward",
 - une couche d'entrés de nœuds-source,
 - une ou plusieurs couches cachées,
 - une couche de sortie,
 - réseaux récurrents,
 - au moins une boucle de rétro-action,
 - structures en treillis,
 - neurones organisés en matrice.

Architectures "feed-forward"

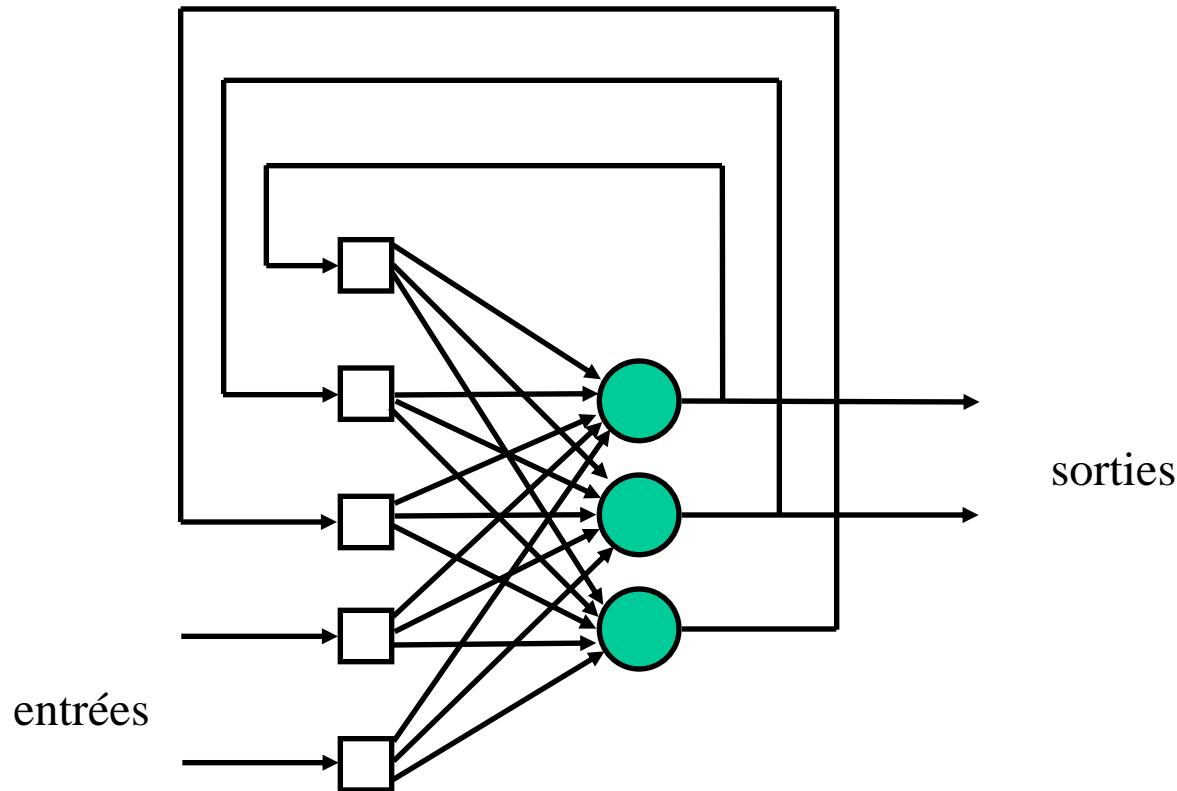


Réseau monocouche "feedforward"



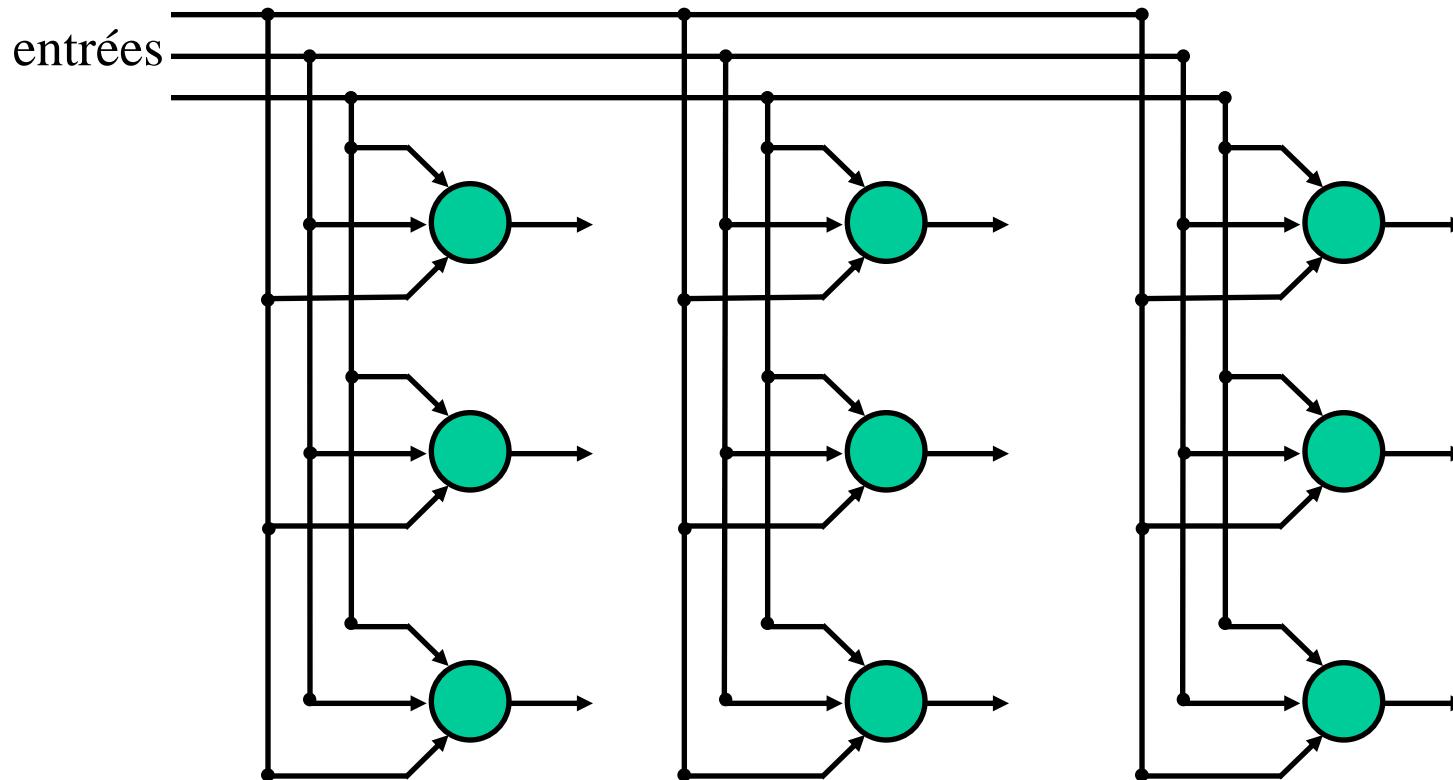
Réseau multicouche "feedforward"

Architecture récurrente



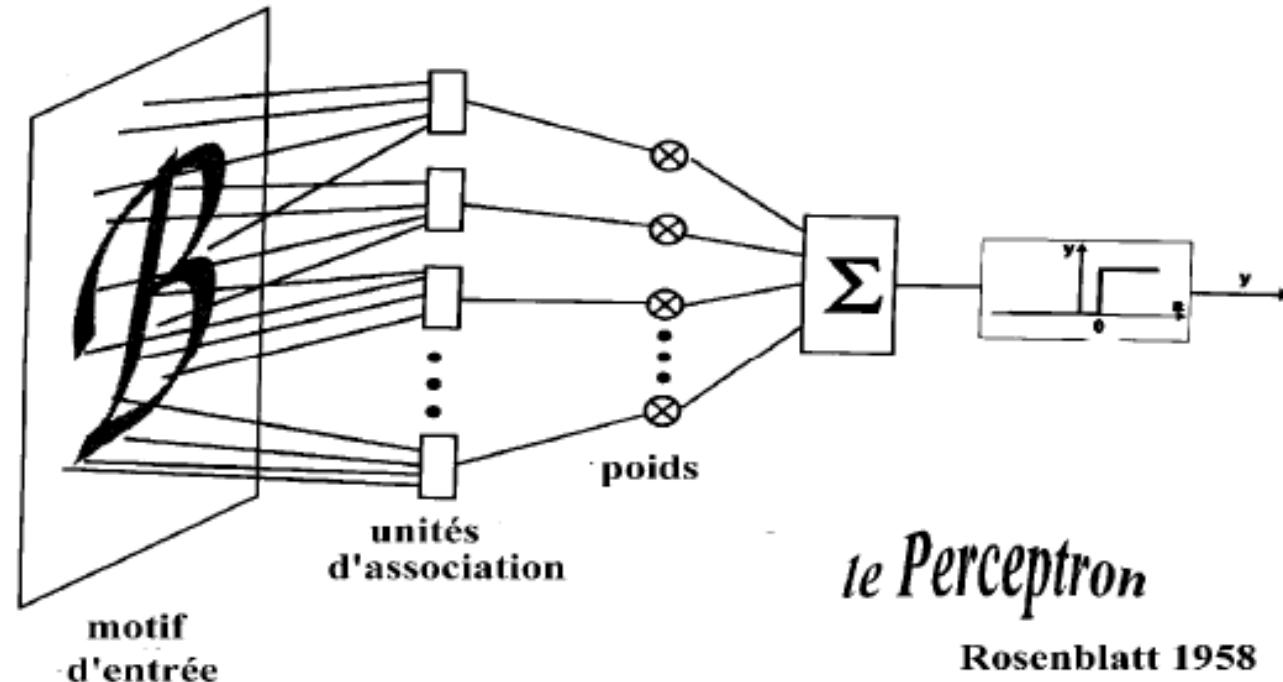
Réseau récurrent avec neurones cachés

Architecture en treillis



Réseau 2-D 3x3

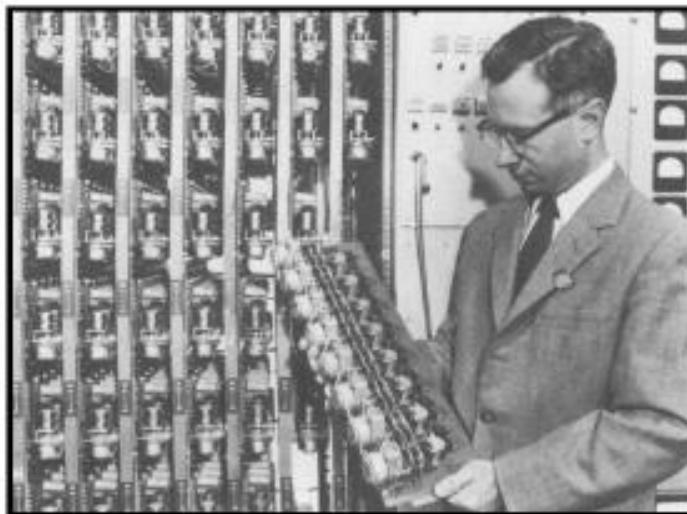
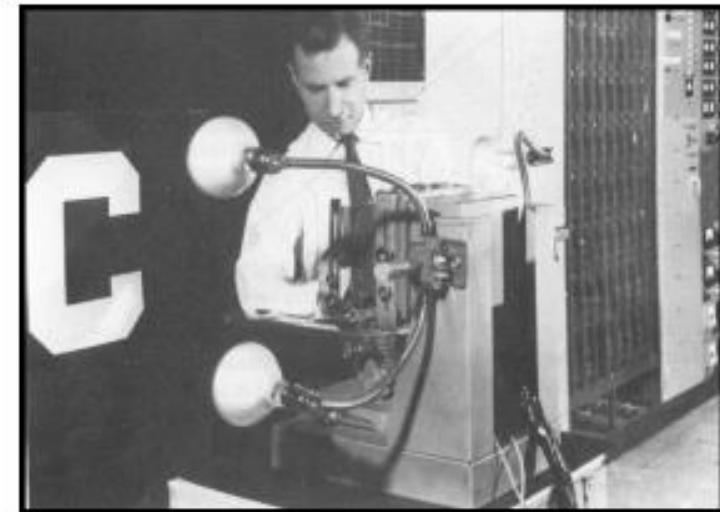
Le Perceptron (F. Rosenblatt, 1958)



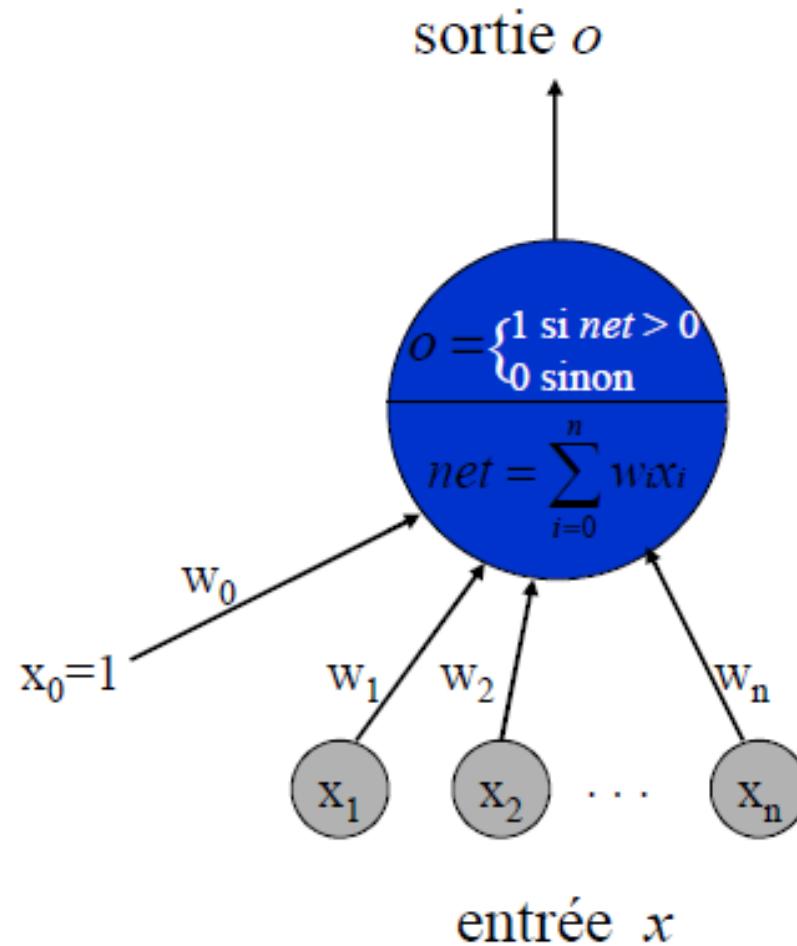
Le Mark I Perceptron



F. Rosenblatt

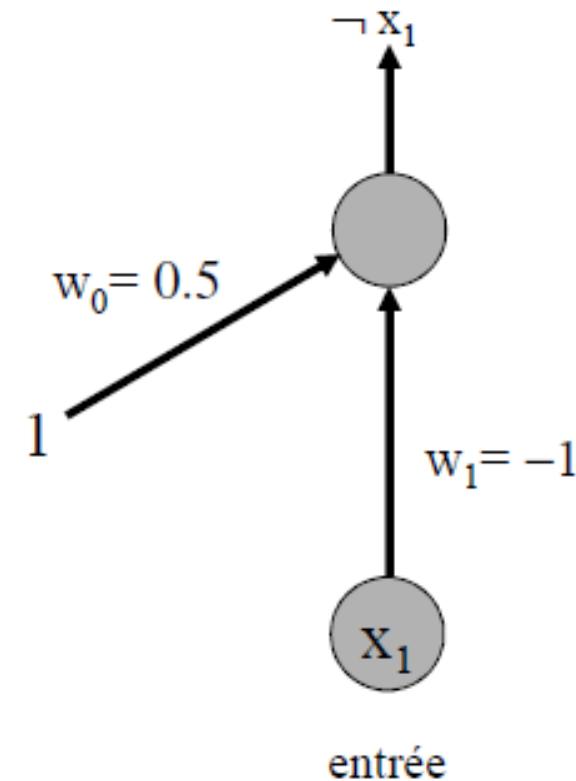


Le modèle "Perceptron"



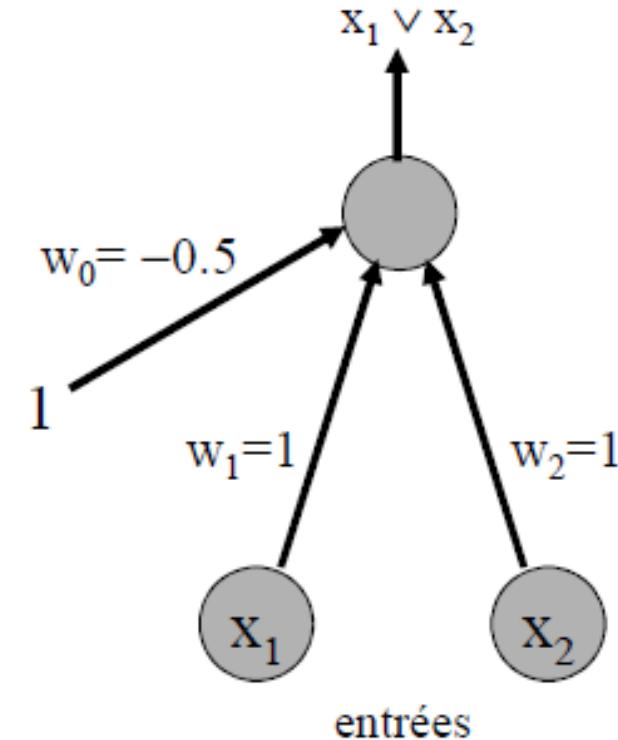
Exemple: négation logique

entrée x_1	sortie
0	1
1	0



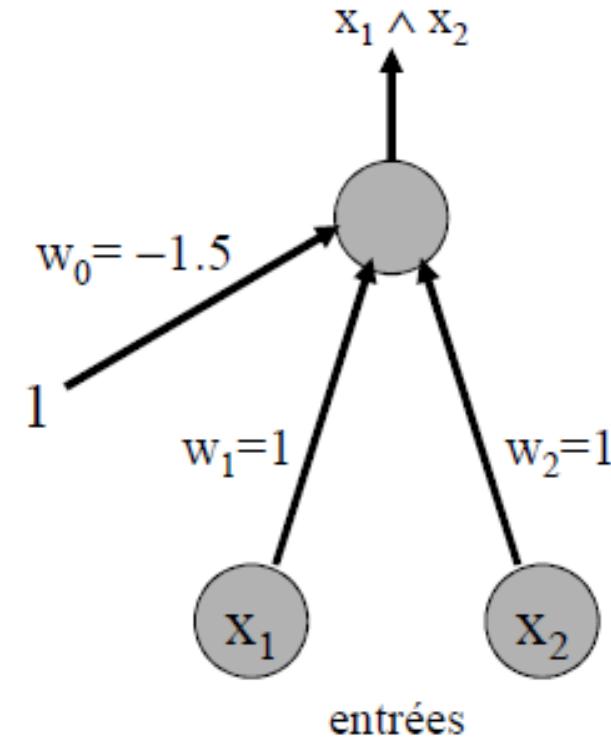
Exemple: "ou" logique

entrée x_1	entrée x_2	sortie
0	0	0
0	1	1
1	0	1
1	1	1



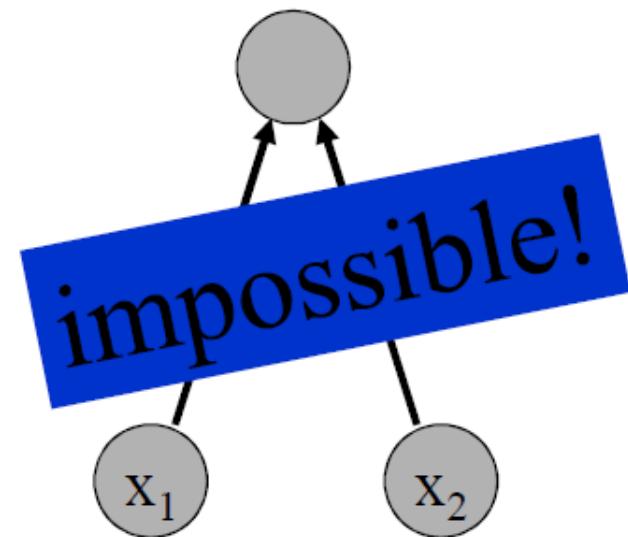
Exemple: "et" logique

entrée x_1	entrée x_2	sortie
0	0	0
0	1	0
1	0	0
1	1	1



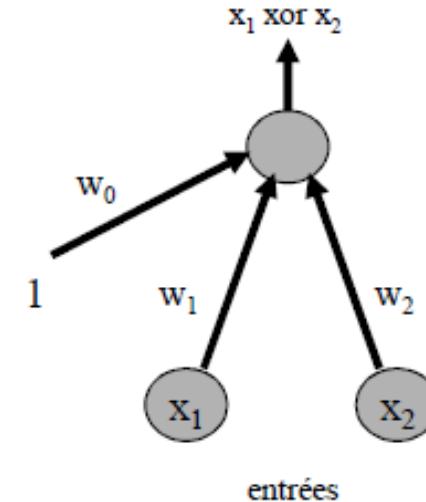
Exemple: "xor" logique (ou exclusif)

entrée x_1	entrée x_2	sortie
0	0	0
0	1	1
1	0	1
1	1	0



Pourquoi "impossible" ?

entrée x1	entrée x2	sortie
0	0	0
0	1	1
1	0	1
1	1	0

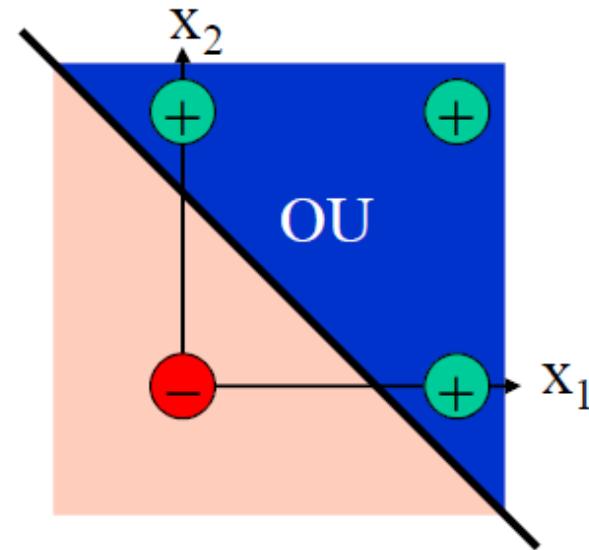


Le système d'équations à résoudre est:

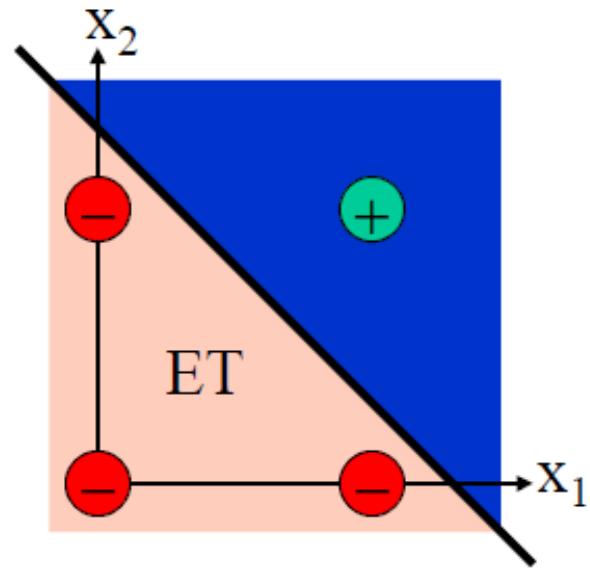
$$\begin{aligned} w_0 + 0.w_1 + 0.w_2 &\leq 0 \\ w_0 + 0.w_1 + 1.w_2 &> 0 \\ w_0 + 1.w_1 + 0.w_2 &> 0 \\ w_0 + 1.w_1 + 1.w_2 &\leq 0 \end{aligned}$$

Il n'y a aucune valeur possible pour les coefficients w_0 , w_1 et w_2 qui satisfasse les inégalités ci-contre.
xor ne peut donc pas être représenté!

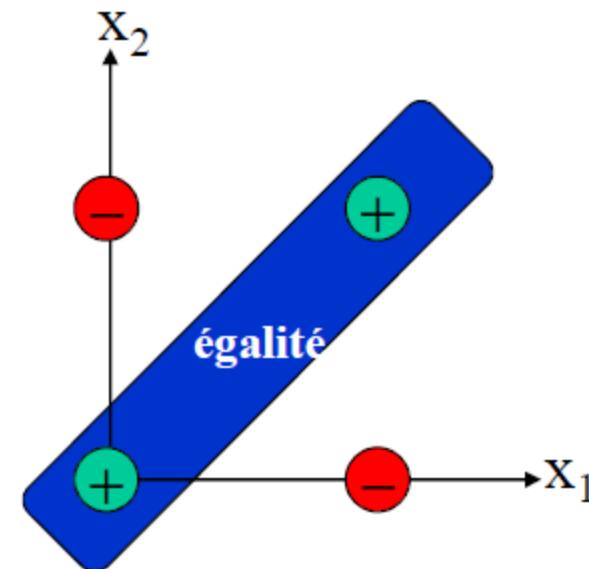
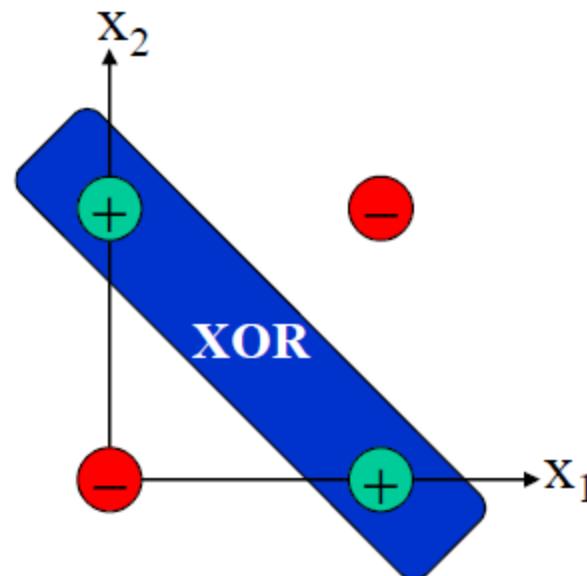
Séparabilité linéaire



Séparabilité linéaire

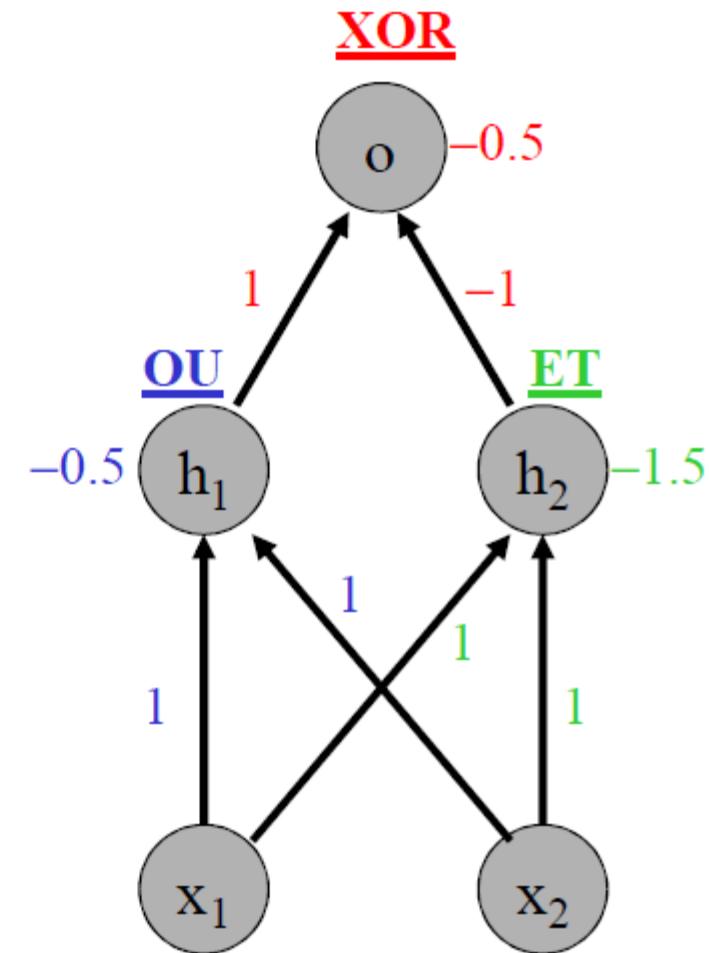


Non-séparabilité linéaire



Exemple: "xor" logique (revu)

entrée x1	entrée x2	sortie
0	0	0
0	1	1
1	0	1
1	1	0



Théorèmes du Perceptron

Théorème de représentation

Un réseau "feedforward" à une seule couche (Perceptron) peut uniquement représenter des fonctions linéairement séparables. C'est-à-dire celles pour lesquelles la surface de décision séparant les cas positifs des cas négatifs est un (hyper-)plan.

Théorème d'apprentissage (F. Rosenblatt)

Étant donné suffisamment d'exemples d'apprentissage, il existe un algorithme qui apprendra n'importe quelle fonction linéairement séparable.

Algorithme d'apprentissage du Perceptron

Entrées: ensemble d'apprentissage $\{(x_1, x_2, \dots, x_n, t)\}$

Méthode

initialiser aléatoirement les poids $w(i)$, $0 \leq i \leq n$

répéter jusqu'à convergence:

pour chaque exemple

calculer la valeur de sortie o du réseau.

ajuster les poids:

$$\Delta w_i = \eta (t - o) x_i$$

$$w_i \leftarrow w_i + \Delta w_i$$

Règle d'apprentissage
du *Perceptron*

Algorithme d'apprentissage du Perceptron (résumé)

$$w_i \leftarrow w_i + \Delta w_i$$

nouveau poids ancien poids modification
 \downarrow \downarrow \downarrow
 w_i \leftarrow w_i + Δw_i

$$\Delta w_i = \eta (t - o) x_i$$

modification taux valeur entrée
 \uparrow \uparrow \uparrow \uparrow
 d'apprentissage d'apprentissage attendue de sortie
 \uparrow \uparrow \uparrow \uparrow
 valeur entrée attendue du Perceptron

Erreur quadratique

- La règle d'apprentissage du *Perceptron* effectue une descente de gradient dans l'espace des poids.
- Considérons une unité linéaire simple pour laquelle

$$o = w_0 + w_1 x_1 + \dots + w_n x_n$$

- définissons l'erreur comme:

$$E[w_0, w_1, \dots, w_n] = \frac{1}{2} \sum_{e \in \text{Exemples}} (t_e - o_e)^2$$

(*erreur quadratique*)

Convergence

La convergence est garantie car l'erreur E est une forme quadratique dans l'espace des poids. Elle possède donc un seul minimum global et la descente du gradient assure de le trouver.

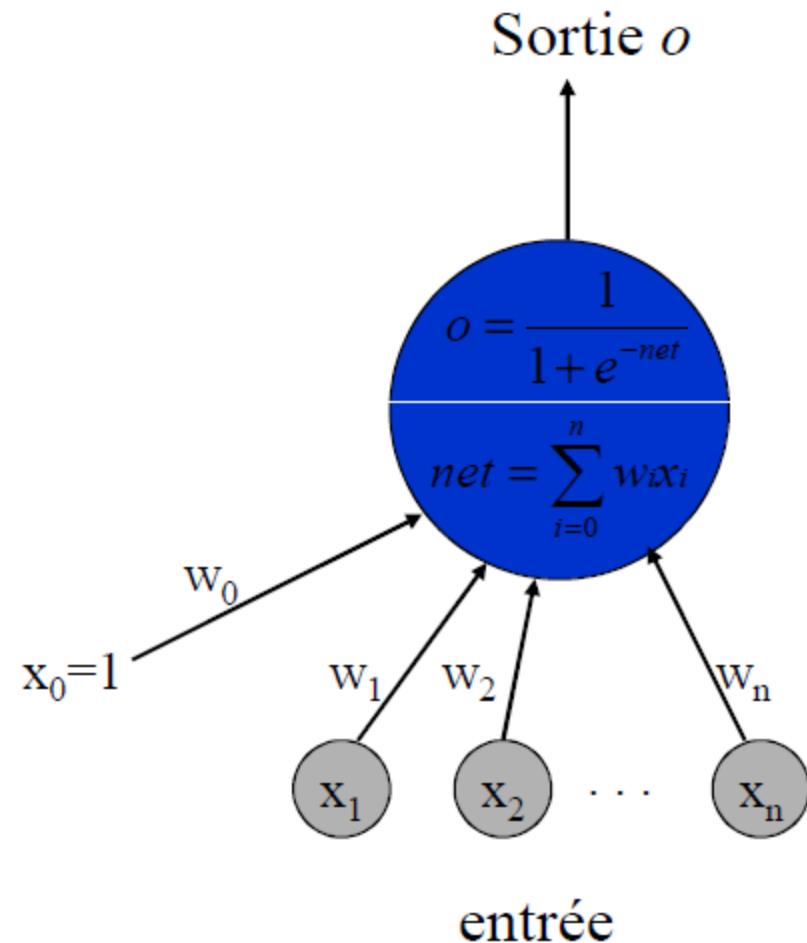
Convergence si:

- ... les données d'apprentissage sont linéairement séparables
- ... le taux d'apprentissage η est suffisamment petit
- ... il n'y a pas d'unités "cachées" (une seule couche)

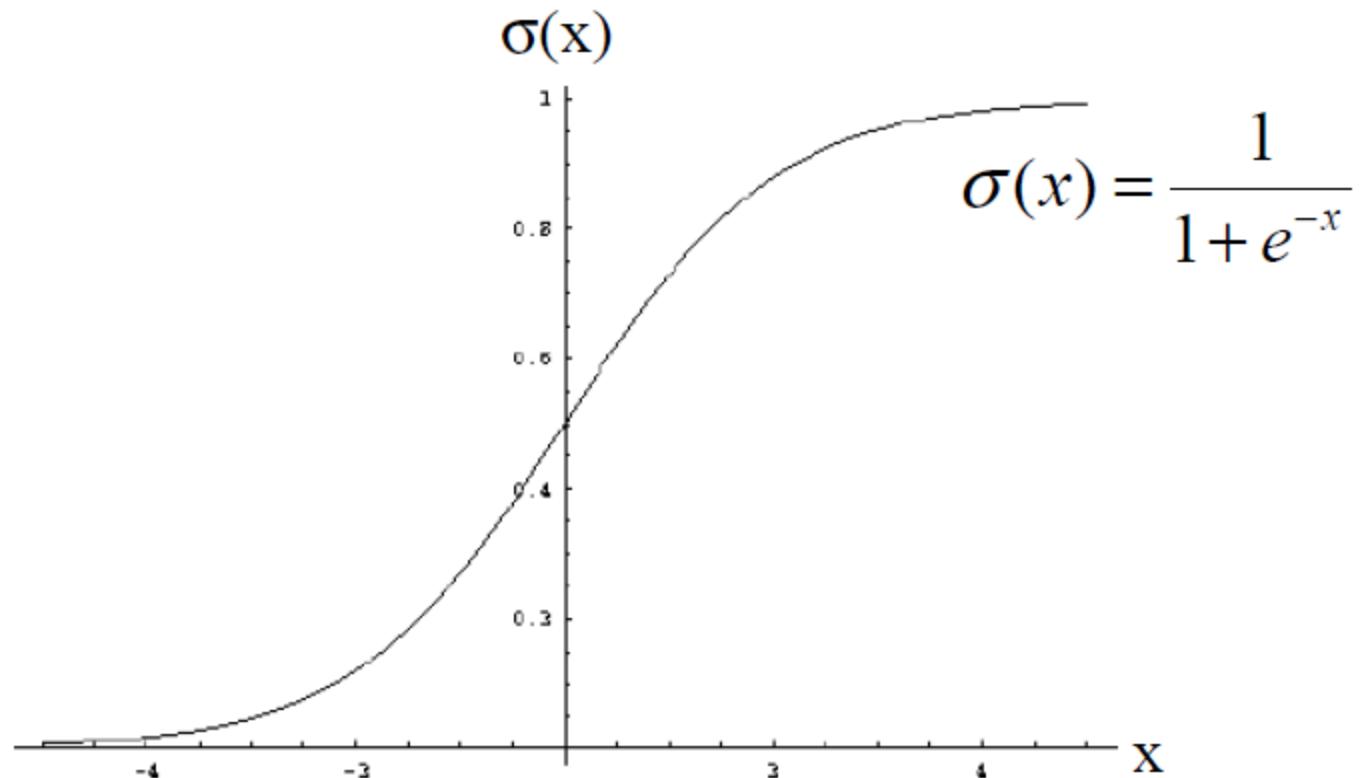
Réseaux multi-couches

- Un réseau à 2 couches (une couche cachée) avec des unités à seuil peut représenter la fonction logique "ou exclusif" (xor).
- Les réseaux multi-couches "feedforward" peuvent être entraînés par rétro-propagation pour autant que la fonction de transition des unités soit différentiable (les unités à seuil ne conviennent donc pas).

Unité "sigmoïde"



Fonction sigmoïde



$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

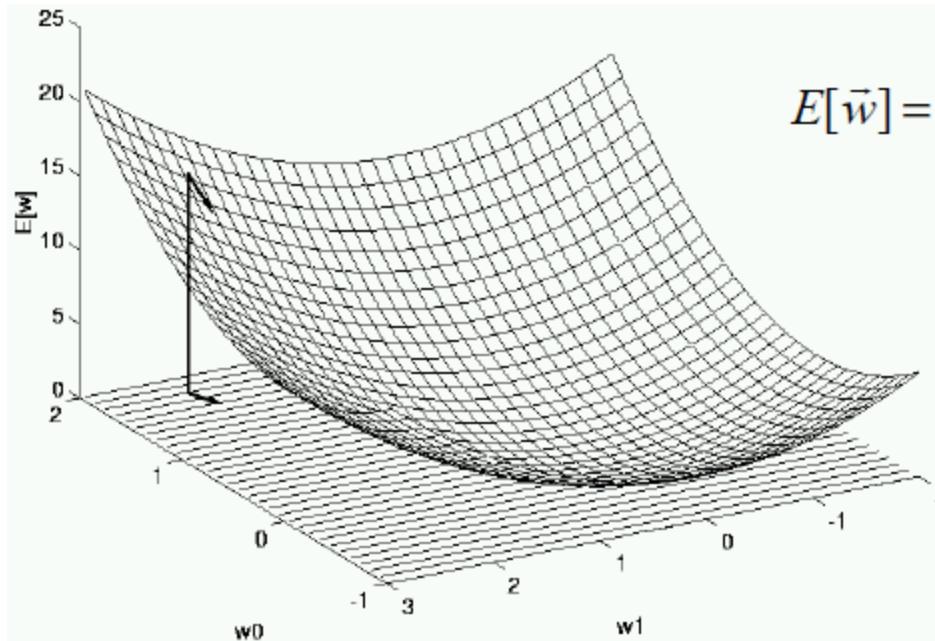
Base pour le procédé de descente gradient

Descente de gradient

- Il faut apprendre les valeurs des poids w_i qui minimisent l'erreur quadratique

$$E[\vec{w}] = \frac{1}{2} \sum_e (t_e - o_e)^2$$

Descente de gradient (suite)



$$E[\vec{w}] = \frac{1}{2} \sum_e (t_e - o_e)^2$$

Gradient:

$$\nabla E[\vec{w}] = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

Règle d'apprentissage:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}] \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Descente de gradient (suite)

$$\begin{aligned}
 \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_e (t_e - o_e)^2 \\
 &= \frac{1}{2} \sum_e \frac{\partial}{\partial w_i} (t_e - o_e)^2 \\
 &= \frac{1}{2} \sum_e 2(t_e - o_e) \frac{\partial}{\partial w_i} (t_e - o_e) \\
 &= \sum_e (t_e - o_e) \frac{\partial}{\partial w_i} (t_e - \sigma(\vec{w} \cdot \vec{x}_e)) \\
 &= \sum_e (t_e - o_e) \left(-\frac{\partial}{\partial w_i} \sigma(\vec{w} \cdot \vec{x}_e) \right) \\
 &= -\sum_e (t_e - o_e) \sigma(\vec{w} \cdot \vec{x}_e) (1 - \sigma(\vec{w} \cdot \vec{x}_e)) x_{i,e} \\
 &= -\sum_e (t_e - o_e) o_e (1 - o_e) x_{i,e}
 \end{aligned}$$

Ajustement des poids

- Le terme d'erreur ($t_e - o_e$) est connu pour les unités de la couche de sortie. Pour ajuster les poids entre la couche cachée et celle de sortie, on peut utiliser le processus de descente de gradient.
- Pour ajuster les poids entre la couche d'entrée et la couche cachée, il faut trouver le moyen d'estimer les erreurs commises par les unités de la couche cachée.

Rétro-propagation de l'erreur

Idée: chaque unité cachée est responsable pour une fraction de l'erreur commise par chacune des unités de sortie.

$$\text{erreur pour l'unité cachée } j = \sum_{i \in \text{sorties}} w_{ij} \delta_i$$

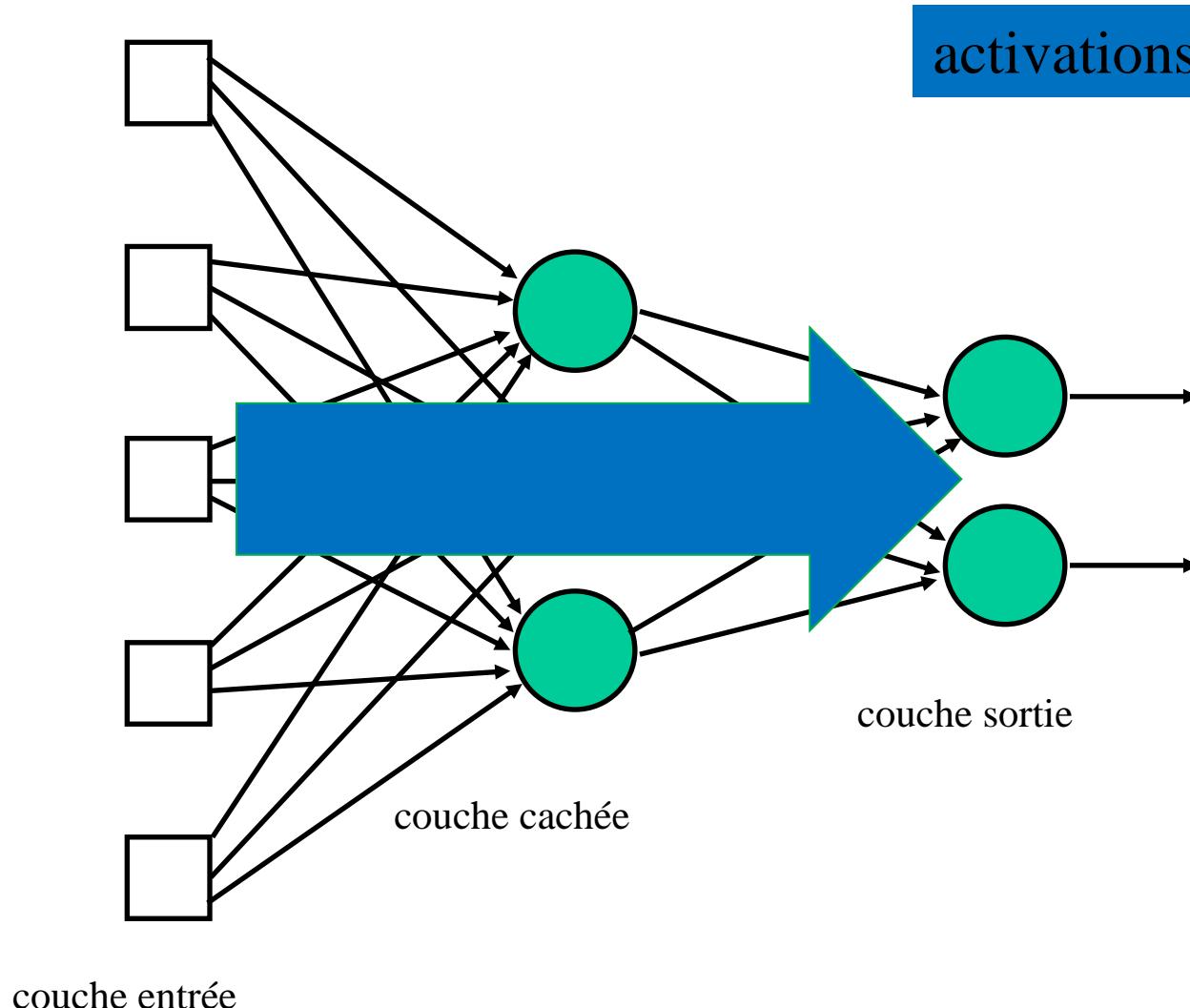
où δ_i est l'erreur pour l'unité de sortie i

Algorithme de rétro-propagation

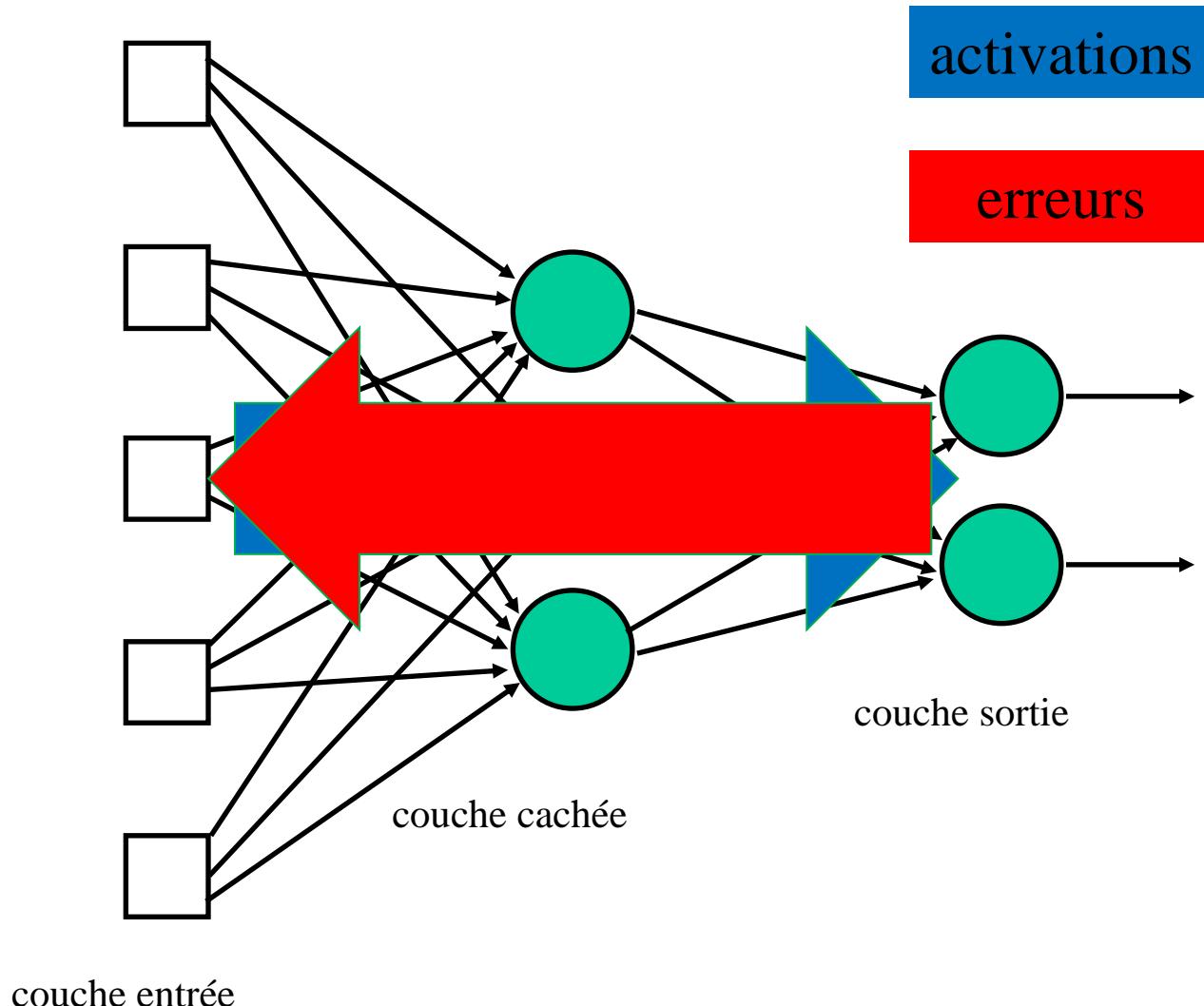
- Initialiser tous les poids du réseau avec de petites valeurs aléatoires.
- Pour chaque exemple de l'ensemble d'apprentissage faire:

- pour chaque unité cachée h , calculer:
$$o_h = \sigma\left(\sum_i w_{hi}x_i\right)$$
- pour chaque unité de sortie k , calculer:
$$o_k = \sigma\left(\sum_k w_{kh}x_h\right)$$
- pour chaque unité de sortie k , calculer:
$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$
- pour chaque unité cachée h , calculer:
$$\delta_h = o_h(1 - o_h)\sum_k w_{hk}\delta_k$$
- mettre à jour chaque poids du réseau:
$$w_{ij} \leftarrow w_{ij} + \eta \delta_j x_{ij}$$

Algorithme de rétro-propagation (suite)



Algorithme de rétro-propagation (suite)



Caractéristiques de la rétro-propagation

- Processus de descente de gradient effectué sur tout le vecteur des poids du réseau.
- Minimum d'erreur local, pas nécessairement global.
- Minimisation de l'erreur sur l'ensemble d'apprentissage; risque de sur-apprentissage comme pour les arbres de décision.
- Apprentissage nécessite des milliers d'itérations -- très lent!
- Décider de la topologie et fixer les valeurs des paramètres (par ex: taux d'apprentissage) est plus un *art* qu'une science.

Apprentissage incrémental (en ligne)

- Initialiser chaque poids w_i à une petite valeur aléatoire
- répéter jusqu'à terminaison

pour chaque exemple d'apprentissage e faire

$$\Delta w_i = 0$$

$$o_e \leftarrow \sigma(\sum_i w_i x_{i,e})$$

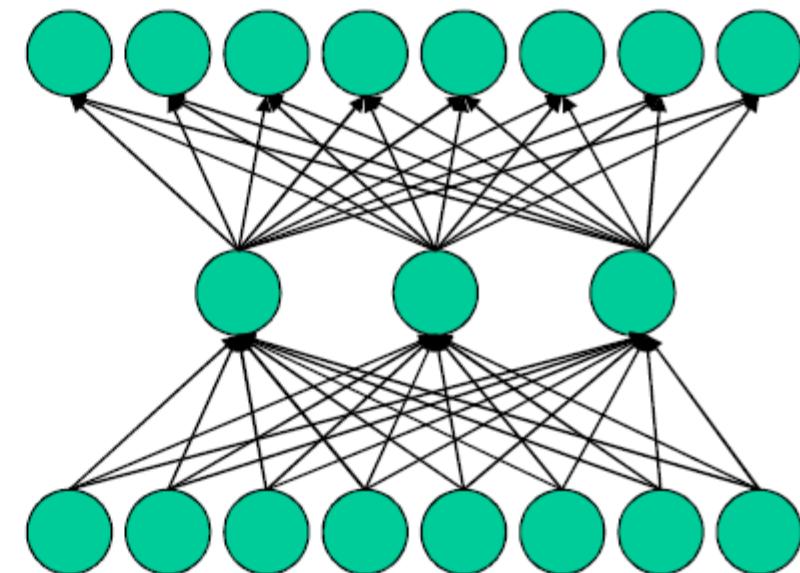
$$\Delta w_i \leftarrow \Delta w_i + \eta (t_e - o_e) o_e (1-o_e) x_{i,e}$$

$$\longrightarrow w_i \leftarrow w_i + \Delta w_i$$

Ceci peut-il être appris ?

encodeur - décodeur

Entrée		Sortie
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

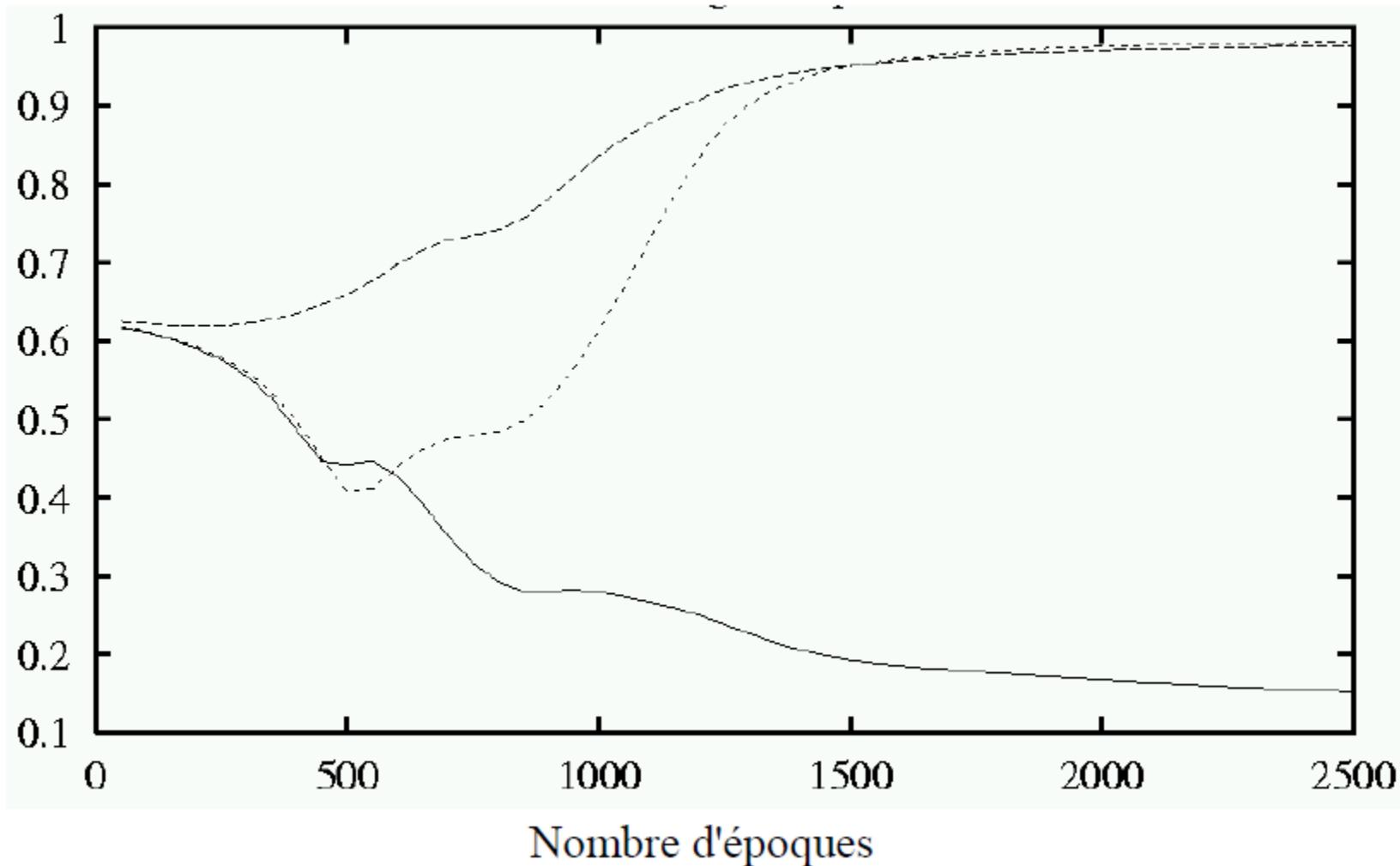


Représentation dans la couche cachée

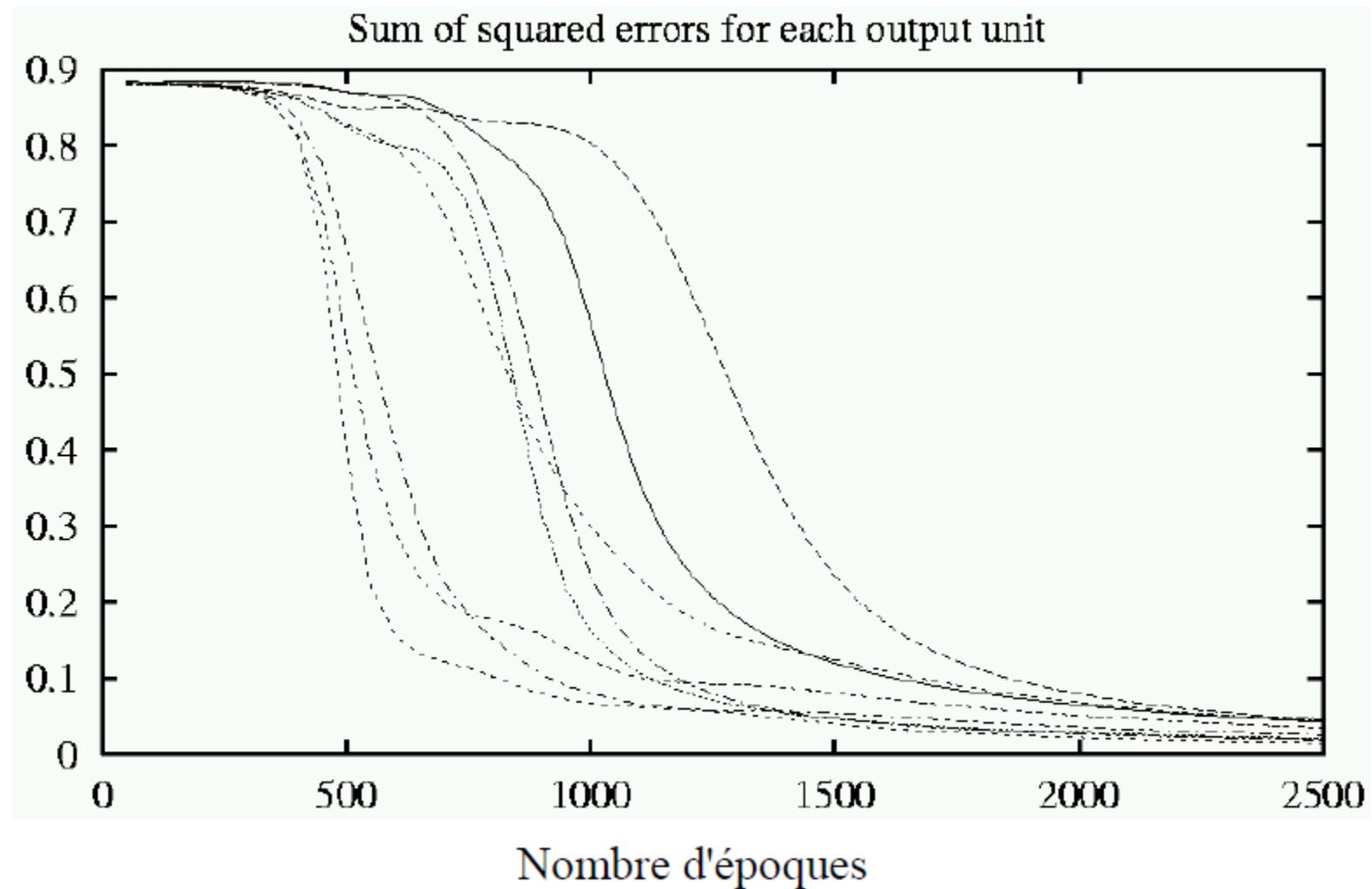
Entrée				Sortie
10000000	→	.89 .04 .08	→	10000000
01000000	→	.01 .11 .88	→	01000000
00100000	→	.01 .97 .27	→	00100000
00010000	→	.99 .97 .71	→	00010000
00001000	→	.03 .05 .02	→	00001000
00000100	→	.15 .99 .99	→	00000100
00000010	→	.80 .01 .98	→	00000010
00000001	→	.60 .94 .01	→	00000001

Apprentissage: représentation interne

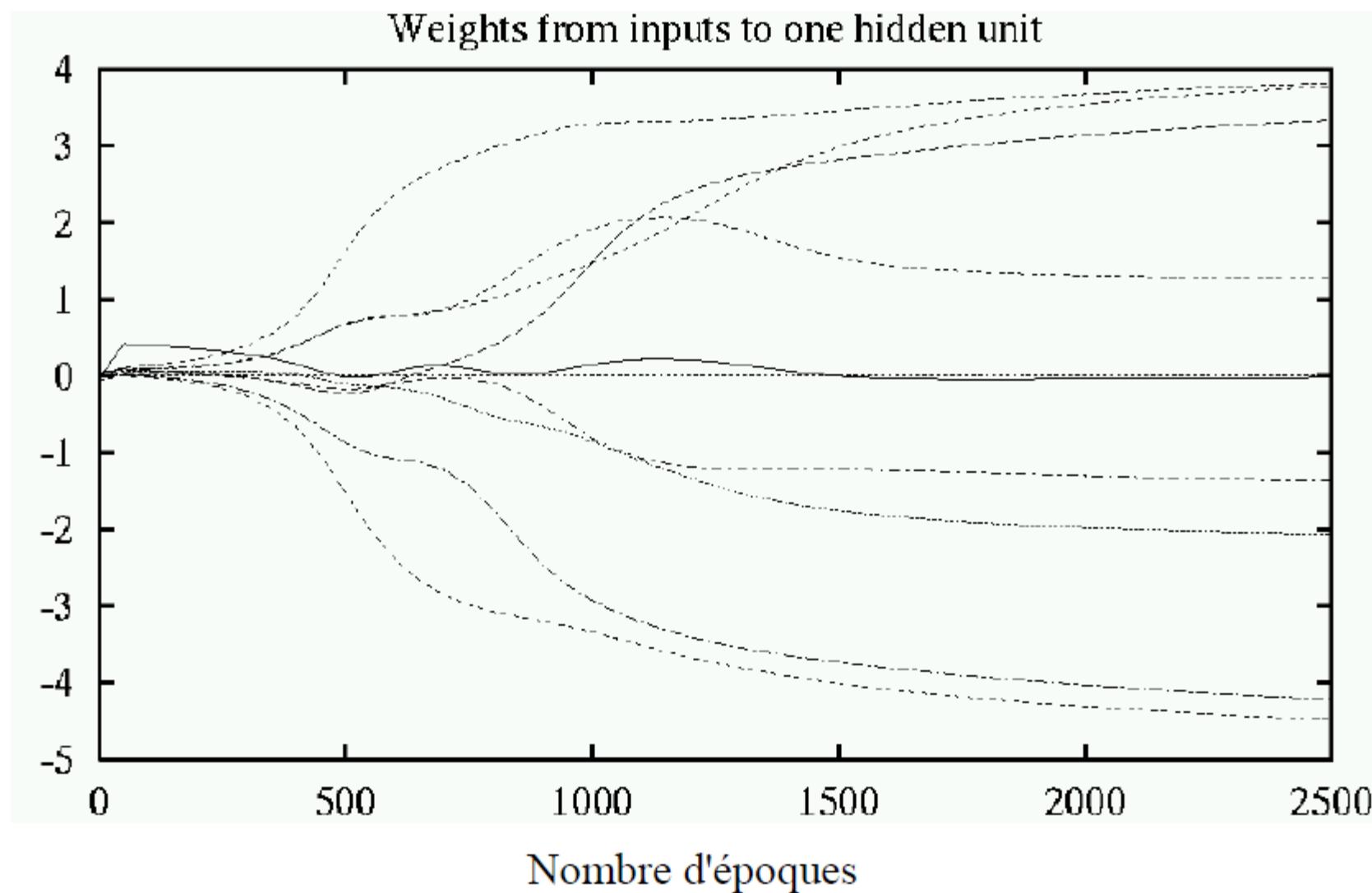
Codage des unités cachées pour l'entrée 00000100



Apprentissage: erreur



Apprentissage: valeurs des poids



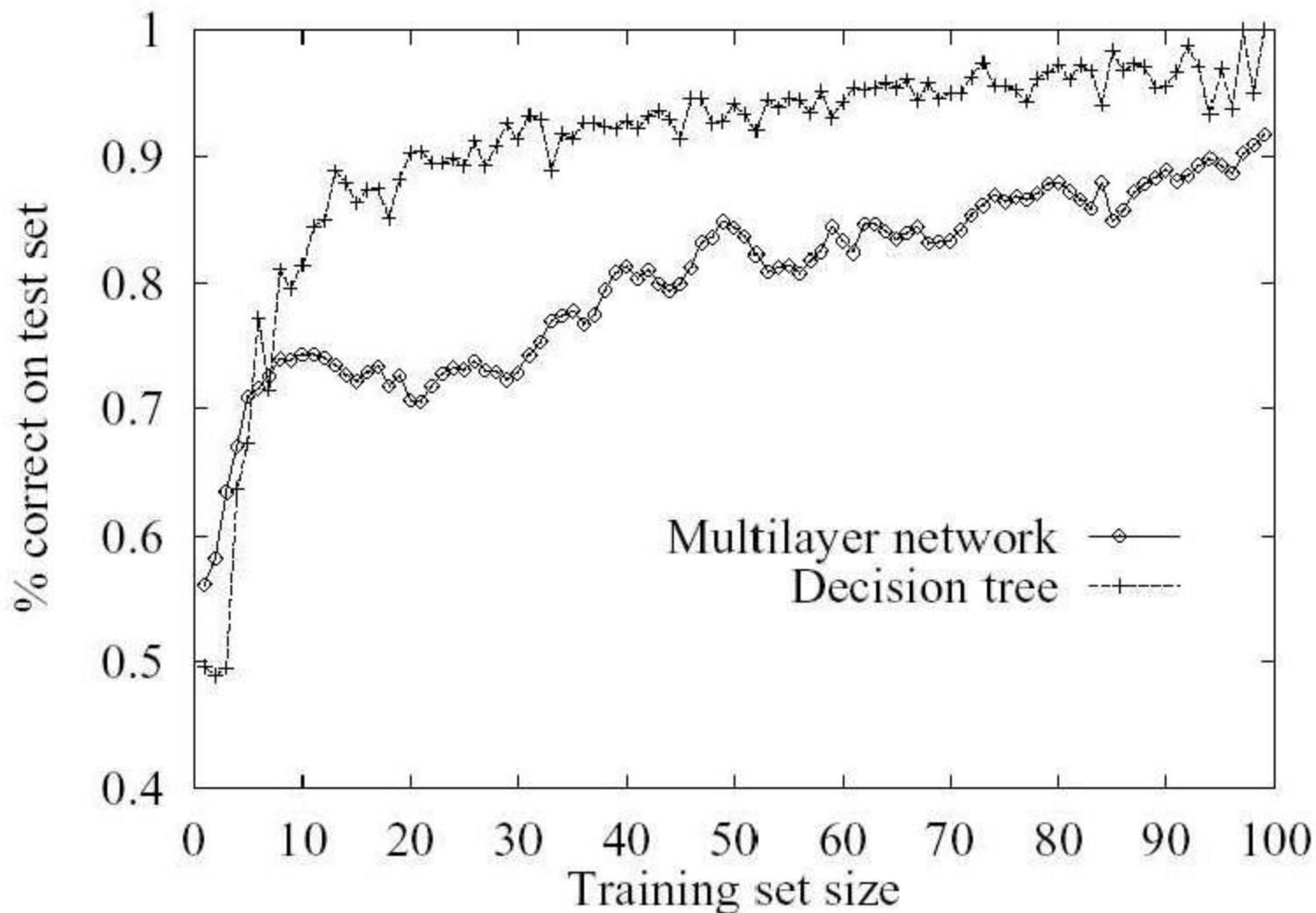
Théorème de Kolmogorov

N'importe quelle fonction continue

$$f: [0,1]^n \rightarrow \mathbb{R}^m$$

peut être réalisée par un réseau à une couche cachée avec n unités d'entrée, $(2n+1)$ unités dans la couche cachée et m unités de sortie.

Réseaux vs arbre de décision



Question ?

Quand utiliser des réseaux de neurones artificiels pour résoudre des problèmes d'apprentissage?

- lorsque les données sont représentées par des paires attributs-valeur,
- lorsque les exemples d'apprentissage sont bruités,
- lorsque des temps d'apprentissage (très) longs sont acceptables,
- lorsqu'une évaluation rapide de la fonction apprise est nécessaire,
- lorsque la compréhension par l'utilisateur de la fonction apprise est sans importance.

Observations (suite)

- Expressivité — des réseaux avec une couche cachée et assez d'unités dans cette couche peuvent représenter n'importe quelle fonction continue (théorème de V. Vapnik):
 - dans certains cas la représentation est concise,
 - ex: un perceptron de taille n peut représenter la fonction "majorité" de n -entrées; cette même fonction nécessiterait un arbre de décision de taille 2^n .
 - dans d'autres cas la représentation n'est pas aussi concise,
 - ex: $2^n/n$ unités cachées sont nécessaires pour représenter toutes les fonctions logiques de taille n .
- Interprétabilité — la connaissance d'un RNA réside dans les valeurs des poids des connexions; interpréter ces valeurs est très difficile,
 - les arbres de décision et les formules logiques sont beaucoup plus faciles à interpréter et donc d'expliquer leur valeurs de sortie,
 - c'est particulièrement important lorsqu'on prescrit un médicament ou qu'on refuse un prêt à quelqu'un.

Bibliographie "papier"

- Hopfield, J. J. (1982). *Neural networks and physical systems with emergent collective computational abilities*. Proceedings of the National Academy of Sciences 79:2554-2558.
- Hertz, J., A. Krogh, and R. G. Palmer. (1991). *Introduction to the Theory of Neural Computation*. Redwood City, CA: Addison-Wesley.
- Rumelhart, D. E., J. McClelland, and the PDP Research Group. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press.
- Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. New York: Oxford University Press.
- Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. New York: Macmillan College Publishing.
- Churchland, P. S., and T. J. Sejnowski. (1992). *The Computational Brain*. Cambridge, MA: MIT Press.
- Arbib, A. M. (1995). *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA: MIT Press.
- Sejnowski, T. (1997). *Computational neuroscience*. *Encyclopedia of Neuroscience*. Amsterdam: Elsevier Science Publishers.

Bibliographie Internet

Références dans "*MIT Encyclopedia of Cognitive Science*"

- home page:
 - <http://cognet.mit.edu/MITECS/Front/introduction.html>
- neural networks (M. Jordan):
 - <http://cognet.mit.edu/MITECS/Entry/jordan2.html>
- cognitive modeling, connectionism (J. McClelland):
 - <http://cognet.mit.edu/MITECS/Entry/mcclelland.html>
- computation and the Brain (P. Churchland and R. Grush):
 - <http://cognet.mit.edu/MITECS/Entry/churchland.html>
- computational Neuroscience (T. Sejnowski)
 - <http://cognet.mit.edu/MITECS/Entry/sejnowski.html>