

Objectifs

- Concepts de base en apprentissage automatique (*machine learning*)
- Algorithme des k plus proches voisins
- Classification linéaire avec le Perceptron et la régression logistique
 - ◆ dérivées partielles
 - ◆ descente de gradient (stochastique)
- Réseau de neurones artificiel
 - ◆ dérivation en chaîne
 - ◆ rétropropagation (*backpropagation*)

Apprentissage automatique

- Un agent **apprend** s'il améliore sa performance sur des tâches futures avec l'expérience
- On va se concentrer sur un problème d'apprentissage simple mais ayant beaucoup d'applications:

« Étant donnée une collection de paires (**entrées, sorties**) appelées **exemples d'apprentissage**, comment apprendre une **fonction** qui peut prédire correctement une sortie étant donnée une **nouvelle entrée**. »

Apprentissage automatique

- Pourquoi programmer des programmes qui apprennent:
 - ◆ il est trop difficile d'anticiper toutes les entrées à traiter correctement
 - ◆ il est possible que la relation entre l'entrée et la sortie **évolue dans le temps** (ex.: classification de pourriels)
 - ◆ parfois, on a aucune idée comment programmer la fonction désirée (ex.: reconnaissance de visage)

Types de problèmes d'apprentissage

- Il existe plusieurs sortes de problèmes d'apprentissage, qui se distinguent par la nature de la supervision offerte par nos données
 - ◆ **apprentissage supervisé** : sortie désirée (cible ou « *target* ») est fournie explicitement par les données
 - » ex.: reconnaissance de caractères, à l'aide d'un ensemble de paires (images, identité du caractère)

Types de problèmes d'apprentissage

- Il existe plusieurs sortes de problèmes d'apprentissage, qui se distinguent par la nature de la supervision offerte par nos données
 - ◆ **apprentissage non-supervisé** : les données ne fournissent pas de signal explicite et le modèle doit extraire de l'information uniquement à partir de la structure des entrées
 - » ex.: identifier différents thèmes d'articles de journaux en regroupant les articles similaires (« *clustering* »)

Types de problèmes d'apprentissage

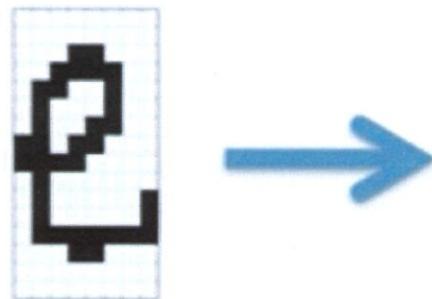
- Il existe plusieurs sortes de problèmes d'apprentissage, qui se distinguent par la nature de la supervision offerte par nos données
 - ◆ **apprentissage par renforcement** : le signal d'apprentissage correspond seulement à des récompenses et punitions
 - » ex.: est-ce que le modèle a gagné sa partie d'échec (1) ou pas (-1)

Intelligence Artificielle

Apprentissage automatique - définitions

Représentation des données

- L'entrée **X** est représentée par un vecteur de valeurs d'attributs réels (représentation factorisée)
 - ◆ ex.: une image est représentée par un vecteur contenant la valeur de chacun des pixels



```
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
       0.,  0.,  0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,
       1.,  1.,  1.,  1.,  0.,  0.,  1.,  1.,  1.,  0.,  0.,  1.,  0.,
       0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.,
       1.,  1.,  0.,  0.,  1.,  1.,  0.,  1.,  1.,  0.,  0.,  1.,  0.,
       1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.,
       1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,
       0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,
       0.,  0.,  0.,  0.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
       0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
       0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

- La sortie désirée ou **cible** *y* aura une représentation différente selon le problème à résoudre:
 - ◆ problème de classification en *C* classes: valeur discrète (index de 0 à *C*-1)
 - ◆ problème de régression: valeur réelle ou continue

Apprentissage supervisé

- Un problème d'apprentissage supervisé est formulé comme suit:
« Étant donné un **ensemble d'entraînement** de N exemples:

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N) \} D$$

où chaque y_j a été générée par une **fonction inconnue** $y = f(\mathbf{x})$,
découvrir une nouvelle fonction h (**modèle ou hypothèse**)
qui sera une bonne approximation de f (c'est à dire $f(\mathbf{x}) \approx h(\mathbf{x})$) »

- Un algorithme d'apprentissage peut donc être vu comme étant une fonction A
à laquelle on donne un ensemble d'entraînement et qui donne en retour cette
fonction h

$$A(D) = h$$

Apprentissage automatique - k plus proches voisins

Classifieur k plus proches voisins

- Possiblement l'algorithme d'apprentissage pour la classification le plus simple
- **Idée:** étant donnée une entrée \mathbf{X}
 1. trouver les k entrées \mathbf{x}_t parmi les exemples d'apprentissage qui sont les plus « proches » de \mathbf{X}
 2. faire voter chacune de ces entrées pour leur classe associée y_t
 3. retourner la classe majoritaire
- Le succès de cet algorithme va dépendre de deux facteurs
 - ◆ la quantité de données d'entraînement (plus il y en a, meilleure sera la performance)
 - ◆ la qualité de la mesure de distance (2 entrées similaires sont-elles de la même classe?)
 - » en pratique, on utilise souvent la distance Euclidienne:

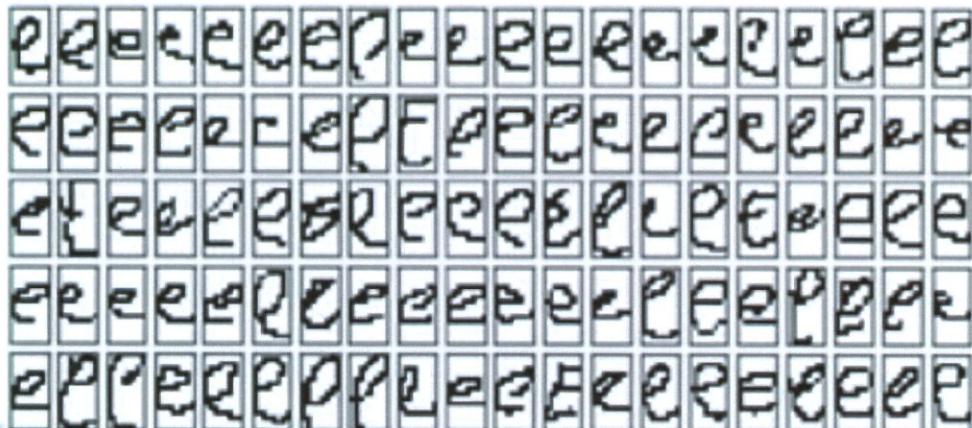
$$d(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_k (x_{1,k} - x_{2,k})^2}$$

\mathbf{x} = vecteur
 x = scalaire

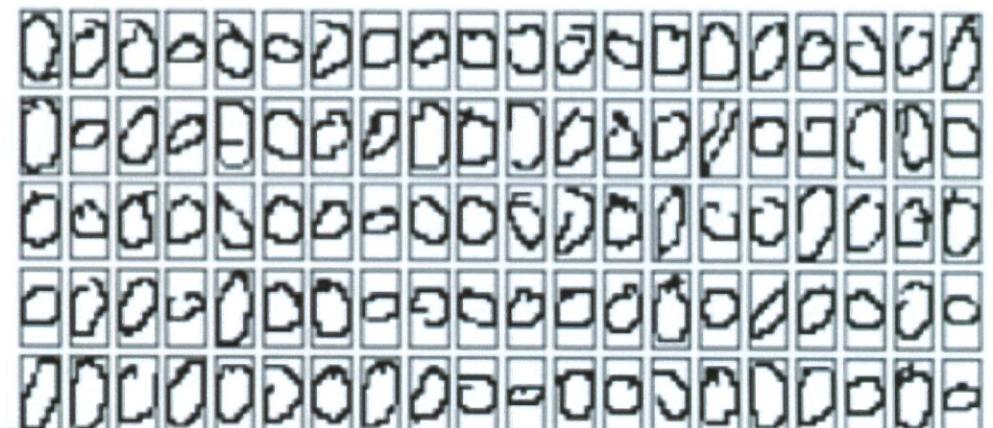
Illustration: 3 plus proches voisins

- Reconnaissance de caractère: est-ce un 'e' ou un 'o'?

Ensemble d'entraînement
(100 exemples d'apprentissage par classe)



Classe 'e'



Classe 'o'

Illustration: 3 plus proches voisins

- Reconnaissance de caractère: est-ce un 'e' ou un 'o'?

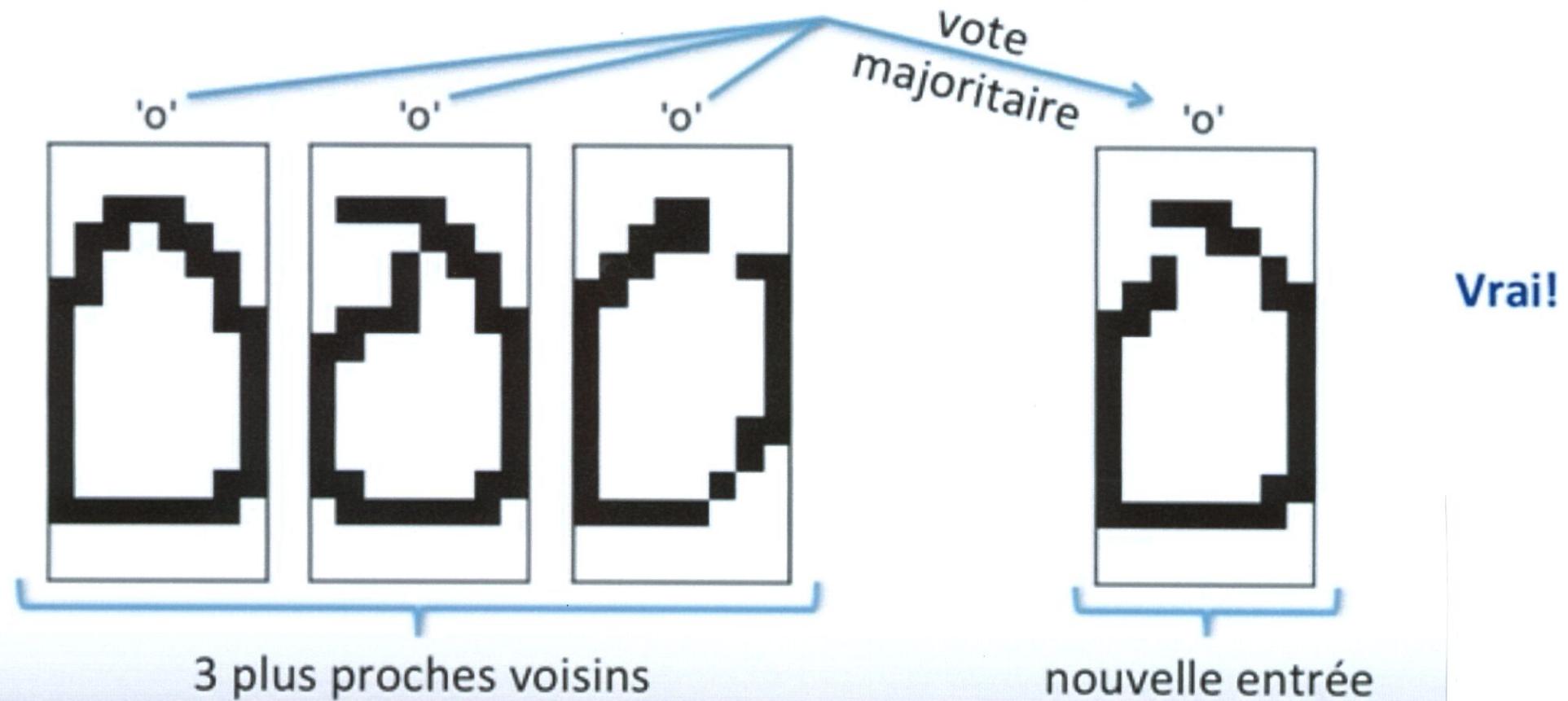


Illustration: 3 plus proches voisins

- Reconnaissance de caractère: est-ce un 'e' ou un 'o'?

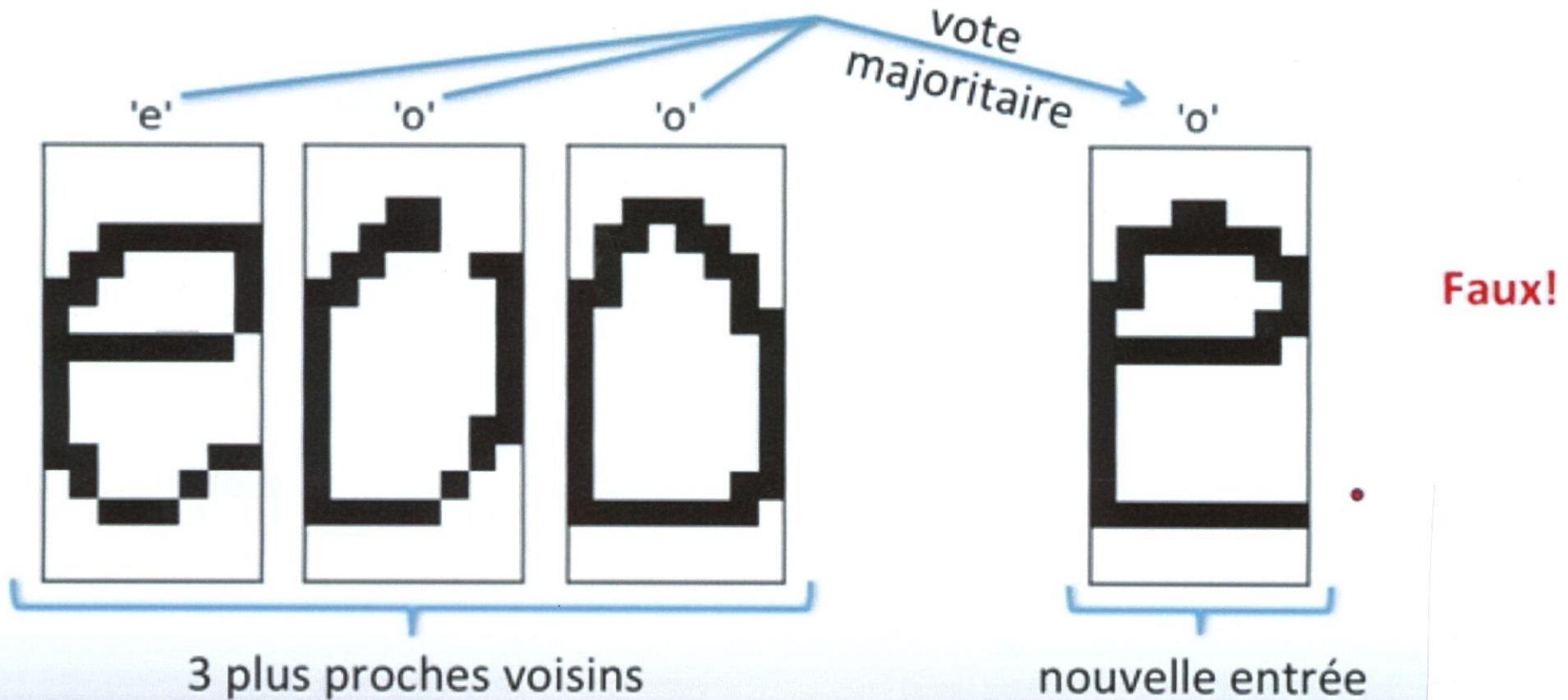
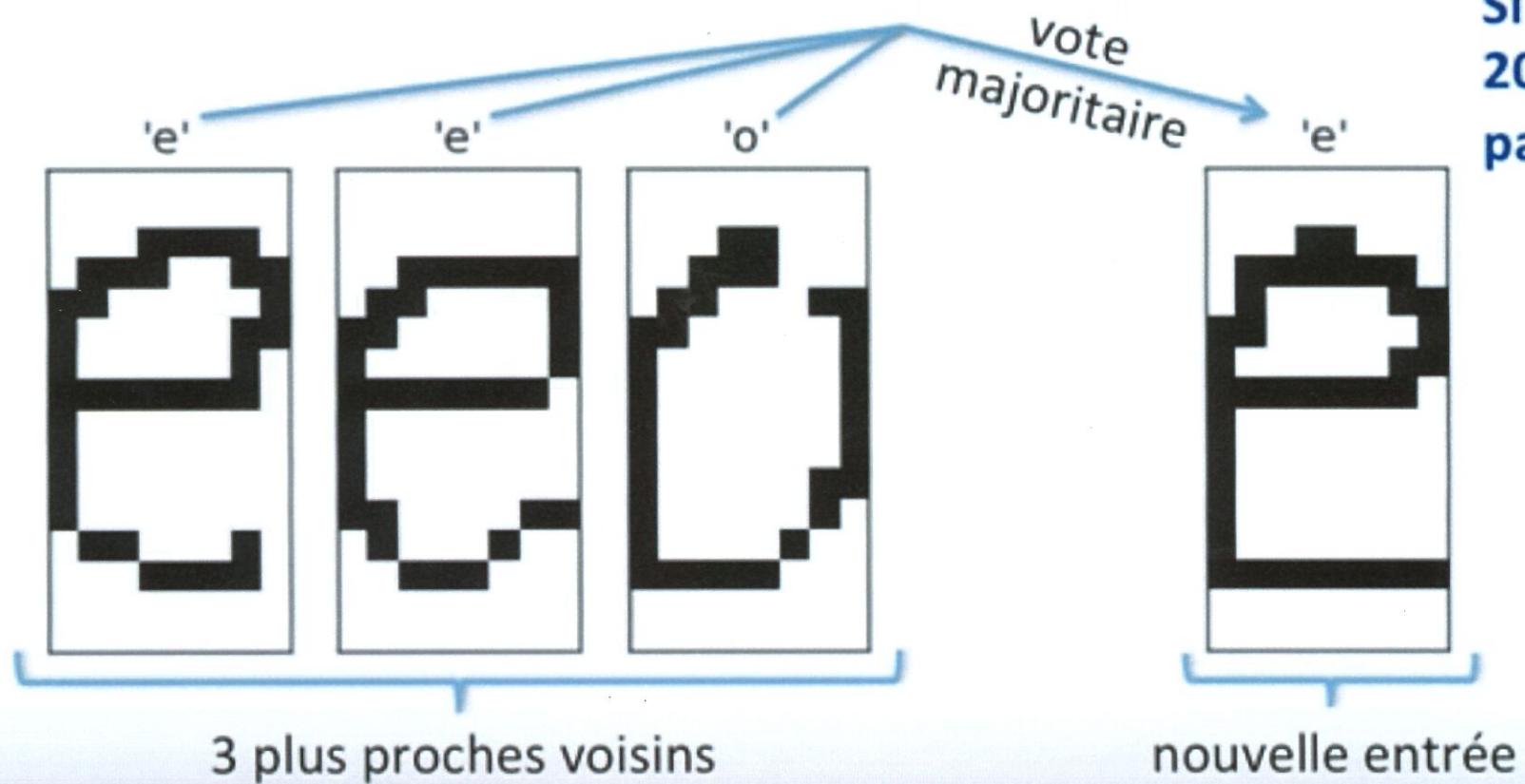


Illustration: 3 plus proches voisins

- Reconnaissance de caractère: est-ce un 'e' ou un 'o'?



Si on ajoute
200 exemples
par classe...

Vrai!

Apprentissage supervisé

- Un problème d'apprentissage supervisé est formulé comme suit:
« Étant donné un **ensemble d'entraînement** de N exemples:

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N) \quad D$$

où chaque y_j a été généré par une **fonction inconnue** $y = f(\mathbf{x})$,
découvrir une nouvelle fonction h (**modèle ou hypothèse**)
qui sera une bonne approximation de f (c'est à dire $f(\mathbf{x}) \approx h(\mathbf{x})$) »

- Un algorithme d'apprentissage peut donc être vu comme étant une fonction A
à laquelle on donne un ensemble d'entraînement et qui donne en retour cette
fonction h

$$A(D) = h$$

Retour sur classifieur k plus proches voisins

- Dans le cas de l'algorithme k plus proches voisins:
 - ◆ A est un programme qui produit lui-même un programme, soit celui qui fait une prédiction à l'aide de la procédure k plus proches voisins
 - ◆ $h = A(D)$ est le programme qui fait voter les k plus proches voisins dans D d'une entrée donnée
 - ◆ $h(\mathbf{x})$ est la sortie du programme pour l'entrée \mathbf{X} , c'est à dire une prédiction de la classe de \mathbf{X}
 - ◆ f est la « fonction » qui a généré nos données d'entraînement
 - » ex.: l'être humain qui a étiqueté les images de caractères
- On peut démontrer que plus D est grand, plus h sera une bonne approximation de f
 - ◆ intuition: en augmentant la taille de l'ensemble d'entraînement, les k plus proches voisins ne peuvent qu'être plus proches (plus similaires) à l'entrée

Apprentissage automatique - perceptron

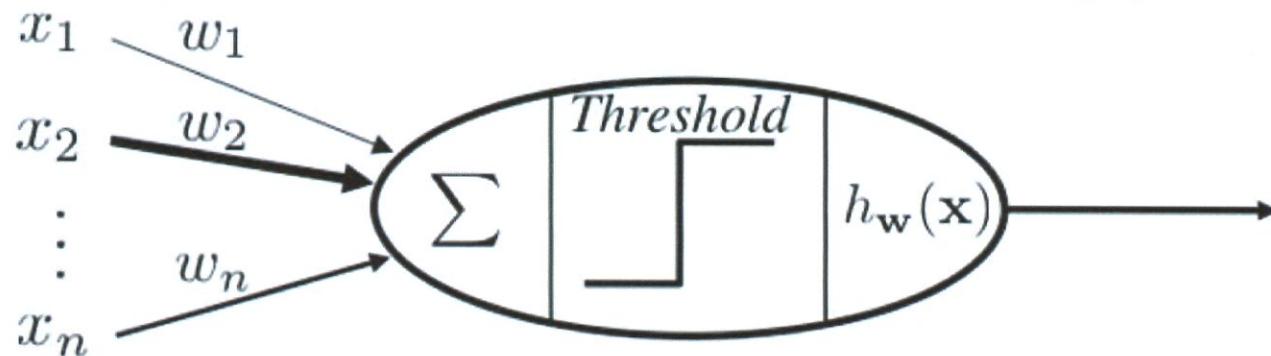
Deuxième algorithme: Perceptron

(Rosenblatt, 1957)

- Idée: modéliser la décision à l'aide d'une fonction linéaire, suivi d'un seuil:

$$h_{\mathbf{w}}(\mathbf{x}) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x})$$

où $\text{Threshold}(z) = 1$ si $z \geq 0$, sinon $\text{Threshold}(z) = 0$



- Le vecteur de poids \mathbf{W} correspond aux paramètres du modèle

Deuxième algorithme: Perceptron

(Rosenblatt, 1957)

- L'algorithme d'apprentissage doit adapter la valeur des paramètres de façon à ce que $h_{\mathbf{w}}(\mathbf{x})$ soit la bonne réponse sur les données d'entraînement

Algorithme du perceptron

1. pour chaque paire $(\mathbf{x}_t, y_t) \in D$
 - a. calculer $h_{\mathbf{w}}(\mathbf{x}_t) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x}_t)$
 - b. si $y_t \neq h_{\mathbf{w}}(\mathbf{x}_t)$
 - $w_i \leftarrow w_i + \alpha(y_t - h_{\mathbf{w}}(\mathbf{x}_t))x_{t,i} \quad \forall i$ (mise à jour des poids et biais)
 2. retourner à 1 jusqu'à l'atteinte d'un critère d'arrêt
(nb. maximal d'itérations atteint ou nb. d'erreurs est 0)
- La mise à jour des poids est appelée la **règle d'apprentissage du perceptron**. Le multiplicateur α est appelé le **taux d'apprentissage**

Deuxième algorithme: Perceptron

(Rosenblatt, 1957)

- L'algorithme d'apprentissage doit adapter la valeur des paramètres de façon à ce que $h_{\mathbf{w}}(\mathbf{x})$ soit la bonne réponse sur les données d'entraînement

Algorithme du perceptron

- pour chaque paire $(\mathbf{x}_t, y_t) \in D$
 - calculer $h_{\mathbf{w}}(\mathbf{x}_t) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x}_t)$
 - si $y_t \neq h_{\mathbf{w}}(\mathbf{x}_t)$
 - $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_t - h_{\mathbf{w}}(\mathbf{x}_t))\mathbf{x}_t$ ← forme vectorielle (mise à jour des poids et biais)
 - retourner à 1 jusqu'à l'atteinte d'un critère d'arrêt
(nb. maximal d'itérations atteint ou nb. d'erreurs est 0)
-
- La mise à jour des poids est appelée la **règle d'apprentissage du perceptron**.
Le multiplicateur α est appelé le **taux d'apprentissage**

Apprentissage automatique - exemple d'exécution du perceptron

Exemple

- Simulation avec biais, $\alpha = 0.1$
- Initialisation : $\mathbf{w} \leftarrow [0, 0]$, $b = 0.5$
- Paire (\mathbf{x}_1, y_1) :
 - ◆ $h(\mathbf{x}_1) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x}_1 + b) = \text{Threshold}(0.5) = 1$
 - ◆ puisque $h(\mathbf{x}_1) = y_1$, on ne fait pas de mise à jour de \mathbf{w} et b

D ensemble entraînement

\mathbf{x}_t	y_t
[2,0]	1
[0,3]	0
[3,0]	0
[1,1]	1

Exemple

- Simulation avec biais, $\alpha = 0.1$
- Valeur courante : $w \leftarrow [0, 0]$, $b = 0.5$
- Paire (x_2, y_2) :
 - ◆ $h(x_2) = \text{Threshold}(w \cdot x_2 + b) = \text{Threshold}(0.5) = 1$
 - ◆ puisque $h(x_2) \neq y_2$, on met à jour w et b
 - » $w \leftarrow w + \alpha (y_2 - h(x_2)) x_2 = [0, 0] + 0.1 * (0 - 1) [0, 3] = [0, -0.3]$
 - » $b \leftarrow b + \alpha (y_2 - h(x_2)) = 0.5 + 0.1 (0 - 1) = 0.4$

D ensemble entraînement

x_t	y_t
[2,0]	1
[0,3]	0
[3,0]	0
[1,1]	1

Exemple

- Simulation avec biais, $\alpha = 0.1$
- Valeur courante : $\mathbf{w} \leftarrow [0, -0.3]$, $b = 0.4$
- Paire (\mathbf{x}_3, y_3) :
 - ◆ $h(\mathbf{x}_3) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x}_3 + b) = \text{Threshold}(0.4) = 1$
 - ◆ puisque $h(\mathbf{x}_3) \neq y_3$, on met à jour \mathbf{w} et b
 - » $\mathbf{w} \leftarrow \mathbf{w} + \alpha (y_3 - h(\mathbf{x}_3)) \mathbf{x}_3 = [0, -0.3] + 0.1 * (0 - 1) [3, 0] = [-0.3, -0.3]$
 - » $b \leftarrow b + \alpha (y_3 - h(\mathbf{x}_3)) = 0.4 + 0.1 (0 - 1) = 0.3$

D ensemble entraînement

\mathbf{x}_t	y_t
[2,0]	1
[0,3]	0
[3,0]	0
[1,1]	1

Exemple

- Simulation avec biais, $\alpha = 0.1$
- Valeur courante : $\mathbf{w} \leftarrow [-0.3, -0.3]$, $b = 0.3$
- Paire (\mathbf{x}_4, y_4) :
 - ◆ $h(\mathbf{x}_4) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x}_4 + b) = \text{Threshold}(-0.3) = 0$
 - ◆ puisque $h(\mathbf{x}_4) \neq y_4$, on met à jour \mathbf{w} et b
 - » $\mathbf{w} \leftarrow \mathbf{w} + \alpha (y_4 - h(\mathbf{x}_4)) \mathbf{x}_4 = [-0.3, -0.3] + 0.1 * (1 - 0) [1, 1] = [-0.2, -0.2]$
 - » $b \leftarrow b + \alpha (y_4 - h(\mathbf{x}_4)) = 0.3 + 0.1 (1 - 0) = 0.4$

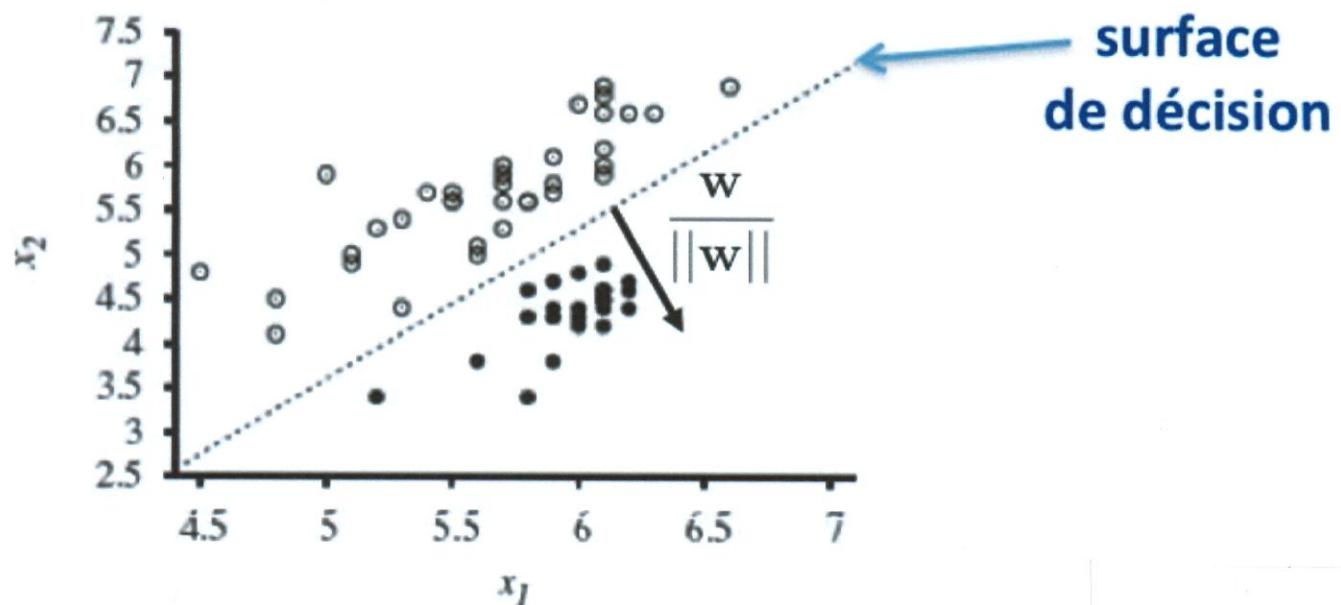
D ensemble entraînement

\mathbf{x}_t	y_t
[2,0]	1
[0,3]	0
[3,0]	0
[1,1]	1

Apprentissage automatique - propriétés du perceptron

Surface de séparation

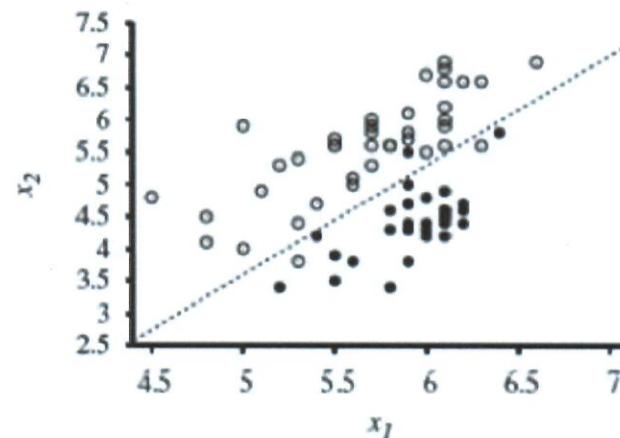
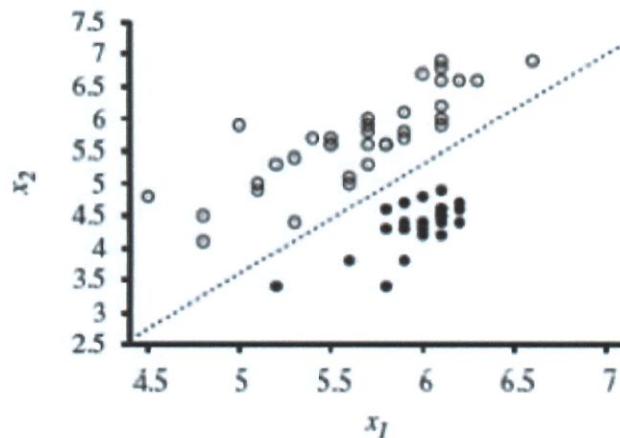
- Le perceptron cherche donc un **séparateur linéaire** entre les deux classes



- La **surface de décision** d'un classifieur est la surface qui sépare les deux régions classifiées dans les deux classes différentes

Convergence et séparabilité

- Si les exemples sont linéairement séparables (gauche), le perceptron est garanti de converger à une solution avec une erreur nulle sur l'ensemble d'entraînement, pour tout α



- Sinon, pour garantir la convergence à une solution ayant la plus petite erreur possible en entraînement, on doit décroître le taux d'apprentissage, par ex. selon $\alpha_k = \frac{\alpha}{1 + \beta k}$

$$\alpha_k = \frac{\alpha}{1 + \beta k}$$

Apprentissage vue comme la minimisation d'une perte

- Le problème de l'apprentissage peut être formulé comme un problème d'optimisation
 - ◆ pour chaque exemple d'entraînement, on souhaite minimiser une certaine distance $Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t))$ entre la cible y_t et la prédiction $h_{\mathbf{w}}(\mathbf{x}_t)$
 - ◆ on appelle cette distance une **perte**
- On peut dériver l'algorithme du perceptron de cette façon ...

Apprentissage automatique - dérivées partielles et gradients

Gradient

- On va appeler **gradient** ∇f d'une fonction f le vecteur contenant les dérivées partielles de f par rapport à toutes les variables
- Dans l'exemple avec la fonction $f(x, y)$:

$$\begin{aligned}\nabla f(x, y) &= \left[\frac{\partial f(x, y)}{\partial x}, \frac{\partial f(x, y)}{\partial y} \right] \\ &= \left[\frac{2x}{y}, \frac{-x^2}{y^2} \right]\end{aligned}$$

Apprentissage automatique - minimisation de perte

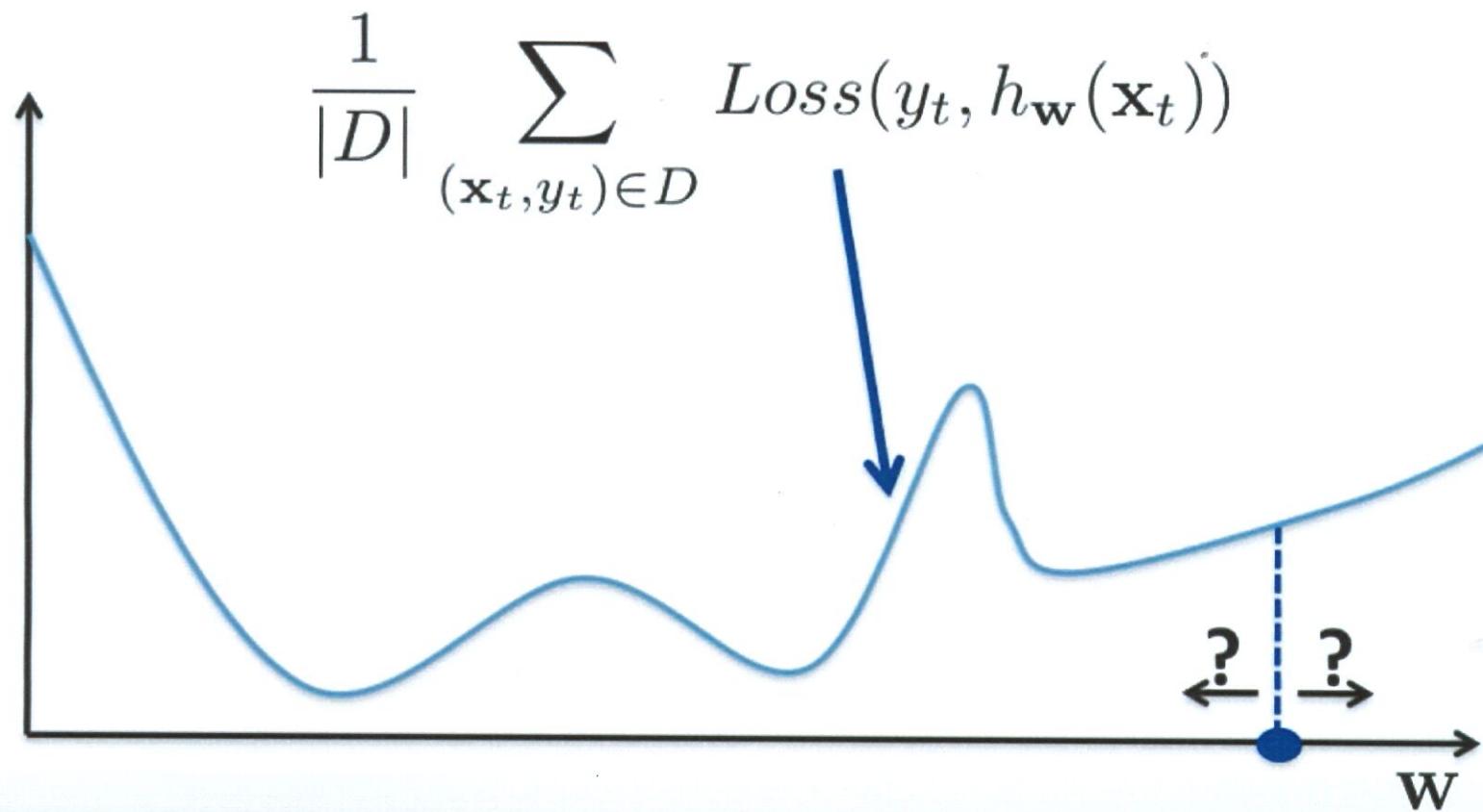
Apprentissage vue comme la minimisation d'une perte

- Le problème de l'apprentissage peut être formulé comme un problème d'optimisation
 - pour chaque exemple d'entraînement, on souhaite minimiser une certaine distance $Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t))$ entre la cible y_t et la prédiction $h_{\mathbf{w}}(\mathbf{x}_t)$
 - on appelle cette distance une **perte**
- Dans le cas du perceptron:

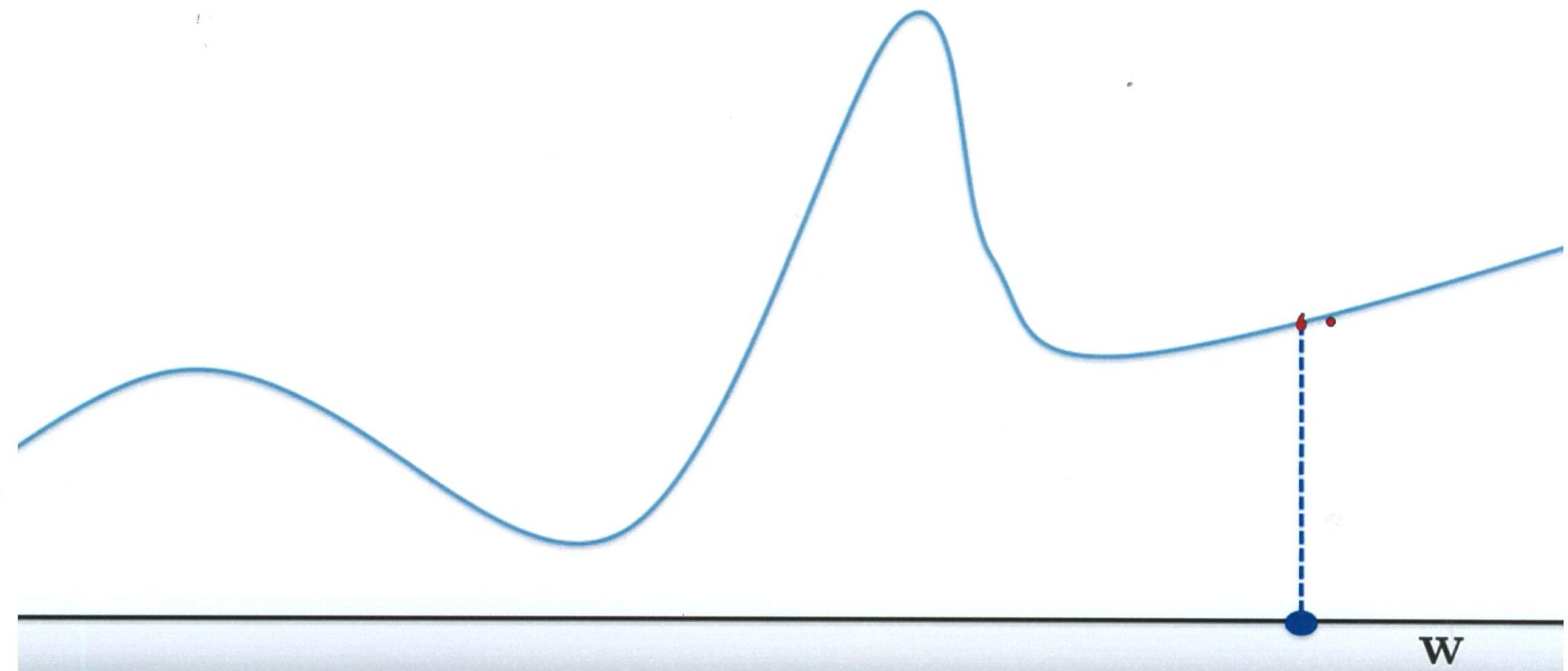
$$Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) = -(y_t - h_{\mathbf{w}}(\mathbf{x}_t)) \mathbf{w} \cdot \mathbf{x}_t$$

- si la prédiction est bonne, le coût est 0
- si la prédiction est mauvaise, la perte est la distance entre $\mathbf{w} \cdot \mathbf{x}_t$ et le seuil à franchir pour que la prédiction soit bonne

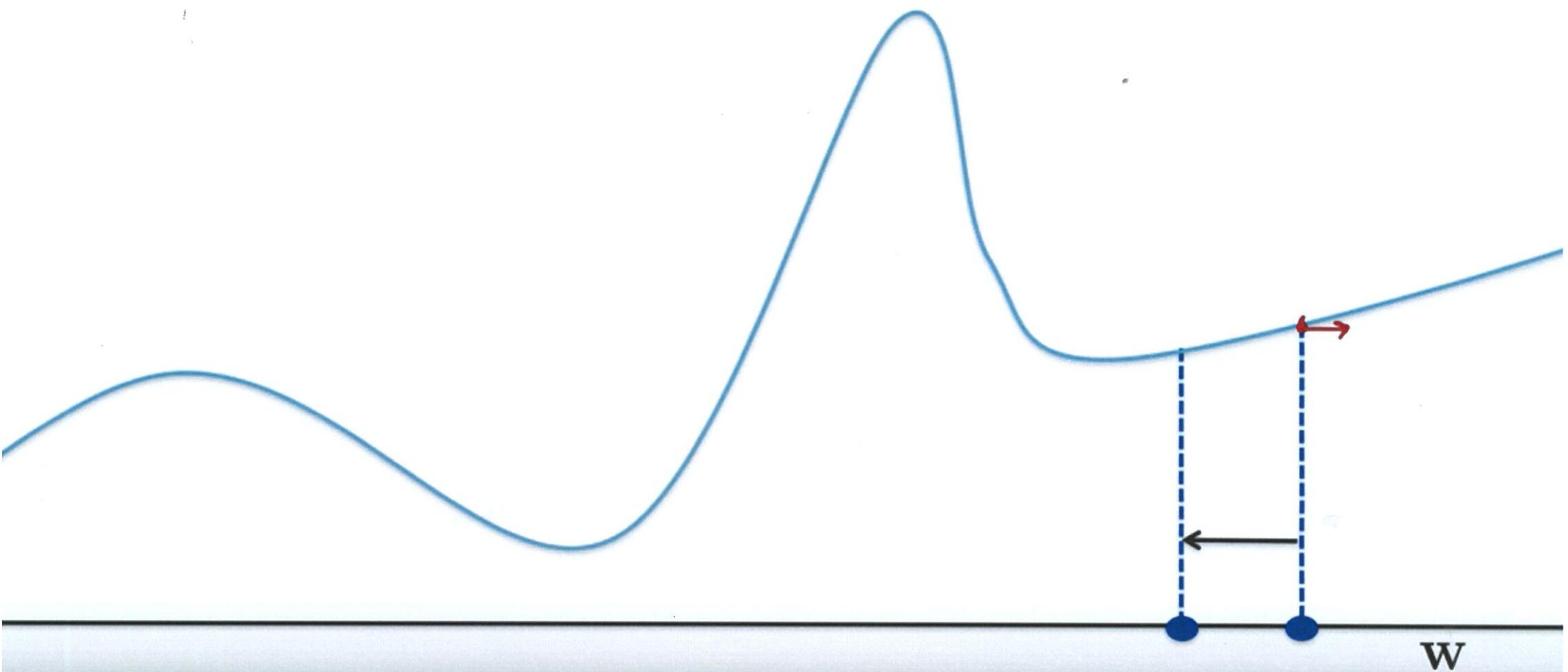
Recherche locale pour la minimisation d'une perte



Algorithme de descente de gradient

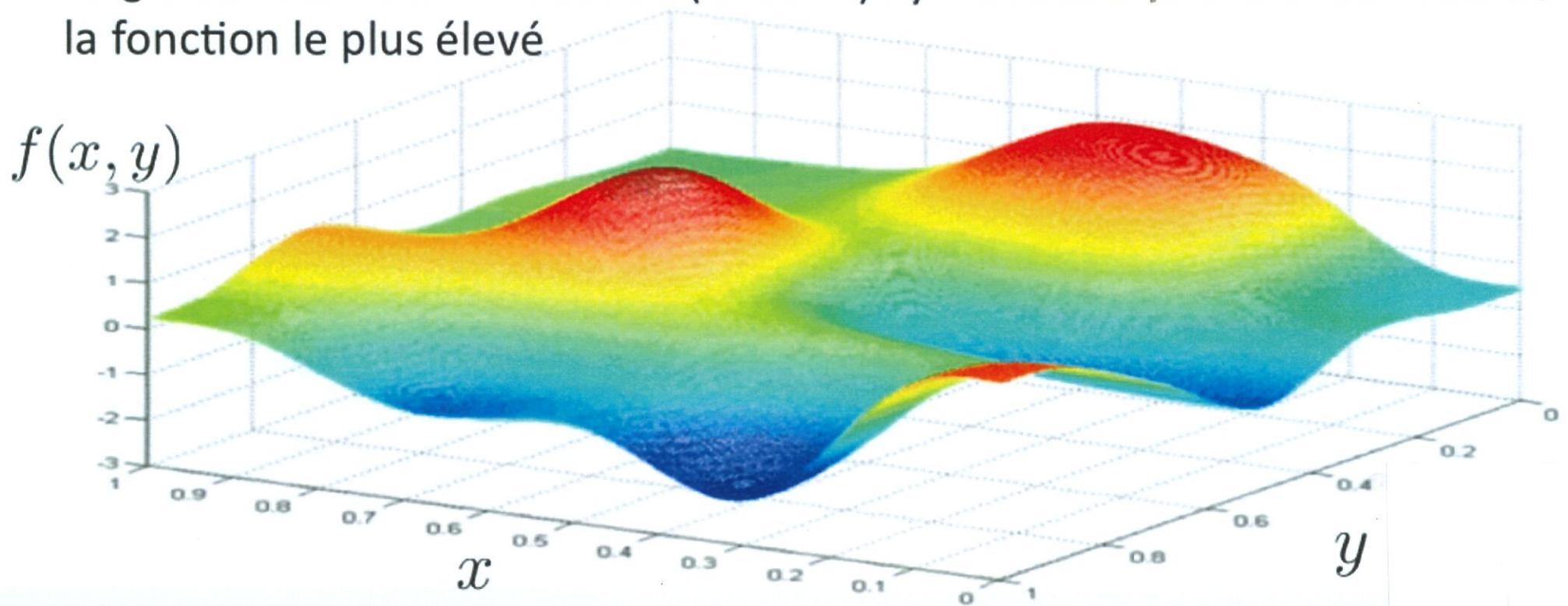


Algorithme de descente de gradient



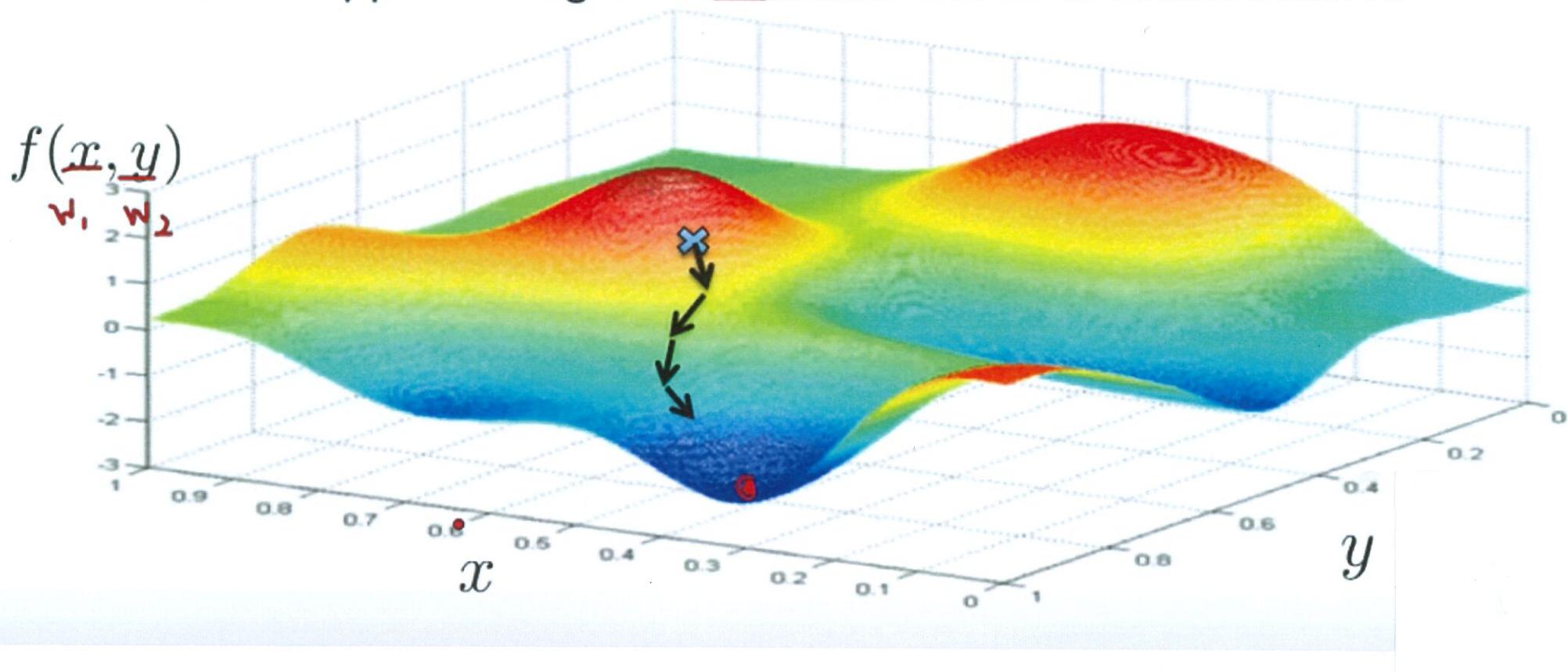
Descente de gradient

- Le gradient donne la direction (vecteur) ayant le taux d'accroissement de la fonction le plus élevé



Descente de gradient

- La direction opposée au gradient nous donne la direction à suivre



Apprentissage vue comme la minimisation d'une perte

- En apprentissage automatique, on souhaite optimiser:

$$\frac{1}{|D|} \sum_{(\mathbf{x}_t, y_t) \in D} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t))$$

- Le gradient par rapport à la perte moyenne contient les dérivées partielles:

$$\frac{1}{|D|} \sum_{(\mathbf{x}_t, y_t) \in D} \frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$$

- Devrait calculer la moyenne des dérivées sur tous les exemples d'entraînement avant de faire une mise à jour des paramètres!

Descente de gradient stochastique

- **Descente de gradient stochastique:** mettre à jour les paramètres à partir du gradient (c.-à-d. des dérivées partielles) d'un seul exemple, choisi aléatoirement:

```
- Initialiser  $\mathbf{w}$  aléatoirement  
- Pour  $T$  itérations  
    - Pour chaque exemple d'entraînement  $(\mathbf{x}_t, y_t)$   
        -  $w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$ 
```

- Cette procédure est plus efficace lorsque l'ensemble d'entraînement est grand
 - ◆ on fait $|D|$ mises à jour des paramètres après chaque parcours de l'ensemble d'entraînement, plutôt qu'une seule mise à jour avec la descente de gradient normale

Retour sur le perceptron

- On utilise le gradient (dérivée partielle) pour déterminer une direction de mise à jour des paramètres:

$$\frac{\partial}{\partial w_i} \text{Loss}(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) = \frac{\partial}{\partial w_i} - (y_t - h_{\mathbf{w}}(\mathbf{x}_t)) \mathbf{w} \cdot \mathbf{x}_t \cong -(y_t - h_{\mathbf{w}}(\mathbf{x}_t)) x_{t,i}$$

- Par définition, le gradient donne la direction (locale) d'augmentation la plus grande de la perte

◆ pour mettre à jour les paramètres, on va dans la direction opposée à ce gradient:

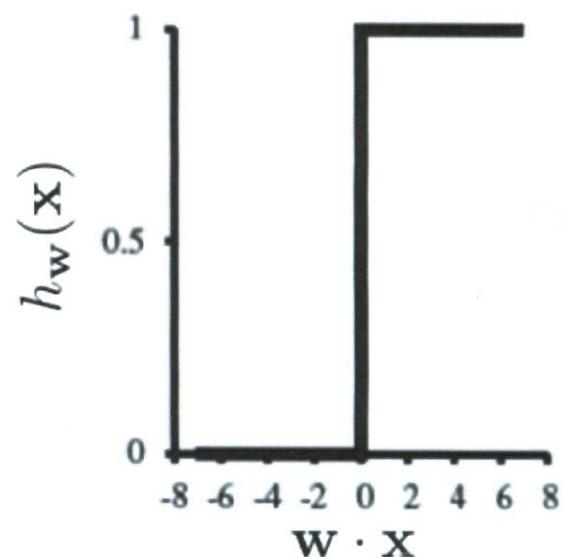
$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{Loss}(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$$

◆ on obtient à la règle d'apprentissage du perceptron

$$w_i \leftarrow w_i + \alpha (y_t - h_{\mathbf{w}}(\mathbf{x}_t)) x_{t,i} \quad \forall i$$

Apprentissage vue comme la minimisation d'une perte

- La procédure de descente de gradient stochastique est applicable à n'importe quelle perte dérivable partout
- Dans le cas du perceptron, on a un peu triché:
 - ◆ la dérivée de $h_w(x)$ n'est pas définie lorsque $w \cdot x = 0$
- L'utilisation de la fonction *Threshold* (qui est constante par partie) fait que la courbe d'entraînement peut être instable



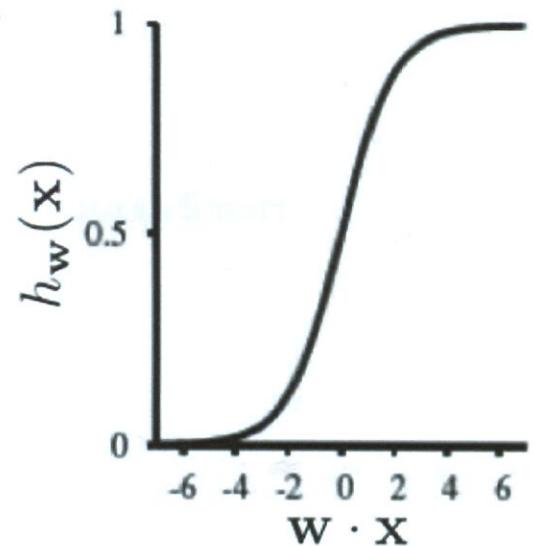
Apprentissage automatique - régression logistique

Régression logistique

- **Idée:** plutôt que de prédire une classe, prédire une probabilité d'appartenir à la classe 1

$$p(y = 1|\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}) = \text{Logistic}(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

- Choisir la classe la plus probable selon le modèle
 - ◆ si $h_{\mathbf{w}}(\mathbf{x}) \geq 0.5$ choisir la classe 1
 - ◆ sinon, choisir la classe 0



Dérivation de la règle d'apprentissage

- Pour obtenir une règle d'apprentissage, on définit d'abord une perte

$$Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) = -y_t \log h_{\mathbf{w}}(\mathbf{x}_t) - (1 - y_t) \log(1 - h_{\mathbf{w}}(\mathbf{x}_t))$$

- si $y_t = 1$, on souhaite maximiser la probabilité $p(y_t = 1 | \mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}_t)$
- si $y_t = 0$, on souhaite maximiser la probabilité $p(y_t = 0 | \mathbf{x}) = 1 - h_{\mathbf{w}}(\mathbf{x}_t)$

- On dérive la règle d'apprentissage comme une descente de gradient

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$$

ce qui donne

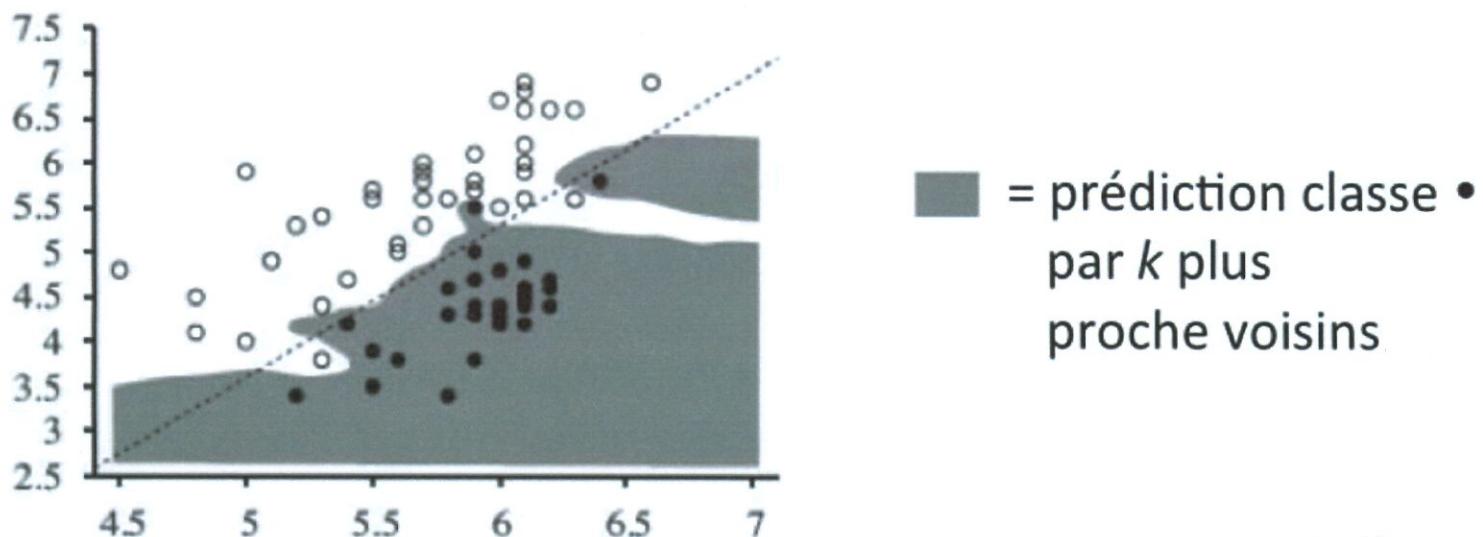
$$w_i \leftarrow w_i + \alpha(y_t - h_{\mathbf{w}}(\mathbf{x}_t))x_{t,i} \quad \forall i$$

- La règle est donc la même que pour le Perceptron, mais la définition de $h_{\mathbf{w}}(\mathbf{x}_t)$ est différente

Apprentissage automatique - réseau de neurones

Limitation des classifieurs linéaires

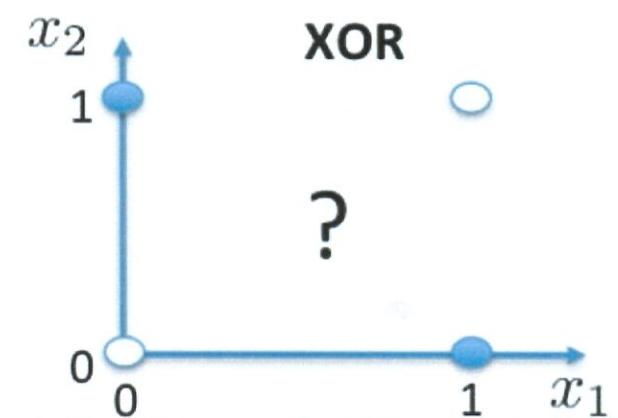
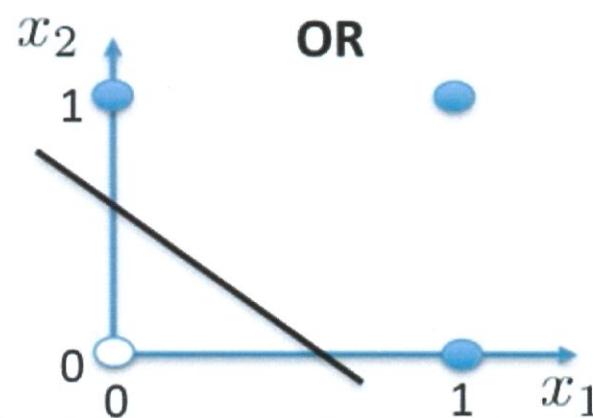
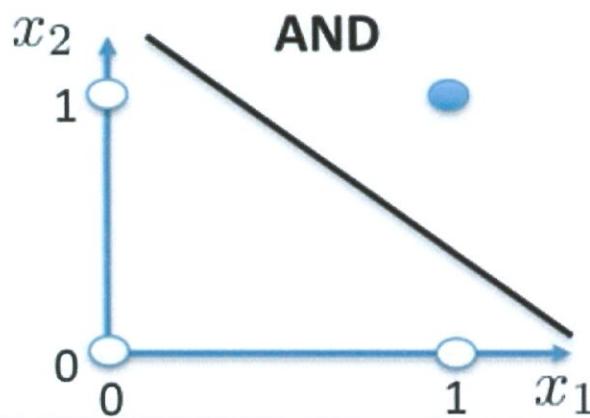
- Si les données d'entraînement sont séparables linéairement, le perceptron et la régression logistique vont trouver cette séparation



- k plus proche voisins est non-linéaire, mais coûteux en mémoire et temps de calcul (pas approprié pour des problèmes avec beaucoup de données)

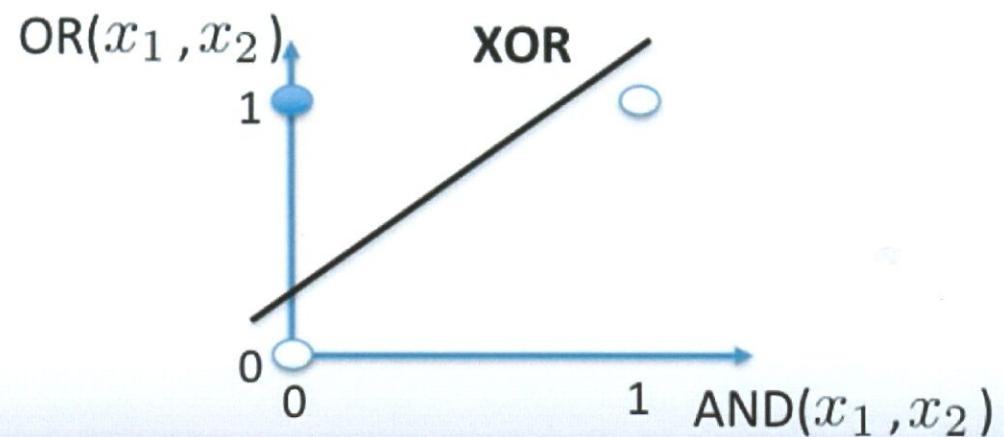
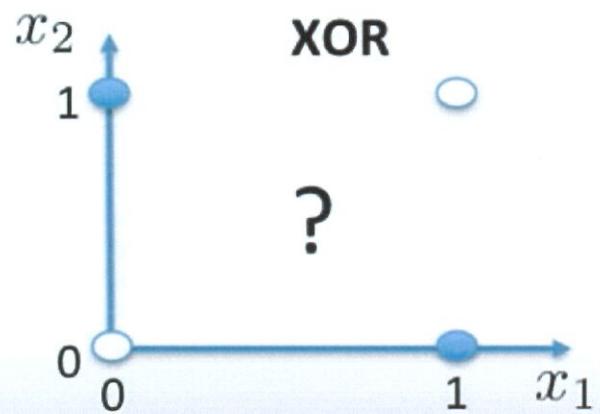
Limitation des classifieurs linéaires

- Cependant, la majorité des problèmes de classification ne sont pas linéaires
- En fait, un classifieur linéaire ne peut même pas apprendre XOR!



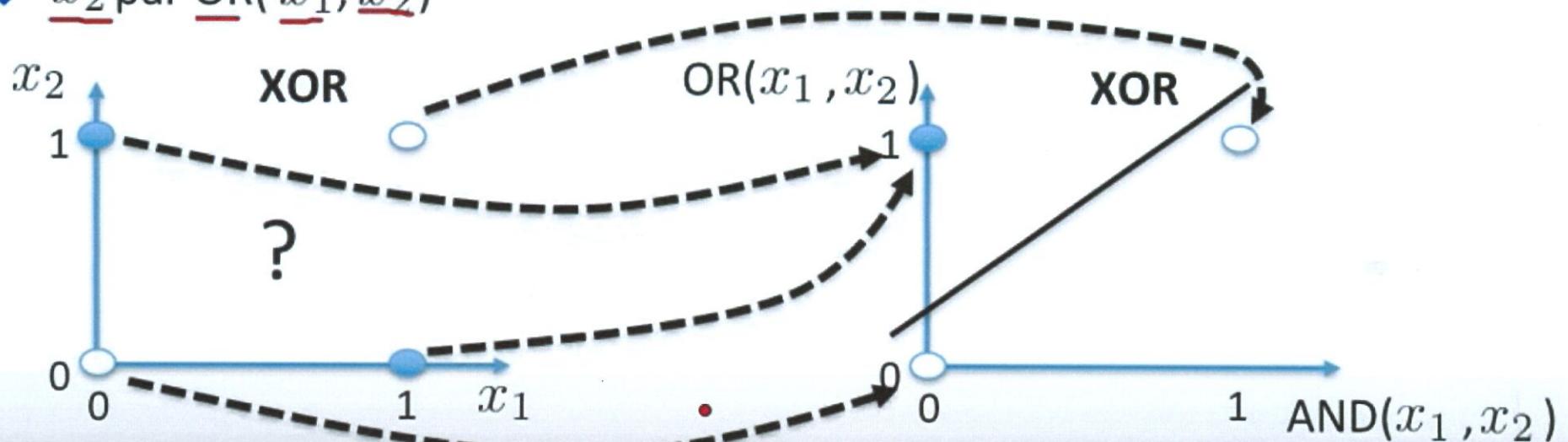
Limitation des classifieurs linéaires

- Par contre, on pourrait transformer l'entrée de façon à rendre le problème linéairement séparable sous cette nouvelle représentation
- Dans le cas de XOR, on pourrait remplacer
 - ◆ x_1 par $\text{AND}(x_1, x_2)$ et
 - ◆ x_2 par $\text{OR}(x_1, x_2)$



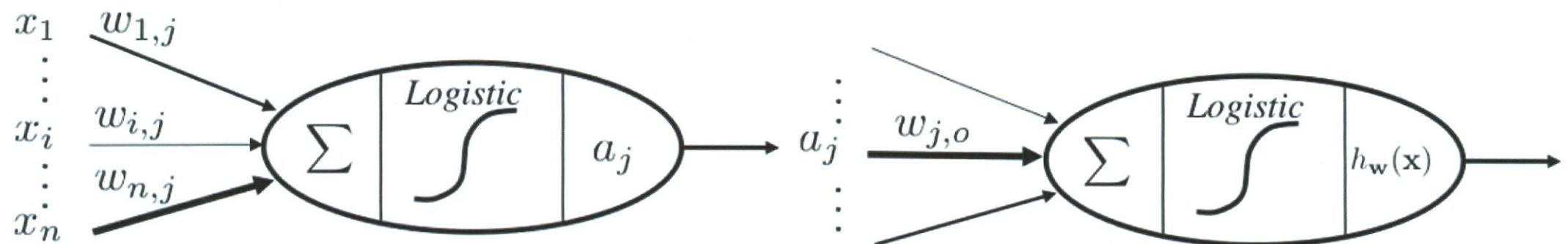
Limitation des classifieurs linéaires

- Par contre, on pourrait transformer l'entrée de façon à rendre le problème linéairement séparable sous cette nouvelle représentation
- Dans le cas de XOR, on pourrait remplacer
 - ◆ x_1 par AND(x_1, x_2) et
 - ◆ x_2 par OR(x_1, x_2)



Quatrième algorithme: réseau de neurones artificiel

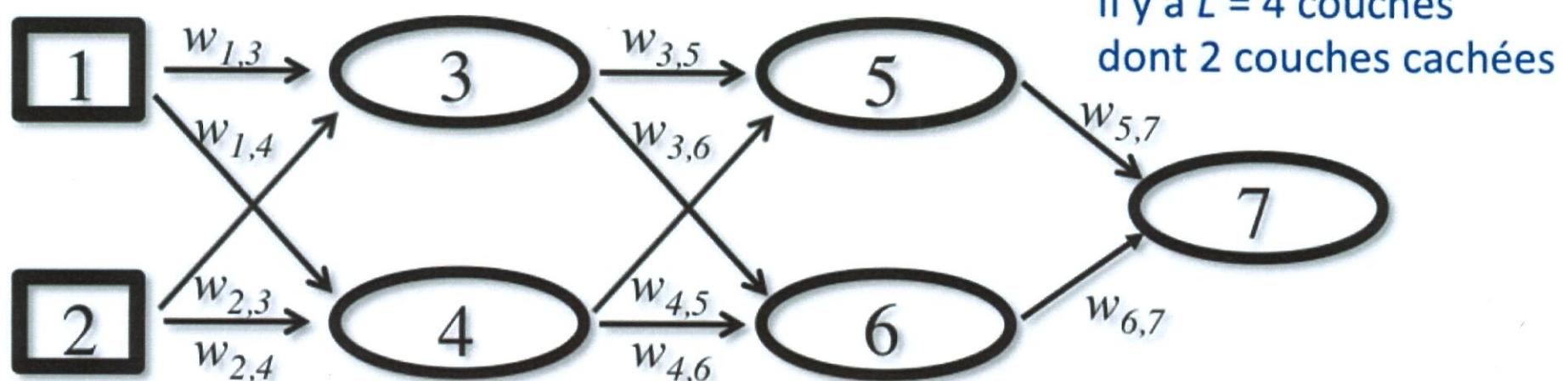
- Idée: apprendre les poids du classifieur linéaire **et** une transformation qui va rendre le problème linéairement séparable



Réseau de neurones à une seule couche cachée

Cas général à L couches

- Rien n'empêche d'avoir plus d'une couche cachée



- On note a_j l'activité du j^e « neurone », incluant les neurones d'entrée et de sortie.
Donc on aura $a_i = x_i$
- On note in_j l'activité du j^e neurone avant la non-linéarité logistique, c'est à dire

$$a_j = \text{Logistic}(in_j) = \text{Logistic}(\sum_i w_{i,j} a_i)$$

Dérivation de la règle d'apprentissage

- La dérivation de la règle d'apprentissage se fait encore avec les gradients

$$\underline{w_{i,j}} \leftarrow w_{i,j} - \alpha \frac{\partial}{\partial w_{i,j}} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i, j$$

- Calculer ces dérivées partielles peut paraître ardu
- Le calcul est grandement facilité en utilisant **la règle de dérivation en chaîne**

Dérivation en chaîne

- Si on peut écrire une fonction $f(x)$ à partir d'un résultat intermédiaire $g(x)$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

- Si on peut écrire une fonction $f(x)$ à partir de résultats intermédiaires $g_i(x)$, alors on peut écrire la dérivée partielle

$$\frac{\partial f(x)}{\partial x} = \sum_i \frac{\partial f(x)}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x}$$

Dérivation de la règle d'apprentissage

- La dérivation de la règle d'apprentissage se fait encore avec les gradients

$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial}{\partial w_{i,j}} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i, j$$

- Par l'application de la dérivée en chaîne, on a:

$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial}{\partial in_j} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \frac{\partial}{\partial w_{i,j}} in_j$$

- Par contre, un calcul naïf de toutes ces dérivées serait très inefficace
 - ♦ on utilise la procédure de **rétropropagation des gradients (ou erreurs)**

Dérivation de la règle d'apprentissage

- La dérivation de la règle d'apprentissage se fait encore avec les gradients

$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial}{\partial w_{i,j}} \text{Loss}(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i, j$$

- Par l'application de la dérivée en chaîne, on a:

$$w_{i,j} \leftarrow w_{i,j} - \alpha \underbrace{\frac{\partial}{\partial \text{in}_j} \text{Loss}(y_t, h_{\mathbf{w}}(\mathbf{x}_t))}_{\begin{array}{l} \text{gradient de la perte p/r} \\ \text{à la somme des entrées du neurone} \end{array}} \underbrace{\frac{\partial}{\partial w_{i,j}} \text{in}_j}_{\begin{array}{l} \text{gradient de la} \\ \text{somme p/r} \\ \text{au poids } w_{i,j} \end{array}}$$

- Par contre, un calcul naïf de toutes ces dérivées serait très inefficace
 - on utilise la procédure de **rétropropagation des gradients (ou erreurs)**

Rétropropagation des gradients

- Utiliser le fait que la dérivée pour un neurone à la couche l peut être calculée à partir de la dérivée des neurones connectés à la couche $l+1$

$$\begin{aligned}\frac{\partial \text{Loss}}{\partial \text{in}_j} &= \frac{\partial \text{Loss}}{\partial a_j} \frac{\partial a_j}{\partial \text{in}_j} \\ &= \left(\sum_k \frac{\partial \text{Loss}}{\partial \text{in}_k} \frac{\partial \text{in}_k}{\partial a_j} \right) \frac{\partial g(\text{in}_j)}{\partial \text{in}_j} \\ &= \left(\sum_k \frac{\partial \text{Loss}}{\partial \text{in}_k} w_{j,k} \right) g(\text{in}_j)(1 - g(\text{in}_j))\end{aligned}$$

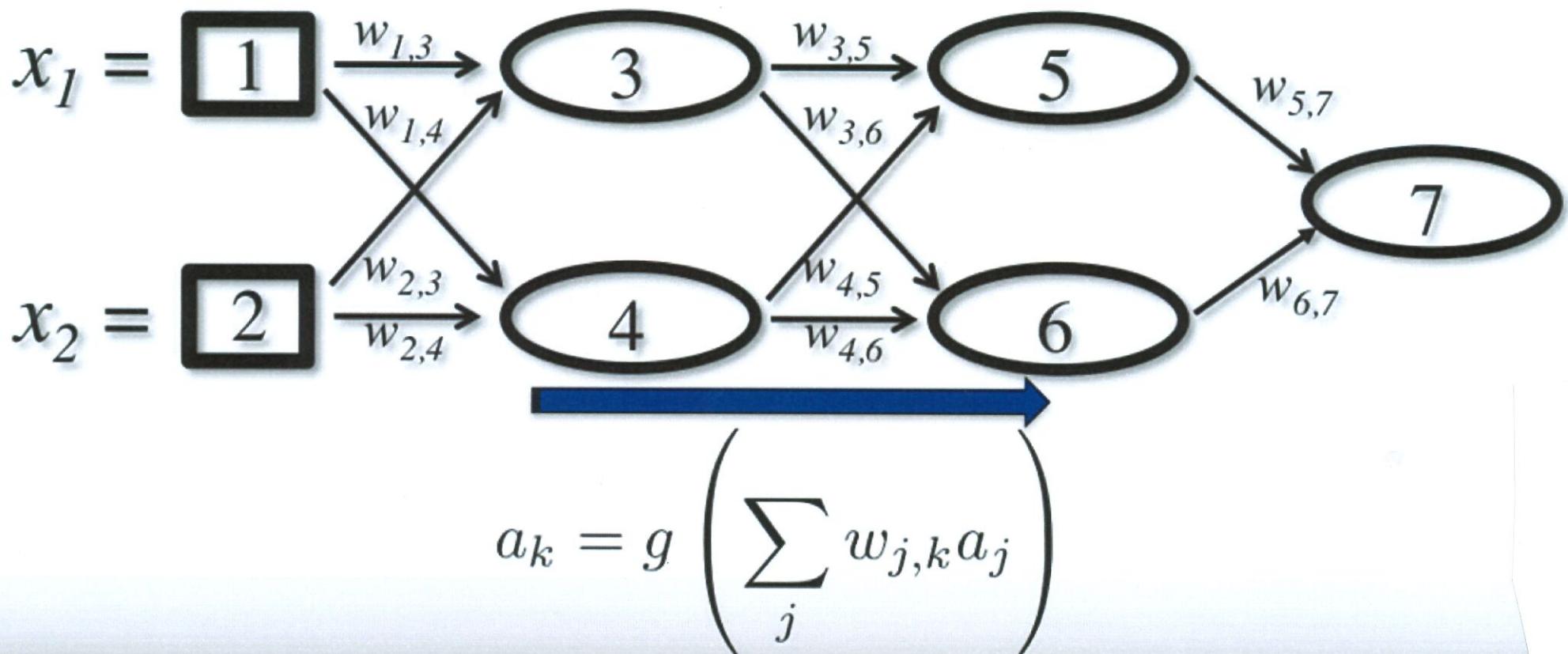
k itère sur les neurones cachés de la couche $l+1$

où $\text{in}_k = \sum_i w_{i,k} a_i$ et

$\boxed{\text{Logistic}(\cdot) \equiv g(\cdot)}$
(pour simplifier notation)

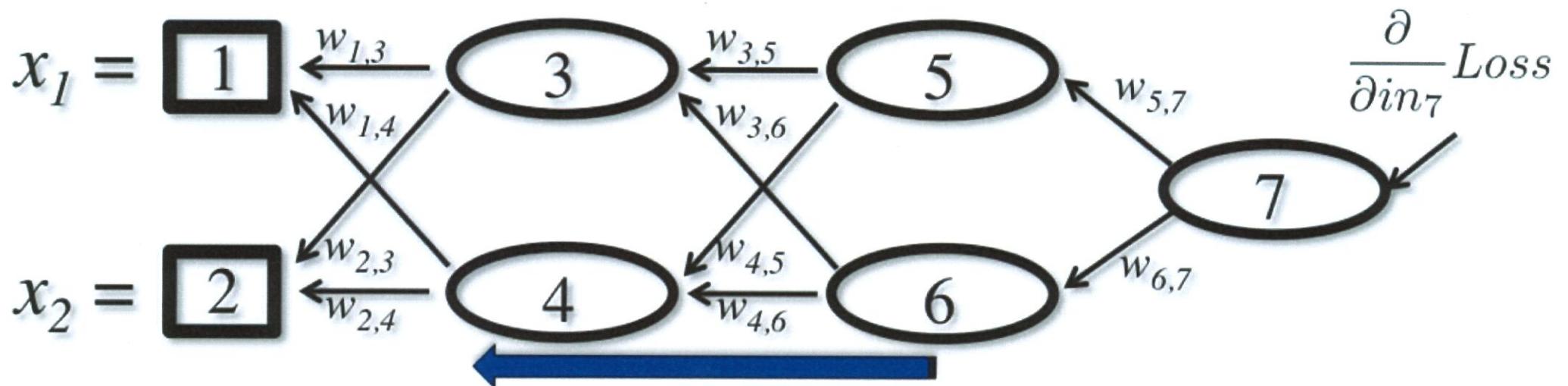
Visualisation de la rétropropagation

- L'algorithme d'apprentissage commence par une **propagation avant**



Visualisation de la rétropropagation

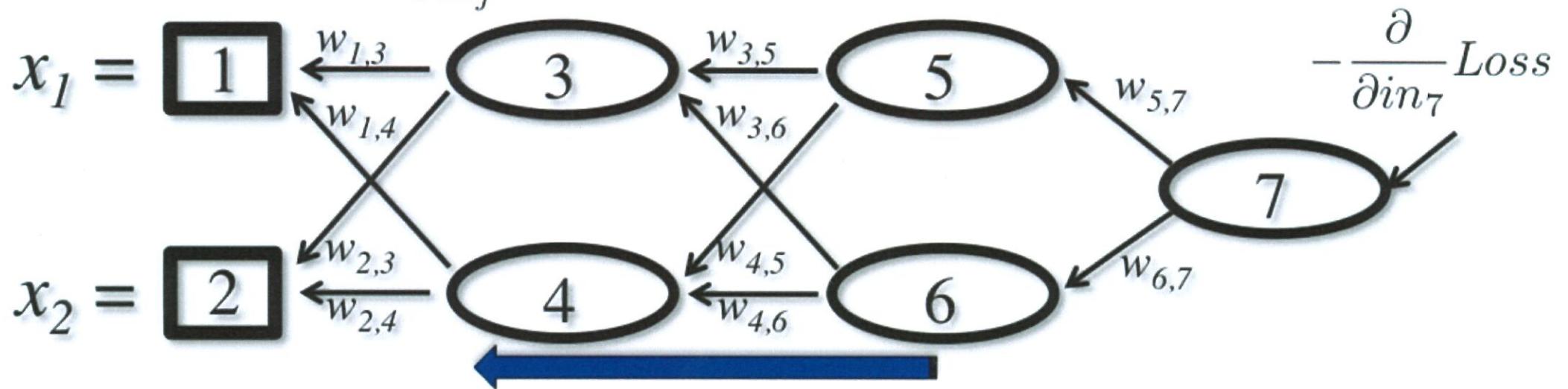
- Ensuite, le gradient sur la sortie est calculé, et le gradient rétropropagé



$$\frac{\partial}{\partial \text{in}_j} \text{Loss} = g(\text{in}_j)(1 - g(\text{in}_j)) \sum_k w_{j,k} \frac{\partial}{\partial \text{in}_k} \text{Loss}$$

Visualisation de la rétropropagation

- Peut propager $-\frac{\partial}{\partial in_j} Loss = \Delta[j]$ aussi (décomposition équivalente du livre)



$$\Delta[j] = g(in_j)(1 - g(in_j)) \sum_k w_{j,k} \Delta[k]$$

Retour sur la règle d'apprentissage

- La dérivation de la règle d'apprentissage se fait encore avec les gradients

$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial}{\partial in_j} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \frac{\partial}{\partial w_{i,j}} in_j$$


- Donc la règle de mise à jour peut être écrite comme suite:

$$w_{i,j} \leftarrow w_{i,j} + \alpha a_i \Delta[j]$$

```

function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector  $\mathbf{x}$  and output vector  $\mathbf{y}$ 
  network, a multilayer network with  $L$  layers, weights  $w_{i,j}$ , activation function  $g$ 
  local variables:  $\Delta$ , a vector of errors, indexed by network node

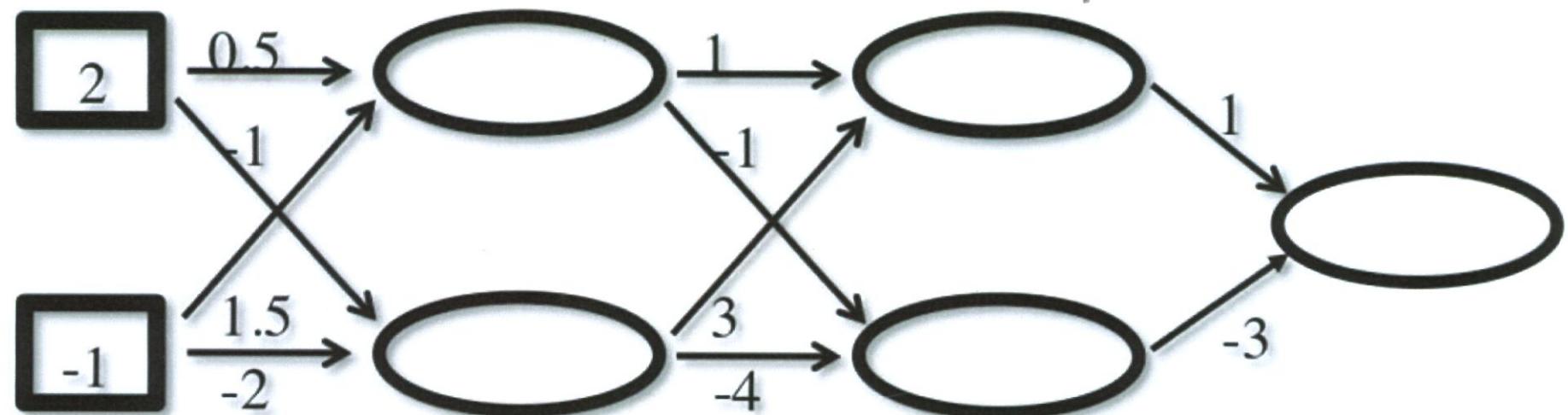
  for each weight  $w_{i,j}$  in network do
     $w_{i,j} \leftarrow$  a small random number
  repeat
    for each example  $(\mathbf{x}, \mathbf{y})$  in examples do
      /* Propagate the inputs forward to compute the outputs */
      for each node  $i$  in the input layer do
         $a_i \leftarrow x_i$ 
      for  $\ell = 2$  to  $L$  do
        for each node  $j$  in layer  $\ell$  do
           $in_j \leftarrow \sum_i w_{i,j} a_i$ 
           $a_j \leftarrow g(in_j)$ 
      /* Propagate deltas backward from output layer to input layer */
      for each node  $j$  in the output layer do
         $\Delta[j] \leftarrow y_j - a_j$  ( $= -\partial Loss / \partial in_j$ )
      for  $\ell = L - 1$  to 1 do
        for each node  $i$  in layer  $\ell$  do
           $\Delta[i] \leftarrow g(in_i)(1 - g(in_i)) \sum_j w_{i,j} \Delta[j]$ 
      /* Update every weight in network using deltas */
      for each weight  $w_{i,j}$  in network do
         $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ 
    until some stopping criterion is satisfied
  return network

```

$Logistic(\cdot) \equiv g(\cdot)$
 (pour simplifier notation)

Exemple

- Exemple: $x = [2, -1]$, $y = 1$

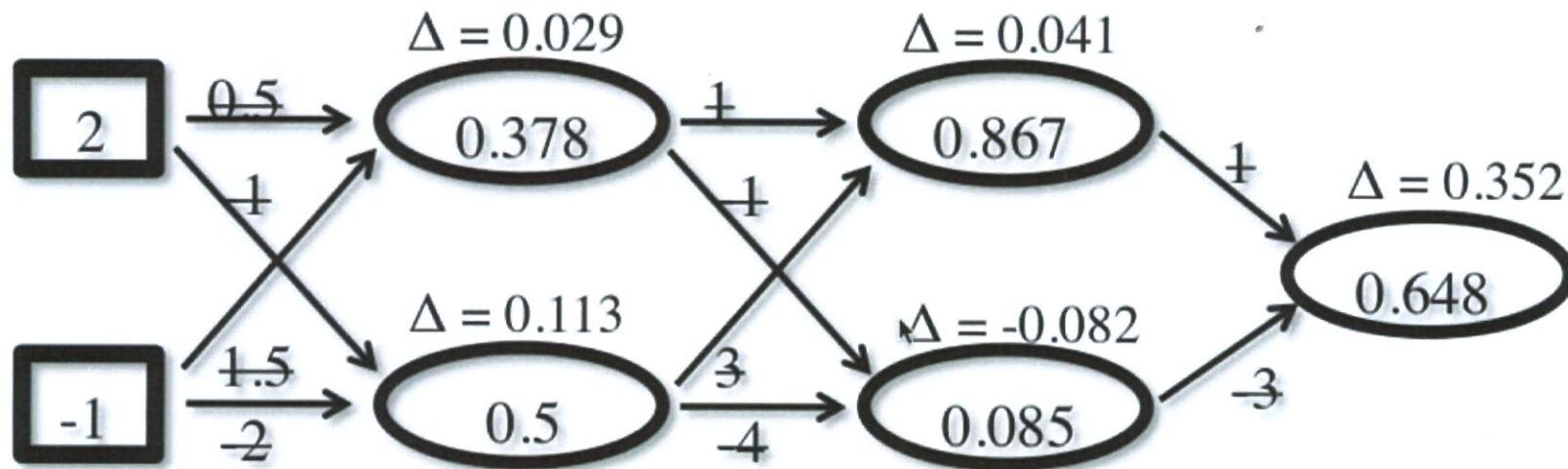


propagation avant

$$a_k = g \left(\sum_j w_{j,k} a_j \right)$$

Exemple

- Exemple: $x = [2, -1]$, $y = 1$



mise à jour ($\alpha=0.1$)

$$w_{1,3} \leftarrow 0.5 + 0.1 * 2 * 0.029 = 0.506$$

$$w_{1,4} \leftarrow -1 + 0.1 * 2 * 0.113 = -0.977$$

$$w_{2,3} \leftarrow 1.5 + 0.1 * -1 * 0.029 = 1.497$$

$$w_{2,4} \leftarrow -2 + 0.1 * -1 * 0.113 = -2.011$$

$$w_{3,5} \leftarrow 1 + 0.1 * 0.378 * 0.041 = 1.002$$

$$w_{3,6} \leftarrow -1 + 0.1 * 0.378 * -0.082 = -1.003$$

$$w_{4,5} \leftarrow 3 + 0.1 * 0.5 * 0.041 = 3.002$$

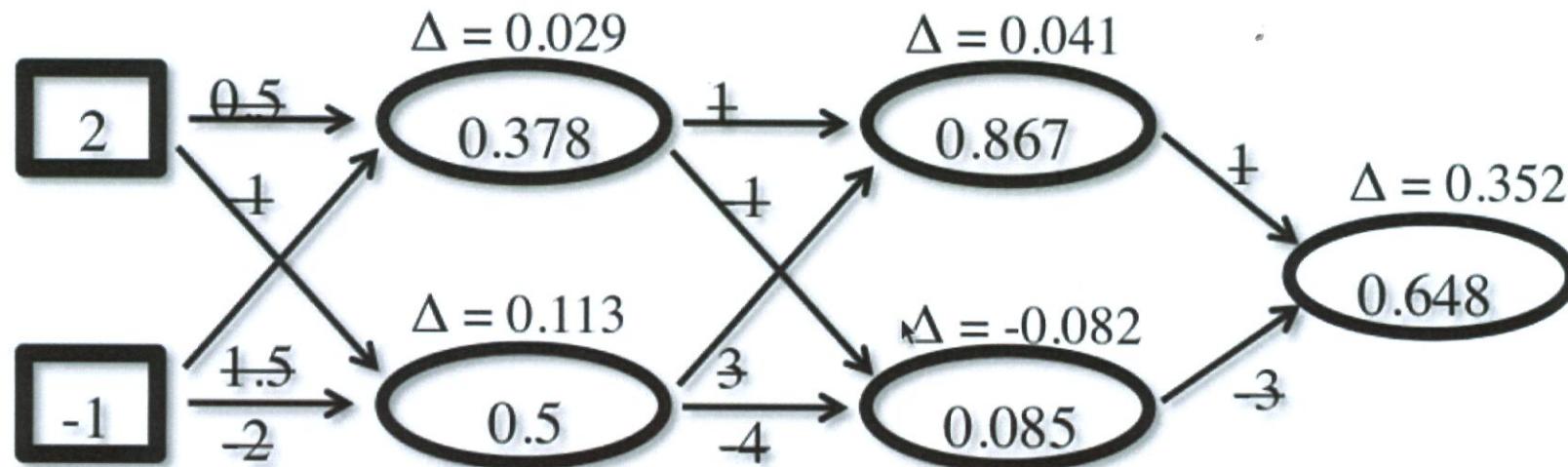
$$w_{4,6} \leftarrow -4 + 0.1 * 0.5 * -0.082 = -4.004$$

$$w_{5,7} \leftarrow 1 + 0.1 * 0.867 * 0.352 = 1.031$$

$$w_{6,7} \leftarrow -3 + 0.1 * 0.085 * 0.352 = -2.997$$

Exemple

- Exemple: $x = [2, -1]$, $y = 1$



mise à jour ($\alpha=0.1$)

$$w_{1,3} \leftarrow 0.5 + 0.1 * 2 * 0.029 = 0.506$$

$$w_{1,4} \leftarrow -1 + 0.1 * 2 * 0.113 = -0.977$$

$$w_{2,3} \leftarrow 1.5 + 0.1 * -1 * 0.029 = 1.497$$

$$w_{2,4} \leftarrow -2 + 0.1 * -1 * 0.113 = -2.011$$

$$w_{3,5} \leftarrow 1 + 0.1 * 0.378 * 0.041 = 1.002$$

$$w_{3,6} \leftarrow -1 + 0.1 * 0.378 * -0.082 = -1.003$$

$$w_{4,5} \leftarrow 3 + 0.1 * 0.5 * 0.041 = 3.002$$

$$w_{4,6} \leftarrow -4 + 0.1 * 0.5 * -0.082 = -4.004$$

$$w_{5,7} \leftarrow 1 + 0.1 * 0.867 * 0.352 = 1.031$$

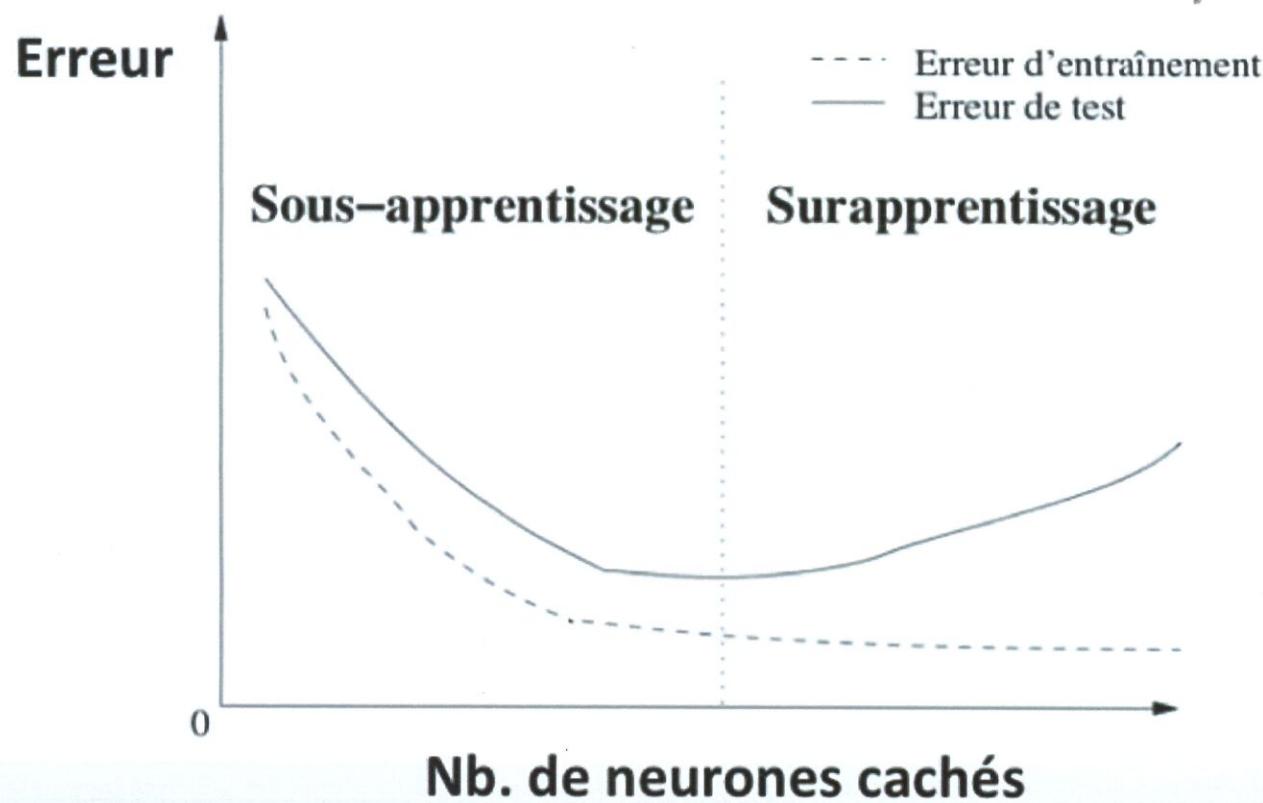
$$w_{6,7} \leftarrow -3 + 0.1 * 0.085 * 0.352 = -2.997$$

Mesure de la performance d'un algorithme d'apprentissage

- Comment évaluer le succès d'un algorithme?
 - ◆ on pourrait regarder l'erreur moyenne commise sur les exemples d'entraînement, mais cette erreur sera nécessairement optimiste
 - » h a déjà vu la bonne réponse pour ces exemples!
 - » on mesure donc seulement la capacité de l'algorithme à **mémoriser**
 - » dans le cas 1 plus proche voisin, l'erreur sera de 0!
- Ce qui nous intéresse vraiment, c'est la capacité de l'algorithme à **généraliser** sur de **nouveaux exemples**
 - ◆ reflète mieux le contexte dans lequel on va utiliser h
- Pour mesurer la généralisation, on met de côté des exemples étiquetés, qui seront utilisés seulement à la toute fin, pour calculer l'erreur
 - ◆ on l'appelle l'**ensemble de test**

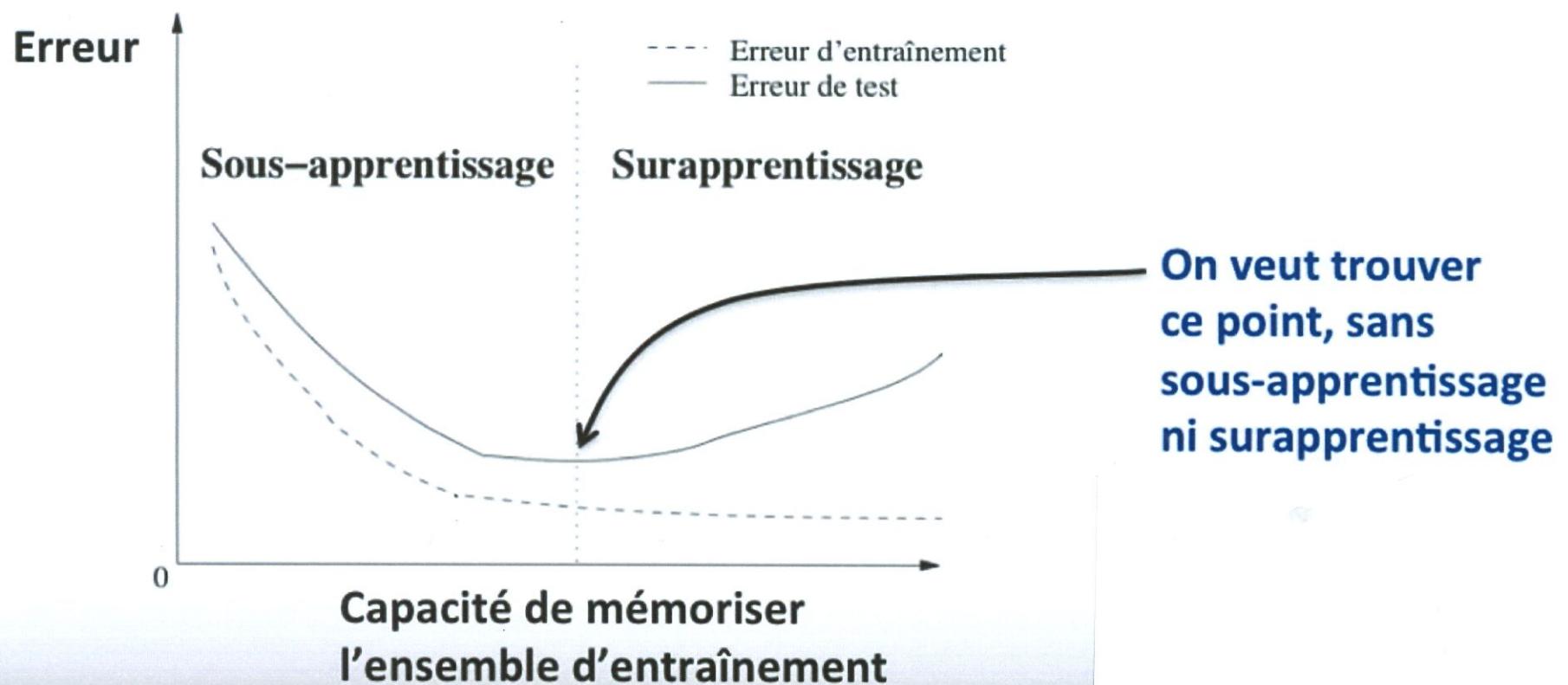
Retour sur la notion de généralisation

- Quelle est la relation entre l'erreur d'entraînement et de test ?



Retour sur la notion de généralisation

- Quelle est la relation entre l'erreur d'entraînement et de test ?



Hyper-paramètres

- Les algorithmes d'apprentissage ont normalement des « options » à déterminer
 - ◆ k plus proche voisins: la valeur de « k »
 - ◆ Perceptron et régression logistique: le taux d'apprentissage α , nb. itérations N
 - ◆ réseau de neurones: taux d'apprentissage, nb. d'itérations, nombre de neurones cachés, fonction d'activation $g(\cdot)$
- On appelle ces « options » des **hyper-paramètres**
 - ◆ ne pas choisir en fonction de l'erreur d'entraînement (mène à du surapprentissage)
 - » pour le k plus proche voisin, l'optimal sera toujours $k=1$
 - ◆ on ne peut pas utiliser l'ensemble de test non plus, **ça serait tricher!**
 - ◆ on garde plutôt un autre ensemble de côté, l'**ensemble de validation**
- Sélectionner les valeurs d'hyper-paramètres est une forme d'apprentissage

Procédure d'évaluation complète

- Utilisation typique d'un algorithme d'apprentissage
 - ◆ séparer nos données en 3 ensembles:
entraînement (70%), validation (15%) et test (15%)
 - ◆ faire une liste de valeurs des hyper-paramètres à essayer
 - ◆ pour chaque élément de cette liste, lancer l'algorithme d'apprentissage sur l'ensemble d'entraînement et mesurer la performance sur l'ensemble de validation
 - ◆ réutiliser la valeur des hyper-paramètres avec la meilleure performance en validation, pour calculer la performance sur l'ensemble de test
- La performance sur l'ensemble de test est alors une **estimation non-biaisée** (non-optimiste) de la performance de généralisation de l'algorithme
- On peut utiliser la performance pour comparer des algorithmes d'apprentissage différents