IFT 615 – Intelligence artificielle

Apprentissage automatique

Hugo Larochelle
Département d'informatique
Université de Sherbrooke
http://www.dmi.usherb.ca/~larocheh/cours/ift615.html

Sujets couverts

- 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**. »
- 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)

Apprentissage dans un agent

- Dans un agent, on distingue 6 composantes:
  1. une correspondance entre les conditions de l'état courant et les actions (plan ou politique)
  2. un moyen de déduire, à partir des perceptions reçues, des propriétés pertinentes du monde
  3. des informations sur la façon dont le monde évolue et sur les résultats des actions de l'agent
  4. des informations sur l'utilité / la désirabilité des états
  5. des informations sur la valeur / la désirabilité des actions
  6. des buts, identifiant les classes d'états dont l'atteinte maximise l'utilité de l'agent
- Chacune de ces composantes peut être « apprise » à l'aide d'un algorithme d'apprentissage approprié

Types de problèmes d'apprentissage

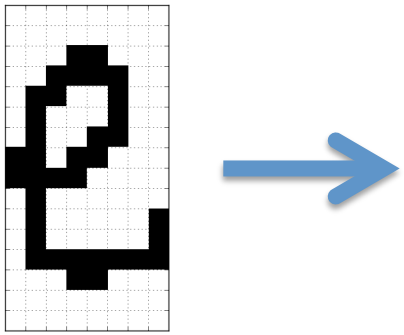
- Il existe plusieurs sortes de problèmes d'apprentissage, qui se distinguent par la nature de la supervision offerte par nos données
  - **apprentissage supervisé** : sortie désirée (cible ou « target ») est fournie explicitement par les données (sujet de ce cours)
    - » ex.: reconnaissance de caractères, à l'aide d'un ensemble de paires (images, identité du caractère)
  - **apprentissage par renforcement** : le signal d'apprentissage correspond seulement à des récompenses et punitions (vu plus tard dans le cours)
    - » ex.: est-ce que le modèle a gagné sa partie d'échec (1) ou pas (-1)
  - **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 »)
    - » voir **IFT 603 - Techniques d'apprentissage, IFT 501 - Forage de données**
  - et plusieurs autres!

Types de problèmes d'apprentissage

- Dans ce cours, on va se concentrer sur l'apprentissage supervisé
- Plus spécifiquement, on va s'intéresser au problème de la classification
- Voir le cours **IFT 603 - Techniques d'apprentissage** pour en savoir plus sur le grand monde de l'apprentissage automatique

Représentation des données

- L'**entrée** $\mathbf{x}$ 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.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,
        1.,  1.,  1.,  1.,  0.,  0.,  0.,  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.,  0.,  1.,
        1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  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.])
```

- 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

Exemple: classifieur *k* plus proches voisins

- Possiblement l'algorithme d'apprentissage de 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 (est-ce que deux entrées jugées similaires sont 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’?

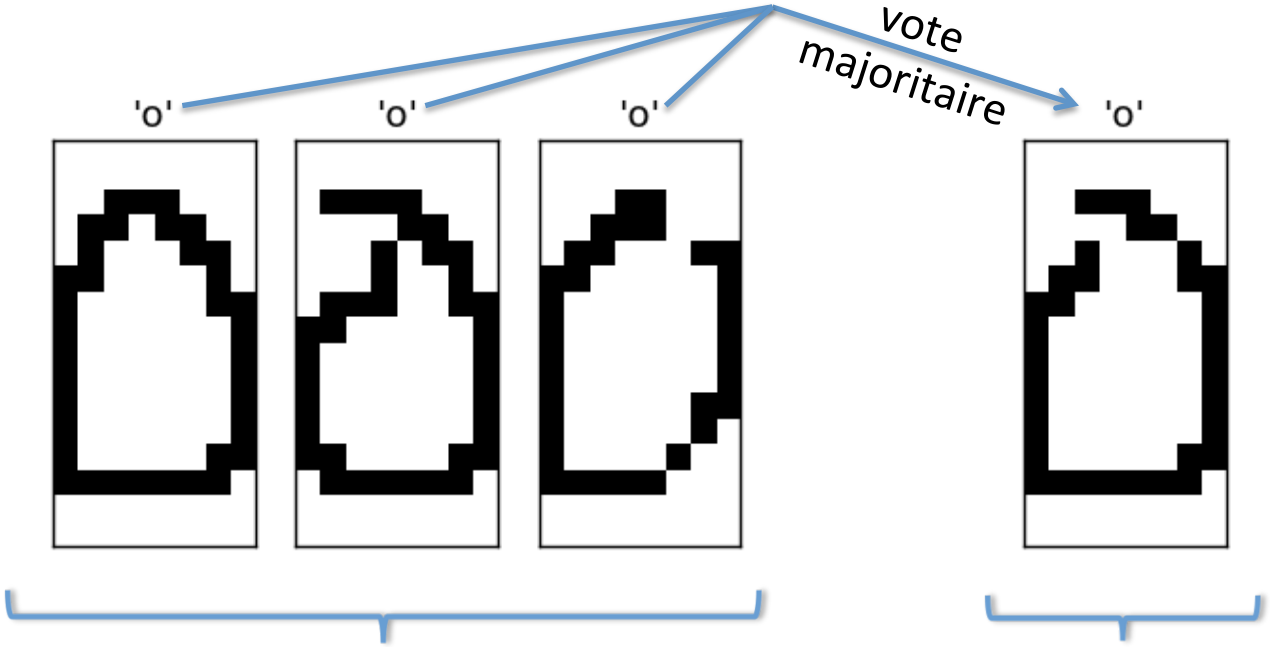
Ensemble d’entraînement

(100 exemples d’apprentissage par classe)

Classe ‘e’ Classe ‘o’

Illustration: 3 plus proches voisins

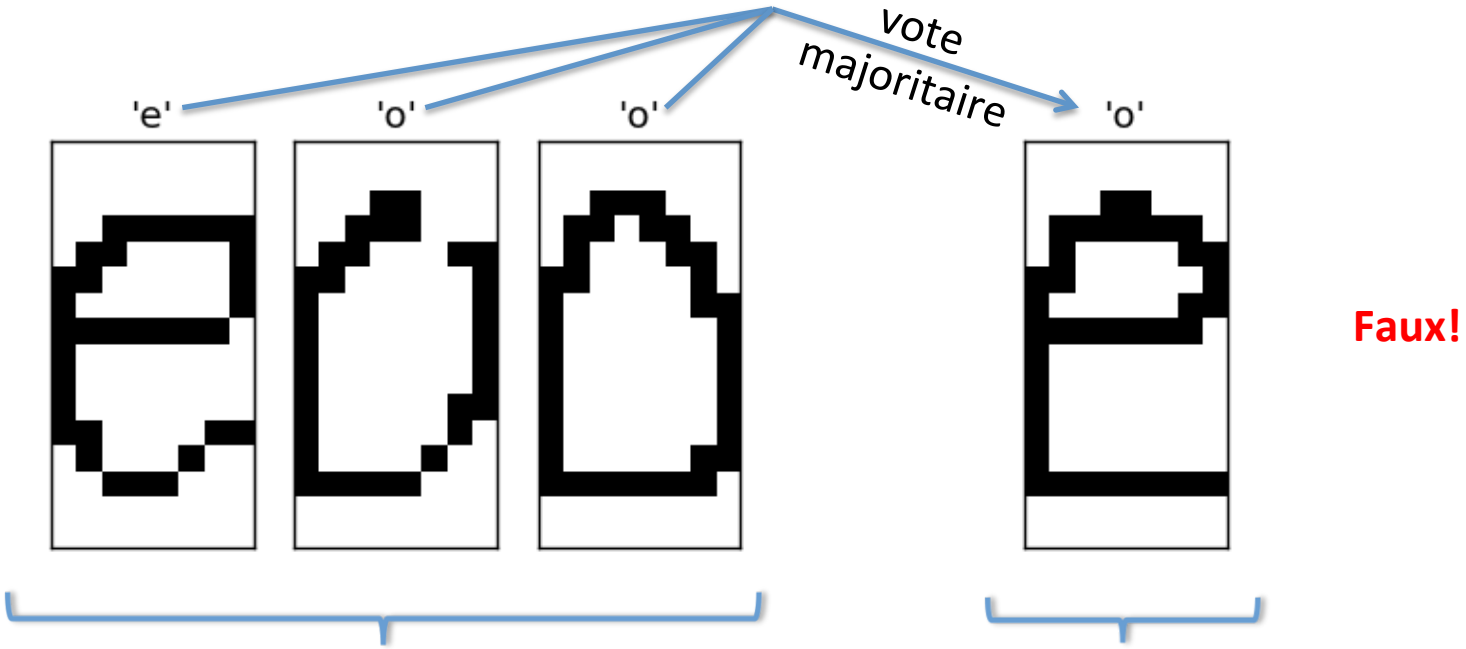
- Reconnaissance de caractère: est-ce un ‘e’ ou un ‘o’?



Vrai!

Illustration: 3 plus proches voisins

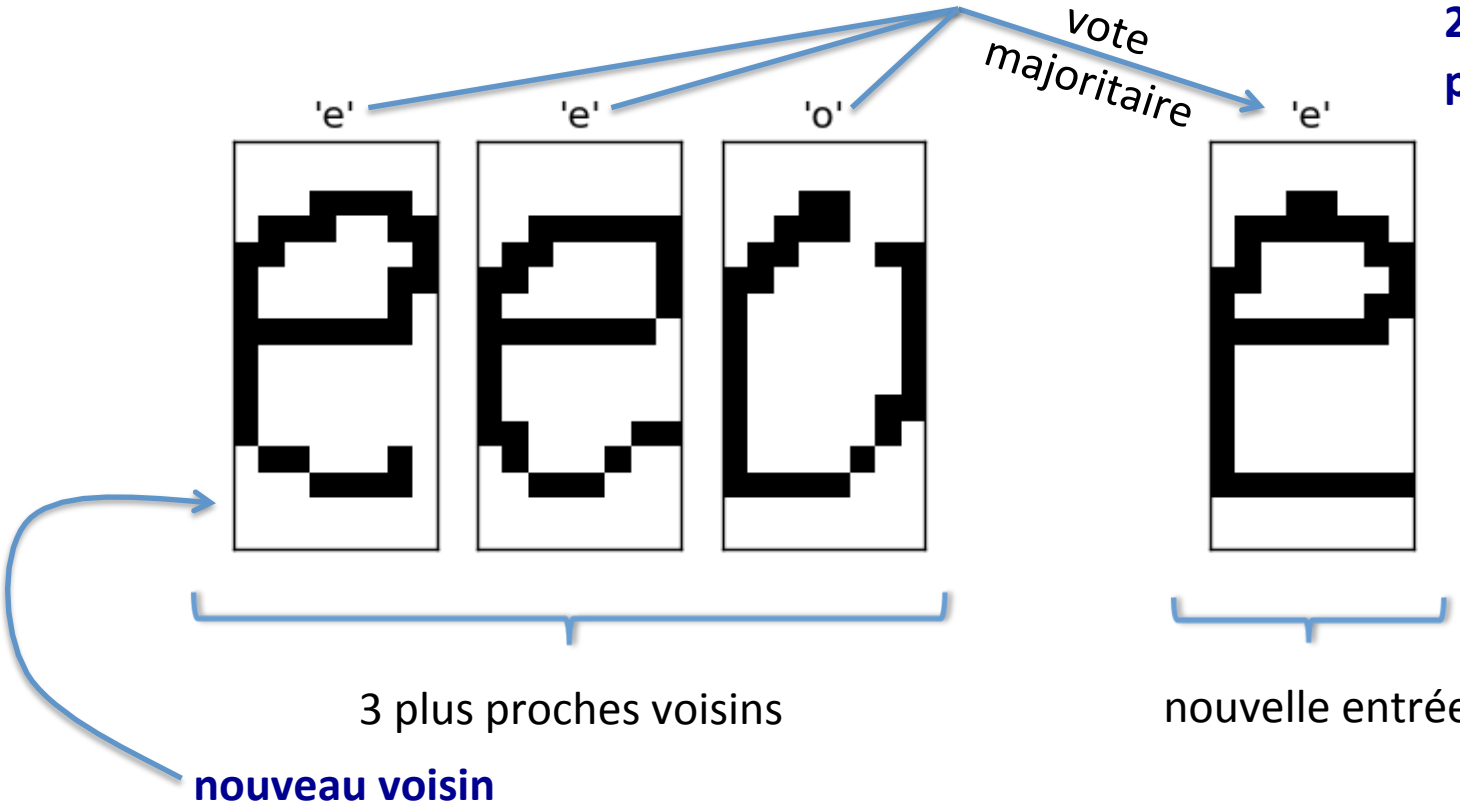
- Reconnaissance de caractère: est-ce un ‘e’ ou un ‘o’?



Faux!

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:

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

  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 changer qu'en étant encore plus proches (plus similaires) à l'entrée

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**
  - ça reflète mieux le contexte dans lequel on va utiliser h

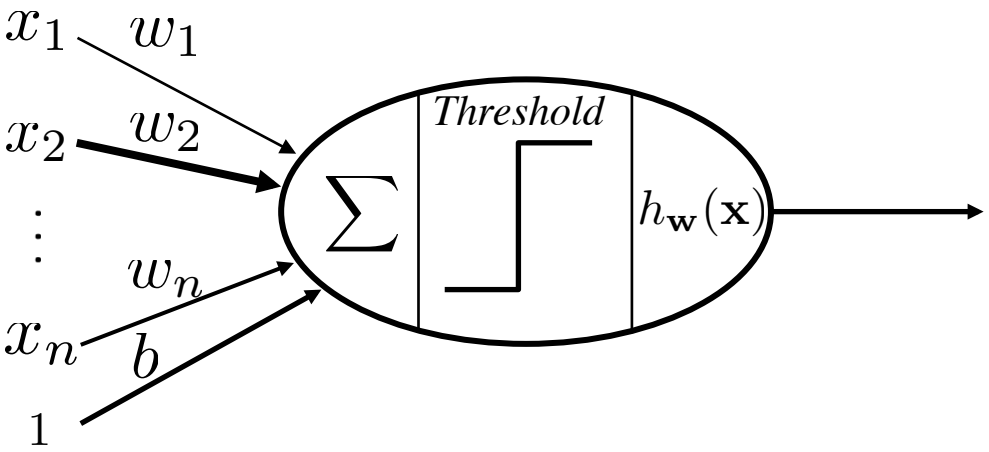
- Pour mesurer la généralisation, on met de côté des exemples étiquetés, qui seront utilisés seulement à la toute fin, pour calculer l'erreur
  - on l'appel l'**ensemble de test**

Deuxième algorithme: Perceptron
(Rosenblatt, 1957)

- Un des plus vieux algorithmes de classification
- **Idée**: modéliser la décision à l'aide d'une fonction linéaire, suivi d'un seuil:

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

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



- Le **vecteur de poids** $\mathbf{W}$ correspond aux **paramètres** du modèle
- On ajoute également un biais b, qui équivaut à ajouter une entrée $x_{n+1} = 1$

Deuxième algorithme: Perceptron
(Rosenblatt, 1957)

- L'algorithme d'apprentissage doit adapter la valeur des paramètres (c'est-à-dire les poids et le biais) 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 $h_{\mathbf{w}}(\mathbf{x}_t) = Threshold(\mathbf{w} \cdot \mathbf{x}_t)$
     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**. La multiplicateur α est appelé le **taux d'apprentissage**

Deuxième algorithme: Perceptron
(Rosenblatt, 1957)

- L'algorithme d'apprentissage doit adapter la valeur des paramètres (c'est-à-dire les poids et le biais) 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 $h_{\mathbf{w}}(\mathbf{x}_t) = Threshold(\mathbf{w} \cdot \mathbf{x}_t)$
     b. si $y_t \neq h_{\mathbf{w}}(\mathbf{x}_t)$
        - $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_t - h_{\mathbf{w}}(\mathbf{x}_t))\mathbf{x}_t$ (mise à jour des poids et biais)

     forme vectorielle
  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**. La multiplicateur α est appelé le **taux d'apprentissage**

Exemple

D ensemble entraînement

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

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

Exemple

D ensemble entraînement

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

- Simulation **avec biais**, $\alpha = 0.1$
- Valeur courante : $\mathbf{w} \longleftarrow [0, 0]$, $b = 0.5$
- Paire $(\mathbf{x}_2, y_2)$:
  - $h(\mathbf{x}_2) = \textit{Threshold}(\mathbf{w} \cdot \mathbf{x}_2 + b) = \textit{Threshold}(0.5) = 1$
  - puisque $h(\mathbf{x}_2) \neq y_2$, on met à jour $\mathbf{w}$ et b
    - » $\mathbf{w} \longleftarrow \mathbf{w} + \alpha (y_2 - h(\mathbf{x}_2)) \mathbf{x}_2 = [0, 0] + 0.1 * (0 - 1) [0, 3] = [0, -0.3]$
    - » $b \longleftarrow b + \alpha (y_2 - h(\mathbf{x}_2)) = 0.5 + 0.1 (0 - 1) = 0.4$

Exemple

D ensemble entraînement

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

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

Exemple

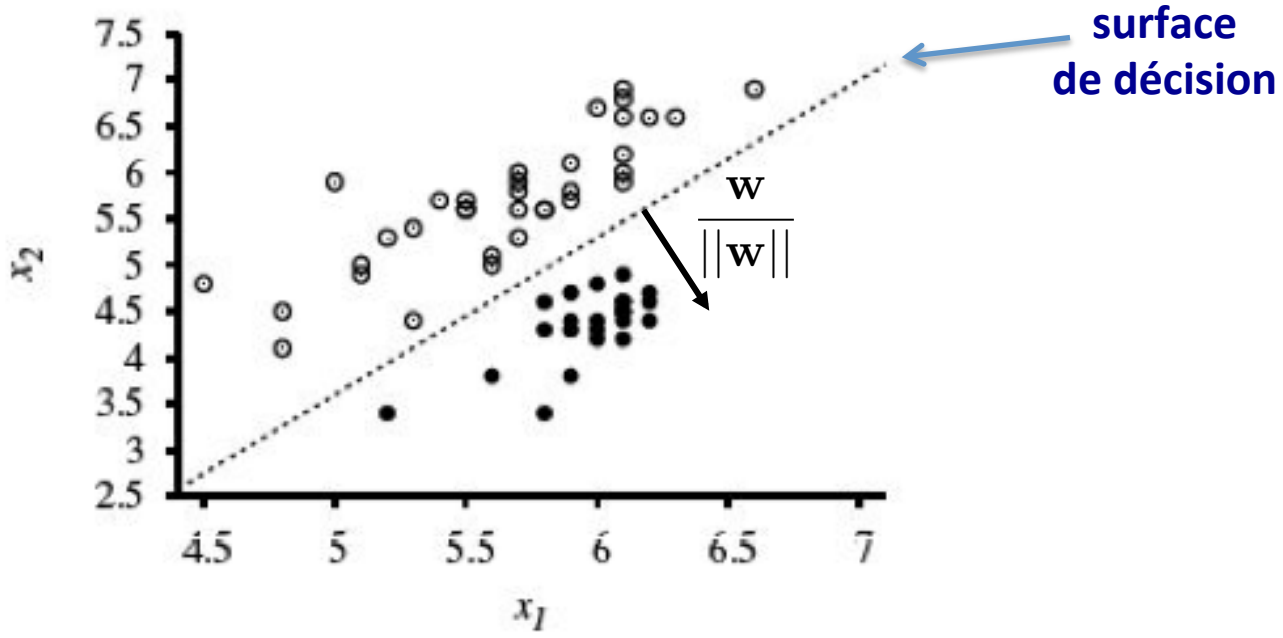
D ensemble entraînement

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

- Simulation **avec biais**, $\alpha = 0.1$
- Valeur courante : $\mathbf{w} \longleftarrow [-0.3, -0.3]$, $b = 0.3$
- Paire $(\mathbf{x}_4, y_4)$:
  - $h(\mathbf{x}_4) = \mathit{Threshold}(\mathbf{w} \cdot \mathbf{x}_4 + b) = \mathit{Threshold}(-0.3) = 0$
  - puisque $h(\mathbf{x}_4) \neq y_4$, on met à jour $\mathbf{w}$ et b
    - » $\mathbf{w} \longleftarrow \mathbf{w} + \alpha (y_4 - h(\mathbf{x}_4)) \mathbf{x}_4 = [-0.3, -0.3] + 0.1 * (1 - 0) [1, 1] = [-0.2, -0.2]$
    - » $b \longleftarrow b + \alpha (y_4 - h(\mathbf{x}_4)) = 0.3 + 0.1 (1 - 0) = 0.4$
- Et ainsi de suite, jusqu'à l'atteinte d'un critère d'arrêt...

Surface de séparation

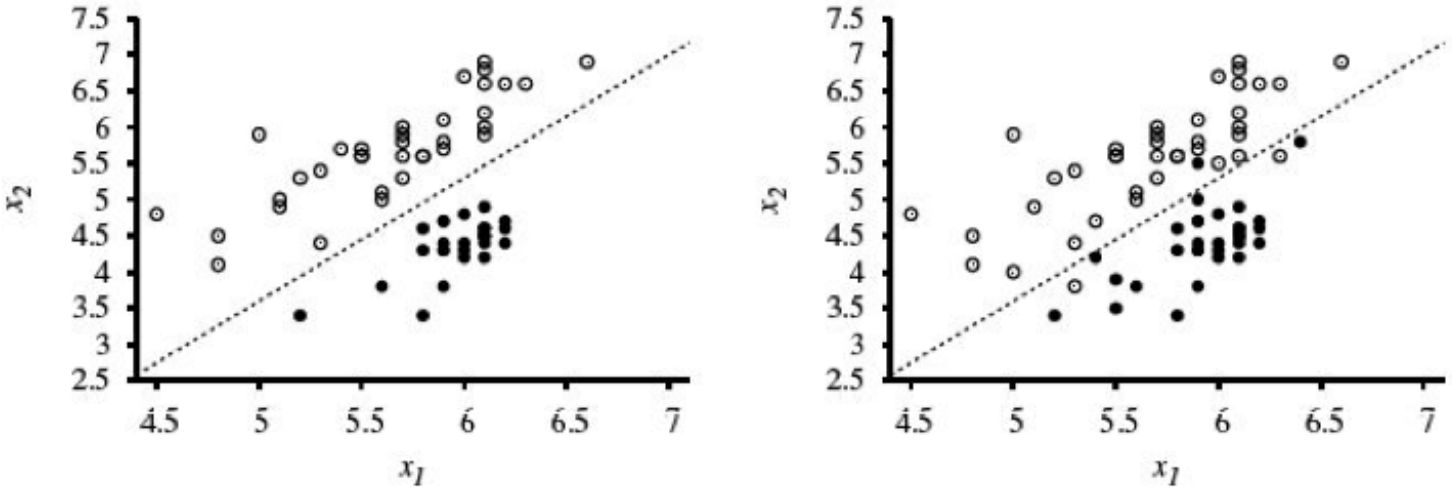
- Le Perceptron cherche donc un **séparateur linéaire** entre les deux classes



- La **surface de décision** d'un classifieur est la surface (dans le cas du perceptron en 2D, une droite) qui sépare les deux régions classifiées dans les deux classes différentes

Convergence et séparabilité

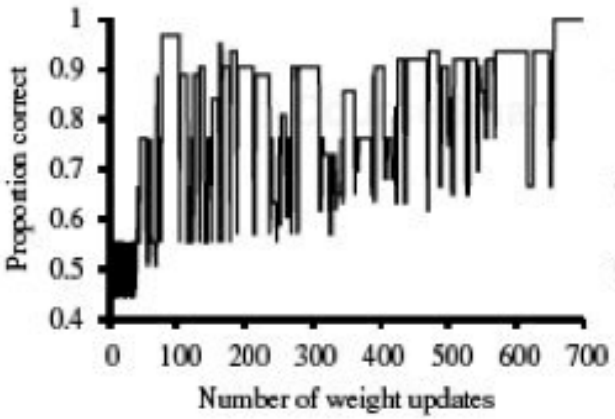
- Si les exemples d'entraînement sont **linéairement séparables** (gauche), l'algorithme est garanti de converger à **une solution avec une erreur nulle** sur l'ensemble d'entraînement, quel que soit le choix de α



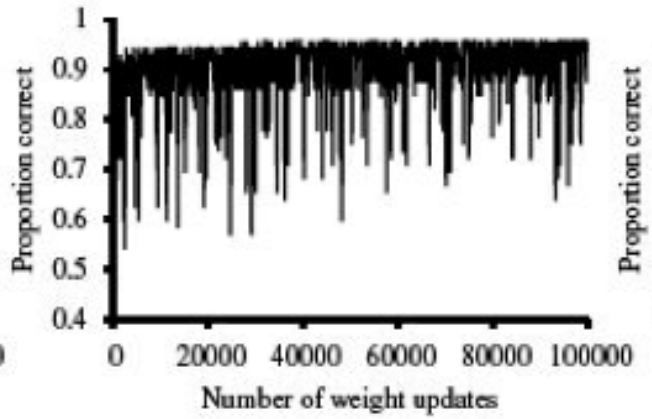
- Si non-séparable linéairement (droite), pour garantir la convergence à une **solution avec la plus petite erreur possible en entraînement**, on doit décroître le taux d'apprentissage, par ex. selon $\alpha_k = \dfrac{\alpha}{1+\beta k}$

Courbe d'apprentissage

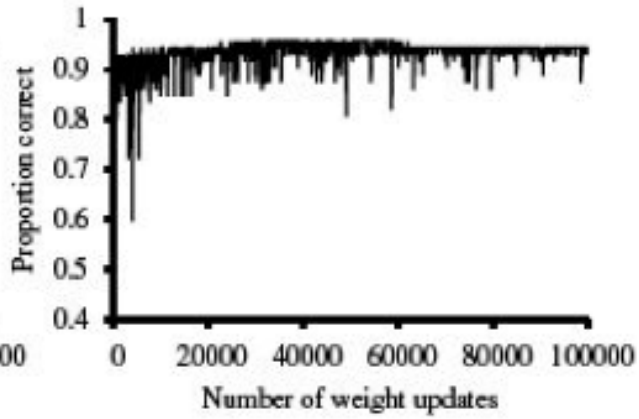
- Pour visualiser la progression de l'apprentissage, on peut regarder la **courbe d'apprentissage**, c'est-à-dire la courbe du taux d'erreur (ou de succès) en fonction du nombre de mises à jour des paramètres



linéairement séparable



pas linéairement séparable



pas linéairement séparable, avec taux d'app. décroissant

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**

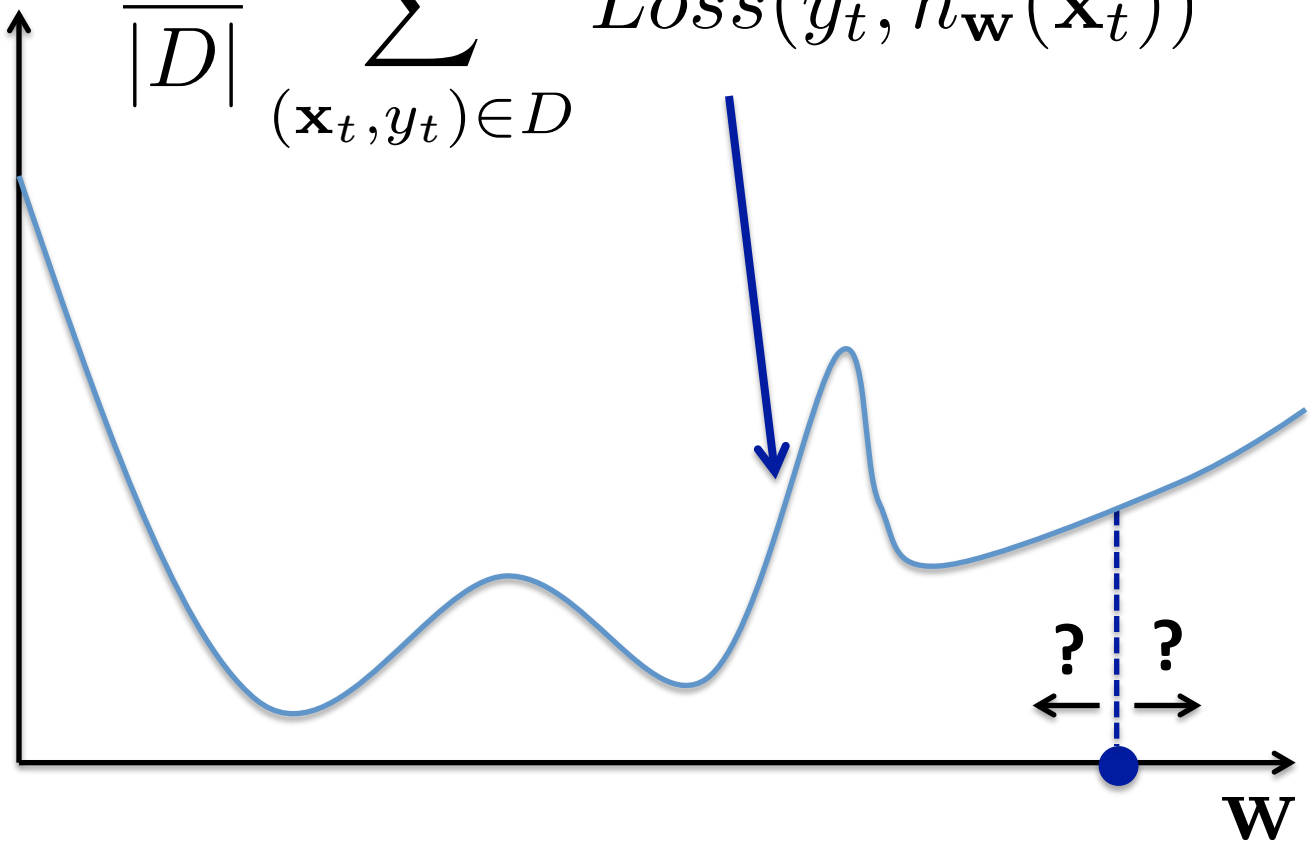
- Dans le cas du perceptron:

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

  - si la prédiction est bonne, le coût est 0
  - si la prédiction est mauvaise, le coût 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

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



Dérivées

- On peut obtenir la direction de descente via la **dérivée**

$$f'(a) = \frac{df(a)}{da} = \lim_{\Delta \to 0} \frac{f(a+\Delta) - f(a)}{\Delta}$$

- Le signe de la dérivée est la **direction d'augmentation** de f
  - signe positif indique que $f(a)$ augmente lorsque a augmente
  - signe négatif indique que $f(a)$ diminue lorsque a augmente
- La valeur absolue de la dérivée est le **taux d'augmentation** de f

- Plutôt que d, je vais utiliser le symbole ∂

Dérivées

- Les dérivées usuelles les plus importantes sont les suivantes:

$$\frac{\partial a}{\partial x} = 0$$

$$\frac{\partial x^n}{\partial x} = nx^{n-1}$$

$$\frac{\partial \log(x)}{\partial x} = \frac{1}{x}$$

$$\frac{\partial \exp(x)}{\partial x} = \exp(x)$$

a et n sont des constantes

Dérivées

- On peut obtenir des dérivées de composition de fonctions

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

$$\frac{\partial f(x)^n}{\partial x} = n f(x)^{n-1} \frac{\partial f(x)}{\partial x}$$

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

$$\frac{\partial \log(f(x))}{\partial x} = \frac{1}{f(x)} \frac{\partial f(x)}{\partial x}$$

a et n sont des constantes

Dérivées

- Exemple 1: $f(x) = 3x^4$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial 3x^4}{\partial x} = 3\frac{\partial x^4}{\partial x} = 12x^3$$

Dérivées

- Exemple 2: $f(x) = \exp\left(\frac{x^2}{3}\right)$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial \exp\left(\frac{x^2}{3}\right)}{\partial x} = \exp\left(\frac{x^2}{3}\right) \frac{\partial \frac{x^2}{3}}{\partial x}$$

$$= \frac{1}{3} \exp\left(\frac{x^2}{3}\right) \frac{\partial x^2}{\partial x} = \frac{2}{3} \exp\left(\frac{x^2}{3}\right) x$$

Dérivées

- Pour des combinaisons plus complexes:

$$\frac{\partial g(x) + h(x)}{\partial x} = \frac{\partial g(x)}{\partial x} + \frac{\partial h(x)}{\partial x}$$

$$\frac{\partial g(x)h(x)}{\partial x} = \frac{\partial g(x)}{\partial x}h(x) + g(x)\frac{\partial h(x)}{\partial x}$$

$$\frac{\partial \frac{g(x)}{h(x)}}{\partial x} = \frac{\partial g(x)}{\partial x}\frac{1}{h(x)} - \frac{g(x)}{h(x)^2}\frac{\partial h(x)}{\partial x}$$

Dérivées

- Exemple 3: $f(x) = x \exp(x)$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial x}{\partial x} \exp(x) + x \frac{\partial \exp(x)}{\partial x}$$

$$= \exp(x) + x \exp(x)$$

Dérivées

- Exemple 4: $f(x) = \frac{\exp(x)}{x}$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial \exp(x)}{\partial x}\frac{1}{x} - \frac{\exp(x)}{x^2}\frac{\partial x}{\partial x}$$

$$= \frac{\exp(x)}{x} - \frac{\exp(x)}{x^2}$$

Dérivées

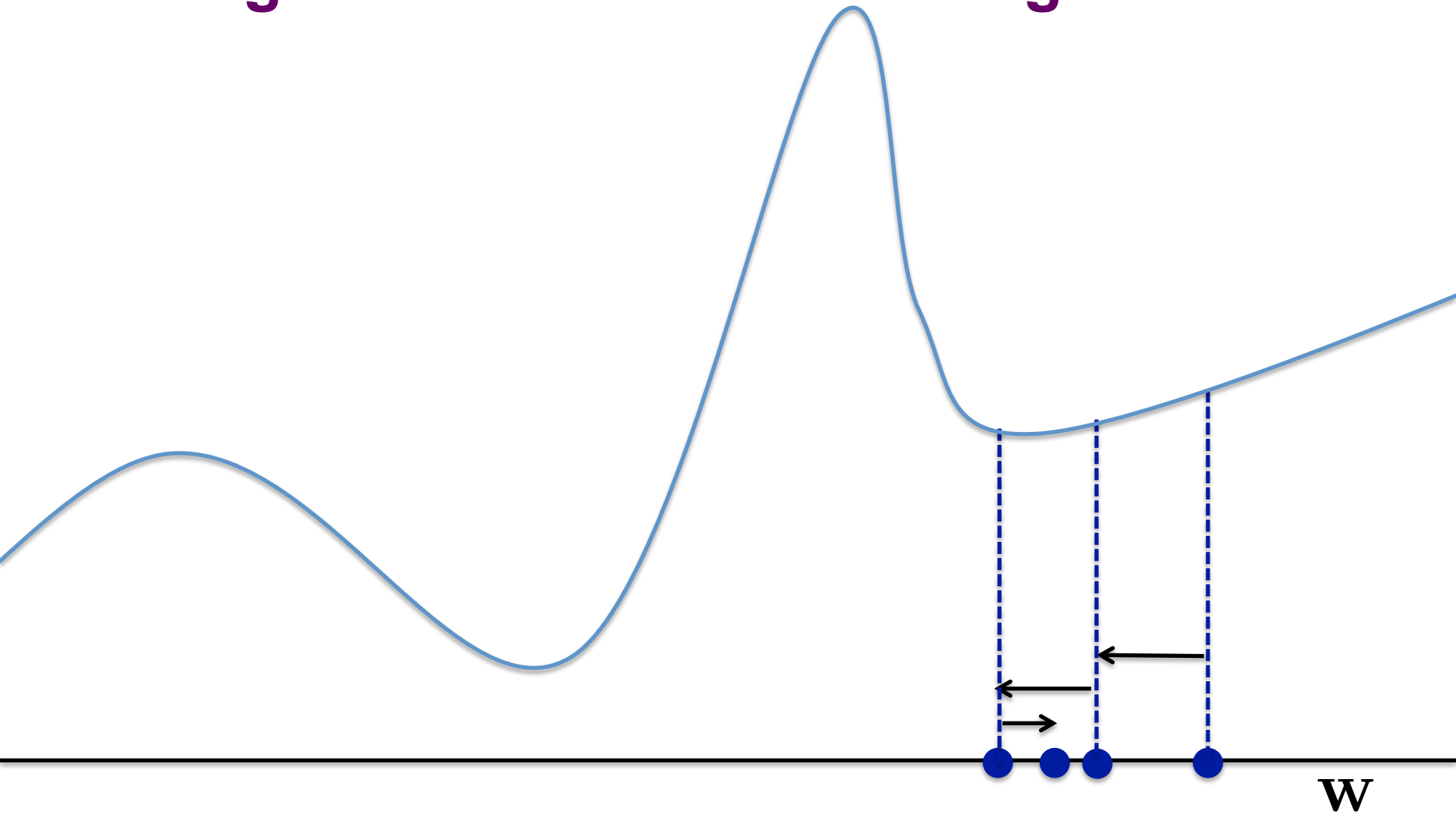
- Exemple 4: $f(x) = \frac{\exp(x)}{x}$

dérivation alternative!

$$\frac{\partial f(x)}{\partial x} = \frac{\partial \exp(x)}{\partial x}\frac{1}{x} + \exp(x)\frac{\partial \frac{1}{x}}{\partial x}$$

$$= \frac{\exp(x)}{x} - \frac{\exp(x)}{x^2}$$

Algorithme de descente de gradient



Dérivée partielle et gradient

- Dans notre cas, la fonction à optimiser dépend de plus d'une variable
  - elle dépend de tout le vecteur $\mathbf{w}$
- Dans ce cas, on va considérer les **dérivées partielles**, c.-à-d. la dérivée par rapport à chacune des variables en supposant que les autres sont constantes:

$$\frac{\partial f(a,b)}{\partial a} = \lim_{\Delta \to 0} \frac{f(a+\Delta, b) - f(a,b)}{\Delta}$$

$$\frac{\partial f(a,b)}{\partial b} = \lim_{\Delta \to 0} \frac{f(a, b+\Delta) - f(a,b)}{\Delta}$$

Dérivée partielle et gradient

- Exemple de fonction à deux variables:

$$f(x,y) = \frac{x^2}{y}$$

- Dérivées partielles:

$$\frac{\partial f(x,y)}{\partial x} = \underbrace{\frac{2x}{y}}_{\text{traite } y \text{ comme une constante}} \qquad \frac{\partial f(x,y)}{\partial y} = \underbrace{\frac{-x^2}{y^2}}_{\text{traite } x \text{ comme une constante}}$$

Dérivée partielle et gradient

- Un deuxième exemple:

$$f(\mathbf{x}) = \frac{\exp(x_2)}{\exp(x_1) + \exp(x_2) + \exp(x_3)}$$

- Dérivée partielle $\frac{\partial f(\mathbf{x})}{\partial x_1}$:

  équivaut à faire la dérivée de $f(x) = \frac{a}{\exp(x) + b}$

  où $x = x_1$
  et on a des **constantes** $a = \exp(x_2)$ et $b = \exp(x_2) + \exp(x_3)$

Dérivée partielle et gradient

- Un deuxième exemple: $f(x) = \frac{a}{\exp(x) + b}$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial}{\partial x} \frac{a}{\exp(x) + b} = a \frac{\partial}{\partial x} \frac{1}{\exp(x) + b}$$

$$= \frac{-a}{(\exp(x) + b)^2} \frac{\partial}{\partial x} (\exp(x) + b)$$

$$= \frac{-a \exp(x)}{(\exp(x) + b)^2}$$

Dérivée partielle et gradient

- Un deuxième exemple:

$$\frac{\partial f(x)}{\partial x} = \frac{-a \exp(x)}{(\exp(x) + b)^2}$$

où $x = x_1$, $a = \exp(x_2), b = \exp(x_2) + \exp(x_3)$

- On remplace:

$$\frac{\partial f(\mathbf{x})}{\partial x_1} = \frac{-\exp(x_2)\exp(x_1)}{(\exp(x_1) + \exp(x_2) + \exp(x_3))^2}$$

Dérivée partielle et gradient

- Un troisième exemple:

$$f(\mathbf{x}) = \frac{\exp(x_2)}{\exp(x_1) + \exp(x_2) + \exp(x_3)}$$

- Dérivée partielle $\frac{\partial f(\mathbf{x})}{\partial x_2}$:

  équivaut à faire la dérivée de $f(x) = \frac{\exp(x)}{\exp(x) + b}$

  où $x = x_2$
  et on a une constante $b = \exp(x_1) + \exp(x_3)$

Dérivée partielle et gradient

- Un troisième exemple: $f(x) = \frac{\exp(x)}{\exp(x) + b}$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial \exp(x)}{\partial x} \frac{1}{\exp(x) + b} - \frac{\exp(x)}{(\exp(x) + b)^2} \frac{\partial(\exp(x) + b)}{\partial x}$$

$$= \frac{\exp(x)}{\exp(x) + b} - \frac{\exp(x)\exp(x)}{(\exp(x) + b)^2}$$

Dérivée partielle et gradient

- Un troisième exemple:

$$\frac{\partial f(x)}{\partial x} = \frac{\exp(x)}{\exp(x) + b} - \frac{\exp(x)\exp(x)}{(\exp(x) + b)^2}$$

où $x = x_2$, $b = \exp(x_1) + \exp(x_3)$

- On remplace:

$$\frac{\partial f(\mathbf{x})}{\partial x_2} = \frac{\exp(x_2)}{\exp(x_2) + \exp(x_1) + \exp(x_3)} - \frac{\exp(x_2)\exp(x_2)}{(\exp(x_2) + \exp(x_1) + \exp(x_3))^2}$$

Dérivée partielle et gradient

- On va appeler **gradient** ∇f d'une fonction f le vecteur contenant les dérivées partielles de f par rapport à toutes les variables

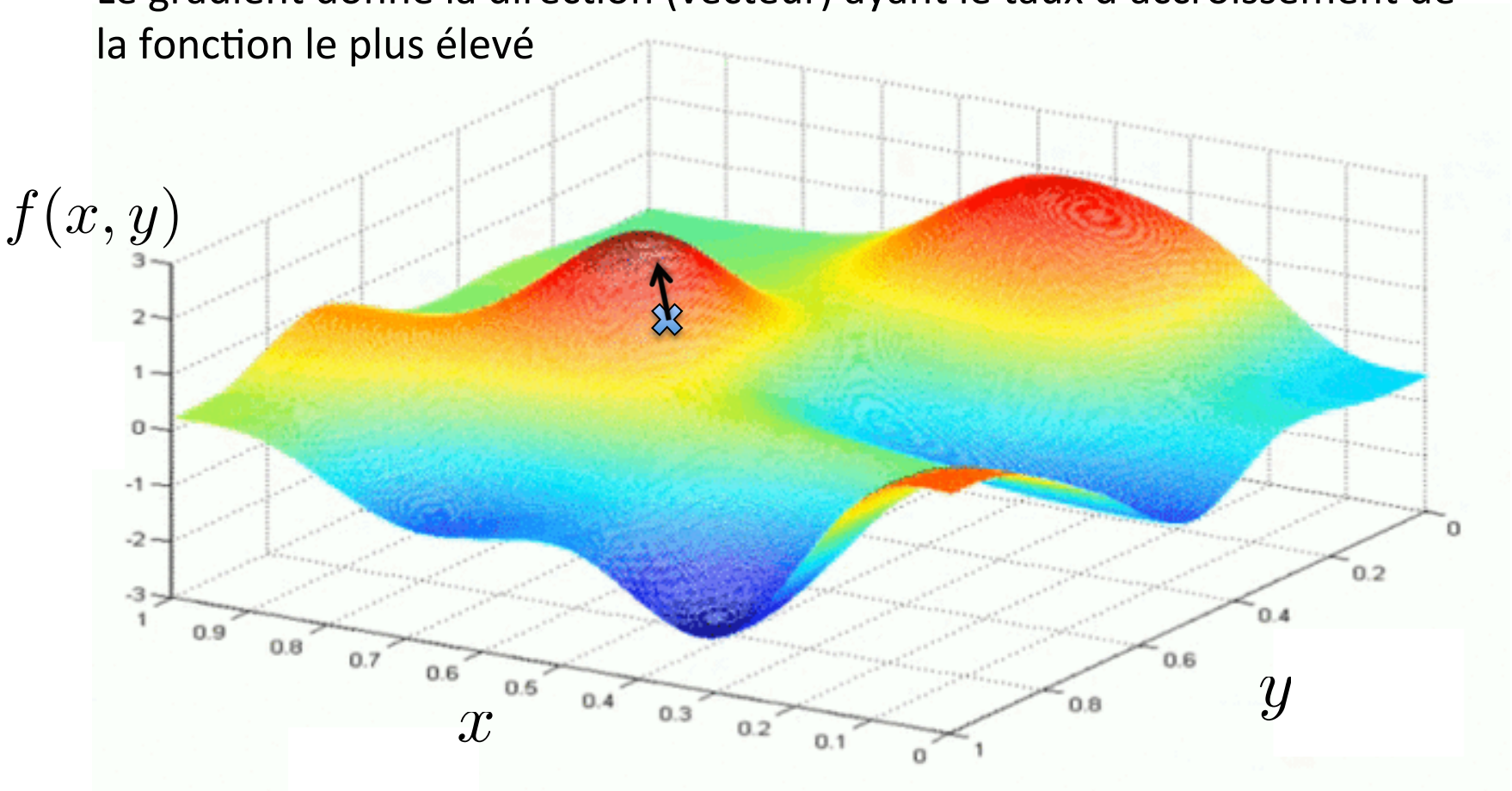
- Dans l'exemple avec la fonction $f(x, y)$:

$$\nabla f(x,y) = \left[\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right]$$

$$= \left[\frac{2x}{y}, \frac{-x^2}{y^2}\right]$$

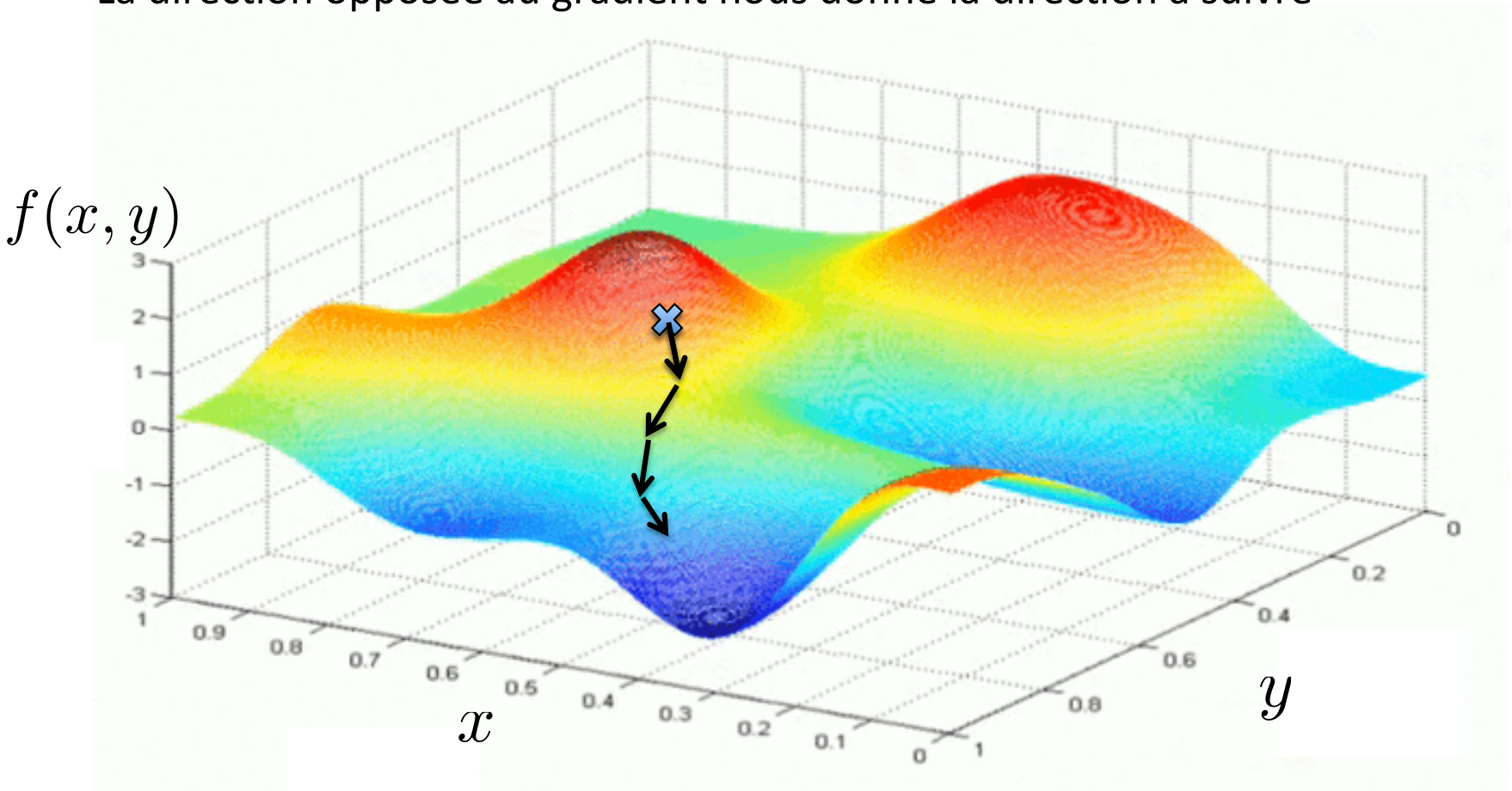
Descente de gradient

- 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 (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 beaucoup 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 pourrait utiliser le gradient (dérivée partielle) pour déterminer une direction de mise à jour des paramètres:

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

- 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:

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

  - on obtient à nouveau la règle d'apprentissage du Perceptron

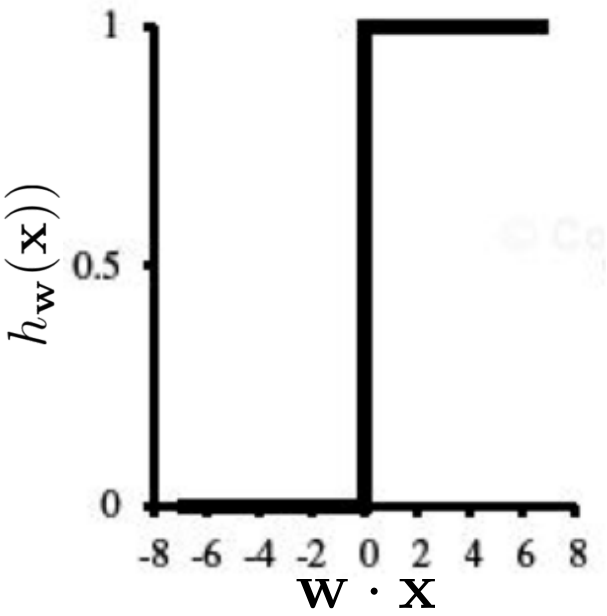
$$w_i \leftarrow w_i + \alpha(y_t - h_{\mathbf{w}}(\mathbf{x}_t))x_{t,i} \quad \forall i$$

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 $h_{\mathbf{w}}(\mathbf{x}))$ n'est pas définie lorsque $\mathbf{w} \cdot \mathbf{x} = 0$

- L'utilisation de la fonction *Threshold* (qui est constante par partie) fait que la courbe d'entraînement peut être instable

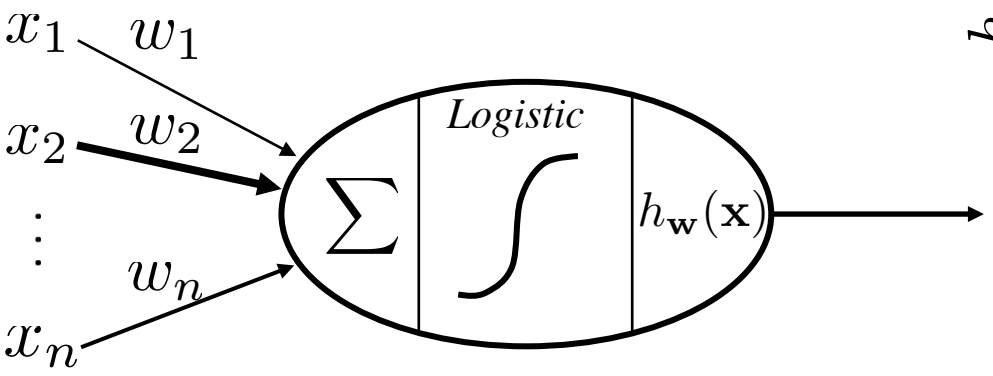


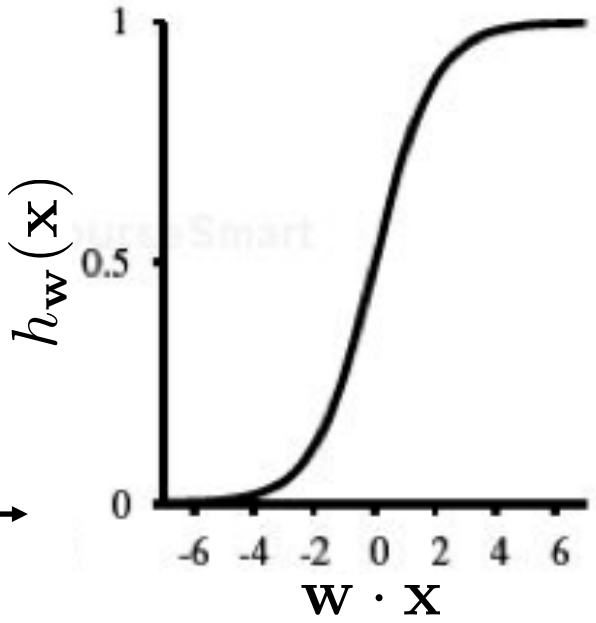
Troisième algorithme: régression logistique

- **Idée**: plutôt que de prédire une classe, prédire une probabilité d'appartenir à la classe 1 (ou la classe 0, ça marche aussi)

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

- Pour choisir une classe, prendre la plus probable selon le modèle
  - si $h_{\mathbf{w}}(\mathbf{x}) >= 0.5$ 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

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

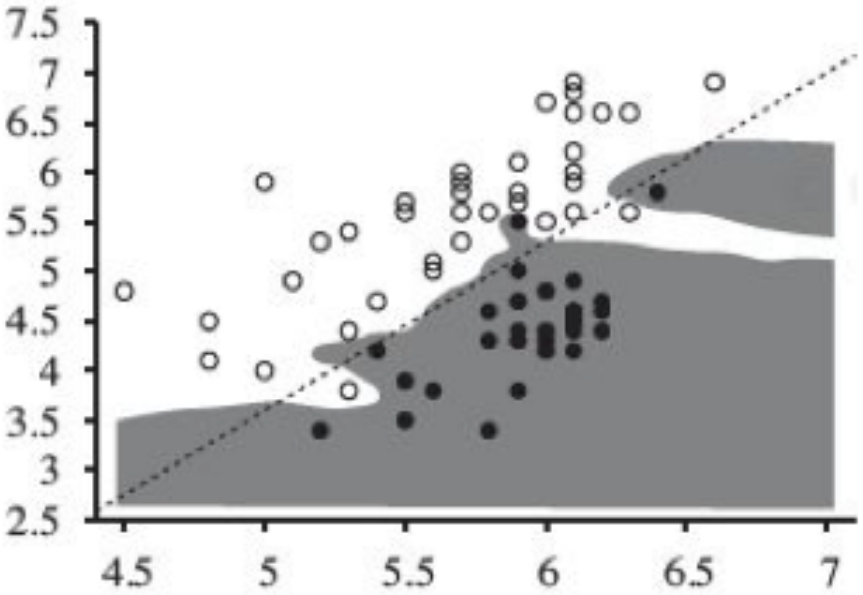
  ce qui donne

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

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

Limitation des classifieurs linéaires

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

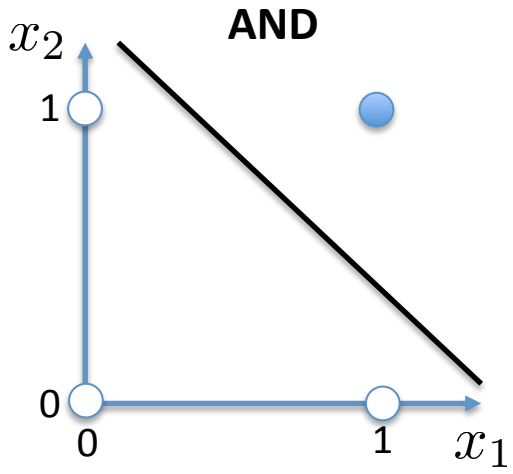


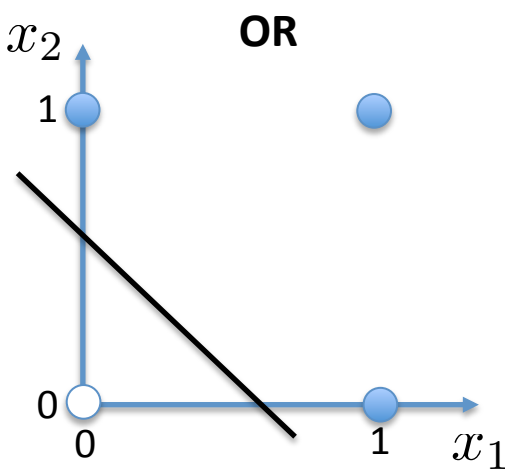
= prédiction classe • par k plus proche voisins

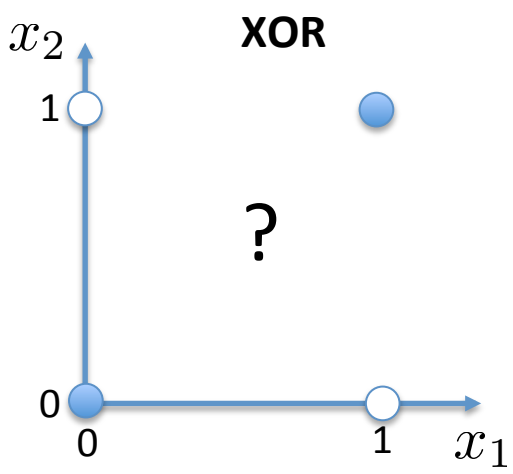
- *k* plus proche voisins est non-linéaire, mais coûteux en mémoire et temps de calcul (pas approprié pour des problèmes avec beaucoup de données)

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!

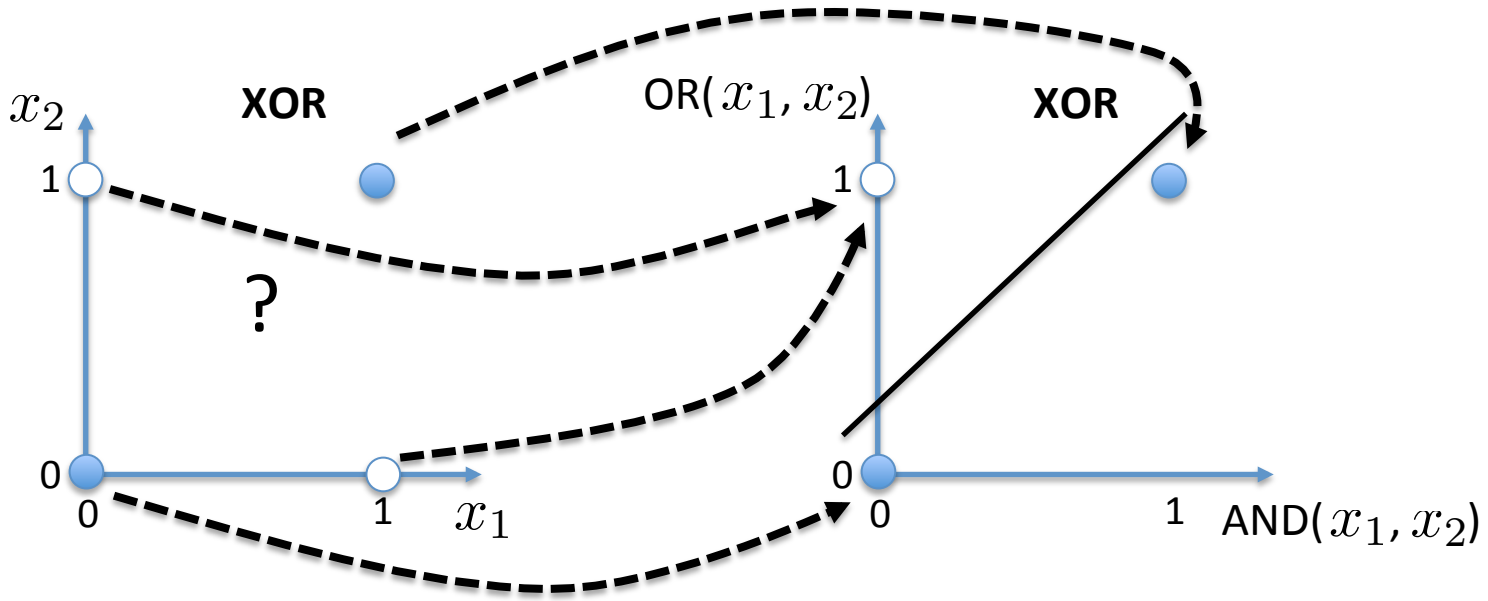






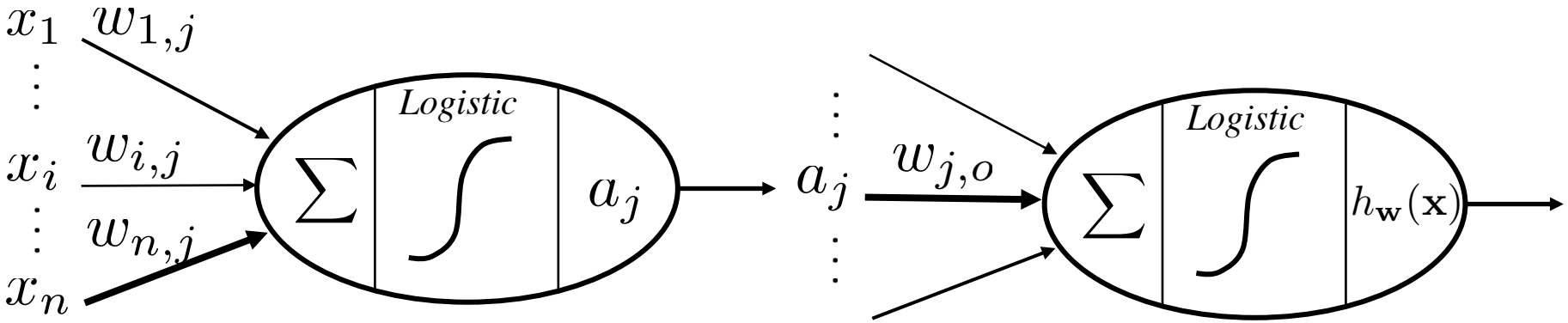
Limitation des classifieurs linéaires

- 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 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

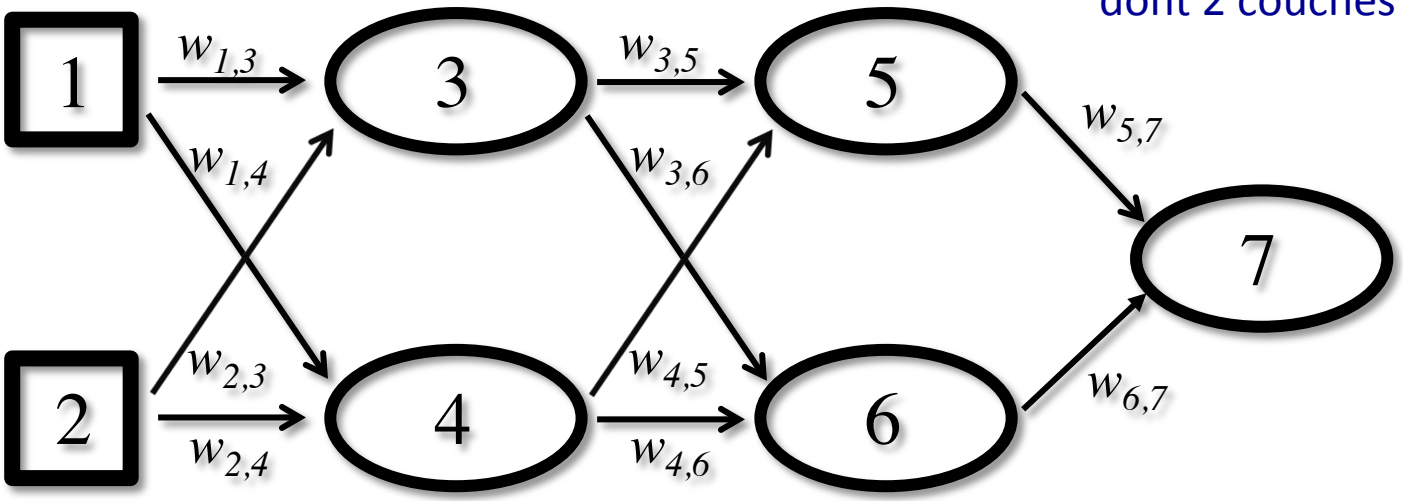


Réseau de neurones à une seule couche cachée

Cas général à L couches

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

Il y a L = 4 couches
dont 2 couches cachées



- 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 $a_j = Logistic(in_j) = Logistic(\Sigma_i\ w_{i,j} a_i)$

Dérivation de la règle d'apprentissage

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

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

- 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 $g(x)$

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

- De façon récurrente, si on peut exprimer $g(x)$ à partir de $h(x)$

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

- 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 en chaîne

- Exemple: $f(x) = 4\exp(x) + 3(1+x)^3$

- On considère $g_1(x) = \exp(x)$ et $g_2(x) = 1 + x$
- Donc on peut écrire $f(x) = 4g_1(x) + 3g_2(x)^3$
- On peut obtenir la dérivée partielle avec les morceaux:

$$\frac{\partial f(x)}{\partial g_1(x)} = 4 \qquad \frac{\partial g_1(x)}{\partial x} = \exp(x)$$

$$\frac{\partial f(x)}{\partial g_2(x)} = 9g_2(x)^2 \qquad \frac{\partial g_2(x)}{\partial x} = 1$$

- Donc: $\frac{\partial f(x)}{\partial x} = 4\exp(x) + 9g_2(x) = 4\exp(x) + 9(1+x)^2$

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 peut décomposer cette règle d'apprentissage comme suit:

$$w_{i,j} \leftarrow w_{i,j} - \alpha \underbrace{\frac{\partial}{\partial a_j} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t))}_{\text{gradient du coût p/r au neurone}} \underbrace{\frac{\partial}{\partial in_j} Logistic(in_j)}_{\text{gradient du neurone p/r à la somme des entrées}} \underbrace{\frac{\partial}{\partial w_{i,j}} in_j}_{\text{gradient de la somme p/r au poids } w_{i,j}}$$

- Par contre, un calcul naïf de tous ces gradients serait très inefficace
- Pour un calcul efficace, 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 $l+1$

$$\frac{\partial}{\partial a_j} Loss = \sum_k \frac{\partial}{\partial a_k} Loss \ \frac{\partial}{\partial a_j} a_k$$

$$= \sum_k \frac{\partial}{\partial a_k} Loss \ \frac{\partial}{\partial in_k} g(in_k) \frac{\partial}{\partial a_j} in_k$$

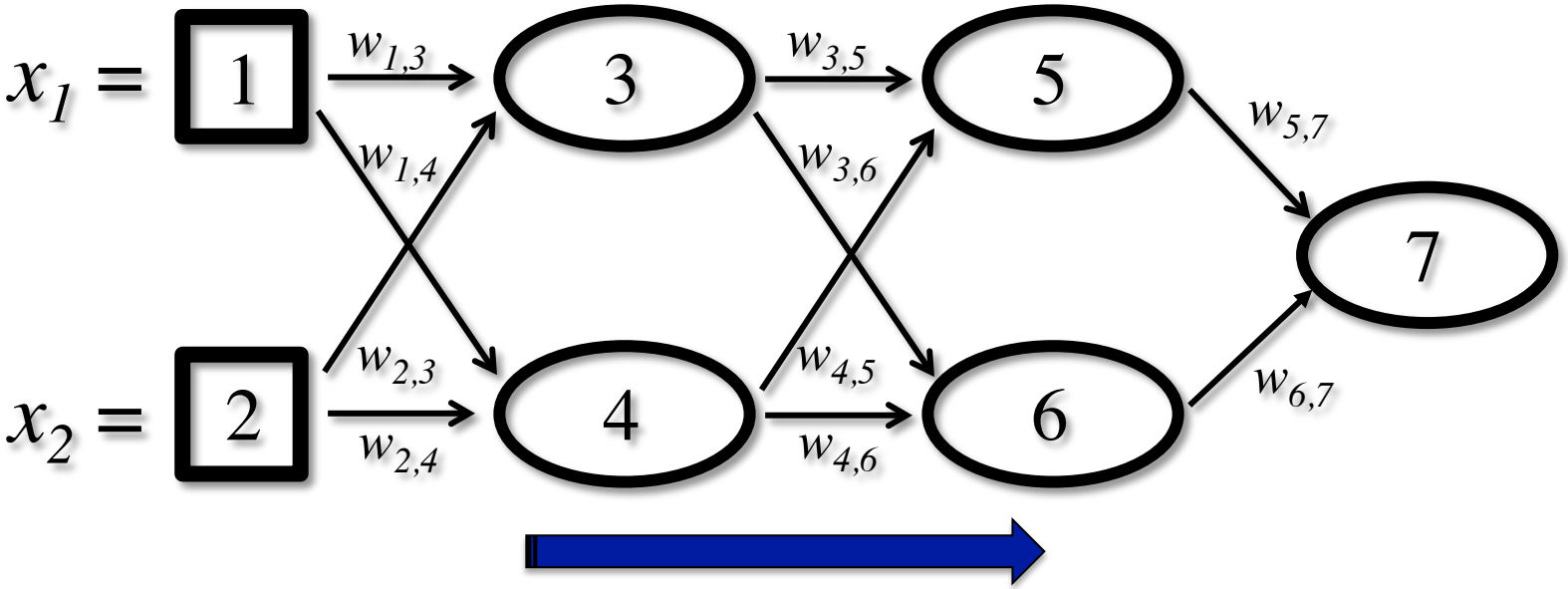
$$= \sum_k \frac{\partial}{\partial a_k} Loss \ g(in_k)(1 - g(in_k)) \ w_{j,k}$$

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

où $in_k = \sum_j w_{j,k} a_j$ et $Logistic(\cdot) \equiv g(\cdot)$ (pour simplifier notation)

Visualisation de la rétropropagation

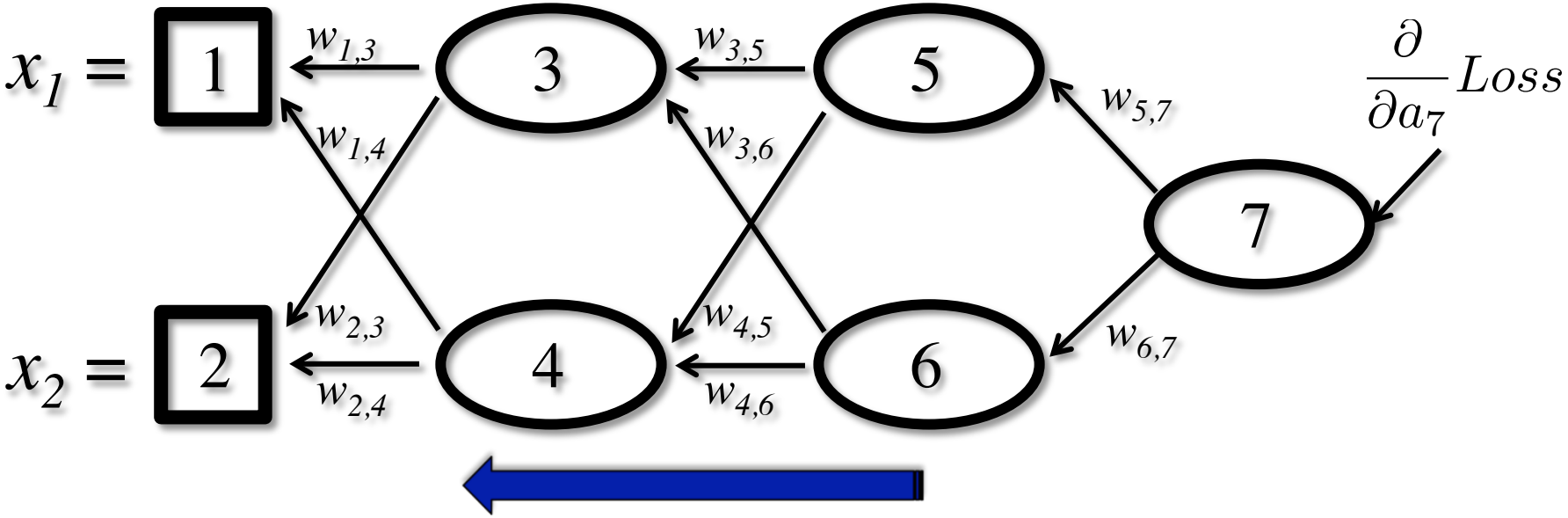
- L'algorithme d'apprentissage commence par une **propagation avant**



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

Visualisation de la rétropropagation

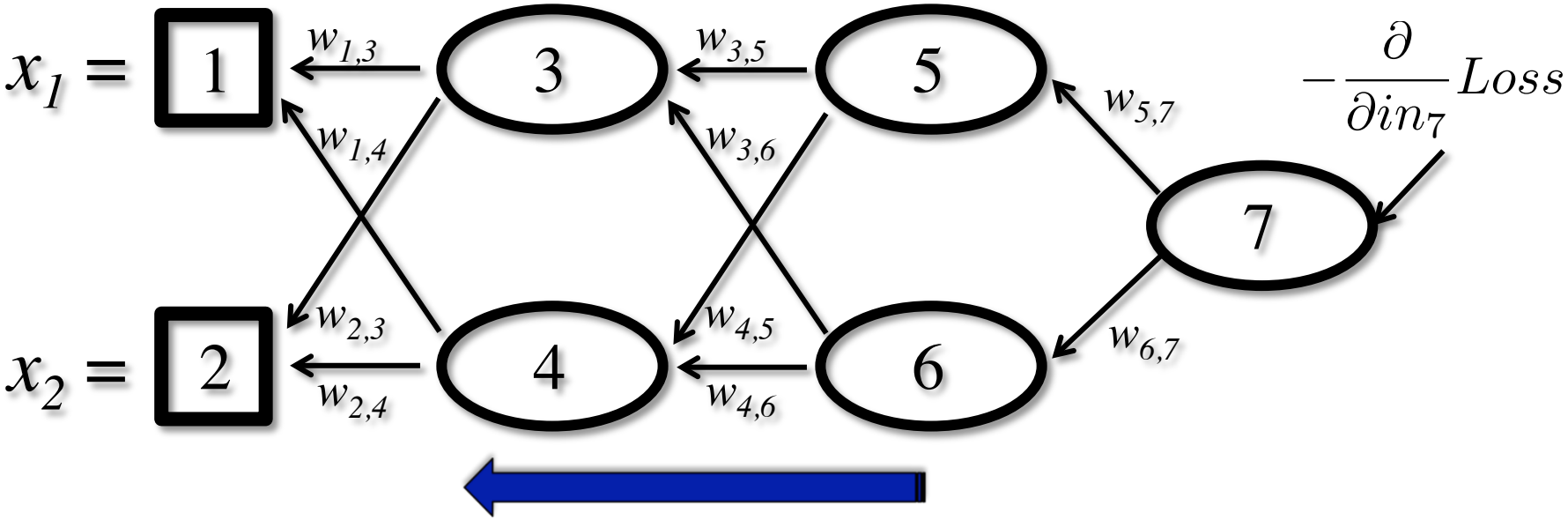
- Ensuite, le gradient sur la sortie est calculé, et le gradient rétropropagé



$$\frac{\partial}{\partial a_j} Loss = \sum_k \frac{\partial}{\partial a_k} Loss \frac{\partial}{\partial a_j} a_k$$

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

$$w_{i,j} \leftarrow w_{i,j} - \alpha \underbrace{\frac{\partial}{\partial a_j} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t))}_{\text{gradient du coût p/r au neurone}} \underbrace{\frac{\partial}{\partial in_j} Logistic(in_j)}_{\text{gradient du neurone p/r à la somme des entrées}} \underbrace{\frac{\partial}{\partial w_{i,j}} in_j}_{\text{gradient de la somme p/r au poids } w_{i,j}}$$

$$\underbrace{\qquad\qquad}_{-\Delta[j]} \qquad \underbrace{\qquad}_{a_i}$$

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

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

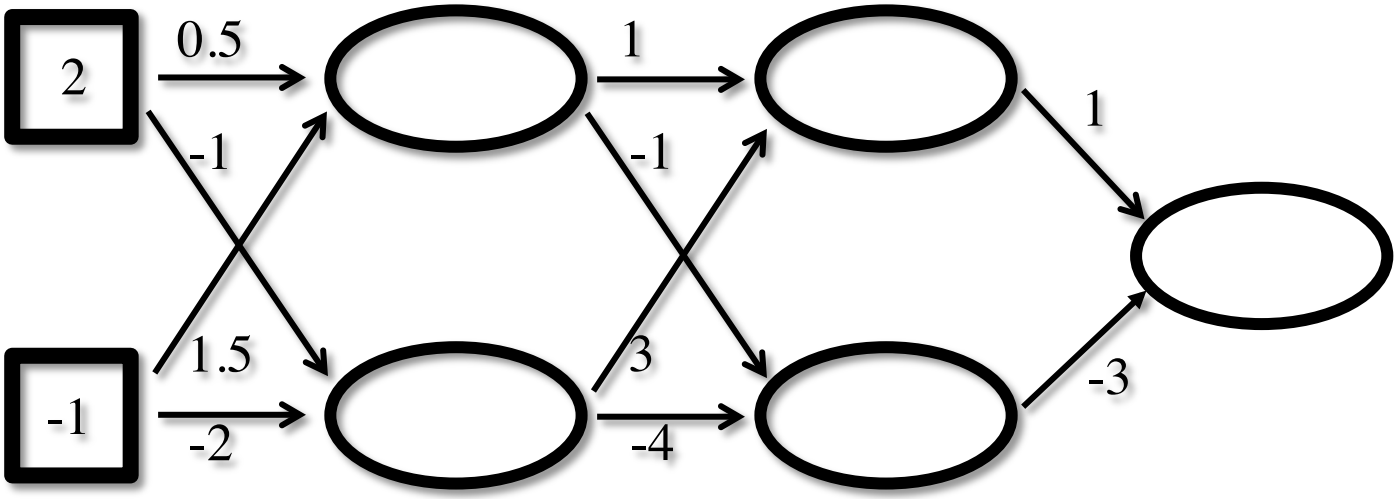
function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network
inputs: *examples*, a set of examples, each with input vector **x** and output vector **y**
network, a multilayer network with L layers, weights $w_{i,j}$, activation function g
local variables: Δ, a vector of errors, indexed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

for each weight $w_{i,j}$ in *network* **do**
$w_{i,j} \leftarrow$ a small random number
repeat
for each example (**x**, **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 ℓ **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**
$\Delta[j] \leftarrow y_j - a_j \quad (= -\partial Loss/\partial in_j)$
for $\ell = L - 1$ **to** 1 **do**
for each node i in layer ℓ **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: $\mathbf{x} = [2, -1]$, $y = 1$

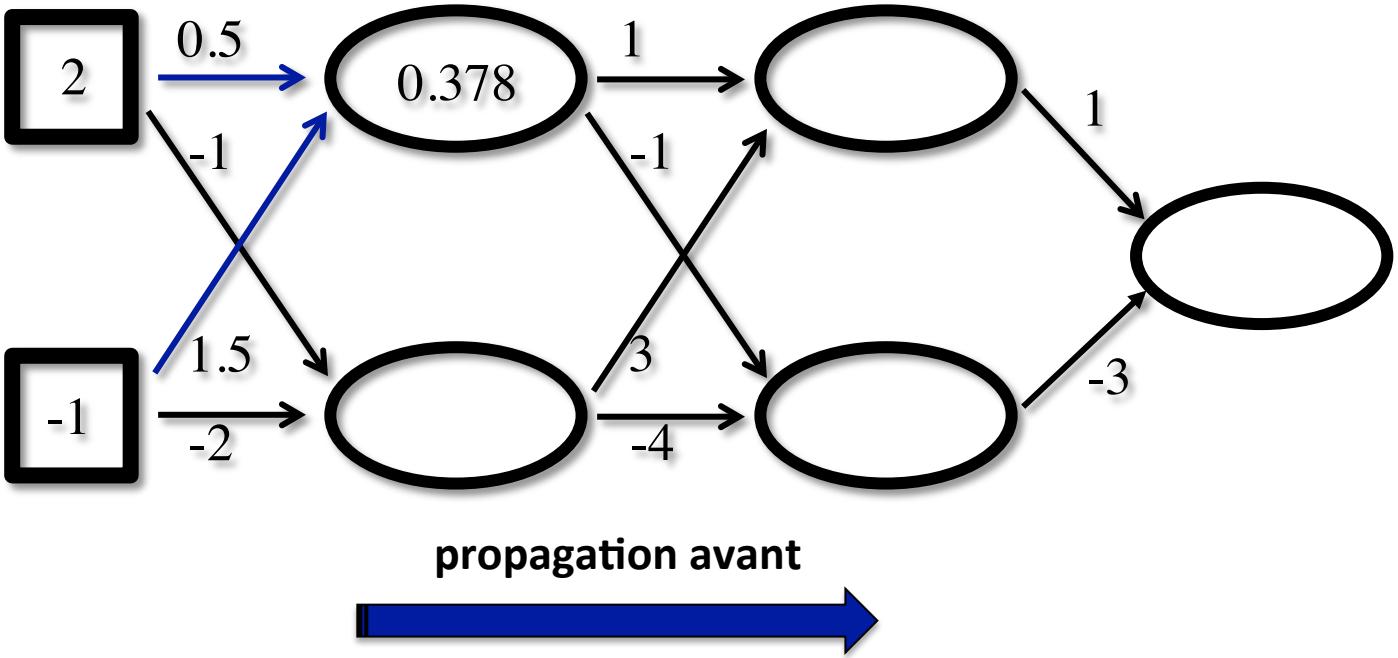


propagation avant

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

Exemple

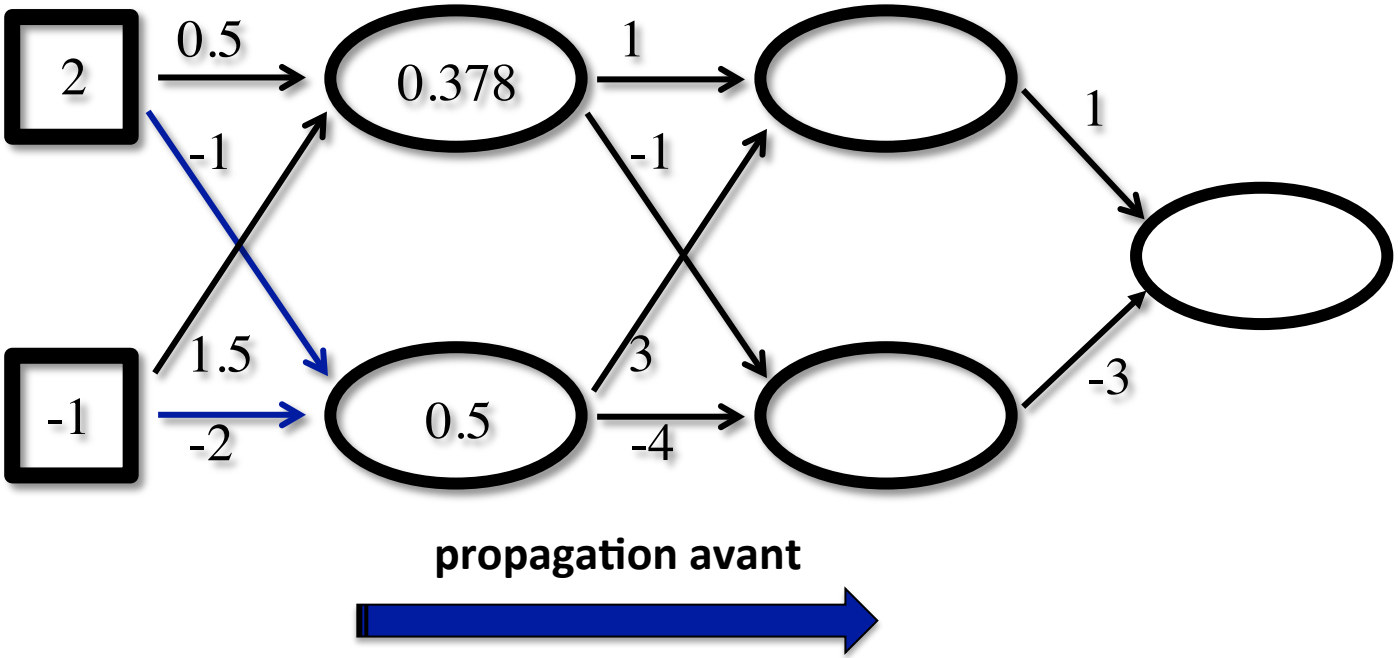
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$Logistic(0.5 * 2 + 1.5 * -1) = Logistic(-0.5) = 0.378$$

Exemple

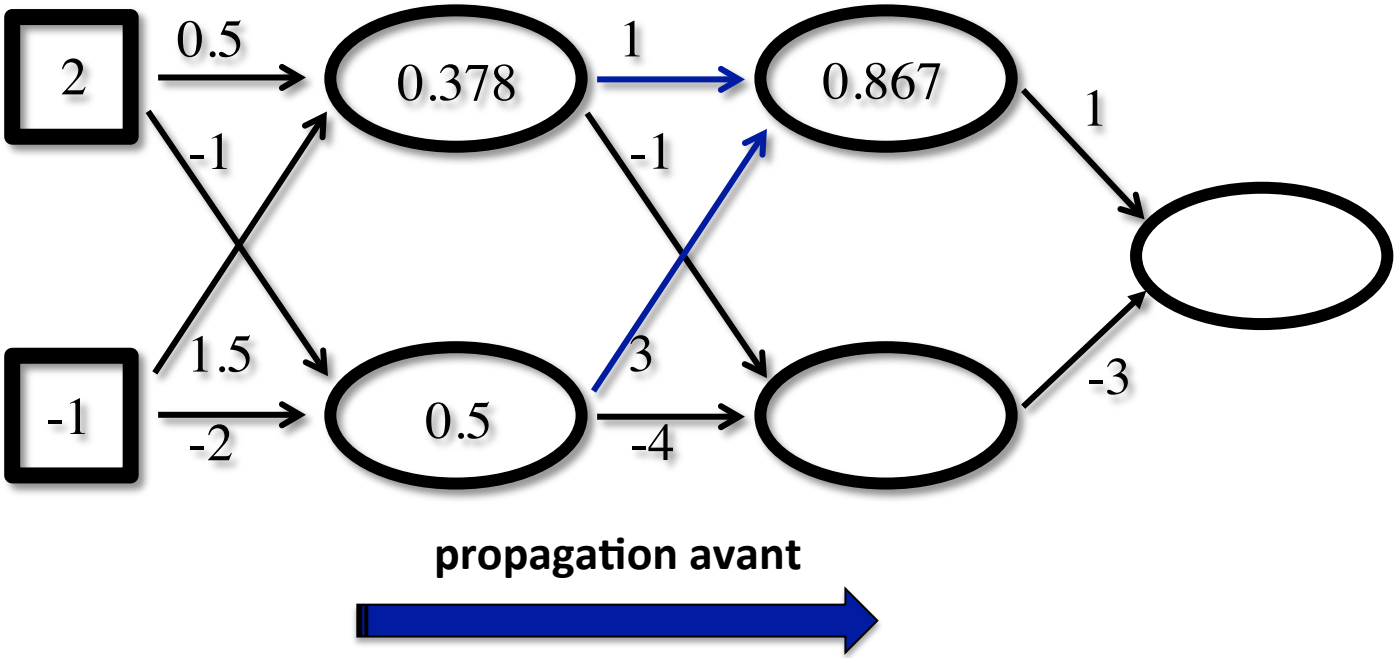
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$Logistic(\text{-}1 * 2 + \text{-}2 * \text{-}1) = Logistic(0) = 0.5$$

Exemple

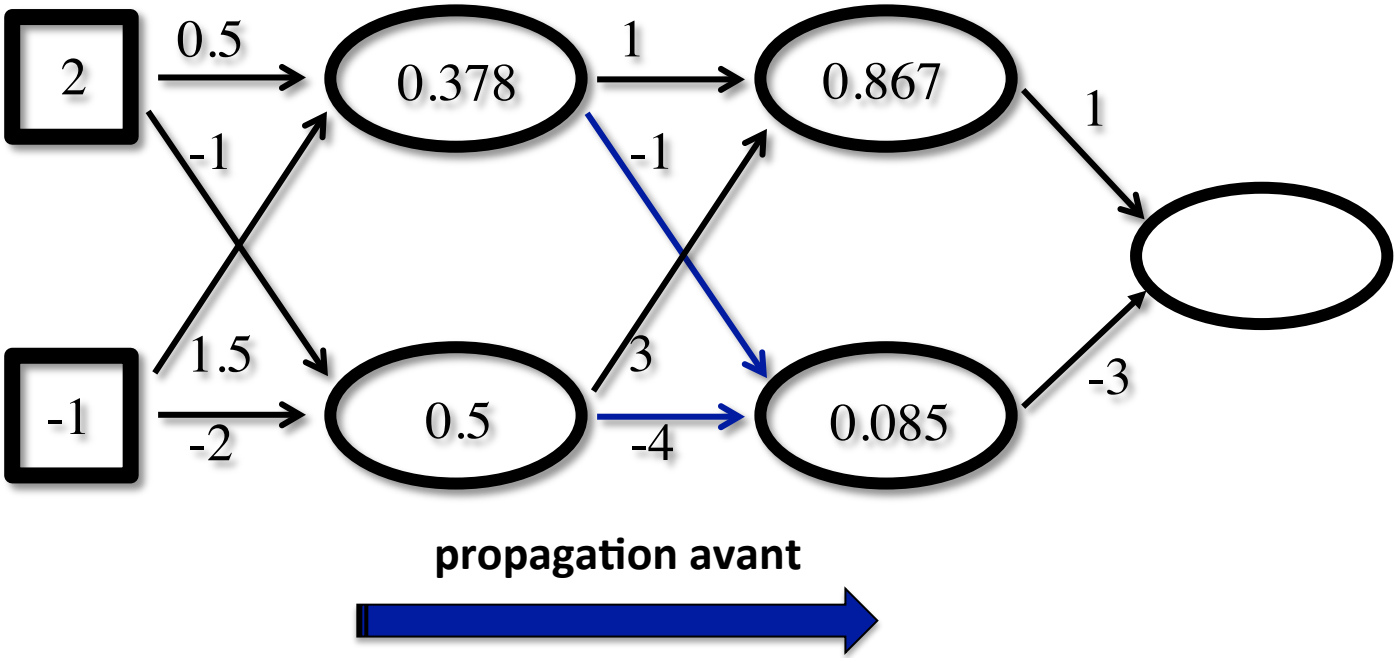
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$Logistic(1 * 0.378 + 3 * 0.5) = Logistic(1.878) = 0.867$$

Exemple

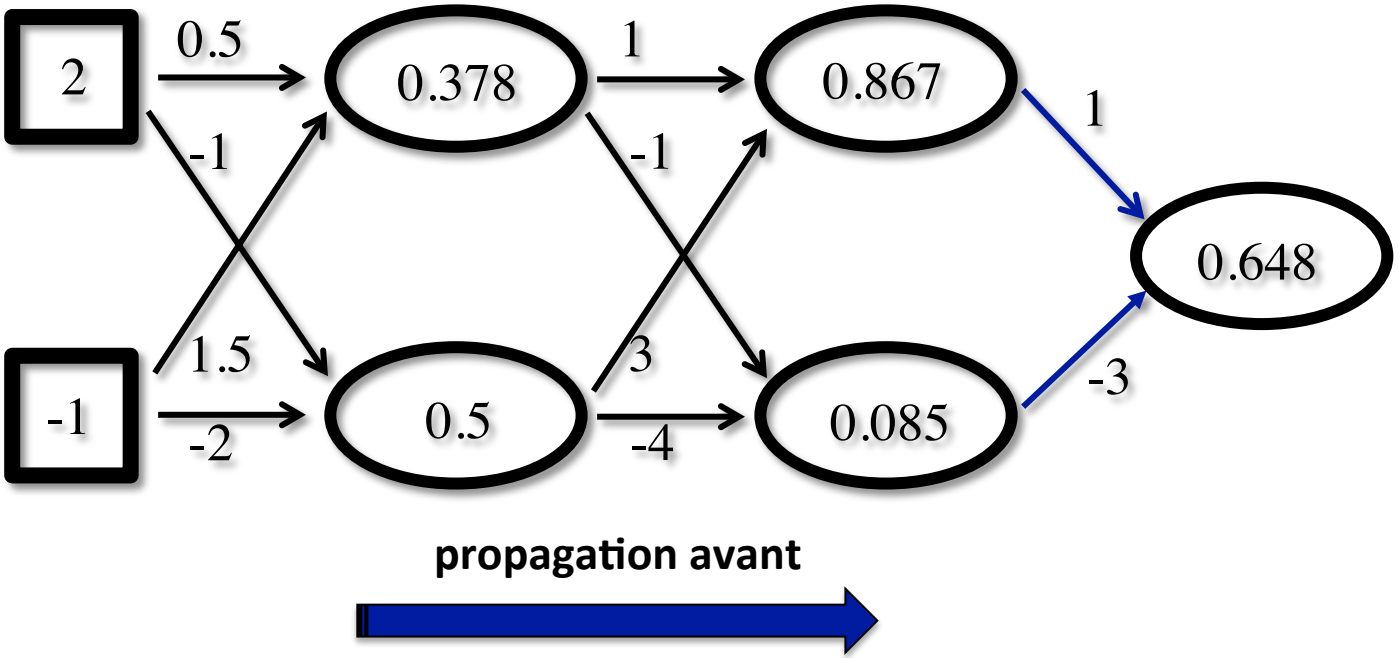
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$Logistic(-1 * 0.378 + -4 * 0.5) = Logistic(-2.378) = 0.085$$

Exemple

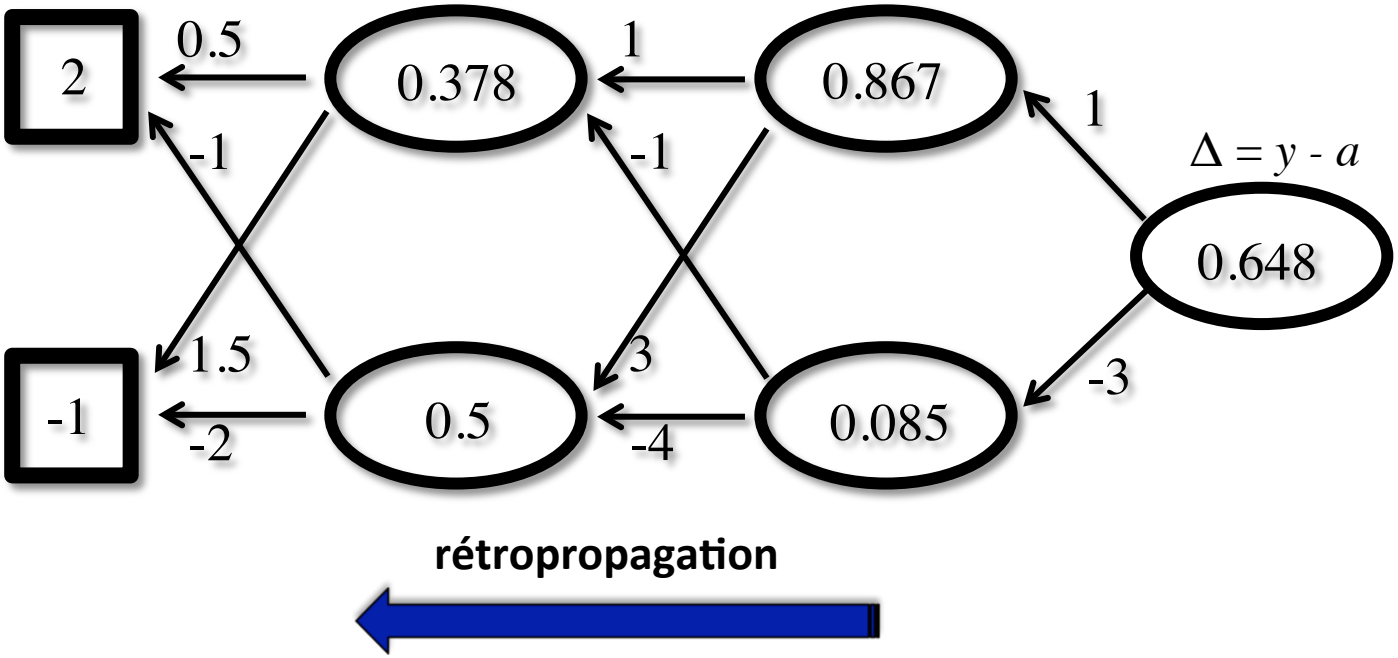
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$Logistic(1 * 0.867 + -3 * 0.085) = Logistic(0.612) = 0.648$$

Exemple

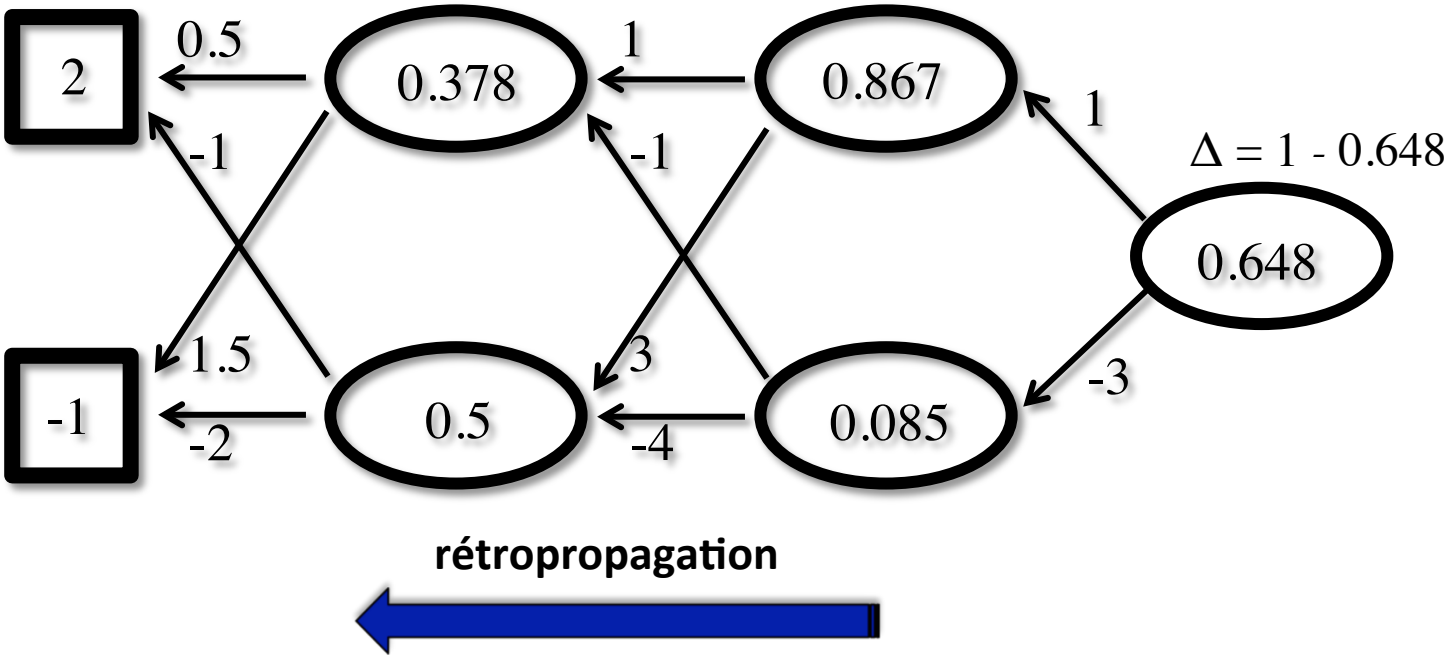
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



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

Exemple

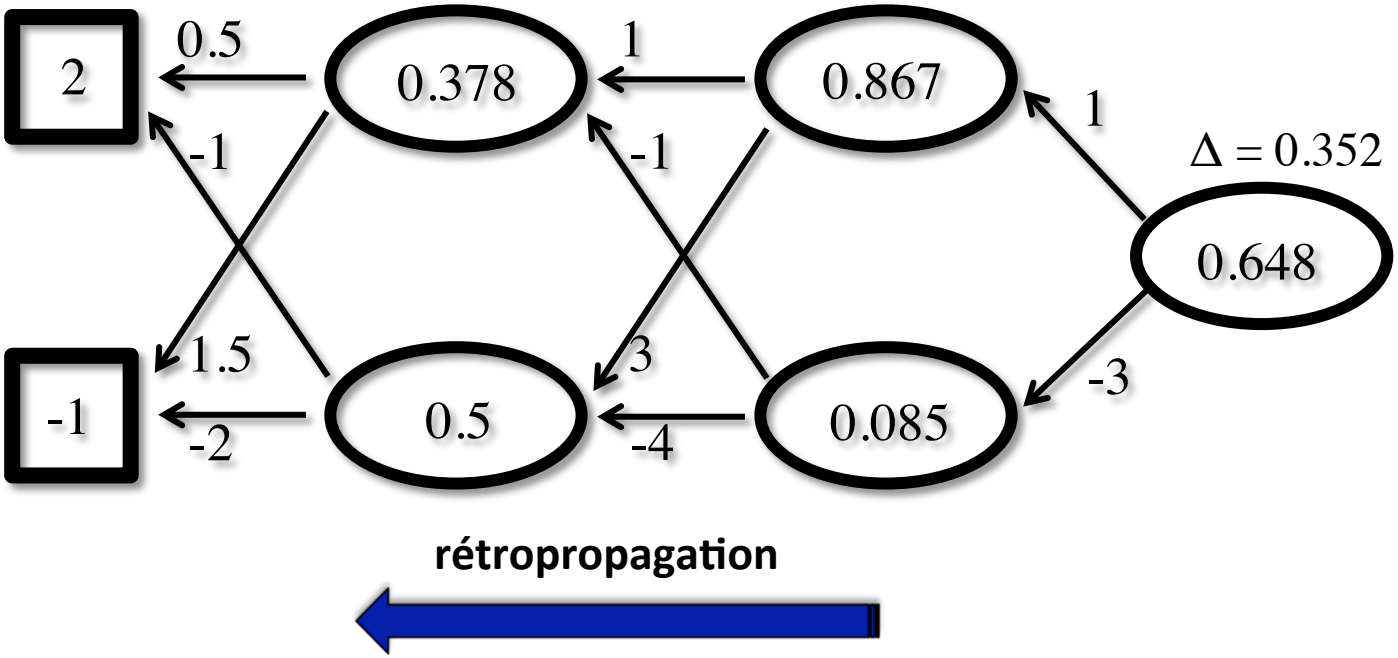
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



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

Exemple

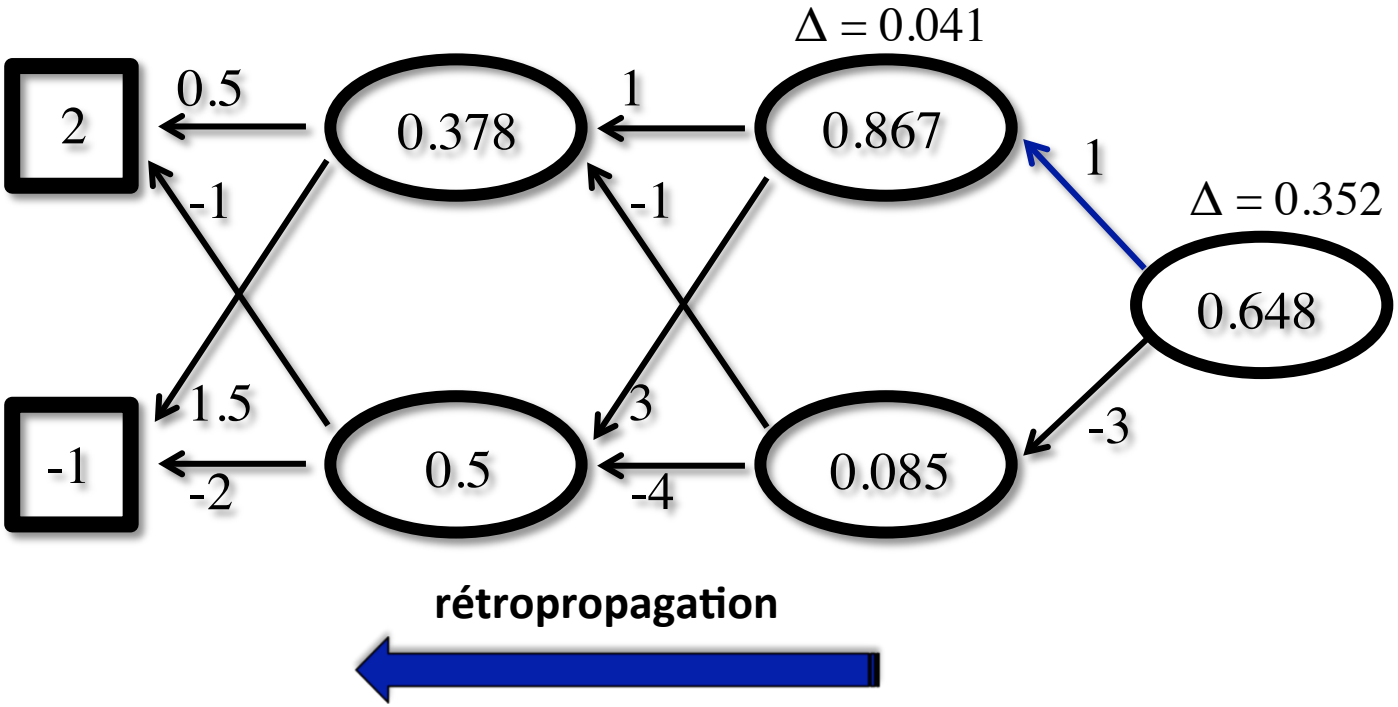
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



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

Exemple

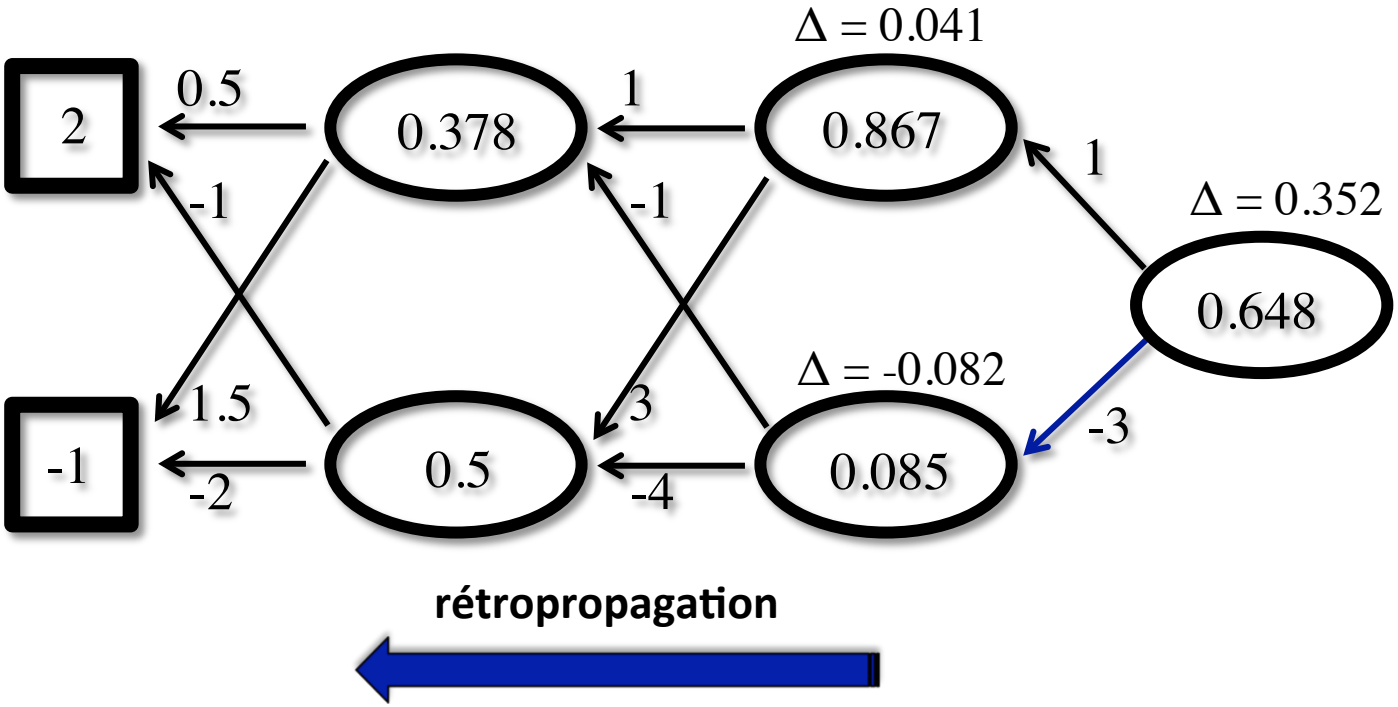
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$\Delta = 0.867 * (1\text{-}0.867) * 1 * 0.352 = 0.041$$

Exemple

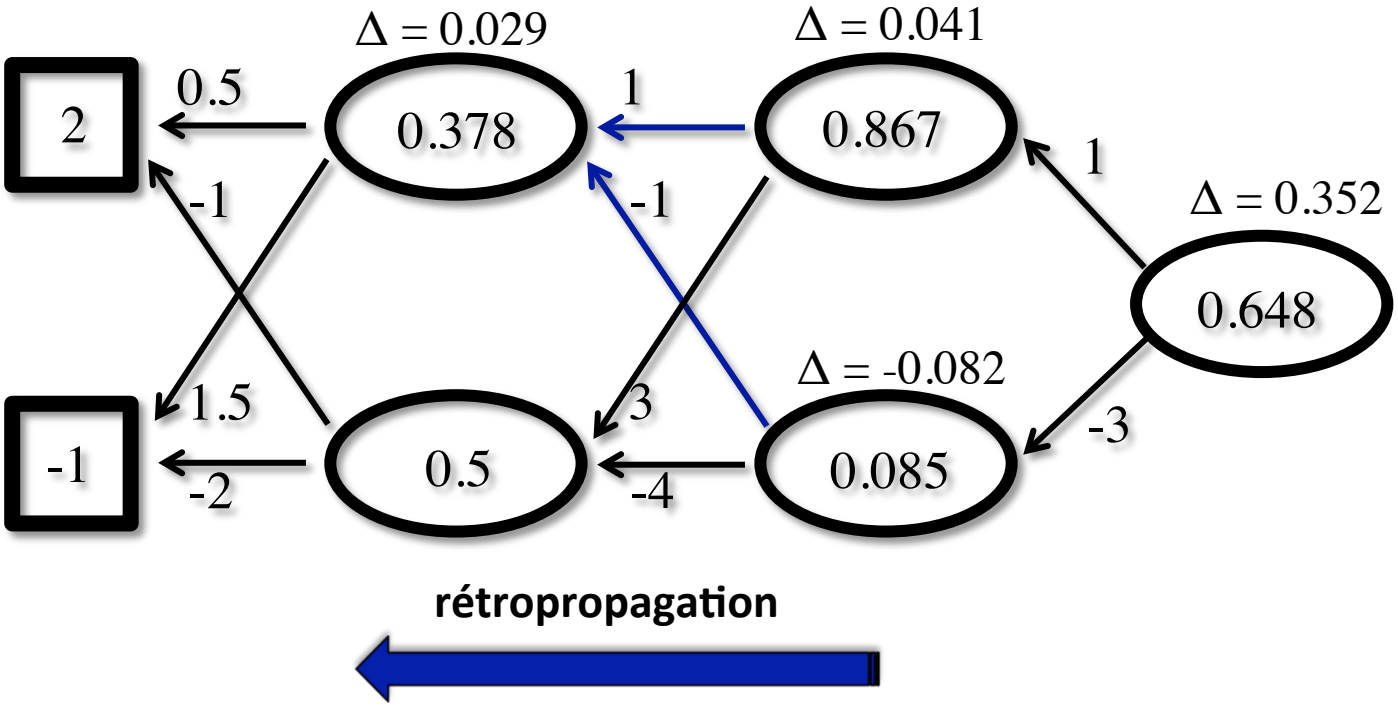
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$\Delta = 0.085 * (1\text{-}0.085) * \text{-}3 * 0.352 = \text{-}0.082$$

Exemple

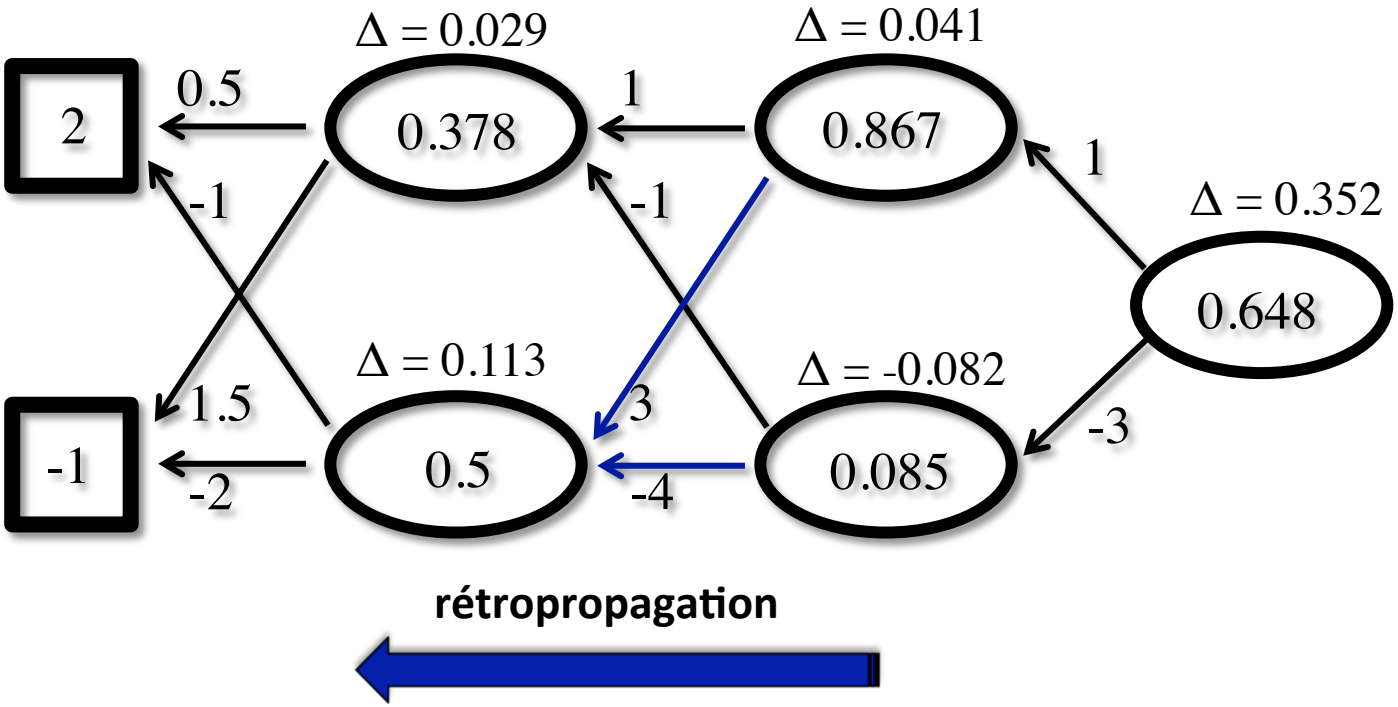
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$\Delta = 0.378 * (1\text{-}0.378) * (1 * 0.041 + \text{-}1 * \text{-}0.082) = 0.029$$

Exemple

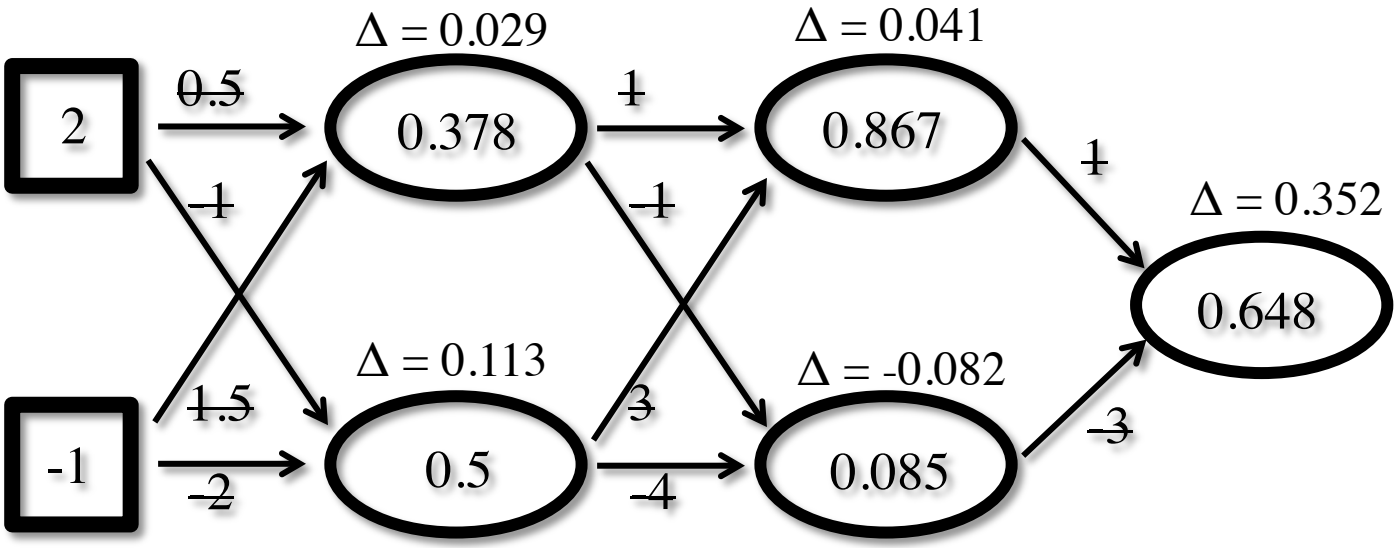
- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



$$\Delta = 0.5 * (1\text{-}0.5) * (3 * 0.041 + \text{-}4 * \text{-}0.082) = 0.113$$

Exemple

- Exemple: $\mathbf{x} = [2, -1]$, $y = 1$



mise à jour (α=0.1)

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

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

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

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

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

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

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

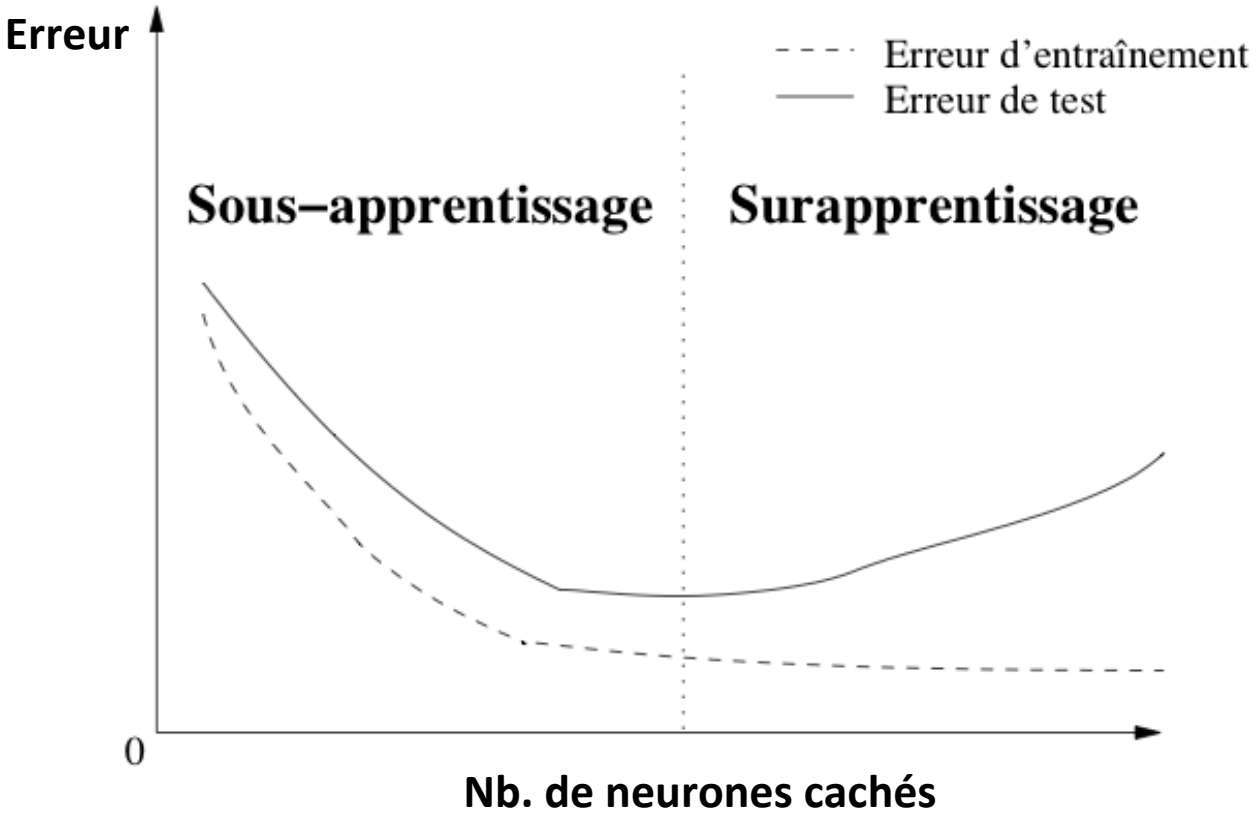
$w_{4,6} \leftarrow -4 + 0.1 * 0.5 * -0.082 = -4.004$

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

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

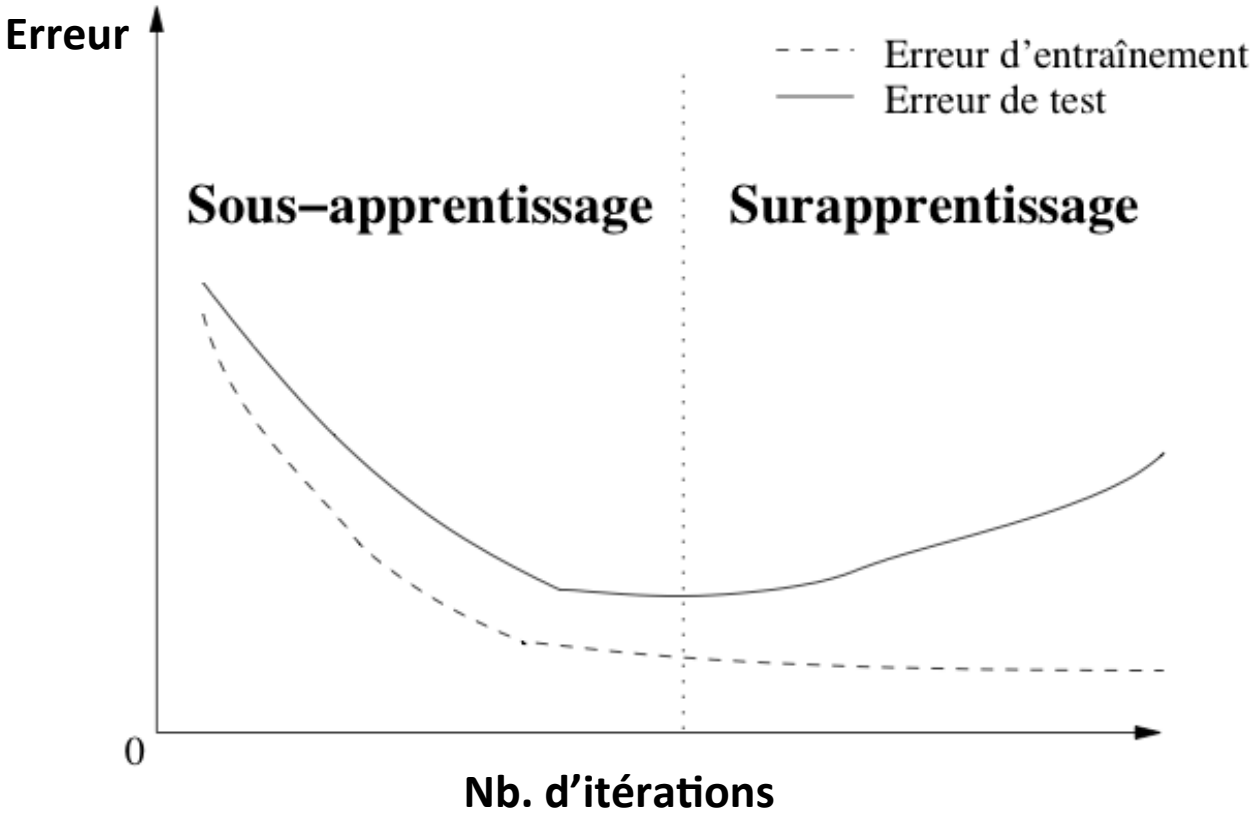
Retour sur la notion de généralisation

- Comment choisir le nombre de neurones cachés?



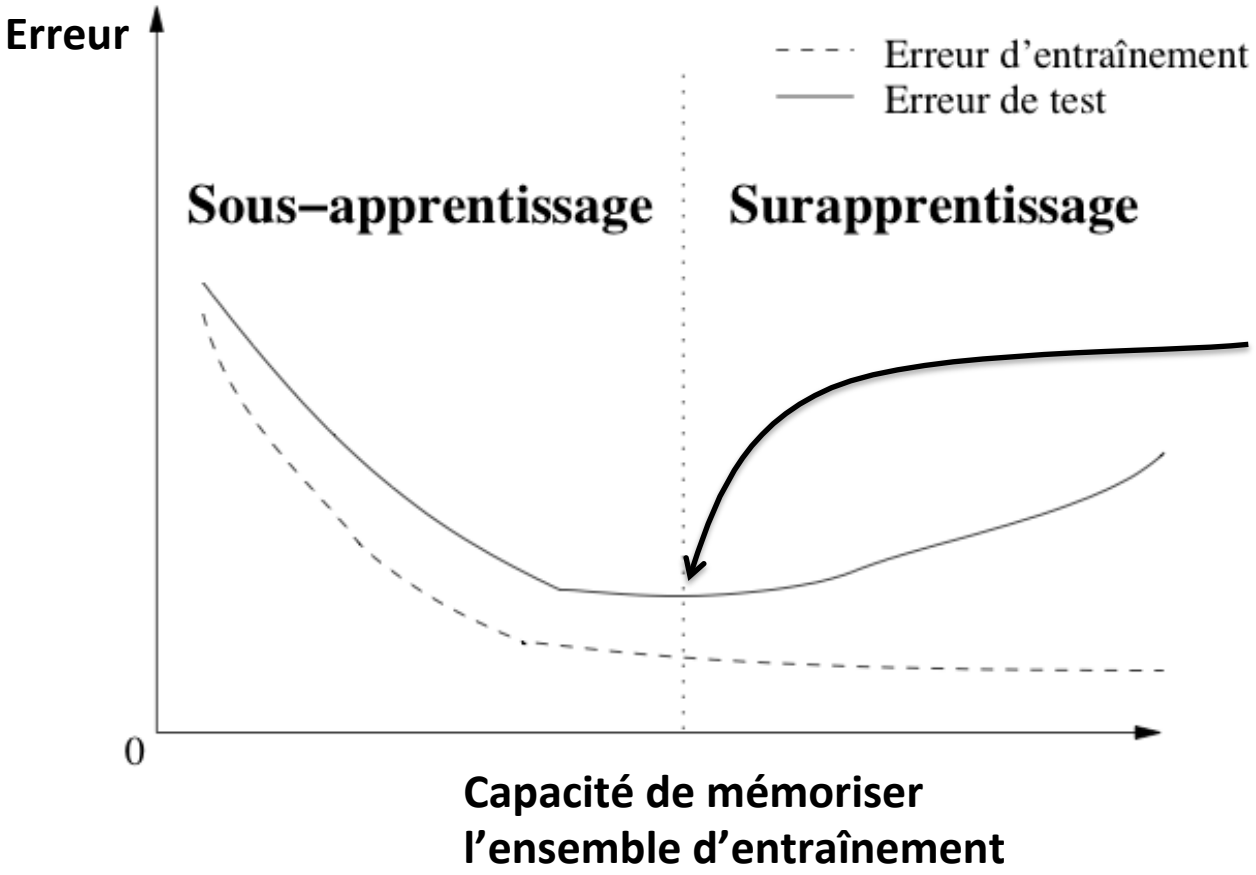
Retour sur la notion de généralisation

- Comment choisir le nombre de neurones cachés?



Retour sur la notion de généralisation

- Comment choisir le nombre de neurones cachés?



On veut trouver ce point, sans sous-apprentissage ni surapprentissage

Hyper-paramètres

- Dans tous les algorithmes d'apprentissage qu'on a vu jusqu'à maintenant, il y avait 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**
  - choisir la valeur qui marche le mieux sur l'ensemble d'entraînement est en général une mauvaise idée (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!**
  - en pratique, on garde un autre ensemble de côté, l'**ensemble de validation**, pour choisir la valeur de ce paramètre
- 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

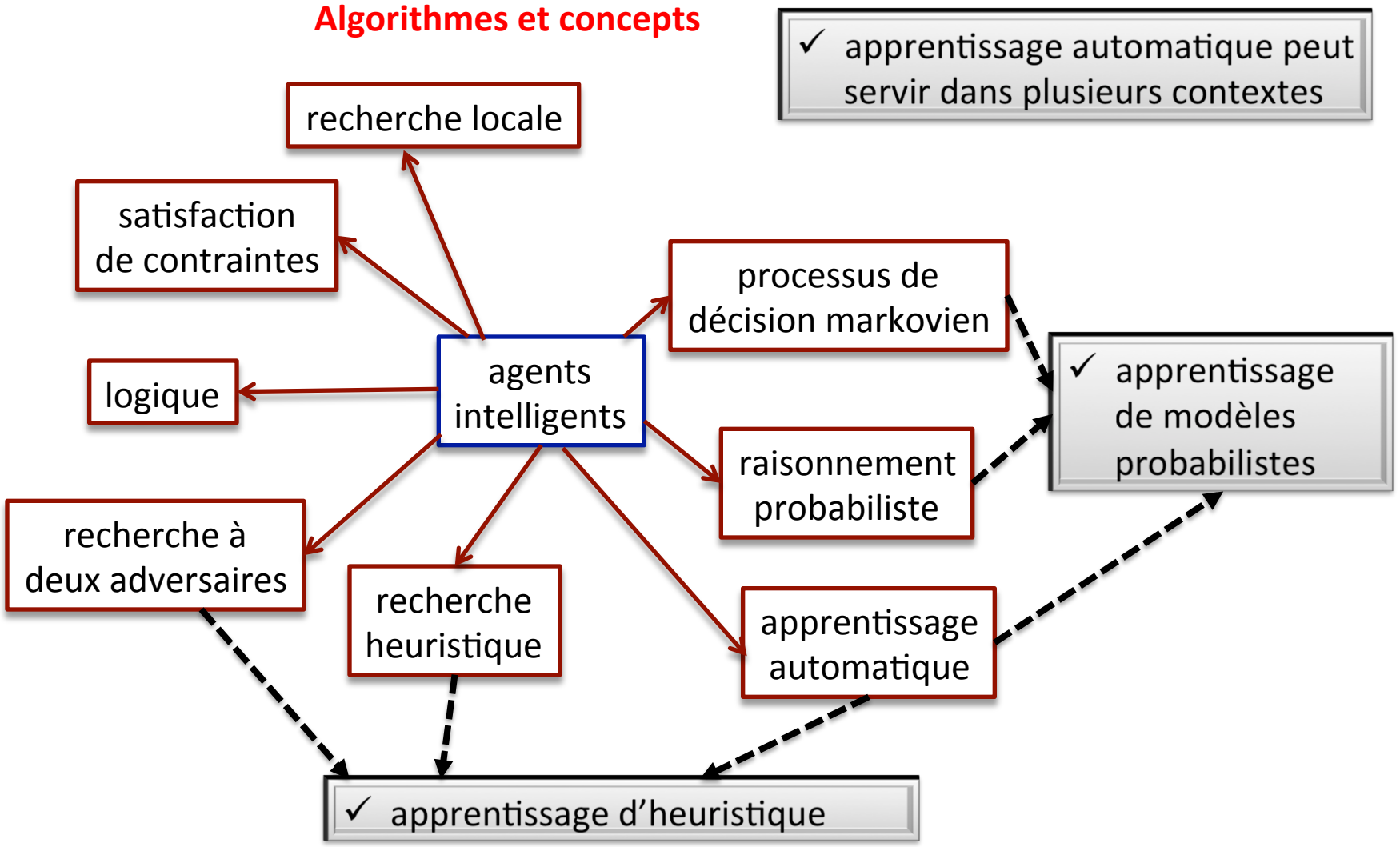
Autres définitions

- **Capacité** d'un modèle h : habilité d'un modèle à réduire son erreur d'entraînement, à mémoriser ces données
- **Modèle paramétrique**: modèle dont la capacité n'augmente pas avec le nombre de données (Perceptron, régression logistique, réseau de neurones avec un **nombre de neurones fixe**)
- **Modèle non-paramétrique**: l'inverse de paramétrique, la capacité augmente avec la taille de l'ensemble d'entraînement (*k* plus proche voisin, réseau de neurones avec un **nombre de neurones adapté aux données d'entraînement**)
- **Époque**: une itération complète sur tous les exemples d'entraînement
- **Fonction d'activation**: fonction non-linéaire $g(\cdot)$ des neurones cachés

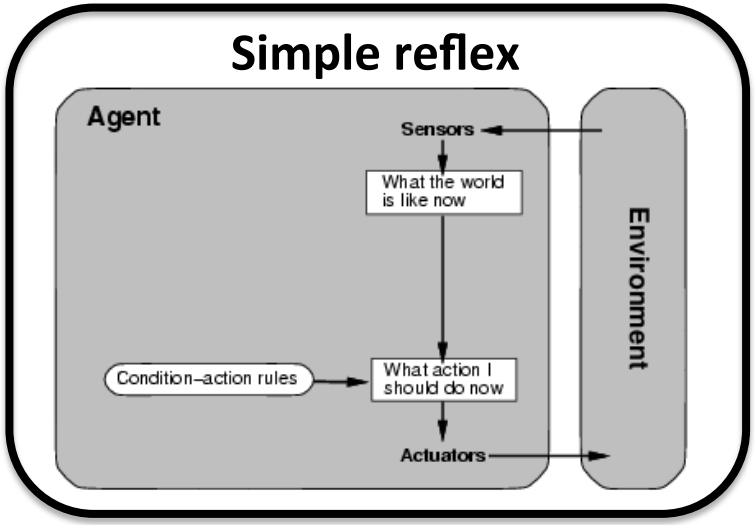
Conclusion

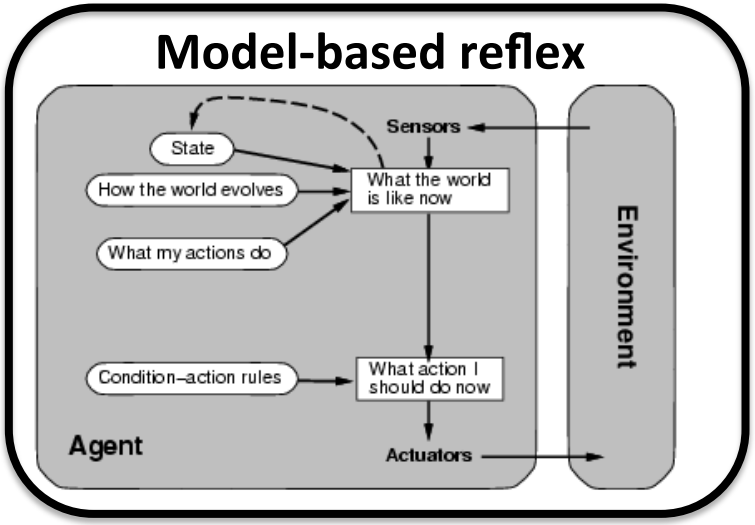
- L'apprentissage automatique permet d'extraire une expertise (humaine) à partir de données
- Nous avons vu le cas spécifique de la classification
  - il existe plusieurs autres problèmes pour lesquels l'apprentissage automatique peut être utile (voir le cours **IFT 603 - Techniques d'apprentissage**)
- L'algorithme des *k* plus proches voisins est simple et puissant (non-linéaire), mais peut être lent avec de grands ensembles de données
- Les algorithmes linéaires du Perceptron et de la régression logistique sont moins puissants mais efficaces
- Les réseaux de neurones artificiel peut avoir la puissance (capacité) d'une classifieur des *k* plus proches voisins, tout en étant plus efficace

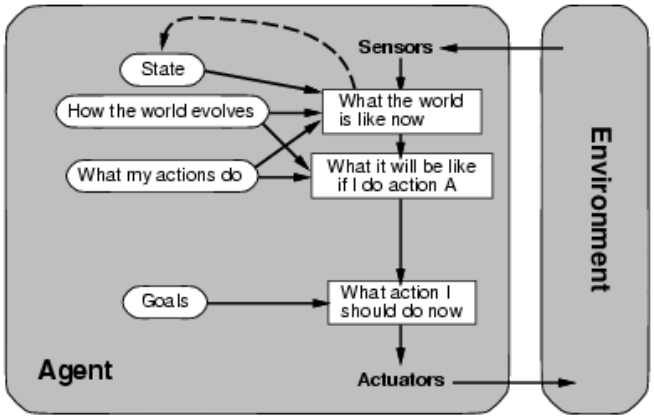
Objectifs du cours

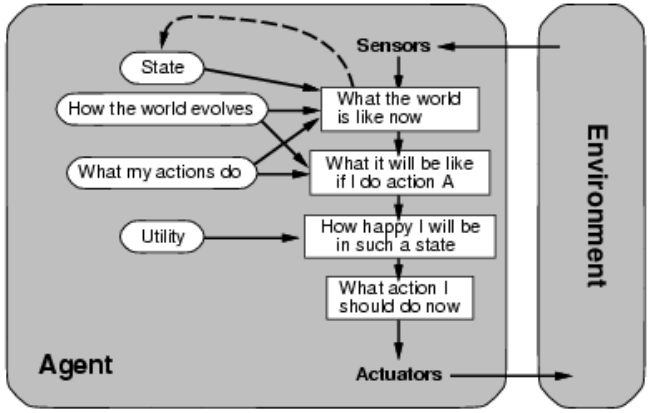


Recherche pour jeux à deux adversaires : pour quel type d'agent?









Vous devriez être capable de...

- Simuler les algorithmes vus
  - *k* plus proches voisins
  - Perceptron
  - régression logistique
  - réseau de neurones
- Calculer une dérivée partielle
- Décrire le développement et l'évaluation (de façon non-biasée) d'un système basé sur un algorithme d'apprentissage automatique
- Comprendre les notions de sous-apprentissage et surapprentissage
- Savoir ce qu'est un hyper-paramètre