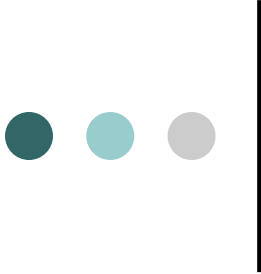
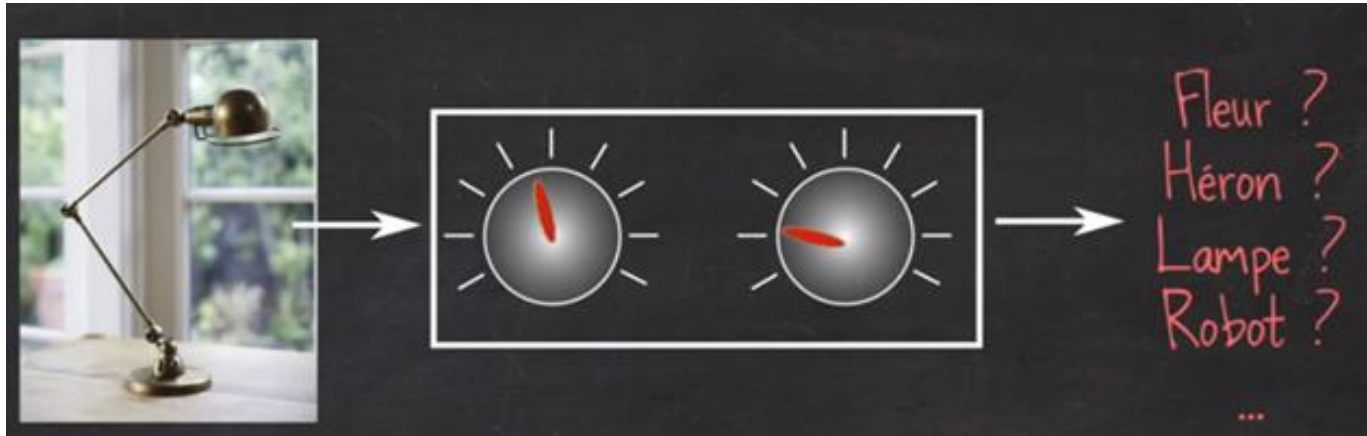


# Les réseaux de neurones artificiels: Des perceptrons aux réseaux convolutifs



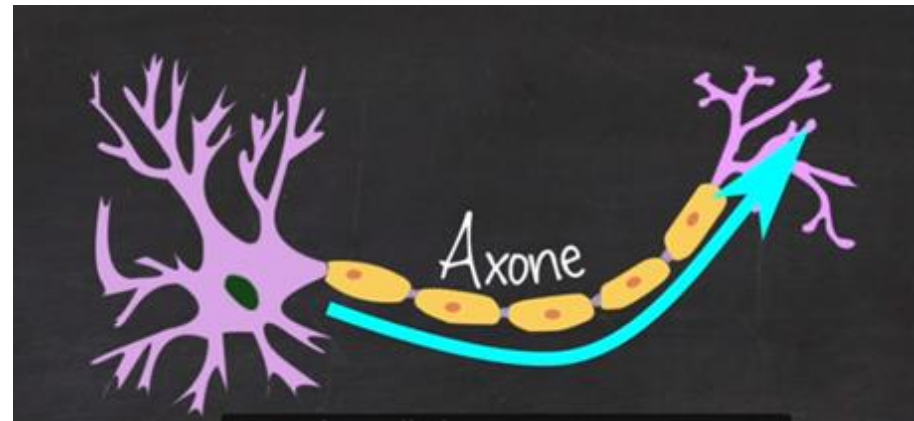
- 
- Introduction
    - Neurone artificiel
    - Fonctions d'activation
  - Classificateurs de type perceptron monocouche
    - Limite du perceptron
    - Apprentissage du Perceptron et convergence
    - L'algorithme d'apprentissage du perceptron
    - Perceptron vs SVM
  - Réseaux de neurones multicouches (MLP)
    - Représentation de XOR
    - Apprentissage d'un MLP: Backpropagation – Rétropropagation
      - Principe
      - Minimisation du risque
      - Descente de gradient
      - Algorithme de Backpropagation
      - Performance de la Backpropagation
      - Exemple
      - Condition d'arrêt
  - Réseaux pour l'apprentissage profond (DL)
    - Principe
    - Réseaux de neurones convolutifs
      - Convolution
      - Pooling

# Reconnaissance d'image



- Une simple régression est largement insuffisante
- Il faut une méthode qui prend en considération plusieurs données en input et capturer des relations complexes entre input et output
- C'est là qu'interviennent les **Réseaux de Neurones**
- Un neurone « artificiel » imite le neurone réel

# Neurone

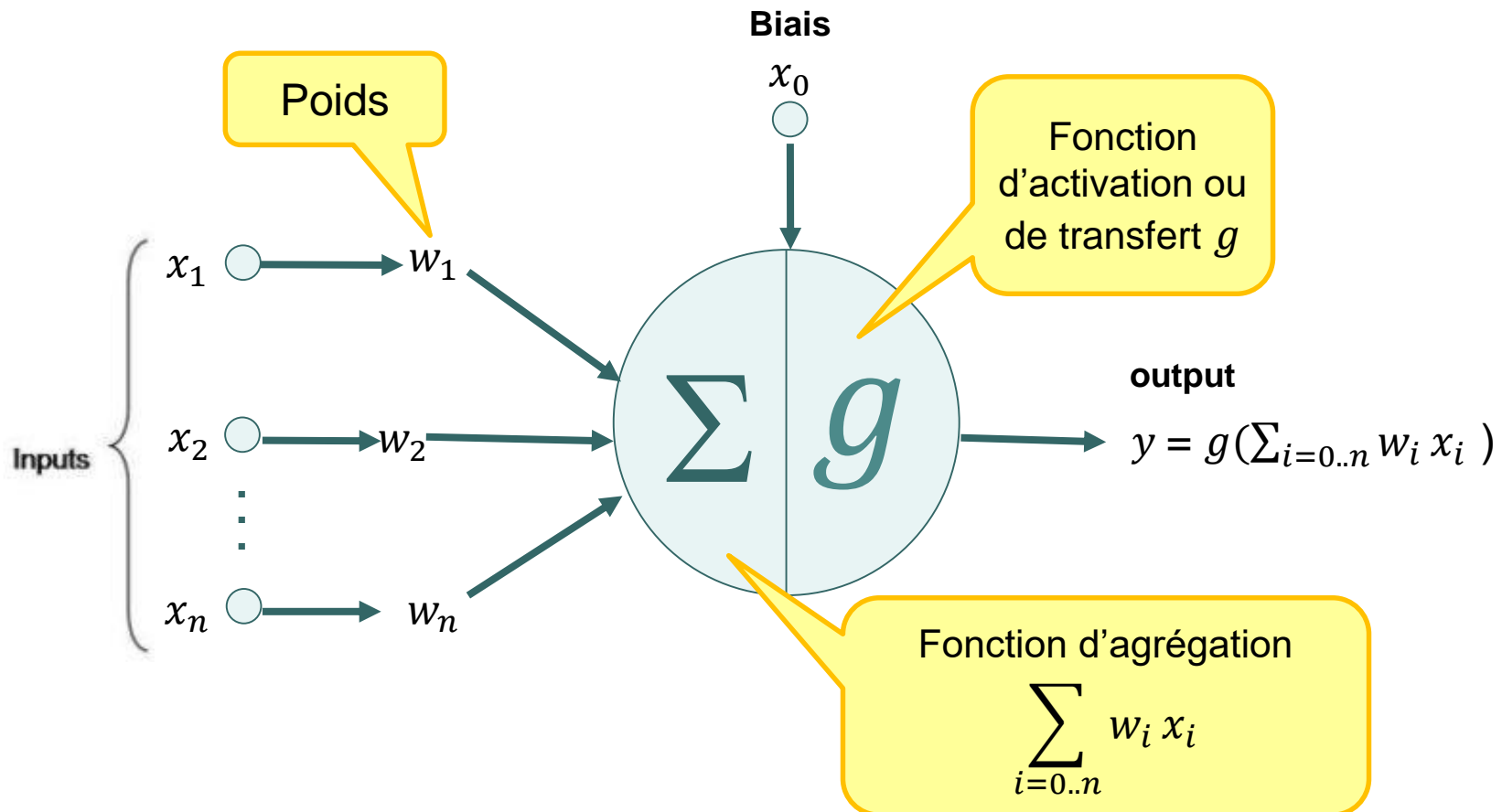


- Les vrais neurones biologiques sont des cellules qu'on trouve dans notre système nerveux et qui sont connectés les uns aux autres par des axones qui permettent d'envoyer les signaux
- Les neurones reçoivent, ou pas un signal électrique des autres neurones qui sont connectés à lui et en fonction de ces signaux :
  - soit il n'envoie rien dans son axone
  - soit il « décharge » c'est-à-dire qu'il envoie un signal

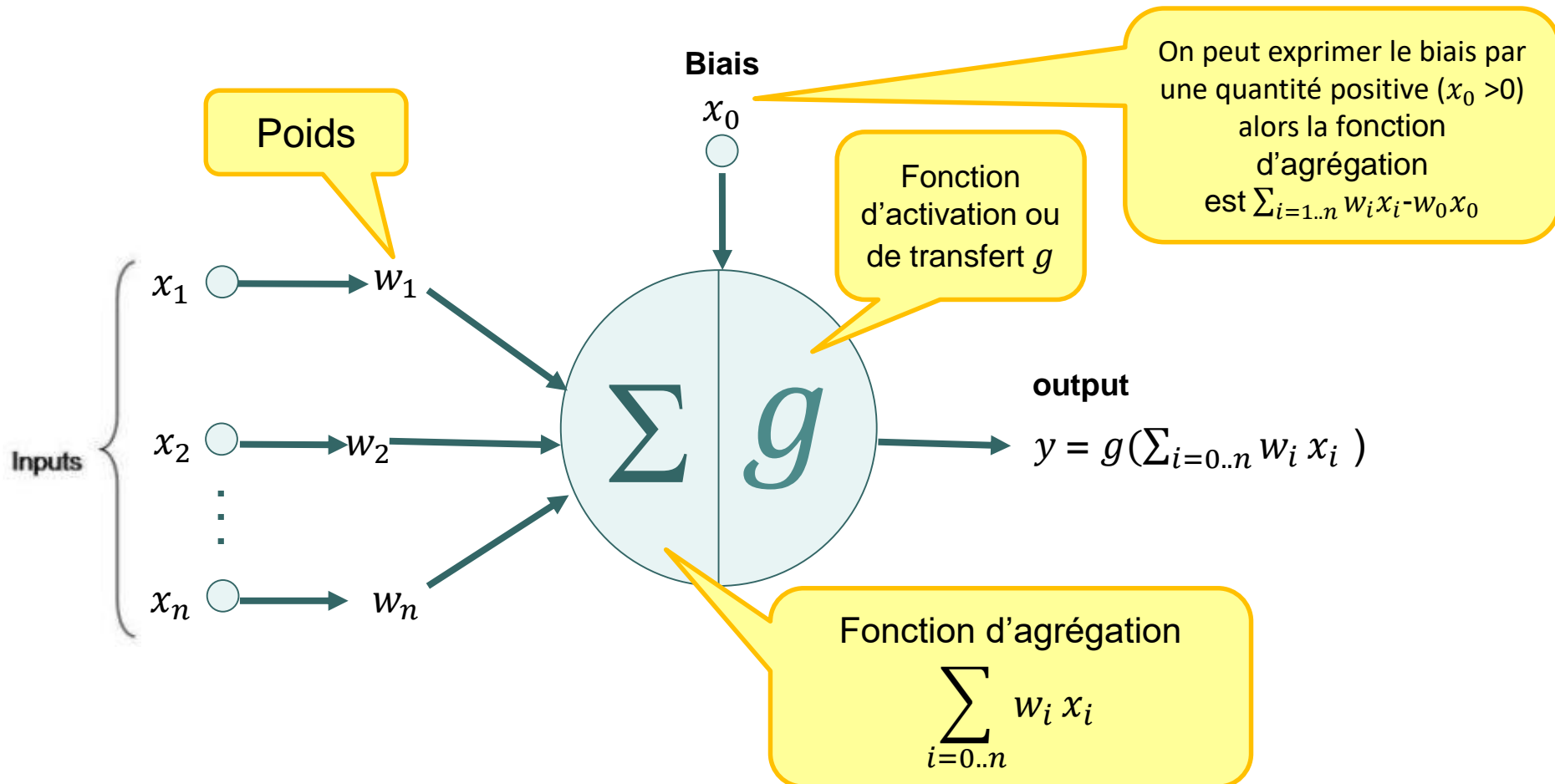


# Neurone artificiel

- L'idée des neurones artificiels est de mimer ce comportement
- C'est en 1943 que Mc Culloch (neurophysiologiste) et Pitts (logicien) ont proposé la première modélisation mathématique de neurone formel.

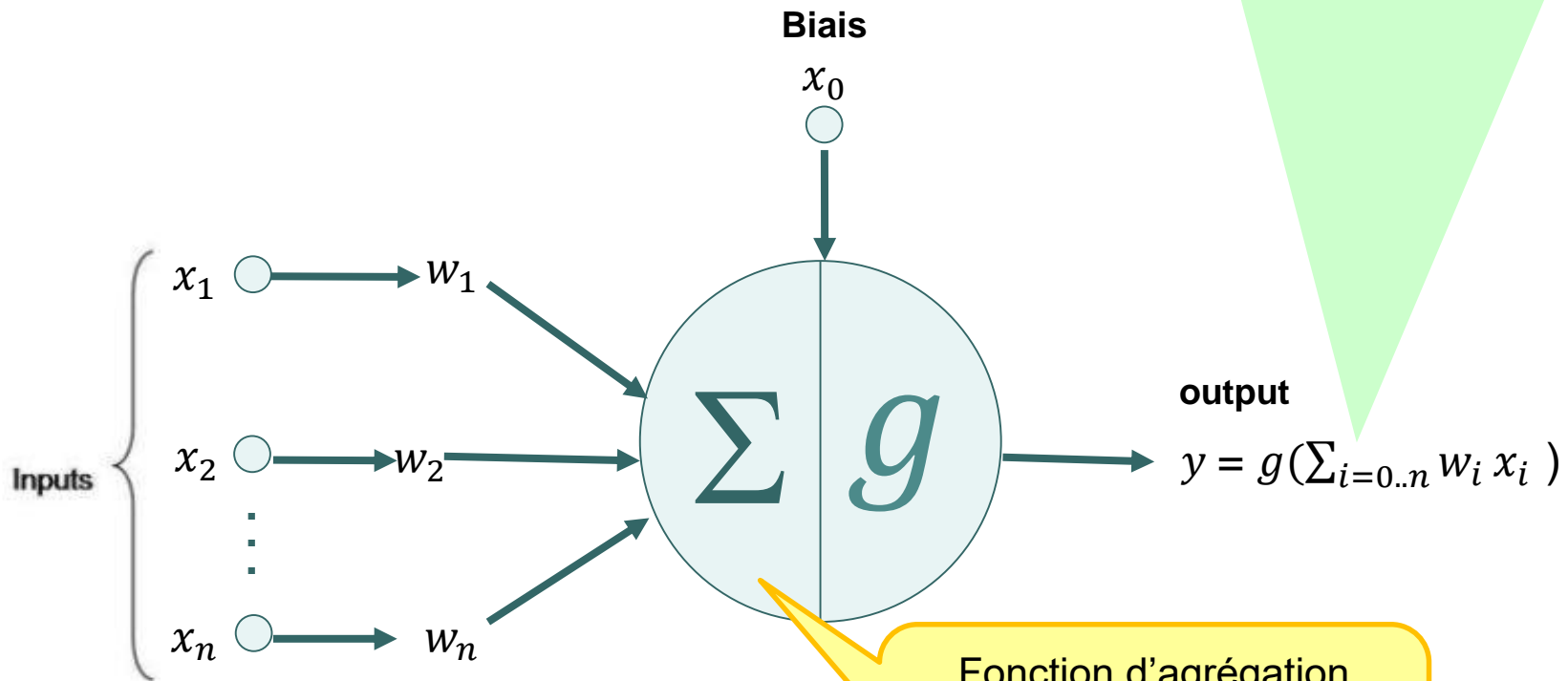


- Formellement un neurone artificiel est défini par :
  - entrées/inputs ( $x_i$ )  $i=1..n$
  - une fonction d'activation (ou de transfert),  $g : \mathbf{R} \rightarrow \mathbf{R}$
  - sortie/output ( $y$ ) %on peut avoir plusieurs sorties
- Un neurone à  $n$  entrées est associé à  $n + 1$  paramètres numériques :
  - $n$  poids synaptiques  $w_i$   $i=1..n$
  - un seuil d'activation du neurone (threshold) ou biais (bias)  $x_0$  ( $x_0 < 0$ )
  - Un neurone à  $n$  entrées calcule la fonction suivante :  $g(\sum_{i=0..n} w_i x_i)$



# Neurone artificiel

Le résultat de la somme pondérée des entrées  $\sum_{i=1..n} w_i x_i$  s'appelle le niveau d'activation du neurone. Lorsque le niveau d'activation atteint ou dépasse le seuil  $x_0$ , alors l'argument de  $g$  cad  $\sum_{i=0..n} w_i x_i$  devient positif (ou nul). Sinon, il est négatif.






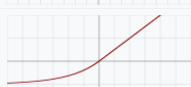
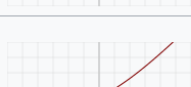


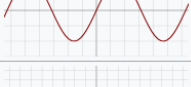
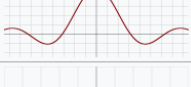


Fonction d'agrégation

$$\sum_{i=0..n} w_i x_i$$

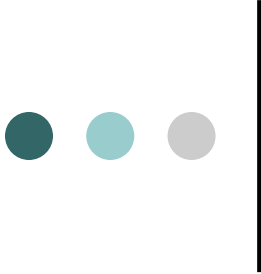
# Fonctions d'activation (ou de transfert)

Les plus  
utilisées

| Nom  | Graphe  | Équation   | Dérivée   |
|--|---|--|---|
| Identité/Rampe   |    | $f(x) = x$   | $f'(x) = 1$   |
| Marche/Heaviside / signe                                       |    | $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$                 | $f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$                                       |
| Logistique (ou marche douce, ou sigmoïde)                      |    | $f(x) = \frac{1}{1 + e^{-x}}$  | $f'(x) = f(x)(1 - f(x))$  |
| Tangente Hyperbolique (TanH)                                   |    | $f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$  | $f'(x) = 1 - f(x)^2$  |
| Arc Tangente (ArcTan ou Tan <sup>-1</sup> )                    |    | $f(x) = \tan^{-1}(x)$  | $f'(x) = \frac{1}{x^2 + 1}$   |
| Unité Exponentielle Linéaire(ELU) <sup>9</sup>                 |    | $f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$   | $f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$                           |
| Unité de Rectification Linéaire Douce (SoftPlus) <sup>10</sup> |    | $f(x) = \log_e(1 + e^x)$   | $f'(x) = \frac{1}{1 + e^{-x}}$  |
| Identité courbée   |   | $f(x) = \frac{\sqrt{x^2 + 1} - 1}{2} + x$  | $f'(x) = \frac{x}{2\sqrt{x^2 + 1}} + 1$   |
| Sinusoïde  |  | $f(x) = \sin(x)$   | $f'(x) = \cos(x)$   |
| Sinus cardinal (Sinc)  |  | $f(x) = \begin{cases} 1 & \text{for } x = 0 \\ \frac{\sin(x)}{x} & \text{for } x \neq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 & \text{for } x = 0 \\ \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} & \text{for } x \neq 0 \end{cases}$ |
| Fonction Gaussienne  |  | $f(x) = e^{-x^2}$  | $f'(x) = -2xe^{-x^2}$   |

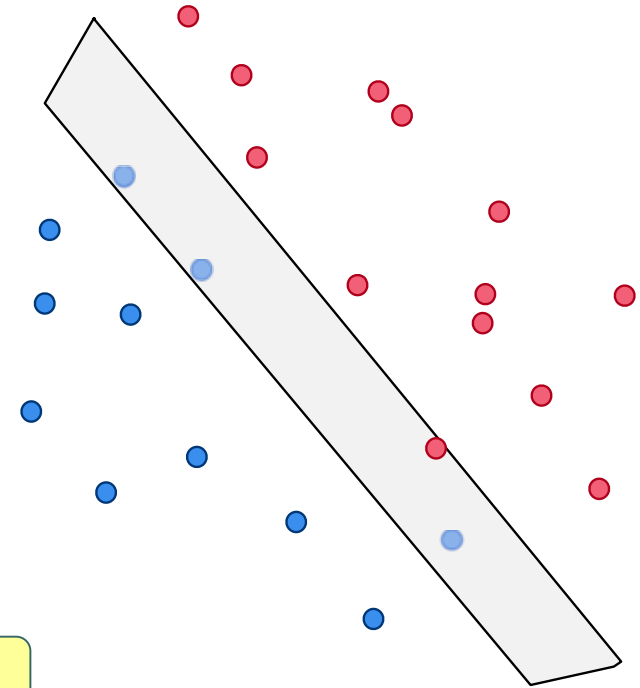
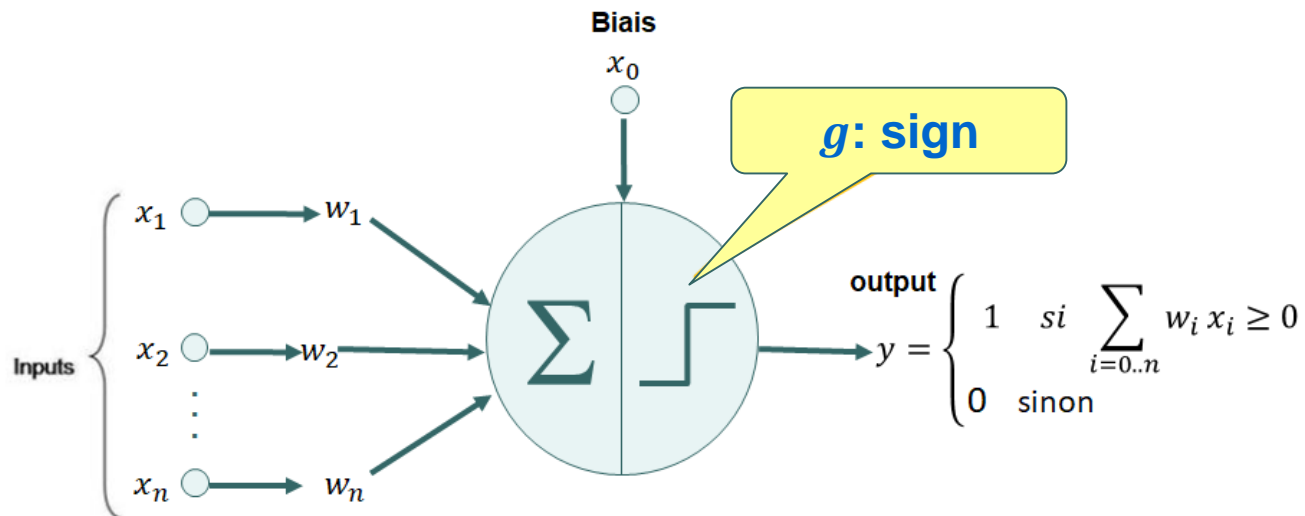
% il existe d'autres fonctions :  
celles-ci c'est es plus courantes  
et les plus citées dans la  
littérature



- 
- Introduction
    - Neurone artificiel
    - Fonctions d'activation
  - Classificateurs de type perceptron monocouche
    - Limites du perceptron
    - Apprentissage du Perceptron et convergence
    - L'algorithme d'apprentissage du perceptron
    - Perceptron vs SVM
  - Réseaux de neurones multicouches (MLP)
    - Représentation de XOR
    - Apprentissage d'un MLP: Backpropagation – Rétropropagation
      - Principe
      - Minimisation du risque
      - Descente de gradient
      - Algorithme de Backpropagation
      - Performance de la Backpropagation
      - Exemple
      - Condition d'arrêt
  - Réseaux pour l'apprentissage profond (DL)
    - Principe
    - Réseaux de neurones convolutifs
      - Convolution
      - Pooling

# Perceptron

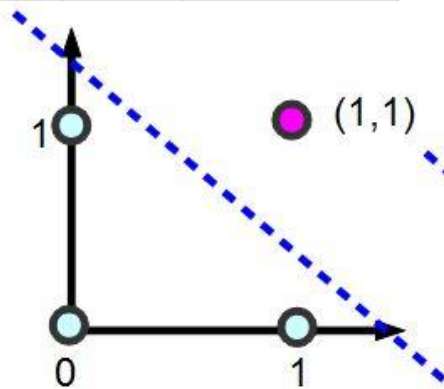
- Le **perceptron** peut être vu comme le type de réseau de neurones le plus simple [Rosenblatt 1957]
- C'est un classifieur linéaire / un discriminant linéaire
- Il s'agit **d'un** neurone formel muni d'une règle d'apprentissage qui permet de déterminer automatiquement les poids synaptiques de manière à séparer un problème d'apprentissage supervisé.
- Dans sa version simplifiée, le perceptron est mono-couche et n'a **qu'une seule sortie** à laquelle toutes les entrées sont connectées et les entrées et la sortie sont booléennes.



# Limite du perceptron

- Le perceptron est incapable de distinguer les patterns non séparables linéairement [Minsky 69]
- Exemple: les portes logiques :

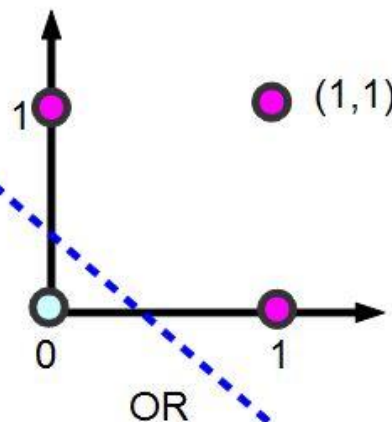
| $x_1$ | $x_2$ | $x_1 \text{ AND } x_2$ |
|-------|-------|------------------------|
| 0     | 0     | 0                      |
| 0     | 1     | 0                      |
| 1     | 0     | 0                      |
| 1     | 1     | 1                      |



AND

linearly separable

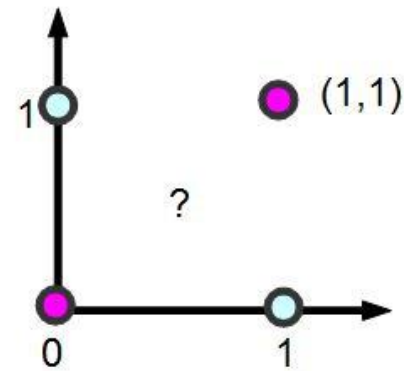
| $x_1$ | $x_2$ | $x_1 \text{ OR } x_2$ |
|-------|-------|-----------------------|
| 0     | 0     | 0                     |
| 0     | 1     | 1                     |
| 1     | 0     | 1                     |
| 1     | 1     | 1                     |



OR

linearly separable

| $x_1$ | $x_2$ | $x_1 \text{ XOR } x_2$ |
|-------|-------|------------------------|
| 0     | 0     | 0                      |
| 0     | 1     | 1                      |
| 1     | 0     | 1                      |
| 1     | 1     | 0                      |



XOR

not linearly separable

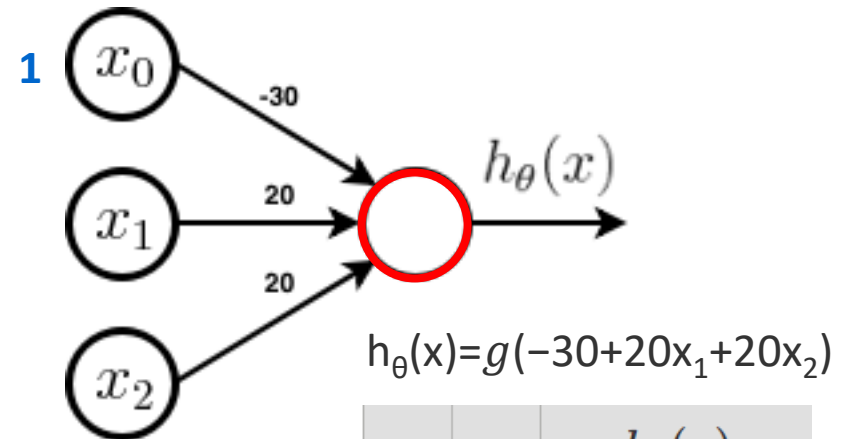
# Représentation des portes logiques

$g$  : sigmoid

## AND Gate

Using a single neuron, it is possible to achieve the approximation of an **AND** gate.

| $x_1$ | $x_2$ | $x_1 \text{ AND } x_2$ |
|-------|-------|------------------------|
| 0     | 0     | 0                      |
| 0     | 1     | 0                      |
| 1     | 0     | 0                      |
| 1     | 1     | 1                      |



| $x_1$ | $x_2$ | $h_{\theta}(x)$    |
|-------|-------|--------------------|
| 0     | 0     | $g(-30) \approx 0$ |
| 0     | 1     | $g(-10) \approx 0$ |
| 1     | 0     | $g(-10) \approx 0$ |
| 1     | 1     | $g(10) \approx 1$  |

$$\begin{aligned} w_1 \times 0 + w_2 \times 0 + w_0 &< 0 \\ w_1 \times 0 + w_2 \times 1 + w_0 &< 0 \\ w_1 \times 1 + w_2 \times 0 + w_0 &< 0 \\ w_1 \times 1 + w_2 \times 1 + w_0 &\geq 0 \end{aligned}$$



$$\begin{aligned} w_0 &< 0 \\ w_2 &< -w_0 \\ w_1 &< -w_0 \\ w_1 + w_2 &\geq -w_0 \end{aligned}$$

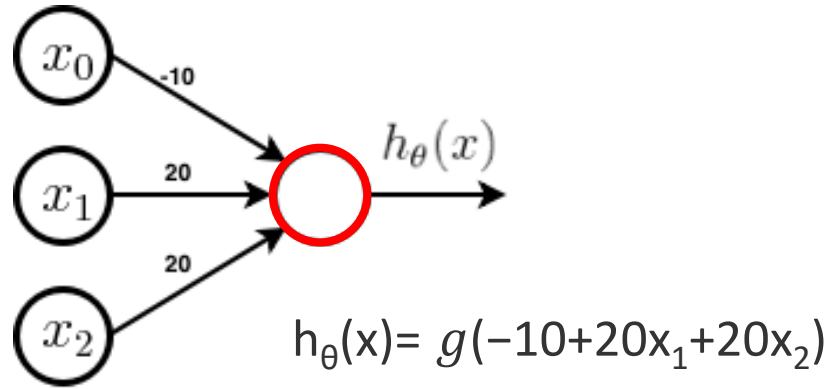
$$\begin{aligned} w_0 &= -30 \\ w_1 &= 20 \\ w_2 &= 20 \end{aligned}$$

ou

$$\begin{aligned} w_0 &= -20 \\ w_1 &= 10 \\ w_2 &= 10 \end{aligned}$$

1

OR Gate



$$\begin{aligned} w_1 \times 0 + w_2 \times 0 + w_0 &< 0 \\ w_1 \times 0 + w_2 \times 1 + w_0 &\geq 0 \\ w_1 \times 1 + w_2 \times 0 + w_0 &\geq 0 \\ w_1 \times 1 + w_2 \times 1 + w_0 &\geq 0 \end{aligned}$$

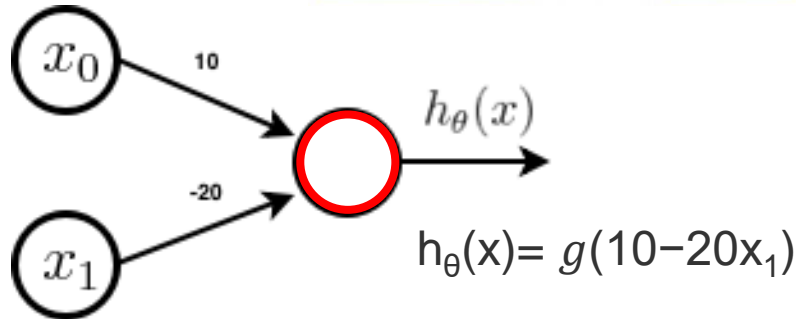


$$\begin{aligned} w_0 &< 0 \\ w_2 &\geq -w_0 \\ w_1 &\geq -w_0 \\ w_1 + w_2 &\geq -w_0 \end{aligned}$$

| $x_1$ | $x_2$ | $h_{\theta}(x)$    |
|-------|-------|--------------------|
| 0     | 0     | $g(-10) \approx 0$ |
| 0     | 1     | $g(10) \approx 1$  |
| 1     | 0     | $g(10) \approx 1$  |
| 1     | 1     | $g(30) \approx 1$  |

1

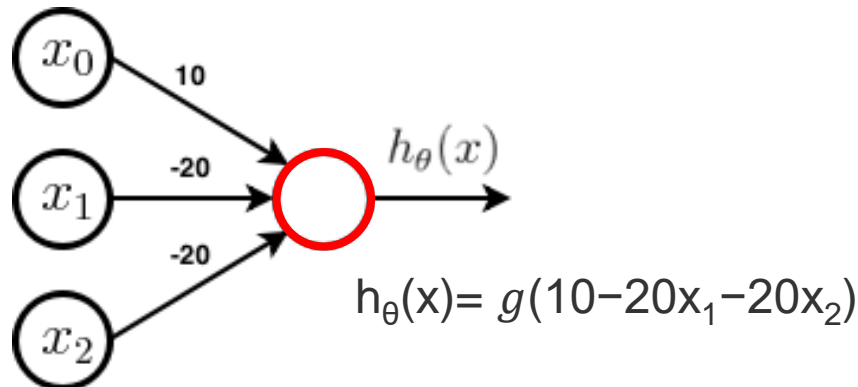
NOT Gate



| $x_1$ | $h_{\theta}(x)$    |
|-------|--------------------|
| 0     | $g(10) \approx 1$  |
| 1     | $g(-10) \approx 0$ |

(NOT  $x_1$ ) AND (NOT  $x_2$ )

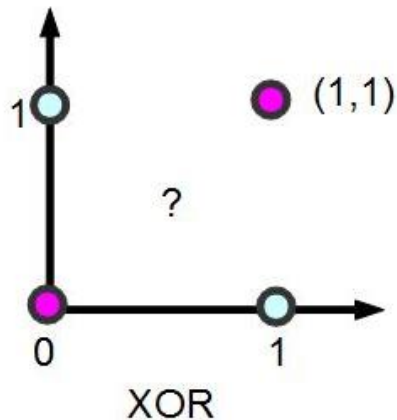
1



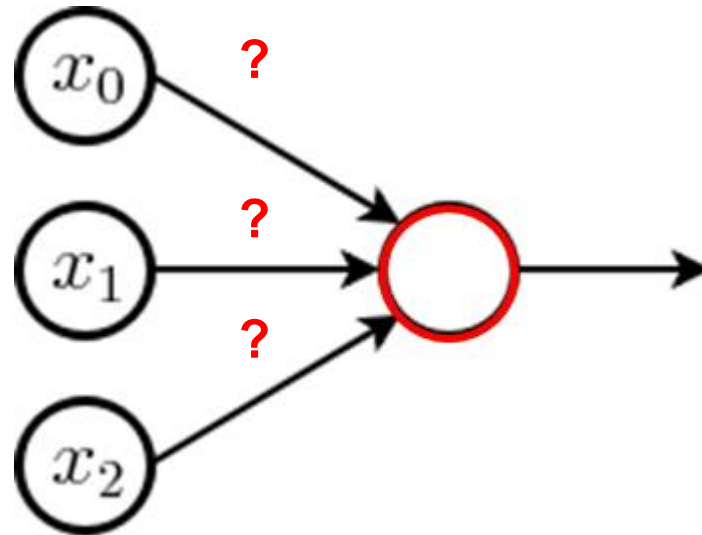
| $x_1$ | $x_2$ | $h_{\theta}(x)$    |
|-------|-------|--------------------|
| 0     | 0     | $g(10) \approx 1$  |
| 0     | 1     | $g(-10) \approx 0$ |
| 1     | 0     | $g(-10) \approx 0$ |
| 1     | 1     | $g(-30) \approx 0$ |

# Représentation du XOR

| $x_1$ | $x_2$ | $x_1 \text{ XOR } x_2$ |
|-------|-------|------------------------|
| 0     | 0     | 0                      |
| 0     | 1     | 1                      |
| 1     | 0     | 1                      |
| 1     | 1     | 0                      |



not linearly seperable



$$\begin{aligned}
 w_1 \times 0 + w_2 \times 0 + w_0 &< 0 \\
 w_1 \times 0 + w_2 \times 1 + w_0 &\geq 0 \\
 w_1 \times 1 + w_2 \times 0 + w_0 &\geq 0 \\
 w_1 \times 1 + w_2 \times 1 + w_0 &< 0
 \end{aligned}$$



$$\begin{aligned}
 w_0 &< 0 \\
 w_2 &\geq -w_0 \\
 w_1 &\geq -w_0 \\
 w_1 + w_2 &< -w_0
 \end{aligned}$$

2eme et 3eme inégalités sont incompatibles avec la 4eme :  
Impossible de trouver les poids pour modéliser XOR



# Apprentissage du Perceptron et convergence

- Apprendre le perceptron = Apprendre les poids des connexions  $w_i$
- A chaque phase de l'apprentissage les poids sont MAJ jusqu'à convergence (les exemples sont bien classés)

## PERCEPTRON CONVERGENCE THEOREM:

The perceptron learning algorithm will always find weights to classify the inputs if such a set of weights exists.

Minsky and Papert show that such weights exist iff the problem is linearly separable

# The perceptron learning algorithm: intuition

%  $\hat{y}$  : résultat du perceptron  
%  $y$  : le vrai output

Le perceptron se trompe dans deux cas

**Cas 1:  $\hat{y} = 0$  et  $y = 1$  cad  $(\sum_i w_i x_i) < 0$**

→ On voudrait que  $\sum_i w_i x_i$  soit plus grand on doit donc augmenter  $w_i x_i$

- Si l'entrée à laquelle est connectée  $w_i$  est positive cad  $x_i > 0$  : on doit augmenter  $w_i$   
 $(y - \hat{y})x_i = x_i > 0$  donc  $w_i$  va augmenter quand on lui ajoute  $\alpha (y - \hat{y})x_i$
- Si l'entrée à laquelle est connectée  $w_i$  est négative cad  $x_i < 0$  : on doit diminuer  $w_i$   
 $(y - \hat{y})x_i = x_i < 0$  donc  $w_i$  va diminuer quand on lui ajoute  $\alpha (y - \hat{y})x_i$

%  $\alpha$  : taux d'apprentissage  $> 0$



# The perceptron learning algorithm: intuition

Le perceptron se trompe dans deux cas

**Cas 1:**  $\hat{y} = 0$  et  $y = 1$  cad  $(\sum_i w_i x_i) < 0$

→ On voudrait que  $\sum_i w_i x_i$  soit plus grand on doit donc augmenter  $w_i x_i$

- Si l'entrée à laquelle est connectée  $w_i$  est positive cad  $x_i > 0$  : on doit augmenter  $w_i$   
 $(y - \hat{y})x_i > 0$  donc  $w_i$  va augmenter quand on lui ajoute  $\alpha (y - \hat{y})x_i$
- Si l'entrée à laquelle est connectée  $w_i$  est négative cad  $x_i < 0$  : on doit diminuer  $w_i$   
 $(y - \hat{y})x_i < 0$  donc  $w_i$  va diminuer quand on lui ajoute  $\alpha (y - \hat{y})x_i$

**Cas 2:**  $\hat{y} = 1$  et  $y = 0$  cad  $(\sum_i w_i x_i) > 0$

→ On voudrait que  $\sum_i w_i x_i$  soit plus petit on doit donc diminuer les  $w_i x_i$

- Si l'entrée à laquelle est connectée  $w_i$  est positive cad  $x_i > 0$  : on doit diminuer  $w_i$   
 $(y - \hat{y})x_i < 0$  donc  $w_i$  va diminuer quand on lui ajoute  $\alpha (y - \hat{y})x_i$
- Si l'entrée à laquelle est connectée  $w_i$  est négative cad  $x_i < 0$  : on doit augmenter  $w_i$   
 $(y - \hat{y})x_i > 0$  donc  $w_i$  va augmenter quand lui ajoute  $\alpha (y - \hat{y})x_i$

Règle  
d'apprentissage  
du perceptron

$$w_i \leftarrow w_i + \alpha (y - \hat{y})x_i$$

Il existe d'autres  
règles comme la  
règle de Hebb



# Taux d'apprentissage

- $\alpha$  : taux d'apprentissage: contrôle la vitesse avec laquelle on voudrait que le processus d'apprentissage se déroule (contrôle l'amplitude des pas utilisés pour ajuster les poids).
- Le taux d'apprentissage  $\alpha$  est compris généralement entre 0 et 1.
  - Si  $\alpha$  est proche de 1: les poids vont varier avec une assez grande amplitude,
    - si  $\alpha$  est trop proche de 1, on pourrait dans de nombreux cas faire osciller les poids sans jamais converger
  - Si  $\alpha$  est éloigné de 1: les poids vont varier petit à petit (sans une grosse variation d'amplitude)
    - si  $\alpha$  est trop éloigné de 1, les poids varieront tellement doucement qu'ils risquent de ne jamais converger vers de bonnes valeurs

% généralement on prend  $\alpha = 0.7$  ou  $0.8$

- **Remarque:** Il existe d'autres hyperparamètres comme le momentum : (un terme d'inertie) qui permet de prendre en considération les erreurs passées au fil des itérations pour aider l'algorithme à sortir de ces minimums locaux.

# The perceptron learning algorithm

**Function** PERCEPTRON-LEARNING (*examples*, *network*,  $\alpha$ ) returns a perceptron

**Inputs:** *examples*: a set of examples (linearly separable), each with input vector  $\mathbf{x}$  and output vector  $\mathbf{y}$

*network*: weights  $w_i$ , activation function  $g$

$\alpha$ : learning rate

**for each** weight  $w_i$  in network **do**

$w_i \leftarrow$  a small random number

Initialiser les poids à de  
petites valeurs  
aléatoires

**Repeat**

**for each** example  $(\mathbf{x}, \mathbf{y})$  in examples **do**

$\hat{y} \leftarrow g(\sum_i w_i x_i)$

**If**  $\hat{y} \neq y$  **then**

**for each**  $i$  **do**

$w_i \leftarrow w_i + \alpha (y - \hat{y}) x_i$

**Until** convergence

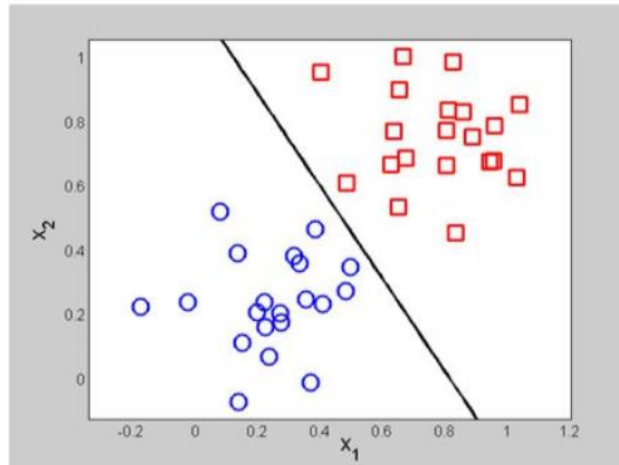
**Return** Perceptron

Epoch: cycle through the examples

# Perceptron vs SVM

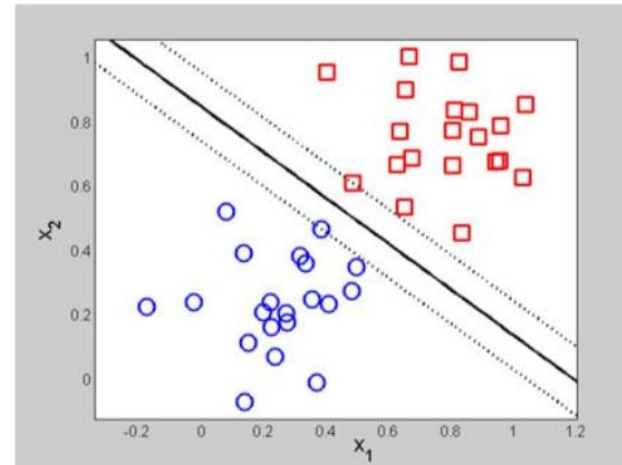
- Les perceptrons ont un principe proche des machines à vecteurs de support (Support Vector Machine, SVM) [Vladimir Vapnik ,1990]
- Le SVM, tente de maximiser le "vecteur de support", c'est-à-dire la distance entre la frontière de séparation et les exemples les plus proches.
- Le Perceptron n'essaie pas d'optimiser la distance de séparation. Tant qu'il trouve un hyperplan qui sépare les exemples (les 1 et les 0)
- Les SVM ont une assise théorique solide

**Perceptron:**

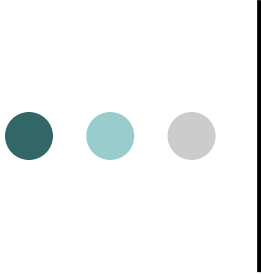


**Minimizes least square error  
(gradient descent)**

**SVM:**



**Maximizes margin**

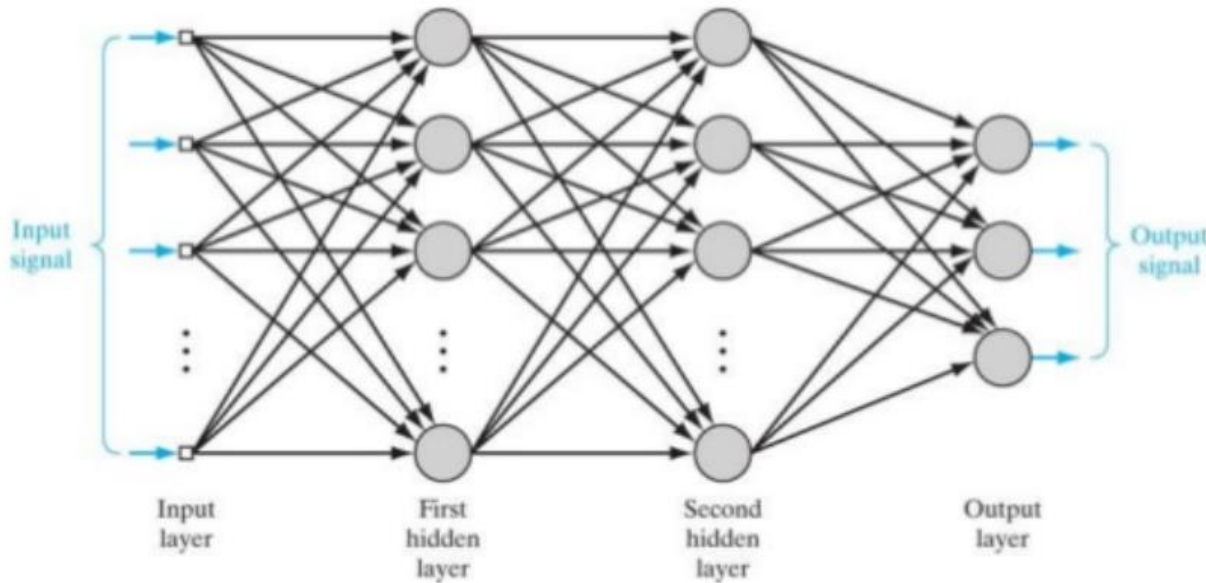
- 
- Introduction
    - Neurone artificiel
    - Fonctions d'activation
  - Classificateurs de type perceptron monocouche
    - Limite du perceptron
    - Apprentissage du Perceptron et convergence
    - L'algorithme d'apprentissage du perceptron
    - Perceptron vs SVM
  - Réseaux de neurones multicouches (MLP)
    - Représentation de XOR
    - Apprentissage d'un MLP: Backpropagation – Rétropropagation
      - Principe
      - Minimisation du risque
      - Descente de gradient
      - Algorithme de Backpropagation
      - Performance de la Backpropagation
      - Exemple
      - Condition d'arrêt
  - Réseaux pour l'apprentissage profond (DL)
    - Principe
    - Réseaux de neurones convolutifs
      - Convolution
      - Pooling



# Les réseaux de neurones artificiels

- Un seul neurone seul ne suffit pas pour modéliser les relations complexes
- Combiner plusieurs neurones:
  - = Réseaux de neurones artificiels
  - = Perceptrons Multi-couches (PMC)
  - = Multilayer Perceptron (MLP)

# Multilayer Perceptron (MLP)



3 types de couches (layers)

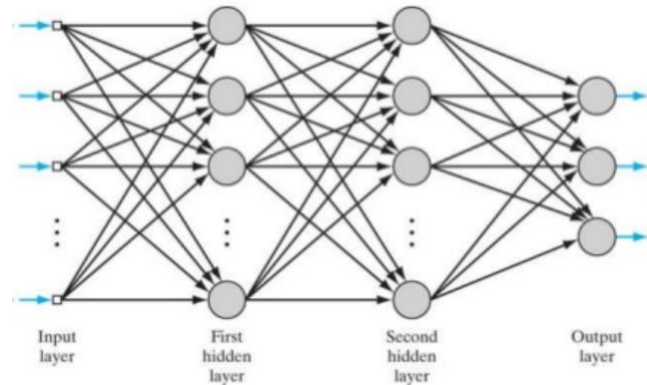
- 1 couche d'entrée
- 1 ou ++ couches cachées
- 1 couche sortie

- Une couche : un groupe de neurones uniformes sans connexion les uns avec les autres
- Les neurones de la couche  $i$  servent d'entrées aux neurones de la couche  $i + 1$  (structure sans cycle : une couche ne peut utiliser que les sorties des couches précédentes)
- Toutes les connections  $i \rightarrow j$  sont pondérées par le poids  $w_{ji}$
- Il réalise une transformation vectorielle :
  - une couche reçoit un vecteur d'entrée et le transforme en vecteur de sortie
  - une couche au moins n'est pas linéaire
  - les dimensions d'entrée et de sortie peuvent être différentes
- Les MLP sont très polyvalents on peut les adapter à ++ entrées et ++ sorties

# ● ● ● | Multilayer Perceptron (MLP)

PHASE  
D'APPRENTISSAGE

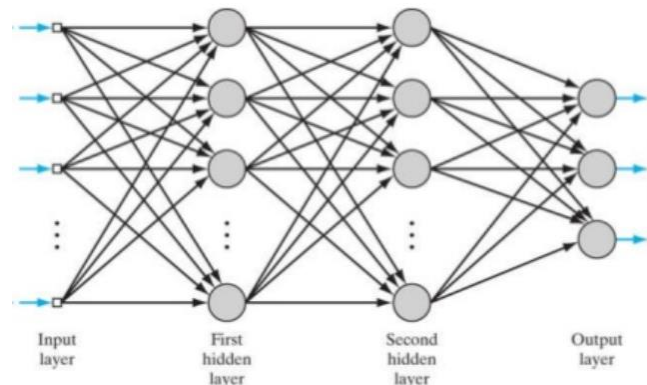
Entrés X



Sorties Y

PHASE DE  
PRÉDICTION

Nouvel exemple X



Prédiction Y





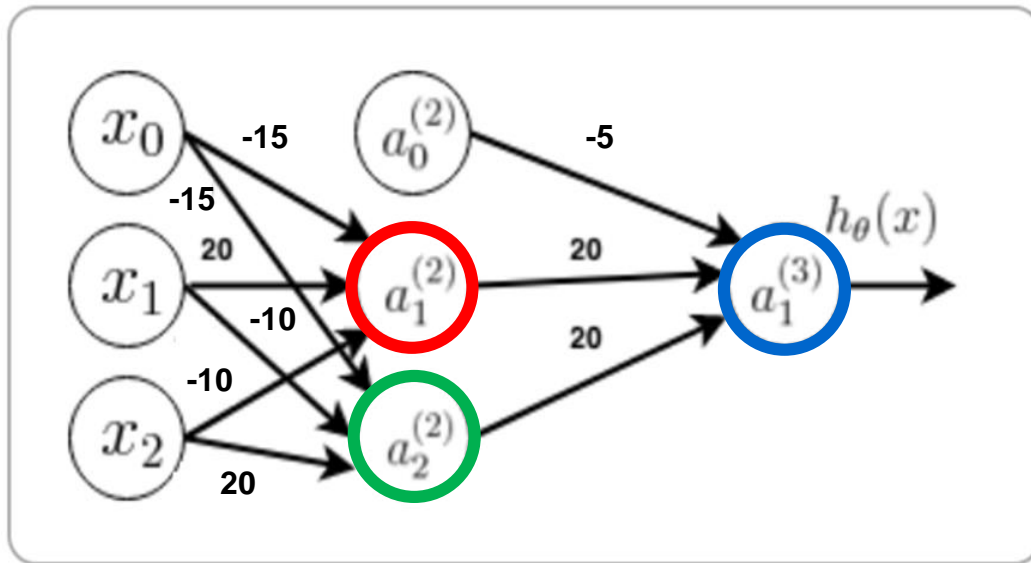
# Représentation de XOR

**XOR Gate**

$(\text{NON}(x_1) \text{ ET } x_2) \text{ OU } (x_1 \text{ ET } (\text{NON}(x_2)))$

# Représentation de XOR

## XOR Gate



| $x_1$ | $x_2$ | $a_1^{(2)}$ | $a_2^{(2)}$ | $h_\theta(x)$ |
|-------|-------|-------------|-------------|---------------|
| 0     | 0     | 0           | 0           | 0             |
| 0     | 1     | 1           | 0           | 1             |
| 1     | 0     | 0           | 1           | 1             |
| 1     | 1     | 0           | 0           | 0             |

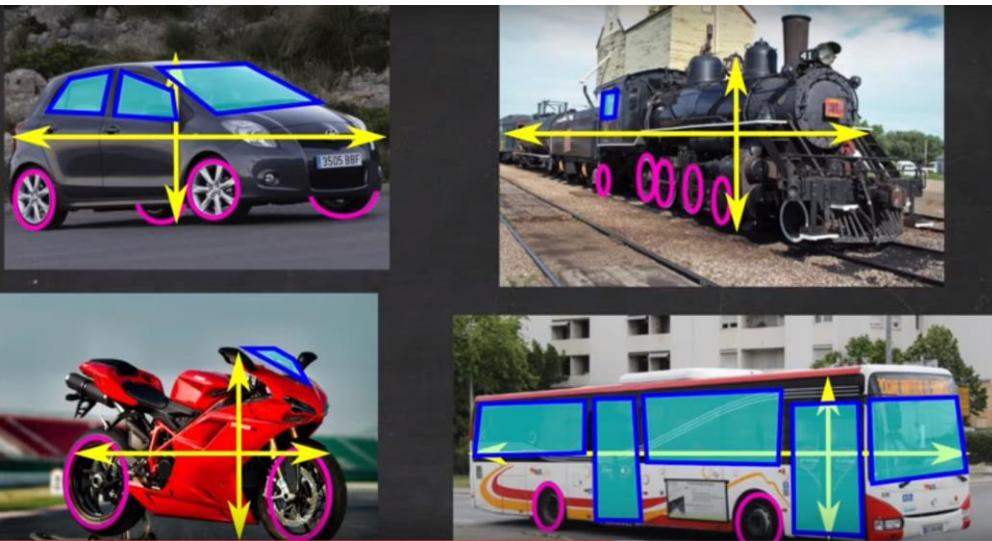
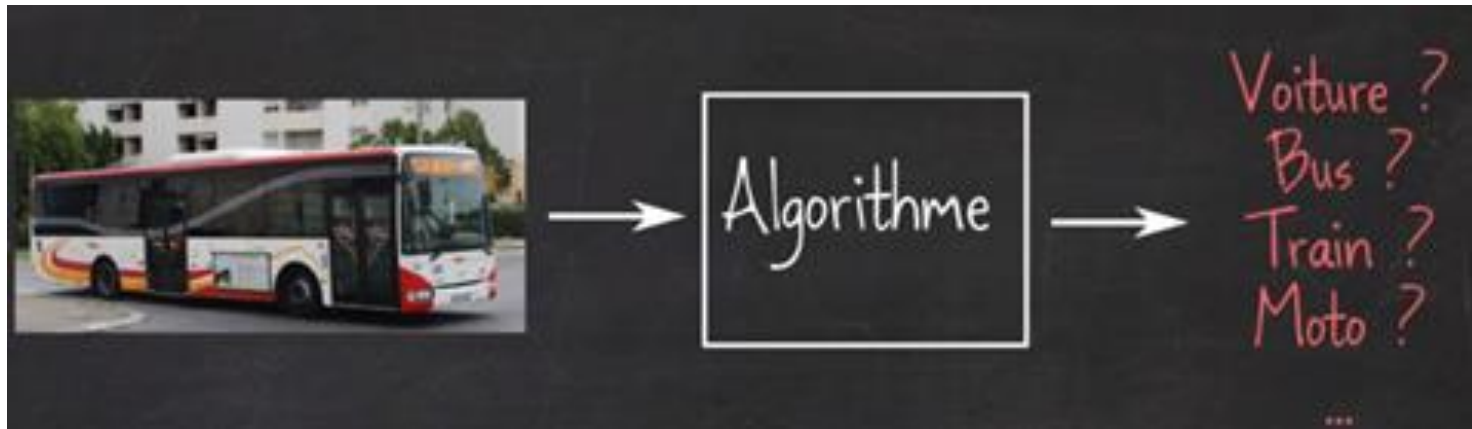
(NON( $x_1$ ) ET  $x_2$ ) OU ( $x_1$  ET (NON( $x_2$ )))

$a_1^{(2)}$  : (NOT( $x_1$ ) ET  $x_2$ )

$a_2^{(2)}$  : ( $x_1$  ET (NON( $x_2$ )))

$a_1^{(3)}$  (output neuron) : OR gate

# Reconnaissance d'images



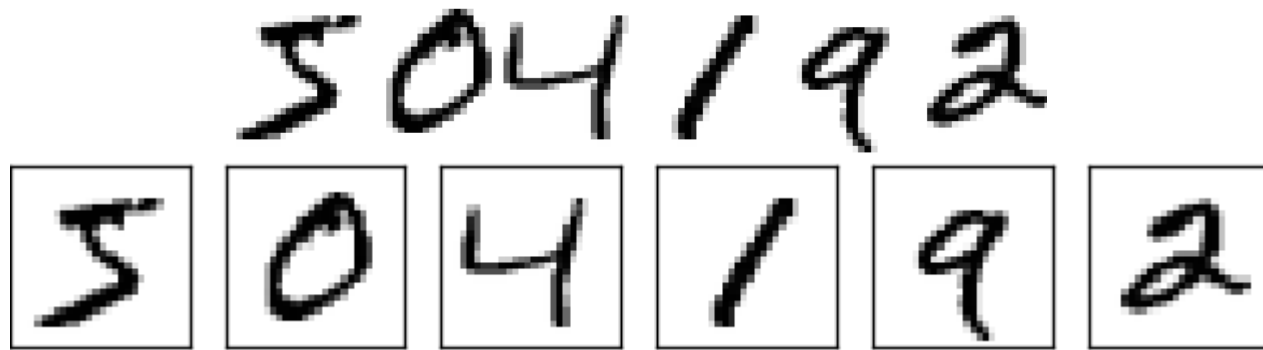
**Principe: Fixer des caractéristiques de l'image**

- Roues (nombre forme)
- Rapport hauteur largeur
- Emplacement des vitres
- Etc.

# Un réseau simple pour classer les chiffres manuscrits

source: <http://neuralnetworksanddeeplearning.com/chap1.html>

- Find a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit. For example, we'd like to break the image into six separate images,

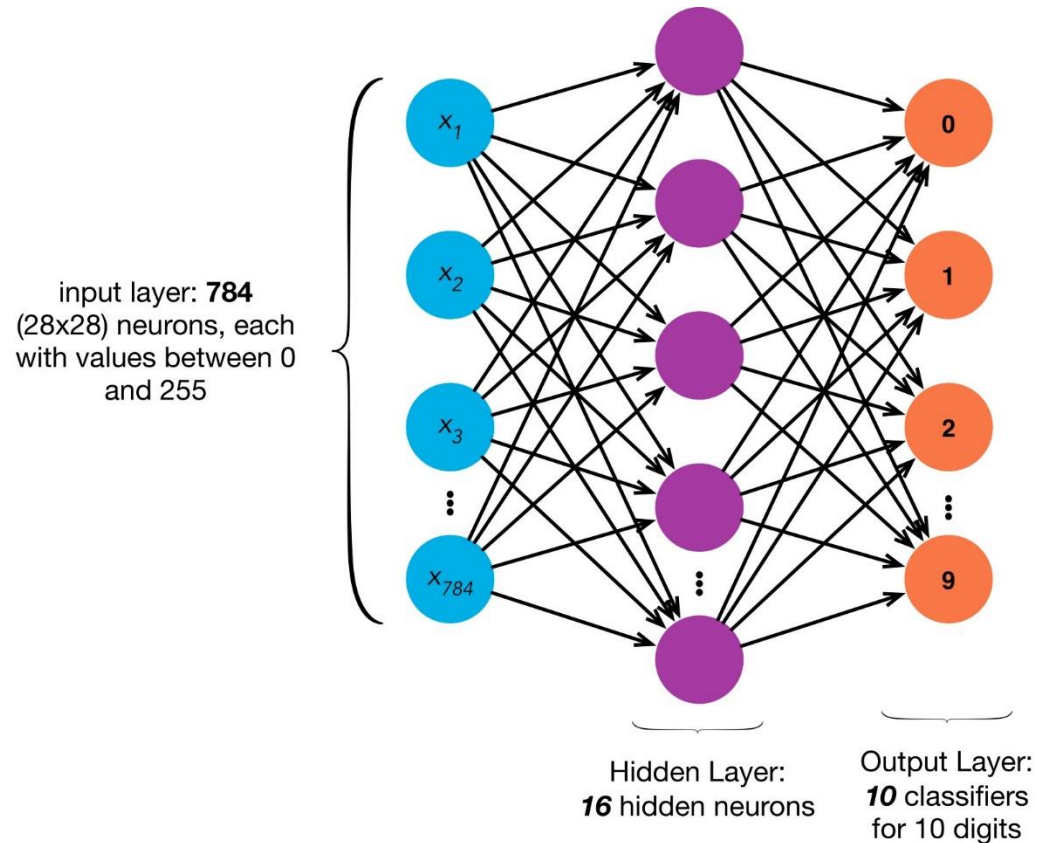


- Once the image has been segmented, the program then needs to classify each individual digit. So, for instance, we'd like our program to recognize that the first digit above, is a 5.



# Un réseau simple pour classer les chiffres manuscrits

- To recognize individual digits we will use a three-layer neural network:
- The training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains  $784=28 \times 28$  neurons. **(Mnist)**
- Each pixel takes a value between 0 and 255 (RGB color code). 0 = white and 255 = black. Each image = an array of 784 numbers like [0, 0, 180, 16, 230, ..., 4, 77, 0, 0, 0]
- The 10 output neurons, returned to us in an array, will each be in charge to classify a digit from 0 to 9. So if the neural network thinks the handwritten digit is a zero, then we should get an output array of [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]. If the neural network thinks the handwritten digit is a 5, then we should get [0, 0, 0, 0, 0, 0, 1, 0, 0, 0].



# mnist.arff

```
@relation mnist_0_9999
@attribute a_0 numeric
@attribute a_1 numeric
@attribute a_2 numeric
@attribute a_3 numeric
@attribute a_4 numeric
@attribute a_5 numeric
@attribute a_6 numeric
@attribute a_7 numeric
@attribute a_8 numeric
@attribute a_9 numeric
@attribute a_10 numeric
@attribute a_11 numeric
@attribute a_12 numeric
@attribute a_13 numeric
```

...

```
@attribute a_777 numeric
@attribute a_778 numeric
@attribute a_779 numeric
@attribute a_780 numeric
@attribute a_781 numeric
@attribute a_782 numeric
@attribute a_783 numeric
@attribute digit {0,1,2,3,4,5,6,7,8,9}
@data
```

```
{202 84,203 185,204 159,205 151,206 60,207 36,230 222,231 254,232 254,233 254,234 254,235 241,236 198,237 198,238 198,239 198,240 198,241 198
,242 198,243 198,244 170,245 52,258 67,259 114,260 72,261 114,262 163,263 227,264 254,265 225,266 254,267 254,268 254,269 250,270 229,271 254
,272 254,273 140,291 17,292 66,293 14,294 67,295 67,296 67,297 59,298 21,299 236,300 254,301 106,326 83,327 253,328 209,329 18,353 22,354 233
,355 255,356 83,381 129,382 254,383 238,384 44,408 59,409 249,410 254,411 62,436 133,437 254,438 187,439 5,463 9,464 205,465 248,466 58,491
126,492 254,493 182,518 75,519 251,520 240,521 57,545 19,546 221,547 254,548 166,572 3,573 203,574 254,575 219,576 35,600 38,601 254,602 254,
603 77,627 31,628 224,629 254,630 115,631 1,655 133,656 254,657 254,658 52,682 61,683 242,684 254,685 254,686 52,710 121,711 254,712 254,713
219,714 40,738 121,739 254,740 207,741 18,784 7}
```

...

la base de données MNIST qui comporte les descriptions de chiffres manuscrits en format  $28 \times 28$ , soit 784 pixels, et les étiquettes associées. La base contient en tout 70 000 exemples (7000 exemples de chaque chiffre).

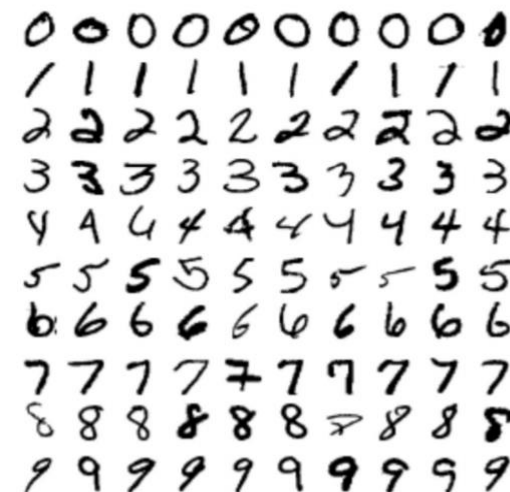


FIGURE 2 – Un échantillon de chiffres de la base MNIST.

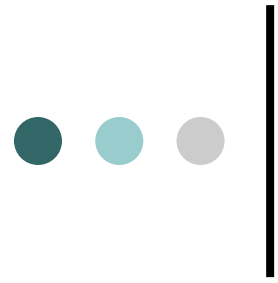


# Comment construire un MLP ?

En pratique on se limite à une structure simple avec généralement 3 couches de neurones

## **Questions:**

- Comment définir l'architecture du réseau?
- Comment apprendre les paramètres (poids des connexions)?

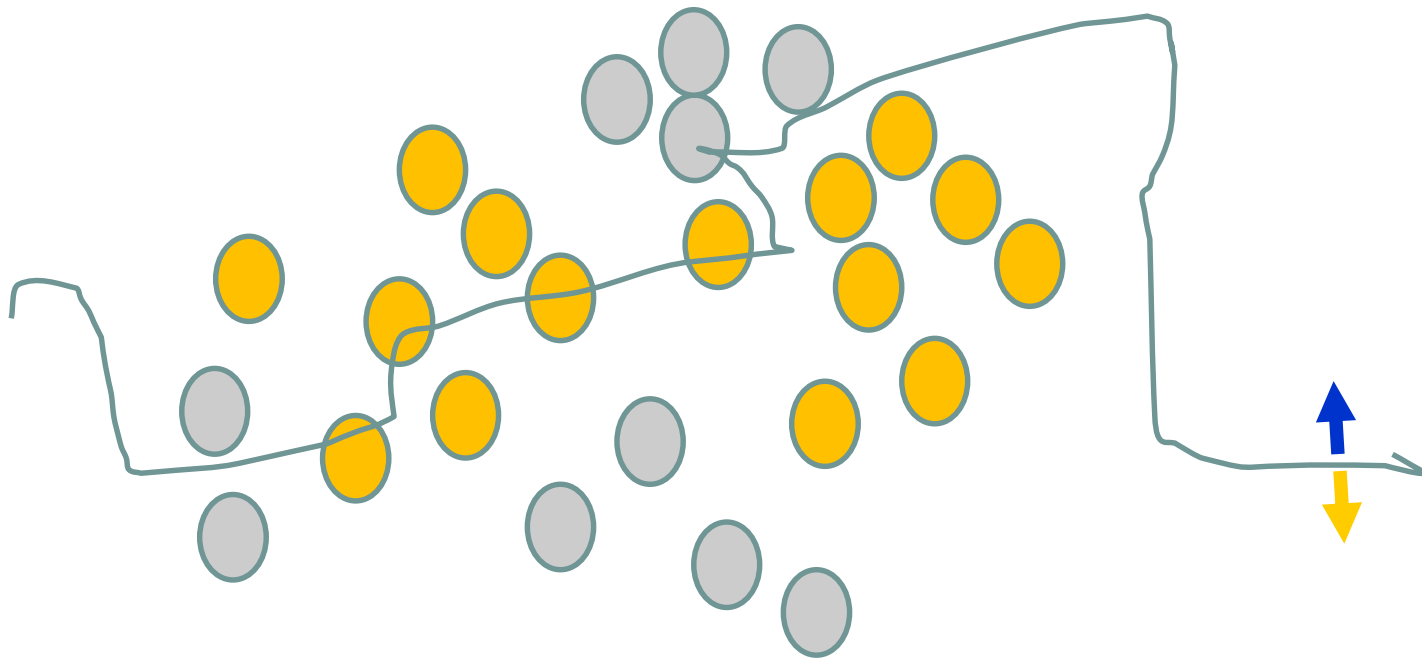


# Apprentissage d'un MLP: Backpropagation - Rétropropagation



# Principle

Initial random weights



# Principe

Present a training instance / adjust the weights



# Principe

Present a training instance / adjust the weights



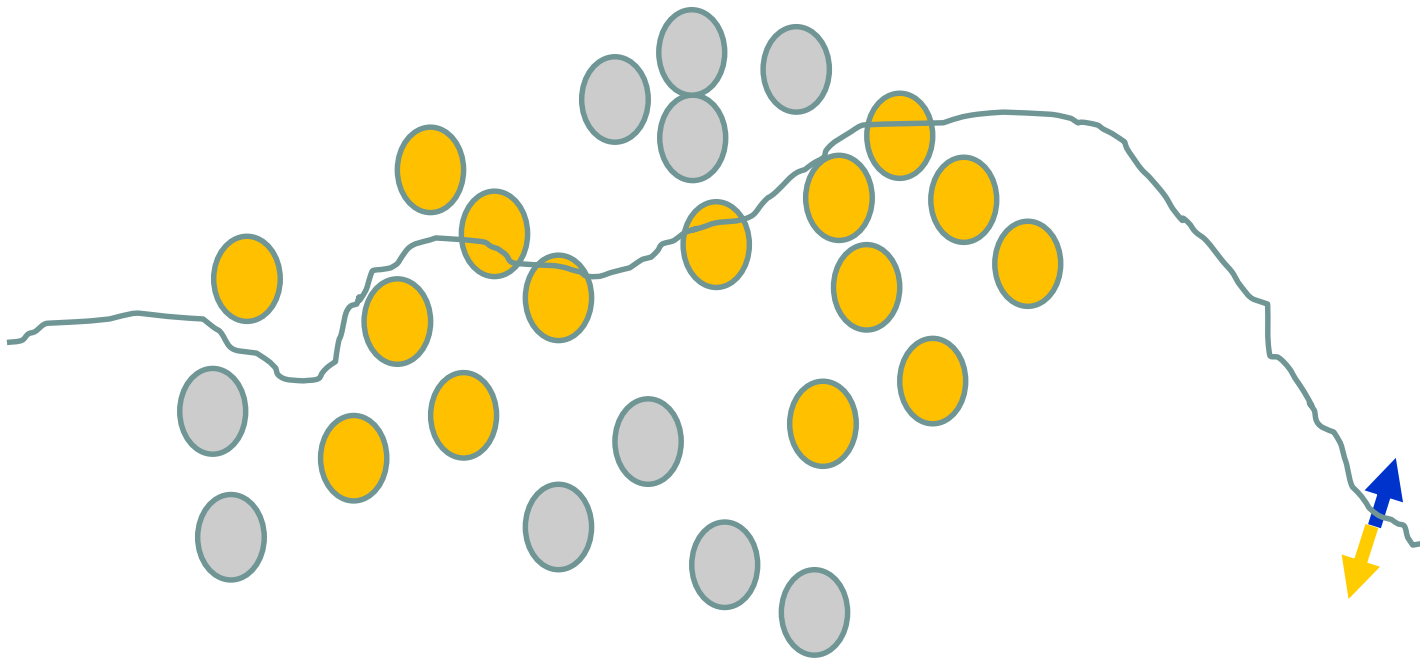
# Principe

Present a training instance / adjust the weights



# Principe

Present a training instance / adjust the weights



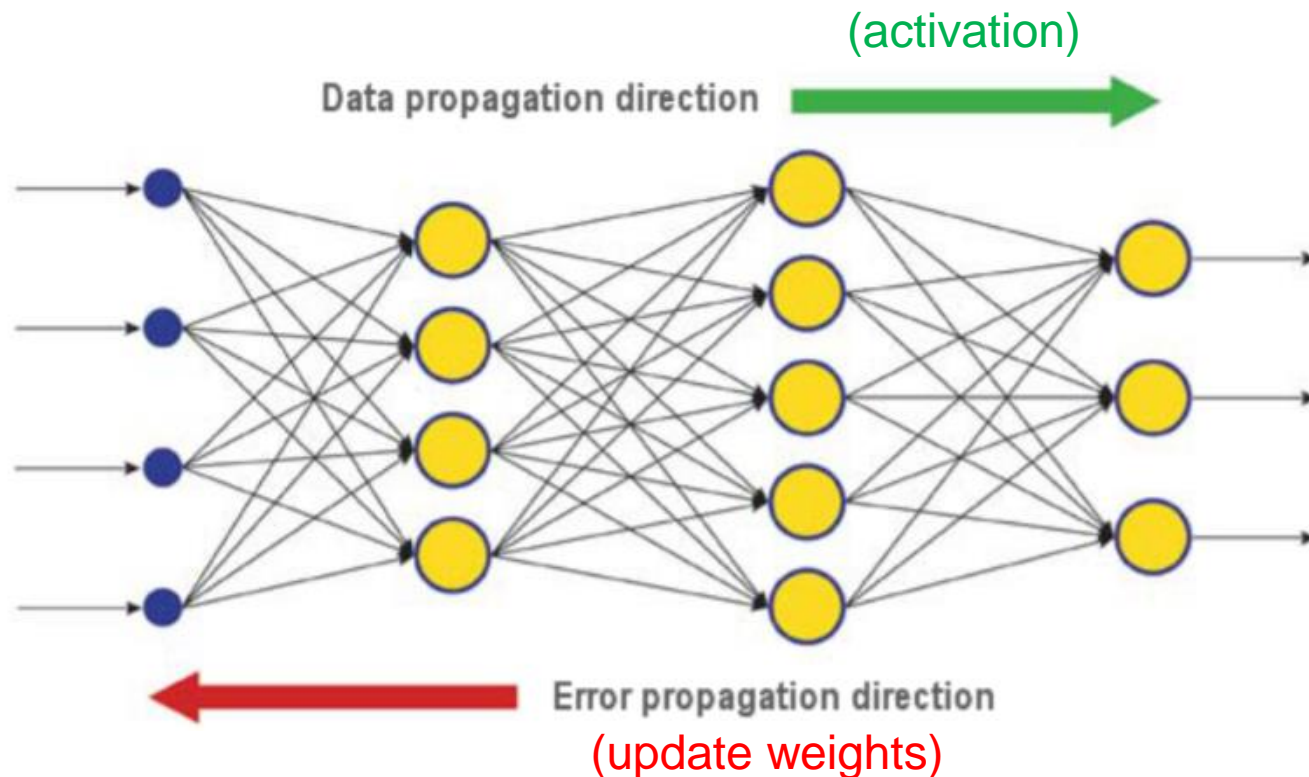
# Principe

Eventually,



# Principe de l'apprentissage d'un MLP: Backpropagation - Rétropropagation

Key idea: Learn by adjusting weights to reduce error on training set.  
→ update weights repeatedly for each example.





# Minimisation du risque

%  $l$  : function de perte  
%  $x$  is a vector of inputs  
%  $y$  is the vector of the true outputs  
%  $h_w(x)$ : the output of the  
perceptron on the example  
%  $w$  is the vector of weights  
(including the bias)

- $S = \{(\mathbf{x}_i, y_i) : i = 1, 2, \dots, m\}$  le training set.
- $h_w$  : la fonction de prediction qui depend des poids  $w$
- $h_w(x)$  la fonction prédite pour  $x$
- ➔ **Objectif** : Minimiser la perte  $l(y, h_w(x))$
- L'erreur dépend des poids  $w$  on va donc utiliser le **gradient (dérivée partielle)** pour déterminer une direction de mise à jour des poids

$$\frac{\partial}{\partial w} l(y, h_w(x))$$

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



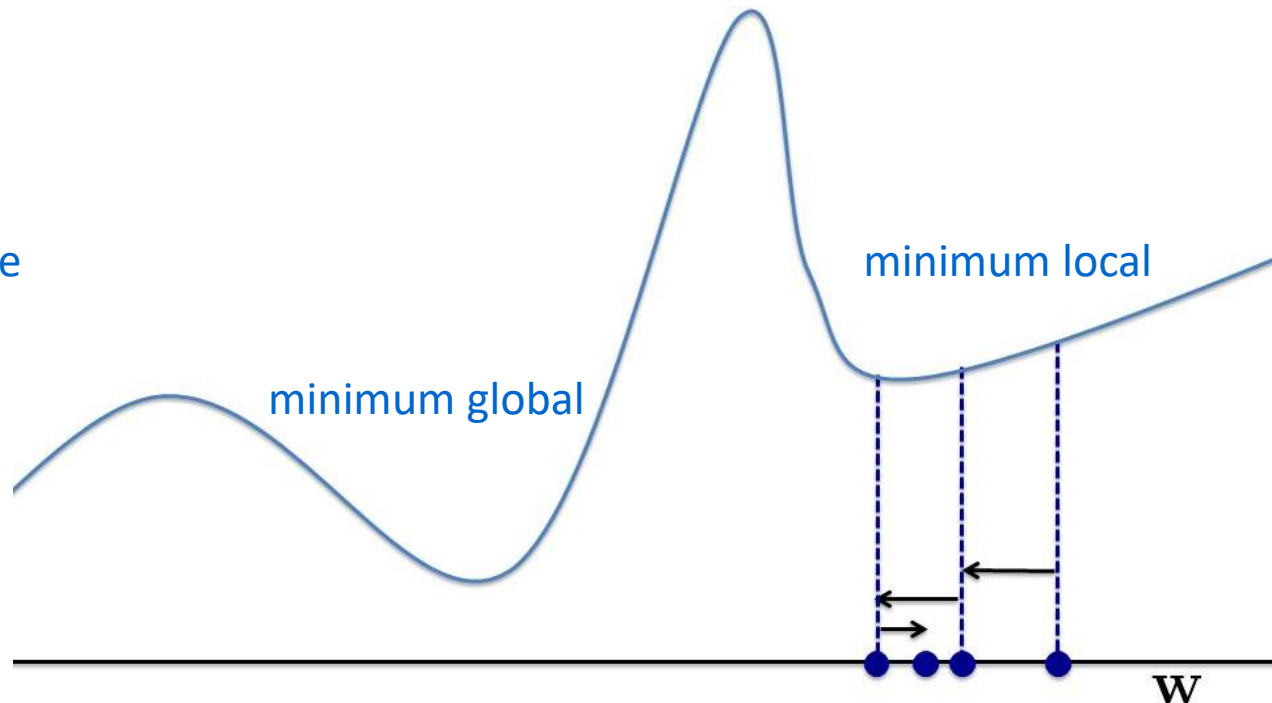
# Descente de gradient

- Recherche locale pour la minimisation de la perte
- Pour mettre à jour du paramètre  $w$  on va dans la direction opposée de ce gradient

$$w \leftarrow w - \alpha \frac{\partial}{\partial w} l(y, h_w(x))$$

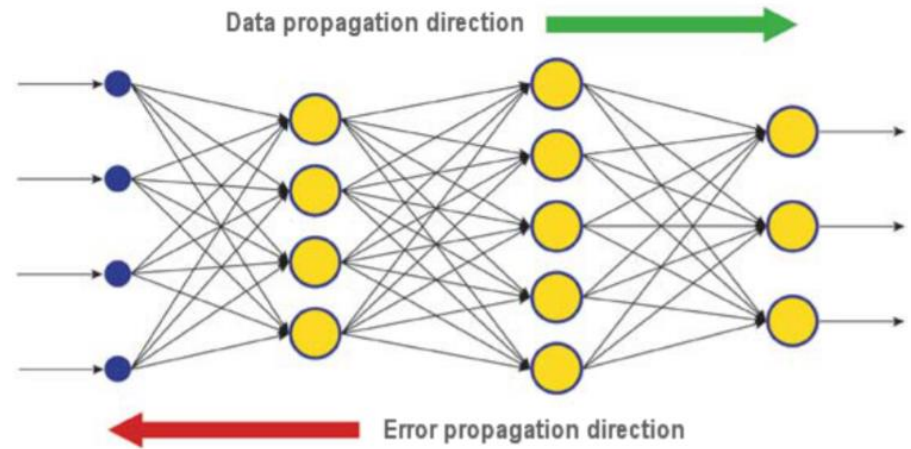
→ Principe de la Descente de gradient (Gradient descent)

L'Algorithme de descente de gradient peut trouver un minimum local au lieu du minimum global



# Backpropagation Learning Algorithm

Backpropagation Learning Algorithm:  
Généralisation de la descente du gradient



*Initialisation de tous les poids du réseau (petites valeurs)*

**Repeat**

**for each** example in examples **do**

*/\*propagate the inputs **forward** to compute the outputs \*/*

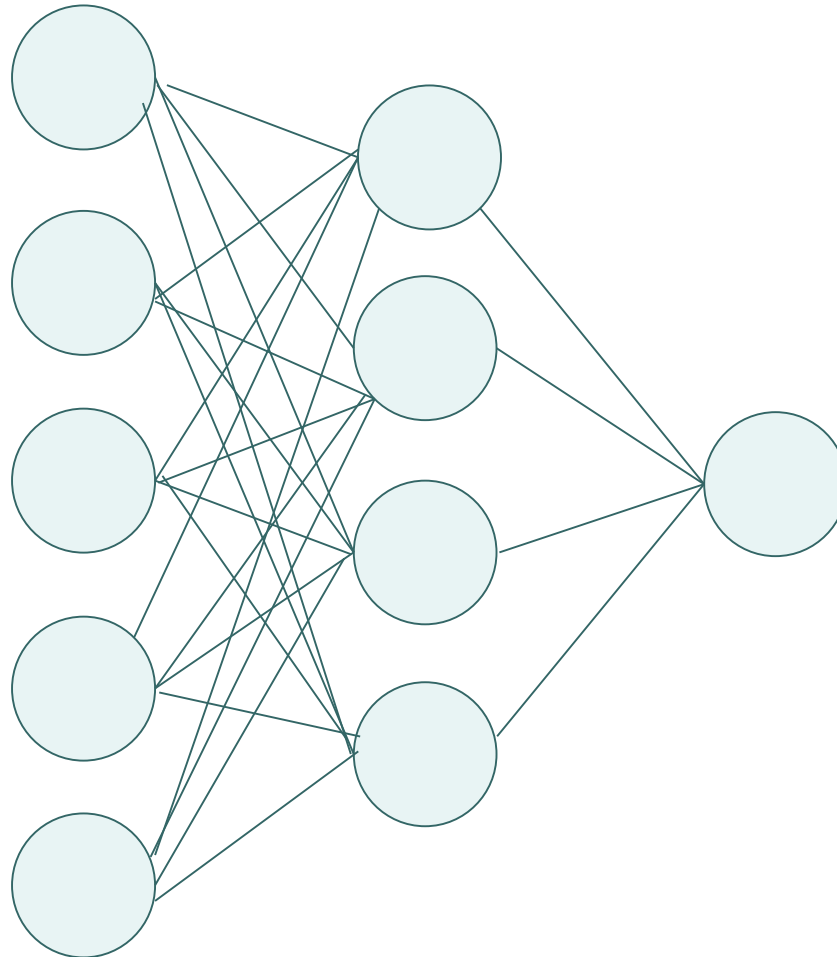
*/\*propagate deltas **backward** from output layer to input layer\*/*

*/\*update every weight in network using deltas\*/*

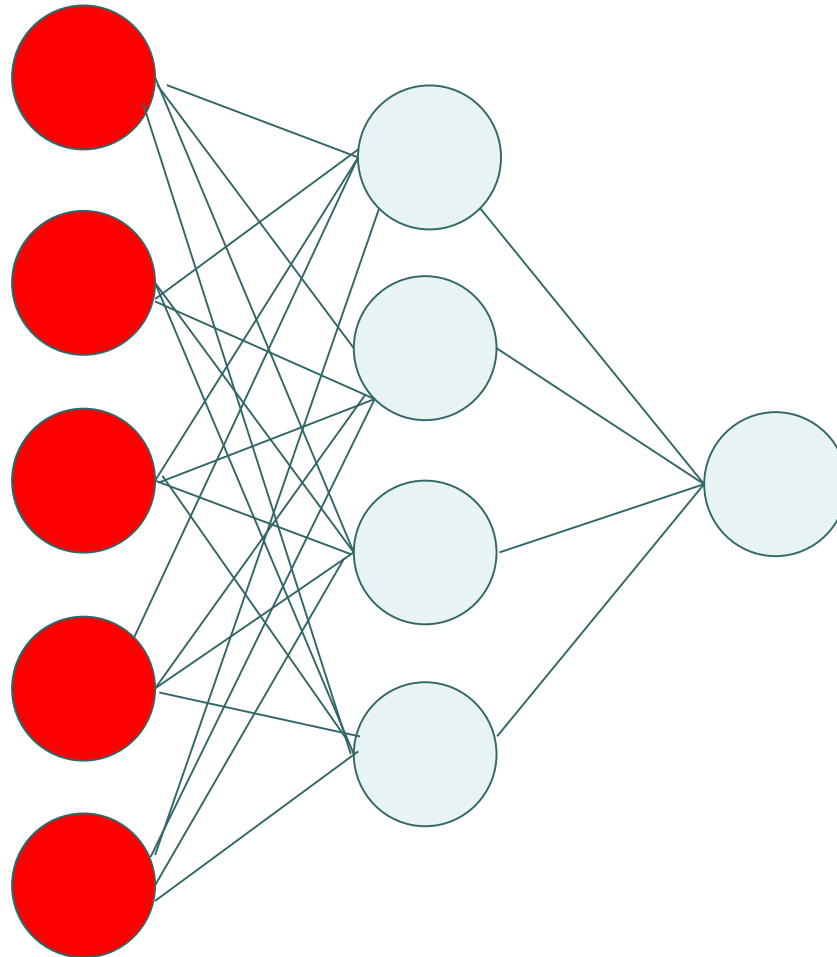
**Until some stopping criterion is satisfied**

**Return Network**

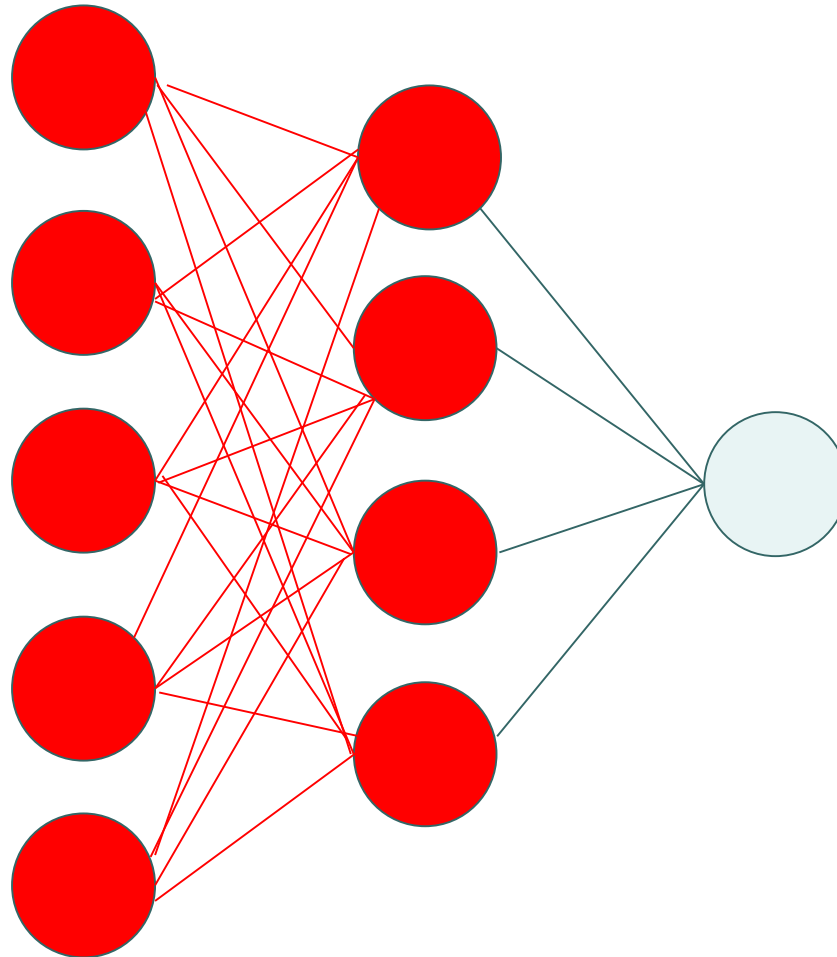
# ● ● ● | Principe de la rétropropagation



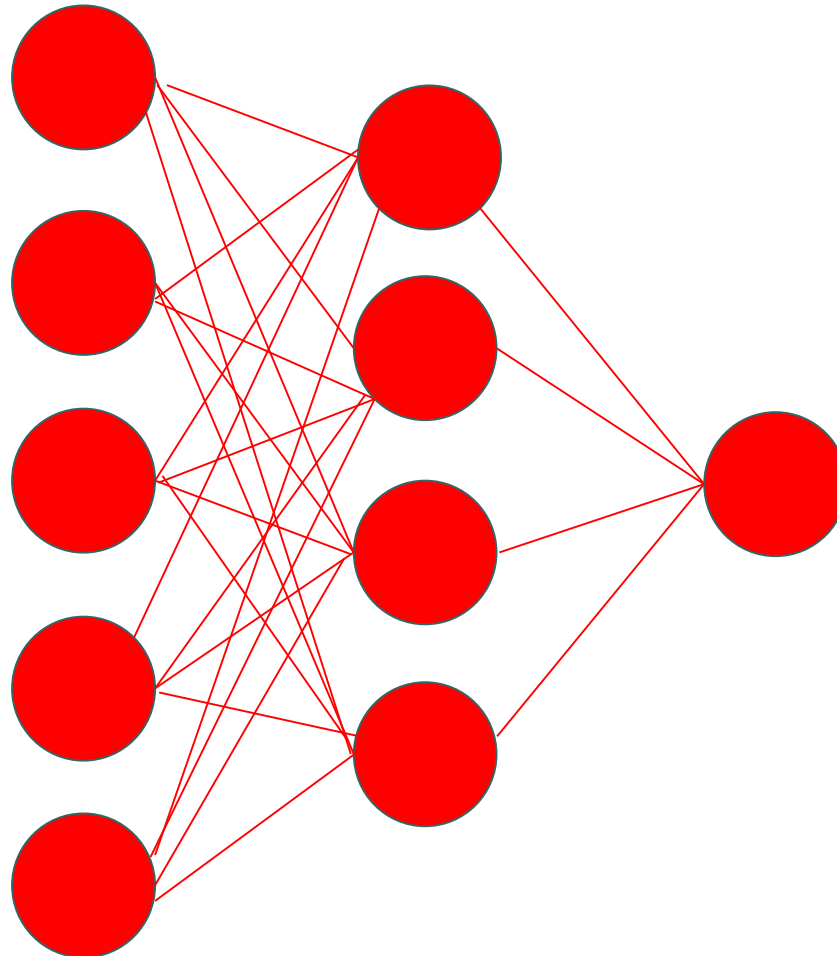
# Principe de la rétropropagation



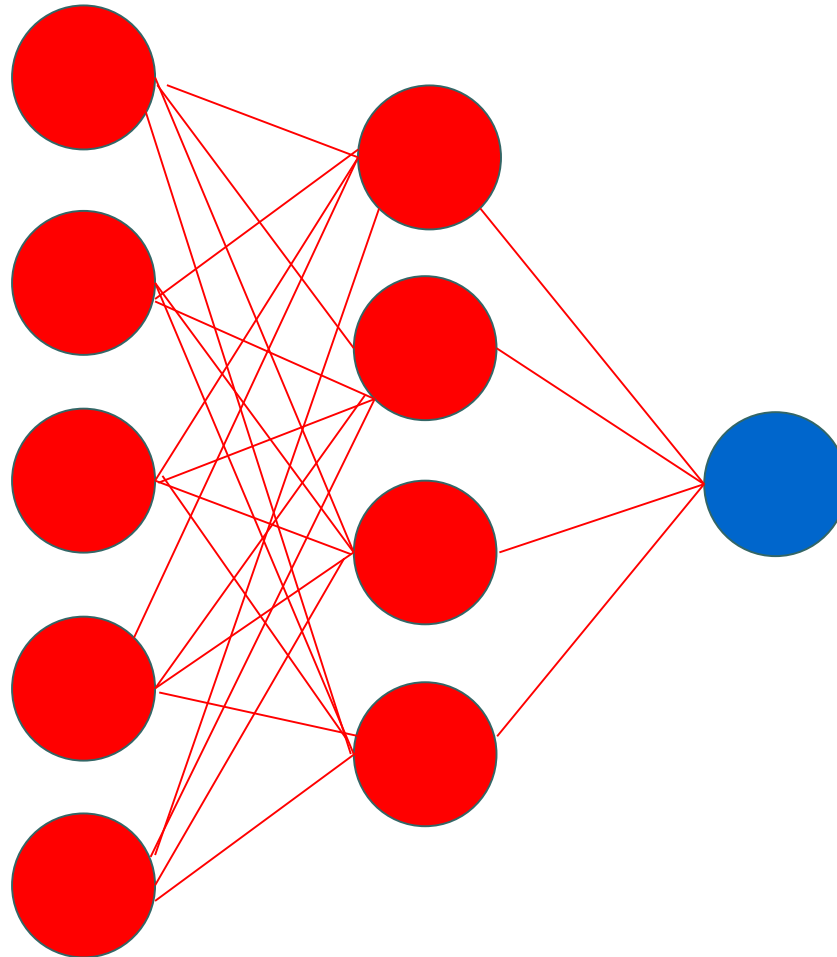
# Principe de la rétropropagation



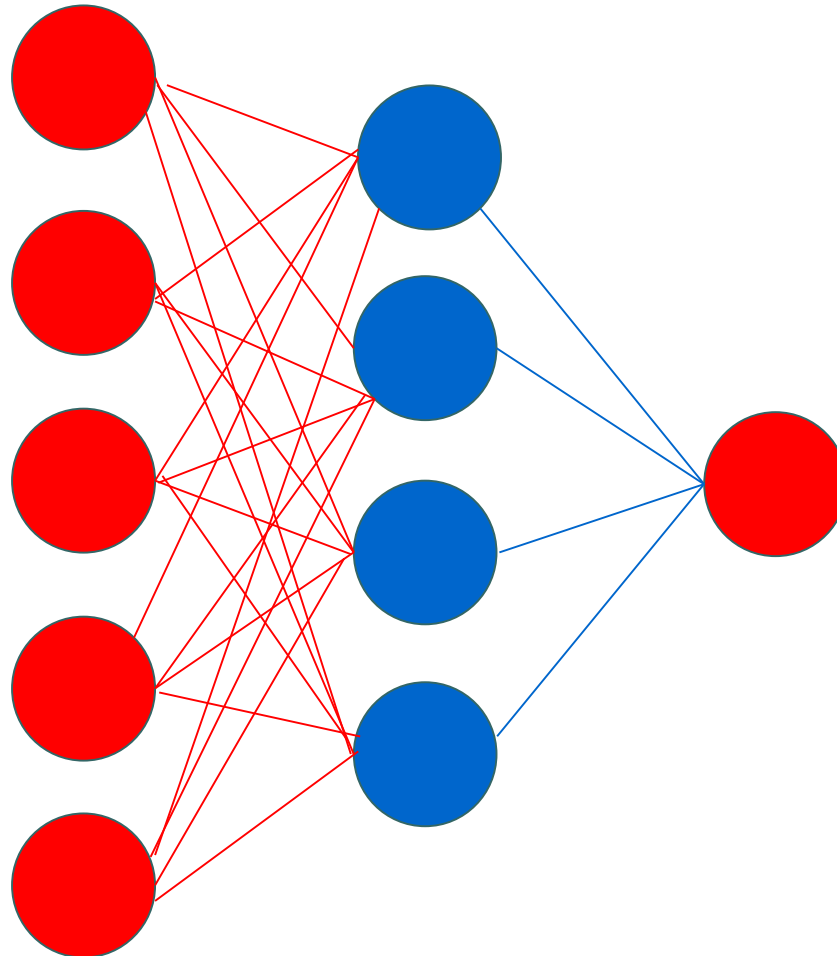
# Principe de la rétropropagation



# ● ● ● | Principe de la rétropropagation

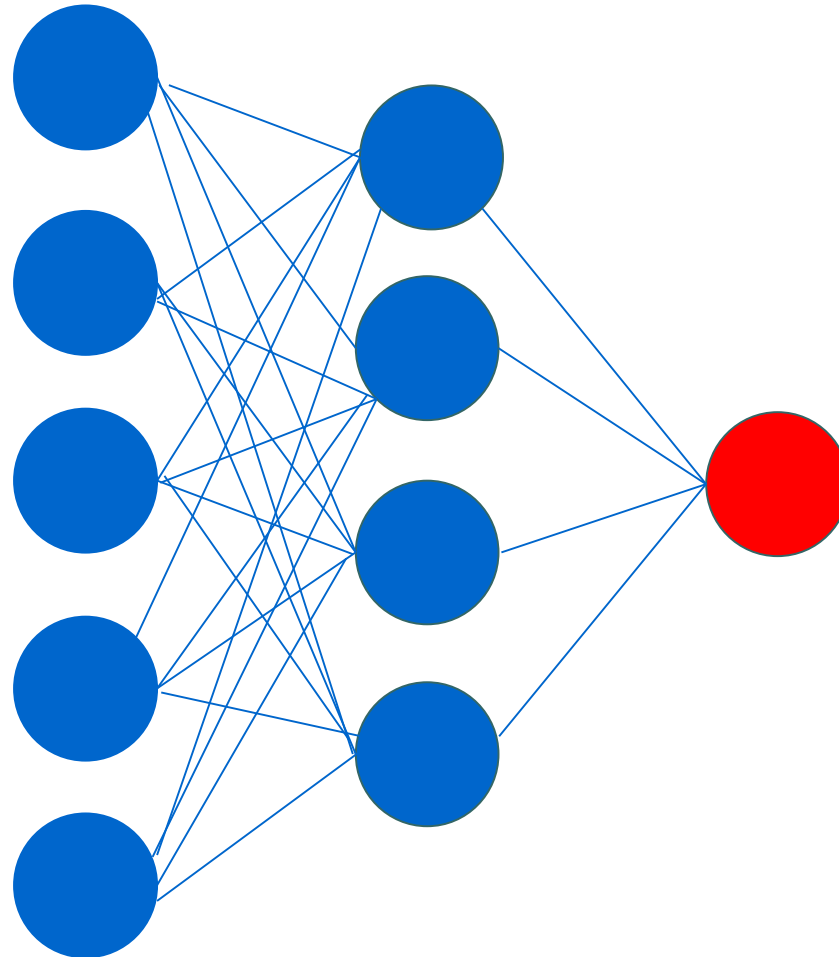


# Principe de la rétropropagation



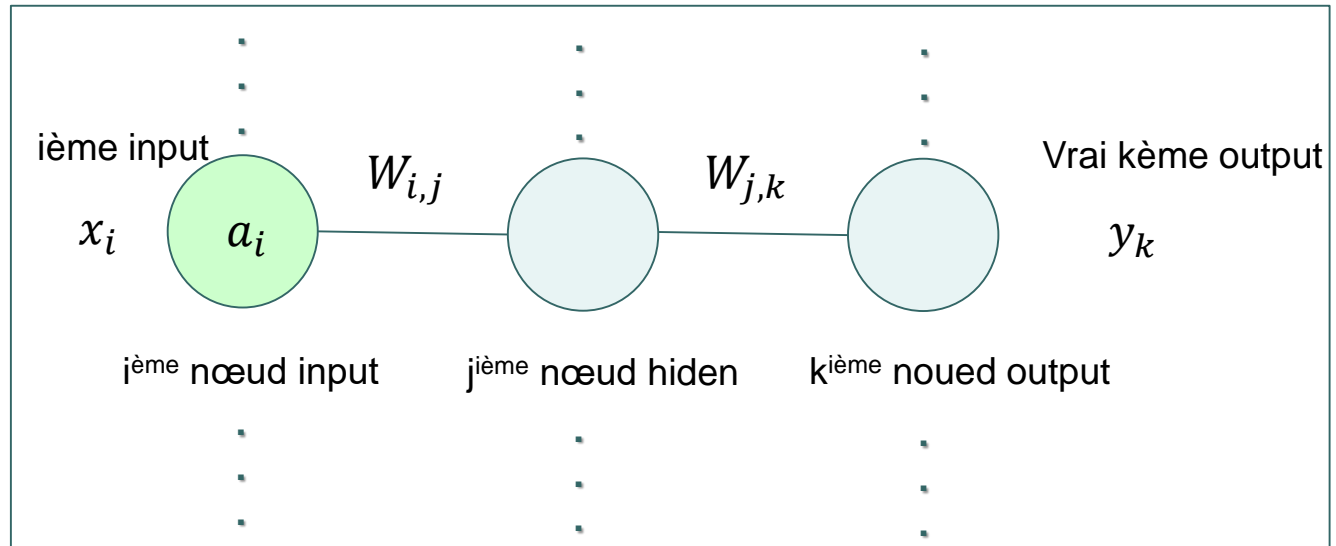


# Principe de la rétropropagation



# Principe de la rétropropagation (1)

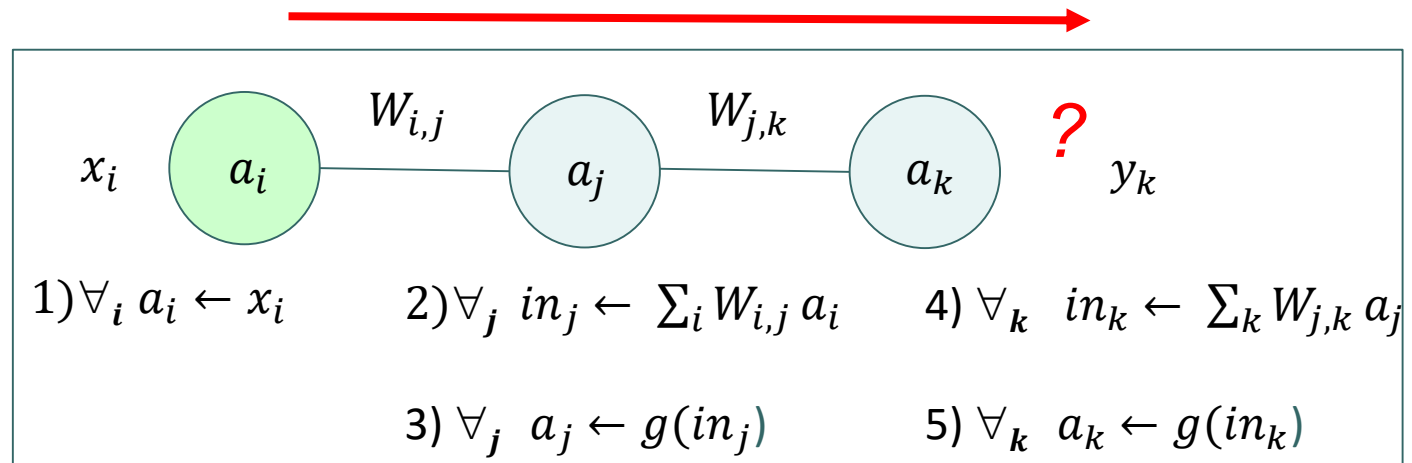
On suppose (sans perte de généralité) qu'on a une seule couche cachée



# Principe de la rétropropagation (2)

## Phase 1: Propagation avant

*/\*propagate the inputs forward to compute the outputs \*/*

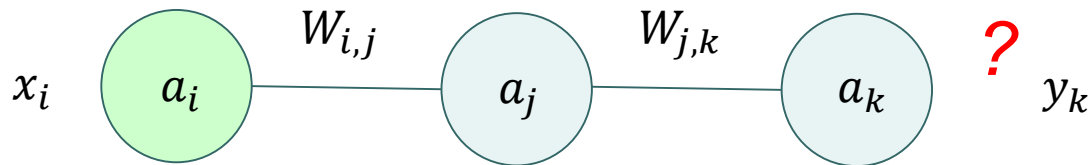


Puis on calcule l'erreur observée ( $y_k - a_k$ ) pour chaque  $a_k$  (dans output layer) à la fin de cette phase puis on corrige les poids si nécessaire par propagation arrière

# Principe de la rétropropagation (3)

## Phase 1: Propagation arrière

*/\*propagate deltas backward from output layer to input layer\*/*



pas la peine de calculer les delta de la couche d'entrée (on ne va pas l'utiliser dans les phases suivantes)

$$3) \forall_j \Delta_j \leftarrow g'(in_j) \sum_k W_{j,k} \Delta_k$$

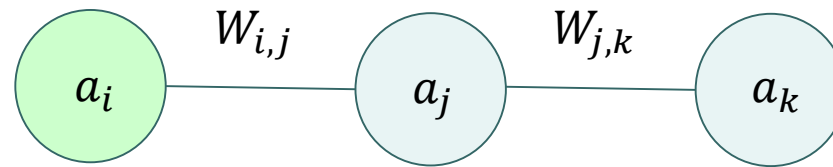
$$1) \forall_k \text{Err}_k \leftarrow y_k - a_k$$

$$2) \forall_k \Delta_k \leftarrow g'(in_k) \times \text{Err}_k$$
$$\forall_k \Delta_k \leftarrow g'(in_k) \times (y_k - a_k)$$

# Principe de la rétropropagation (4)

## Phase 3: Mise à jour des poids

*/\*update every weight in network using deltas\*/*



$\forall W_{j,k}$

$$W_{j,k} \leftarrow W_{j,k} + \alpha \times a_j \times \Delta_k$$

% cad que le gradient =  $-a_j \times \Delta_k$

$\forall W_{i,j}$

$$W_{i,j} \leftarrow W_{i,j} + \alpha \times a_i \times \Delta_j$$

% cad que le gradient =  $-a_i \times \Delta_j$

**% preuve dans la suite**

# Back-Propagation Learning Algorithm

**function** BACK-PROP-LEARNING (*examples*, *network*,  $\alpha$ ) 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  $W_{j,i}$ , activation function  $g$

$\alpha$ : learning rate

**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

Initialiser les poids à de petites valeurs aléatoires

**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**  $l=2$  **to**  $L$  **do** %  $L$  nb de layers

**for each node**  $j$  **in layer**  $l$  **do**

$in_j \leftarrow \sum_i W_{i,j} a_i$

$a_j \leftarrow g(in_j)$

Propagation avant

*/\*propagate deltas **backward** from output layer to input layer\*/*

**for each node**  $j$  **in the output layer do**

$\Delta_j \leftarrow g'(in_j) \times (y_j - a_j)$

**for**  $l=L-1$  **to**  $2$  **do**

**for each node**  $i$  **in layer**  $l$  **do**

$\Delta_i \leftarrow g'(in_i) \sum_j W_{i,j} \Delta_j$

Propagation arrière

*/\*update every weight in network using deltas\*/*

**for each weight**  $W_{i,j}$  **in network do**

$W_{i,j} \leftarrow W_{i,j} + \alpha \times a_i \times \Delta_j$

Si la condition d'arrêt n'est pas atteinte on refait le traitement avec tous les exemples

epoch

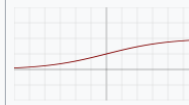
Pour mettre à jour les paramètres on va dans la direction opposée du gradient

$$W_{i,j} \leftarrow W_{i,j} - \alpha \frac{\partial E}{\partial W_{i,j}}$$

la valeur du gradient :  $-a_i \times \Delta_i$

**Until some stopping criterion is satisfied**

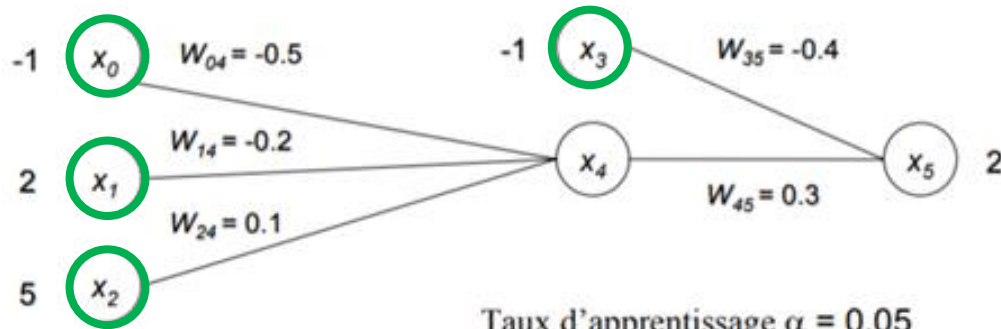
**Return Network**



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

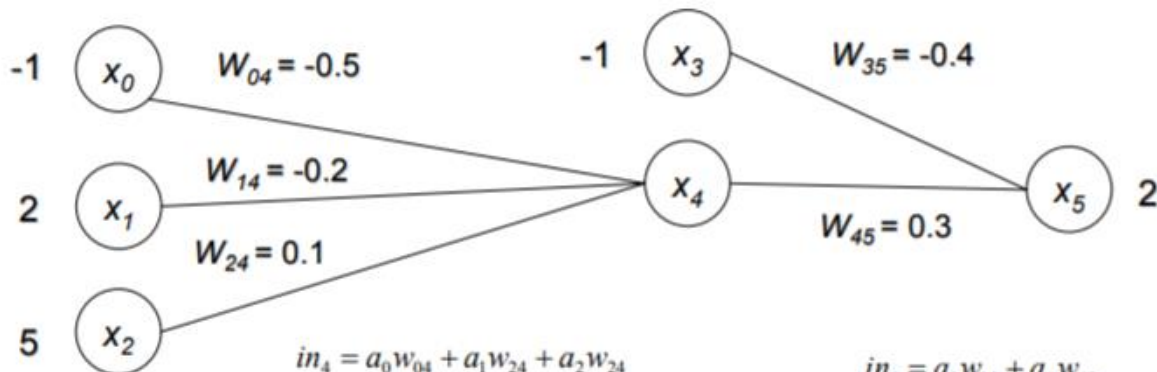
$$f'(x) = f(x)(1 - f(x))$$

# Exemple



Taux d'apprentissage  $\alpha = 0.05$

Fonction sigmoïde  $g$  aux noeuds  $x_4$  et  $x_5$



$$\begin{aligned} a_0 &= -1 \\ a_1 &= 2 \\ a_2 &= 5 \\ a_3 &= -1 \end{aligned}$$

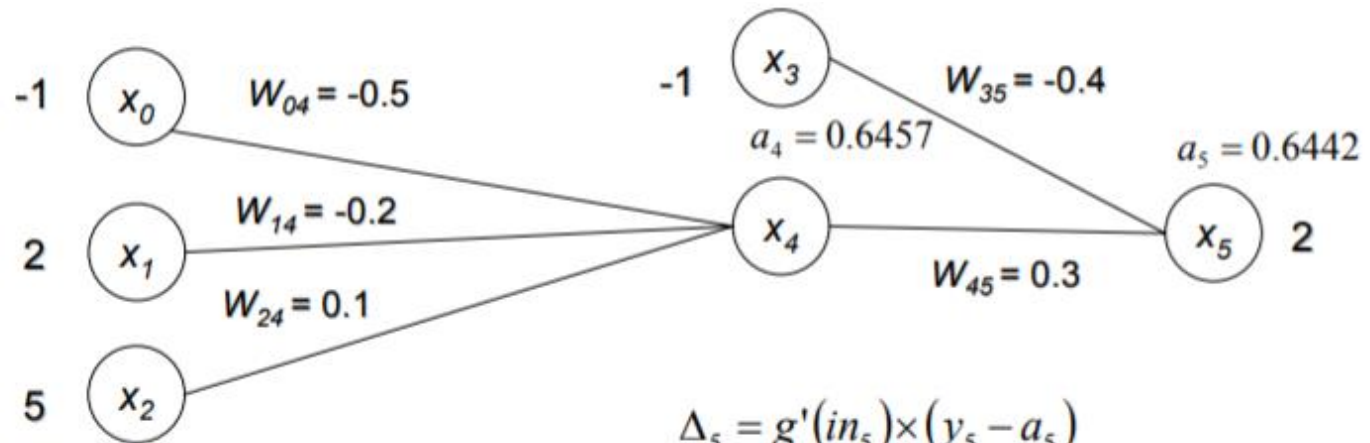
$$\begin{aligned} in_4 &= a_0 w_{04} + a_1 w_{14} + a_2 w_{24} \\ &= (-1 \times -0.5) + (2 \times -0.2) + (5 \times 0.1) \\ &= 0.6 \end{aligned}$$

$$a_4 = g(in_4) = \frac{1}{1 + e^{-0.6}} = 0.6457$$

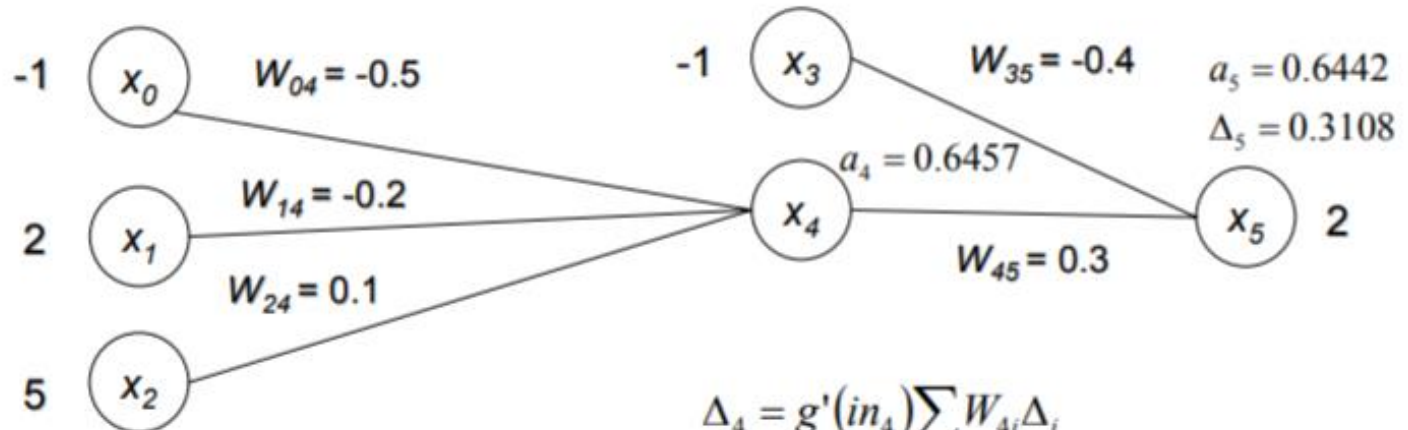
$$\begin{aligned} in_5 &= a_3 w_{35} + a_4 w_{45} \\ &= (-1 \times -0.4) + (0.6457 \times 0.3) \\ &= 0.5937 \end{aligned}$$

$$a_5 = g(in_5) = \frac{1}{1 + e^{-0.5937}} = 0.6442$$

Ça correspond à  $P(y=1|X)$   
Cad le niveau de croyance  
que  $X$  appartient à la classe 1

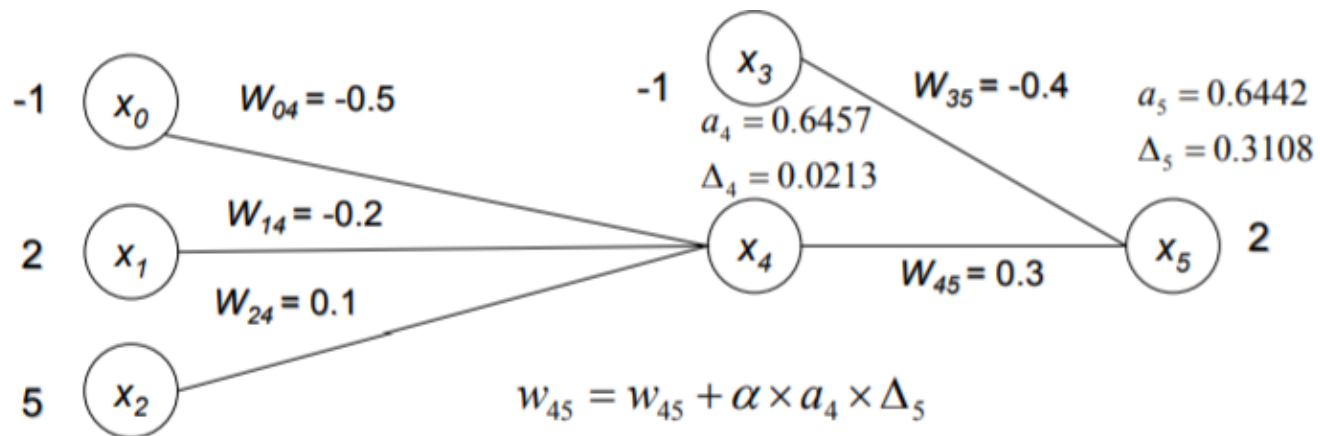


$$\begin{aligned}\Delta_5 &= g'(in_5) \times (y_5 - a_5) \\ &= a_5 \times (1 - a_5) \times (y_5 - a_5) \\ &= 0.6442 \times (1 - 0.6442) \times (2 - 0.6442) \\ &= 0.3108\end{aligned}$$



$$\begin{aligned}\Delta_4 &= g'(in_4) \sum_i W_{4i} \Delta_i \\ &= a_4 \times (1 - a_4) \times W_{45} \Delta_5 \\ &= 0.6457 \times (1 - 0.6457) \times (0.3 \times 0.3108) \\ &= 0.0213\end{aligned}$$



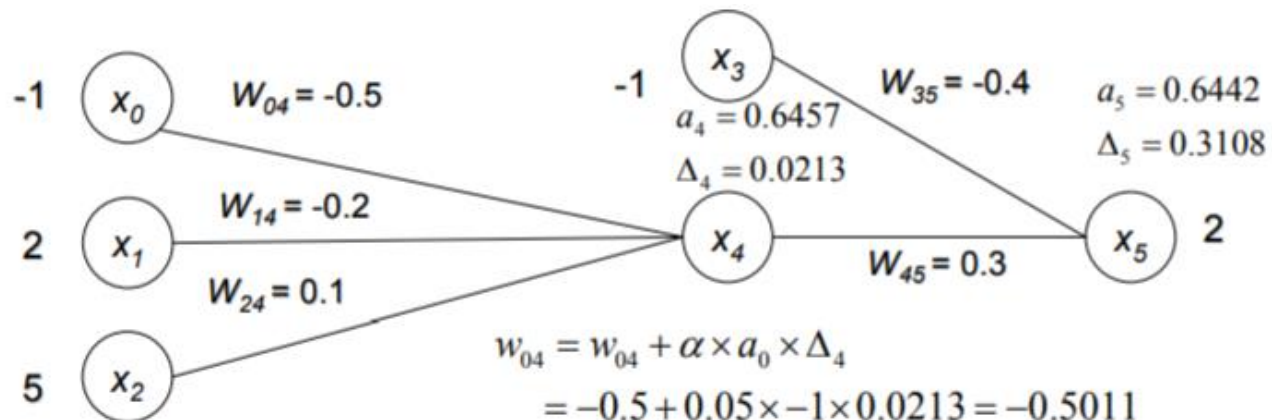


$$w_{45} = w_{45} + \alpha \times a_4 \times \Delta_5$$

$$= 0.3 + 0.05 \times 0.6457 \times 0.3108 = 0.3100$$

$$w_{35} = w_{35} + \alpha \times a_3 \times \Delta_5$$

$$= -0.4 + 0.05 \times -1 \times 0.3108 = -0.4155$$



$$w_{04} = w_{04} + \alpha \times a_0 \times \Delta_4$$

$$= -0.5 + 0.05 \times -1 \times 0.0213 = -0.5011$$

$$w_{14} = w_{14} + \alpha \times a_1 \times \Delta_4$$

$$= -0.2 + 0.05 \times 2 \times 0.0213 = -0.1979$$

$$w_{24} = w_{24} + \alpha \times a_2 \times \Delta_4$$

$$= 0.1 + 0.05 \times 5 \times 0.0213 = 0.1053$$

# Gradient pour les poids de la couche output

## Dérivation (1)

%  $a_k$  valeur "résultat" du k<sup>ème</sup> noeud de la couche output

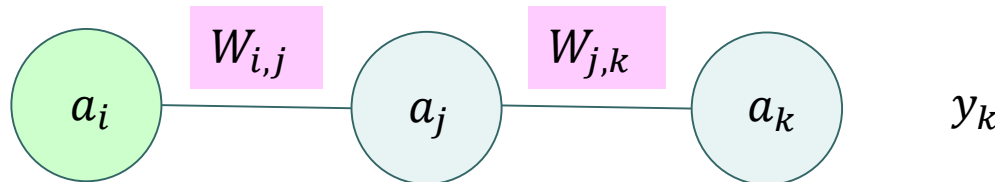
%  $y_k$  valeur réelle du k<sup>ème</sup> noeud de la couche output

% L'erreur quadratique pour un exemple  $x_1 \dots x_N \ y_1 \dots y_O$

% Somme sur les erreurs de la couche output

$$E = \frac{1}{2} \sum_k (a_k - y_k)^2$$

Dividing the sum (of 'n') squares by 'n' is so that the number of elements in the set doesn't effect the output (allowing more consistent results across different sized data sets). As for dividing by 2: you could divide by any constant (5, 10, 20), it doesn't matter: however, dividing by 2 makes the subsequent math easier, since it cancels out the extra '2' introduced when taking the derivative of the squared value (i.e., derivative of  $j^2$  is  $2j$ , dividing by 2 makes it simply 'j').



On dérive l'erreur  $E = \frac{1}{2} \sum_k (a_k - y_k)^2$  par rapport aux:

1. poids  $W_{j,k}$
2. poids  $W_{i,j}$

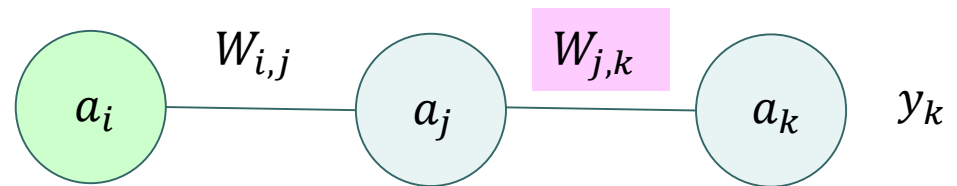
# Gradient pour les poids de la couche output

## Dérivation (2)

$$E = \frac{1}{2} \sum_k (a_k - y_k)^2$$

- On calcule le gradient (on derive l'erreur) pour un seul noeud de la couche output (kième noeud) par rapport à  $W_{j,k}$
- The summation disappears in the derivative. This is because when we take the partial derivative with respect to the j-th dimension/node, the only term that survives in the error gradient is j-th, and thus we can ignore the remaining terms in the summation).

$$\begin{aligned}
 \frac{\partial E}{\partial W_{j,k}} &= (a_k - y_k) \frac{\partial a_k}{\partial W_{j,k}} \\
 &= (a_k - y_k) \frac{\partial g(in_k)}{\partial W_{j,k}} \\
 &= (a_k - y_k) g'(in_k) \frac{\partial in_k}{\partial W_{j,k}} \\
 &= (a_k - y_k) g'(in_k) \frac{\partial}{\partial W_{j,k}} \sum_j W_{j,k} a_j \\
 &= (a_k - y_k) g'(in_k) a_j \\
 &= -a_j \Delta_k
 \end{aligned}$$



$$\begin{aligned}
 \% a_k &= g(in_k) \\
 \% in_k &= \sum_j W_{j,k} a_j \\
 \% \Delta_k &= -(a_k - y_k) g'(in_k) \\
 &= (y_k - a_k) g'(in_k)
 \end{aligned}$$

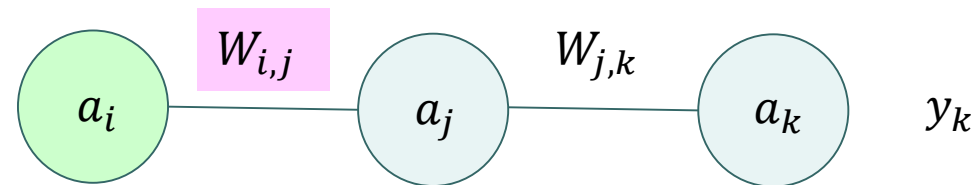
# Gradient pour les poids de la couche cachée

## Dérivation (1)

$$E = \frac{1}{2} \sum_k (a_k - y_k)^2$$

- Calcul du gradient pour les poids  $W_{i,j}$  entre la couche output et la couche cachée (hypothèse 1 seule couche cachée).
- On calcule le gradient (on dérive l'erreur) pour un seul noeud de la couche output (kième noeud) par rapport à  $W_{i,j}$
- Notice here that the sum does not disappear because, due to the fact that the layers are fully connected, each of the hidden unit outputs affects the state of each output unit.

$$\begin{aligned} \frac{\partial E}{\partial W_{i,j}} &= \sum_k (a_k - y_k) \frac{\partial a_k}{\partial W_{i,j}} \\ &= \sum_k (a_k - y_k) \frac{\partial g(in_k)}{\partial W_{i,j}} \\ &= \sum_k (a_k - y_k) g'(in_k) \frac{\partial in_k}{\partial W_{i,j}} \end{aligned}$$



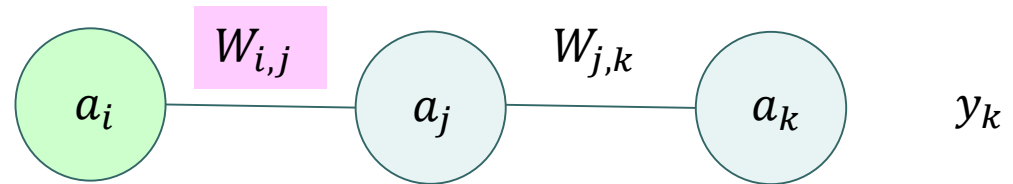
$$\% a_k = g(in_k)$$

# Gradient pour les poids de la couche cachée

## Dérivation (2)

$$E = \frac{1}{2} \sum_k (a_k - y_k)^2$$

$$\begin{aligned} in_k &= \sum_j W_{j,k} a_j \\ &= \sum_j W_{j,k} g(in_j) \\ &= \sum_j W_{j,k} g\left(\sum_i W_{i,j} a_i\right) \end{aligned}$$



% From the last term we see that  $in_k$  is *indirectly* dependent on  $W_{i,j}$

$$\begin{aligned} \% a_j &= g(in_j) \\ \% in_j &= \sum_i W_{i,j} a_i \end{aligned}$$

# Gradient pour les poids de la couche cachée

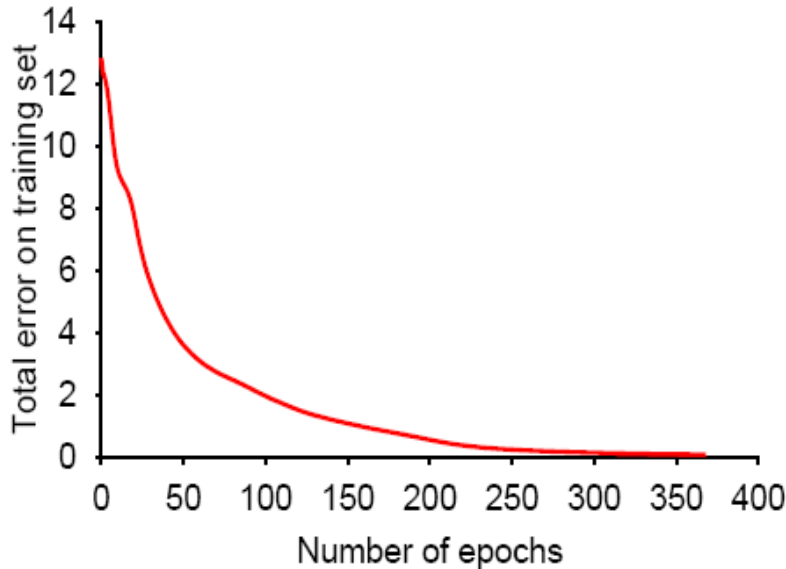
## Dérivation (3)

$$\begin{aligned}
 \frac{\partial in_k}{\partial W_{i,j}} &= \frac{\partial in_k}{\partial a_j} \frac{\partial a_j}{\partial W_{i,j}} \\
 &= \frac{\partial}{\partial a_j} a_j W_{j,k} \frac{\partial a_j}{\partial W_{i,j}} \\
 &= W_{j,k} \frac{\partial a_j}{\partial W_{i,j}} \\
 &= W_{j,k} \frac{\partial g(in_j)}{\partial W_{i,j}} \\
 &= W_{j,k} g'(in_j) \frac{\partial in_j}{\partial W_{i,j}} \\
 &= W_{j,k} g'(in_j) \frac{\partial}{\partial W_{i,j}} (\sum_i W_{i,j} a_i) \\
 &= W_{j,k} g'(in_j) a_i
 \end{aligned}$$

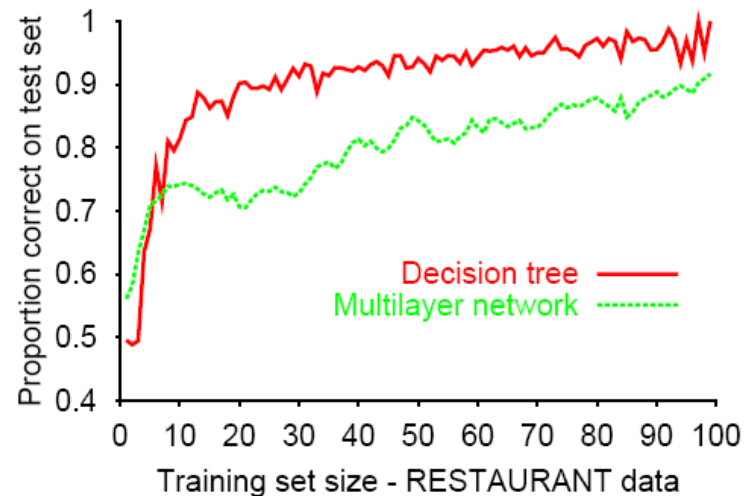
$$\begin{aligned}
 \frac{\partial E}{\partial W_{i,j}} &= \sum_k (a_k - y_k) g'(in_k) \frac{\partial in_k}{\partial W_{i,j}} \\
 &= -\sum_k \Delta_k \frac{\partial in_k}{\partial W_{i,j}} \\
 &= -\sum_k \Delta_k W_{j,k} g'(in_j) a_i \\
 &= -a_i g'(in_j) \sum_k \Delta_k W_{j,k} \\
 &= -a_i \Delta_j
 \end{aligned}$$

$$\Delta_j = g'(in_j) \sum_k W_{j,k} \Delta_k$$

# Back Propagation performance



(a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain.



(b) Comparative learning curves showing that decision-tree learning does slightly better than back-propagation in a multilayer network (4 hidden units).



# Batch / Séquentiel

- 2 façons d'appliquer l'algorithme Back Propagation :
  - « **batch** » : mise à jour des poids après la présentation de tous les exemples (version étudiée)
    - Convergence to the unique global minimum is guaranteed (as long as we pick  $\alpha$  small enough) but may be very slow: we have to cycle through all the training data for every step, and there may be many steps.
    - calculs et stockage plus lourds si trop d'exemples
  - **séquentiel** : (on-line, **stochastique**)  
mise à jour des poids après chaque exemple
    - besoin de tirer l'exemple au hasard
    - problèmes de convergence
    - It is often faster than batch gradient descent. With a fixed learning rate  $\alpha$ , however, it does not guarantee convergence; it can oscillate around the minimum without settling down





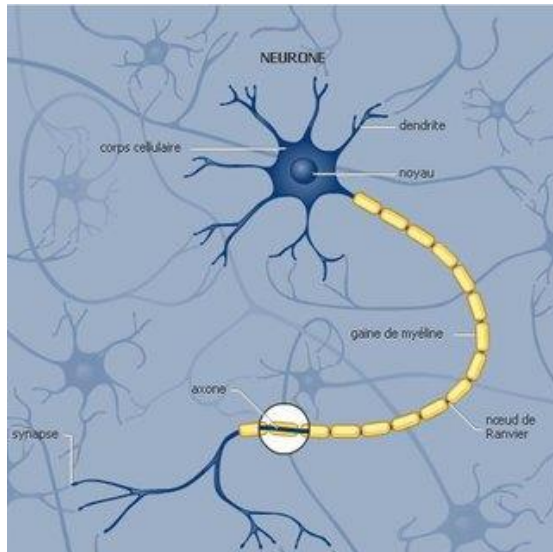
# Condition d'arrêt

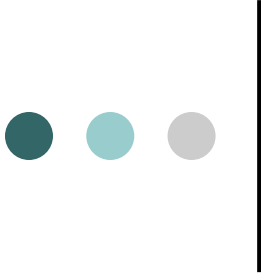
- Le nombre d'itérations est important car:
  - Si trop faible, l'erreur n'est pas suffisamment réduite.
  - Si trop grand, le réseau devient trop spécifique aux données d'entraînement (sur-apprentissage )

→ une technique de régularisation : arrêt prématuré (early stopping) : éviter le sur-apprentissage en arrêtant l'algorithme avant d'avoir atteint le minimum
- Il y a plusieurs conditions d'arrêt possibles
  - Après un certain nombre fixe d'itérations.
  - Lorsque les poids se stabilisent
  - Lorsque l'erreur dans les sorties des exemples d'entraînement descend en dessous d'une certaine borne.
  - Lorsque l'erreur avec les exemples de validation descend en dessous
  - d'une certaine borne.

# MLP = Boîte noire ?

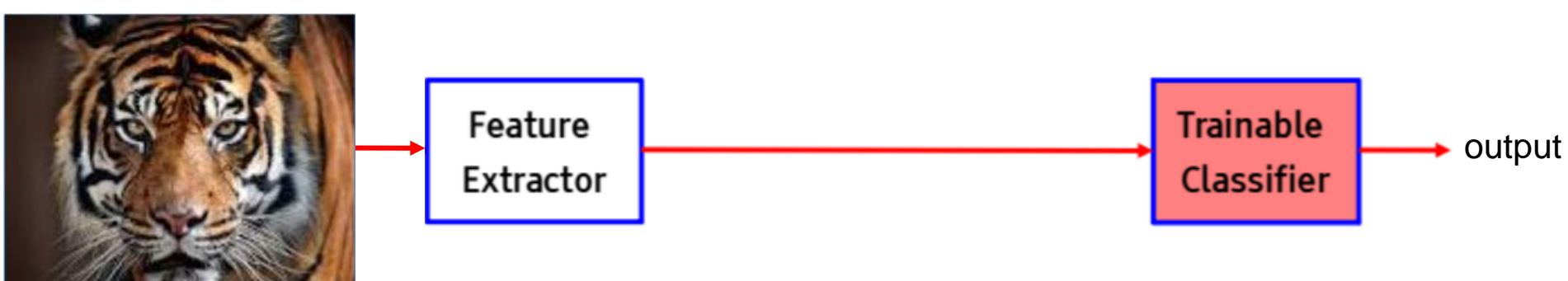
- La critique majeure des MLP c'est le manque d'une assise mathématique solide
- Signification des valeurs des poids finaux ? Que codent les nœuds cachés
- Ce comportement est assez semblable au fonctionnement des vraies neurones



- 
- Introduction
    - Neurone artificiel
    - Fonctions d'activation
  - Classificateurs de type perceptron monocouche
    - Limite du perceptron
    - Apprentissage du Perceptron et convergence
    - L'algorithme d'apprentissage du perceptron
    - Perceptron vs SVM
  - Réseaux de neurones multicouches (MLP)
    - Représentation de XOR
    - Apprentissage d'un MLP: Backpropagation – Rétropropagation
      - Principe
      - Minimisation du risque
      - Descente de gradient
      - Algorithme de Backpropagation
      - Performance de la Backpropagation
      - Exemple
      - Condition d'arrêt
  - Réseaux pour l'apprentissage profond (DL)
    - Principe
    - Réseaux de neurones convolutifs
      - Convolution
      - Pooling

# Apprentissage classique

- Les modèles traditionnels de reconnaissance de formes utilisent des « hand-crafted features » (caractéristiques extraites à la main)



- Processus fastidieux et coûteux
- Les features dépendent beaucoup des applications et ne peuvent pas être facilement généralisés à d'autres applications.

# Apprentissage profond



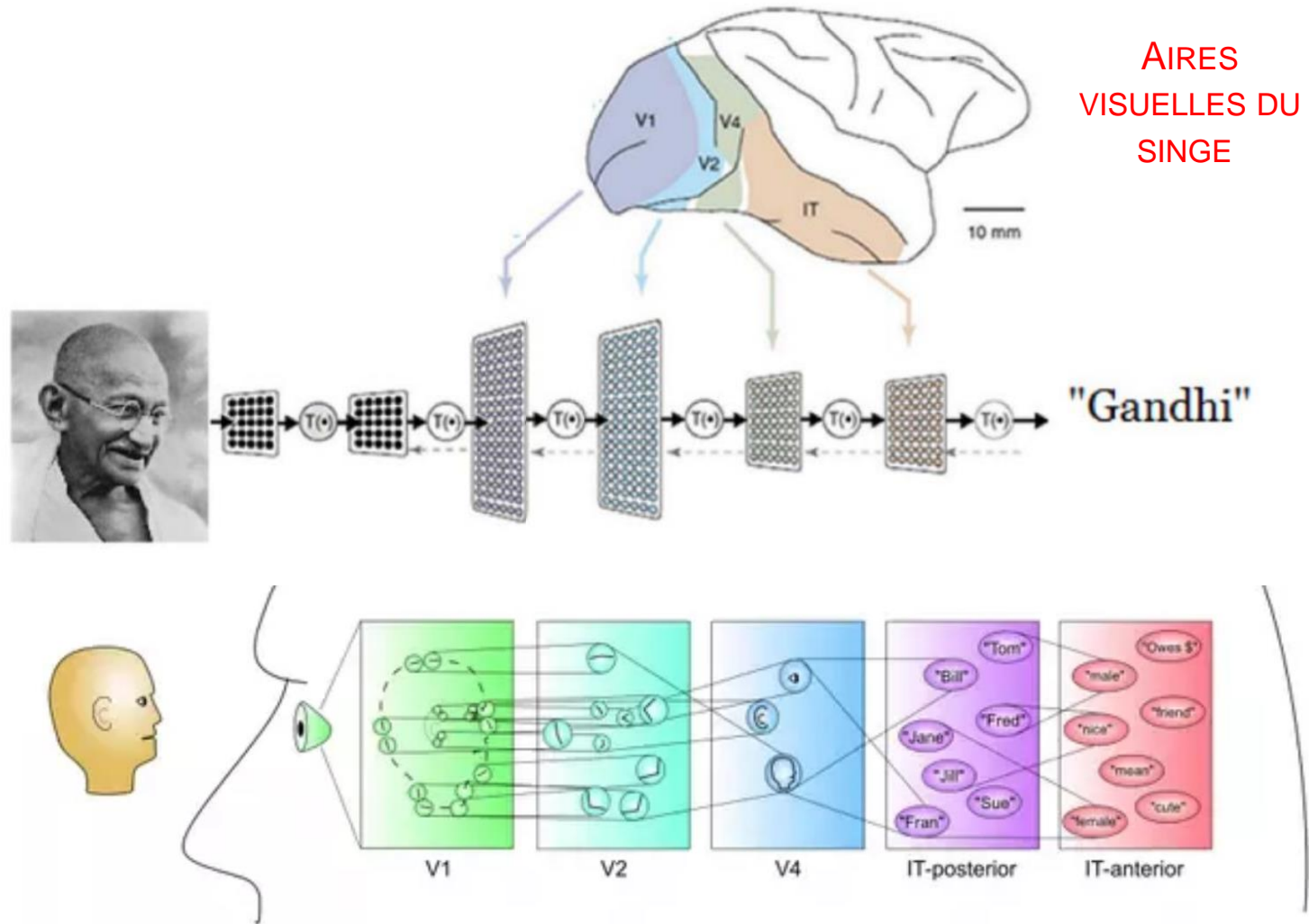
Image 255\*255 RGB

Impossible à représenter par MLP classique  
car ça suppose en input  $255*255*3=195075$   
neurones en entrée

L'apprentissage en profondeur (Deep learning) cherche à  
comprendre automatiquement des concepts en analysant les  
données à un haut niveau d'abstraction

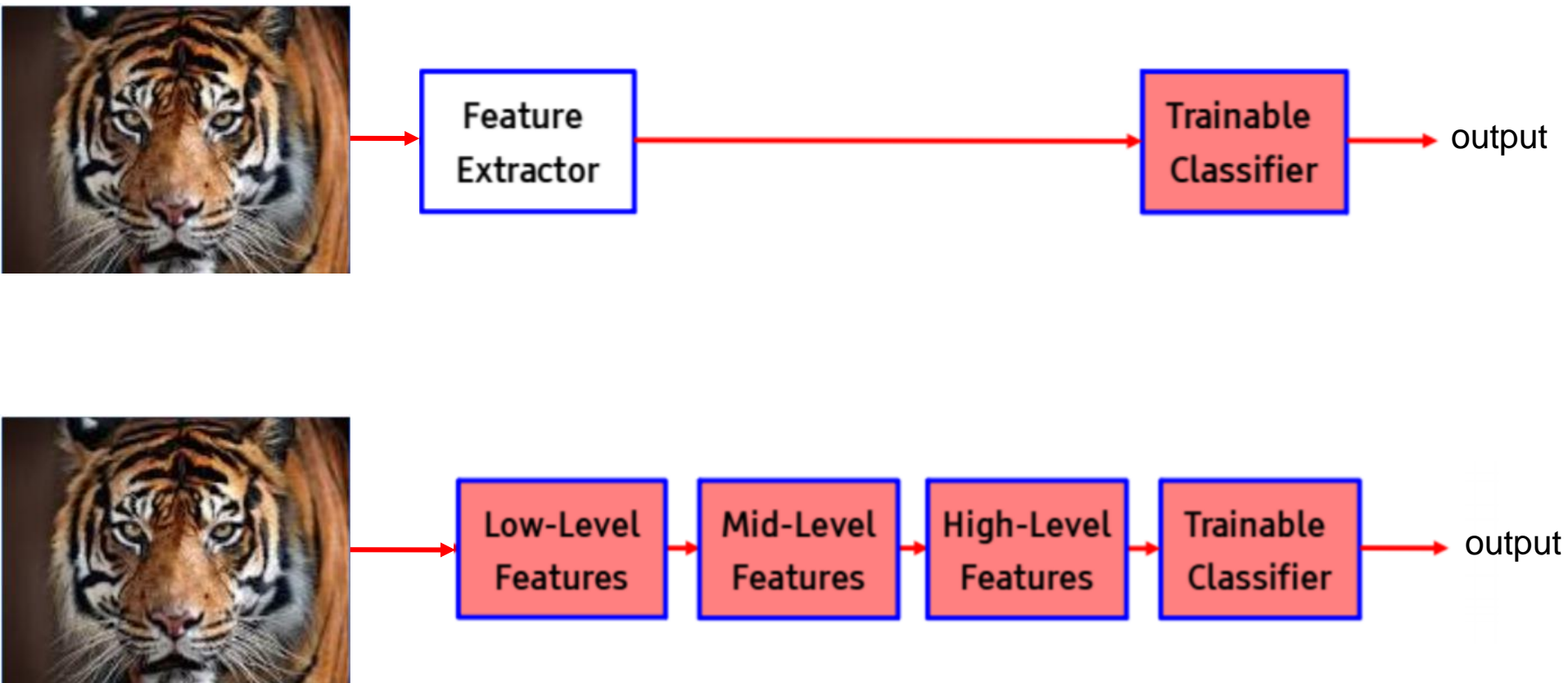
>> Les algorithmes phares du Deep Learning : Les réseaux de neurones  
convolutifs, aussi appelés CNN ou ConvNet (Convolutional Neural Network)

Un **réseau de neurones convolutifs** est un type de réseau de neurones artificiels dans lequel le motif de connexion entre les neurones est inspiré par le cortex visuel des animaux





# Apprentissage classique vs profond



# Deep learning

- Un des pionniers de cette idée: le français **Yann Le Cun** qui a commencé à développer ces outils en 90
- Il n'a pas réussi à convaincre la communauté de l'IA jusqu'à 2012 où de Deep Learning a remporté la compétition : **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** qui évalue les algorithmes de détection d'objets et de classification d'images à grande échelle.
  - Il s'agit d'un **réseau de neurones convolutif** appelé AlexNet (SuperVision group).
- ...Depuis **Yann Le Cun** est directeur du laboratoire IA de FB



Conférence de Yann Le Cun :

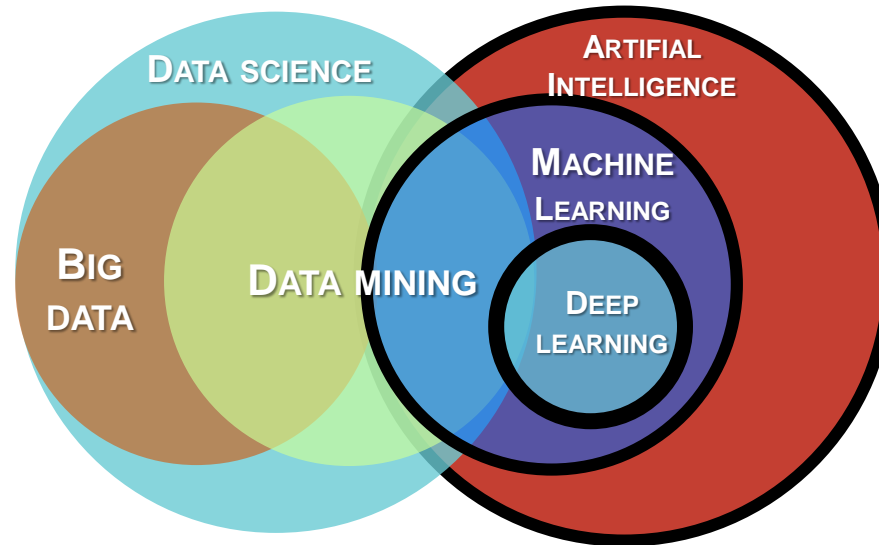
[https://www.youtube.com/watch?v=RgUcQceqC\\_Y&t=1700s](https://www.youtube.com/watch?v=RgUcQceqC_Y&t=1700s)

| 2010           |     | 2011            |     |
|----------------|-----|-----------------|-----|
| 1. NEC         | 28% | 1. XRCE         | 26% |
| 2. XRCE        | 34% | 2. Uv A         | 31% |
| 3. ISIL        | 45% | 3. ISI          | 36% |
| 4. UCI         | 47% | 4. NII          | 50% |
| 5. Hminmax     | 54% |                 |     |
| 2012           |     | 2013            |     |
| 1. SuperVision | 16% | 1. Clarifai     | 12% |
| 2. ISI         | 26% | 2. NUS          | 13% |
| 3. VGG         | 27% | 3. ZeilerFergus | 13% |
| 4. XRCE        | 27% | 4. A.Howard     | 13% |

%plus le pourcentage est faible plus la méthode est bonne  
<http://image-net.org/challenges/LSVRC/2012/>



# Deep learning



- Le **DL** a révolutionné le domaine de l'IA en permettant à la machine d'apprendre de façon totalement autonome
- Le **DL** est une méthode très gourmande (A titre de comparaison, un grand maître du Jeu de Go aura joué environ 10000 parties dans sa vie quand l'ordinateur aura besoin de jouer 409600000 parties pour atteindre le même niveau)



# Pourquoi ça marche ?

Le DL est devenu possible (alors que pendant 20 ans tout le monde pensait que ça ne marcherait jamais) grâce à :

1. La progression des algorithmes (théoriquement)
2. l'accélération des vitesses de calcul (progrès des processeurs et des cartes graphiques)
3. l'explosion de la quantité de données disponibles (big data) surtout en reconnaissance d'image
  - En 2007 le Stanford Vision Lab développe un agrégateur d'images où sont consignés et étiquetés quelques millions de photos : IMAGENET.
  - En 2010, IMAGENET regroupe 15 000 000 d'images toutes catégorisées en fonction de leurs caractéristiques propres (véhicules, animaux,...).

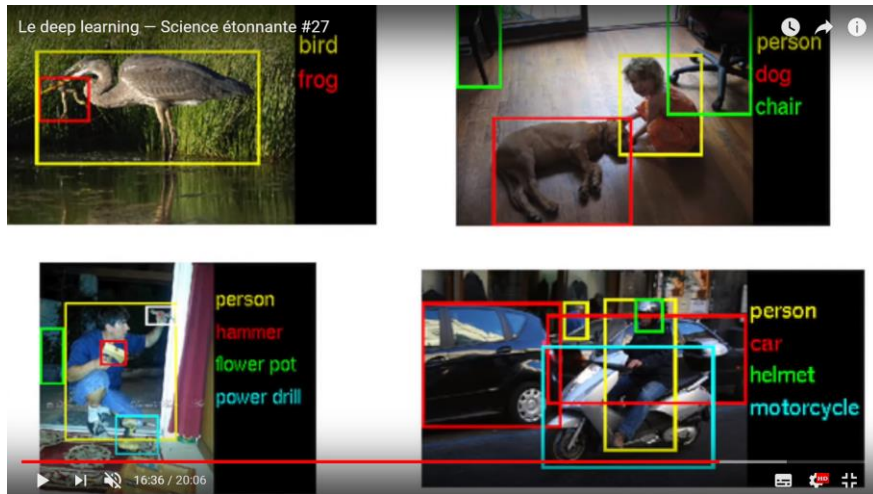


<http://www.image-net.org/about-stats>

# Applications

Le DL ne se limite pas à la classification d'images

## Reconnaitre les objets



On peut même analyser des scènes...



A person riding a motorcycle on a dirt road.



Two dogs play in the grass.

# Réseaux de neurones convolutifs (ludique)

## Classification

[Click for a Quick Example](#)



Maximally accurate

Maximally specific

cat

1.79306

feline

1.74269

domestic cat

1.70760

tabby

0.94807

domestic animal

0.76846

CNN took 0.064 seconds.

Try out a live demo at  
<http://demo.caffe.berkeleyvision.org/>

# Réseaux de neurones convolutifs (ludique)

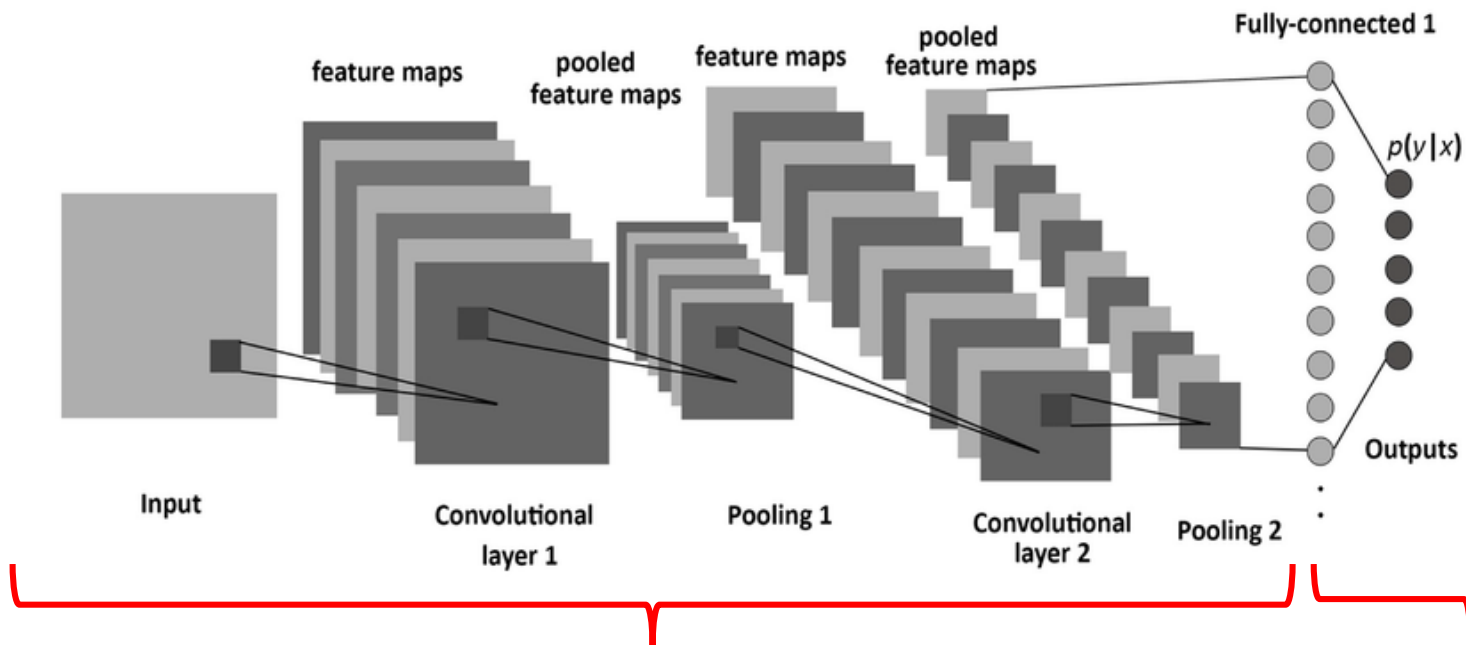
- Quick Draw: application proposée par google et qui utilise DL (Réseaux de neurones convolutifs)



<https://quickdraw.withgoogle.com/>

# Réseaux de neurones convolutifs

- Dans un CNN on distingue 2 parties:
  - Partie convolutive qui fonctionne comme un extracteur de caractéristiques des images et se base sur une succession de deux opérations successives (**convolution** / **pooling**)
  - La deuxième partie (connectée à la première) est un MLP



# Convolution (1)



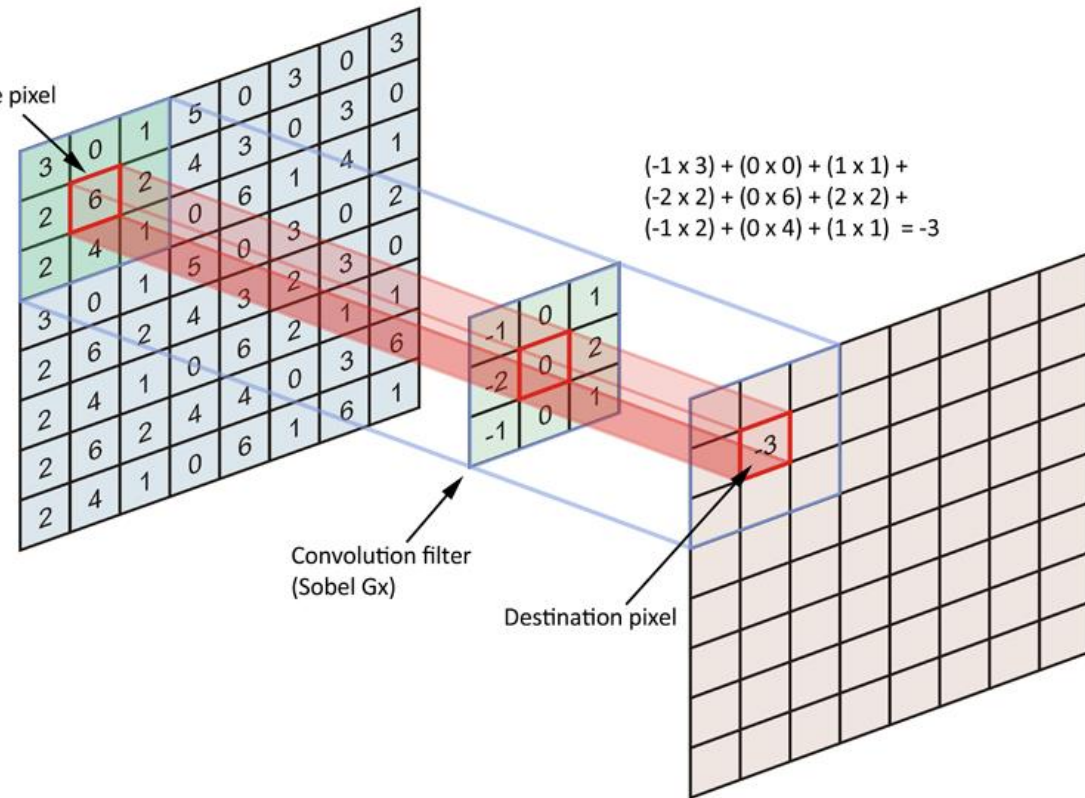
Image 255\*255 RGB

Impossible à représenter par MLP classique  
car ça suppose en input  $255*255*3=195075$   
neurones en entrée

- **Idée** est de créer plusieurs filtres de cette images qui se focalisent chacun sur un pattern
- Convolutionner l'image = appliquer plusieurs filtres

# Convolution (2)

- L'opération de convolution consiste à remplacer la valeur de chaque pixel par une combinaison linéaire (sommation de multiplications) de ses voisins
- Le filtre de convolution (*kernel*) est une matrice de valeurs (généralement 3 par 3).
- Un filtre sert à faire ressortir certaines caractéristiques d'une image donnée (couleur, contour, luminosité, netteté, etc...).
- Exemple Filtres de **Sobel** – **Prewitt** (vertical / horizontal)



*Sobel Vertical*

|    |    |    |
|----|----|----|
| 1  | 2  | 1  |
| 0  | 0  | 0  |
| -1 | -2 | -1 |

*Sobel Horizontal*

|    |   |   |
|----|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

*Prewitt vertical*

|    |   |    |
|----|---|----|
| -1 | 0 | +1 |
| -1 | 0 | +1 |
| -1 | 0 | +1 |

*Prewitt Horizontal*

|    |    |    |
|----|----|----|
| +1 | +1 | +1 |
| 0  | 0  | 0  |
| -1 | -1 | -1 |

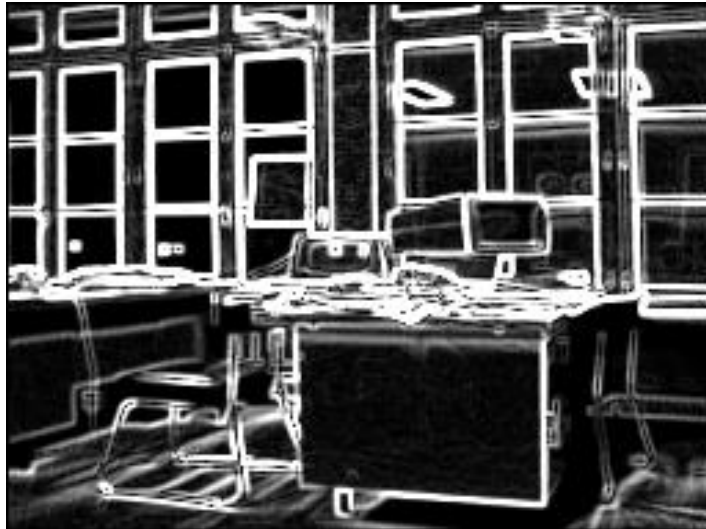


# Exemple

Original



Sobel

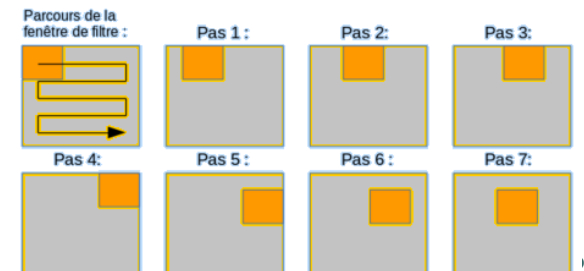
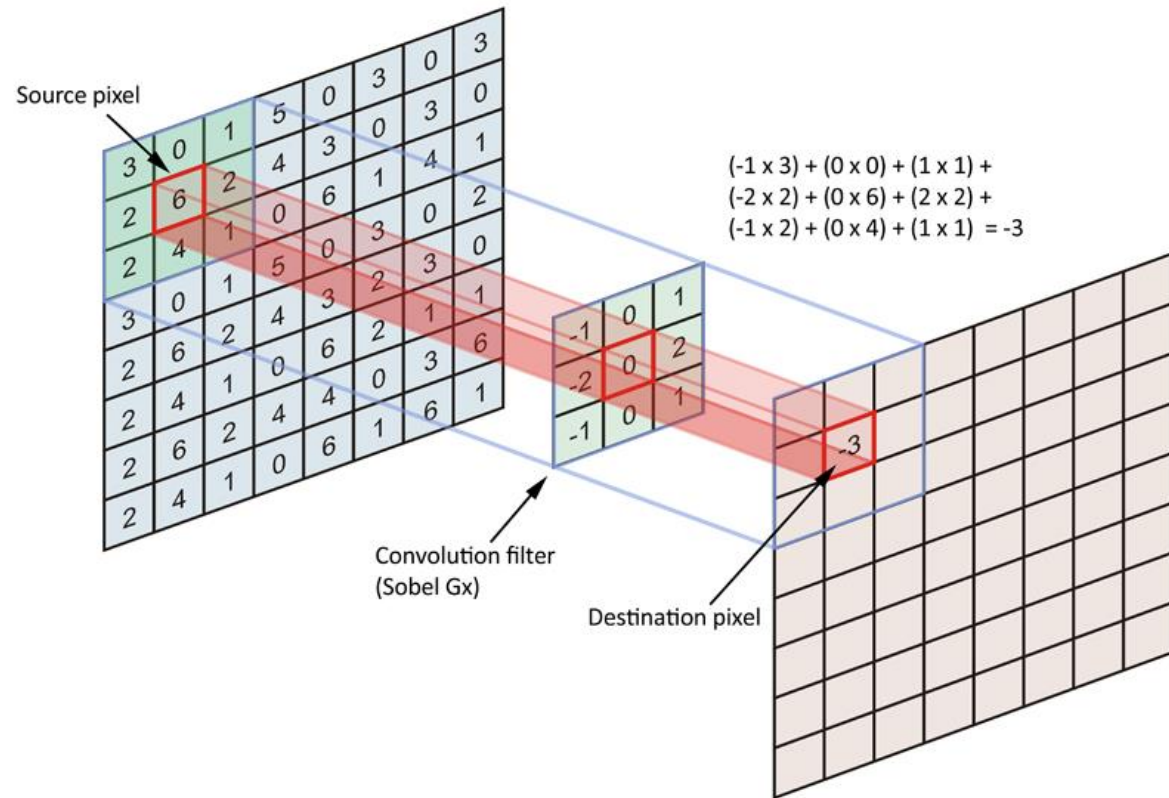


Prewitt



# Convolution (3)

- Le filtre va être déplacé par pas successifs sur l'ensemble de l'image.
- Pour les valeurs des bords plusieurs solutions (remplacer par 0, garder les valeurs initiales (effet miroir), convolution partielle avec les voisins existants etc.)
- Par convention pratique la taille de l'image initiale est la même que la taille de l'image convolutionnée mais elle peut être réduite



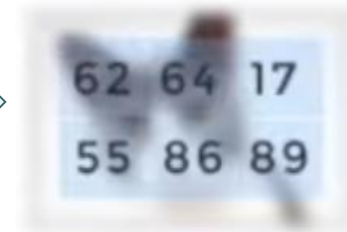
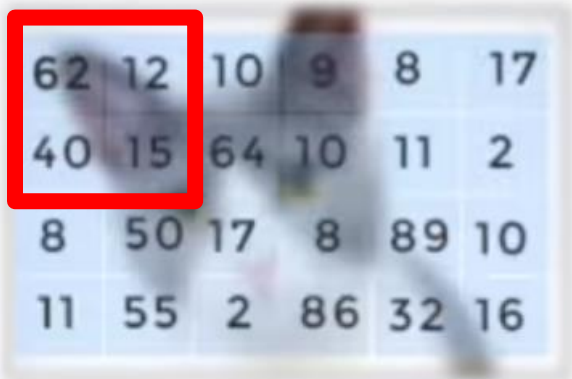
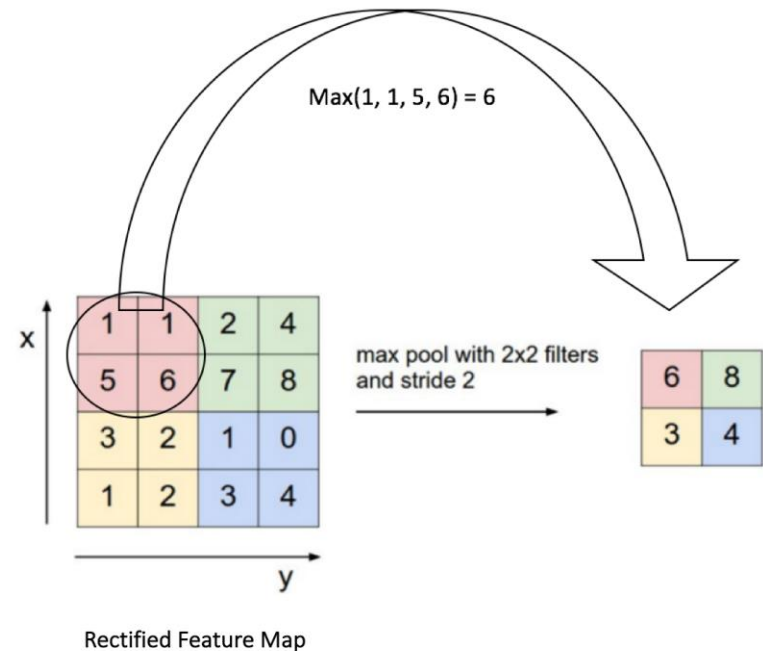
# Convolution (4)

- L'usage de filtres est la base dans un réseau à convolution.
- En pratique, lors de la phase d'entraînement, plusieurs filtres sont testés avec des valeurs différentes. Les meilleures, par rapport au jeu d'entraînement, sont retenues.
- Le nombre de filtres déterminera le nombre de caractéristiques détectées. Ce nombre est appelé la profondeur. C'est à dire que si 10 filtres seront appliqués à une image donnée, la valeur de sa profondeur sera de 10.
- Il est préférable de ne pas utiliser un nombre de filtres trop important car cela peut entraîner un sur-apprentissage.



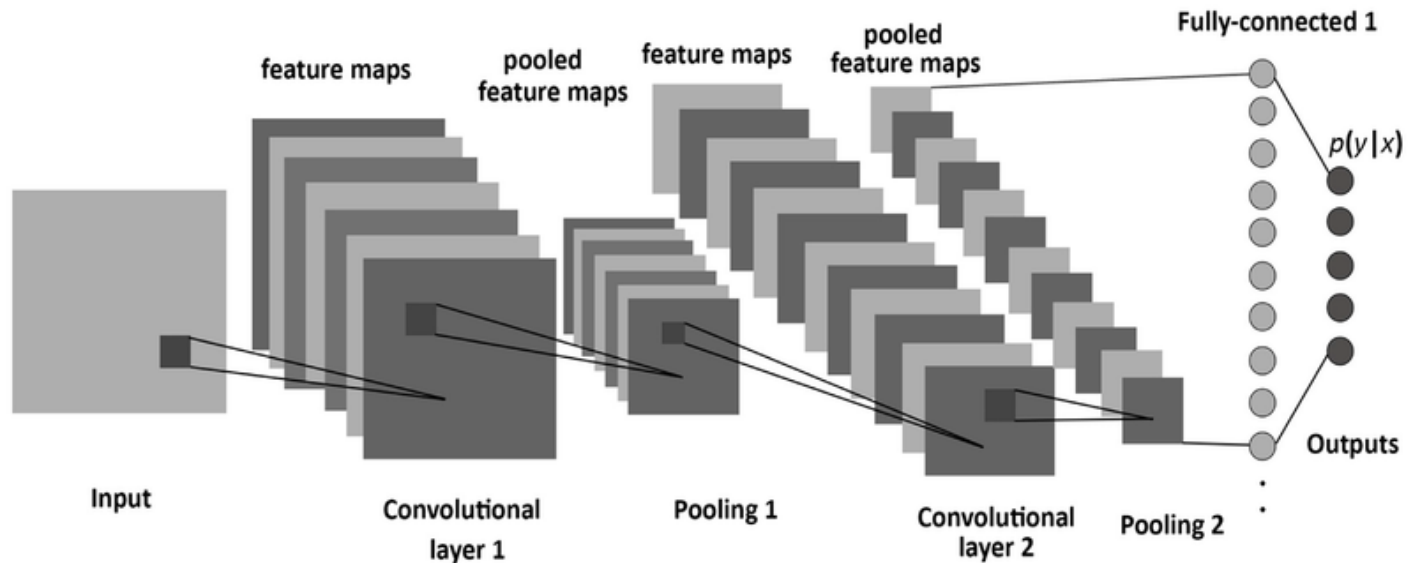
# Pooling

- Pooling : extraire les valeurs importantes des pixels, il permet de **réduire une image** tout en conservant les caractéristiques pertinentes.
- Il existe plusieurs méthodes de pooling:
  - **Max Pooling** : Max de toutes les valeurs recouvertes par la zone du pooling (tuile)
  - **Average Pooling** : Moyenne de toutes les valeurs recouvertes par la tuile
  - **Pooling stochastique** : Ne retient qu'une seule valeur, comme le "Max Pooling", mais en se basant sur une méthode probabiliste
- La méthode la plus utilisée est le Max Pooling (on conserve les valeurs remarquables de l'image)



# La classification

- Une fois que les étapes de convolutions ont opéré, les patterns obtenues en sortie sont injectés comme données d'entrée dans un réseau neuronal classique en **mettant à plat** toutes les données dans un seul vecteur.
- Toutes les "images" !! sont ainsi mises bout à bout dans ce même vecteur.
- Ce vecteur permettra la création d'une première couche de neurones entièrement connectée. C'est à dire que chacune des valeurs de ce vecteur sera connectée aux neurones de la première couche du réseau permettant la classification de l'image.

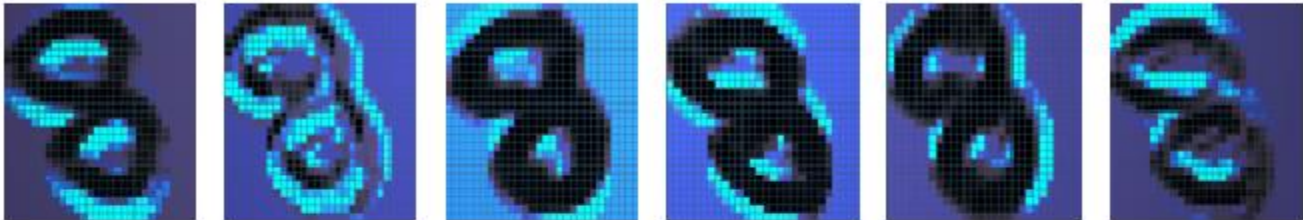




# Exemple



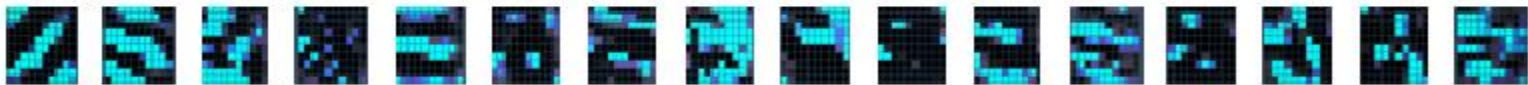
Image d'entrée



1ère convolution



1er Pooling



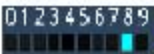
2ème convolution



2ème Pooling



Couches connectées

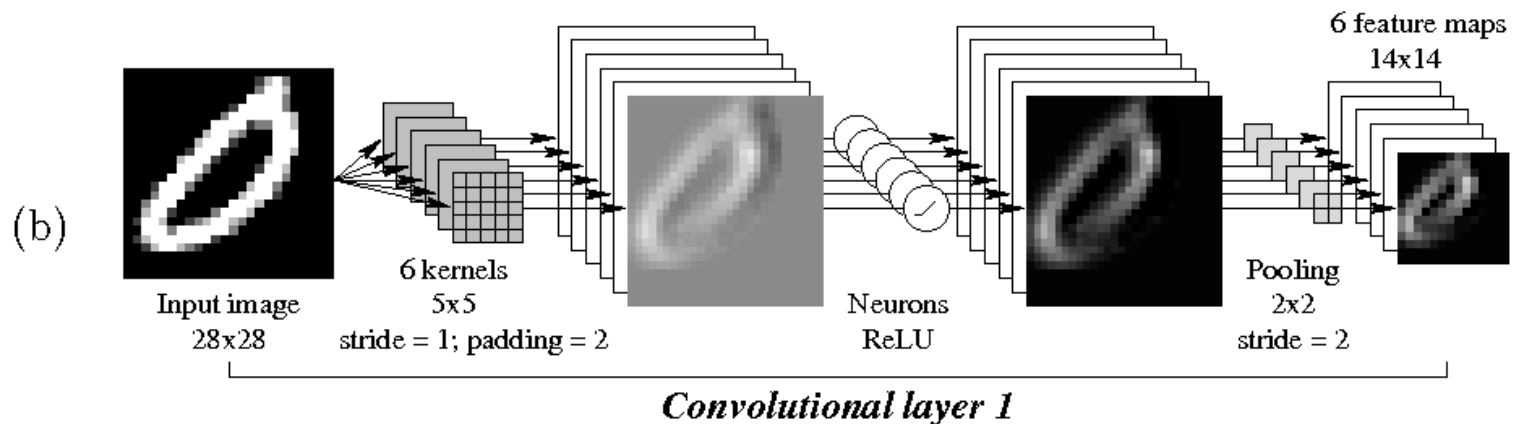
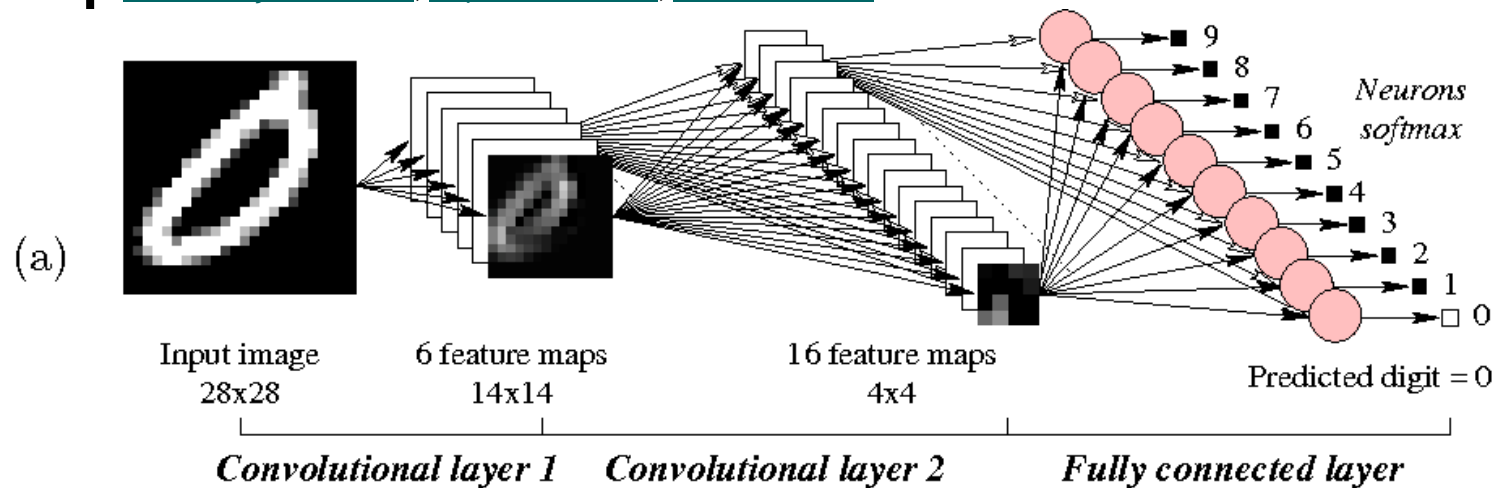


Résultat de la classification

# Exemple MNIST

Steganalysis via a Convolutional Neural Network using Large Convolution Filters

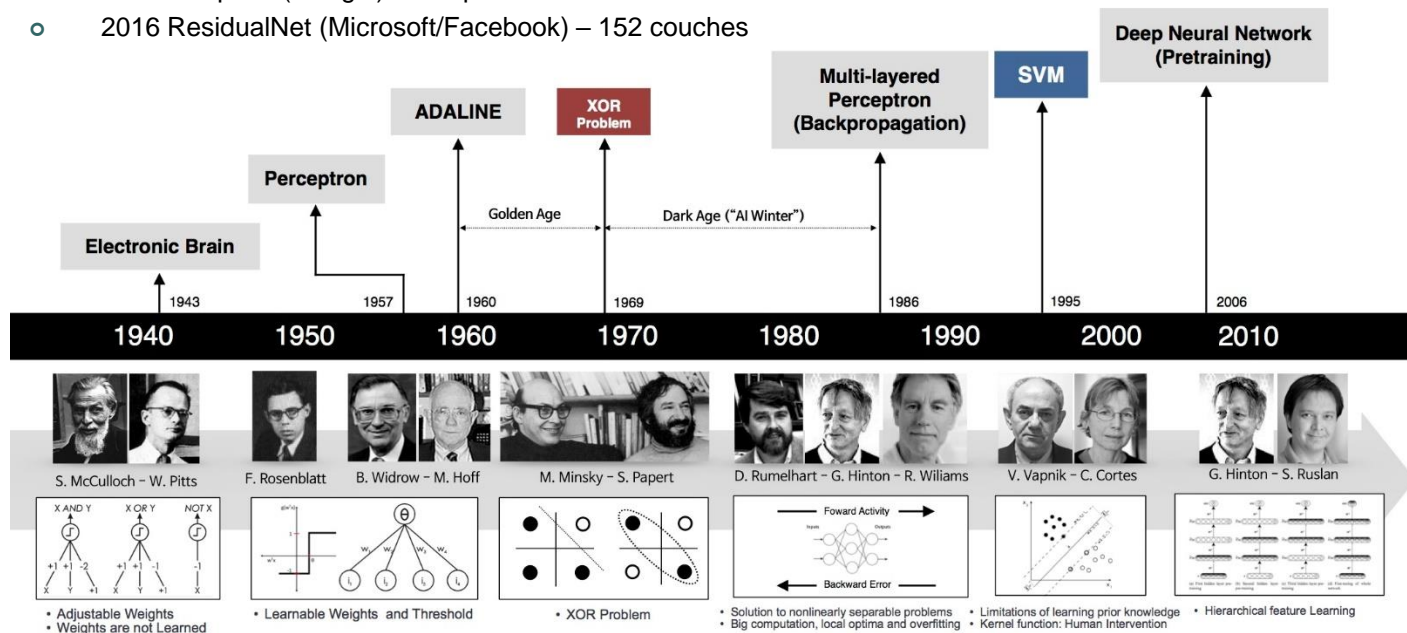
[Jean-François Couchot](#), [Raphaël Couturier](#), [Michel Salomon](#)



A convolutional neural network for the MNIST problem: global architecture (a) and detailed view of the first convolutional layer (b).

# Du perceptron a l'apprentissage profond

- 1957 (Rosenblatt) Perceptron
- 1960 (Widrow, Hoff) ADALINE
  - 1969 (Minsky, Papert) Probleme XOR – impossibilité de classer des configurations non linéairement séparables
  - abandon (financier) des recherches sur les RNA
- [1967 - 1982] :
  - Mise en sommeil des recherches sur les RNA. Elles continuent sous le couvert de domaines divers. Grossberg, Kohonen, Anderson, ...
- 1982 (Hopfield) :
  - modèle des verres de spins
- 1986 (Rumelhart et. al) MLP et backpropagation
- 1992 (Vapnik et. al) SVM
- 1998 (LeCun et. al) LeNet
- 2010 (Hinton et. al) Deep Neural Networks
- 2012 (Krizhevsky, Hinton et. al) t, ILSVRC'2012, GPU – 8 couches AlexNe
- 2014 GoogleNet – 22 couches
- 2015 Inception (Google) – Deep Dream
- 2016 ResidualNet (Microsoft/Facebook) – 152 couches







# Quelques remarques

- Outils
  - WEKA <http://www.cs.waikato.ac.nz/ml/weka/>.
  - scikit-learn with python
- challenge in 2018 involves classifying 3D objects using natural language

Artificial Intelligence: A Modern Approach  
(Third edition) by Stuart Russell and Peter Norvig  
(chapter 18.7) <http://aima.cs.berkeley.edu/>

S. Haykin, Neural Networks and Learning Machines - Prentice Hall, 2009.

