Programmation et IA

Cours 8

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-ia-25

Plan

- fonctions d'activation
- neurone et perceptron
- réseau de neurones
- rétro-propagation
- exemples

Machine Learning [Andrew Ng] http://cs229.stanford.edu
http://coursera.org/share/5aa61ab89328fc47de71f57999bf14b2

Deepmath [Bodin & Récher] http://exo7.emath.fr/cours/livre-deepmath.pdf

Rappels mathématiques

- dérivée de la composition de fonctions (*chain rule*)

$$(f(g(x))' = f'(g(x))\, g'(x)$$

$$\frac{dz}{dx} = \frac{dz}{dy}\,\frac{dy}{dx}$$ [z dépend de y] [y dépend de x]

- dérivée d'une fonction de plusieurs arguments

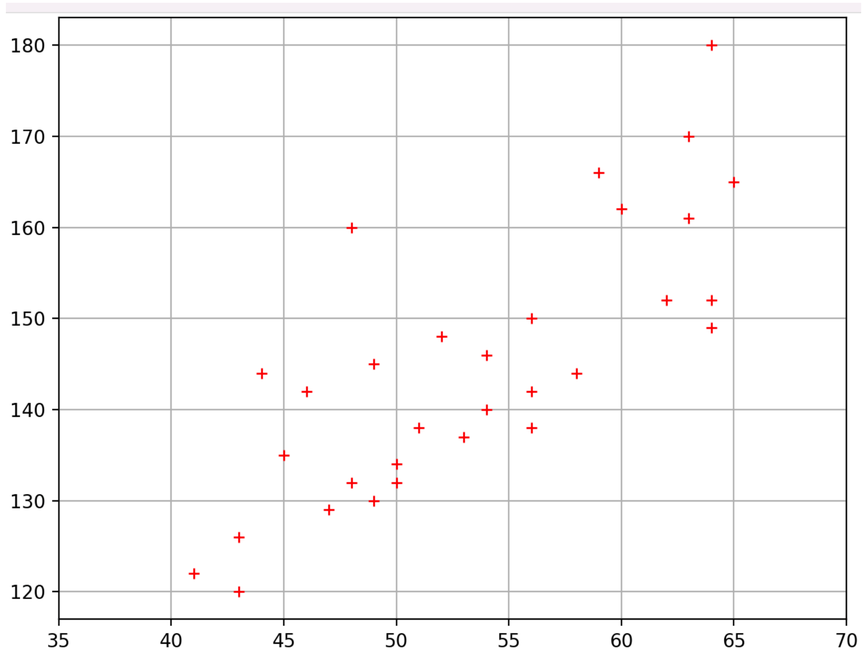
$$\frac{\partial}{\partial t} f(x,y) = \frac{\partial}{\partial x} f(x,y)\,\frac{\partial x}{\partial t} + \frac{\partial}{\partial y} f(x,y)\,\frac{\partial y}{\partial t}$$

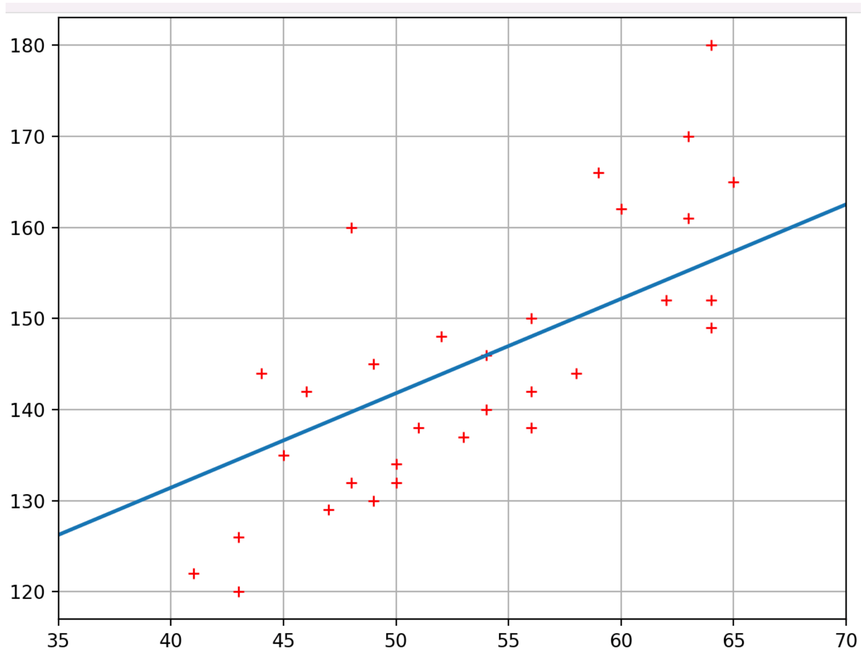
$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x}\,\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\,\frac{\partial y}{\partial t}$$ [f dépend de x et y] [x et y dépendent de t]

régression linéaire

Régression linéaire

- on cherche une approximation linéaire du jeu de données $\left(x^{(i)}, y^{(i)}\right)$





$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix} \qquad x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}$$

$$f(x) = \theta_0 + \theta_1 x_1 \cdots \theta_d x_d = \theta^T x$$

Régression linéaire

- on cherche une approximation linéaire du jeu de données $\left(x^{(i)}, y^{(i)}\right)$
- dans l'exemple de la pression artérielle en fonction de l'âge, on a $d = 1$
- dans l'exemple du taux de carbone en fonction du volume et du poids, on a $d = 2$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix} \qquad x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}$$

$$f(x) = \theta_0 + \theta_1 x_1 \cdots \theta_d x_d = \theta^T x$$

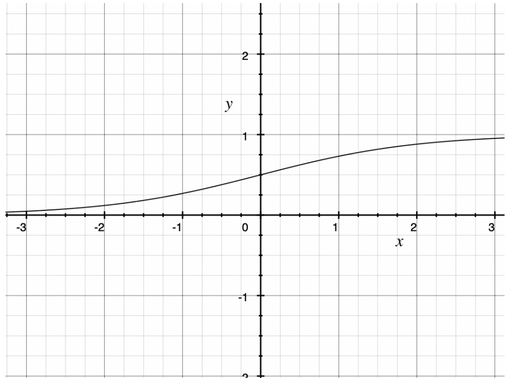
Classification - régression logistique

- pour séparer par une **droite** les zones rouge et bleue, on cherche θ tel que :

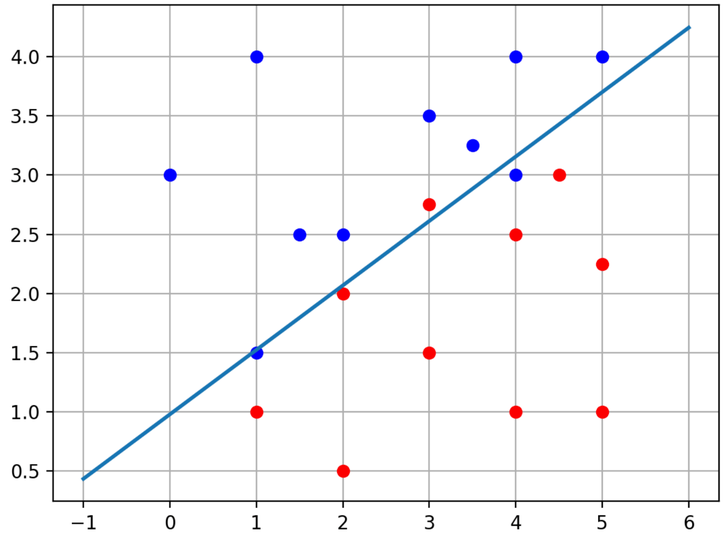
$$\sigma(\theta^T x) = \sigma(\theta_0 + \theta_1 x_1 + \cdots \theta_d x_d) > \frac{1}{2} \quad \Longleftrightarrow \quad \theta^T x = \theta_0 + \theta_1 x_1 + \cdots \theta_d x_d > 0$$

avec la fonction sigmoïde définie par:

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



σ

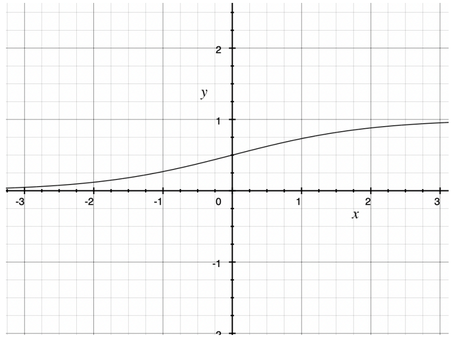


Fonctions d'activation

- fonctions d'activation possibles

la fonction sigmoïde

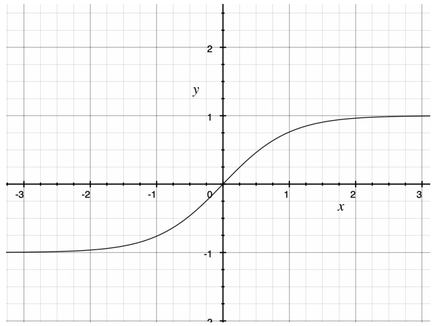
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



σ

la tangente hyperbolique

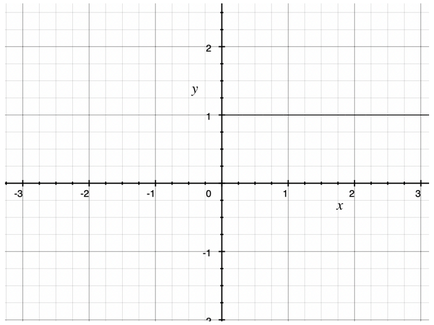
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



tanh

la fonction marche de Heaviside

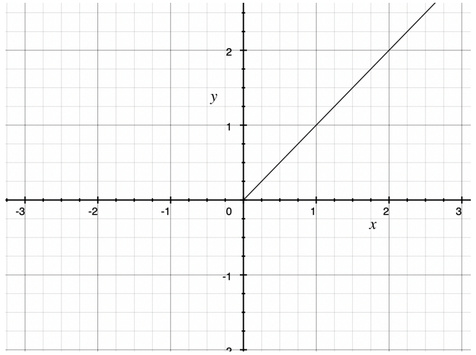
$$\mathrm{H}(x) = \begin{cases} 1 \text{ si } x \geq 0 \\ 0 \text{ sinon} \end{cases}$$



H

la fonction linéaire rectifiée

$$\mathrm{ReLU}(x) = max(0, x)$$



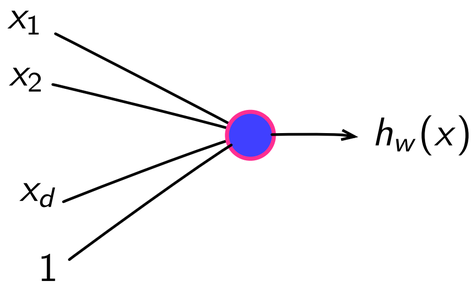
ReLU

- certaines ne sont pas dérivables en 0.. ce n'est pas grave si on n'utilise pas 0

perceptron

Neurone

- neurone à *d* entrées et 1 sortie

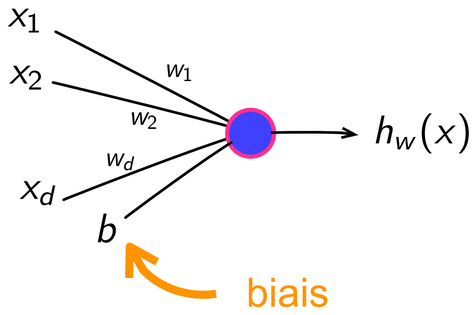


$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \\ 1 \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix}$$

$$h_w(x) = f(w^T x) = f\left(\sum_{i=1}^{d} w_i x_i + b\right)$$

fonction d'activation

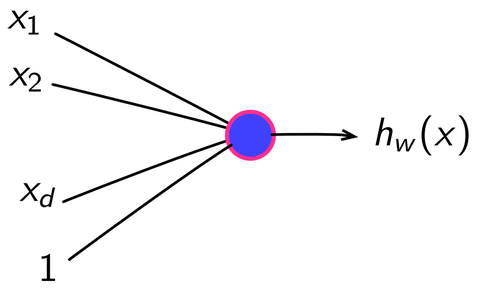
- autre présentation graphique



- avec 1 neurone, on fait la régression (ou classification) linéaire

Neurone

attention: changement de notations



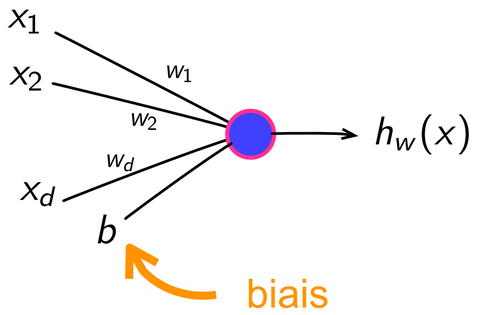
$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \\ 1 \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix}$$

$$h_w(x) = f(w^T x) = f(\sum_{i=1}^{d} w_i x_i + b)$$

fonction d'activation

après

- autre présentation graphique



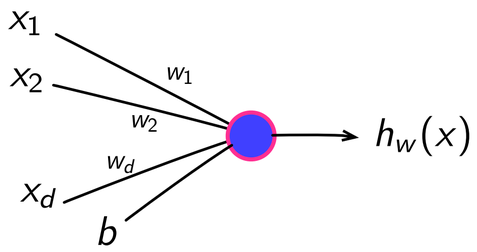
$$x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$$

$$f(x) = \theta_0 + \theta_1 x_1 \cdots \theta_d x_d = \theta^T x$$

avant

Neurone - perceptron

- *d* entrées et 1 sortie

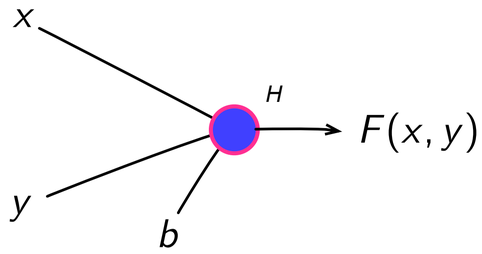


$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \\ 1 \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix}$$

$$h_w(x) = f(w^T x) = f\left(\sum_{i=1}^{d} w_i x_i + b\right)$$

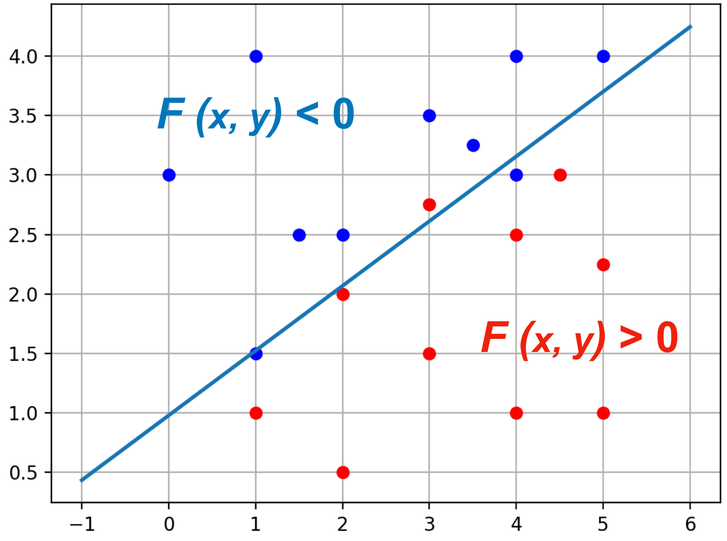
fonction d'activation

- classification linéaire avec $f = H$ (Heaviside)



$$\mathrm{X} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad w = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

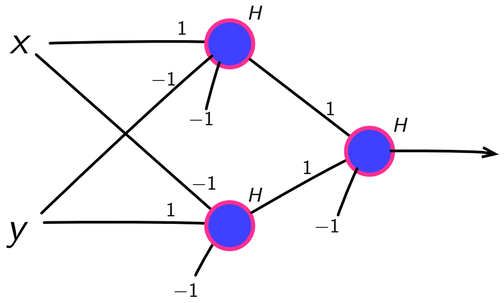
$$F(x, y) = h_w(\mathrm{X}) = f(w^T \mathrm{X}) = x - 2y + 1$$



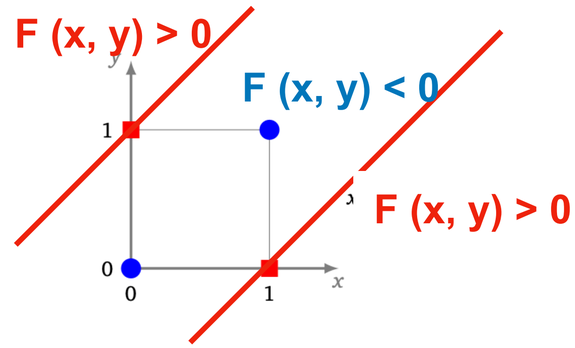
Neurone - perceptron

- on ne peut pas faire le OU exclusif (XOR) avec 1 seul neurone

- il faut plus de neurones



$$H(H(x-y-1)+H(-x+y-1)-1)$$



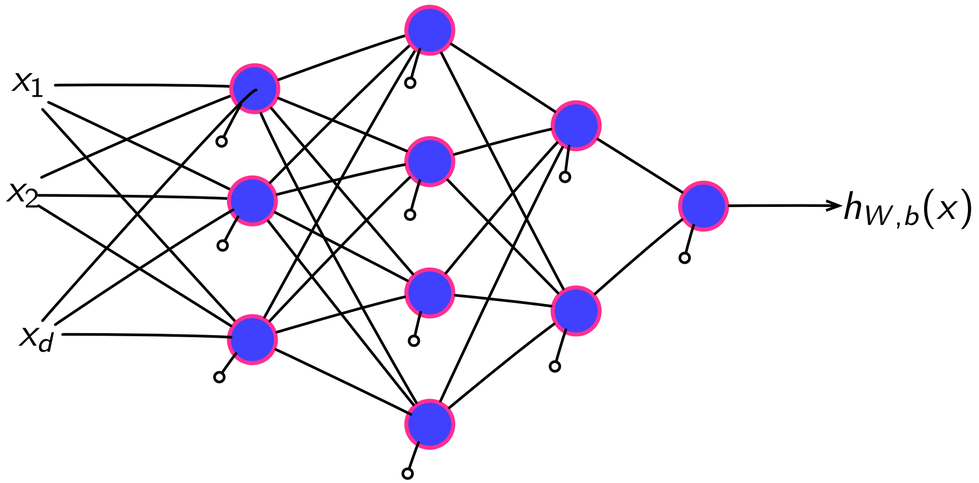
XOR

x\	0	1
0	0	1
1	1	0

réseau de neurones

Réseau de neurones

• exemple de réseau avec d entrées, 1 sortie, 5 couches $(n_\ell = 5)$ et une fonction d'activation uniforme f



$$x = a^{[1]}$$

$$z^{[\ell+1]} = W^{[\ell]} a^{[\ell]} + b^{[\ell]}$$

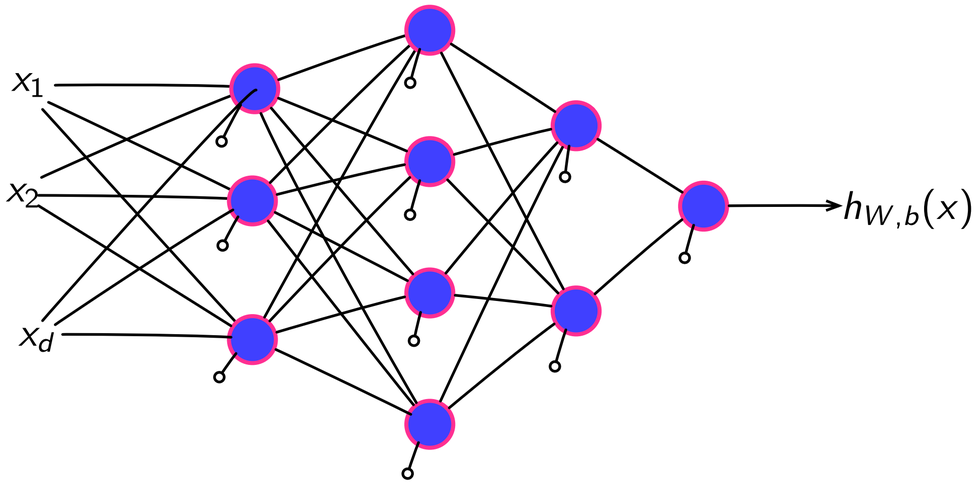
$$a^{[\ell+1]} = f(z^{[\ell+1]})$$

$$h_{W,b}(x) = a^{[n_\ell]} \qquad (1 \leq \ell \leq n_\ell)$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \qquad W^{[\ell]} = \begin{bmatrix} w_{11}^{[\ell]} & w_{12}^{[\ell]} & \cdots & w_{1d_\ell}^{[\ell]} \\ w_{21}^{[\ell]} & w_{22}^{[\ell]} & \cdots & w_{2d_\ell}^{[\ell]} \\ \vdots & & & \vdots \\ w_{d_{\ell+1}1}^{[\ell]} & w_{d_{\ell+1}2}^{[\ell]} & \cdots & w_{d_{\ell+1}d_\ell}^{[\ell]} \end{bmatrix} \qquad b^{[\ell]} = \begin{bmatrix} b_1^{[\ell]} \\ b_2^{[\ell]} \\ \vdots \\ b_{d_\ell}^{[\ell]} \end{bmatrix} \qquad z^{[\ell]} = \begin{bmatrix} z_1^{[\ell]} \\ z_2^{[\ell]} \\ \vdots \\ z_{d_\ell}^{[\ell]} \end{bmatrix} \qquad a^{[\ell]} = \begin{bmatrix} a_1^{[\ell]} \\ a_2^{[\ell]} \\ \vdots \\ a_{d_\ell}^{[\ell]} \end{bmatrix}$$

Réseau de neurones

• exemple de réseau avec d entrées, 1 sortie, 5 couches $(n_\ell = 5)$ et une fonction d'activation uniforme *f*



$$x = a^{[1]}$$

$$z^{[\ell+1]} = W^{[\ell]} a^{[\ell]} + b^{[\ell]}$$

$$a^{[\ell+1]} = f(z^{[\ell+1]})$$

$$h_{W,b}(x) = a^{[n_\ell]}$$

$d = d_1$ $\quad n_\ell = 5$ $\quad W^{[1]} : 3 \times d$ $\quad W^{[2]} : 4 \times 3$ $\quad W^{[3]} : 2 \times 4$ $\quad W^{[4]} : 1 \times 2$

• 31 paramètres quand *d* = 3 !!

Calcul de l'erreur

- on dispose donc d'un jeu de n données $(x^{(i)},\ y^{(i)})$ $\qquad (1 \leq i \leq n)$

- fonction d'erreur (ou coût)

$$J(W, b; x, y) = \tfrac{1}{2}(h_{W,b}(x) - y)^2$$

$$J(W, b) = \tfrac{1}{n} \sum_{i=1}^{n} J(W, b; x^{(i)}, y^{(i)}) \ + \ \tfrac{\lambda}{2} \sum_{\ell=1}^{n_\ell - 1} \sum_{i=1}^{d_\ell} \sum_{j=1}^{d_{\ell+1}} (w_{ij}^{[\ell]})^2$$

moyenne

correction pour équilibrer l'influence des poids [*weight decay*]

- il faut deviner les nombreux paramètres par **apprentissage supervisé**

- descente du gradient sur J pour calculer tous les poids !

Calcul des dérivées

- descente du gradient en calculant les dérivées partielles par rapport aux poids

$$w_{ij}^{[\ell]} = w_{ij}^{[\ell]} - \alpha \frac{\partial}{\partial w_{ij}^{[\ell]}} J(W, b)$$

$$\frac{\partial}{\partial w_{ij}^{[\ell]}} J(W, b) = \left[\frac{1}{n} \sum_{k=1}^{n} \frac{\partial}{\partial w_{ij}^{[\ell]}} J(W, b;\ x^{(k)}, y^{(k)}) \right] + \lambda\, w_{ij}^{[\ell]}$$

$$b_i^{[\ell]} = b_i^{[\ell]} - \alpha \frac{\partial}{\partial b_i^{[\ell]}} J(W, b)$$

$$\frac{\partial}{\partial b_i^{[\ell]}} J(W, b) = \frac{1}{n} \sum_{k=1}^{n} \frac{\partial}{\partial b_i^{[\ell]}} J(W, b;\ x^{(k)}, y^{(k)})$$

- comment calculer ces nombreuses dérivées partielles ?

$$J(W, b; x, y) = \tfrac{1}{2}(h_{W,b}(x) - y)^2$$

$$x = a^{[1]}$$

$$z^{[\ell+1]} = W^{[\ell]} a^{[\ell]} + b^{[\ell]}$$

$$a^{[\ell+1]} = f(z^{[\ell+1]})$$

$$h_{W,b}(x) = a^{[n_\ell]}$$

$$\frac{\partial}{\partial w_{ij}^{[\ell]}} J(W, b;\ x, y)$$

$$\frac{\partial}{\partial b_i^{[\ell]}} J(W, b;\ x, y)$$

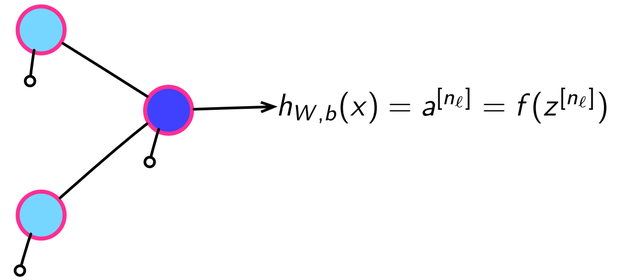
Rétro-propagation

• retour aux cours (élémentaires) de mathématiques en calculant d'abord la dérivée par rapport aux $z_i^{[\ell]}$

[Seppo Linnainmaa 1970, Yann LeCun 1985]

$$\begin{aligned}
\delta_i^{[n_\ell]} &= \frac{\partial}{\partial z_i^{[n_\ell]}} J(W, b; x, y) \\
&= \frac{\partial}{\partial z_i^{[n_\ell]}} \frac{1}{2}(h_{W,b}(x) - y)^2 \\
&= (a_i^{[n_\ell]} - y) \frac{\partial a_i^{[n_\ell]}}{\partial z_i^{[n_\ell]}} \\
&= (a_i^{[n_\ell]} - y) f'(z_i^{[n_\ell]})
\end{aligned}$$

dernière couche



• hypothèses

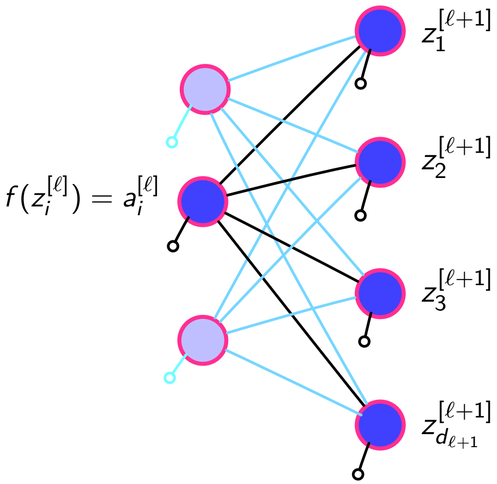
$$J(W, b; x, y) = \frac{1}{2}(h_{W,b}(x) - y)^2$$

$$x = a^{[1]}$$

$$z^{[\ell+1]} = W^{[\ell]} a^{[\ell]} + b^{[\ell]}$$

$$a^{[\ell+1]} = f(z^{[\ell+1]})$$

$$h_{W,b}(x) = a^{[n_\ell]}$$



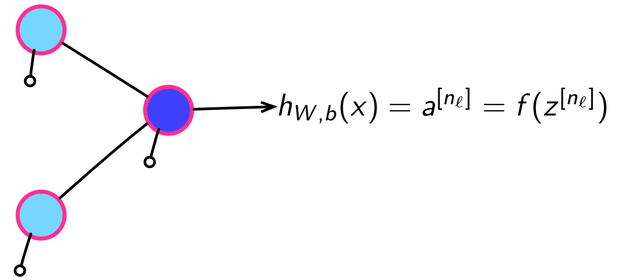
Rétro-propagation

• retour aux cours (élémentaires) de mathématiques en calculant d'abord la dérivée par rapport aux $z_i^{[\ell]}$

[Seppo Linnainmaa 1970, Yann LeCun 1985

$$
\begin{aligned}
\delta_i^{[\ell]} &= \frac{\partial}{\partial z_i^{[\ell]}} J(W, b; x, y) \\
&= \sum_{j=1}^{d_{\ell+1}} \frac{\partial}{\partial z_j^{[\ell+1]}} J(W, b; x, y) \frac{\partial z_j^{[\ell+1]}}{\partial z_i^{[\ell]}} \\
&= \sum_{j=1}^{d_{\ell+1}} \delta_j^{[\ell+1]} \frac{\partial z_j^{[\ell+1]}}{\partial a_i^{[\ell]}} \frac{\partial a_i^{[\ell]}}{\partial z_i^{[\ell]}} \\
&= \sum_{j=1}^{d_{\ell+1}} \delta_j^{[\ell+1]} w_{ji}^{[\ell]} f'(z_i^{[\ell]}) \\
&= \left(\sum_{j=1}^{d_{\ell+1}} w_{ji}^{[\ell]} \delta_j^{[\ell+1]}\right) f'(z_i^{[\ell]})
\end{aligned}
$$

couche intermédiaire



• hypothèses

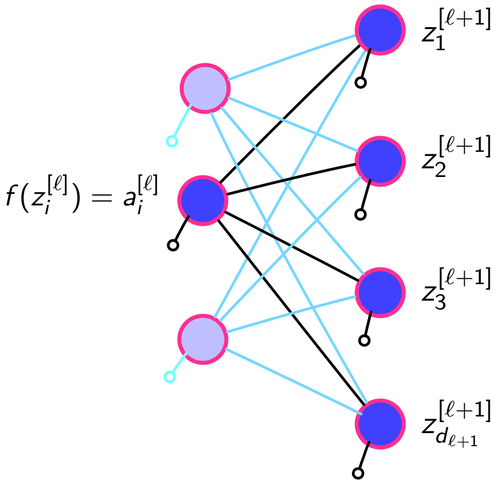
$$J(W, b; x, y) = \tfrac{1}{2}(h_{W,b}(x) - y)^2$$

$$x = a^{[1]}$$

$$z^{[\ell+1]} = W^{[\ell]} a^{[\ell]} + b^{[\ell]}$$

$$a^{[\ell+1]} = f(z^{[\ell+1]})$$

$$h_{W,b}(x) = a^{[n_\ell]}$$



Rétro-propagation

• calcul des dérivées partielles par rapport aux poids [Seppo Linnainmaa 1970, Yann LeCun 1985]

$$\delta_i^{[n_\ell]} = \frac{\partial}{\partial z_i^{[n_\ell]}} J(W, b; x, y) = (a_i^{[n_\ell]} - y)\, f'(z_i^{[n_\ell]})$$

$$\delta_i^{[\ell]} = \frac{\partial}{\partial z_i^{[\ell]}} J(W, b; x, y) = \Big(\sum_{j=1}^{d_{\ell+1}} w_{ji}^{[\ell]}\, \delta_j^{[\ell+1]}\Big)\, f'(z_i^{[\ell]})$$

$$\begin{aligned}
\frac{\partial}{\partial w_{ij}^{[\ell]}} J(W, b; x, y) &= \sum_{k=1}^{d_{\ell+1}} \delta_k^{[\ell+1]}\, \frac{\partial z_k^{[\ell+1]}}{\partial w_{ij}^{[\ell]}} \\
&= \delta_i^{[\ell+1]}\, \frac{\partial z_i^{[\ell+1]}}{\partial w_{ij}^{[\ell]}} \\
&= \delta_i^{[\ell+1]}\, \frac{\partial}{\partial w_{ij}^{[\ell]}} \sum_{k=1}^{d_\ell} w_{ik}^{[\ell]}\, a_k^{[\ell]} \\
&= \delta_i^{[\ell+1]}\, a_j^{[\ell]}
\end{aligned}$$

• hypothèses

$$J(W, b; x, y) = \tfrac{1}{2}(h_{W,b}(x) - y)^2$$

$$x = a^{[1]}$$

$$z^{[\ell+1]} = W^{[\ell]} a^{[\ell]} + b^{[\ell]}$$

$$a^{[\ell+1]} = f(z^{[\ell+1]})$$

$$h_{W,b}(x) = a^{[n_\ell]}$$

$$\frac{\partial}{\partial w_{ij}^{[\ell]}} J(W, b; x, y) = a_j^{[\ell]}\, \delta_i^{[\ell+1]}$$

$$\frac{\partial}{\partial b_i^{[\ell]}} J(W, b; x, y) = \delta_i^{[\ell+1]}$$

Rétro-propagation

- calcul des dérivées partielles par rapport aux poids

1) on calcule les $a_i^{[\ell]}$ par une **passe en avant**

2) pour la couche de sortie (ou les sorties si plusieurs), on calcule

$$\delta_i^{[n_\ell]} = (a_i^{[n_\ell]} - y)\, f'(z_i^{[n_\ell]})$$

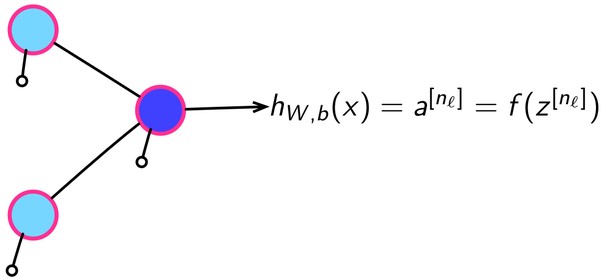
3) pour chaque neurone des couches intermédiaires, on calcule

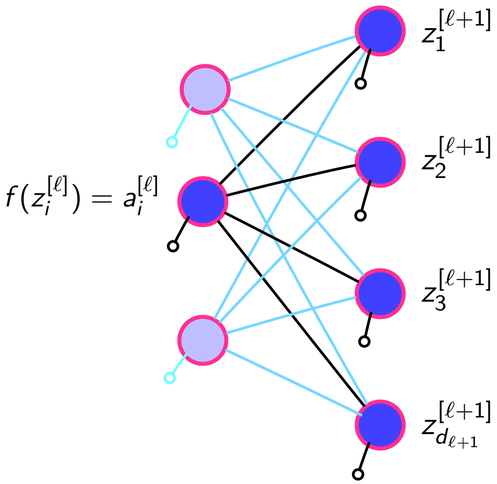
$$\delta_i^{[\ell]} = (\sum_{j=1}^{d_{\ell+1}} w_{ji}^{[\ell]}\, \delta_j^{[\ell+1]})\, f'(z_i^{[\ell]})$$

4) les dérivées sont maintenant établies

$$\frac{\partial}{\partial w_{ij}^{[\ell]}} J(W, b; x, y) = a_j^{[\ell]}\, \delta_i^{[\ell+1]}$$

$$\frac{\partial}{\partial b_i^{[\ell]}} J(W, b; x, y) = \delta_i^{[\ell+1]}$$





Rétro-propagation

- calcul des dérivées partielles par rapport aux poids

1) on calcule les $a_i^{[\ell]}$ et les $z_i^{[\ell]}$ par une passe en avant

2) pour la sortie (ou les sorties si plusieurs), on calcule

$$\delta^{[n_\ell]} = (a^{[n_\ell]} - y) \bullet f'(z^{[n_\ell]})$$

- $\bullet$ multiplication point par point

3) pour chaque neurone des couches intermédiaires, on calcule

$$\delta^{[\ell]} = ((W^{[\ell]})^T \ \delta^{[\ell+1]}) \bullet f'(z^{[\ell]})$$

4) les dérivées sont maintenant établies

$$\nabla_{W^{[\ell]}} J(W, b; x, y) = \delta^{[\ell+1]} (a^{[\ell]})^T$$

$$\nabla_{b^{[\ell]}} J(W, b; x, y) = \delta^{[\ell+1]}$$

- ∇ opérateur de dérivation par rapport à tous les éléments de la matrice

apprentissage profond

Apprentissage profond - calcul des poids

- descente du gradient (par lots) avec rétro-propagation pour calculer les poids

1) initialisation $dW^{[\ell]} = 0$ et $db^{[\ell]} = 0$ pour tout ℓ

2) pour toutes les données (x, y)

   (i) par rétropropagation, calculer $\nabla_{W^{[\ell]}} J(W, b; x, y)$ et $\nabla_{b^{[\ell]}} J(W, b; x, y)$

   (ii) additionner les dérivées pour chaque donnée

$$dW^{[\ell]} = dW^{[\ell]} + \nabla_{W^{[\ell]}} J(W, b; x, y)$$
$$db^{[\ell]} = db^{[\ell]} + \nabla_{b^{[\ell]}} J(W, b; x, y)$$

3) modifier tous les poids

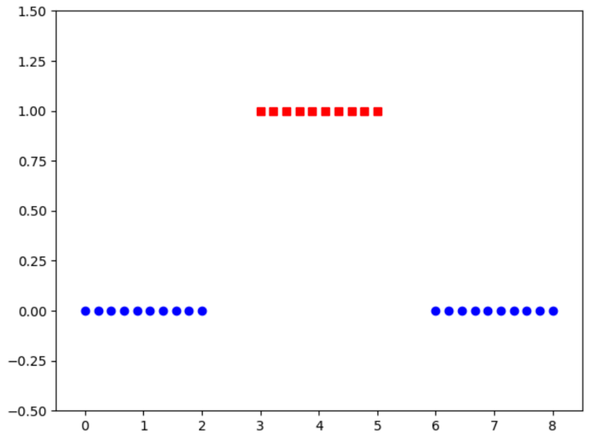
$$W^{[\ell]} = W^{[\ell]} - \alpha \left[\left(\tfrac{1}{n} \, dW^{[\ell]} \right) + \lambda \, W^{[\ell]} \right]$$
$$b^{[\ell]} = b^{[\ell]} - \alpha \left[\tfrac{1}{n} \, db^{[\ell]} \right]$$

4) et on recommence la descente jusqu'à convergence

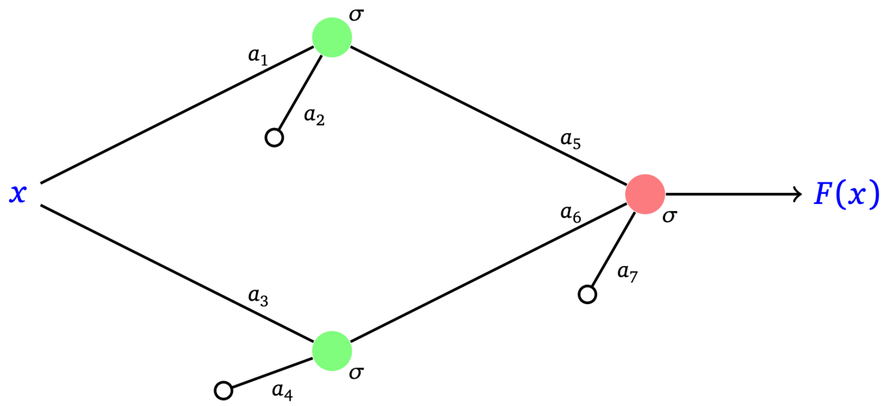
5) PROBLÈMES: initialisation des W et vitesse de cet algorithme

Calcul des poids

Exercice Ecrire les fonctions Python pour le calcul des poids avec les jeux de données et le modèle suivant



données



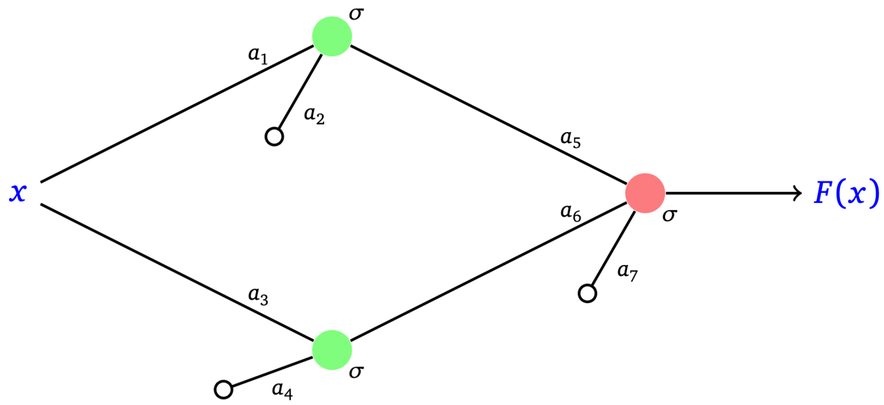
modèle

- on veut trouver un modèle (*neural net*) qui approxime les données d'entrainement
- ce qui permet de prevoir des résultats en dehors des données d'entrainement (*predictions*)
- dans notre exercice, on suppose un réseau à 3 couches (entrée, intermédiaire avec 2 neurones, sortie) et une fonction sigmoïde d'activation
- il reste à calculer les paramètres (poids)

Apprentissage profond

• jeu de données et coefficients initiaux

```
bleus = [(i/10, 0.2) for i in range (20)] + \
        [(6 + i/10, 0.2) for i in range (20)]
rouges = [(3 + i/10, 1) for i in range (20)]

dataset = bleus + rouges

# --- paramètres (poids) du réseau ---
a1, a2, a3, a4 = (0.0, 1.0, 0.0, -1.0)
a5, a6, a7 = (1.0, 1.0, -1.0)
```



• fonction d'activation (et sa dérivée)

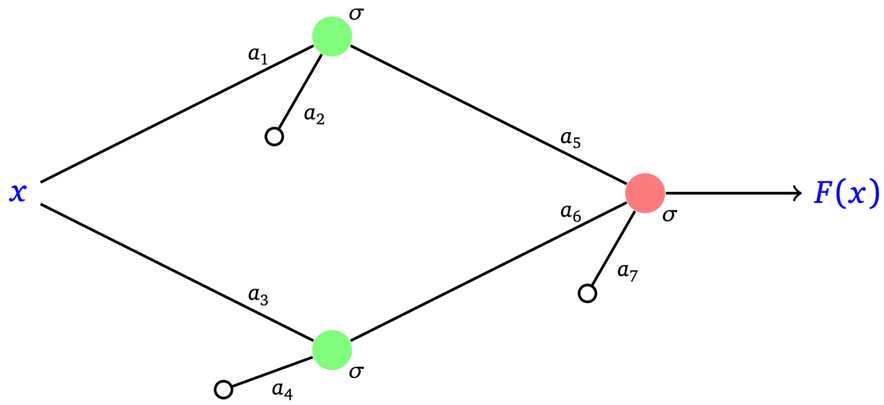
```
def sigma (x) :
    return 1 / (1 + np.exp(-x))

def dsigma (x) :
    return (sigma (x)) * (1 - sigma(x))
```

• on va calculer les poids par une descente de gradient sur les 7 coefficients

Apprentissage profond

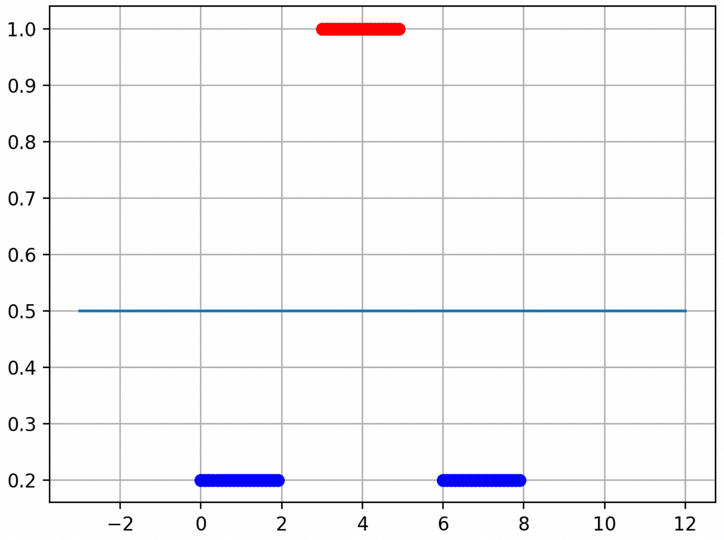
- fonction calculée

```
def F1(x):
    global a1, a2, a3, a4, a5, a6, a7
    return sigma (a1 * x + a2)

def F2(x):
    global a1, a2, a3, a4, a5, a6, a7
    return sigma (a3 * x + a4)

def F(x):
    global a1, a2, a3, a4, a5, a6, a7
    return sigma (a5 * F1(x) + a6 * F2(x) + a7)
```



- affichage des données et de la fonction calculée

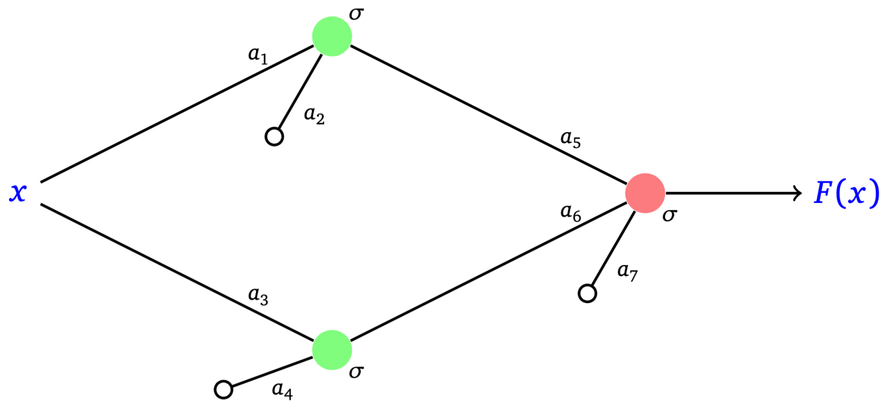
```
xlim = (-3, 12)
def show (dataset) :
    global xlim
    for (x, y) in dataset :
        plt.plot (x, y, marker='o', color='r' if y == 1 else 'b')
    X = np.linspace (*xlim, 100)
    Y = F(X)
    plt.plot (X, Y)
    plt.grid()
    plt.show()
```



Apprentissage profond

- dérivées de la fonction calculée

```
def dF1(x):
    global a1, a2, a3, a4, a5, a6, a7
    return np.array ([dsigma (a1 * x + a2) * x, \
            dsigma (a1 * x + a2), \
            0, 0, 0, 0, 0])

def dF2(x):
    global a1, a2, a3, a4, a5, a6, a7
    return np.array ([0, 0, dsigma (a3 * x + a4) * x,
            dsigma (a3 * x + a4), 0, 0, 0])

def dF(x):
    global a1, a2, a3, a4, a5, a6, a7
    dA3 = dsigma(a5 * F1(x) + a6 * F2(x) + a7)
    return np.array ([ \
            dA3 * a5 * dF1(x)[0],
            dA3 * a5 * dF1(x)[1],
            dA3 * a6 * dF2(x)[2],
            dA3 * a6 * dF2(x)[3],
            dA3 * F1(x),
            dA3 * F2(x),
            dA3 ])
```



Apprentissage profond

• la fonction J de coût

```
def Jxy(x, y) :
    return 0.5 * (F (x) - y)**2

def J(dataset) :
    s = 0; n = len(dataset)
    for (x, y) in dataset :
        s += Jxy (x, y)
    return s
```

• dérivées du coût

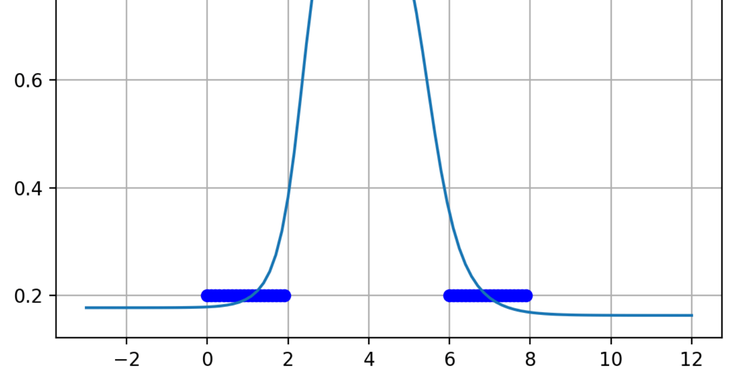
```
def derJxy (x, y) :
    return (F(x) - y) * dF (x)

def derJ (dataset) :
    s = 0
    for (x, y) in dataset :
        s += derJxy (x, y)
   return s
```

• descemte du gradient

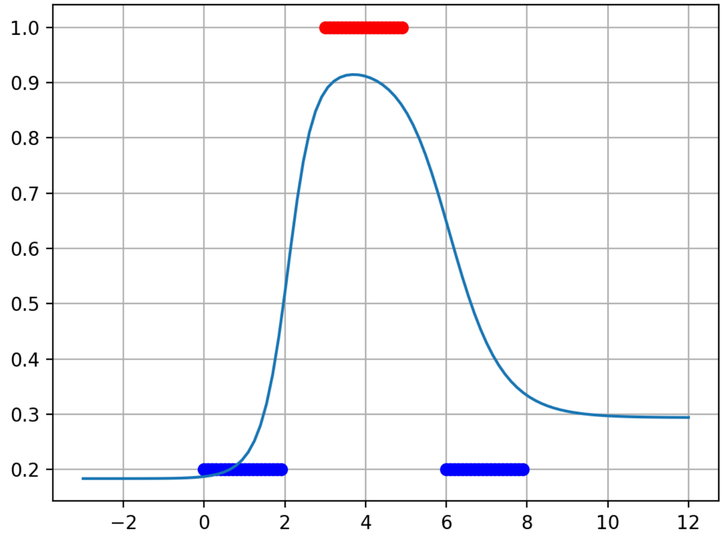
```
def descent (dataset) :
    global a1, a2, a3, a4, a5, a6, a7
    global alpha
    for epoch in range (800) :
        dd = derJ (dataset)
        a1 -= alpha * dd [0]
        a2 -= alpha * dd [1]
        a3 -= alpha * dd [2]
        a4 -= alpha * dd [3]
        a5 -= alpha * dd [4]
        a6 -= alpha * dd [5]
        a7 -= alpha * dd [6]
```

• programme principal

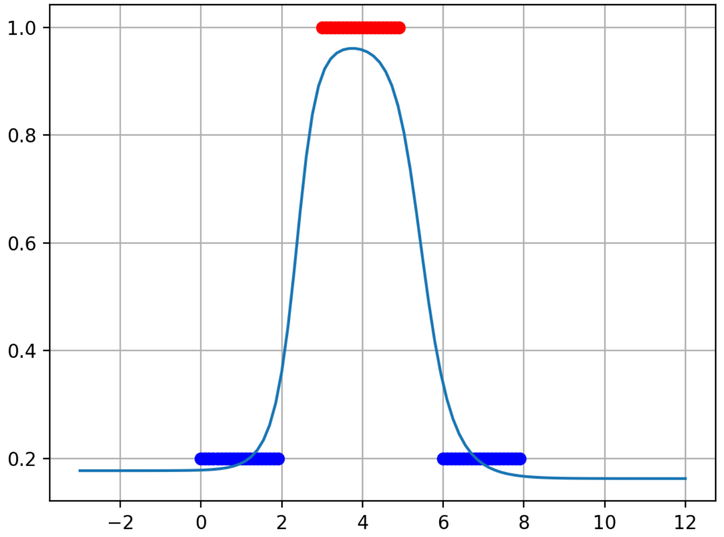
```
alpha = 0.1
print ('J = ', J (dataset))
print ('a1a2a3a4a5a6a7 --> ', a1, a2, a3, a4, a5, a6, a7)
show (dataset)

descent (dataset)

print ('J = ', J (dataset))
print ('a1a2a3a4a5a6a7 --> ', a1, a2, a3, a4, a5, a6, a7)
show (dataset)
```



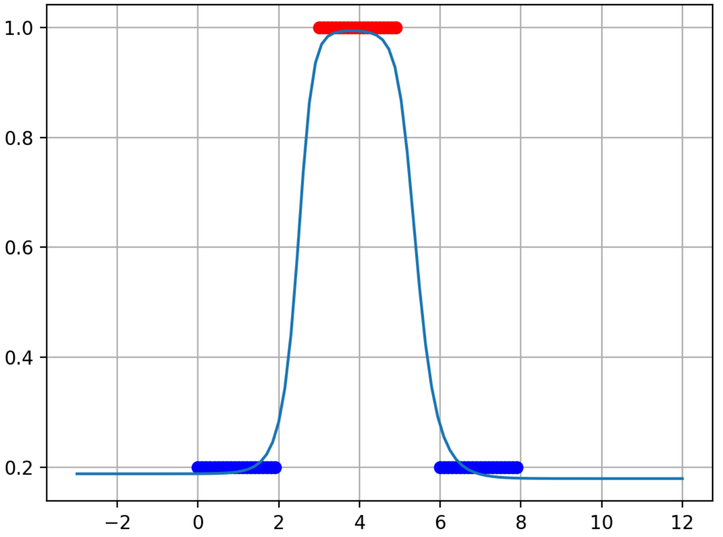
Apprentissage profond



• 500 itérations



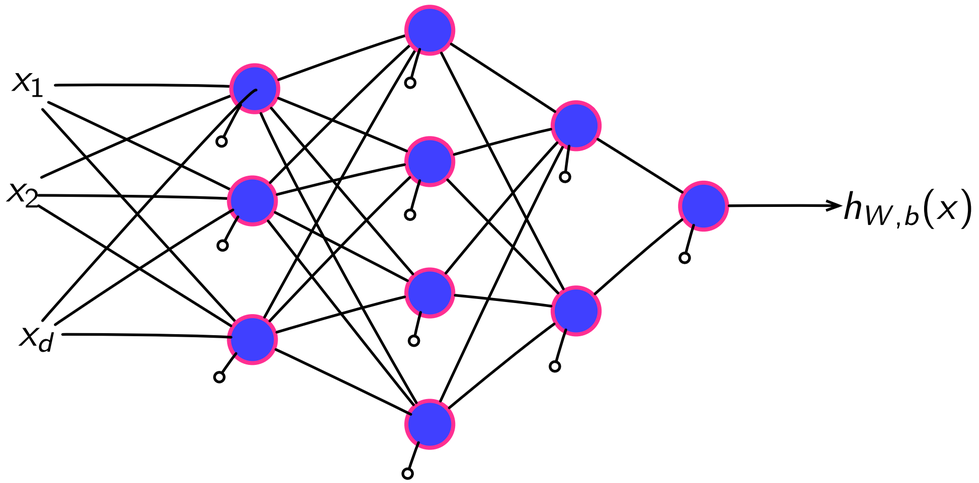
• 1000 itérations



• 5000 itérations

Réseau de neurones (rappel)

• exemple de réseau avec d entrées, 1 sortie, 5 couches $(n_\ell = 5)$ et une fonction d'activation uniforme *f*



$$x = a^{[1]}$$

$$z^{[\ell+1]} = W^{[\ell]} a^{[\ell]} + b^{[\ell]}$$

$$a^{[\ell+1]} = f(z^{[\ell+1]})$$

$$h_{W,b}(x) = a^{[n_\ell]}$$

$d = d_1$ $\quad n_\ell = 5$ $\quad W^{[1]} : 3 \times d$ $\quad W^{[2]} : 4 \times 3$ $\quad W^{[3]} : 2 \times 4$ $\quad W^{[4]} : 1 \times 2$

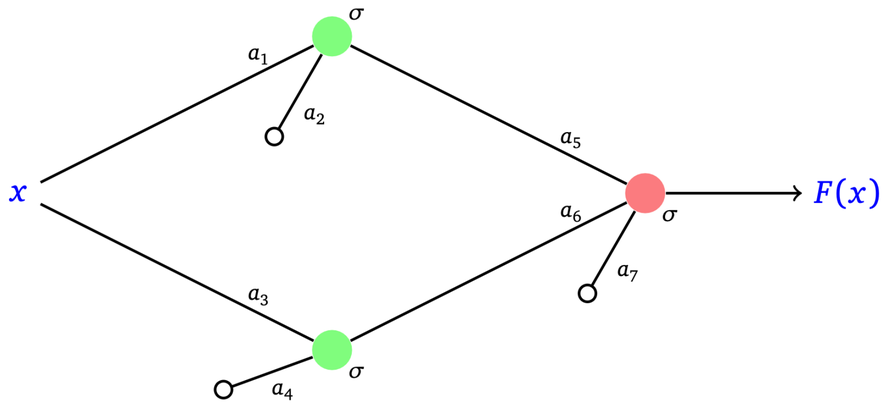
• 31 paramètres quand *d* = 3 !!

Apprentissage profond

[version algébrique]

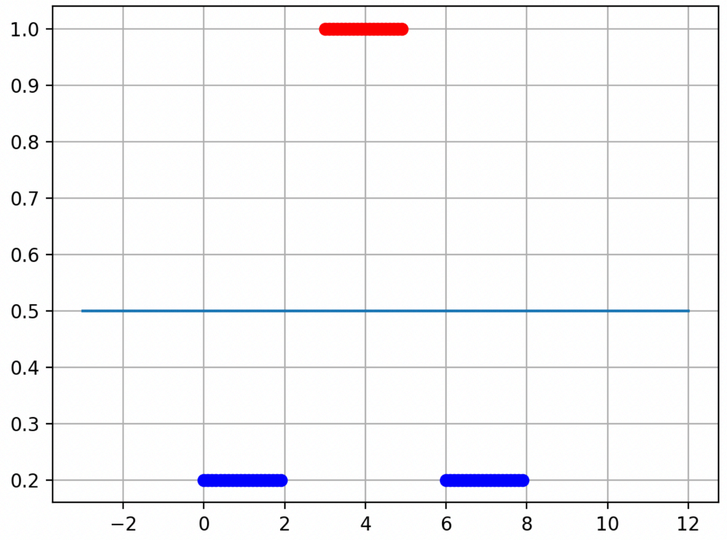
- fonction calculée

```
W1 = np.array ([[a1], [a3]]) # shape 2x1
B1 = np.array ([[a2], [a4]]) # shape 2x1
W2 = np.array ([[a5, a6]]) # shape 1x2
B2 = np.array ([[a7]])     # shape 1x1

def F(x):
    global W1, W2, B1, B2
    X = np.array([[x]])
    Z2 = np.dot (W1, X)+ B1; A2 = sigma (Z2)
    Z3 = np.dot (W2, A2) + B2; A3 = sigma (Z3)
    return A3[0,0]
```



- affichage des données et de la fonction calculée

```
xlim = (-3, 12)
def show (dataset) :
    global xlim
    for (x, y) in dataset :
        plt.plot (x, y, marker='o', color='r' if y == 1 else 'b')
    X = np.linspace (*xlim, 100)
    Y = np.array ([F(x) for x in X])
    plt.plot (X, Y)
    plt.grid()
    plt.show()
```



Rétro-propagation (rappel)

- calcul des dérivées partielles par rapport aux poids

1) on calcule les $a_i^{[\ell]}$ et les $z_i^{[\ell]}$ par une passe en avant

2) pour la sortie (ou les sorties si plusieurs), on calcule

$$\delta^{[n_\ell]} = (a^{[n_\ell]} - y) \bullet f'(z^{[n_\ell]})$$

- $\bullet$ multiplication point par point

3) pour chaque neurone des couches intermédiaires, on calcule

$$\delta^{[\ell]} = ((W^{[\ell]})^T \ \delta^{[\ell+1]}) \bullet f'(z^{[\ell]})$$

4) les dérivées sont maintenant établies

$$\nabla_{W^{[\ell]}} J(W, b; x, y) = \delta^{[\ell+1]} (a^{[\ell]})^T$$

$$\nabla_{b^{[\ell]}} J(W, b; x, y) = \delta^{[\ell+1]}$$

- ∇ opérateur de dérivation par rapport à tous les éléments de la matrice

Apprentissage profond

• fonction de coût

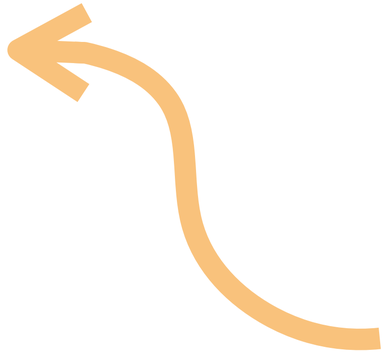
```
def Jxy(x, y) :
    return 0.5 * (F (x) - y)**2

def J(dataset) :
    s = 0
    for (x, y) in dataset :
        s += Jxy (x, y)
    return s
```

• dérivée du coût

```
def derJxy (x, y) :
    # forward pass
    global W1, W2, B1, B2
    X = np.array([[x]])
    Z2 = np.dot (W1, X) + B1; A2 = sigma (Z2)
    Z3 = np.dot (W2, A2) + B2; A3 = sigma (Z3)

    # back propagate
    Y = np.array([[y]])
    D3 =  (A3 - Y) * dsigma(Z3)
    D2 = np.dot (W2.T, D3) * dsigma (Z2)
    dJW1 = np.dot (D2, X.T); dJB1 = D2
    dJW2 = np.dot (D3, A2.T); dJB2 = D3
    return dJW1, dJW2, dJB1, dJB2

def derJ (dataset) :
    sW1, sW2, sb1, sb2 = 0, 0, 0, 0
    for (x, y) in dataset :
        (dW1, dW2, dB1, dB2) = derJxy (x, y)
        sW1 += dW1; sW2 += dW2
        sb1 += dB1; sb2 += dB2
    return (sW1, sW2, sb1, sb2)
```

• descente du gradient

```
def descent (dataset) :
    global W1, W2, B1, B2
    global alpha
    for epoch in range (1000) :
        (dW1, dW2, dB1, dB2) = derJ (dataset)
        W1 -= alpha * dW1; B1 -= alpha * dB1
        W2 -= alpha * dW2; B2 -= alpha * dB2
```

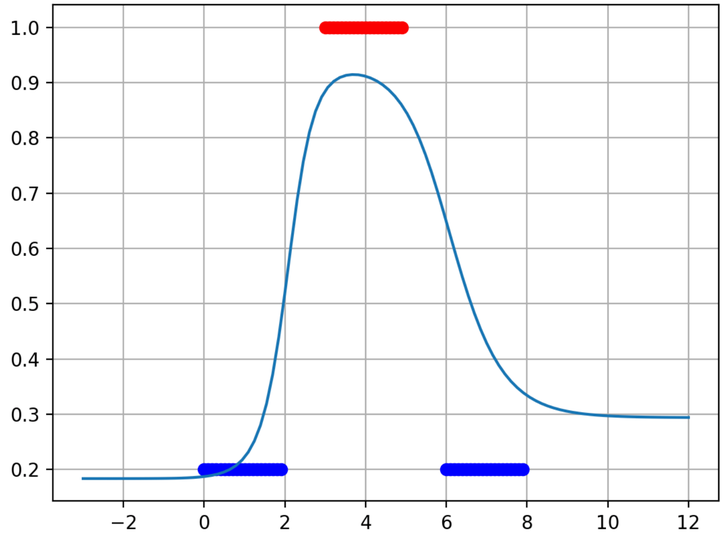


$$\delta^{[3]} = (a^{[3]} - y) \bullet f'(z^{[3]})$$
$$\delta^{[2]} = ((W^{[2]})^T \delta^{[3]}) \bullet f'(z^{[2]})$$
$$\nabla_{W^{[1]}} J(W, b; x, y) = \delta^{[2]} (a^{[1]})^T$$
$$\nabla_{b^{[1]}} J(W, b; x, y) = \delta^{[2]}$$
$$\nabla_{W^{[2]}} J(W, b; x, y) = \delta^{[3]} (a^{[2]})^T$$
$$\nabla_{b^{[2]}} J(W, b; x, y) = \delta^{[3]}$$

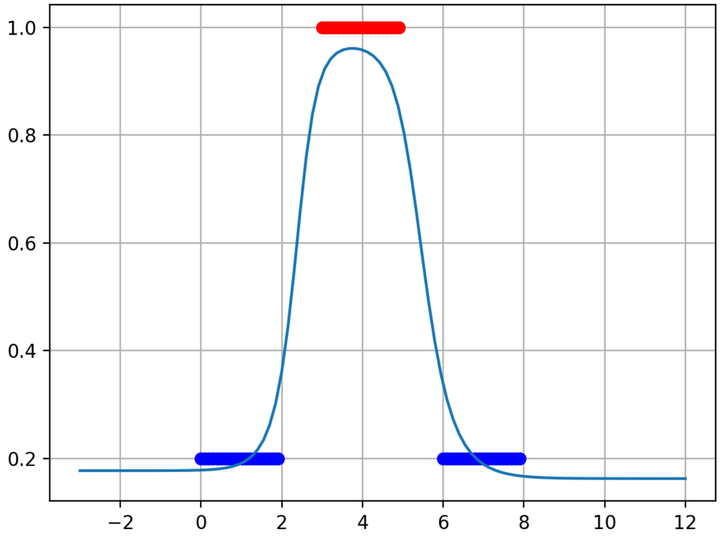
• programme principal

```
alpha = 0.1
print ('J = ', J (dataset))
print ('w1b2w2b2 --> ', W1, B1, W2, B2)
show(dataset)
descent (dataset)
print ('J = ', J (dataset))
print ('w1b2w2b2 --> ', W1, B1, W2, B2)
show(dataset)
```

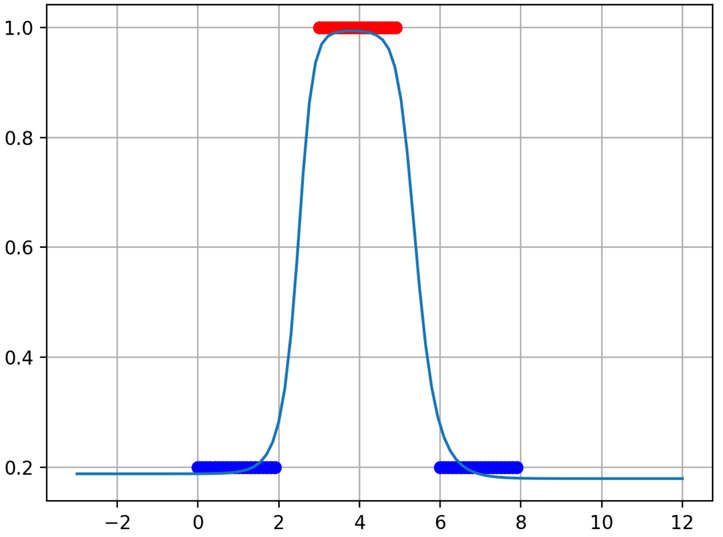
Apprentissage profond



• 500 itérations



• 1000 itérations



• 5000 itérations

Apprentissage profond

- l'**initialisation** des poids est importante pour diminuer l'erreur

- beaucoup d'opérations peuvent être **vectorisées** avec un GPU (processeur graphique)

  - la multiplication de matrices exécutée en parallèle [https://youtu.be/cGEIEnekmRM]
  - l'addition de matrices exécutée en parallèle
  - chaque étape de la rétro-propagation en parallèle
  - tout calcul sans variables modifiables (ce n'est pas le cas de la descente de gradient)

- NVDIA est actuellement le plus important fabricant de GPU, aussi rebaptisé *Neural Engine* chez Apple

- Tensorflow, Pytorch marchent avec GPU sur PC avec chip AMD, macbook-pro, et iMac avec les chips M1, .. , M4 (chip Apple/ARM). Google a développé les TPU (*Tensor Processor Unit*)

Reconnaissance de texte

[exemple pris de Deepmath Exo7 (cité dans la 2ème diapositive), page 192]

Le but de cet exemple est de décider si une critique de film est positive ou négative. Voici un exemple de critique (la critique numéro 123) de la base que nous utiliserons :

beautiful and touching movie rich colors great settings good acting and one of the most charming movies i have seen in a while i never saw such an interesting setting when i was in china my wife liked it so much she asked me to log on and rate it so other would enjoy too

C'est un critique positive !

La base IMDB fournit 25000 critiques d'apprentissage, chacune étant déjà catégorisée positive (valeur 1) ou négative (valeur 0)

Voici trois (fausses) critiques :
« bon film à voir absolument»
« début moyen mais après c'est très mauvais»
« film drôle et émouvant on passe un bon moment»

• Tout d'abord, parmi toutes les critiques, on ne retient que les 10 mots les plus fréquents. Imaginons que ce sont les mots : **film, bon, mauvais, voir, éviter, très, bien, mal, moyen, super**

En réalité on retiendra les 1000 (voire 10 000) mots les plus fréquents.

Reconnaissance de texte

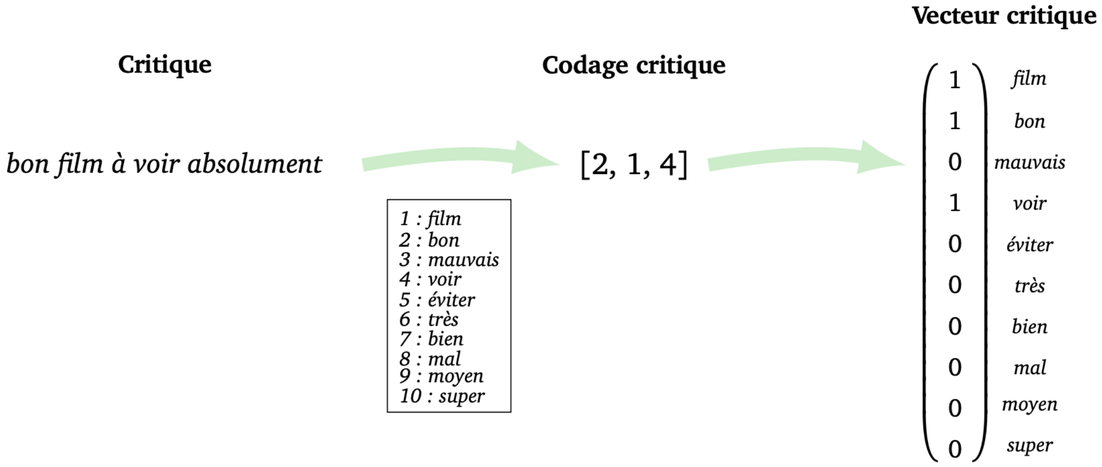
[exemple pris de Deepmath (cité dans la 2ème diapositive), page 192]

• Tout d'abord, parmi toutes les critiques, on ne retient que les 10 mots les plus fréquents. Imaginons que ce sont les mots : **film, bon, mauvais, voir, éviter, très, bien, mal, moyen, super**

En réalité on retiendra les 1000 (voire 10 000) mots les plus fréquents.

• On code une phrase comme une liste d'indices de mots. Le mot « film » est remplacé par 1, le mot « bon » par 2,. . . jusqu'à « super ». Les autres mots ne sont pas pris en compte. Par exemple la critique « bon film à voir absolument » devient la liste d'indices [2, 1, 4]. Les deux autres critiques deviennent [9, 3] et [1, 2].

• On transforme ensuite chaque liste en un vecteur de taille fixe n = 10 de coordonnées 0 ou 1. On place un 1 en position i si le mot numéro i apparaît dans la critique. **Ainsi la phrase « bon film à voir absolument », codée en [2, 1, 4] devient le vecteur X = (1, 1, 0, 1, 0, 0, 0, 0, 0, 0)**. La phrase codée [9, 3] donne le vecteur X′= (0, 0, 1, 0, 0, 0, 0, 0, 1, 0).



Apprentissage supervisé, réinforcé ou non

• apprentissage **supervisé** avec un jeu de données d'entrainement où la sortie est étiquetée

- ce qu'on a vu jusqu'à présent

• apprentissage **non supervisé** avec un jeu de données non étiquetées

- on essaie de les classer en groupes à partir de données prises au hasard initialement

• apprentissage **renforcé** est un mélange des deux techniques

- on modifie l'apprentissage automatique avec quelques interventions humaines

• l'apprentissage génératif désigne la prédiction à partir d'une des méthodes précédentes

- par exemple, GPT (*generative pre-training transformer*) utilisé dans les LLM ou Chat-GPT

Modèles de réseaux

- `nn`: modèles séquentiels
  - suite de couches (plus ou moins denses, couches de convolution possibles)

- `rnn`: modèles récurrents
  - boucles possibles dans l'enchaînement des couches

- modèles fonctionnels (en Keras)
  - topologie à la carte

Conclusion

- l'apprentissage est **bien loin** des algorithmes
- la contribution informatique est : le stockage possible de **grand nombre de données**, les **processeurs parallèles**
- l'apprentissage est une discipline plutôt **expérimentale** (statistiques et probabilités)
- l'apprentissage a été souvent le domaine de la **vision par ordinateur** (*computer vision*)
- de manière surprenante, l'apprentissage marche pour l'**analyse de la langue naturelle** (*NLP*) avec les LLM et GPT

- il reste à l'intégrer à la **démonstration automatique** de théorèmes, à l'aide à la programmation (*copilot*) ou à la vérification du logiciel
- etc..