Algorithmes, Programmation, IA

Cours 9

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/algo-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

- 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]

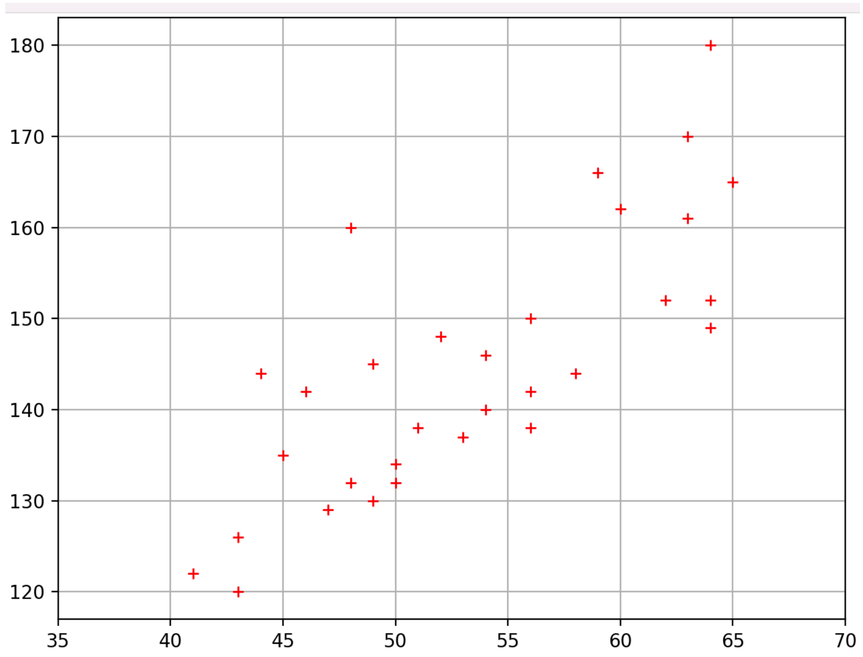
- dérivée d'une fonction de plusieurs arguments

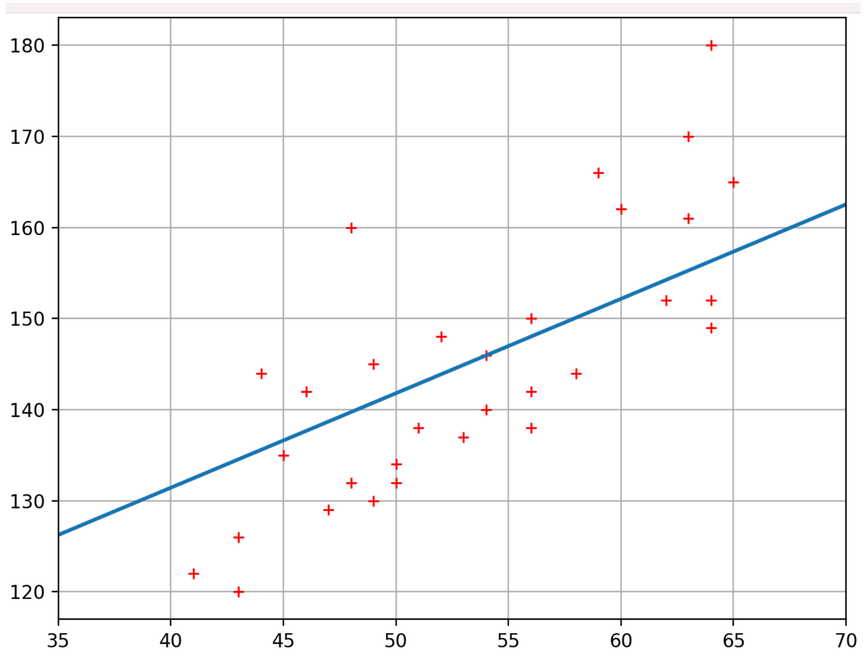
$$\frac{\partial}{\partial t} f(x,y) = \frac{\partial}{\partial x} f(x,y)\,\frac{\partial x}{\partial t} + \frac{\partial}{\partial y} f(x,y)\,\frac{\partial y}{\partial t}$$

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

Régression linéaire

- on cherche une approximation linéaire du jeu de données





$$y = \theta_0 + \theta_1 x_1 \cdots \theta_d x_d = \theta^T x$$

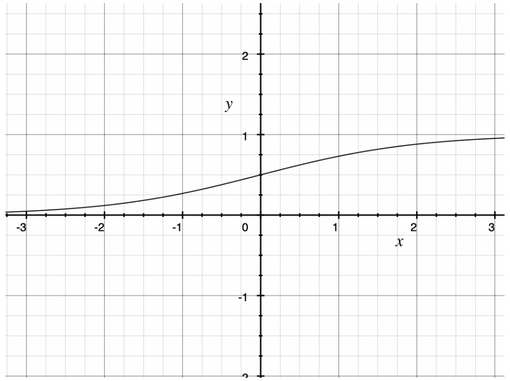
Classification

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

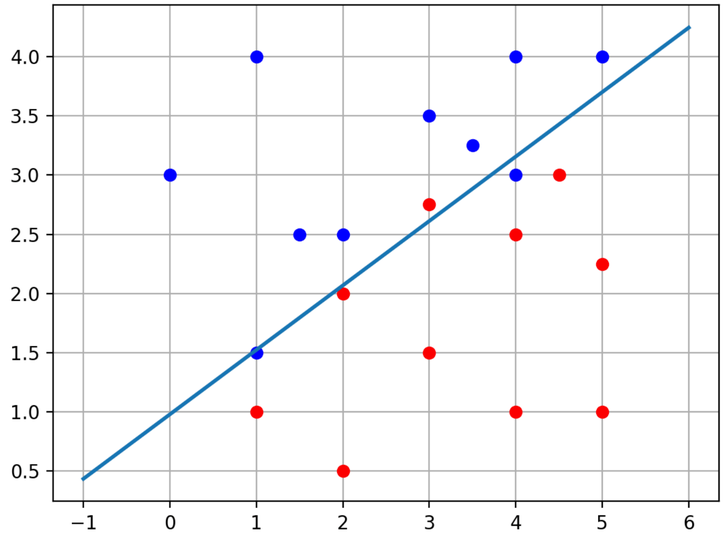
$$\sigma(\theta^T x) = \sigma(\theta_0 + \theta_1 x_1 + \cdots \theta_d x_d) > \frac{1}{2} \quad \Longleftrightarrow \quad \theta^T x = \theta_0 + \theta_1 x_1 + \cdots \theta_d x_d > 0$$

avec la fonction sigmoïde définie par:

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



σ

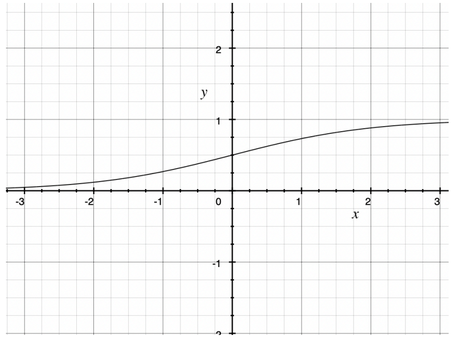


Fonctions d'activation

- fonctions d'activation possibles

la fonction sigmoïde

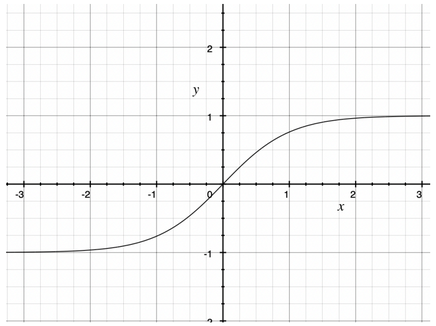
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



σ

la tangente hyperbolique

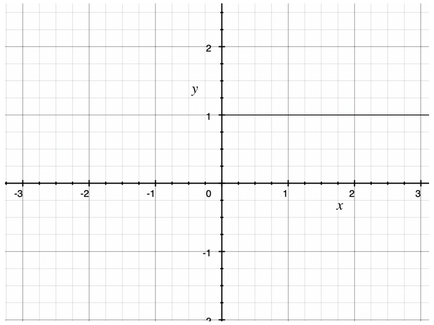
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



tanh

la fonction marche de Heaviside

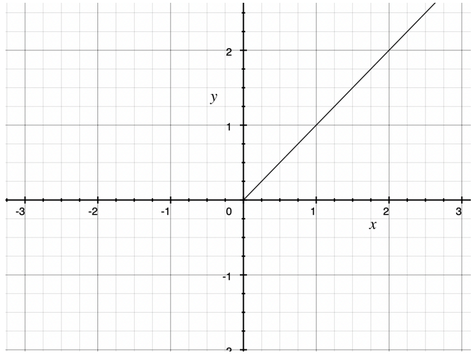
$$\mathrm{H}(x) = \begin{cases} 1 \text{ si } x \geq 0 \\ 0 \text{ sinon} \end{cases}$$



H

la fonction linéaire rectifiée

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

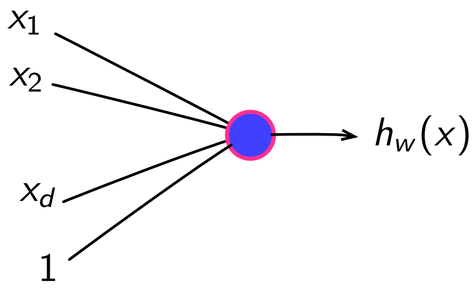


ReLU

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

Neurone

- neurone à *d* entrées et 1 sortie

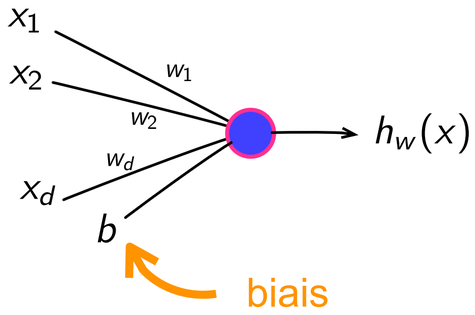


$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \\ 1 \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix}$$

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

fonction d'activation

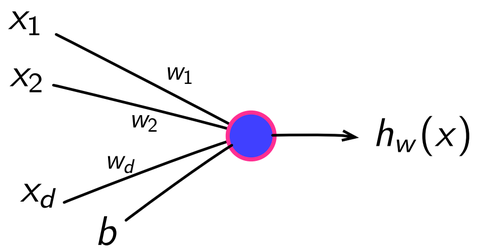
- autre présentation graphique



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

Neurone - perceptron

- *d* entrées et 1 sortie

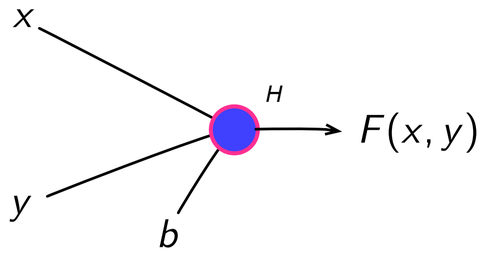


$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \\ 1 \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix}$$

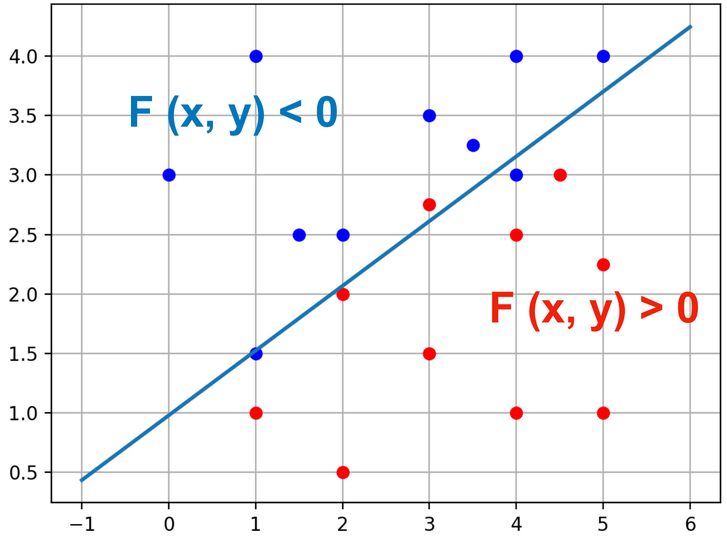
$$h_w(x) = f(w^T x) = f\left(\sum_{i=1}^{d} w_i x_i + b\right)$$

fonction d'activation

- classification linéaire avec *F* = *H* (Heaviside)

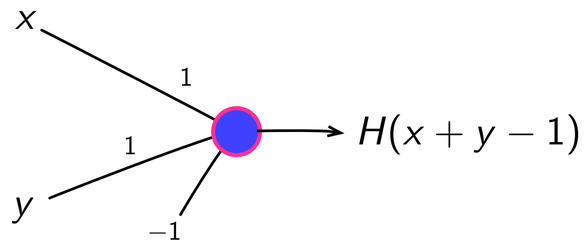


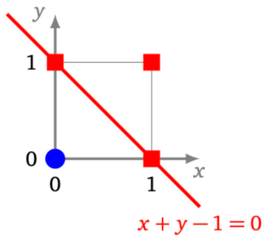
$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad w = \begin{bmatrix} 1 \\ -2 \\ 2 \end{bmatrix}$$



Neurone - perceptron

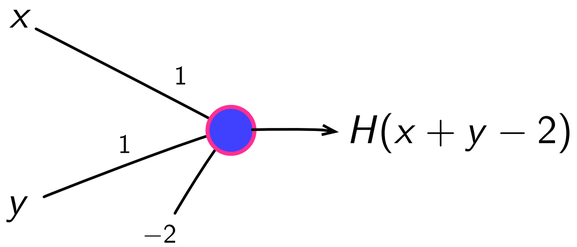
- avec le perceptron, on peut faire les fonctions OU, le ET et le NOT avec Heaviside

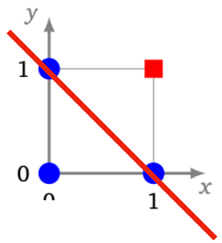




OU

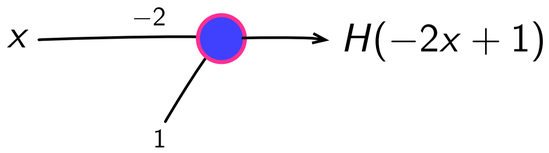
x\	0	1
0	0	1
1	1	1





ET

x\	0	1
0	0	0
1	0	1



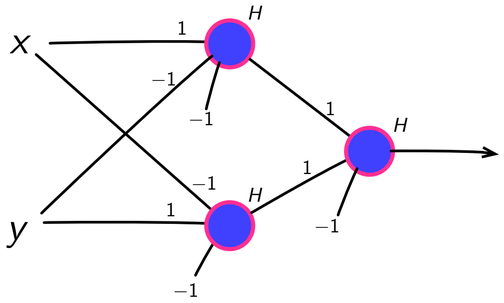
NOT

x	
0	1
1	0

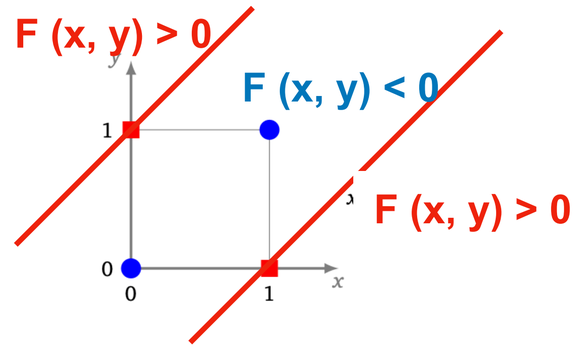
Neurone - perceptron

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

- il faut plus de neurones



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

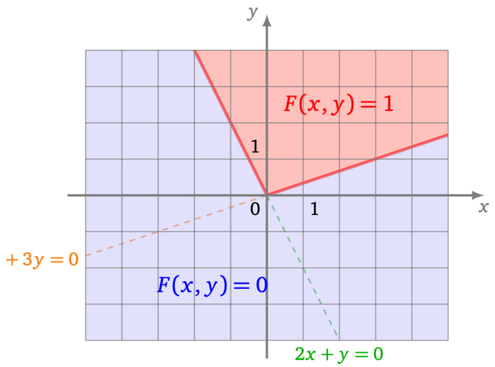


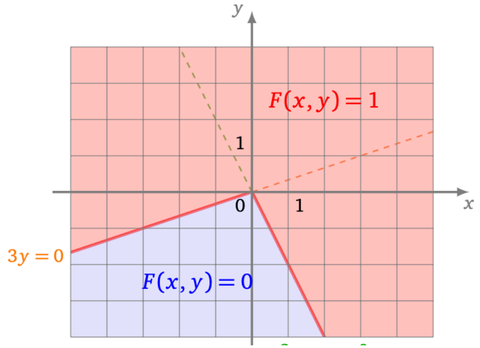
XOR

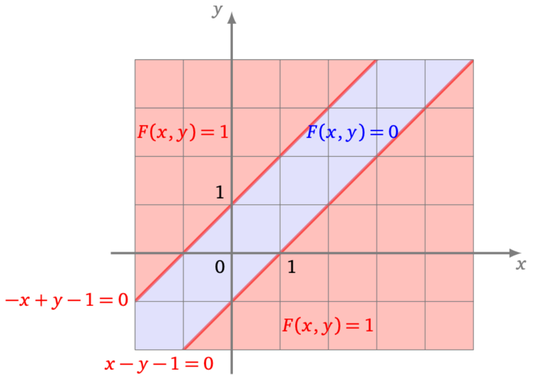
x\	0	1
0	0	1
1	1	0

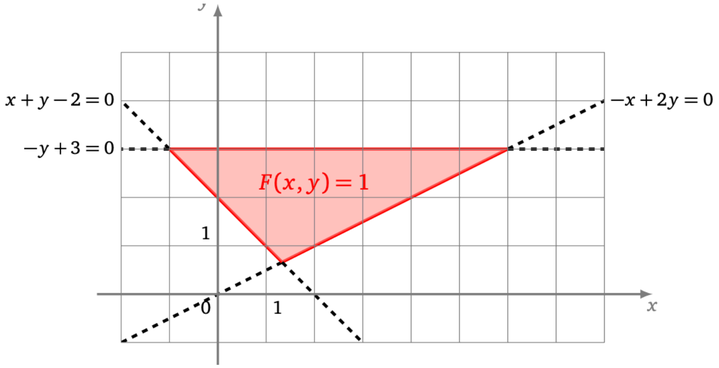
Neurones

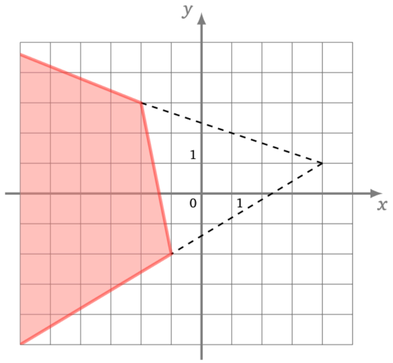
Exercice Trouver le nombre de neurones pour définir les zones rouges suivantes :







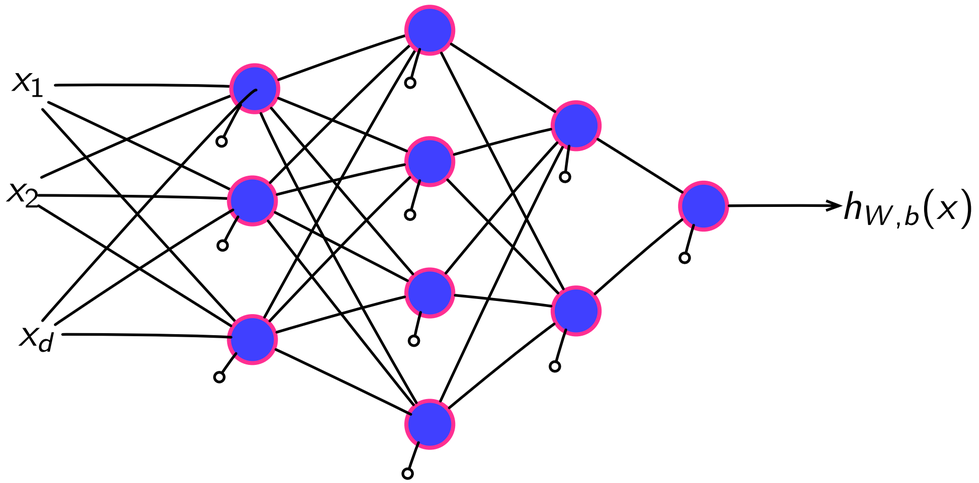




- On peut donc approximer l'intérieur d'une courbe fermée en l'approximant par suffisamment de triangles

Réseau de neurones

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



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

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

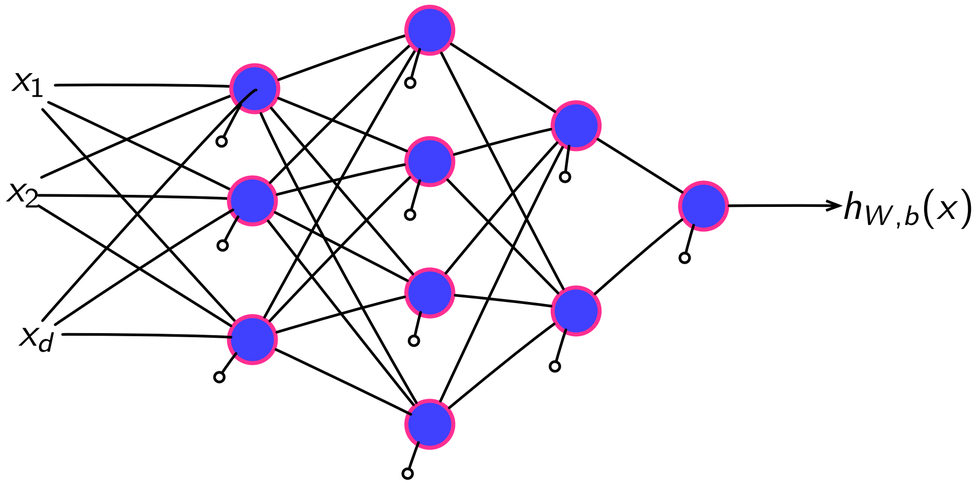
$$a^{[\ell+1]} = f(z^{[\ell+1]})$$

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

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

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 !!

Apprentissage profond - calcul de l'erreur

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

- 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 ?

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

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

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

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

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

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

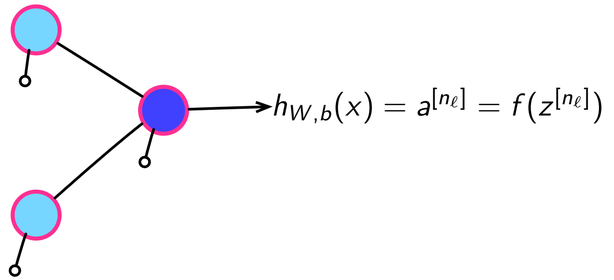
$$\frac{\partial}{\partial b_i^{[\ell]}} J(W, b;\ x, y)$$

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

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



• hypothèses

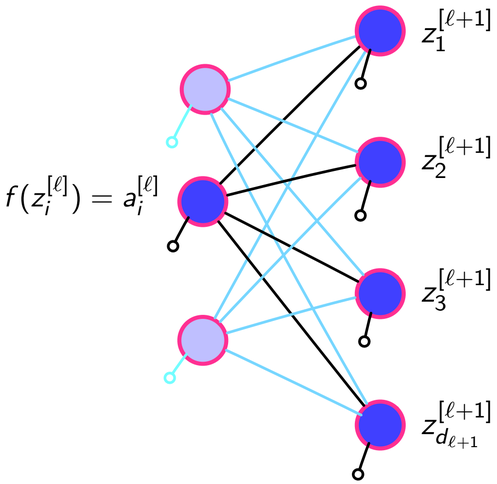
$$J(W, b; x, y) = \frac{1}{2}(h_{W,b}(x) - y)^2$$

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

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

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

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

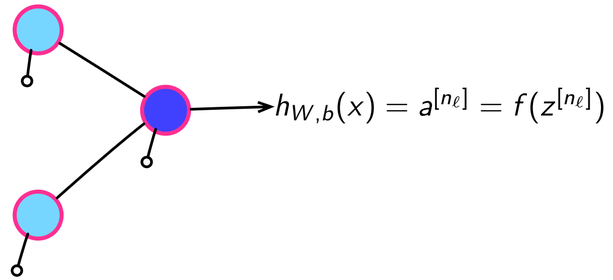


Rétro-propagation

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

[Seppo Linnainmaa 1970, Yann LeCun 1985

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



• hypothèses

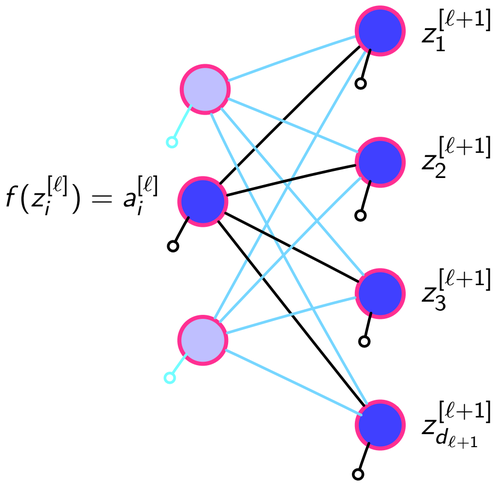
$$J(W, b; x, y) = \tfrac{1}{2}(h_{W,b}(x) - y)^2$$

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

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

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

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



Rétro-propagation

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

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

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

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

• 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

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

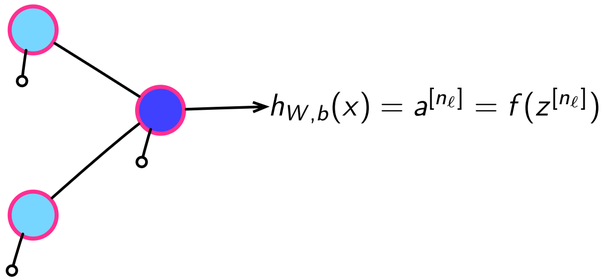
3) pour chaque neurone des couches intermédiaires, on calcule

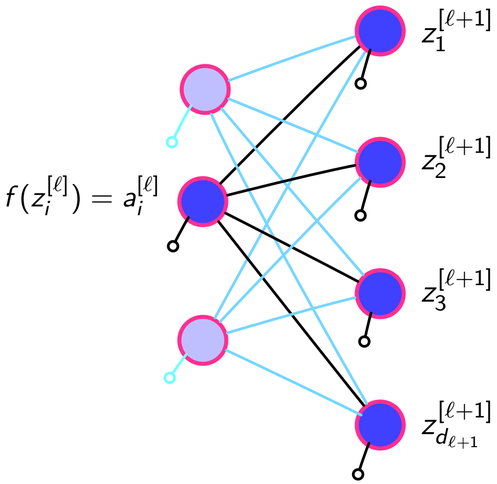
$$\delta_i^{[\ell]} = (\sum_{j=1}^{d_{\ell+1}} w_{ji}^{[\ell]}\, \delta_j^{[\ell+1]})\, f'(z_i^{[\ell]})$$

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 - 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

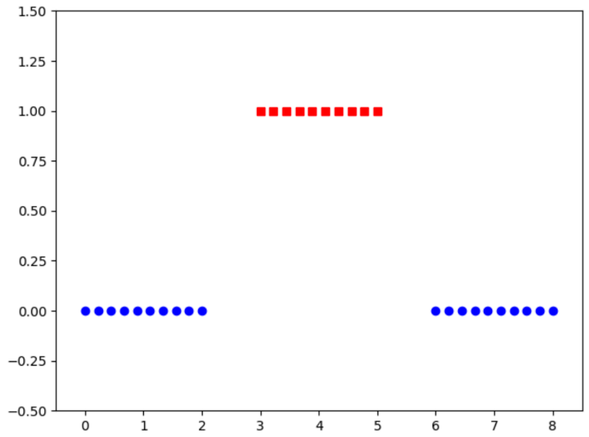
$$W^{[\ell]} = W^{[\ell]} - \alpha \left[\left(\tfrac{1}{n} \, dW^{[\ell]} \right) + \lambda \, W^{[\ell]} \right]$$
$$b^{[\ell]} = b^{[\ell]} - \alpha \left[\tfrac{1}{n} \, db^{[\ell]} \right]$$

4) et on recommence la descente jusqu'à convergence

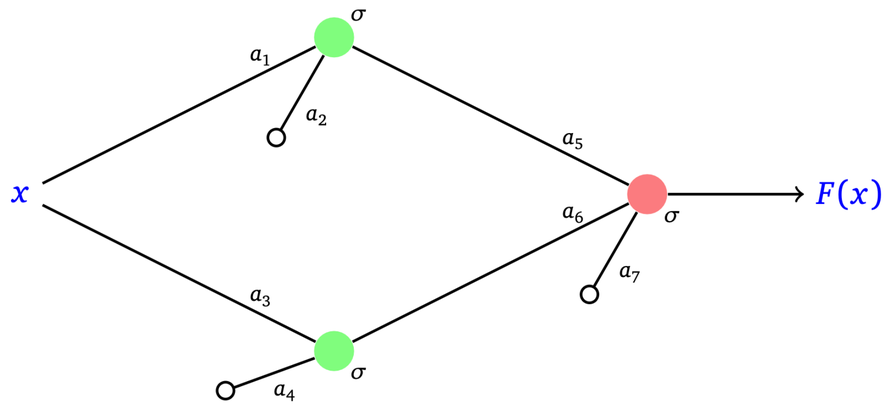
5) PROBLÈMES: initialisation des W et vitesse de cet algorithme

Calcul des poids

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



données



modèle

Essayer une descente de gradient directement sur les 7 poids du réseau.

Essayer ensuite avec rétro-propagation..

Essayer encore plus tard avec Pytorch.

Conclusion

- vectorisation pour optimiser le calcul des poids
- bibliothèques `Pytorch` (facebook), `Tensorflow` (google) au dessus de `numpy` pour la rétro-propagation
- l'apprentissage supervisé marche bien pour reconnaissance d'images, de langage naturel et la traduction automatique
- bien dimensionner le modèle pour éviter sous-apprentissage ou sur-apprentissage
- compression des données par convolution
- etc..