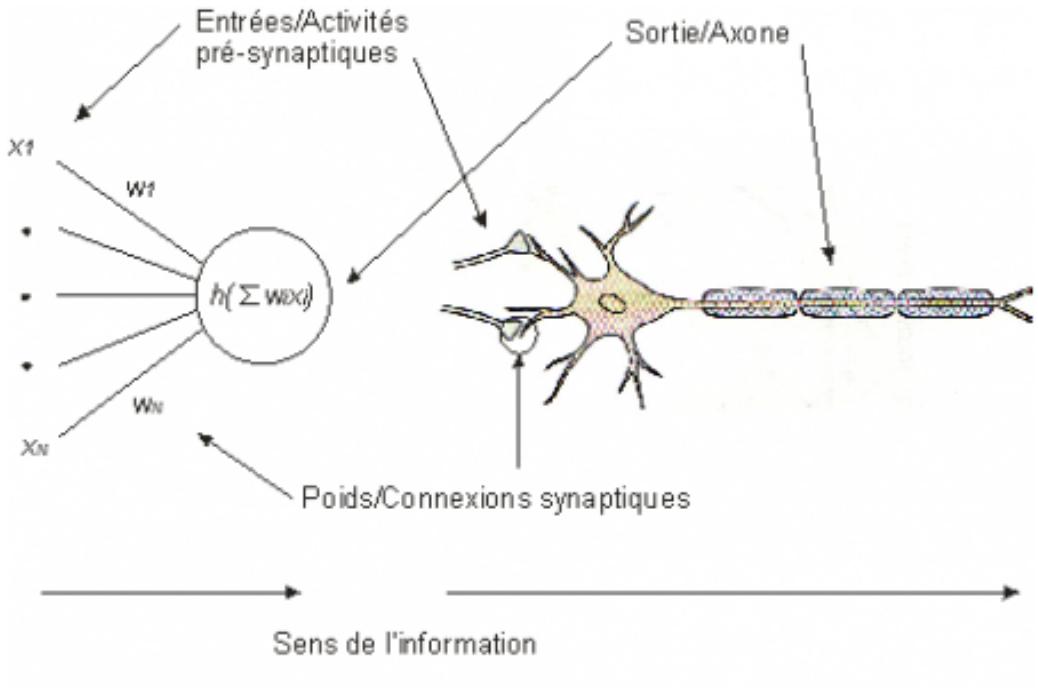
Réseau de neurones, rétropropagation du gradient de l'erreur

September 16, 2019

A la fin de ce TP, les mots clefs suivant devront vous être familié.

- réseau de neurone= réseau connectionniste → convolutional neuronal networks (CNN)
- entrainement=apprentissage= training
- couche d'entrée, couche de sortie, couche cachée, intercept=biais
- optimisation → descente de gradient → rétropropagation du gradient de l'erreur

1 Qu'est-ce qu'un réseau de neurone



1.1 Qu'est-ce qu'un neurone formel

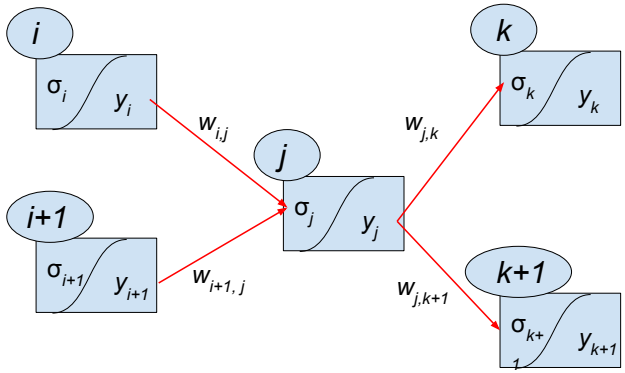
Un neurone générique j reçoit des poids des couches précédentes $prec(j)$, il sommes ces poids

$$\sigma_j = \sum_{i \in prec(j)} y_i w_{ij}$$

puis applique une fonction S

$$y_j = S(\sigma_j)$$

puis transmet y_j à tous ses successeurs $succ(j)$.



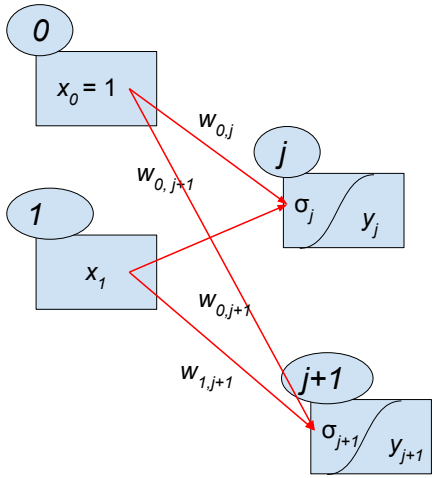
La fonction S est appelée fonction d'activation. Elle est nulle ou négative quand sont entrée est trop faible. C'est souvent une des fonctions suivantes:

- Sigmoide : $\frac{1}{1+e^{-x}}$.
- Relu : $x1_{\{x>0\}}$
- Arctan

L'important c'est que ces fonctions écrasent les petites valeurs et qu'elle soient non-linéaire.

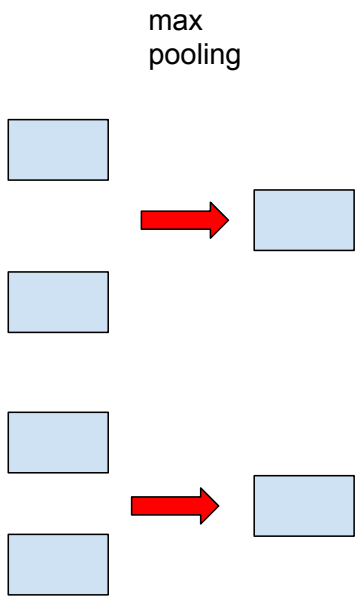
1.2 neurones d'entrées

Les neurones d'entrée sont particuliers: Leur valeurs $x_0, x_1, ...$ ne sont pas calculées, elle sont fournies dans les données. Souvent $x_0 = 1$ (quelles que soient les données). Les coefficients correspondant sont appelé biais ou intercept.



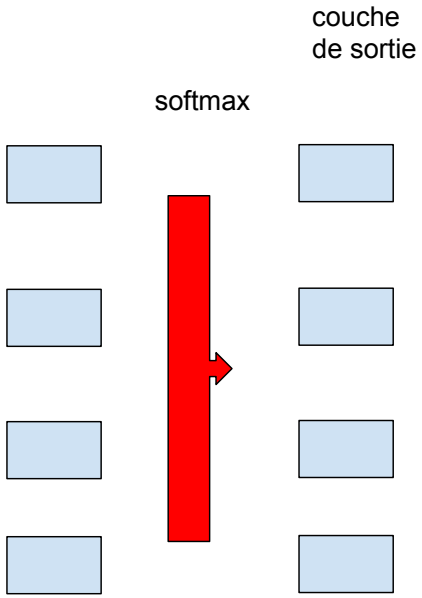
1.3 Transformation par paquet de neurones

La transmission générique de l'information se fait donc par sommation et transformation non-linaire. Cela immite les neurones naturels. Mais puisque l'on est artificiel, on peut se permettre d'autre manipulation. On peut aussi intercaler des transformations par paquets de neurones. Par exemple faire un maxpooling qui permet de réduire la taille d'une couche :


max
pooling

1.4 Neurones de sorties

Les neurones en bout de chaine, ceux qui n'ont pas de successeurs sont appelés neurones de sortie. Souvent, entre l'avant dernière couche et la couche de sortie, on intercale une fonction vectorielle, un sorte de traitement final



On utilise souvent la fonction de $\mathbb{R}^n$ dans $\mathbb{R}^n$ suivante :

$$\text{softmax}(y)_k = \frac{e^{y_k}}{\sum_j e^{y_j}}$$

Ainsi : les éléments de y sont passés à l'exponentiel, puis le résultat est renormalisé pour en faire un vecteur de probabilité. Les probabilité $\text{softmax}(y)_k$ seront grande là où y_k était grand.

1.5 De l'entrée à la sortie

Nous avons des neurones d'entrée In, des neurones de sorties Out, et les autres, les neurones cachés.

Ainsi un réseau de neurones prend en entrée un vecteur $(x_i)_{i \in In}$ et ressort un vecteur $(y_k)_{k \in Out}$.

- Exemple de réseau de neurones à une couche sans fonction d'activation :

$$\forall k \in Out \qquad y_k = w_{0k} + \sum_{i \in In} x_i w_{ik}$$

  C'est le fameux modèle ...

- Exemple de réseau de neurones à une couche avec une fonction softmax :

$$y = \text{softmax}(w_{0.} + \sum_{i \in In} x_i w_{i.})$$

  C'est le fameux modèle ...

- Exemple de réseau à deux couches :

$$\forall k \in Out \qquad y_k = S\Big[\tilde{w}_{0k} + \sum_j \tilde{w}_{jk}\ S\Big(w_{0j} + \sum_{i \in In} x_i w_{ij}\Big)\Big]$$

  C'est le multi-perceptron à deux couches.

1.6 Architecture

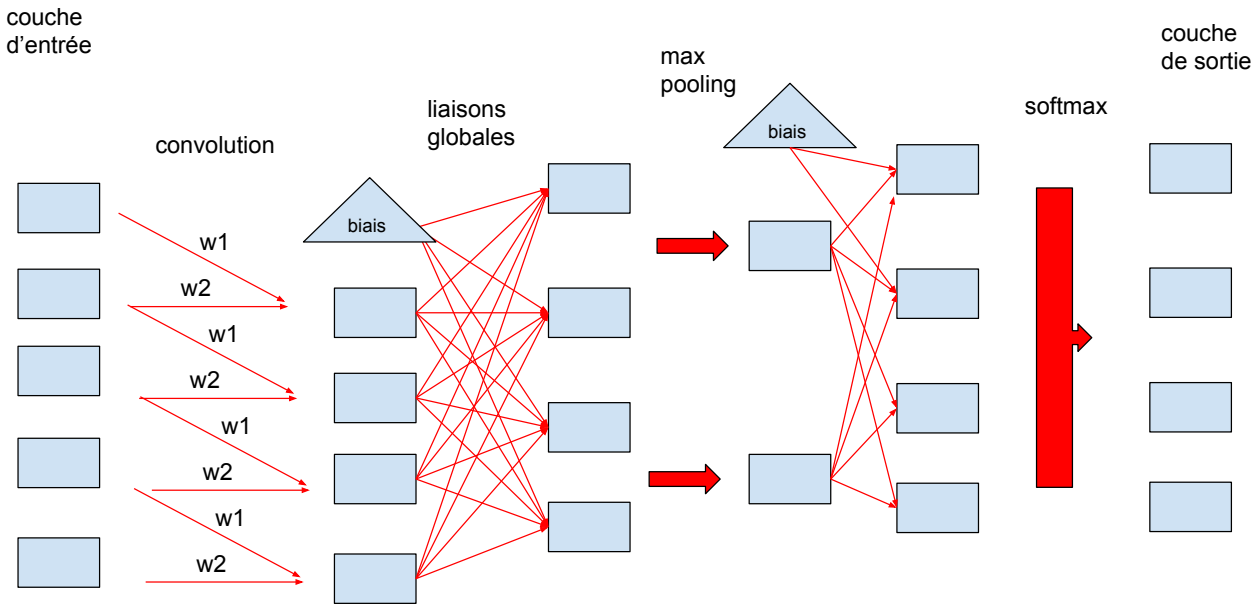
D'une couche à l'autre, on peut mettre toutes les liaisons (full connection), ou pas.

Il faut rajouter des biais à chaque couches intermédiare, qui permettent de recentrer les combinaisons linéaire.

On peut réutiliser les mêmes poids pour des liaisons différentes. C'est notamment le cas pour les réseaux de convolutions. Exemple:

$$y_j = x_{j-1}w_1 + x_j w_2 + x_{j+1}w_3$$

Trouver le bon réseau c'est l'art du modélisateur.



2 Apprentissage

2.1 Descente de gradient

Ainsi un réseau de neurone est une fonction f_w qui à $x = (x_i)_{i \in In}$ renvoie une sortie $y = (y_i)_{i \in Out}$.

Notre but est de trouver le meilleurs w pour que

$$dist(f_w, f^?) \sim 0$$

Mais $f^?$ fonction est inconnue dans son ensemble. On dispose seulement d'observations x^b, u^b. On considère la quantité

$$\sum_b dist\Big(f^?(x^b), f_w(x^b)\Big) = \sum_b dist\Big(u^b, y^b\Big)$$

Et on cherche à minimiser cette quantité.

Plus précisément, plutôt que de considérer toutes nos données d'un coup, on considére un paquet de point: $Batch$. On regarde l'erreur provoquée par ces points:

$$E = E(w) = \sum_b dist\Big(u^b, y^b\Big)$$

(on l'appelle aussi la loss).

On modifie w pour minimiser cette erreur:

$$w \to w - \ell \nabla E(w)$$

où $\ell > 0$ est un paramètre petit (ex 10e-3) appelé le learning rate.

Ici w est le tenseur représentant tous les paramètres (toutes les flèches). Pour un paramètre (i, j) particulier cela donne

$$w_{ij} \to w_{ij} - \ell \frac{\partial E}{\partial w_{ij}}$$

En suite on prend un second *Batch* et on recommence jusqu'à ce que l'on soit satisfait de l'ajustement.

Cet algorithme itératif s'appelle la descente de gradient. Il admet des tas de variantes, qui ont toutes en commun le fait de calculer des gradients. Mais comment calculer le gradient d'une fonction aussi complexe de E ?

2.2 Dérivation composée (Chain rule)

Mais comment calcule-t-on des dérivée déjà ? Ok, on sait le faire sur papier, mais avec un ordinateur ? Curieusement, la bonne réponse est venue tardivement (1985 environ). Ce s'appelle la rétropopagation de l'erreur. Et c'est cela qui a permit de développer le deep learning.

Analysons la dérivée de la fonction $t(x) = h \circ g \circ f(x)$. Son graphe de dépendance :

$$x \xrightarrow{f} y \xrightarrow{g} z \xrightarrow{h} t$$

Les accroissements infinitésimaux se multiplient (c'est la chain rule) :

$$\frac{\partial t}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial z}{\partial y}\frac{\partial t}{\partial z} \tag{1}$$

$$= f'(x)g'(y)h'(z) \tag{2}$$

Exercice : considérons les fonctions affines $f(x) = ax + A$ et $g(x) = bx + B$. Calculez explicitement $g \circ f(x + \varepsilon) - g \circ f(x)$.

En comparant cet exo et la chain-rule, vous comprendrez que : les accroissement infinitésimaux des fonctions lisses, se composent de la même manière que les accroissement des fonctions affines. En bref : toute fonction lisse est localement une fonction affine.

2.3 Fonction à plusieurs variables

Quand un fonction a plusieurs variables, $g(a, b, ..)$, ces dérivée partielles se calculent sans difficulté. Par ex, pour calculer $\frac{\partial g(a,b,..)}{\partial a}$ il suffit de considérer uniquement la fonction $a \to g(a, b, ..)$.

Par contre, quand une variable x intervient plusieurs fois:

$$z(x) = g[f_1(x), f_2(x), ...] = g[y_1, y_2, ...]$$

Graphe de dépendance :

$$x \xrightarrow{f} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \end{bmatrix} \xrightarrow{g} z$$

Les accroissements s'additionnent:

$$\frac{\partial z}{\partial x} = \sum_i \frac{\partial y_i}{\partial x} \frac{\partial z}{\partial y_i} = \sum_i f_i'(x) g'(y_i)$$

Avec ces 2 règles, on peut calculer tous les gradients que l'on veut.

2.4 Réseau classique

considérons un réseau de neurone classique (sans maxpooling, ou solftmax) :

$$\sigma_j = \sum_{i \in prec(j)} y_i w_{ij} \tag{3}$$

$$y_j = S(\sigma_j) \tag{4}$$

$$E = \sum_{j \in Out} \Big(y_j - u_j \Big)^2 \tag{5}$$

La dépendance $w_{ij} \to E$ se découpe en

$$w_{ij} \longrightarrow \sigma_j \longrightarrow y_j \longrightarrow E$$

ce qui donne pour les accroissements

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial \sigma_j}{\partial w_{ij}} \frac{\partial y_j}{\partial \sigma_j} \frac{\partial E}{\partial y_j}$$

Les fonctions $\sigma_j \to y_j$ et $w_{ij} \to \sigma_j$ sont explicites et se dérivent facilement :

$$\frac{\partial y_j}{\partial \sigma_j} = S'(\sigma_j)$$

$$\frac{\partial \sigma_j}{\partial w_{ij}} = y_i$$

Ainsi on a pour tout j :

$$\frac{\partial E}{\partial w_{ij}} = \quad y_i \quad S'(\sigma_j) \quad \frac{\partial E}{\partial y_j}$$

Dans le cas $j \in Out$, la fonction $y_j \to E$ est explicite et se dérive ainsi :

$$\frac{\partial E}{\partial y_j} = 2(y_j - u_j)$$

Nous savons donc calculer $\frac{\partial E}{\partial w_{ij}}$ pour les $j \in Out$ et nous les calculons en premier.

2.5 seconde étape

Si maintenant j ne correspond pas à un neurone de sortie ($j \notin Out$), disons qu'ils se situe juste dans la couche avant.

La dépendance $y_j \to E$ se fait via les variables $(y_k)_{k \in succ(j)}$.

$$y_j \longrightarrow \begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \end{bmatrix} \longrightarrow E$$

Que l'on peut détaillé ainsi

$$y_j \longrightarrow \begin{bmatrix} \sigma_k \to y_k \\ \sigma_{k+1} \to y_{k+1} \\ \vdots \end{bmatrix} \longrightarrow E$$

$$\frac{\partial E}{\partial y_j} = \sum_{k \in succ(j)} \frac{\partial \sigma_k}{\partial y_j} \frac{\partial y_k}{\partial \sigma_k} \frac{\partial E}{\partial y_k}$$
$$= \sum_{k \in succ(j)} w_{jk} \; S'(y_k) \; \frac{\partial E}{\partial y_k}$$

Si j est un neurone de l'avant dernière couche, k est un neurone de la dernière couche, et nous avons calculer $\frac{\partial E}{\partial y_k}$ dans la section précédente.

Et si j était encore plus loin de la sortie? Il faudrait itéré.

Ainsi toutes les dérivées partielles sont calculée, en commençant par la sortie, et en remondant vers l'entrée; c'est la rétropropagation de l'erreur.