Réseaux de neurones artificiels

Introduction

Depuis un quart de siècle, les réseaux de neurones ont démontré leur efficacité dans la classification des ressources numériques. Au delà de la métaphore biologique, les réseaux de neurones artificiels constituent un outil précieux dans des domaines aussi variés que l'analyse de données, la prévision, la classification ou la reconnaissance de formes.

Comment mettre en oeuvre un réseau de neurones pour construire des prévisions de températures sur la base de données météorologiques ? Après avoir défini les composantes d'un réseau de neurones et le principe de l'apprentissage automatisé, nous simulons ce réseau et l'appliquons à notre problème.

Je remercie le Professeur Mathias Quoy, Directeur du laboratoire ETIS - Neurocybernétique de l'Université de Cergy-Pontoise pour ses remarques, ses conseils et le temps qu'il a bien voulu m'accorder lors de ce travail de TIPE (cf Annexe).

1 Généralités sur les réseaux de neurones

1.1 Bref historique

Le neurone biologique fut mis en évidence en 1870 par Golgi et Cajal. Le modèle du neurone formel, construit par analogie avec le neurone biologique, apparaît pour la première fois en 1943 dans les travaux de Mc Culloch et Pitts.

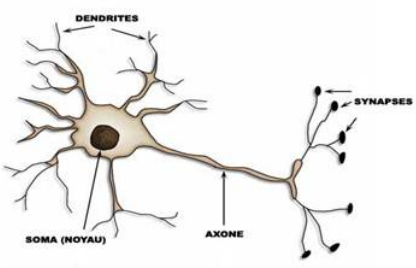


Fig. 1 : schéma d'un neurone biologique

En 1949, Hebb propose une règle d'apprentissage par renforcement des couplages synaptiques, fondamentale pour l'apprentissage supervisé. En 1958, Rosenblatt simule le premier réseau artificiel : le perceptron simple. Les réseaux multicouches apparaissent en 1986, avec l'algorithme de rétropropagation du gradient.

1.2 Modèle du neurone artificiel

1.2.1 Définition

Le neurone artificiel **[1]** est constitué d'un *vecteur poids* $\vec{w} = (w_1, \ldots, w_p) \in \mathbb{R}^p$ (p est la dimension du neurone), d'un paramètre de *biais* $\sigma \in \mathbb{R}$ et d'une *fonction d'activation* $f : \mathbb{R} \to [-1, 1]$ (il s'agit d'une convention, on peut également prendre une fonction à valeurs dans $[0, 1]$).

Le neurone reçoit en entrée un vecteur $\vec{x} = (x_1, \ldots, x_p) \in \mathbb{R}^p$. Il renvoie en sortie la valeur :

$$s(\vec{x}) = f(\langle \vec{x}, \vec{w} \rangle - \sigma) = f\big(\big(\sum_{i=1}^{p} x_i w_i\big) - \sigma\big)$$

On note $h = \big(\sum_{i=1}^{p} x_i w_i\big) - \sigma$, de sorte que $s(\vec{x}) = f(h)$.

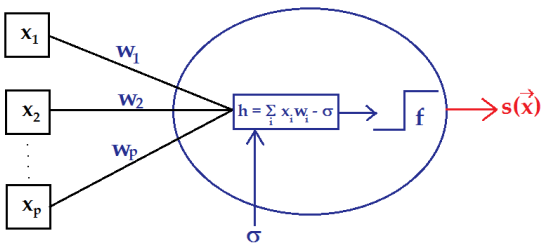


Fig. 2 : modèle du neurone artificiel

1.2.2 Fonctions d'activation

On dispose de quelques fonctions d'activation qui sont souvent utilisées. Les plus courantes sont les suivantes :

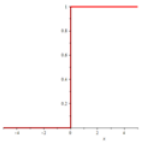
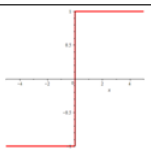
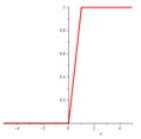
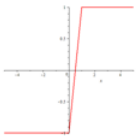
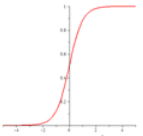
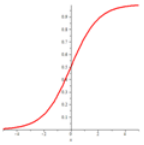
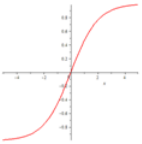
Nom	Valeur de la fonction Φ normalisée entre 0 et 1	Représentation de Φ	Valeur de la fonction Ψ normalisée entre -1 et 1	Représentation de Ψ
Fonction de Heaviside	$\Phi(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ 1 & \text{si } x > 0 \end{cases}$		$\Psi(x) = \begin{cases} -1 & \text{si } x < 0 \\ 1 & \text{si } x > 0 \end{cases}$ et $\Psi(0) = 0$ *note : la valeur en 0 est indifférente*	
Linéaire saturée	$\Phi(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } 0 \leq x \leq 1 \\ 1 & \text{si } x > 1 \end{cases}$		$\Psi(x) = \begin{cases} -1 & \text{si } x < 0 \\ 2x - 1 & \text{si } 0 \leq x \leq 1 \\ 1 & \text{si } x > 1 \end{cases}$	
Tangente hyperbolique	$\Phi(x) = \frac{1}{2}(\frac{e^x - e^{-x}}{e^x + e^{-x}} + 1)$		$\Psi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	
Sigmoïde	$\Phi(x) = \frac{1}{1+e^{-x}}$		$\Psi(x) = 2\frac{1}{1+e^{-x}} - 1$	

Fig. 3 : les principales fonctions d'activation

1.2.3 Elimination du paramètre de biais

Le paramètre de biais induit une asymétrie dans le fonctionnement du neurone. Il est avantageux de le supprimer. Pour cela, on le remplace par un poids supplémentaire $w_0 = \sigma$. Le vecteur de poids devient $\vec{w'} = (w_0, w_1, \ldots, w_p) \in \mathbb{R}^{p+1}$. L'entrée est également modifiée : on ajoute un paramètre $x_0 = -1$, de sorte que $\vec{x'} = (x_0, x_1, \ldots, x_p) \in \mathbb{R}^{p+1}$. La sortie n'est pas modifiée :

$$s'(\vec{x}) = f\big(\big\langle \vec{x'}, \vec{w'} \big\rangle\big) = f\big(\sum_{i=0}^{p} x_i w_i\big) = f\big(\big(\sum_{i=1}^{p} x_i w_i\big) - w_0\big) = s(\vec{x})$$

Dans la suite, quand on considérera des *neurones sans biais*, la première coordonnée des entrées sera prise égale à -1.

1.3 Modèle du réseau de neurones

On considère les réseaux de neurones multicouches **[2]** non bouclés.

1.3.1 Notations

Soit p le nombre de coordonnées de l'entrée. On appelle p la dimension du réseau.

On note N le nombre de couches dans le réseau, et $\mathcal{C}^n$ la n-ème couche du réseau, pour $n \in \{1, \dots, N\}$.
Les $N-1$ premières couches sont appelées "couches cachées", la dernière est la "couche de sortie".
Soit $f^{(n)}$ la fonction d'activation de la couche n.
On note $m^{(n)}$ le nombre de neurones dans la couche n, pour $n \in \{1, \dots, N\}$. On a toujours $m^{(N)} = 1$ (la sortie du réseau est un scalaire) et on prend $m^{(0)} = p$

On note $\mathcal{N}_i^{(n)}$ le i-ème neurone de la couche n, pour $i \in \left\{1, \dots, m^{(n)}\right\}$. Ce neurone a un vecteur poids noté $\vec{w}_i^{(n)}$ qui a $m^{(n-1)}$ coordonnées. On le note $\vec{w}_i^{(n)} = (w_{i,j}^{(n)})_{1 \leq j \leq m^{n-1}}$

1.3.2 Calcul de la sortie

Le réseau reçoit une entrée $\vec{x^{in}} = (x_1, \dots, x_p) \in \mathbb{R}^p$, et calcule de proche en proche une sortie $s(\vec{x^{in}})$.
On définit la suite : $(\vec{x}^{(n)})_{0 \leq n \leq N} = ((x_1^{(n)}, \dots, x_{m^{(n)}}^{(n)}))_{0 \leq n \leq N}$ par :
$\vec{x}^{(0)} = \vec{x^{in}}$ et

$$\forall n \in \{1, \dots, N\}, \ \vec{x}^{(n)} = \left(f^{(n)}\left(\sum_{j=1}^{m^{(n-1)}} x_j^{(n-1)} w_{1,j}^{(n)}\right), \dots, f^{(n)}\left(\sum_{j=1}^{m^{(n-1)}} x_j^{(n-1)} w_{m^{(n)},j}^{(n)}\right)\right)$$

Pour chaque couche n, le vecteur reçu en entrée par chaque neurone $\mathcal{N}_i^{(n)}$ est le vecteur $\vec{x}^{(n-1)}$; il renvoie le scalaire $x_i^{(n)}$. La couche n renvoie donc le vecteur $\vec{x}^{(n)} = (x_i^{(n)})_{1 \leq i \leq m^{(n)}}$.
Le réseau renvoie le vecteur à une coordonnée $s(\vec{x}^{in}) = \vec{x}^{(N)} = x^{out}$.

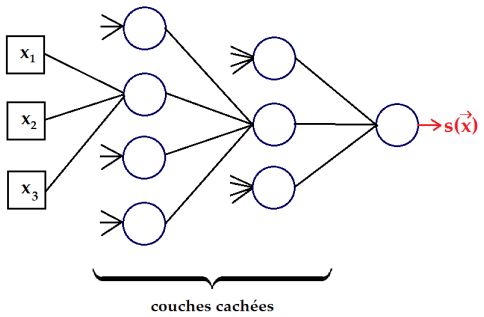


Fig. 4 : réseau de neurones à 2 couches cachées

Sur le schéma, on a omis certaines connexions pour plus de clarté. Chaque neurone d'une même couche reçoit la même entrée $\vec{x}^{(n)}$.
Sur l'exemple, on a

$$s(\vec{x}^{in}) = f^{(3)}\left(\sum_{j=1}^{3} w_{1,j}^{(3)} \cdot f^{(2)}\left(\sum_{i=1}^{4} w_{j,i}^{(2)} \cdot f^{(1)}\left(\sum_{k=1}^{3} w_{i,k}^{(1)} x_k^{in}\right)\right)\right)$$

1.4 Premier exemple

On cherche à créer un réseau qui prend en entrée un vecteur (x_1, x_2) et qui renvoie en sortie $x_1 \text{ET} x_2$, conformément à la table suivante :

x_1	x_2	$x_1 \text{ET} x_2$
-1	-1	-1
-1	1	-1
1	-1	-1
1	1	1

On prend un réseau élémentaire constitué d'une couche à un neurone : il s'agit d'un perceptron. On choisit une fonction de Heaviside entre -1 et 1.
On remarque que le poids $\vec{w} = (1, 1)$ et le biais $\sigma = -1,5$ permettent de répondre au problème. Cette solution n'est pas unique.

Cependant, la plupart du temps les poids ne sont pas aussi évidents à ajuster. Le problème de l'apprentissage réside dans la façon dont les poids adéquats sont déterminés pour répondre au problème posé.

2 Apprentissage supervisé

L'apprentissage consiste en la présentation d'exemples où l'on fournit à la fois l'entrée et la sortie souhaitée. Le réseau doit adapter ses poids pour donner la sortie adéquate pour ces exemples. S'ils sont suffisamment nombreux et représentatifs, le réseau sera opérationnel sur des entrées inconnues.

2.1 Règle de Hebb

« Des neurones stimulés en même temps sont des neurones qui se lient ensemble. » Autrement dit, la règle de Hebb consiste à renforcer les poids synaptiques entre les neurones simultanément actifs.

Les poids des connexions entre neurones sont mis à jour entre les instants t et $t+1$ de la façon suivante :

$$w_{i,j}(t+1) = w_{i,j}(t) + \eta x_i x_j$$

où $w_{i,j}$ est le poids de la liaison entre deux neurones pré et post-synaptiques d'activation respective x_i et x_j dans $[-1,1]$; et $\eta \in \mathbb{R}_+^\star$ est le pas.

On remarque que si x_i et x_j sont tous deux proches de 1 ou -1, $w_{i,j}$ augmente car les neurones ont été actifs ou inactifs en même temps.

En revanche, si $x_i \simeq 1$ et $x_j \simeq -1$ (ou l'inverse), $w_{i,j}$ diminue car les neurones ont eu des comportements opposés.

2.2 Apprentissage du perceptron

Dans cette partie, on considère un unique neurone dont la fonction d'activation est une fonction de Heaviside normalisée entre -1 et 1.

2.2.1 Algorithme du perceptron

On suppose que l'on dispose d'une base d'exemples $\mathcal{B} = ((\vec{x}_k, t_k))_{1\leq k\leq n}$ constituée de $n \geq 1$ couples $(\vec{x}_k, t_k)$, où $\vec{x}_k$ est une entrée et $t_k \in \{-1, 1\}$ est la sortie souhaitée pour cette entrée.

Entrées :
η, $((\vec{x}_k, t_k))_{1\leq k\leq n}$
Initialiser le poids (on peut aussi l'initialiser aléatoirement) :
$\vec{w} \longleftarrow \vec{0}$
Jusqu'à stabilisation, répéter :
    Choisir un vecteur d'entrée $\vec{x}_k$
    Calculer la sortie $s(\vec{x}_k)$
    Mettre à jour le vecteur de poids :
        $\vec{w} \longleftarrow \vec{w} + \eta(t_k - s(\vec{x}_k))\vec{x}_k$
        (si $s(\vec{x}_k) = t_k$, les poids sont inchangés)

Les exemples de $\mathcal{B}$ peuvent être présentés dans l'ordre ou aléatoirement.

La stabilisation correspond soit à un nombre d'itérations fixées, soit au fait qu'aucune erreur n'ait été commise sur les exemples de $\mathcal{B}$, soit à un pourcentage de réussite fixé sur les exemples de $\mathcal{B}$.

2.2.2 Séparation linéaire

Soit $\mathcal{B} = ((\vec{x}_k, t_k))_{1\leq k\leq n}$ un ensemble d'exemples où les $\vec{x}_k \in \mathbb{R}^p$.

$\mathcal{B}$ est linéairement séparable en deux classes $\mathcal{B}^+$ et $\mathcal{B}^-$ telles que $\begin{cases} \mathcal{B}^+ \cap \mathcal{B}^- = \emptyset \\ \mathcal{B}^+ \cup \mathcal{B}^- = \mathcal{B} \end{cases}$ lorsqu'il existe un hyperplan affine $\mathcal{H} = \{\vec{x} \in \mathbb{R}^p / \langle \vec{w}, \vec{x}\rangle - \sigma = 0\}$ tel que $\begin{cases} \langle \vec{w}, \vec{x}\rangle - \sigma > 0 & \text{si } \vec{x} \in \mathcal{B}^+ \\ \langle \vec{w}, \vec{x}\rangle - \sigma < 0 & \text{si } \vec{x} \in \mathcal{B}^- \end{cases}$. On a $\dim \mathcal{H} = p - 1$.

Visuellement cela signifie qu'on peut trouver un hyperplan affine qui sépare l'espace en deux parties $\mathcal{B}^+$ et $\mathcal{B}^-$.

En considérant les entrées $\vec{x}'_k \in \mathbb{R}^{p+1}$ où la première coordonnée vaut -1, $\mathcal{B}$ est linéairement séparable si et seulement si il existe un hyperplan vectoriel $H = \{\vec{x'} = (-1, x_1, ..., x_p) \in \mathbb{R}^{p+1} / \langle \vec{w'}, \vec{x'}\rangle = 0\}$ tel que $\begin{cases} \langle \vec{w'}, \vec{x'}\rangle > 0 & \text{si } \vec{x}' \in \mathcal{B}^+ \\ \langle \vec{w'}, \vec{x'}\rangle < 0 & \text{si } \vec{x'} \in \mathcal{B}^- \end{cases}$.
On a de même $\dim H = p - 1$.

Le perceptron simple classifie correctement des exemples si et seulement si ils sont linéairement séparables. Dans ce cas $\vec{w}$ est le vecteur poids et σ le biais.

2.2.3 Convergence de l'algorithme

Montrons que l'algorithme du perceptron converge **[3]** vers des poids qui classifient correctement tous les exemples de $\mathcal{B}$.

Théorème

Supposons qu'il existe un vecteur $\vec{w^\star}$ de norme 1, et $\gamma > 0$ tel que $\forall k \in \{1, \ldots, n\}$, $t_k \cdot \langle \vec{x}_k, \vec{w^\star} \rangle \geq \gamma$. On note $R = \max\{\| \vec{x}_k \|\}_{1 \leq k \leq n}$.

Alors le perceptron fait au plus $\dfrac{R^2}{\gamma^2}$ erreurs (une erreur étant un instant où $s(\vec{x}_k) \neq t_k$ pour un couple $(\vec{x}_k, t_k) \in \mathcal{B}$).

La première hypothèse du théorème correspond au fait que $\mathcal{B}$ soit linéairement séparable : $\vec{w^\star}$ est un vecteur normal à l'hyperplan affine séparateur. γ correspond à une marge entre les deux classes : $\begin{cases} \langle \vec{w^\star}, \vec{x} \rangle \geq \gamma & \text{si } \vec{x} \in \mathcal{B}^+ \\ \langle \vec{w^\star}, \vec{x} \rangle \leq -\gamma & \text{si } \vec{x} \in \mathcal{B}^- \end{cases}$.

Preuve du théorème

Notons $\vec{w}_k$ le vecteur poids du perceptron quand il fait sa k-ème erreur. On a $\vec{w}_1 = \vec{0}$. Soit $k \geq 1$. Supposons que la k-ème erreur soit faite sur l'exemple i.

On a $\langle \vec{w}_{k+1}, \vec{w^\star} \rangle = \langle \vec{w}_k + \eta t_i \vec{x}_i, \vec{w^\star} \rangle = \langle \vec{w}_k, \vec{w^\star} \rangle + \eta t_i \langle \vec{x}_i, \vec{w^\star} \rangle \geq \langle \vec{w}_k, \vec{w^\star} \rangle + \eta\gamma$.

Comme $\langle \vec{w}_1, \vec{w^\star} \rangle = 0$, on a, par récurrence sur k, $\langle \vec{w}_{k+1}, \vec{w^\star} \rangle \geq k\eta\gamma$.

De plus, d'après l'inégalité de Cauchy-Schwarz, $\| \vec{w}_{k+1} \| \| \vec{w^\star} \| \geq \langle \vec{w}_{k+1}, \vec{w^\star} \rangle \geq k\eta\gamma$. Or $\| \vec{w^\star} \| = 1$, donc

$$\| \vec{w}_{k+1} \| \geq k\eta\gamma \qquad (1)$$

Majorons maintenant $\| \vec{w}_{k+1} \|$:

$\| \vec{w}_{k+1} \|^2 = \| \vec{w}_k + \eta t_i \vec{x}_i \|^2 = \| \vec{w}_k \|^2 + \eta^2 t_i^2 \| \vec{x}_i \|^2 + 2\eta t_i \langle \vec{w}_k, \vec{x}_i \rangle$. Or $t_i \langle \vec{w}_k, \vec{x}_i \rangle \leq 0$ car le perceptron a fait une erreur sur l'exemple i, et $t_i^2 = 1$, donc $\| \vec{w}_{k+1} \|^2 \leq \| \vec{w}_k \|^2 + \eta^2 R^2$.

Comme $\vec{w}_1 = \vec{0}$, il vient, par récurrence sur k :

$$\| \vec{w}_{k+1} \|^2 \leq k\eta^2 R^2 \qquad (2)$$

D'où, avec (1) et (2) :

$$k \leq \frac{R^2}{\gamma^2}$$

Notons que la convergence ne dépend pas du pas η. Il est en général pris proche de 0.

2.2.4 Un exemple de convergence du perceptron

On soumet un perceptron au problème suivant. Etant donné un point $M = (x, y) \in [-10, 10]^2$, le perceptron doit renvoyer 1 si M est en dessous de la droite $\mathcal{D}$: $y - x = 2$ et -1 sinon.

On lui présente une base de n exemples avec laquelle il apprend jusqu'à convergence, puis on le teste sur des exemples inconnus. On observe pour 50 exemples une convergence vers les poids $\mathsf{W} = [6.922, -3.876, 3.717]$. Cela correspond à une droite d'équation $-3,876x + 3,717y = 6,922$, soit, en normalisant : $0,959y - x = 1,786$, proche du résultat attendu.

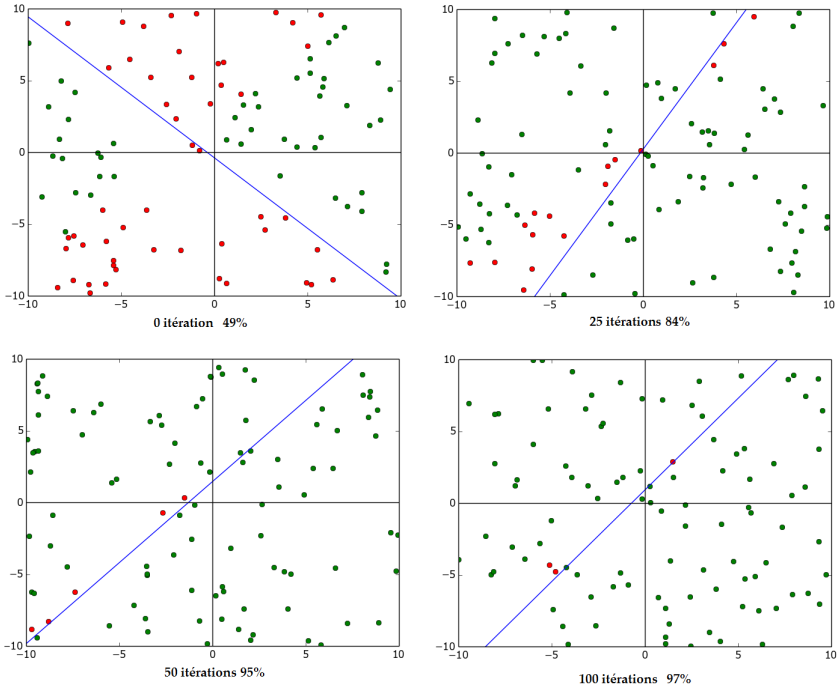


Fig. 5 : résultats de l'expérience (les points correctement classés sont en vert, les erreurs en rouge)

2.3 Apprentissage d'un réseau de neurones

2.3.1 Optimisation par descente de gradient

L'algorithme de descente de gradient permet d'approximer la position d'un minimum local d'une fonction f à valeurs réelles différentiable.

Le principe est le suivant : on prend x_0 proche d'un minimum, et pour $n \geq 1$, $x_{n+1} = x_n - \eta \nabla f(x_n)$. Si f est suffisamment régulière, et η proche de 0, (x_n) converge vers un minimum.

Le principal inconvénient de cette méthode est que l'on peut obtenir un minimum local et non global. Il est donc pertinent de relancer plusieurs fois l'algorithme avec des poids initialisés aléatoirement.

2.3.2 Rétropropagation du gradient

On considère une base d'exemples indexée par μ. L'erreur quadratique sur l'exemple μ est notée $E(\mu) = \frac{1}{2}(t(\mu) - x^{out}(\mu))^2$.

On définit la fonction d'erreur quadratique globale : $E = \sum_{\mu} E(\mu)$. On cherche à la minimiser. Pour cela, on lui applique l'algorithme de descente du gradient **[4].**

Justification de l'algorithme de rétropropagation *(je propose cette méthode, elle peut donc certainement être améliorée)*

E étant une fonction des $\left(w_{i,j}^{(n)}\right)_{i,j,n}$, la mise à jour des poids doit se faire de la façon suivante : $\Delta w_{i,j}^{(n)} = -\eta \dfrac{\partial E}{\partial w_{i,j}^{(n)}}$, avec η suffisamment petit.

Notons $h_i^{(n)}(\mu) = \sum_{j=1}^{m^{(n-1)}} w_{i,j}^{(n)} x_j^{(n-1)}(\mu)$. En dérivant, on a : $\dfrac{\partial E}{\partial w_{i,j}^{(n)}} = \sum_{\mu} \dfrac{\partial E(\mu)}{\partial h_i^{(n)}(\mu)} \dfrac{\partial h_i^{(n)}(\mu)}{\partial w_{i,j}^{(n)}} = -\sum_{\mu} \delta_i^{(n)}(\mu) x_j^{(n-1)}(\mu)$, en posant $\delta_i^{(n)}(\mu) = -\dfrac{\partial E(\mu)}{\partial h_i^{(n)}(\mu)}$.

Donc $\Delta w_{i,j}^{(n)} = -\eta \sum_{\mu} \delta_i^{(n)}(\mu) x_j^{(n-1)}(\mu)$. Il reste à calculer les $\delta_i^{(n)}(\mu)$.

- *Pour la couche de sortie ($n = N$) :*

$E(\mu) = \frac{1}{2}(t(\mu) - f^{(N)}(h_1^{(N)}(\mu)))^2$, donc $\dfrac{\partial E(\mu)}{\partial h_1^{(n)}(\mu)} = \left(t(\mu) - f^{(N)}(h_1^{(N)}(\mu))\right) f'^{(N)}(h_1^{(N)}(\mu)) = \left(t(\mu) - x^{out}(\mu)\right) f'^{(N)}(h_1^{(N)}(\mu))$.

$$\delta_1^{(N)}(\mu) = \left(t(\mu) - x^{out}(\mu)\right) f'^{(N)}(h_1^{(N)}(\mu)) \qquad (1)$$

- *Pour les autres couches* $(1 \leq n \leq N-1)$ *:*

$$\delta_j^{(n-1)}(\mu) = -\frac{\partial E(\mu)}{\partial h_j^{(n-1)}(\mu)} = -\sum_{i=1}^{m^{(n)}} \frac{\partial E(\mu)}{\partial h_i^{(n)}(\mu)} \frac{\partial h_i^{(n)}(\mu)}{\partial h_j^{(n-1)}(\mu)} = \sum_{i=1}^{m^{(n)}} \delta_i^{(n)}(\mu) \frac{\partial h_i^{(n)}(\mu)}{\partial h_j^{(n-1)}(\mu)}.$$

Or $h_i^{(n)}(\mu) = \sum_{j=1}^{m^{(n-1)}} w_{i,j}^{(n)} x_j^{(n-1)}(\mu) = \sum_{j=1}^{m^{(n-1)}} w_{i,j}^{(n)} f^{(n-1)}(h_j^{(n-1)}(\mu))$. Donc $\frac{\partial h_i^{(n)}(\mu)}{\partial h_j^{(n-1)}(\mu)} = w_{i,j}^{(n)} f'^{(n-1)}(h_j^{(n-1)}(\mu))$

$$\delta_j^{(n-1)}(\mu) = f'^{(n-1)}(h_j^{(n-1)}(\mu)) \sum_{i=1}^{m^{(n)}} \delta_i^{(n)}(\mu) w_{i,j}^{(n)} \qquad (2)$$

Les équations (1) et (2) définissent les $\delta_i^{(n)}(\mu)$ par récurrence, d'où le terme de rétropropagation du gradient.

2.3.3 Algorithme BackProp

Entrées : η, base d'exemples $(\vec{x}^{in}(\mu), t(\mu))_\mu$
Initialiser les poids aléatoirement
Pour tout exemple μ :
Présenter $\vec{x}^{in}(\mu)$ en entrée
Propager le signal en avant $x_k^{(n-1)} \rightsquigarrow x_j^{(n)}$

$$x_j^{(n)} = f^{(n)}(h_j^{(n)}) = f^{(n)}\Big(\sum_{k=1}^{m^{(n-1)}} w_{j,k}^{(n)} x_k^{(n-1)}\Big)$$

Calculer l'erreur de sortie
$\delta_1^{(N)}(\mu) = (t(\mu) - x^{out}(\mu)) f'^{(N)}(h_1^{(N)}(\mu))$
Rétro-propager l'erreur en arrière $\delta_i^{(n)} \rightsquigarrow \delta_j^{(n-1)}$

$$\delta_j^{(n-1)}(\mu) = f'^{(n-1)}(h_j^{(n-1)}(\mu)) \sum_{i=1}^{m^{(n)}} \delta_i^{(n)}(\mu) w_{i,j}^{(n)}$$

Mettre à jour les poids
$w_{i,j}^{(n)} \leftarrow w_{i,j}^{(n)} + \eta \delta_i^{(n)} x_j^{(n-1)}$

2.3.4 Apprentissage sur un problème non linéairement séparable : le XOR

On peut facilement montrer que le problème du OU exclusif (XOR) n'est pas linéairement séparable. Un perceptron ne peut donc pas le calculer. On a donc utilisé un perceptron multicouche avec $\eta = 0,1$, une fonction d'activation tangente hyperbolique, et une architecture à deux couches cachées de 3 neurones, sur une base d'apprentissage exhaustive. La figure de gauche montre les sorties calculées en fonction du nombre d'itérations ; on observe qu'il y a bien convergence. Celle de droite montre le résultat après 500 itérations.

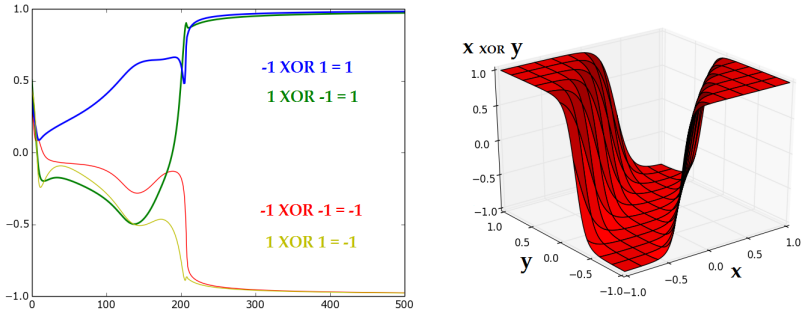


Fig. 6 : simulation du XOR avec un réseau multicouche

3 Application à la prévision des températures

3.1 Développement d'un réseau de neurones

J'ai développé, au cours de ce TIPE, un programme Python permettant de simuler n'importe quel réseau de neurones non bouclé. Il est possible de choisir l'architecture du réseau (c'est-à-dire le nombre de neurones, de couches...) Ce

programme est donc potentiellement adaptable à un grand nombre de problèmes résolubles par réseau de neurones. Je l'ai utilisé pour traiter chacun des exemples présentés dans ce TIPE.

```
f=vectorize(lambda x:tanh(x))
ff=vectorize(lambda x:1-f(x)**2)
g=vectorize(lambda x:1/2*log((1+x)/(1-x)))

class Neurone:
    def __init__(self,couche,l,fact,num):
        # initialisation aléatoire des poids
        self.poids=2*rand(l)-ones(l)
        self.couche=couche
        self.fact=fact
        self.num=num
    def sortie(self,entree):
        assert len(entree)==len(self.poids), 'erreur dimension'
        S=self.fact(np.vdot(self.poids,entree))
        return S
    def majpoids(self,nvpoids):
        self.poids=nvpoids

class Couche:
    def __init__(self,taille,dim,num):
        #neurones = liste des objets
        self.neurones=[Neurone(num,dim,f,i) for i in range(taille)]
        self.taille=taille
        self.dim=dim
        self.ncouche=num
    def sortiecouche(self,entree):
        #entree : vecteur entree a inserer ds chaq neur
        vectsort=[self.neurones[i].sortie(entree) for i in range (self.taille)]
        return vectsort

class Reseau:
    def __init__(self,archi):
        self.archi=archi
        #archi:[couche,couche,couche...]
    def sortieres(self,entree):
        S=entree
        for i in range(len(self.archi)):
            S=self.archi[i].sortiecouche(S)
        return S[0]
    def apprendre(self,Base,iterations,eta):
        n=len(self.archi)
        #n=nbre de couches
        for a in range(iterations):
            for E in Base:
                S=self.sortieres(E[0])
                X=[E[0]]
                for C in self.archi:
                    X.append(C.sortiecouche(X[-1]))
                X=X[1:]
                # calcul des h
                H=[g(X[i]) for i in range (len(X))]
                D=[[] for i in range(n)]
                # calcul des delta
                D[-1]=array([[ff(H[-1][k])*(E[1]-S) for k in range(self.archi[-1].taille)]])
                for i in range(2,n+1):
                    C=self.archi[-i]
                    Csucc=self.archi[-i+1]
                    D[-i]=[eta*ff(H[-i][j])*sum([Csucc.neurones[k].poids[j]*D[-i+1][k] for k in range(len(D[-i+1]))]) for j in range(C.taille)]
                X.append(E[0])
                # mise a jour des poids
                for C in self.archi:
                    for N in C.neurones:
                        N.majpoids(array(N.poids)+D[C.ncouche][N.num]*array(X[C.ncouche-1]))

"""Exemple de réseau de dimension 4, à 2 couches cachées de 3 neurones"""
C1=Couche(3,4,0)
C2=Couche(3,3,1)
CS=Couche(1,3,2)
R=Reseau([C1,C2,CS])
```

Estimation de la complexité de l'algorithme d'apprentissage :

On considère une base d'apprentissage de n exemples et un réseau à m neurones. Si l'on répète l'apprentissage a fois, l'algorithme de rétropropagation du gradient fait $O(a \times n \times m)$ opérations élémentaires (produits scalaires). Cette complexité devient vite très grande quand on considère de grands réseaux, comme cela est nécessaire pour les applications à la reconnaissance de caractères. Les temps de calculs sont alors trop importants, et l'algorithme de rétropropagation du gradient n'est plus adapté. Il existe d'autres algorithmes plus adaptés **[5]**, comme celui de KOHONEN, que je n'ai pas étudié ici.

3.2 Prévision des températures

3.2.1 Principe

On cherche, grâce à un réseau de neurones, à prévoir la température d'un lieu, connaissant les températures antérieures.

Pour cela, j'ai utilisé les données météorologiques fournies par le site http://www.infoclimat.fr/. Je disposais donc des températures relevées par la station Limoges-Bellegarde toutes les heures, entre le 1er et le 15 mars 2015. J'ai entraîné le réseau pendant les 5 premiers jours. Les 10 jours suivants, le réseau doit prévoir des températures à partir de situations qu'il n'avait jamais rencontrées.

On donne en entrée les températures des p instants précédents : $T(t_{i-p}), \ldots, T(t_{i-1})$, il doit renvoyer en sortie la

température $T(t_i)$.

3.2.2 Résultats et influence des paramètres

Architecture du réseau

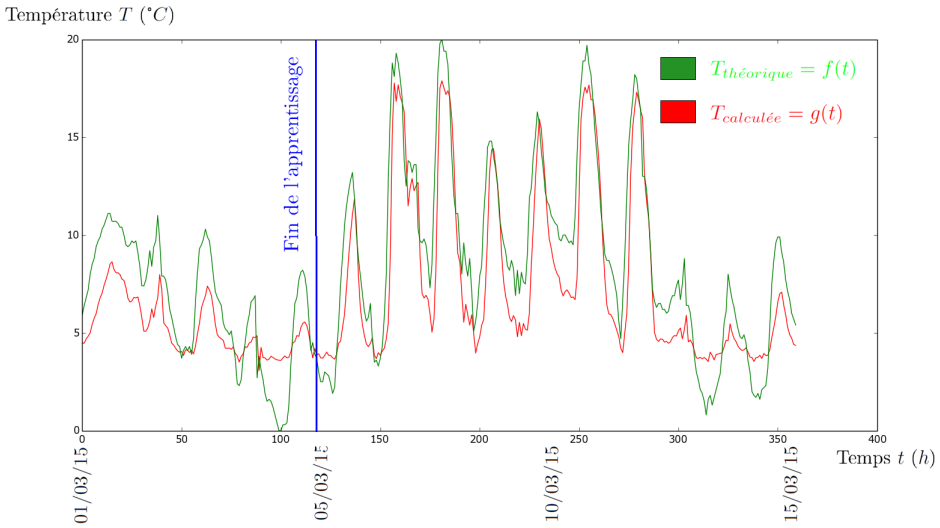
On constate empiriquement que le réseau qui offre le meilleur compromis entre temps de calcul et précision est constituée de trois couches cachées de 4 neurones. Ajouter davantage de couches et de neurones n'améliore pas le résultat, et rend l'apprentissage plus long.

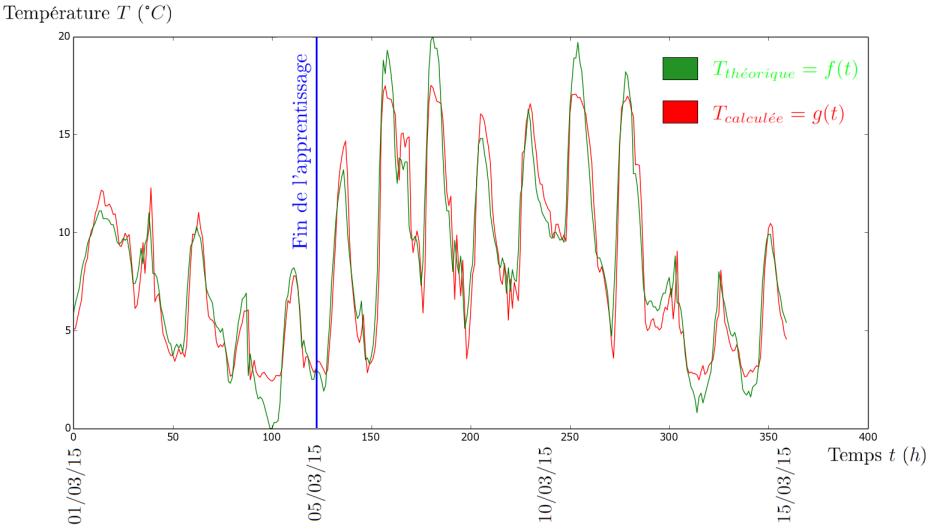
Nombre d'itération à l'entraînement

Les poids se stabilisent après environ 100 répétitions, ce qui correspond à environ 1min30 de calcul (processeur i3-3110M, 2,4GHz). Au delà on ne constate plus d'évolution notable. Le nombre d'opérations est déjà de l'ordre de 200 000.

Nombre d'antériorités

On peut penser que plus p est élevé, plus les résultats seront satisfaisants. On fait donc varier p entre 2 et 4, pour $\eta = 0,01$.





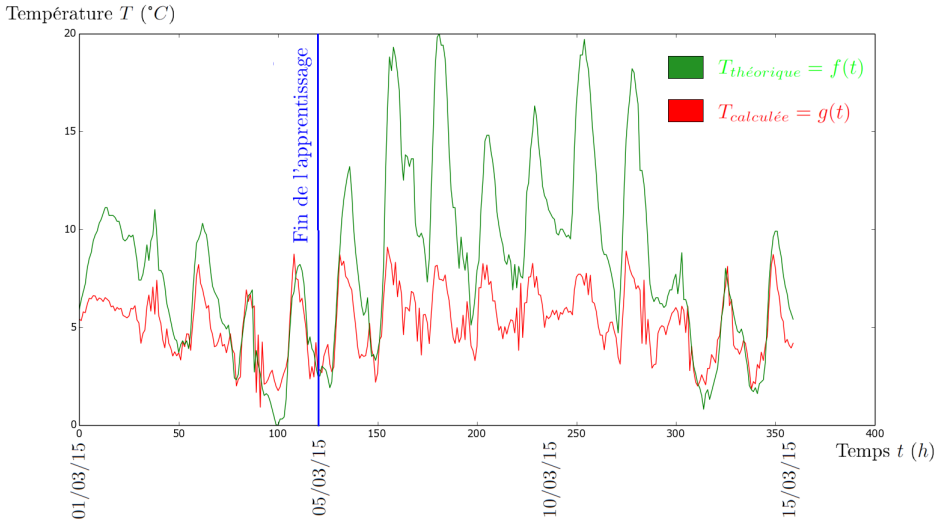


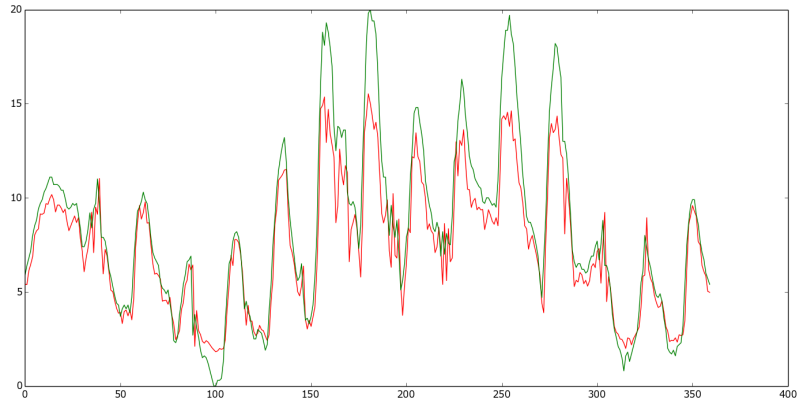
Fig. 7 : résultats pour $p = 2$, $p = 3$ et $p = 4$ ($\eta = 0,01$)

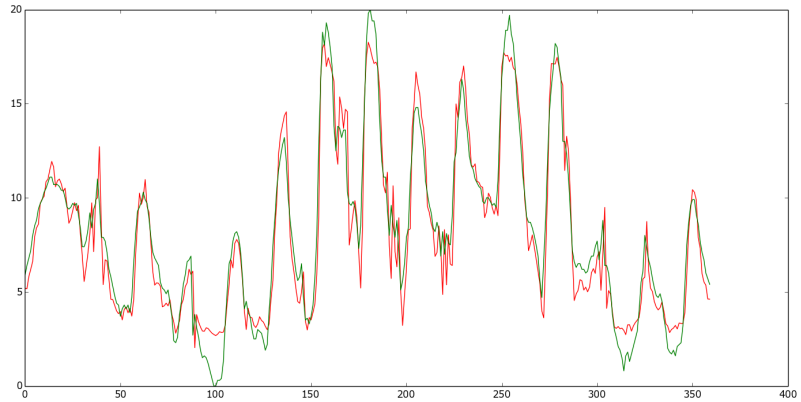
Il y a bien une amélioration entre 2 et 3 antériorités car davantage d'informations sont fournies en entrée. Pour 3 antériorités, l'erreur commise est faible (de l'ordre de 1°C), elle n'excède pas 3°C, même pour des entrées inédites. Cependant, pour 4 antériorités, le résultat n'est pas satisfaisant ; les températures trop éloignées constituent plus un bruit qu'une information utile.

Influence de η

Le choix de η est le paramètre qui influence le plus la convergence de l'algorithme. S'il est trop faible, l'algorithme converge trop lentement, s'il est trop élevé, les poids oscillent sans converger.

Les graphes suivants résultent des mêmes expériences avec $\eta = 0,1$.





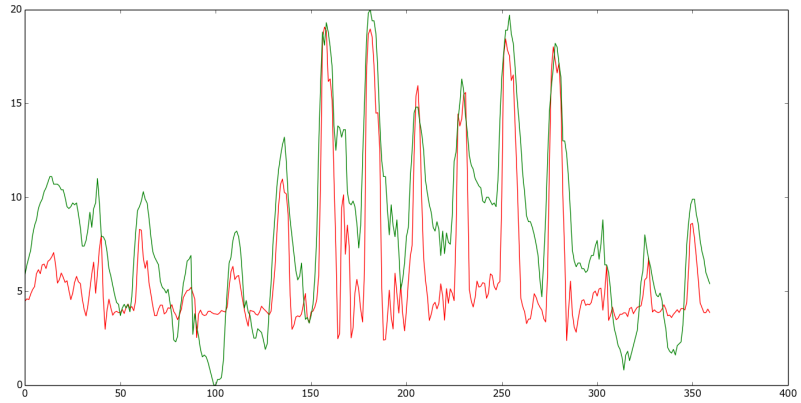


Fig. 8 : résultats pour $p = 2$, $p = 3$ et $p = 4$ ($\eta = 0,1$)

Il n'y a pas d'amélioration notable pour $p = 2$ ou 3. Pour $p = 4$, le résultat s'est amélioré mais n'est toujours pas satisfaisant.

Conclusion

Le choix qui s'impose pour cette étude est celui d'un réseau à trois couches cachées de 4 neurones, où l'on présente 3 antériorités de température, avec $0,01 \lesssim \eta \lesssim 0,1$ et 100 répétitions de l'apprentissage.

3.2.3 Une amélioration

Comme il existe une relative périodicité dans les températures, il peut être pertinent de présenter l'heure du relevé en entrée, en plus des autres paramètres.

Les résultats sont satisfaisants car on arrive à la même précision que précédemment avec seulement 1 couche cachée de 2 neurones et 50 itérations.

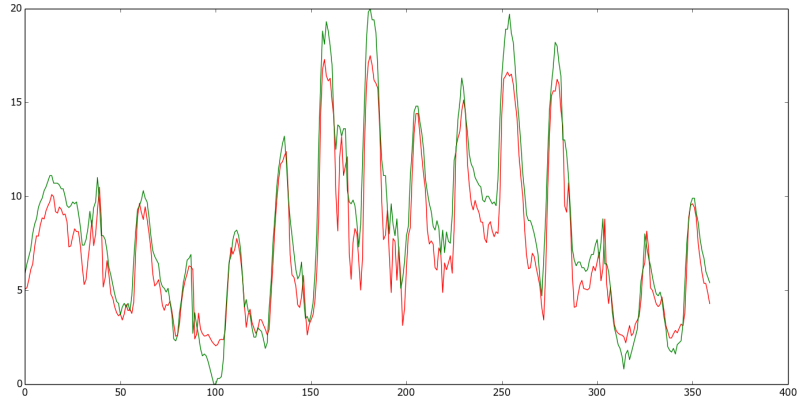


Fig. 9 : Ajout de l'heure de la journée en entrée ($\eta = 0,2$)

Conclusion

Le réseau de neurones mis en œuvre, appliqué à la prévision météorologique de températures s'est révélé performant et a prouvé l'efficacité des processus d'apprentissage. La problématique du temps de calcul liée à la fois au langage de développement et à la machine utilisée reste néanmoins centrale. L'implémentation réalisée peut être appliquée dans de nombreux contextes, sous réserve de disposer de données expérimentales suffisantes.