ROUSSILLON Cyril

Lycée Victor Hugo
BESANCON

TIPE 2003 – 2004
Dossier

PRÉVISION DE TEMPÉRATURE PAR RÉSEAUX NEURONAUX

INTRODUCTION

Les réseaux de neurones artificiels, ou réseaux neuromimétiques, sont des modèles inspirés du fonctionnement du cerveau animal, et dont le but est de voir surgir des propriétés analogues au système biologique.

Ils en reprennent quelques grands principes :

- le **parallélisme** : les neurones sont des entités réalisant une fonction très simple, mais ils sont très fortement interconnectés entre eux, ce qui rend le traitement du signal massivement parallèle.
- les **poids synaptiques** : les connexions entre les neurones ont des poids variables, qui déterminent la force de l'interaction entre chaque paire de neurones.
- l'**apprentissage** : ces coefficients synaptiques sont modifiables lors de l'apprentissage, dans le but de faire réaliser au réseau la fonction désirée.

La fonction que réalise un réseau dépend de sa structure (connexions et forces des connexions) ainsi que de l'opération effectuée par les neurones. L'essentiel de l'étude des réseaux neuronaux (le connexionnisme), consiste en la mise au point et l'optimisation de méthodes d'apprentissage.

Parmi leurs nombreuses applications, ils sont en particulier utilisés pour effectuer des prévisions, notamment météorologiques, et en reconnaissance de formes. J'ai choisi de me consacrer à ces applications car elles offrent des retombées dans le domaine de l'environnement, en permettant par exemple des économies d'énergie, de prévoir les pics d'ozone ou de détecter des nappes de pétrole dans l'océan.

Je commence dans ce dossier par une présentation générale des réseaux de neurones artificiels, ainsi que quelques notions d'optimisation. Je m'intéresse ensuite aux propriétés des réseaux multicouches et à leur apprentissage, pour finalement présenter leurs applications ainsi que mes réalisations pratiques.

PLAN DU DOSSIER

1. NOTIONS GÉNÉRALES

1.1. PRÉSENTATION DES RÉSEAUX DE NEURONES ARTIFICIELS

➢ **BRÈVE HISTORIQUE**

1943 : Mc Culloch et Pitts présentent le premier neurone formel

1949 : Hebb propose un mécanisme d'apprentissage

1958 : Rosenblatt présente le premier réseau de neurones artificiels : le Perceptron. Il est inspiré du système visuel, et possède deux couches de neurones : perceptive et décisionnelle.

Dans la même période, le modèle de l'ADALINE (ADAptive LINear Element) est présenté par Widrow Ce sera le modèle de base des réseaux multicouches.

1969 : Minsky et Papert publient une critique des perceptrons en montrant leurs limites, ce qui va faire diminuer la recherche sur le sujet.

1972 : Kohonen présente ses travaux sur les mémoires associatives

1982 : Hopfield démontre l'intérêt d'utiliser les réseaux récurrents pour la compréhension et la modélisation des fonctions de mémorisation.

1986 : Rumelhart popularise l'algorithme de rétropropagation du gradient, conçu par Werbos, qui permet d'entraîner les couches cachées des réseaux multicouches.

Les réseaux neuronaux ont été depuis été beaucoup étudiés, et ont trouvé énormément d'applications.

1.1.1. LE NEURONE ARTIFICIEL

➢ **LE NEURONE FORMEL DE MC CULLOCH ET PITTS**

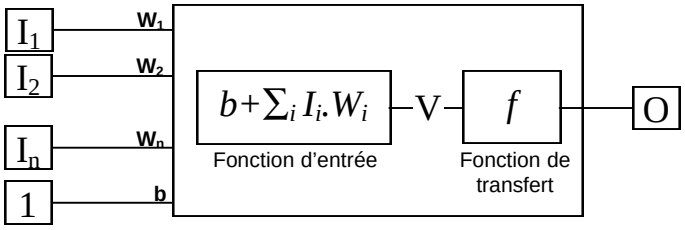
La notion de neurone formel a été pour la première fois avancée en 1943 par deux biophysiciens de Chicago, Mc Culloch et Pitts, dans l'article « A logical calculus of the ideas immanent in nervous activity » paru dans le journal « Bulletin of mathematical ».

Ils ont prouvé que pour des poids judicieusement choisis ce modèle offrait la puissance d'une machine de Turing universelle, voulant ainsi démontrer que le cerveau est équivalent à une machine de Turing. Ils déclarent ainsi en 1955 : « Plus nous apprenons de choses au sujet des organismes, plus nous sommes amenés à conclure qu'ils ne sont pas simplement analogues aux machines, mais qu'ils sont une machine ».

Je ne m'étendrai pas sur ce modèle, cas particulier du neurone artificiel général, qui présentait la particularité d'être binaire, c'est à dire qu'il était soit actif, soit non actif.

➢ **LE NEURONE ARTIFICIEL GÉNÉRAL**

Un neurone artificiel peut être représenté comme ceci :

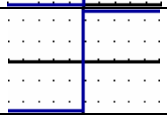
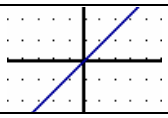
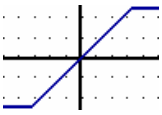
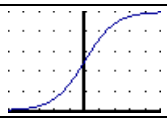
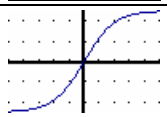


Chaque neurone possède plusieurs entrées (*In*), à chacune desquelles est affecté un poids (*Weight*). L'entrée fixée à 1 de poids *b* est facultative et représente le seuil (ou biais).

A chaque entrée peuvent être connectées plusieurs sorties d'autres neurones. La sortie est calculée à partir des entrées et des poids synaptiques :

- une **fonction d'entrée** évalue la stimulation reçue en calculant le potentiel V du neurone. Elle est très souvent la somme pondérée (par les poids synaptiques) des entrées, augmentée d'un seuil.
- une **fonction de transfert** (ou fonction d'activation) génère alors la sortie grâce à ce potentiel. Cette fonction de transfert est très importante, et détermine le fonctionnement du neurone et du réseau. Elle peut prendre de nombreuses formes comme nous allons le voir dans le paragraphe suivant.

1.1.2. LES FONCTIONS DE TRANSFERT

Catégorie	Type	Relation	Allure	Dérivées
Seuil	Binaire (fonction de Heaviside)	$f(x)=\begin{cases}0 \text{ si } x<0\\1 \text{ si } x\geq 0\end{cases}$		-
	Signe	$f(x)=\begin{cases}-1 \text{ si } x<0\\1 \text{ si } x\geq 0\end{cases}$		-
	Stochastique (binaire ou signe) *(T est appelé température)*	$f(x,T)=1$ avec la probabilité $\frac{1}{1+e^{-\frac{x}{T}}}$	-	-
Linéaire	Identité	$f(x,k) = k.x$		$f'(x,k) = k$
	Saturé positif	$f(x,k)=\begin{cases}0 \text{ si } x<0\\1 \text{ si } x\geq \frac{1}{k}\\k.x\end{cases}$		$f(x,k)=\begin{cases}0 \text{ si } x<0\\0 \text{ si } x\geq \frac{1}{k}\\k\end{cases}$
	Saturé symétrique	$f(x,k)=\begin{cases}-1 \text{ si } x<-\frac{1}{k}\\1 \text{ si } x\geq \frac{1}{k}\\k.x\end{cases}$		$f(x,k)=\begin{cases}0 \text{ si } x<-\frac{1}{k}\\0 \text{ si } x\geq \frac{1}{k}\\k\end{cases}$
Sigmoïde	Positive (type logistique)	$f(x,k)=\frac{1}{1+e^{-k.x}}$		$f'(x,k)=\frac{k}{2+e^{-k.x}+e^{k.x}}$
	Symétrique (type tanh)	$f(x,k)=\frac{2}{1+e^{-k.x}}-1$		$f'(x,k)=\frac{2.k}{2+e^{-k.x}+e^{k.x}}$

Plus généralement, une fonction est dite **sigmoïdale** si elle est infiniment dérivable, non linéaire, bornée, croissante et que sa dérivée admet un et un seul maximum global.
On peut également considérer qu'il s'agit d'une approximation infiniment dérivable de la fonction à seuil de Heaviside.

Il faut cependant être prudent avec les fonctions sigmoïdes, car pour des entrées grandes la fonction exponentielle peut provoquer un débordement (*overflow*), et il peut être nécessaire de vérifier que l'entrée n'est pas trop grande avant d'appliquer l'équation générale.

Il est bien sûr possible de discrétiser ces fonctions (afin de stocker des entiers). On peut également appliquer un coefficient additif à la fonction, et simuler un seuil comme vu plus haut avec une entrée supplémentaire fixe au réseau.

1.1.3. L'APPRENTISSAGE

On peut distinguer deux principaux types d'apprentissages :

- **supervisé** : on fournit au réseau le couple (entrée, sortie) et on modifie les poids en fonction de l'erreur entre la sortie désirée et la sortie obtenue.
- **non supervisé** : le réseau doit détecter des points communs aux exemples présentés, et modifier les poids afin de fournir la même sortie pour des entrées aux caractéristiques proches.

➢ LA RÈGLE DE HEBB

Méthode d'apprentissage la plus ancienne (1949), elle est inspirée de la biologie. Le principe est de renforcer les connexions entre deux neurones lorsque ceux-ci sont actifs simultanément. Cette règle peut être classée comme apprentissage non supervisé, ou supervisé car on sait calculer directement les poids correspondant à l'apprentissage d'un certains nombres d'exemples.

➢ LA RÈGLE DU DELTA (OU RÈGLE DE WIDROW-HOFF)

Son but est de faire évoluer le réseau vers le minimum de sa fonction d'erreur (erreur commise sur l'ensemble des exemples). Elle est utilisée dans le modèle de l'ADALINE (ADAptive LINear Element).

L'apprentissage est réalisé par itération (les poids sont modifiés après chaque exemple présenté), et on obtient le poids à l'instant $t+1$ par la formule :

$$W(t+1) = W(t) + h.(T - O).E$$

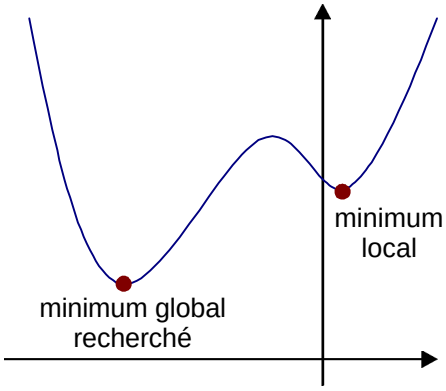
si W est le poids, T la sortie théorique et O la sortie réelle, E l'entrée et h un coefficient d'apprentissage (entre 0 et 1) que l'on peut diminuer au cours de l'apprentissage.

Cette règle s'interprète par le fait que l'on modifie le poids proportionnellement à l'erreur (différence entre sortie théorique et sortie donnée) et à l'état d'excitation de l'entrée gérée par le poids. Ainsi le poids ne sera pas modifié si la sortie est celle voulue, ou si le poids n'a joué aucun rôle dans le résultat.
C'est en fait un cas particulier de l'algorithme de rétropropagation du gradient pour un réseau à une couche.

➢ LA RÉTROPROPAGATION DU GRADIENT (*backpropagation*)

Il a fallu attendre le début des années 1980 pour qu'une règle efficace soit mise au point pour l'apprentissage des réseaux multicouches. Cette règle, découverte simultanément par des équipes française et américaine, est en fait une généralisation de la règle du delta.

Elle consiste simplement en une descente de gradient, qui est une méthode d'optimisation universelle. On cherche à minimiser une fonction d'énergie d'erreur (qui représente l'erreur entre la sortie désirée et la sortie obtenue), en suivant les lignes de plus grande pente. Une fonction d'erreur rapportée à une dimension peut se représenter ainsi :



On peut se représenter la descente de gradient comme une bille que l'on poserait sur la courbe, et qui descendrait logiquement la pente (le gradient représente la pente selon chaque dimension).

L'inconvénient de cette méthode est qu'elle va s'arrêter dans le premier minimum local rencontré. C'est pourquoi diverses améliorations ont été apportées, détaillées dans la partie 2.

1.1.4. LES PRINCIPAUX TYPES DE RÉSEAUX

➢ **MÉMOIRES AUTO-ASSOCIATIVES**

Dans la désignation mémoire associative, le terme mémoire fait référence à la fonction de stockage de ces réseaux, et le terme associative au mode d'adressage, puisque qu'il faut fournir de l'information au réseau pour obtenir celle qui est mémorisée : c'est une mémoire adressable par son contenu.

Avec les mémoires auto-associatives, il faut fournir une partie de l'information stockée pour obtenir l'information stockée (par exemple : partie du visage pour obtenir le visage en entier).

Structure

Chaque neurone est relié à tous les autres, et à toutes les entrées. La fonction de transfert est habituellement l'identité, et l'évaluation peut se faire de manière synchrone (toutes les unités évoluent en même temps) ou asynchrone (les éléments du réseaux évoluent les uns après les autres).

Apprentissage

Il est de type supervisé, c'est-à-dire que la base d'apprentissage est constituée de couples de vecteurs entrée et sortie associés, et est basé sur la règle de Hebb.

Applications

Les applications des ces mémoires sont essentiellement la reconstruction de signaux et leur reconnaissance.

Exemple : Réseau de Hopfield

Le modèle de Hopfield est une mémoire auto-associative dont la réponse des neurones est asynchrone, et dont les neurones ont pour fonction de transfert la fonction signe, c'est-à-dire dont la réponse est binaire. Les fondements mathématiques de ces réseaux sont très bien compris, et il est possible réaliser l'apprentissage par un calcul direct à partir des exemples à mémoriser grâce à la règle de Hebb généralisée.

➢ **MÉMOIRES HÉTÉRO-ASSOCIATIVES ET CARTES TOPOLOGIQUES DE KOHONEN**

Avec les mémoires hétéro-associatives, on fournit une information au réseau et celui-ci rend une information différente (par exemple : visage donne le nom).

Structure

Chaque neurone est relié à toutes les entrées, mais les neurones ne sont pas reliés entre eux. La fonction de transfert est linéaire.

Apprentissage

Diverses méthodes peuvent être utilisées, comme la règle de Hebb ou la règle du delta.

Applications

Les mémoires à apprentissage supervisé sont principalement utilisées pour la mémorisation et la reconnaissance de formes (associer le nom d'une lettre à l'image de la lettre par exemple). Les réseaux à apprentissage non supervisés sont en revanche utilisés pour des fonctions de classification et d'agrégation (*clustering*).

Exemple : les cartes auto-organisatrices de Kohonen

Kohonen s'est inspiré de la topologie de certaines zones du cortex qui présentent la même organisation que les capteurs sensoriels : des zones proches correspondent à des neurones proches dans le cortex.
Tous les neurones sont interconnectés, mais seuls les neurones proches ont de l'influence, selon une fonction DOG (Difference Of Gaussian) en forme de "chapeau mexicain" : les neurones les plus proches sont excitateurs, ceux un peu plus éloignés inhibiteur et les plus éloignés ont une influence nulle.
La fonction d'entrée est de type sigmoïde, et le réseau présente la particularité de n'avoir qu'un seul neurone actif à la fois. Il réalise une tâche de classification : des entrées aux caractéristiques proches donnent une même sortie.

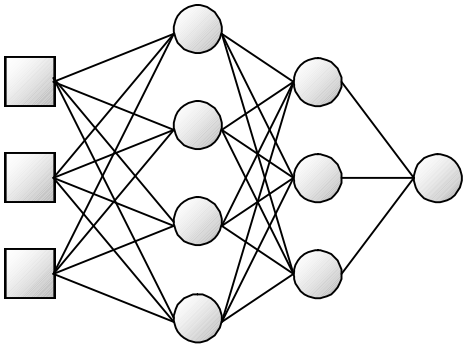
➢ LES RÉSEAUX MULTICOUCHES (NON BOUCLÉS)

Les MLP (*multi-layer perceptron*), ou réseaux à couches, forment la très grande majorité des réseaux. Ils sont intemporels (ce sont des réseaux statiques et non dynamiques).

Structure

Les neurones sont organisés en couches : chaque neurone est connecté à toutes les sorties des neurones de la couche précédente, et nourrit de sa sortie tous les neurones de la couche suivante (ces réseaux sont d'ailleurs qualifiés de *feedforward* en anglais : "nourrit devant"). Pour la première couche ses entrées sont l'entrée du réseau. D'ailleurs une couche est souvent rajoutée pour constituer les entrées, appelée couche d'entrée, mais elle n'en est pas une puisqu'elle ne réalise aucun traitement.
Les fonctions d'entrée et de transfert sont les mêmes pour les neurones d'une même couche, mais peuvent différer selon la couche. Ainsi la fonction de transfert de la couche de sortie est généralement l'identité.

Un réseau à trois couches (deux couches cachées et une de sortie) et à trois entrées peut donc se représenter ainsi :



Apprentissage

Les réseaux multicouches utilisent la règle de rétropropagation du gradient décrite plus haut. Les fonctions de transferts doivent donc être différentiables, c'est pourquoi on utilise des fonctions sigmoïdes, qui sont des approximations infiniment dérivables de la fonction à seuil de Heaviside.

Applications

Les applications des réseaux non bouclés sont très diverses et étendues. Elles vont de la reconnaissance de motifs, à la prévision en passant par l'apprentissage de comportements ou de jeux.

Cas particulier : Les réseaux à fonctions radiales de base

Ces réseaux, appelés réseaux RBF (Radial Basis Functions) sont des réseaux multicouches, à une couche couchée. Cependant, contrairement aux perceptrons multicouches, les fonctions de transfert de la couche cachée dépendent de la distance entre le vecteur d'entrée et un vecteur centre. Dans la pratique cette fonction de transfert est souvent le noyau gaussien :

$s = \exp\left(-\left(\frac{\|e-c\|}{s} \right)^2 \right)$ où s est le facteur de largeur et c le vecteur centre.

La fonction d'activation de la couche de sortie est quant à elle une fonction linéaire.

Les trois types de paramètres - centres, largeurs, poids - sont optimisés indépendamment afin d'éviter une lourdeur excessive des calculs.

CONCLUSION

Les réseaux de neurones présentent donc une très grande diversité. En effet un type de réseau neuronal est défini par sa topologie, sa structure interne, son algorithme d'apprentissage, et beaucoup de combinaisons sont viables.
Je me restreindrai par la suite aux réseaux non bouclés (ou multicouches), car ils forment le modèle qui a le plus d'applications, et dont les techniques d'apprentissage sont les mieux connues.

1.2. NOTIONS D'OPTIMISATION

Nous verrons que l'apprentissage supervisé des réseaux multicouches est une simple question d'optimisation, c'est-à-dire consiste en la minimisation d'un critère d'erreur.

Il existe divers types d'optimisation. Elle peut être différentiable ou non, opérant sur des données linéaires ou non. Dans le cas des réseaux de neurones nous nous intéresserons à l'optimisation différentiable non linéaire.

1.2.1. OPTIMISATION CONTRAINTE ET NON CONTRAINTE

Le but est de trouver un point de l'espace des variables en lequel le critère d'erreur est minimal, c'est-à-dire un minimum global du critère d'erreur.

Définition : Le point X_0 de l'espace E est un MINIMUM GLOBAL de la fonction f définie sur E si et seulement si, pour tout X de E on a : $f(X_0) \leq f(X)$.

Définition : Le point X_0 de l'espace E est un MINIMUM LOCAL de la fonction f définie sur E si et seulement si il existe un voisinage V de X_0 dans E tel que, pour tout X de V on a : $f(X_0) \leq f(X)$.

Les minima locaux sont faciles à reconnaître : une condition nécessaire est que le gradient s'annule en ce point, une condition nécessaire et suffisante est que le gradient s'annule et que la matrice hessienne des dérivées secondes $\left(\frac{\partial^2 f}{\partial x_i \partial x_j} \right)_{i,j}$ soit définie positive.

En revanche les minima globaux sont beaucoup plus difficiles à déterminer. En effet mis à part quelques cas particuliers (fonctions convexes …) où la solution est facilement accessible, dans le cas général de fonctions non linéaires une exploration exhaustive de la fonction est nécessaire pour construire l'enveloppe convexe, ce qui impensable en dimension quelconque.

➢ OPTIMISATION CONTRAINTE

Des contraintes peuvent être appliquées, consistant à limiter l'espace des variables d'entrée.
Dans le cas des réseaux de neurones, ces contraintes peuvent être dues à l'injection de connaissances du problème à traiter, ou à des possibilités de réalisation physique. Elles peuvent également être impératives, on parle alors de contraintes *strictes*, ou seulement des préférences, on les appelle contraintes *relaxables*.

On se ramène en fait souvent à un problème d'optimisation non contrainte en ajoutant au critère d'erreur une fonction pénalité qui pénalise les domaines ne rentrant pas dans les contraintes. Le choix de cette fonction est délicat car trop forte elle empêchera le réseau de passer par certains domaines pour y découvrir des zones de minima, et pas assez forte elle ne l'empêchera pas de se stabiliser dans un domaine ne satisfaisant pas les contraintes.

1.2.2. MÉTHODES ITÉRATIVES D'OPTIMISATION

Devant la difficulté d'obtenir une solution analytique lorsque les données sont non linéaires, l'émergence de méthodes itératives numériques basées sur une approximation du critère d'erreur (son développement à plusieurs variables de Taylor) a été favorisée.

On veut donc minimiser à chaque itération la valeur $f(x_t)$. Cela est le cas si l'on a à chaque itération :

$$f(x_{t+1}) < f(x_t)$$

Si la variation $\Delta x_t = x_{t+1} - x_t$ est suffisamment faible, on peut approcher la valeur $f(x_{t+1})$ par le développement de Taylor à plusieurs variables en x_t :

$$f(x_{t+1}) = f(x_t + \Delta x_t) \approx f(x_t) + \left(\nabla f \big|_{x_t} \right)^{\mathrm{T}} . \Delta x_t$$

où $\nabla f \big|_{x_t}$ est le gradient de f au point x_t.

Notre objectif est donc d'avoir le produit scalaire entre le vecteur gradient et le vecteur des variations de x négatif :

$$\left(\nabla f\big|_{x_t}\right)^{\mathrm{T}}.\Delta x_t < 0$$

On a une décroissance maximale de la fonction f si ce produit scalaire est négatif et minimal, donc si et seulement si les deux vecteurs sont de même direction et de sens opposé.

On obtient pour règle d'apprentissage :

$x_{t+1} = x_t - h.\nabla f\big|_{x_t}$ où h est un coefficient compris entre 0 et 1

On constate donc qu'il faut suivre le gradient, d'où le terme de descente de gradient.

CONCLUSION

L'apprentissage d'un réseau de neurones artificiels se réduisant à un problème d'optimisation : trouver le minimum d'une fonction d'erreur, on pourra donc mettre à profit cette méthode d'optimisation universelle de descente de gradient, qui constituera la règle de rétropropagation du gradient pour les réseaux multicouches, étudiée dans la seconde partie.

2. APPRENTISSAGE SUPERVISÉ DES RÉSEAUX NON BOUCLÉS

2.1. L'APPROXIMATION

Lors d'un apprentissage supervisé, on veut faire correspondre à chaque vecteur d'entrée, un vecteur de sortie. En d'autres termes le réseau doit réaliser une fonction vectorielle voulue. Si cette fonction est connue analytiquement, il s'agit d'une tâche d'approximation de fonction. Si on ne dispose que d'un certain nombre de mesures (souvent entachées de bruit), il s'agit d'une tâche de régression.

Ainsi **un réseau en apprentissage supervisé réalise une tâche d'approximation**.

Je vais donc réaliser toutes les expériences d'apprentissage sur de l'approximation à une variable, puisqu'elle présente un intérêt pédagogique particulier permettant de visualiser immédiatement à chaque étape l'efficience et la vitesse de l'apprentissage.

2.1.1. APPROXIMATEURS UNIVERSELS

Les approximateurs universels les plus connus sont les polynômes.
En effet n'importe quelle fonction suffisamment continue de plusieurs variables peut être approchée avec une précision arbitraire (sur un domaine fini de l'espace de ses variables) par une fonction polynôme.
Cette fonction polynôme peut être déterminée par le développement limité si la fonction est connue, ou par une régression polynomiale si la fonction est inconnue. On peut également citer toutes les formules d'approximation polynomiale telles les polynômes d'interpolation de Lagrange, qui permet de déterminer le polynôme de degré n passant par n+1 points, ou les polynômes de Tchebychev.

Les réseaux de neurones sont également des approximateurs universels :

Théorème [Dreyfus]
> Toute fonction bornée suffisamment régulière peut être approchée uniformément, avec une précision arbitraire, dans un domaine fini de l'espace de ses variables, par un réseau de neurones comportant une couche de neurones cachés en nombre fini, possédant toutes la même fonction d'activation, et un neurone de sortie linéaire.

Ceci n'est qu'un théorème d'existence, et ne donne aucune indication sur la manière d'obtenir les poids adéquats, ni sur le nombre de neurones dans la couche cachée. De plus une couche cachée est un minimum, mais en pratique des réseaux avec plusieurs couches cachées peuvent être plus performants.

La démonstration de ce théorème, connu sous le nom de théorème de Kolmogorov, a été donnée par de nombreux auteurs (Cybenko, Funahashi, Hornik, Stinchcombe, White et Leshno). Mais cette existence est purement théorique : J. Wray a montré que ces preuves d'existence ne sont pas justifiées si l'on considère les limitations imposées par une simulation numérique sur un ordinateur.

2.1.2. APPROXIMATEURS PARCIMONIEUX

L'avantage de certains réseaux de neurones réside dans le fait qu'ils réalisent des approximations plus parcimonieuses que les polynômes par exemple, c'est-à-dire que le nombre de paramètres ajustables est plus faible pour une même précision. Ceci résulte du théorème suivant :

Théorème [Dreyfus]
> Le nombre de paramètres, pour une précision donnée, croît exponentiellement avec le nombre de variables dans le cas des approximateurs linéaires par rapport à leurs paramètres, alors qu'il croît linéairement avec ce nombre dans le cas des approximateurs non linéaires par rapport à leurs paramètres.

➢ **ETUDE DES POLYNÔMES**

Un polynôme à deux variables x et y de degré deux s'écrit :

$$P(x,y) = a_{00} + a_{10}.x + a_{01}.y + a_{11}.x.y + a_{20}.x^2 + a_{02}.y^2 + a_{21}.x^2.y + a_{12}.x.y^2 + a_{22}.x^2.y^2$$

Plus généralement, un polynôme à m variables de degré n s'écrit formellement :

$$P(x,a) = \sum_{0 \le i_1, i_2, \ldots, i_m \le n} \left[a_{i_1, i_2, \ldots, i_m} . \prod_{j=1}^{m} x_j^{\,i_j} \right]$$

où x représente le vecteur variables et a le vecteur paramètres.

On constate ainsi d'une part la linéarité du polynôme par rapport à ses paramètres :
$P(x, l.a+b) = l.P(x,a) + P(x,b)$

D'autre part on vérifie que le nombre de coefficients (paramètres) est $(n+1)^m$ donc qu'en ajoutant à précision égale (degré égal) une variable on le multiplie par n+1 : il s'agit bien d'une progression exponentielle.

- **ETUDE DES RÉSEAUX DE NEURONES**

En revanche contrairement aux polynômes, si la fonction d'activation n'est pas linéaire, alors le réseau de neurones n'est pas linéaire par rapport à ses paramètres. En effet la sortie est le résultat de compositions des fonctions d'activations, et la composée de deux fonctions non linéaires est, sauf cas exceptionnel voulu (fonctions réciproques), non linéaire.

- **CONSÉQUENCES**

La différence est ainsi d'autant plus importante que le nombre de paramètres croît. Ainsi pour une ou deux variables il est indifférent d'utiliser l'un ou l'autre, mais l'avantage des réseaux de neurones devient net lorsque le nombre de variables augmente.

En effet on imagine aisément que pour obtenir un bon ajustement des paramètres, le nombre d'exemples doit être grand devant le nombre de paramètres ajustables. Ainsi l'avantage d'une parcimonie accrue est que l'approximation est également parcimonieuse en exemples, dont la collecte peut être longue et/ou coûteuse.

CONCLUSION

On retiendra donc que les réseaux de neurones artificiels permettent une approximation d'une meilleure qualité pour un même nombre d'exemples que les fonctions polynomiales par exemple, et que cet avantage n'est réel que pour un grand nombre de variables d'entrée, ce qui est généralement le cas dans l'utilisation des réseaux de neurones.

2.2. L'APPRENTISSAGE : UN PROBLÈME D'OPTIMISATION

Notations utilisées dans la modélisation du réseau multicouches

- n : nombre de couches du réseau (la pseudo couche d'entrée n'est pas comptée)
- $(m_i)_{0\le i\le n}$: nombre de neurones de la couche i

  $q = m_0$: nombre d'entrées du réseau (dimension du vecteur d'entrée $\vec{I}$)

  $r = m_n$: nombre de sorties du réseau (dimension du vecteur de sortie $\vec{O}$)
- $(S_{ij})_{\substack{0\le i\le n \\ 1\le j\le n_i}}$: sortie du neurone j de la couche i (et entrée j des neurones de la couche i+1)

  $(I_j)_{1\le j\le q} = (S_{0,j})_{1\le j\le q}$: entrées (*in*) du réseau (coordonnées du vecteur d'entrée $\vec{I}$)

  $(O_j)_{1\le j\le r} = (S_{n,j})_{1\le j\le r}$: sorties (*out*) du réseau (coordonnées du vecteur de sortie $\vec{O}$)
- $(W_{ijk})_{\substack{1\le i\le n \\ 1\le j\le m_i \\ 1\le k\le m_{i-1}}}$: poids (*weight*), dans le neurone j de la couche i, de la sortie du neurone k de la couche i-1
- $(V_{ij})_{\substack{1\le i\le n \\ 1\le j\le n_i}}$: potentiel du neurone j de la couche i (résultat de la fonction d'entrée)
- $(f_i)_{1\le i\le n}$: fonction de transfert de la couche i
- p : nombre d'exemples de la base

  $(E_{k,j})_{\substack{0\le k<p \\ 1\le j\le q}}$: exemples de la base (entrées du réseau)

  $(T_{k,j})_{\substack{0\le k<p \\ 1\le j\le r}}$: sorties théoriques du réseau pour l'exemple k

On notera également W le vecteur poids du réseau, dont les coordonnées sont l'ensemble des poids W_{ijk} du réseau. Ces notations sont également utilisées dans les structures et algorithmes, dont le code se trouve en annexe B.

2.2.1. LA FONCTION D'ERREUR

➢ CONTEXTE

On souhaite modéliser une fonction inconnue $y = f(x)$, et l'on dispose seulement de p échantillons : p observations $\{y_1, y_2, \ldots, y_p\}$ de y correspondant à p observations $\{x_1, x_2, \ldots, x_p\}$ de x, les x_i et y_i étant entachés de **bruit**, dû à une imprécision de mesure ou à des parasites.

On peut donc définir la relation entre les x_i et y_i par $y = f(x) + e$ où e est un vecteur d'erreur aléatoire.

On supposera de plus que l'erreur moyenne pour une valeur x est nulle (il n'y a pas d'*erreur systématique*, simplement une *erreur accidentelle*) :

$$\mathrm{E}[e \mid x] = 0$$

et que e et $f(x)$ sont non corrélés :

$$\mathrm{E}[e.f(x)] = 0$$

où $\mathrm{E}[.]$ désigne l'espérance mathématique et $\mathrm{E}[.|.]$ l'espérance conditionnelle

Il est ainsi possible d'estimer la relation f par $f(x) = \mathrm{E}[y \mid x]$, ce qui est raisonnable pour pouvoir espérer modéliser f. En effet si un décalage constant intervient par exemple sur les mesures, ce ne serait alors plus f qui serait modélisée.

➢ Erreur instantanée

En fournissant alors au réseau une entrée x, celui-ci va fournir une sortie $O = N(x,w)$ où w est le vecteur poids du réseau. Il faut alors estimer l'**erreur instantanée** qui représente le coût encouru à confondre la sortie du réseau à la réponse désirée y. Celui-ci peut l'être par n'importe quelle distance entre ces deux vecteurs. La distance euclidienne quadratique (correspondant à la fonction de coût des moindres carrées) est usuellement utilisée :

$$C(x,w) = \|y - O\|^2$$

ou encore avec les notations générales :

$$C_k = C(E_k) = \sum_{j=1}^{r} (T_{x,j} - O_{x,j})^2$$

Son intérêt, qui fait qu'elle est largement employée dans tous les domaines, est que les calculs restent simples, qu'elle est infiniment dérivable et que dans le cas d'une distribution normale des erreurs (99% des cas), la minimisation du coût des moindres carrées est celle qui donne le maximum de vraisemblance.

➢ Erreur moyenne

L'**erreur moyenne** du réseau dans la modélisation de f est alors l'espérance du coût instantané C_k : c'est la moyenne de C_k pour toutes les valeurs de x et y, pondérée par leur probabilité de rencontre. Il faut donc en théorie utiliser la fonction de distribution cumulative de probabilité de x et y considérées comme variables aléatoires :

$F_X(x) = \Pr(X \leq x)$ représente la probabilité pour qu'une réalisation de la variable aléatoire X soit inférieure à x.

En pratique on ne dispose que de nos p échantillons, et on ne connaît pas la fonction de distribution cumulative de probabilité. On est aussi réduit à utiliser une approximation empirique de ce critère d'erreur moyenne, qui est simplement la moyenne du coût sur les p échantillons : $C = \frac{1}{p} . \sum_{k=1}^{p} C_k$.

2.2.2. L'algorithme de rétropropagation du gradient

On a présenté dans le paragraphe 1.1 par une approche intuitive l'algorithme de rétropropagation du gradient, et on a démontré dans le paragraphe 1.2 grâce à un développement de Taylor qu'il fallait suivre le gradient. Il reste donc à déterminer ce gradient.

➢ Calcul

On cherche donc, pour chaque poids (i, j et k fixés), à exprimer la pente de la courbe représentative de chaque application partielle associée à la fonction coût $C(W)$ en W (i.e. la dérivée partielle de C par rapport à chacun des poids), ce en fonction de **paramètres connus** du réseau (les poids, les potentiels, les fonctions de transfert …)

Puisque ce coût total est la somme des coûts instantanés : $C = \sum_{x=0}^{N-1} C_x$, et que ces derniers sont positifs (c'est d'ailleurs pourquoi on préfère le terme coût au terme erreur), le coût total est minimal lorsque chacun des coûts instantanés est minimal. Il faut donc minimiser chaque coût instantané C_x.

Ceci n'est bien entendu rigoureusement valable qu'à condition de ne pas modifier les poids entre deux exemples.

Hypothèses

→ Fonction d'entrée (produit scalaire entre le vecteur S_{i-1} et le vecteur W_{ij}) :

(1) $V_{ij} = \sum_{k=1}^{m_{i-1}} W_{ijk} . S_{i-1,k}$

→ Fonctions de transfert f_i quelconques mais dérivables :

(2) $S_{ij} = f_i(V_{ij})$

→ Fonction de coût instantané $C_x(W)$ pour chaque exemple x (fonction de coût des moindres carrées) :

(3) $C_x = \frac{1}{2} . \sum_{j=1}^{r} (T_{x,j} - O_{x,j})^2$

→ On pose de plus pour chaque neurone :

(4) $d_{ij} = -\frac{\partial C_x}{\partial V_{ij}}$

Démonstration

$\frac{\partial C_x}{\partial W_{ijk}} = \frac{\partial C_x}{\partial V_{ij}} . \frac{\partial V_{ij}}{\partial W_{ijk}} = -d_{ij} . \frac{\partial V_{ij}}{\partial W_{ijk}}$ (dérivation des fonctions composées)

$\frac{\partial V_{ij}}{\partial W_{ijk}} = S_{i-1,k}$ d'après (1)

$d_{ij} = -\frac{\partial C_x}{\partial V_{ij}} = -\frac{\partial C_x}{\partial S_{ij}} . \frac{\partial S_{ij}}{\partial V_{ij}}$

$\frac{\partial S_{ij}}{\partial V_{ij}} = f_i'(V_{ij})$ d'après (2)

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

$\frac{\partial C_x}{\partial S_{nj}} = \frac{\partial C_x}{\partial O_j} = -(T_j - O_j)$ d'après (3)

$d_{nj} = f_n'(V_{nj}) . (T_j - O_j)$

- Pour une couche autre que celle de sortie ($i < n$) :

$\frac{\partial C_x}{\partial S_{ij}} = \sum_{l=1}^{m_{i+1}} \frac{\partial C_x}{\partial V_{i+1,l}} . \frac{\partial V_{i+1,l}}{\partial S_{ij}}$ (puisque les $V_{i+1,l}$ sont fonctions de S_{ij})

$\frac{\partial V_{i+1,l}}{\partial S_{ij}} = \frac{\partial}{\partial S_{ij}} \left(\sum_{k=1}^{m_i} W_{i+1,l,k} . S_{ik} \right) = W_{i+1,l,j}$ d'après (1)

$\frac{\partial C_x}{\partial V_{i+1,l}} = -d_{i+1,l}$ d'après (4)

$\frac{\partial C_x}{\partial S_{ij}} = -\sum_{l=1}^{m_{i+1}} d_{i+1,l} . W_{i+1,l,j}$

$d_{ij} = f_i'(V_{ij}) . \sum_{l=1}^{m_{i+1}} d_{i+1,l} . W_{i+1,l,j}$

$\frac{\partial C_x}{\partial W_{ijk}} = -S_{i-1,k} . d_{ij}$

➢ **RÉSULTAT**

Or on a vu en 1.2 que la règle d'apprentissage était :

$$W_{ijk}(t+1) = W_{ijk}(t) - h.\frac{\partial C_x}{\partial W_{ijk}}$$

Ce qui donne donc :

$$W_{ijk}(t+1) = W_{ijk}(t) + \varsigma.S_{i\text{-}1,k}.\ddot{a}_{ij} \quad \text{avec} \begin{cases} d_{nj} = f_n'(V_{nj}).(T_j - O_j) \\ d_{ij} = f_i'(V_{ij}).\sum_{l=1}^{m_{i+1}} d_{i+1,l}.W_{i+1,l,j} \quad \text{si } i < n \end{cases}$$

Le coefficient h est le *coefficient d'apprentissage* ou *d'adaptation*, qui permet d'ajuster les modifications et qui doit être soigneusement choisi. En effet plus il est élevé et plus l'apprentissage est rapide, mais au détriment de la stabilité. En général sa valeur est fixée entre 0 et 1, et une valeur entre 0.3 et 0.7 représente un bon compromis.

2.2.3. LOCALITÉ DE L'OPTIMISATION ET AMÉLIORATIONS

Le problème de cette optimisation est qu'elle reste locale, c'est-à-dire que si le réseau se trouve dans un minimum local de la fonction d'erreur, il y restera.

Diverses solutions permettent de sortir de minima locaux, visant à modifier l'allure locale de l'hyper surface d'erreur.

➢ **LA DESCENTE STOCHASTIQUE**

Comme dans la règle de Widrow-Hoff, on minimise itérativement l'erreur due à chaque exemple. Les apprentissages de chaque exemple s'influençant les uns les autres, cela permet de passer sur des petits minima locaux.
Une présentation aléatoire des exemples donne généralement de meilleurs résultats.

➢ **LA DESCENTE AVEC INERTIE**

On introduit ici un moment d'inertie a (terme de momentum), qui correspond dans notre image de la bille, à l'inertie qu'elle acquiert en descendant la courbe : son élan lui permettra de ne pas s'arrêter dans le premier minimum local rencontré.

La règle d'apprentissage devient alors : $\Delta W_{ijk}(t) = W_{ijk}(t+1) - W_{ijk}(t) = \varsigma.S_{i\text{-}1,k}.\ddot{a}_{ij} + a.\Delta W_{ijk}(t-1)$

On peut également calculer dynamiquement les coefficients grâce à la règle suivante :

- Calculer un coefficient de corrélation : $\sum_{i,j,k} \Delta W_{ijk}(t).\Delta W_{ijk}(t-1)$
- S'il est négatif (ce qui signifie que les signes sont majoritairement différents, donc qu'il y a oscillation), on pose $h = 0.01$ et $a = 0$
- S'il est positif, on pose $h = h + d_n$ et $a = a + 0.01$

Il peut être nécessaire de borner ces coefficients, et on peut également décider de ne les augmenter que si le coefficient de corrélation est supérieur d'au moins 5% à sa précédente valeur.

➢ **LE GRADIENT CONJUGUÉ, LA QUICKPROP, LA RPROP**

Ce sont d'autres améliorations plus complexes de cet algorithme, qui peuvent combiner plusieurs corrections, et que je n'ai pas étudiées par manque de temps.

2.2.4. VITESSE DE CONVERGENCE ET AMÉLIORATIONS : MÉTHODES DU SECOND ORDRE

La convergence peut se révéler très lente à proximité des minima si la pente est faible. Des méthodes dites du second ordre, utilisant une approximation quadratique (au second ordre) du critère d'erreur permettent d'approcher beaucoup plus vite le minima grâce à la courbure quadratique de la fonction d'erreur au voisinage immédiat des minima (la convergence s'effectue en une seule itération si la courbure est idéalement quadratique).

La méthode de base est connue sous la dénomination d'optimisation de Newton, mais en pratique ne se fiant pas à la pente, cette méthode est sujette à converger vers des points non minimaux où le gradient s'annule, comme les points selles ou les points d'inflexion. C'est pourquoi la méthode de Levenberg-Macquart par exemple combine une optimisation du premier ordre à une optimisation du second ordre au voisinage des minima.

Cependant une optimisation du second ordre nécessite la matrice hessienne inverse des dérivées partielles du second ordre, dont le calcul et le stockage peut se révéler handicapant pour des réseaux de grande taille. Des méthodes, dites quasi-Newton, utilisent une estimation itérative de la matrice hessienne, réduisant le coût opératoire.

CONCLUSION

Les méthodes d'optimisation du premier et second ordre ont fait les preuves de leur efficacité, même si elles ne peuvent garantir de ne pas converger vers un minimum local. Les diverses corrections présentées se font au prix d'un coût opératoire, et tout l'apprentissage est un compromis entre rapidité et qualité de l'apprentissage.

3. APPLICATIONS ET RÉALISATIONS

Les applications des réseaux de neurones sont multiples, et en plus de pouvoir en utiliser à des fins de prévention, le caractère fondamental d'optimisation des réseaux neuronaux permet de réaliser des économies d'énergies.

➢ **APPLICATIONS**

Voici un bref aperçu des applications possibles des réseaux neuronaux :

- **Reconnaissance de formes :** traitement des chèques, signatures, codes postaux …
- **Modélisation et prévisions :** prévisions météorologiques, financières …
- **Traitement du signal :** compression de données, analyse du signal pour les télécommunications, reconnaissance et synthèse de la parole …
- **Classification et analyse discriminante :** informatique décisionnelle …
- **Robotique :** auto-apprentissage, évolution, vie artificielle, robotique autonome …
- **Secteur industriel :** contrôle de qualité, commande de processus, diagnostic de pannes
- **Problèmes NP Complexes :** problème du voyageur de commerce …
- **Neurologie :** étude du cerveau …

➢ **RÉALISATIONS**

J'ai donc réalisé en particulier un programme de reconnaissance de caractères, ainsi qu'un programme de prévision des températures, que je présente maintenant.

J'ai programmé les différents logiciels en C++. J'ai choisis ce langage d'une part parce que je le connais bien, et d'autre part parce qu'il permet une interface graphique plus pratique et conviviale que CAML, puisque j'utilise le logiciel Borland C++Builder, analogue à Delphi ou VisualBasic.

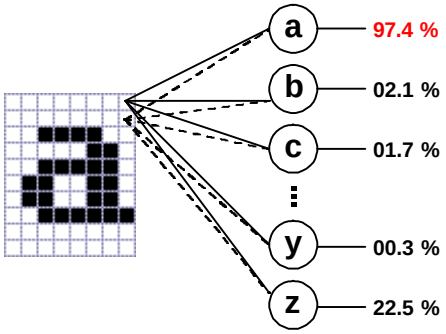
Ceci s'est révélé indispensable pour réaliser de l'acquisition analogique par exemple, et très utile pour permettre assez facilement un paramétrage complet des réseaux.

3.1. RECONNAISSANCE DE FORMES : PREMIER PROGRAMME

Ma toute première application pratique a été la reconnaissance de forme. Celle-ci est très utilisée dans différents domaines, de la reconnaissance de caractères dans des logiciels d'OCR (Optical Character Recognition) au déchiffrage des codes postaux par La Poste, mais aussi pour la détection de nappes de pétrole dans l'océan, ou la détection de défauts dans les matériaux, qui permet des économies d'énergie par la réduction de la fréquence des arrêts de production.

J'ai donc réalisé un premier projet, relativement simple puisque utilisant un réseau à une couche auquel on applique la règle du delta, et réalisant de la reconnaissance de caractères non manuscrits.

3.1.1. PRINCIPE GÉNÉRAL



Ce réseau ne comporte qu'une seule couche, avec autant de neurones que de formes à reconnaître, c'est-à-dire que chaque neurone est responsable d'une forme. Chaque neurone est relié en entrée à tous les pixels de la forme présentée, et il donne en sortie la probabilité (%) qu'on lui ait présenté la forme dont il est responsable.

L'apprentissage, issu de la règle du delta, est de type supervisé : si la forme reconnue est la bonne on renforce l'apprentissage selon l'écart avec la seconde forme reconnue, si le réseau s'est trompé on renforce les valeurs de la forme qui aurait due être reconnue et on les baisse pour la forme reconnue.

3.1.2. FONCTIONNEMENT

Concernant les neurones :

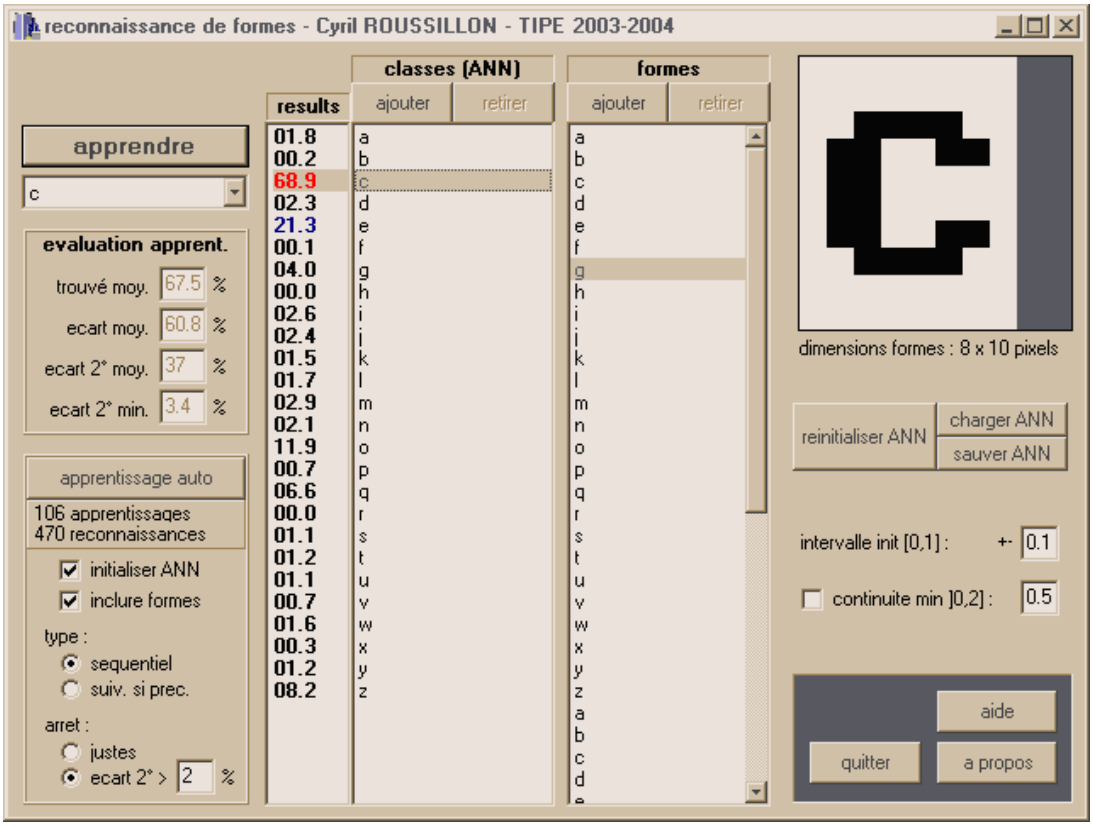
- Tous les poids sont des réels entre -1 et 1.
- Les valeurs des entrées sont simplement l'intensité du pixel en niveaux de gris (de 0 à 255), ramenée à un réel entre 0 et 1.
- La fonction d'entrée appliquée et une somme pondérée des entrées
- La fonction de transfert appliquée est une sigmoïde ($f(x) = 1 / (1 + e^{-x})$) qui se charge de ramener la sortie à un réel entre 0 et 1, qui représente la probabilité que ce soit la forme correspondante.

Au cours de l'apprentissage il s'agit donc pour chaque neurone d'affecter un poids important aux pixels activés de sa forme, d'autant plus s'il n'est pas activé pour les autres formes (c'est alors un pixel discriminant), et un poids beaucoup plus faible aux autres :

- **Correction.** Le réseau s'est trompé (la forme de plus grande probabilité n'est pas la bonne).
  On modifie les poids proportionnellement à la différence entre les probabilités de la forme trouvée et la forme réelle, en augmentant le poids des entrées activées pour la forme réelle et en le baissant pour la forme trouvée.
- **Renforcement.** Le réseau a trouvé la bonne forme.
  On modifie les poids proportionnellement au complémentaire de la différence entre les probabilités de la forme trouvée (et réelle) et de la forme arrivant en seconde position, en augmentant le poids des entrées activées pour la forme trouvée.

Un ajustement soigné (et expérimental) des coefficients de proportionnalité a été nécessaire afin d'obtenir de bons résultats, que vous pourrez apprécier avec le code source dans l'annexe B.

3.1.3. RÉSULTATS



➢ UTILISATION

Reconnaissance

Il faut commencer par charger dans "classes (ANN)" les formes qui constitueront les types de formes à reconnaître (fichiers bitmap .BMP qui doivent tous avoir les mêmes dimensions).

A chaque clic sur une forme, celle-ci se dessine et une reconnaissance s'effectue automatiquement, les résultats pour chaque classe s'affichant à gauche (probabilité en % pour qu'il s'agisse de la bonne forme). On peut charger de même dans "formes" les images que l'on souhaite, qui seront identifiées comme appartenant à la classe correspondante.

Apprentissage

Après chaque reconnaissance, on peut choisir "apprendre", après avoir éventuellement sélectionné la bonne classe dans la liste déroulante sous le bouton, que le réseau se soit trompé ou non .

Le programme comporte de plus une fonction d'apprentissage automatique :

Celui-ci peut être de deux types :

- "séquentiel" : chaque forme est reconnue dans l'ordre, si le réseau s'est trompé on procède à un apprentissage, puis on passe au suivant dans tous les cas.
- "suiv. si prec." : chaque forme est reconnue dans l'ordre, mais si le réseau s'est trompé on procède à un apprentissage puis on vérifie que les formes précédentes sont toujours correctement reconnues et on corrige si nécessaire. Ce mode est légèrement plus long mais donne des résultats un peu meilleurs.

L'apprentissage peut de plus s'arrêter quand toutes les formes sont identifiées sans erreur, ou avec une condition supplémentaire sur l'écart minimum entre une forme reconnue et celle arrivant en seconde position.

On peut également inclure les images chargées dans "formes" pour l'apprentissage. Dans ce cas il faut que leur nombre soit multiple du nombre de classes, et chaque forme est supposée comme appartenant à la classe correspondante dans "classes (ANN)". Les formes sont alors prises en compte pour tous les paramètres et dans l'évaluation de l'apprentissage.

Evaluation apprentissage

Après chaque apprentissage, une évaluation est effectuée, affichant plusieurs paramètres :

– "trouvé moy. " : pourcentage moyen de reconnaissance des formes.
– "ecart moy. " : écart moyen entre la forme reconnue et les autres.
– "ecart 2° moy. " : écart moyen entre la forme reconnue et la seconde.
– "ecart 2° min" : écart minimum entre une forme reconnue et la seconde.

Autres

On peut également préciser l'intervalle d'initialisation des poids du réseau, ou faire en sorte que la différence entre deux poids de pixels voisins ne dépasse pas la valeur spécifiée. Cette option était prévue pour de ne pas rendre un pixel trop décisif afin d'optimiser les capacités de généralisation du réseau, mais elle s'est révélée peu efficace.

Il est finalement possible d'enregistrer les poids d'un réseau sur le disque dur pour pouvoir le recharger lors d'une utilisation ultérieure.

➢ RÉSULTATS

Mon réseau parvient à reconnaître parfaitement les 26 lettres minuscules en une quarantaine d'apprentissages, en reconnaissant la lettre en moyenne à 80%, avec une moyenne d'écart de 50% avec les autres lettres, 20% avec la lettre arrivant en seconde position, et un écart minimum entre la lettre reconnue et la seconde tournant autour de quelques %.

On arrive à atteindre un écart moyen avec le second de 80% et un écart minimum de plus de 70% au bout d'environ 300 apprentissages.
Il parvient également à identifier les lettres parmi trois polices différentes avec un écart minimum supérieur à 5% après quelques 1500 apprentissages, c'est à dire que si on lui présente un 'a' d'une police ou d'une autre, il reconnaîtra un 'a'.

3.2. PRÉVISION DE TEMPÉRATURE

J'ai ensuite programmé un réseau multicouches, muni de l'algorithme d'apprentissage de rétropropagation du gradient et quelques unes de ses améliorations.

J'ai commencé par appliquer ce réseau à l'approximation de fonction, afin de le tester. En effet on voit ainsi très facilement la vitesse et l'efficience de l'apprentissage : basiquement, la courbe doit se rapprocher des points.

Puis une fois que j'étais sûr qu'il fonctionnait, j'ai pu l'appliquer à la prévision des valeurs d'une fonction à partir des précédentes, et finalement à la prévision de température après avoir fabriqué un capteur de température pour avoir une vraie base d'exemples d'apprentissage et de test.

3.2.1. RÉALISATION D'UN CAPTEUR

Plusieurs solutions s'offraient à moi pour réaliser un capteur de température et acquérir les données sur l'ordinateur, de la plus sophistiquée qui consiste à utiliser un capteur spécialisé et un convertisseur analogique–numérique connecté au port parallèle que l'on commande grâce à quelques lignes d'assembleur, à la plus simple que j'ai adoptée.

- **LE CAPTEUR : UNE THERMISTANCE**

En effet je voulais me débrouiller avec les moyens du bord, et je n'avais à disposition que quelques thermistances, dont la résistance constitue une grandeur thermométrique. Ainsi alimentant un diviseur de tension dont une thermistance constitue un élément, on dispose d'une tension dépendant de la température.
Après quelques essais à l'ohmmètre pour choisir celle qui proposait la meilleure amplitude de variation, le problème était donc réglé pour le capteur.

- **ACQUISITION DU SIGNAL : LA CARTE SON**

Je me suis ensuite souvenu avoir entendu parler d'utiliser la carte son pour acquérir un signal, en effet pourquoi acheter un convertisseur alors que l'on en dispose d'un avec une résolution de 16 bits ? Mais comment commander la carte son ? Connaissant la structure des fichiers WAV, ma première intention fut donc d'enregistrer de manière classique le « son », l'enregistrer sur le disque dur puis traiter le fichier WAV. En enregistrant par exemple une seconde à 1000 Hz, on fait la moyenne de ces 1000 acquisitions ce qui permet d'éliminer le bruit. Par chance C++Builder dispose d'un composant appelé TMediaPlayer permettant de lire et enregistrer des fichiers WAV en provenance de l'entrée micro ou ligne (« line in »), ce qui permet d'automatiser tout le processus avec un unique logiciel.

J'avais lu dans un magazine d'électronique que l'entrée micro proposait une tension à ses bornes afin d'alimenter les microphones piézo-électriques, ce que je vérifiai aisément, aussi valait-il mieux utiliser l'entrée ligne. Une fois réalisé le logiciel, j'ai donc tenté d'effectuer une mesure de tension, mais je n'ai obtenu aucun résultat, c'est-à-dire une tension quasi-nulle quelle que soit la tension appliquée. Après réflexion j'ai soupçonné l'existence d'un condensateur faisant office de filtre coupe-bas afin de bloquer le courant continu, qui il est vrai n'a aucun intérêt pour l'utilisation première de la carte son. J'ai donc appliqué la sortie d'un petit oscillateur et le signal est apparu sur mon écran !

Reste que l'impossibilité de mesurer une tension continue complique sérieusement la tâche : il me faut donc intégrer un oscillateur au montage ? En fait non, car désirant alimenter mon montage avec un transformateur, il m'a suffit de démonter le transformateur afin de court-circuiter le pont de diodes et je disposais alors d'une tension parfaitement sinusoïdale de 50 Hz.

- **STABILISATION**

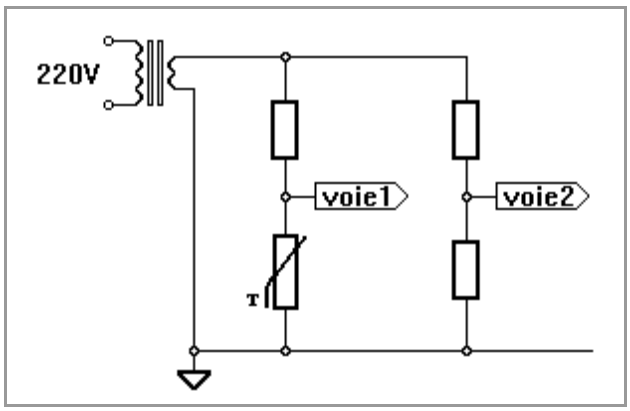
Mon dernier problème était que j'avais prévu de stabiliser la tension d'alimentation du diviseur de tension, car la thermistance ayant une résistance variable le courant débité par le transformateur l'est aussi, et les transformateurs ayant une forte résistance interne les mesures seraient faussées. Or je ne savais pas comment stabiliser une amplitude de tension variable.
J'ai donc eu l'idée d'utiliser la deuxième voie (stéréo) pour mesurer la tension aux bornes du transformateur et pouvoir ainsi corriger la tension obtenue par une banale règle de trois.
J'ai par la même occasion abandonné l'adaptateur d'impédance qui devait être constitué par un amplificateur opérationnel en montage suiveur, sachant que l'étalonnage devrait prendre en compte un éventuel courant d'entrée.

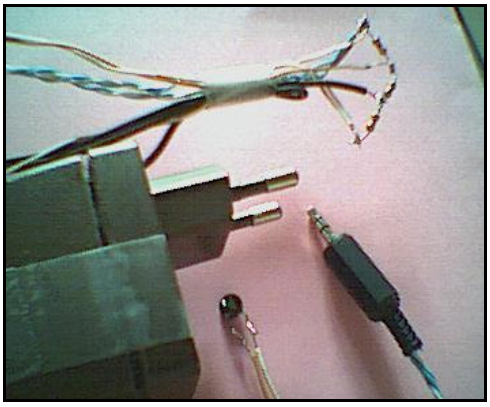
➢ **DERNIER DÉTAIL**

Il restait que la méthode d'enregistrer le fichier sur le disque dur ne me convenait pas totalement, car je craignais qu'enregistrer à répétition une dizaine de kilooctets toujours au même emplacement ne détériore le disque. J'ai donc cherché à enregistrer en stockant directement en mémoire, mais c'est finalement une autre solution, que l'on m'a proposée sur un forum, que j'ai choisie : utiliser un disque virtuel. C'est un pilote qui simule un disque dur, mais constitué de la mémoire vive, comme par exemple RamDrive que j'ai utilisé.

➢ **RÉSULTAT**

Le montage final, extrêmement simple comme promis se résume donc ainsi :





3.2.2. EXPLOITATION DU CAPTEUR

Une acquisition est un enregistrement de 2 secondes dont on fait la moyenne des amplitudes. Une acquisition est faite toutes les 10 secondes, et une mesure est un mixte moyenne-médiane des acquisitions sur une durée de 5 minutes (je retire le quart des valeurs extrêmes et je fais la moyenne des données restantes).

La valeur d'une mesure est donc un entier signé sur deux octets (type *short*) qu'il faut par conséquent convertir en une température. C'est le rôle de l'étalonnage.

➢ **ETALONNAGE**

J'ai donc disposé à côté du capteur plusieurs thermomètres à alcool, et relevé les valeurs des mesures pour des températures s'étalant de -10°C à +22°C à l'extérieur et à l'intérieur. Je me suis fié à trois thermomètres qui donnaient à peu près les mêmes températures (moins d'un degré celsius d'écart) pour éliminer ceux qui donnaient des valeurs éloignées de jusqu'à 3°C des autres.

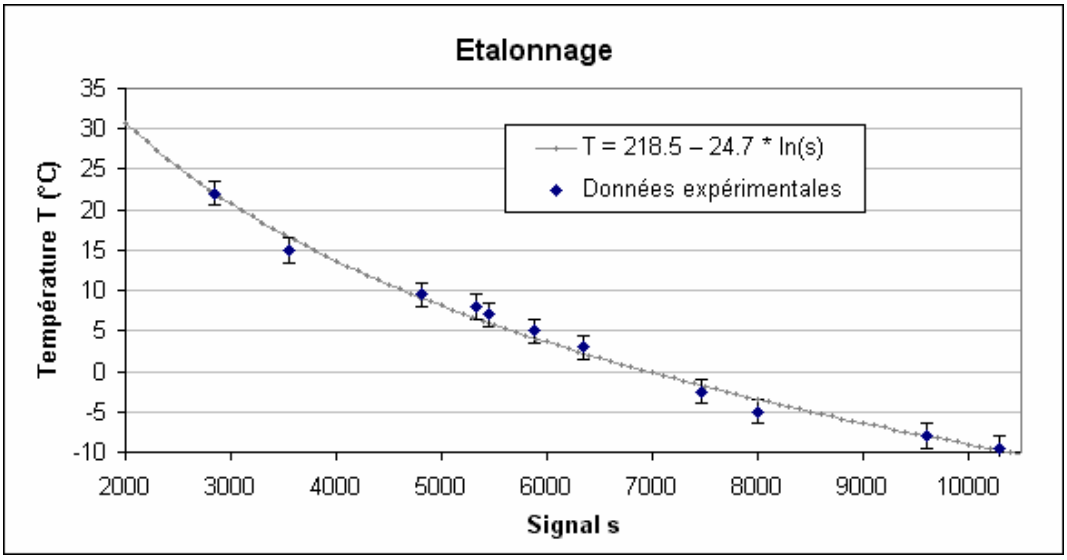
J'ai alors obtenu la série suivante de mesures :

Signal s	Température T (°C)
10300	-9.5
9600	-8
8000	-5
7470	-2.5
6350	3
5880	5
5450	7
5320	8
4810	9.5
3560	15
2850	22

J'ai ensuite entré les données dans ma calculatrice pour effectuer des régressions, et c'est une régression logarithmique qui s'est révélée être la plus adaptée, ce qui est cohérent avec un comportement exponentiel des composants électroniques non linéaires en général.

La formule de passage de la valeur donnée par la carte son à la température est :

$$T = 218.5 - 24.7 * \ln(s)$$



On obtient alors une précision du dixième de degré en stabilité, c'est-à-dire que à température constante les acquisitions restent dans une fourchette de 0.1°C. Une précision absolue est impossible à obtenir en l'absence d'étalon fiable, mais cela reste un problème d'étalonnage, secondaire.

➢ LOGICIEL D'ACQUISITION

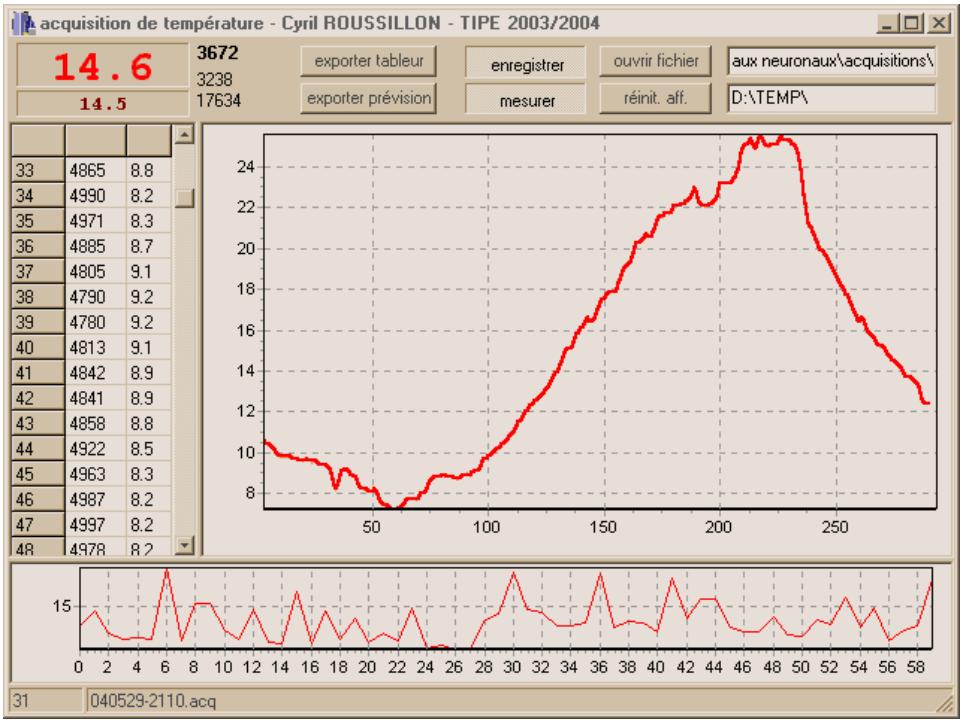
J'ai donc écris un logiciel d'acquisition de température, qui réalise les mesures comme décrit précédemment, et qui enregistre les valeurs dans un fichier afin d'être sauvegardées et qui les affiche simultanément dans un graphique.

On a également la possibilité d'ouvrir des fichiers précédemment enregistrés pour les visualiser, ainsi que d'exporter les données dans un fichier texte pour pouvoir être ouverts dans un tableur.

J'ai utilisé pour cela la structure des fichiers INI, et il faut dans le tableur choisir comme séparateur de colonnes le caractère '='. Il faut également prendre garde lors du tracé du graphique à ne pas sélectionner le titre, sous peine que les valeurs de l'axe des abscisses (l'heure) ne soient pas utilisées.

On peut également exporter les données pour être ouvertes dans le logiciel de prévision que je présente juste après. Le fichier est constitué des couples (x,y) de variables copiées directement (type float : sur 4 octets). On a aussi la possibilité d'appliquer une échelle sur les températures, ainsi que de sous échantillonner (voir suite).

Le logiciel se présente donc ainsi :



➢ UTILISATION

J'ai donc pu obtenir les températures de plusieurs journées avec une résolution de cinq minutes. Je souhaitais au début laisser tourner l'ordinateur pendant mon absence, mais il s'est révélé qu'au bout de plusieurs heures Windows ralentissait, se bloquant même par moments et allant jusqu'à faire retarder l'horloge, ce qui faussait les mesures. J'ai donc effectué les mesures quand j'étais là, et je le redémarrais régulièrement.

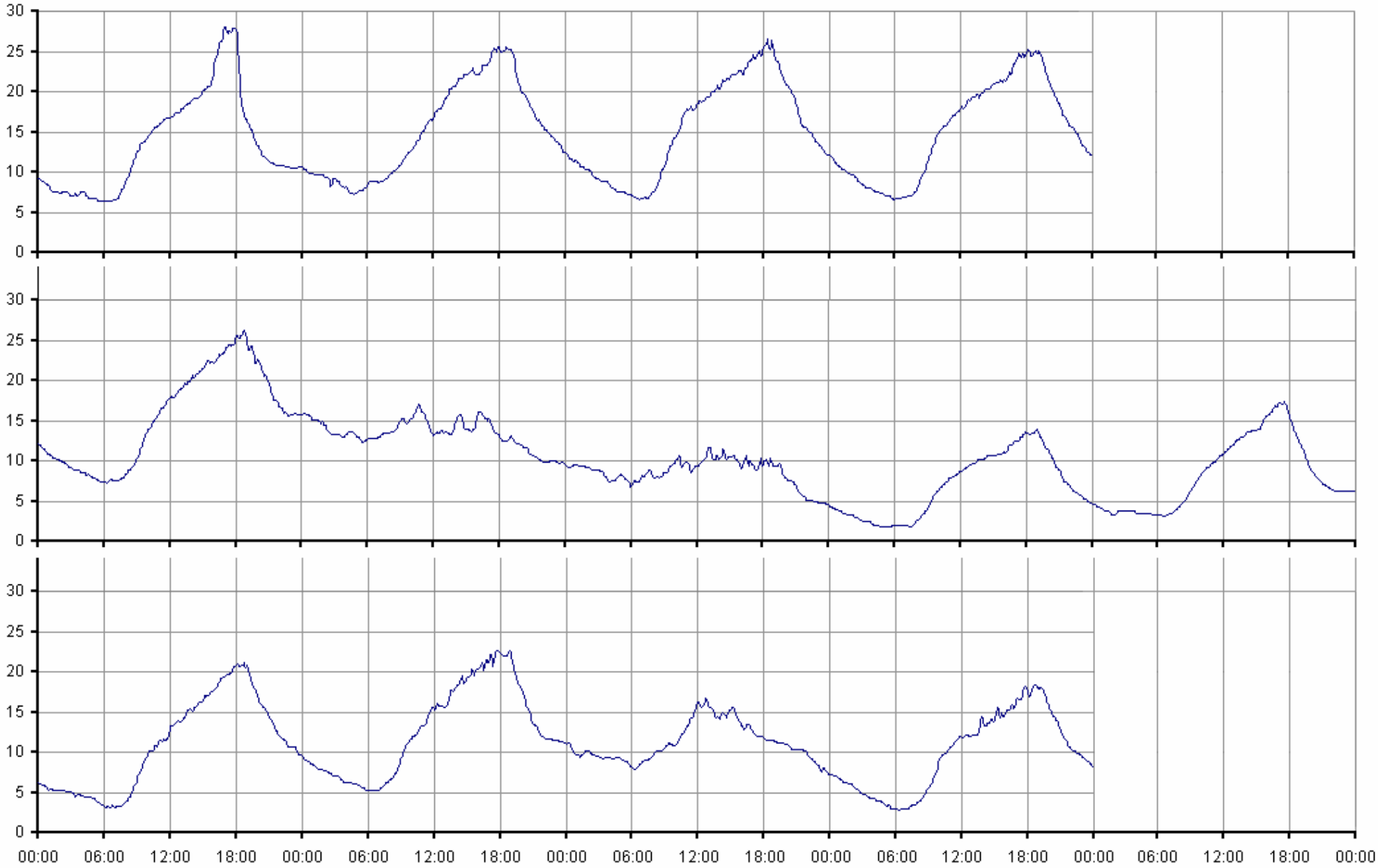
Les mesures obtenues sont présentées à la page suivante.

On constate que la tendance générale est très nette lors des journées ensoleillées : le minimum de température se trouve juste avant le lever du soleil après 6h, et le maximum un peu avant son coucher. Il devrait normalement se situer vers 15 ou 16 heures, mais le capteur se trouve sur une façade orientée Nord-Ouest qui commence à voir le soleil vers 16 heures. Le capteur étant à l'abri du rayonnement direct du soleil (dans un cadre de fenêtre), on peut affirmer que c'est bien la température de l'air de ce côté de la maison qui augmente lorsque le soleil arrive.

On remarque également des irrégularités, qui sont bien de réelles variations de température puisque s'étalant sur plusieurs acquisitions, voire plusieurs heures.

En revanche lors des journées pluvieuses, ou du moins couvertes par un manteau nuageux, les irrégularités augmentent considérablement, et il n'y a plus de tendance très nette, sinon qu'il fait plus froid la nuit que la journée, et encore.
On imagine aisément que cela peut poser de sérieux problèmes à une quelconque tentative de prévision de la température, mais on verra qu'il en est de même des quelques irrégularités lors des journées ensoleillées. On procèdera donc à un sous échantillonnage des températures, en faisant des moyennes sur une heure.

Température extérieure du 16/05/04 au 28/05/04 à PONTARLIER (25)



3.2.3. RÉALISATION D'UN PROGRAMME DE RÉGRESSION ET DE PRÉVISION

J'ai donc mis en œuvre la méthode d'apprentissage par rétropropagation du gradient dans un programme réalisant l'approximation de fonctions, ainsi que prévision des futures valeurs de la fonction lorsqu'on lui présente des valeurs passées consécutives.

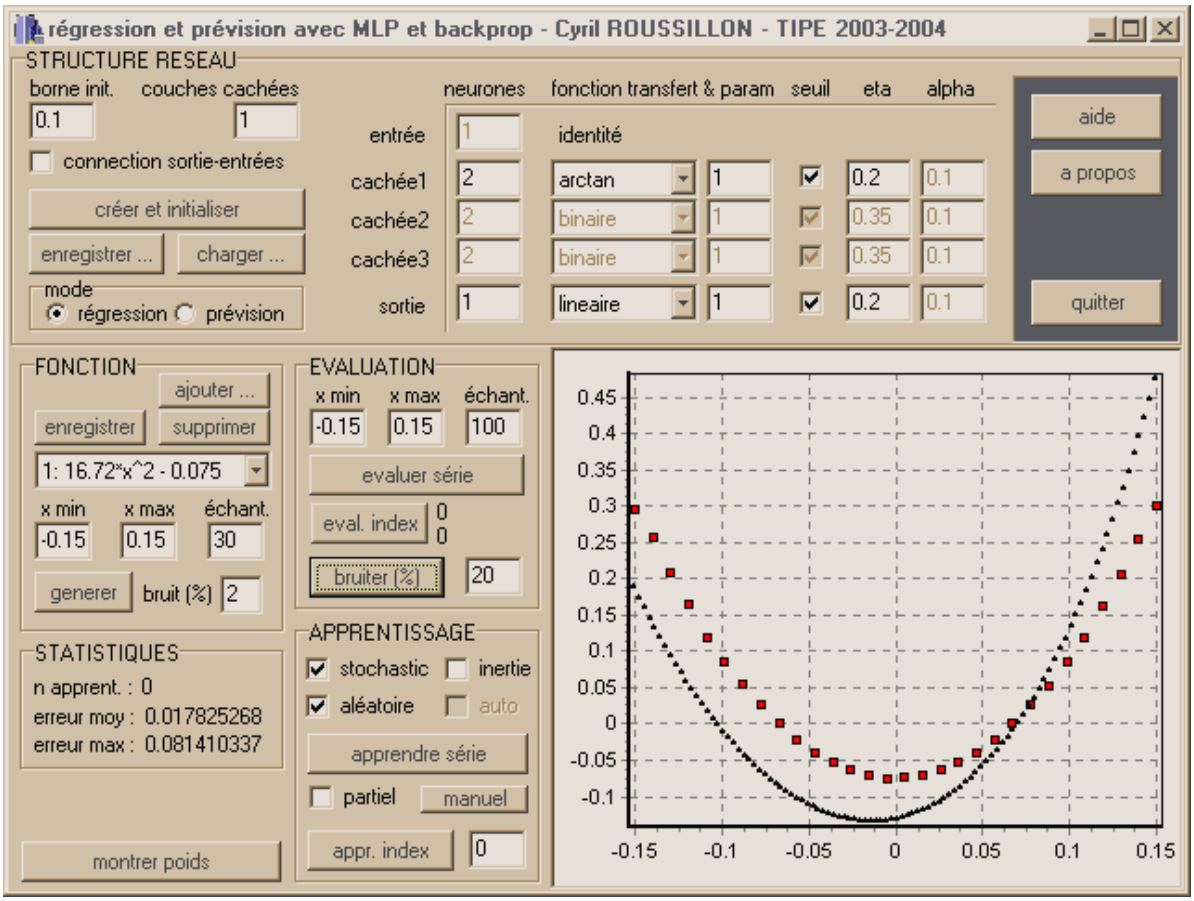
J'ai fait le choix de l'approximation de fonctions (ou régression) car elle permet de visualiser immédiatement la vitesse et l'efficience de l'apprentissage. C'est donc uniquement dans un but pédagogique, car je le rappelle, les réseaux de neurones n'ont aucun intérêt particulier pour des fonctions à une variable, si ce n'est qu'il est généralement constaté qu'ils sont moins sensibles au bruit que d'autres méthodes. Mais il n'empêche que le principe et les méthodes sont les mêmes que pour tout apprentissage supervisé à l'aide de l'algorithme de rétropropagation du gradient. Cette application montre également les capacités de généralisation des réseaux de neurones, qui s'exprime ici par l'inférence statistique : le réseau retrouve la valeur que doit avoir la fonction entre les points d'échantillonnage.
En revanche l'application prend tout son intérêt pour effectuer de la prévision, puisque le nombre d'entrées devient grand.

➢ **MISE EN OEUVRE**

J'ai modélisé le réseau dans des *classes* C++ : une première modélise un neurone et contient son état interne ainsi que ses poids, stockés dans un tableau. Une seconde classe modélise une couche de neurones, contenant les neurones dans un tableau et le nombre de neurones de la couche. Enfin une dernière classe modélise le réseau lui-même, contenant les couches de neurones, ainsi que les *méthodes* d'utilisation du réseau (une méthode d'une classe est une fonction dont la classe est propriétaire et qui a accès à toutes ses *propriétés*).
J'ai également mis en place une série de fonctions permettant d'accéder à tous les paramètres du réseau (poids, sorties …) avec les mêmes notations que les démonstrations du dossier.

Le code complet entièrement commenté de la modélisation et de l'apprentissage se trouve en annexe B.

➢ **DESCRIPTION & UTILISATION**



Structure du réseau

On a la possibilité de totalement paramétrer le réseau, avec un maximum de trois couches cachées, ce qui laisse un large éventail de possibilités.

Pour chaque couche de neurone, on peut choisir :
- le nombre de neurones de la couche
- la fonction de transfert utilisée par les neurones de la couche ainsi que son paramètre (voir 1.1.2).
- le fait que les neurones de la couche possèdent un seuil, ajustable par l'apprentissage
- les coefficients d'apprentissage qui seront utilisés pour les neurones de cette couche

On peut également préciser si on veut qu'il existe une connexion entre les entrées et les neurones de la couche de sortie, ce qui peut permettre de traiter la partie linéaire d'un problème, ainsi qu'une borne qui délimite les valeurs possibles d'initialisation des poids lors de la création du réseau.

Il est aussi possible d'enregistrer sur le disque dur un réseau satisfaisant, puis de le recharger ultérieurement.

Enfin, il existe deux modes :
- régression : le réseau tente, quand on lui fournit une valeur d'abscisse, de fournir la valeur associée (le réseau comporte alors une seule entrée, modélisée par le nombre de neurones de la pseudo couche d'entrée).
- prévision : on fournit au réseau un certain nombre de valeurs consécutives de la fonction (le nombre d'entrées), et il tente de prédire la valeur suivante. Cette prévision peut être étendue à une plus grande échéance (précisée dans "Durée max prev." de la zone "Prévision"), en réutilisant des valeurs déjà prédites en entrée.

Fonction

On peut choisir ici dans la liste déroulante la fonction de son choix à utiliser pour l'apprentissage du réseau en régression ou prévision. Celle-ci est stockée avec son intervalle d'utilisation et le nombre d'échantillonnages à utiliser lors de son tracé, échantillonnages qui constituent la base d'exemples d'apprentissage du réseau. De plus, afin de simuler des données expérimentales, du bruit peut être ajouté à la fonction lors de son tracé.

Afin de pouvoir utiliser n'importe quelle fonction, j'ai écrit un **interpréteur d'expressions arithmétiques**. Le code source se trouve en annexe B : l'expression est analysée puis convertie en un arbre, qui permet le calcul de la fonction en n'importe quel point.

On peut donc ajouter une fonction, la supprimer de la liste, ou simplement modifier son intervalle d'utilisation (bouton "enregistrer"). Il est également possible d'obtenir les données à partir d'un fichier, sous le format décrit dans le paragraphe sur le capteur de température (3.2.2).

Evaluation

On peut dans cette zone définir l'intervalle d'évaluation du réseau, qui peut être avantageusement plus grand que celui d'utilisation de la fonction (et donc d'apprentissage), pour visualiser le comportement du réseau dans un domaine ou il n'a subit aucun apprentissage.

On peut ensuite évaluer le réseau sur tout l'intervalle d'évaluation ("evaluer série"), ou un seulement un indice précis de l'échantillonnage de la fonction en mode régression. S'affichent alors la valeur attendue et la valeur rendue.

Il est également possible de bruiter les poids du réseau pour dégrader l'apprentissage, afin de pouvoir examiner comment le réseau converge de nouveau.

Statistiques

Lors de chaque évaluation du réseau, l'erreur qu'il commet par rapport à la fonction est calculée : il s'agit de l'erreur relative (par rapport à l'amplitude de la fonction sur son intervalle d'utilisation) moyenne, ainsi que de l'erreur relative maximale.

Est également affiché le nombre d'apprentissages qu'à subit le réseau depuis son initialisation.

Apprentissage

Plusieurs algorithmes d'apprentissage sont disponibles :

- *Gradient simple* : Si aucune case à cocher n'est cochée, correspond à l'algorithme de base. Chaque exemple est soumis au réseau, l'erreur est calculée, rétropropagée, et les corrections sont stockées. Une fois que tous les exemples ont été traités, on applique la moyenne des corrections au réseau.
- *Gradient stochastique* : Ici les corrections sont aussitôt appliquées après chaque exemple soumis. On peut également présenter les exemples dans un ordre aléatoire, ce qui peut améliorer la convergence mais aussi augmenter l'instabilité.
- *Gradient inertie* : Un terme est ajouté à la correction, constitué du produit d'un coefficient alpha et de la correction précédente. Les coefficients eta et alpha peuvent être calculés automatiquement par l'algorithme présenté en 2.2.3.
- *Manuel* : Pour un réseau à une couche cachée de deux neurones, trace les applications partielles, permettant de localiser manuellement chaque minimum.

On peut alors effectuer l'apprentissage sur toute la base d'apprentissage, ou seulement sur un indice (l'indice est le même que celui de la zone Evaluation). On peut également n'utiliser qu'un exemple sur deux pour l'apprentissage, mais tous pour calculer l'erreur, ceci afin de vérifier la capacité de généralisation.

Il est finalement possible de visualiser en temps réel les poids du réseau, en cliquant sur le bouton "montrer poids".

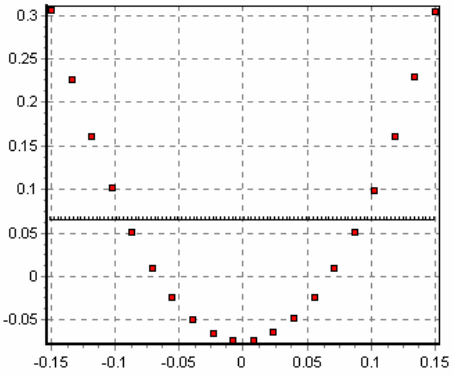
3.2.4. RÉSULTATS

➢ RÉGRESSION

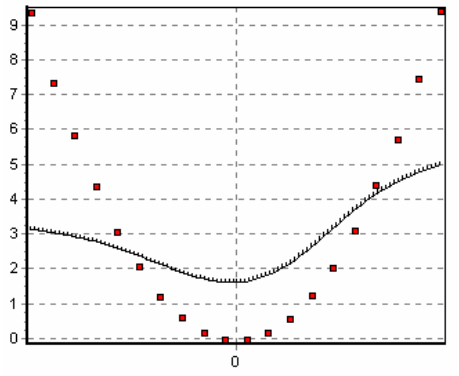
La méthode qui s'est révélée la plus efficace est la descente stochastique avec présentation aléatoire. L'ajout d'un terme d'inertie n'a curieusement pas beaucoup amélioré les résultats.

Le réseau s'est révélé très sensible à la représentation des données d'entrée, ce qui est d'ailleurs caractéristique des réseaux de neurones, en l'occurrence aux échelles.
Il est également parfois nécessaire de jouer finement sur les coefficients d'apprentissage pour obtenir une convergence.

Prenons l'exemple de la parabole $y = 16.72.x^2 - 0.075$, avec un réseau à une couche cachée comportant deux unités, et muni de la fonction de transfert arc tangente ainsi que de seuils ajustables.
On utilise pour l'apprentissage une descente stochastique.

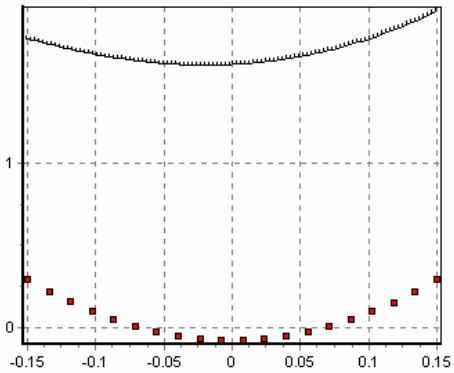


Si l'on tente un apprentissage sur l'intervalle -0.15 à 0.15, le réseau ne parvient pas à prendre la forme de la fonction du fait de la symétrie, et ne converge pas du tout, et ce quels que soient les coefficients d'apprentissage.

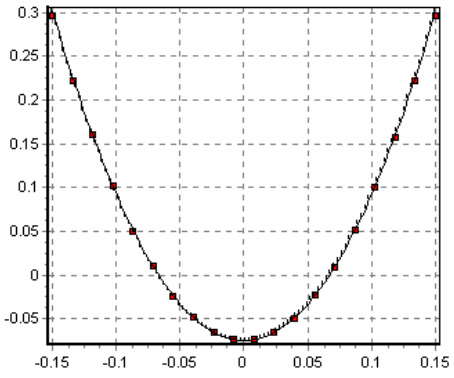


Si l'on passe entre -0.75 et 0.75, avec un petit coefficient d'apprentissage même si le réseau bouge plus, il ne parvient toujours pas à prendre la forme.

En revanche avec un coefficient de 0.5, il commence immédiatement à prendre la courbure de la parabole.



En revenant dans l'intervalle -0.15 à 0.15, le réseau est très éloigné mais possède maintenant la bonne concavité.

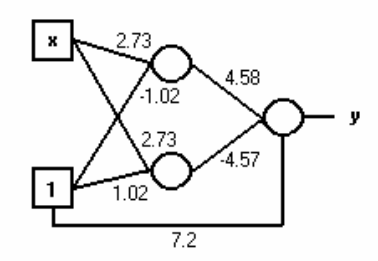


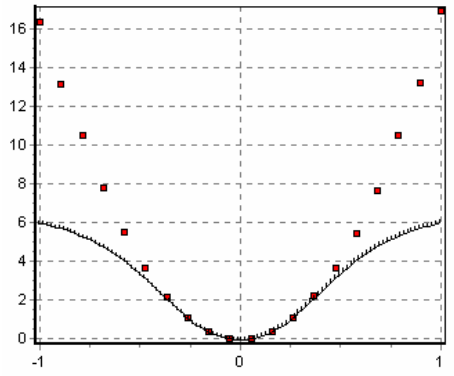
La convergence se fait ensuite très facilement, et en quelques centaines d'apprentissage on obtient une très bonne approximation, avec une erreur moyenne de .

On constate en examinant les poids synaptiques que l'on retrouve une symétrie, due à celle du problème :

spy

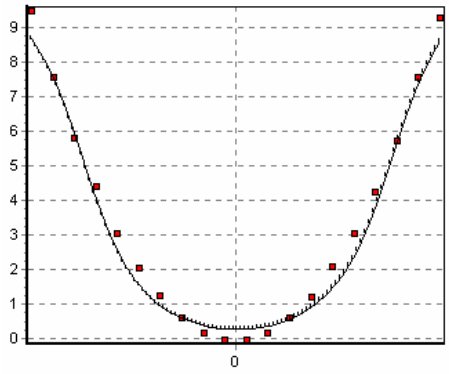
	poids	couche 1	poids	couche 2
sortie	**2.73**	-.55	**4.58**	.3
potentiel		-.61	**-4.57**	.3
seuil		**-1.02**		**7.2**
delta		1.817		
sortie	**2.73**	.96		
potentiel		1.43		
seuil		**1.02**		
delta		3.530		





On peut cependant constater que l'approximation se dégrade nettement en dehors de l'intervalle d'apprentissage.

Ceci est normal puisque le réseau n'a été soumis à ces valeurs.
Cela ne remet pas en cause sa capacité de généralisation, qui s'exprime entre les points d'échantillonnage de la fonction sous forme d'inférence statistique.



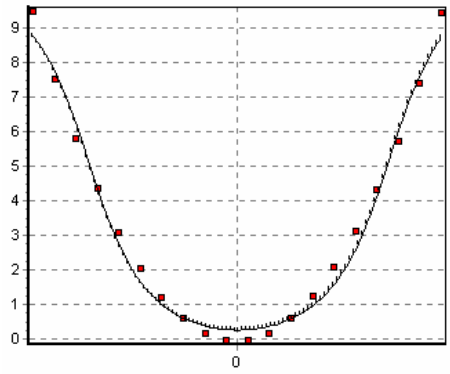
spy

	poids	couche 1	poids	couche 2
sortie	**-5.71**	-1.44	**4.38**	8.6
potentiel		-7.48	**4.43**	8.6
seuil		**-3.2**		**11.44**
delta		-.18		-.3
sortie	**5.63**	.78		
potentiel		.99		
seuil		**-3.23**		
delta		-.08		

On peut également à partir de l'étape 2 continuer l'apprentissage sur l'intervalle -0.75 à 0.75.

Il faut cependant veiller à diminuer le coefficient d'apprentissage sous peine d'osciller, voire même tendre vers l'infini.

L'approximation est un peu moins bonne sur cet intervalle, probablement à cause de la forme de la fonction de transfert.

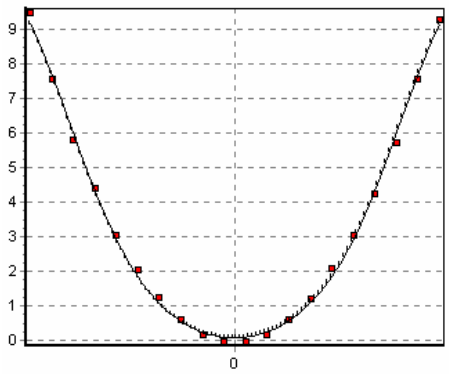


spy

	poids	couche 1	poids	couche 2
sortie	**-5.76**	-1.44	**4.35**	8.75
potentiel		-7.46	**0.00**	8.75
seuil		**-3.14**	**4.44**	**11.35**
delta		1.817		
sortie	**0.00**	.00		
potentiel		.00		
seuil		**.00**		
delta		8.380		
sortie	**5.64**	.82		
potentiel		1.08		
seuil		**-3.15**		
delta				

Si on passe à 3 neurones cachés, on constate que le réseau peut converger vers des configurations où un neurone est tout simplement éliminé, et que les coefficients des deux autres correspondent à ceux du cas précédent avec 2 neurones cachés.

Cela signifie donc que le minimum global du réseau à 2 neurones cachés est un minimum local du réseau à 3 neurones cachés.

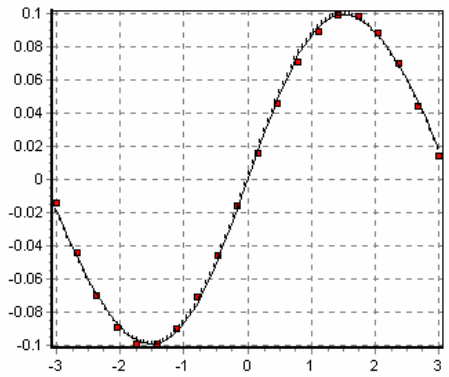


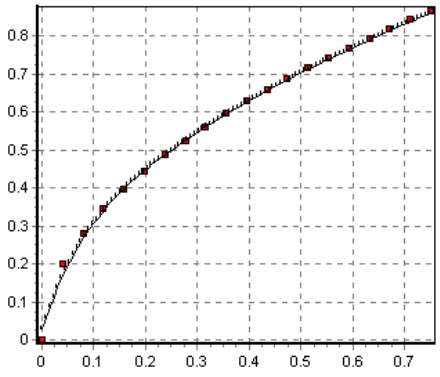
spy

	poids	couche 1	poids	couche 2
sortie	**-3.44**	-1.36	**7.02**	9.18
potentiel		-4.65	**-6.44**	9.18
seuil		**-2.07**	**7.16**	**4.82**
delta				
sortie	**-3.57**	-.51		
potentiel		-.56		
seuil		**2.12**		
delta				
sortie	**4.31**	1.48		
potentiel		11.41		
seuil		**8.18**		
delta				

En effet en utilisant réellement les 3 neurones cachés on obtient une bien meilleure approximation.

On peut ainsi approcher d'autres fonctions, comme par exemple la fonction sinus ou racine carrée :





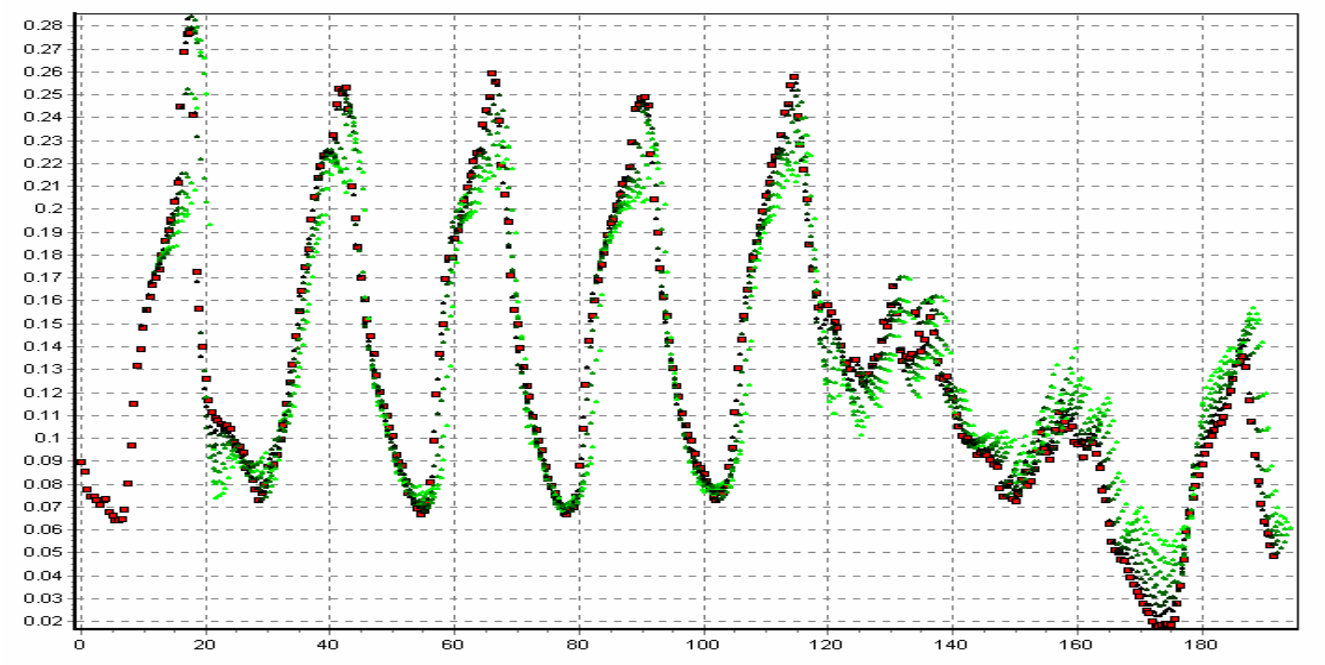
➢ PRÉVISION

Le réseau effectue par défaut une prévision à un pas de temps : on fournit à ses n entrées les n dernières températures, et il donne en sortie sa prévision pour la suivante. On peut cependant prolonger à plusieurs pas de temps en réutilisant les valeurs prédites par le réseau en les injectant en entrée.

J'ai choisi cette méthode car elle est la plus simple : elle ne nécessite qu'un réseau. On aurait pu aussi penser à une série de réseaux, celui d'indice k effectuant une prévision à k pas de temps, et ayant en entrée les prévisions des réseaux d'indice inférieur à k. A priori cela revient au même, mais en fait une même entrée ne pourra pas être tantôt initialisée avec une valeur prédite, tantôt avec une valeur réelle. Ainsi chaque réseau pourra accorder plus de crédit aux entrées correspondant aux valeurs réelles, naturellement plus fiables.

Grâce aux mesures de température effectuées j'ai pu faire un jeu d'exemples d'apprentissage avec les 8 premiers jours, et un jeu d'exemples de test avec les 5 derniers. J'ai regroupé les mesures pour avoir un échantillon toutes les 30 minutes, afin d'avoir des variations significatives.

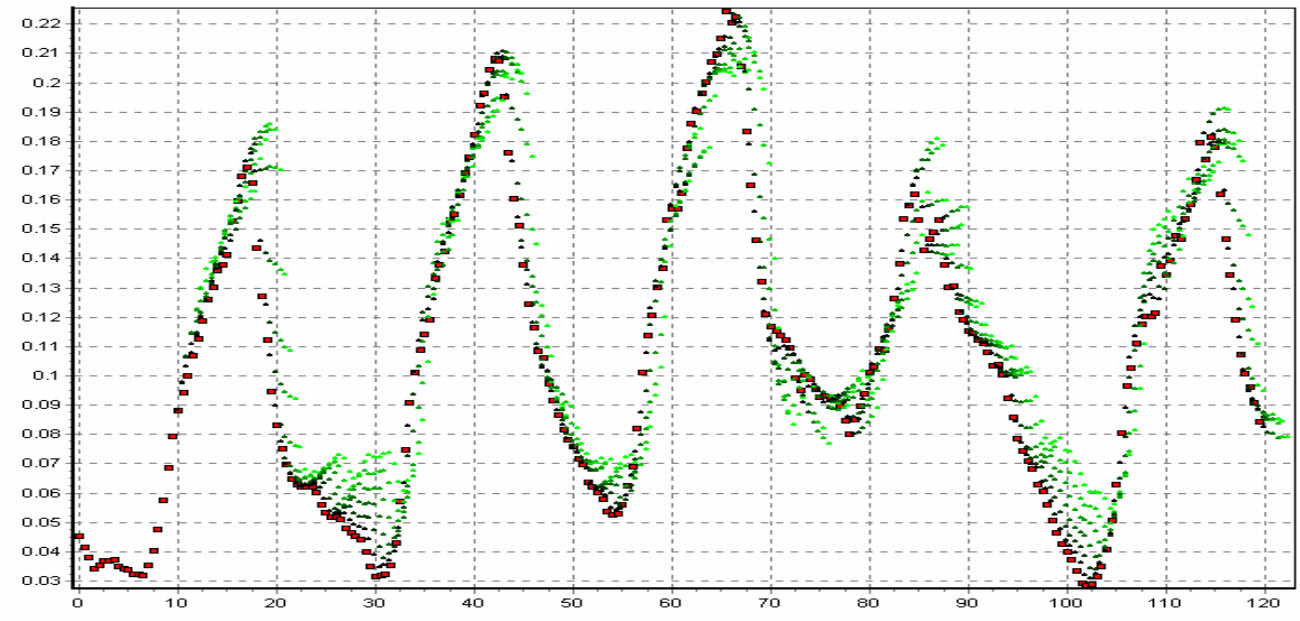
Avec un réseau muni d'une couche cachée de 8 neurones on obtient après apprentissage le résultat suivant :



Les vraies températures sont en rouge, et les prévisions en vert. Plus la prévision est lointaine, plus le vert est pur. Ainsi les prévisions à un pas de temps (30 minutes) sont en noir, et les prévisions à 6 pas de temps (3 heures) sont en vert pur. L'axe des températures a été divisé par 100, afin de mettre les données à une échelle traitable par le réseau. Ainsi 0.28 correspond à 28°C.

On peut ainsi juger la prévision au fait que les points verts doivent être le plus proche possible des points rouge, l'éparpillement doit être minimal.
On constate ainsi que par moments le réseau est trompé par des variations accidentelles de la courbe, mais l'erreur de la prévision à 3 heures ne dépasse jamais 3°C, et est de l'ordre de 1°C, ce qui est tout à fait satisfaisant.

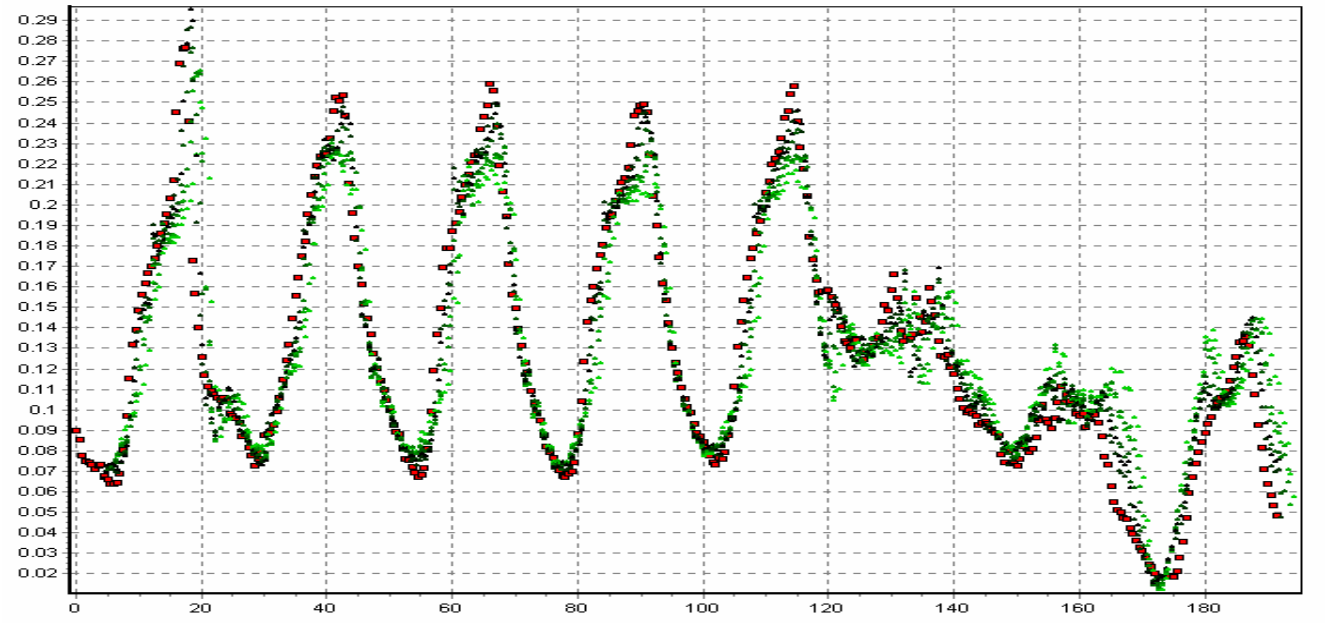
On peut ensuite tester le réseau sur la base d'exemples de test. On constate une performance similaire, alors que le réseau n'a jamais été soumis à ces données auparavant, ce qui montre les capacités de généralisation des réseaux de neurones.



Afin d'améliorer la prévision, on peut fournir l'heure au réseau : il est évident que connaître le moment de la journée permet d'interpréter différemment les précédentes valeurs de la température, pour une prévision plus pertinente.

J'ai fourni cette information au réseau sous la forme d'un réel entre 0 et 1 communiqué à une entrée du réseau.

On constate alors que les résultats obtenus sont meilleurs, puisque l'on arrive à un éparpillement plus réduit même avec seulement 10 neurones d'entrée (les températures des 5 dernières heures sont fournies en entrée) contre 20 neurones précédemment (10 heures), toujours avec une échéance maximale de 3 heures :



Cela montre qu'il est possible de prendre des paramètres extérieurs en considération, et on pourrait donc prendre en compte de la même façon l'ensoleillement ou l'hygrométrie, qui sont liés à l'évolution de la température, afin d'améliorer encore la prévision.

3.3. APPLICATIONS DE LA PRÉVISION DE TEMPÉRATURE

Un tel module de prévision de température peut trouver de nombreuses applications en s'incorporant directement dans de plus gros projets, mais aussi le principe même de la prévision peut très facilement s'adapter à d'autres grandeurs.

Je présente maintenant deux applications pratiques, qui ont effectivement été réalisées, et qui nécessitent une prévision de température.

3.3.1. RÉGULATION DE CHAUFFAGE NEURO-FLOUE

Ce principe a été concrètement mis en place par une équipe suisse, sous le nom de projet NEUROBAT, et par le chercheur Pierre-Yves GLORENNEC dans une crèche rennaise. Ils ont obtenus des résultats supérieurs à ceux des régulations classiques de chauffage, et cette installation présente l'avantage de ne pas nécessiter de réglage régulier puisque le système s'adapte à l'environnement. Les références de ces projets se trouvent dans la bibliographie.

J'ai pour ma part réaliser un avant-projet, de manière à me faire une idée de la façon dont pourrait être constituée une telle installation.

➢ PRÉSENTATION

Le but est de réguler la température d'un système (pièce, bâtiment) à l'aide d'un système de chauffage (chaudière, radiateurs électriques d'appoint) en utilisant au maximum les apports gratuits (gains solaires, gains internes dus aux personnes et machines) afin de réaliser des économies d'énergie, tout en offrant un confort maximal.

Différents éléments sont à prendre en compte :
- chauffage au sol (chaudière) : économe mais plusieurs heures de réaction
- chauffage électrique d'appoint : réactif mais peu économe
- température extérieure (isolation, aération)
- ensoleillement (vitrage)
- personnes présentes (apports de chaleur, et température souhaitée)

Le principe est d'utiliser un réseau neuronal (multicouches avec rétropropagation du gradient) dont le but est de prévoir les conditions de l'environnement et du système plusieurs heures à l'avance (en particulier la température prévue de la pièce), couplé à un système flou dont le but est à partir de ces prévisions de commander le système de chauffage.

➢ AVANT-PROJET

J'ai donc réalisé un avant-projet du réseau de prévision, que je n'ai pu concrétiser par manque de temps. En effet il m'aurait fallu pour le tester modéliser complètement l'environnement ce qui est relativement long.

Le réseau devrait donc avoir les propriétés suivantes :

Entrées :
- Horloge (réel de 0 à 1)
- Température extérieure (capteur)
- Température intérieure (capteur)
- Température dalle chauffante (capteur)
- Ensoleillement (photorésistance)
- Créneau de présence
- Température souhaitée bâtiment occupé
- Température souhaitée bâtiment vide
- Etat du système de chauffage

Sortie : température intérieure prévue dans plusieurs heures

Il s'agit alors lors de l'apprentissage de trouver le minimum d'une fonction de coût dépendant a priori de l'énergie consommée et du confort des occupants. On a alors deux possibilités :

- Le chauffage électrique d'appoint, thermostaté et indépendant, se charge de maintenir la pièce à la bonne température. Sa consommation est incluse à la fonction de coût énergétique, le but étant de réduire son rôle. Le confort des occupants est maximal, mais l'entrée température intérieure, constante, devient inutile.

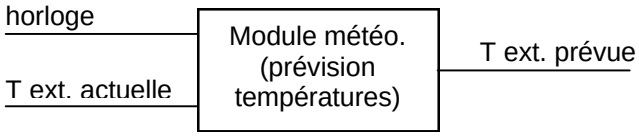
- Il n'y a pas de chauffage électrique d'appoint, la fonction de coût inclus le confort des occupants (que l'on peut ramener à l'énergie que consommerait le chauffage électrique d'appoint s'il devait maintenir la pièce à la bonne température. L'avantage est que le réseau connaît la température intérieure due à ce qu'il contrôle.

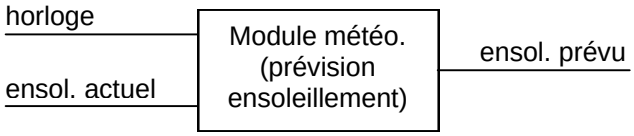
Structure :
Les entrées d'un réseau doivent être homogènes pour permettre un apprentissage et un fonctionnement efficaces (on ne peut pas fournir les entrées en vrac). Aussi il faut séparer ce réseau en différents modules et blocs dont l'apprentissage sera indépendant dans un premier temps, puis que l'on assemblera.

Module prévisions météorologiques sur quelques heures (réseau neuronal)
Entrées :
- Horloge
- Température extérieure actuelle
- Ensoleillement actuel

Sorties :
- Température extérieure prévue dans x heures
- Ensoleillement prévu dans x heures

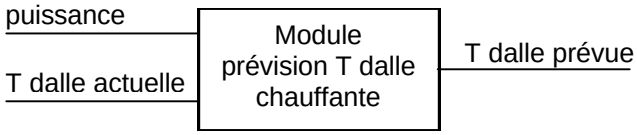
On constate qu'il y a trois données non homogènes : horloge, température, ensoleillement. Mais on peut séparer ce module en deux blocs, température et ensoleillement afin de réduire cette non homogénéité.

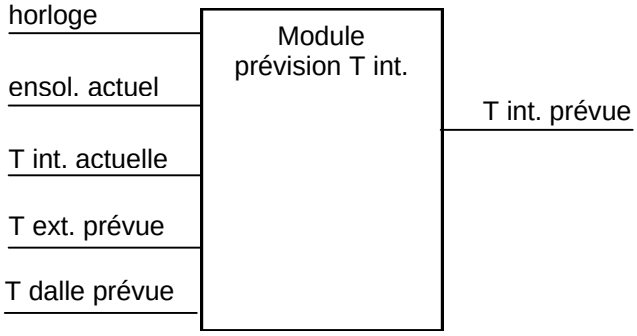




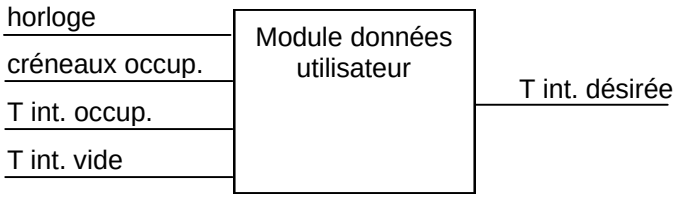
On réalisera son apprentissage indépendamment en conditions réelles.

Module prévisions température intérieure sur quelques heures (réseau neuronal)
La température de la dalle pouvant être mesurée, on sera certainement gagnant à prévoir indépendamment la température de la dalle, puis l'utiliser pour prévoir la température intérieure.





Module données utilisateur (algorithme classique)



Le système flou, connaissant la température actuelle, la future température ainsi que la température désirée pourra régler au mieux la chaudière afin d'atteindre l'objectif.

3.3.2. PRÉVISION DES PICS D'OZONE

Les réseaux neuronaux peuvent aussi être utilisés pour tenter de prédire les taux d'ozone, ceci afin de prévenir les pics d'ozone, dangereux pour la santé des personnes fragiles. Une telle prévision nécessite d'une part la connaissance et une prévision de la température, et d'autre part il s'agit du même principe de prévision que pour la température.

Des équipes de recherche françaises ont mis en place un tel système ; leur réseau prenait pour entrée :
- les mesures d'ozone précédentes, ainsi que d'autres taux tels NO ou NO2
- les températures extrêmes passées ainsi que les températures prédites

Ils ont obtenus des résultats encourageants, puisqu'ils ont constaté une erreur moyenne de prédiction de 23 $\mu g/m^3$ alors que le seuil d'alerte était fixé à 180 $\mu g/m^3$.

➢ **CONCLUSION**

Les réseaux de neurones artificiels peuvent donc donner de bons résultats dans la prévision de température. Bien sûr le programme que j'ai réalisé reste primitif, dans le sens où il pourrait donner de bien meilleurs résultats en prenant en compte d'autres paramètres comme l'ensoleillement ou l'hygrométrie.

Cependant la méthode utilisée peut être appliquée à toutes sortes d'autres prévisions.

CONCLUSION

Les réseaux de neurones artificiels donnent donc des résultats encourageants dans des applications comme la reconnaissance de formes et la prévision de température.

Cependant il faut prêter garde à ne pas négliger les techniques plus classiques qui ont fait leurs preuves, ou qui peuvent encore être améliorées, car les réseaux neuronaux présentent l'inconvénient d'être des boîtes noires, c'est-à-dire qu'une fois l'apprentissage réalisé, il est très difficile d'interpréter les coefficients synaptiques pour déterminer sur quels critères se base le réseau.
Mais c'est aussi ce qui rend les réseaux neuronaux fascinants, car ils parviennent à extraire la connaissance de données brutes que constituent les exemples, alors que nous sommes incapables de la leur communiquer directement.

ANNEXE A : BIBLIOGRAPHIE

Je précise après la source ce que j'en ai tiré principalement, c'est-à-dire ce que j'ai trouvé le plus intéressant à retenir, parce que je ne l'ai trouvé que là ou que c'était le mieux présenté. Les chapitres indiqués sont soit ceux qui traitaient le sujet, ou les seuls dont je disposais dans le cas des ouvrages électroniques au format PDF téléchargés sur Internet.

LIVRES

Jean-Marc Culloch, Intelligence Artificielle et Informatique Théorique, Cépaduès-Editions, 2002
chap. 20 *Les réseaux de neurones*
Démonstration rétropropagation du gradient

Hélène Ventsel, Théorie des probabilités, Editions Mir, 1973
chap. 5, 6, 8, 10
Système de variables aléatoires, fonctions de variables aléatoires, loi normale

François Blayo et Michel Verleysen, Les réseaux de neurones artificiels, PUF, 1996
Collection Que sais-je? N°3042
Présentation générale

Jean-Baptiste Hiriart-Urruty, L'optimisation, PUF, 1996
Collection Que sais-je? N°3184
Conditions de minimalité

REVUES

Pour la science, L'apprentissage en informatique, N°212, Juin 1995
Intérêt de l'injection de connaissances dans un réseau

La Recherche, Darwin revisité par la sélection artificielle, N°350 Spécial « Les nouveaux robots », 02/02
Robotique évolutionniste : découverte des réseaux neuronaux

OUVRAGES ÉLECTRONIQUES (FORMAT PDF)

http://www.trop.univ-mulhouse.fr/rapportsPDF/these/BretonChap3.pdf, Thèse Université de Mulhouse, 1999
Stéphane BRETON, Une approche neuronale du contrôle robotique utilisant la vision binoculaire par reconstruction tridimensionnelle, chap. 3 *Les réseaux de neurones artificiels*
Approche probabiliste de la fonction d'erreur, développement de Taylor à plusieurs variables

http://www.editions-eyrolles.com/Chapitres/9782212110197/chap01.pdf, Editions Eyrolles, 2002
Gérard DREYFUS, Réseaux de neurones, chap. 1 *Les réseaux de neurones, pourquoi et pour quoi faire ?*
Approche statistique de l'approximation

www.cavalex.com/pdf/livre_touzet.pdf, juillet 1992
Claude Touzet, Les réseaux de neurones artificiels, Introduction au connexionnisme
Présentation générale

PAGES INTERNET

ftp://ftp.sas.com/pub/neural/FAQ.html (anglais)
FAQ, conseils pratiques d'implémentation de réseaux de neurones

http://www.csem.ch/detailed/pdf/e_Conference_Report_UAW_May_2000_French.pdf
http://www.sciences-ouest.org/default.asp?chaine=/contenu.asp?rub=5044
Régulations de chauffage neuro-floue : NEUROBAT et crèche intelligente

http://www.nrcan.gc.ca/es/etb/cetc/cetc01/htmldocs/factsheet_harnessing_artificial_intelligence_in_heavy_industry_f.html
Utilisation des réseaux neuronaux dans l'industrie lourde

REMERCIEMENTS

Je tiens à remercier Pierre-Yves GLORENNEC, chercheur à l'INSA de Rennes pour avoir répondu à mes questions sur son installation de chauffage neuro-floue. Je lui avais demandé pourquoi ne pas utiliser uniquement un réseau de neurones à la place de la commande floue, et voici sa réponse :

Cyril,

Je vais aborder plusieurs points.

1- D'abord, le "neuro-flou". Bien que j'ai peut-être été le premier en France à proposer un système dit "neuro-flou" (mais il y a prescription !!), je considère maintenant qu'il s'agit d'une sorte d'escroquerie intellectuelle. Un tel système est en réalité un RBF (réseau de neurones à base radiale), sur lequel on adapte des algorithmes stupides car pas du tout adaptés, du style rétropropagation. (Attention : je ne dis pas que la rétropropagation est un algorithme stupide !! C'est son utilisation pour optimiser des règles floues qui l'est). On aboutit en effet à des boîtes noires ininterprétables, ce qui est un comble quand on fait de la logique floue (qui permet d'exprimer une connaissance en langage naturel).

A l'origine, le neuro-flou était intéressant, car c'était la première méthode d'optimisation (ou apprentissage) pour des SIF (systèmes d'inférence floue). La démarche se basait sur une idée totalement fausse : les réseaux de neurones ont des capacités d'apprentissage, donc pour faire de l'apprentissage il faut passer par des réseaux de neurones...

a- les réseaux de neurones n'ont pas plus de capacités d'apprentissage qu'un modèle ARMA, un polynôme ou un SIF... Parler d'apprentissage est souvent de l'anthropomorphisme primaire. En réalité, on fait de l'optimisation paramétrique. La rétropropagation n'est que l'adaptation de la méthode du gradient pour la structure particulière des RN.

b- Tous les paramètres d'un SIF ont une signification précise, contrairement aux poids synaptiques d'un RN. De plus, il y a des relations de dépendance entre les paramètres. Il faut donc trouver des algorithmes d'optimisation
-> qui garantissent la sémantique des règles, pour qu'elles restent interprétables du début jusqu'à la fin,
-> qui tiennent compte des relations de dépendance entre les paramètres.

c- Transformer un SIF en un RN pour l'optimiser (outre que cela traduit souvent un manque de connaissances en math) impose des conditions de dérivabilité. On va donc trouver des gaussiennes comme fonctions d'appartenance et le produit comme T-norme.
-> Avec des gaussiennes, on ne peut pas avoir des partitions floues fortes.
-> Implémenter ça avec un contrôleur 8 bits et en nombres entiers est quasiment une mission impossible.

d- Tout ceci conduit à des algorithmes complexes et à de très nombreuses itérations sur l'ensemble d'apprentissage, pour aboutir à une boîte noire... A l'opposé, j'ai des algorithmes très simples, qui garantissent la sémantique, qui peuvent même fonctionner immédiatement en-ligne en partant d'une connaissance nulle, sans ensemble d'apprentissage !

2- J'utilise un SIF ou un RN selon le type de problème. La principale qualité d'un RN (non, ce n'est pas l'apprentissage !!!!!) c'est d'être un approximateur universel parcimonieux : c'est le seul modèle qui n'explose pas combinatoirement quand le nombre d'entrées augmente. Par contre, ils sont ininterprétables et (c'est une conséquence) on ne peut pas les initialiser a priori. Quand la connaissance n'est pas un critère et quand le nombre d'entrées est important, un RN est tout-à-fait indiqué. Un de mes thésards travaille sur des RN évolutifs (ceci pour dire que je ne suis pas anti-RN !!). J'utilise un SIF chaque fois que la notion de connaissance a un sens :
-> avant l'apprentissage, pour permettre une initialisation correcte et obtenir un fonctionnement initial acceptable,
-> après l'apprentissage, pour interpréter et valider la connaissance extraite.
Il faut donc que la connaissance traitée soit manipulable par un cerveau humain. S'il y a moins de 5 ou 6 entrées, j'utilise un SIF. Au delà, j'utilise un arbre de décision flou qui permet de mettre en évidence les variables les plus significatives, selon leur ordre d'importance.

3- En thermique du bâtiment, nous avons des connaissances : donc il est possible d'utiliser un SIF. Celui-ci pourra être immédiatement fonctionnel et être optimisé au fur et à mesure. Il n'aura pas besoin d'avoir une saison de chauffe comme données d'apprentissage.

A titre d'exemple, j'utilise un SIF pour prédire le niveau d'ozone du lendemain. Il y a 4 ou 5 entrées selon les villes. Un tel SIF démarre avec une connaissance nulle (de façon à pouvoir être utilisé sans données historiques). Il est fonctionnel dès le premier jour et indique, en plus, le degré de confiance qu'il attribue à sa prévision. Cet été, le SIF n'a pas prévu le premier pic d'ozone (car il n'avait jamais rencontré une telle situation). Il a donc fait une prévision très sous-estimée, mais, en même temps, il annonçait un degré de confiance très faible et affichait les règles utilisées. Comme les règles sont interprétables, il était alors facile d'intervenir et de corriger "à la main" la prévision. Evidemment, la prévision du lendemain était bien meilleure.

4- Je joins un logiciel de démo et son manuel utilisateur, plus un fichier d'apprentissage et un fichier de test, nommés respectivement m3_app et m3_tst.

J'espère avoir répondu à tes questions.

P.Y.

Il souligne donc qu'il ne faut pas abandonner les méthodes classiques car les réseaux neuronaux ne sont pas toujours plus avantageux, principalement à cause de leur côté "boîte noire", mais également qu'ils présentent l'avantage d'être les approximateurs universels les plus parcimonieux.

Je remercie également Jean-Michel JUSSIAUX, mon professeur de physique de Terminale et d'IESP de seconde au lycée Xavier Marmier à PONTARLIER, qui m'a à l'époque parlé d'utiliser la carte son de l'ordinateur pour acquérir un signal analogique, ce qui m'a particulièrement servi pour réaliser mes acquisitions de température.

ANNEXE B : CODE SOURCE DES ÉLÉMENTS CLÉS

Tout le code a été écrit en langage C++ et compilé sous Borland C++Builder 4.
J'ai choisis ce langage d'une part parce que je le connais bien, et d'autre part parce qu'il permet une interface graphique plus pratique et conviviale que CAML. Cela s'est de plus révélé indispensable pour réaliser de l'acquisition analogique, lire bit à bit un fichier sur le disque dur ou permettre un paramétrage complet des réseaux.

Je présente ici la totalité du code "algorithmique", c'est-à-dire qui ne consiste pas simplement en de la gestion d'interface.

Vous trouverez dans l'ordre l'implémentation de :

- **La reconnaissance de formes avec un réseau à une couche**
- **La régression et prévision par rétropropagation du gradient sur réseau multicouche**
- **Lecture et interprétation d'un fichier WAV (acquisition de température)**
- **L'interpréteur d'expressions arithmétiques**

RECONNAISSANCE DE FORME

Voici le code du programme de reconnaissance de formes utilisant un réseau à une couche, et un apprentissage supervisé de type règle du delta.

```
//*****************************************************************************************
// DÉCLARATIONS DIVERSES
//*****************************************************************************************

//_________________________________________________________________________________________
// Pour faciliter la lecture du programme par la suite : avant la compilation exit(42) est remplacé par ((42)+1)*(sw*sh+1)-1

#define exit(k)        ((k)+1)*(sw*sh+1)-1 ..... // donne l'indice de la sortie du neurone k
#define poids(k,i,j) (k)*(sw*sh+1)+(j)*sw+(i) ..... // du poids i,j du neurone k
#define bmp(k,i,j)     (k)*sh*sw+(j)*sw+(i) ..... // du pixel i,j de la forme k

//_________________________________________________________________________________________
// Le réseau, des variables globales, et stocker les formes

float *ANN; ..... // le réseau (Artificial Neuron Network)
int nb_shapes[2]={0,0}, sw=8, sh=8 ..... // nombre de formes (classes et test), dimensions formes
int trouve, trouve2; ..... // index des formes trouvée et arrivant en deuxième d'après la dernière reconnaissance
unsigned char *shapesL = NULL, *shapesT = NULL; ..... // tableau pour les formes (classes et test)

//_________________________________________________________________________________________
// Quelques fonctions utiles

float sqr(float x) { return x*x; } ..... // mettre au carré (square)
float sigmoid(float x) { return 1 / (1+exp(-x)); } ..... // la sigmoïde


//*****************************************************************************************
// FONCTIONS D'UTILISATION
//*****************************************************************************************

//_________________________________________________________________________________________
// REINITIALIZE : réinitialise le réseau

void reinitialize()
{
  ANN = (float*) realloc(ANN, nb_shapes[0]*(sw*sh+1)*sizeof(float)); .....//allouer la mémoire
  float d = StrToFloat(F->edIntInit->Text); ..... // les bornes de l'initialisation : entre –d et +d
  if (d < 0) d = 0; else if (d > 1) d = 1; ..... // que l'on contrôle quand même

  for(int k = 0; k < nb_shapes[0]; k++) ..... // pour chaque classe (neurone)
    for(int i = 0; i < sw * sh; i++) ..... // et chaque pixel (entrée)
      ANN[poids(k,i,0)] = (random(2000001)-1000000)/1000000. * d ..... // initialiser le poids
}

//_________________________________________________________________________________________
// RECOGNIZE : reconnaître l'indice index du tableau de formes shapes

int recognize(unsigned char *shapes, int index)
{
  int found = -1, found2 = -1; ..... // indice du neurone qui obtient le plus grand score et le deuxième plus grand
  float max = 0, max2 = 0; ..... // plus grand et second plus grand scores trouvés

  for(int k = 0; k < nb_shapes[0]; k++) ..... // pour chaque forme
  {
    float value = 0; ..... // variable temporaire lors du calcul qui contiendra la valeur de la sortie
    for(int j = 0; j < sh; j++) for(int i = 0; i < sw; i++) ..... // pour chaque pixel
      value += ANN[poids(k,i,j)]*(255-shapes[bmp(index,i,j)])/255.; ..... // ajouter contribution
    ANN[exit(k)] = sigmoid(value); ..... // puis appliquer fonction de transfert

    //--- recherche du plus grand et second plus grand .....
    if (ANN[exit(k)] > max) ..... // si ce neurone a un plus grand score que tous ceux qui précèdent
    {
      max2 = max; found2 = found; ..... // l'ancien plus grand devient le deuxième plus grand
      max = ANN[exit(k)]; found = k; ..... // et le plus grand c'est notre sortie
    } else
    if (ANN[exit(k)] > max2) ..... // si on bat juste le deuxième plus grand
    {
      max2 = ANN[exit(k)]; trouve2 = k; ..... // et bien on modifie
    }
  }
  trouve = found; trouve2 = found2; ..... // on sauve les résultats dans les variables globales pour le réutiliser
  return found; ..... // et on renvoie quand même l'indice de la forme identifiée
}

//_________________________________________________________________________________________
// LEARN : apprendre l'indice ids du tableau de formes shapes correspondant à la classe index (théoriquement)
void learn(unsigned char *shapes, int index, int ids)
{
  int found = trouve, second = trouve2; ..... // on récupère les valeurs issues de la dernière reconnaissance

  if (index != found) ..... // si s'est trompé -> correction
  {
    float erreur = (ANN[exit(found)]-ANN[exit(index)]) / 10; ..... // erreur

    for(int j = 0; j < sh; j++) for(int i = 0; i < sw; i++) ..... // pour chaque pixel (entrée, poids)
    {
      float diff = (shapesL[bmp(found,i,j)]-shapes[bmp(ids,i,j)])/255.;
            ..... // différence entre la classe reconnue en deuxième et la forme qu'on a soumis au réseau : pixels discriminants

      // on augmente le théorique
      ANN[poids(index,i,j)] += (255-shapes[bmp(ids,i,j)])/255. * erreur/3; .. // pixels activés
      ANN[poids(index,i,j)] += diff * erreur*3; ..... // et encore un petit coup pour les discriminants
      // et on diminue le trouvé
      ANN[poids(found,i,j)] -= (255-shapes[bmp(ids,i,j)])/255. * erreur; ..... // pixels activés
      ANN[poids(found,i,j)] -= diff * erreur/2; ..... // pixels discriminants
    }
  } else ..... // si a trouvé le bon -> renforcement
  {
    float erreur = (1-ANN[exit(found)]+ANN[exit(second)]) / 10; ..... // bon ok, pseudo erreur

    for(int j = 0; j < sh; j++) for(int i = 0; i < sw; i++) ..... // pour chaque pixel (entrée, poids)
    {
      float diff = (shapesL[bmp(second,i,j)]-shapes[bmp(ids,i,j)])/255.;
            ..... // différence entre la classe trouvée et la forme qu'on a soumis au réseau : pixels discriminants

      // on augmente le trouvé
      ANN[poids(found,i,j)] += (255-shapes[bmp(ids,i,j)])/255. * erreur/3; .. // pixels activés
      ANN[poids(found,i,j)] += diff * erreur*3; ..... // renforcer un peu plus les pixels discriminants
      // et on diminue le second
      ANN[poids(second,i,j)] -= (255-shapes[bmp(ids,i,j)])/255. * erreur; ..... // pixels activés
      ANN[poids(second,i,j)] -= diff * erreur/2; ..... // et encore pour les pixels discriminants
    }
  }
}
```

RÉSEAUX NON BOUCLÉS ET RÉTROPROPAGATION DU GRADIENT

Voici le code complet de la modélisation d'un réseau multicouches, de son utilisation ainsi que de son apprentissage par la méthode de rétropropagation du gradient et diverses améliorations, telles le gradient stochastique, avec inertie etc …

```
//*****************************************************************************************************
// DÉCLARATIONS
//*****************************************************************************************************

//_____________________________________________________________________________________________________
// TANN : le réseau multicouches avec apprentissage
class TANN
{
 //*** déclarations privées (accessibles seulement aux autres
 private:
  //--- définitions ..................................................................................
  struct TNeuron
  {
    double V, S;  ..... // potentiel et sortie
    double *W, b, *D;  ..... // poids, seuil et poids connections directes entrées
    double *dWt, dbt, *dDt;  ..... // variable temporaire pour stocker la correction calculée
    double *dWp, dbp, *dDp;  ..... // correction due à l'apprentissage précédent
    double delta;  ..... // delta de l'apprentissage rétropropagation du gradient
  };
  struct TLayer
  {
    TNeuron *neuron;
    int m;  ..... // nb neurones dans couche
    int typ_f; double par_f;  ..... // type fonction de transfert, paramètre eventuel
    bool seuils, connection_e;  ..... // seuils?, connection aux entrées du réseau?
    double eta, alpha;  ..... // coefficients d'apprentissage
  };

  //--- données ......................................................................................
  TLayer *layer;
  int n;  ..... // nb de couches
  double *T;  ..... // vecteur theorique de sortie
  //--- fonctions transfert..........................................................................
  double ft(int t, double a, double x);  ..... // fonctions de transfert : t = type, a = coeff, x = x
  double dft(int t, double a, double x);  ..... //  dérivées

  //--- notation indicielle = notation démonstrations ................................................
  int q, r;  ..... // dimensions des vecteurs d'entrée et sortie
  int m(int i) { return layer[i].m; }  ..... // nb de neurones des couches
  // niveaux
  double &S(int i, int j) { return layer[i].neuron[j].S; }  ..... // sorties des neurones
  double &I(int j       ) { return layer[0].neuron[j].S; }  ..... // entrées du réseau
  double &O(int j       ) { return layer[n].neuron[j].S; }  ..... // sorties du réseau
  double &V(int i, int j) { return layer[i].neuron[j].V; }  ..... // potentiels des neurones
  // poids
  double &W(int i, int j, int k) { return layer[i].neuron[j].W[k]; }  .// poids entrées neurones
  double &D(int i, int j, int k) { return layer[i].neuron[j].D[k]; }  ..... // connections entrées
  double &b(int i, int j)        { return layer[i].neuron[j].b; }  ..... // seuils neurones
  bool  seuil(int i) { return layer[i].seuils; }  ..... // neurones couche ont seuil ?
  bool  connexionE(int i) { return layer[i].connection_e; }  ..... // neurones couche reliés entrées ?
  double &delta(int i, int j) {return layer[i].neuron[j].delta;}  ..... // delta des neurones
  double f(int i, double x) {return  ft(layer[i].typ_f,layer[i].par_f,x);}  ..... // f transfert
  double df(int i, double x) {return dft(layer[i].typ_f,layer[i].par_f,x);}  ..... // dérivées

  // stockage variations poids temporaire et précédent
  double &dWt(int i, int j, int k) { return layer[i].neuron[j].dWt[k]; }
  double &dDt(int i, int j, int k) { return layer[i].neuron[j].dDt[k]; }
  double &dbt(int i, int j) { return layer[i].neuron[j].dbt; }
  double &dWp(int i, int j, int k) { return layer[i].neuron[j].dWp[k]; }
  double &dDp(int i, int j, int k) { return layer[i].neuron[j].dDp[k]; }
  double &dbp(int i, int j) { return layer[i].neuron[j].dbp; }

 //*** déclarations publiques (accessibles pour l'utilisation)
 public:
  TANN(int n_couches, int n_neurones[], bool connection_e);  ..... // constructeur du réseau
  void set_ftransfert(int couche, int type, double param = 1);  ..... // définit f transfert de la couche
  void set_poids(int couche, bool seuils=false,
                 double initMax=1, double initMin=0);  ..... // initialise les poids de la couche
  ~TANN  ..... // destructeur (libère mémoire)

  double evaluate_out();  ..... // évalue la sortie, renvoie l'erreur
  void learn_gradient();  ..... // apprend l'exemple
  void learning_apply(int count, bool inertie, bool autom,
                      double ETA = -1, double ALPHA = -1);  ..... // applique la correction
  void learning_reset();  ..... // réinitialise la correction
  void learn_user();  ..... // affiche une fenêtre pour apprentissage manuel
  void bruiter_poids(double pourcent);  ..... // bruite les poids de pourcent %

  // notation indicielle
  double &eta(int i) { return layer[i].eta; }
  double &alpha(int i) { return layer[i].alpha; }
  double &vecteur_e (int j) {return layer[0].neuron[j<q?(j>=0?j:0):(q-1)].S;}  ..... // entrée
  double  vecteur_s (int j) {return layer[n].neuron[j<r?(j>=0?j:0):(r-1)].S;}  ..... // sortie
  double &vecteur_th(int j) {return T[j<r?(j>=0?j:0):(r-1)];}  ..... // vecteur théorique

  int const dim_e, dim_s, n_couches;  ..... // dimensions vecteurs entrée et sortie, nb couches
  void trace(TStringGrid *sg);  ..... // affiche dans la grille sg l'état interne du réseau
  void permute(int *tableau, int taille);  ..... // permute aléatoirement les éléments du tableau
  double randb(double borne);  ..... // renvoie un flottant aléatoire entre -borne et +borne
  double randi(double borneMin, double borneMax);
};

enum {binary = 0, sym_binary, linear, sat_linear, sym_sat_linear,
      sigmoid, sym_sigmoid, gaussian, stochastic, arctan, identity};

double sqr(double x) { return x*x; }
double abs(double x) { return x>=0?x:-x; }


//*****************************************************************************************************
// FONCTIONS DE TRANSFERT
//*****************************************************************************************************

//_____________________________________________________________________________________________________
// FT : fonctions de transfert
double TANN::ft(int t, double a, double x)
{
  switch (t)
  {
    //--- seuils
```

```
    case binary: return x>=0;                    // binaire 0 ou 1
    case sym_binary: return (x>=0)*2-1;          // binaire -1 ou 1
    //--- linéaires
    case linear: return a*x;                     // linéaire
    case sat_linear: return a*x<0?0:(a*x>1?1:a*x);   // linéaire saturé à 0 et 1
    case sym_sat_linear: return a*x<-1?-1:(a*x>1?1:a*x);   // linéaire saturé à -1 et 1
    //--- sigmoides
    case sigmoid:                                // sigmoïde de 0 à 1
    { double lim = 46./a;                        // empeche overflow
      return x>lim?1:(x<-lim?0:1/(1+exp(-a*x)));
    }
    case sym_sigmoid:                            // sigmoïde de -1 à 1
    { double lim = 46./a;                        // empeche overflow
      return x>lim?1:(x<-lim?-1:2/(1+exp(-a*x))-1);
    }
    //--- divers
    case gaussian:                               // gaussienne
    { double lim = sqrt(46*a);                   // empeche overflow
      return ((x>lim)||(x<-lim))?0:exp(-x*x/a);
    }
    case stochastic: return (random(1+exp(-x/a)) == 0);   // stochastique : a = température
    case arctan: return atan(x);                 // arc tangente
    case identity: return x;                     // identité
    default: return 0;
  }
}

//-------------------------------------------------------------------------------------------
// DFT : dérivées des fonctions de transfert
double TANN::dft(int t, double a, double x)
{
  switch (t)
  {
    //--- seuils
    case binary: return 0;
    case sym_binary: return 0;
    //--- linéaires
    case linear: return a;
    case sat_linear: return a*x<0?0:(a*x>1?0:a);
    case sym_sat_linear: return a*x<-1?0:(a*x>1?0:a);
    //--- sigmoides
    case sigmoid:
    { double lim = 46./a;
      return ((x>lim)||(x<-lim))?0:a/(exp(-a*x)+exp(a*x)+2);
    }
    case sym_sigmoid:
    { double lim = 46./a;
      return ((x>lim)||(x<-lim))?0:a/(exp(-a*x)+exp(a*x)+2)*2;
    }
    //--- divers
    case gaussian:
    { double lim = sqrt(46*a);
      return ((x>lim)||(x<-lim))?0:-2*x*exp(-x*x/a)/a;
    }
    case stochastic: return 0;
    case arctan: return 1/(1+x*x);
    case identity: return 1;
    default: return 0;
  }
}


//*******************************************************************************************
// CRÉATION DU RÉSEAU
//*******************************************************************************************

//-------------------------------------------------------------------------------------------
// TANN : constructeur
TANN::TANN(int n_couches, int n_neurones[], bool connection_e)
 : dim_e(q), dim_s(r), n_couches(n_couches)
{
  layer = new TLayer[n_couches+1];               // créer les couches
  n = n_couches; layer[0].m = n_neurones[0];     // nb de couches et nb d'entrées
  layer[0].neuron = new TNeuron[n_neurones[0]];  // neurones d'entrée
  T = new double[n_neurones[n_couches]];         // vecteur théorique de sortie

  for(int i = 1; i <= n_couches; i++) // pour chaque autre couche
  {
    layer[i].connection_e = false;               // seule la dernière couche peut être connectée aux entrées
    layer[i].m = n_neurones[i];                  // nombre de neurones de la couche
    layer[i].typ_f = 0; layer[i].par_f = 1;      // type et paramètre fonction par défaut
    layer[i].eta = 0.3; layer[i].alpha = 0;      // coeffs d'apprentissage par défaut
    layer[i].seuils = false;                     // toujours par défaut
    layer[i].neuron = new TNeuron[n_neurones[i]];   // on crée les neurones de la couche

    for(int j = 0; j < n_neurones[i]; j++)       // pour chaque nurone
    {
      //--- les poids-----------------------------------------------------------------------
      layer[i].neuron[j].W = new double[n_neurones[i-1]];   // créer
      layer[i].neuron[j].dWt = new double[n_neurones[i-1]];   "
      layer[i].neuron[j].dWp = new double[n_neurones[i-1]];   "
      for(int k = 0; k < m(i-1); k++)            // initialiser par défaut
      {
        W(i,j,k) = randb(0.1);
        dWt(i,j,k) = 0; dWp(i,j,k) = 0;
      }
      //--- le seuil -----------------------------------------------------------------------
      b(i,j) = 0;
      dbt(i,j) = 0; dbp(i,j) = 0;
      //--- les poids des connexions aux entrées-------------------------------------------
      if (i == n_couches && connection_e)
      {
        layer[i].connection_e = true;            // on dit que la couche est connectée aux entrées
        layer[i].neuron[j].D = new double[q];    // créer
        layer[i].neuron[j].dDt = new double[q];   "
        layer[i].neuron[j].dDp = new double[q];   "
        for(int k = 0; k < m(0); k++)            // initialiser par défaut
        {
          D(i,j,k) = randb(0.1);
          dDt(i,j,k) = 0, dDp(i,j,k) = 0;
        }
      } else                                     // sinon ne servent à rien
      {
        layer[i].neuron[j].D = NULL;
        layer[i].neuron[j].dDt = NULL;
        layer[i].neuron[j].dDp = NULL;
      }
    }
  }
  q = layer[0].m; r = layer[n_couches].m;        // nombre d'entrées et de sorties
}
```

```
//-------------------------------------------------------------------------------------------
// ~TANN : destructeur
TANN::~TANN()
{
  for(int i = 0; i <= n; i++) ..... // pour chaque couche
  {
    for(int j = 1; j < m(i); j++) ..... // pour chaque neurone
    {
      if (layer[i].neuron[j].W) delete layer[i].neuron[j].W; ..... // on libère si pas NULL
      if (layer[i].neuron[j].D) delete layer[i].neuron[j].D; ..... "
      if (layer[i].neuron[j].dWt) delete layer[i].neuron[j].dWt; ..... "
      if (layer[i].neuron[j].dDt) delete layer[i].neuron[j].dDt; ..... "
      if (layer[i].neuron[j].dWp) delete layer[i].neuron[j].dWp; ..... "
      if (layer[i].neuron[j].dDp) delete layer[i].neuron[j].dDp; ..... "
    }
    if (layer[i].neuron) delete layer[i].neuron; ..... // et la couche elle-même
  }
  if (layer) delete layer; ..... // et le tableau de couches
  if (T) delete T; ..... // et le vecteur théorique
}


//-------------------------------------------------------------------------------------------
// SET_FTRANSFERT : initialisation des fonctions de transfert
void TANN::set_ftransfert(int couche, int type, double param)
{
  couche = couche<n?(couche>=0?couche:0):n-1; ..... // on vérifie que la couche existe
  layer[couche+1].typ_f = type; ..... // on affecte le type
  layer[couche+1].par_f = param; ..... // et le paramètre
}

//-------------------------------------------------------------------------------------------
// SET_POIDS : initialisation aléatoire des poids
void TANN::set_poids(int couche, bool seuils, double initMax, double initMin)
{
  couche = couche<n?(couche>=0?couche:0):n-1; ..... // on vérifie que la couche existe
  layer[couche+1].seuils = seuils; ..... // seuils ?
  for(int j=0; j < m(couche+1); j++) ..... // pour chaque neurone
  {
    if (seuils) b(couche+1,j) = randi(initMin,initMax);
      else b(couche+1,j) = 0; ..... // initialier seuils
    for(int k=0; k < m(couche); k++)
      W(couche+1,j,k) = randi(initMin,initMax); ..... // et poids avec initMin < | | < initMax
  }
}


//*******************************************************************************************
// UTILISATION DU RÉSEAU
//*******************************************************************************************

//-------------------------------------------------------------------------------------------
// EVALUATE_OUT : évalue la sortie du réseau d'après les entrées déjà écrites dans le réseau
double TANN::evaluate_out()
{
  //--- calculer sortie
  for(int i=1; i <= n; i++) ..... // pour chaque couche
    for(int j=0; j < m(i); j++) ..... // pour chaque neurone
    {
      V(i,j) = b(i,j); ..... // déjà le seuil
      for(int k=0; k < m(i-1); k++) ..... // puis pour chaque poids
        V(i,j) += S(i-1,k) * W(i,j,k); ..... // les autres entrées pondérées du neurone
      S(i,j) = f(i, V(i,j)); ..... // et enfin fonction de transfert
    }
  //--- calculer l'erreur (coût des moindres carrées)
  double erreur = 0;
  for(int j=0; j < r; j++) erreur += abs(T[j] - O(j)); ..... // ajouter l'erreur de chaque sortie du réseau
  return erreur/r; ..... // on renvoie une erreur moyenne
}

//-------------------------------------------------------------------------------------------
// LEARN_GRADIENT : calcule la correction pour l'exemple écrit dans le réseau et le stocke dans dWt
// sans les coeffs d'apprentissage qui seront appliqués après
void TANN::learn_gradient()
{
  evaluate_out(); ..... // on commence par évaluer pour avoir le bon état interne
  //--- on commence par la couche de sortie
  for(int j = 0; j < r; j++) ..... // pour chaque neurone de la couche
  {
    delta(n,j) = (T[j] - O(j)) * df(n, V(n,j)); ..... // calcul du delta

    for(int k = 0; k < m(n-1); k++) ..... // pour chaque poids
      dWt(n,j,k) += delta(n,j) * S(n-1,k); // on calcule la correction que l'on stocke dans dWt sans l'appliquer
    dbt(n,j) += delta(n,j) * 0.1; ..... // de meme pour le seuil
  }
  //--- puis on rétropropage aux autres couches
  for(int i = n-1; i > 0; i--) ..... // pour chaque couche
    for(int j = 0; j < m(i); j++) ..... // pour chaque neurone de la couche
    {
      //--- calcul du delta
      delta(i,j) = 0;
      for(int l = 0; l < m(i+1); l++)
        delta(i,j) += delta(i+1,l) * W(i+1,l,j);
      delta(i,j) *= df(i, V(i,j));

      //--- puis pour chaque poids on calcule la correction
      for(int k = 0; k < m(i-1); k++)
        dWt(i,j,k) += delta(i,j) * S(i-1,k);
      dbt(i,j) += delta(i,j) * 0.1; // le seuil aussi
    }
}

//-------------------------------------------------------------------------------------------
// LEARNING_RESET : remet à 0 les corrections temporaires
void TANN::learning_reset()
{
  for(int i=1; i <= n; i++) ..... // pour chaque couche
    for(int j=0; j < m(i); j++) ..... // pour chaque neurone
    {
      for(int k=0; k < m(i-1); k++) dWt(i,j,k) = 0; ..... // pour chaque poids

      if (connexionE(i)) for(int k=0; k < m(0); k++) dDt(i,j,k) = 0; ..... // pour chaque entrée
      dbt(i,j) = 0; ..... // pour le seuil
    }
}
```

```
//-------------------------------------------------------------------------
// LEARNING_APPLY : applique les correction temporaires, en rajoutant l'inertie si nécessaire,
// et en calculant les coeffs si automatique
void TANN::learning_apply(int count, bool inertie, bool autom, double ETA, double
ALPHA)
{
  if (ETA != -1) for(int i = 0; i <= n; i++) eta(i) = ETA; .....// si meme coeff pour chaque couche
  if (ALPHA != -1) for(int i = 0; i <= n; i++) alpha(i) = ALPHA; ..... "
  for(int i = 1; i <= n; i++) ..... // pour chaque couche
  {
    //--- calculer correl de la couche avec coeffs actuels.....
    float correl = 0;
    for(int j = 0; j < m(i); j++) ..... // pour chaque neurone
    {
      for(int k = 0; k < m(i-1); k++) ..... // les poids
    correl += (dWt(i,j,k)*eta(i)/count + (inertie?alpha(i)*dWp(i,j,k):0))*dWp(i,j,k);
      if (connexionE(i)) for(int k = 0; k < m(0); k++) ..... // les poids connexion entrée
        correl += (dDt(i,j,k)*eta(i)/count + (inertie?alpha(i)*dDp(i,j,k):0)) *
dDp(i,j,k);
      if (seuil(i)) ..... // le seuil
        correl += (dbt(i,j) * eta(i)/count + (inertie?alpha(i)*dbp(i,j):0)) *
dbp(i,j);
    }

    //--- ajuster coeffs d'apprentissage de la couche si oscillation .....
    if (inertie && autom) if (correl < 0)
    {
      eta(i) = 0.1; // on corrige
      alpha(i) = 0;
    }

    //--- appliquer la correction avec les (éventuellement nouveaux) poids.....
    for(int j = 0; j < m(i); j++) ..... // pour chaque neurone
    {
      //--- poids
      for(int k = 0; k < m(i-1); k++) ..... // pour chaque poids
      {
        dWt(i,j,k) *= eta(i)/count; ..... // on applique et on fait une moyenne des corrections
        if (inertie) dWt(i,j,k) += alpha(i)*dWp(i,j,k); ..... // éventuellement on ajoute inertie
        W(i,j,k) += dWt(i,j,k); ..... // on applique la correction
        dWp(i,j,k) = dWt(i,j,k); ..... // et on stocke la correction pour le suivant (pour inertie)
      }
      //--- poids connexions entrées
      if (connexionE(i)) for(int k = 0; k < m(0); k++) ..... // pour chaque entrée
      {
        dDt(i,j,k) *= eta(i)/count; ..... // on applique et on fait une moyenne des corrections
        if (inertie) dDt(i,j,k) += alpha(i)*dDp(i,j,k); ..... // éventuellement on ajoute inertie
        D(i,j,k) += dDt(i,j,k); ..... // on applique la correction
        dDp(i,j,k) = dDt(i,j,k); ..... // et on stocke la correction pour le suivant (pour inertie)
      }
      //--- seuil
      if (seuil(i))
      {
        dbt(i,j) *= eta(i)/count; ..... // on applique et on fait une moyenne des corrections
        if (inertie) dbt(i,j) += alpha(i)*dbp(i,j); ..... // éventuellement on ajoute inertie
        b(i,j) += dbt(i,j); ..... // on applique la correction
        dbp(i,j) = dbt(i,j); ..... // et on stocke la correction pour le suivant (pour inertie)
      }
    }

    // ajuster coeffs d'apprentissage si pas oscillation
    if (inertie && autom) if (correl >= 0)
    {
      eta(i) += 0.01;
      alpha(i) += 0.01;
    }
  }
}


//-------------------------------------------------------------------------
// BRUITER_POIDS : bruite les poids
void TANN::bruiter_poids(double pourcent)
{
  for(int i = 1; i <= n; i++) // pour chaque couche
    for(int j = 0; j < m(i); j++) // pour chaque neurone
      for(int k = 0; k < m(i-1); k++) // pour chaque poids (entrée)
        W(i,j,k) *= 1 + randb(pourcent); // on bruite de pourcent %
}


//*************************************************************************************************
// PETITES FONCTIONS UTILES
//*************************************************************************************************

//-------------------------------------------------------------------------
// RANDB : renvoie un flottant aléatoire entre -borne et +borne
double TANN::randb(double borne)
{
  return (2.*(random(1000001)/1000000.)-1.)*borne;
}

//-------------------------------------------------------------------------
// RANDI : renvoie un flottant aléatoire de val abs comprise entre borneMin et borneMax
double TANN::randi(double borneMin, double borneMax)
{
  double rap = borneMin / borneMax;
  rap = ((random((int)(1000001*(1-rap)))+rap*1000000)/1000000.);
  if (random(100)%2) rap = -rap;
  return rap*borneMax;
}


//-------------------------------------------------------------------------
// PERMUTE : permute aléatoirement les éléments d'un tableau
void TANN::permute(int *tableau, int taille)
{
  for(int i = 0; i < taille; i++)
  {
    int index = random(taille-i);
    int tmp = tableau[taille-i-1]; tableau[taille-i-1] = tableau[index];
tableau[index] = tmp;
  }
}
```

LECTURE ET INTERPRÉTATION D'UN FICHIER WAV

Le plus intéressant est ici l'ouverture et l'interprétation d'un fichier .WAV. La structure du format a été trouvé sur Internet, il suffit ensuite de parcourir convenablement le fichier.

```
//---------------------------------------------------------------------------
// COMPARE : compare deux tableaux de caractères sur la longueur nb
bool compare(char *data1, char *data2, int nb)
{
  for(int i=0; i < nb; i++) if (data1[i] != data2[i]) return false;
  return true;
}

//---------------------------------------------------------------------------
// OCTETSTOINT : convertit au plus 4 octets dans un tableau en entier (format intel : avec LSB (octet poids faible) en tête)
const int power256[4] = {1, 256, 65536, 16777216};
int octetsToInt(unsigned char *data, int nb)
{
  if (nb > 4) return 0;
  int res = 0;
  for(int i=0; i < nb; i++) res += data[i]*power256[i]
  return res;
}

//---------------------------------------------------------------------------
// WAV_FILE : structure contenant les infos sur un fichier wave
struct wave_file
{
  int nb_canaux, f_echant, profondeur, duree; // variables minimales de description

  AnsiString nom_fichier; // variables pour l'ouverture du fichier
  int handle_fichier; //"
  int nb_echant, taille_fichier, taille_donnees, niveaux; //"

  short voie1[10000], voie2[10000]; // pour stocker le fichier
  wave_file() { nb_canaux=2; f_echant=5000; profondeur=16; duree=2000; } // constructeur
} wav; // on en déclare un

//---------------------------------------------------------------------------
// OUVRIR_WAV : ouvre un fichier wave
void ouvrir_wav(wave_file &wav)
{
  class Format{}, Taille{}, Supporte{}, Occupe{}, // déclaration des types "erreur"

  const TaillePasse = 4096; // nb d'octets que l'on lit à la fois (= une passe)
  int nbPasse; // nb de passes nécessaires pour lire le fichier
  unsigned char buf[TaillePasse]; // créer le buffer (= tampon = tableau temporaire)


  try { try
  {
    wav.handle_fichier = FileOpen(wave.nom_fichier, fmShareDenyWrite); // ouvrir le fichier
    wav.taille_fichier = FileSeek(wave.handle_fichier, 0, 2); // stocker taille fichier
    if (wave.taille_fichier == -1) throw Occupe(); // c'est qu'il y a eu une erreur

    FileSeek(wave.handle_fichier, 0, 0); // positionner pointeur sur le début du fichier
    FileRead(wave.handle_fichier, buf, 20); // lire 1° chunk (infos sur le fichier) + début 2° (20 octets)

    if (!compare(buf, "RIFF", 4)) throw Format();
    if ((octetsToInt(buf+4, 4) + 8) != wav.taille_fichier) throw Taille();
    if (!compare(buf+8, "WAVE", 4)) throw Format();
    if (!compare(buf+12, "fmt ", 4 throw Format();

    int Chunk = octetsToInt(buf+16, 4); // on récupère la taille du 2° chunk
    FileRead(wave.handle_fichier, buf, Chunk + 8); // lire fin 2° chunk (format) + début 3° chunk
    wave.nb_canaux = octetsToInt(buf+2, 2); // lire le nombre de canaux utilisés
    if (wave.nb_canaux != 2) throw Supporte(); // si c'était pas pour mon capteur on supporterait aussi 1
    wave.f_echant = octetsToInt(buf+4, 4); // fréquence d'échantillonnage utilisée
    wave.profondeur = octetsToInt(buf+14, 2); // profondeur : échantillonnage 8 bits ou 16 bits ?
    if (wave.profondeur != 16) throw Supporte(); // idem on pourrait supporter 8 …
    wave.niveaux = 65536; // et avec 8 ça donnerait 256
    int Debut = Chunk + 28;

    //--- on cherche le chunk de données "data" (peut y avoir "fact" et peut-être d'autres)
    while (!compare(buf+chunk, "data", 4))
    {
      chunk = octetsToInt(buf+chunk+4, 4);
      FileSeek(wave.handle_fichier, Debut, 0);
      FileRead(wave.handle_fichier, buf, chunk+8);
      debut += chunk+8;
    }

    wave.taille_donnees = octetsToInt(buf+chunk+4, 4);
    chunk = debut;

    wave.nb_echant = wave.taille_donnees / (wave.profondeur/8 * wave.nb_canaux);

    //--- lecture des données
    nbPasse = wave.taille_donnees / TaillePasse;
    if (wave.taille_donnees % TaillePasse != 0) nbPasse++;

    for(int i = 0; i < nbPasse; i++)
    {
      int count = TaillePasse;
      if (i == nbPasse-1) count = wave.taille_donnees % TaillePasse;

      FileSeek(wave.handle_fichier, chunk + i*TaillePasse, 0); // positionner pointeur
      FileRead(wave.handle_fichier, buf, count); // lire une passe

      short *value;
      if ((wave.profondeur == 16) && (wave.nb_canaux == 2))
        for(int j=0; j < count/4; j++)
        {
          value = (short*) &buf[4*j];
          wave.voie1[i*(TaillePasse/4)+j] = *value;
          value = (short*) &buf[4*j+2];
          wave.voie2[i*(TaillePasse/4)+j] = *value;
        }
    }
    //-------------

  } __finally
  {
    FileClose(wave.handle_fichier); // fermer le fichier
    delete buf; // effacer le buf
  }}
  catch (Supporte) { ShowMessage("Le format wav de ce fichier n'est pas supporté"); }
  catch (Taille)   { ShowMessage("Ce fichier est endommagé"); }
  catch (Format)   { ShowMessage("Ce fichier n'est pas du type .wav"); }
  catch (Occupe) { ShowMessage("Ce fichier est utilisé par une autre application"); }
  catch (...)      { ShowMessage("Il s'est produit une erreur"); }
}
```

INTERPRÉTEUR D'EXPRESSIONS ARITHMÉTIQUES

Il m'a fallu ici d'abord recréer la structure de liste chaînée, comme en CAML, pour pouvoir créer un arbre associé à l'expression de manière totalement récursive.

```
//*****************************************************************************************************
// CONSTANTES
//*****************************************************************************************************

static char dec = (FormatFloat("0.0",0))[2]; ..... // récupérer le séparateur décimal (. ou , ?)
const nbin = 5, nun = 18, ncte = 2; ..... // nombre d'opérateurs binaires, unaires, et de constantes
const AnsiString symboles_bin[nbin] = { "-", "+", "*", "/", "^" }; ..... // opérateurs binaires
const AnsiString symboles_un[nun] = { "-", "ln", "log", "exp", "cos", "sin", "tan",
        "acos", "asin", "atan", "sqrt", "abs", "cosh", "sinh", "tanh",
        "round", "random", "fact" }; ..... // opérateurs unaires
const AnsiString symboles_cte[ncte] = { "pi", "e" }; ..... // noms des constantes
const double values_cte[ncte] = { 3.1415926535897, 2.718281828459 }; .// valeurs des constantes

//_____________________________________________________________________________________________________
// CODE_OP : renvoie le code opératoire associé au nom d'opérateur donné (chaîne)
int code_op(AnsiString lex)
{
  for(int i = 0; i < nbin; i++) if (symboles_bin[i] == lex) return i;
  for(int i = 0; i < nun; i++) if (symboles_un[i] == lex) return 128+i;
  return -1; ..... // c'est une constante ou une variable
}

//_____________________________________________________________________________________________________
// CONSTANTES : renvoie la valeur de la constante
double constantes(AnsiString lex)
{
  for(int i = 0; i < ncte; i++) if (symboles_cte[i] == lex) return values_cte[i];
  return 0; ..... // si ce n'est pas une constante
}

//_____________________________________________________________________________________________________
// divers
int round(double x) {return ((x-(int)x)>=0.5?(int)x:(int)x+1);} ...// arrondi entier le plus proche
int fact(double x) { int y = (int) x; return (y<=0?1:y*fact(y-1)); } ..... // factorielle
bool IsFloat(AnsiString s) ..... // est-ce que la chaîne représente un nombre en virgule flottante ?
  { try {StrToFloat(s); return true;} catch(...) {return false;} }


//*****************************************************************************************************
// CLASSES
//*****************************************************************************************************

//_____________________________________________________________________________________________________
// STRING_LIST : liste chaînée contenant des chaînes de caractères
struct string_list
{
  AnsiString tete; ..... // tête de la liste
  string_list *queue; ..... // pointeur vers la queue (une autre string_list)
  bool par; ..... // ajouter des parenthèses ?

  string_list() { tete=""; queue=NULL; par=false; } ..... // constructeur
  ~string_list() { if (queue!=NULL) delete queue; tete=""; } ..... // destructeur
};

//_____________________________________________________________________________________________________
// NOEUD : noeud d'un arbre représentant une expression arithmétique
class noeud
{
  int op; ..... // le code opérateur
  noeud *opG, *opD; ..... // pointeurs vers les opérandes gauches et droites
  double valeur; ..... // valeur du noeud

  int extract_elements(AnsiString expr, string_list **res); // décomposer en liste d'éléments simples
  bool cherche(string_list *elts, int idebut, int ifin, arbre *proprio); .....//

 public:
  noeud(AnsiString expr, arbre *proprio); ..... // constructeur
  ~noeud() { if (opG!=NULL) delete opG; if (opD!=NULL) delete opG; } ..... // destructeur

  double eval(arbre *proprio); ..... // évalue la valeur du noeud
  AnsiString print(arbre *proprio); ..... // renvoie l'expression interprétée du noeud (comprise)
};


//_____________________________________________________________________________________________________
// ARBRE : arbre représentant une expression arithmétique
class arbre
{
  int next_lexeme(AnsiString &expr, int l, int &c); .....// déplace le curseur c vers le lexeme suivant

 public:
  bool verifie(AnsiString &expr); ..... // vérifie que l'expression est syntaxiquement correcte

  noeud *racine; ..... // la racine de l'arbre
  AnsiString expression; ..... // l'expression telle qu'elle lui a été donnée

  //--- variables
  int nvar; ..... // nombre de variables utilisées dans l'expression
  TStringList *var_symboles; ..... // et leurs symboles (noms)
  double *var; ..... // et leurs valeurs

  //--- méthodes
  arbre(AnsiString expr) ..... // constructeur
    { if (verifie(expr.LowerCase())) var_symboles = new TStringList();
      nvar=0; racine = new noeud(expression, this);}
  ~arbre()..... // destructeur
    { if (racine!=NULL) delete racine; if (var!=NULL) delete var;
      if (var_symboles!=NULL) delete var_symboles;}

  double eval() { return racine->eval(this); } ..... // évalue la valeur de l'expression
  AnsiString print() { return racine->print(this); } ..... // renvoie l'expression interprétée (chaîne)
};


//*****************************************************************************************************
// MÉTHODES
//*****************************************************************************************************

//_____________________________________________________________________________________________________
// EXTRACT_ELEMENTS : crée une liste des éléments de l'expression expr.  Un élément est un opérateur, une valeur, une
variable ou une expression entre parenthèses.  **res est un pointeur sur le pointeur de la string_list, pour pouvoir l'allouer
int noeud::extract_elements(AnsiString expr, string_list **res)
{
  int l = expr.Length(); ..... // longueur de l'expression (nombre de caractères)
  int i; for(i = l; (i>0) && (expr[i] == ' '); i--); .....// on passe tous les espaces éventuels à la fin
  if (i <= 0) { *res = NULL; return 0; } ..... // s'il n'y a que des espaces ou rien on arrête
```

```
        else (*res) = new string_list(); ..... // sinon on continue
  //--- si c'est une expression entre parenthèses
  if (expr[i] == ')')
  {
    int count = 0; ..... // niveau de parenthésage
    for(int j=i; j > 0; j--)
    {
      if (expr[j] == ')') count++; else if (expr[j] == '(') count--; ..... // mettre à jour
      if (count <= 0) ..... // c'est qu'on arrivé à l'ouvrante dont on avait la fermante
      {
        (*res)->tete = expr.SubString(j+1,i-j-1); (*res)->par=true; ..... // ça fait une expression
        return 1 + extract_elements(expr.SubString(1,j-1), &((*res)->queue));
      }
    }
  } else
  //--- si c'est un opérateur symbole
  if (expr[i]=='*' || expr[i]=='/' || expr[i]=='-' || expr[i]=='+' || expr[i]=='^')
  {
    (*res)->tete = expr.SubString(i,1);
    return 1 + extract_elements(expr.SubString(1,i-1), &((*res)->queue));
  } else
  //--- si c'est un opérateur texte ou une variable ou un nombre
  {
    int j; for(j=i; (j>0) && expr[j]!=' ' && expr[j]!='*' && expr[j]!='/' &&
                             expr[j]!='-' && expr[j]!='+' && expr[j]!='^' &&
                             expr[j]!='(' && expr[j]!=')'                    ; j--);
    (*res)->tete = expr.SubString(j+1, i-j);
    return 1 + extract_elements(expr.SubString(1,j), &((*res)->queue));
  }
  return -1;
}

//-------------------------------------------------------------------------------------------
// CHERCHE : cherche dans la liste d'elements le premier opérateur dont le code est compris entre idebut et ifin et l'exécute
bool noeud::cherche(string_list *elts, int idebut, int ifin, arbre *proprio)
{
  // on cherche
  string_list *tmp = elts->queue;
  int c;
  while (tmp != NULL) ..... // tant qu'il y a une queue
  {
    c = code_op(tmp->tete);
    if (c<idebut || c>ifin) tmp = tmp->queue; else break;
  }
  // on en a trouvé un
  if (tmp != NULL) ..... // c'est qu'on s'est arrêté parce qu'on en a trouvé un
  {
    //--- opérande droite
    AnsiString droite = "";
    tmp = elts;
    do
    {
      droite = (tmp->par?"(":" ") + tmp->tete + (tmp->par?")":" ") + droite;
      tmp = tmp->queue;
      c = code_op(tmp->tete);
    } while (c<idebut || c>ifin);
    opD = new noeud(droite, proprio);
    //--- opérateur
    op = c;
    //--- opérande gauche
    AnsiString gauche = "";
    tmp = tmp->queue;
    while (tmp != NULL)
    {
      gauche = (tmp->par?"(":" ") + tmp->tete + (tmp->par?")":" ") + gauche;
      tmp = tmp->queue;
    }
    opG = new noeud(gauche, proprio);
    //--- pour transformer 0-1 en -1 unaire
    if ((c == 0) && (opG->valeur == 0) && (opG->op == -1))
      { delete opG; op = 128; }
    return true;
  } else return false;
}

//-------------------------------------------------------------------------------------------
// NŒUD : constructeur du noeud
noeud::noeud(AnsiString expr, arbre *proprio)
{
 debut: ..... // étiquette
  opG = NULL; opD = NULL;
  string_list *elts;
  int nelt = extract_elements(expr, &elts); ..... // extrait les élements dans elts et en stocke le nombre

  //--- si pas d'éléments on renvoie 0 (utile pour - unaire : (-1) = (0-1))
  if (nelt == 0) { valeur = 0; op = -1; } else
  //--- si un seul élément
  if (nelt == 1)
  {
    // possibilité1 : c'est un nombre
    if (IsFloat(elts->tete)) { op=-1; valeur = StrToFloat(elts->tete); } else
    {
      // possibilité2 : c'est une constante
      int l = elts->tete.Length();
      int i; for(i = 1; (i <= l) && (elts->tete[i]!=' ') && (expr[i]!='*') &&
                        (expr[i]!='/') && (expr[i]!='-') && (expr[i]!='+') &&
                        (expr[i]!='^') && (expr[i]!='(') && (expr[i]!=')'); i++);
      if (i == l+1)
      {
        valeur = constantes(elts->tete);
        if (valeur != 0) op = -1; else
        {
          // possibilité3 : c'est une variable
          op = -2;
          for(op = -2; (op > -proprio->nvar-2) &&
                       (proprio->var_symboles->Strings[-op-2] != elts->tete); op--);
          if (op == -proprio->nvar-2) // si c'est une nouvelle
          {
            if (proprio->nvar == 0) proprio->var = NULL;
            proprio->nvar++;
            proprio->var = (double*) realloc(proprio->var, (proprio->nvar) * 8);
            proprio->var_symboles->Add(elts->tete);
          }
        }
      // possibilité4 : c'est une expression qui était entre parenthèses
      } else { expr = elts->tete; goto debut; }
    }
  } else
  {
    if (!cherche(elts, 0, 1, proprio)) ..... // on cherche déjà + ou - (moins prioritaire)
    if (!cherche(elts, 2, 3, proprio)) ..... // sinon * ou /
    if (!cherche(elts, 4, 4, proprio)) ..... // sinon ^
    if (!cherche(elts, 128, 255, proprio)) ..... // sinon opérateur unaire (plus prioritaire)
    if (!cherche(elts, 312, 312, proprio)); ..... // sinon opérateur unaire (plus prioritaire)
  }
}
```

```
//-------------------------------------------------------------------------------
// NEXT_LEXEME : donne le type du lexeme précédent l'indice l dans la référence c, renvoie le nouvel indice
int arbre::next_lexeme(AnsiString &expr, int l, int &c)
{
  //--- on identifie le lexeme
  AnsiString res = "";
  int i; int j = -1;
  for(i = l; (i>0) && (expr[i] == ' '); i--);
  if ((expr[i] == '*') || (expr[i] == '/') || (expr[i] == '-') || (expr[i] == '+') ||
      (expr[i] == '^') || (expr[i] == '(')  || (expr[i] == ')'))
    res = expr.SubString(i,1); else
  {
    for(j = i; (j>0) && (expr[j] != ' ') && (expr[j] != '*') && (expr[j] != '/') &&
                        (expr[j] != '-') && (expr[j] != '+') && (expr[j] != '^') &&
                        (expr[j] != '(') && (expr[j] != ')'); j--);
    res = expr.SubString(j+1,i-j);
  }
  //--- chercher le type
  c = 4;
  if (res == "(") c = 0; else if (res == ")") c = 1; else
  if (c == 4) if (res == symboles_un[0]) c = 5;
  if (c == 4) for(int i = 0; i < nun; i++) if (res == symboles_un[i]) c = 3;
  if (c == 4) for(int i = 0; i < nbin; i++) if (res == symboles_bin[i]) c = 2;
  if (c == 4) ..... // si la variable commence par un chiffre => multiplication implicite
  {
    if ((((res[1] >= '0') && (res[1] <= '9')) || (res[1]==dec)) && (!IsFloat(res)))
    {
      int k = 0; while ((j+1+k<=l) && IsFloat(res.SubString(1,k+1))) k++;
      expr.Insert("*", j+1+k);
      return next_lexeme(expr, l+1, c);
    }
  }
  //--- renvoyer le nouvel indice
  if (j == -1) return i-1; else return j;
}

//-------------------------------------------------------------------------------
// VERIFIE : vérifie si l'expression est correcte : caractères autorisés et syntaxe
bool arbre::verifie(AnsiString &expr)
{
  int l = expr.Length(), lp=l, llp=l, lllp=l; ..... // longueur de l'expression, et les 3 précédentes
  const AnsiString symb[6] = {"(", ")", "operateur_binaire",
                                        "operateur_unaire", "valeur", "-"};
  //--- vérifie caractères incorrects
  for(int i = 1; i <= l; i++)
    if ((expr[i] != ' ') && (expr[i] != '(') && (expr[i] != ')') &&
        (expr[i] != '-') && (expr[i] != '+') && (expr[i] != '*') &&
        (expr[i] != '/') && (expr[i] != '^') && (expr[i] != '_') &&
        (expr[i] != dec) && ( (expr[i] < '0') || ((expr[i] > '9') &&
        (expr[i] < 'a')) || (expr[i] > 'z') ))
  {
    ShowMessage("Caractère incorrect : '" + (AnsiString)expr[i] + "'");
    return false;
  }
  //--- vérifie symétrie parenthèses
  int count = 0;
  for(int i=1; i<=l; i++) if (expr[i]=='(') count++; else if (expr[i]==')') count--;
  if (count != 0) { ShowMessage("Parenthèses asymétriques"); return false; }
  //--- vérifie syntaxe et explicite multiplication implicite
  int c = 1, prevc = 1; ..... // code et code de l'expression précédente
  bool res = true;
  while (l>0 && res)
  {
    prevc = c; l = next_lexeme(expr, l, c);
    switch (c)
    {
      case 4: { switch(prevc){case 0: case 4: {expr.Insert("*",lp+1);  break;}
                              case 3: res=false;}
                break; }
      case 1: { switch(prevc){case 4:case 0:case 3: {expr.Insert("*",lp+1); break;} }
                break; }
      case 0: case 2: { switch(prevc){case 1:case 2: {res=false; break;}
                                      case 5: {expr.Insert("(0",lp+1);
                                               expr.Insert(")",lllp+3); break;} }
                        if (c==2 && l<=0) res = false; break; }
      case 3: case 5: { switch (prevc) { case 1: case 2: case 3: case 5: res=false;}
                        break; }
    }
    lllp=llp; llp=lp; lp=l; ..... // on décale tout
  }
  if (!res)
  {
    ShowMessage("Syntaxe incorrecte : '" + symb[c] +" "+ symb[prevc] +"' non valide");
    expression = 0;
  } else expression = expr;
  return res;
}

//-------------------------------------------------------------------------------
// PRINT : renvoie une chaîne contenant l'expression telle qu'elle a été interprétée, complètement parenthésée
AnsiString noeud::print(arbre *proprio)
{
  if (op == -1) return FloatToStr(valeur); else
  if (op>=0 && op<128)
    return (proprio->racine!=this?"(":"") + opG->print(proprio) + " " +
            symboles_bin_p[op] + " " +
            opD->print(proprio) + (proprio->racine!=this?")":"");
  else if (op == 128)
    return symboles_un[0] + (opD->op>=128?"(":"") +
           opD->print(proprio) +(opD->op>=128?")":"");
  else if (op > 128)
    return symboles_un[op-128] + ((opD->op<0)||(opD->op>127)?"(":"") +
           opD->print(proprio) + ((opD->op<0)||(opD->op>127)?")":"");
  else return proprio->var_symboles->Strings[-op-2]
```

CONTACTS

Pour toute remarque ou question, vous pouvez m'envoyer un email :

cyril@crtek.fr.st

Vous pourrez trouver la totalité du TIPE (dossier, soutenance, programmes) à l'adresse suivante :

www.crteknologies.fr.st