INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

DEA en Sciences Cognitives

INPG/Universités Joseph Fourier/Pierre Mendès-France/Stendhal

MONOPLAN: Un réseau constructif avec la règle *Minimerror*

Juan Manuel Torres Moreno

Stage effectué sous la direction de Mirta B. Gordon

Jury:

Bernard Amy

Jeanny Hérault

Mirta B. Gordon

CEA/Département de Recherche Fondamentale sur la Matière Condensée

Centre d'Etudes Nucléaires de Grenoble

FRANCE ***20 Juin 1994***

Quien lo comprenda podría
destejer un arcoiris...

Qui le comprenait pourrait
détisser un arc-en-ciel...

[J.L. Borges]

REMERCIEMENTS

Tout d'abord, mes remerciements vont à Mirta Gordon, laquelle par sa grande patience et ses encouragements a contribuée à la réussite de ce travail de stage. Ces mots de reconnaissance s'addressent à la fois au professeur, au chercheur mais aussi au personnage humaine.

Un remerciement à Valérie par la vérification du nombre maximum des fautes (de grammaire). Au ***SUN*** du **DRFMC** par son excellente mémoire. Et finalement à ***KARELLEN*** chez moi, par des heures et des heures de travail que tous les deux avons passés pendant des milliers d'exécutions de **MONOPLAN-R**, sans que'elle se fâche, au moins jusqu'à maintenant.

Ce travail a été effectué avec le support du *Consejo Nacional de Ciencia y Tecnología* (***CONACYT***) et de l'Universidad Autónoma Metropolitana—Azcapotzalco (**UAM**), à Mexico.

TABLE DE MATIERES

§1. INTRODUCTION

Le stage du **DEA** m'a permis de travailler au **Centre d'Etudes Nucléaires de Grenoble** (**CENG**) dans le **Département de Recherche Fondamentale sur la Matière Condensée** (**DRFMC**).

On a suivi une approche physique pour l'étude des réseaux de neurones, avec la règle ***Minimerror*** pour **perceptrons**. J'ai étudié cette règle et son implantation en réseaux multicouches constructivistes en utilisant l'algorithme ***Monoplan*** pour apprendre des fonctions non linéairemente séparables.

Dans une première partie du rapport je ferai des rappels sur le perceptron. Ensuite, je présenterai la règle ***Minimerror***. Le point suivant sera consacré aux réseaux multicouches et à l'algorithme ***Monoplan***, pour arriver aux résultats des simulations numériques avec le logiciel écrit en langage ***C***, **MONOPLAN-R** version **1.0,** qui utilise les deux algorithmes.

§2. LE PERCEPTRON

2.1 Rappels.

Le groupe de Frank Rosenblatt a proposé le modèle connexionniste le plus simple pour effectuer des tâches computationnelles: **le perceptron**, ou **perceptron simple** [1]. Dans ce modèle, le perceptron n'a plus que deux couches: l'entré et la sortie. Pour cette classe de perceptrons, Rosenblatt [2] propose en 1962 l'**algorithme du perceptron,** une manière de changer les poids synaptiques $\{J_i\}$.

Le perceptron simple montré sur la figure 1, est constitué d'unités appelées **neurones,** disposées en la couche d'entrée et la couche de sortie. Les neurones de l'entrée sont connectés à la sortie, mais il n'existe aucun lien dans une même couche. Les neurones de la couche d'entrée sont de simples récepteurs, ils ne traitent pas l'information: seulement l'unité de sortie joue un rôle actif.

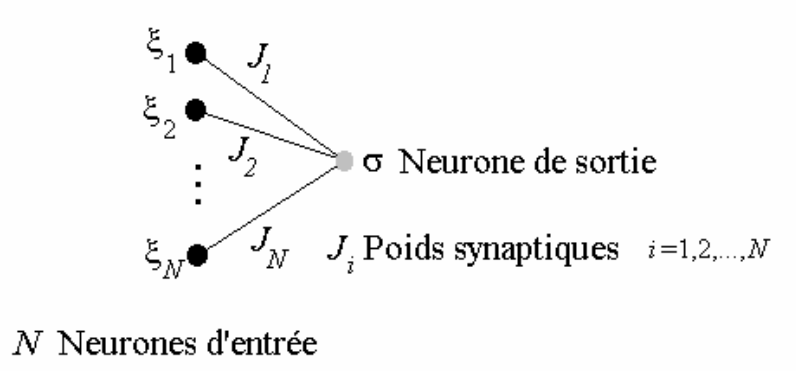


Figure 1. Le perceptron

Les communications entre neurones se font par des **synapses**. A chaque synapse on associe un nombre réel, qui représente son efficacité ou poids synaptique. Si la synapse est excitatrice il est positif, négatif si elle est inhibitrice. Dans le perceptron il y a N neurones d'entrée, donc N efficacités $\{J_i\}$ (i=1,2,...,N)**.** Lorsqu'un neurone d'entrée i se trouve dans un état ξ_i il agit sur la sortie avec une intensité proportionnelle à son état d'activité.

L'influence des neurones d'entrée crée un **champ** h sur la sortie donné par:

$$h = \sum_{i=1}^{N} J_i \xi_i \tag{2.1}$$

Quand ce champ dépasse le seuil d'activation Θ, la sortie devient active, dans le cas contraire elle est inactive:

$$\sigma = \begin{cases} +1 & \text{si} \quad h > \Theta \\ -1 & \text{si} \quad h < \Theta \end{cases} \tag{2.2}$$

On peut aussi écrire $\sigma = signe\ (h - \Theta)$.

On considère que les poids $\{Ji\}$ sont les composantes d'un vecteur $\vec{J} = (J_1, J_2, ..., J_N)$, et que les entrées ξ_i sont composantes d'un vecteur $\vec{\xi} = (\xi_1, \xi_2, ..., \xi_N)$, alors on met (2.2) sous la forme compacte suivante:

$$\sigma\ (\vec{J} \cdot \vec{\xi} - \Theta) > 0 \tag{2.3}$$

On peut représenter $\vec{\xi}$ dans un espace à *N* dimensions, appelé l'**espace des entrées**.

Souvent, on remplace dans les formules antérieures le seuil Θ par une efficacité $J_0 = -\Theta$, qui provient d'un neurone "**fictif**" numéroté 0, dont l'état est invariable à $\xi_0 = +1$. On ajoute cette influence ($J_0\ \xi_0$) au champ de la sortie et donc, il y aura *N*+1 neurones d'entrée, avec *N*+1 poids synaptiques $\{J_i\}$ i=0,1,2,...,*N*. L'utilisation de cette convention permet de transformer l'espace d'entrées de dimension *N* à dimension *N*+1. On discutera en **§3.4** le rôle de ce seuil.

2.2 Apprentissage.

Dans les systèmes naturels, on pense que l'architecture cognitive est déterminée en partie génétiquement et en partie par l'apprentissage. De ce point de vue, dans les systèmes neuronaux, l'architecture et les efficacités synaptiques devraient être déterminées afin d'obtenir un comportement désiré du réseau face un problème posé. La manière de poser le problème passe à travers exemples, appelés l'**ensemble d'apprentissage**, constitué de *P* couples { entrée, sortie }. Dans ce qui suit, on suppose qu'on connaît la réponse que le réseau doit donner aux exemples (ou *patterns*) de l'ensemble d'apprentissage. On parle alors, d'apprentissage **supervisé.**

En général, on sépare le problème de l'architecture du réseau et de celui du calcul des poids synaptiques: ceux-ci sont déterminés suivant une **règle d'apprentissage**, en supposant l'architecture fixe.

On étudiera le cas où les états des neurones d'entrée sont binaires, c'est-à-dire, ils peuvent uniquement prendre des valeurs ± 1. L'apprentissage peut être **exhaustif** ou non. On parle d'**apprentissage exhaustif** si le nombre d'exemples $P=2^N$ ou **non exhaustif** si $P=\alpha N<2^N$. C'est ce dernier cas qui est intéressant, l'apprentissage exhaustif étant utilisé pour tester les algorithmes.

Pour le perceptron, le but consiste à trouver les poids synaptiques des connections $\boldsymbol{J}=\{\boldsymbol{J}_i;i=0,1,...,N\}$, tels que sur un ensemble d'apprentissage de P exemples $\boldsymbol{\xi}^{\mu}=\{\boldsymbol{\xi}_i^{\mu}=\pm 1;i=0,1,...,N;\mu=1,2,...,P\}$, on obtienne à la sortie les valeurs désirées $\boldsymbol{\tau}^{\mu}$ à la sortie,

$$\tau^{\mu} = signe\left(\sum_{i=0}^{N} J_i\, \xi_i^{\mu}\right); \qquad 1 \le \mu \le P \tag{2.4}$$

Autrement dit, les poids trouvés après l'apprentissage devraient satisfaire:

$$\tau^{\mu}\left(\vec{J}\cdot\vec{\xi}^{\mu}\right) > 0 \tag{2.5}$$

Cela consiste à trouver dans l'espace N+1 dimensionnel des exemples, un hyperplan séparateur défini par son vecteur normal $\vec{J}=\{J_0,J_1,...,J_N\}$. Cet hyperplan sépare l'espace en deux sous-espaces, celui des exemples à sortie +1 et celui à sortie -1. On dit qu'un perceptron fait des séparations linéaires car l'équation de la hypersurface séparatrice est du **premier ordre**.

Il est utile d'introduire la notion de **stabilité** [3] des exemples qui est une fonction des poids synaptiques. On définit la **stabilité** $\boldsymbol{\gamma}^{\mu}$ d'un exemple μ dont l'entrée est $\boldsymbol{\xi}^{\mu}$ comme il suit:

$$\gamma^{\mu} = \tau^{\mu}\sum_{i=0}^{N}\frac{J_i}{\left\|\vec{J}\right\|}\xi_i^{\mu} = \frac{\tau^{\mu}}{\left\|\vec{J}\right\|}\left(\vec{J}\,\vec{\xi}^{\mu} - J_0\right) = \tau^{\mu}\frac{\vec{J}}{\left\|\vec{J}\right\|}\cdot\vec{\xi}^{\mu} \tag{2.6}$$

où

$$\left\|\vec{J}\right\| = \sqrt{\sum_{i=0}^{N} J_i^2} \tag{2.7}$$

est la norme du vecteur des poids qui peut être choisi égal à 1 ou à autre valeur.

Une grande stabilité positive assure une certaine robustesse de la réponse du perceptron. Puisque la quantité $\boldsymbol{\tau}^{\mu}\,\vec{J}\cdot\boldsymbol{\xi}^{\mu}$ peut être aussi grande que l'on veut par simple multiplication de $\vec{J}$ par une constante, ce qui ne modifie pas la

qualité de la solution trouvée, on divise $\vec{J}$ par sa norme pour que les stabilités ne dépendent pas de ce type de transformation. Ainsi, *la stabilité est la distance de l'exemple* μ *à l'hyperplan*; $\gamma^\mu>0$ si l'exemple est bien appris et $\gamma^\mu<0$ autrement.

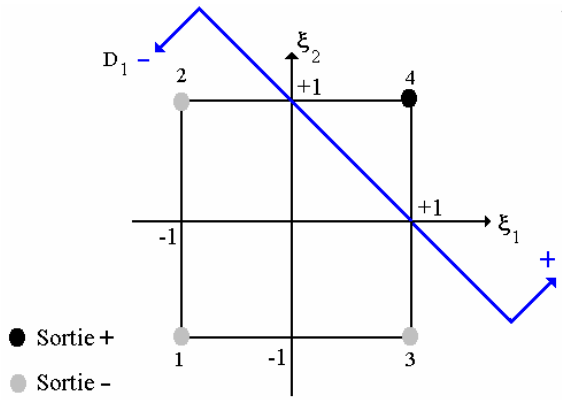
On travaillera en particulier avec $\|\vec{J}\| = \sqrt{N+1}$ comme en [4], pour avoir concordance avec des résultats. Ainsi, γ^μ devient une quantité *proportionnelle* à la distance des exemples à l'hyperplan.

2.3 Limites d'apprentissage avec le perceptron.

Le perceptron peut apprendre seulement des fonctions binaires qui sont **linéairemente séparables**. Un exemple d'une telle fonction est la fonction **ET** (*AND*). En observant la figure 2, il est facile de voir qu'il existe un plan séparateur des exemples de la classe +1 des ceux de la classe -1. En revanche, le problème du **ou exclusif** (*XOR*) n'admet pas une séparation linéaire par un plan, quel que soit son orientation. La raison peut s'expliquer facilement de façon géométrique. De la figure 3, on voit qu'il n'existe aucun plan du type:

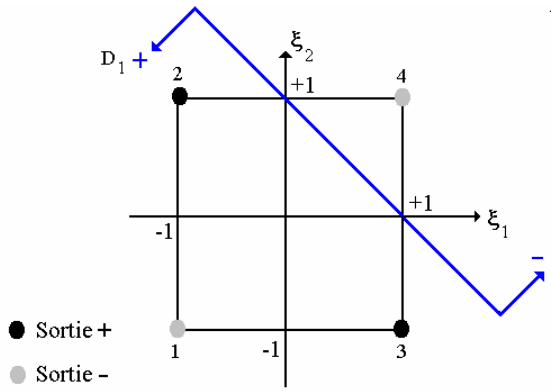
$$J_0 + \xi_1 J_1 + \xi_2 J_2 = 0 \qquad (2.8)$$

qui puisse séparer les sous-espaces à sortie +1 et -1.



μ	ξ1	ξ2	τμ
1	– 1	– 1	– 1
2	– 1	1	– 1
3	1	– 1	– 1
4	1	1	1

Figure 2. ET (*AND*)



μ	ξ1	ξ2	τ
1	– 1	– 1	1
2	– 1	1	– 1
3	1	– 1	– 1
4	1	1	1

Figure 3. XOR

Les fonctions booléennes qui sont lineairemente séparables appartiennent à un sous ensemble parmi toutes les fonctions booléennes possibles**.**

L'ensemble de toutes les fonctions booléennes à N entrées est:

$$N_B = 2^{2^N} \tag{2.9}$$

tandis que, l'ensemble des fonctions linéairement séparables est donné par [5]:

$$N_L \cong 2 \frac{2^{N^2}}{N!} \tag{2.10}$$

N_L est un nombre très réduit, même quand N est petit. Par exemple, pour N=4 il y a N_B = 65536 fonctionnes booléennes et on sait que N_L doit entre compris entre 130 et 170, approximativement.

Pour les problèmes linéairemente séparables il y a plusieurs règles d'apprentissage, par exemple la règle de Widrow-Hoff ou la règle du perceptron, mais les problèmes avec elles sont les suivants:

a) Les algorithmes du type perceptron ne convergent que si une séparation linéaire est possible. Si ce n'est pas le cas, le comportement des algorithmes est erratique.

b) S'il existe plusieurs solutions (plusieurs hyperplans qui séparent les deux classes), la *qualité* de la solution trouvée par les différents algorithmes n'est pas optimale, c'est-à-dire on doit se contenter avec une solution parmi l'ensemble total sans qu'elle fasse la meilleure séparation.

§3. LA REGLE MINIMERROR

3.1 L'erreur moyenne.

La règle *Minimerror* [4],[5] utilise des considérations de physique statistique pour trouver l'erreur moyenne. On considère un perceptron à N entrées qui réalise une sortie unique via des efficacités synaptiques $\{\boldsymbol{J}_i\}$. Chaque exemple de l'ensemble d'apprentissage $\{\boldsymbol{\xi}_i^{\mu}\}$ peut être vu comme le sommet d'un hypercube de dimension N, et sa sortie désirée $\boldsymbol{\tau}^{\mu}$.

L'idée fondamentale de la règle consiste à trouver la configuration des poids qui minimise le nombre de fautes; c'est-à-dire, le nombre d'exemples dont les stabilités sont négatives. On sera également intéressé à ce que *les stabilités soient les plus grandes possibles* pour avoir un apprentissage robuste.

Il n'est donc pas suffisant de travailler sur une fonction de coût qui compte exactement le nombre de fautes, du type $n = \sum_{\mu=1}^{P} \theta\left(-\gamma^{\mu}\right)$, où $\boldsymbol{\theta}(x)$ est la fonction de Heaviside, définie par $\theta(x) = \begin{cases} 0 & \text{si } x \le 0 \\ 1 & \text{si } x > 0 \end{cases}$ car elle compte de la même façon tous les exemples non appris quelles que soient leurs stabilités. Par exemple une modification des poids $\{\boldsymbol{J}i\}$ qui fait passer d'une stabilité $\boldsymbol{\gamma}^{\boldsymbol{\mu}_1} = -1$ à $\boldsymbol{\gamma}^{\boldsymbol{\mu}_1} = -0.1$ est autant prise en compte qu'une modification transformant $\boldsymbol{\gamma}^{\boldsymbol{\mu}_2} = -6$ à $\boldsymbol{\gamma}^{\boldsymbol{\mu}_2} = -5$ car toutes deux comptabilisent exactement la même quantité d'erreurs. Pourtant, la deuxième modification n'a aucun intérêt parce que l'exemple μ_2 pourrait ne pas être *apprenable*, tandis que la première modification pourrait amener l'exemple μ_1 à être appris dans l'itération suivante.

Pour remédier à cela, il faut modifier la fonction de coût. On peut proposer une fonction de coût avec des arguments physiques, de la manière suivante: si on considère chaque exemple comme un état, l'énergie associée à l'état μ est définie par

$$\mathrm{E}^{\mu} = \gamma^{\mu}(\{\mathrm{J_i}\}) \tag{3.1}$$

Les états d'énergie E^{μ} peuvent être occupés ou vides, de façon similaire aux états d'énergie d'un système de particules dans des niveaux d'énergie.

On définit le nombre d'états occupés par $n^{\mu}=1$ et le nombre d'états vides par $n^{\mu}=0$ [5]. L'énergie du système s'écrit donc ainsi:

$$E = \sum_{\mu=1}^{P} E^{\mu} n^{\mu} \tag{3.2}$$

Ou bien:

$$E = \sum_{\mu=1}^{P} \gamma^{\mu} n^{\mu} \quad \text{avec } n^{\mu} = 0 \text{ ou } 1 \tag{3.3}$$

De cette manière, minimiser E revient à occuper tous les niveaux pour lesquels $\gamma^{\mu}<0$. Dans ce cas, la somme $n = \sum_{\mu=1}^{P} n^{\mu}$ est exactement le nombre d'exemples ayant $\gamma^{\mu}<0$. Or, on a montré que le nombre exact d'erreurs ne donnait pas suffisamment d'informations au perceptron sur les directions vers lesquelles il doit évoluer pour mieux apprendre les exemples. De ce fait on introduit un nouveau paramètre appelé bruit ou agitation thermique. En mécanique statistique le paramètre $\beta = \frac{1}{T}$ contrôle le taux de bruit. Si β est grand, le taux de bruit est faible. Ce paramètre permet à chaque niveau d'être occupé avec une probabilité dépendant de β et proportionnelle à $e^{-\beta E^{\mu}}$. En fait, plus la stabilité γ^{μ} est négative, plus la probabilité que le niveau d'énergie correspondante soit occupé est grande, Par suite, les niveaux de basse énergie sont peuplés de façon préférentielle.

L'état général du système est défini par le vecteur $\left\{n^{\mu}\right\}_{\mu=1}^{P}$, et la probabilité de chaque état s'exprime par:

$$p\left(\left\{n^{\mu}\right\}_{\mu=1}^{P}\right) = \frac{e^{-\beta \sum_{\mu=1}^{P} \gamma^{\mu} n^{\mu}}}{Z} \tag{3.4}$$

Où p est la distribution de Boltzmann et Z est une fonction de partition [4],[5],[6] garantit que la probabilité est bien normalisée. On a:

$$Z = \sum_{\{n^{\mu}\}} e^{-\beta \sum_{\mu=1}^{P} \gamma^{\mu} n^{\mu}} = \sum_{\{n^{\mu}\}} \left(\prod_{\mu=1}^{P} e^{-\beta \gamma^{\mu} n^{\mu}} \right) \tag{3.5}$$

La somme sur tous les vecteurs { n^{μ} } (qui ne prennent que les valeurs 0 ou 1) se développe en la somme de toutes les combinaisons de produits des termes $e^{-\beta \gamma^{\mu} n^{\mu}}$. Donc, la fonction de partition peut s'écrire:

$$Z=\prod_{\mu=1}^{P}\left(1+e^{-\beta\gamma^{\mu}}\right) \tag{3.6}$$

La somme $n = \sum_{\mu=1}^{P} n^{\mu}$ est différente pour chaque état global. Toutefois, la moyenne de *n* où chaque état a une probabilité donné par (3.4) rend compte des stabilités négatives, puisque les niveaux dont la stabilité est grande ne sont pratiquement jamais peuplés. L'erreur moyenne corresponde à la valeur moyenne de *n*, définie de la manière suivante:

$$\langle n\rangle = \sum_{\left\{n^{\mu}\right\}}^{P}\left(\sum_{\mu=1}^{P} n^{\mu}\right) p\left(\left\{n^{\mu}\right\}\right) \tag{3.7}$$

$$\langle n\rangle = \sum_{\mu=1}^{P}\frac{1}{2}\left(1-\tanh\frac{\beta\gamma^{\mu}}{2}\right) \tag{3.8}$$

qu'on peut écrire:

$$\langle n\rangle = \sum_{\mu=1}^{P} V\left(\gamma^{\mu},\beta\right) \tag{3.9}$$

La règle *Minimerror* consiste à minimiser $\langle n\rangle$ à une température finie. On rappelle que T=1/β. Lorsque $\beta\to\infty$, T tend vers 0 et la fonction $V(\gamma^{\mu},\beta)$ tend vers la fonction échelon. En revanche, quand T devient très grande ($\beta\to 0$) et $V(\gamma^{\mu},\beta)$ tend vers une droite horizontale d'ordonnée $\frac{1}{2}$.

3.2 L'erreur moyenne avec deux températures.

En [4] s'analyse la performance de la règle *Minimerror* avec l'utilisation de deux températures. La performance générale est très améliorée si on considère une température T^{-} pour les exemples mal classés, avec γ^{μ} négative et une autre, T^{+} pour ceux avec stabilité positive. Ainsi, le nombre moyen d'erreurs avec deux températures est:

$$\langle n\rangle = \frac{1}{2}\left\{\left(1-\tanh\frac{\beta^{+}\gamma^{\mu}}{2}\right)\theta\left(\gamma^{\mu}\right)+\left(1-\tanh\frac{\beta^{-}\gamma^{\mu}}{2}\right)\theta\left(-\gamma^{\mu}\right)\right\} \tag{3.10}$$

3.3 La descente en gradient.

Comme la fonction $\langle n \rangle$ est continue et dérivable partout, on peut utiliser une descente en gradient pour trouver son minimum:

$$\vec{J}_{t+1} = \vec{J}_t + \delta \vec{J}_t \tag{3.11}$$

Avec:

$$\delta \vec{J}_t = -\varepsilon \frac{\partial \langle n \rangle}{\partial \vec{J}_t} \tag{3.12}$$

Le paramètre ε s'appelle le **taux d'apprentissage**, **gain** ou **pas de l'algorithme**, souvent il est compris entre 0 et 1 [6],[7].

De (3.12) et utilisant (3.8), et seulement une température T=1/β pour simplifier les calculs, on a:

$$\frac{\partial \langle n \rangle}{\partial J_i} = -\frac{\beta}{4\|\vec{J}\|} \frac{\sum_{\mu=1}^{P} \xi_i^\mu \tau^\mu}{\cosh^2\left(\frac{\beta \gamma^\mu}{2}\right)} \tag{3.13}$$

Et avec deux températures T^+ et T^- on aura:

$$\frac{\partial \langle n \rangle}{\partial J_i} = -\frac{\sum_{\mu=1}^{P} \xi_i^\mu \tau^\mu}{4\|\vec{J}\|} \left\{ \frac{\beta^+}{\cosh^2\left(\frac{\beta^+ \gamma^\mu}{2}\right)} \theta(\gamma^\mu) + \frac{\beta^-}{\cosh^2\left(\frac{\beta^- \gamma^\mu}{2}\right)} \theta(-\gamma^\mu) \right\} \tag{3.14}$$

Avec (2.7), on contraint la recherche du minimum dans une hypersphère de radius $\sqrt{N+1}$. Le choix des températures a une influence directe sur l'aspect du "paysage" de l'erreur moyenne défini par la fonction $\langle n \rangle$: à haute température on aura un paysage de "collines" douces avec des variations d'amplitude peu importantes entre un sommet et une vallée, en tout point on a une indication locale de modification des $\{Ji\}$ pour diminuer $\langle n \rangle$, mais il est

difficile de savoir si un minimum est plus profond qu'un autre. Par contre, à basses températures, le paysage aura des formes des paliers avec des variations d'amplitude très fortes entre configurations voisines.

Pour minimiser $\langle n \rangle$, on peut commencer avec des $\{Ji\}$ tirés au hasard ou avec la règle d'Hebb. On cherche d'abord la direction de la descente en gradient à haute température avec un rapport $T^+/T^-=6$ [4], et puis, on fait un refroidissement au fur et à mesure des itérations afin de trouver le minimum global du paysage. A la fin, on fait une dernière minimisation à $T^+=T^-=T$.

3.4 Le calcul des stabilités dans le même espace des entrées.

Pendant des simulations, on a trouvé quelque fois que l'algorithme ne convergeait pas à certains ensembles d'apprentissage (aléatoires ou de la parité). Ceci a posé la question si l'utilisation de la stabilité tel quelle était la plus performante pour *Minimerror*. Plus précisément, est-ce qu'il existe une autre mesure de "*stabilité*" qui peut être utilisé en lieu de γ^μ? Ce pour quoi on a décidé de calculer les stabilités dans *le même espace des entrées* et de les utiliser dans *Minimerror*.

Le terme de **biais** est parfois utilisé comme un synonyme de **seuil**. En **§2.1** on a parlé que *Minimerror* utilise des stabilités pour trouver $\langle n \rangle$, et bien entendu, elles dépendent des exemples et du vecteur $\vec{J}$.

Si on regarde la figure 4, on voit que la stabilité γ^μ mesure la distance du plan P_1 à l'exemple μ dans l'espace tridimensionnel, qui est donnée par (2.8); bien qu'on sache que la distance efficace qu'on *voudrait maximiser* soit la distance d^μ de la droite D_1 au même exemple (mesuré sur un espace bidimensionnel, de la même dimension que l'espace des entrées). De cette figure on obtient:

$$sin\,\alpha = \frac{J_0}{\|\vec{J}\|} \tag{3.15}$$

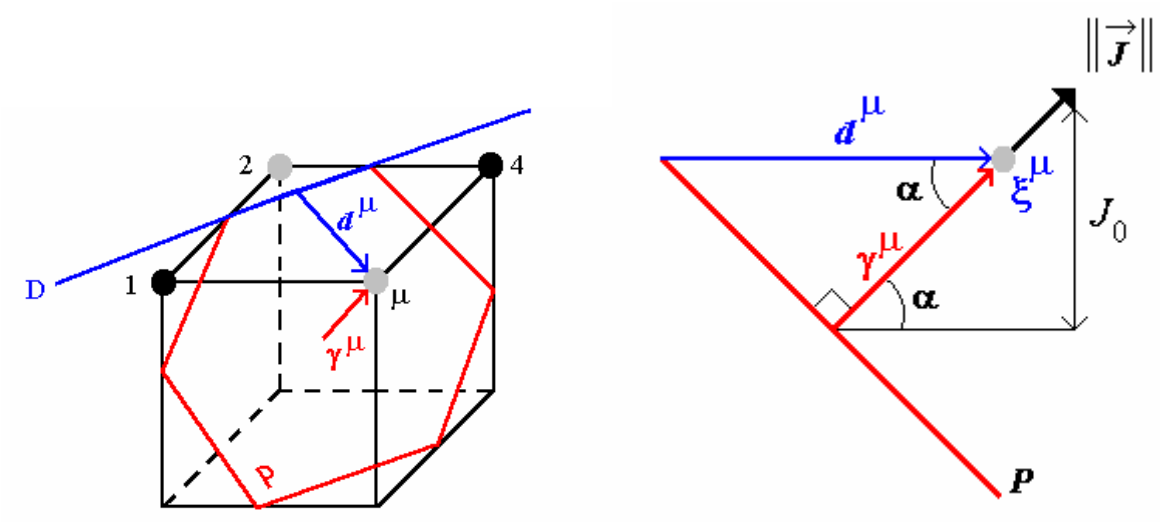


Figure 4. γ^μ vs. d^μ

$$d^\mu = \frac{\gamma^\mu}{\cos\alpha} = \frac{\gamma^\mu}{\sqrt{1-\sin^2\alpha}} = \frac{\gamma^\mu}{\sqrt{1-J_0^2/\|\vec{J}\|^2}} = \gamma^\mu \frac{\|\vec{J}\|}{\sqrt{\|\vec{J}\|^2 - J_0^2}} \qquad (3.16)$$

Or, par (2.7) et le choix $\|\vec{J}\|^2 = N+1$:

$$d^\mu = \gamma^\mu \frac{1}{\sqrt{1-\frac{J_0^2}{N+1}}} = \frac{\tau^\mu}{\sqrt{1-\frac{J_0^2}{N+1}}} \left\{ J_0 + \sum_{i=1}^{N} J_i \xi_i^\mu \right\} \qquad (3.17)$$

Pour la minimisation, il faut calculer:

$$\frac{\partial \langle n \rangle}{\partial J_i} = \frac{\partial \langle n \rangle}{\partial d^\mu} \frac{\partial d^\mu}{\partial J_i} \qquad (3.18)$$

On aura, alors:

$$\frac{\partial \langle n \rangle}{\partial d^\mu} = -\frac{\beta}{4} \frac{1}{\cosh^2\left(\frac{\beta d^\mu}{2}\right)} \qquad (3.19)$$

En utilisant (3.17), on aura pour J_0 et pour $\{Ji\}$ (i=1,2,...,N) respectivement:

$$\frac{\partial d^{\mu}}{\partial J_0} = \sum_{\mu=1}^{P} \left\{ \frac{\tau^{\mu}}{\sqrt{1 - J_0^2/N+1}} + \frac{\gamma^{\mu} J_0}{\sqrt{\left(1 - J_0^2/N+1\right)^3 (N+1)}} \right\} \qquad (3.20)$$

$$\frac{\partial d^{\mu}}{\partial J_i} = \frac{\sum_{\mu=1}^{P} \xi_i^{\mu} \tau^{\mu}}{\sqrt{1 - J_0^2/(N+1)}} \qquad i = 1,2,\ldots,N \qquad (3.21)$$

Pour la minimisation de $\langle n \rangle$, on doit utiliser (3.19) avec deux températures T^+ et T^- de la même façon qu'en (3.14).

3.5 Le problème de la parité à *N* entrées: le nombre minimum de fautes.

La fonction *XOR* à deux entrées ne peut pas être réalisée par un perceptron avec *N*=2 neurones, car elle est non linéairement séparable. Le problème *XOR* est souvent cité comme un problème difficile pour l'apprentissage en réseaux de neurones [6], [7], [8]. La généralisation de ce problème appelé la Parité, dont la sortie doit être +1 si le nombre de neurones dans l'état +1 à l'entrée est pair, et -1 autrement. On peut alors parler de Parité d'ordre *N*, qu'on notera *N*-Parité et donc, le *XOR* correspond à la 2-Parité. Le tableau 1 montre la 3-Parité.

La *N*-Parité est une fonction intéressante car on sait qu'un perceptron ne peut pas résoudre le problème, mais avec un bon algorithme d'apprentissage on peut réaliser le nombre minimum de fautes. Souvent [9],[10],[11],[12],[13], ce problème est utilisé comme test d'apprentissage pour des réseaux de neurones. En particulier, nous avons testé numériquement *Minimerror* avec ce problème.

μ	ξ1	ξ2	ξ3	τ
1	– 1	– 1	– 1	1
2	– 1	– 1	1	– 1
3	– 1	1	– 1	– 1
4	– 1	1	1	1
5	1	– 1	– 1	– 1
6	1	– 1	1	1
7	1	1	– 1	1
8	1	1	1	– 1

Tableau 1. La 3-Parité

Mais, quel est le nombre minimum de fautes pour la *N*-Parité?

Si on peut déterminer ce nombre, on est capable de vérifier si l'algorithme utilisé est correct.

Pour trouver le nombre minimum de fautes qu'un perceptron fait sur la *N*-Parité, on commence d'abord par le *XOR*. Sur la figure 5, la droite D_1 sépare le plan en deux, et on voit que les exemples 2, 3 et 4 sont bien classés, tandis que l'exemple 1 est mal classé. Donc D_1 fait, et on ne peut pas faire mieux, une erreur. Pour la 3-Parité, on doit regarder un espace à trois dimensions. Sur la figure 6, le plan P_1 sépare l'espace en deux régions, où les entrées 1,2,3,6,7 y 8 sont bien classés et les entrées 5 et 4 mal classés. Alors, P_1 fait deux fautes.

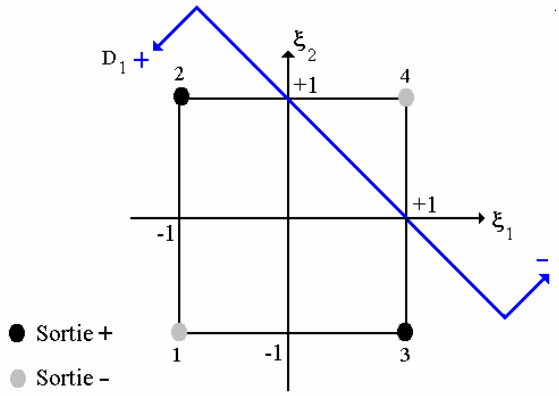


Fig 5 Le XOR

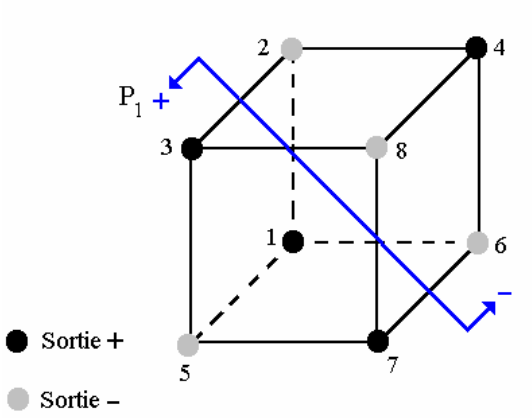


Fig. 6 La 3-Parité

En généralisant ce résultat, et puisque les entrées et sorties de la *N*-Parité quand on a $P=2^N$ exemples, sont disposées de façon symétrique dans un espace *N* dimensionnel, on peut construire le tableau suivant:

	k=0	1	2	3	4	5	6	7	8	9	10	
N	-	+	-	+	-	-	+	-	+	-	+	**Fautes**
2	1	2	1									**1**
3	1	3	3	1								**2**
4	1	4	6	4	1							**5**
5	1	5	10	10	5	1						**10**
6	1	6	15	20	15	6	1					**22**
7	1	7	21	35	35	21	7	1				**44**
8	1	8	28	56	70	56	28	8	1			**93**
9	1	9	36	84	126	126	84	36	9	1		**186**
10	1	10	45	120	210	252	210	120	45	10	1	**386**

Tableau 2. Nombre minimum de fautes pour la *N*-Parité

Où *N* correspond à l'ordre de la *N*-Parité. On définit $\boldsymbol{\nu}_k$ la distribution du nombre des sommets à sortie -1 ou +1 de façon alternée, séparés par des hyperplans successifs. Par exemple, pour *N*=3, on a un exemple à sortie -1 ($\boldsymbol{\nu}_0$), trois à sortie +1 ($\boldsymbol{\nu}_1$), trois à sortie -1 ($\boldsymbol{\nu}_2$) et un à sortie +1 ($\boldsymbol{\nu}_3$). L'hyperplan séparateur qui minimise le nombre de fautes doit donc se situer entre $\boldsymbol{\nu}_1$ et $\boldsymbol{\nu}_2$ (figure 4), ce qui donne deux fautes.

Le tableau 2 représente le triangle de Pascal. La distribution des sommets $\boldsymbol{\nu}_k$, k=0,1,..,N est donnée par les coefficients du binôme:

$$\nu_k = \binom{N}{k} = \frac{N!}{k!\ (N-k)!} \tag{3.22}$$

La dernière colonne représente le nombre minimum de fautes pour la *N*-Parité commises par un perceptron à *N* entrées. D'où, une analyse a montré que si *N* est impair, ce nombre est égal à deux fois le nombre des fautes commises par un perceptron à *N*-1 entrées. Mais si *N* est pair, grâce à la symétrie du problème (sommets disposés de façon alternée) on doit comptabiliser seulement les erreurs commises par les premiers *N*/2 hyperplans.

Ce qui donne que le nombre minimum de fautes qu'on désignera *f(N)* soit égal à:

$$f(N) = \begin{cases} f(N=2p) = \sum_{i=1}^{p} \binom{2\mathrm{p}}{2\mathrm{p}+i-1} \\ f(N=2p+1) \ = \ 2f(2p) \end{cases} \qquad \mathrm{p} = 1,2,3,...,\mathrm{N} \tag{3.23}$$

§4. MONOPLAN

4.1 Les réseaux multicouches.

Le perceptron est limité car il peut seulement apprendre des fonctions linéairemente séparables. Une manière de résoudre ce problème consiste à construire des réseaux de neurones avec des unités intermédiaires entre l'entrée et la sortie. Chaque unité peut être considérée comme un perceptron simple. Ces perceptrons sont connectés entre eux formant des couches cachées contenant des neurones qui reçoivent des informations de l'entrée et qui transmettent leurs états aux neurones de sortie ou éventuellement à d'autres neurones cachés, comme sur la figure 7. La plupart des architectures multicouches utilisent cette idée, mais il y a deux questions qui émergent par rapport à ces réseaux:

1.- *Quelle est l'architecture la plus performante du réseau?*

2.- *Combien d'unités cachées doit-on mettre dans chaque couche?*

Malgré les efforts des chercheurs, on ne sait pas bien répondre à ces questions. En principe, rien n'empêche d'avoir plusieurs couches cachées [14], mais en pratique, il est difficile d'avoir des réseaux avec plus d'une ou deux couches cachées. Il y a deux causes principales à cela: la première est théorique et repose sur le théorème de Kolmogorov [6],[15]. Ce théorème dit qu'il est toujours possible d'implémenter n'importe quelle fonction booléenne avec une couche cachée si elle possède suffisamment d'éléments. Mais la démonstration est inutilisable pour déterminer les fonctions d'activation et l'algorithme d'apprentissage. L'autre est une raison de temps, car les temps de convergence des algorithmes deviennent vite prohibitifs même avec des machines performantes.

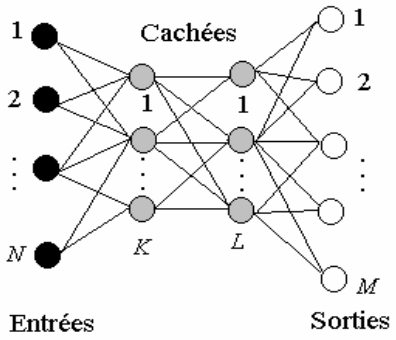


Figure 7. Réseau multicouches

4.2 La rétro-propagation de l'erreur.

La technique de rétro-propagation de l'erreur [16],[17] est une technique très populaire dans les applications des réseaux de neurones. Pour répondre à un stimulus donné, le réseau propage le signal d'entrée à la couche cachée via des synapses. Les neurones cachés calculent leur activation et puis la transforment en une réponse qui est envoyé aux neurones de la couche de sortie, lesquels calculent leur activation et donnent la réponse du réseau.

Pour l'apprentissage, le réseau doit modifier les poids des synapses entre les couches. De la même façon que le perceptron, le réseau a besoin d'un apprentissage supervisé afin de pouvoir calculer un signal d'erreur. A chaque étape, un exemple lui est présenté et une sortie calculée. Ce calcul est effectué de proche en proche de l'entrée vers la sortie dans une phase appelée **relaxation du réseau**. Ensuite, l'erreur (somme au carré des sorties calculées moins les sorties réelles) est calculée et rétro-propagée vers l'entrée donnant lieu à une modification des poids. La procédure est répétée pour tous les exemples et, si l'erreur est inférieure à une limite choisie, on dit que le réseau a convergé. L'apprentissage consiste à minimiser l'erreur quadratique commise sur l'ensemble d'apprentissage par une descente en gradient.

4.3 Réseaux constructivistes.

La rétro-propagation de l'erreur n'est pas sans problèmes: le processus d'apprentissage est quelque fois *attrapé* dans un minimum local, ce qui donne des mauvaises réponses du réseau. Le nombre de couches cachées et d'unités par couche n'est pas connu à l'avance, et doivent être déterminés par essai et erreur! [12], [18]. De plus, des neurones analogiques doivent toujours être utilisés même pour des problèmes qui nécessitent uniquement des neurones booléens. Enfin, la minimisation n'est pas garantie pour converger au minimum ayant une erreur nulle [13]. En [6] on montre trois solutions pour le *XOR* en utilisant une, deux et trois unités cachées. Si on veut résoudre ce problème avec rétro-propagation, il est indispensable définir l'architecture du réseau a priori avec une, deux, trois ou plus d'unités selon on veut, bien qu'on sache que pour la *N*-Parité dans un réseau où on impose que chaque couche puisse être connectée seulement à la couche précédente, le nombre optimal d'unités cachées est précisément *N*.

4.4 Les représentations internes.

Dans un réseau qui possède de couchées cachées, Les **représentations internes** (***RI***) sont l'ensemble d'états des neurones cachés qui prennent leurs valeurs en fonction des champs h_i produits par les entrées. Alors, on a pour les *NH* unités cachées $\sigma^{\mu} = (\sigma_j^{\mu} = \pm 1, (j=1,2,...,NH, \mu=1,2,...,P)$.

L'exemple du *XOR* pourra éclaircir la situation. La figure 8 montre la droite D_1 qui sépare les exemples de l'ensemble d'apprentissage A^1 en deux régions: les exemples 1,2 et 3 sont bien classés tandis que l'exemple 4 lui, non, ce qui donne lieu à un nouvel ensemble d'apprentissage A^2. La droite D_2 divise maintenant le nouvel ensemble où tous les exemples sont bien classés. En termes de représentations internes, cette figure montre qu'il y a une représentation interne σ_i^{μ} (ι=1,2; μ=1,2,3,4), pour chaque exemple ξ^{μ}.

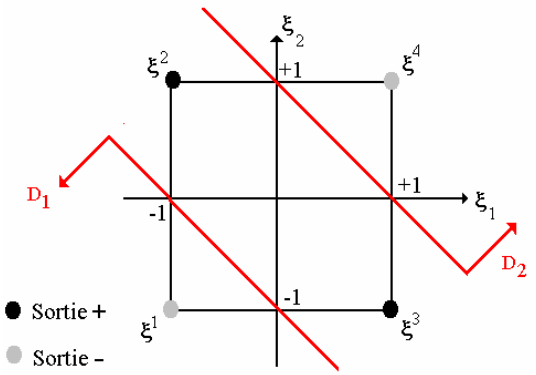


Fig. 8. Entrées et solution pour le *XOR*

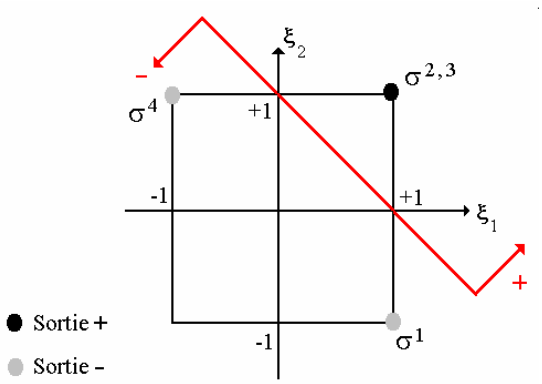


Fig. 9. Représentations internes σ^{μ}

Sur la figure 9, on observe les *RI* des entrées pour le *XOR*. Les exemples 2 et 3 ont été classés avec la même *RI*. Alors, on a la transformation:

μ	ξ1	ξ2	τμ	σ1	σ2
1	−1	−1	−1	1	−1
2	−1	1	1	1	1
3	1	−1	1	1	1
4	1	1	−1	−1	1

Entrées ξ^{μ} *RI* σ^{μ}

Tableau 3. Représentations internes pour le *XOR*

4.5 Algorithmes constructivistes.

Les réseaux constructivistes sont une approche différente des réseaux avec d'unités cachées. Le premier algorithme constructiviste est celui de M. Mézard et J.P. Nadal [13], appelé ***Algorithme de Pavage*** (*the tiling algorithm*). Unités et couches sont ajoutées en suivant une heuristique jusqu'à ce que le système apprenne la fonction booléenne désirée. Chaque unité rajoutée doit apprendre une représentation interne avec une règle du type perceptron. Le mécanisme d'apprentissage repose sur une idée simple: *deux représentations internes identiques sont incapables de produire deux différentes sorties.*

Un deuxième algorithme s'appelle **CHIR** (*Learning by choice of internal representations*), de Grossman, Meir et Domany [6]. D'abord on part des poids $\{\boldsymbol{J}j,k\}$ entre la couche d'entrée et la couche cachée, et $\{\boldsymbol{J}i,j\}$ de la couche cachée et la sortie tirés au hasard. Ensuite on construit les $P \times N$ représentations internes. On modifie $\{\boldsymbol{J}i,j\}$ par l'algorithme du perceptron. Si la sortie obtenue associée aux représentations internes est égale à la sortie désirée, l'algorithme est fini; sinon, on calcule le nombre d'erreurs et on doit modifier les synapses $\{\boldsymbol{J}i,j\}$ jusqu'à ce que le numéro change. Ensuite, on utilise la règle du perceptron pour modifier les $\{\boldsymbol{J}j,k\}$. Si la sortie calculée est correcte, l'algorithme s'arrête, sinon les efficacités $\{\boldsymbol{J}j,k\}$ déterminent un nouvel ensemble de représentations internes et la procédure doit être iterée jusqu'à la convergence.

The Upstart Algorithm de Frean [9], est autre algorithme constructiviste qui utilise l'idée suivante: on a un perceptron à N entrées avec un neurone de sortie Z. Si la sortie $\sigma^\mu \neq \tau^\mu$ pour au moins un exemple μ, on intercale deux neurones *fils* X et Y reliés aux entrées. On enseigne 1 à X si $\sigma^\mu=1$ et $\tau^\mu=-1$, ou -1 en autre cas (*wrongly "on"*). Et pour Y on enseigne 1 si $\sigma^\mu=-1$ et $\tau^\mu=1$, ou -1 en autre cas (*wrongly "off"*). Ensuite on relie X et Y à Z et on essaye d'apprendre la sortie. Des nouveaux neurones sont ajoutés tandis qu'il y a des erreurs.

4.6 L'algorithme *Monoplan*.

L'algorithme constructiviste *Monoplan* de Peretto et Gordon [11] utilise aussi les représentations internes. On suppose qu'on a un ensemble d'apprentissage $\{\xi^\mu, \tau^\mu\}$. Pour la couche cachée on part des poids $\{J1,j\}$ tirés au hasard ou avec la règle d'Hebb, et on enseigne au premier neurone τ^μ. Avec la règle *Minimerror* on calcule son état σ_1^μ. Si le problème est non lineairemente séparable, il y aura des erreurs, alors on ajoute la deuxième unité mais pour lui, on enseigne la projection du neurone avant, c'est-à-dire $\sigma_1^\mu \tau^\mu$; ainsi elle doit *uniquement d'apprendre les exemples non appris* par le premier neurone: on a transformé la fonction booléenne originelle en une nouvelle fonction définie par les ***RI***. La procédure est répétée tandis qu'il y a des erreurs en faisant toujours la projection $\sigma^\mu_{NH-1}\tau^\mu_{NH-1}$ pour le neurone *NH.*

Quand on obtient zéro fautes, alors on enseigne au perceptron de sortie l'ensemble $\{\sigma^\mu, \tau^\mu\}$ et on calcule son état σ, avec *Minimerror*. On compte le nombre des fautes: S'il n'y a plus des erreurs la procédure termine, sinon on doit ajouter une autre unité *NH*+1 à la couche cachée, et répéter la procédure maintenant avec la projection $\sigma\ \tau^\mu$, jusqu'à la convergence.

Les simulations ont montré que par la *N*-Parité on obtient les meilleurs résultats si les poids sont tirés au hasard pour les unités cachées et suivant la règle d'Hebb pour la sortie.

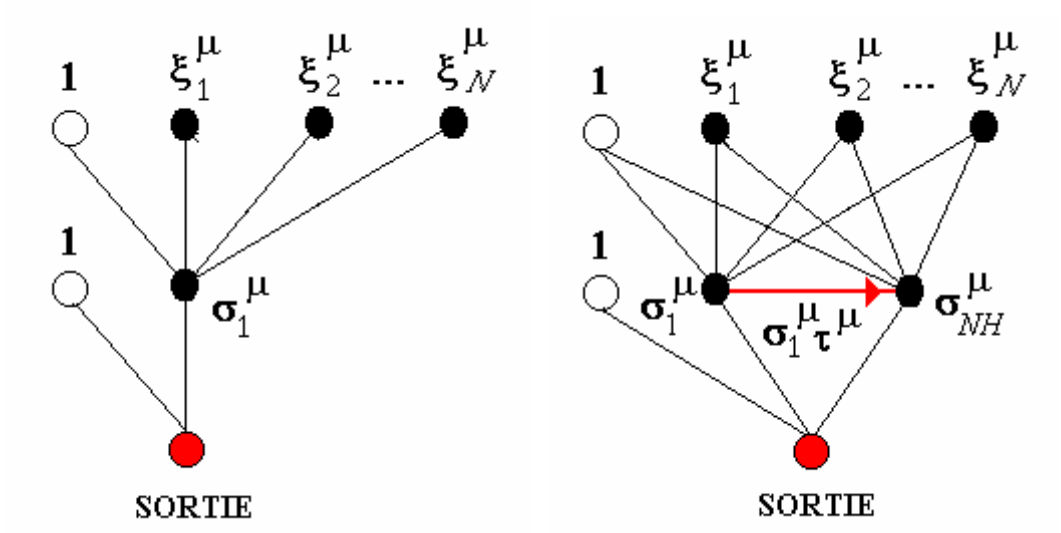


Fig. 10. L'algorithme *Monoplan*

Monoplan construit un réseau à trois couches, comme celui sur la fig. 11.

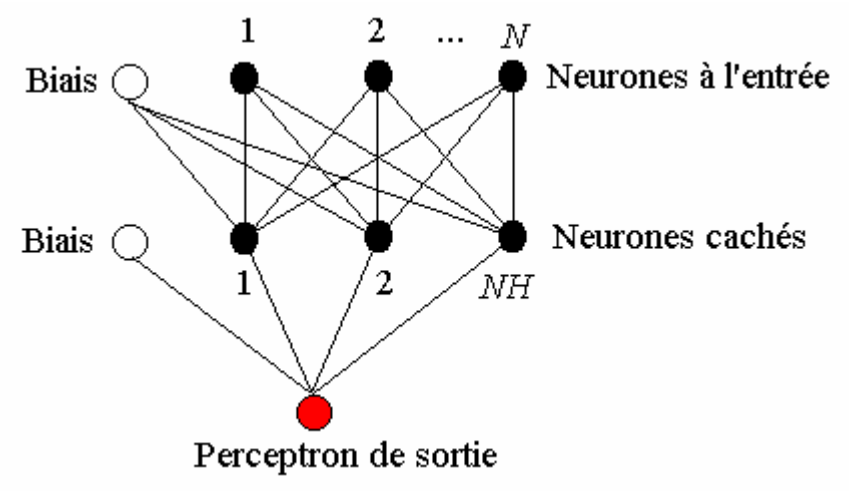


Fig 11. Réseau à trois couches

4.7 Dégénérescence des représentations internes.

Dans la couche cachée, à chaque exemple de l'entrée on associe une ***RI***. Mais, ceci n'empêche pas le fait que plusieurs exemples soient associés à la même ***RI.*** Par exemple, pour le *XOR*, les quatre exemples sont associés à trois états σ^{μ} différents (Tableau 3). Pour la 3-Parité, on aura quatre ***RI*** différentes et, en général, pour la *N*-Parité on aura *N*+1. Ceci est un phénomène désirable appelé **contraction de l'espace des entrées**. De cette manière, on aura pour les *P* exemples, seulement *P** ***RI*** différentes, avec $P^* < P$. Mais, il reste la question: on doit travailler avec toutes ou sans les ***RI*** répétés?.

Supposant que dans la couche cachée on a k_1 représentations internes répétées σ^1 correspondantes aux exemples de l'entrée $\xi^1,\xi^2,...,\xi^k$, et k_2 représentations σ^2 pour le reste $\xi^{k+1},...,\xi^p$ et $k_1>>k_2$. Si on utilise toutes les k_1+k_2 ***RI*** comme des entrées valables pour le perceptron de sortie, ceci signifie présenter au réseau k_1 fois le même exemple σ^1 et seulement k_2 fois σ^2 en privilégiant artificiellement σ^1 sur σ^2, ce qui n'a pas aucune raison d'être car tous les deux sont exemples autant valables. Ceci provoque un phénomène qu'on appellera **dégénérescence des représentations internes,** qui peut empêcher de trouver l'hyperplan séparateur optimum.

Pour le perceptron de sortie, comme deux ***RI*** égales ne peuvent pas produire deux différentes sorties, il est préférable de les éliminer. Ainsi, on peut atteindre trouver la sortie la plus stable possible, car le perceptron "voit" tous les exemples sans aucun renforcement artificiel. On a fait des simulations pour vérifier l'effet de la dégénérescence des ***RI***.

§5. SIMULATIONS AVEC **MONOPLAN-R**

MONOPLAN-R utilise la règle *Minimerror* pour l'apprentissage et l'algorithme *Monoplan* pour créer l'architecture du réseau en éliminant les représentations internes répétés, ce qui donne une diminution de la quantité de mémoire et un traitement très rapide à la sortie. Ce logiciel a été construit en ANSI *C* et il y a une version FORTRAN du SUN™. Les simulations ont été faites sur une machine SUN sous UNIX-V© et sur un PC 80486 à 66 MHz sous MS-DOS©.

5.1 Sensibilité au taux d'apprentissage ε.

Les figures 12 à 17 montrent la $\gamma>0$ minimum trouvé par rapport à ε de le premier neurone caché. Ce paramètre est important car s'il est trop grand on ne convergera jamais. Par contre, s'il est trop petit la convergence sera très ralentie. En pratique, $\varepsilon \in [0.02 ,0.03]$ fonctionne bien. Pour les tests, on a choisi $\beta_0=0.001$ et $\delta\beta_0=0.001$ avec une moyenne sur 10 tests pour les exemples aléatoires.

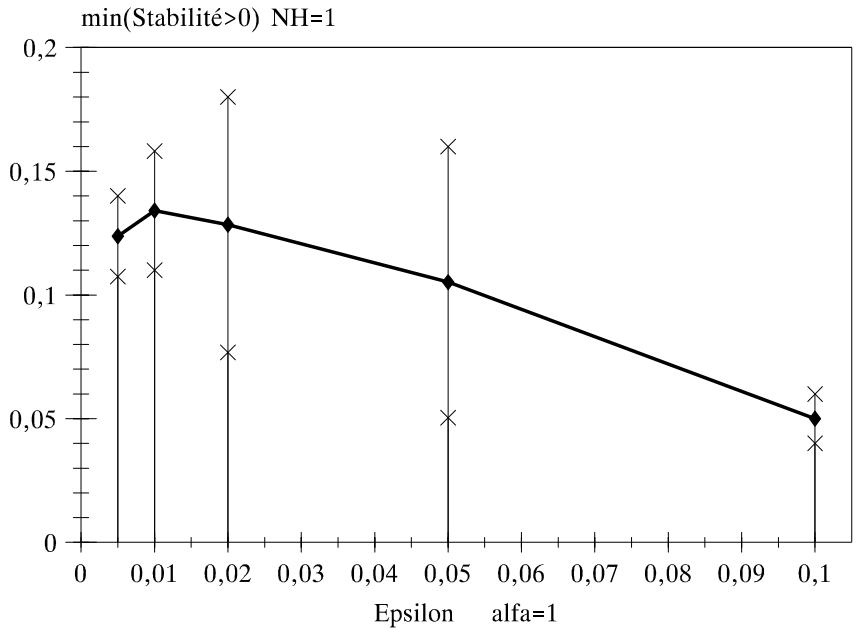


Figure 12.

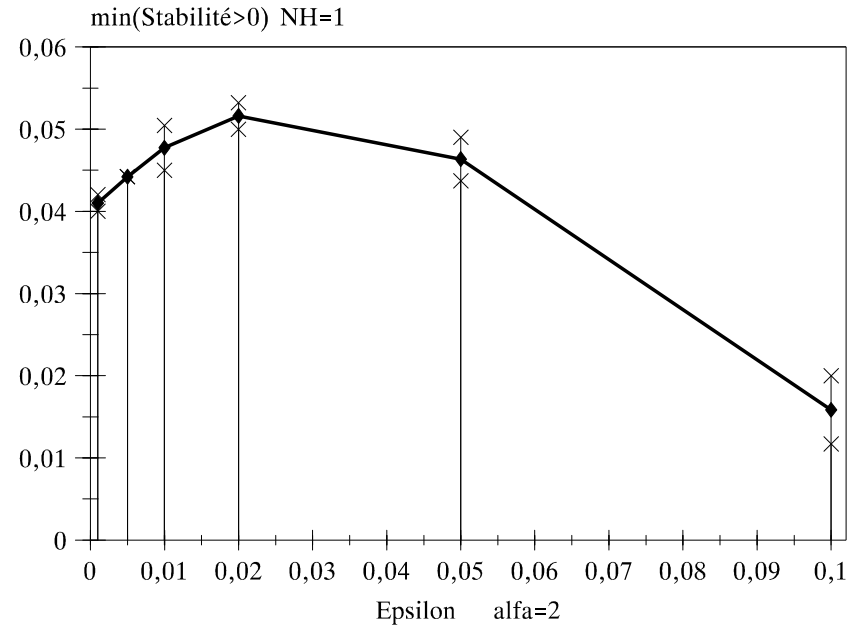


Figure 13

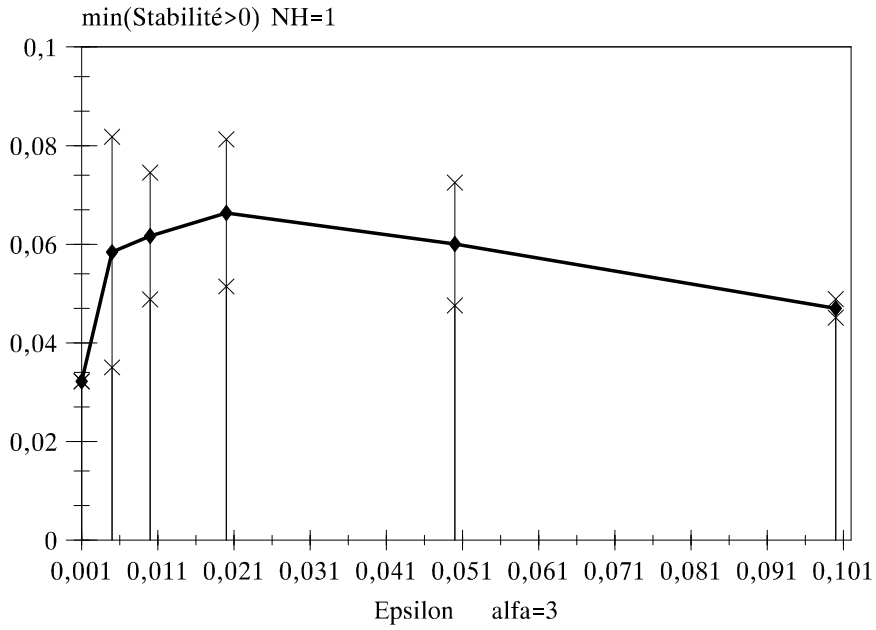


Figure 14

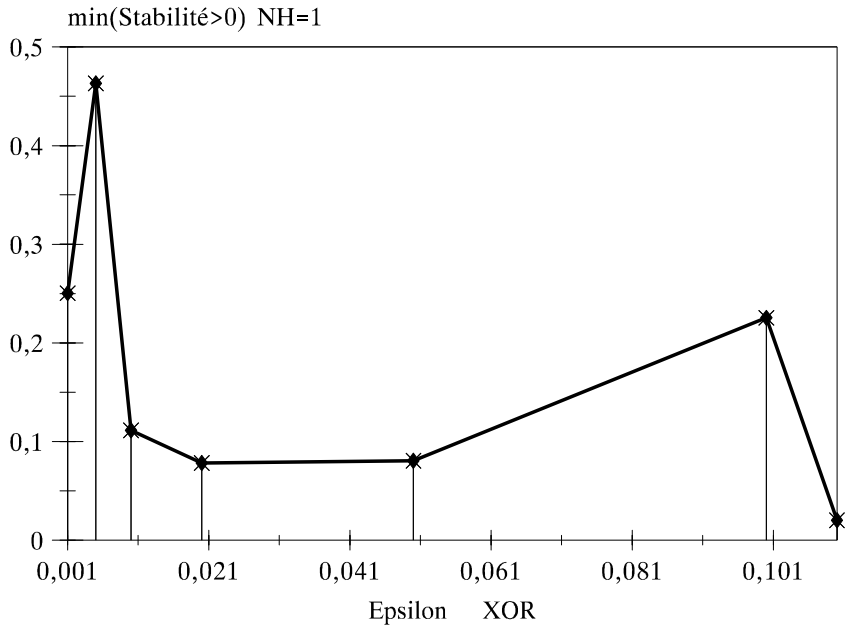


Figure 15

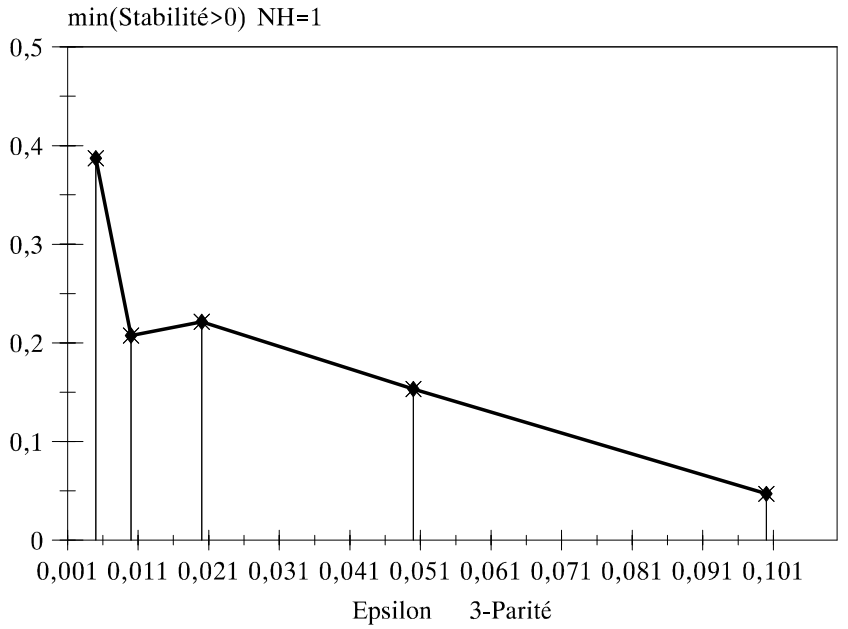


Figure 16

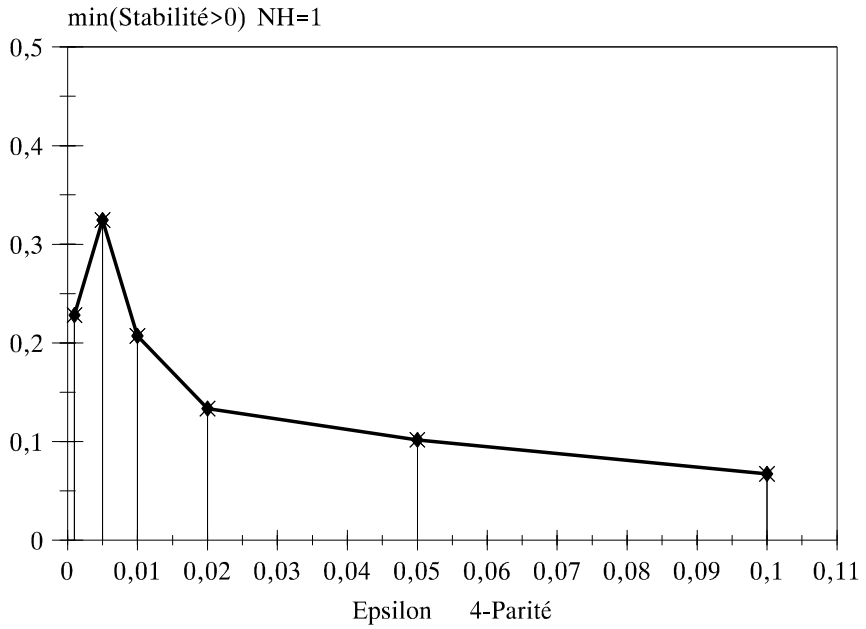


Figure 17.

5.2 Apprentissage exhaustif de la *N*-Parité.

On a testé **MONOPLAN-R** pour la *N*-Parité avec $N \leq 10$, et on a trouvé toujours la solution optimale avec $NH=N$ unités cachées. Sur le tableau 4, la 10-Parité. On a mesuré la dépendance du paramètre β par rapport au nombre d'unités cachées *NH*. Les résultats sont présentés sur la figure 18.

		Couche cachée									
i	Biais	Synapses (couche d'entrée à l'unité i)									
1	-1.04	-1.10	0.52	1.00	1.03	-1.03	-1.07	-1.02	-1.00	-1.07	-1.00
2	1.44	-0.93	0.88	0.92	0.92	-0.96	-0.93	-0.97	-1.06	-0.93	-0.93
3	2.45	0.68	-0.69	-0.73	-0.71	0.68	0.72	0.74	0.73	0.71	0.68
4	2.47	-0.70	0.72	0.69	0.70	-0.70	-0.71	-0.69	-0.71	-0.68	-0.69
5	2.87	-0.54	0.54	0.51	0.53	-0.52	-0.52	-0.54	-0.50	-0.54	-0.52
6	2.84	0.49	-0.50	-0.58	-0.53	0.54	0.54	0.57	0.56	0.54	0.55
7	3.03	0.39	-0.37	-0.46	-0.45	0.41	0.42	0.43	0.47	0.42	0.45
8	3.06	-0.45	0.46	0.33	0.39	-0.42	-0.43	-0.40	-0.37	-0.43	-0.39
9	3.12	0.23	-0.17	-0.62	-0.40	0.26	0.19	0.31	0.49	0.25	0.39
10	3.12	-0.49	0.63	0.17	0.22	-0.28	-0.42	-0.33	-0.22	-0.38	-0.15

	Unité de sortie									
Biais	Synapses (couche cachée à la sortie)									
1.00	1.00	-1.00	1.00	1.00	-1.00	-1.00	1.00	1.00	-1.00	-1.00

Tableau 4. La 10-Parité

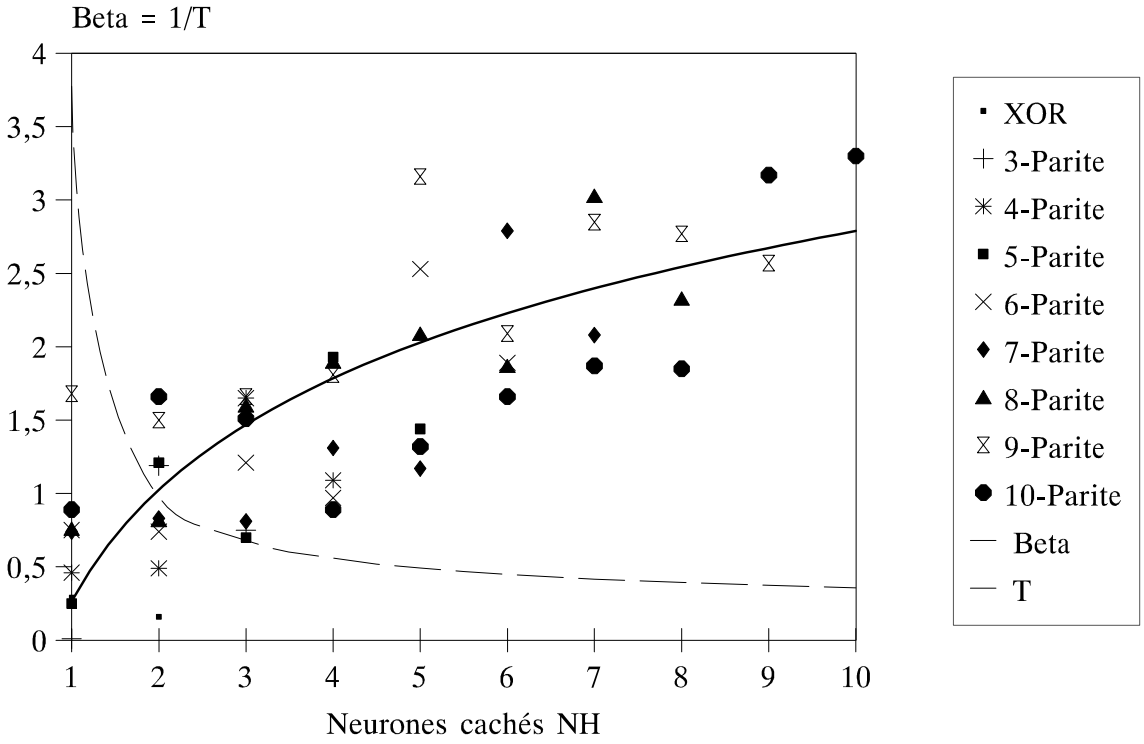


Figure 18. Dépendance de β par rapport *NH*

5.2.1 Avec ou sans toutes les représentations internes?

Le tableau 5 montre la solution pour la 6-Parité avec toutes les ***RI***. On peut y observer que la sortie ne donne pas la solution *élégante* [15] mais elle est très proche. La raison a été montrée en **§4.7**. Le tableau 6 montre la solution pour la 6-Parité en éliminant les ***RI*** répétés (la couche cachée a les mêmes poids). On peut constater une amélioration de la sortie, ce qui donne la solution exacte.

	Couche cachée						
i	Biais	Synapses (couche d'entrée à l'unité i)					
1	1.00	-1.00	1.00	-1.00	-1.00	1.00	-1.00
2	1.42	0.91	-0.91	0.91	0.91	-0.91	0.91
3	2.17	-0.62	0.62	-0.62	-0.62	0.62	-0.62
4	2.17	0.62	-0.62	0.62	0.62	-0.62	0.62
5	2.38	-0.47	0.47	-0.47	-0.47	0.47	-0.47
6	2.38	0.47	-0.47	0.47	0.47	-0.47	0.47

	Unité de sortie					
Biais	Synapses (couche cachée à la sortie)					
-0.73	**1.16**	**1.16**	**-1.07**	**-1.07**	**0.86**	**0.86**

Tableau 5. 6-Parité avec toutes les *RI*

	Unité de sortie					
Biais	Synapses (couche cachée à la sortie)					
-1.00	**1.00**	**1.00**	**-1.00**	**-1.00**	**1.00**	**1.00**

Tableau 6. 6-Parité sans *RI* répétés

5.2.2 Effet du biais sur les stabilités.

Le biais a un effet sur les stabilités: si on utilise les ***RI*** différentes et comme fonction à maximiser les stabilités γ^μ, on attend pour le XOR trois stabilités à la sortie égales à $1/\sqrt{3}$ comme le montre la fig. 9. Or, si on utilise d^μ il est raisonnable d'attendre un comportement différent: maintenant on cherche une maximisation mais en fonction des distances effectives, ce qui donne deux stabilités égales entre elles, mais plus grandes que la troisième. On a ensuite les résultats de la N-Parité ($N \leq 6$) des $N+1$ stabilités γ^μ et d^μ distances à la sortie.

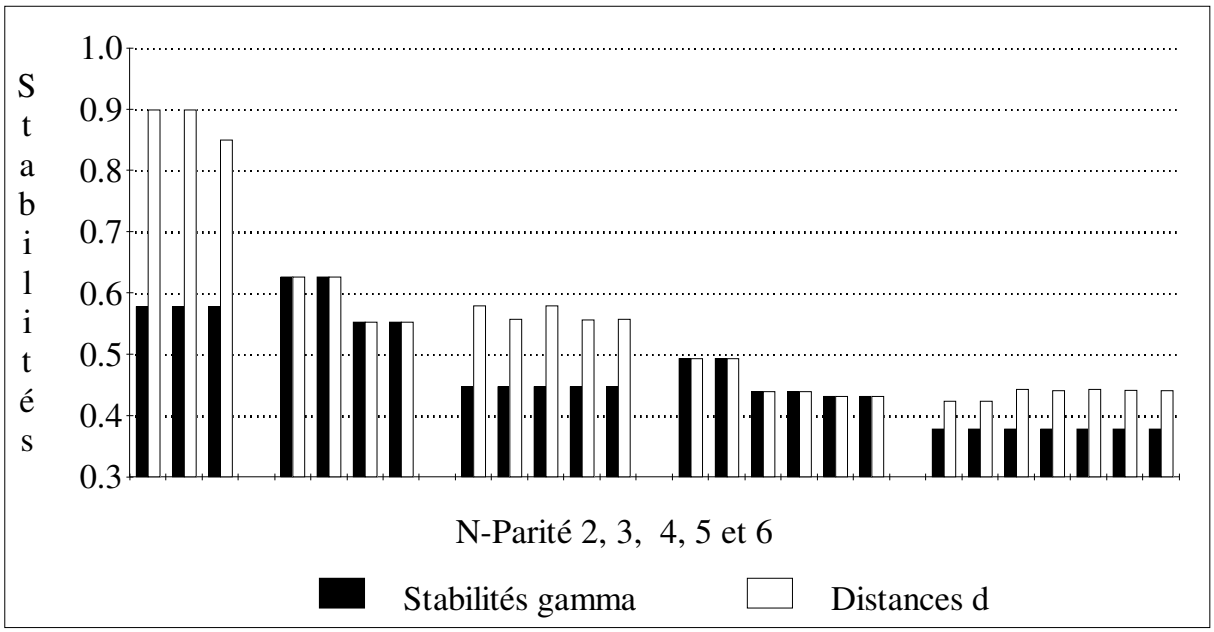


Figure 19. γ^μ vs d^μ

Si on regarde les poids synaptiques, il est intéressant de voir que les poids des neurones cachés *N* et *N-1* presque ne changent pas entre les deux tests: par exemple, les tableaux 7 et 8 montrent les poids pour la 5-Parité.

i	Biais	1	2	3	4	5
1	0.00	-1.10	1.10	-1.10	-1.10	1.10
2	1.83	0.73	-0.73	0.73	0.73	-0.73
3	1.83	-0.73	0.73	-0.73	-0.73	0.73
4	2.15	0.53	-0.53	0.53	0.53	-0.53
5	2.15	-0.53	0.53	-0.53	-0.53	0.53
	-0.00	**1.06**	**1.07**	**-1.07**	**-1.14**	**1.14**

Tableau 7. 5-Parité (γ^{μ})

i	Biais	1	2	3	4	5
1	0.00	-1.10	1.10	-1.10	-1.10	1.10
2	1.79	0.75	-0.75	0.75	0.75	-0.75
3	1.79	-0.75	0.75	-0.75	-0.75	0.75
4	2.15	0.53	-0.53	0.53	0.53	-0.53
5	2.15	-0.53	0.53	-0.53	-0.53	0.53
	-0.00	**1.06**	**1.07**	**-1.07**	**-1.14**	**1.14**

Tableau 8. 5-Parité (d^{μ})

5.3 Apprentissage non exhaustif de la *N*-Parité: la généralisation.

La généralisation de la *N*-Parité est montrée sur la figure 20. L'étude a été fait pour *N*=2,...,7. On a désigné le pourcentage de réponses erronées par rapport à $\alpha = P/N$

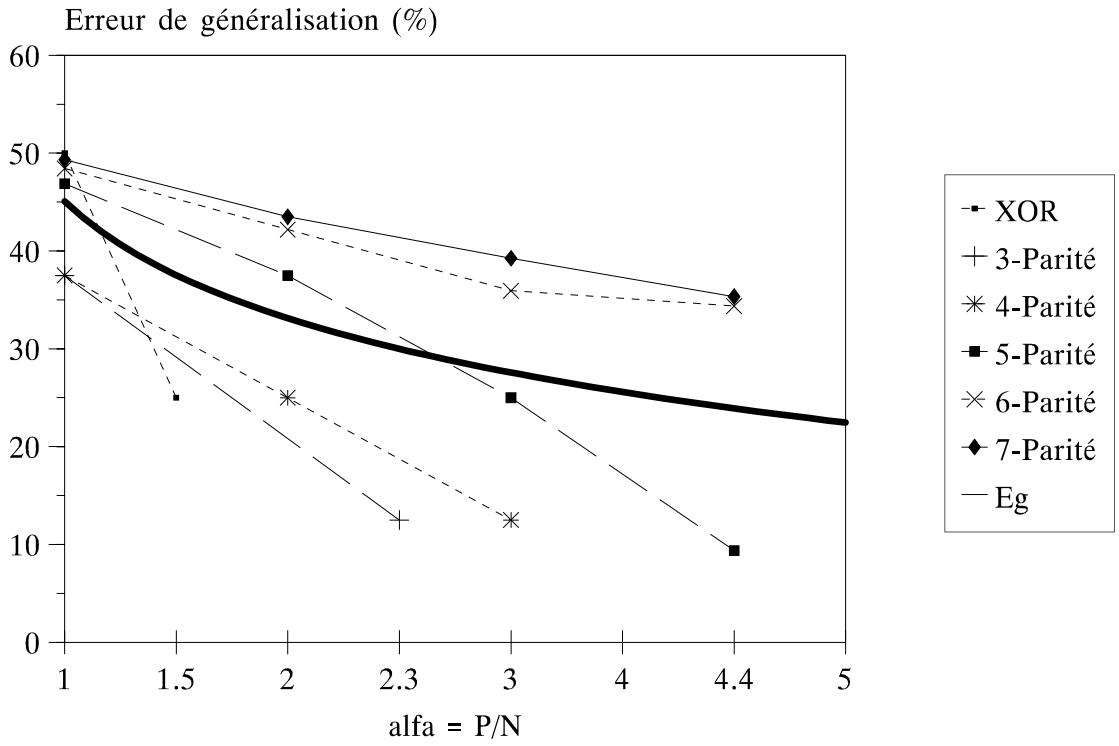


Figure 20. Généralisation pour la *N*-Parité

§6. CONCLUSION ET PERSPECTIVES

On a construit le logiciel **MONOPLAN-R**, à partir des algorithmes *Minimerror*, *Monoplan* et de l'élimination des représentations internes répétées. Les performances obtenues avec ce logiciel sont:

1. La solution trouvée est l'optimum avec les stabilités les plus grandes possibles et le nombre minimum des fautes sur l'ensemble d'apprentissage, grâce à *Minimerror* et à l'élimination des représentations internes répétées.

2. Le réseau final est à architecture simple: une seule couche cachée.

3. Les tests sur la *N*-Parité sont très satisfaisants: on trouve la solution de plus grande stabilité [14].

4. En ce qui concerne le taux d'apprentissage, on a trouvé que dans la région $\varepsilon \in [0.01, 0.05]$, on a les résultats les plus *sures* pour la convergence..

5. On a fait le calcul des distances d^{μ} avec biais (minimisation des stabilités dans l'espace N+1 dimensionnel), ce qui théoriquement doit rendre plus stables les stabilités. En effet, on a trouvé des stabilités plus grandes qu'avec le calcul des γ^{μ} dans le *N*-Parité, mais de plus en plus pour d^{μ} est proche à γ^{μ} peut-être en raison de la densité des exemples qui ne permette pas à l'hyperplan beaucoup degrés de liberté. A la limite quand $N \to \infty$, on doit avoir $d^{\mu} \equiv \gamma^{\mu}$.

Perspectifs.

On peut envisager quelques perspectives intéressantes pour ce travail:

1. Tests sur l'altération aléatoire des sorties.

i). Apprendre une fonction à N entrées fourni par un perceptron professeur.

ii). Pour k=1,...,N

Altérer k bits de la sortie du professeur et essayer d'apprendre la nouvelle fonction avec NH unités cachées. Vérifier que $NH \leq k$+1. Ce problème est actuellement en traitement.

2. L'utilisation d'un paramètre de momentum α, pour la descente en gradient pour améliorer la procédure d'apprentissage de *Minimerror*.

3. Tests sur l'altération d'une ou plusieurs synapses, ce qui signifie tester la robustesse du réseau face à perturbations externes.

ANNEXES

A1. Algorithmes.

NOTATION

N	Neurones de l'entrée.
P	Nombre d'exemples.
NH	Nombre d'unités cachées.
$\{\xi_i^\mu, \tau^\mu\}$	Ensemble d'apprentissage (i=0,1,2,...,N), (μ=1,2,...,P)
σ^μ	Représentations internes.
P^*	Nombre des *RI* différentes
$\{\sigma^\mu{}_*, \tau^\mu{}_*\}$	Ensemble d'apprentissage pour la sortie (μ=1,2,...,P^*)
JW=JW(N x NH)	Matrice des poids couche entrée → couche cachée.
JO=JO(NH)	Vecteur des poids couche cachée → sortie.
$\beta = 1/T$	Agitation thermique.
γ^μ	Stabilité de l'exemple μ

***Synapses*(J,n,ξ^μ, τ^μ,Façon)**

DEBUT

Façon Hebb: Ji ← $\Sigma\, \xi^\mu\, \tau^\mu$ i=0,1,2,...,n

Façon Aléatoire: Ji ← Aléatoires {+1, -1} i=0,1,2,...,n

J ← **J** / | **J** |

FIN *Synapses*

***CRC* (*P*,σ^μ,τ^μ)**

DEBUT

Trouver les P^* représentations internes différentes (RI)

$P \leftarrow P^*$

$\sigma^\mu \leftarrow \sigma^\mu{}_*$

$\tau^\mu \leftarrow \tau^\mu{}_*$

FIN *CRC*

***Minimerror* (J,ξ^μ,τ^μ, β)**

DEBUT

$\gamma^\mu \leftarrow \tau^\mu\, \Sigma\, J\, \xi^\mu$

δ**J** ← β/(4 *cosh*² ($\beta\gamma^\mu$/2))

J ← **J** + $\varepsilon\delta$**J**

J ← **J** / | **J** |

RETOUR nombre de $\gamma^\mu < 0$

FIN *Minimerror*

***Monoplan*(JW,*NH*, σ^μ, τ^μ)**

DEBUT

erreur ← *P*

TANDIS QUE(erreur > 0)

Synapses(**JW**(*NH*),*N*,ξ^μ,τ^μ,Aléatoire)

β^+/β^-=6

erreur ← *Minimerror*(**JW**(*NH*),ξ^μ,τ^μ,β)

$\beta^+=\beta^-$= constant

erreur ← *Minimerror*(**JW**(*NH*),ξ^μ,τ^μ, β)

σ^μ ← signe(Σ **JW**(*NH*),ξ^μ)

$\tau^\mu \leftarrow \tau^\mu \sigma^\mu$

NH ← *NH*+1

FIN

FIN *Monoplan*

MONOPLAN-R

DEBUT

LIRE FICHIER [*N*, *P*, $\left\{ \xi_i^\mu , \tau^\mu \right\}$ $\mu=1,2,...,P;\ i=0,1,2,...,N$]

erreur_sortie ← *P*

NH ← 1

TANDIS QUE erreur_sortie > 0

Monoplan(**J**(*NH*),*NH*, σ^μ,τ^μ)

CRC(*P*, σ^μ, τ^μ)

Synapses(**JO**,*NH*,σ^μ,τ^μ,Hebb)

β^+/β^-=6

erreur_sortie ← *Minimerror*(**JO**,σ^μ,τ^μ, β)

$\beta^+=\beta^-$= constant

erreur_sortie ← *Minimerror*(**JO**,σ^μ,τ^μ, β)

σ^μ ← signe(Σ **JO** ξ^μ)

$\tau^\mu \leftarrow \tau^\mu \sigma^\mu$

NH ← *NH*+1

FIN

ECRIRE FICHIER [*N*, *P*, *NH,* **J**(*NH*), **JO**]

FIN *MONOPLAN-R*

A2. Programmes source du logiciel MONOPLAN-R Version 1.0

UNIX: cc -DDOBLE Monoplan.c -o Monoplan

```
/*********************************************************************************************
*         INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE (INPG)                                 *
*                DEA en Sciences Cognitives                                                  *
*          CENTRE D'ETUDES NUCLEAIRES DE GRENOBLE (CENG)                                    *
*    Departament de Recherche Fondamentale Sur la Matiere Condensee *
*    Programa    : MONOPLAN-R.C                                                             *
*    Descripcion : Red constructivista Minimerror+Monoplan-RI                               *
*    Version     : 7.0                                                                      *
*    Creacion    : 1994.II 1     Version 1  -Fortran- 9:15 Am                               *
*    Modificacion: 1994.VI.16  Version 7  ANSI C 21:29 Pm                                   *
*    Autor            : Juan Manuel Torres                                                  *
*    Responsable : Mirta B. Gordon                                                          *
*********************************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
typedef  double real;                   /* Tipo de datos real      */
#include "arrays.h"                     /* Arreglos dinamicos      */
#include "vars.h"                       /* Variables globales      */
#include "util.h"                       /* Funciones utiles        */
#include "crc.h"                        /* Calculos CRC            */
#include "rep_int.h"                    /* Rep. interna            */
#include "hmu.h"                        /* Estabilidades           */
#include "dmu.h"                        /* Distancias              */
#include "learn.h"                      /* Apprentissage           */
#include "general.h"                    /* Generalisation          */

int menu(void)
{
     int op;
     system(CLS);
     printf("--------------------------------\n");
     printf(PROGRAMA);
     printf(VERSION);
     printf(AUTOR);
     printf(CIUDAD);
     printf("--------------------------------\n");
     printf("\t%d.- Apprentissage\n", LEARN);
     printf("\t%d.- Generalisation\n",GENER);
     printf("\t%d.- Systeme\n",       SYSTEM);
     printf("\t%d.- Fin\n",           FIN);
     printf("\tVotre choix? ");
     scanf("%d",&op);
     return op;
}

main(void)
{
  int opcion;                   /* Opcion del menu */
  do
   {
     opcion = menu();
     switch(opcion)
     {
          case LEARN:  Monoplan  (); pausa(); break;
```

```
            case GENER:  generaliza(); pausa(); break;
            case SYSTEM: system(SHELL);        break;
       }
    }  while(opcion!=FIN);
   return BIEN;
}

/*************************************************************************
*    Programa     : ARRAYS.H                                              *
*    Descripcion : Rutinas para arreglos dinamicos                        *
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#define NR_END 1
#define FREE_ARG char*
#ifndef __ARRAYS_H
#define __ARRAYS_H

real *vector(long nl, long nh)
{
   real *v;

   v = (real *) malloc((size_t) ((nh-nl+1+NR_END)*sizeof(real )));
   if(!v) { printf("no hay memoria"); exit(0); }
   return v-nl+NR_END;
}

char *ivector(long nl, long nh)
{
   char *v;

   v = (char *) malloc((size_t) ((nh-nl+1+NR_END)*sizeof(char)));
   if(!v) { printf("No hay memoria para vector..."); exit(0); }
   return v-nl+NR_END;
}

real **matriz(long nrl, long nrh, long ncl, long nch)
{
   long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
   real **m;

   m = (real **) malloc((size_t) ((nrow+NR_END)*sizeof(real *)));
   if(!m) { printf("no hay memoria"); exit(0); }
   m += NR_END;
   m -= nrl;
   m[nrl] = (real *) malloc((size_t) ((nrow*ncol+NR_END)*sizeof(real)));
   if(!m[nrl]) { printf("no hay memoria para columnas"); exit(0); }
   m[nrl] += NR_END;
   m[nrl] -= ncl;

   for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;
   return m;
}

char **imatriz(long nrl, long nrh, long ncl, long nch)
{
   long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
   char **m;
   m = (char **) malloc((size_t) ((nrow+NR_END)*sizeof(char *)));
   if(!m) { printf("no hay memoria"); exit(0); }
```

```
  m += NR_END;
  m -= nrl;
  m[nrl] = (char *) malloc((size_t) ((nrow*ncol+NR_END)*sizeof(char)));
  if(!m[nrl]) { printf("no hay memoria para columnas"); exit(0); }
  m[nrl] += NR_END;
  m[nrl] -= ncl;
  for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;
  return m;
}

void free_vector(real *v, long nl)
{
   free((FREE_ARG) (v+nl-NR_END));
}

void free_ivector(char *v, long nl)
{
   free((FREE_ARG) (v+nl-NR_END));
}

void free_matriz(real **m, long nrl, long ncl)
{
   free((FREE_ARG) (m[nrl]+ncl-NR_END));
   free((FREE_ARG) (m+nrl-NR_END));
}

void free_imatriz(char **m, long nrl, long ncl)
{
   free((FREE_ARG) (m[nrl]+ncl-NR_END));
   free((FREE_ARG) (m+nrl-NR_END));
}
#endif

/*************************************************************************
*   Programa     : vars.h                                                 *
*   Descripcion : Definiciones y macros                                   *
*************************************************************************/
#define PROGRAMA "     MONOPLAN-R v1.0\n"
#define AUTOR    "     Auteur: Juan Manuel Torres\n"
#define RESP     "Responsable: Mirta B. Gordon\n"
#define CIUDAD   "Grenoble, FRANCE. Printemps 1994\n"
         /*         Numeros Magicos                    */
#define HEBB     0                      /* JW <- Hebb(TauúXi)   */
#define RANDOM   1                      /* JW <- Rand(-1,1)     */
#define NBENTMAX 100                    /* Maximo de neuronas   */
#define NBPATMAX 2500                   /* Maximo de patterns   */
#define COSH_MAX 100.0                  /* precision de cosh(x) */
#define alea()   (double) 1.0 - 2.0 * rand() / RAND_MAX
#define sgn(x)   (int) ((x)<=0.0)?-1:+1 /* sgn(x) = +-1 signo x        */
#define BIEN     0                      /* Banderas de retorno   */
#define ERROR !BIEN
#define SI       1                      /* Sinonimos             */
#define NO       !SI
         /*         Opciones del menu       */
#define LEARN    1                      /* Aprendizaje           */
#define GENER    2                      /* Generalizacion        */
#define SYSTEM   3                      /* Sistema operativo     */
#define FIN      4                      /* Terminar              */
         /* Funciones estandar             */
double sqrt(), pow(), cosh(), fabs();
```

```
          /* Variables globales                */
FILE  *control,                                 /* Parametros iniciales   */
      *pat,                                     /* P Patterns {Xi, Tau}   */
      *red,                                     /* Resultados             */
      *neu,                                     /* Sinapsis JW y JWO      */
      *gen;                                     /* Generalizacion         */
int   iteraciones;                              /* Maximo de iteraciones*/
int   intentos;                                 /* Intentos de salida      */
int   desechadas;                               /* Neuronas desechadas*/
int   errold_s;                                 /* Error anterior salida  */
int   erreur_s;                                 /* Error de salida        */
int   errold;                                   /* Error anterior ocult   */
int   erreur;                                   /* Error ocultas          */
int   NHMAX;                                    /* Maximo neur ocultas    */
int   IMP  = NO;                                /* Imprime estabilidades*/
int   DMU  = NO;                                /* Calcula Dæ             */
int   RI   = SI;                                /* Elimina Rep.Internas */
int   ALEA = 12345;                             /* Generador aleatorios */

#ifdef DOS
  #define VERSION "Version MS-DOS\n"
  #define SHELL   "command.com"
  #define CLS     "cls"
  #define pausa()  getch()
#else
  #define VERSION "Version UNIX\n"
  #define SHELL   "csh"
  #define CLS     "clear"
  #define pausa() {int i; scanf("%d",&i);}
#endif

/*************************************************************************
*    Programa      : UTIL.H                                              *
*    Descripcion : Funciones utiles y de E/S                             *
*************************************************************************/
void error(char *msg)                           /* Mensajes de error      */
{      printf("*** ERREUR!: %s ***\n",msg); }

real lee_valor(FILE *e)                         /* Leer real de archivo */
{
   real x;
#ifdef DOBLE                                    /* Doble precision        */
   fscanf(e,"%lf",&x);
#else                                           /* Precision sencilla     */
   fscanf(e,"%f",&x);
#endif
   return x;
}
int lee_entero(FILE *e)                         /* Leer entero de archivo */
{   int x;
    fscanf(e,"%d",&x);
    return x;
}

FILE *abre(char *archivo)                       /* Abre un archivo        */
{
  FILE *f;
  if((f = fopen(archivo,"w"))==NULL)
             error(archivo);
```

```
  return f;
}

real norme(real v[],int N)            /* norme <- sqrt[ Vi^2]) */
{
     register int i;
     real s = 0.0;
     for(i=0; i<=N; i++)
            s += v[i]*v[i];                    /* norme <- v[i]^2          */
     return sqrt(s);                           /* sqrt(norme)              */
}

void normalise(real v[],int N)       /* A 1                              */
{
    register int i;
    real norv, factor;

    norv = norme(v,N);
    if(norv!=0.0)
     {
          factor = sqrt( N+1.0 )/norv;
          for(i=0; i<=N; i++)
             v[i] *= factor;
     }
}

void filas(real *vec, real **mat, int NH,int N,int op)
{
     int i;
     if(op==0)
          for(i=0; i<=N; i++)             /* Vec real <- Matriz */
             vec[i] = mat[NH][i];
     else
          for(i=0; i<=N; i++)             /* Mat real <- Vector */
             mat[NH][i] = vec[i];
}

void ifilas(char vec[], char **mat, int NH, int P,int op)
{
     int   i;
     if(op==0)
          for(i=1; i<=P; i++)             /* Vector <- Matriz*/
            vec[i] = mat[NH][i];
     else
          for(i=1; i<=P; i++)             /* Matriz <- Vector */
            mat[NH][i] = vec[i];
}
void patterns(int  N, int P, char **xi, char tau[], char tau_orig[],
               char **sigma)
{
     int i,mu;

     for(mu=1; mu<=P; mu++)
     {
          sigma[0][mu] = xi[0][mu] = 1;    /* Rep int Neurona 0 */
          for(i=1; i<=N; i++)
            xi[i][mu] = lee_entero(pat);
          tau_orig[mu] = tau[mu] = lee_entero(pat);          /* {Xi,Tau} */
     }
     fclose(pat);
```

```
}

void titulos(FILE *a,char *mensaje)
{
     fprintf(a,PROGRAMA);
     fprintf(a,"   Reseau constructiviste de perceptrons\n");
     fprintf(a,"        CENTRE D'ETUDES NUCLEAIRES\n");
     fprintf(a,"              CEA/CENG-DRFMC\n");
     fprintf(a,"   INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE\n");
     fprintf(a,"           DEA Sciences Cognitives\n");
     fprintf(a,AUTOR );
     fprintf(a,RESP  );
     fprintf(a,CIUDAD);
     fprintf(a,"%s",mensaje);
}

void estadisticas(int N,int P,int NH, real **jw, real *jwo)
{
     int i,j;

     fprintf(neu,"%d %d %d %d\n",N,P,NH,RI);     /* Sinapsis */
     fprintf(red,"-------------------------------------\n");
     fprintf(red,"               Couche cachee\n");
     fprintf(red," i Biais  Synapses (couche d'entree a l'unite i)\n");
     fprintf(red,"-------------------------------------\n");
     for(i=1; i<=NH; i++)
          {
           fprintf(red,"%2d",i);
           for(j=0; j<=N; j++)
             {
               fprintf(red,"%6.2f",jw[i][j]);
               fprintf(neu,  "%f ",jw[i][j]);       /* Sinapsis */
             }
           fprintf(red,"\n");
           fprintf(neu,"\n");
          }
     fprintf(red,"-------------------------------------\n");
     fprintf(red,"               Unite de sortie\n");
     fprintf(red," Biais  Synapses (couche cachee a la sortie)\n");
     fprintf(red,"-------------------------------------\n");
     for(i=0; i<=NH; i++)
          {
           fprintf(red,"%6.2f",jwo[i]);
           fprintf(neu,  "%f ",jwo[i]);                    /* Sinapsis */
          }
     fprintf(red,"\n-------------------------------------\n");
     fprintf(red,"    Neurones cach,s    = %d\n",NH);
     fprintf(red,"    Neurones rejet,s   = %d\n",desechadas);
     fprintf(red,"    Attentes de sortie = %d\n",intentos);
     fprintf(red,"    Total d'iterations = %d\n",iteraciones);
     fprintf(red,"-------------------------------------\n");
     fclose(red);
     fclose(neu);
}


/*************************************************************************
*    Programa      : CRC.H                                                 *
*    Descripcion : Polinomio caracteristico CRC de cada sigma     *
*                          Eliminacion de codigos repetidos                 *
```

```
**************************************************************************/
void CRC(char **sigma, int P, int NH, real *poli)
{
     int    i,mu;
     real  code;

     fprintf(red,"****************************************** CRC\n");
      printf(    "****************************************** CRC\n");
      printf("    Calcul de P = %2d codes CRC...\n",P);
     for(mu=1; mu<=P; mu++)
      {
            code = 0;
            for(i=1; i<=NH; i++)
               if(sigma[i][mu]==1) code += pow(2,NH-i);
            poli[mu] = code;
      }

}

void decode_CRC(int NH, char **sigma, int P_, real *poli_)
{
     int   i, mu, x;
     real a;

     printf("    Decode de P* = %2d codes CRC...\n",P_);
     for(mu=1; mu<=P_; mu++)
      {
           a = poli_[mu];
           for(i=NH; i>=1; i--)
            {
              x = ((int) a )% 2;
              a /= 2;
              if(x==1) sigma[i][mu] = +1;
                  else sigma[i][mu] = -1;
            }
      }
}

void elimina(real *poli, real *poli_, char *tau_orig, char *tau_, int P, int  *P_)
{
     int i, j, k;
     printf("    Eliminantion des representations internes degenerees...\n");
     *P_ = 0;
     for(i=1; i<=P; i++)                 /* [P]                          */
      {
            k = 0;                       /* k rep. diferentes */
            for(j=i+1; j<=P; j++)        /* [P-i]                */
               if(poli[i]!=poli[j]) k++;  /* poli[i] <> poli[j]           */
            if(k==(P-i))
                 {                       /* poli[i] diferente con todos */
                       (*P_) ++;
                       poli_[*P_] = poli[i];
                       tau_ [*P_] = tau_orig[i];
                 }
      }
     fprintf(red,"    (P* = %2d rep. internes differentes)\n",*P_);
      printf(    "    (P* = %2d rep. internes differentes)\n",*P_);
}

/**************************************************************************
```

```
*   Programa     : Rep_int.H                                          *
*   Descripcion : Representacion interna                              *
*************************************************************************/
void rep_interna(int P, int N, real w[], char **x, char s[])
{
     register int mu;
     real  Hmu;
     for(mu=1; mu<=P; mu++)
     {
          register int i;
          Hmu = 0;
          for(i=0; i<=N; i++)
             Hmu += w[i] * (real) x[i][mu];
          s[mu] = sgn(Hmu);
     }
}

/*************************************************************************
*   Programa     : Hmu.H                                              *
*   Descripcion : Calculo de DeltaJW = f(Hmu)                         *
*************************************************************************/
void minimerror_H(real *DeltaJW, char **x,  char *t, real *H,  real  betad,  real betagd, int n, int P)
{
     int   i,  mu;
     real ch, Hmu;
     for(i=0; i<=n; i++)
     {
          DeltaJW[i] = 0;
          for(mu=1; mu<=P; mu++)
            {
              Hmu = H[mu]*betad;                    /* B+ */
              if(Hmu<=0.0) Hmu /= betagd;           /* B- */
              if(fabs(Hmu)<COSH_MAX)
                    {
                     ch = cosh(Hmu);
                     DeltaJW[i] += (real) x[i][mu]*t[mu]/(ch*ch);
                    }
            }
     }
}

/*************************************************************************
*   Programa     : Dmu.H                                              *
*   Descripcion : Calculo de DeltaJW = f( Dμ(Hμ,Jo) )                 *
*************************************************************************/
void minimerror_D(real *DeltaJW, char **x,  char *t, real *H,  real  betad,   real betagd, int  n, int   P,
                  real Jo)
{
     int   i,  mu;
     real ch;
     real a, b, c, d, e, Dmu, Hmu;

     a = n + 1.0;
     b = a - Jo*Jo;                    /* N + 1 - Jo²                    */
     c = sqrt(b/a);                    /* sqrt[ 1 - Jo²/(N+1) ]          */
     d = pow (b/a,1.5)*a; /* (N+1)ú( 1 - Jo²/(N+1) )^3/2      */
     e = 1/c;                          /* sqrt[ (N+1)/(N+1 - Jo²) ]      */
     DeltaJW[0] = 0;
     for(mu=1; mu<=P; mu++)
             {
```

```
                Hmu = H[mu]*betad;
                if(Hmu<=0) Hmu /= betagd;       /* B- */
                Dmu = e*Hmu;
                if(fabs(Dmu)<COSH_MAX)
                      {
                        ch = cosh(Dmu);
                        DeltaJW[0] += (real) t[mu]*(e + Hmu*Jo/d)/(ch*ch);
                      }
              }

      for(i=1; i<=n; i++)     /* i=1,2,...,N */
      {
            DeltaJW[i] = 0;
            for(mu=1; mu<=P; mu++)
              {
                Hmu = H[mu]*betad;
                if(Hmu<=0) Hmu /= betagd;       /* B- */
                Dmu = e*Hmu;
                if(fabs(Dmu)<COSH_MAX)
                      {
                        ch = cosh(Dmu);
                        DeltaJW[i] += ((real) x[i][mu]*t[mu])*e/(ch*ch);
                      }
              }
      }
}

/*************************************************************************
*    Programa     : LEARN.H                                                *
*    Descripcion : Aprendizaje                                            *
*************************************************************************/
int lecture(int *N, int *P, int *itmax, real *betadmax, real *betadinit, real *dbetadinit, real *betagd,
            real *epsiloninit)
{
      char archivo[20], tmp[20];

      if((control=fopen("control.dat","r"))==NULL) error("control");
       ALEA        = lee_entero(control);      /* Random     */
      *itmax       = lee_entero(control);
      *betadmax    = lee_valor(control);
      *betadinit   = lee_valor(control);
      *dbetadinit  = lee_valor(control);
      *betagd      = lee_valor(control);
      printf("\tEpsilon 0? ");
      scanf("%lf",&*epsiloninit);
      fclose(control);                          /* control file */
      printf("\tFichier a traiter? ");
      scanf("%s",archivo);
      if((pat=fopen(archivo,"r"))==NULL)
            {
              error(archivo);
              return ERROR;
            }
      *N = lee_entero(pat);
      printf("\tExemple 1< P <= %.f? ",pow(2,*N));
      scanf("%d",&*P);
      if(*N<2 || *P<2 || (real) *P>pow(2,*N))
            {
              error("Limites erronees");
              return ERROR;
```

```
        }
    printf("\t1.- Calcul des distances  Dμ\n");
    printf("\t2.- Calcul des stabilites Hμ\n");
    printf("\tVotre choix? ");
    scanf("%d",&DMU);
    if(DMU!=2) DMU=SI;
          else DMU=NO;
    printf("\t1.- Eliminer RI r,p,t,s\n");
    printf("\t2.- Non les eliminer\n");
    printf("\tVotre choix? ");
    scanf("%d",&RI);
    if(RI!=2) RI=SI;
         else RI=NO;
    strcpy(tmp,archivo);
    if(DMU==1) strcat(tmp,".rdu");          /* Dμ              */
          else strcat(tmp,".red");                  /* Hμ              */
    red = abre(tmp);                                /* Resultados      */
    strcpy(tmp,archivo);
    if(DMU==1) strcat(tmp,".sdu");                  /* Dμ              */
          else strcat(tmp,".sin");          /* Hμ              */
    neu = abre(tmp);                                /* Sinapsis        */
    titulos(red,"\nAPPRENTISSAGE ");
    if(DMU==SI) fprintf(red,"(Distances Dμ)\n");
          else fprintf(red,"(Stabilités Hμ)\n");
    fprintf(red," --- Parametres initials ---\n");
    fprintf(red,"Fichier des entrees : %s\n" ,archivo);
    fprintf(red,"Neurones (N)        : %5d\n",*N);
    fprintf(red,"Exemples (P)        : %5d\n",*P);
    fprintf(red,"Alfa  =  (P/N)      : %9.3f\n",(real) *P/ *N);
    if((real)*P==pow(2,*N))
    fprintf(red,"Apprentissage       : EXHAUSTIF\n");
    fprintf(red,"B+/B-               : %9.3f\n",*betagd);
    fprintf(red,"Max. d'iterations   : %8d\n",  *itmax);
    fprintf(red,"Beta+ maximum       : %9.3f\n",*betadmax);
    fprintf(red,"Beta+ initial       : %9.3f\n",*betadinit);
    fprintf(red,"delta Beta+ initial : %9.3f\n",*dbetadinit);
    fprintf(red,"Epsilon             : %9.3f\n",*epsiloninit);
    fprintf(red,"Random              : %8d\n",  ALEA);
    if(RI==SI) fprintf(red,"Elimination des RI r,p,tes\n");
    return BIEN;
} /* lecture */

void graba(real *jw, int n, int erreur, int it, real b_mas, real b_menos)
{
    int i;

     printf(    "Erreur = %3d  Iter = %4d  B+ = %6.2f  B- = %6.2f\n",
               erreur,it,b_mas,b_menos);
    fprintf(red,"Erreur = %3d  Iter = %4d  B+ = %6.2f  B- = %6.2f\n",
               erreur,it,b_mas,b_menos);
    fprintf(red,"J:{");
    for(i=0; i<=n; i++) fprintf(red,"%7.2f",jw[i]);
    fprintf(red," }\n");
}

void sinapsis(real *w, char **x, char *t, int P, int N, int facon)
{
    int i, mu;

    if(facon==HEBB)                             /* Hebb            */
```

```
            for(i=0; i<=N; i++)
              {
                w[i]=0;
                for(mu=1; mu<=P; mu++)
                        w[i] += (real) x[i][mu] * t[mu];
              }
       else                                                    /* RANDOM       */
             for(i=0; i<=N; i++) w[i] = alea();
      normalise(w,N);
      printf(" %d Synapses ",N*P);
      if(facon==HEBB) printf("Hebb\n");
                        else printf("aleatoires\n");
}

void champ(real *H, real *jw, char **x, char *t, int *err, int N, int P)
{
      register int mu, i;

      *err = 0;
      for(mu=1; mu<=P; mu++)
      {
            H[mu]=0;
            for(i=0; i<=N; i++)
              H[mu] += jw[i] * (real) x[i][mu];
            H[mu] *= (real) t[mu];
            if(H[mu]<=0.0) (*err)++;
      }
}

void mod_tau(char *s, char *tau, int P)
{
      register int mu;
      for(mu=1; mu<=P; mu++)
             tau[mu] *= s[mu];
}
void gradiente(int  itmax, real betadinit, real dbetadinit, real betadmax,  real betagd,  real *betacrit,
                    real epsiloninit,  real *jw, int N, int   P, char **x, char *t, int  *erreur,   int *it)
{
      int  mu, i, j, oldf,                  /* J iteraciones              */
             count=0;                              /* para log                   */
      real *DeltaJW,                               /* Incremento JW              */
             *H,                            /* Campos                     */
             epsilon, oldbetad, betad, dbetad;
      real norma, norma_1;
      DeltaJW = vector(0,N);
             H = vector(1,P);
             *it = 0;
       *erreur = P;
      epsilon  = epsiloninit;
      oldbetad = 0.0;
      betad    = betadinit;
      dbetad   = dbetadinit;
      *betacrit= betad;
      champ(H,jw,x,t,&oldf,N,P);
      for(j=0; j<itmax; j++)
      {
            betad += dbetad;
            if(!(j % 4))
            {
              if(betad==oldbetad)               /* Betad no cambia */
```

```
            epsilon *= 0.8;                 /* 0.8 */
          oldbetad = betad;
        }
        if(betad>betadmax) goto SALIR;
        if(DMU==SI) minimerror_D(DeltaJW,x,t,H,betad,betagd,N,P,jw[0]);
                else minimerror_H(DeltaJW,x,t,H,betad,betagd,N,P);
        for(i=0; i<=N; i++)
           jw[i] += DeltaJW[i]*epsilon;
        normalise(jw,N);
        champ(H,jw,x,t,&*erreur,N,P);
        if(oldf==*erreur) count++;
        else
          {       count = 1;
           *betacrit = betad;
          }
        dbetad = log( (real) count )*dbetadinit;
          oldf = *erreur;
    } /* for ITMAX */
SALIR:        *it = j;
    iteraciones += *it;
    norma = norme(jw,N);
    norma_1 = norma*norma - jw[0]*jw[0];
    for(mu=1; mu<=P; mu++)
    {     real Hmu = H[mu]/norma;                   /* Hμ/|J|*/
          if(DMU==SI)
            Hmu = H[mu]*norma/norma_1;          /* Dμ/|J|*/
          if(IMP==SI)  {
            printf(    "S[ %2d ] = %8.5f\n",mu,Hmu);
           fprintf(red,"S[ %2d ] = %8.5f\n",mu,Hmu);  }
    }
    free_vector(DeltaJW,0);       /* Libera memoria */
    free_vector(H,0);
}
void learning(int  itmax,    real betadinit, real dbetadinit,
              real betadmax, real betagd,    real epsiloninit, real *jw, int  N, int P, char **x, char *t,
              int *erreur)
{
    int  i, e2, e3, e4, indice, it;
    real betad,             /* Temperatura de trancision                */
         *j5,               /* Vector de pesos B+ = B- = 5   */
         *j10,              /* Vector de pesos B+ = B- = 10  */
         *j15;              /* Vector de pesos B+ = B- = 15  */

    j5  = vector(0,N);
    j10 = vector(0,N);
    j15 = vector(0,N);
                                          /* B+ / B- = 6     */
    gradiente(itmax,betadinit,dbetadinit,betadmax,betagd,
                  &betad,epsiloninit,jw,N,P,x,t,&*erreur,&it);
    indice = 1;
    graba(jw,N,*erreur,it,betad,betad/betagd);
    if(betadmax <= 5.0) goto SALIR;

       /*         Minimizaciones a igual temperatura B+ / B- = 1 */
    betagd = 1.0;
    betad  = 5.0;                         /* B+ = B- = 5    */
    for(i=0; i<=N; i++) j5[i] = jw[i];
    gradiente(1,betad,dbetadinit,betad+1,betagd,
                  &betad,epsiloninit,j5,N,P,x,t,&e2,&it);
    if(e2==*erreur) { indice = 2;
```

```
                        graba(j5,N,e2,it,betad,betad);
                    }
    betad  = 10.0;                      /* B+ = B- = 10  */
    for(i=0; i<=N; i++) j10[i] = jw[i];
    gradiente(1,betad,dbetadinit,betad+1,betagd,
                &betad,epsiloninit,j10,N,P,x,t,&e3,&it);
    if(e3==*erreur) { indice = 3;
                        graba(j10,N,e3,it,betad,betad);
                    }
    betad  = 15.0;                      /* B+ = B- = 15  */
    for(i=0; i<=N; i++) j15[i] = jw[i];
    gradiente(1,betad,dbetadinit,betad+1,betagd,
                &betad,epsiloninit,j15,N,P,x,t,&e4,&it);
    if(e4==*erreur) { indice = 4;
                        graba(j15,N,e4,it,betad,betad);
                    }
SALIR: if(indice>1)
         for(i=0; i<=N; i++)
            switch(indice)
            {
              case 2: jw[i] = j5 [i]; break;
              case 3: jw[i] = j10[i]; break;
              case 4: jw[i] = j15[i]; break;
            }
    free_vector(j5, 0);    /* Libera memoria */
    free_vector(j10,0);
    free_vector(j15,0);
}


int monoplan(int N, int P, int *NH, int itmax, real betadinit, real dbetadinit,  real betadmax,
             real betagd, real epsiloninit, real **jw, char **xi, char *tau, char **sigma)
{
    real *jw_fila;                      /* Pesos por fila entrada        */
    real  epsi, bi;                     /* Epsilon, beta inicial         */
    char *sigma_fila;           /* sigma por filas                  */
    int   rechazo;
    real  delta;
    int mu;

    jw_fila    = vector(0,N);
    sigma_fila = ivector(1,P);

        epsi = epsiloninit;
        bi   = betadinit;
    delta   = (real)(1 - bi)/N;       /* Aumento de B automatico */
    erreur  = errold = P;
    rechazo = 0;
    IMP = NO;

    while(erreur>0)
    {
        fprintf(red,"---------------------------------- NEURONE CACHE %d\n",*NH);
        printf (    "---------------------------------- NEURONE CACHE %d\n",*NH);
         printf(    "Bo = %5.4f, dBo = %5.4f,", bi,dbetadinit);
        fprintf(red,"Bo = %5.4f, dBo = %5.4f\n",bi,dbetadinit);
        sinapsis(jw_fila,xi,tau,P,N,RANDOM);
        learning(itmax,bi,dbetadinit,betadmax,betagd,epsi,
                 jw_fila,N,P,xi,tau,&erreur);
```

```
        filas(jw_fila,jw,*NH,N,1);          /* jw <- fila                  */
        rep_interna(P,N,jw_fila,xi,sigma_fila);   /* sigma=sign[H mu]    */
        ifilas(sigma_fila,sigma,*NH,P,1);       /* sigma_fila[i) <- sigma */
        if(erreur==0) goto SALIR;               /* EXITO                  */
        if(erreur<errold)
         {
           mod_tau(sigma_fila,tau,P);           /* tau[i+1] = tau[i]úsigma*/
           errold = erreur;
           bi = bi+ delta;
           rechazo = 0;
         }
        else
         {
          printf("Erreur majeur ou egal a l'erreur anterieure...\n");
          rechazo++;
          desechadas++;
          switch(rechazo)
          {
           case 1: {
                     printf("Agrandir beta initial...\n");
                     if(bi<=0.02) bi *= 5.0;
                              else bi *= 1.1;
                   } break;
           case 2: {
                     printf("Agrandir beta initial...\n");
                     if(bi<=0.06) bi *= 10.0;
                              else bi *= 1.1;
                   } break;

           case 3: {
                     printf("Changer B+/B-\n");
                     if(bi<=0.09) bi *= 10.0;
                              else bi *= 1.1;
                     betagd+= 1;
                    } break;
           default: {
                     free_vector(jw_fila,0);
                     free_ivector(sigma_fila,0);
                     error("PROBLEME NON RESOLU");
                     return ERROR;
                    }
          }

          fprintf(red,"  *** Neurone rejeté: %d\n",*NH);
           printf(    "  *** Neurone rejeté: %d\n",*NH);
          (*NH)--;
         }  /* else */
        if(*NH < NHMAX) (*NH)++; /* Autre unite cachee */
                  else {
                          free_vector(jw_fila,0);
                          free_ivector(sigma_fila,0);
                          error("PROBLEME NON RESOLU");
                          return ERROR;
                         }
    }     /* while (erreur > 0) */
SALIR: free_vector(jw_fila,0);
       free_ivector(sigma_fila,1);
       return BIEN;
}
```

```
void monoplan(void)
{
    real *poli, *poli_, **jw, *jwo;
    char *tau_,*tau,*tau_orig,**xi,**sigma,*sortie;

    real  epsiloninit,                  /* Epsilon inicial */
          betadmax,                     /* B+ MAX          */
          betadinit,                    /* B+ inicial      */
          dbetadinit,                   /* Delta B+        */
          betagd,                       /* Relacion B+/B-  */
          bi;                           /* Beta inicial  local */
    int   N, P,                         /* alfa = P/N      */
          P_,                           /* Rep. Int diferentes */
          NH,                           /* NH unidades ocultas */
          itmax,                        /* Maximo de iteraciones*/
          mu;

    srand(ALEA);
    if(lecture(&N,&P,&itmax,&betadmax,&betadinit,&dbetadinit,
               &betagd,&epsiloninit)==ERROR) return;
    NHMAX  = 2*N+1;
    poli   = vector(1,P);               /* Polinomio original  */
    poli_  = vector(1,P);               /* Polinomio depurado  */
    jwo    = vector(0,NHMAX);           /* Sinapsis de sortie  */
    jw     = matriz(1,NHMAX,0,N);       /* Sinapsis            */
    tau    = ivector(1,P);          /* sortie                 */
    tau_   = ivector(1,P);              /* sortie sin RI repet */
    tau_orig= ivector(1,P);             /* sortie original     */
    xi     = imatriz(0,N,1,P);          /* Patterns            */
    sigma  = imatriz(0,NHMAX,1,P);      /* represent. internas */
    sortie = ivector(1,P);              /* salida              */

    patterns(N,P,xi,tau,tau_orig,sigma);
          NH  = 1;              /* Inicializaciones        */
       erreur_s = P;                    /* Error a la salida   */
       errold_s = P;                    /* Error anterior sal  */
    desechadas  = 0;                    /* Neuronas desechadas */
    intentos    = 0;                    /* Intentos de salida  */
    iteraciones = 0;                    /* Iteraciones totales */
                bi = betadinit;

    while(erreur_s)                     /* Mientras haya error salida */
    {
       if(monoplan(N,P,&NH,itmax,betadinit,dbetadinit,betadmax,
                   betagd,epsiloninit,jw,xi,tau,sigma)==ERROR) goto SALIR;
       if(RI==NO)
        {
          if(intentos==0)
            {
               P_ = P;
               for(mu=1; mu<=P_; mu++)
                  tau_[mu] = tau_orig[mu];
            }
        }
        else
        {
           CRC(sigma,P,NH,poli);
           elimina(poli,poli_,tau_orig,tau_,P,&P_);
           decode_CRC(NH,sigma,P_,poli_);
        }
```

```
            fprintf(red,"****************************************** SORTIE\n");
            printf (    "****************************************** SORTIE\n");
            printf ("    Perceptron [ %d neurones, %d Patterns ]\n",NH,P_);
             printf(    "Bo = %5.4f, dBo = %5.4f,", bi,dbetadinit);
            fprintf(red,"Bo = %5.4f, dBo = %5.4f\n",bi,dbetadinit);
            IMP = SI;
            sinapsis(jwo,sigma,tau_,P_,NH,HEBB);
            learning(itmax,bi,dbetadinit,betadmax,betagd,
                        epsiloninit,jwo,NH,P_,sigma,tau_,&erreur_s);
            rep_interna(P_,NH,jwo,sigma,sortie);     /* sigma_o = sign[H mu] */
            mod_tau(sortie,tau_,P_);
            for(mu=1; mu<=P; mu++)
              tau[mu] = tau_[mu];
            NH++;
            intentos++;
            if(RI==SI)
              {
                P = P_;                   /*Ir a monoplan con P_ rep. internas*/
                for(mu=1; mu<=P; mu++)
                      tau[mu] = tau_[mu];
              }
        }  /* While */



SALIR: NH--;
       estadisticas(N,P_,NH,jw,jwo);
       free_vector(poli, 1);             /* Libera la memoria */
       free_vector(poli_,1);
       free_vector(jwo,  0);
       free_matriz(jw, 1,0);
       free_ivector(tau, 1);
       free_ivector(tau_,1);
       free_ivector(tau_orig,1);
       free_imatriz(xi,    0,1);
       free_imatriz(sigma, 0,1);
       free_ivector(sortie,1);
}

/*************************************************************************
*    Programa     : GENERAL.H                                             *
*    Descripcion : Generalizacion                                         *
*************************************************************************/
void lecture_general(int *N,int *P)
{
     char archivo[20], tmp[20];

     printf("\tFichier … generaliser? ");
     scanf("%s",archivo);
     if((pat=fopen(archivo,"r"))==NULL) error(archivo);
     *N = lee_entero(pat);
     printf("\tPatterns 1< P <= %.f? ",pow(2,*N));
     scanf("%d",&*P);
     if(*N>NBENTMAX || *P>NBPATMAX) error("Limites erronees");

     strcpy(tmp,archivo);                          /* Resultados    */
     if(DMU==SI) strcat(tmp,".gdu");               /* Dµ            */
              else strcat(tmp,".gen");             /* Hµ            */
     gen = abre(tmp);
     titulos(gen,"\nGENERALISATION\n");
```

```
    fprintf(gen,VERSION);
} /* lecture_general */

void neuronas(int N, int *P, int *NH, int PP, real **jw, real jwo[])
{
    FILE *f;
    char  archivo[20];                          /* Archivo de conocimiento*/
    int   i,j,k;

    printf("\tFichier des connaissances [.sin]? ");
    scanf("%s",archivo);
    if((f=fopen(archivo,"r"))==NULL) error(archivo);
    fscanf(f,"%d%d%d%d",&k,&*P,&*NH,&RI);
    if(k!=N)
            { error("Diferentes neuronas de entrada!");
              return;
            }
    if(RI==SI) printf("\tRI eliminées dans l'apprentissage...\n");
    for(i=1; i<=*NH; i++)
          for(j=0; j<=N; j++)
             jw[i][j]=lee_valor(f);    /*  { JW  }          */
    for(i=0; i<=*NH; i++)
           jwo[i]=lee_valor(f);       /*  { JWO }          */
    fclose(f);
    printf("\tN = %d Neurones de l'entrée\n",N);
    printf("\tNH =%d Neurones de la couche cachée\n",*NH);
    printf("\tP  =%d Patterns differentes appris\n",*P);
    printf("\tP* =%d Patterns à calculer\n",PP);
}

void generaliza(void)
{
    real *poli,*poli_, *jwo, **jw, *jw_fila;
    char  *tau,*tau_,*tau_orig,*out,**xi,**sigma,*sigma_fila;
    int   err_g = 0;                    /*error de generalizacion*/
    int   N, P, PP, P_, NH;                     /* N  neurones P patterns P_ RI*/
    int   i, mu;

    lecture_general(&N,&PP);
    NHMAX   = 2*N + 1;
    poli    =  vector(1,PP);                    /* Polinomio original      */
    poli_   =  vector(1,PP);          /* Polinomio depurado    */
    jwo     =  vector(0,NHMAX);               /* Sinapsis de sortie      */
    jw      =  matriz(1,NHMAX,0,N);           /* Sinapsis                */
    jw_fila =  vector(0,N);                   /* Sinapsis                */
    tau     = ivector(1,PP);                  /* sortie            */
    tau_    = ivector(1,PP);          /* sortie sin RI repet       */
    tau_orig= ivector(1,PP);                  /* sortie original         */
    out     = ivector(1,PP);                  /* sortie calculada        */
    xi      = imatriz(0,N,1,PP);              /* Patterns                */
    sigma   = imatriz(0,NHMAX,1,PP);  /* represent. internas     */
    sigma_fila= ivector(1,PP);                /* represent. internas     */

    patterns(N,PP,xi,tau,tau_orig,sigma);
    neuronas(N, &P, &NH, PP, jw, jwo);
    for(i=1; i<=NH; i++)              /* REPRESENTATION INTERNE */
         {
          filas(jw_fila, jw, i, N, 0);
          rep_interna(PP,N,jw_fila,xi,sigma_fila);
          ifilas(sigma_fila,sigma,i,PP,1);
```

```
        }
    if(RI==SI)
    {
     CRC(sigma,PP,NH,poli);              /* CRC                    */
     elimina(poli,poli_,tau,tau_,PP,&P_);
     decode_CRC(N,sigma,P_,poli_);
    }
    else
    {
        P_ = PP;
        for(mu=1; mu<=P_; mu++)
        {
          tau_[mu]= tau[mu];
        }
    }

    rep_interna(P_,NH,jwo,sigma,out);   /* SORTIE                 */
    for(mu=1; mu<=P_; mu++)             /* GENERALISATION         */
     if(tau_[mu]!=out[mu])
          {
           err_g++;
           printf("    \tTau[%3d]=%3d  réseau=%3d\n",mu,tau_[mu],out[mu]);
          fprintf(gen,"\tTau[%3d]=%3d  réseau=%3d\n",mu,tau_[mu],out[mu]);
          }
    printf("\t>> Erreur de généralisation = %d\n",err_g);
    printf("\t>> Pourcentage = %f\n",(P_ - err_g)*100.0/P_);
   fprintf(gen,"\t>> Erreur de généralisation = %d\n",err_g);
   fprintf(gen,"\t>> Pourcentage = %f\n",(P_ - err_g)*100.0/P_);

   fclose (gen);
   free_vector(poli, 1);              /* Libera la memoria */
   free_vector(poli_,1);
   free_vector(jwo,  0);
   free_matriz(jw, 1,0);
   free_vector(jw_fila,0);
   free_ivector(tau, 1);
   free_ivector(tau_,1);
   free_ivector(tau_orig,1);
   free_ivector(out,1);
   free_imatriz(xi,    0,1);
   free_imatriz(sigma, 0,1);
   free_ivector(sigma_fila,1);
}
```

Si l'Homme est "Neuronal", le Neurone, lui, est très certainement inhumain.

REFERENCES

[1] J. Hertz, A. Krogh, R. G. Palmer (1991). *Introduction to the theory of Neural Computation. Santa Fe Institute in the Sciences of Compexity*. Addison Wesley, USA.

[2] Rosenblatt, F. (1962). *Principles de neurodynamics*. New York: Spartan Books.

[3] Krauth, W., Nadal, J.P. et Mézard, M. (1988). *The roles of stability and simmetry in the dynamics of neural networks.* Journal Physics. A: Math. Gen. 21 pp. 2995-3011.

[4] Gordon, M., Berchier, D. *Minimerror: A Perceptron Learning Rule that Finds the Optimal Weights*. ESANN'93. Broussels, April 7-9, 1993. Proceedings, Michel Verleysen Ed. pp. 105-110.

[5] Gordon, M.B., Peretto, P., Berchier, D.(1993). *Learning Algorithms for Perceptrons from Statistical Phhysics.* Journal Physics I. France 3 pp. 377-387.

[6] Peretto P. (1992). *An introduction to the Modeling of Neural Networks.* Cambridge, University Press.

[7] P. Bourret, J. Reggia, M. Samuelides (1991). *Réseaux Neuronaux: une approche connexionniste de l'Intelligence Artificielle*. Teknea, France.

[8] Minsky, M., Papert, S. (1969). *Perceptrons.* MIT Press, Cambridge.

[9] Frean, M. (1990). *The upstart Algorithm: A method for Constructing and Training Feedforward Neural Networsks*. Neural Computation 2, 198-209. MIT

[10] Nadal, J.P. (1989). *Study of a Growth Algorithm for a Feedforward Network.* L.P.S.E.N.S. preprint March. Submitted to International Journal of Neural Systems.

[11] Peretto, P., Gordon, M. *A constructive Learning Algorithme for One-hidden Layer Feed-Forward Neural Networks.* Neural Networks for Computing. Snowbird, Utha April 7-10, 1992.

[12] Hirose, Y., Yamashita K., Hijiya, S. (1991) *Back-Propagation Algorithm Wich Varies the Number of Hidden Units.* Neural Networks Vol. 4. pp. 61-66.

[13] Mézard, M., Nadal, J.P. (1989). *Learning in feedforward layered networks: the tiling algorithm* Journal Physics. A: Math. Gen. 22 pp. 2191-2203.

[14] Abdi, H. (1994). *Les réseaux de neurones*. Press Universitaires de Grenoble.

[15] Hecht-Nielsen, R. (1990). *Neurocomputing*. Reading: Addison-Wesley.

[16] Le Cun, Y. (1985). *Une procédure d'apprentissage pour réseau à seuil asymétrique.* Proceedings of Cognitiva 85, pp. 599-604. Paris

[17] Rumelhart., D.E., Hinton, G.E., Williams, R.J. (1986). *Learning representations by back-propagating errors.* Nature 323, pp. 533-536.

[18] Van Camp, D. (1992). *Neurons for computers.* Scientific American 194 pp.128-131.