12/96

الجمهـــوريـــة الجـزائـريـة الـديمـقـراطيـة الشعبـيـة
REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

وزارة التـــربيـــة الـــوطنـــيـــة
MINISTERE DE L'EDUCATION NATIONALE

ECOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
BIBLIOTHEQUE — المكتبة
Ecole Nationale Polytechnique

DEPARTEMENT ELECTRONIQUE

PROJET DE FIN D'ETUDES

SUJET

IMPLEMENTATION DES RESEAUX DE NEURONES SUR UN P.C.
APPLICATION A LA RECONNAISSANCE D'IMAGES

Proposé par :

Mme BEDDEK
Mr SADOUN

Etudié par:

Mr R. BOUBERTAKH
Mr A. BENKRID

Dirigé par:

Mme BEDDEK
Mr SADOUN

PROMOTION

Septembre 1996

E. N. P 10, Avenue Hassen Badi - EL-HARRACH - ALGER

الجمهـــوريـــة الجـزائـريـة الـديمـقـراطيـة الشعبـيـة
REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

وزارة التـــربيـــة الـــوطنـــيـــة
MINISTERE DE L'EDUCATION NATIONALE

ECOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
BIBLIOTHEQUE — المكتبة
Ecole Nationale Polytechnique

DEPARTEMENT ELECTRONIQUE

PROJET DE FIN D'ETUDES

SUJET

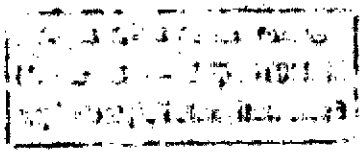
IMPLEMENTATION DES RESEAUX DE NEURONES SUR UN P.C. APPLICATION A LA RECONNAISSANCE D'IMAGES

Proposé par :	Etudié par:	Dirigé par:
Mme BEDDEK	*Mr R. BOUBERTAKH*	*Mme BEDDEK*
Mr SADOUN	*Mr A. BENKRID*	*Mr SADOUN*

PROMOTION

Septembre 1996

E. N. P 10, Avenue Hassen Badi - EL-HARRACH - ALGER



المدرسة الوطنية المتعددة التقنيات
BIBLIOTHEQUE — المكتبة
Ecole Nationale Polytechnique

Dédicaces

Je dédie ce travail :

A la mémoire de ma grand mère, en souvenir de son immense générosité, de son incommensurable gentillesse ainsi que son incomparable bonté et son altruisme sans limites.

A la mémoire de mon oncle Ali

A mes grands parents

A ma mère et a mon père que j'aime

A ma grande soeur et a mon petit frère

A tous les membres de ma famille

Rédha

Je dédie ce travail :

A ma très chère mère

A mon très chère père

A mes frères et soeurs

A tous mes Amis

A toute ma famille

Abdasamad

المدرسة الوطنية المتعددة التقنيات
BIBLIOTHEQUE — المكتبة
Ecole Nationale Polytechnique

REMERCIEMENTS

Nous tenons tout d'abord à remercier tous les enseignants de l'ENP qui ont contribués de prés ou de loin à notre formation.

Nous adressons nos remerciements à nos promoteurs Mme BEDDEK et Mr SADOUN pour leur suivi, leur aide et leur soutient tout au long de ce projet, nous les remercions encore pour les moyens qu'ils ont mis à notre disposition.

Nos remerciements vont également au groupe d'étudiants travaillant au Labo. 11 : Ahmed, Neceredine, Fayçal, Yazid, Ali et Amor.

Enfin, nous remercions tout le personnel de la bibliothèque.

المدرسة الوطنية المتعددة التقنيات
BIBLIOTHEQUE — المكتبة
Ecole Nationale Polytechnique

ملخص :

يهدف هذا العمل إلى تحقيق إنجاز لشبكات الأعصاب الإصطناعية بواسطة الحاسوب. يجب على ذلك الإنجاز أن يمثل عملاً حقيقياً لتلك الشبكات، أي بمعناه عملاً "بالتوازي". سوف يساعدنا تطبيقا (التعرف على الحروف اللاتينية) على المقارنة بين إنجازنا وبين برمجة متسلسلة عادية لشبكات الأعصاب.

RESUME

L'objet de ce travail consiste à réaliser une implémentation des réseaux de neurones sur un P.C. Cette implémentation devra reproduire un fonctionnement réel de ces réseaux, c'est à dire un fonctionnement parallèle.

Une application, la reconnaissance de caractères, servira à comparer les performances de notre implémentation logicielle avec une simulation séquentielle classique des réseaux de neurones.

ABSTRACT

The purpose of this work consist of realize an implementation of artificial neural networks on a P.C.. This implementation must reproduce a real working of these networks, which mean a parallel working.

An application, caracter's recognition, serves to compare performances of our software's implementation with a classic sequential simulation of neural networks.

المدرسة الوطنية المتعددة التقنيات
BIBLIOTHEQUE — المكتبة
Ecole Nationale Polytechnique

SOMMAIRE

INTRODUCTION GENERALE....4

CHAPITRE UN : THEORIE DES RESEAUX DE NEURONES

1. INTRODUCTION....6

2. LE NEURONE BIOLOGIQUE....6

2.1 ORGANISATION EN RESEAUX....7
2.2 PLASTICITE SYNAPTIQUE....7

3. LE NEURONE FORMEL....8

3.1 MODELISATION GENERALE....9

4. LES MODELES DE RESEAUX....12

5. REGLES DE PLASTICITE SYNAPTIQUE....15

5.1 MODELISATION DE LA REGLE DE HEBB....16

6. APPRENTISSAGE DES RESEAUX DE NEURONES....16

6.1 RESEAUX STATIQUES....17
6.1.1 APPRENTISSAGE SUPERVISE....17
6.1.1.1 REGLE DE WIDROW-HOFF....17
6.1.1.2 RETROPROPAGATION DU GRADIENT....19
6.1.2 APPRENTISSAGE NON SUPERVISE....24
6.1.2.1 CARTES AUTO-ORGANISATRICES DE KOHONEN....24
6.2 RESEAUX DYNAMIQUES....27
6.2.1 MODELE DE HOPFIELD....27
6.3 APPRENTISSAGE ET GENERALISATION....30

7. DOMAINES D'APPLICATIONS DES RESEAUX DE NEURONES....30

CONCLUSION....31

CHAPITRE DEUX : METHODES D'IMPLEMENTATION DES RESEAUX DE NEURONES

1. INTRODUCTION....32

2. METHODES D'IMPLEMENTATION DES RESEAUX DE NEURONES....32

2.1 IMPLEMENTATIONS ANALOGIQUES....32

2.1.1 EXPLICITATION DU BLOC 2 34
1. CAS DE FONCTIONS D'ACTIVATIONS A DEUX NIVEAUX... 34
2. CAS DE LA FONCTION D'ACTIVATION LINEAIRE..... 35
3. CAS DES FONCTIONS NON LINEAIRES..... 35
2.2 IMPLEMENTATIONS DIGITALES OU NUMERIQUES 39
2.2.1 IMPLEMENTATIONS PAR LOGIQUE CABLEE 39
2.2.2 IMPLEMENTATION PAR LOGIQUE MICROPROGRAMMEE 41
a) TOPOLOGIE EN BUS SIMPLE..... 43
b) TOPOLOGIES A CONNEXIONS DIRECTES..... 43
2.3 REALISATION LOGICIELLE 45
CONCLUSION..... 46

CHAPITRE TROIS : IMPLEMENTATION LOGICIELLE DES RESEAUX DE NEURONES A L'AIDE DES "DDE"

1. INTRODUCTION..... 47

2. INTERFACE UTILISATEUR..... 48

2.1 EXPLOITATION..... 49
2.1.1 RESEAU MULTICOUCHES..... 50
2.1.2 RESEAU AUTO-ORGANISATEUR..... 53
2.1.3 RESEAU DE HOPFIELD 55

3. IMPLEMENTATION LOGICIELLE A L'AIDE DES DDE..... 56

3.1 ETUDE DES DDE..... 57
3.1.1 GESTION DE LA CONVERSATION..... 58
a) INITIALISATION..... 58
b) ETABLISSEMENT DE LA CONVERSATION..... 58
c) ECHANGE DE DONNEES..... 59
d) TERMINER LA CONVERSATION..... 61
3.2 MODELE D'UN NEURONE UTILISANT LES DDE 62
3.3 IMPLEMENTATION D'UN RESEAU MULTICOUCHES A L'AIDE DES DDE..... 65
CONCLUSION..... 68

CHAPITRE QUATRE : APPLICATION A LA RECONNAISSANCE DE CARACTERES

1. INTRODUCTION..... 69

المدرسة الوطنية المتعددة التقنيات
BIBLIOTHEQUE — المكتبة
Ecole Nationale Polytechnique

2. CARACTERES GENERES 69

3. SPECIFICATION DES RESEAUX 70

3.1 DIMENSIONS DES RESEAUX UTILISES 71

4. APPRENTISSAGE 72

5. CAPACITE DE GENERALISATION 75

6. INTERPRETATION DES RESULTATS 79

CONCLUSION 81

CONCLUSION GENERALE 82

BIBLIOGRAPHIE 84

ANNEXE 85

المدرسة الوطنية المتعددة التقنيات
BIBLIOTHEQUE — المكتبة
Ecole Nationale Polytechnique

INTRODUCTION GENERALE

Copier le cerveau humain restera longtemps encore une ambition exagérée, mais vouloir s'inspirer des architectures et du fonctionnement du système nerveux n'est pas un rêve inaccessible. C'est ainsi que sont nées les techniques neuromimétiques (ressemblance avec les neurones), issues des mariages entre la neurobiologie d'une part et la physique, l'informatique et les mathématiques d'autre part.

Les méthodes neuromimétiques, appelées aussi connexionistes, essaient de comprendre les principes selon lesquels les systèmes biologiques traitent l'information et s'en inspirent pour élaborer de nouvelles techniques en sciences de l'ingénieur. Elle s'appuient pour cela sur une structure de base : le neurone artificiel ou neurone formel.

L'intérêt de ces méthodes est que malgré la constante augmentation de puissance des calculateurs et les approches théoriques de plus en plus sophistiquées, un certain nombre de tâches résistent encore aux algorithmes et aux méthodes classiques de traitement des signaux et des données. Ces tâches relèvent typiquement du traitement, en temps réel, de très grands flots de données souvent multidimensionnelles et arrivant à des cadences élevées, comme capter une image, la numériser, la segmenter en élément de contour, détecter un objet mobile, capter le son d'une voix au milieu du bruit ambiant et reconnaître les mots qui sont prononcés. Or toutes ces opérations se trouvent réalisées de manière naturelle chez les êtres vivants: un visage est reconnu en quelques dixièmes de secondes, une voix est reconnue au milieu du bruit ambiant et le discours est perçu. Les réseaux de neurones tentent donc de modéliser mathématiquement le cerveau humain afin de produire de nouvelles machines plus efficaces que ceux dont nous disposons. Les principales caractéristiques de ces réseaux sont leurs faculté d'apprentissage et leurs fonctionnement parallèle.

Notre étude portera sur la théorie des réseaux de neurones et leurs différents modèles ainsi que sur les différentes méthodes d'implémentation de ces réseaux. On présentera une méthode d'implémentation, basée sur des techniques logicielles, qui tient compte du fonctionnement parallèle de ces réseaux.

Notre travail a été divisé en quatre parties:

Le premier chapitre est consacré aux fondements théoriques des réseaux de neurones, leurs différents modèles ainsi que les algorithmes d'apprentissages associés à chaque modèle de réseau.

Le deuxième chapitre expose différentes méthodes d'implémentation des réseaux de neurones : méthodes analogiques, méthodes numériques ou digitales et les méthodes logicielles.

Le troisième chapitre présente la méthode d'implémentation adoptée. C'est une méthode logicielle utilisant les DDE (Dynamic Data Exchange : échange dynamique de données) qui reproduit un fonctionnement parallèle des réseaux de neurones.

Le quatrième chapitre est une application des réseaux de neurones, à savoir la reconnaissance de caractères, destinée à comparer les performances de notre implémentation logicielle avec une simulation séquentielle classique.

Enfin, on termine par une conclusion générale qui résume les objectifs atteints.

CHAPITRE 1

THEORIE DES RESEAUX DE NEURONES

1. INTRODUCTION

Les réseaux de neurones formels sont des structures simulées par des algorithmes qui tentent d'imiter certaines des fonctions du cerveau humain en reproduisant certaines de ses structures de base.

Ils sont utilisés essentiellement pour résoudre des problèmes de classification, de reconnaissance de formes ou de traitement du signal

Ce premier chapitre a pour but d'introduire le modèle du neurone formel, ainsi que les différents types d'architectures des réseaux de neurones et leurs algorithmes d'apprentissage.

2. LE NEURONE BIOLOGIQUE

L'élément de base du système nerveux est le neurone (Figure 1.1) .
Un neurone est une cellule constituée de 3 parties : les dendrites, le soma et l'axone.

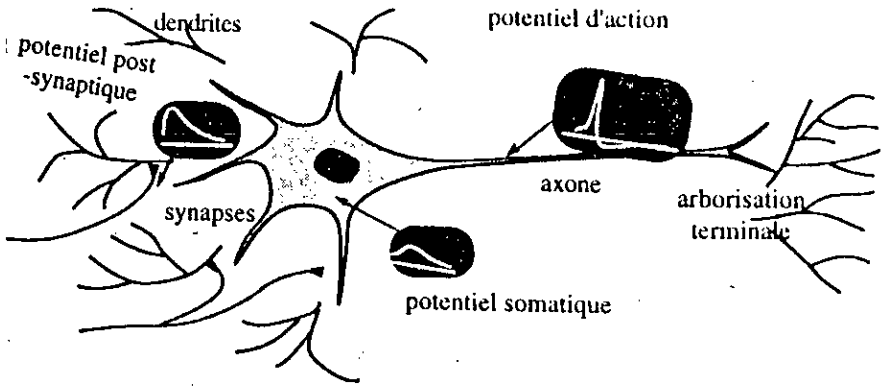


Fig.1.1 : Cellule nerveuse.

Les dendrites reçoivent l'information au niveau de points de contacts avec les autres neurones. Ces points de contacts sont appelés synapses. Certaines cellules nerveuses comptent jusqu'à 100.000 synapses. L'information est ensuite acheminée vers le corps cellulaire ou soma.

L'influx nerveux est une impulsion électrique, et les signaux dendritiques et somatiques sont des variations de potentiel électrique.

Entre deux cellules nerveuses, au niveau de la synapse, la transmission se fait par l'intermédiaire d'un médiateur chimique appelé neurotransmetteur.

Le soma recueille et concentre l'ensemble des informations reçues par les dendrites et en fait une sommation. Si le potentiel somatique dépasse un certains seuil, il y a émission d'un potentiel d'action ou "spike". Ce signal très bref (1ms) est transmis sans atténuation le long de l'axone parfois sur plus d'un mètre, et réparti sur les synapses des neurones cibles grâce à l'arborisation terminale [1].

2.1 ORGANISATION EN RESEAUX

Pour former le système nerveux, les neurones sont connectés les uns aux autres suivant des répartitions spatiales complexes.

On estime à environ 100 à 1 000 Milliards le nombre de neurones du système nerveux humain, et on compte en moyenne de 1 000 à 100 000 synapses par neurone. Les neurones sont donc fortement connectés entre eux, mais ces connections ne sont pas aléatoires. Elles correspondent à des réseaux dont les architectures sont assez bien connues mais dont les propriétés fonctionnelles restent encore difficiles à comprendre.

2.2 PLASTICITE SYNAPTIQUE

Une propriété très importante des neurones est la plasticité du système nerveux. C'est la faculté d'évolution des cellules nerveuses et de leurs interconnexions. Les notions d'apprentissage et de mémoire sont très étroitement liées à cette propriété.

A la naissance, le système nerveux est caractérisé par un précâblage inné redondant des réseaux neuronaux. Ce n'est qu'après quelques années (vers 4 ans) et grâce aux stimulations extérieures, que les circuits neuronaux se spécialisent par sélection, élimination ou renforcement des connexions. On aboutit à la stabilisation des réseaux de neurones par plasticité de leur connectivité [1].

3. LE NEURONE FORMEL

Le premier modèle de neurone fut élaboré en 1943 par Mc CULLOCH et PITTS.
Le neurone est modélisé par 2 opérateurs (Fig.1.2) [1] :

- Un opérateur de sommation. Il calcule un "potentiel" p égal à la somme de ses entrées, pondérées par des coefficients appelés poids synaptiques.
- Un opérateur calculant l'état de la sortie y du neurone en fonction de son potentiel p.

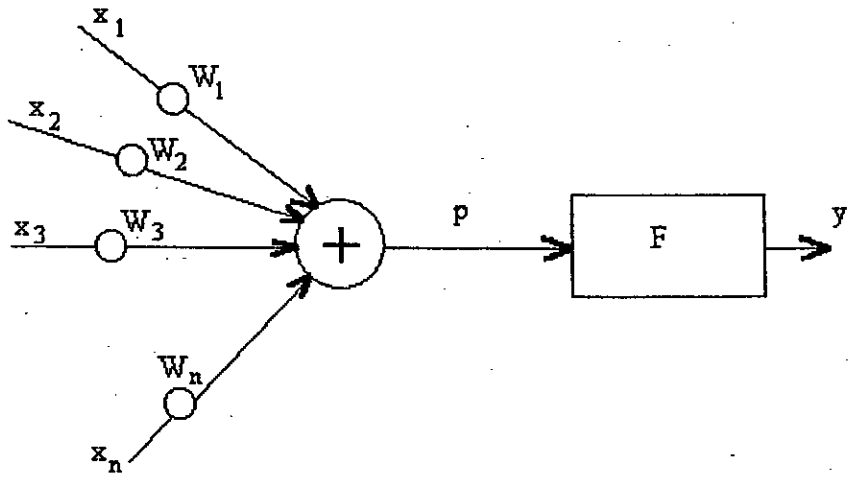


Fig.1.2 : Modèle du neurone de McCULLOCH et PITTS.

On note :

$(x_i)_{i=1,...,n}$:les entrées du neurone formel.
y : sa sortie.
β : seuil de la fonction d'activation.
W_i : les paramètres de pondération (poids synaptiques)
F : la fonction de seuillage.

Les entrées x_i sont des sorties d'autres neurones ou encore des entrées extérieures.
Les calculs du potentiel P et de la sortie y s'expriment par les relations suivantes :

$$p = \sum_{i=1}^{n} W_i \cdot x_i$$

et $y = F(p)$

avec $\begin{cases} F(p) = 1 & si \quad p > \beta \\ F(p) = 0 & si \quad p \leq \beta \end{cases}$

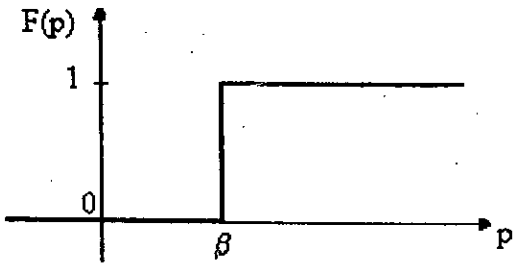


Fig.1.3 : Fonction de seuillage du neurone.

3.1 MODELISATION GENERALE

D'une façon plus générale, on peut définir un neurone formel par les éléments suivants: [2]

- La nature de ses entrées. Elles peuvent être binaires (0,1), bipolaires (1,-1), ou réelles.
- La fonction d'entrée totale qui définit le prétraitement effectué sur les entrées.
- La fonction d'activation (ou d'état) qui détermine l'état interne du neurone en fonction de la valeur de sa fonction d'entrée.
- La fonction de sortie qui calcule la sortie du neurone en fonction de son état d'activation.

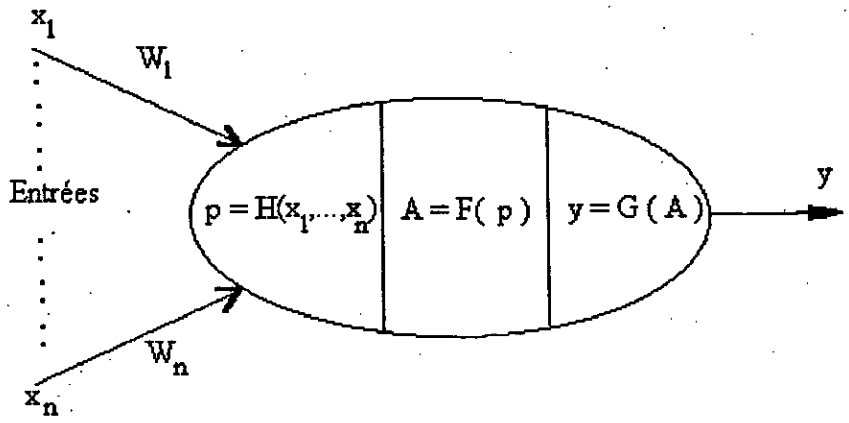


Fig.1.4 : Modélisation générale d'un neurone

On adopte les notations suivantes :

$(x_i)_{i=1...n}$: les entrées du neurone.

$(W_i)_{i=1...n}$: les poids synaptiques du neurone.

H : la fonction d'entrée totale , ou "potentiel" du neurone.

F : la fonction d'activation.

G : la fonction de sortie.

Et $P = H(x_1, ..., x_n)$: Entrée totale du neurone, ou potentiel du neurone.

$A = F(P)$: état du neurone.

$y = G(A)$: sortie du neurone.

La fonction d'entrée totale : H peut être :

- linéaire : $H(x_1, ..., x_n) = \sum_{i=1}^{n} W_i \cdot x_i$
- affine : $H(x_1, ..., x_n) = \sum_{i=1}^{n} W_i \cdot x_i + b$

Le dernier cas est le plus fréquents.

Si la fonction H est affine; le terme constant b peut être entré sous le signe somme. Il correspond à la contribution d'une entrée fictive x_o de valeur constante +1, connectée par un poids W_o qui est précisément ce terme constant (Figure 1.5).

donc : $H(x_1, ..., x_n) = \sum_{i=0}^{n} W_i \cdot x_i \qquad (x_0 = 1, W_0 = b)$

b est appelé "biais" du neurone. Par la suite, nous supposerons que cette entrée x_o existe toujours.

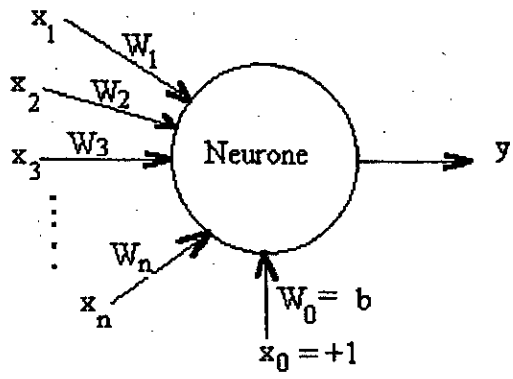


Figure.1.5 : Neurone formel avec biais.

La fonction d'activation :

La fonction d'activation F peut prendre différentes formes.

Elle peut être :

- Une fonction linéaire (fonction identité) : F (p) = p (Figure 1.6.a).
- Une fonction avec un seuil β, une partie linéaire à pente constante "a" et une valeur de saturation A_{max} (Figure 1.6.b).

$$F(p) = \begin{cases} 0 & si \quad p \le \beta \\ a.p & si \quad \beta < p \le \dfrac{A_{max}}{a} \\ A_{max} & si \quad p > \dfrac{A_{max}}{a} \end{cases}$$

- Une fonction signe (valeurs -1 ou +1, Figure 1.6.c) ou une fonction échelon (valeur 0 ou +1, Figure 1.6.d) avec éventuellement un seuil β.

$$F(p) = Sgn(p - \beta)$$

ou

$$F(p) = U(p - \beta)$$

- Une fonction d'allure "sigmoïde". C'est une fonction dont la forme générale est celle d'une tangente hyperbolique avec des valeurs comprises entre -1 et +1 (Figure 1.6.e) ou avec des valeurs comprises entre 0 et +1. (Figure 1.6.f). La pente à l'origine est contrôlée par un paramètre T. Elle augmente si T tend vers 0. Ces fonctions sigmoïdales tendent alors vers les fonctions signe et échelon.

$$F(p) = \frac{\exp\left((p-\beta)/T\right) - 1}{\exp\left((p-\beta)/T\right) + 1} \qquad \in]-1, +1[$$ c'est la fonction tangente hyperbolique

ou

$$F(p) = \frac{\exp\left((p-\beta)/T\right)}{\exp\left((p-\beta)/T\right) + 1} \qquad \in]0, +1[$$ c'est la fonction sigmoïde.

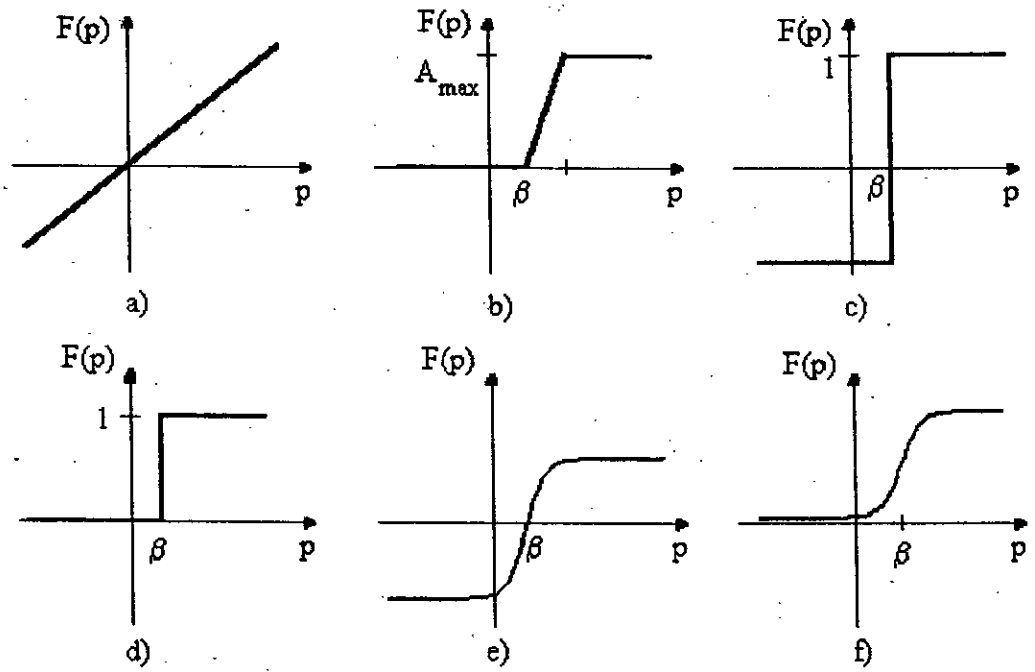


Fig.1.6 : Quelques exemples de fonctions d'activations.

La fonction de sortie :

En général, cette fonction G est considérée comme la fonction identité, donc:

$y = F(p) = A$.

Par la suite on confondra toujours activation et sortie du neurone.

4. LES MODELES DE RESEAUX

L'association de plusieurs neurones formels connectés entre eux suivant différentes structures est appelée réseaux de neurones. C'est en 1958 que Rosenblatt décrivit le premier modèle opérationnel de réseaux de neurones : le Perceptron [1] . C'est un réseau mono-couche, inspiré du système visuel, capable d'apprendre à calculer certaines fonctions logiques en modifiant ces connexions synaptiques.

L'architecture des réseaux de neurones peut aller d'une connectivité totale (tous les neurones sont reliés entre eux) à une connectivité locale où les neurones ne sont reliés qu'à leurs plus proches voisins. On utilise le plus souvent des réseaux à structure régulière pour faciliter leur utilisation.

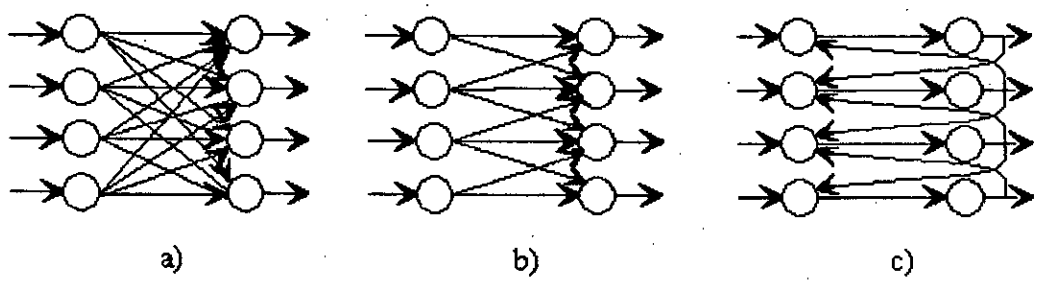


Fig.1.7. Modèles de réseaux à connexions:
a) Totales entre 2 couches, b) Locales, c) récurrente.

Dans un réseau constitué de plusieurs neurones, chaque neurone est repéré par son numéro d'ordre k. Son entrée totale et sa sortie sont notées respectivement p_k et y_k et les poids sont à double indice W_{ki} , le premier indice correspondant au neurone cible k le second, au neurone émetteur i (Figure 1.8).

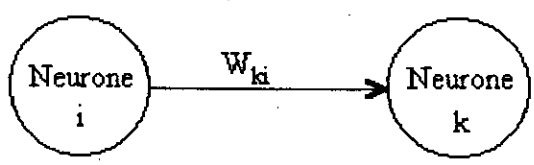


Fig.1.8 : Notation des poids synaptiques entre neurones.

Pour un réseau de K neurones à N entrées, le calcul de la sortie d'un neurone se fait alors à l'aide des relations suivantes :

$$p_k = \sum_{i=0}^{N} W_{ki}.x_i \quad \text{,avec k = 1 à K}$$

$$\text{et } y_k = F(p_k) \quad \text{,avec k = 1 à K}$$

on peut utiliser une notation matricielle :

$\mathbf{p} = \mathbf{W}\,\mathbf{x}$ et $\mathbf{y} = F(\mathbf{p})$.

Avec les vecteurs : $\mathbf{x} = \{ x_n \}^T = \{ x_1 , x_2 , ..., x_N \}^T$, $\mathbf{p} = \{ p_k \}^T$, $\mathbf{y} = \{ y_k \}^T$ et la matrice des poids $\mathbf{W} = \{ W_{kn} \}$. $n = 1, ..., N$ et $k = 1, ..., K$.

Il existe deux modèles classiques pour les réseaux de neurones [2] :

- Les réseaux à couches (Figure 1.9.a).
- Les réseaux entièrement connectés (figure 1.9.b).

1) Dans les réseaux à couches, le réseau de neurones est partagé en plusieurs couches, tel que les neurones qui appartiennent à une même couche ne soient pas connectés entre eux. On définit un sens préférentiel du transfert de l'information : chacune des couches recevant des signaux de la couche précédente et transmettant le résultat de ses traitements à la couche suivante. La première couche correspond à la couche qui reçoit ses entrées du milieu extérieur, et la dernière couche fournit le résultat des traitements effectués. Les couches intermédiaires sont appelées couches cachées. Leur nombre est variable.

2) Dans les réseaux entièrement connectés, chaque neurone est relié à tous les autres et possède même un retour sur lui-même. L'évolution de ces réseaux est dépendante du temps. C'est pour cela qu'ils sont appelés "réseaux dynamiques", contrairement aux réseaux à couches qui sont des "réseaux statiques".

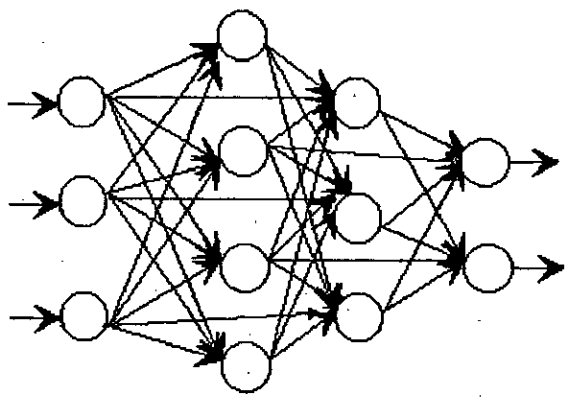


Fig.1.9.a : Réseaux à couches.

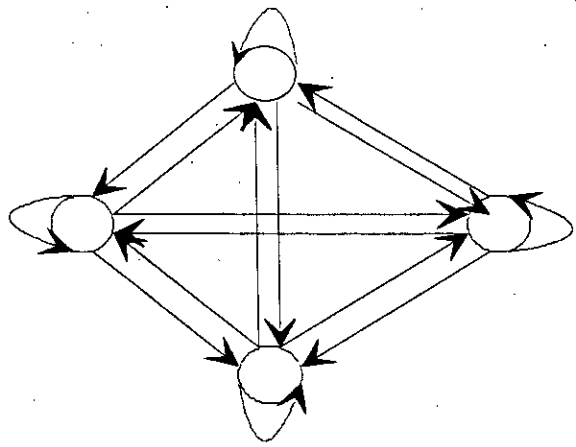


Fig.1.9.b : Réseaux entièrement connectés.

5. REGLES DE PLASTICITE SYNAPTIQUE

C'est en 1949 que HEBB a proposé le premier énoncé d'une règle qualitative régissant la plasticité synaptique [2]. Le point fondamental de cette règle est que le renforcement synaptique intervient lorsqu'il y a activité conjointe du neurone pré-synaptique (cause) et du neurone post-synaptique (effet). Chaque neurone présente deux états possibles : actif ou inactif. L'efficacité de la synapse augmente, d'après HEBB, si les deux neurones sont actifs simultanément. Le tableau suivant schématise cette règle.

Neurone pré-synaptique	Neurone post-synaptique	Efficacité synaptique
Actif	Actif	renforcement
Inactif	Actif	pas de modification
Actif	Inactif	" "
Inactif	Inactif	" "

Tableau 1 : Règle de plasticité de HEBB.

On remarque que la règle de HEBB prévoit exclusivement le renforcement des efficacités synaptiques : le poids de la synapse ne peut qu'augmenter. Ce qui conduirait, en l'absence de phénomènes limitants, à la saturation des réseaux.

5.1 MODELISATION DE LA REGLE DE HEBB

Soit $W_{ij}(t)$ le poids de la connexion entre le neurone j et le neurone i à l'instant t, et A_i et A_j les activations des neurones i et j. La règle de HEBB s'écrit :

$W_{ij}(t+\Delta t) = W_{ij}(t) + \eta A_i A_j$ avec $\eta > 0$.

η est un paramètre positif reflétant l'intensité de l'apprentissage.

Pour pallier a l'inconvénient de cette règle, RAUSCHECKER et SINGER [1], ont proposé les 3 règles suivantes (Tableau 2) :

1) l'efficacité des synapses augmente à chaque fois que les éléments pré- et post-synaptique sont actifs simultanément.

2) l'efficacité d'une synapse décroît si le neurone post-synaptique est actif, tandis que le neurone pré-synaptique est inactif.

3) l'efficacité synaptique décroît plus lentement (oubli) indépendamment de l'activité pré-synaptique, si le neurone post-synaptique est inactif.

Neurone pré-synaptique	Neurone post-synaptique	Efficacité synaptique
Actif	Actif	renforcement
Inactif	Actif	Décroissance
Actif	Inactif	Décroissance lente
Inactif	Inactif	" "

Tableau 2 : Règle de plasticité de RAUSCHECKER et SINGER.

6. APPRENTISSAGE DES RESEAUX DE NEURONES

L'apprentissage est l'opération par laquelle le réseau de neurones acquiert la capacité de faire une tâche déterminée en modifiant ses paramètres internes (poids des connexions) en utilisant un algorithme d'adaptation appelé algorithme d'apprentissage.

Il existe deux classes de règles d'apprentissage :

- règles d'apprentissages à source d'inspiration biologique : Règle de HEBB, règle de RAUSCHECKER et SINGER.
- règles d'apprentissages à source d'inspiration mathématique : le réseau est considéré comme une fonction de transfert des entrées qui lui sont présentées. Pour calculer les paramètres de cette fonction de transfert (les poids et les biais de chaque neurone du réseaux), on utilise des algorithmes qui dépendent de la structure du réseau utilisé : réseau statique ou réseau dynamique.

6.1 RESEAUX STATIQUES

L'apprentissage des réseaux statiques peut être divisé en deux parties: apprentissage supervisé et non supervisé.

6.1.1 APPRENTISSAGE SUPERVISE

L'apprentissage supervisé implique l'existence d'un "professeur" qui a pour rôle d'évaluer le succès ou l'échec du réseau quand on lui présente un stimulus connu (on dit que ce stimulus est un exemple appartenant à la base d'apprentissage). Cette supervision consiste à renvoyer au réseau une information, lui permettant de faire évoluer ses connexions (parfois aussi sa propre architecture) afin de diminuer son taux d'échec.

Le plus souvent cette information sera l'erreur entre la sortie du réseau et la sortie désirée.

6.1.1.1 REGLE DE WIDROW-HOFF

L'apprentissage du perceptron (réseau mono-couche) est un apprentissage supervisé qui se fait par correction d'erreur. On utilise pour cela la règle de WIDROW-HOFF. C'est une méthode de minimisation de l'erreur par descente de gradient [3].

On considère un réseau constitué de n neurones recevant des vecteurs à K composantes (Figure 1.10).

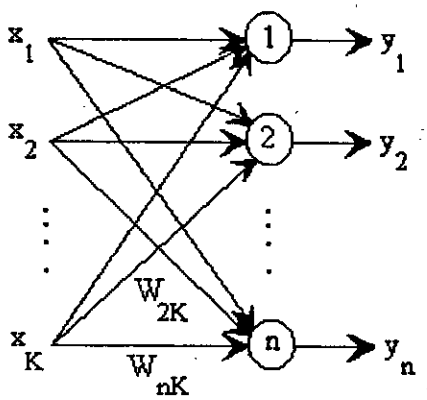


Fig.1.10 : Réseau à une couche.

les K entrées x_j du réseau sont distribuées sur tous les neurones. La sortie du neurone i vaut :

$$y_i = F(p_i) = F\left(\sum_{j=1}^{K} W_{ij} \cdot x_j + b_i\right)$$

où b_i est le biais du neurone i.

A un vecteur d'entrée **x**, on veut associer un vecteur de sortie désirée **Yd**. Si les poids W_{ij} ont des valeurs quelconques, le vecteur de sortie **y** observé est différent de **Yd**. On associe à cette différence l'erreur quadratique :

$$E = \frac{1}{2}\sum_{j=1}^{n}\left(Yd_j - y_j\right)^2$$

on calcule le gradient de cette erreur par rapport à W_{ik} :

$$\frac{\partial E}{\partial W_{ik}} = \sum_{j=1}^{n}\left(Yd_j - Y_j\right)\frac{\partial\left(Yd_j - Y_j\right)}{\partial W_{ik}}$$

en remplaçant y_j par son expression, on trouve :

$$\frac{\partial E}{\partial W_{ik}} = -\left(Yd_i - y_i\right)x_k \cdot F'(p_i)$$

ou F' est la dérivée de F.

En posant $\delta_i = \left(Yd_i - y_i\right)$, le gradient de l'erreur s'écrit :

$$\frac{\partial E}{\partial W_{ik}} = -\delta_i\, x_k \cdot F'\left(\sum_{j=1}^{K} W_{ij} \cdot x_j + b_i\right)$$

Pour diminuer l'erreur , pour l'exemple **x**, il faut calculer un ΔW_{ik} dans le sens opposé à ce gradient, soit donc :

$\Delta W_{ik} = \eta \delta_i x_k F'(p_i)$ k=1,....,K et i=1,....,n.
et $W_{ik}(t+1) = W_{ik}(t) + \Delta W_{ik}$

Le coefficient d'apprentissage η joue un rôle important puisqu'il règle la vitesse avec laquelle se fait la descente du gradient. Trop petit, il ne permet d'atteindre une valeur suffisamment faible de l'erreur qu'après un très grand nombre de pas et on risque même de tomber dans un minimum local dont il est impossible de sortir. Trop grand, il peut conduire à s'éloigner du minimum d'erreur recherché.

L'apprentissage prend fin sur décision de l'utilisateur : après un nombre fixé d'itérations ou lorsqu'on a atteint une erreur acceptable sur les sorties du réseau.

6.1.1.2 RETROPROPAGATION DU GRADIENT

L'incapacité des réseaux mono-couches (Perceptrons) à résoudre des problèmes simples, notamment la classification de données en plusieurs classes, a entraîné un ralentissement de la recherche sur les réseaux de neurones. Ce n'est que dans le courant des années 70, que des travaux démontrèrent qu'en ajoutant une ou plusieurs couches entre la couche de neurones d'entrée et la couche de neurones de sortie, on est certain de pouvoir approximer à la sortie du réseau n'importe quelle fonction de l'espace d'entrée. Mais la théorie ne donne aucune indication sur le nombre et la taille de ces couches intermédiaires (appelées couches cachées). Ce nombre devant être d'autant plus grand que la fonction à implémenter est plus irrégulière. Dans la quasi-totalité du cas, on constate qu'une couche cachée est suffisante [1].

L'apprentissage d'un réseau multicouche ne peut pas être réalisé par la règle de WIDROW-HOFF. En effet, si cette méthode est directement applicable pour ajuster les poids de la dernière couche, elle ne l'est pas pour ceux des couches internes, car on ne connaît pas la sortie désirée pour ces couches et par conséquent, on ne connaît pas directement le terme d'erreur associé à chaque couche interne. Il faut donc tenter d'exprimer l'erreur à la sortie de chaque neurone d'une couche quelconque à partir de l'erreur de la dernière couche; seule erreur directement mesurable.

La solution à ce problème a été apportée par l'algorithme de "rétropropagation du gradient d'erreur" , appelé "backpropagation algorithm" dans la littérature anglaise [4].

Considérons le réseau à trois couches de la Figure 1.11 : le réseau comporte une couche d'entrée, une couche cachée et une couche de sortie.

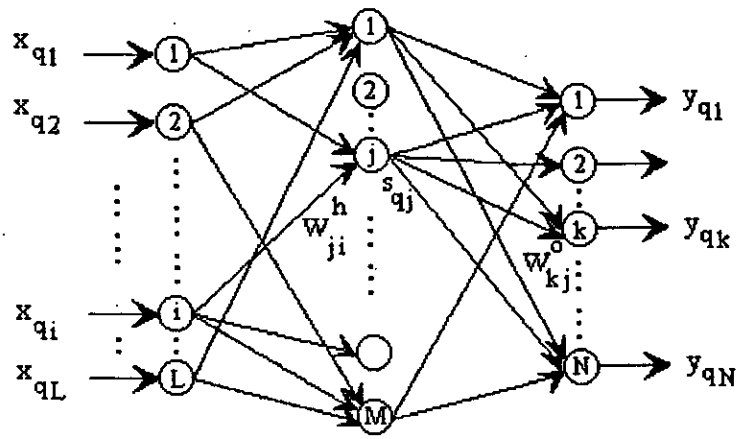


Fig.1.11 : Réseau de neurones multicouches.

On considère que la couche d'entrée ne fait que transmettre les entrées : le réseau ne comporte que deux couches réelles de traitement : la couche interne et la couche de sortie.

On adopte les notations suivantes :

- $\mathbf{x}_q = (x_{q1}, x_{q2}, \ldots, x_{qL})$: vecteur d'entrée, où q est l'indice de l'exemple. q=1,...,n. n représente de nombre d'exemples de la base d'apprentissage.
- $\mathbf{y}_q = (y_{q1}, y_{q2}, \ldots, y_{qN})$: vecteur de sortie du réseau en réponse à l'entrée $\mathbf{x}_q$.
- $\mathbf{yd}_q = (yd_{q1}, yd_{q2}, \ldots, yd_{qN})$: vecteur de sortie désiré pour l'exemple $\mathbf{x}_q$.
- $s_q = (s_{q1}, s_{q2}, \ldots, s_{qM})$: vecteur de sortie des M neurones de la couche cachée.

- W^h_{ji} : poids synaptique entre le $i^{éme}$ neurone de la couche d'entrée et le jéme neurone de la couche cachée.
- W^o_{kj} : poids synaptique entre le $j^{éme}$ neurone de la couche cachée et le $k^{éme}$ neurone de la couche de sortie.
- b^h_j : biais du neurone j de la couche cachée.
- b^o_k : biais du neurone k de la couche de sortie.
- p^h_{qj} : l'entrée totale du neurone j de la couche cachée, pour l'exemple q.
- p^o_{qk} : l'entrée totale du neurone k de la couche de sortie, pour l'exemple q.
- F^h : fonction d'activation de la couche cachée.
- F^o : fonction d'activation de la couche de sortie.

Avec : i = 1,...,L ; j = 1,...,M et k = 1,...,N.

L'algorithme d'apprentissage par rétropropagation du gradient est alors [4] :

1. choix de la taille du réseau. Initialisation aléatoire des poids et des biais des neurones à de petites valeurs.
2. on applique le vecteur $\mathbf{x_q}$ à la couche d'entrée. $\mathbf{x_q}$ est choisit aléatoirement dans la base d'apprentissage.
3. on calcule l'entrée totale de chaque neurone de la couche cachée :
$$p^h_{qj} = \sum_{i=1}^{L} W^h_{ji} \cdot x_{qi} + b^h_j$$
4. on calcule la sortie de la couche cachée.
$$s_{qj} = F^h\left(p^h_{qj}\right)$$
5. on calcule l'entrée totale des neurones de la couche de sortie:
$$p^o_{qk} = \sum_{j=1}^{M} W^o_{kj} \cdot s_{qj} + b^0_k$$
6. calcul de sortie du réseau : $y_{qk} = F^o\left(p^o_{qk}\right)$
7. on calcule sur la couche de sortie le terme :
$$\delta^o_{qk} = \left(yd_{qk} - y_{qk}\right) \cdot F^{o'}\left(p^o_{qk}\right)$$ ou $(yd_{qk} - y_{qk})$ est l'erreur commise par le neurone k.
8. calcul de l'erreur commise sur la couche cachée :
$$\delta^h_{qj} = F^{h'}\left(p^h_{qj}\right) \cdot \sum_{k=1}^{N} \delta^o_{qk} \cdot W^o_{kj}$$
9. mise à jours des poids par :
$W^h_{ji}(t+1) = W^h_{ji}(t) + \eta \delta^h_{qj} x_{qi}$ sur la couche cachée, et
$W^o_{kj}(t+1) = W^o_{kj}(t) + \eta \delta^o_{qk} s_{qj}$ sur la couche de sortie.
Où η est le coefficient d'apprentissage.
Et des biais par :
$b^h_j(t+1) = b^h_j(t) + \eta \delta^h_{qj}$ sur la couche cachée, et
$b^o_k(t+1) = b^o_k(t) + \eta \delta^o_{qk}$ sur la couche de sortie.
10. si le test d'arrêt n'est pas vérifié, retourner à l'étape 2.

Le test d'arrêt est l'erreur quadratique totale qu'on veut atteindre : $E = \frac{1}{2}\sum_{q=1}^{n} E_q^2$, ou E_q est l'erreur entre la sortie $\mathbf{y_q}$ du réseau et la sortie désirée $\mathbf{yd_q}$.

ASPECTS PRATIQUES DE L'ALGORITHME

L'algorithme proposé précédemment est rarement utilisé tel quel en pratique. En effet, c'est un algorithme de type gradient stochastique : on doit présenter les exemples dans un ordre aléatoire, et la mise à jour des poids a lieu à la suite de la présentation de chaque exemple. On minimise donc une erreur instantanée; ce qui conduit à une vitesse d'apprentissage assez lente.

Pour remédier à ce problème, on accumule les erreurs de chaque exemple et on met les poids à jour après la présentation de tous les exemples de la base d'apprentissage : c'est l'algorithme du gradient global [1, 5].

L'algorithme modifié devient alors :

1. Initialisation aléatoire des poids et des biais des neurones à des petites valeurs (entre -0.5 et 0.5).
2. q=1.
3. on applique le vecteur d'entrée $\mathbf{x}_q$.
4. calcul de l'entrée totale de chaque neurone de la couche cachée:

$$p_{qj}^{h} = \sum_{i=1}^{L} W_{ji}^{h} \cdot x_{qi} + b_j^{h}$$

5. calcul de la sortie de la couche cachée: $s_{qj} = F^{h}\left(p_{qj}^{h}\right)$
6. calcul de la sortie des neurones de la couche de sortie :

$$p_{qk}^{o} = \sum_{j=1}^{M} W_{kj}^{o} \cdot s_{qj} + b_k^{0} \quad \text{et} \quad y_{qk} = F^{o}\left(p_{qk}^{o}\right).$$

7. on calcule sur la couche de sortie le terme:

$$\delta_{qk}^{o} = \left(yd_{qk} - y_{qk}\right) \cdot F^{o'}\left(p_{qk}^{o}\right)$$

8. calcul de l'erreur de la couche cachée :

$$\delta_{qj}^{h} = F^{h'}\left(p_{qj}^{h}\right) \cdot \sum_{k=1}^{N} \delta_{qk}^{o} \cdot W_{kj}^{o}$$

9. si $q < n$, alors $q = q+1$ et retourner à l'étape 3.
10. mise à jours des poids et des biais:

$$W_{ji}^{h}(t+1) = W_{ji}^{h}(t) + \eta \sum_{q=1}^{n} \delta_{qj}^{h} x_{qi}$$

pour la couche cachée, et :

$$b_j^{h}(t+1) = b_j^{h}(t) + \eta \sum_{q=1}^{n} \delta_{qj}^{h}$$

$$W^0_{kj}(t+1) = W^0_{kj}(t) + \eta \sum_{q=1}^{n} \delta^o_{qk}\, s_{qi}$$

pour la couche de sortie.

$$b^o_k(t+1) = b^o_k(t) + \eta \sum_{q=1}^{n} \delta^o_{qk}$$

11. si le test d'arrêt n'est pas vérifié, retourner à l'étape 2.

Cet algorithme peut être appliqué à un réseau comportant un nombre quelconque de couches cachées.

a) Coefficient d'apprentissage

Le coefficient d'apprentissage η ne doit pas être trop grand sinon il entraînerait des oscillations de l'erreur autour d'un minimum qu'on ne pourra pas atteindre et si η est trop petit, le temps d'apprentissage sera très grand. Une solution a ce problème consiste à utiliser un coefficient d'apprentissage adaptatif [5] : si l'erreur à l'itération (t) dépasse l'erreur à l'itération (t-1) d'un rapport c fixé (appelé rapport d'erreur), alors η est diminué par multiplication par une constante inférieure à 1. Si l'erreur à l'itération (t) est inférieure à l'erreur à l'itération (t-1), alors η est augmenté par multiplication par une constante supérieure à 1:

- Si Erreur(t) > c * Erreur(t-1) alors $\eta = \eta * (\eta\_déc)$ avec $\eta\_déc < 1$.
- Si Erreur(t) ≤ Erreur(t-1) alors $\eta = \eta * (\eta\_inc)$ avec $\eta\_inc > 1$.

Cette procédure accélère fortement le temps d'apprentissage du réseau.

b) Momentum

Un moyen efficace pour accélérer l'apprentissage et aussi pour pouvoir sortir des minimums locaux de la surface d'erreur est d'utiliser un terme "d'inertie" dans la correction des poids [1, 5] dans lequel on tient compte de la correction des poids à l'étape précédente.

On a vu que la correction des poids se faisait par la relation :
$W_{ij}(t+1) = W_{ij}(t) + \Delta W_{ij}(t+1)$.
où la formule de $\Delta W_{ij}(t+1)$ dépend de la couche sur laquelle porte les corrections.
Avec la technique du momentum, $\Delta W_{ij}(t+1)$ devient :

$$\Delta W_{ij}(t+1)_{|\text{avec momentum}} = \gamma \,.\, \Delta W_{ij}(t) + (1-\gamma) \,.\, \Delta W_{ij}(t+1)_{|\text{sans momentum}}$$

Le paramètre γ est le momentum. Il est variable suivant l'évolution de l'erreur du réseau

Comme pour le cas du coefficient d'apprentissage, on définit un rapport c entre l'erreur à l'itération (t) et l'erreur à l'itération (t-1) :

- Si Erreur(t) > c * Erreur(t-1) alors $\gamma = 0$.
- Si Erreur(t) ≤ Erreur(t-1) alors γ est égal à une constante inférieure à "1". Généralement γ est très proche de 1.

L'utilisation dans l'algorithme d'apprentissage d'un coefficient d'apprentissage adaptatif, et du momentum assure une convergence rapide vers une erreur minimale, et permet d'éviter les minimums locaux.

6.1.2 APPRENTISSAGE NON SUPERVISE

L'apprentissage non supervisé implique la fourniture à un réseau une quantité suffisante d'exemples contenant des corrélations (autrement dit de la redondance), tel que celui-ci en dégage les régularités automatiquement. Ces réseaux sont appelés "auto-organisateur" ou à "apprentissage compétitif".

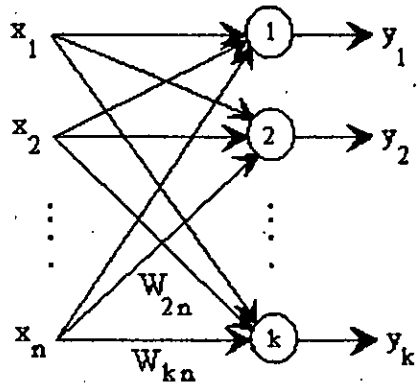
Ce type d'apprentissage possède souvent une moindre complexité dans les calculs en comparaison avec l'apprentissage supervisé.

6.1.2.1 CARTES AUTO-ORGANISATRICES DE KOHONEN

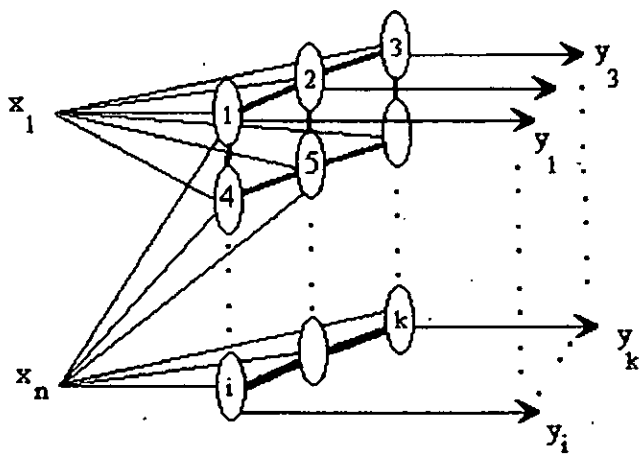
Ce modèle a été présenté par KOHONEN en 1982 en se basant sur des constatations biologiques [1,2]. Il a pour objectif de représenter des données complexes et appartenant généralement à un espace de grande dimension, dans un espace discret dont la topologie est limitée à 1 ou 2 dimensions. Il s'agit donc d'un modèle de quantification vectorielle (V. Q).

a) LE RESEAU

Les cartes organisatrices de KOHONEN sont réalisées à partir d'un réseau de k neurones à n entrées. Le réseau possède ainsi k sorties. Ces neurones sont connectés suivant une structure mono ou bidimensionelle. chaque neurone possède donc des voisins dans cette structure (Figure 1.12). Les entrées sont des vecteurs à n composantes; toutes totalement connectées aux k neurones du réseau par n*k connexions modifiables.



a)



b)

Fig.1.12 : Réseau auto-organisateur de KOHONEN.
a) structure mono-dimensionnelle b) structure bidimensionnelle

b) L'ALGORITHME

On suppose que les poids des connexions modifiables sont initialement aléatoire. Considérons un vecteur d'entrée **x**. La sortie du réseau est le vecteur **y** à k composantes :

$$y_i = \sum_{j=1}^{n} W_{ij} \cdot x_j = \mathbf{x} \,.\, \mathbf{W_i}^T$$

Dans cette relation on note $\mathbf{W_i}$ le vecteur poids du neurone i, c'est à dire, le vecteur à n composantes { W_{i1} , W_{i2} , ...,W_{in} } . Selon le vecteur **x** et la configuration initiale des poids, il existe un neurone i_0 dont la sortie est la plus grande. Ce neurone sera appelé neurone gagnant.

On considère que dans un voisinage de i_0 , les neurones i sont actifs ($y_i = 1$), et qu'ailleurs ils sont inactifs ($y_i = 0$).

La règle d'adaptation des poids est :

$$\Delta \mathbf{W}_i = \eta(t)\, G(i, i_0)\, (\mathbf{x} - \mathbf{W}_i) \qquad , \quad \eta(t) \text{ est le coefficient d'apprentissage.}$$

Pour obtenir une convergence presque sure, Le coefficient d'apprentissage $\eta(t)$ est pris variable avec le temps. Son évolution est en 1/t [1].

$G(i, i_0)$ est la fonction de voisinage. Elle est maximale pour le neurone gagnant i_0 , et décroît plus ou moins rapidement sur les neurones voisins, devient négative au-delà, pour s'annuler loin du neurone gagnant. Cette fonction d'inhibition latérale, qui a la forme représentée sur la Figure 1.13 , est dite "en chapeau mexicain" [1, 3]. Mais on peut par simplification la représenter par une fonction constante sur une certaine distance, nulle au-delà (Figure 1.13.b). Sa largeur est un paramètre de l'apprentissage : grande au début ceci afin d'ordonner rapidement le réseau, puis de plus en plus faible par la suite , elle fixe la carte auto-organisatrice dans sa configuration finale. En pratique cette évolution est en 1/t . Dans le cas le plus simple on utilise : $G(i, i_0) = \delta_{i\,i0}$ où $\delta_{i i_0} = \begin{cases} 1 & si\ i = i_0 \\ 0 & \text{sinon} \end{cases}$

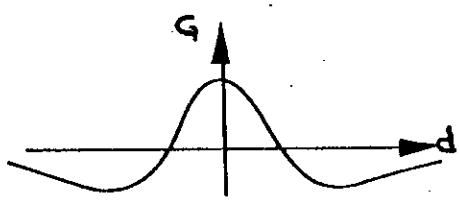


Fig.1.13 : Fonction d'inhibition latérale.

L'algorithme se résume à 7 étapes:

1. Choix du nombre k de neurones et de la structure du réseau.
2. Initialisation du voisinage et du pas d'adaptation $\eta(t)$. Initialisation aléatoire des poids.
3. Tirage aléatoire d'une entrée **x** dans la base d'apprentissage .
4. Recherche du neurone i_0 le "plus proche" (le gagnant).
5. Modification des poids $\mathbf{W}_i$ pour i tel que $G(i, i_0) \neq 0$ ($i \in$ voisinage de i_0).
6. Modification du voisinage $G(i, i_0)$ et du pas d'adaptation $\eta(t)$.
7. Retour à l'étape 3 sauf si le test d'arrêt (le nombre d'itérations) est vérifié.

Le résultat de l'apprentissage est que chaque neurone soit représentatif d'un certain type d'entrées, et que des neurones voisins dans le réseau tendent à représenter des domaines voisins de l'espace d'entrée.

A la convergence de l'algorithme, les incréments $\Delta \mathbf{W}_i$ sont en moyenne nuls. Ce qui signifie, qu'en moyenne , $\mathbf{W}_i$ tend vers $\mathbf{x}$. Or lorsque le voisinage se réduit le poids $\mathbf{W}_i$ n'est mis à jour que pour les vecteurs $\mathbf{x}$ qui sont les plus proches de $\mathbf{W}_i$. Si on représente l'espace d'entrée en domaine R_i (représentation dite de VORONOI), alors le vecteur poids $\mathbf{W}_i$ du neurone i est le barycentre des vecteurs $\mathbf{x}$ du domaine R_i que ce neurone représente.

6.2 RESEAUX DYNAMIQUES

6.2.1 MODELE DE HOPFIELD

C'est une architecture simple qui possède certaines fonctionnalité de type cognitif (mémoire associative).

La mémoire associative est un dispositif capable de mémoriser des informations que l'on peut ensuite retrouver, non par leur adresse comme dans un mémoire classique, mais en fournissant des données mêmes incomplètes ou bruitées relatives aux informations stockées : ce type de mémoire s'appelle aussi Mémoire Adressable par le Contenu (CAM) [1].

Dans le modèle de HOPFIELD, un neurone est décrit par une variable y, qui est la sortie du neurone, qui peut prendre deux états -1 ou +1. L'état d'un réseau de N neurones (Figure 1.14) est donc décrit par un vecteur binaire $y \in \{-1,+1\}^N$ tous les neurones sont connectés entre eux. Il en résulte qu'il y a N^2 connexions portant des poids. Les interconnexions entre les neurones sont symétriques.

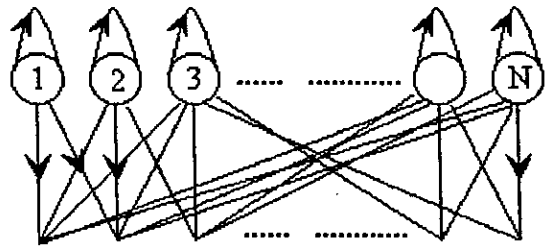


Fig.1.14 : Modèle de Hopfield.

Les influences mutuelles des neurones sont contenues dans la matrice des poids **W**. Comme les interconnexions entre les mesures sont symétriques, la matrice **W** est symétrique : $W_{ij} = W_{ji}$.

L'apprentissage consiste à calculer les coefficients synaptique W_{ij} connaissant L vecteurs $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_L$ de dimension N dont on veut faire des états stables du réseau. On appelle ces vecteurs les prototypes appris par le réseau. La matrice des poids est donnée par la relation suivante [4] :

$$W = \sum_{i=1}^{L} xi.x_i^T$$

a) DYNAMIQUE DU RESEAUX

On procède de la manière suivante pour faire évoluer l'état **y** à l'instant (t) du réseau:[3]

- on choisit un neurone k :
- on calcul le potentiel $p_k(t)$ du neurone k.

$$p_k(t) = \sum_{j=1}^{N} W_{kj} \cdot y_j(t)$$

- l'état du neurone k est alors mis à jours selon le signe du potentiel synaptique :

$$y_k(t+1) \longleftarrow Sign(p_k(t)) \qquad \text{donc} \quad y_k(t+1) = \begin{cases} +1 \\ -1 \end{cases} ou$$

- On choisit une nouvelle valeur de k et on recommence.

Au cours de ces itérations, les états des neurones finissent par se stabiliser à des valeurs qui dépendent des états initiaux et de la matrice des poids W.

Un neurone stable lorsque : $y_j(t+1) = y_j(t)$

b) FONCTION D'ENERGIE

On considère l'état **y** du réseau constitué par l'ensemble des sorties des neurones $\mathbf{y} = (y_1, y_2, ..., y_N)$. On peut associer à l'état **y** une énergie totale E(y) [1, 4]:

$$E(\mathbf{y}) = -\frac{1}{2}\mathbf{p}^T\mathbf{y} = -\frac{1}{2}\left(\sum_{i=1}^{N} p_i \cdot y_y\right) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{k=1}^{N} W_{ik} \cdot y_k \cdot y_i$$

$$E(\mathbf{y}) = -\frac{1}{2}\mathbf{p}^T\mathbf{y} = -\frac{1}{2}(\mathbf{W}\mathbf{y})^T\mathbf{y} = -\frac{1}{2}\mathbf{y}^T\mathbf{W}^T\mathbf{y}.$$

On montre que l'évolution du réseau se fait à énergie décroissante [4].

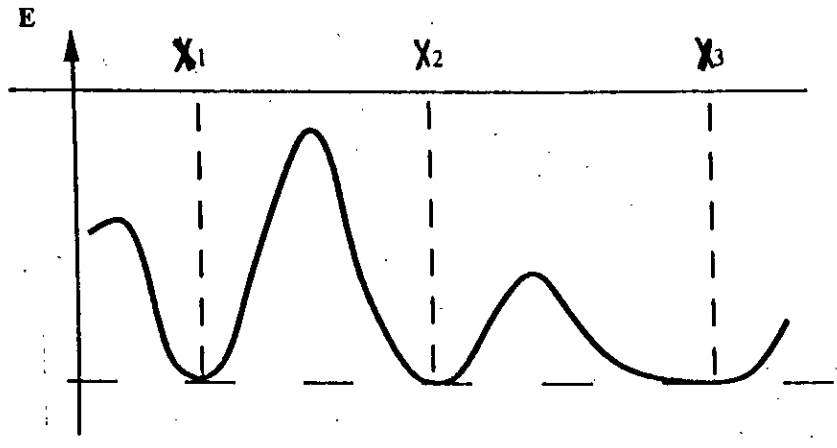


Fig.1.15 : Energie d'un réseau de hopfield.

L'énergie du réseau de HOPFIELD passe par des minima E_{min} lorsque l'état **y** passe par les différents états mémorisés $\mathbf{x}_1$, $\mathbf{x}_2$,.. .

L'utilisation du réseau, après apprentissage, consiste à imposer à chaque neurone une composante d'un vecteur stimulus, et à laisser le réseau évoluer librement (relaxation) sous l'effet de sa dynamique. La règle d'apprentissage garantit d'atteindre un état stable ("attracteur") après un nombre fini d'étapes. Si l'on mesure la distance de Hamming entre cet attracteur et chacun des prototypes appris, on constate que généralement l'attracteur est très voisin ou confondu avec un des prototypes : le réseau se comporte donc comme une mémoire auto-associative, restituant une information apprise sous l'effet d'une stimulation.

On remarque cependant l'apparition quelquefois d'attracteurs ne correspondant à aucun prototype : on les appelle les états parasites. Cela se produit si l'état initial $\mathbf{y} = \{y_1, y_2, ..., y_N\}^T$ du réseau est trop éloigné d'un état mémorisé, ou si un grand nombre de prototypes ont été appris. De nombreuses simulations évaluent à 0,15N le nombre de vecteurs que peut mémoriser un réseau de taille N sans qu'il se trouve piéger dans des minima d'énergie locaux en phase d'utilisation [4].

Le réseau de HOPFIELD peut présenter un comportement indésirable : l'oubli catastrophique. Ainsi , un réseau peut apprendre K exemples sans problèmes, et les "oublier" tous dés lors qu'on souhaite lui en apprendre un de plus [2].

6.3 APPRENTISSAGE ET GENERALISATION

Après la phase d'apprentissage, il faut être en mesure d'estimer le degré de confiance qu'on peut placer dans le réseau. C'est pourquoi on divise la base de données contenant les exemples connus en deux sous-ensembles : la base d'apprentissage proprement dite, à l'aide de laquelle on effectue l'apprentissage, et la base de test sur laquelle on essaie de tester la capacité du réseau à reconnaître des exemples non appris. Cette opération permet d'estimer la capacité de généralisation du réseau, qui est le critère déterminant pour son utilisation effective. Pour cela, il faut respecter 2 conditions :

1. Les exemples utilisés doivent être équitablement représentatif des classes à reconnaître.
2. Il faut que le rapport du nombre d'exemples d'apprentissage au nombre de connexions du réseau soit supérieur à 1 : plus ce rapport est élevé, meilleur sera la généralisation [3].

7. DOMAINES D'APPLICATIONS DES RESEAUX DE NEURONES

Les réseaux de neurones sont bien adaptés à la résolution des problèmes ayant les caractéristiques suivantes :

1. Les règles qui permettent de résoudre le problème sont inconnues ou très difficiles à expliciter.
2. Le problème fait intervenir des données bruitées.
3. Le problème nécessite une grande rapidité de traitement.

Les domaines d'applications suivants possèdent pratiquement toutes les caractéristiques exposées précédemment. Ils constituent les principales applications des réseaux de neurones:

- Reconnaissance des formes.
- Traitement du signal.
- Vision, parole.
- Robotique.
- Prévision et modélisation.

CONCLUSION

L'étude des modèles des réseaux de neurones nous a permis de dégager les principales caractéristiques qui les différencient des méthodes classiques utilisées dans la résolution des problèmes posés par plusieurs domaines : l'intelligence artificielle, le traitement du signal, la reconnaissance de formes.... Ces caractéristiques sont :

- Leur parallélisme intrinsèque.
- Leur capacité d'adaptation (capacité d'apprentissage)
- La mémoire distribuée : la mémorisation de la fonction à réaliser par le réseau est distribuée sur plusieurs neurones, ce qui introduit une résistance au bruit car la perte d'un neurone ne correspond pas à la perte de la fonction mémorisée.
- Leur capacité de généralisation.

Le développement des domaines d'applications des réseaux de neurones a conduit les chercheurs à passer de la simulation numérique de ces réseaux à leur implémentation sur des structures matérielles diverses en vue d'une utilisation effective afin de profiter au mieux de leurs avantages.

Le modèle multicouches est le plus utilisé parmi les différents modèles de réseaux car il permet d'approximer n'importe qu'elle fonction de transfert donnée, après un choix approprié de ses paramètres [1]. De ce fait c'est le modèle le plus intéressant à implémenter.

CHAPITRE 2

METHODES D'IMPLEMENTATION DES RESEAUX DE NEURONES

1. INTRODUCTION

L'une des principales caractéristiques des réseaux de neurones est qu'ils sont massivement parallèles : c'est un ensemble d'entités élémentaires, les neurones, qui peuvent être vus comme des processeurs indépendants aux fonctions très simples :

- Multiplication
- Addition
- Fonction de seuillage

La simulation informatique sur une machine séquentielle d'un réseau de neurones pour une petite application est simple. Mais lorsqu'il s'agit d'applications complexes, comme dans la perception ou l'intelligence artificielle, le grand nombre de données à traiter exige un très large espace mémoire et entraîne rapidement des temps de calcul importants pour la plupart des modèles de réseaux.

C'est seulement en créant de vrais réseaux de processeurs inspirés du modèle des neurones formels que l'on pourra profiter des avantages des réseaux de neurones : leur vitesse de calcul et leur tolérance aux pannes.

Dans ce chapitre, on expose différentes méthodes d'implémentation des réseaux de neurones à partir desquels un choix sera fait quant à la manière d'implémenter, dans notre cas, ces réseaux.

2. METHODES D'IMPLEMENTATION DES RESEAUX DE NEURONES

Il existe deux approches différentes pour la réalisation des réseaux de neurones :

- Approche analogique.
- Approche digitale.

2.1 IMPLEMENTATIONS ANALOGIQUES

La réalisation analogique des réseaux de neurones se fait en exploitant les propriétés fonctionnelles des circuits électroniques de base.

L'amplificateur opérationnel est apparu comme le neurone artificiel par excellence : il est capable de sommer des courants en " parallèle " sur son noeud d'entrée (Fig.2.1), puis de délivrer une tension de sortie qui sera la somme des tensions d'entrées pondérées par des coefficients. Un deuxième bloc sert à réaliser la fonction de seuillage du neurone [3].

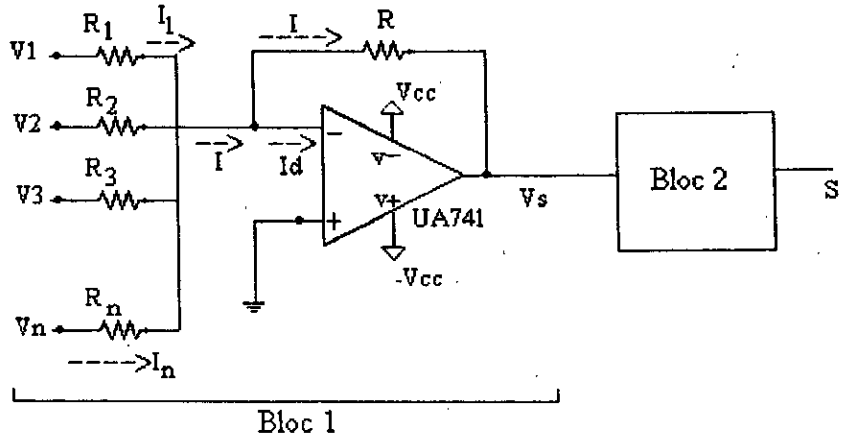


Fig.2.1 : Neurone artificiel réalisé à partir d'amplificateur opérationnels

on a : $i_d \approx 0, \quad i = i_1 + i_2 + ... + i_n$.

$$Vs = -R.\,i \quad \text{et} \quad i_1 = \frac{V_1}{R_1}, \ldots\ldots\ldots\ldots, i_n = \frac{Vn}{R_n}$$

d'où :

$$Vs = -R\left[\frac{1}{R_1}V_1 + \frac{1}{R_2}V_2 + \ldots\ldots\ldots + \frac{1}{R_n}V_n\right]$$

$$Vs = \sum_{k=1}^{n} \frac{-R}{R_k} V_k$$

$(-V_k)_{k=1,2,\ldots,n}$: représentent les entrées du neurone.

$\left(\frac{R}{R_k}\right)_{k=1,2,\ldots,n}$: représentent les poids synaptiques.

2.1.1 EXPLICITATION DU BLOC 2

1.CAS DE FONCTIONS D'ACTIVATIONS A DEUX NIVEAUX:

- **La fonction sign :**

Cette fonction est facilement réalisable par un simple comparateur représenté par la Figure 2.2. La sortie S1 du comparateur varie entre +Vcc et -Vcc. [6]

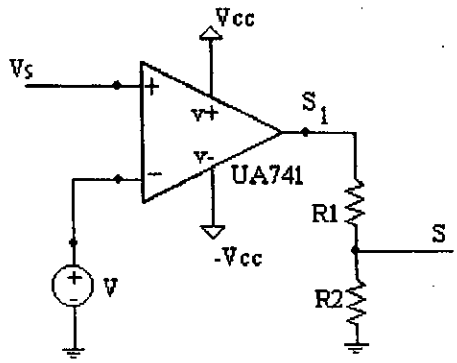


Fig.2.2 : Réalisation de la fonction Sign.

- **La fonction échelon :**

C'est le même principe que pour la réalisation de la fonction Sign. Mais le basculement du comparateur ce fait ici entre +Vcc et 0.

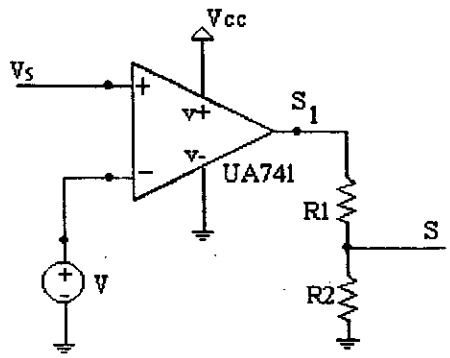


Fig.2.3 : Réalisation de la fonction échelon.

Remarques:

1) La tension de seuil de basculement (ou de transition) V est constante. Elle est égale à la valeur du biais associé au neurone.

2) Le rôle de pont des résistances (R_1 , R_2) est de régler la tension de sortie S (sortie du neurone) à la valeur désirée.

2. CAS DE LA FONCTION D'ACTIVATION LINEAIRE

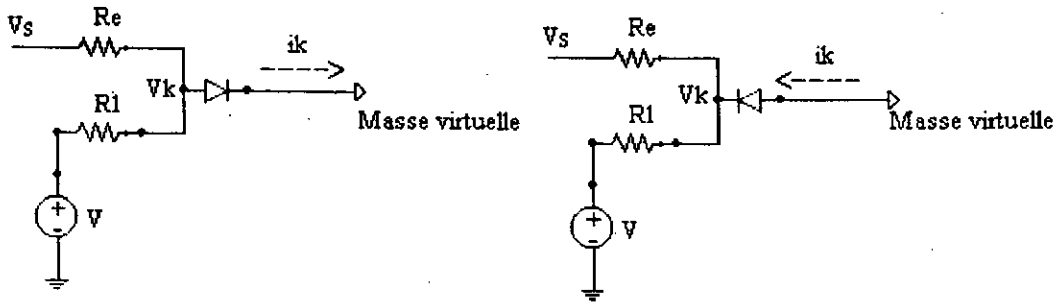
Pour réaliser un neurone avec une fonction d'activation linéaire on a pas besoin du bloc 2 . En effet dans ce cas la sortie du neurone est égale à son potentiel (S = Vs).

3. CAS DES FONCTIONS NON LINEAIRES

On regroupe dans cette catégorie les fonctions sigmoïde et tangente hyperbolique. Pour essayer de reproduire ces fonctions d'activations on peut adopter la méthode basée sur l'approximation des caractéristiques non linéaires à l'aide des segments de droites [6].

Description :

Il est possible de réaliser n'importe quelle caractéristique de transfert non linéaire monotone en introduisant, dans un convertisseur courant-tension ,un courant qui varie en fonction de la valeur du potentiel d'entrée . ce courant peut être obtenu par une somme de courants qui sont nuls jusqu'à des valeurs déterminées du potentiel d'entrée et qui croissent ensuite linéairement avec ce dernier . Les schémas de la figure 2.4 sont des dispositifs permettant d'atteindre cet objectif [6].



a) b)

Fig.2.4 : Cellules de base pour réaliser l'approximation de fonctions non linéaires.

• **Cellule (a) :** (Figure 2.4.a)

Le courant est nul jusqu'à que le potentiel Vk atteigne la valeur Uj de conduction de la diode . Cette condition est réalisée lorsque le potentiel Vs atteint le potentiel de conduction Vsk :

$$Vsk = Uj\left(\frac{Rek + Rk}{Rk}\right) - \frac{Rek}{Rk}V \qquad (1)$$

En effet de la Figure 2.4.a on peut tirer facilement :

$$I_k = \frac{Vs}{Rek} - \left[Vk\left(\frac{1}{Rk} + \frac{1}{Rek}\right) - \frac{Rek}{Rk}V\right] \qquad (2)$$

En mettant l'expression (2) égale à zéro quand Vk = Uj , nous obtenons l'équation (1).

Pour $Vs \geq Vsk$, Le courant I_k a pour expression :

$$I_k = \frac{Vs - Vsk}{Rek} \qquad (3)$$

A la Figure 2.5 , on a représenté le courant I_k en fonction du potentiel d'entrée Vs.

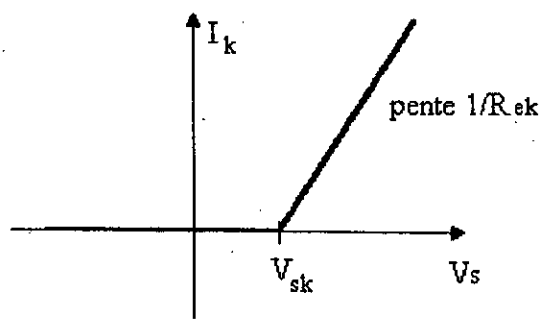


Fig.2.5 : Caractéristique $Vs = f(I_k)$ de la cellule a.

• **Cellule (b) :** (Figure 2.4.b)

En suivant la même démarche que pour la cellule (a), on démontre que :

$$Vsk = -\left(\frac{Rek}{Rk}V + \frac{Rek + Rk}{Rk}Uj\right) \qquad (4) \quad \text{à} \quad Vk = -Uj$$

et que pour $Vs \leq Vsk$ le courant $I_k = \frac{Vsk - Vs}{Rek}$ (5).

La caractéristique $Vs = f(I_k)$ de cette cellule est donnée par la figure suivante :

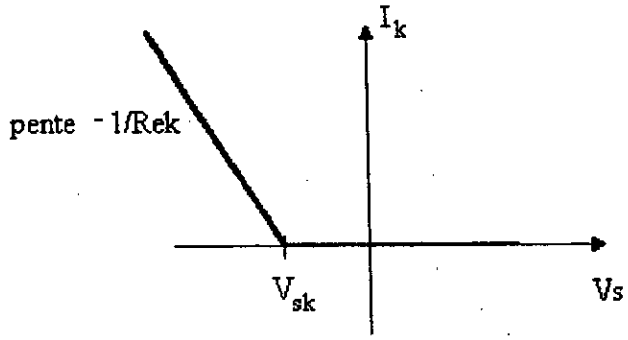


Fig.2.6 : Caractéristique Vs = f (I_k) de la cellule (b).

Après avoir réalisé un réseau constitué de cellules similaires à celles de la figures 2.4 , et choisi le potentiel de conduction Vsk et la résistance Rek de chaque cellule , on somme les courants I_k à l'entrée d'un convertisseur courant-tension (Figure 2.7).

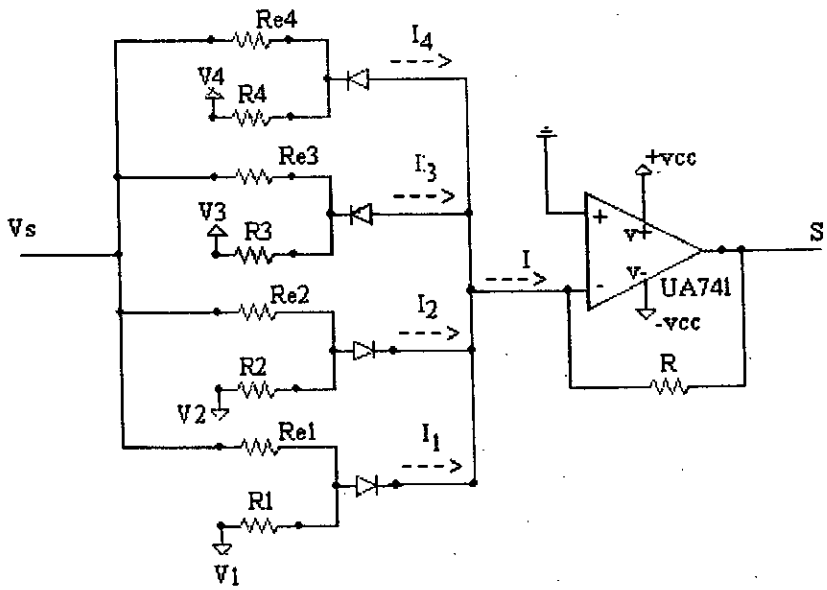


Fig.2.7 : Schéma-bloc d'un système pour approximer une fonction non linéaire.

La caractéristique de transfert de ce dispositif (Figure 2.8) est une fonction non linéaire constituée de segments de droites. Le potentiel de sortie est donnée par l'expression : $S = - R \sum_{k=1}^{n} I_k$, n étant le nombre de cellules.

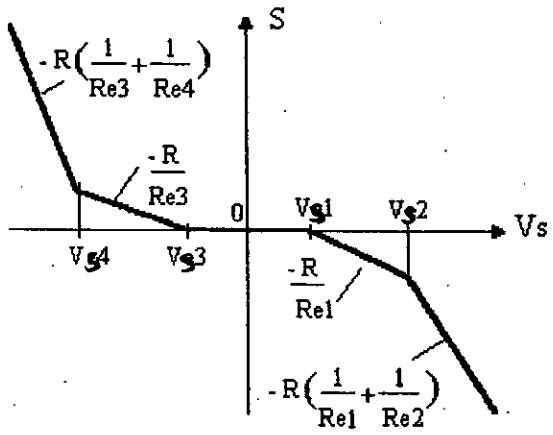


Fig.2.8 : Caractéristique de transfert du dispositif d'approximation.

En se basant sur ce modèle de neurone artificiel, on peut donc implémenter n'importe quel réseau de neurone. Pour une application donnée, on choisi la structure du réseau et on réalise la phase d'apprentissage par simulation numérique. La phase de réalisation matérielle consistera à choisir les valeurs des résistances correspondant aux poids synaptiques déterminés par l'algorithme d'apprentissage, et aussi à déterminer les paramètres du deuxième bloc pour que sa caractéristique entrée-sortie s'approche le plus de la fonction d'activation choisie.

Mais ces réalisations se heurtent à de nombreux problèmes. En effet, les poids synaptiques reliant les neurones doivent être en principe précis, modifiables et stables dans le temps. Or ces conditions sont difficiles à réaliser à cause de :

- la sensibilité aux bruits des circuits électroniques.
- la sensibilité des composants à la température.
- la dérive des caractéristiques avec le temps (vieillissement des composants).
- l'architecture difficilement évolutive.

Ces inconvénients limitent la précision des calculs que peuvent atteindre les réalisations analogiques.

Ce type d'implémentation peut être utilisé lorsque l'application envisagée requiert de grandes vitesses de traitement, que seules les techniques analogiques peuvent atteindre.

2.2 IMPLEMENTATIONS DIGITALES OU NUMERIQUES

On a vu que les principaux inconvénients des implémentations analogiques des réseaux de neurones sont :

- Difficultés d'évolutions du réseau.
- Manque de précision des calculs.

Pour pallier à ces inconvénients il est nécessaire d'avoir recours à une implémentation digitale (numérique), cela permet d'inclure plusieurs fonctionnalités au réseau :

- Possibilité de définir la précision désirée sur les calculs par le choix de la largeur du mot des données et du type d'arithmétique utilisé (calcul sur des entiers, sur des nombres à virgules fixes ou à virgules flottantes)
- Apprentissage interne.
- Programmabilité de l'architecture du réseau : choix du nombre de neurones, du nombre de couches, des fonctions d'activations et des paramètres de l'apprentissage.
- Possibilités de relier les circuits selon des topologies spatiales complexes. On peut réaliser ainsi des réseaux de grandes dimensions (en mettant en cascade plusieurs circuits de base, ou en les assemblant selon une grille à deux dimensions..)

Il existe plusieurs possibilités pour une implémentation numérique des réseaux de neurones .

2.2.1 IMPLEMENTATIONS PAR LOGIQUE CABLEE

Un neurone ne réalise que des opérations élémentaires. On peut donc le réaliser à partir de registres à décalage, d'additionneurs et de multiplieurs. Un exemple d'une telle réalisation est donné, pour un neurone ayant N entrées, par la Figure 2.9 [7].

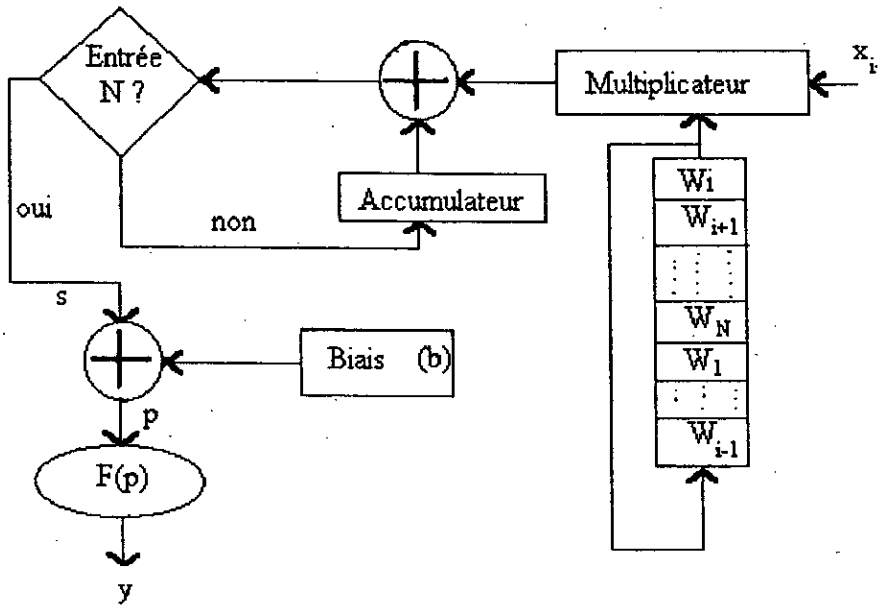


Fig.2.9 : Implémentation d'un neurone par logique câblée.

Ce modèle est constitué :

- d'une entrée qui reçoit l'entrée i du neurone (x_i). i = 1, 2, ..., N.
- d'un registre à décalage contenant les poids synaptiques du neurone.
- d'un multiplieur qui calcule le produit $x_i . W_i$.
- d'un additionneur faisant la somme entre la sortie du multiplieur et le contenu de l'accumulateur.

Lorsque le multiplicateur aura reçu sa $N^{ième}$ entrée, la sortie de l'additionneur contiendra la somme $S = \sum_{i=1}^{N} x_i . W_i$.

Un deuxième additionneur qui calcule l'entrée totale du neurone $p = S + b = \sum_{i=1}^{N} W_i . x_i + b$, où b est le biais du neurone. Il est mémorisé dans un registre.

La sortie du neurone y est calculée par : $y = F(p)$ où F est la fonction d'activation.

La difficulté dans ce modèle réside dans la réalisation de la fonction d'activation F avec la précision requise. Elle peut être générée grâce à l'implémentation matérielle d'une approximation, ou bien, elle peut être tabulée [8].

On peut, à partir de ce modèle, réaliser un réseau de neurones quelconque en connectant plusieurs neurones à l'aide d'un réseau d'interconnexions programmable via un système hôte (un P.C. par exemple) , Fig.2.10 .

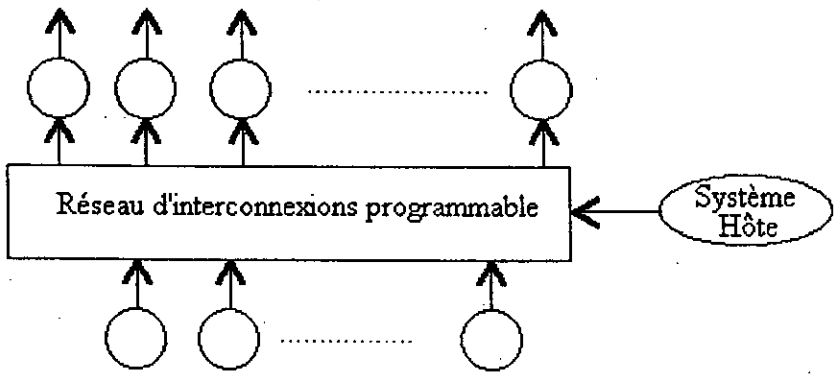


Fig.2.10 : Modèle d'implémentation digitales d'un réseau de neurones.

Les réseaux de neurones réalisés par cette méthode font partie de la classe des processeurs cellulaires [9]. Un processeur cellulaire est constitué d'un assemblage de cellules simples et identiques (dans ce cas, ce sont les neurones) réalisant des opérations élémentaires. La communication avec l'extérieur ne peut se produire qu'à partir des cellules spécialisées.

Dans de tels réseaux, il y a exécution en parallèle d'opérations élémentaires sur les entrées des neurones : c'est le parallélisme à " grain fin " [9]. La granulité est évaluée par la capacité de traitement de chaque noeud du réseau et de sa mémoire associée (pour un processeur cellulaire 1 noeud représente 1 neurone).

2.2.2 IMPLEMENTATION PAR LOGIQUE MICROPROGRAMMEE

Une autre possibilité pour la réalisation numérique des réseaux de neurones est l'utilisation de microprocesseurs. Dans ce cas chaque noeud du réseau est un microprocesseur classique (telles les séries 68000 de MOTOROLA, les processeurs 80X86 d'INTEL, ou les

transputers d'INMOS). On affecte généralement plus d'un neurone par processeur. Chaque processeur possède sa propre mémoire locale, et est relié aux autres processeurs par un réseau d'interconnexions. Une unité de contrôle ,un système hôte, synchronise toutes les opérations parallèles (Fig.2.11).

Tous les processeurs effectuent la même tâche opérant chacune sur des données différentes (comme le calcul de la sortie de tous les neurones du réseau). Ce type d'architecture est appelé SIMD (Single Instruction Multiple Data Stream). Une telle structure est appelée parallélisme à " grain moyen " [9].

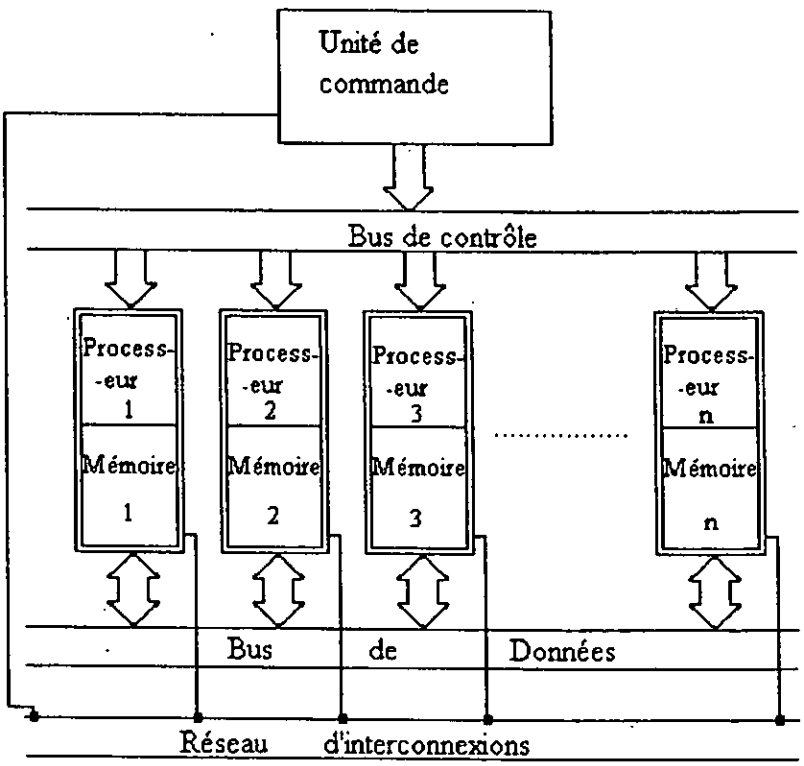


Fig.2.11 : Architecture SIMD.

La topologie du réseau est déterminée par le réseau d'interconnexions qui relie les processeurs.

Une topologie avec interconnexion totale est la plus souhaitable, mais la complexité croît très vite avec le nombre de processeurs. Il existe diverses topologies possibles pour relier les processeurs.

a) TOPOLOGIE EN BUS SIMPLE [9]

C'est une topologie simple à réaliser (Fig.2.12). Le bus est un chemin de communication reliant tous les processeurs et l'unité de commande. Elle est très répandue dans les réseaux locaux (réseaux de type ETHERNET). Son principal inconvénient est le problème des conflits d'accès au bus unique (si on dispose d'un grand nombre de processeurs). Ce qui réduit inévitablement la vitesse de communication.

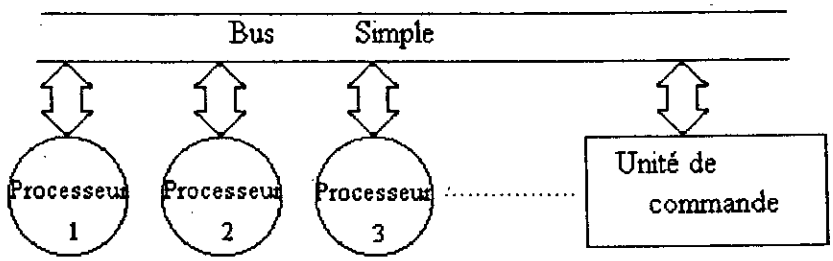


Fig.2.12 : Schéma de principe d'un réseau en bus.

b) TOPOLOGIES A CONNEXIONS DIRECTES

Réseaux totalement connectés

Tout processeur est relié aux autres processeurs par une connexion. On ne peut mettre en pratique cette solution que pour 1 petit nombre n de processeurs. Le nombre d'interconnexions est égal à $n * \frac{(n-1)}{2}$ (Fig.2.13). [9]

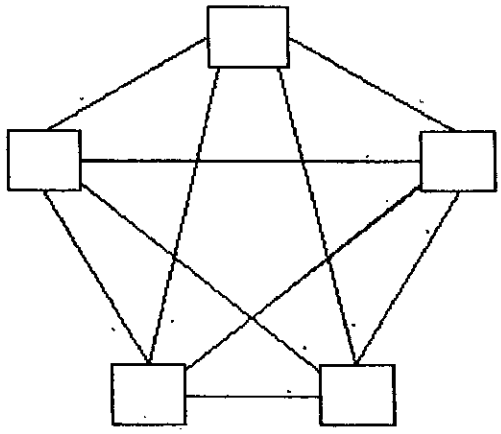


Fig 2.13 : Réseau totalement connecté.

Réseau linéaire ou PIPELINE [9]

C'est une structure très simple (Fig.2.14) qui peut être utilisée lorsque les données subissent une série de transformation successives. Ce qui est le cas des réseaux de neurones à couches où les sorties de chaque couche sont transmises à la couche suivante. On peut donc affecter à chaque processeur une couche du réseau de neurones.

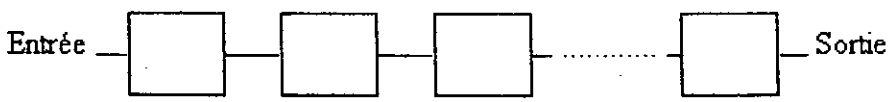


Fig.2.14 : Topologie en pipeline.

Le résultat final du traitement des entrées est récupéré sur le processeur de l'étage final.

L'avantage de cette structure est qu'après un temps d'initialisation (qui dépend du nombre d'étages du pipeline) , le pipeline produit un résultat à sa sortie pour chaque cycle de traitement.

Plusieurs implémentations des réseaux de neurones ont été réalisées avec des microprocesseurs. On cite [2] :

- SIGMA 1 (SAIC)

C'est 1 extension pour IBM-PC, basée sur un processeur spécialisé pour exécuter des multiplications et des additions en virgules flottantes, avec 12 Mega-octets de mémoire

additionnelle. La plupart des modèles des réseaux de neurones sont implémentés. Cette carte est fournie avec un langage de spécification de réseaux .

• MARK III et IV (TEW)

C'est 1 carte additionnelle pour bus VME (VAX) comportant 15 processeurs M68020 et M68881, avec laquelle est fourni un environnement graphique de spécification de réseaux.

2.3 REALISATION LOGICIELLE [3]

Il s'agit de langages permettant de choisir plusieurs types d'architectures et d'algorithmes d'apprentissage des réseaux de neurones.

Ces logiciels sont dotés de bibliothèques de programmes spécialisés et peuvent être associés à des interfaces graphiques interactives pour faciliter le travail de l'utilisateur. Ils permettent de réaliser toutes les étapes nécessaires à la simulation d'un réseau de neurones pour une application donnée:

- choix d'une structure pour le réseau.
- choix des paramètres d'apprentissage.
- réaliser l'apprentissage du réseau, et le tester en phase d'utilisation pour déterminer sa capacité de généralisation.

On peut grâce à ces logiciels concevoir et tester de nouveaux algorithmes d'apprentissage et évaluer leurs performances.

CONCLUSION

Des trois méthodes d'implémentations exposées, c'est l'implémentation digitale par des microprocesseurs qui offre le plus de souplesse à l'utilisateur pour la programmation du réseau, la réalisation de l'apprentissage, ainsi que pour faire évoluer la structure du réseau. C'est aussi l'implémentation qui offre la plus grande capacité de traitement, ce qui est très important pour pouvoir implémenter des applications nécessitant un traitement d'un grand nombre de données (tel que le traitement d'image).

Notre choix s'est porté sur une implémentation logicielle puisant son parallélisme de l'architecture du processeur utilisé à savoir : le parallélisme de flux et le parallélisme de contrôle [10] :

- le parallélisme de contrôle intervient dans notre application informatique car elle est composée d'actions que l'on peut "faire en même temps". En effet tous les neurones d'une couche déterminée travaillent en même temps pour calculer la sortie de cette couche.
- Le parallélisme de flux ou le mode de fonctionnement en Pipeline intervient dans le mode du travail à la chaîne. On dispose d'un flux de données (les entrées du réseau de neurone), sur lesquelles on effectue une suite d'opérations en cascades. Chaque couche du réseau reçoit ses entrées de la couche précédente , calcule sa sortie et la transmet à la couche suivante jusqu'à obtenir la sortie finale du réseau.

CHAPITRE 3

IMPLEMENTATION LOGICIELLE DES RESEAUX DE NEURONES A L'AIDE DES "DDE"

1. INTRODUCTION

On propose dans ce chapitre une méthode d'implémentation des réseaux de neurones qui reflète leur fonctionnement réel ; c'est à dire un ensemble d'unités élémentaires aux fonctions simples, travaillant en parallèle et pouvant communiquer entre eux.

L'idée de départ de cette implémentation est de réaliser un programme modélisant un seul neurone et ayant les caractéristiques suivantes : [2]

- •Autonomie : il faut que le neurone intègre les fonctions nécessaires à son fonctionnement autonome : commande, communication, mémoire locale, calcul, extinction et naissance.
- •Programmabilité : on doit pouvoir programmer la fonction d'activation pour être compatible avec un large gamme de modèles, et le nombre d'entrées du neurone.
- •Simplicité : le neurone doit être aussi simple que possible.

On pourra par la suite construire un réseau de neurones en créant autant de tâches élémentaires (chaque tâche modélise un neurone) dont on aura besoin.

Un programme hôte se chargera de toutes les opérations nécessaires au bon fonctionnement du réseau (Figure 3.1) :

- •Production des neurones selon la configuration choisie pour le réseau.
- •Paramétrage des variables du neurone : poids synaptiques, fonction d'activation, biais, nombre d'entrées du neurone....
- •Supervision de la transmission et de la réception des données entre les neurones d'une part, et entre les neurones et le programme hôte d'autre part.
- •Synchronisation du fonctionnement du réseau.

Mais avant de présenter cette méthode d'implémentation, on présentera un programme que nous avons développé (une interface utilisateur) qui servira de "processus hôte ". Son rôle est double :

- Simuler, en séquentiel, un réseau de neurones y inclue les phases d'apprentissage et de test.
- Réaliser l'interface avec le réseau implémenté en vue d'assurer la configuration, la transmission et la réception de données ainsi que leur mise en forme finale.

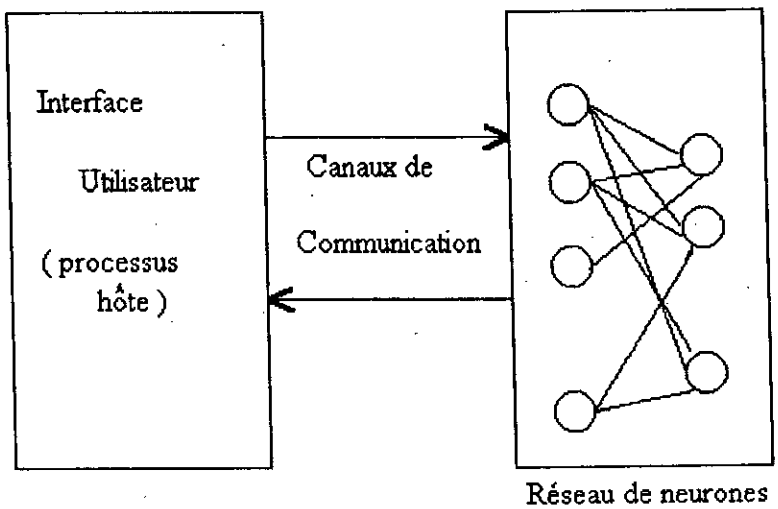


Fig.3.1 : Modèle d'implémentation logicielle.

2. INTERFACE UTILISATEUR

Ce programme a été développé à l'aide de Matlab sous Windows à cause des nombreux avantages qu'il offre : [11]

- •Un environnement intégré composé d'un éditeur de texte et d'un interpréteur de commande.
- •Une grande facilité pour les manipulations et les calculs portant sur les matrices et les tableaux de grandes dimensions, ce qui procure de grandes vitesses de calcul.
- •De puissantes fonctions graphiques faciles d'utilisation.
- •Il est extensible par l'ajout de bibliothèques spécialisées ou *Toolboxes* . Ces bibliothèques sont des ensembles de fonctions qui permettent d'étendre les possibilités de Matlab dans un domaine déterminé.

Le programme réalisé se présente comme une interface graphique constituée de menus faciles à manipuler disposants de boites de dialogues, de boites de saisies et de boutons actions. De ce fait, il est beaucoup plus facile d'utilisation que le *Neural Network toolbox* de Matlab car contrairement à ce dernier, il ne nécessite de la part de l'utilisateur aucune connaissance des fonctions qui servent à simuler le fonctionnement d'un réseau de neurones.

2.1 EXPLOITATION

Pour lancer l'exécution du logiciel, on exécute la commande " modelise " sur la ligne de commande de Matlab. On a alors le choix entre les trois modèles de réseaux présentés dans le chapitre 1 (Figure 3.2) :

- Réseau Multicouches.
- Réseau Auto-organisateur.
- Réseau de Hopfield.

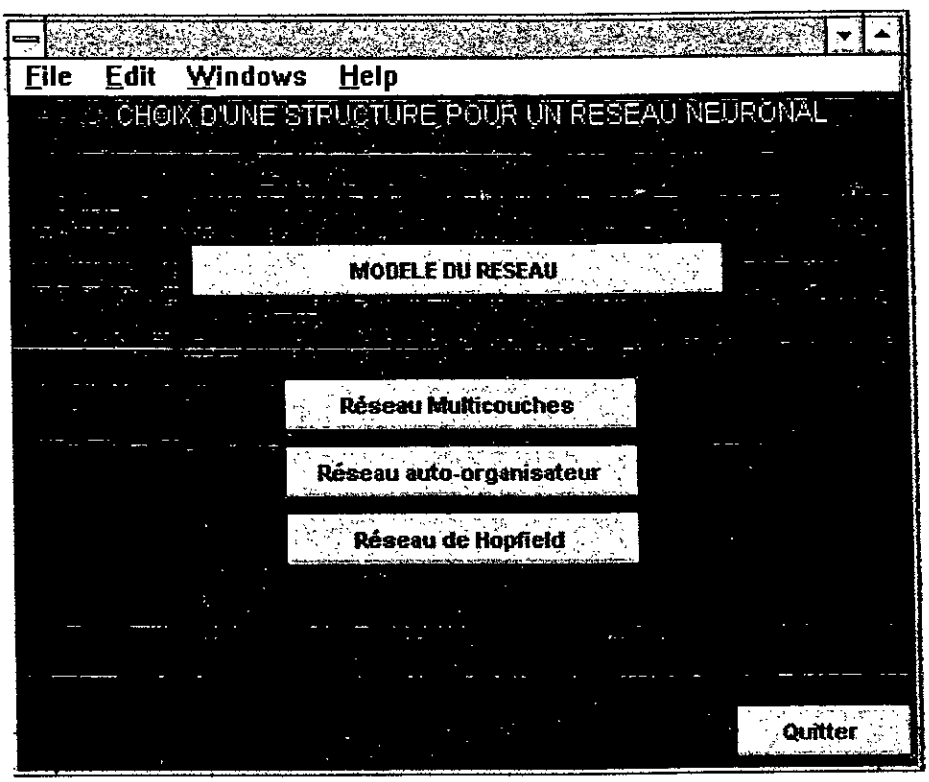


Fig.3.2 : Menu Principal.

On n'a alors qu'à choisir à l'aide de la souris le modèle désiré en activant l'elvis[1] correspondant.

[1] elvis : Elément Visuel.

2.1.1 RESEAU MULTICOUCHES

Le choix de ce modèle fait apparaître le menu de la Figure 3.3 : on peut choisir le nombre de couches du réseau. Le nombre maximum de couches est fixé à 6.

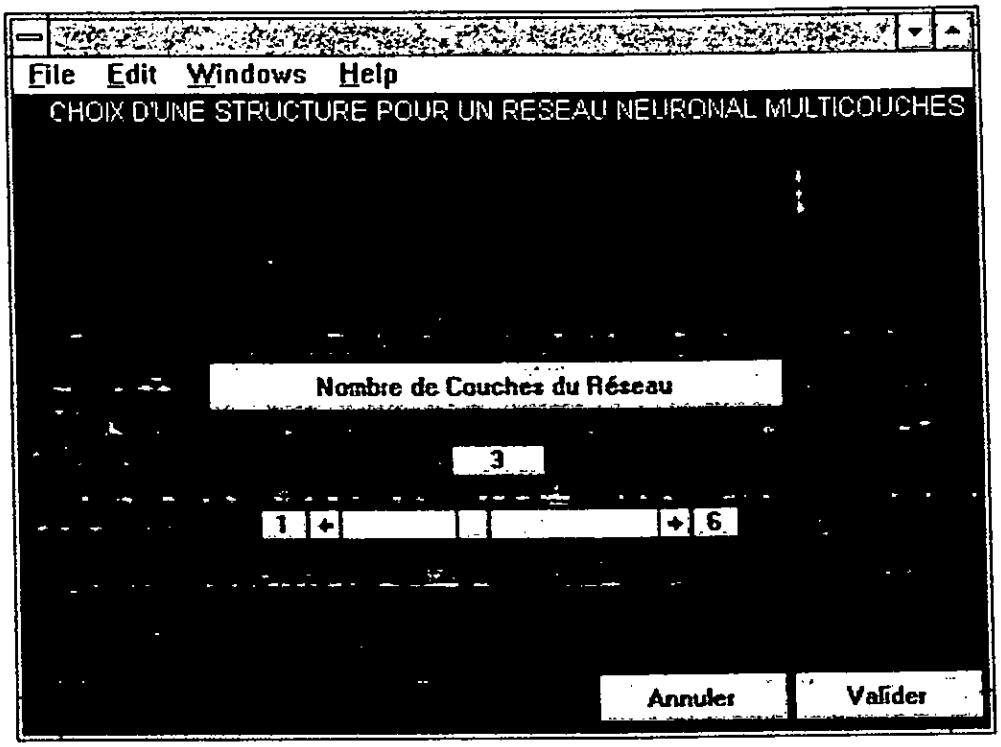


Fig.3.3 : Choix du nombre de couches.

La validation d'un choix par l'elvis " Valider", fait apparaître le menu de la Figure 3.4 destiné à la sélection des paramètres d'apprentissage du réseau.

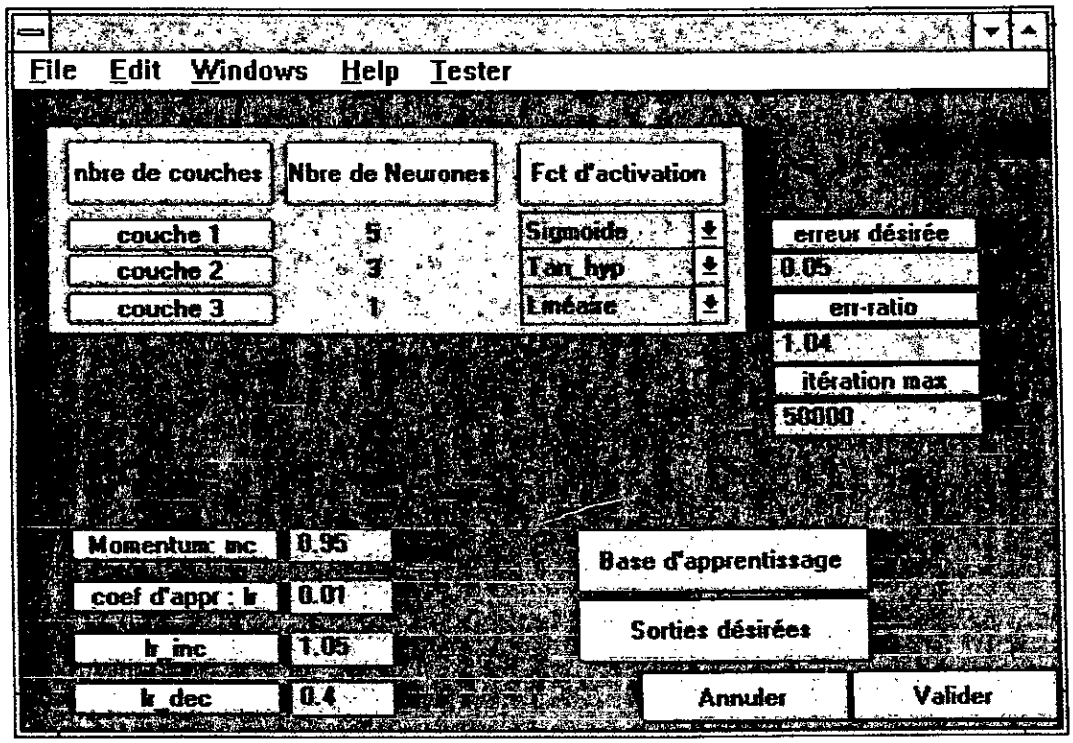
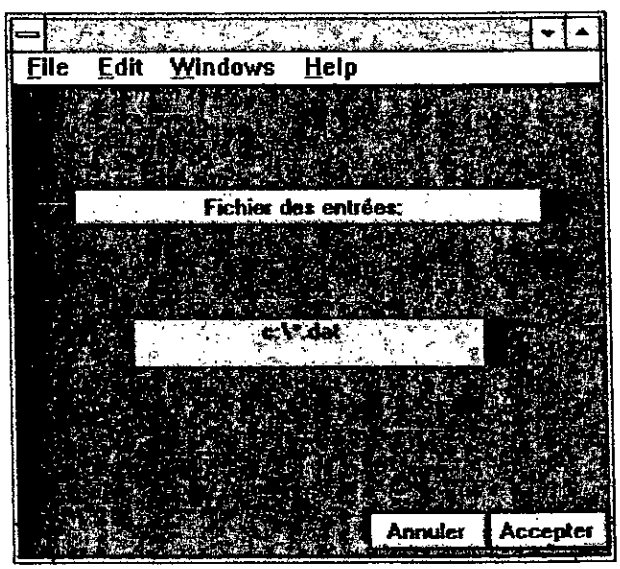


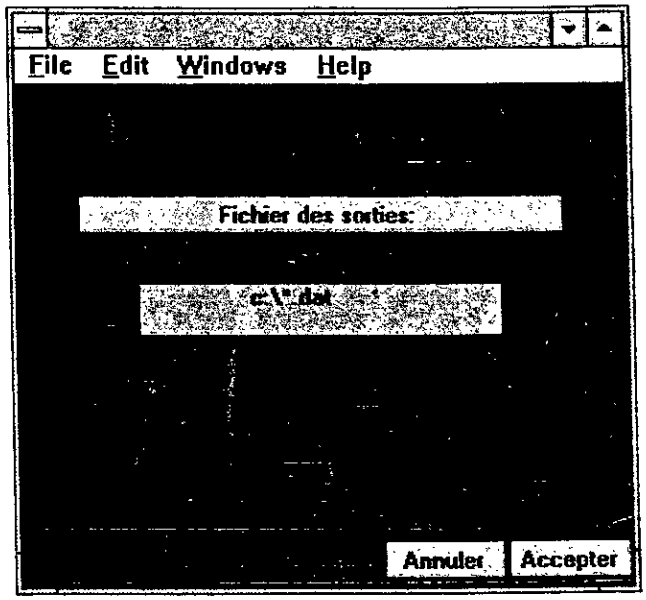
Fig.3.4: Choix des paramètres du réseau.

On fixe :

- le nombre de neurones par couche.
- la fonction d'activation des neurones de chaque couche.
- les entrées (la base d'apprentissage) et les sorties désirées du réseau, qu'on pourra lire à partir d'un fichiers (Fig.3.5 a) et b)).



a)



b)

Fig.3.5 : Lecture des entrées (a) et des sorties désirées (b) du réseau.

- le coefficient d'apprentissage (lr).
- l'erreur finale désirée.
- le nombre maximum d'itérations .
- le rapport d'erreur (err_ratio), utilisé pour avoir un coefficient d'apprentissage adaptatif et pour utiliser un momentum.
  - Si err_ratio = 1 : on utilise l'algorithme de rétropropagation simple.
  - Si err_ratio >1 : on utilise l'algorithme de rétropropagation avec un coefficient d'apprentissage adaptatif et momentum.

Dans ce cas on choisi :

La constante d'incrémentation (lr_inc) du coefficient d'apprentissage.

La constante de décrémentation (lr_dec) du coefficient d'apprentissage.

Chaque fenêtre possède un elvis " Annuler " qui permet de revenir à la fenêtre précédente, et d'un bouton " Valider " qui valide toutes les informations entrées dans la fenêtre.

Après validation des paramètres du réseau, la phase d'apprentissage commence. On obtient alors la Figure 3.6 . Le réseau est représenté sous forme graphique et on peut suivre l'évolution de l'erreur quadratique totale au cours de l'apprentissage, ce qui permet d'apprécier la convergence du réseau.

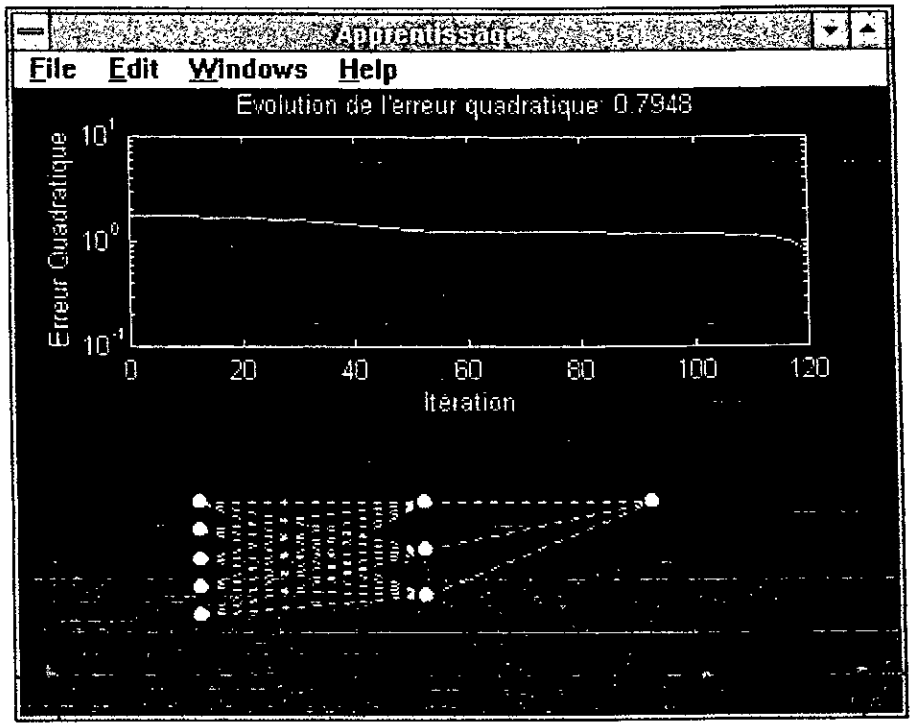


Fig.3.6 : Phase d'apprentissage.

L'apprentissage prend fin si on atteint la valeur désirée de l'erreur, ou après avoir atteint le nombre maximum d'itération. On obtient à la fin de l'apprentissage les matrices des poids et des biais du réseau.

On peut tester la réponse du réseau à d'autres entrées différentes de la base d'apprentissage et cela dans le but d'évaluer sa capacité de généralisation sur des exemples qu'il n'a pas appris. Pour cela on utilise le menu " Tester " de la bare des menus (Figure 3.4) d'ou on pourra choisir le fichier contenant notre base de test.

2.1.2 RESEAU AUTO-ORGANISATEUR

Le choix de modèle fait apparaître le menu présenté dans la Figure 3.8.

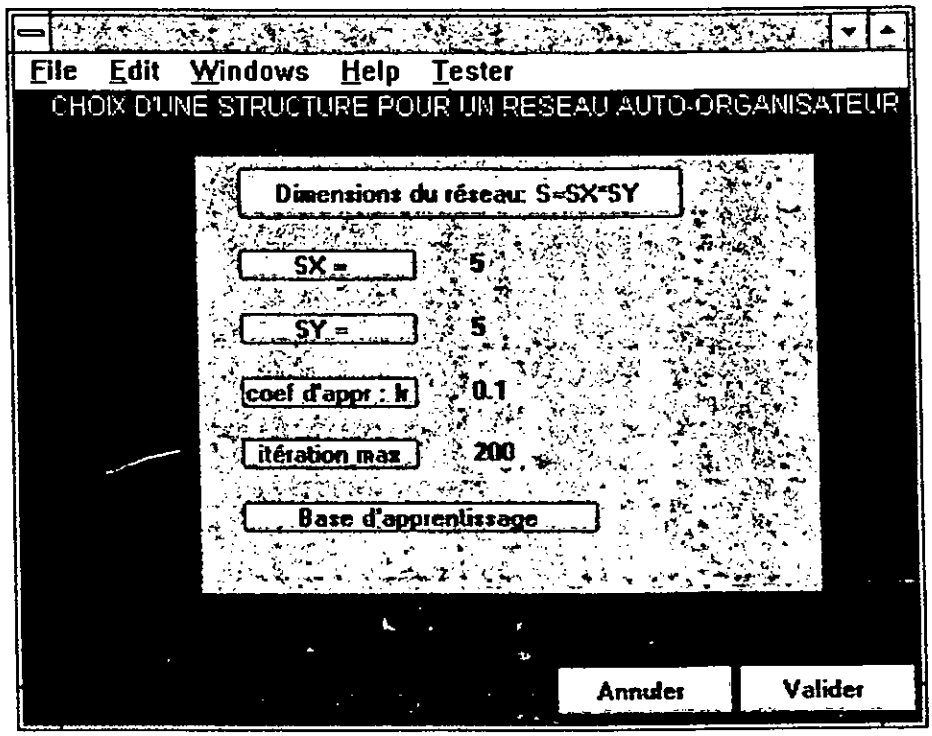


Fig.3.8 : Choix des paramètres d'un réseau auto-organisateur.

La configuration des paramètres du réseau se fait par le choix de :

- La dimension du réseau S, c'est à dire le nombre de neurones.
  $S = Sx * Sy$
  Le réseau peut avoir une topologie à une dimension (Sy ou Sx = 1) ou à deux dimensions (Sx et Sy ≠ 1).
- Le coefficient d'apprentissage initial (lr).
- Le nombre maximum d'itérations : fixe la durée de l'apprentissage.
- Les entrées du réseau (c'est la base d'apprentissage) qu'on lit à partir d'un fichier.

Après avoir choisi les paramètres du réseau ainsi que ses entrées, on active le bouton de validation, ce qui lance l'exécution de l'algorithme d'apprentissage. A la fin de l'apprentissage, on aura comme résultat les poids synaptiques et les biais de tous les neurones du réseau.

On peut par la suite tester le réseau en lui présentant des entrées différentes de la base d'apprentissage pour observer sa réponse. Cela peut être réalisé grâce au menu " Tester" de la bare des menus.

2.1.3 RESEAU DE HOPFIELD

Le menu qui correspond à ce modèle est présenté dans la Figure 3.9.

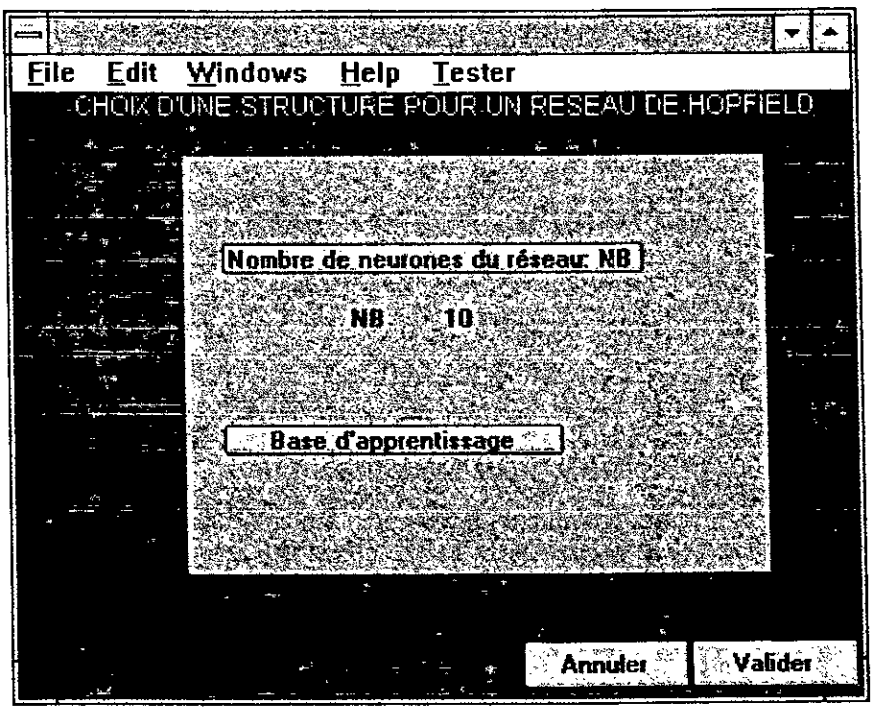


Fig.3.9 : Choix des paramètres d'un réseau de HOPFIELD.

On peut choisir :

- Le nombre de neurones du réseau.
- Ainsi que les entrées du réseau (base d'apprentissage), qu'on lit toujours à partir d'un fichier.

Le calcul des poids et des biais des neurones se fera après validation du choix précédent par le bouton " Valider ".

Après l'apprentissage, on peut tester le réseau grâce au menu " Tester " de la base des menus.

3. IMPLEMENTATION LOGICIELLE A L'AIDE DES DDE

Pour réaliser une implémentation des réseaux de neurones telle qu'elle est décrite par la Figure 3.1 , on a vu que les neurones doivent répondre à certaines caractéristiques :

- Autonomie.
- Programmabilité.
- Simplicité.

Chaque neurone étant modélisé par un programme.

La construction d'un réseau de neurones particulier se fait en lançant l'exécution d'autant de programmes qu'il y a de neurones dans ce réseau.

Ces conditions impliquent :

- L'utilisation d'un système d'exploitation multitâche capable de supporter l'exécution de plusieurs programmes simultanément.

- De définir un protocole de communication très précis pour l'échange des données entre les différents neurones, ainsi qu'une gestion des interconnexions entre les différents neurones.

Ces deux problèmes sont résolus grâce à l'utilisation du système d'exploitation Microsoft Windows. En effet Windows 3.1 permet l'exécution de plusieurs programmes à la fois en exploitant le mode protégé des processeurs 80X86 (80286, 80386 et 80486). Ce mode a été introduit pour permettre l'exécution simultanée de plusieurs programmes (appelés tâches ou " Tasks ") indépendants. Cette exécution en parallèle implique une protection des ressources spécifiques à des tâches contre l'écriture dans des zones de mémoire étrangère, c'est pour cela qu'on qualifie le mode de " protégé".

De plus Windows possède un protocole de communication que peuvent utiliser les applications Windows pour l'échange de données : c'est les DDE (Dynamic Data Exchange) échange dynamique de données.

On profite donc de cette opportunité pour développer nos programmes modélisant les neurones comme des Applications Windows. Comme Matlab ne peut pas produire des programmes exécutables indépendants, on utilisera pour cela le langage Borland Pascal 7.0 sous Windows car il intègre entre autres des bibliothèques permettant la programmation d'applications Windows.

3.1 ETUDE DES DDE

L'échange dynamique des données (DDE) [12] est une forme de communication interprocessus qui permet l'échange de données entre applications Windows. On utilise pour cela les fonctions que fournit la bibliothèque DDEML (Dynamic Data Exchange Management Library). C'est une bibliothèque de liaison dynamique (Dynamic Link Library : DLL) qui permet l'ajout de possibilités DDE à une application Windows.

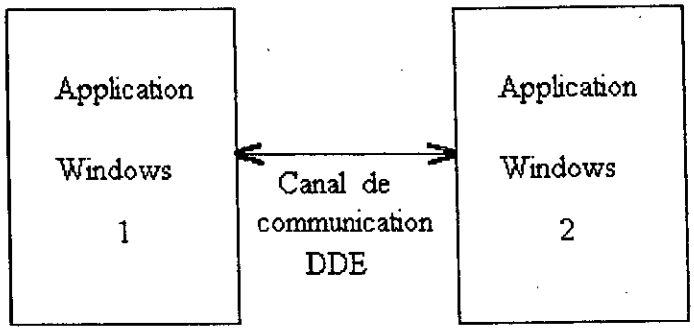


Fig.3.10 : Principe de l'échange dynamique de données (DDE).

L'échange dynamique de données prend toujours place entre une application cliente et une application serveur. Le client procède à l'échange de données en établissant une conversation avec le serveur, à travers laquelle il peut envoyer une transaction au serveur (une transaction est une demande de données ou de service). Un serveur peut avoir plusieurs clients au même moment, et un client peut demander des données à plusieurs serveurs.

Le client termine la conversation lorsqu'il n'a plus besoin de données ou de services.

3.1.1 GESTION DE LA CONVERSATION

Une application DDE utilise 3 niveaux de hiérarchies pour identifier la donnée sur laquelle portera l'échange durant la conversation.

- Nom de l'application (ServiceName) : c'est une chaîne de caractères contenant le nom du serveur DDE vers lequel on ouvre un canal de communication.

- Non de rubrique (TopicName) : c'est une chaîne de caractères qui décrit un objet dans l'application serveur vers laquelle on ouvre un canal. Par exemple si on établit une conversation avec un tableur, le nom de rubrique sera un nom de fichier.
- Le nom de l'élément (ItemName) : une chaîne de caractères qui indique la donnée sur laquelle porte la transaction.

Ces noms représentent une sorte "d'adresse" de l'application serveur. Chaque application cliente qui voudra établir une conversation DDE avec une application serveur devra spécifier le Nom de l'application serveur, son Nom de rubrique ainsi que le Nom de l'élément (la donnée) sur lequel portera l'échange.

Toute application DDE possède une fonction appelée " Fonction callback " qui gère les transactions DDE associées à cette application.

a) INITIALISATION

Une application doit appeler la fonction Dde_Initialize avant de faire appel à toute autre fonction DDEML. Cette fonction obtient un identificateur d'instance de l'application, et enregistre la fonction callback avec la bibliothèque DDEML.

Syntaxe : DdeInitialize (Inst, Callback, 0 , 0)

paramètres :

inst : l'identificateur d'instance de l'application. Windows repère chacune des applications actives (encours d'exécution) par un numéro (appelé en anglais "handle").

Callback : pointe vers la fonction callback de l'application.

b) ETABLISSEMENT DE LA CONVERSATION

Une application cliente établie une conversation avec une autre application serveur en appelant la fonction Dde_Connect et en spécifiant le nom de l'application serveur ainsi que son nom de rubrique.

DDEML répond en envoyant la transaction XTYP_CONNECT à la fonction callback du serveur, ainsi que le nom de l'application et le nom de rubrique spécifiés par Dde_Connect.

Le serveur examine les noms envoyés et retourne la valeur VRAI s'ils correspondent bien à son nom d'application et à son nom de rubrique, et la valeur FAUX dans le cas contraire.

Si le serveur retourne la valeur FAUX, aucune conversation n'est établie.

Si le serveur retourne la valeur VRAI, la conversation est établie, et le client reçoit un Handle qui identifie la conversation. Ce Handle sera utilisé par le client pour tous les appels ultérieurs aux fonctions DDE pour l'échange de données. Le serveur reçoit la transaction XTYP_CONNECT_CONFIRM. Cette transaction envoie le Handle de conversation au serveur et confirme la connexion avec l'application cliente.

Syntaxe : DdeConnect (Inst, Service, Topic, point)
paramètres :
Inst : l'identificateur d'instance de l'application.
Service : handle qui spécifie le nom de l'application serveur avec laquelle on veut établir une conversation.
Topic : handle qui spécifie le nom de rubrique de l'application serveur.
point : c'est un pointeur. Il a la valeur NIL.

Ces handles seront crées au préalable avec la fonction DdeCreateStringHandle.

Syntaxe : handle = DdeCreateStringHandle (Inst, Chaîne de caractères, cp_WinAnsi)

paramètres:
Inst : l'identificateur d'instance de l'application.
Chaîne de caractères : peut être le Nom de l'application, son nom de rubrique ou le nom d'un élément de l'application. On doit créer un handle (un identificateur) pour chacun de ces noms.
cp_WinAnsi : spécifie le code page de la chaîne de caractère.

c) ECHANGE DE DONNEES

Les DDE utilisent la mémoire globale pour échanger des données entre applications.

Toutes les transactions qui procèdent à l'échange de données doivent appeler la fonction DdeCreateDataHandle. Cette fonction copie les données dans la mémoire globale, et retourne à l'application un Handle identifiant ces données.

Pour échanger des données, une application envoie ce Handle à la bibliothèque DDEML, qui l'envoie à son tour à la fonction Callback de l'application recevant les données.

Il existe plusieurs transactions pour l'échange de données :

- Envoie de données :

On utilise pour cela la transaction XTYP_POKE : DDEML envoie cette transaction au serveur en spécifiant le handle de la conversation, le handle de l'élément (ItemName) sur lequel portera la transaction, ainsi que le format des données transmises. Si le serveur accepte l'envoie de la donnée, il retourne la valeur DDE_FACK à DDEML. On ne peut envoyer qu'une seule donnée à la fois (par exemple on ne pas envoyer un vecteur vers une application en une seule transaction, mais on l'envoi élément par élément), et cette donnée devra être mise sous forme de chaîne de caractères avant d'être transmise.

- Demande de données :

Une application peut utiliser la transaction XTYP_REQUEST pour faire une demande de données à une application serveur.

DDEML envoie cette transaction au serveur en spécifiant le handle de la conversation, le handle de l'élément (ItemName) et le format des données. Le serveur renvoie un handle qui identifie la valeur de la donnée. DDEML passe ce handle à l'application cliente, il servira à repérer cette donnée en mémoire vive.

Syntaxe : DdeClientTransaction(Data, DataLen, ConvHdl, ItemHld, cf_Text, Transaction, time, point)

paramètres :

Data : un pointeur qui indique le l'emplacement en mémoire de la donnée sur laquelle porte la transaction.

DataLen : spécifie la longueur, en octets, de la donnée sur laquelle porte la transaction.

ConvHdl : le handle de la conversation DDE.

ItemHdl : le handle de l'élément sur lequel portera la transaction.

ct_text : c'est un format de donnée. Les données seront transmises sous forme de chaîne de caractères.

Transaction : pour un envoie de données : Transaction = XTYP_POKE.

pour une demande de données : Transaction = XTYP_REQUEST.

Time : spécifie le temps maximum, en millisecondes, que l'application cliente devra attendre pour avoir une réponse de l'application serveur

point : c'est un pointeur qui a la valeur NIL.

d) TERMINER LA CONVERSATION

Une application termine une conversation DDE en appelant la fonction DdeDisconnect. Cette fonction met fin à la conversation entre l'application cliente et l'application serveur.

Si une application n'a plus besoin d'établir de liaisons DDE, elle appelle la fonction DdeUninitialize qui libère les ressources DDEML que le système a alloué pour l'application.

Syntaxe : Ddedisconnect (ConvHdl)
DdeUninitialize (Inst)

paramètres :
ConvHdl : le handle de la conversation DDE qu'on veut terminer.
Inst : l'identificateur d'instance de l'application.

3.2 MODELE D'UN NEURONE UTILISANT LES DDE

Le modèle du neurone proposé utilisant l'échange dynamique de données est donné dans la figure suivante :

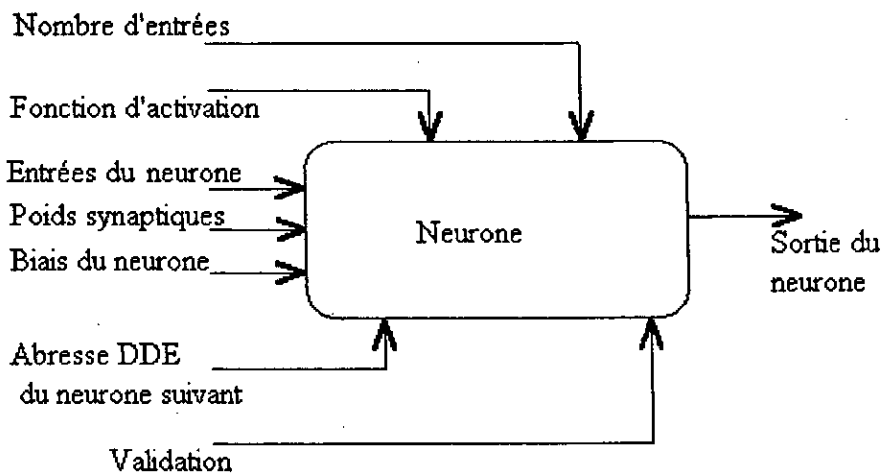


Fig.3.11 : Modèle du neurone.

Il possède :

- Des entrées de données : entrées du neurone, poids synaptiques, biais, nombre des entrées du neurone, la fonction d'activation ...
- Une sortie.
- Une entrée de commande (l'entrée validation).
- Une entrée d'identification ou adresse pour établir des conversations DDE avec d'autres neurones.

On accède à toutes ces données à travers un canal de communication DDE.

Les variables qui définissent les paramètres du neurones sont :

- Nombre d'entrées du neurone : (variable nb_e)

C'est une variable entière qui permet de spécifier le nombre d'entrées qu'aura le neurone, c'est à dire le nombre de connexions qu'il peut avoir ($1 \leq nb\_e \leq 40$).

- La fonction d'activation : (variable Fct_activ)

C'est un nombre entier qui permet de choisir la fonction d'activation du neurone.

Si Fct_activ = 1 $\rightarrow$ Fonction d'activation sigmoïde.
Si Fct_activ = 2 $\rightarrow$ Fonction d'activation Tangente hyperbolique.
Si Fct_activ = 3 $\rightarrow$ Fonction d'activation linéaire.

- Biais : (variable B)

C'est un nombre réel qui permet de fixer le biais du neurone.

- Poids synaptiques : (variable Poids_synap)

C'est un vecteur de réels. Il est constitué de nb_e éléments (un poids pour chaque entrée du neurone).

- Entrées du neurone : (variable TData)

C'est un vecteur de réels constitué de nb_e éléments.

- Sortie : (variable s)

C'est un nombre réel. Lorsque le neurone reçoit toutes ses entrées, il calcule sa sortie en utilisant la fonction d'activation précédemment choisie.

- Validation : (variable ok)

C'est une entrée de commande. Lorsque le neurone reçoit un "1" sur cette entrée, il envoi sa sortie vers le neurone avec lequel il est connecté.

- Adresse du neurone suivant : (variable N_S)

C'est par cette entrée qu'on informe le neurone avec quel neurone il est connecté. Cette adresse est le nom de l'application du neurone auquel on envoi la sortie.

Toutes ces variables sont définies dans l'unité ' NData.PAS '. Elle est donnée en Annexe, ainsi que le programme ' N1.PAS ' qui modélise le neurone .

Pour pouvoir accéder aux variables du neurone lors des appels aux fonctions DDE, on attribue au programme :

- Un nom d'application (ServiceName) : '1'.
- Un nom de rubrique (TopicName) : 'entree'.
- Et à chaque variable du programme un nom d'élément :

Entrées du neurone	→	'Data1'
Poids synaptiques	→	'Data2'
Fonction d'activation	→	'Data3'
Nombre d'entrées	→	'Data4'
Biais du neurone	→	'Data5'
Validation	→	'Data6'
Adresse DDE du neurone suivant	→	'Data7'
Sortie du neurone	→	'Data8'

Le programme qui a servi à modéliser ce neurone est une application Windows développée avec le langage BORLAND PASCAL 7.0. Il peut communiquer avec d'autres applications en utilisant les DDE, et peut s'exécuter sous Windows simultanément avec d'autres applications.

Le développement d'applications Windows nécessite l'utilisation de la programmation orienté objet (P.O.O). On a utilisé pour cela les bibliothèques de types objet intégrées dans Borland Pascal [13].

Le neurone ainsi développé est un objet descendant du type objet Twindow. Il peut être divisé en deux parties :

- Une partie communication : c'est l'ensemble des méthodes qui gèrent les communications DDE avec d'autres applications. C'est la plus grande partie du programme.
- Une partie qui calcule la sortie du neurone : elle n'est constituée que d'une seule fonction (la Fonction "Sortie").

On peut par la suite créer autant de processus modélisant des neurones qu'on le désire en les dérivants du processus "source" 'N1.EXE'. On ne changera pour chaque nouveau processus que son nom d'application pour que chaque neurone ait sa propre adresse DDE :

Neurone 1 (modélisé par N1.EXE) : son nom d'application est '1'.

Neurone 2 (modélisé par N2.EXE) : son nom d'application est '2'.

Neurone 3 (modélisé par N3.EXE) : son nom d'application est '3'.....

La figure suivante est un exemple de réseau constitué de 3 neurones. Elle détaille les liaisons qui existent entre les neurones et le processus de commande qui gère le fonctionnement du réseau.

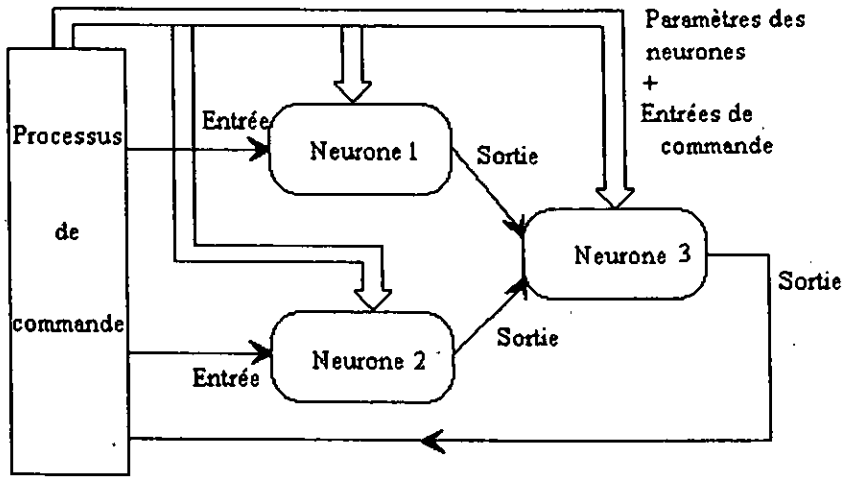


Fig.3.12 : Réseau constitué de 3 neurones.

Les paramètres du neurone regroupent : les poids synaptiques, le biais, la fonction d'activation, le nombre d'entrées des neurones ainsi que l'adresse des neurones suivants. L'entrée de commande est l'entrée validation.

3.3 IMPLEMENTATION D'UN RESEAU MULTICOUCHES A L'AIDE DES DDE

Dans cette partie on va appliquer le modèle d'implémentation des réseaux de neurones présenté dans la Figure.3.1 au modèle des réseaux multicouches car c'est le modèle le plus utilisé dans les réseaux de neurones.

Le processus hôte sera l'interface réalisé sous Matlab. (Fig.3.13).

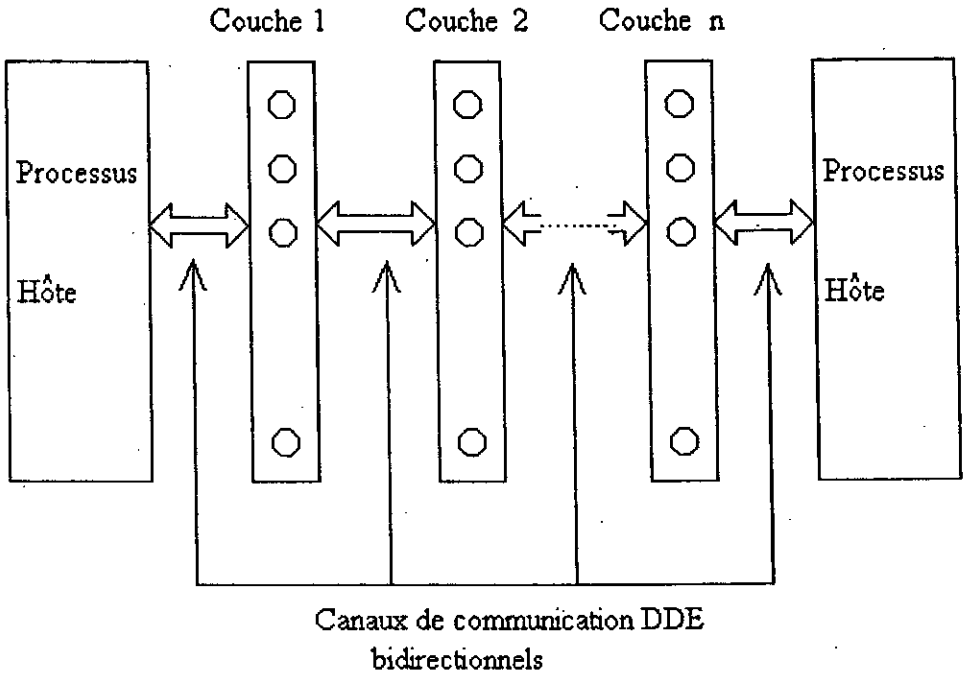


Fig. 3.13 : Implémentation d'un réseau Multicouches à l'aide des DDE.

La validation des choix des paramètres du réseau (nombre de couches, nombre de neurones par couche, fonctions d'activations ...), lance l'exécution d'autant de programmes qu'on aura choisi de neurones dans notre réseau : chaque neurone est ainsi représenté par une application totalement indépendante du processus hôte.

On peut, grâce aux liaisons DDE, fixer les paramètres de chaque neurone : nombre de ses entrées, sa fonction d'activation, les neurones auxquels il est connecté, et on attribue à chaque couche le nombre de neurones nécessaires.

Grâce au réseau ainsi construit, on peut réaliser les phases d'apprentissage du réseau pour une application précise.

L'algorithme d'apprentissage d'un réseau multicouches (chapitre 1) peut être divisé en 4 étapes :

1) Phase de présentation de la base d'apprentissage à la couche d'entrée du réseau.
2) Phase de rétropropagation de l'erreur de la couche de sortie vers la couche d'entrée.
3) Phase de mise à jour des poids synaptiques et des biais du réseau.
2) Phase de calcul de l'erreur quadratique totale sur la sortie du réseau.
Ces étapes sont répétées jusqu'à atteindre l'erreur désirée sur les sorties du réseau.

C'est lors de l'étape 1 que l'on utilise vraiment le réseau : on présente un exemple à la couche d'entrée et on transmet les sorties des neurones vers la couche suivante, et ainsi de suite jusqu'à obtenir les sorties de la dernière couche du réseau.

Le transfert des sorties d'une couche (i) vers les entrées de la couche suivante (i+1) se fait par le programme hôte en envoyant sur l'entrée "Validation" de tous les neurones de la couche (i) la valeur "1", ce qui provoque l'envoi par ces neurones de leurs sorties vers les neurones de la couche suivante auxquels ils sont connectés. Arrivé sur la dernière couche du réseau, le programme hôte récupère les sorties des neurones et pourra passer aux étapes suivantes de l'apprentissage (étapes 2 à 4).

Tous les transferts de données entre le programme hôte et les neurones, et entre les neurones se fait par l'intermédiaire de canaux de communication DDE.

Exemple de programmation d'un neurone à partir du processus hôte:

C'est le processus hôte qui fixe les paramètres de chaque neurone du réseau. On utilise pour cela les fonctions DDE intégrées à Matlab :

- C = DdeInit (nom de l'application, nom de la rubrique)

Initialise une conversation DDE avec l'application spécifiée. La variable C représente le numéro alloué au canal de cette conversation; il sera utilisé ultérieurement pour envoyer ou recevoir des données à travers ce canal.

- DdePoke (canal, nom de l'élément, donnée à envoyer) :

Envoi sur le canal de conversation spécifié (obtenu par DdeInit) une donnée vers un élément de l'application. La donnée à envoyer devra être donnée sous forme de chaîne de caractères, car c'est le seul format reconnu par Matlab.

- a=DdeReq (canal, nom de l'élément)

lit la valeur de la donnée représentée par le nom de l'élément spécifié et l'affecte a la variable "a".

exemple

- Initialisation d'une conversation DDE avec le neurone 1 :

le nom d'application de ce neurone est '1'

son nom de rubrique est 'entree'. L'initialisation du canal de conversation DDE se fait grâce à l'instruction suivante : C =DdeInit ('1', 'entree').

- Choix du nombre d'entrées de ce neurone à 20 :

Le nom de l'élément de la variable nb_e qui fixe le nombre d'entrées du neurone est 'Data4'. Pour donner à cette variable la valeur 20 on exécute l'instruction suivante : DdePoke (C, 'Data4', '20').

- Choix de la fonction d'activation du neurone :

pour une fonction sigmoïde on devra envoyer la valeur "1" vers la variable Fct_activ du neurone : DdePoke (C, ' Data3 ', '1').

- Lecture de la sortie du neurone :

La sortie du neurone est représenté par le nom de l'élément 'Data8' :

S = DdeReq (C, ' Data8 ') .

CONCLUSION

L'implémentation des réseaux multicouches réalisée à l'aide des DDE essaie de reproduire un fonctionnement naturel des neurones : chaque neurone étant indépendant des autres et pouvant communiquer avec ses voisins à travers des connexions préétablies.

La modélisation de chaque neurone comme un processus élémentaire paramétrable permet d'implémenter n'importe quel modèle de réseau de neurones.

Ce concept d'implémentation est un premier pas à notre connaissance vers une implémentation des réseaux de neurones sur un réseau de P.C. En effet, on simule le fonctionnement d'un tel réseau sur un seul P.C. : le processus hôte réalisé avec Matlab représente le serveur du réseau, et chaque application modélisant un neurone représente un processeur de ce réseau.

Mais l'inconvénient majeur de cette implémentation pour le moment est la lenteur de la phase de calcul de la sortie du réseau causée par les phases de communication entre les différents processus (processus hôte $\longleftrightarrow$ neurones, et neurone $\longleftrightarrow$ neurone).

CHAPITRE 4

APPLICATION A LA RECONNAISSANCE DE CARACTERES

1. INTRODUCTION

Dans cette partie on propose une application des réseaux de neurones pour la reconnaissance de caractères. Le but de cette application est de comparer les performances de l'implémentation logicielle à l'aide des DDE (en terme de vitesse d'apprentissage) avec celles du programme entièrement réalisé sous Matlab.

2. CARACTERES GENERES

Dans cette application on génère les images des 26 lettres de l'alphabet latin, en noir et blanc. C'est à dire que leurs matrices représentatives ne possèdent que des 0 et des 1 (image binaire).

Chaque lettre est représentée par une matrice de dimension 5x7. Ainsi la lettre A sera représentée par la matrice:

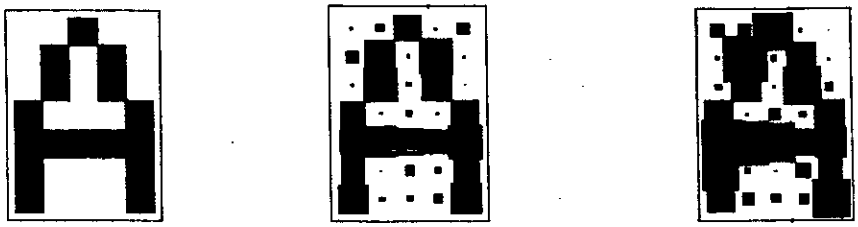
$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Pour diminuer la sensibilité du réseau vis à vis du bruit, on réalise l'apprentissage avec une base de données regroupant:

a) les 26 caractères non bruités.
b) les 26 caractères contaminés avec un bruit de distribution normale, de moyenne nulle et d'écart type 0.2.
c) les 26 caractères contaminés avec un bruit de distribution normale, de moyenne 0.1 et d'écart type 0.2.

Ces bruits sont générés grâce à la fonction "Randn" de Matlab [5].

La base d'apprentissage sera donc constituée de 78 caractères. On donne comme exemple les trois représentations graphiques de la lettre A utilisées pour l'apprentissage:



a) b) c)

Fig.4.1 : Lettre A utilisée pour l'apprentissage.
a) Lettre non bruitée b) bruit de moyenne nulle et écart type 0.2
c) bruit de moyenne 0.1 et écart type 0.2

3. SPECIFICATION DES RESEAUX

Les réseaux qu'on a utilisé sont des réseaux multicouches. Ils ont les caractéristiques suivantes:

- la fonction d'entrée des neurones est affine.
- La fonction d'activation de la couche d'entrée est la fonction sigmoïde.

On utilise deux structures différentes selon le codage choisi pour représenter la sortie du réseau:

- Dans le premier code, la couche de sortie est constituée de 26 neurones. Chaque neurone représente une lettre de l'alphabet (Tableau 1). La fonction d'activation des neurones de la couche de sortie est la fonction sigmoïde.
- Dans le deuxième code, la couche de sortie est constituée d'un seul neurone. La fonction d'activation de la couche de sortie est la fonction linéaire pour représenter chaque lettre par son numéro d'ordre (Tableau 2).

Caractère	Code 1
A	00000000000000000000000001
B	00000000000000000000000010
C	00000000000000000000000100
D	00000000000000000000001000
..	.....
..	.....
..	.....
Z	10000000000000000000000000

Tableau 1 : Codage de la sortie des réseaux a 26 neurones de sortie.

Caractère	Code 2
A	1
B	2
C	3
D	4
..	.....
..	.....
..	.....
Z	26

Tableau 2 : Codage de la sortie des réseaux a 1 neurone de sortie.

3.1 DIMENSIONS DES RESEAUX UTILISES

Les dimensions des réseaux qu'on a utilisé sont données dans le Tableau 3. On a utilisé des réseaux comportant 2 couches : une couche d'entrée et une couche de sortie.

Réseau	nbre de neurones de la couche d'entrée	nbre de neurones de la couche de sortie
1	10	26
2	5	26
3	10	1
4	15	1

Tableau 3 : Dimensions des réseaux utilisés.

4. APPRENTISSAGE

On réalise l'apprentissage des quatre réseaux précédents avec les paramètres suivants:

- coefficient d'apprentissage : 0.01 .
- momentum : 0.95 .
- facteur d'incrémentation du coefficient d'apprentissage : 1.05 .
- facteur de décrémentation du coefficient d'apprentissage: 0.4 .

Tous ces paramètres ont été choisi après plusieurs essais afin de déterminer leurs valeurs qui assurent une convergences la plus rapide possible des différents réseaux.

L'apprentissage prend fin si on atteint une erreur quadratique totale sur les sorties des quatre réseaux égale 0.5 . On obtient alors les matrices des poids et des biais des différents réseaux.

On effectue l'apprentissage de chaque réseau suivant deux méthodes:

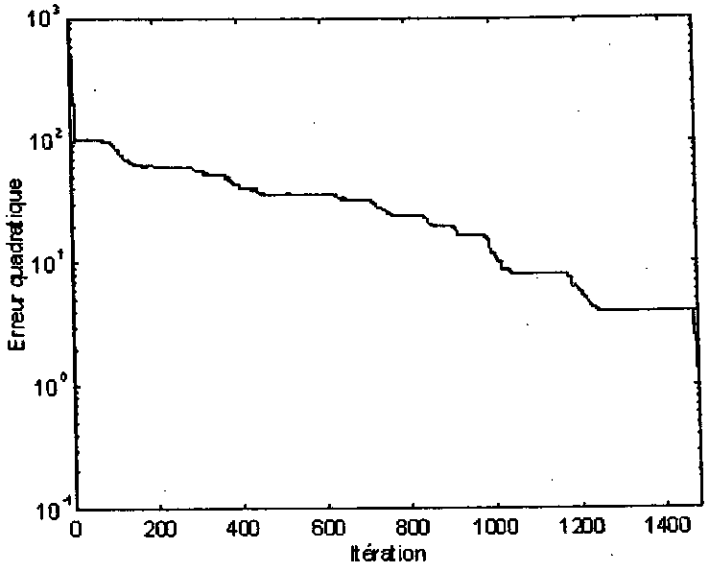
1. Apprentissage réalisé avec le programme écrit entièrement avec Matlab.
2. Apprentissage réalisé avec l'implémentation logicielle à l'aide des DDE (programme hôte sous Matlab + les programmes modélisant les neurones).

Les résultats de l'apprentissage sont donnés dans le tableau suivant:

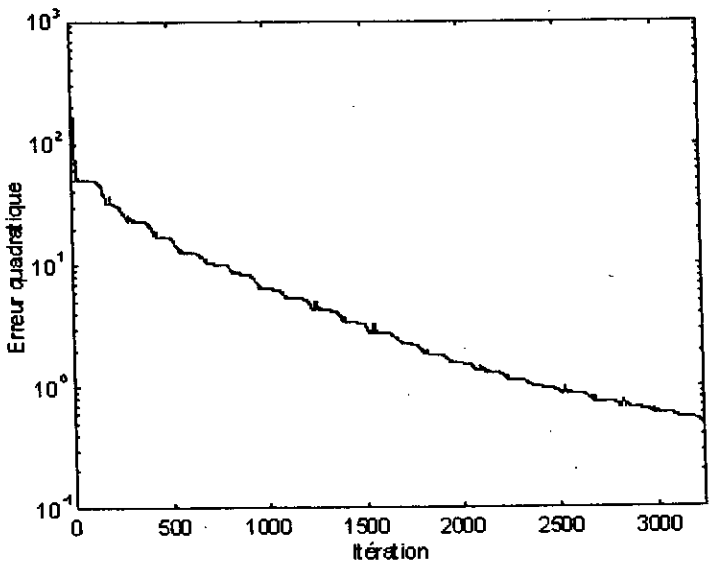
Réseau	Erreur atteinte	Temps d'apprentissage		Nombre d'itérations	
		Matlab	DDE	Matlab	DDE
1	0.1455	13min 54s	2h52min	1482	1497
2	0.4999	20min 48s	3h35 min 20s	3237	3207
3	0.4958	7min 2s	2h32 min 27s	883	872
4	0.4988	11min 47s	3h15min 32s	2084	2090

Tableau 4 : Résultats de l'apprentissage.

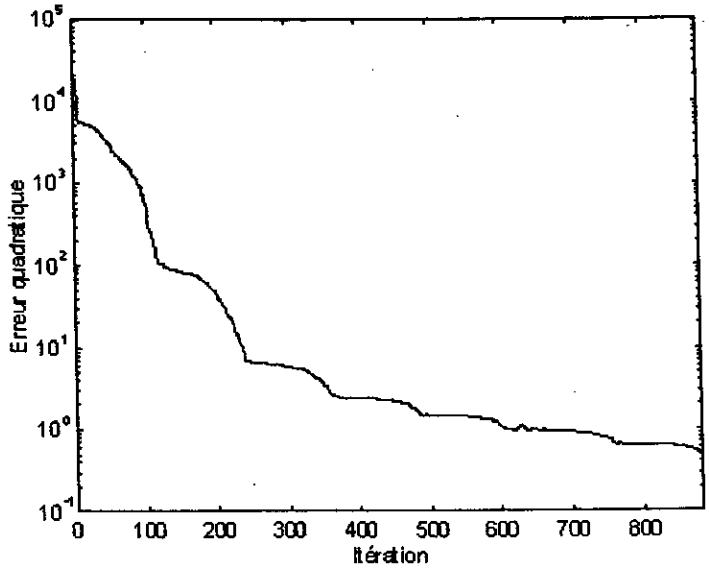
Les graphes des erreurs relatives aux quatres réseaux sont donnés par la figure suivante:


Erreur quadratique
Itération
10^3
10^2
10^1
10^0
10^{-1}
0
200
400
600
800
1000
1200
1400

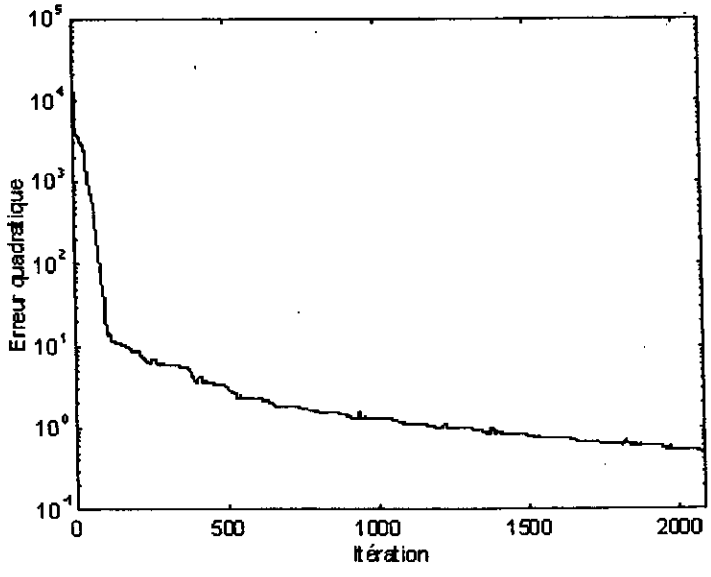
a)


Erreur quadratique
Itération
10^3
10^2
10^1
10^0
10^{-1}
0
500
1000
1500
2000
2500
3000

b)



c)



d)

Fig.4.2 : Graphes des erreurs des 4 réseaux.
a) réseau 1(10-26) , b) réseau 2 (5-26) , c) réseau 3 (10-1), d) réseau 4 (15-1).

5. CAPACITE DE GENERALISATION

La capacité de généralisation des différents réseaux se mesure en les testant avec différents vecteurs d'entrées contaminés par des bruits de moyenne et d'écart type variables. Pour cela on suit la procédure suivante:

1. on présente un vecteur d'entrée à la couche d'entrée du réseau.
2. on calcule la sortie du réseau.

une fois la sortie calculée, on passe à l'étape de décision. L'étape de décision dépend du codage utilisé pour représenter les caractères.

Pour le premier code (Tableau 1), l'algorithme de décision est:

a) Lire le vecteur de sortie **y** du réseau.

b) Repérer le numéro du neurone dont la sortie est la plus proche de 1. Ce neurone est appelé le neurone gagnant.

c) Si le numéro du neurone gagnant correspond à l'ordre de la lettre présentée alors le caractère est reconnu. Sinon, le caractère n'est pas reconnu.

Pour le deuxième code (Tableau 2) on a:

a) Lire la sortie du neurone de la couche de sortie.

b) Si la sortie du neurone correspond à l'ordre de la lettre présentée alors le caractère est reconnu. Sinon, le caractère n'est pas reconnu.

En faisant varier l'écart type du bruit entre 0 et 0.5 pour 3 moyennes du bruit (0, 0.2 et 0.5) on obtient les résultats suivants :

Moyenne / Ecart type	0	0.2	0.5
0	26	26	26
0.05	26	26	24
0.10	26	26	24
0.15	26	25	24
0.20	26	24	22
0.25	23	20	19
0.30	22	19	18
0.35	18	19	13
0.40	18	18	13
0.45	17	15	13
0.50	13	12	11

Tableau 5.a : Résultats de la phase de test du réseau 1 (10-26).

Moyenne / Ecart type	0	0.2	0.5
0	26	17	4
0.05	26	15	2
0.10	25	15	0
0.15	20	13	0
0.20	18	13	0
0.25	12	11	0
0.30	12	11	0
0.35	11	10	0
0.40	9	8	0
0.45	6	5	0
0.50	6	5	0

Tableau 5.b : Résultats de la phase de test du réseau 2 (5-26).

Moyenne / Ecart type	0	0.2	0.5
0	26	15	5
0.05	17	13	3
0.10	13	10	3
0.15	9	8	1
0.20	6	8	0
0.25	4	5	0
0.30	4	4	0
0.35	4	4	0
0.40	3	3	0
0.45	1	1	0
0.50	1	0	0

Tableau 5.c : Résultats de la phase de test du réseau 3 (10-1).

Moyenne / Ecart type	0	0.2	0.5
0	26	24	19
0.05	21	22	19
0.10	14	11	12
0.15	12	10	8
0.20	7	6	5
0.25	6	6	5
0.30	6	6	5
0.35	3	3	3
0.40	3	2	3
0.45	3	2	0
0.50	2	0	0

Tableau 5.d : Résultats de la phase de test du réseau 4 (15-1).

On représente tous ces résultats sous forme d'histogrammes (Figure 4.3).

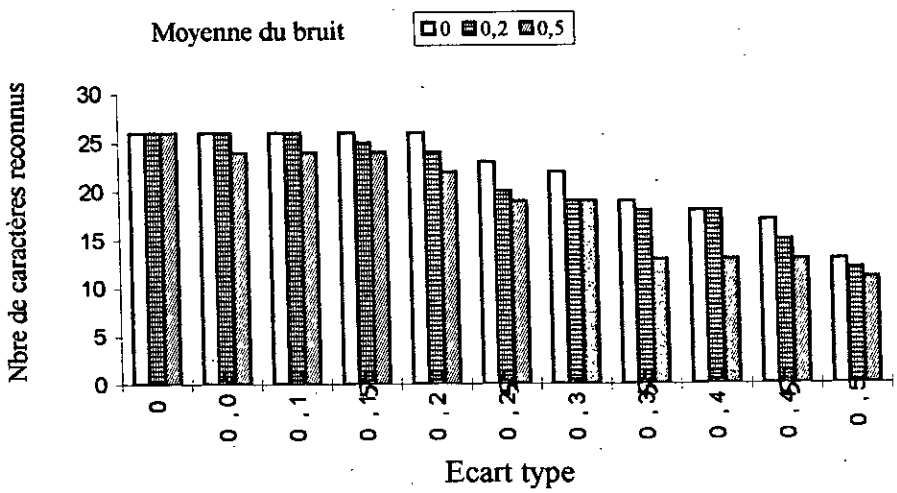


Fig.4.3.a : Résultats du réseau 1.

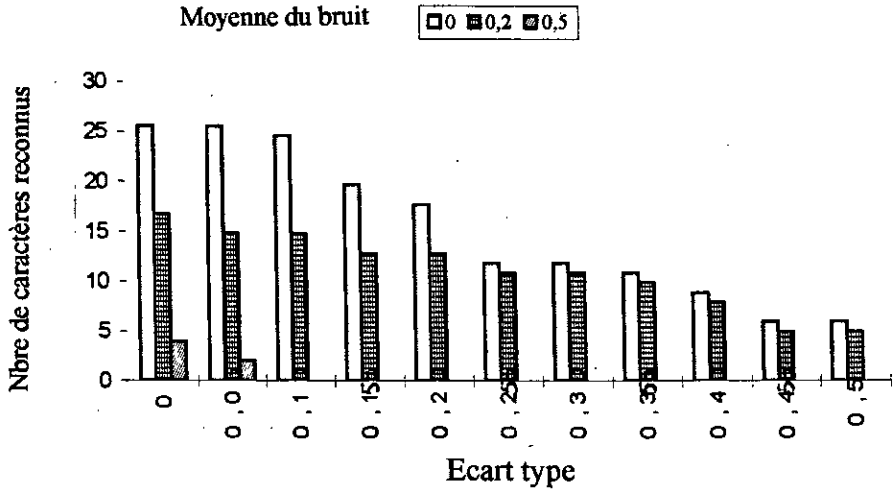


Fig.4.3.b : Résultats du réseau 2.

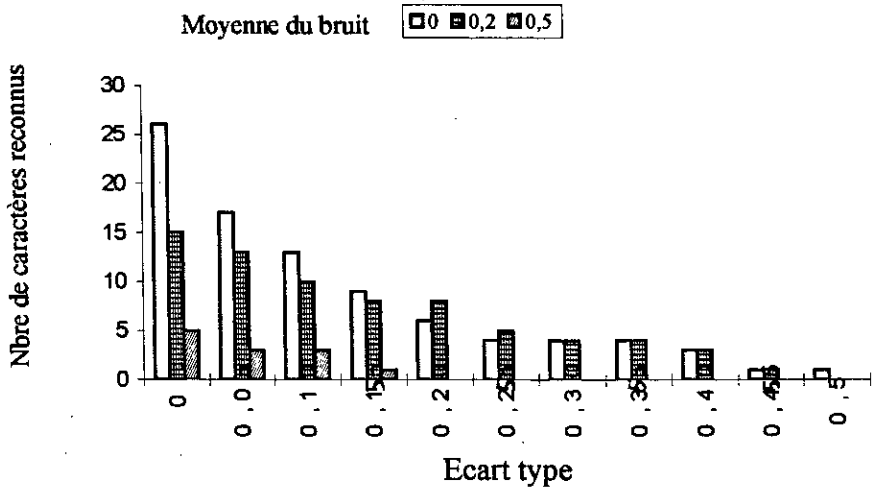


Fig.4.3.c : Résultats du réseau 3.

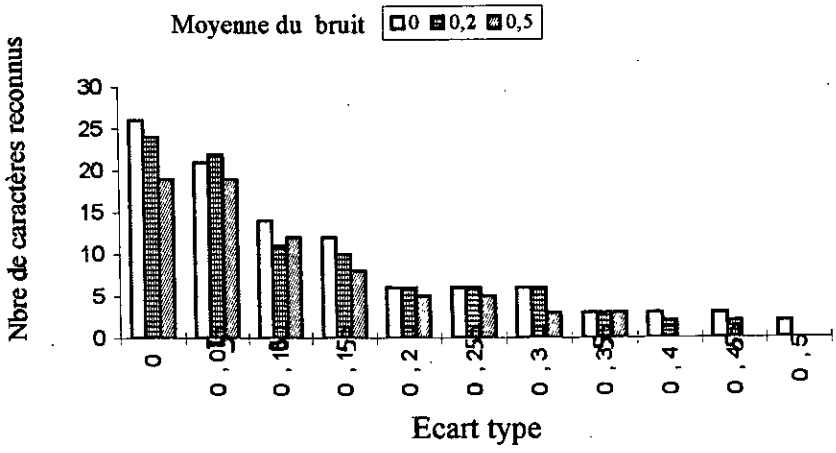


Fig.4.3.c : Résultats du réseau 4.

6. INTERPRETATION DES RESULTATS

On remarque de tous ces résultats que c'est le réseau qui comporte 10 neurones sur sa couche d'entrée et 26 neurones sur sa couche de sortie (réseau 1) qui réalise les meilleurs taux de reconnaissance. Des quatre réseaux testés, il présente donc la meilleur immunité au bruit et la meilleur capacité de généralisation : malgré que l'apprentissage ait été réalisé avec une base contaminée avec des bruit de moyennes 0 et 0.1, le réseau présente des taux de reconnaissance de :

- 77.92% pour une base de test comportant un bruit de moyenne 0.2 et de variance 0.25
- 73% pour une base de test comportant un bruit de moyenne 0.5 et de variance 0.25 .

La phase d'apprentissage des réseaux avec 26 neurones sur la couche de sortie prends beaucoup plus de temps que pour les réseaux avec un seul neurone sur la couche de sortie. Cela est du au grand nombre de connexions que comporte les réseaux 1 et 2.

Les résultats obtenus par les deux méthodes d'apprentissage (Matlab et DDE) sont exactement les mêmes en ce qui concerne les taux de reconnaissance et le nombre d'itérations nécessaire pour atteindre l'erreur désirée. La légère différence qui existe entre ces deux méthodes est due à l'initialisation aléatoire des poids au début de la phase d'apprentissage. Mais en ce qui concerne le temps nécessaire pour atteindre l'erreur désirée, on note une très grande différence entre les deux méthodes. L'implémentation utilisant les DDE nécessite un temps beaucoup plus grand que le programme écrit avec Matlab pour atteindre la même erreur finale. Cela est causé par :

1) Les phases de communication DDE entre les neurones, et entre les neurones et Matlab ralentissent les calculs et la mise à jours des poids synaptiques du réseau.

2) L'utilisation des DDE nous oblige à transmettre les vecteurs d'entrée de chaque neurone élément par élément, ce qui conduit à l'utilisation de nombreuses boucles imbriquées et ralentit d'autant les calculs.

CONCLUSION

L'application qu'on a proposé nous a permis de comparer entre notre implémentation logicielle et une simulation classique des réseaux de neurones.

L'implémentation logicielle des réseaux de neurones à l'aide des DDE permet de réaliser les phases d'apprentissage et de test d'un réseau multicouche destiné a une application déterminée. Mais la phase d'apprentissage prend beaucoup plus de temps qu'une simulation séquentielle classique du réseau.

L'intérêt de cette implémentation n'est donc pas de se substituer aux simulations classiques, mais de simuler un fonctionnement réel (parallèle) des réseaux de neurones. Cette implémentation pourra servir de base à une implémentation matérielle digitale qui permettra de profiter de tous les avantages des réseaux de neurones. En effet cette implémentation logicielle permet de simuler un fonctionnement parallèle de plusieurs neurones et de voir les problèmes de synchronisation qui existent lorsqu'on exécute plusieurs programmes en même temps et devant communiquer entre eux.

CONCLUSION GENERALE

L'objectif de notre travail était d'implémenter les réseaux de neurones sur un P.C., ce qu'on a réalisé grâce à une implémentation logicielle utilisant l'échange dynamique de données (DDE). Cette implémentation permet de simuler dans un environnement multitâches un fonctionnement parallèle d'un réseau de neurones multicouches. Cette simulation permet de se rapprocher du fonctionnement naturel des réseaux neuronaux.

Dans notre implémentation ont s'est limité aux réseaux multicouches car ce sont des réseaux qui peuvent être utilisé pour résoudre de grands nombres de problèmes, et parce que leur simulation fait intervenir les deux notions de parallélisme vuent dans le deuxième chapitre: le parallélisme de contrôle et le parallélisme de flux. Ce double parallélisme complique la simulation de ces réseaux et fait ressortir l'importance de la synchronisation entre processus dans toute exécution parallèle.

La conception des neurones du réseau comme des cellules élémentaires qu'on peut dupliquer et aux fonctions simples, permet d'implémenter non seulement le modèle des réseaux multicouches mais aussi n'importe quel modèle de réseaux. Cela est possible à cause de la programmabilité des neurones. On peut programmer leurs fonctions d'activations, le nombre de leurs entrées, les connexions entre neurones... .

L'application des réseaux de neurones qu'on a proposé, la reconnaissance de caractères, a servi à comparer les performances de l'implémentation réalisée à l'aide des DDE avec celles d'une simulation séquentielle classique de ces réseaux. Ces performances sont mesurées en terme de temps d'apprentissage. On a remarqué que les deux méthodes donnaient les mêmes résultats pour le nombre d'itérations nécessaire à la convergence vers une erreur déterminée. Mais l'implémentation à l'aide des DDE nécessite une durée d'apprentissage beaucoup plus grande. Cela est dû essentiellement aux problèmes de communication.

L'intérêt de notre implémentation est qu'elle simule sur un P.C. le fonctionnement d'un réseau de neurones sur un réseau de P.C. C'est une première étape vers une implémentation des réseaux de neurones sur un tel réseau. Une telle implémentation permettra de profiter pleinement des avantages qu'offrent les réseaux de neurones en terme de vitesse de calcul et de robustesse aux pannes.

Notre travail peut aussi constituer un outil d'initiation aux techniques neuronales et cela à travers le programme développé avec Matlab. En effet, il se présente sous la forme d'une

interface graphique réalisée sous windows simple d'utilisation qui permet d'aborder le domaine des réseaux neuronaux sans rebuter l'utilisateur.

Le principal problème de notre implémentation est la lenteur de l'apprentissage. Cela est dû au principe des DDE : à chaque transaction on ne peut envoyer qu'une seule donnée. La transmission d'un vecteur se fera donc élément par élément ce qui retarde les calculs. Une amélioration éventuelle à apporter à notre travail serait donc d'améliorer les phases de communication entre processus. Cela peut être réalisé en ayant recours à d'autres méthodes de communication qui permettent la transmission de vecteurs en une seule transaction, ce qui accélérera la vitesse d'apprentissage. Ces nouvelles méthodes de communication peuvent être développées à l'aide de langages permettant la programmation windows (tel que le Pascal sous windows ou le C++). Notre implémentation pourra donc être améliorée en changeant les parties gérant la communication entre les neurones et le processus hôte.

BIBLIOGRAPHIE

[1] J. Hérault, C. Jutten, *Réseaux neuronaux et traitement du signal*, Ed. Hermès, Paris, 1994.

[2] E. Davalo, P. Naïm, *Des réseaux de neurones*, Ed. Eyrolles, Paris, 1990.

[3] M. Weinfeld, "Réseaux de neurones", Techniques de l'ingénieur, traité Informatique, H 1990.

[4] J. A. Freeman, D. M. Skapura, *Neural networks applications and programming techniques*, Ed. Adison Wesley publishing company, Houston, 1991.

[5] H. Demuth, M. Beale, *Neural Network Toolbox : user's guide*, The MathWorks Inc, June 1992.

[6] J. D. Chatelain, R. Dessoulavy, *Electronique Volume VIII*, Presses Polytechniques romandes, Lausanne, 1985.

[7] J. A. Vlontzos, S. Y. Kung, " Digital neural network architecture and implementation", VLSI Design of Neural Networks, Kluwer Academic Publishers, 1991.

[8] S. Y. Kung, "Parallel architectures for artificial neural nets", Internationnal Conference on Systolic Arrays, Computer Society Press, May 25-27, 1988.

[9] J. L. Jacquemin, *Informatique parallèle et systèmes multiprocesseurs*, Ed. Hermès, 1993.

[10] J. P. Sansonet, " Architecture des ordinateurs parallèles", Techniques de l'ingénieur, traité Informatique, H 170.

[11] *Building a graphical user interface*, The MathWorks Inc, June 1993.

[12] *Windows API guide Volume 3*, Borland , 1992.

[13] B. Frala, *Turbo Pascal facile sous windows*, Marabout, 1992.

ANNEXE

PROGRAMME PARAMETRANT UN NEURONE

1) PROGRAMME 'N1.PAS' PARAMETRANT LE NEURONE 1 :

```
PROGRAM Model_Neurone;
uses Strings, WinTypes, WinProcs, OWindows, ODialogs, Win31, DDEML,
  ShellAPI, BWCC, Data2;
var
   DataEntryName : PChar ;

   { Variables de communication DDE }
   Inst, Inst2         : Longint;
   CallBack, CallBackPtr      : TCallback;
   ServiceHSz, ServiceHSz2    : HSz;
   TopicHSz,TopicHSz2  : HSz;
   ItemHSz             : array [1..NumValues] of HSz;
   ConvHdl   : HConv;
   ConvHdl2  : HConv;

   { Paramètres du neurone }
   DataSample : TData;
   Poids_syn  : Poids;
   Fct    : Fct_activ;
   nb_e   : Nbr_entree;
   B      : Biais;
   OK     : Validation;
   N_s    : Neur_suiv;
   S      : N_sortie;
   K      : Integer;  { Indice sur le vecteur des entrées}
   L      : Integer;  { Indice sur le vecteur des poids}

type
{ Fenetre Principale de l'Application }
  PDDEServerWindow = ^TDDEServerWindow;
  TDDEServerWindow = object(TWindow)
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    destructor  Done; virtual;
    procedure SetupWindow; virtual;
    procedure WMLButtonDown(Var Msg : TMessage);Virtual
     wm_First+wm_LButtonDown;

    function  MatchTopicAndService(Topic, Service: HSz): Boolean; virtual;
    function  MatchTopicAndItem(Topic, Item: HSz): Integer; virtual;
    function  WildConnect(Topic, Service: HSz;
      ClipFmt: Word): HDDEData; virtual;
    function  AcceptPoke(Item: HSz; ClipFmt: Word;
          Data: HDDEData): Boolean; virtual;
    function  DataRequested(TransType: Word; ItemNum: Integer;
      ClipFmt: Word): HDDEData; virtual;
  end;

 { Objet TApplication  }
  TDDEServerApp = object(TApplication)
    procedure InitMainWindow; virtual;
```

```
 end;
{ Objet TApplication }
const
 DemoTitle  : PChar = 'Neurone1';
{ Global variables }
var
 App: TDDEServerApp;
{ Calcul de la sortie du neurone }
function Sortie : Real;
var
    I    : Integer;
    somme : real;
begin
 somme := 0;
 for I:= 1 to nb_e do
  somme := somme + Datasample[I]*Poids_syn[I];
 somme := somme + B;
 if (Fct=1) then     { Fct d'activation Sigmoide }
  begin
   somme := 1/(1+exp(- somme));
  end;
 if (Fct=2) then     { Fct d'activation Tan_Hyp }
  begin
   somme := 2 /(1 + exp(-2* somme))-1;
  end;
 Sortie := somme ;
 end;


{ Function "CallBack" pour DDEML }
{ Cette fonction répond a toutes les transactions générées par DDEML.}
function CallbackProc(CallType, Fmt: Word; Conv: HConv; HSz1, HSz2: HSZ;
 Data: HDDEData; Data1, Data2: Longint): HDDEData; export;
var
 ThisWindow: PDDEServerWindow;
 ItemNum  : Integer;
begin
 CallbackProc := 0;   { See if proved otherwise }
 ThisWindow := PDDEServerWindow(App.MainWindow);
 case CallType of
  xtyp_Register:
   begin
       { retourne 0 }
   end;
  xtyp_Unregister:
   begin
       { retourne 0 }
   end;
  xtyp_xAct_Complete:
   begin
       { returne 0 }
   end;
  xtyp_Disconnect:
```

```
    begin
        { retourne 0 }
    end;
  xtyp_WildConnect:
    CallbackProc := ThisWindow^.WildConnect(HSz1, HSz2, Fmt);
  xtyp_Connect:
    if Conv = 0 then
    begin
        if ThisWindow^.MatchTopicAndService(HSz1, HSz2) then
          CallbackProc := 1;   { Connecté! }
    end;
{ Quand une connexion est confirmée on enregistre son Handle.}
  xtyp_Connect_Confirm:
    ConvHdl := Conv;
{ Une application fait une demande de donées (Data Raquest).}
  xtyp_Request:
    begin
        ItemNum := ThisWindow^.MatchTopicAndItem(HSz1, HSz2);
        if ItemNum > 0 then
          CallbackProc := ThisWindow^.DataRequested(CallType, ItemNum, Fmt);
    end;
{ Répond a un envoi de données }
  xtyp_Poke:
    begin
        if ThisWindow^.AcceptPoke(HSz2, Fmt, Data) then
          CallbackProc := dde_FAck;
    end;
 end; { Case CallType }
end;
{ Les Méthodes de TDDEServerWindow }
constructor TDDEServerWindow.Init(AParent: PWindowsObject; ATitle: PChar);
var
 I : Integer;
begin
 TWindow.Init(AParent, ATitle);
 Inst     := 0; Inst2 :=0;    { Doit etre a zéro au premier appel de DdeInitialize }
 @CallBack := nil;   { MakeProcInstance est applelée dans SetupWindow   }
 for I := 1 to Max_entree do
   begin
    DataSample[I]:= 0;   { Initialisation des }
    Poids_syn[I] := 0;          {  entrées et des poids}
   end;
  Fct := 1;   { Fonction d'activation }
  nb_e := 20 ; { Le nombre des entrées du neurone }
  B := 0;     { Biais du neurone }
  S := 0;     { Sortie du neurone }
  K := 1;
  L := 1;
end;

Procedure TDDEServerWindow.WMLButtonDown(Var Msg:TMessage);
 Var
   Valeur:Integer;
```

```
begin
  ShowWindow(hWindow,sw_Minimize)
end;
{ Détruit l'instance de l'application et fait Appel DdeUninitialize pour terminer
 les conversations DDE.
}
destructor TDDEServerWindow.Done;
var
 I : Integer;
begin
 if ServiceHSz <> 0 then
  DdeFreeStringHandle(Inst, ServiceHSz);
 if TopicHSz <> 0 then
  DdeFreeStringHandle(Inst, TopicHSz);
 for I := 1 to NumValues do
  if ItemHSz[I] <> 0 then
   DdeFreeStringHandle(Inst, ItemHSz[I]);
 if Inst <> 0 then
  DdeUninitialize(Inst);
 if @CallBack <> nil then
  FreeProcInstance(@CallBack);
 TWindow.Done;
end;
{ Complète l'initialisation de la fenetre de l'application.
}
procedure TDDEServerWindow.SetupWindow;
var
 I : Integer;
begin
 TWindow.SetupWindow;
 @CallBack:= MakeProcInstance(@CallBackProc, HInstance);
 if DdeInitialize(Inst, CallBack, 0, 0) = dmlErr_No_Error then
 begin
  ServiceHSz:= DdeCreateStringHandle(Inst, DataEntryName, cp_WinAnsi);
  TopicHSz  := DdeCreateStringHandle(Inst, DataTopicName, cp_WinAnsi);
  for I := 1 to NumValues do
   ItemHSz[I] := DdeCreateStringHandle(Inst, DataItemNames[I],
       cp_WinAnsi);
  if DdeNameService(Inst, ServiceHSz, 0, dns_Register) = 0 then
  begin
   MessageBox(HWindow, 'Registration failed.', Application^.Name,
       mb_IconStop);
   PostQuitMessage(0);
  end;
 end
 else
  PostQuitMessage(0);
end;

procedure PokeData(I : integer);
var Err : integer;
   Datastr : TDataString;
begin
```

```
@CallBackPtr := MakeProcInstance(@CallBackProc, HInstance);
{ Initialise les  DDE
}
 if @CallBackPtr <> nil then
 begin
  if DdeInitialize(Inst, TCallback(CallBackPtr), 0,
   0) = dmlErr_No_Error then
  begin
   Str(I, DataStr);
   ServiceHSz2:= DdeCreateStringHandle(Inst, DataStr, cp_WinAnsi);
   TopicHSz2  := DdeCreateStringHandle(Inst, 'entree', cp_WinAnsi);
   if (ServiceHSz2 <> 0) and (TopicHSz2 <> 0)  then
   begin
       ConvHdl2 := DdeConnect(Inst, ServiceHSz2, TopicHSz2, nil);
       if ConvHdl2 = 0 then
       begin
        MessageBox(0, 'Ne peut pas établir la Conversation!',
         Application^.Name, mb_IconStop);
        PostQuitMessage(0);
       end
   end
   else
   begin
       MessageBox(0, 'Ne peut pas créer les Handles!', Application^.Name,
        mb_IconStop);
       PostQuitMessage(0);
   end
  end
  else
  begin
   MessageBox(0, 'Ne peut pas initialiser!', Application^.Name,
       mb_IconStop);
   PostQuitMessage(0);
  end;
 end;
end;
{ Retourne la valeur TRUE si le "Topic" et "Service" correspondent à ceux
 supportés par cette application, ou FALSE dans le cas contraire.
}
function TDDEServerWindow.MatchTopicAndService(Topic, Service: HSz): Boolean;
begin
 MatchTopicAndService := False;
 if DdeCmpStringHandles(TopicHSz, Topic) = 0 then
  if DdeCmpStringHandles(ServiceHSz, Service) = 0 then
   MatchTopicAndService := True;
end;
{ Determine si le "Topic" et "Item" correspondent à ceux
 supportés par cette application.  Retourne le numéro de l'Item
 trouvé, et zero si aucun Item ne correspond.
}
function TDDEServerWindow.MatchTopicAndItem(Topic, Item: HSz): Integer;
var
 I : Integer;
```

```
begin
  MatchTopicAndItem := 0;
  if DdeCmpStringHandles(TopicHSz, Topic) = 0 then
    for I := 1 to NumValues do
      if DdeCmpStringHandles(ItemHSz[I], Item) = 0 then
          MatchTopicAndItem := I;
end;
{ Wildcard connect
}
function TDDEServerWindow.WildConnect(Topic, Service: HSz;
  ClipFmt: Word): HDDEData;
var
  TempPairs: array [0..1] of THSZPair;
  Matched  : Boolean;
begin
  TempPairs[0].hszSvc  := ServiceHSz;
  TempPairs[0].hszTopic:= TopicHSz;
  TempPairs[1].hszSvc  := 0;
  TempPairs[1].hszTopic:= 0;
  Matched := False;
  if (Topic= 0) and (Service = 0) then
    Matched := True
  else
    if (Topic = 0) and (DdeCmpStringHandles(Service, ServiceHSz) = 0) then
      Matched := True
    else
      if (DdeCmpStringHandles(Topic, TopicHSz) = 0) and (Service = 0) then
          Matched := True;
  if Matched then
    WildConnect := DdeCreateDataHandle(Inst, @TempPairs, SizeOf(TempPairs),
      0, 0, ClipFmt, 0)
  else
    WildConnect := 0;
end;
{ Accepte un envoi de données "Poke requests" .}
function TDDEServerWindow.AcceptPoke(Item: HSz; ClipFmt: Word;
  Data: HDDEData): Boolean;
var
  DataStr          : TDataString;
  Err, J           : Integer;
  TempSample, Tpoids : Real;
  TFct_activ       : Fct_activ;
  TNbr_entree      : Nbr_entree;
  TBiais           : Biais;
  TOK              : Validation;
  TNBr             : NBr_neur;
  TN_s             : Word;
begin
  { Data1  : Les Entrées du neurones }
  if (DdeCmpStringHandles(Item, ItemHSz[1]) = 0) and (ClipFmt = cf_Text) then
  begin
    DdeGetData(Data, @DataStr, SizeOf(DataStr), 0);
    Val(DataStr, TempSample, Err);
```

```
   DataSample[K] := TempSample;
   K := K+1;
   if (K = nb_e+1)  then
   begin
     K := 1;
     S := Sortie;   { Calcul de la sortie du neurone}
end;
  end;

 { Data2  : Les Poids synaptiques }
 if (DdeCmpStringHandles(Item, ItemHSz[2]) = 0) and (ClipFmt = cf_Text) then
  begin
   DdeGetData(Data, @DataStr, SizeOf(DataStr), 0);
   Val(DataStr, TPoids, Err);
            Poids_syn[L] := TPoids;
            L := L+1;
            if L = nb_e+1 then
               L := 1;
  end;

  { Data3  : La fonction d'activation du neurone }
  if (DdeCmpStringHandles(Item, ItemHSz[3]) = 0) and (ClipFmt = cf_Text) then
  begin
   DdeGetData(Data, @DataStr, SizeOf(DataStr), 0);
   Val(DataStr, TFct_activ, Err);
            Fct := TFct_activ;
  end;

  { Data4  : Le nombre des entrées du neurones }
  if (DdeCmpStringHandles(Item, ItemHSz[4]) = 0) and (ClipFmt = cf_Text) then
  begin
   DdeGetData(Data, @DataStr, SizeOf(DataStr), 0);
   Val(DataStr, TNbr_entree, Err);
            nb_e := TNbr_entree;
  end;

  { Data5  : Le biais du neurone }
  if (DdeCmpStringHandles(Item, ItemHSz[5]) = 0) and (ClipFmt = cf_Text) then
  begin
   DdeGetData(Data, @DataStr, SizeOf(DataStr), 0);
   Val(DataStr, TBiais, Err);
            B := TBiais;
  end;

  { Data6  : Variable de control  }
  if (DdeCmpStringHandles(Item, ItemHSz[6]) = 0) and (ClipFmt = cf_Text) then
  begin
   DdeGetData(Data, @DataStr, SizeOf(DataStr), 0);
   Val(DataStr, TOK, Err);
   OK := TOK;
   if OK = 1 then
   begin
    Str(S, DataStr);
```

```
  PokeData(N_s);   { connexion aux autres neurones}
  DdeClientTransaction(@DataStr, StrLen(DataStr) + 1, ConvHdl2,
    ItemHSz[1], cf_Text, xtyp_Poke, 1000, nil);
  DdeDisconnect(ConvHdl2);
 end;
 OK:=0;
end;

{ Data7  : "Adresse" des neurones de la couche suivante}
 if (DdeCmpStringHandles(Item, ItemHSz[7]) = 0) and (ClipFmt = cf_Text) then
 begin
  DdeGetData(Data, @DataStr, SizeOf(DataStr), 0);
  Val(DataStr, TN_s, Err);
          N_s := TN_s;
 end;
 AcceptPoke := True;
end;

{ Retourne les données demandées par une autre application}
function TDDEServerWindow.DataRequested(TransType: Word; ItemNum: Integer;
  ClipFmt: Word): HDDEData;
var
  ItemStr: TDataString;   {défini dans l'unité DATA2.TPW}
begin
  if ClipFmt = cf_Text then
  begin
   if ItemNum = 8 then
   begin
    Str(S, ItemStr);
    DataRequested := DdeCreateDataHandle(Inst, @ItemStr, StrLen(ItemStr) + 1,
      0, ItemHSz[8], ClipFmt, 0);
   end;
  end
  else
    DataRequested := 0;
end;

{ Les Méthodes de TDDEServerApp  }
procedure TDDEServerApp.InitMainWindow;
Var
  Valeur:Integer;
begin
 Valeur:=MessageBox(0,' neurone 1',
           '- Information -',mb_IconAsterisk);
 IF valeur=id_OK Then
  begin
   MainWindow := New(PDDEServerWindow, Init(nil, Application^.Name));
  end
end;

{ Programme Principal }
BEGIN
  DataEntryName := '1';
```

```
 App.Init(DemoTitle);
 App.Run;
 App.Done;
END.
```

2) UNITE 'NDATA.PAS' :

```
{*************************************************}
{                                                 }
{   Unité Définissant les variables du neurone    }
{                                                 }
{*************************************************}

{Cette unité définit la structure des données qui sont utilisée
 par le les neurones.}
unit NData;
interface
const
  NumValues = 8;
  Max_entree = 40;
type
  TDataString = array [0..20] of Char;
{ Structure des données}
  TData      = array [1..Max_entree] of Real;
  Poids      = array [1..Max_entree] of Real;
  Fct_activ  = word;
  Nbr_entree = Integer;
  Ligne      = integer;
  Biais      = Real;
  Depart     = Word;
  N_sortie   = Real;
const
  {DataEntryName : PChar = 'neurone1';}
  DataTopicName : PChar = 'entree';
  DataItemNames : array [1..NumValues] of PChar = ('Data1',
                                                   'Data2',
                                                   'Data3',
                                                   'Data4',
                                                   'Data5',
                                                   'Data6',
                                                   'Data7',
                                                   'Data8');
implementation
end.
```