Projet de programmation objet en C++ :
« Programmation de réseaux de neurones »

I) Présentation du projet

I.1) Introduction

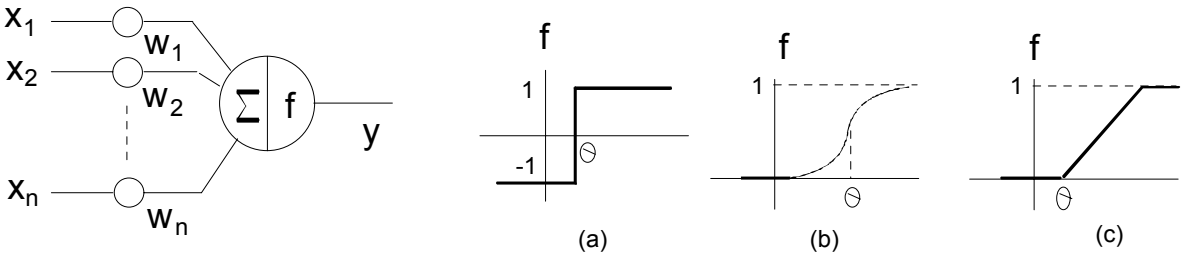
Le domaine des réseaux de neurones est une branche de l'intelligence artificielle permettant de réaliser des systèmes adaptatifs, fonctionnant par apprentissage.

Leur champ d'application est vaste : ils peuvent être appliqués à des problèmes de reconnaissance de formes (écriture manuscrite, parole, visages, etc), d'optimisation, de contrôle automatique, de traitement d'image, etc.

I.2) Qu'est-ce qu'un neurone ?

Les neurones dont on parle ici sont des modèles très grossiers des neurones naturels. Le modèle de neurone le plus utilisé possède une sortie et plusieurs entrées. Il réalise une simple somme pondérée de ses entrées et passe le résultat dans une fonction d'activation. Les coefficients de pondération sont appelés poids synaptiques (par analogie avec les neurones naturels), ou plus simplement poids. En général, ils sont adaptés au problème à traiter lors d'une phase d'apprentissage.

La fonction d'activation peut avoir diverses formes (voir schéma ci-dessous) : **(a)** seuil logique, **(b)** sigmoïdale ou **(c)** linéaire avec ou sans saturation.



Pour la fonction "seuil logique" (a), on peut avoir une sortie binaire : 0 ou 1, ou une sortie dite bipolaire : –1 ou +1.

Une fonction couramment utilisée pour *f* est la fonction sigmoïde (b), définie par :

$$f(x) = \frac{1}{1 + e^{-a.(x-\theta)}}$$

Le paramètre *a* permet de régler l'importance de la non-linéarité, puisque la pente au niveau du seuil θ est égale à *a*. Lorsque *a* tend vers l'infini, la forme de la courbe tend vers le seuil logique.

On utilise également souvent la tangente hyperbolique f(x)=tanh(x), qui possède une forme proche de la sigmoïde mais varie entre –1 et +1.

L'opération réalisée par ce neurone sur ses entrées est définie par :

$$y = f(\sum_i w_i x_i).$$

où x_i sont les entrées du neurone, w_i les poids associés à ces entrées et f la fonction d'activation. y est appelée sortie du neurone.

L'opération terme à terme $\sum_i w_i x_i$ est également appelée produit de convolution, et peut se noter :

$$W * X$$

où W et X sont les vecteurs des poids du neurone et de ses entrées, respectivement.

Les neurones sont regroupés ensemble pour former des réseaux. Pour créer ces réseaux, la sortie d'un neurone constitue l'entrée d'un ou plusieurs autres.

Il existe de nombreux modèles de réseaux de neurones, dont les plus connus sont peut-être :

- le **réseau d'Hopfield** est un réseau complètement interconnecté, c'est à dire dont tous les neurones sont connectés avec tous les autres, sauf avec eux-mêmes ;
- les réseaux multicouche avec apprentissage par **rétro-propagation du gradient** de l'erreur de sortie ;
- les **cartes auto-oganisatrices de Kohonen** ;
- les **fonctions radiales de base** ;
- etc.

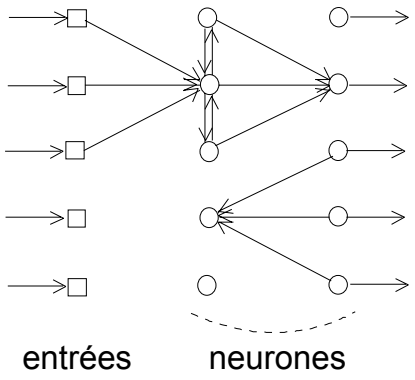
Ces différents modèles sont caractérisés par :

- leur **architecture**, c'est à dire la façon dont les neurones sont interconnectés,
- l'**algorithme d'apprentissage**, c'est à dire la règle de modification des poids, et
- le mode de **mise à jour des sorties** des neurones : synchrone ou asynchrone aléatoire, etc.

I.3) Architecture des réseaux de neurones

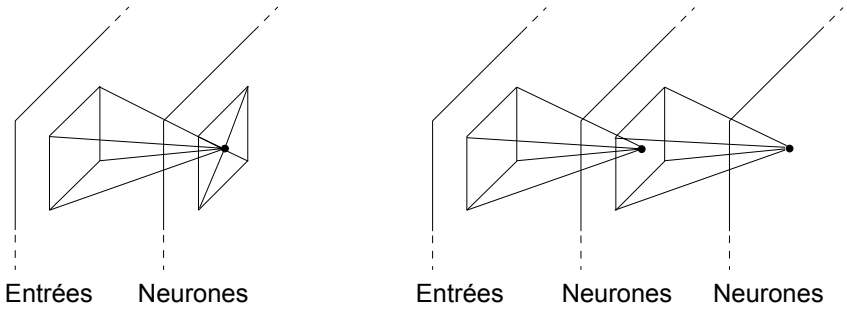
Les réseaux peuvent avoir une structure à 1 dimension ou à 2 dimensions.

Dans le cas d'une dimension, un réseau de neurones peut possèder des connexions directes (flèches vers la droite sur la figure ci-dessous), latérales (flèches vers le haut et le bas) et, plus rarement, de retour (flèches vers la gauche).



Par exemple, le réseau complètement interconnecté d'Hopfield ne comporte que des connexions latérales.

Le cas de 2 dimensions et bien adapté au traitement des images (analyse, traitement, reconnaissance, etc). La figure ci-dessous représente un réseau à 2 dimensions, à 1 couche (a) ou à 2 couches (b).

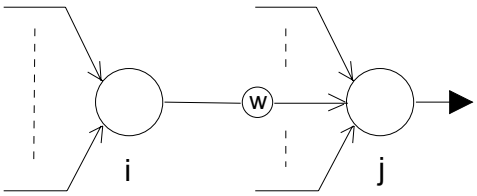


(a) (b)

L'ensemble des connexions d'entrée d'un neurone sur une couche est appelé champ récepteur. Le champ récepteur d'un neurone désigne l'ensemble des autres neurones qu'il "perçoit", par l'intermédiaire de leur sortie. Par abus de langage, un champ récepteur désignera également un ensemble de liens. Un neurone peut avoir plusieurs champs récepteurs, sur des couches différentes ou sur différents endroits d'une même couche. Par exemple, dans le cas (a), la couche de neurones possède un champ récepteur sur la couche d'entrée et un autre sur elle-même ; dans le cas (b), la 1ère couche de neurones possède un champ récepteur sur la couche d'entrée, et la 2e couche de neurone possède un champ récepteur sur la 1ère.

I.4) Apprentissage

La connexion entre 2 neurones est affectée d'un poids synaptique. Dans la plupart des modèles de réseaux de neurones, ce poids est modifié par apprentissage.



L'apprentissage consiste à modifier les poids du réseau en fonction de couples entrée/sortie désirées (on parle alors d'apprentissage supervisé) ou indépendamment d'une sortie désirée (on parle alors d'auto-organisation). L'apprentissage est itératif : il est obtenu par des petites adaptations successives des poids. Soit $w_{ij}(t)$ le poids de la connexion entre les neurones i et j, à un instant donné (discrêt) t. On écrit la nouvelle valeur du poids, après une itération d'apprentissage, de la manière suivante :

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}$$

Voici 3 exemples de règles d'apprentissage courantes (on ommet l'indice t pour alléger les relations) :

- règle de Hebb :

$$\Delta w_{ij} = \alpha . y_i . y_j$$

- règle d'apprentissage compétitif :

$$\Delta w_{ij} = \alpha . (y_i - w_{ij}) . y_j$$

- règle d'apprentissage supervisé :

$$\Delta w_{ij} = \alpha . y_i . (y_{j,d} - y_j)$$

où :

- w_{ij} est le poids de la connexion entre les neurones i et j ; cette connexion est un lien d'entrée du neurone j ;
- y_i et y_j sont respectivement les sorties des neurones source i et cible j ;
- α est un coefficient d'apprentissage, en général inférieur ou égal à 1 : il règle la quantité d'adaptation des poids à chaque itération ;
- $y_{j,d}$ est la sortie désirée du neurone j.

I.5) Activation

L'activation d'un neurone consiste à calculer sa sortie telle qu'elle a été définie plus haut avec les différentes fonctions d'activation possibles.

Dans le cas de réseaux à couches ne possédant que des connexions directes, l'activation des différentes couches est réalisée de manière synchrone : la sortie de tous les neurones de la 1ère couche est calculée, puis celle de la 2e, et ainsi de suite.

Dans le cas d'un réseau possédant des connexions latérales comme le réseau de Hopfield, l'ordre de mise à jour des sorties des différents neurones est important, car chaque nouvelle sortie va pouvoir modifier le calcul de la sortie d'un autre neurone. Dans ce cas-là en général on utilise 2 grands modes de mise à jour des sorties

- Asynchrone aléatoire : les neurones dont les sorties sont mises à jour sont choisis aléatoirement ;
- Synchrone : toutes les sorties sont mises à jour en même temps, en fonction de leurs valeurs précédentes .

II) Programmation en C++

La programmation objet est bien adaptée à la programmation des réseaux de neurones : ces derniers possèdent une structure hiérarchique, dans laquelle :

- un réseau est constitué d'un ou plusieurs couches,
- un couche est constituée par un ensemble de neurones,
- un neurone possède un ou plusieurs champs récepteurs (pour faciliter la programmation on ne considèrera que le cas d'un seul),
- un champ récepteur est un ensemble de liens.

A partir de ces considérations, on peut déduire que le le programme permettant de tester des réseaux de neurones doit comporter :

- des classes
- des fonctions (ou méthodes)
  - de création, réalisant entre autres l'allocation-mémoire (=constructeurs)
  - de connexion
  - d'activation
  - d'apprentissage

Les classes comportent les membres (ou variables) associés à l'entité qu'elles représentent (par exemple une classe *Neurone* doit comporter un membre correspondant à la sortie d'un neurone).

II.1) Classes

Le programme devra comporter au moins les 3 classes suivantes :

- Réseau
- Couche
- Neurone

Du fait qu'un neurone possède en général plusieurs champs récepteurs, donc chacun peut comporter plusieurs liens d'entrée, on peut également constituer une classe "champ récepteur" et une classe "lien".

II.2) Fonctions de création

Les allocations-mémoire nécessaires aux réseaux de neurones sont réalisées par les *constructeurs* des classes.

Du fait qu'il existe une grande variété de réseaux de neurones différents, il paraît difficile de paramétrer toute l'architecture du réseau lors de l'appel du constructeur de réseau. Il y aurait trop de paramètres à lui passer. Par contre on peut envisager de définir tous les paramètres de l'architecture à l'intérieur de ce constructeur. Il faudrait alors sélectionner le type de réseau par simple passage d'un paramètre entier, par exemple avec :

```
Reseau reseau(int type);
```

A l'intérieur du constructeur d'un réseau, on peut alors effectuer toutes les allocations-mémoire nécessaires à la création du réseau. Le constructeur de réseau peut en effet faire

appel au constructeur de couches, qui peut lui-même faire appel lui-même au constructeur de neurones, etc.

Pour limiter le nombre de paramètres à passer au constucteur d'une couche, on peut envisager de lui passer un modèle de champ récepteur, créé auparavant.

La syntaxe de la méthode de création d'un champ récepteur pourrait être par exemple :

```
ChR champ(nb_liens_l, nb_liens_h);
```

La syntaxe de la méthode de création d'une couche pourrait donc se limiter à :

```
Couche c_e(largeur, hauteur, &champ);     //création de la couche d'entrée
```

où *champ* est un modèle de champ récepteur (de type *ChR* dans cet exemple) créé auparavant. L'avantage de cette méthode est d'éviter d'avoir à passer les paramètres du champ récepteur, ce qui permet d'envisager d'étendre ce programme à plusieurs champs récepteurs par neurone.

II.3) Fonctions de connexion

Les méthodes de connexion permettent de constituer l'architecture du réseau de neurones proprement dite, une fois les allocations mémoire effectuées.

La fonction de connexion entre 2 couches pourrait avoir par exemple le protoype suivant :

```
void Couche::connect_src(Couche *c_src)
```

où `c_src` est la couche-source, c'est à dire celle à laquelle la couche dans laquelle on se trouve doit être connectée.

II.4) Fonctions d'apprentissage

Par exemple, la fonction permettant d'appliquer l'apprentissage supervisé aux neurones d'une couche pourrait ressembler à :

```
void Couche::apprentissage_supervise(float alpha, float *sd)
```

où `sd` est un tableau des sorties désirées de la couche.

II.5) Fonctions d'activation

L'utilisation d'un réseau de neurone consiste à activer les couches les unes après les autres dans le sens de propagation des signaux d'entrée, une fois l'apprentissage terminé.

Par exemple, l'appel de la fonction d'activation pourrait s'écrire :

```
reseau.activation();
```

Elle aurait pour effet de calculer les sorties des différentes couches du réseau les unes après les autres, jusqu'à la couche de sortie.

II.6) Applications

On se propose d'utiliser ces outils de "programmation neuronale" avec 2 applications :

- Filtrage d'image,
- Mémoire associative.

L'application au filtrage sort un peu du cadre des réseaux de neurones, mais permet de montrer leur généralité. L'application à la mémoire associative est l'une des plus simple, et permet de mettre en évidence certaine des propriétés des réseaux de neurones.

On pourra chercher à utiliser le plus possible de variables privées, mais l'usage de variables publiques est autorisé si cela simplifie la programmation.

Le **travail demandé** consiste à programmer l'une de ces 2 applications.

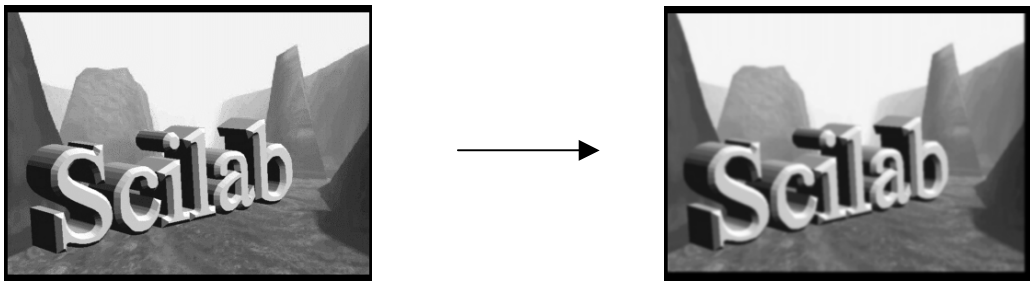
a) Application au filtrage d'images

Le filtrage d'images est basé sur l'opération de convolution, qui est justement celle réalisée par un neurone. Il consiste convoluer une grande image avec un filtre, qui n'est rien d'autre qu'une autre petite image : l'image à filtrer est balayée par le filtre, et à chaque position le produit de convolution du filtre et de la partie de l'image superposée est calculé. Le résultat de ce produit pour une position donnée constitue un pixel (élément d'image) de l'image filtrée).

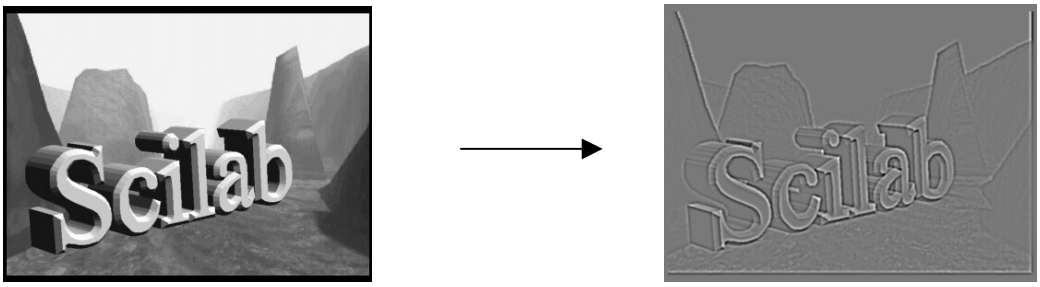
Une couche de neurones possédant des champs récepteurs locaux à poids partagés (contenant tous le même filtre) sur une image (=la couche d'entrée du réseau) peut donc réaliser cette opération.

Pour cette application, la partie du code permettant la gestion des filtres et des images est donnée en annexe. Ce code permet notamment de calculer :

- un filtre gaussien permettant de "flouer" une image ;
- un filtre en forme de "chapeau mexicain", calculé par une différence de 2 gaussiennes, permettant d'extraire les contrastes des images.



Convolution avec filtre gaussien



Convolution avec DOG (Difference Of Gaussians)

Dans le cas du 2e filtrage, on voit bien qu'il n'y a qu'aux endroits des contrastes (=variations des intensités lumineuses) de l'image à traiter que l'image résultante est non nulle. Il faut remarquer que, du fait du filtre utilisé, cette image comporte des valeurs négatives, représentées par les teintes claires, des valeurs positives représentées par les teintes foncées et que les grandes zones grises correspondent à des valeurs proches de zéro.

Dans le cas de l'application des réseaux de neurone au filtrage d'images, on calcule les poids des neurones en fonction des paramètres du filtre, il n'y a pas d'apprentissage. Ces paramètres peuvent être calculés de la manière décrite ci-dessous.

En général, la taille du filtre en nombre de coefficients par côté est impair. Prenons un filtre carré de taille (2p+1)(2p+1), le filtre comprend $(2p+1)^2$ coefficients que nous notons w_{ij} (i et j variant de 0 à 2p).

Soit γ (i,j,p,σ) la fonction de gauss :

$$\gamma(i,j,p,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(i-p)^2-(j-p)^2}{2\sigma^2}}$$

Pour un filtre gaussien de "flouage", on a :

$$w_{ij} = \gamma(i,j,p,\sigma)$$

Pour un chapeau mexicain, différence entre 2 gaussiennes de variances différentes, on a :

$$w_{ij} = \frac{\gamma(i,j,p,\sigma)}{S^+} - \frac{\gamma(i,j,p,1.5\sigma)}{S^-}$$

avec

$$S^+ = \sum_{i=0}^{2p}\sum_{j=0}^{2p}\gamma(i,j,p,\sigma) \qquad \text{et} \qquad S^- = \sum_{i=0}^{2p}\sum_{j=0}^{2p}\gamma(i,j,p,1.5\sigma)$$

Soit g_{kl} le niveau de gris du point (k,l) de l'image d'entrée ($0 \le g_{kl} \le g_{max}$), et w_{ij} les coefficients du filtre (2p+1)(2p+1), pour obtenir le niveau de gris G_{kl} d'un point de l'image filtrée, on calcule :

$$G_{kl} = \sum_{i=0}^{2p}\sum_{j=0}^{2P} w_{ij} g_{k+i-p,l+j-p}$$

Cette valeur doit ensuite être ramenée dans l'intervalle [0, g_{max}].

Il faut de plus traiter le problème des bords de l'image, car pour les points situés sur ou près de ces bords, on irait chercher des points de l'image d'entrée qui n'existent pas.

Le **travail à réaliser** consiste à créer l'architecture de réseau de neurones adéquate pour appliquer l'un des filtrages possibles dans le code déjà écrit, à une image de type PGM

(Portable Grayscale Map : monochrome). Ce format est les plus simples : les fichiers commencent par une entête, comportant les informations suivantes sur des lignes séparées :

- la chaîne de caractère "P5"
- une ou plusieurs ligne(s) de commentaires commençant par "#"
- le nom du fichier
- la largeur de l'image en nombre de pixels
- la hauteur de l'image
- le nombre de niveaux de gris
- la zone des pixels

L'entête de l'image est formatée, c'est à dire écrite en clair : on peut la lire et l'écrire avec les opérateurs >> et <<. La zone de donnée est non formatée, il faut la lire et l'écrire avec les instructions *read* et *write*.

Finalement, le programme principal pourrait ressembler à :

```
void
main()
{
        Image8 im_e("scilab.pgm");              //chargement de l'image destinée à être traitée par le réseau

        Filtre f(7, 7);                         //Filtre 7x7 (par exemple)

        int choix_filtre=1;                     //ou 2
        if(choix_filtre==1)
          f.calcul_d2gauss_DOG(0.7);            //calcul du filtre "chapeau mexicain" (pour extraction de contrastes)
        else if(choix_filtre==2)
          f.gauss(0.7);                         //calcul du filtre pour filtrage gaussien

        ChR chr1(1, 1);                         //inutilisé mais nécesssaire à la couche d'entrée
        ChR chr2(7, 7);                         //modèle de champ récepteur pour la construction du réseau
        chr2.initial_poids_synapt(f.don);       //copie des valeurs du filtre dans le champ récepteur
        chr2.affich_liens();                    //affichage des valeurs des poids à l'écran

        Couche c_e(im_e.l, im_e.h, &chr1, 1, 1);        //création de la couche d'entrée
        Couche c_s(im_e.l, im_e.h, &chr2, 1, 1);        //création de la couche de sortie
        Image8 im_s(im_e.l, im_e.h);                    //image pour visualiser la sortie de la couche de sortie

        c_s.connect_src(&c_e);                  //connexion de la couche d'entrée à la couche de sortie
        c_e.copie_image_sortie(&im_e);          //copie de l'image à traiter en sortie de la couche d'entrée
        c_s.copie_chr(&chr2);                   //copie du modèle de champ récepteur dans la couche de sortie
        c_s.activ();                            //activation du neurone : calcul de la somme pondérée de ses entrées
        c_s.calcul_sortie(num_fct=2);           //mise à jour des sorties : passage de la valeur d'activation
                                                //dans la fonction de transfert du neurone
        c_s.recadr_sorties(0, 255);             //recadrage des pixels de l'image entre 0 et 255 (niveaux de gris)
        c_s.copie_sortie_image(&im_s);          //copie de la sortie de la couche de sortie dans une image

        im_s.sauve_pgm("tmp2.pgm");             //sauvegarde de l'image de la sortie du réseau
}
```

b) Application à la mémoire associative

Une mémoire associative permet de faire progressivement ressortir de la mémoire une forme préalablement mémorisée, à partir d'une version partielle ou bruitée de cette forme.

On peut réaliser une mémoire associative à l'aide d'un réseau appelé réseau de Hopfield. Ce modèle ne permet pas de réaliser des applications pratiques de mémorisation, par exemple en reconnaissance de formes, mais l'étude de ses propriétés est intéressante. Parmi ses limitations, il y a notamment :

- le fait que le nombre de formes pouvant être mémorisées est limité à 15% du nombre de neurones ;

- l'augmentation très rapide du nombre de connexions dans le réseau avec le nombre de neurones : pour N neurones, il y a en effet $(N-1)^2$ connexions dans le réseau ;
- la mémorisation est parfaite pour des formes complètement décorrélées ; on dit également "orthogonales" : soient $X = (x_1, x_2, ..., x_n)$ et $Y = (y_1, y_2, ..., y_n)$ 2 vecteurs de dimension n représentant 2 formes différentes, ces 2 formes sont orthogonales si :

$$\sum_{i=1}^{n} x_i . y_i = 0$$

Par exemple, les 2 formes suivantes (lettres "A" et "B") ne sont pas orthogonales car le produit 2 à 2 de leurs éléments n'est pas nul.

0 0 1 0 0	1 1 1 1 0
0 1 0 1 0	1 0 0 0 1
1 0 0 0 1	1 0 0 0 1
1 1 1 1 1	1 1 1 1 0
1 0 0 0 1	1 0 0 0 1
1 0 0 0 1	1 0 0 0 1
1 0 0 0 1	1 1 1 1 0

Les 2 phases d'utilisation du réseau de Hopfield sont :

- Une phase d'apprentissage, consistant à présenter toutes les formes à mémoriser en entrée du réseau (éventuellement plusieurs fois), successivement, en appliquant la règle d'apprentissage à chacune ;
- Une phase d'utilisation, consistant à présenter une forme, identique à celles mémorisées ou légèrement différentes, en entrée du réseau, et à calculer les sorties de ce dernier de manière itérative, jusqu'à leur stabilisation (il est démontré que cette stabilisation se produit toujours si la mise à jour des sorties est asynchrone et aléatoire).

Le **travail à réaliser** est le suivant : écrire un constructeur de réseau complétement interconnecté pour créer un réseau de Hopfield à 2 dimensions d'une centaine de neurones maximum. Lui appliquer la fonction d'apprentissage de Hebb pour mémoriser quelques formes composées d'éléments binaires. Puis tester cette mémorisation en utilisant d'autres exemples des mêmes formes mémorisées, légèrement différents, déformés ou bruités.

Pour vous aider, vous trouverez un exemple d'implémentation de la méthode en C à cette adresse :

http://lava.cs.virginia.edu/shirley_benchmark/HopfieldModel.c