CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103

NANCY – METZ

PROJET

Année 2009 – 2010, deuxième semestre

Coefficient : 1/3

Travail à réaliser au plus tard avant l'examen de juin 2010, et à remettre en version électronique :

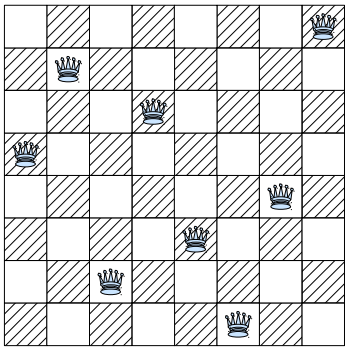
- au surveillant au début de l'examen final (avant la distribution du sujet),
- ou à envoyer **avant** l'examen à E. Desvigne : emmanuel@desvigne.org

Programmation d'un réseau neuromimétique

1. Présentation du concept

1.1 Problématique

Intuitivement, les problèmes informatiques sont souvent résolus à l'aide d'algorithmes séquentiels, qui se déroulent instruction après instruction. Les disciplines informatiques tel que le « *génie logiciel* » ont démontré mathématiquement que ce modèle a ses limites, même pour des problèmes qui peuvent sembler simples. Exemple : supposons que vous disposiez d'un échiquier et de huit reines, et qu'on vous demande de disposer ces huit pièces de façon à ce qu'aucune reine ne prenne une des sept autres. Classiquement, ce genre de problème se résout en étudiant toutes les possibilités de disposition des huit reines sur l'échiquier, ce qui demandera... plusieurs siècles de calculs aux ordinateurs les plus puissants.



Pour réduire le temps de calcul, l'idée est alors venue assez rapidement d'étudier toutes ces possibilités en parallèle sur plusieurs machines (chacune d'entre elles ne travaillant que sur une partie de l'arbre des possibles). Seulement, traduire des algorithmes séquentiels en algorithmes parallèles, ou inventer un algorithme intrinsèquement parallèle et efficace n'est pas si simple.

Le travail se complique si la formulation du problème est floue, ou si elle évolue dans le temps. Par exemple, la prévision de l'évolution du cours de la bourse, ou la reconnaissance d'une écriture manuscrite (qui varie d'un homme à l'autre, et qui varie aussi dans le temps pour un homme donné) ne sont pas des problèmes pour lesquels l'énoncée formel est simple à exprimer. L'idéal, pour résoudre ce genre de problème, serait de disposer d'une machine capable d'apprendre. Le problème devient : comment apprendre à apprendre à une machine.

Une des pistes pour résoudre ce problème : imiter le fonctionnement d'une « *machine* » réalisant ce travail à merveille, le cerveau. Dès la fin des années 1950, des neurobiologistes (Warren McCulloch et Walter Pitts pour les plus célèbres) ont commencé à décrire le fonctionnement des cellules qui semblaient constituer la partie efficace du cerveau : les **neurones**.

1.2 Fonctionnement des neurones

Schématiquement, un neurone est une cellule constituée :

- d'un corps (le **soma**),
- des sortes de tubes par lesquels entrent les signaux en provenance des autres neurones (les **dendrites**),
- et d'un tube par lequel sort l'influx nerveux (l'**axone**). C'est cet axone (souvent très long par rapport aux autres constituants) qu'on reconnaît comme étant la « fibre nerveuse ». Ce dernier ce termine en arborescence (l'**arborisation terminale**) dont chacune de ses terminaisons se « connecte » à une dendrite d'un autre neurone. Le « bourgeon » qui constitue cette liaison (la **synapse**) forme une connexion appelée **liaison synaptique**. Toutes les liaisons synaptiques n'ont pas la même efficacité (autrement dit, l'information en provenance d'un neurone sera plus ou moins bien transmise à un autre neurone).

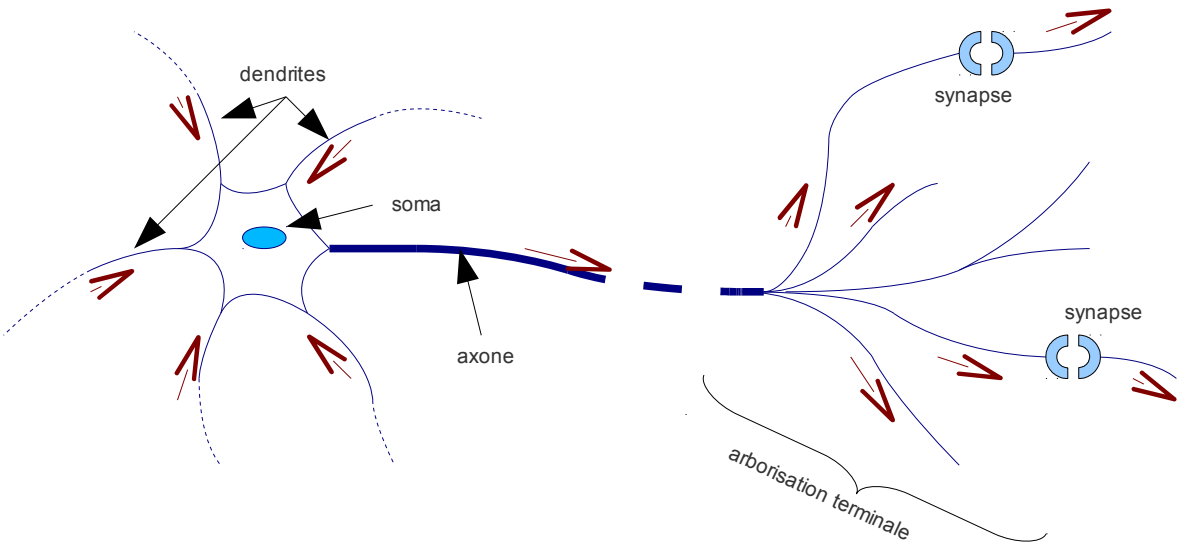


schéma d'un neurone

À noter que le soma fonctionne un peu à la façon du « *tout ou rien* ». Où bien la somme des influx nerveux se situe en dessous d'un certain seuil, et alors, il ne se passe rien (aucun influx nerveux n'est émis sur l'axone). Ou bien la somme des influx nerveux dépasse ce seuil, et l'influx de sortie envoyé sur l'axone est maximal.

1.3 Le neurone formel

Schématiquement, nous pouvons simuler un neurone selon le diagramme suivant :

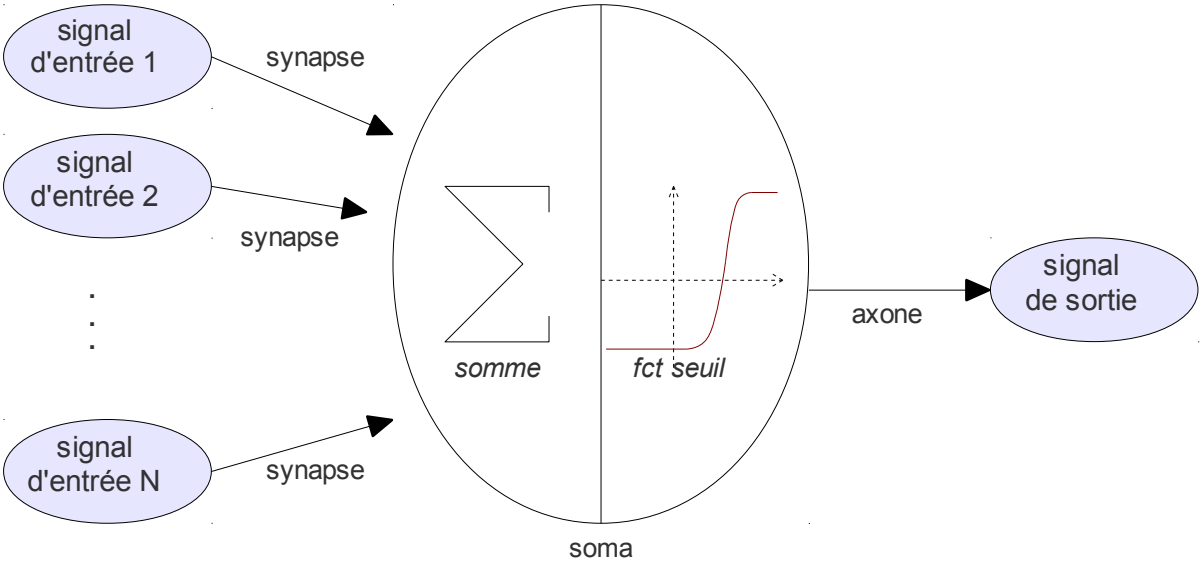


schéma d'un neurone formel

Nous avons vu que dans un neurone, les liaisons synaptiques sont plus ou moins efficaces. Pour simuler cette propriété, nous allons affecter à chaque synapse une pondération. Ou bien la liaison synaptique sera de mauvaise qualité, et cette pondération sera proche de zéro. Ou bien la synapse sera efficace, et cette pondération sera proche de 1.

Voici le principe de fonctionnement d'un neurone qui aurait N « entrées » (autrement dit, N dendrites) tel que nous allons le simuler :

- le signal d'entrée (qui est émis par un autre neurone) est un nombre à virgule flottante (de type « double » en C) compris entre 0 et 1 ;

- pour chacune des N entrées, le neurone calcule le signal qu'il va recevoir, en multipliant cette valeur d'entrée par le poids qui simule l'efficacité de la synapse (ce poids est lui aussi un nombre à virgule flottante de type « double » en C, compris entre 0 et 1) ;
- puis, il fait la somme de ces signaux qui arrivent par les N dendrites ;
- cette somme est divisée par N pour obtenir un nombre entre 0 et 1 ;
- ou bien le nombre obtenu est en dessous d'un certain seuil, et alors, la valeur de sortie sera proche de 0, ou bien la valeur obtenue est supérieure à ce seuil, et la valeur de sortie sera tout de suite proche de 1. Pour simuler ce mécanisme, nous utiliserons une fonction sigmoïde, dont la définition vous est donnée dans le code ci-dessous ;
- le résultat de cette fonction sera la valeur de sortie du neurone. À noter que cette valeur de sortie correspondra à une valeur d'entrée d'un autre neurone, connecté à celui dont nous venons de simuler le fonctionnement.

Voici le code (en C) qui simule le fonctionnement d'un tel neurone formel à huit entrées :

```
/* Permet de définir les fonctions mathématiques sur       */
/* les nombres à virgule (indispensable pour pouvoir       */
/* utiliser la fonction exponentiel exp() :                */
#include <math.h>

/* Définition de la fonction sigmoïde :                    */
double sigmoide(double x)
{
    return(1.0 / (exp((0.65 - x) * 25.0) + 1.0));
}

/* Définition du nombre d'entrées N                        */
#define N   8

/* Création du tableau "e" des N entrées, et d'un tableau  */
/* "p" des N poids qui simulent l'efficacité des N         */
/* synapses. Rappel : en C, les tableaux à N éléments      */
/* sont numérotés de 0 à N-1 (et non pas de 1 à N).        */
double e[N-1];
double p[N-1];

/* Initialisation des N entrées et des N poids des N       */
/* synapses. Cette partie de la simulation n'est pas à     */
/* traiter par le candidat. Il pourra mettre ce qu'il      */
/* veut comme valeur, par exemple en faisant appel à une   */
/* fonction qui retourne un nombre aléatoire compris       */
/* entre 0.0 et 1.0, ou en affectant des valeurs en dur,   */
/* comme par exemple : e[0]=0.25; e[1]=0.01; e[2]=0.98;    */
/* etc. ... p[0]=0.50; p[1]=0.10; p[2]=0.45; etc.          */
Votre code d'initialisation ici;

/* Code qui simule le fonctionnement d'un neurone formel : */
double calcule_sortie()
{
    int i;
    double somme_ponderee = 0.0;
    for (i = 0 ; i < N ; i++)
        somme_ponderee += e[i] * p[i];
    return(sigmoide(somme_ponderee / N));
}
```

1.4 Les réseaux de neurones formels (réseaux neuromimétiques)

L'utilisation d'un seul neurone formel ne présente pas un grand intérêt. L'idée est de simuler un grand nombre de neurones, massivement connectés les uns avec les autres. Tout l'art consiste à choisir :

- comment les neurones sont connectés les uns avec les autres ;
- quelle fonction seuil utiliser. Nous avons choisi dans notre exemple une fonction sigmoïde dont les coefficients ont été définis arbitrairement. De plus, ces derniers restent fixes dans le temps, ce qui n'est pas toujours le cas dans les diverses simulations.
- comment évoluent dans le temps les poids des synapses ;
- quelle est la stratégie d'apprentissage. Par exemple, certains types de réseau passent par une phase d'apprentissage durant laquelle les poids des synapses sont corrigés pour minimiser les erreurs évaluées sur des jeux d'essais, avant de passer à la phase d'utilisation durant laquelle les poids n'évoluent plus. Dans d'autres réseaux, les poids des synapses évoluent durant toute la vie du neurone.

Le premier réseau neuromimétique programmé dans les années 1960 était le « perceptron ». Les entrées et les poids étaient de simples bits. Les règles du perceptron lui permettaient d'apprendre une combinaison de portes logiques (non, et, ou naturel, etc.). Ce réseau a connu une certaine popularité, avant qu'il ne soit démontré que les règles de convergence ne permettaient pas à un neurone d'apprendre le « ou exclusif » (sortie à 1 si et seulement si une et une seule entrée à 1, l'autre étant à 0).

Cette limitation a été levée par la suite en changeant les règles d'apprentissage, la géométrie du réseau, etc. Voici quelques exemples de réseaux célèbres :

- les « réseaux multicouches à rétropropagation d'erreur »,
- les « réseaux de Hopfield »,
- les « réseaux de Kohonen »,
- la « machine de Boltzmann »,
- etc.

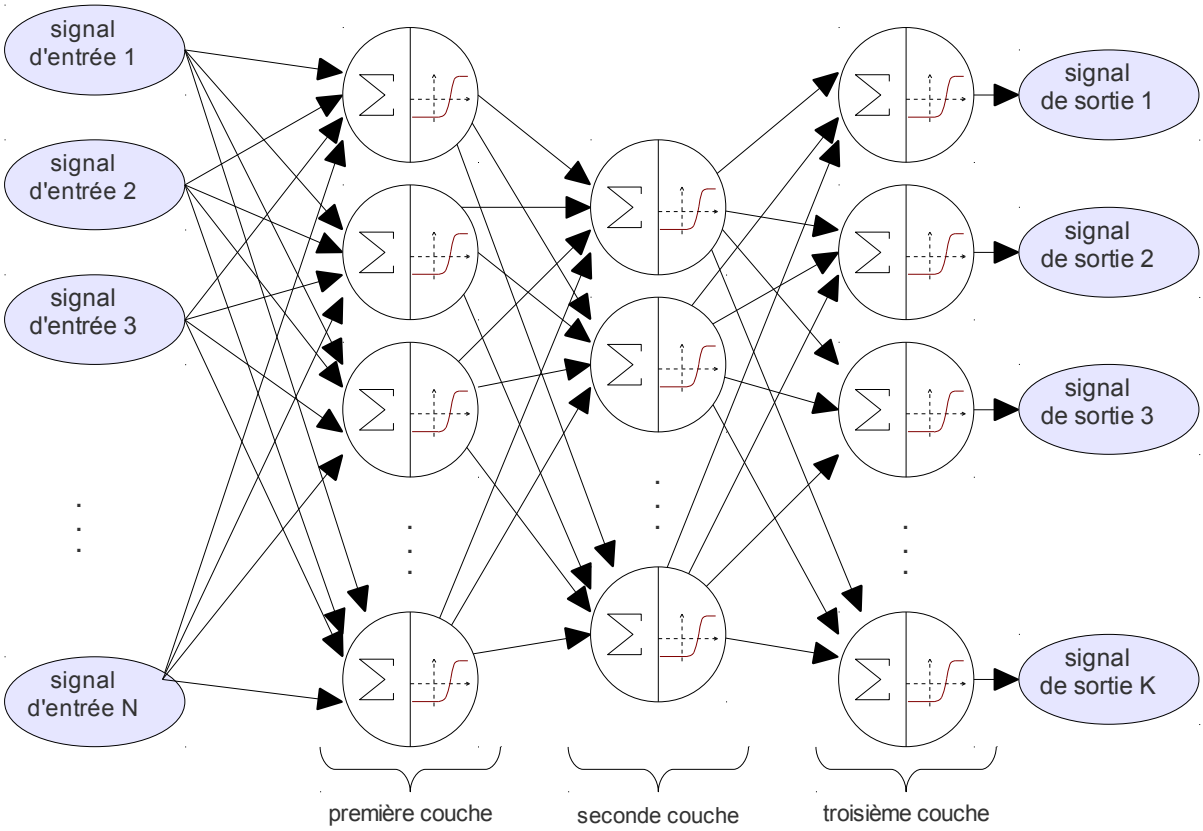
2. Sujet : programmation d'un réseau multicouches

2.1 Le principe d'un réseau multicouches

Les auditeurs auront à programmer un réseau de neurones multicouches. En voici le principe :

- une première couche est constituée de neurones qui sont tous connectés à des entrées (exemple : cellules photosensibles d'une caméra numérique, fournissant pour chaque pixel un niveau de gris simulé par une valeur comprise entre 0 – pour le noir – et 1 – pour le blanc –). Ces neurones sont tous isolés les un des autres (aucune sortie d'un de ces neurones n'est l'entrée d'un autre, et aucun neurone a pour entrée sa propre sortie) ;
- Une seconde couche utilise comme entrée les sorties de la couche précédente, et fonctionne suivant les mêmes règles ;
- une troisième couche reprend ce même principe. La sortie du réseau de neurones est l'ensemble des sorties de cette troisième couche.

Voici le schéma d'un tel réseau :



2.2 Formalisme du sujet

Les candidats doivent réaliser la programmation sous Linux d'un réseau multicouches composé de trois couches, ayant :

- huit entrées,
- huit neurones formels dans la première couche,
- quatre neurones formels dans la seconde couche,
- et six neurones formels dans la troisième couche (le réseau a ainsi six sorties).

La phase d'apprentissage n'est pas à traiter. Le candidat pourra choisir arbitrairement les poids des différentes synapses qui constituent le réseau. Ces poids (compris entre 0 et 1) seront fixes, et ne changeront pas dans le temps. La fonction seuil de tous les neurones sera la fonction sigmoïde avec les mêmes coefficients que ceux définis dans le paragraphe 1.3.

Important : le fonctionnement du réseau devra être le plus « parallélisé » possible, en respectant les règles suivantes :

- tous les neurones d'une même couche doivent effectuer leurs calculs **en parallèle** ;
- par contre, lorsque tous les neurones d'une couche travaillent, les autres couches doivent attendre. En effet, ça n'aurait pas de sens que les neurones de la seconde couche fassent des calculs à partir des résultats du travail des neurones de la première couche, si tous les neurones de cette dernière n'ont pas fini. De même, les neurones de la troisième couche ne s'activeront que lorsque tous les neurones de la seconde couche auront fini leurs calculs.

Le candidat a une totale liberté pour simuler l'aspect « multiprogrammation » du réseau (utilisation de threads, de processus, voire même s'il le souhaite de machines différentes situées sur un réseau IP). De même, la communication des résultats entre les neurones pourra être faite avec les outils de son choix : MSQ, pipes, shared memory, etc.

2.3 Travail attendu, barème

Le candidat s'attachera à justifier tous les choix qu'il aura fait (pourquoi avoir réalisé le traitement du parallélisme suivant telle méthode plutôt que telle autre, pourquoi avoir choisi tel mode de communication entre neurones plutôt que tel autre, etc.).

Cet argumentaire, ainsi que la description du réseau final comptera pour la moitié de la note.

Le candidat fournira aussi les algorithmes et le code de son réseau (en pseudo code ou en C). Ce travail constituera la seconde moitié de la note.

Bonus : si le candidat fournit du code en C opérationnel en complément ou à la place des algorithmes, un « bonus » lui sera attribué. Ce bonus pourra rattraper une éventuelle moyenne tangente à 10 après l'examen.

2.4 Modalités de remise du projet

Le projet devra impérativement être remis sous forme électronique (sur CD-Rom ou par e-mail) **AVANT** l'examen final du NSY103.

Un CD-ROM pourra être remis à la personne qui surveille l'examen **avant** la distribution des sujets. Le candidat signera alors une feuille d'émargement. Sinon (solution préférée), l'ensemble du projet pourra être envoyé par email à l'adresse emmanuel@desvigne.org **avant** l'examen. Chaque réception d'email sera suivie d'accusé de réception de la part du correcteur, indiquant que le projet a bien été reçu.

Le rapport du projet (obligatoire) pourra être rédigé dans un document en texte brut (ASCII ou UTF8), ou en RTF, ou au format MS-Word doc 1997-2003, ou préférentiellement au format OpenOffice Oasis Open Document (format odt).

Les algorithmes pourront faire partie de ce document, ou être mis dans des fichiers texte séparés. Enfin, l'éventuel code source en C sera fourni dans des fichiers séparés ayant les extensions « .c », « .h », etc.

L'ensemble de tous ces documents pourra être intégré dans un seul fichier (archive 7Z, ZIP, RAR, .tar.gz, ou .tar.bz2).