SN : Un simulateur pour réseaux connexionnistes.

Léon-Yves BOTTOU*, Yann LE CUN **

* Laboratoire d'Intelligence Artificielle de Paris 5, Ecole des Hautes Etudes en Informatique, Université de Paris 5, 45 Rue des Saints Pères, PARIS-FRANCE.

** Department of Computer Science, University of Toronto, King's College Road, TORONTO- ONTARIO, M5S-1A4- CANADA

1- Introduction.

L'écriture des parties centrales d'un programme de simulation de réseaux connexionnistes est souvent assez simple: les principaux algorithmes tiennent souvent en quelques lignes. Cependant, pour utiliser avec efficacité un réseau, l'utilisateur doit pouvoir examiner à chaque instant le comportement de son réseau: il lui faut alors écrire une quantité impressionnante de code pour permettre cette interactivité.

En particulier, il est souvent utile de disposer de sorties graphiques, ou de pouvoir définir un protocole de tests sans avoir à réécrire des parties trop importantes des programmes. Par ailleurs, la description même des réseaux devient de plus en plus complexe: en effet, la plupart des résultats intéressants [7] utilisent des structures complexes de connexions, intégrant la connaissance du domaine traité. Par exemple, on peut souhaiter imposer des connexions locales, des structures hiérarchisées en couches ou contraindre certaines connexions à partager un même coefficient synaptique.

Il existe aujourd'hui beaucoup d'outils de simulation de réseaux: la tendance consiste à développer des langages dédiés pour la simulation ou pour la description des réseaux[8,3].

Pour nos besoins propres, nous avons écrit, et utilisons depuis un an, un simulateur 'SN' ('Simulateur de Neurones'). Dans sa version actuelle, ce simulateur permet de définir des architectures de réseaux très générales, il implémente l'algorithme de rétro-propagation du gradient, et quelques variantes: réseaux itératifs [6], rétro-propagation d'états désirés [4], rétro-propagation du second ordre [5]. Ce simulateur est écrit en langage C, et fonctionne sur un certain nombre de machines: stations de travail (SUN, APOLLO), machines UNIX, et machines vectorielles (CONVEX, CRAY).

Les sections 2 et 3 décrivent les choix principaux faits dans SN. Nous exposons dans les sections 4 à 7 les principales fonctions implémentées et les éléments de la bibliothèque. La section 8 présente un exemple d'utilisation de SN et la section 9 les développements en cours.

2- LISP pour décrire les réseaux, et en diriger la simulation.

SN est architecturé autour d'un langage de description et de simulation de réseaux. Il s'agit en l'occurrence d'un petit interpréteur Lisp. Pour faciliter la portabilité, cet interpréteur est lui- même écrit en C. De nombreuses extensions concernent les opérations coûteuses nécessaires au traitement connexionniste, qu'il n'était évidemment pas question d'écrire en Lisp.

Certains simulateurs sont développés autour d'un langage ad-hoc, créé spécifiquement pour la manipulation de réseaux. En particulier, on propose souvent des langages orientés objets, pour la facilité de manipulation hiérarchique des "objets" réseaux: neurones, connexions, couches, poids synaptiques...

Nous avons décidé pour notre part de porter notre choix sur un langage existant. Créer de toutes pièces un langage nouveau nous aurait demandé des délais plus importants, et nous aurait conduits à des activités de développement de langage éloignées du domaine connexionniste lui-même. Cette volonté de rapidité de développement orientait notre choix vers un langage facile à implémenter. A ce stade, nous pouvions hésiter entre Lisp et Forth. Quoique peu adapté au traitement numérique, Lisp l'a emporté, d'une part parce qu'il est universellement connu, d'autre part, parce qu'il permet d'établir plus facilement un lien entre systèmes symboliques classiques, et systèmes connexionnistes.

Comme cet interpréteur Lisp est écrit lui même en C, son ensemble de primitives peut être étendu facilement. Il suffit de faire une édition de liens avec un fichier objet C contenant des routines aux arguments bien définis.

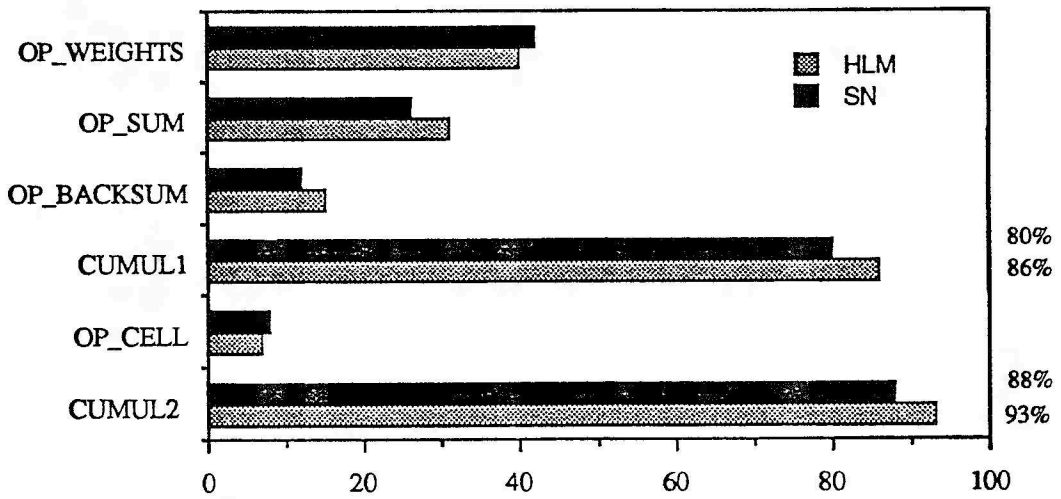
3- C pour calculer.

Le temps de calcul dans un simulateur connexionniste est déterminé:

- en tout premier lieu par les opérations concernant les connexions: calcul de la somme pondérée des entrées, calcul d'erreur et modification des poids,
- puis par celles concernant les cellules: calcul des états, passage de la fonction de transition des cellules (e.g. une tangente hyperbolique),
- enfin, de façon moins importante, (sauf pour les réseaux de très petite taille), par les opérations effectuées une fois lors de chaque itération: accès aux motifs d'apprentissage, mesure des performances, ...

Dans SN, toutes les fonctions de simulation concernant les connexions, ou les cellules sont écrites en C, alors que celles concernant les itérations effectuées sont écrites en Lisp: il en résulte que les contributions de ces différentes tâches au temps de calcul global sont dans SN assez peu pénalisées par la lenteur relative de Lisp (fig.1). Cette légère dégradation des performances est largement compensée, à notre avis, par les facilités offertes par Lisp dans la description de la dynamique du réseau.

Répartition des temps comparée HLM - SN (fig 1)



Comparaison de l'utilisation du temps dans SN et HLM (simulateur pour la rétro-propagation écrit en Pascal). Il s'agit ici d'une tâche de rétro-propagation sur un réseau à 3 couches de 100 cellules totalement connectées. Sur cet exemple, SN est très légèrement plus rapide que HLM.

OP_WEIGHTS: *ajustement des poids.*

OP_SUM: *calcul des sommes pondérées pour le calcul de l'état.*

OP_BACKSUM: *calcul des sommes pondérées pour le calcul du gradient.*

CUMUL1: *cumul des trois proportions précédentes. i.e. le temps passé à boucler sur les connexions.*

OP_CELL: *opérations relatives aux cellules. Il s'agit principalement du calcul de la fonction sigmoïde.*

CUMUL2: *temps éffectif passé au calculs connexionnistes. Le temps résiduel concerne les opérations éffectuées une fois lors de chaque itération. Seules ces opérations dans SN sont écrites en Lisp.*

4- Description de réseaux.

La description des réseaux dans SN est en fait une opération de construction. Décrire un réseau, c'est définir une fonction Lisp, paramétrée ou non, qui le construit.

Nous avons donc défini deux types de primitives:

- Les premières créent de nouvelles cellules, une à une ou par couches. Pour l'algorithme de rétro-propagation, par exemple, on dispose de la primitive **newneurons**:

```
(setq layer (newneurons 5))
```

met une liste de cinq nouvelles cellules dans la variable `layer`.

- Les secondes connectent les cellules créées avec les précédentes: connexion simple, connexion avec poids partagé, connexion complète de couche à couche.

La connexion simple est définie par la primitive **connect** :

```
(connect a1 a2)
```

qui établit une connexion de la cellule a1 vers la cellule a2.

Dans la version actuelle, toutes les autres fonctions de description de réseaux sont construites en Lisp à partir de ces deux classes de fonctions. Les réseaux simples peuvent donc être construits en appelant une seule fonction, elle-même définie en Lisp dans la bibliothèque standard. En particulier, la fonction **build-net:**

```
(build-net  '((input 400) (hidden 50) (output 76))
            '((input hidden)     (hidden output))      )
```

construit un réseau multi-couches contenant 400 cellules dans une couche d'entrée nommée `input`, 50 cellules cachées dans une couche nommée `hidden` et 76 cellules en sortie dans une couche nommée `output`. En outre, on indique que les connexions sont complètes de couche à couche.

5- Simulation.

On peut classer les primitives de simulation en quatre catégories principales.

1- Les premières servent à positionner ou examiner les états d'un groupe de cellules.

La fonction **get-pattern** :

```
(get-pattern array n layer)
```

permet en particulier de forcer les états d'une couche de cellules `layer` avec des valeurs extraites de la n^ieme^ ligne d'un tableau `array`. C'est elle qui est en général utilisée pour imposer au réseau des états en entrée.

La fonction **state** :

```
(state layer)
```

permet de récupérer les états de la couche `layer` dans une liste.

2 - D'autres ont pour but de mettre à jour les états, ou toute autre variable, d'un groupe de cellules, en fonction des états des cellules de leur voisinage.

Par exemple, les deux fonctions principales de l'algorithme de rétro-propagation du gradient, sont :

```
(update-state layer)          et
(update-gradient layer).
```

Elles calculent simplement les états et les gradients d'erreur associés à une couche de cellules en fonction des états des cellules amont, et du gradient des cellules aval.

3 - Les troisièmes appliquent un algorithme d'adaptation des coefficients synaptiques. Toujours dans le cas de la rétro-propagation, la primitive:

```
(update-weight)
```

modifie les poids, selon l'algorithme de rétro-propagation, en tenant compte des états et gradients des cellules amont et aval, de la valeur courante de la vitesse d'apprentissage, du momentum, et du decay.

4 - Les fonctions de la dernière classe permettent de contrôler les divers paramètres de fonctionnement du réseau : valeurs initiales des coefficients synaptiques, type et paramètres des fonctions de transfert associées aux cellules, vitesse d'apprentissage, etc... Chacun de ces paramètres peut être contrôlé indépendamment.

De plus, des fonctions Lisp de bibliothèques implémentent des concepts de plus haut niveau: par exemple, elles permettent d'effectuer des cycles d'apprentissage, des mesures de performances, des sorties graphiques. Elles sont également chargées de positionner tous les paramètres de façon cohérente. Voir ci-dessous dans l'exemple la fonction **epsi**.

6- Graphisme.

Nous l'avons dit plus haut, il peut être très utile, et toujours très efficace, de disposer de représentations graphiques du réseau, ou de la courbe d'erreur, ou encore des ordres de grandeur des variables du réseau (états d'activité, gradients, poids...).

Dans ce but, SN est équipé de fonction graphiques. Outre certaines des fonctions usuelles des systèmes graphiques disponibles sur stations de travail, comme:

```
(create-window posx posy sizex sizey name)
(draw-line x1 y1 x2 y2)
```

il y a également des fonctions spécialisées dans la représentation de valeurs analogiques. Ces fonctions permettent de dessiner la forme d'un réseau, avec des motifs de taille proportionelle à la grandeur de telle ou telle variable du réseau.
Par exemple:

```
(draw-list x y real_list ncol pix apart smax)
```

produit un dessin 'en carrés' des valeurs réelles contenues dans la liste `real_list`. Par assemblage de ces fonctions **draw-list**, on peut construire des fonctions de visualisation de réseaux entiers (fig.2). Pour compléter utilement ces fonctions, nous avons écrit en Lisp une bibliothèque simple de tracé de courbes. Elle permet de dessiner des axes, et de tracer des courbes en coordonnées réelles (fig.2).

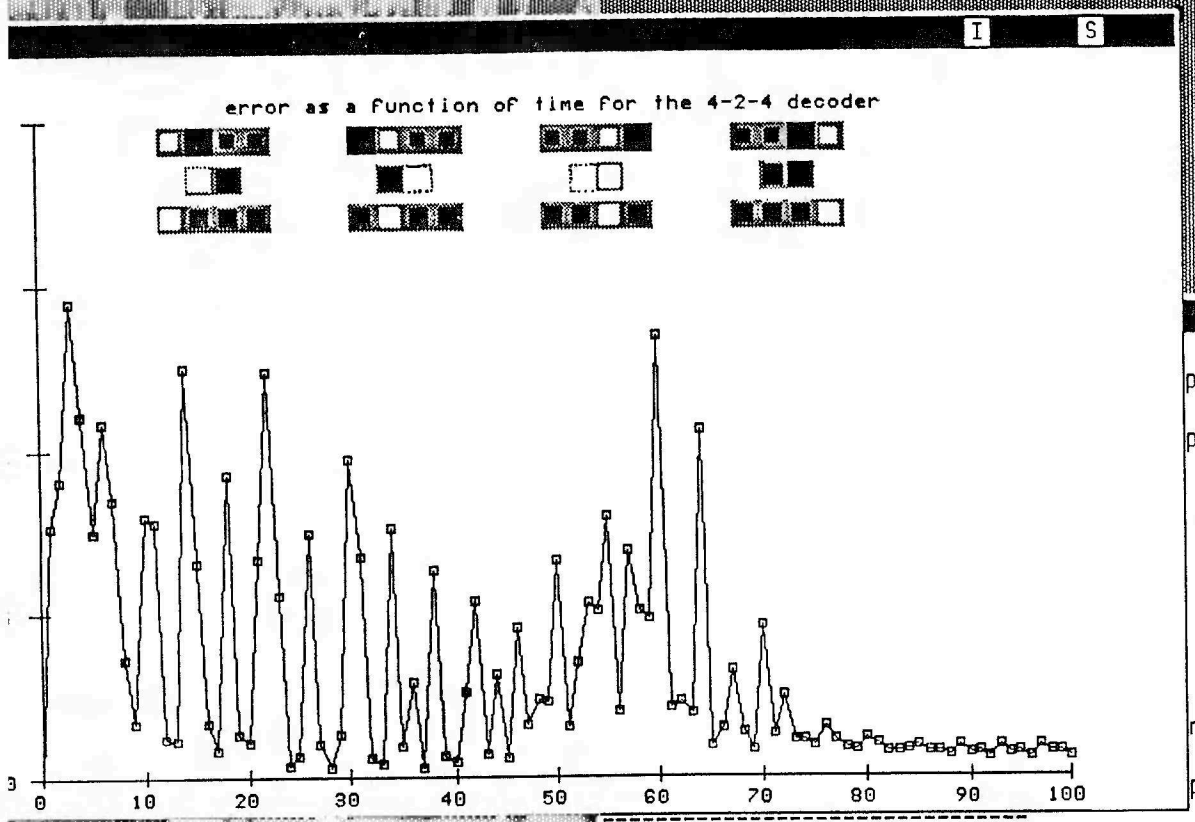
7- Bibliothèques.

L'existence d'un langage de description et de simulation de réseaux permet donc de construire des bibliothèques significatives. Ainsi, la bibliothèque standard associée à l'algorithme de rétro-propagation de gradient contient toutes les définitions de haut niveau permettant de diriger la simulation en termes de cycles d'apprentissage, étapes de test, et critère de mesure de performance:

```
(run 100 20)
```

permet, par exemple, de lancer 20 fois un cycle de 100 itérations d'apprentissage, suivi d'une étape de test.

```
(while (< (perf) 98)
   (learn 200) )
```




```
$
$
$
$ cd /sys/d
$
```

SHELL

performance=0

performance=25

2.380000 over sqrt(Fanin)

rformance=0

performance=0

```
--------------------------------
perf set is {0..3}, age=20, error=0.324564, performance=0
--------------------------------
perf set is {0..3}, age=30, error=0.192019, performance=0
--------------------------------
perf set is {0..3}, age=40, error=0.0954021, performance=75
--------------------------------
perf set is {0..3}, age=50, error=0.0809537, performance=75
--------------------------------
perf set is {0..3}, age=60, error=0.0692033, performance=100
= 1
```

demande d'effectuer des cycles de 200 itérations d'apprentissage, jusqu'à ce que la performance sur l'ensemble d'apprentissage dépasse 98%. On peut envisager ainsi des opérations aussi complexes qu'on le désire.

Ces bibliothèques contiennent donc une partie conséquente de nos connaissances sur les réseaux et la façon de fixer tel ou tel paramètre. Par exemple, toujours dans le cas de la rétro-propagation de gradient, la fonction:

```
(SMARTFORGET)
```

réinitialise les poids à des valeurs aléatoires, dans un intervalle tenant compte, et de la fonction de transfert utilisée, et des lois d'échelle usuelles concernant en particulier la connectivité des cellules.

8- Un exemple : "l'encodeur-décodeur 4.2.4."

Voici un petit exemple concret : "l'encodeur - décodeur 4.2.4". On dispose de 4 formes sur 4 bits +---, -+--, --+-, ---+, dont on souhaite élaborer un code sur 2 bits.

On utilise un réseau multicouche totalement connecté fonctionnant en auto-association. Les couches d'entrée et de sortie comportent donc quatre cellules. En revanche, la couche cachée n'en contient que deux. Pour pouvoir associer correctement chacune de ces 4 formes à elle même, le réseau va exhiber dans sa couche cachée un codage de ces formes sur 2 bits. On utilise pour cela un apprentissage par rétro-propagation du gradient.

Ce problème, d'un strict point de vue mathématique, n'est en fait qu'une Analyse en Composantes Principales[1,2]. Son principal intérêt ici est de montrer comment SN fonctionne et peut être utilisé.

Voici un exemple de programme SN, fonctionnant dans l'environnement standard, pour résoudre ce problème:

```
;alloue de la mémoire pour 40 cellules et 1000 connexions.
(ALLOC-NET 40 1000)

;crée un réseau à trois couches, totalement connectées
;de tailles 4, 2, et 4.
(build-net '( (input 4) (hidden 2) (output 4) )
           '( (input hidden) (hidden output) )   )

;crée un tableau contenant les 'patterns'
```

```
;les symboles pattern-array  et desired-array sont ceux  utilisés
;par la bibliothèque standard.
(dim pattern-array 4 4)
(for (i 0 3)
   (for (j 0 3)
         (pattern-array i j (read) ) )
 1 -1 -1 -1
-1  1 -1 -1
-1 -1  1 -1
-1 -1 -1  1

;on travaille en auto-association.
(setq desired-array pattern-array)

;préciser les 'patterns' à utiliser pour l'apprentissage.
(ensemble 0 3)

;définition de la fonction de transfert.
;i.e. f(x) = 1.68th(2x/3) + 0.05x
(nlf-tanh 0.666 0.05)

;initialisation aléatoire des poids
;i.e. dans [ -2.38/sqrt(FanIn), 2.38/sqrt(FanIn) ]
;cela correspond à cadrer la densité de probabilité des
;sommes pondérées entre les points de courbure maximum
;de la fonction de transfert.
(SMARTFORGET)

;initialisation de la vitesse d'apprentissage
;pour chaque connexion, epsilon vaut 1.0/FanIn.
(epsi 1.0)

;mode d'affichage pendant l'apprentisage : silencieux.
(set-disp-nil)

;critère de mesure : Un résultat est considéré comme
;correct si l'erreur quadratique est inférieure à 0.2
;Ce critère est utilisé par la fonction perf.
(set-class-lms 0.2)
```

```
;apprendre jusqu'à un taux satisfaisant.
(while (< (perf) 100)
    (learn 10) )
```

Cet apprentissage dure environ 10s sur une station Apollo. En enrichissant ce squelette de fonctionnalités graphiques, et en utilisant le mode d'affichage approprié, on obtient l'écran dont la copie est à la figure 2.

9- Développements prévus.

Plusieurs améliorations majeures sont actuellement en cours.

- Réécrire un interpréteur Lisp, plus rapide, et plus souple d'emploi. Celui-ci doit permettre un interfaçage plus facile des fonctions C, et contenir à peu près tout ce que l'on attend d'un interpréteur Lisp non dédié.

En outre, il sera capable de considérer les cellules, ou tout autre objet abstrait propre au domaine connexionniste comme des atomes Lisp à part entière.

- Modifier les fonctions de simulation, pour pouvoir prendre en compte plusieurs types de cellules ou de connexions, ayant des comportements différents. Nous espérons pouvoir simuler ainsi des réseaux hybrides, et des réseaux travaillant en coopération.

- Ecrire des modules en C, implémentant les algorithmes connexionnistes majeurs: non seulement la rétro-propagation de gradient, mais aussi les mémoires associatives, les modèles de cartes topologiques de Kohonen, de Grossberg, la machine de Boltzmann...

- Harmoniser toutes les fonctions qui ont été rajoutées au fur et à mesure de nos expériences avec SN.

Cette nouvelle version devrait être disponible fin 88 sur station de travail (Apollo, Sun).

10- Conclusion

Nous avons élaboré un système général de simulation de réseaux de neurones qui permet, dans sa version actuelle de:

- définir de façon très souple une architecture de réseau
- définir une dynamique sur ce réseau

- piloter des expériences d'apprentissage et de test
- visualiser en temps réel le fonctionnement du réseau

Cet outil nous a permis de réaliser des applications de tailles variables: petits exemples d'école, tests de validité de nouveaux algorithmes, et applications en "vraie grandeur" (reconnaissance d'image et de parole). Notre expérience nous a convaincus que l'usage d'un tel outil permettait de réduire notablement les temps de développement nécessaires.

Ce simulateur est utilisé dans nos Laboratoires, ainsi que dans quelques compagnies.

11 - Bibliographie

[1] - H.BOURLARD, Y.KAMP - Auto association by Multi Layer Perceptron and Singular Value Decomposition - M217 - Philips Research Lab, Brussels, nov 87.

[2] - P.GALLINARI,S.THIRIA,F.FOGELMAN SOULIE. MultiLayer Perceptrons and Data Analysis - ICNN 88 , IEEE Annual International Conférence on Neural Networks, July 88.

[3] - W.A.HANSON & al,, CONE COmputational Network Environment" -- in Procs of the IEEE First International Conference on Neural Networks, pp 531-538. June 87.

[4] - Y.LE CUN - Learning process in an assymmetric threshold network. In 'Disordered Systems and Biological Organization", NATO workshop Les-Houches 85, NATO-ASI Series in Systems and Computer Science, F20, E.Bienenstock, F.Fogelman Soulié, G.Weisbuch Edsd: Springer Verlag, 1986, 233-240.

[5] - D.B.PARKER - "Optimal Algorithms for Adaptive Networks: Second Order Back Propagation, Second Order Direct Propagation, and Second Order Hebbian Learning" - in Procs of the IEEE First International Conference on Neural Networks, II-593. June 87.

[6] - D.E.RUMELHART, G.E.HINTON,R.J.WILLIAMS - Learning internal représentation by error propagation. in 'Parallel Distrubuted Processing: explorations in the microstructures of cognition', MIT Press 86.

[7] - A.WAIBEL, T.HANAZAWA, G.HINTON, K.SHIKANO, K.LANG - Phonème recognition using Time-Delay Neural Networks. - TR 1 0006, ATR Interpreting Telephony Research Laboratories - October 87 .

[8] - D.ZIPSER, D.RABIN - The P3 simulation machine - Parallel Distributed Processing vol1 - MIT press 86