NSK, un noyau pour la simulation orientée objets de réseaux de neurones[1]

Cédric GEGOUT[•◇⋆], Bernard GIRAU[•] & Fabrice ROSSI[2◇⋆]

⋆ THOMSON-CSF/SDC,
DPR/R4, 7 rue des Mathurins, 92223 BAGNEUX FRANCE
Téléphone : (33 1) 40 84 29 05 Télécopie : (33 1) 40 84 29 50

◇ Ecole Normale Supérieure de Paris,
45 rue d'Ulm, 75005 PARIS FRANCE

• Laboratoire de l'Informatique du Parallélisme,
Ecole Normale Supérieure de Lyon,
46 avenue d'Italie, 69007 LYON FRANCE

e-mail : gegout@ens.ens.fr, bgirau@lip.ens-lyon.fr et rossi@ens.ens.fr

Résumé : Nous présentons dans cet article un noyau permettant la conception de simulateurs orientés objets de réseaux de neurones. Ce noyau est basé sur un modèle mathématique décrivant les réseaux de neurones non récurrents. Nous proposons une implémentation en C++ qui satisfait aux exigences suivantes : le noyau est extensible (l'implémentation d'un nouveau modèle neuronal est très simple), portable et efficace (il n'introduit pas de pénalité algorithmique pour les modèles classiques, comparé à une implémentation orientée objets spécifique à l'un quelconque de ces modèles). Les algorithmes d'apprentissage, et en particulier ceux utilisant le gradient, peuvent être mis en place pour des réseaux quelconques et sont de ce fait directement utilisables pour tout modèle particulier.

Mots-clés : simulateur, réseaux neuronaux non récurrents, langages orientés objet, rétro-propagation

Abstract : An object-oriented neural network simulator kernel is presented. It is based on a general mathematical model for arbitrary feedforward nets. We propose a C++ implementation of this model which satisfies the following requirements : expandability (allowing an easy implementation of a new neural model), portability and efficiency (the kernel does not increase significantly computation times for classic models, compared to a direct object-oriented implementation). Learning algorithms such as gradient-based ones can be written for arbitrary nets and are therefore directly available for every particular model.

Keywords : simulator, non-recurrent neural networks, object-oriented languages, back-propagation

Catégorie : outils & techniques (outils de simulation).

---

[1]Publié dans les actes des Journées internationales sur Les Réseaux Neuromimétiques et leurs Applications.
Disponible à l'URL http://apiacoa.org/publications/1994/neuronimes94.pdf

[2]Les coordonnées actuelles de Fabrice Rossi sont disponibles à l'URL http://apiacoa.org/.

1 Introduction

L'absence de théorèmes précis concernant les possibilités pratiques des réseaux de neurones entraîne souvent un recours massif à l'expérimentation. Pour faciliter ces expérimentations, de nombreux simulateurs neuronaux ont été proposés (comme par exemple [1, 4, 5, 6, 11, 12]). Ces simulateurs pèchent souvent par excès de généralité : ils implémentent de très nombreux modèles assez différents (Perceptrons Multicouches (MLP), modèle de Hopfield, modèle de Kohonen, etc. ...), au détriment de l'unification de la représentation. Les possibilités d'extension de ces simulateurs sont en général assez limitées par cette absence d'unification : la notion de rétro-propagation est ainsi étroitement liée à celle de MLP, ce qui veut dire que la mise en place d'un modèle comme celui des réseaux d'ondelettes (cf. [17]) impose de programmer à nouveau la rétro-propagation pour celui-ci. En fait, l'extensibilité est assurée au niveau des fonctions annexes (comme la gestion des simulations), mais pas au niveau des fondements mathématiques. D'autre part, cette extensibilité reste très coûteuse en terme de temps de développement.

Dans cette communication, nous reprenons en l'étendant un modèle des réseaux de neurones non récurrents proposé dans [2] :

> Un réseau de neurones est un graphe orienté non bouclé. Chaque sommet du graphe est un neurone qui possède des entrées, des sorties et des paramètres. Les connexions du graphe transportent les sorties de certains neurones vers les entrées de certains autres. Chaque neurone est capable de calculer ses sorties en fonctions de ses entrées et de ses paramètres.

Ce modèle est extrêmement général et permet entre autres de simuler des MLP, des réseaux d'ondelettes ou des réseaux à base de fonctions radiales (les Radial Basis Functions, cf. [15]). D'autre part, il garantit l'existence d'une rétro-propagation généralisée, qui permet de calculer la différentielle de la sortie du réseau par rapport aux paramètres de celui-ci. Cet algorithme généralisé est aussi efficace algorithmiquement que l'algorithme classique appliqué aux MLP. Ce modèle est parfaitement adapté pour l'approximation de fonction par réseaux de neurones et justifie donc pleinement l'intérêt qui lui est porté.

Nous proposons une implémentation en C++ de ce modèle général, le logiciel NSK (Neural Simulator Kernel). Le noyau de simulation obtenu est très facilement portable en raison du langage retenu. D'autre part, NSK est réellement extensible. En effet, l'implémentation du modèle est totalement générique et il suffit de l'instancier pour obtenir un modèle particulier. Cette instanciation a lieu en deux temps :

1. Programmation du calcul local effectué par un neurone (fonction semi-linéaire pour un MLP par exemple).

2. Mise en place de contraintes sur la structure du réseau (c'est à dire avec des couches uniformes et une connexion totale entre deux couches pour les MLP).

Bien entendu, l'algorithme de rétro-propagation est applicable directement à chaque instanciation du modèle général, et ceci sans aucun ajout de code.

Nous proposons d'autre part une modélisation mathématique simple de l'apprentissage supervisé qui nous permet encore une fois d'obtenir une implémentation générique en C++. La généralité de ce modèle permet de prendre en compte de nombreux algorithmes classiques comme les algorithmes basés sur l'utilisation du gradient (gradient conjugué de Polak-Ribière par exemple) ou les algorithmes génétiques. D'autre part l'implémentation en C++ permet d'avoir des algorithmes indépendants du réseau de neurones considéré. Les

algorithmes sont en fait exprimés au niveau du modèle général et sont donc immédiatement utilisables pour une instance particulière de celui-ci, ceci sans aucun ajout de code. Bien entendu, des algorithmes particuliers, utilisables seulement avec certains modèles peuvent être mis en place.

La fin de cet article est organisée comme suit : la section 2 présente les modèles mathématiques utilisés pour les réseaux et pour l'apprentissage ; la section 3 expose les techniques utilisées pour l'implémentation de NSK ; la section 4 présente quelques exemples d'applications et enfin la section 5 conclut en donnant quelques perspectives.

2 Modèle mathématique

2.1 Les réseaux de neurones

Cette sous-section présente une formulation mathématique des réseaux de neurones. La principale différence avec les modèles classiques réside dans la suppression des pondérations de connexions, les coefficients en question étant déplacés à l'intérieur du neurone par l'intermédiaire de la notion de paramètres.

Les neurones
Le neurone est le composant de base des réseaux de neurone. Mathématiquement, c'est une simple fonction à deux "entrées" :

Définition 2.1 *Soit* $\boldsymbol{I}$, $\boldsymbol{O}$ *et* $\boldsymbol{P}$ *trois espaces vectoriels sur* $\mathbb{R}$ *de dimensions finies. Un* **neurone** *est une fonction* $\boldsymbol{N}$ *de* $\boldsymbol{I} \times \boldsymbol{P}$ *dans* $\boldsymbol{O}$. $\boldsymbol{I}$ *est son espace d'entrée,* $\boldsymbol{P}$ *son espace de paramètres et* $\boldsymbol{O}$ *son espace de sortie. Le neurone* $\boldsymbol{N}$ *est dit différentiable si les fonctions partielles* $\boldsymbol{N_1(.,p)}$ *et* $\boldsymbol{N_2(x,.)}$ *le sont (on notera* $\boldsymbol{dN_1}$ *et* $\boldsymbol{dN_2}$ *les différentielles correspondantes).*

Les graphes ordonnés
Un graphe ordonné est simplement un graphe classique auquel on a ajouté un ordre sur les noeuds et un ordre sur les prédécesseurs :

Définition 2.2 *Un* **graphe ordonné** *est un triplé* $\boldsymbol{(\mathcal{N}, \mathcal{E}, (\leq_n)_{n\in\mathcal{N}})}$ *dont les composants vérifient les propriétés suivantes :*

- $\boldsymbol{\mathcal{N}}$ *est un ensemble fini de noeuds (ou sommets). Il est de plus totalement ordonné (c'est donc une suite finie de sommets) ;*
- $\boldsymbol{\mathcal{E} \subset \mathcal{N}^2}$ *est un ensemble d'arêtes ou de connexions.* $\boldsymbol{(i,j) \in \mathcal{E}}$*, traduit l'existence d'une arête de* $\boldsymbol{i}$ *vers* $\boldsymbol{j}$ *;*
- $\boldsymbol{(\leq_n)_{n\in\mathcal{N}}}$ *est un ensemble d'ordres.* $\boldsymbol{\leq_n}$ *est un ordre total sur l'ensemble des* **prédécesseurs** *de* $\boldsymbol{n}$*,* $\boldsymbol{Pred(n) = \{p \in \mathcal{N} \mid (p,n) \in \mathcal{E}\}}$.

Un tel graphe est **acyclique** *quand il ne comporte pas de cycle, c'est à dire de chemin (suite d'arêtes) d'un sommet vers lui-même.*

L'ensemble des **successeurs** *d'un noeud est défini par* $\boldsymbol{Succ(n) = \{p \in \mathcal{N} \mid (n,p) \in \mathcal{E}\}}$.

Les réseaux de neurones
Un réseau de neurone est simplement un graphe ordonné dont les sommets sont des neurones :

Définition 2.3 *Un* **réseau de neurones** *non récurrent est un graphe ordonné* $\boldsymbol{G = (\mathcal{N}, \mathcal{E}, \leq)}$ *dont les sommets sont des neurones et vérifiant :*

- $\boldsymbol{G}$ *est acyclique ;*
- $\boldsymbol{\forall N \in \mathcal{N},\ \sum_{P \in Pred(N)} \dim O^P = \dim I^N}$

Dans cette dernière formule $\boldsymbol{I^N}$ *désigne l'espace d'entrée du neurone* $\boldsymbol{N}$ *(et bien entendu,* $\boldsymbol{O^P}$ *est l'espace de sortie du neurone* $\boldsymbol{P}$*).*

La formule d'égalité permet d'identifier l'espace produit des sorties des prédécesseurs d'un noeud à l'espace d'entrée de ce noeud[1].

2.2 Calcul dans un réseau

Cette sous-section présente le modèle de calcul retenu pour un réseau de neurones et explique brièvement le principe de la rétro-propagation étendue.

Calcul de la sortie
A partir d'un réseau de neurones $\boldsymbol{G}$, on définit un nouveau neurone $\boldsymbol{GN}$. Pour ce faire, on introduit d'abord deux définitions importantes :

Définition 2.4 *L'ensemble des neurones d'entrée du réseau* $\boldsymbol{G = (\mathcal{N}, \mathcal{E}, \leq)}$*, noté* $\boldsymbol{In}$*, est défini par* $\boldsymbol{In = \{N \in \mathcal{N} \mid Pred(N) = \emptyset\}}$*. Comme* $\boldsymbol{\mathcal{N}}$ *est totalement ordonné, on peut définir une suite ordonnée des éléments de* $\boldsymbol{In}$*, notée* $\boldsymbol{(In^i)}$*.*

De la même façon, on définit l'ensemble des neurones de sortie du réseau, noté $\boldsymbol{Out}$ *par* $\boldsymbol{Out = \{N \in \mathcal{N} \mid Succ(N) = \emptyset\}}$*.*

On peut maintenant définir le neurone "implémenté" par un réseau.

Définition 2.5 *Soit un réseau de neurones* $\boldsymbol{G = (\mathcal{N}, \mathcal{E}, \leq)}$*. Comme* $\boldsymbol{\mathcal{N}}$ *est totalement ordonné et fini, on peut l'écrire sous la forme* $\boldsymbol{\mathcal{N} = \{N^1, \ldots, N^n\}}$*, avec* $\boldsymbol{N^i < N^{i+1}}$*. On définit alors les espaces suivants :*

- $\boldsymbol{I = \prod_i I^{In^i}}$ *est le produit des espaces d'entrées des neurones d'entrée ;*
- $\boldsymbol{P = \prod_{i=1}^{n} P^{N^i}}$ *est le produit des espaces de paramètres de tous les neurones ;*
- $\boldsymbol{O = \prod_i I^{Out^i}}$ *est le produit des espaces de sortie des neurones de sortie.*

$\boldsymbol{G}$ *définit un neurone* $\boldsymbol{GN}$ *dont les espaces d'entrées, de paramètres et de sorties sont respectivement* $\boldsymbol{I}$*,* $\boldsymbol{P}$ *et* $\boldsymbol{O}$*.*

Soit $\boldsymbol{i_1, \ldots, i_p}$ *les numéro d'ordre des éléments de* $\boldsymbol{In}$ *dans la liste des éléments de* $\boldsymbol{\mathcal{N}}$*. Soit de même* $\boldsymbol{o_1, \ldots, o_q}$ *les numéro d'ordre des éléments de* $\boldsymbol{Out}$*. Soit maintenant* $\boldsymbol{x = (x^{i_1}, \ldots, x^{i_p})}$ *un vecteur de* $\boldsymbol{I}$ *et* $\boldsymbol{p = (p^1, \ldots, p^n)}$ *un vecteur de* $\boldsymbol{P}$*. On définit* $\boldsymbol{R^k}$*, la sortie du neurone* $\boldsymbol{N^k}$ *par récurrence :*

- *si* $\boldsymbol{N^k \in In}$*, alors* $\boldsymbol{R^k = N^k(x^k, p^k)}$*.*

[1] En fait, les espaces considérés ne sont pas strictement égaux, mais identifiables par un isomorphisme canonique de concaténation. Ici, on se permettra toujours de confondre les deux espaces. Ainsi, $((x),(y)) \in \mathbb{R} \times \mathbb{R}$ sera identifié à $(x, y) \in \mathbb{R}^2$. Les ordres locaux aux prédécesseurs permettent de définir l'ordre dans lequel la concaténation locale est effectuée.

- *si* $\boldsymbol{N^k \notin In}$, *alors posons* $\boldsymbol{C^k = (R^{Pred(N^k)_1}, \ldots, R^{Pred(N^k)_r})}$. *D'après la condition de compatibilité des dimensions, on peut supposer que* $\boldsymbol{C^k \in I^{N^k}}$. *On pose alors* $\boldsymbol{R^k = N^k(C^k, p^k)}$.

On pose enfin $\boldsymbol{R = (R^{o_1}, \ldots, R^{o_q})}$. *Par définition, on a* $\boldsymbol{GN(x, p) = R}$.

Cette définition un peu formelle ne fait que traduire mathématiquement un processus assez simple. On fournit une entrée à chacun des neurones qui ne sont pas reliés à d'autres neurones. On fournit un paramètre à chaque neurone. Pour calculer la sortie d'un neurone, on "attend" que celles de ses prédécesseurs aient été calculées. On fabrique alors une entrée par concaténation des sorties puis on évalue la sortie du neurone considéré. La sortie du réseau est simplement la concaténation des sorties des neurones de sortie. Pour une démonstration de la possibilité du calcul par récurrence, voir [8].

On remarque que le modèle choisi est intrinsèquement modulaire puisqu'un réseau de neurones est lui-même un neurone.

Rétro-propagation

Notons tout d'abord qu'il existe une certaine confusion entre l'algorithme de rétro-propagation en lui-même et la méthode d'apprentissage la plus simple qui en découle. Nous considérons que l'algorithme de rétro-propagation est simplement la méthode très efficace de calcul du gradient de l'erreur commise par un MLP sur un exemple (la fonction d'erreur étant considérée comme une fonction des paramètres du réseau). La méthode d'apprentissage n'est rien d'autre qu'un gradient stochastique qui peut être utilisé dans de nombreuses autres applications (cf. [2]).

Dans le cas qui nous intéresse, on peut montrer que dans un réseau de neurones $\boldsymbol{G}$ dont tous les neurones sont différentiables, la fonction $\boldsymbol{GN}$ est aussi différentiable et que le calcul de ces deux différentielles partielles $\boldsymbol{dGN_1}$ et $\boldsymbol{dGN_2}$ peut se faire par un algorithme de rétro-propagation généralisée. Nous ne décrirons pas plus précisément la méthode en elle-même car elle nécessite l'introduction de très nombreuses notations et reste assez complexe (on se reportera à [2, 8]). Quelques points sont cependant à retenir :

1. La rétro-propagation généralisée est définie à partir d'un "signal rétro-propagé" qui est calculé par une récurrence arrière : on commence par les neurones de $\boldsymbol{Out}$ et on calcule le signal au niveau d'un neurone en fonction du signal au niveau de ses successeurs.

2. On peut calculer les différentielles par une propagation directe, c'est à dire par une simple application de la formule de la différentielle de fonction composée, mais ce mode de calcul est beaucoup plus coûteux.

3. Les matrices jacobiennes des neurones $\boldsymbol{dN_1^k}$ et $\boldsymbol{dN_2^k}$ n'interviennent dans la rétro-propagation que sous forme de membres de droite de produits matriciels.

4. Si on optimise les calculs matriciels, le coût algorithmique du calcul par la rétro-propagation généralisée est le même que celui du calcul classique dans le cas d'un MLP. Cette optimisation consiste à utiliser la connaissance qu'on peut avoir à priori sur la forme des matrices jacobiennes des neurones (voir la sous-section 3.3).

Partage d'entrées et de paramètres

Le modèle décrit possède une lacune assez gênante : une distinction est faite entre les neurones de $\boldsymbol{In}$ et les autres. En effet, les neurones de $\boldsymbol{In}$ reçoivent tous une entrée

distincte dans le processus de calcul. Au contraire, les neurones "internes" peuvent partager entre eux la sortie d'un autre neurone. Pour remédier à cet inconvénient, on introduit la notion de fonction de partage d'entrées :

Définition 2.6 *Soit* GN *un neurone défini par le réseau de neurones* G *et soit* I *son espace d'entrée. On appelle* **fonction de partage d'entrées** *une fonction* F *d'un espace vectoriel réel* NI *de dimension finie vers* I*. Le couple* (GN, F) *définit un nouveau neurone* GN_F *dont les espaces de paramètres et de sortie sont les mêmes que ceux de* GN *et dont l'espace d'entrée est* NI*. Pour* $x \in NI$ *et* $p \in P$*, on a* $GN_F(x, p) = GN(F(x), p)$*.*

Supposons par exemple que le réseau possède deux neurones d'entrées avec le même espace d'entrée I'. On définit alors F de $NI = I'$ dans $I = (I')^2$ par $F(x) = (x, x)$ ce qui permet de donner aux deux neurones la même entrée.

Si F et GN sont différentiables, GN_F l'est aussi et la rétro-propagation peut encore s'appliquer (pour plus de détails, voir [7]).

On peut de la même façon mettre en place un système de partage de paramètres qui permet d'avoir par exemple deux neurones qui utilisent le même vecteur de paramètres (cf. [7]).

2.3 Exemple des MLP

Deux approches sont possibles pour représenter les MLP dans le cadre de notre modèle : l'approche neurone par neurone ou l'approche couche par couche.

Approche neuronale
On considère que chaque neurone du MLP engendre un neurone du modèle général. La structure du graphe représentant le MLP est alors exactement la même que celle du MLP (c'est à dire une structure par couches de neurones totalement connectés aux neurones de la couche précédente). Le vecteur de paramètre est constitué des valeurs des connexions du neurone suivi de la valeur de son seuil. Si par exemple on a $p = (a_1, \ldots, a_n, s)$ et que la sortie de la couche précédente est $x = (x_1, \ldots, x_n)$, le neurone calcule la fonction $N(x, p) = f(\sum_{i=1}^{n} a_i x_i + s)$, où f désigne sa fonction de transfert. La sortie d'une couche est obtenue automatiquement par concaténation au niveau de chaque neurone de la couche suivante. Pour permettre aux neurones de la première couche de tous recevoir la même entrée, on utilise la technique de partage d'entrées décrite dans la sous-section précédente. A part cette fonction, la principale différence entre ce modèle et le modèle classique des MLP réside dans la transformation des valeurs synaptiques liées aux connexions en paramètres liés au neurone. On peut montrer que la complexité algorithmique du calcul de la sortie du réseau ou de sa différentielle est la même pour notre modèle que pour le modèle classique des MLP (c'est à dire linéaire en le nombre de connexions et de seuils du réseau).

Approche par couche
On considère que chaque couche du MLP engendre un neurone du modèle général. La structure du graphe dans notre modèle est alors simplement linéaire avec une unique connexion entre deux neurones qui se suivent. Cette connexion transmet la sortie vectorielle de la couche considérée. Les paramètres d'un neurone sont en fait la concaténation de la matrice des connexions et du vecteur de seuils de la couche correspondante du MLP de départ. Comme dans le cas précédent, on peut montrer que la complexité des calculs utilisant ce modèle est la même que dans le cadre du modèle classique des MLP, à condition

d'optimiser les produits matriciels mis en jeu dans la rétro-propagation (car les matrices jacobiennes d'une couche sont creuses, avec seulement des blocs diagonaux non nuls).

2.4 L'apprentissage

L'apprentissage supervisé peut en général se ramener à une tentative d'approximation d'une fonction échantillonnée. Plus précisément, on se donne une fonction $\boldsymbol{F}$ définie sur $\boldsymbol{S}$ un ensemble fini de points de $\boldsymbol{I}$ et à valeur dans $\boldsymbol{O}$, respectivement les espaces d'entrées et de sorties d'un neurone $\boldsymbol{N}$. On se donne une norme définie sur l'ensemble des fonctions de $\boldsymbol{S}$ dans $\boldsymbol{O}$ et on suppose que cette norme peut être restreinte, c'est à dire obtenue en ne considérant comme ensemble de départ des fonctions un sous ensemble $\boldsymbol{S'}$ de $\boldsymbol{S}$ (si on considère par exemple une norme "discrète", on a $||\boldsymbol{F}(.) - \boldsymbol{N}(., \boldsymbol{p})|| = \sum_{\boldsymbol{x} \in \boldsymbol{S'}} ||\boldsymbol{F}(\boldsymbol{x}) - \boldsymbol{N}(\boldsymbol{x}, \boldsymbol{p})||$).

Un algorithme d'apprentissage propose trois "méthodes" :

- Initialisation : l'algorithme crée un *état interne* à partir d'un neurone de départ, d'un paramètre de départ pour celui-ci, d'une fonction, d'une norme et d'un sous ensemble de calcul ;
- Itération : l'algorithme passe d'une étape à une autre en modifiant un état interne ;
- Projection : l'algorithme renvoie un neurone et son paramètre, correspondant à un état interne.

L'état interne joue le rôle de mémoire de l'algorithme. La projection sélectionne la "meilleure" approximation (au sens de la norme de l'erreur) obtenue par l'algorithme. A chaque étape, l'algorithme peut modifier l'ensemble qui sert au calcul de l'erreur, ce qui permet de modéliser la notion de changement dynamique d'ensemble d'apprentissage (comme dans le cas du gradient stochastique).

On pourra se reporter à [8] pour une présentation plus formelle de ce modèle.

3 Description du simulateur

3.1 Organisation générale

NSK se présente sous la forme d'une bibliothèque de classes en C++. Ces classes peuvent être approximativement séparées en trois groupes :

1. Les outils généraux comme les vecteurs, matrices, listes, etc. ...
2. L'implémentation des neurones et des réseaux de neurones ;
3. Les algorithmes d'apprentissage.

Les sous-sections suivantes décrivent brièvement chaque sous-ensemble. Une description plus complète est disponible dans [7].

3.2 Outils généraux

NSK utilise intensivement certaines particularités du C++ :

- Le contrôle de la mémoire et des affectations : la plupart des types "conteneurs" (vecteurs, matrices, graphes, etc. ...) utilisent une sémantique de copie paresseuse. Quand on écrit `u=v` pour deux vecteurs, `u` devient un autre nom pour `v` : il n'y a pas copie effective. Celle-ci n'aura lieu que si on cherche à modifier `u` ou `v`. De la même façon, on gère des extractions de "sous-conteneurs" de façon efficace.

- Redéfinition des opérateurs : les types numériques comme les vecteurs peuvent être utilisés avec une notation proche des mathématiques en écrivant par exemple `u=v+w` au lieu de `u=Sum(v,w)`. Cette méthode, ajoutée au contrôle automatique de la mémoire, facilite grandement l'écriture d'algorithmes complexes et clarifie le texte des programmes.

- La généricité (classes *template* en C++) : cette méthode est indispensable pour les classes containers comme une classe liste, mais elle est aussi utilisée pour les types numériques, afin de permettre les expérimentations impliquant une représentation spéciale des paramètres des neurones (réels codés en virgule fixe par exemple), prélude indispensable à une implémentation matérielle.

3.3 Réseaux de neurones

La classe `Neuron` permet de gérer les neurones. Un neurone particulier sera obtenu comme classe dérivée de la classe `Neuron`, trois méthodes importantes étant à redéfinir : le calcul de la sortie et ceux des deux différentielles partielles.

On utilise aussi la notion de `Jacobian`. Cette classe spéciale permet d'optimiser les calculs matriciels faisant intervenir les matrices jacobiennes des neurones. Elle ne fournit que des méthodes de multiplication par une autre matrice (le jacobien intervenant comme membre de droite, comme expliqué dans la sous-section 2.2). Quand on veut ajouter une nouvelle méthode d'optimisation des calculs, il suffit de dériver une classe de `Jacobian`.

Les réseaux de neurones sont gérés par la classe `NeuralNet` qui dérive de la classe `Neuron`. Toutes les fonctions de calcul sont implémentées et donc automatiquement utilisables pour n'importe quel type de neurone et pour toute structure de graphe. Bien entendu, on peut dériver des classes de `NeuralNet` afin d'implémenter des modèles spéciaux ou ajouter des méthodes.

3.4 Algorithmes d'apprentissage

Les algorithmes d'apprentissage sont implémentés par la classe `SupervisedAlgo` qui travaille avec les classes `SampledFunction` (fonction échantillonnée à apprendre), `ErrorFunction` (mesure de l'erreur entre la fonction à apprendre et la sortie du neurone), `SelectSubSet` (méthode de sélection d'un sous-ensemble d'apprentissage) et `NeuralNet` (et non pas `Neuron` comme dans le modèle général, ceci dans le simple but de faciliter l'implémentation d'algorithmes tenant compte de la structure du réseau). Une description du fonctionnement de la classe `SupervisedAlgo` est fournie dans la section 2.4.

4 Applications

A ce jour, NSK a été utilisé pour trois études différentes :

1. Dans un récent article, nous avons proposé une méthode d'accélération de la convergence pour des MLP (cf. [16]). Cette méthode est basée sur un changement de

paramètres pour les neurones des MLP. Comme nous l'avons vu dans la sous-section 2.3, la sortie d'un neurone est donné par $f(\sum_{i=1}^{n} a_i x_i + b)$, c'est à dire en notation vectorielle $f((\vec{a}|\vec{x}) + b)$. Pour obtenir une meilleure stabilité dans l'apprentissage, nous utilisons deux vecteurs, le vecteur de dilatation $\vec{d}$ et celui de translation $\vec{t}$. La sortie d'un neurone devient alors $f(\vec{d}|(\vec{x} - \vec{t}))$. Grâce à NSK, l'implémentation de ce nouveau modèle a seulement demandé la mise en place d'un nouveau neurone. Tous les algorithmes classiques étaient alors automatiquement disponibles, ce qui nous a permis de nous concentrer sur la mise en place d'un algorithme hybride, tenant compte de l'architecture du réseau (cf. [16]).

2. Des expériences comparant les MLP classiques aux réseaux d'ondelettes ont été menées avec NSK ([10]) dans le cadre de l'approximation de fonctions quelconques par réseaux de neurones.

3. Des expériences faisant intervenir les algorithmes génétiques sont en cours. Nous étudions ces algorithmes comme méthode d'optimisation de fonctions en général et comme méthode d'apprentissage pour les réseaux de neurones en particulier (cf. [9]). L'apprentissage s'effectue par l'utilisation conjointe d'un algorithme génétique et d'un algorithme de descente de gradient. Le premier algorithme cherche une bonne solution pour initialiser efficacement la descente de gradient. NSK permet d'appliquer à un réseau de neurones quelconque un algorithme d'optimisation sans contrainte, en introduisant seulement quelques règles de programmation peu contraignantes au niveau de l'implémentation de l'algorithme.

5 Conclusion et perspectives

Dans cette communication, nous avons présenté un modèle général pour les réseaux de neurones non récurrents. NSK, propose une implémentation en C++ de ce modèle. Ce noyau est facilement extensible et portable. Il offre de plus de bonnes performances puisqu'il est algorithmiquement équivalent aux implémentations classiques de certains modèles particuliers comme les MLP.

NSK est encore en phase de développement. Un modèle étendu des réseaux de neurones, permettant de gérer les réseaux récurrents tels qu'ils sont décris dans [13], est en cours de développement. Nous nous proposons d'autre part de mettre en place un générateur automatique de code C pour des machines MIMD (basées sur l'envoi de messages) permettant de réaliser une parallélisation automatique d'un réseau de neurones décrit par NSK. Cette parallélisation prendra deux formes distinctes : une parallélisation des calculs neuronaux en eux-mêmes (cf. [3]) ou une parallélisation basée sur le découpage de l'ensemble d'apprentissage (voir par exemple [14]).

References

[1] M. Azema-Barac, M. Hewetson, M. Recce, J. Taylor, P. Treleaven, and M. Vellasco. Pygmalion : Neural network programming environment. In *Proc. Int. Neural Network Conf.*, volume II, pages 709–712, 1990.

[2] Léon Bottou. *Une Approche théorique de l'Apprentissage Connexioniste ; Applications à la reconnaissance de la Parole.* Thèse de doctorat, Université d'Orsay, 1991.

[3] Hervé Bourdin, Bernard Girau, and Loïc Prylli. Parallelisation interne des réseaux d'opérateurs. Technical report, Ecole Normale Supérieure de Lyon, Oct. 93.

[4] Pim H.W. Buurman, Wim J. M. Philipsen, and John van Spaandonk. PLANNET : a New Neural Net Simulator. In O. Simula T. Kohonen, K. Mäkisara and J. Kangas, editors, *Artificial Neural Networks*, pages 1481–1484. Elsevier Science B.V., 1991.

[5] Benoît Derot, Philippe Escande, and Catherine Moulinoux. Nacre : a neuron-oriented programming environment. In *Proc. Neuro-Nimes*, pages 183–200, 1989.

[6] L. Fuentes, J.F. Aldana, and J.M. Troya. Urano : an object-oriented artificial neural network simulation tool. In *Proc. Int. Workshop on Artificial Neural Networks*, volume 686, pages 364–369. Springer-Verlag, 1993.

[7] Cédric Gégout, Bernard Girau, and Fabrice Rossi. Spécifications de NSK. Technical report, Thomson-CSF/SDC, Août 1993.

[8] Cédric Gégout, Bernard Girau, and Fabrice Rossi. A General Feed-Forward Neural Network Model. Technical report NC-TR-95-041, Neuro-COLT, Royal Holloway, University of London, May 1995. Alaivable at `http://apiacoa.org/publications/1995/neurocolt1995.pdf`.

[9] Cédric Gégout and Fabrice Rossi. Continuous Parameter Optimization with Genetic Algorithms. Application to Neural Network Initialization. Technical report, Thomson-CSF/SDC, Nov. 1993.

[10] Bernard Girau. Réseaux neuronaux dérivés des ondelettes : Application à l'approximation de fonctions. Raport de maîtrise, Sept. 93.

[11] Alexander Linden and Christoph Tietz. Combining Multiple Neural Network Paradigms and Applications using SESAME. In *IJCNN*, volume II, pages 528–534, Baltimore, June 1992.

[12] Edmond Mesrobian and Josef Skrzypek. A Software Environment for Studying Computational Neural Systems. *IEEE Trans. on Software Engineering*, 18(7):575–589, July 1992.

[13] Olivier Nerrand, Pierre Roussel-Ragot, Léon Personnaz, Gérard Dreyfus, Sylvie Marcos, Odile Macchi, and Christophe Vignat. Feedback neural networks for non-linear adaptive filtering. In *Proc. Neuro-Nîmes'91*, pages 753–756, November 1991.

[14] Hélène Paugam-Moisy. On a parallel algorithm for back-propagation by partitioning the training set. In *Proc. Neuro-Nîmes*, pages 53–65, 1992.

[15] Tomaso Poggio and Federico Girosi. Networks for approximation and learning. *Proc. IEEE*, 78(9):1481–1497, September 1990.

[16] Fabrice Rossi and Cédric Gégout. Geometrical Initialization, Parametrization and Control of Multilayer Perceptrons : Application to Function Approximation. In *Int. Conf. on Neural Networks*, volume I, pages 546–550, Orlando (Florida), June 1994. IEEE.

[17] Qinghua Zhang and Albert Benveniste. Wavelet networks. *IEEE Trans. On Neural Networks*, 3(6):889–898, November 1992.