MASTER EN SCIENCES INFORMATIQUES

Prévision à l'aide de réseaux neuronaux

Simon, Nicolas

Award date:
2002

Facultés Universitaires Notre Dame de la Paix, Namur
Institut d'Informatique
Année académique 2001-2002

Prévision à l'aide de réseaux neuronaux

Nicolas Simon

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique

Résumé

Ce travail aborde l'étude des séries temporelles et montre quelques méthodes statistiques de prévision. Nous avons ensuite une introduction sur les réseaux neuronaux avant d'aborder plus en détail le réseau multicouche non bouclé encore appelé Perceptron multicouche (PMC). Nous abordons ensuite l'intégration du réseau neuronal dans un environnement de prévision ainsi que différents traitements à faire subir aux données et principalement en vue d'intégrer de multiples variables quantitatives ou catégorielles. Nous évaluerons ensuite le système sur différentes séries avant de conclure.

Mots-clés : Prévision – Réseaux neuronaux – Perceptron multicouche.

Abstract

This work studies time series and shows some statistical forecasting methods. We have an introduction to neural network before a deep incursion into feed forward neural network also called multilayer Perceptron (MLP). We study the integration of the neural network into a forecasting environment and different data processing to include many numerical and symbolic variables. We evaluate the system on different series before concluding.

Keywords: Forecasting – Neural networks – multilayer Perception

Avant-propos

Je tiens ici à remercier tous ceux qui m'ont aidé de près ou de loin dans la réalisation de ce mémoire. Je pense à Madame Monique Noirhomme, mon promoteur au sein des FUNDP, à Madame d'Udekem-Gevers, conseillère pédagogique, ainsi qu'aux différents enseignants que j'ai eu la chance de fréquenter. Je souhaite également remercier Monsieur Daniel Gonsette, Administrateur de la Société Holydis, pour m'avoir accordé temps et moyen à la réalisation de ce mémoire mais également à la réalisation de ces deux années de licence. Je salue également mes collègues de bureau qui m'ont également bien soutenus. Enfin je voudrais remercier ma femme Xuan et mon fils Henri pour leur immense sacrifice et la patience dont ils ont fait preuve pour m'aider.

Table des matières

Résumé 2
Abstract 2
Avant-propos 3
Table des matières 4
Chapitre 1 Introduction 6
1.1 Contexte 6
1.2 Approche 7
Chapitre 2 Modélisation et prévision des séries chronologiques 8
2.1 Les séries chronologiques 8
2.1.1 Définitions 8
2.1.2 Principaux concepts des séries chronologiques. 9
2.2 La modélisation prévisionnelle 10
2.2.1 Généralités 10
2.2.2 Les modèles linéaires 11
2.2.3 Les modèles à moyennes mobiles (MA) 11
2.2.4 Les modèles autorégressifs (AR) 11
2.2.5 Les modèles autorégressifs à moyennes mobiles (ARMA) 12
2.2.6 Les modèles non linéaires 13
Chapitre 3 Présentation des réseaux neuronaux 14
3.1 Introduction 14
3.2 Le neurone formel 16
3.2.1 Les connexions entre les unités 16
3.2.2 Activation et règles de sortie 16
3.3 Topologie du réseau 17
3.4 Apprentissage 19
3.4.1 Paradigme de l'apprentissage 19
3.4.2 Modification des poids 20
3.5 Classement de quelques réseaux de neurones. 20
3.5.1 Réseau de Hopfield 20
3.5.2 Cartes autoorganisatrices 21
3.5.3 Réseau ART (Adaptive Resonance Theory) 21
3.6 Etude du Perceptron 21
3.6.1 Présentation 21

3.6.2 Algorithme de rétropropagation du gradient 22
3.6.3 Utilisation de la rétropropagation du gradient 24
3.6.4 Problème du surapprentissage (overfitting) 25
3.7 Comparaison du vocabulaire des méthodes statistiques et neuronales. 26
Chapitre 4 Environnement et modélisation des variables 27
4.1 Gestion de variables de périodes différentes. 28
4.2 Gestion de variables catégorielles 29
4.3 Utilisation de variables connues dans le futur. 30
Chapitre 5 Implémentation de la solution 31
5.1 L'environnement 31
5.2 Traitements des variables 33
5.3 Traitements du résultat 33
Chapitre 6 Méthodes d'évaluation des résultats 35
6.1 Erreur quadratique 35
6.2 Erreur quadratique moyenne 35
6.3 Erreur quadratique moyenne normalisée 35
Chapitre 7 Création et test de différents modèles 37
Chapitre 8 Analyses 39
8.1 Analyse des variables catégorielles sur la qualité des résultats 39
8.2 Analyse de l'architecture sur la qualité des résultats. 39
8.3 Analyse de la méthode 39
Chapitre 9 Conclusions et perspectives 41
Bibliographie 42
Annexes 43
Annexe A : Code source de l'implémentation en C du réseau de neurones 43
Annexe B : Format des fichiers utilisés par le réseau de neurones 60
Le fichier de description du réseau 60
Le fichier de description des données 60
Les fichiers de données 60

Chapitre 1

Introduction

1.1 Contexte

Le sujet de ce mémoire résulte d'un intérêt personnel pour la compréhension des réseaux de neurones. J'avais entrevu les réseaux de neurones lors de mon travail de fin d'étude pour mon Graduat en Informatique. J'ai alors choisi de réaliser une librairie de traitement d'images. C'est alors que j'ai aperçu les réseaux de neurones comme un classificateur dans la reconnaissance d'images [Gonzalez, 1993, pp.595-619]. Par la suite, et dans le cadre de mon travail, j'ai eu à développer un logiciel de prévision d'activité en vue de faire de la planification. Enfin, c'est lors de cette licence en Informatique que j'ai pu étudier les réseaux de neurones dans le cadre de la prévision. Je désire ici remercier mon employeur pour avoir accepté que je consacre du temps à ce sujet.

Ce mémoire est réalisé au sein de la société Holydis. Holydis est une société installée à Charleroi et qui depuis 1994, réalise des logiciels de gestion de ressources humaines. Son objectif premier est de réaliser et commercialiser des outils de planification de ressources humaines pour le secteur de la grande distribution.

Nos logiciels calculent des plannings sur base des personnes engagées, de leurs contrats ainsi que d'autres contraintes paramétrables tout en optimisant la couverture d'une charge de travail. Cette charge de travail représente un nombre théorique de personnes devant être présente par unité de temps (ex. par demi-heure) pour satisfaire un objectif (ex. minimisation du temps d'attente aux caisses).

Le calcul de la charge de travail est réalisé sur base du volume d'activité prévue (nombre de clients, nombre d'articles, chiffre d'affaires, ...) et sur base d'une productivité moyenne.

Nous avons également réalisé un logiciel enregistrant les transactions des caisses en vue de constituer un historique et de pouvoir établir des prévisions. Ces prévisions ont suscité deux problèmes : Les données sont-elles correctes ? Comment les extrapoler ?

Nous pouvons remarquer, lorsque les files d'attente aux caisses sont importantes, un décalage dans le temps entre le moment où un client se présente à la caisse, début de l'attente dans la file, et le moment où la transaction est réalisée. Ce décalage nous suggère de corriger les données historiées afin de calculer une charge de travail en phase avec l'affluence des clients aux caisses. Cette correction de l'historique est l'objet de logiciels fonctionnant sur base de l'observation de file d'attente, cette observation étant réalisée à l'aide de caméras et de reconnaissance d'images.

Comment extrapoler les données ? C'est précisément la question récurrente qui justifie qu' Holydis s'intéresse toujours à cette question.

Par ailleurs, Holydis s'est intéressé à d'autres cibles commerciales, et notamment les centres d'appels (Call center). Les centres d'appels sont des sociétés ou des divisions de grandes sociétés qui offrent des services de gestion d'appels téléphoniques. On retrouve ces centres d'appels derrière les numéros verts, les services à la clientèle, ...

Les centres d'appels font massivement appel à du personnel travaillant en horaire variable, et c'est pour cette raison que nous réalisons leurs plannings. La charge de travail est alors établie sur base du nombre d'appels reçus par unité de temps.

1.2 Approche

Je désirais trouver une méthode de prévision qui me permette de réaliser de bonnes prévisions en incluant des données qui me semblaient influencer la prévision. Ces données pouvaient être logiques tout comme les jours fériés

J'ai examiné quelques méthodes statistiques, mais elles avaient en général les défauts suivants : hypothèse forte concernant la stationnarité de la série. Elles n'intégraient pas non plus de multiples variables aux périodes différentes, et enfin elles s'accommodent mal de variables catégorielles.

Au cours de ma recherche, j'ai découvert un article [Bonnet, 1997a] qui m'a éclairé sur différentes pistes. Les réseaux de neurones, et en particulier le Perceptron multicouches, sont de bons outils pour faire des prévisions. De plus, Bonnet [Bonnet, 1997b] donnait une méthode pour encoder des variables symboliques. C'était le point de départ de mon investigation.

Afin de réaliser ce système, je dus m'intéresser successivement aux réseaux de neurones, au pré-traitement de mes données et enfin aux méthodes d'évaluation des résultats.

Ce mémoire est organisé afin de suivre ces étapes, à savoir: une présentation de quelques approches statistiques au chapitre 2 ; une présentation générale des réseaux neuronaux et plus particulièrement des réseaux non bouclés au chapitre 3 ; le chapitre 4 présente l'environnement mis en place autour du réseau de neurones ainsi que la gestion des différents types de variables; l'implémentation du système de prévision et une explication concernant les transformations effectuées sur les données se trouve dans le chapitre 5 ; le chapitre 6 présente les méthodes d'évaluation des résultats alors que le chapitre 7 présente l'utilisation du système sur plusieurs jeux de tests. Viennent ensuite l'analyse de la méthode et les conclusions dans les chapitres 8 et 9.

Chapitre 2

Modélisation et prévision des séries chronologiques

La prévision ou l'anticipation de phénomènes physiques repose sur l'analyse de variables qui se prêtent à l'observation. Si un phénomène est régi par un système d'équations déterministes connues, alors il est en principe possible de résoudre ce système pour prédire le phénomène en connaissant les conditions initiales. Par contre, si ces équations sont inconnues, il est nécessaire d'approximer les lois qui gouvernent l'évolution du phénomène à partir de l'historique de ses observations antérieures. C'est pour cette raison que nous présentons les mécanismes fondamentaux des séries chronologiques afin de comprendre la série et d'en faciliter la prévision, [Coutrot, 1984] [Bonnet, 1997a] [Thiria, 1997]. Nous examinerons ensuite quelques modèles de prévision.

2.1 Les séries chronologiques

La réalisation de prévisions nécessite l'approche d'un domaine particulier de l'économétrie : les statistiques chronologiques ou séries chronologiques (séries temporelles ou chroniques).

Les séries chronologiques sont généralement constituées d'une suite d'observations chiffrées, ordonnées dans le temps.

La grandeur, dont on suit l'évolution, peut être :

- Un stock, se rapportant à des dates déterminées (population, effectifs salariés, clientèle, ...);
- Une intensité (prix, part de marché, taux de réussite à un examen, ...);
- Un flux, se rapportant à un intervalle de temps (ventes, nombre de clients, ...).

Plus précisément, une série chronologique est formée de l'ensemble des observations d'un caractère quantitatif ou qualitatif à des époques successives. Le temps joue ici un rôle de variable explicative.

Nous noterons la série chronologique $\{y_1, y_2, ..., y_n\}$ relative à une variable y sous une forme synthétique $\{y_t\}$. Cette notation repose sur l'hypothèse que les dates auxquelles la variable a été observée sont équidistantes et peuvent être représentées par les n premiers nombres entiers. Dans la forme synthétique de la série $\{y_t\}$, nous sous-entendons que t varie de 1 à n.

2.1.1 Définitions

L'utilisation des séries temporelles impose la compréhension d'un certain nombre de concepts dont les définitions sont présentées ici dans une liste non exhaustive :

Une *prévision* peut être définie comme une affirmation statistique concernant des événements inconnus ou à venir.

L'horizon de prévision est l'intervalle de temps qui sépare le moment où la prévision est effectuée (origine de la prévision) et le moment pour lequel on désire la prévision (fin de la prévision). Selon cet horizon on distinguera les prévisions à court, à moyen et à long terme.

Les variables sont classées suivant cette nomenclature :

Les *variables quantitatives* : ces variables sont représentées par des valeur numériques et peuvent être continues (ex. la température) ou discrètes (ex. nombre de clients).

Les *variables catégorielles* : ces variables sont représentées par des catégories. Nous dirons encore qu'une variable est une variable catégorielle simple si elle ne peut prendre qu'une valeur simultanément et nous parlerons de variable catégorielle multiple si elle est multivaluée.

2.1.2 Principaux concepts des séries chronologiques.

L'analyse descriptive des séries chronologiques met en évidence quatre types de «mouvements» :

- Un mouvement longue durée ou tendance (T)

Les mouvements de long terme, en anglais «trend», traduisent l'allure d'ensemble du phénomène. La série chronologique peut être globalement croissante, décroissante ou stable. La connaissance de cette tendance permet la comparaison de différentes séries chronologiques. De plus, c'est à partir de la tendance que les autres composantes de la série seront étudiées.

- Une composante saisonnière (S)

Les variations saisonnières correspondent à la répétition d'un profil particulier de la variable dépendante autour d'une tendance déterminée. Il est possible de parler de pointes saisonnières et de creux saisonniers. Ces variations sont périodiques et plus ou moins régulières. Leur période peut être journalière (trafic, horaire du métro,...), hebdomadaire (nombre d'heures prestées par jour) ou annuelle.

- Une composante cyclique (C)

La composante cyclique exprime des fluctuations longues que la grandeur peut présenter autour de la tendance à long terme. Les fluctuations cycliques qui traduisent la vie économique peuvent avoir une amplitude de plusieurs années.

- Une composante résiduelle, variations accidentelles ou erreurs (e)

Autour des mouvements précédents se produisent des variations accidentelles de trois types. Elles peuvent correspondre à l'imprécision des mesures statistiques, aux aléas de la vie économique (grèves, sécheresses, ...) ou bien encore représenter la part de l'évolution dont les composantes précédentes ne peuvent rendre compte. On leur donne parfois, pour cette raison, le nom de fluctuations résiduelles.

Au final, la donnée observée x, à la date t, d'une série chronologique peut s'interpréter comme résultant de la superposition de ces quatre composantes.

$$x(t) = \Im(T, S, C, e)$$

Deux approches sont possibles :

- On considère l'effet cumulatif des composantes

$$x(t) = T(t) + S(t) + C(t) + e(t)$$

- On considère l'effet multiplicatif des composantes

$x(t) = T(t).S(t).C(t).(1 + e(t))$

2.2 La modélisation prévisionnelle

2.2.1 Généralités

Les outils prévisionnels permettent d'anticiper l'évolution future du phénomène considéré, sous l'hypothèse que les caractères mesurés, les régularités et autres permanences constatées dans le passé se reproduiront jusqu'à l'horizon retenu. En d'autres termes, il est possible de réaliser des prévisions « toutes choses étant égales par ailleurs ».

Chaque observation du phénomène étudié est la réalisation d'une variable aléatoire. Par conséquence, une chronique est constituée d'une famille de variables aléatoires. Il s'agit donc d'un processus stochastique ou aléatoire.

La modélisation consistera à obtenir des informations pertinentes sur le processus générateur de la série temporelle observée. Ainsi après avoir observé graphiquement la série, on choisit une famille de modèles.

Les paramètres du modèle sont estimés par ajustement aux données. Ensuite le modèle est validé par un certain nombre de tests. Ceci ressemble à une démarche statistique classique sur échantillons. La différence essentielle est que dans le cas des statistiques prévisionnelles, les variables successives sont liées. Il existe un fort lien de dépendance pouvant être mesuré par un coefficient de covariance ou un coefficient de corrélation. C'est le lien qui permettra de réaliser des prévisions.

Il est aussi très important de comprendre que la qualité de la prévision dépend de la façon dont la série évolue. Plus la série est une fonction régulière du temps, plus il sera facile de prévoir. La qualité de la prévision dépend aussi de l'horizon et est généralement meilleure lorsque cet horizon est petit.

Il existe deux grands types de méthodes. Il s'agit des méthodes extrapolatives, qui utilise le passé de la variable, et les méthodes explicatives qui font reposer la prévision de la série, Y, sur celle de facteurs Xi dont la liaison avec Y a été très importante dans le passé.

Le premier objectif des méthodes statistiques a été de nature à expliquer le phénomène générateur de la série et a donné lieu à l'analyse descriptive traditionnelle. L'analyse descriptive va tenter d'isoler chacune des composantes de la série chronologique afin de comprendre la série et plus tard d'en faciliter la prédiction.

On va donc essayer d'isoler ces composantes.

La partie *tendance* va faire l'objet d'une stationnarisation. Cette stationnarisation se fait en général par la méthode des moyennes mobiles. L'idée de base est que la composante périodique à une moyenne constante sur une période, et que l'on va donc pouvoir l'extraire avec un filtre passe bas. L'inconvénient de ce filtre est de noyer le bruit dans la tendance. On se reportera au point 2.3 pour la méthode des moyennes mobiles.

La composante *saisonnière* va faire l'objet d'une dessaisonalisation. Cette dessaisonalisation se fait en général par la méthode de la différenciation dont le principe est de soustraire la tendance de la série originale. La différenciation est un filtre passe haut dont l'inconvénient est d'intégrer les changements de tendance dans le bruit.

La plupart des méthodes de prévision font appel à la dimension autorégressive de la série. Il nous faut donc la définir.

Un processus autorégressif général s'exprime sous la forme :

$$x(t) = f\left[x(t-1),...,x(t-n)\right] + e(t)$$

où *e(t)* est un bruit gaussien.

On en déduit le prédicteur autorégressif associé :

$$\hat{x}(t) = f\left[x(t-1),...,x(t-n)\right]$$

2.2.2 Les modèles linéaires

Une méthode standard pour générer des modèles capables de prévoir est d'utiliser des modèles globalement linéaires tels que les modèles à moyennes mobiles, les modèles autorégressifs ou encore les modèles autorégressifs à moyennes mobiles.

2.2.3 Les modèles à moyennes mobiles (MA)

Supposons que nous ayons une série en entrée $\{e_t\}$ et que nous voulions la modifier pour produire la série observée en sortie $\{y_t\}$. Postulons que le système soit linéaire et que la valeur présente de y soit influencée par la valeur actuelle de la série e ainsi que par les N dernières valeurs de la série e, la relation entre l'entrée et la sortie est :

$$y(t) = \beta_1.e(t-1) + ... + \beta_N.e(t-N) = \sum_{n=1}^{N} \beta_n.e(t-n) \qquad (2.1)$$

L'équation (2.1) décrit un filtre de convolution classique. La nouvelle série y est générée par un filtre linéaire aux coefficients $\beta_1,...,\beta_N$. En statistique, ce modèle est appelé modèle à moyennes mobiles du N^{ième} ordre, ou MA(N) alors qu'en ingénierie, ces modèles sont appelés filtre à réponse finie (FIR) parce que leurs sorties sont garantie d'atteindre zéro N étapes après que l'entrée arrive à zéro.

Les processus MA sont stationnaires et s'appliquent généralement à certaines séries très bruitées.

Inconvénient : La classe des processus MA est très restreinte.

L'ajustement des paramètres est complexe et délicat (phénomène d'oscillation).

L'utilité essentielle de ces modèles est de permettre de présenter simplement la classe des processus ARMA et le prédicteur associé.

2.2.4 Les modèles autorégressifs (AR)

Yule (1927) a proposé une classe de modèles autorégressifs définis comme suit :

$$y(t) = \alpha_0 + \sum_{m=1}^{M} \alpha_m y(t-m) + e(t) = \hat{y}(t) + e(t) \qquad (2.2)$$

L'équation (2.2) est celle d'un modèle autorégressif du M^{ième} ordre, noté AR(M). En fonction de l'application, *e(t)* représente une entrée contrôlée de système ou un bruit.

Les modèles autorégressifs sont également considérés comme des filtres à réponse infinie (IIR) car leurs sorties n'atteignent pas forcément zéro après M étapes.

Propriétés :

- Le prédicteur AR est linéaire.
- Le prédicteur AR est très simple (M paramètres).
- Le calcul des paramètres est très facile et peu coûteux en temps de calcul (méthode des moindres carrés par exemple)

Applications :

- Les séries très simples (quasi-linéaires).
- Les séries très complexes, où la meilleure politique est de reproduire à l'instant t la valeur observée à l'instant $t-1$.

Inconvénient :

- Il leur est impossible de reproduire des séries non linéaires même simples.

2.2.5 Les modèles autorégressifs à moyennes mobiles (ARMA)

L'étape suivante en complexité est de regrouper les parties AR et MA dans le modèle, ce qui donne le modèle communément appelé modèle autorégressif à moyennes mobiles, ou encore ARMA(M,N)

$$y(t) = \sum_{m=1}^{M} \alpha_m y(t-m) + \sum_{n=1}^{N} \beta_n e(t-n) \qquad (2.4)$$

Les modèles ARMA dominent le domaine de l'analyse des séries temporelles depuis plus d'un demi siècle. Si le modèle est bon, il transforme la série en un petit nombre de coefficients et en un bruit résiduel.

Propriétés :

- Les processus ARMA sont donc la superposition d'un processus AR et d'un processus MA. Ils cumulent donc leurs avantages et leurs inconvénients.
- Les processus ARMA sont stationnaires.
- Un théorème d'optimalité indique que tout processus stationnaire peut être approché uniformément (au sens de la norme euclidienne) par un processus ARMA.
- Il est aisé de tirer des renseignements sur la nature de la série à partir du processus ARMA associé.
- Apprentissage peu coûteux en temps de calcul.

Applications :

- Virtuellement toutes les séries stationnaires.

Inconvénients :

- Nécessité de la stationnarité (au moins au second ordre).
- Linéarité

2.2.6 Les modèles non linéaires

La limitation des modèles linéaires face à de simples non régularités nous pousse à envisager des modèles non linéaires. Après le règne des modèles linéaires, sont apparus des modèles semi-linéaires. Il s'agit de modèles autorégressifs à seuil (TAR) développés par Tong et Lim et qui sont globalement non linéaires : l'idée consiste à compartimenter l'espace des phases et d'appliquer dans chaque région un modèle ARMA. Toutefois, cette approximation linéaire par morceau nécessite un grand nombre de partitions lorsque la surface présente des non-linéarités quadratiques (i.e. la fonction logistique). Un autre moyen d'étendre l'équation (2.4) fait appel à un polynôme d'ordre supérieur ; ces modèles portent le nom de suites de Voltera [Voltera 59].

Les modèles TAR, les suites de Voltera, les fonctions splines, radiales, etc. [Castagli 89 et 92] élargissent considérablement le champ des fonctionnalités pour modéliser les suites temporelles mais au prix d'une plus grande complexité pour l'adaptation des paramètres. Pour que des modèles non linéaires soient utiles, il faut un moyen d'exploiter la non-linéarité des données pour guider la construction du modèle. Or le manque de compréhension de ce problème a sérieusement freiné leurs extensions.

L'approche connexionniste fait également partie des modélisations non linéaires et fait l'objet du prochain chapitre.

Chapitre 3

Présentation des réseaux neuronaux

3.1 Introduction

Le neurone artificiel a été développé sur base du neurone biologique (Fig. 3.1) dont il est une simplification extrême. Le neurone biologique permet de faire une intégration spatiale (sommation de la contribution des différentes dendrites) et temporelle (délais de transmissions) des signaux que perçoit une multitude de récepteurs appelés *dendrites*. Lorsque le potentiel électrique à proximité de la membrane du corps cellulaire le permet, un potentiel d'action est généré puis propagé le long de la fibre nerveuse principale appelée *axone*. Enfin, les *synapses* permettent de diffuser le potentiel vers les dendrites des neurones voisins. Une propriété importante du neurone est sa non-linéarité, c'est-à-dire qu'il y a saturation de son potentiel d'action.

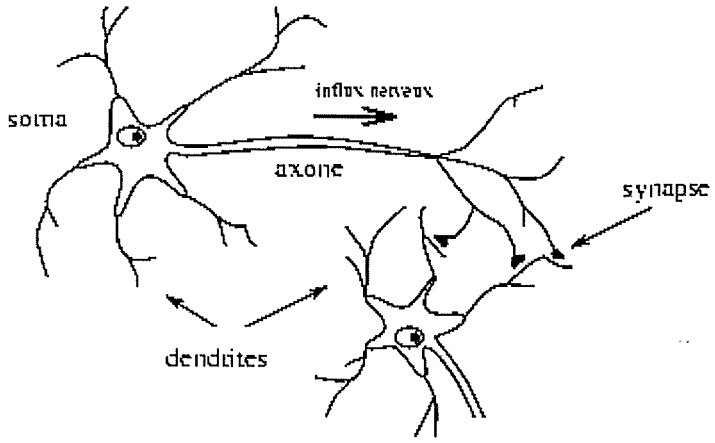

Fig. 3.1 : Illustration d'un neurone biologique. L'influx nerveux généré par le neurone transite par l'axone avant d'aboutir dans les synapses. C'est alors que les neurones voisins reçoivent le signal via leurs dendrites.

Cette description succincte du neurone biologique n'est pas suffisante, nous devrions en effet considérer l'organisation de nos 10^{11} neurones, structurés en couches denses ainsi que la plasticité des synapses et l'apprentissage. On consultera les ouvrages de référence pour une description plus avancée sur ces sujets.

Les réseaux de neurones artificiels sont apparus pour la première fois en 1943, lorsque McCulloch et Pitts ont présenté leurs premiers neurones artificiels comme étant des modèles de neurones biologiques et comme des composants conceptuels de circuits qui pourraient effectuer des tâches de calcul. Ils ont montré, à l'époque, qu'ils pouvaient calculer certaines fonctions logiques à l'aide d'automates linéaires à états finis. En 1949, Hebb souligna l'importance du couplage synaptique et développa un certain nombre de règles, encore largement utilisées, pour la mise à jour des poids synaptiques lors de la phase d'apprentissage. Il faudra attendre 1958 pour voir la naissance d'une méthode analytique rigoureuse

d'adaptation de poids dans un modèle multicouche appelé le *perceptron.* Des réseaux similaires appelés *adalines* furent inventés à la même époque par Widrow. Rossenblatt démontra la convergence d'un algorithme itératif d'adaptation des poids pour la classe élémentaire des perceptrons monocouche. Inspiré des idées de Hebb, de McCulloch et de Pitts, le perceptron était capable d'apprendre à calculer certaines fonctions logiques à l'aide d'exemples en modifiant ses connexions synaptiques.

En 1969, l'analyse de Minsky et Pappert mirent en évidence de sévères limitations du perceptron pour résoudre certains types de problèmes tels que le problème notoire du OU exclusif (XOR). La conséquence fut une désaffection massive des chercheurs pour l'approche connexionniste. Ces limitations provenaient de l'utilisation d'une seule couche de neurones. Le problème était d'avoir une méthode d'apprentissage pour des réseaux multicouches, seuls à pouvoir traiter des problèmes tels que celui du « OU exclusif ». En 1982, Hopfield mit au point un réseau appelé *machine de Boltzmann.* Il avait introduit une fonction d'*énergie* et donnait une dimension stochastique au neurone en incluant une erreur aléatoire à la fonction de transfert. Hopfield avait mis en place une méthode d'apprentissage proche du recuit simulé qui vise à réduire l'énergie représentant l'erreur. Ce modèle levait les limitations du perceptron bien que la convergence de l'algorithme soit particulièrement longue.

C'est avec l'apparition de l'algorithme de *rétropropagation du gradient* que l'on a pu lever les limitations dans l'apprentissage des perceptrons multicouches. Ces importants développements sont dus à Werbos en 1974, et ont été popularisés en 1985 par Rumelhart, Parker et Le Cun. Ceci relança l'intérêt des chercheurs pour les méthodes connexionnistes.

Pour poursuivre la présentation des réseaux de neurones, nous allons introduire quelques notions importantes avant d'aborder les unités de traitements de bases appelées neurones, les différents types de topologie de réseau, ainsi que la stratégie d'apprentissage.

Les réseaux de neurones artificiels sont basés sur le paradigme du traitement parallèle distribué. L'architecture de chaque réseau est basée sur des unités de base très similaires chargées d'effectuer le traitement. Un ensemble de caractéristiques communes peut être distingué pour les modèles de traitements parallèles distribués. [Rumelhart, 1986]

Un ensemble d'unité de traitements, appelé *neurone.* Nous utiliserons un indice k pour désigner le neurone k.

Un *état d'activation* y pour chaque unité correspondant à sa valeur de sortie, noté y_k pour le neurone k.

Des *connexions* entre les unités. Généralement les connexions sont représentées par un *poids* w_{jk} qui détermine l'effet du neurone j sur le neurone k.

Une *loi de propagation* qui détermine l'entrée effective S_k de l'unité sur base de ses entrées externes.

Une *fonction d'activation* f_k qui détermine le nouveau niveau d'activation sur base de l'entrée effective $S_k(t)$ et de l'état courant d'activation $y_k(t)$. La *sortie* ou le *niveau d'activation* du neurone sont deux termes employés qui désignent la même chose.

Une entrée externe θ_k pour chaque unité également appelée *biais*. Le *biais* désigne une constante qui entre dans la somme du neurone. Cette valeur externe est généralement implémentée comme un poids d'un neurone fictif dont le niveau d'activation serait 1.

Une méthode pour organiser l'information, une *règle d'apprentissage*.

Un *environnement* à l'intérieur duquel le système va fonctionner et qui va fournir les signaux en entrée.

3.2 Le neurone formel

Chaque neurone, voir la figure 3.2, va faire un travail relativement simple : recevoir des informations de ses voisins ou d'une source externe et utiliser ces informations pour calculer un signal de sortie qui sera propagé aux autres unités. Nous aurons un second traitement qui sera l'ajustement des poids. Nous avons un parallélisme inhérent au système dans la mesure où les unités peuvent faire leurs calculs en même temps.

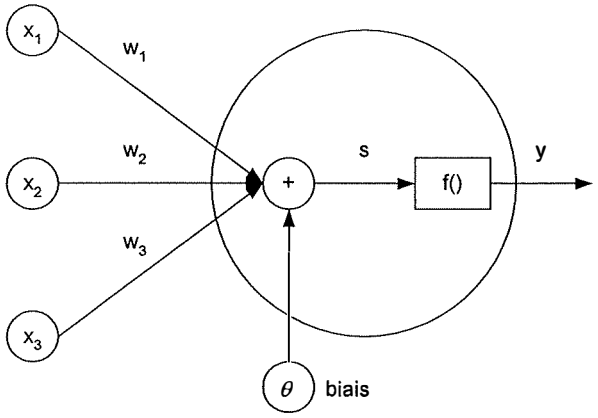

Fig. 3.2 Illustration d'un neurone formel : le modèle effectue une somme pondérée de ses signaux d'entrée puis applique une fonction non linéaire au résultat, habituellement une fonction sigmoïde.

3.2.1 Les connexions entre les unités

Dans la majorité des cas, nous supposerons que les unités auront une loi de propagation additive par rapport aux unités auxquelles elles sont connectées. La partie gauche de la figure 3.2 nous montre où intervient cette loi de propagation. L'entrée totale de l'unité k est simplement la somme pondérée de chaque unité auquel le neurone est connecté plus un terme de biais θ_k.

$$s_k(t) = \sum_j w_{jk}(t) y_j(t) + \theta_k(t) \qquad (3.1)$$

La contribution des w_{jk} positifs est considérée comme une excitation et pour les w_{jk} négatifs comme une inhibition. Nous appellerons une unité avec la règle de propagation (3.1) une unité sigma.

Une autre règle de propagation introduite par Felman et Ballart est connue sous le nom de règle de propagation sigma pi :

$$s_k(t) = \sum_j w_{jk}(t) \prod_m y_{jm}(t) + \theta_k(t) \qquad (3.2)$$

3.2.2 Activation et règles de sortie

Nous avons également besoin d'une règle qui donne l'effet du total des entrées sur l'activation du neurone. La partie droite de la figure 3.2 nous montre où intervient cette règle

d'activation. Nous avons besoin d'une fonction $\Im_k$ qui prend le total des entrées $s_k(t)$ et l'activation courante $y_k(t)$ et produit une nouvelle valeur d'activation pour l'unité k :

$$y_k(t+1) = \Im(y_k(t), s_k(t)) \tag{3.3}$$

Le plus souvent, la fonction d'activation prend seulement le total des entrées $s_k(t)$.

$$y_k(t+1) = \Im_k(s_k(t)) = \Im_k(\sum_j w_{jk}(t) y_j(t) + \theta_k(t)) \tag{3.4}$$

Généralement la fonction d'activation borne la sortie. Ceci est nécessaire pour introduire la non-linéarité dans le modèle.

La fonction sigmoïde est souvent utilisée, elle est représentée sur la figure 3.3 et par l'équation (3.5).

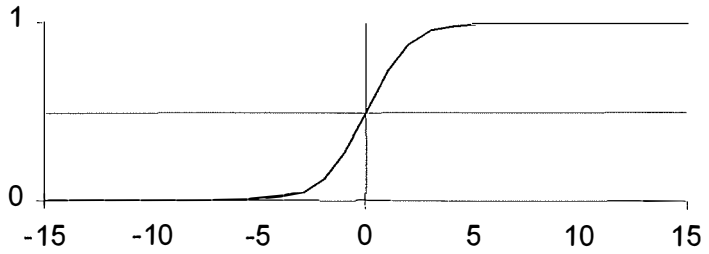

Fig. 3.3 : Graphe de la fonction sigmoïde

$$y_k = \Im(s_k) = \frac{1}{1+e^{-s_k}} \tag{3.5}$$

La fonction semi-linéaire (3.6) est également utilisée car plus rapide à évaluer.

$$y_k = \Im(s_k) = \begin{cases} -1 & si \quad S_k \le -1 \\ S_k & si \quad -1 < S_k < 1 \\ +1 & si \quad S_k \ge 1 \end{cases} \tag{3.6}$$

3.3 Topologie du réseau

Dans les points précédents nous avons montré les propriétés des unités élémentaires, les neurones.

Ici nous allons nous intéresser aux différentes formes d'interconnexions entre les neurones et à la propagation des données.

Les réseaux de neurones sont généralement subdivisés en couches, voir figure 3.4. Nous distinguerons trois types de couches.

La *couche d'entrée* est constituée des entrées venant de l'extérieur du réseau.

La *couche de sortie* est constituée des neurones qui produisent leurs sorties vers l'extérieur du réseau.

Une *couche cachée* est constituée de neurones qui reçoivent leurs entrées et produisent leurs sorties à l'intérieur du réseau.

Le *nombre de couche* d'un réseau est en général le nombre de couches cachées plus la couche de sortie. La couche d'entrée n'est généralement pas comptée car on n'y effectue pas de calcul.

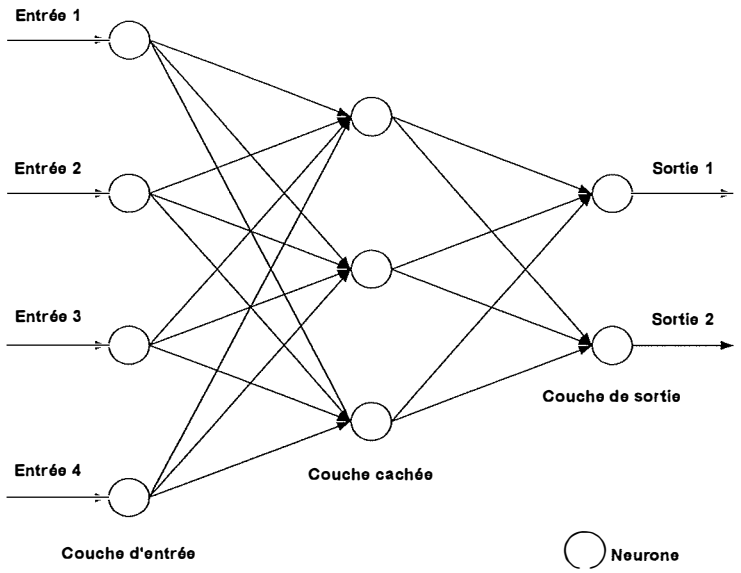

Fig. 3.4 Les réseaux multicouches non bouclés : la sortie d'un neurone dans une couche donnée est une entrée des neurones de la couche suivante. Dans cette illustration, chaque flèche indique une connexion dans le sens gauche-droite.

Une importante distinction doit être faite entre deux types de réseaux.

Les réseaux non bouclés ou «feed forward networks», illustrés par la figure 3.4, où les données vont strictement des neurones d'entrée vers les neurones de sortie. Le traitement des données peut être étendu à de multiples (couches de) neurones mais sans connexions vers l'arrière, c'est-à-dire sans connexion de la sortie du neurone vers une entrée d'un neurone de la même couche ou d'une couche précédente.

Les réseaux récurrents, exemple figure 3.5, qui contiennent des connexions vers l'arrière. Ici, contrairement aux réseaux à propagation avant, les propriétés dynamiques sont importantes.

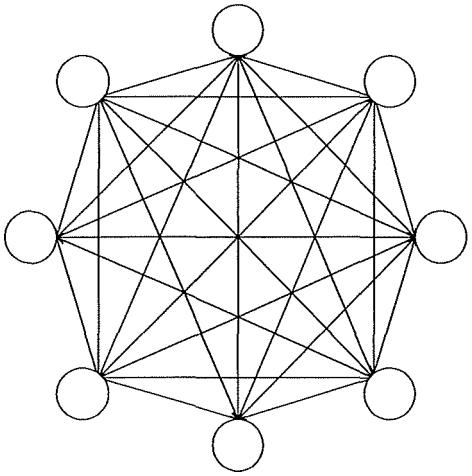

Fig. 3.5 Le réseau autoassociatif de Hopfied. Tous les neurones sont à la fois des neurones d'entrée et de sortie. Lorsqu'un exemple est présenté, le réseau itère vers un état stable et la sortie du réseau consiste en les nouvelles valeurs d'activation des neurones.

3.4 Apprentissage

Un réseau de neurones doit être configuré de façon qu'un ensemble de données entrées produisent les résultats désirés en sortie. Différentes façons d'établir ces connexions existent. Une manière est de mettre les poids explicitement en utilisant une connaissance à priori. Une autre manière est d'entraîner le réseau de neurones en lui soumettant un ensemble de données d'apprentissage et de modifier les poids en fonction d'une règle d'apprentissage.

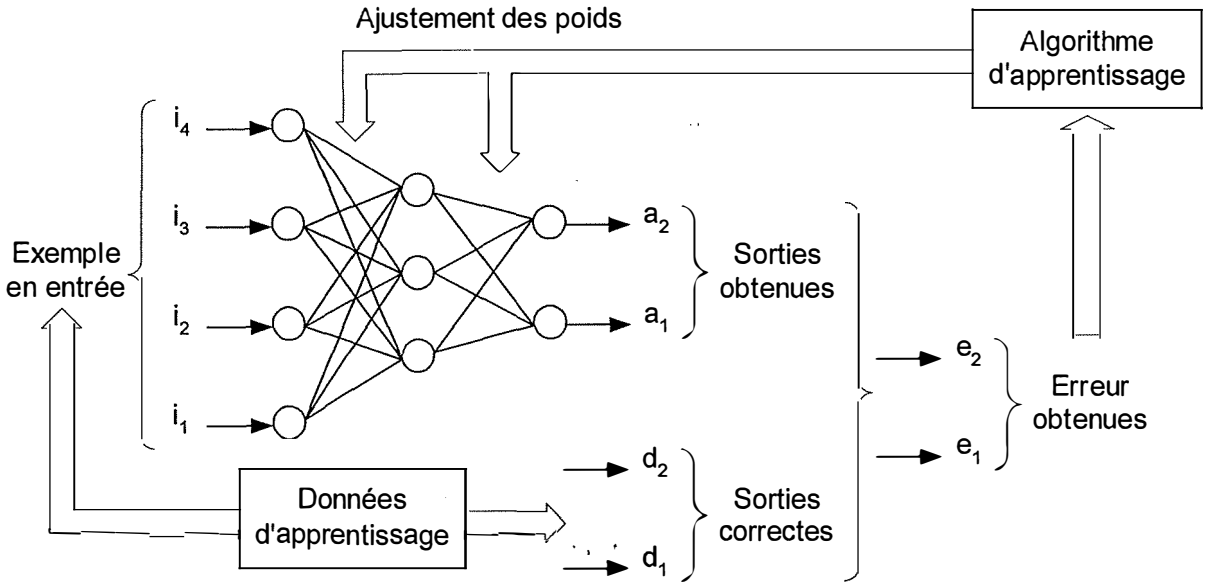

Fig. 3.6 Principe de fonctionnement de l'apprentissage supervisé. On présente les données *i*, ensuite on fait une passe avant dans le réseau, on obtient la sortie *a* que l'on compare à la sortie désirée *d*. L'erreur *e* est la différence entre *d* et *a*. Cette erreur sera utilisée par l'algorithme d'apprentissage pour mettre à jour les poids.

3.4.1 Paradigme de l'apprentissage

Nous pouvons distinguer deux catégories distinctes d'apprentissage.

L'apprentissage supervisé ou l'apprentissage associatif, dans lequel le réseau est entraîné en lui soumettant en ensemble de données en entrée et les données souhaitées en sortie. Ces paires d'entrée et sortie peuvent être fournies par un enseignant externe ou par le système qui contient le réseau (autosupervision)

L'apprentissage non supervisé ou auto organisant, dans lequel un neurone de sortie est entraîné pour différencier le type de données en entrée. Dans ce paradigme, le système est supposé découvrir des propriétés statistiques de la population d'entrée. Contrairement au paradigme de l'apprentissage supervisé, il n'y a pas de catégories fixées à priori dans lesquelles les données doivent être classifiées. Le système doit donc développer sa propre représentation par rapport aux stimuli d'entrée.

3.4.2 Modification des poids

Les deux paradigmes d'apprentissage discutés précédemment nécessitent une adaptation des poids entre les unités en fonction d'une règle de modification. L'idée de base est que si deux neurones j et k sont actifs simultanément, leur interconnexion doit être contrôlée. Si j reçoit son entrée de k, la méthode la plus simple d'apprentissage prescrit de modifier le poids w_{jk} avec

$$\Delta w_{jk} = \gamma y_j y_k \tag{3.7}$$

où γ est une constante positive de proportionnalité représentant la vitesse d'apprentissage. L'équation (3.7) est également connue sous le nom règle d'apprentissage de Hebb.

Une autre règle d'apprentissage largement répandue n'utilise pas l'activation actuelle du neurone k mais la différence entre l'activation actuelle et l'activation désirée, l'erreur locale, pour ajuster les poids :

$$\Delta w_{jk} = \gamma y_j (d_k - y_k) \tag{3.8}$$

dans lequel d_k est l'activation désirée fournie par l'enseignant. L'équation (3.8) est également connue sous le nom règle d'apprentissage de Widrow-Hoff ou la règle du delta.

3.5 Classement de quelques réseaux de neurones.

3.5.1 Réseau de Hopfield

Il s'agit d'un réseau récurrent entièrement connecté fonctionnant comme une mémoire associative non linéaire, capable de retrouver un objet stocké à partir d'une représentation partielle ou bruitée. On peut aussi se servir de ces réseaux comme outil d'optimisation. L'état de chaque cellule peut être mis à jour un nombre aléatoire de fois et ce de façon indépendante des autres cellules, mais en parallèle. Comme toutes les cellules interagissent, la propriété globale du réseau réduit de manière inhérente les temps de calcul.

Dans le cas des machines de Boltzmann, une généralisation des réseaux de Hopfield, l'opération repose sur un concept de la thermodynamique statistique (le recuit simulé).

Les réseaux de Hopfield, les machines de Boltzmann et une de leurs dérivées, la machine ``Mean-Field-Theorem" (MFT) ont été utilisés avec succès dans les cas de segmentation d'images, d'optimisation combinatoire (problème du voyageur de commerce, partitionnement d'un graphe) en plus de leur utilisation comme mémoire à contenu adressable.

3.5.2 Cartes autoorganisatrices

Ce type de réseau (en anglais « Self Organizating Map » ou SOM) à apprentissage non supervisé transforme des modèles (``patterns'') d'entrée de dimension p en une carte discrète de dimension q (le plus souvent 1 ou 2) ordonnée topologiquement. Les points qui sont proches dans la dimension p le sont plus encore dans le treillis en dimension q. Chaque cellule du treillis est représentée par un neurone associé à un vecteur de poids modifiables de dimension p. La correspondance entre chaque vecteur de poids est calculée pour chaque entrée. Le vecteur de poids ayant la meilleure corrélation ainsi que certains de ses voisins sont adaptés pour augmenter encore cette corrélation. De tels réseaux ont été utilisés pour générer des cartes sémantiques, des machines à écrire phonétiques, le ``bi-partitionnement'' de graphes.

Pour un cas particulier de SOM, les LVQ, seul le noeud ayant la meilleure corrélation est adapté. Des réseaux de ce type, dans lesquels seul le vainqueur est sélectionné, sont appelés réseaux à compétition. Il s'agit essentiellement d'un réseau qui ne préserve pas l'ordre topologique, et dont l'usage principal est la catégorisation (``clustering'') ou la compression d'images. Pour plus d'informations on se référera à [Gallinari, 1997] [Kohonen, 2001]

3.5.3 Réseau ART (Adaptive Resonance Theory)

Dans le cadre de l'apprentissage par compétition, il n'y a aucune garantie que les catégories (clusters) formées soient stables, à moins que le coefficient d'apprentissage tende petit à petit vers zéro. Mais alors, le réseau perd en plasticité (i.e. son adaptation au nouvel environnement). L'objectif des réseaux ART est de contourner ce problème. Dans ART, un vecteur de poids (prototype d'une catégorie) n'est adapté que si l'entrée est suffisamment proche du prototype (on parle alors de résonance). Lorsqu'une entrée n'est pas suffisamment voisine des prototypes existants, une nouvelle catégorie est créée dont le vecteur prototype initial est l'entrée qui a provoqué sa création. Une fois la nouvelle catégorie créée, le système cherche à réorganiser les différents clusters pour une meilleure adéquation. Il existe deux classes de réseaux ART : ART-1, défini dans le cas d'entrées binaires, et ART-2 dans le cas de données continues. Pour plus d'informations on se référera à [Tauritz, 2001]

3.6 Etude du Perceptron

3.6.1 Présentation

Le perceptron est un réseau non bouclé structuré en couches (Figure 3.4). Chaque couche est constituée de neurones qui reçoivent leurs entrées des neurones de la couche précédente et envoient leurs sorties vers les neurones de la couche suivante. Il n'y a pas de connexions entre les neurones d'une même couche. Les valeurs d'entrées, considérées comme étant de la première couche, sont en réalité présentées à la première couche cachée. Il n'y a pas de traitement dans la couche d'entrée. La loi de propagation est la somme pondérée des entrées par les poids plus un terme de biais décrite en (3.1). Les sorties des neurones d'une couche sont distribuées sur les neurones de la couche suivante et ainsi de suite jusqu'aux neurones de la couche de sortie. Bien que la rétropropagation puisse s'effectuer sur des réseaux avec n'importe quel nombre de couches cachées, il a été montré [Hornik 1989] que seulement une couche cachée de neurones suffit pour approximer une fonction au nombre fini de discontinuités et ce avec une précision arbitrairement choisie, pour autant que la fonction d'activation des neurones de la couche cachée soit non linéaire.

3.6.2 Algorithme de rétropropagation du gradient

Cet algorithme est au cœur de la justification de la rétropropagation du gradient et figure dans mon implémentation. C'est pour cette raison que j'inclus sa démonstration ici.

Parmi différentes démonstrations que l'on retrouvera dans [Rumelhart, 1986] [Gonzales, 1993] [Kröse, 1996], j'ai traduit celle de Kröse qui me semblait la plus claire.

Il nous faut établir la manière dont on va prendre en compte l'erreur pour la mise à jour des poids en commençant par la description de la sortie.

$$a_i^p = \Im(i_i^p) \tag{3.9}$$

dans lequel

$$i_i^p = \sum_j w_{ij} a_j^p + \theta_i \tag{3.10}$$

Nous définissons

$$\Delta_p w_{ij} = -\gamma \frac{\partial E^p}{\partial w_{ij}} \tag{3.11}$$

La mesure de l'erreur E^p est définie comme étant l'erreur quadratique pour l'exemple p sur les neurones de la couche de sortie comprenant N_o neurones.

$$E^p = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^p - a_i^p)^2 \tag{3.12}$$

alors que $E = \sum_p E^p$ est la mesure de l'erreur globale. Nous pouvons écrire

$$\frac{\partial E^p}{\partial w_{ij}} = \frac{\partial E^p}{\partial i_i^p}\frac{\partial i_i^p}{\partial w_{ij}} \tag{3.13}$$

Le second terme de l'équation (3.10) devient

$$\frac{\partial i_i^p}{\partial w_{ij}} = a_j^p \tag{3.14}$$

Si nous définissons

$$\delta_i^p = -\frac{\partial E^p}{\partial i_i^p} \tag{3.15}$$

L'équation (3.11) devient

$$\Delta_p w_{ij} = \gamma \delta_i^p a_j^p \tag{3.16}$$

Cette règle de modification de poids permet de suivre la descente du gradient sur la surface des erreurs. a_i^p étant connu (c'est la sortie du réseau), il nous reste à calculer δ_i^p. Le résultat que nous allons dériver maintenant est l'existence d'un calcul récursif simple qui peut

être implémenté en propageant le signal d'erreur de la sortie vers l'entrée du réseau d'où le nom de l'algorithme.

Pour calculer δ_i^p, nous allons réécrire cette dérivée partielle comme étant le produit de deux facteurs : le premier reflète le changement de l'erreur comme une fonction de la sortie du neurone et le second exprime le changement de la sortie du neurone en fonction de ses entrées. Nous avons donc :

$$\delta_i^p = -\frac{\partial E^p}{\partial i_i^p} = -\frac{\partial E^p}{\partial a_i^p}\frac{\partial a_i^p}{\partial i_i^p} \tag{3.17}$$

Calculons le second facteur. De l'équation (3.9), il résulte

$$\frac{\partial a_i^p}{\partial i_i^p} = \Im'(i_i^p) \tag{3.18}$$

Il s'agit simplement de la dérivée de la fonction d'activation $\Im$ pour le neurone i. Pour calculer le premier facteur de l'équation (3.17), nous devons envisager deux cas. Premièrement, supposons que le neurone *i* soit un neurone de sortie. Il résulte de la définition de E^p que

$$\frac{\partial E^p}{\partial a_i^p} = -(d_i^p - a_i^p) \tag{3.19}$$

La substitution de ce résultat et de l'équation (3.18) dans l'équation (3.17) donne

$$\delta_i^p = (d_i^p - a_i^p)\Im'(i_i^p) \tag{3.20}$$

pour chaque neurone *i* de sortie. Secundo, si le neurone *i* n'est pas un neurone de sortie, nous ne connaissons pas la contribution du neurone sur l'erreur du réseau. Néanmoins, la mesure de l'erreur peut être écrite comme une fonction des entrées du réseau sur une couche cachée vers la couche de sortie; $E^p = (i_1^p, i_2^p, ..., i_i^p, ...)$ et nous utilisons la règle de chaînage pour écrire

$$\frac{\partial E^p}{\partial a_i^p} = \sum_{h=1}^{N_o}\frac{\partial E^p}{\partial i_h^p}\frac{\partial i_h^p}{\partial a_i^p} = \sum_{h=1}^{N_o}\frac{\partial E^p}{\partial i_h^p}\frac{\partial}{\partial a_i^p}\sum_{k=1}^{N_h} w_{hk}a_k^p = \sum_{h=1}^{N_o}\frac{\partial E^p}{\partial i_h^p}w_{hi} = -\sum_{h=1}^{N_o}\delta_h^p w_{hi} \tag{3.21}$$

avec N_h le nombre de neurones dans la couche cachée.

En substituant ce résultat dans l'équation (3.17), nous obtenons

$$\partial_i^p = \Im'(i_i^p)\sum_{h=1}^{N_o}\delta_h^p w_{hi} \tag{3.22}$$

Les équations (3.20) et (3.22) donnent une procédure récursive pour calculer les δ pour tous les neurones du réseau qui sont utilisés pour calculer les changements des poids tel que décrit dans l'équation (3.16). Cette procédure constitue la règle du delta généralisée pour les réseaux non bouclés aux fonctions d'activation non linéaire.

3.6.3 Utilisation de la rétropropagation du gradient

L'utilisation de la règle du delta généralisée implique deux phases (voir la figure 3.6) : Pendant la première phase les entrées sont propagées en avant à travers le réseau pour calculer les valeurs de sorties $a_i^{l'}$ de chaque neurone de sortie. La seconde phase implique un passage en arrière dans le réseau afin de rétropropager le signal d'erreur vers chaque neurone et de pouvoir ajuster les poids.

L'ajustement des poids avec une fonction d'activation sigmoïde peut être résumé en trois équations en se basant sur les résultats de la section précédente.

Le poids d'une connexion est ajusté proportionnellement au produit d'un signal d'erreur δ_i sur le neurone i recevant l'entrée et à la sortie du neurone j produisant le signal sur la connexion.

$$\Delta_p w_{ij} = \gamma \delta_i^p a_j^p \tag{3.23}$$

Si le neurone est un neurone de sortie alors le signal d'erreur est donné par

$$\delta_i^p = (d_i^p - a_i^p)\Im'(i_i^p) \tag{3.24}$$

Reprenons la fonction d'activation sigmoïde $\Im$

$$a_i^p = \Im(i_i^p) = \frac{1}{1+e^{-i_i^p}} \tag{3.25}$$

dans ce cas la dérivée est égale à

$$\begin{aligned} \Im'(i_i^p) &= \frac{\partial}{\partial i_i^p}\frac{1}{1+e^{-i_i^p}} \\ &= \frac{1}{(1+e^{-i_i^p})^2}(-e^{-i_i^p}) \\ &= \frac{1}{(1+e^{-i_i^p})}\frac{e^{-i_i^p}}{(1+e^{-i_i^p})} \\ &= a_i^p(1-a_i^p) \end{aligned} \tag{3.26}$$

et donc nous pouvons réécrire le signal d'erreur pour un neurone de sortie

$$\delta_i^p = (d_i^p - a_i^p)a_i^p(1-a_i^p) \tag{3.27}$$

Le signal d'erreur pour un neurone d'une couche cachée est déterminé récursivement en terme du signal d'erreur des neurones auxquels il est directement connecté et du poids de ces connexions. Ce qui donne avec une fonction d'activation sigmoïde

$$\delta_i^p = \Im'(i_i^p)\sum_{h=1}^{N_o}\delta_h^p w_{hi} = a_i^p(1-a_i^p)\sum_{h=1}^{N_o}\delta_h^p w_{hi} \tag{3.28}$$

Vitesse d'apprentissage et moment

La procédure d'apprentissage nécessite que le changement de poids soit proportionnel à $\partial E^p / \partial w$ Une véritable descente du gradient nécessite de procéder par des étapes infinitésimales. La constante de proportionnalité est la *vitesse d'apprentissage* γ. Pour des

raisons pratiques, nous choisissons la plus grande vitesse d'apprentissage possible sans engendrer des oscillations. Une manière d'éviter les oscillations avec un grand γ est de faire le changement de poids en incluant le changement de poids précédent et en incluant un terme de *moment.*

$$\Delta w_{ij}(t+1) = \gamma \delta_i^p a_i^p + \alpha \Delta w_{ij}(t) \qquad (3.29)$$

où t représente le temps, ici l'itération, et α une constante qui détermine l'effet du changement de poids précédent.

Le terme de moment permet de stabiliser les oscillations lorsque l'on augmente la vitesse d'apprentissage. Si aucun terme de moment n'est utilisé, nous avons une lente descente avant d'atteindre le minimum avec une petite vitesse d'apprentissage, alors qu'avec une grande vitesse d'apprentissage nous n'atteignons pas le minimum à cause des oscillations. Si nous ajoutons un terme de moment, le minimum est atteint plus rapidement.

L'apprentissage après chaque exemple : théoriquement, l'algorithme de rétropropagation effectue une descente du gradient sur l'erreur totale seulement si l'on effectue la mise à jour des poids après le passage complet de tout l'ensemble d'apprentissage. En pratique, il est fréquent d'appliquer la règle d'apprentissage après chaque exemple, c'est-à-dire que lorsqu'on présente l'exemple p, E^p est calculée et les poids sont adaptés. Il existe des indications empiriques qui indiquent que l'on obtient une convergence plus rapide avec cette méthode. Il a donc également été montré que, si l'on présente les exemples plusieurs fois dans la même séquence, le réseau peut ne prendre en compte que les premiers exemples. Ce problème est surmonté en permutant l'ordre de présentation des exemples pendant l'entraînement.

3.6.4 Problème du surapprentissage (overfitting)

Lors de la phase d'apprentissage, l'objectif est de minimiser l'erreur. Pour atteindre cet objectif, nous itérons l'algorithme tant qu'il y a décroissance de l'erreur. Il en résulte un paramétrage (poids des connexions) optimal du réseau sur les données soumises à l'apprentissage. L'inconvénient est que le réseau devient alors moins précis dans sa phase prédictive.

Solution : Diviser les données de l'historique disponible en plusieurs parties.

Un *ensemble d'apprentissage*, servant directement à la phase d'apprentissage. Les données sont soumises en entrée, la propagation donne les valeurs associées, enfin l'erreur permet la rétropropagation de l'erreur. Il s'agit d'un processus itératif, qui si on le laisse fonctionner parvient à minimiser l'erreur sur l'ensemble d'apprentissage. Il faut donc un autre ensemble de donnée permettant d'évaluer ce que donnerait le réseau en phase de prévision. A cette fin nous aurons un deuxième ensemble de données appelé ensemble de test.

L'*ensemble de test* permet, au cours de l'apprentissage, d'évaluer l'état du réseau dans un état prédictif. A intervalle régulier de l'apprentissage, nous allons effectuer un test qui consiste à soumettre les données de l'ensemble de test au réseau, de faire la propagation avant dans le réseau et de déterminer l'erreur. Ici l'erreur ne sera pas rétropropagée, mais servira comme condition d'arrêt de l'apprentissage.

L'algorithme général d'apprentissage devient donc :

Présentons tous les exemples de l'ensemble d'apprentissage (dans un ordre aléatoire) et adaptons les poids.

Réalisons une prévision sur l'ensemble de test

Réitérer ces étapes tant que l'erreur sur l'ensemble de test décroît.

Cette méthode permet de stopper prématurément l'algorithme avant que la qualité prédictive du réseau ne s'estompe.

Enfin nous devons définir un troisième ensemble, appelé *ensemble de validation*, pour procéder à une comparaison objective du modèle sans attendre que la prévision ne se réalise et qui n'est dévoilé qu'au moment de l'évaluation finale du modèle.

J'ai divisé mes données historiques en prenant 70 % pour l'ensemble d'apprentissage, 20 % pour l'ensemble de test et les 10 % restant pour l'ensemble de validation.

3.7 Comparaison du vocabulaire des méthodes statistiques et neuronales.

Il y a un chevauchement important entre les domaines des statistiques et des réseaux de neurones. En effet quelque soit l'approche il faut préparer l'expérience, collecter les données, calculer l'estimateur, évaluer les performances, interpréter les résultats. Malgré ces similitudes, la terminologie dans la littérature entre les réseaux de neurones et les statistiques est fort différente. Le tableau 3.7 présente ces différences.

Réseaux de neurones	**Statistiques**
apprentissage, adaptation, autoorganisation	estimation
poids (synaptiques)	paramètres
Connaissance	valeur des paramètres
apprentissage supervisé	classification, régression
Classification	discrimination, classement
apprentissage non supervisé, autoassociation	estimation de densité, réduction de donnée
Clustering	classification
réseau de neurones	modèle
grand : 100 000 poids	grand : 50 paramètres
ensemble d'apprentissage	échantillon
grand : 50 000 exemples	grand : 200 cas
fonction erreur, fonction coût	critère d'estimation

Tableau 3.7 Glossaire réseaux de neurones / statistiques

Chapitre 4

Environnement et modélisation des variables

Le réseau a besoin d'un environnement à l'intérieur duquel il va fonctionner.

Les rôles de cet environnement sont de :

- réaliser une passerelle entre le système de base de données contenant l'historique des variables à prédire et le réseau de neurones;
- permettre à l'utilisateur de choisir une variable à prédire et d'autres variables explicatives;
- définir la durée de l'historique à prendre en compte pour chaque variable;
- définir l'horizon de prévision;
- sauver les choix précédents;
- définir la topologie du réseau de neurones;
- fournir au réseau les données nécessaires à l'apprentissage et à la prévision;
- sauver les résultats de la prévision.

Afin de pouvoir faire des prévisions, il y a lieu de suivre quelques étapes :

La première étape est de sélectionner la variable à prévoir ainsi que des variables supplémentaires que l'on pense explicatives. Nous allons sauver ces choix sous forme d'un *modèle de prévision.*

Un modèle de prévision est une description de tous les éléments nécessaires pour faire une prévision à l'aide des réseaux de neurones. A ce titre, le modèle va reprendre :

La liste des variables à inclure dans la prévision et pour chacune de ces variables :

- Le type : quantitative ou catégorielle
- La fréquence de disponibilité.
- Le nombre de jours historiques à prendre en compte.
- Pour la variable à prédire, l'horizon de prévision.
- Connaissance future de certaines variables : Les jours fériés, les périodes de vacances scolaires, ...
- Les paramètres de transformations pour le pré et post traitement.

Le modèle va également reprendre les paramètres du réseau :

- Nombre de couches
- Nombre de neurones par couches

- La vitesse d'apprentissage γ, et le terme de moment α

Je vais présenter un modèle théorique afin d'illustrer les différents types de données que je souhaite pouvoir inclure. A cette fin, définissons un exemple où nous utilisons les 30 derniers jours d'historique pour prévoir le lendemain soit un horizon à 1 jour.

Prenons la prévision d'un nombre d'appels téléphoniques (NA), ceux-ci étant connus par demi-heure. Nous aurons 48 valeurs par jour sur 30 jours soit 1440 valeurs en entrée.

Pour avoir la prévision de ce nombre d'appels téléphoniques à un horizon d'un jour nous aurons 48 valeurs en sortie.

Il peut sembler y avoir un facteur périodique, et donc nous pourrions donner le jour (J) à prévoir. Un encodage binaire simple sera utilisé. Nous aurons 7 valeurs en entrée.

Un facteur sociologique important est la présence de jours fermés (JF), combinaison des jours de congés scolaires et des jours fériés. Ce type de variable permet de montrer que nous pouvons associer une variable exogène aux valeurs passées mais également comme indicateur du type de jour à prévoir. Nous utiliserons des variables binaires, 30 + 1 valeurs

Nous pouvons également inclure une variable du type événement exceptionnel (Coupe du Monde de football, Jeux Olympiques, ...) que nous coderons sous forme de 30 variables binaires représentant la présence ou l'absence d'événement.

Avec cet exemple, nous aurons un réseau avec 1508 neurones en entrée et 48 valeurs en sortie correspondant au nombre d'appels prévus.

Cet exemple met en évidence différente caractéristique des données à traiter

Les variables n'ont pas toutes la même période (parfois 30 minutes, parfois un jour)

Certaines variables représentent des variables catégorielles (J ou JF).

Certaines variables peuvent être connue dans le futur (ex. : JF).

Nous allons donc étudier la manière de soumettre ces différentes variables au réseau de neurones.

4.1 Gestion de variables de périodes différentes.

Si nous nous remettons dans le contexte de l'exemple, nous aurons un modèle reprenant 30 jours d'historiques pour prévoir à un horizon de 1 jour

Une unité de temps pratique dans cet exemple est le jour. Cette unité de temps sera commune à toutes les variables.

Le nombre de données à considérer pour une variable par unité de temps est l'inverse de la période soit la fréquence. (Ex. : Une variable disponible toutes les 30 minutes donnera 48 données par jour).

Le nombre de neurones en entrée pour traiter une variable sera le nombre de jours de l'historique multiplié par la fréquence de cette variable.

Il faut noter que prendre une période de référence d'un jour est une hypothèse permettant de simplifier la compréhension par rapport aux exemples choisi, mais n'est en rien une hypothèse restrictive quant à l'applicabilité de la méthode à d'autres problèmes.

4.2 Gestion de variables catégorielles

Nous parlons ici de variables telles que :

Le jour de la semaine ayant 7 catégories (Lundi, …)

Le type de ciel (ensoleillé, couvert, pluvieux, neigeux, …)

La présence d'événements exceptionnels (Coupe du Monde de football, Jeux Olympiques, Attentat,…)

La méthode d'encodage utilisée dans ce mémoire est l'encodage binaire simple.

Bonnet a prouvé dans [Bonnet 1997a] que l'encodage binaire simple est optimal s'il n'existe pas de relation établie entre les catégories. Cet encodage consiste à coder la variable catégorielle en autant de variables binaires que de catégories possibles. Ces variables binaires représentent alors l'appartenance de la variable catégorielle à la catégorie correspondante

Notons que l'encodage binaire simple supporte les variables catégorielles simples ainsi que les variables catégorielles multivaluées

Nous devons considérer trois aspects de ces variables catégorielles :

- Combien de catégories sont possibles pour chaque variable ?
- Une variable peut-elle être dans plusieurs catégories simultanément ? Cela revient à savoir si elle est multivaluée ou non ?
- Doit-on tenir compte d'une relation pouvant exister entre les différentes catégories ?

Le nombre de catégories pour une variable est connu dans l'environnement à l'intérieur duquel le système va fonctionner. Ici nous pouvons noter que le nombre de catégories peut augmenter dans l'environnement pilotant le réseau de neurones (par exemple si un utilisateur ajoute un nouveau type d'événement exceptionnel). Cette remarque a des implications concernant la pérennité d'un réseau, ou d'un modèle de prévision en général. La prise en compte de nouvelles catégories (comme de nouvelles variables) nécessitera la création d'un nouveau réseau, faute de quoi ces nouvelles catégories seraient ignorées.

Doit-on tenir compte d'une relation pouvant exister entre les différentes catégories ? Nous aurions envie de répondre «oui». Malheureusement il est difficile d'avoir une représentation générale de toutes les relations possibles qu'il pourrait exister entre les différentes catégories. De plus dans le cadre d'une méthode générale nous voulons limiter les hypothèses restrictives. Cela veut-il dire que les relations que nous connaissons à priori entre les catégories demeureront inexploitées ? Eh bien non, car le réseau, lors de sa phase d'apprentissage pourra reconstituer une certaines formes de relations entre les catégories tels

que nous l'avons nous même appris. Mieux, le réseau peut trouver des relations entres les catégories qui nous étaient inconnues et qui lui seront utile dans le cadre précis de son objectif de prévision.

4.3 Utilisation de variables connues dans le futur.

La connaissance à priori de certaines variables dans le futur semble intéressante dans la mesure où elles sont en relation directe avec le résultat escompté de la prévision.

Prenons le cas d'appels téléphoniques et des jours de congé. Soumettons en entrée du réseau de neurones les appels téléphoniques, nous comparons alors les sorties du réseau avec appels téléphoniques correspondant à l'horizon de prévision afin de déterminer l'erreur et de poursuivre l'apprentissage. Si maintenant nous savons que les appels téléphoniques correspondant à l'horizon de prévision sont relatifs à un jour de congés, nous pourrions vouloir le signifier au réseau. Cette indication peut-être donnée au réseau en incluant les valeurs connues relatives à l'horizon de prévision en entrée du réseau.

Dans ce cas nous allons non seulement fournir en entrées les valeurs historiques de cette variable mais également les valeurs correspondantes à l'horizon de prévision.

Il faut noter que pour ces variables, nous devrons être capable, en phase prédictive, de fournir les valeurs correspondantes à l'horizon de prévision. L'environnement à l'intérieur duquel le système va fonctionner devra donc connaître ces valeurs avant de pouvoir effectuer une prévision.

Chapitre 5

Implémentation de la solution

5.1 L'environnement

La société Holydis dispose déjà d'un outil de prévision appelé SimFlow. SimFlow inclus un dictionnaire comprenant la description de toutes les variables (dans le sens séries temporelles) du système. Nous avons développé un nouveau module dans SimFlow afin de créer les modèles de prévisions tels que nous l'avons envisagé au chapitre précédant et nous avons implémenté le réseau de neurones dans un programme distinct dont on peut voir le principe sur la figure 5.1. D'un point de vue langage de développement, SimFlow est réalisé avec Windev qui est un environnement de développement de quatrième génération (4GL) de la société française PCSOFT, alors que le réseau de neurone (bpn) est développé en langage C.

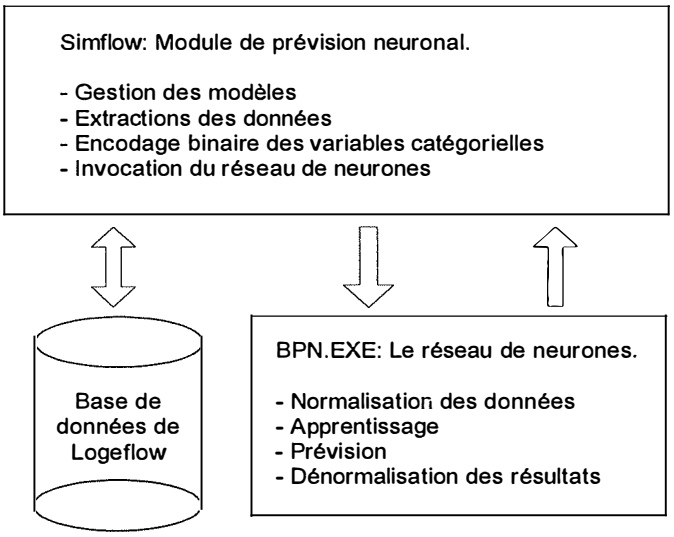

Figure 5.1 : Schéma de la solution développée. On peut y voir le module de prévision développé au sein du logiciel SimFlow, ainsi que le nouveau module BPN comprenant le réseau de neurones.

Le nouveau module de SimFlow, dont l'interface est présentée sur la figure 5.2, permet à l'utilisateur de créer son modèle de prévision en choisissant la variable à prédire, l'horizon de prévision ainsi que d'autres variables qu'il pense explicatives.

Pour chacune des variables sélectionnées, l'interface va lui indiquer si elle est catégorielle, si la variable est connue dans le futur et sa période (en minutes).

L'utilisateur devra ensuite, pour chacune des variables, indiquer le nombre de jours de l'historique à prendre en compte. De plus, pour les variables connues dans le futur, il a la

possibilité de sélectionner un nombre de jours dans l'horizon de prévision à inclure en entrée du réseau.

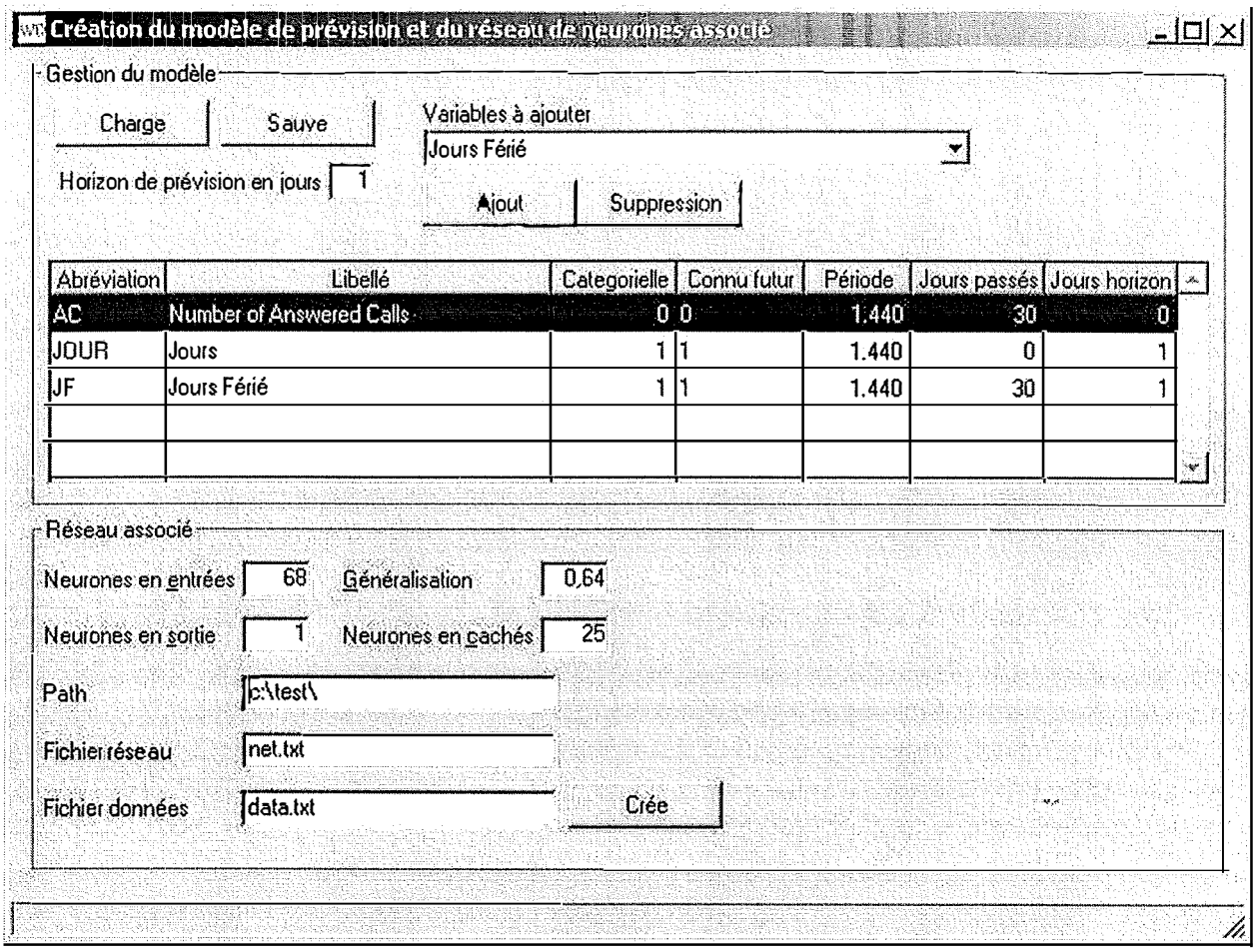

Figure 5.2 : Interface du nouveau module de prévision au sein de SimFlow. Cette interface permet de sélectionner les variables participantes au modèle de prévision et d'extraire les données pour le réseau de neurones associés. Il faut noter que la première variable de la liste est implicitement la variable à prédire.

Durant cette étape de sélection de variables, l'utilisateur peut voir la configuration du réseau associé à sa sélection. Dans l'exemple illustré sur la figure 5.2, on peut voir que le réseau a 68 neurones dans sa couche. Ce nombre de neurones est déterminé comme suit :

30 neurones pour recevoir les données de la variables 'Number of Answered Calls'.

7 neurones pour modéliser le jour de la semaine du premier jour de l'horizon. Il faut noter que cette variable à subit un encodage binaire.

31 neurones pour savoir si les 30 jours de l'historique et le jour de prévision sont des jours fériés. L'encodage binaire de cette variable indique ou non la présence du férié.

Le nombre de neurones de sortie est déterminé en multipliant la durée de l'horizon de prévision par la fréquence de la variable à prédire

L'utilisateur a enfin la possibilité, de sélectionner le nombre de neurones dans la couche cachée. Ce nombre de neurones dans la couche cachée Nh est généralement compris entre le nombre de neurones dans la couche d'entrée Ni et le nombre de neurones dans la couche de sortie No. Plus Nh est proche de Ni, mois le réseau pourra généraliser. A l'inverse, plus Nh est proche de No, plus le réseau aura tendance à généraliser. J'ai donc défini un paramètre de généralisation g permettant de calculer Nh comme suit :

$$Hh = Ni.(1-g) + No.g \qquad (5.1)$$

Avec l'équation (5.1), lorsque g vaut 0, $Nh = Ni$ ce qui reflète l'absence de généralisation alors que lorsque g vaut 1, $Nh = No$ ce qui reflète la plus grande généralisation possible.

Nous avons également inclus une possibilité de sauvegarde et de récupération des modèles créés.

Les données nécessaires à l'execution du réseau de neurones sont fournies dans des fichiers créés par SimFlow. Enfin le programme bpn utilise ces fichiers pour effectuer la phase d'apprentissage et ensuite la prévision.

5.2 Traitements des variables

Le modèle étant défini, nous allons pouvoir traiter les données afin d'effectuer une prévision. SimFlow crée un premier fichier comprenant la structure du réseau c'est à dire le nombre de neurones dans chacune des couches. Un second fichier décrira le modèle et comportera des références vers autant de fichiers que de variables du modèle. Ces derniers fichiers comprennent les données liées aux variables.

Nous utiliserons les données les plus récentes disponibles pour chaque variable. Afin d'avoir des données synchronisées entre les différentes variables, nous déterminons *la période du modèle* comme étant la période correspondant à l'intersection des périodes disponibles pour chacune des variables.

L'étape suivante consiste à extraire les données des variables du modèle pour la période du modèle et leurs faire subir un prétraitement en fonction de leur type.

Les variables catégorielles sont transformées avec l'encodage binaire simple en autant de variables binaires que de catégories.

Ces premiers traitements sont réalisés au sein de SimFlow et ont pour résultats les fichiers nécessaires à l'exécution du module bpn. Une documentation du format de ces fichiers est fournie en annexe B.

La normalisation des données dans l'intervalle [0; 1]; et la détermination parmi les données disponibles, d'un ensemble de données d'apprentissage, de test et de validation sont réalisés dans le module du réseau de neurone écrit en langage C. Le lecteur désireux d'en savoir plus pourra se reporter au code source en annexe A.

5.3 Traitements du résultat

Le réseau neuronal produit des résultats sur base des données normalisées en entrée, il convient donc de faire la transformation inverse en sortie du réseau neuronal afin que les résultats soient comparables aux données originales.

C'est alors que l'on effectue l'évaluation générale du modèle. Cette évaluation consiste à comparer les données que le réseau fournit en phase prédictive avec les données de l'ensemble de validation, ensemble de données qui avait été « cachée » pendant l'apprentissage.

Chapitre 6

Méthodes d'évaluation des résultats

Le problème est d'évaluer les prévisions réalisées sur des séries différentes par des méthodes différentes.

Nous devons donc avoir une méthode d'évaluation indépendante du problème (insensible à la longueur de la série, à ses moments d'ordre 1 et 2) et indépendante du modèle (insensible aux hypothèses du modèle et aux paramètres du modèle). Voici quelques mesures de l'erreur qui peuvent être appliquées aux données de l'ensemble de validation dans la phase d'évaluation. $x(t)$ représente la valeur attendue à l'instant t et $\hat{x}(t)$ la valeur fournie par la prévision à l'instant t. N représente le nombre de valeurs dans l'ensemble de validation.

6.1 Erreur quadratique

Appelée Square Error en anglais, elle est insensible aux paramètres et à la moyenne (moment d'ordre 1)

$$SE = \sum_{t=1}^{N} \left(x(t) - \hat{x}(t)\right)^2$$

6.2 Erreur quadratique moyenne

Ou Mean Square Error (MSE) en anglais

SE + Insensible à la longueur de la série.

$$MSE = \frac{SE}{N}$$

6.3 Erreur quadratique moyenne normalisée

Une mesure usuelle de l'erreur pour évaluer et comparer la puissance de prédiction d'un modèle est d'utiliser l'erreur quadratique moyenne normalisée, qui donne en anglais *Normalized Mean Square Error* (NMSE) ou encore parfois AVR pour *Average Relative Variance*.

NMSE = MSE + insensibilité au moment d'ordre 2.

$$NMSE = \frac{MSE}{\sigma^2} = \frac{\sum_{t=1}^{N}\left(x(t) - \hat{x}(t)\right)^2}{\sum_{t=1}^{N}\left(x(t) - \bar{x}(t)\right)^2}$$

où σ^2 représente la variance sur x dans l'ensemble de validation et $\bar{x}$ la valeur moyenne de x, également dans l'ensemble de validation.

Si nous utilisons la moyenne $\bar{x}$ comme valeur pour la prévision, nous aurons NMSE = 1. La dépendance (linéaire) R^2 entre les paires des valeurs désirées et des valeurs prédites est donnée par la relation $R^2 = 1 - NMSE$. Ceci implique que plus NMSE est proche de 0, plus la dépendance R^2 est proche de 1 et en d'autre terme que la prévision est bonne. Par ailleurs si NMSE est supérieur à 1, alors cela indique que le prédicteur utilisé donne de moins bon résultat que l'utilisation de la moyenne $\bar{x}$ comme valeur prédictive.

De ces trois méthodes d'évaluation de l'erreur, nous retiendrons la troisième car elle est largement répandue pour ses qualités comparatives.

Chapitre 7

Création et test de différents modèles

Les données utilisées pour l'évaluation sont une série comprenant le *nombre d'appels téléphoniques* entrants d'un centre d'appel ouvert en permanence (24 heures sur 24, 365 jours par an). Je dispose de ces données pour la période du 1/9/1998 au 31/7/2001 par intervalle de 30 minutes. Les valeurs de la série représentent donc les appels entrants par 30 minutes. Il s'agit de données quantitatives entières.

Ce nombre d'appels semble fluctuer en fonction du jour de la semaine et de la présence ou non de jours fériés. Il m'a donc semblé intéressant de les inclure afin d'évaluer leurs apports.

Les variables incluses dans les quatre modèles sont :

Modèle *A* : La variable à prédire, à savoir, le nombre d'appels téléphoniques.

Modèle *B* : Le nombre d'appels téléphoniques et les jours fériés.

Modèle *C* : Le nombre d'appels téléphoniques et le jour de la semaine.

Modèle *D* : Le nombre d'appels téléphoniques, les jours fériés et le jour de la semaine.

Ces quatre modèles ont étés créer, sur une même base pour la comparaison. On utilise 30 jours pour prévoir le $31^{ième}$ c'est-à-dire que l'on travail à un horizon d'un jour.

Lors de mes premiers essais, je suis arrivé à un temps de calcul pour la phase d'apprentissage de 8 jours sur un Pentium IV à 1600Mhz. J'aborde ce problème au chapitre 8.

Afin de poursuivre l'évaluation, j'ai dû modifier la série comprenant le *nombre d'appels téléphoniques*. J'ai considéré la somme des appels par jour au lieu des données par intervalle de 30 minutes. Ceci réduit considérablement la taille du réseau et permet de calculer les résultats en quelques minutes.

La période du 1/9/1998 au 31/7/2001 comprend 1035 jours. De ces 1035 jours, 724 seront utilisés comme ensemble d'apprentissage, 207 pour l'ensemble de test et 104 pour l'ensemble de validation.

Suite au prétraitement des données, les différents modèles nous donnent les réseaux suivants :

Modèle *A* : La couche d'entrée comporte 30 neurones.

Modèle *B* : La couche d'entrée comporte 61 neurones, 30 pour traiter les appels téléphoniques, 31 pour indiquer la présence des jours fériés (à noter que nous somme dans le cas d'une variable connue dans le futur)

Modèle *C* : La couche d'entrée comporte 37 neurones, 30 pour traiter les appels téléphoniques, 7 pour indiquer le jour de la semaine du jour à prédire.

Modèle *D* : La couche d'entrée comporte 68 neurones, 30 pour traiter les appels téléphoniques, 31 pour indiquer la présence des jours fériés et 7 pour indiquer le jour de la semaine du jour à prédire.

Il faut noter que et la couche de sortie comporte un neurone dans tous les modèles. C'est la valeur de ce neurone de sortie qui est comparée avec la valeur souhaitée.

Différentes architectures ont été testée en faisant varier le nombre de neurones dans la couche cachée. On trouvera les résultats dans le tableau 7.1

Modèle	# neurones en entrée	# neurones cachés				
		10	15	20	25	30
A	30	0.5511	0.5338	0.5283	0.5565	0.5180
B	61	0.5811	0.5745	0.5893	0.6033	0.5738
C	37	0.5490	0.5331	0.5091	0.5180	0.5116
D	68	0.5819	0.5712	0.5448	0.5429	0.5461

Tableau 7.1 Résultat comparatif des modèles sur l'ensemble de validation. Les valeurs représentent l'erreur quadratique moyenne normalisée (NMSE).

Chapitre 8

Analyses

8.1 Analyse des variables catégorielles sur la qualité des résultats.

Le tableau 7.1 nous révèle que le modèle B est toujours plus mauvais que le modèle A en terme d'erreur quadratique moyenne normalisée. Nous pouvons donc en conclure que contrairement à notre hypothèse, la présence des jours fériés n'améliore pas la qualité de la prévision.

Le modèle C est toujours meilleur que le modèle A en terme d'erreur quadratique moyenne normalisée. Nous pouvons donc en conclure que l'indication du jour à prévoir est bien une variable explicative, qui de plus est exploitée par le réseau de neurones.

Le modèle D à un résultat compris entre ceux du modèle A et du modèle C. Il semble donc que l'influence du jour à prévoir soit toujours bien présente mais cela montre également les perturbations introduites par l'indication des jours fériés. Seul le réseau avec 25 neurones parvient à faire mieux que le modèle de base. Il faut donc rester vigilant à l'introduction de nouvelles variables.

8.2 Analyse de l'architecture sur la qualité des résultats.

Les différents modèles ont leurs meilleurs résultats avec un nombre de neurones dans la couche cachée N_h supérieure ou égale à 20. Il semble donc que l'information nécessaire ne puisse être résumée en moins de 20 paramètres. Nous voyons également que les deux plus gros modèles en nombre de neurones améliorent leurs résultats avec un N_h supérieur. Nous pouvons donc constater que l'information supplémentaire fournie au réseau n'est correctement exploitée qu'en augmentant la taille du réseau.

8.3 Analyse de la méthode

Malgré le succès apparent de l'algorithme de rétropropagation du gradient, il y a certaines choses qui peuvent perturber l'algorithme, ce qui est un obstacle à son usage universel.

Le plus troublant est la durée du processus d'apprentissage. Cette durée peut résulter d'une vitesse d'apprentissage γ et d'un facteur du moment α inadapté. Différentes méthodes d'adaptation de ces constantes sont discutées dans [Rumelhardt 1986] mais n'ont pas été exploitées ici. Ce défaut de paramétrage induit généralement deux types de problèmes : Une paralysie de l'apprentissage et le fait d'aboutir dans un minimum local.

La paralysie du réseau résulte généralement d'un ajustement d'un poids à une très grande valeur. L'entrée totale d'un neurone peut alors atteindre une très grande valeur positive (ou négative). Du fait de l'utilisation de la fonction sigmoïde comme fonction d'activation, nous obtenons une valeur de sortie proche de 1 (ou proche de 0). En regardant les équations (3.27) et (3.28) on constate que l'ajustement des poids étant proportionnel à

$a_i^p(1-a_i^p)$, expression dont le résultat est proche de 0 dans ces cas limites. Il en résulte un arrêt virtuel de la phase d'apprentissage.

Un second problème peut-être d'aboutir dans un minimum local lorsque la surface d'erreur est complexe. Il semble exister des méthodes stochastiques proche du recuit simulé pour contourner ce problème. Une autre voie, est d'ajouter des neurones dans les couches cachées, ce qui a pour effet d'augmenter la dimensionnalité de la surface d'erreur et minimise les chances de tomber dans un minimum local. Cette dernière approche montre une limite supérieure au nombre de neurones cachés car on semble à nouveau tomber dans des minima locaux.

Concernant la rapidité de l'implémentation, il faut noter que le code développer a été réalisé avec un souci de lisibilité afin de pouvoir facilement le modifier et le déboguer. Il est donc certainement possible de l'optimiser.

J'aimerai faire une remarque concernant la création de modèle, à propos de l'impact de l'ajout de nouvelles catégories pour les variables catégorielles. Ces modifications ont des implications concernant la pérennité d'un réseau, ou d'un modèle de prévision en général. La prise en compte de nouvelles catégories (comme de nouvelles variables) nécessitera la création d'un nouveau réseau, faute de quoi ces nouvelles catégories seraient ignorées.

Un autre point concerne l'utilité de l'utilisation de certaines variables. Comme nous l'avons vu au point 7.2., certaines variables peuvent dégrader la qualité des résultats. Un simple test pour valider l'utilisation d'une variable supplémentaire est de tester un modèle comprenant la variable de base et cette variable supplémentaire. Si les résultats s'améliorent alors nous pouvons conserver cette variable supplémentaire pour l'élaboration de modèles plus complexes et peut-être globalement meilleurs.

Chapitre 9

Conclusions et perspectives

Ce travail m'a permis d'approfondir l'étude des réseaux de neurones et des méthodes de prévisions en les situant dans leurs contextes. C'est ainsi que j'ai pu me plonger dans le monde des séries temporelles et voir l'approche statistique descriptive et prédictive. Ensuite en étudiant les réseaux de neurones, j'ai compris les mécanismes d'apprentissage et malheureusement plus tard, les limites de mon implémentation.

Ce travail m'a également permis de lire un très grand nombre d'articles tous plus intéressants les uns que les autres. Les nombreuses explications, méthodes et pistes qu'ils présentaient, éveillaient une grande curiosité mais m'ont laisser devant un grand sentiment de frustration faute de pouvoir les tester. En effet là où j'ai mis beaucoup de temps à développer du code, j'aurai mieux fait d'essayer de trouver des outils de simulations existants afin de valider les différentes pistes proposées. Ce qui m'aurait certainement mené bien plus loin. Il s'agit là probablement d'un reliquat de mon Graduat en Informatique.

En ce qui concerne les perspectives, je pense qu'il faudrait lever les limitations du système existant, à savoir :

La mise en place d'une heuristique pour obtenir une bonne architecture du réseau en terme de nombre de couches, nombres de neurones. Des pistes sont disponibles dans [Kröse, 1996, p.40] et dans [Thiria et all., 1997, p.60]. Cet aspect me semble important pour augmenter la qualité des résultats et pour ne pas surdimensionner le réseau, ce qui a pour conséquence de ne plus pouvoir le calculer en des temps raisonnables. Ceci nous ouvrira également des portes pour traiter des modèles plus complexes. Dans cette même optique, il faudrait également une méthode permettant de déterminer les paramètres de vitesse d'apprentissage et du terme de moment.

Par ailleurs, et quelque soit le système, il serait intéressant de mettre en place un système de suivi de la prévision en terme de dérive par rapport aux données réelles et qui, pourquoi pas, proposerait le recalcul du modèle sur des données plus récentes.

Ce module de prévision fera certainement l'objet de recherche plus approfondie en vue de le rendre opérationnel en clientèle. L'approche d'Holydis est de collaborer avec des clients « pilotes » afin d'adapter le logiciel aux situations réelles.

Bibliographie

Bonnet D., Perrault V. et Grumbach A., *Vers une approche modulaire de la prévision*, ENST, Paris, France, 1997.

Bonnet D., Perrault V. and Grumbach A., *Using Symbolic Data to Improve Connexionnist Forecasting : A Methodology*, ENST technical report 97C002, Paris, France, 1997.

Coutrot B. et Droesbeke F., *Les méthodes de prévisions*, coll. "Que sais-je?" n°2157, PUF, Paris, France, 1984

Gallinari P., *Méthodes neuronales de discrimination*, dans : Thiria S., Lechevalier Y.; Gascuel O. et Canu S., *Statistique et méthodes neuronales*, Dunod, Paris, 1997

Gonzalez R., Woods R., *Digital Image Processing*, Addison-Wesley, 1993

Hornik K., Stinchcombe M. and White H., Multilayer *feedforward networks are universal approximators*, Neural Networks, Volume 2, pp359-366, 1989.

Kohonen T., *Self organizing maps*, http://www.cis.hut.fi/research/som-research/index.shtml (du 24-1-2001) accédé le 22-4-2002

Kröse B. and van der Smagt P., *An introduction to neural netwoks*, University of Amsterdam, 8th. edition, 1996.

Rumelhart D.E. and McClelland J.L., Parallel Distributed Processing: Exploration in the Microstructure of Cognition, Volume 1, The MIT Press, 1986

Tauritz D., *The Adaptive Resonance Theory (ART) clearinghouse,*http://www.liacs.nl/art (du 17-9-2001) accédé le 22-4-2002

Thiria S., Lechevalier Y., Gascuel O. et Canu, S., *Statistique et méthodes neuronales*, Dunod, Paris, 1997

Annexes

Annexe A : Code source de l'implémentation en C du réseau de neurones

```
/*****************************************************************************
  BPN.C

  Perceptron Multi Couches (PMC)

  Algorithme de rétropropagation du gradiant
  avec biais et Moment d'inertie.

  Nicolas Simon

  Reference:    D.E. Rumelhart, G.E. Hinton, R.J. Williams
                Learning Internal Representations by Error Propagation
                in:
                D.E. Rumelhart, J.L. McClelland (Eds.)
                Parallel Distributed Processing, Volume 1
                MIT Press, Cambridge, MA, pp. 318-362, 1986
 *****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <string.h>
#include <assert.h>
#include <time.h>


/* Définition des types de données */
typedef int           BOOL;
typedef int           INT;
typedef double        REAL;

typedef struct {                  /* A LAYER OF A NET:                     */
        INT           Units;        /* - number of units in this layer       */
        REAL*         Output;       /* - output of ith unit                  */
        REAL*         Error;        /* - error term of ith unit              */
        REAL**        Weight;       /* - connection weights to ith unit      */
        REAL**        WeightSave;   /* - saved weights for stopped training  */
        REAL**        dWeight;      /* - last weight deltas for momentum     */
} LAYER;
```

```
typedef struct {                    /* A NET:                                  */
        INT           NumLayers;      /* - number of layers in this net        */
        LAYER**       Layer;          /* - layers of this net                  */
        LAYER*        InputLayer;     /* - input layer                         */
        LAYER*        OutputLayer;    /* - output layer                        */
        REAL          Alpha;          /* - momentum factor                     */
        REAL          Eta;            /* - learning rate                       */
        REAL          Error;          /* - total net error                     */
        INT           N;              /* - number of units in InputLayer       */
        INT           M;              /* - number of units in OutputLayer      */
} NET;

typedef struct {                  /* A VARIABLE:                               */
        INT     ValuesPerPattern; /* - Number of values per pattern, ie. per day */
        INT     Input;            /* - Number of values used in input          */
        INT     Output;           /* - Number of values used in input          */
        INT     Offset;           /* - Indice correspondant a la première valeur à
prédire */
        REAL    Mean;             /* - Moyenne                                 */
        REAL    a, b;             /* - Coefficient de normalisation            */
        INT     NumData;
        REAL*   Data;
} VARIABLE;

typedef struct {                  /* A MODEL:                                  */
        INT       NumVar;         /* - number of variables                     */
        VARIABLE* Var;            /* - list of variable                        */
} MODELE;


/* Quelques constantes */

#define FALSE         0
#define TRUE          !FALSE

#define MAX_REAL      FLT_MAX
#define MIN_REAL      FLT_MIN
#define HI            (REAL)0.9
#define LO            (REAL)0.1

#define MIN(x,y)      ((x)<(y) ? (x) : (y))
#define MAX(x,y)      ((x)>(y) ? (x) : (y))

#define BIAS          1.0
#define sqr(x)        ((x)*(x))

/* Variables globales */
MODELE  Modele;
NET     Net;
```

```
/* These are for dividing data set into subset with specific purpose */
INT     TRAIN_LWB, TRAIN_UPB, TRAIN_PATTERNS;
INT     TEST_LWB,  TEST_UPB,  TEST_PATTERNS;
INT     EVAL_LWB,  EVAL_UPB,  EVAL_PATTERNS;

/* Array that store values for current iteration, global for optimisation */
REAL*   Input;
REAL*   Output;
REAL*   Target;

REAL    Mean;
REAL    TrainError;
REAL    TrainErrorPredictingMean;
REAL    TestError;
REAL    TestErrorPredictingMean;

FILE*   f;
time_t  ltime;

/******************************************************************************
       R A N D O M S   D R A W N   F R O M   D I S T R I B U T I O N S
 ******************************************************************************/

void InitializeRandoms(void)
{
  srand(4711);
}

INT RandomEqualINT(INT Low, INT High)
{
  return rand() % (High-Low+1) + Low;
}

REAL RandomEqualREAL(REAL Low, REAL High)
{
  return ((REAL) rand() / RAND_MAX) * (High-Low) + Low;
}

void RandomWeights()
{
  INT l,i,j;

  for (l=1; l<Net.NumLayers; l++) {
    for (i=1; i<=Net.Layer[l]->Units; i++) {
      for (j=0; j<=Net.Layer[l-1]->Units; j++) {
        Net.Layer[l]->Weight[i][j] = RandomEqualREAL(-0.5, 0.5);
        Net.Layer[l]->dWeight[i][j] = 0.0;
      }
    }
```

```
  }
}

/*****************************************************************************
                APPLICATION-SPECIFIC CODE
 *****************************************************************************/

void GenerateNetwork( INT NumLayers, INT Units[] )
{
  INT l,i;

  Net.NumLayers = NumLayers;
  Net.Layer = (LAYER**) calloc(Net.NumLayers, sizeof(LAYER*));
  for (l=0; l<Net.NumLayers; l++) {
    Net.Layer[l]             = (LAYER*) malloc(sizeof(LAYER));
    Net.Layer[l]->Units      = Units[l];
    Net.Layer[l]->Output     = (REAL*)  calloc(Net.Layer[l]->Units+1, sizeof(REAL));
    Net.Layer[l]->Error      = (REAL*)  calloc(Net.Layer[l]->Units+1, sizeof(REAL));
    Net.Layer[l]->Weight     = (REAL**) calloc(Net.Layer[l]->Units+1, sizeof(REAL*));
    Net.Layer[l]->WeightSave = (REAL**) calloc(Net.Layer[l]->Units+1, sizeof(REAL*));
    Net.Layer[l]->dWeight    = (REAL**) calloc(Net.Layer[l]->Units+1, sizeof(REAL*));
    Net.Layer[l]->Output[0]  = BIAS;

    if (l != 0) {
      for (i=1; i<=Net.Layer[l]->Units; i++) {
        Net.Layer[l]->Weight[i]     = (REAL*) calloc(Net.Layer[l-1]->Units+1,
sizeof(REAL));
        Net.Layer[l]->WeightSave[i] = (REAL*) calloc(Net.Layer[l-1]->Units+1,
sizeof(REAL));
        Net.Layer[l]->dWeight[i]    = (REAL*) calloc(Net.Layer[l-1]->Units+1,
sizeof(REAL));
      }
    }
  }
  Net.InputLayer  = Net.Layer[0];
  Net.OutputLayer = Net.Layer[Net.NumLayers - 1];

  Net.N = Net.InputLayer->Units;
  Net.M = Net.OutputLayer->Units;
}


int LoadNet(char *FileName)
{
  FILE *hf;
  INT  Layers;
  INT  Units[5];
  INT  i;

  hf = fopen(FileName,"rt");
  if (hf == NULL) { return 0; }
```

```
  fscanf(hf,"%d\n",&Layers);
  for (i=0;i<Layers;i++)
    fscanf(hf,"%d\n",&Units[i]);
  fclose(hf);

  GenerateNetwork( Layers, Units );
  RandomWeights();

  Net.Alpha = (REAL) 0.9;
  Net.Eta   = (REAL) 0.05;

  return 0;
}

int SaveNet(char *FileName)
{
  FILE *hf;
  INT l,i,j;

  hf = fopen(FileName,"wt");
  if (hf == NULL) { return 0; }

  fprintf(hf,"%d\n",Net.NumLayers);
  for (l=0; l<Net.NumLayers; l++) {
    fprintf(hf,"%d\n",Net.Layer[l]->Units);
  }
  for (l=1; l<Net.NumLayers; l++) {
    for (i=1; i<=Net.Layer[l]->Units; i++) {
      for (j=0; j<=Net.Layer[l-1]->Units; j++) {
        fprintf(hf,"%f\n",Net.Layer[l]->Weight[i][j]);
      }
    }
  }
  fclose(hf);
  return 1;
}

int LoadData(char *FileName)
{
  FILE *hf,*hd;
  char DataFileName[512];
  INT  i,j;
  char *StrPtr;
  long Length;
  int iIn, iOut;
  long nPat, NumPat;
  REAL Min, Max;
  float InputValue;
  FILE *fd;
```

```
        hf = fopen(FileName,"rt");
        if (hf == NULL) { return -1; }

        fscanf(hf,"%d\n",&Modele.NumVar);

        Modele.Var = (VARIABLE*) malloc( Modele.NumVar * sizeof(VARIABLE));
        if (Modele.Var == NULL) {
          fclose(hf);
          return -2;
        }

        iIn = iOut = 0;

        for (i=0;i<Modele.NumVar;i++) {
            fscanf(hf,"%d,%d,%d,", &Modele.Var[i].ValuesPerPattern,
                                   &Modele.Var[i].Input,
                                   &Modele.Var[i].Output);

            fgets(DataFileName,sizeof(DataFileName),hf);

            // Remove trailing '\n'
            StrPtr = DataFileName + strlen( DataFileName ) - 1; /* Last char */
            if (*StrPtr == '\n') *StrPtr = '\0';

            hd = fopen(DataFileName,"rb");
            if (hd == NULL) { return -3; }

            fseek(hd, 0L, SEEK_END);
            Length = ftell(hd);
            fseek(hd, 0L, SEEK_SET);

            Modele.Var[i].NumData = (INT)(Length / sizeof(float)); /* External data in float
format */
            Modele.Var[i].Data = malloc( (size_t)Modele.Var[i].NumData * sizeof(REAL) );
            if (Modele.Var[i].Data == NULL) {
                fclose(hf);
                fclose(hd);
                return -4;
            }

            for (j=0; j<Modele.Var[i].NumData; j++) {
              if (fread( &InputValue, sizeof(float), 1, hd) != 1) {
                fclose(hf);
                fclose(hd);
                return -5;
             }
              Modele.Var[i].Data[j] = (REAL)InputValue;
            }
```

```
        /*
              if (fread( Modele.Var[i].Data, (size_t)Length, 1, hd) != 1) {
                  fclose(hf);
                  fclose(hd);
                  return -5;
              }
        */

              fclose(hd);

              /* Dump  */
              fd = fopen("DATADUP.TXT","w");
              if (fd != (FILE*)NULL) {
               for (j=0;j<Modele.Var[i].NumData;j++) {
                      fprintf(fd,"%04d %f\n",j,Modele.Var[i].Data[j] );
                }
                fclose(fd);
              }


              /* Normalisation */
              Min = MAX_REAL;
              Max = MIN_REAL;
              for (j=0; j<Modele.Var[i].NumData; j++) {
                Min = MIN(Min, Modele.Var[i].Data[j]);
                Max = MAX(Max, Modele.Var[i].Data[j]);
              }
              /* Scale [ Min ; Max ] to [ LO ; HI ] */
              Modele.Var[i].a = (HI-LO) / (Max-Min);
              Modele.Var[i].b = LO - (Min * Modele.Var[i].a);
              Modele.Var[i].Mean = 0.0;
              for (j=0; j<Modele.Var[i].NumData; j++) {
                        Modele.Var[i].Data[j] = Modele.Var[i].Data[j] * Modele.Var[i].a +
Modele.Var[i].b;
                        Modele.Var[i].Mean  += Modele.Var[i].Data[j];
              }
              Modele.Var[i].Mean /= Modele.Var[i].NumData;

              iIn += Modele.Var[i].Input;
              iOut += Modele.Var[i].Output;

          }
          fclose(hf);

          /* Check if Data match number of neurons in input layer */
          if (iIn != Net.N) {
             return -6;
          }
          /* Check if Data match number of neurons in output layer */
          if (iOut != Net.M) {
```

```
      return -7;
    }

    NumPat = 32000;
    for (i=0;i<Modele.NumVar;i++) {
      Modele.Var[i].Offset = (int)ceil( Modele.Var[i].Input /
Modele.Var[i].ValuesPerPattern ) * Modele.Var[i].ValuesPerPattern;

      nPat = (INT)floor(Modele.Var[i].NumData / Modele.Var[i].ValuesPerPattern)   //
Pattern available
           - (INT)ceil( Modele.Var[i].Input  / Modele.Var[i].ValuesPerPattern )   //
Pattern used for input
           - (INT)ceil( Modele.Var[i].Output / Modele.Var[i].ValuesPerPattern )   //
Pattern used for output
           + 1;

      if (nPat < NumPat) NumPat = nPat;
    }

    TRAIN_LWB   = 0;
    TRAIN_UPB   = (INT)(NumPat * 0.7) - 1 ;      /* use 70 % of data in training set */
    TEST_LWB    = TRAIN_UPB + 1;
    TEST_UPB    = (INT)(NumPat * 0.9) - 1;       /* use 20 % of data in test set */
    EVAL_LWB    = TEST_UPB + 1;                  /* keep 10 % of data for evaluation */
    EVAL_UPB    = NumPat-1;

    TRAIN_PATTERNS = TRAIN_UPB - TRAIN_LWB + 1;
    TEST_PATTERNS  = TEST_UPB - TEST_LWB + 1;
    EVAL_PATTERNS  = EVAL_UPB - EVAL_LWB + 1;

    return 0;
  }

  /*****************************************************************************
                          I N I T I A L I Z A T I O N
   *****************************************************************************/

  void InitializeApplication()
  {
    INT  NumPattern, i, Off;
    REAL Out, Err, Mean;

    Input  = malloc( Net.N * sizeof(REAL));
    Output = malloc( Net.M * sizeof(REAL));
    Target = malloc( Net.M * sizeof(REAL));

    Mean = Modele.Var[0].Mean;

    TrainErrorPredictingMean = 0;
    for (NumPattern=TRAIN_LWB; NumPattern<=TRAIN_UPB; NumPattern++) {
      Off = Modele.Var[0].Offset + (Modele.Var[0].ValuesPerPattern * NumPattern);
      for (i=0; i<Net.M; i++) {
```

```
      Out = Modele.Var[0].Data[Off+i];
      Err = Mean - Out;
      TrainErrorPredictingMean += sqr(Err);
    }
  }
  TrainErrorPredictingMean *= 0.5;

  TestErrorPredictingMean = 0;
  for (NumPattern=TEST_LWB; NumPattern<=TEST_UPB;NumPattern++) {
    Off = Modele.Var[0].Offset + (Modele.Var[0].ValuesPerPattern * NumPattern);
    for (i=0; i<Net.M; i++) {
      Out = Modele.Var[0].Data[Off+i];
      Err = Mean - Out;
      TestErrorPredictingMean += sqr(Err);
    }
  }
  TestErrorPredictingMean *= 0.5;

  f = fopen("BPN.txt", "W");
}

void FinalizeApplication()
{
  INT l,i;

  for (l=0; l<Net.NumLayers; l++) {
    if (l != 0) {
      for (i=1; i<=Net.Layer[l]->Units; i++) {
         free(Net.Layer[l]->Weight[i]);
         free(Net.Layer[l]->WeightSave[i]);
         free(Net.Layer[l]->dWeight[i]);
      }
    }
    free(Net.Layer[l]->Output);
    free(Net.Layer[l]->Error);
    free(Net.Layer[l]->Weight);
    free(Net.Layer[l]->WeightSave);
    free(Net.Layer[l]->dWeight);

    free(Net.Layer[l]);
  }
  free(Net.Layer);

  for (i=0;i<Modele.NumVar;i++) {
     free(Modele.Var[i].Data);
  }
  free(Modele.Var);

  free(Input);
  free(Output);
```

```
  free(Target);

  fclose(f);
}


void PrepareInputOutput( int NumPattern, REAL *Input, REAL *Output )
{
  int v,n,Src;
  int iOff,oOff;

  iOff = oOff = 0;
  for (v=0;v<Modele.NumVar;v++) {

    n = Modele.Var[v].Input;
    Src = Modele.Var[v].Offset + (Modele.Var[v].ValuesPerPattern * NumPattern) - n;

    // test nombre de pattern disponible
    assert((Src + n) <= Modele.Var[v].NumData );

    while (n--) {
      Input[ iOff++ ] = Modele.Var[v].Data[Src++];
    }

    n = Modele.Var[v].Output;

    // test nombre de pattern disponible
    assert((Src + n) <= Modele.Var[v].NumData );

    while (n--) {
      Output[ oOff++ ] = Modele.Var[v].Data[Src++];
    }
  }
}

void SetInput(REAL* Input)
{
  INT i;

  for (i=1; i<=Net.InputLayer->Units; i++) {
    Net.InputLayer->Output[i] = Input[i-1];
  }
}

void GetOutput(REAL* Output)
{
  INT i;

  for (i=1; i<=Net.OutputLayer->Units; i++) {
    Output[i-1] = Net.OutputLayer->Output[i];
```

```
  }
}

/******************************************************************************
           S U P P O R T   F O R   S T O P P E D   T R A I N I N G
 ******************************************************************************/

   /
void SaveWeights()
{
  INT l,i,j;

  for (l=1; l<Net.NumLayers; l++) {
    for (i=1; i<=Net.Layer[l]->Units; i++) {
      for (j=0; j<=Net.Layer[l-1]->Units; j++) {
        Net.Layer[l]->WeightSave[i][j] = Net.Layer[l]->Weight[i][j];
      }
    }
  }
}


void RestoreWeights()
{
  INT l,i,j;

  for (l=1; l<Net.NumLayers; l++) {
    for (i=1; i<=Net.Layer[l]->Units; i++) {
      for (j=0; j<=Net.Layer[l-1]->Units; j++) {
        Net.Layer[l]->Weight[i][j] = Net.Layer[l]->WeightSave[i][j];
      }
    }
  }
}

/******************************************************************************
                   P R O P A G A T I N G   S I G N A L S
 ******************************************************************************/

void PropagateNet()
{
  INT l,i,j;
  REAL Sum;

  REAL *lWeight, *lOutput;

  for (l=0; l<Net.NumLayers-1; l++) {
    lOutput = Net.Layer[l]->Output;
    for (i=1; i<=Net.Layer[l+1]->Units; i++) {
      lWeight = Net.Layer[l+1]->Weight[i];
```

```
      Sum = 0;
      for (j=0; j<=Net.Layer[l]->Units; j++) {
        Sum += lWeight[j] * lOutput[j];
      }
      Net.Layer[l+1]->Output[i] = (1/(1+ (REAL)exp(-Sum)));
    }
  }
}

/*****************************************************************************
                B A C K P R O P A G A T I N G   E R R O R S
 *****************************************************************************/

void ComputeOutputError(REAL* Target)
{
  INT  i;
  REAL Out, Err;

  Net.Error = 0;
  for (i=1; i<=Net.OutputLayer->Units; i++) {
    Out = Net.OutputLayer->Output[i];
    Err = Target[i-1]-Out;
    Net.OutputLayer->Error[i] = Out * (1-Out) * Err; // * Net.Gain
    Net.Error += sqr(Err);
  }
  Net.Error *= 0.5;
}

void BackpropagateNet()
{
  INT l, i, j;
  REAL Out, Err;

  REAL **lWeight, *lError;

  for (l=Net.NumLayers-1; l>1; l--) {
    lWeight = Net.Layer[l]->Weight;
    lError  = Net.Layer[l]->Error;
    for (i=1; i<=Net.Layer[l-1]->Units; i++) {
      Err = 0;
      for (j=1; j<=Net.Layer[l]->Units; j++) {
        Err += lWeight[j][i] * lError[j];
      }
      Out = Net.Layer[l-1]->Output[i];
      Net.Layer[l-1]->Error[i] = Out * (1-Out) * Err;  // * Net.Gain
    }
  }
}

void AdjustWeights()
```

```
{
  INT  l,i,j;
  REAL Out, Err, dWeight;

  REAL **ldWeight,**lWeight, Corr;

  for (l=Net.NumLayers-1; l>0; l--) {
    ldWeight = Net.Layer[l]->dWeight;
    lWeight =  Net.Layer[l]->Weight;
    for (i=0; i<=Net.Layer[l-1]->Units; i++) {
      for (j=1; j<=Net.Layer[l]->Units; j++) {
        Out = Net.Layer[l-1]->Output[i];
        Err = Net.Layer[l]->Error[j];
        Corr = Net.Eta * Err * Out;
        dWeight = ldWeight[j][i];
        lWeight[j][i] += Corr + Net.Alpha * dWeight;
        ldWeight[j][i] = Corr;
      }
    }
  }
}


/*****************************************************************************
                    S I M U L A T I N G   T H E   N E T
 *****************************************************************************/

void SimulateNet(INT PatternNb, BOOL Training, REAL *Output)
{
//  REAL iE;

  PrepareInputOutput( PatternNb, Input, Target );

  SetInput(Input);
  PropagateNet();

  ComputeOutputError(Target);

  if (Training) {

//  iE = Net.Error;

    BackpropagateNet();
    AdjustWeights();

/*
      Attention à la convergence si Alpha > 0

      PropagateNet(Net);
      ComputeOutputError(Net, Target);
```

```
          if (iE < Net.Error) {
          fprintf(f,"Initial error %f\n",iE);
                  fprintf(f,"Final   error %f\n",Net.Error);
                  fflush(f);
      }
*/

  }
  else
    GetOutput(Output);

}

void TrainNet(INT Epochs)
{
  INT  PatternNb, n;

  for (n=0; n<Epochs*TRAIN_PATTERNS; n++) {
    PatternNb = RandomEqualINT(TRAIN_LWB, TRAIN_UPB);

  /*  fprintf(f,"before training %d ",PatternNb);
    fflush(f);
  */
    SimulateNet(PatternNb, TRUE, NULL);

    /*fprintf(f," - error %f\n",Net.Error);
    fflush(f);
    */
  }
}

void TestNet() // NET* Net)
{
  INT  Pattern;

  TrainError = 0;

  for (Pattern=TRAIN_LWB; Pattern<=TRAIN_UPB; Pattern++) {

    SimulateNet(Pattern, FALSE, Output);
    TrainError += Net.Error;
  }

  TestError = 0;
  for (Pattern=TEST_LWB; Pattern<=TEST_UPB; Pattern++) {
    SimulateNet(Pattern, FALSE, Output);
    TestError += Net.Error;
  }
```

```
  printf("\nNMSE is %0.3f on Training Set and %0.3f on Test Set",
            TrainError / TrainErrorPredictingMean,
            TestError / TestErrorPredictingMean);

  fprintf(f, "\nNMSE is %0.3f on Training Set and %0.3f on Test Set",
            TrainError / TrainErrorPredictingMean,
            TestError / TestErrorPredictingMean);
}


void EvaluateNet(REAL* Output)
{
  INT  Pattern;
  REAL LoopError;
  REAL Data;
  INT  i;

  REAL *Real;
  REAL *Result;
  REAL Mean,Num,Denom;

  LoopError = 0.0;
  Real   = malloc( sizeof(REAL) * EVAL_PATTERNS );
  Result = malloc( sizeof(REAL) * EVAL_PATTERNS );
  Mean  = 0.0;

  fprintf(f, "\n\n\n");
  fprintf(f, "Item    Value    Open-Loop Prediction\n");
  fprintf(f, "\n");

  for (Pattern=EVAL_LWB; Pattern<=EVAL_UPB; Pattern++) {

    SimulateNet(Pattern, FALSE, Output );

i = Modele.Var[0].Offset + (Modele.Var[0].ValuesPerPattern * Pattern);
Data = (Modele.Var[0].Data[ i ] - Modele.Var[0].b) / Modele.Var[0].a;
    Output[0] = (Output[0] - Modele.Var[0].b) / Modele.Var[0].a;

    Real[Pattern-EVAL_LWB] = Data;
    Result[Pattern-EVAL_LWB] = Output[0];

    fprintf(f, "%d       %0.4f                %0.4f\n",
              i, // Pattern,
              Data,
              Output[0]);

    LoopError += (REAL)fabs( Data - Output[0] );
Mean += Data;
  }
```

```
  LoopError /= EVAL_PATTERNS;

  fprintf(f, "\nOpen-Loop Error: %0.4f\n", LoopError);

  Mean /= EVAL_PATTERNS;
  Num = Denom = 0.0;
  for (Pattern=0; Pattern< EVAL_PATTERNS; Pattern++) {
    Num += sqr( Real[Pattern] - Result[Pattern] );
    Denom += sqr( Real[Pattern] - Mean);
  }
  LoopError = Num / Denom;
  fprintf(f, "\nNMSE: %0.4f\n", LoopError);

  free(Result);
  free(Real);
}

/*****************************************************************************
                                    M A I N
 *****************************************************************************/

int main(int argc, char **argv)
{
  BOOL Stop;
  REAL MinTestError;
  INT  iter = 0;

  if (argc != 4) {
    /*
     argv[0] = pgm
     argv[1] = Net def
     argv[2] = Data
     argv[3] = OutPut  Net weights */

    return -1;
  }

  InitializeRandoms();

  LoadNet( argv[1] );
  if (LoadData( argv[2] )) return -1;

  InitializeApplication();

  fprintf( f, "\n#training set\t%d",   TRAIN_PATTERNS );
  fprintf( f, "\n#test set\t%d",       TEST_PATTERNS );
  fprintf( f, "\n#evaluation set\t%d", EVAL_PATTERNS );
  fflush( f );

/*
```

```
  time( &ltime );
  fprintf( f, "start time %s\n", ctime( &ltime ) );
  fflush( f );
*/

  Stop = FALSE;
  MinTestError = MAX_REAL;
  do {
      TrainNet(10);
      TestNet();
      if (TestError < MinTestError) {
        fprintf(f, " - saving Weights ...\n");
        fflush( f );

        MinTestError = TestError;
        SaveWeights();
      }
      else if (TestError > 1.2 * MinTestError) {
        fprintf(f, " - stopping Training and restoring Weights ...\n");
        fflush( f );
        if (iter>50) Stop = TRUE;
        RestoreWeights();
      }
      iter++;
      /*if (iter >=100) {
        fprintf(f, " - stopping Training and restoring Weights ...\n");
        fflush( f );

        Stop = TRUE;
        RestoreWeights();

      } */
  } while (! Stop);

  /*time( &ltime );
  fprintf( f, "start time %s\n", ctime( &ltime ) );
  fflush( f );
  */
  TestNet();
  SaveNet( argv[3]);

  EvaluateNet(Output);

  FinalizeApplication(); // Free net from memory

  return 0;
}
```

Annexe B : Format des fichiers utilisés par le réseau de neurones

Le fichier de description du réseau

Net.txt

1 ère ligne : nombre de couches dans le réseau (ex.3)

2 ème ligne : nombre de neurones dans la couche d'entrée

3 ème ligne : nombre de neurones dans la couche cachée

4 ème ligne : nombre de neurones dans la couche de sortie

Le fichier de description des données

Data.txt

1 ère ligne : nombre de variables à inclure (égale au nombre de lignes suivantes)

Lignes suivantes : description des variables

i1,i2,i3,filename

avec

i1 : le nombre de valeurs par exemple (pattern), par jour dans ce cas

i2 : le nombre de valeurs historiques et futures en entrée (pas forcement un multiple de i1)

i3 : le nombre de valeurs en sortie

filename : le nom du fichier contenant les données.

Les fichiers de données

Ce fichier contient les valeurs codées en réel 32 bits

Ce fichier doit ce trouver dans le même répertoire que le fichier de descripton