Université de Lille

Travail Encadré de Recherche

Intelligence artificielle et apprentissage de réseaux connexionnistes

Aurélien Pottiez
Marc Duquesnoy

Professeur encadrant :
Bernhard Beckermann

11 mai 2018

Remerciements

Nous adressons nos remerciements aux personnes qui nous ont aidé dans la réalisation de ce Travail Encadré de Recherche.

En premier lieu, nous remercions M. Beckermann, enseignant chercheur à l'université de Lille. En tant que tuteur de ce projet, il nous a guidés dans notre travail, nous a aidés à résoudre quelques problèmes au cours de rendez-vous hebdomadaires et s'est montré très disponible.

Nous remercions également le corps enseignant du Master Mathématiques Appliquées, Statistique, pour la qualité des enseignements dispensés, ce qui nous a permis d'acquérir les connaissances et compétences nécessaires à la réalisation de ce Travail Encadré de Recherche.

Table des matières

Introduction

Les réseaux de neurones artificiels tirent leur origine des neurones biologiques qui composent le cerveau humain. On peut alors percevoir ces réseaux connexionnistes comme une tentative de modéliser, d'un point de vue mathématique et informatique, le cerveau humain. Dès lors, leur conception vise à reproduire une de ses caractéristiques les plus importantes : la capacité d'apprentissage. Apprendre, comme le cerveau apprend grâce à l'expérience, puis reconnaître des règles, les généraliser dans le but de classifier, de prédire. Ainsi, ils sont aujourd'hui utilisés dans divers domaines, comme la reconnaissance d'images, la classification de données, le traitement de la parole, ... Revenons sur leur historique.

En 1943, W. McCulloch et W. Pitts proposent une première définition et modélisation, appelé **neurone formel** [1]. Profondément inspiré du neurone biologique, ce neurone à l'architecture très simplifié renvoie une valeur booléenne, *oui* ou *non*, en affectant à chaque entrée des *poids de connexion* (ou *poids synaptiques*, en analogie avec les synapses du corps humain).
Néanmoins, si l'on veut reproduire la capacité d'apprentissage du cerveau, ces poids se doivent de ne pas être fixes mais variables, afin de s'adapter à certaines règles. C'est ainsi qu'en 1949, D. Hebb établit la **Règle de Hebb** [2], qui décrit les changements d'adaptation neuronale pendant un processus d'apprentissage, ce qui va permettre de sérieusement envisager la possibilité d'apprendre pour un réseau de neurones artificiels.

Dans la continuité, en 1957, F. Rosenblatt, en se basant sur les travaux de D. Hebb, propose le modèle du **perceptron** [3] : il s'agit d'un neurone formel possédant une règle d'apprentissage qui permet de déterminer automatiquement ses poids synaptiques.
On semble donc s'orienter vers des avancées majeures dans le domaine. Cependant, en 1969, M. Minsky et S. Papert mettent en avant des limites du perceptron [4]. Pour résumer leur argumentation, les fonctions de type **XOR** (*eXclusive OR*, traduit *OU exclusif*) sont reconnus indispensables en reconnaissance des formes. Or, le perceptron étant un séparateur linéaire, il ne peut apprendre ce type de fonctions. Il faudrait alors des réseaux de neurones plus complexes, et aucune règle ni aucun algorithme d'apprentissage pour ces réseaux n'est envisageable pour les chercheurs. Les recherches sont peu à peu abandonnées, même si S. Grossberg et T. Kohonen par exemple vont continuer, discrètement, leurs recherches.

C'est en 1982 que l'on constate un regain d'intérêt pour les réseaux de neurones, en grande partie dû aux travaux de J.J. Hopfield, aboutissant au **Modèle de Hopfield** [5]. Ce modèle discute de la théorie du fonctionnement et des possibilités des réseaux de neurones et, même si les limitations présentées par M. Minsky ne sont pas levées, les recherches sont relancées.
C'est alors qu'en 1985, Y. LeCun propose le modèle de **Réseaux de neurones multicouches**, aussi appelés **perceptron multicouches** [6]. Ces réseaux se basent sur le concept de **rétropropagation du gradient** comme règle d'apprentissage (dont le calcul est détaillé dans la suite du document) et permettent d'apprendre des fonctions telles la fonction **XOR**. Aussi, le perceptron multicouches n'est pas un séparateur linéaire : il permet ainsi de classer des données non linéairement séparables. De nos jours, les réseaux multicouches et la rétropropagation du gradient reste le modèle le plus étudié et le plus productif au niveau des applications.

Nous nous intéresserons ici à **l'apprentissage supervisé**, qui sera défini ultérieurement. Dans ce contexte, le modèle du perceptron multicouches sera l'objet de nos recherches. Ainsi, nous articulerons ce document autour de 3 grands axes : tout d'abord, nous allons définir et expliciter le fonctionnement d'un réseau de neurones multicouches ; ensuite, nous présenterons différents algorithmes et leur implémentation ; enfin, nous réaliserons des simulations numériques et interpréterons les résultats associés.

Le premier chapitre sera donc consacré aux différentes définitions. Après avoir introduit les premières notions, nous approfondirons le calcul central dans l'apprentissage d'un réseau de neurones multicouches : la rétro-propagation du gradient.

Le détail des algorithmes d'apprentissage sera l'objet du deuxième chapitre. Nous étudierons l'algorithme dit **gradient stochastique**, prépondérant dans les applications, mais nous essaierons également de le comparer à des algorithmes se basant sur des méthodes d'optimisation, tels **steepest descent** ou encore la méthode de **Broyden Fletcher Goldfarb Shanno (BFGS)**. Puis, nous verrons comment implémenter ces algorithmes en Fortran.

Nous réaliserons ensuite des simulations numériques. Tout d'abord, nous détaillerons quelques esquisses de code réalisées sous R qui nous permettront d'appréhender l'algorithme d'apprentissage ainsi que les exemples sur lesquels nous allons le tester : les iris de Fisher et l'addition de deux nombres binaires. Par la suite, nous effectuerons les simulations en Fortran sur ces deux exemples.

Enfin, nous essaierons d'apporter quelques compléments à notre travail. Nous ferons dans ce cadre quelques liens avec des méthodes d'analyse de données. De plus, nous proposerons des pistes visant à améliorer l'algorithme d'apprentissage. Aussi, nous verrons les limites des réseaux de neurones, en particulier l'interprétation des neurones constituant le réseau, le choix des paramètres de l'algorithme. Nous analyserons alors quelles méthodes d'apprentissage peuvent être préférées en pratique

Chapitre 1

Les réseaux de neurones

1.1 Neurone formel

Un neurone dit "formel" peut être vu comme un sommet d'où l'on tire une information. En effet, un réseau de neurones peut se représenter comme un graphe orienté. Par la suite, surtout dans la section 1.4, nous aurons besoin de quelques notions de théorie des graphes orientés, que l'on récapitule succinctement ici :

- on note $G = (S, A)$ un graphe orienté ;
- S est l'ensemble des sommets (les neurones) ;
- A est l'ensemble des arcs (les connexions) ;
- un chemin, de source s_0 et de destination s_n est une suite $\gamma = [s_0, ..., s_n]$ où $s_i \in S$, $i \in 0, ..., n$ et $(s_i, s_i + 1) \in A$, $i \in 0, ..., n - 1$

Si l'on parle donc d'information sur chaque neurone, ou sommet, il faut y définir une quantité y_j, $j \in S$, avec $S = \bigcup_{l=1}^{q} S_l$ où q est le nombre de couches, S_1 la couche comprenant les neurones d'entrée, S_q la couche comprenant les neurones de sorties et S_l, $l \in 2, ..., q-1$ les couches cachées. On notera

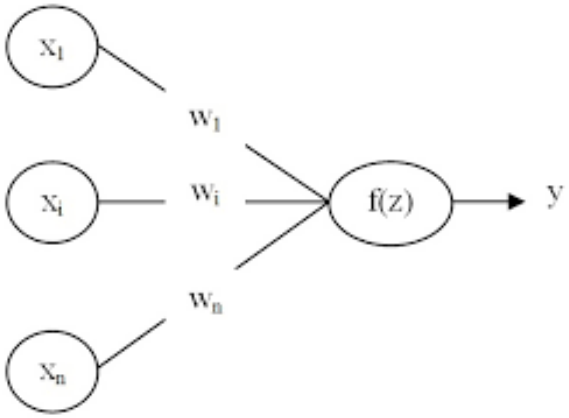
$$y_j = f\left(-b_j + \sum_{(i,j)\in A} w_{i,j} \cdot y_i\right), \qquad j \in S \setminus S_1 \tag{1.1}$$

avec $w_{i,j}$ le poids associé à l'arc (i, j) et b_j le seuil, que l'on considérera comme un poids, associé à chaque neurone qui n'est pas un neurone d'entrée.

1.2 Perceptron

1.2.1 Définition

Le perceptron est un type particulier de neurone formel : il peut être vu comme un réseau de neurones simple ; en effet, ce réseau de neurones ne possède qu'une seule sortie sans couche cachée. On prend $x_1, x_2, ..., x_n$ les n entrées, et $w_1, w_2, ..., w_n$ les n poids associés aux entrées, telle que la somme pondérée des entrées par les poids vaut une certaine valeur z, qui sera évaluée par une fonction d'activation f. Voici comment représenter le perceptron :



1.2.2 Fonction d'activation

Plusieurs fonctions d'activation s'offrent à nous. Étant donné que le but de ce perceptron est de classifier, on veut que la variable de sortie soit comprise entre 0 et 1 : si la sortie est proche de 1 ("proche" est à définir selon le contexte), on dira que le neurone de sortie est actif, si la sortie est proche de 0, il sera dit inactif (d'où le terme de fonction d'activation). Dans la pratique, on choisira des fonctions comme la fonction de Heaviside, la sigmoïde logistique, la tangente hyperbolique (renormalisée par la suite pour que l'on se situe dans l'intervalle $[0, 1]$) ou encore signe doux :

Heaviside	$f(x) = 0$ si $x < 0$, 1 si $x > 0$
Sigmoïde logistique	$f(x) = \frac{1}{1+e^{-x}}$
Tangente Hyperbolique	$f(x) = \frac{2}{1+e^{-2x}} - 1$
Signe doux	$\frac{x}{1+\|x\|}$

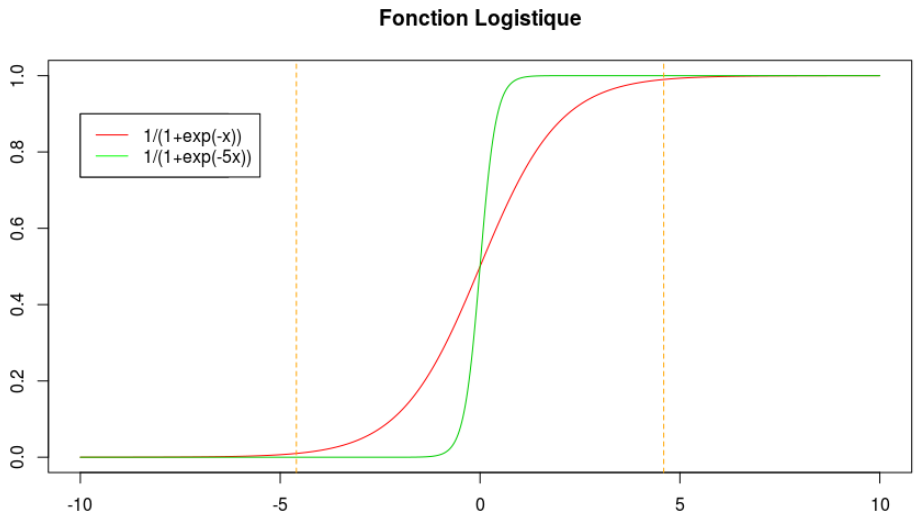
Cependant, il est important de préciser que la fonction doit être suffisamment continue et dérivable telle que l'on puisse en calculer le gradient. De plus, si la fonction est solution d'une équation différentielle de la forme

$$f'(x) = F(f) \tag{1.2}$$

cela nous permettra de simplifier des calculs. Nous verrons cela dans la partie consacrée au calcul du gradient.

Dans la suite, nous avons choisi la fonction logistique, choix motivé par le fait qu'elle renvoie des valeurs comprises entre 0 et 1, qu'elle est $\mathbf{C}^\infty$ et qu'elle vérifie l'équation différentielle (1.2) :

$$f(x) = \frac{1}{1 + e^{-\alpha * x}} \tag{1.3}$$



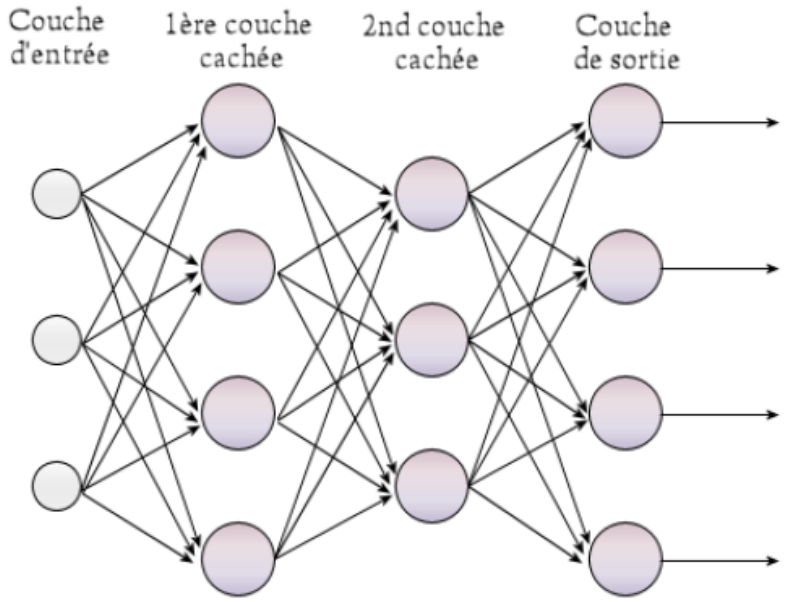
Ci-dessus on affiche une représentation de la fonction logistique pour des valeurs $\alpha = 1$ et 5. Nous avons choisi de prendre $\alpha = 1$ de manière empirique. En effet plus α est grand plus il serait possible d'avoir une explosion des valeurs dans le calcul du gradient car la droite d'équation $x = 0$ pourrait être une tangente à la courbe, on se rapproche donc d'un cas de singularité, le choix de la valeur $\alpha = 1$ est sans perte de généralité : les poids $w_{i,j}$ avec $\alpha = c$ correspondent aux poids $c * w_{i,j}$ avec $\alpha = 1$.

Ce perceptron classe ensuite selon la valeur de sortie. Cette valeur de sortie se doit d'être interprétée comme booléenne : en effet, le perceptron est un classificateur linéaire et vise donc à séparer deux classes. Il ne s'applique donc efficacement qu'à des données linéairement séparables. C'est pourquoi on introduit la notion de perceptron multicouches.

1.3 Perceptron multicouches

1.3.1 Définition

Le perceptron multicouches possède un plus grand nombre de poids répartis selon des couches dites intermédiaires ou cachées (où le nombre de neurones peut varier selon le problème posé), et des seuils sur chaque sommet qui n'est pas un sommet d'entrée. Voici un exemple d'un perceptron multicouches :



On remarque tout d'abord que les sommets sont reliés entre eux dans un et un seul sens : en effet, un neurone (sommet) qui constitue ce réseau reçoit des informations de la couche située juste avant la couche dont il fait partie, puis transmet une information vers la couche suivante. De plus, aucun neurone n'est relié à un neurone situé sur la même couche. Aussi, les poids reliant les neurones ne "sautent" pas de couches ; ainsi, il n'existe pas de poids reliant un neurone de la couche l à un neurone de la couche $l+2, l+3, ...$ Enfin, il est important de noter que chaque sommet d'une couche est relié à tous les sommets de la couche suivante : on parle dans ce cas d'un graphe plein. Cependant, cela n'est pas obligatoire et on peut tout à fait envisager de ne pas relier tous les sommets entre eux. Il faudrait alors réfléchir à une structure de graphe particulière, qui nous permettrait de comprendre le rôle de chaque neurone, et d'ainsi pouvoir les interpréter, ce qui n'est pas envisageable dans la grande majorité des cas. Nous reviendrons sur ce point dans l'exemple de l'addition de nombres binaires, puis en développant les limites d'un réseau de neurones.

1.3.2 Apprentissage des poids

On introduit tout d'abord ces notations :

— X un ensemble dit ensemble d'apprentissage
— $card(X) = L$
— $(x^{(i)}, z^{(i)})$ un exemple de l'ensemble d'apprentissage, $i \in 1, ..., L$
— $x^{(i)}$ est l'entrée d'un exemple
— $z^{(i)}$ est la sortie théorique associée, connue

L'apprentissage choisi pour notre réseau de neurones est un apprentissage dit supervisé. Ce type d'apprentissage consiste à produire des règles basées sur l'ensemble d'apprentissage contenant les exemples. A l'inverse de l'apprentissage non-supervisé, le réseau est entraîné pour converger vers un état final précis, c'est-à-dire qu'à chaque présentation d'un exemple, on tend à se rapprocher de la sortie théorique. On vise donc à minimiser l'erreur quadratique, lorsque l'on donne en entrée les entrées $x^{(i)}$, entre les sorties $y_k^{(i)}, k \in S_q$ du perceptron multicouche et les sorties théoriques $z_k^{(i)}$ associées que l'on notera :

$$E^{(i)} = \frac{1}{2} \sum (y_k^{(i)} - z_k^{(i)})^2 \quad \forall k \in S_q, \forall i \in 1, ..., L$$

Dans la suite, nous allons noter $x_k^{(i)} = x_k$, $y_k^{(i)} = y_k$ et $z_k^{(i)} = z_k$, car les calculs seront valables pour chaque exemple de l'ensemble d'apprentissage.

1.4 Calcul du gradient par rétro-propagation

Tout d'abord, nous aurons besoin de la formule de dérivée d'une composée de fonctions. Soient :

$$f : \begin{cases} \mathbb{R}^n \mapsto \mathbb{R}^m \\ x \mapsto f(x) \end{cases}, \quad g : \begin{cases} \mathbb{R}^p \mapsto \mathbb{R}^n \\ y \mapsto g(y) \end{cases}, \quad f \circ g : \begin{cases} \mathbb{R}^p \mapsto \mathbb{R}^m \\ c \mapsto f(g(y)) \end{cases},$$

Alors, nous avons que :

$$\frac{\partial (f \circ g)_j}{\partial y_k}(y) = \sum_\ell \frac{\partial f_j}{\partial x_\ell}(g(y)) \frac{\partial g_\ell}{\partial y_k}(y). \tag{1.4}$$

Par abus de notation, on notera, en posant $h = f \circ g$:

$$\frac{\partial h_j}{\partial y_k} = \sum_\ell \frac{\partial h_j}{\partial x_\ell} \frac{\partial x_\ell}{\partial y_k}, \tag{1.5}$$

où h dépend de x explicitement et de y implicitement.

Comme défini précédemment, l'ensemble des neurones d'entrée est noté S_1. Mais ici, pour ne pas se contenter de traiter le cas d'un graphe plein et donc pour ne pas perdre en généralité dans les calculs ci-dessous, on va introduire la notation $niveau(k)$ pour k sommet (neurone) qui n'est ni d'entrée ni de sortie. Elle définira la longueur maximale, en nombre d'arcs, d'un chemin d'un neurone d'entrée vers un neurone k. On notera alors S_l l'ensemble des sommets de niveau l. Ainsi, on aura défini une partition $S_1, ..., S_q$ de S.
On remarque que :

$$(j,k) \in A \quad \Longrightarrow \quad niveau(k) \geq niveau(j) + 1$$

Cependant, on peut imaginer avoir $niveau(k) \geq niveau(j)+2$, $niveau(k) \geq niveau(j)+3$,... Nous développerons ceci dans la section 3.2.1, lorsque l'on discutera de la structure optimisée d'un graphe pour l'addition de nombres binaires.

Rappelons la formule :

$$\forall k \in S \setminus S_1 : \quad y_k = f\Big(-b_k + \sum_{(j,k)\in A} y_j w_{j,k}\Big) \tag{1.6}$$

Comme nous le verrons dans le code, nous numérotons les sommets par ordre croissant de niveau. On pose maintenant le vecteur ligne y contenant les valeurs de chaque neurone. Aussi, on pose W la matrice des poids et b le vecteur ligne contenant les seuils, tous deux à ajuster lors de l'apprentissage. On a donc :

$$\forall i, j \in 1, ..., n \quad W_{i,j} = \begin{cases} w_{i,j} & \text{si l'arc (i,j) existe} \\ 0 & \text{sinon} \end{cases}$$

et

$$y = [y_k], k \in S, \quad b = [b_k], k \in S \setminus S_0,$$

Remarquons que par (1.4), W est triangulaire supérieure strictement. De ce fait, on peut calculer y_k :

$$y_k = g\Big((y \cdot W - b)_k\Big), \quad \forall k \in S \setminus S_0 \tag{1.7}$$

Le produit $y \cdot W$ fera apparaître des y_j avec $niveau(j) < niveau(k)$.
Comme défini en (1.3.2), le but est de minimiser l'erreur définie par la fonction

$$E^{(i)} = \frac{1}{2}\sum_{k\in S_q}\left(y_k - z_k^{(i)}\right)^2 \tag{1.8}$$

avec $y = \Big[x^{(i)}, g(y\cdot W - b)\Big]$, $x^{(i)} = (x_k^{(i)})_{k\in S_1}$ l'entrée pour l'exmple i et $z^{(i)} = (z_k^{(i)})_{k\in S_q}$ la sortie attendue.

Par (1.7), on observe que $E_{k-1}^{(i)}$ s'obtient si l'on réécrit y_k en fonction de b_k ainsi que de $y_1, ..., y_{k-1}$, et donc, par récursivité, on obtient la formule :

$$\text{Pour } l \in S \setminus S_1 : \quad \frac{\partial E^{(i)}}{\partial b_l} = \frac{\partial E_0^{(i)}}{\partial b_l}$$

Intéressons nous maintenant au calcul de $\frac{\partial E^{(i)}}{\partial b_l}$, $\quad l \in S \setminus S_0$, en considérant les poids $w_{j,k}$ comme des constantes. On remarque tout d'abord que $y_1, ..., y_{k-1}$ ne dépendent pas de b_k mais uniquement de $b_1, ..., b_{k-1}$. Alors, on peut écrire que :

$$\begin{aligned}\text{Pour } l \in S \setminus S_1 : \quad \frac{\partial E^{(i)}}{\partial b_l} &= \frac{\partial E_{l-1}^{(i)}}{\partial b_l} \\ &= \frac{\partial E_l^{(i)}}{\partial y_l}\frac{y_l}{\partial b_l} \\ &= -\frac{\partial E_l^{(i)}}{\partial y_l}G(y_l)\end{aligned} \tag{1.9}$$

avec $G(y) = g'(y)$. De plus, pour $k > l$, on a :

$$\frac{\partial y_k}{\partial y_l} = G(y_k w_{l,k}) \quad \text{si} \quad (l,k) \in A$$

Par conséquent

$$\text{pour } k > l : \quad \frac{\partial E_{k-1}^{(i)}}{\partial y_l} = \frac{\partial E_k^{(i)}}{\partial y_l} + \frac{\partial E_k^{(i)}}{\partial y_k}\frac{\partial y_k}{\partial y_l} = \frac{\partial E_k^{(i)}}{\partial y_l} + \begin{cases} \frac{\partial E_k^{(i)}}{\partial y_k}G(y_k)w_{l,k} & \text{si } (l,k) \in A \\ 0 & \text{si } (l,k) \notin A \end{cases}$$

Sur la couche de sortie, le calcul du gradient est trivial. En effet, nous avons que

$$\forall l \in S_q \quad \frac{\partial E^{(i)}}{\partial y_l} = \frac{\partial}{\partial y_l}\Big(\frac{1}{2}\sum_k (y_k^{(i)} - z_k^{(i)})^2\Big) = y_m^{(i)} - z_m^{(i)} \tag{1.10}$$

Finalement, en combinant ces résultats, et en multipliant par $-G(y_l)$ (par (1.10)), nous pouvons proposer la formule suivante :

$$\forall l \in S \setminus S_1 : \qquad \frac{\partial E^{(i)}}{\partial b_l} = \begin{cases} -G(y_l)(y_l - z_l^{(i)}) & \text{si } l \in S_q, \\ \sum_{(l,k)\in A} G(y_l)\cdot w_{l,k}\frac{\partial E^{(i)}}{\partial b_k} & \text{sinon.} \end{cases} \tag{1.11}$$

Nous avons donc la formule par rapport aux b_k. Cependant, si l'on remarque que les poids $w_{j,k}$ avec $(j,k) \in A$ apparaissent au mêm niveau que b_k, on en déduit la formule par rapport aux $w_{j,k}$:

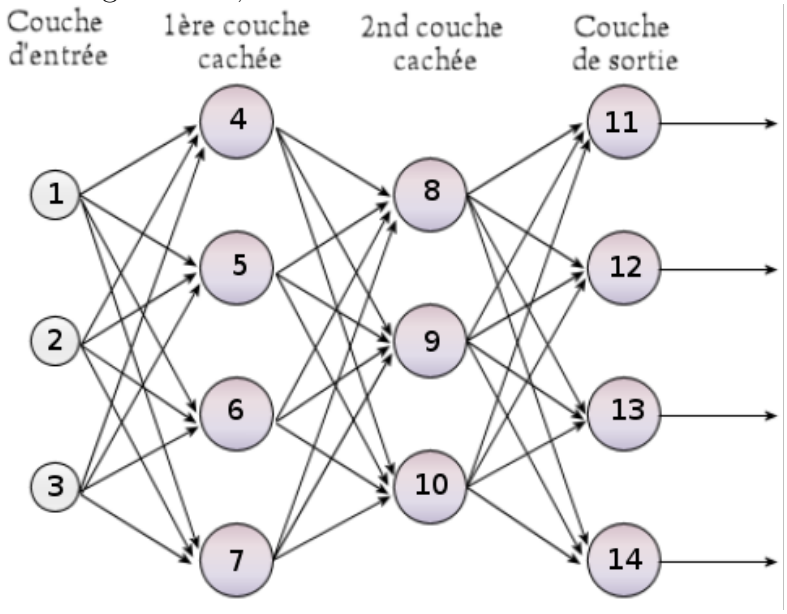
$$\begin{aligned}\text{Pour } (j,k) \in A : \quad \frac{\partial E^{(i)}}{\partial w_{j,k}} &= \frac{\partial E_0^{(i)}}{\partial w_{j,k}} \\ &= -y_j\frac{\partial E^{(i)}}{\partial b_k}\end{aligned} \tag{1.12}$$

Chapitre 2

Algorithmes d'apprentissage

2.1 Algorithme "gradient stochastique"

Avant de proposer un algorithme d'apprentissage, il faut tout d'abord discuter de l'initialisation des poids w_{ij}. D'après Philippe Preux [7], qui cite lui-même un article de Yann Le Cun [8], il faudrait initialiser les $w_{i,j}$ selon *une loi normale, centrée, et d'écart-type la racine carrée du nombre d'entrées de l'unité à laquelle se trouve ce poids*. Ceci dans le but d'avoir la somme pondérée des entrées proche du point d'inflexion de la sigmoïde logistique. C'est cette initialisation que nous allons choisir.
Pour mieux visualiser ce choix, prenons l'exemple du perceptron multicouches donné en 1.3.1. Dans la suite, pour nos algorithmes, nous allons numéroter les neurones comme suit :



Ainsi, en suivant la règle d'initialisation des poids donnée précédemment, les poids sortant des neurones 4 , 5, 6 et 7 suivront une loi normale $\mathcal{N}(0, \sqrt{3})$, ceux sortant des neurones 8, 9 et 10 suivront une $\mathcal{N}(0, 2)$ et enfin ceux sortant des neurones 11, 12, 13 et 14 une $\mathcal{N}(0, \sqrt{3})$. Pour les neurones d'entrée, étant donné qu'il n'y a pas de poids entrants, on initialise les poids selon une $\mathcal{N}(0, 1)$.

Voici l'algorithme, en pseudo-code, que l'on implémente (voir Annexe B pour le code Fortran) :

Algorithme gradient stochastique

fixer l'ensemble d'apprentissage X
initialiser les poids $w_{i,j}$ et b_j, stockés dans des vecteurs lignes, respectivement w et b
fixer le pas d'apprentissage α
boucle tant que critère d'arrêt non-valide
    mélanger X
    boucle pour tous les exemples dans X
        calculer la sortie du réseau pour l'exemple
        mettre a jour l'erreur (norme 2 de la différence avec la sortie attendue)
        calculer le gradient par rapport à w
        calculer le gradient par rapport à b
        mettre à jour b
        mettre à jour w
    fin boucle pour
fin boucle tant que

Aussi, il faut maintenant discuter du critère d'arrêt : lequel choisir ? Quelles valeurs pour ce choix ? Plusieurs suggestions s'offrent à nous :

— stabilisation de l'erreur d'une itération du tant que à la suivante ;
— nombre d'itérations maximum fixé au préalable ;
— scindé X en un échantillon d'apprentissage X' et un échantillon de validation Y' (voir 4.3.) ;
— erreur inférieure à un seuil.

C'est ce dernier que nous allons préférer, couplé à un nombre maximum d'itérations. Mais le seuil n'est pas établi théoriquement, et il n'y a pas de règle ou de démonstration pouvant nous indiquer comment choisir ce paramètre. En effet, dans les ouvrages mathématiques traitant du sujet, on trouve souvent la mention *à définir par l'utilisateur*. On voit cependant que ce critère va être influencé par la taille de l'échantillon d'apprentissage : effectivement, on calcule l'erreur en sommant les normes 2 des différences entre les sorties du réseau et les sorties attendues. Il en résulte que plus les exemples sont nombreux, plus l'erreur sera grande et plus il sera difficile d'obtenir une valeur petite, d'où la nécessité de prendre en considération la taille de l'ensemble d'apprentissage X. De plus, un choix de seuil trop petit pourra entraîner un phénomène appelé "sur-apprentissage".
Le sur-apprentissage est un apprentissage qui colle trop aux données. On peut même faire l'analogie avec l'être humain et parler d'apprentissage par coeur : l'ordinateur va apprendre trop en profondeur les données passés en exemple et va, de ce fait, apprendre les biais qui existent dans tout jeu de données fini. Bien qu'il puisse classer parfaitement tous les exemples appris, cela va engendrer une perte de la généralisation, c'est-à-dire que pour de nouvelles données, l'ordinateur sera incapable de prévoir de manière fiable la sortie attendue. Ceci peut s'éviter en utilisant un échantillon de validation (sujet développé dans la section 4.3) par exemple, mais ce problème de sur-apprentissage fait partie de nombreuses recherches sur le sujet de l'intelligence artificielle et plus particulièrement dans le contexte de *deep learning* [9].
Nous pouvons également préciser les autres étapes de notre algorithme :

— **mélanger** X : utile pour que l'ordinateur ne retraite pas les exemples dans le même ordre, ce qui serait beaucoup moins efficace et l'amènerait à tirer les mêmes règles de notre ensemble d'apprentissage à chaque présentation des exemples ;
— **calculer la sortie du réseau** : pour les entrées de l'exemple à traiter, une fonction renvoie les neurones de sortie en propageant l'information dans le réseau, en utilisant les

poids stockés dans w et b ;
— **mettre à jour erreur** : ajouter à l'erreur la norme 2 la différence entre la sortie calculée par le réseau et la sortie théorique (erreur à réinitialiser après présentation de tous les exemples de X) ;
— **calcul du gradient par rapport à** b : comme défini en 1.4. ;
— **calcul du gradient par rapport à** w : comme défini en 1.4. ;
— **mise à jour de** w : w <- $w - \alpha \cdot$ gradient par rapport à w ;
— **mise à jour de** b : b <- $b - \alpha \cdot$ gradient par rapport à b.

Notre algorithme devient alors :

Algorithme gradient stochastique

fixer l'ensemble d'apprentissage X
initialiser les poids $w_{i,j}$ et b_j, stockés dans des vecteurs lignes, respectivement w et b
fixer le pas d'apprentissage α (0.5 ou 0.7)
fixer le seuil ϵ
fixer un nombre d'itérations maximum, $maxit$
boucle tant que erreur $> \epsilon$ et iterations $< maxit$
    erreur <- 0
    mélanger X
    boucle pour tous les exemples $(x^{(i)}, z^{(i)})$ dans X
        calculer la sortie $y^{(i)}$ du réseau pour l'entrée $x^{(i)}$
        erreur <- erreur + norme2$(y^{(i)} - z^{(i)})$
        calculer le gradient par rapport à w
        calculer le gradient par rapport à b
        w <- $w - \alpha \cdot$ gradient par rapport à w
        b <- $b - \alpha \cdot$ gradient par rapport à b
    fin boucle pour
fin boucle tant que

2.2 Algorithme "gradient steepest descent"

On peut remarquer que dans l'algorithme présenté précédemment, on met à jour les poids à chaque passage d'un exemple : cette méthode est appelée "on-line". Cependant, on peut très bien imaginer sommer les erreurs commises sur chaque exemple, puis, quand tous les exemples ont été traités, on calcule le gradient et on le rétropropage en utilisant l'erreur totale commise. Cette méthode s'appelle "batch". Dès lors, on peut voir un lien avec les méthodes d'optimisation convexe, comme par exemple la méthode de la plus forte pente (*steepest descent* en anglais).

2.2.1 Présentation brève de la méthode

L'algorithme de la plus forte pente est un algorithme d'optimisation à direction de descente, qui vise à minimiser une fonction. Deux quantités sont à définir : la direction de descente et le pas de descente. Dans le cas de la méthode *steepest descent*, la direction sera l'opposée du gradient et le pas est obtenu par une recherche linéaire (nous allons ici utiliser une recherche par *backtracking*). Voici l'algorithme :

Algorithme *steepest descent* pour minimiser une fonction f

x_0 donné
boucle pour k = 0,1,... jusqu'à <critère d'arrêt>
    direction de déplacement d_k = -$\nabla f(x_k)^T$
    choisir un pas t_k par backtracking $(*)$
    nouvel itéré : $x_{k+1} = x_k + t_k d_k$
fin boucle pour

avec $(*)$:

Méthode de backtracking

on fixe 2 paramètres $0 < \alpha < \frac{1}{2}$ et $0 < \beta < 1$
Etape k : on se donne x_k et la direction de descente d_k
on fixe $t = 1$
boucle tant que $f(x_k + td_k) > f(x_k) + \alpha t \nabla f(x_k) d_k$
    faire $t \longleftarrow \beta t$
fin boucle tant que
$t_k \longleftarrow t$
$x_{k+1} = x_k + t_k d_k$

2.2.2 Algorithme d'apprentissage par steepest descent

On obtient donc, si l'on veut utilisr la méthode de la plus forte pente pour rétropropager l'erreur, l'algorithme d'apprentissage suivant :

Algorithme gradient steepest descent

fixer l'ensemble d'apprentissage X
initialiser les poids $w_{i,j}$ et b_j, stockés dans des vecteurs lignes, respectivement w et b
fixer le seuil ϵ
fixer un nombre d'itérations maximum, $maxit$
$grad_w = \frac{\partial E}{\partial w}$, $grad_b = \frac{\partial E}{\partial b}$, stockés en vecteur ligne
boucle tant que erreur $> \epsilon$ et iterations $< maxit$
    erreur <- 0
    mélanger X
    boucle pour tous les exemples $(x^{(i)}, z^{(i)})$ dans X
        calculer la sortie $y^{(i)}$ du réseau pour l'entrée $x^{(i)}$
        erreur <- erreur + norme2$(y^{(i)} - z^{(i)})$
        $grad_w = grad_w + \frac{\partial E^{(i)}}{\partial w}$
        $grad_b = grad_b + \frac{\partial E^{(i)}}{\partial b}$
    fin boucle pour
    recherche de α par **backtracking (*)**
    w <- $w - \alpha \cdot grad_w$
    b <- $b - \alpha \cdot grad_b$
fin boucle tant que

2.3 Algorithme "gradient BFGS"

Un troisième algorithme est envisageable, s'appuyant sur la méthode BFGS, dont la méthode générale est donnée ci dessous :

Algorithme BFGS

fixer l'ensemble d'apprentissage X
initialiser les poids $w_{i,k}$ et b_k, stockés dans un vecteur ligne x_k
résoudre $B_k p_k = -\nabla f(x_k)$
recherche de α par **backtracking (*)**
calculer le gradient par rapport à x_{k+1}
calculer $\delta_k = \nabla\ f(x_{k+1})$ - $\nabla\ f(x_k)$
mettre à jour $B_{k+1} = B_k + (\delta_k \delta_k^T)/(\delta_k^T s_k) - (B_k s_k s_k^T B_k)/(s_k^T B_k s_k)$

En effet, dans le cas d'un algorithme de descente, il nous permet de calculer une approximation du Hessien tout en calculant les poids et les seuils jusqu'à la convergence de ces derniers. Pour cette méthode, on appelle B_k l'approximation du Hessien telle que : $B_{k+1} = B_k + (\delta_k \delta_k^T)/(\delta_k^T s_k) - (B_k s_k s_k^T B_k)/(s_k^T B_k s_k)$.

Or, la récurrence est basée sur le produit de deux matrices de tailles k*2 où k est la longueur de y_k dans le cas où le nombre d'inconnues est plus faible que le nombre d'itérations nécessaires à la convergence, c'est à dire :

$$(\delta_k \delta_k^T)/(\delta_k^T s_k) - (B_k s_k s_k^T B_k)/(s_k^T B_k s_k) = \begin{pmatrix} \delta_k & B_k s_k \end{pmatrix} \times \begin{pmatrix} \delta_k^T/(\delta_k^T s_k) \\ (-s_k^T B_k)/(s_k^T B_k s_k) \end{pmatrix} \tag{2.1}$$

C'est pourquoi, Il est possible d'écrire $B_{k+1} = I + F_{k+1} G_{k+1}^T$ où F_{k+1} et G_{k+1} sont deux matrices issues de la récurrence de BFGS où l'on rajoute les deux lignes (vues au dessus) à chaque itération pour F_k et G_k. Après avoir généralisé :

$$F_k = \left(\begin{array}{cc|cc|cc|c|cc} \delta_0 & B_0 s_0 & \delta_1 & B_1 s_1 & \delta_2 & B_2 s_2 & ... & \delta_k & B_k s_k \end{array} \right) \tag{2.2}$$

$$G_k^T = \left(\begin{array}{c} \delta_0^T/(\delta_0^T s_0) \\ -s_0^T B_0/(s_0^T B_0 s_0) \\ \hline \delta_1^T/(\delta_1^T s_1) \\ -s_1^T B_1/(s_1^T B_1 s_1) \\ \hline \delta_2^T/(\delta_2^T s_2) \\ -s_2^T B_2/(s_2^T B_2 s_2) \\ \hline ... \\ \hline \delta_k^T/(\delta_k^T s_k) \\ -s_k^T B_k/(s_k^T B_k s_k) \end{array} \right) \tag{2.3}$$

Dès lors, il faut préciser que les données sont récupérées en creux, mais rien ne dit que F_k, G_k et les produits matriciels associés définissent des matrices creuses que l'on stockerait sous format CSC.
En reprenant l'hypothèse précédente, on a que :

$$\begin{aligned} & B_k p_k = -\nabla E(x_k) \\ & \Leftrightarrow p_k = -(I - F_k(I + G_k^T)^{-1} G_k T) \nabla E(x_k) \\ & \Leftrightarrow p_k = -\nabla E(x_k) + F_k z_k \end{aligned} \tag{2.4}$$

Ce qui nous amène finalement à utiliser BFGS de cette manière, où le premier système est défini en plein :

$$\begin{cases} (I + G_k^T F_k) z_k = G_k^T \nabla E(x_k) \\ p_k = -\nabla E(x_k) + F_k z_k \\ \text{recherche linéaire du pas} \\ x_{k+1} = x_k + \alpha p_k = x_k + s_k \\ \delta_k = \nabla E(x_{k+1}) - \nabla E(x_k) \\ B_k = I + F_k G_k^T \text{par mise à jour} \end{cases} \tag{2.5}$$

Une première esquisse en pseudo code permet de voir comment l'implémenter :

Algorithme gradient BFGS

fixer l'ensemble d'apprentissage X
initialiser les poids $w_{i,j}$ et b_j, stockés dans des vecteurs lignes, respectivement w et b
fixer le seuil ϵ
fixer un nombre d'itérations maximum, $maxit$
$grad_w = \frac{\partial E}{\partial w}$, $grad_b = \frac{\partial E}{\partial b}$, stockés en vecteur ligne
fixer n_1 comme la taille de $grad_w$ et n_2 comme celle de $grad_b$
boucle tant que erreur $> \epsilon$ et iterations $< maxit$
    erreur <- 0
    boucle pour tous les exemples $(x^{(i)}, z^{(i)})$ dans X
        résolution de $(I + G_k^T F_k) z_k = G_k^T \nabla E(x_k)$
        calcul de $p_k = -\nabla E(x_k) + F_k z_k$
        conservation de $\nabla E(x_k) = rg$
        recherche de α par **backtracking (*)**
        $grad_w = grad_w + \alpha * p_k[1 : n_1]$
        $grad_b = grad_b + \alpha * p_k[n_1 + 1 : n_1 + n_2]$
        calculer la sortie $y^{(i)}$ du réseau pour l'entrée $x^{(i)}$
        calcul de δ_k grâce à rg
        erreur <- erreur + norme2$(y^{(i)} - z^{(i)})$
        implémentation de F_k, G_k, B_k
    fin boucle pour
fin boucle tant que

Malheureusement, la méthode n'a pas abouti. En effet, l'hypothèse faite au départ nécessitait de poser F_k et G_k avec une taille fixée de départ où les parties non utilisées contenaient des 0 (ce qui ne change pas le produit matriciel). De plus, on cherche à mettre à jour les poids et seuils ce qui implique que $x_k = [w_k, b_k]$ stocké en vecteur plein. Aussi, la méthode de BFGS nécessite le stockage d'une matrice trop grande. En particulier, la transition entre les données stockées en creux et le système en plein n'a pas permis d'aboutir à un résultat.

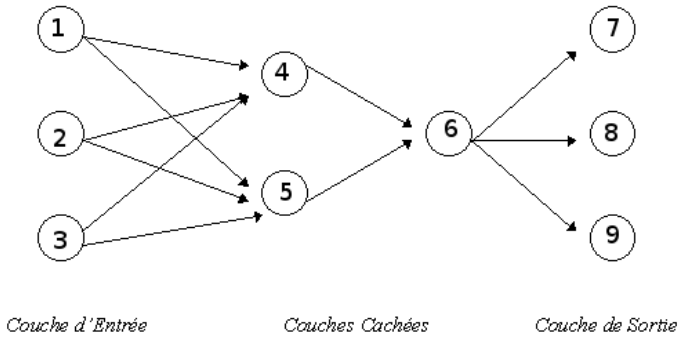
2.4 Implémentation en Fortran

Dans cette section, nous allons détailler quelques points cruciaux du code qui nous permettent d'avoir un algorithme optimisé en temps de calcul, comparativement à la première version de l'algorithme implémenté sous R.

2.4.1 Détails de l'implémentation

Parlons plus spécifiquement de la programmation. Nous allons rester ici dans le cas d'un graphe plein. Tout d'abord, comme dit précédemment, pour passer d'une couche à une autre,

on applique la fonction d'activation à la somme des entrées de chaque neurone. Les entrées des neurones étant pondérées par les poids que l'on a a ajusté au cours de l'apprentissage, on remarque qu'il s'agit en fait d'une multiplication d'un vecteur ligne par une matrice.
Explicitons cela en prenant le réseau simple suivant :



Rappelons certains éléments :
— $w_{i,j}$ est le poids sur l'arc (i, j) ;
— on note f la fonction d'activation ;
— les valeurs des neurones 1, 2 et 3 sont connus puisqu'il s'agit des neurones d'entrée.

Si l'on veut par exemple calculer la valeur du neurone 4, notée y_4, pour des entrées x_1, x_2 et x_3, on aura :

$$y_4 = f\left(w_{1,4}x_1 + w_{2,4}x_2 + w_{3,4}x_3\right)$$

De même, pour les mêmes entrées, on aura pour y_5 la valeur du neurone 5 :

$$y_5 = f\left(w_{1,5}x_1 + w_{2,5}x_2 + w_{3,5}x_3\right)$$

Soit n le nombre de neurones constituant le réseau. On pose y le vecteur de taille n défini par $y = (x_1, x_2, x_3, ..., x_k, 0, ..., 0)$ si le réseau comporte k entrées, et W matrice carrée de taille n définie comme suit :

$$\forall i, j \in 1, ..., n \quad W_{i,j} = \begin{cases} w_{i,j} & \text{si l'arc (i,j) existe} \\ 0 & \text{sinon} \end{cases}$$

En calculant le produit $y \cdot W$, on obtiendra alors $y^{(1)}$ comprenant de la position $k+1$ jusqu'à $k + 1 + m$ (en notant m le nombre de neurones sur la couche 2) les valeurs des neurones sur la couche 2.
En réitérant, c'est-à-dire en calculant $y^{(1)} \cdot W$, on obtiendra $y^{(2)}$, puis $y^{(3)}$,... De cette façon, on disposera alors des valeurs de chaque neurone du réseau : les valeurs des neurones de sortie nous donneront le résultat pour les entrées du réseau, les valeurs des neurones sur les couches intermédiaires nous seront utiles dans le calcul du gradient détaillé en (1.4.).

L'évaluation par le réseau d'une entrée spécifique se fait donc de la sorte. Cependant, remarquons une chose primordiale en explicitant la matrice W pour le réseau pris en exemple :

$$W \in \mathcal{M}_{9x9}, \quad W = \begin{pmatrix} 0 & 0 & 0 & w_{1,4} & w_{1,5} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{2,4} & w_{2,5} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{3,4} & w_{3,5} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{4,6} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{5,6} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & w_{6,7} & w_{6,8} & w_{6,9} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

La matrice W est en effet creuse, d'où la nécessité d'introduire un stockage afin de gagner en temps de calcul et en espace de stockage. Nous avons choisi le stockage CSC (Compressed Sparse Column) car Fortran stocke les éléments d'un tableau en colonnes, mais aussi car on a besoin d'un produit vecteur matrice comme détaillé en (1.7). Nous rappelons comment est défini le stockage CSC :

— Soit nnz le nombre d'élements non-nuls de la matrice A à stocker, n le nombre de neurones du réseau ;
— $Atab$ un vecteur ligne de taille nnz contenant les éléments non-nuls de la matrice A, stockés dans l'ordre en parcourant les colonnes ;
— $INDL$ un vecteur ligne de taille nnz contenant l'indice ligne dans A de chaque élément stocké dans $Atab$;
— $INDPC$ un vecteur ligne de taille $n+1$ contenant l'indice dans $Atab$ du premier élément de chaque colonne, avec par convention $INDPC(n+1) = nnz + 1$.

Nous aurons donc W stockée sous format CSC, et n'oublions pas le vecteur ligne b contenant des poids (seuils) à ajouter à chaque neurone qui n'est pas un neurone d'entrée. Ce sont ces deux tableaux qu'il nous faudra ajuster lors de l'apprentissage.

De plus, afin de naviguer plus aisément dans notre réseau de neurones, nous définissons un pointeur *couche*, de taille $q+1$ où q est le nombre de couches du réseau, avec $couche(l)$ donnant le numéro du premier neurone de la couche l, $l \in 1, ..., q$, et $couche(q+1) = n+1$.

2.4.2 Comparaison avec R

A titre de comparaison, nous avons lancé plusieurs fois l'algorithme sous R et sous Fortran, sur la classification des iris de Fisher, avec les mêmes données initiales : 80 exemples (les mêmes) dans l'exemple d'apprentissage, un seuil de 1, et un pas d'apprentissage fixe de 0.5. Ci-dessous, nous donnons un aperçu des différences en temps de calcul, pour des simulations ayant convergés :

	R	Fortran
	Temps de calcul (en s)	Temps de calcul (en s)
Simulation 1	72.64	2.417
Simulation 2	141.52	9.870
Simulation 3	52.62	7.543
Simulation 4	91.37	3.836
Simulation 5	81.61	5.781
...	...	...
Moyenne sur 20 simulations	85.71	5.234

L'implémentation en Fortran et l'optimisation du code, notamment le stockage de la matrice

creuse sous format CSC ainsi que la redéfinition des produits vecteur-matrice pour respecter ce format, nous ont permis de gagner de nombreuses secondes en temps de calcul.

2.5 Comparaison des différentes méthodes

Nous allons ici comparer uniquement l'algorithme gradient steepest descent et l'algorithme gradient stochastique, car nous n'avons pas réussi à implémenter parfaitement l'algorithme gradient BFGS.
Tout d"abord, intéressons nous à la méthode *steepest descent*. Le gradient est calculé ici en utilisant la totalité des exemples de l'ensemble d'apprentissage. Or, si l'on se place dans le cas d'un ensemble d'apprentissage de cardinal très grand, le coût de l'algorithme peut très vite devenir considérable. En effet, la mise à jour des poids ne se fait qu'une fois après avoir traité l'ensemble des exemples : dès lors, plus le cardinal sera grand, plus notre algorithme convergera lentement vers le minimum global (notons que la fonction que l'on vise à minimiser, l'erreur quadratique, est convexe).
En revanche, la méthode du gradient stochastique met à jour les poids après chaque passage d'un exemple, ce qui permet de faire baisser le coût de l'algorithme en accélérant la convergence. On parle ici de gradient "stochastique" car nous ne calculons pas le gradient global mais plutôt une approximation de ce gradient, que l'on peut définir comme "stochastique".
Lors de nos simulations, nous avons remarqué cette convergence plus lente quand on utilise l'algorithme gradient steepest descent. L'erreur ne diminue que très lentement à chaque passage de l'ensemble des exemples, comparativement à la diminution constatée lors de l'utilisation de l'algorithme basé sur le gradient stochastique. De plus, la recherche linéaire du pas (qui est en fait le pas d'apprentissage dans nos algorithmes) ralentit encore l'algorithme en nécessitant plus d'opérations élémentaires.

Chapitre 3

Simulations numériques

3.1 Présentation des exemples et prémices sous R

Avant d'implémenter un algorithme général et dans le but de comprendre comment un perceptron multi-couches fonctionne d'un point de vue informatique, nous avons réalisé un script R (voir Annexe A) spécifique aux deux exemples que nous allons traiter par la suite : l'addition de deux nombres binaires codés sur 3 bits, et la classification des iris de Fisher.
Nous nous sommes fixés deux structures de graphe : pour l'addition de nombres binaires, 6 neurones d'entrée (deux nombres codés sur 3 bits), 15 neurones sur une seule couche cachée (cf [10]) et 4 neurones de sortie (l'addition peut en effet donner en résultat un nombre codé sur 4 bits) ; pour les iris, 4 neurones d'entrée (correspondant aux 4 caractéristiques recensées : longueur et largeur des pétales, longueur et largeur des sépales), 10 neurones sur une seule couche cachée, 1 neurone de sortie (nous donnant la classe , on discutera de l'interprétation des neurones de sortie en évoquant les limites des réseaux de neurones, ainsi que du choix, judicieux ou non, d'avoir prévu un seul neurone de sortie).

Étudions maintenant ces deux exemples.

3.1.1 L'addition de nombres binaires sous R

On considère l'addition de deux nombres binaires b_1 et b_2, codés sur 3 bits. Nous allons noter

— $b_1 = m_0 + 2m_1 + 4m_2$
— $b_2 = n_0 + 2n_1 + 4n_2$
— $b_3 = b_1 + b_2 = p_0 + 2p_1 + 4p_2 + 8p_3$

Voici les résultats obtenus. Nous avons à fixer plusieurs paramètres avant de faire tourner les algorithmes : le nombre d'exemples L à utiliser comme échantillon d'apprentissage, le pas d'apprentissage α et le critère d'arrêt ϵ. Il n'existe pas de théorème de convergence, ni de règles bien définies pour choisir le pas d'apprentissage, ou le critère d'arrêt, comme dit précédemment. Le seul moyen est de faire des simulations et de voir ce qui fonctionne le mieux en pratique. Empiriquement, nous avons choisi un pas fixe, de 0.5 ou de 0.7. Les motivations sont les suivantes :

— on veut un pas d'apprentissage compris entre 0 et 1 ;
— **pas d'apprentissage α trop proche de 1 :** oscillation des poids trop grande, l'algorithme pourrait ne jamais converger ou bien nécessitera beaucoup trop d'itérations ;
— **pas d'apprentissage α trop proche de 0 :** variation des poids trop faible, risque de ne pas voir les poids converger vers les "bonnes" valeurs ou, encore une fois, en un nombre déraisonnable d'itérations.

Nous voulons donc un α pas trop proche de 0 ni de 1, ce qui explique le choix $\alpha = 0.5$ ou 0.7. Nous avons pris comme critère d'arrêt un seuil pour l'erreur à minimiser (on rappelle que nous considérons ici l'erreur quadratique), avec un nombre d'itérations maximum $itmax$.

Pour l'addition de deux nombres binaires codés sur 3 bits, nous avons 64 opérations possibles. On se propose d'utiliser un échantillon d'apprentissage de 60 exemples.

•$L = 60$, $\alpha = 0.5$, $\epsilon = 0.01$, $itmax = 50000$:

Prévision					Théorie			
p_3	p_2	p_1	p_0		p_3	p_2	p_1	p_0
0.00	0.00	0.01	0.01		0	0	0	0
0.00	0.00	0.00	0.99		0	0	0	1
0.00	0.00	1.00	0.00		0	0	1	0
0.00	0.01	1.00	0.99		0	0	1	1
0.00	1.00	0.00	0.99		0	1	0	1
0.00	1.00	1.00	0.00		0	1	1	0
0.00	1.00	1.00	1.00		0	1	1	1
0.00	0.00	0.01	1.00		0	0	0	1
0.00	0.00	1.00	0.00		0	0	1	0
0.00	0.01	1.00	0.99		0	0	1	1
0.00	1.00	0.00	0.00		0	1	0	0
0.00	1.00	0.00	1.00		0	1	0	1
0.00	1.00	1.00	0.00		0	1	1	0
0.00	1.00	1.00	1.00		0	1	1	1
1.00	0.01	0.01	0.00		1	0	0	0
0.00	0.00	0.99	0.00		0	0	1	0
0.00	0.00	1.00	1.00		0	0	1	1
0.01	0.99	0.00	0.00		0	1	0	0
0.00	1.00	0.00	1.00		0	1	0	1
0.00	1.00	1.00	0.00		0	1	1	0
0.00	0.99	1.00	1.00		0	1	1	1
1.00	0.01	0.01	0.00		1	0	0	0
1.00	0.00	0.01	1.00		1	0	0	1
0.00	0.01	1.00	1.00		0	0	1	1
0.00	0.99	0.00	0.01		0	1	0	0
0.01	1.00	0.00	1.00		0	1	0	1
0.00	0.99	0.99	0.00		0	1	1	0
0.00	0.99	0.99	1.00		0	1	1	1
1.00	0.01	0.00	0.00		1	0	0	0
1.00	0.00	0.01	1.00		1	0	0	1
1.00	0.00	0.99	0.01		1	0	1	0

Prévision					Théorie			
p_3	p_2	p_1	p_0		p_3	p_2	p_1	p_0
0.00	1.00	0.00	0.00		0	1	0	0
0.00	1.00	0.00	1.00		0	1	0	1
0.00	1.00	1.00	0.01		0	1	1	0
0.01	0.99	1.00	1.00		0	1	1	1
1.00	0.01	0.00	0.00		1	0	0	0
1.00	0.00	0.00	1.00		1	0	0	1
1.00	0.01	0.99	0.00		1	0	1	0
0.99	0.01	0.99	1.00		1	0	1	1
0.00	1.00	0.00	1.00		0	1	0	1
0.00	1.00	1.00	0.00		0	1	1	0
0.00	0.99	1.00	1.00		0	1	1	1
1.00	0.01	0.01	0.00		1	0	0	0
1.00	0.00	0.00	0.99		1	0	0	1
1.00	0.00	1.00	0.01		1	0	1	0
1.00	0.01	0.99	0.99		1	0	1	1
1.00	0.98	0.01	0.01		1	1	0	0
0.00	1.00	1.00	0.00		0	1	1	0
0.01	1.00	1.00	1.00		0	1	1	1
1.00	0.01	0.00	0.00		1	0	0	0
1.00	0.01	0.00	1.00		1	0	0	1
1.00	0.00	1.00	0.00		1	0	1	0
1.00	0.00	1.00	1.00		1	0	1	1
1.00	0.98	0.01	0.00		1	1	0	0
1.00	0.99	0.01	1.00		1	1	0	1
0.00	0.99	1.00	1.00		0	1	1	1
1.00	0.00	0.00	0.00		1	0	0	0
0.99	0.00	0.00	1.00		1	0	0	1
1.00	0.00	1.00	0.99		1	0	1	1
1.00	0.12	0.00	0.01		1	1	0	0
0.99	0.93	0.01	1.00		1	1	0	1
1.00	0.04	0.98	0.01		1	1	1	0

On remarque que, sur les 60 exemples qui constituent l'échantillon d'apprentissage, on obtient de très bons résultats : les sorties calculées par l'ordinateur sont très proches des sorties théoriques. Cependant, on note que sur les 4 additions restantes, l'ordinateur est un petit peu moins précis sur le résultat (lorsqu'en théorie on s'attend à obtenir 1 comme valeur d'un neurone de sortie, on a une valeur de 0.93 tandis que sur les additions d'apprentissage on avait des valeurs de 0.98, 0.99 ou 1), mais surtout, pour deux d'entre elles, il commet une erreur : le neurone de sortie correspondant à p_2 ne nous permet pas de conclure à une valeur de 1 pour ce

bit, en se basant sur la règle suivante :

— Soit n_S un neurone de sortie.
— Si $n_S < 0.15$, on conclut que la sortie est 0 ;
— Si $n_S > 0.85$, on conclut que la sortie est 1 ;
— Sinon, on conclut qu'on ne peut pas décider précisément de la sortie.

On pourrait conclure naïvement que l'ordinateur s'est trompé et donc qu'il n'a pas assez appris : il faudrait dans ce cas autoriser un seuil encore plus petit. Néanmoins, comme développé dans la partie sur le gradient stochastique, il est possible que l'ordinateur est en réalité "sur-appris".
On se propose donc maintenant de refaire tourner l'algorithme, mais en augmentant le seuil, de sorte que l'on ne colle pas trop aux données, et que l'on ne perde pas en généralisation :

•$L = 60$, $\alpha = 0.5$, $\epsilon = 0.5$, $itmax = 50000$:

Prévision					Théorie			
0.98	0.02	1.00	0.99		1	0	1	1
1.00	0.98	0.01	0.01		1	1	0	0
1.00	0.99	0.01	0.99		1	1	0	1
1.00	0.99	1.00	0.01		1	1	1	0

On n'affiche ici que les 4 additions ne faisant pas partie de l'ensemble d'apprentissage.

On constate alors que, tout en gardant la précision sur les exemples d'apprentissage, on arrive à mieux généraliser et donc à avoir, pour ces 4 additions, des résultats très satisfaisants. Nous reviendrons plus en détail sur cet exemple dans la section 3.2.

3.1.2 Les iris de Fisher sous R

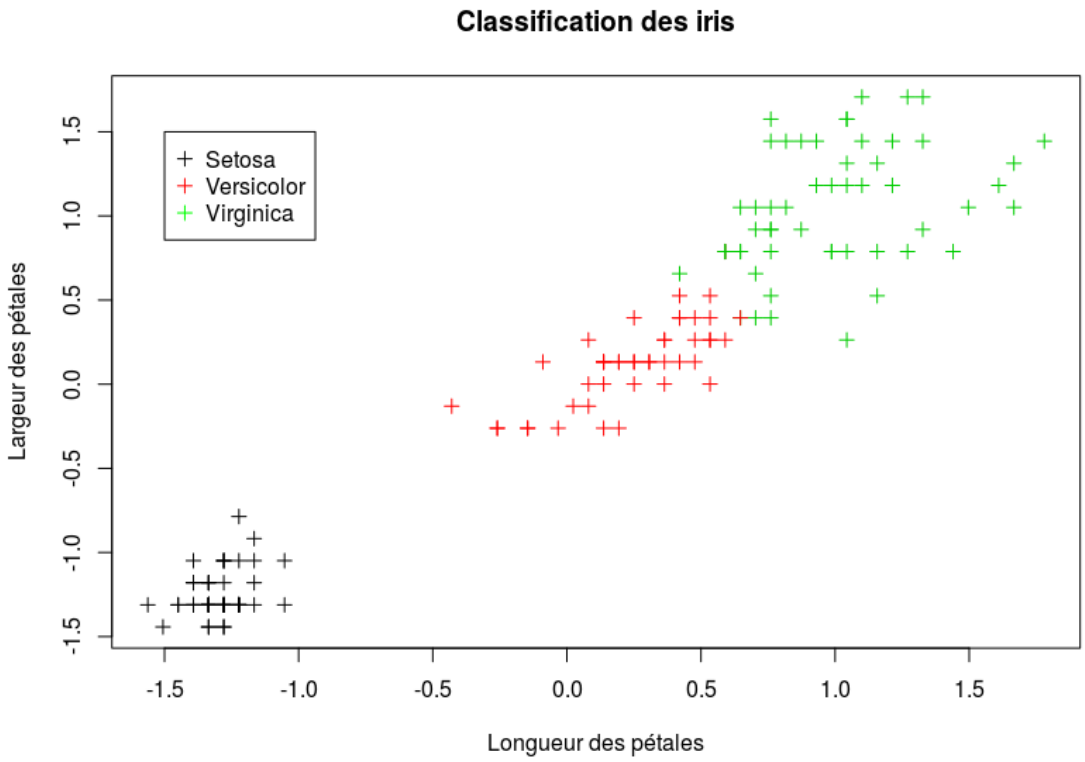
On se propose d'utiliser un réseau de neurones afin de classifier les iris de Fisher. Au préalable, on normalise les entrées du réseau, ceci dans le but de ne pas avoir de valeurs trop grandes qui pourraient faire saturer le réseau et donc empêcher l'apprentissage pour cet exemple. Ensuite, on a choisi de ne prendre qu'un seul neurone de sortie, noté n_S : il prendra sa valeur entre 0 et 1. De ce fait, comme l'on a 3 classes, on introduit une méthode pour classifier :

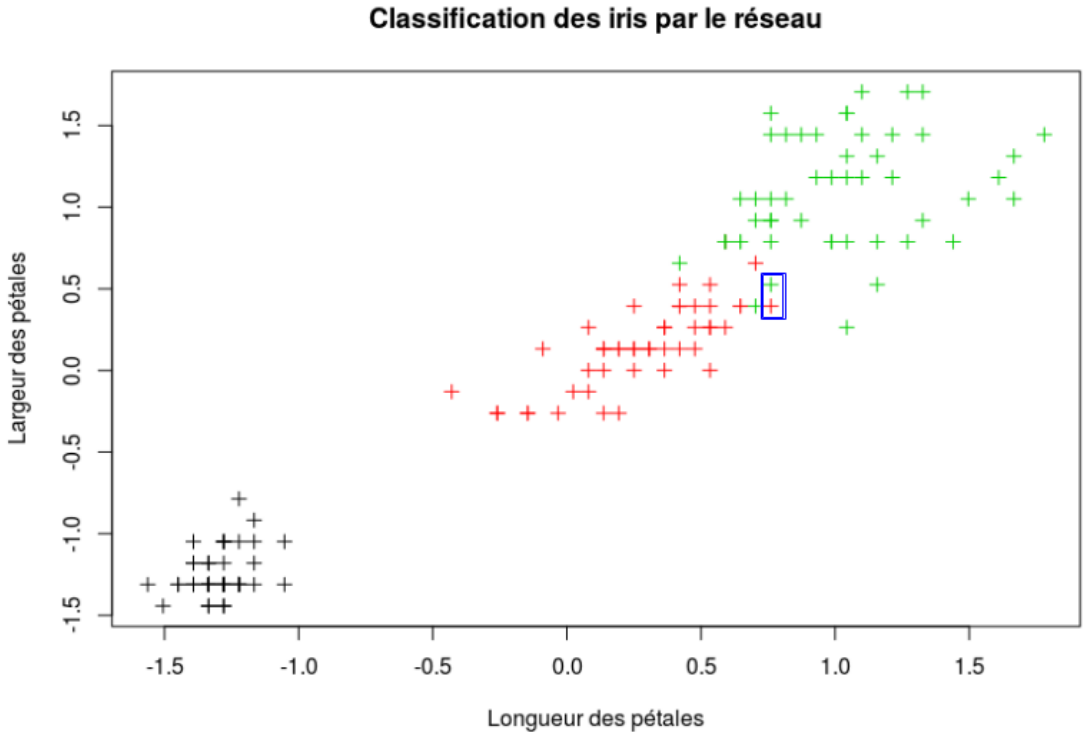
— si $n_S \in [0.3 + / - 0.05]$, on classe "setosa"
— si $n_S \in [0.6 + / - 0.05]$, on classe "versicolor"
— si $n_S \in [0.9 + / - 0.05]$, on classe "virginica"

Lorsque l'on lance l'algorithme, en s'autorisant une valeur seuil de 2 pour l'erreur (obtenue empiriquement après plusieurs essais, de sorte que la généralisation reste intéressante et efficace, c'est-à-dire sans que le réseau ne sur-apprendre et sans qu'il n'apprenne pas assez), en choisissant un pas d'apprentissage de 0.5, et en prenant 60 exemples pour l'échantillon d'apprentissage, on obtient sur les 90 iris restantes à classer :

Bien classée	Mal classée
97,78 %	2.22%

On a réussi à bien classer 88 nouvelles iris sur les 90 présentées, ce qui est très honorable. Représentons la largeur des pétales en fonction de la longueur de celles-ci, pour la classification théorique et la classification par notre réseau de neurones, afin de visualiser les 2 erreurs commises :


Classification des iris
Setosa
Versicolor
Virginica
Largeur des pétales
Longueur des pétales
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5


Classification des iris par le réseau
Largeur des pétales
Longueur des pétales
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5

3.2 L'addition de deux nombres binaires en Fortran

3.2.1 Structure du graphe

La structure du graphe de l'addition de deux nombres binaires, codés sur 3 bits, repose sur les portes logiques XOR et ET. En effet, comme il s'agit d'utiliser des variables bivalentes 0 ou 1 , il nous faut ces deux fonctions qui manipulent deux opérandes.

• Porte XOR (appelée couramment Porte OU EXCLUSIVE) : Soit 1=VRAI et 0=FAUX les deux opérandes booléennes. Il est nécessaire de déclarer la table de vérité permettant d'exposer tous les résultats possibles des combinaisons.

A	B	A OU B
1	0	1
0	1	1
1	1	0
0	0	0

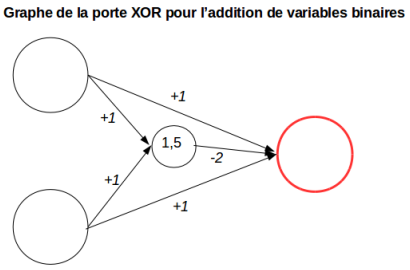
Le résultat de la porte sera vrai si une et une seule opérande est vraie.

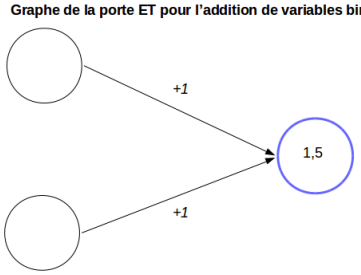
• Porte ET : Selon la même structure que pour la porte précédente, la table de vérité donne :

A	B	A ET B
1	0	0
0	1	0
1	1	1
0	0	0

Le résultat de la porte est vraie uniquement si les deux opérandes sont vraies.

Par conséquent, le graphe ci dessous sera une accumulation de ces deux portes telles que pour deux neurones sur une couche, reliés à un neurone sur la couche suivante, on utilisera le même processus que pour les opérations dans les deux tables de vérités selon si l'on utilise la porte XOR, ou la porte ET. On donnera l'exemple d'un graphe pour l'addition de 2 binaires, codés sur 2 bits afin d'avoir un graphe des plus simples possible.
Voici le graphe des deux portes utiles à la construction du graphe finale où l'on instaure un code couleur. En effet le neurone rouge est le résultat d'une porte XOR alors que le neurone bleu est celui d'une porte ET.



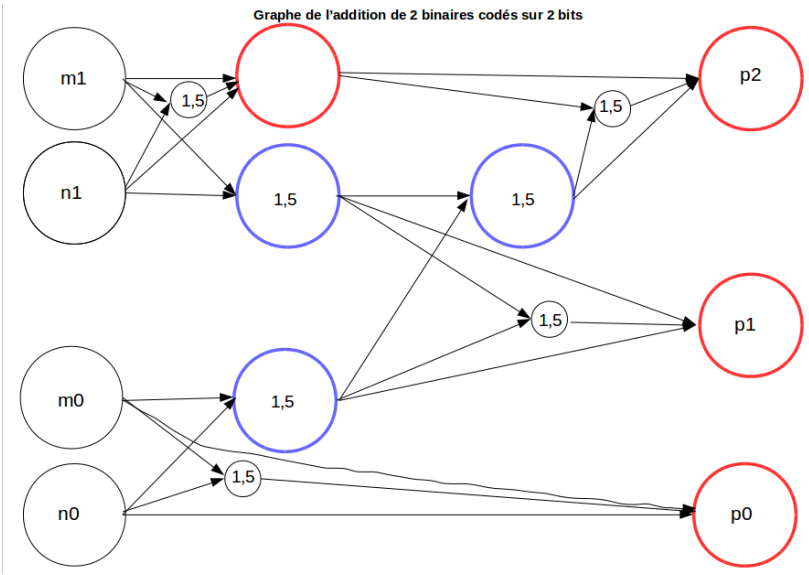


Les valeurs au dessus des liaisons sont les poids et on fixe les seuils à 0. Les valeurs dans les neurones sont les valeurs avec lesquelles on effectue la comparaison de la valeur du neurone : si

la valeur du neurone est supérieure, alors le neurone renvoie 1, sinon il renvoie 0.
On définit :

— $b_1 = m_0 + m_1$
— $b_2 = n_0 + n_1$
— $b_3 = b_1 + b_2 = p_0 + 2p_1 + 4p_2$

On peut remarquer tout de suite que p_0 ne sera influencé que par m_0 et n_0. En revanche, p_1 sera influencé par m_1 et n_1 mais aussi par m_0 et n_0 puisque ceux-ci peuvent engendrer une retenue. De même, p_2 est influencé par m_1, n_1, m_0 et n_0. Pour le graphe final, on a enlevé les poids pour ne pas surcharger le dessin, cependant avec l'introduction des deux figures ci dessus, la compréhension du graphe est possible :



A titre d'exemple, si l'on s'intéresse au deux chiffres des unités m0 et n0, la première couche cachée offre tous les résultats possibles de l'addition de ces deux valeurs grâce aux portes. En voici le résumé dans le tableau :

m0	n0	m0 ET n0 (retenue envisageable)	m0 OU n0
1	0	0	1
0	1	0	1
1	1	1	0
0	0	0	0

On en déduit bien que :
-si les deux chiffres des unités valent 1, il y aura une retenue matérialisée par la porte ET.
-si un seul des deux chiffres vaut 1, pas de retenue comme l'indique la porte ET, mais on a une valeur de 1 pour le chiffre des unités, p0, du résultat final.
-si les deux chiffres valent 0, le résultat est 0.

3.2.2 Résultats

Grâce à l'algorithme du gradient stochastique, a pu effectuer l'addition de deux nombres binaires pour un nombre d'exemple d'apprentissage de 62 sur 64 (il y a $2^6 = 64$ possibilités d'entrées dans le graphe), avec un pas d'apprentissage de 0.7 et un seuil de 2.5 (pour éviter le sur-apprentissage, cela correspond à un écart sur chaque sortie de 0.1 environ). On a utilisé ici un graphe plein : en effet, on a essayé de reconstituer le graphe précédemment exposé mais il semblerait que le fait de violer la règle selon laquelle un poids ne peut pas "sauter" de couches entraîne un problème dans la rétro-propagation de l'erreur et donc dans la mise à jour des

poids, que nous n'avons pas su résoudre.
Voici les résultats affichés dans un tableau. Pour la lisibilité, nous n'affichons qu'une vingtaine de résultats :

Graphe : 6 neurones d'entrée, 15 neurones sur une couche cachée, 4 neurones de sortie, graphe complet ;
Pas d'apprentissage : 0.7
Taille de l'ensemble d'apprentissage : 62 exemples
Seuil : 2.5
Nombre d'itérations : 24140
Temps de calcul : 11.084 secondes

Prévision					Théorie			
p3	p2	p1	p0		p3	p2	p1	p0
1.00000	0.00012	0.00004	0.99992		1.00000	0.00000	0.00000	1.00000
0.00000	0.00027	0.99183	0.99083		0.00000	0.00000	1.00000	1.00000
1.00000	0.00000	0.00000	1.00000		1.00000	0.00000	0.00000	1.00000
0.00000	0.00052	0.00610	0.02682		0.00000	0.00000	0.00000	0.00000
0.00000	0.00399	0.99940	0.00000		0.00000	0.00000	1.00000	0.00000
1.00000	0.00000	1.00000	0.00039		1.00000	0.00000	1.00000	0.00000
0.99985	1.00000	0.01758	0.00027		1.00000	1.00000	0.00000	0.00000
1.00000	0.00001	0.00325	0.99481		1.00000	0.00000	0.00000	1.00000
0.00000	0.99559	0.99960	0.99996		0.00000	1.00000	1.00000	1.00000
0.99914	0.00398	0.00012	0.00001		1.00000	0.00000	0.00000	0.00000
0.00000	1.00000	0.00000	0.98112		0.00000	1.00000	0.00000	1.00000
1.00000	0.00009	0.00070	0.00127		1.00000	0.00000	0.00000	0.00000
0.00000	0.00007	0.00019	0.98722		0.00000	0.00000	0.00000	1.00000
0.00000	1.00000	0.00000	0.99518		0.00000	1.00000	0.00000	1.00000
1.00000	0.01501	0.99718	0.97918		1.00000	0.00000	1.00000	1.00000
0.00000	0.99996	0.01610	0.02589		0.00000	1.00000	0.00000	0.00000
0.00000	1.00000	0.00000	0.00000		0.00000	1.00000	0.00000	0.00000
0.00000	0.77225	0.99783	0.99946		0.00000	1.00000	1.00000	1.00000
0.00003	0.99999	0.00000	0.99964		0.00000	1.00000	0.00000	1.00000

Nous avons utilisé 62 exemples dans l'ensemble d'apprentissage. Essayons maintenant de baisser la taille de cet ensemble d'apprentissage. A noter que les exemples sont choisis aléatoirement. Il se peut donc, si les additions ne faisant pas partie de l'ensemble d'apprentissage ont des propriétés particulières qu'aucune autre addition utilisée pour l'apprentissage ne possède, que l'ordinateur n'arrive pas à généraliser les résultats à ces additions spécifiques. Aussi, le biais présent dans l'ensemble d'apprentissage (dont on a parlé en évoquant le sur-apprentissage) peut être plus ou moins conséquent du fait de la présence ou non d'additions particulières. Un exemple trivial : imaginons que l'on ne prenne, dans notre ensemble d'apprentissage, que des additions dont le résultat peut se coder sur 3 bits. Alors, l'ordinateur aura une probabilité

non-négligeable de se tromper lorsque l'on va lui demander de donner le résultat d'une addition qui entraîne un résultat sur 4 bits.
Quels paramètres choisir ? Nous avons dans un premier temps repris les paramètres précédents, en ne changeant que la taille de l'ensemble d'apprentissage. Cependant, l'erreur commise oscillait, et l'algorithme ne semblait pas converger. Nous avons donc ajusté le pas d'apprentissage, en le baissant à 0.5 au lieu de 0.7, pour que la modification des poids soit moins brutale. Nous avons relancé l'algorithme, et l'erreur est passée sous le seuil de 2.5. Cependant, les résultats étaient mauvais : bien que le réseau donnait la bonne réponse pour les 60 exemples constituant l'ensemble d'apprentissage, il s'est trompé sur les 4 additions "nouvelles". Le seuil était donc à revoir, car il semblait un peu trop élevé. Nous avons alors réduit celui-ci empiriquement, en relançant l'apprentissage jusqu'à avoir une bonne généralisation. Voici les paramètres choisis :

Graphe : 6 neurones d'entrée, 15 neurones sur une couche cachée, 4 neurones de sortie, graphe complet ;
Pas d'apprentissage : 0.5
Taille de l'ensemble d'apprentissage : 60 exemples
Seuil : 0.5
Nombre d'itérations : 53244
Temps de calcul : 26.848 secondes

Prévision					Théorie			
p3	p2	p1	p0		p3	p2	p1	p0
1.00000	0.00049	0.00331	0.00040		1.00000	0.00000	0.00000	0.00000
0.99995	0.00000	1.00000	0.00000		1.00000	0.00000	1.00000	0.00000
1.00000	0.00000	0.99808	0.00000		1.00000	0.00000	1.00000	0.00000
0.00000	0.00000	0.99450	0.00003		0.00000	0.00000	1.00000	0.00000
0.00165	0.99798	0.00016	0.00000		0.00000	1.00000	0.00000	0.00000
0.99999	0.00000	0.00001	0.00000		1.00000	0.00000	0.00000	0.00000
0.99866	0.00000	0.00000	1.00000		1.00000	0.00000	0.00000	1.00000
0.00227	0.00000	0.99511	0.00001		0.00000	0.00000	1.00000	0.00000
0.00000	0.99943	0.00196	0.99948		0.00000	1.00000	0.00000	1.00000
0.99980	0.00153	0.00389	0.00401		1.00000	0.00000	0.00000	0.00000
0.99869	0.00126	0.99983	0.99934		1.00000	0.00000	1.00000	1.00000
0.99804	0.00000	0.00060	0.99858		1.00000	0.00000	0.00000	1.00000
1.00000	0.99855	0.00000	1.00000		1.00000	1.00000	0.00000	1.00000
0.99994	0.99947	0.00001	0.99964		1.00000	1.00000	0.00000	1.00000
0.00045	0.99681	0.99978	0.99965		0.00000	1.00000	1.00000	1.00000
0.99990	0.00151	0.99996	0.99576		1.00000	0.00000	1.00000	1.00000
0.00000	0.99907	0.07947	0.99971		0.00000	1.00000	0.00000	1.00000
0.98792	0.00000	0.00000	1.00000		1.00000	0.00000	0.00000	1.00000

3.3 Les iris de Fisher en Fortran

Intéressons-nous maintenant aux iris de Fisher avec notre algorithme implémenté en Fortran. Rappelons que ces iris proviennent de 3 classes, *setosa*, *versicolor* et *virginica*, et que la base de données contient 4 informations : la longueur et la largeur des pétales ainsi que la longueur et la largeur des sépales. Contrairement à l'addition de nombres binaires, nous ne voyons pas dans cet exemple une structure précise de notre réseau de neurones. En effet, à l'aide de portes ou de fonctions que l'on pourrait modéliser par des connexion entre neurones, le résultat d'une combinaison des données d'entrée ne serait pas interprétable. On décide donc d'utiliser un graphe plein.

Ensuite, nous avions, dans le code R, utilisé un seul neurone de sortie. Ici, nous allons procéder différemment : étant donné qu'il existe 3 classes d'iris dans notre jeu de données, nous allons utiliser 3 neurones de sortie. Si l'on note n_{S_1}, n_{S_2} et n_{S_3} ces 3 neurones, on aura les sorties suivantes :

— $n_{S_1} = 1$, $n_{S_2} = 0$ et $n_{S_3} = 0$ si la classe est *setosa*
— $n_{S_1} = 0$, $n_{S_2} = 1$ et $n_{S_3} = 0$ si la classe est *versicolor*
— $n_{S_1} = 0$, $n_{S_2} = 0$ et $n_{S_3} = 1$ si la classe est *virginica*

En utilisant comme fonction d'activation la sigmoïde logistique, les sorties du réseau ne seront pas forcément 0 ou 1 comme escompté. De ce fait, on introduit la règle suivante, comme précédemment :

— Soit n_S un neurone de sortie.
— Si $n_S < 0.1$, on conclut que la sortie est 0 ;
— Si $n_S > 0.9$, on conclut que la sortie est 1 ;
— Sinon, on conclut qu'on ne peut pas décider précisément de la valeur de n_S.

Notre graphe aura donc 4 neurones d'entrée, 3 neurones de sortie, et nous utilisons, par analogie avec la structure du graphe de l'addition de nombres binaires, 10 neurones sur la couche cachée.
Maintenant, discutons des différents paramètres que l'on va utiliser pour les simulations numériques. Pour une première simulation, nous allons prendre 100 des 150 exemples, qui vont constituer l'ensemble d'apprentissage. Ensuite, nous allons garder un pas d'apprentissage de 0.5, qui a donné de bons résultats lors de nos simulations sur l'addition de nombres binaires. Enfin, le seuil sera fixé à 0.5. Voici les résultats obtenus sur 20 des 50 iris ne faisant pas partie de l'ensemble d'apprentissage :

Graphe : 4 neurones d'entrée, 10 neurones sur une couche cachée, 3 neurones de sortie, graphe complet ;
Pas d'apprentissage : 0.5
Taille de l'ensemble d'apprentissage : 100 exemples
Seuil : 0.5
Nombre d'itérations : 26706
Temps de calcul : 5.803 secondes

Prévision				Théorie		
setosa	*versicolor*	*virginica*		*setosa*	*versicolor*	*virginica*
0.00000	0.00113	0.99998		0.00000	0.00000	1.00000
0.00000	0.00124	0.99999		0.00000	0.00000	1.00000
0.00000	0.99989	0.00000		0.00000	1.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.99999	0.00000		0.00000	1.00000	0.00000
0.00000	0.00226	1.00000		0.00000	0.00000	1.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.00551	1.00000		0.00000	0.00000	1.00000
0.00000	0.99995	0.00000		0.00000	1.00000	0.00000
0.00000	0.99990	0.00000		0.00000	1.00000	0.00000
0.00000	0.00508	1.00000		0.00000	0.00000	1.00000
0.00000	0.00099	0.99997		0.00000	0.00000	1.00000
0.00000	0.00336	1.00000		0.00000	0.00000	1.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.00457	1.00000		0.00000	0.00000	1.00000
0.00000	0.99996	0.00000		0.00000	1.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.99995	0.00000		0.00000	1.00000	0.00000

Tentons de réduire la taille de notre ensemble d'apprentissage. On veut utiliser 50 exemples, pour réduire le coût de notre algorithme. Le problème est qu'il est possible de ne pas avoir suffisamment d'exemples variés pour aboutir à un bon apprentissage. Nous gardons les mêmes paramètres que la simulation précédente, mais en abaissant le seuil de l'erreur, puisqu'il y a moins d'exemples lors de l'apprentissage.

Graphe : 4 neurones d'entrée, 10 neurones sur une couche cachée, 3 neurones de sortie, graphe complet ;
Pas d'apprentissage : 0.5
Taille de l'ensemble d'apprentissage : 50 exemples
Seuil : 0.1
Nombre d'itérations : 21950
Temps de calcul : 4.008 secondes

Prévision				Théorie		
setosa	*versicolor*	*virginica*		*setosa*	*versicolor*	*virginica*
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.99994	0.00000		0.00000	1.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.99590	0.02349		0.00000	1.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.00113	0.99997		0.00000	0.00000	1.00000
0.00000	0.99989	0.00000		0.00000	1.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.00118	0.99998		0.00000	0.00000	1.00000
0.00000	0.00456	1.00000		0.00000	0.00000	1.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.99998	0.00000		0.00000	1.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.01796	0.99659		0.00000	0.00000	1.00000
0.00000	0.00484	1.00000		0.00000	0.00000	1.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
1.00000	0.00000	0.00000		1.00000	0.00000	0.00000
0.00000	0.99988	0.00000		0.00000	1.00000	0.00000
0.00000	0.90252	0.01618		0.00000	1.00000	0.00000

Nous obtenons de très bons résultats. En effet, les erreurs commises sur chaque neurone de sortie sont de l'ordre de $2 \cdot 10^{-2}$ au plus, sauf pour un seul individu (valeur sur-lignée en rouge dans le tableau). Pour ce dernier, qui se situe dans la classe *versicolor*, on obtient une valeur de 0.90252 au lieu de 1. Néanmoins, selon la règle établie au préalable, on classe tout de même cet individu dans la classe correspondant à sa classe théorique.
De plus, dans chacune des deux simulations, aucune iris n'a été mal classée. On constate également que les iris *setosa* sont toujours classées en affectant exactement 1 au neurone de sortie correspondant, et 0 aux autres neurones correspondant aux deux autres classes. En considérant le graphe des iris qui représente la largeur des pétales en fonction de la longueur des pétales, donné précédemment, on se rend compte que les *setosa* sont linéairement séparables des autres iris, ce qui explique la plus grande précision de classification pour cette classe.

Chapitre 4

Compléments

4.1 Lien avec la régression linéaire

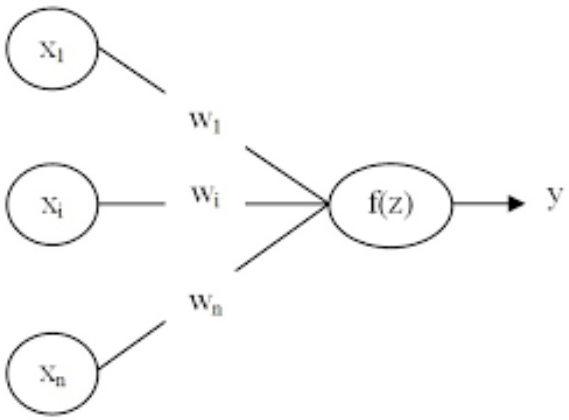
En premier lieu, rappelons la définition d'une régression linéaire. Un modèle de régression linéaire est un modèle de régression qui cherche à établir une relation linéaire entre une variable y, que l'on appelle *variable expliquée*, et une ou plusieurs variables x, appelées *variables explicatives*. Si l'on veut représenter cela en une notation scalaire, on obtient, pour un individu i :

$$y_i = \beta_0 + \beta_1 x_{i,1} + \ldots + \beta_k x_{i,k} + \epsilon_i \tag{4.1}$$

avec ϵ_i l'erreur, et $\beta_0, \ldots, \beta_k \in \mathbb{R}$ à fixer.
Reprenons alors l'équation :

$$y_j = f\left(-b_j + \sum_{(i,j)\in A} w_{i,j} \cdot y_i\right), \qquad j \in S \setminus S_1 \tag{4.2}$$

et le schéma d'un perceptron :



On voit, par identification, que les β à fixer sont ici les poids $w_{i,j}$ et b_j. Ainsi, si f est une fonction linéaire, un perceptron peut être vu comme une régression linéaire des neurones d'entrée, qui prédit la valeur du neurone de sortie.

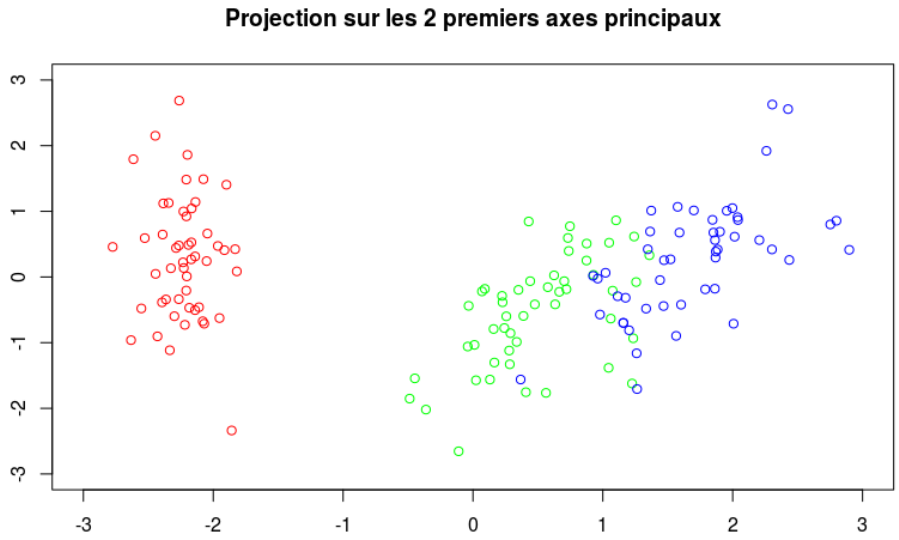
4.2 Visualisation du nuage des iris

Les iris de Fisher nous donnent un bon exemple de classification. En effet étant donné que l'on classe un certain nombre d'individus selon plusieurs caractéristiques, il est possible de se les représenter dans un espace de dimension suffisamment petit pour pouvoir l'observer.

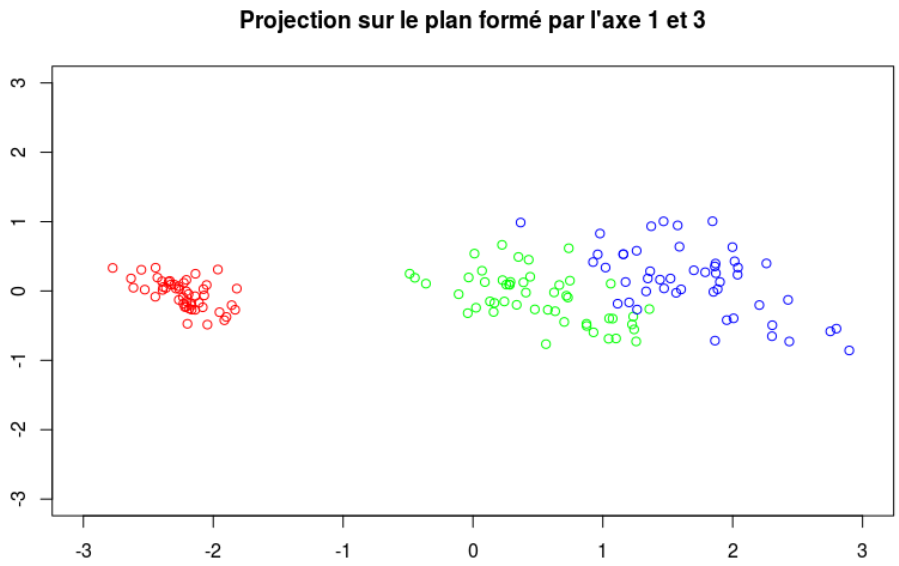
Une ACP (Analyse en Composantes Principales) permet en effet de visualiser les individus (vivant dans un espace de dimension 4) dans un espace de dimension 2 voire 3. En utilisant de jeu de données sous R, on va pouvoir projeter le nuage des individus dans un espace de dimension 2 car les deux premiers axes principaux reconstituent environ 95% de l'information, ce qu'on illustre ci dessous :

	eigenvalue	percentage of variance	cumulative percentage of variance
comp 1	2.91849782	72.9624454	72.96245
comp 2	0.91403047	22.8507618	95.81321
comp 3	0.14675688	3.6689219	99.48213
comp 4	0.02071484	0.5178709	100.00000

Dès lors, on va projeter le nuage sur les deux axes principaux en identifiant les trois groupes d'espèces déjà donnés sous R. De ce fait on va pouvoir voir si l'ACP identifie bien chaque individu dans un des 3 groupes :

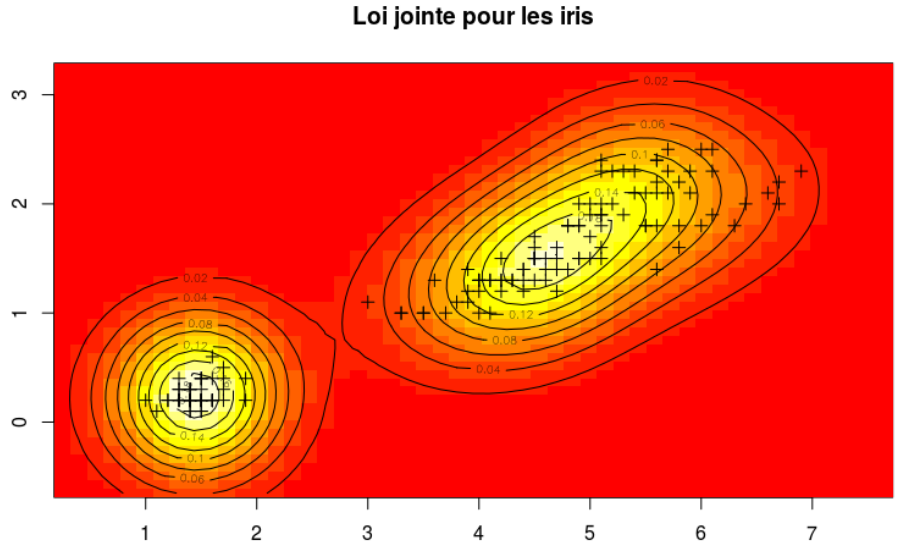


On observe la distinction d'au moins 2 groupes. Cependant le groupe de droite est constitué en réalité de 2 groupes (un en vert et un en bleu), cette représentation dans ce plan ne permet pas de distinguer précisément les individus de chaque groupe (dans le cas où l'on n'a pas coloré les individus). Mais elle permet de voir un premier indice sur la classification.
Si l'on avait décidé de projeter sur le plan formé par le premier et le troisième axe principal, on aurait obtenu ce résultat :



La visualisation des trois groupes est plus intéressante. On en a déduit que selon le plan sur lequel on projeté, la distinction des 3 groupes n'est pas si évidente que cela.

En complément de cette ACP, il est possible de voir, certes de manière brève, la classification selon les groupes grâce à une Analyse Discriminante. On observera sur le graphe la formation des groupes (de la même manière que pour la première image sous R, un groupe bien visible et deux autres trop proches pour pouvoir précisément les distinguer) la loi jointe des 150 individus uniquement pour les variables "longueur" et "largeur" des pétales.



4.3 Amélioration de l'algorithme

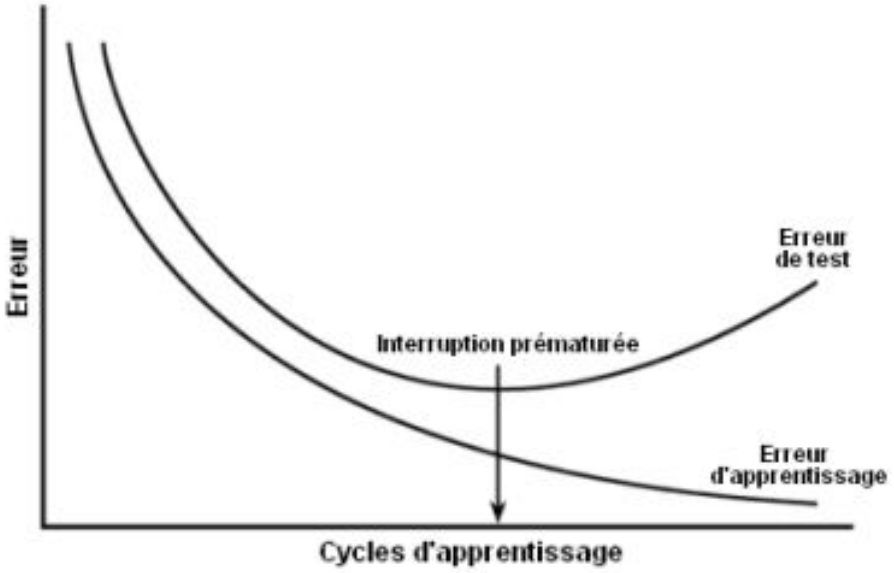
Premièrement, il faut noter que la courbe de l'erreur lors de l'apprentissage est très "accidentée". Il en résulte que de nombreux minima locaux existent. L'algorithme d'apprentissage peut donc converger vers un minimum local de la fonction d'erreur au lieu du minimum global désiré (l'algorithme basé sur le gradient stochastique permet de mieux "explorer" les courbes d'erreur [11]).
Pour éviter ces problèmes liés à une stabilisation dans un minimum local, on peut, lors de la

correction des poids, ajouter un moment d'inertie, noté η. Tout d'abord, on note $\Delta_w(k)$ la correction pour le vecteur ligne w des poids, à l'itération $k \geq 0$. La mise à jour des poids devient alors :

$$\Delta_w(k+1) = \alpha grad_w + \eta \Delta_w(k)$$

en prenant $\Delta_w(0) = 0$. On constate que la mise à jour tient compte du changement des poids effectué à l'itération précédente.

Deuxièmement, pour éviter le sur-apprentissage, il est possible de scinder l'ensemble d'apprentissage en un échantillon d'apprentissage et un échantillon test. On réalise alors l'apprentissage comme défini précédemment en utilisant uniquement l'échantillon d'apprentissage. Puis, après chaque passage de l'ensemble des exemples, on teste notre réseau de neurones (avec les poids calculés jusqu'alors) sur l'échantillon test, et on garde en mémoire l'erreur commise, que l'on notera "erreur de test". On arrête ensuite l'apprentissage dès le moment où, d'une itération à l'autre, l'erreur de test augmente. Ceci est résumé sur le graphique suivant :



4.4 Limites des réseaux de neurones

4.4.1 Choix de l'architecture et interprétation des résultats

Nous avons vu que, pour l'addition de nombres binaires, il est possible de choisir une architecture pertinente afin de bien pouvoir interpréter chaque neurone qui constitue le réseau. Ainsi, si on se réfère à l'image et au tableau de la section 3.2.1, on sait quels neurones représentent les retenues par exemples. Cependant, dans la grande majorité des cas, cela se révèle impossible. En effet, les réseaux de neurones sont souvent comparés à des boîtes noires, c'est-à-dire que l'on ne sait pas interpréter les neurones au sein du réseau. On connaît les données d'entrées, on sait interpréter les neurones de sortie, mais les valeurs des neurones situés sur les couches cachées n'ont pas de signification particulière. De plus, comment choisir une architecture pour un problème précis ? Pour l'addition de nombres binaires, on sait quels neurones seront liés et pourquoi, on connaît le nombre de couches qu'il y aura à l'avance, etc... Mais pour des problèmes de classification, pour les iris par exemple, on ne peut pas penser à un graphe doté d'une structure précise car on ne sait pas quels neurones d'entrées seront liés et pourquoi, quel résultat sortira de cette liaison...

Un autre problème se pose aussi, celui de l'interprétation des neurones de sortie. Nous avons vu

que de manière naïve, lors de nos premières expérimentations sous R, nous n'avons utilisé qu'un seul neurone de sortie pour classifier les iris, donnant la classe du neurone selon l'intervalle dans lequel se trouvait la valeur de sortie. Cependant, on peut prétendre qu'en n'utilisant un seul neurone de sortie, on perd en information : il se pourrait par exemple que l'ordinateur ne sache pas vraiment prédire le classement d'un individu dans un groupe ou l'autre. On pourrait alors avoir une interprétation qui n'affecterait pas de classe à l'individu en question.
En règle général, dans les problèmes de classification, il y aura autant de neurones de sortie que de classes, et le neurone le plus actif (ayant la valeur la plus proche de 1) donnera le numéro de la classe du neurone (cela est aussi connu sous le nom de *winner takes it all*). Néanmoins, imaginons que deux neurones de sortie aient des valeurs sensiblement égales pour un individu donné. Il ne serait pas tellement pertinent d'affecter la classe du neurone ayant la plus grande valeur tout en négligeant totalement la possibilité que l'individu en question puisse faire partie de l'autre classe. Il faudrait donc essayer de définir une interprétation qui tient compte de cela, comme par exemple "on classe cet individu possiblement dans 2 classes" ou encore "on ne classe pas cet individu".

C'est pourquoi les réseaux de neurones nécessitent beaucoup d'expérience et font souvent appel à l'intuition de l'utilisateur, au sens où le phénomène de boîte noire ne permet pas d'avoir une généralisation d'une quelconque structure du réseau, et la façon d'interpréter les résultats de sortie semble dépendre de la problématique donnée

4.4.2 Choix des paramètres

Lors des simulations numériques, nous avons régulièrement eu des cas où l'erreur ne convergeait pas. Cependant, en relançant le même algorithme avec les mêmes paramètres, il arrivait que l'erreur converge. De ce fait, nous ne savions pas vraiment comment paramétrer l'algorithme.
C'est un des problèmes des réseaux de neurones : le nombre d'exemples dans l'ensemble d'apprentissage, le pas d'apprentissage, le critère d'arrêt de l'algorithme sont autant de paramètres primordiaux pour un bon apprentissage mais qui se fixent empiriquement selon la problématique étudiée, en réalisant des simulations numériques. Là encore, cela fait appel à l'expérience et à l'intuition de l'utilisateur.

Aussi, nous avons évoqué lors des simulations numériques sur les iris de Fisher en Fortran, le fait qu'un ensemble d'apprentissage trop restreint pouvait ne pas être intéressant pour réaliser un bon apprentissage. Ceci peut faire l'objet de recherches : pour des exemples simples, dont la taille de l'ensemble d'apprentissage n'est pas trop importante, est-il possible de bien choisir les exemples à utiliser pendant l'apprentissage ? Les cas extrêmes (par exemple, pour l'addition de nombres binaires, les opérations donnant la plus grande ou la plus petite valeur de sortie) doivent-ils impérativement y figurer ?

Conclusion

L'objectif de ce Travail Encadré de Recherche était de comprendre le fonctionnement d'un réseau de neurones, en particulier le calcul du gradient par rétro-propagation permettant l'apprentissage d'un réseau à plusieurs couches, puis d'implémenter un algorithme pour ensuite réaliser des simulations numériques sur divers exemples.

Pour cela, avant de commencer la partie informatique, nous avons réalisé de nombreuses recherches pour comprendre le contexte, appréhender les définitions, mais surtout pour étudier le calcul au coeur de ce projet : la rétro-propagation du gradient. La compréhension des liens implicites entre les variables que sont les poids nous a permis de déduire la formule clé.

Nous avons ensuite implémenté un algorithme d'apprentissage, que l'on a nommé "gradient stochastique", qui est le plus utilisé en pratique. Cependant, pour faire le lien avec des algorithmes de descente étudiés en optimisation convexe, nous en avons également implémenté deux autres. Le premier est basé sur la méthode *steepest descent* : sur nos simulations, cet algorithme s'est révélé beaucoup moins performant que l'algorithme "gradient stochastique". Le deuxième s'appuie sur la méthode BFGS : malheureusement, nous n'avons pas réussi à l'implémenter correctement et, par manque de temps, nous ne sommes pas revenus sur celui-ci.

Ensuite, nous avons poursuivi notre travail en testant notre algorithme sur deux exemples : l'addition de deux nombres binaires codés sur 3 bits et la classification des iris de Fisher. Nous avons comparé les résultats obtenus avec les premiers résultats des simulations sous R, réalisées en amont afin d'appréhender la partie informatique. L'optimisation du code, notamment le stockage CSC de la matrice creuse des poids, ainsi que la redéfinition des produits entre un vecteur ligne et une matrice, nous ont permis de gagner considérablement en temps de calcul. Cependant, nous avons constaté que la convergence de l'algorithme n'était pas tout le temps établie, le choix des paramètres ayant une grande importance et donc un lien étroit avec la convergence.

Ces paramètres sont les suivants : le pas d'apprentissage, le critère d'arrêt de l'algorithme et enfin la taille de l'ensemble d'apprentissage. Cependant, aucune méthode n'existe pour donner des choix raisonnables de ces paramètres et donc l'expérience et le savoir-faire de l'utilisateur entrent en compte pour les définir au mieux. Ceci est l'une des limites des réseaux de neurones. Nous avons également critiqué le fait que ces réseaux soient des boîtes noires, ce qui restreint la compréhension à l'intérieur du réseau et l'interprétation des neurones cachés. Aussi, la difficulté du choix de l'architecture d'un réseau complique la maîtrise des réseaux de neurones.

Si nous disposions de plus de temps, nous nous serions consacrés à la reconnaissance de chiffres manuscrits, sujet complexe qui permet d'effectuer un très bon test sur des algorithmes d'apprentissage tels les réseaux de neurones [15]. Nous aurions également développé l'algorithme avec la méthode BFGS, et nous aurions eu l'occasion d'étudier plus en détail les différents choix des paramètres. Enfin, nous aurions tenté d'améliorer l'algorithme d'apprentissage autour de deux axes : la modification de la mise à jour des poids et la création d'un échantillon de test permettant d'éviter le sur-apprentissage.

Finalement, le développement des réseaux de neurones artificiels a permis l'émergence de méthodes d'apprentissage automatique beaucoup plus sophistiquées. Ainsi, il existe aujourd'hui des "réseaux de neurones convolutifs" dépassant les performances des réseaux de neurones plus traditionnels. Ceci s'inscrit dans le contexte de *deep learning*, *apprentissage profond* en français, qui est au coeur de nombreuses recherches actuelles dans le domaine de l'Intelligence Artificielle.

Bibliographie

[1] W.Pitts W.S. McCulloch. **A logical calculus of the ideas immanent in nervous activity**. *Bulletin of Mathematical Biophysics*, 1943.

[2] Donald Olding HEBB. **The Organization of Behavior**. *New York, Wiley Sons*, 1949.

[3] Frank ROSENBLATT. **The Perceptron : A Perceiving and Recognizing Automaton (Project PARA)**. janvier 1957.

[4] S. Papert M. Minsky. **Perceptrons**. *MIT Press Cambridge*, 1969.

[5] J.J. Hopfield. **Neural networks and physical systems with emergent collective computational abilities**. *Proceedings of the National Academy of Sciences of the United States of America*, Avril 1982.

[6] Yann LeCun. **Une procédure d'apprentissage pour réseau a seuil asymmetrique (a Learning Scheme for Asymmetric Threshold Networks)**. *Proceedings of Cognitiva 85*, 1985.

[7] Philippe Preux. **Fouille de données - Note de cours**. *http ://www.grappa.univ-lille3.fr/ ppreux/Documents/notes-de-cours-de-fouille-de-donnees.pdf*, 2011.

[8] Yves Lecun. **Neural networks, tricks of the trade**. *http ://citeseer.nj.nec.com/lecun98efficient.html*, 1998.

[9] Alex Krizhevsky Ilya Sutskever Ruslan Salakhutdinov Nitish Srivastava, Geoffrey Hinton. **A Simple Way to Prevent Neural Networks from Overfitting**. *https ://www.cs.toronto.edu/ hinton/absps/JMLRdropout.pdf*, 2014.

[10] Daniel Collobert. **Réseaux de neurones**. *http ://daniel.collobert.com/Textes/TextesCoursNN/*, 2003.

[11] Xavier Dupre. **Réseau de neurones, apprentissage**. *http ://www.xavierdupre.fr/app/mlstatpy/helpsphinx/c_ml/rn_6_apprentissage.html*.

[12] Marc Parizeau. **Le perceptron multicouche et son algorithme de rétropropagation des erreurs**. *https ://reussirlem1info.files.wordpress.com/2012/05/mlp.pdf*, septembre 2004.

[13] Laura Gay. **Descente Stochastique du Gradient**. *https ://www-ljk.imag.fr/membres/Marianne.Clausel/Fichiers/ProjectGay.pdf*, 2015.

[14] Sebastian Raschka. **Difference between gradient descent and stochastic gradient descent in Machine Learning**. *https ://www.quora.com/Whats-the-difference-between-gradient-descent-and-stochastic-gradient-descent*, 2015.

[15] Christopher J.C. Burges Yann LeCun, Corinna Cortes. **The MNIST database of handwritten digits**. *http ://yann.lecun.com/exdb/mnist/*.

Annexe A

Code R

```
#TER : Reseau de neurones
#Ce script R a pour but d'implementer un premier algorithme d'apprentissage spécifique
#à l'addition de deux nombres binaires codes sur 3 bits, et a la classification des
#iris de Fisher.

#TRAVAIL SUR L'ADDITION DE NOMBRES BINAIRES

#On utilisera un perceptron multi-couches, compose d'une seule couche cachee.
#Sur celle-ci, on aura 15 neurones.
#Structure du graphe: 6 neurones d'entree, 15 sur la couche cachee, 4 de sortie.

library("R.utils")
library("stringr")

#Fonctions d'activation : la fonction logistique

logist = function(x){
  return(1/(1+exp(-x)))
}

logist3 = function(x){
  return(3/(1+exp(-x)))
}

#On genere nos exemples :

X = intToBin(0:63) #les entrees
Z = intToBin(c(0:7,1:8,2:9,3:10,4:11,5:12,6:13,7:14)) #les sorties associees aux
                                                       entrees

EX = vector("list",64) #on cree une liste d'exemple : on va concatener les entrees et
                         sorties
for(i in 1:length(EX)){
  for(j in 1:10){
    EX[[i]][j] = as.numeric(str_sub(str_c(X[i],Z[i]),j,j))
  }
}

#Algo d'apprentissage
```

```
apprentissage=function(EX,seuil){
  W01 = matrix(rnorm(90,0,1),15,6)
  W12 = matrix(rnorm(60,0,sqrt(7)),4,15)
  b=rep(19,0)
  b[1:15] = rnorm(15,0,sqrt(7))
  b[16:19] = rnorm(4,0,sqrt(16))
  delta = rep(0,19)
  E = 10
  cpt = 0
  while((E > seuil) && cpt <= 30000){
    E = 0
    sample(EX)
    cpt = cpt +1
    alpha = 0.5
    for(exemple in EX){
      y1 = logist(W01%*%exemple[1:6] - b[1:15])
      y2 = logist(W12%*%(y1) - b[16:19])
      E = E + norm(exemple[7:10]-y2,"2")
      #Couche de sortie
      for(k in 1:4){
        delta[k] = y2[k]*(1-y2[k])*(exemple[k+6]-y2[k])
        W12[k,1:15] = W12[k,1:15] + alpha*delta[k]*y1[1:15]
        b[k+15] = b[k+15] - alpha*delta[k]
      }
      #Couche cachée
      for(k in 1:15){
        delta[k+4] = y1[k]*(1-y1[k])*delta[1:4]%*%W12[,k]
        W01[k,1:6] = W01[k,1:6] + alpha*delta[k+4]*exemple[1:6]
        b[k] = b[k] - alpha*delta[k+4]
      }
    }
    print(E)
  }
  return(list("W01"=W01,"W12"=W12,"b"=b,"cpt"=cpt))
}

evaluation=function(X,poids_opt){
  #De la couche d'entree a la couche cachee :
  res = logist(poids_opt$W01%*%X - poids_opt$b[1:15])

  #De la couche cachee a la couche de sortie :
  sortie = logist(poids_opt$W12%*%res - poids_opt$b[16:19])
  #On retourne avec pr?cision au centi?me
  return(round(sortie,2))
}


poids = apprentissage(EX[1:60],0.5)

eval = matrix(rep(0),64,4)
theo = matrix(rep(0),64,4)
```

```
for(i in 1:64){
  eval[i,] = evaluation(EX[[i]][1:6],poids)
  theo[i,] = EX[[i]][7:10]
}

#TRAVAIL SUR LES IRIS DE FISHER

#Données

iris2 = data.matrix(iris)
copie = iris2

#On mélange directement les exemples

melange = sample.int(150)
for(k in 1:150){
  iris2[k,] = copie[melange[k],]
}
iris2

#Normalisation et stockage des exemples

for(j in 1:(dim(iris2)[2]-1)){
  iris2[,j] = (iris2[,j]-mean(iris2[,j]))/sd(iris2[,j])
}
iris2[,5] = iris2[,5]/3

EXiris = list()
for(i in 1:dim(iris2)[1]){
  EXiris[[i]] = iris2[i,]
}
EXiris

#Graphe complet 4-10-1

apprentissageIris=function(EX,seuil){
  W01 = matrix(rnorm(40,0,1),10,4)
  W12 = matrix(rnorm(30,0,sqrt(5)),1,10)
  b=rep(11,0)
  b[1:10] = rnorm(10,0,sqrt(5))
  b[11] = rnorm(1,0,sqrt(11))
  delta = rep(0,11)
  E = 10
  cpt = 0
  while((E > seuil) && cpt <= 30000){
    E = 0
    sample(EX)
    cpt = cpt +1
    alpha = 0.5
    for(exemple in EX){
      y1 = logist(W01%*%exemple[1:4] - b[1:10])
```

```
      y2 = logist(W12%*%(y1) - b[11])
      #print(y2)
      E = E + norm(exemple[5]-y2,"2")

      #Couche de sortie
      delta[1] = y2*(1-y2)*(exemple[5]-y2)
      W12[1,1:10] = W12[1,1:10] + alpha*delta[1]*y1[1:10]
      b[11] = b[11] - alpha*delta[1]
      #Couche cachée
      for(k in 2:11){
        delta[k] = y1[k-1]*(1-y1[k-1])*delta[1]%*%W12[,k-1]
        W01[k-1,1:4] = W01[k-1,1:4] + alpha*delta[k]*exemple[1:4]
        b[k-1] = b[k-1] - alpha*delta[k]
      }
    }
    print(E)
  }
  return(list("W01"=W01,"W12"=W12,"b"=b,"cpt"=cpt))
}

evaluationIris=function(X,poids_opt){
  #De la couche d'entree a la couche cachee :
  res = logist(poids_opt$W01%*%X - poids_opt$b[1:10])

  #De la couche cachee a la couche de sortie :
  sortie = logist(poids_opt$W12%*%res - poids_opt$b[11])
  #On retourne avec precision au centieme
  return(round(sortie,2))
}

poidsiris = apprentissageIris(EXiris[1:100],2)
especes = c("setosa","versicolor","virginica")

vect_eval = rep(0,150)
vect_theo = rep(0,150)

for(i in 1:150){
  vect_eval[i] = round(3*evaluationIris(EXiris[[i]][1:4],poidsiris))
  vect_theo[i] = round(3*EXiris[[i]][5])
}

Mat_comp=matrix(c(vect_eval,vect_theo),150,2,dimnames = list(NULL,c("Prédiction",
"Espèce théorique")))
Mat_comp[,1]==Mat_comp[,2]

plot(iris2[,3],iris2[,4],main="Largeur des pétales en fonction de la longueur
des pétales",pch=3,col=vect_eval,xlab="",ylab="")



#-------------------------------------------------------------------------------

#Ce script R a pour but d'effectuer une ACP afin d'observer la classification
```

```
#des espèces en 3 groupes.
#iris de Fisher.
#TRAVAIL SUR L'ACP

x <- iris

# On va effectuer l'ACP afin de se representer les groupes de classification.

require(FactoMineR)

y=cbind(x$Sepal.Length,x$Sepal.Width,x$Petal.Length,x$Petal.Width)
acp <- PCA(y, scale.unit = T)

#Il faut utliser, dans le premier graphe, les coordonnées sur les 2 premiers axes
#puis pour le second, celles sur le premier et le troisième axe.

z <- acp$ind$coord
z1 <- matrix(z,nrow=150,ncol=4)
z2 <- z1[,-4]

plot(z2[,1][1:50],z2[,2][1:50], col="red",xlim = c(-3,3),ylim = c(-3,3),xlab="",
ylab="",main="Projection sur les 2 premiers axes principaux")
points(z2[,1][51:100],z2[,2][51:100],col="green")
points(z2[,1][101:150],z2[,2][101:150],col="blue")

plot(z2[,1][1:50],z2[,3][1:50], col="red",xlim = c(-3,3),ylim = c(-3,3),xlab="",
ylab="",main="Projection sur le plan formé par l'axe 1 et 3")
points(z2[,1][51:100],z2[,3][51:100],col="green")
points(z2[,1][101:150],z2[,3][101:150],col="blue")
```

Annexe B

Code Fortran

Module

```
!POTTIEZ Aurélien
!DUQUESNOY Marc
!Travail Encadré de Recherche
!Intelligence artificielle et apprentissage de réseaux connexionnistes

!Ce module Fortran a pour but d'implémenter les différents
!algorithmes d'apprentissage étudiés.

MODULE ter

  USE manipul_mat_mod
  implicit none

  INTERFACE logistique
     module procedure logistique_reel
     module procedure logistique_vect
  END INTERFACE logistique

  INTERFACE Dlogistique
     module procedure Dlogistique_reel
     module procedure Dlogistique_vect
  END INTERFACE Dlogistique

CONTAINS

  SUBROUTINE gen_norm(n,m,sd,val)
  !Fonction qui génère un n échantillon d'une loi normale de moyenne
  !m et d'écart-type sd.

    integer,intent(in) :: n
    integer :: i
    real, intent(in) :: m,sd
    real :: u,v
    double precision, dimension(n), intent(inout):: val
    real(16), parameter :: pi = 4 * atan (1.0_16)

    do i=1,n
```

```
      u = rand()
      v = rand()
      val(i) = m + sd * sqrt(-2 * Log(1 - u)) * Cos(2 * pi * v)
    end do


  END SUBROUTINE  gen_norm

!-------------------------------------------------------------------------------

  SUBROUTINE stockage_CSC(matrice,ATAB, INDL, INDPC)

    !Stocke les elements non nuls d'une matrice dans le Vecteur ATAB
    !Stocke les indices lignes des elements non nuls dans INDL
    !Stocke dans INDPC la position dans ATAB du 1er element non nul de chaque colonne

    double precision, dimension(:), allocatable, intent(inout) :: ATAB
    integer, dimension(:), allocatable, intent(inout) :: INDL, INDPC
    integer :: i,j,k,l,n
    double precision, dimension(:,:) :: matrice


    n=0
    !Calcul la taille du vecteur ATAB

    DO j=1,SIZE(matrice,2)
       DO i=1,SIZE(matrice,1)
          IF (matrice(i,j)/=0.) THEN
             n=n+1
          END IF
       END DO
    END DO

    ALLOCATE(ATAB(n))
    ALLOCATE(INDL(n))
    ALLOCATE(INDPC(SIZE(matrice,2)+1))

    k=0
    l=0

    DO j=1,SIZE(matrice,2)
       !Remplissage de INDPC
       IF(j==1) THEN
          INDPC(j)=1
       ELSE
          INDPC(j)=INDPC(j-1)+l
       END IF
       l=0
       !Remplissage de ATAB et INDL
       DO i=1,SIZE(matrice,1)
          IF (matrice(i,j)/=0.) THEN
             k=k+1
```

```
            l=l+1
            ATAB(k)=matrice(i,j)
            INDL(k)=i
         END IF
      END DO
   END DO
   INDPC(SIZE(matrice,2)+1)=n+1

END SUBROUTINE stockage_CSC

!-------------------------------------------------------------------------------

SUBROUTINE produit_matvect_CSC(ATAB,INDL,INDPC,x,y)

!Effectue un produit matrice vecteur avec matrice stockée sous format CSC

   double precision, dimension(:), intent(in) :: x, ATAB
   double precision, dimension(:), intent(inout) :: y
   integer, dimension(:), intent(in) :: INDPC, INDL
   integer :: k,l

   DO l=1,SIZE(x)
      y(INDL(INDPC(l):INDPC(l+1)-1))=y(INDL(INDPC(l):INDPC(l+1)-1))+
      ATAB(INDPC(l):INDPC(l+1)-1)*x(l)
   END DO

END SUBROUTINE produit_matvect_CSC

SUBROUTINE produit_vectmat_CSC(x,ATAB,INDL,INDPC,y)

!Effectue un produit vecteur matrice avec matrice stockée sous format CSC

   double precision, dimension(:), intent(in) :: x,ATAB
   double precision, dimension(:), intent(inout) :: y
   integer, dimension(:), intent(in) :: INDPC, INDL
   integer :: k,l

   DO l=1,SIZE(x)
      y(l) = y(l)+
      dot_product(ATAB(INDPC(l):INDPC(l+1)-1),x(INDL(INDPC(l):INDPC(l+1)-1)))
   END DO

END SUBROUTINE produit_vectmat_CSC

SUBROUTINE produit_vectmattrans_CSC(x,ATAB,INDL,INDPC,y)

!Effectue le produit vecteur fois matrice transposée, avec la matrice
!stockée sous format CSC

   double precision, dimension(:), intent(in) :: x,ATAB
   double precision, dimension(:), intent(inout) :: y
   integer, dimension(:), intent(in) :: INDPC, INDL
```

```
    integer :: k,l

    DO l=1,size(x)
       y = y + ATAB(INDPC(l):INDPC(l+1)-1)*x(l)
    END DO

  END SUBROUTINE produit_vectmattrans_CSC

  !Remarquons que les 3 fonctions précédentes ont été optimisées afin
  !de ne contenir plus qu'une seule boucle pour.


!-------------------------------------------------------------------------------

  FUNCTION evaluation(wtab,b,indl,indpc,exemple,couche)
!Evalue la sortie d'un réseau de neurones pour un exemple donné,
!avec la matrice des poids stockés sous format CSC et les poids-seuils
!stockés en vecteur ligne.

    double precision, dimension(:), intent(in)            :: wtab,b,exemple
    integer, dimension(:), intent(in)                     :: indl,indpc,couche
    double precision, dimension(couche(size(couche))-1)   :: evaluation
    double precision, dimension(couche(size(couche))-1)   :: stock
    integer :: l

    evaluation(:) = 0.
    !stock(:) = 0.
    evaluation(:couche(2)-1) = exemple(:couche(2)-1)

    do l=2,(size(couche)-1)
       stock(:) = 0.
       CALL produit_vectmat_CSC(evaluation,wtab,indl,indpc,stock)
       evaluation(couche(l):couche(l+1)-1) =
       logistique(stock(couche(l):couche(l+1)-1)-b(couche(l):couche(l+1)-1))
    end do

  END FUNCTION evaluation


  !-------------------------------------------------------------------------------

  FUNCTION gradient_b(b,wtab,indl,indpc,y,couche,exemple)

  !Calcule le gradient de la fonction d'erreur a minimiser E (erreur quadratique)
  ! par rapport aux poids-seuils sotckés dans b

    double precision, dimension(:),intent(in) :: b, wtab,y,exemple
    integer, dimension(:),intent(in) :: indl, indpc,couche
    double precision, dimension(couche(size(couche))-1) :: gradient_b,stock
    integer :: l, q, nb_sorties

    q = size(couche)
```

```
  nb_sorties = couche(q) - couche(q-1)

  gradient_b(:) = 0.
  gradient_b(couche(q-1):) = -Dlogistique(y(couche(q-1):couche(q)-1)) &
       *(y(couche(q-1):couche(q)-1)-exemple(size(exemple)-nb_sorties+1:))
  !print*,gradient_b

  do l=(q-2),1,-1
     stock(:) = 0.
     CALL produit_vectmattrans_CSC(gradient_b,wtab,indl,indpc,stock)
     gradient_b(couche(l):couche(l+1)-1) = Dlogistique(y(couche(l):couche(l+1)-1)) &
          *stock(couche(l):couche(l+1)-1)
  end do

END FUNCTION gradient_b

!-------------------------------------------------------------------------------


FUNCTION gradient_w(b,wtab,indl,indpc,gradient_b,y,couche)

!Calcule le gradient de la fonction d'erreur a minimiser E (erreur quadratique)
! par rapport aux poids stockés dans la matrice W sous format CSC

  double precision, dimension(:),intent(in) :: b,y,wtab,gradient_b
  integer, dimension(:) :: indl, indpc,couche
  double precision, dimension(size(wtab)) :: gradient_w
  integer :: l, q, k

  q = size(couche)
  gradient_w(:) = 0.

  do l=1,(q-2)
     do k=couche(l+1),(couche(l+2)-1)
        gradient_w(indpc(k):indpc(k+1)-1) = -y(couche(l):couche(l+1)-1)*gradient_b(k)
     end do
  end do


END FUNCTION gradient_w

!-------------------------------------------------------------------------------

SUBROUTINE melanger(matrice)

!Fonction qui mélange les colonnes d'une matrice

  integer :: j,i,n
  double precision, dimension(:,:) :: matrice
  double precision, dimension(size(matrice,1)) :: temp
  integer,dimension(3) ::  timeArray ! Holds the hour, minute, and second
```

```
    n = size(matrice,2)


    call itime(timeArray) ! Generate sequence of random number

    i = int(rand(timeArray(1)+timeArray(2)+timeArray(3))*(n-1)+1)
    temp = matrice(:,1)
    matrice(:,1) = matrice(:,i)
    matrice(:,i) = temp

    do j=2,n
       i = int(rand(0)*(n-1)+1)
       temp = matrice(:,j)
       matrice(:,j) = matrice(:,i)
       matrice(:,i) = temp
    end do

  ENDSUBROUTINE melanger


  !-------------------------------------------------------------------------------

  SUBROUTINE gradstoch(INDL,INDPC,wtab,b,couche,training_set,cpt)

!Algorithme d'apprentissage gradient stochastique

    !Structure de l'ensemble d'apprentissage :
!matrice -> parcourir les colonnes = parcourir les exemples
    !Un exemple est un vecteur avec sur ses premiers coeffs les entrees puis les
    !sorties attendues

    double precision,dimension(:,:),intent(in)          :: training_set
    integer, dimension(:)                               :: INDL, INDPC,couche
    double precision, dimension(:)                      :: b,wtab
    double precision, dimension(size(wtab))             :: grad_w
    double precision, dimension(couche(size(couche))-1) :: y,grad_b
    double precision                                    :: erreur, seuil, alpha
    integer                                             :: cpt,l
    real                                                :: start, end

!On demande le seuil

    print *,"Seuil : "
    read *, seuil

!On demande le pas

    print *,"Pas d'apprentissage (fixe ici) : "
    read *, alpha

    call CPU_TIME( start )
```

```
    erreur = 100
    cpt = 0

    do while((erreur > seuil) .AND. (cpt < 75000))
       erreur = 0

   !On mélange les exemples pour ne pas les repasser dans le même ordre
   !a chaque itération

       CALL melanger(training_set)
       cpt = cpt +1

       do l=1,size(training_set,2)

  !On évalue la sortie de notre réseau, stockée dans y

          y = evaluation(wtab,b,indl,indpc,training_set(:,l),couche)

  !On calcule l'erreur effectuée (que l'on ajoute dans erreur, l'erreur globale)

          erreur = erreur + norme2(training_set(couche(2):,l)-&
          y(couche(size(couche)-1):))/2.

  !On calcule le gradient par rapport aux poids dans la matrice W stockée
  !sous format CSC et par rapport aux poids-seuils stockés dans le
  !vecteur ligne b

          grad_b = gradient_b(b,wtab,indl,indpc,y,couche,training_set(:,l))
          grad_w = gradient_w(b,wtab,indl,indpc,grad_b,y,couche)

  !On met a jour les poids A CHAQUE PASSAGE D'UN EXEMPLE ("on-line")

          wtab = wtab - alpha*grad_w
          b = b - alpha*grad_b(couche(2):)
       end do

       print *, erreur !affichage de l'erreur
    end do

    call CPU_TIME( end )
    print *, end-start !on affiche le temps de calcul

  END SUBROUTINE gradstoch

  !-------------------------------------------------------------------------------

  SUBROUTINE grad_steepest_descent(INDL,INDPC,wtab,b,couche,training_set,cpt)

!Algorithme d'apprentissage gradient steepest descent

    !Structure de l'ensemble d'apprentissage :
!matrice -> parcourir les colonnes = parcourir les exemples
```

```
!Un exemple est un vecteur avec sur ses premiers coeffs les entrees puis les
!sorties attendues

double precision,dimension(:,:),intent(in)          :: training_set
integer, dimension(:)                               :: INDL, INDPC,couche
double precision, dimension(:)                      :: b,wtab
double precision, dimension(size(wtab))             :: grad_w
double precision, dimension(couche(size(couche))-1) :: y,grad_b
double precision                                    :: erreur, seuil, alpha,tho,c
integer                                             :: cpt,l
real                                                :: start, end

print *,"Seuil : "
read *, seuil

call CPU_TIME( start )

erreur = 100
cpt = 0
grad_b(:) = 0
grad_w(:) = 0

do while((erreur > seuil) .AND. (cpt < 75000))
   erreur = 0
   CALL melanger(training_set)
   cpt = cpt +1

   do l=1,size(training_set,2)

      y = evaluation(wtab,b,indl,indpc,training_set(:,l),couche)

      erreur = erreur + norme2(training_set(couche(2):,l)-&
      y(couche(size(couche)-1):))/2.

      grad_b = grad_b + gradient_b(b,wtab,indl,indpc,y,couche,training_set(:,l))
      grad_w = grad_w + gradient_w(b,wtab,indl,indpc,grad_b,y,couche)

   end do

   !Recherche linéaire du pas : backtracking (regle d'Armijo)

   alpha = 1
   tho = 1/3.
   c = 10**(-2.)

   do while (error(INDL,INDPC,wtab- alpha*grad_w ,b,couche,training_set) > &
        (error(INDL,INDPC,wtab,b,couche,training_set) - c*alpha*norme2(grad_w)))
      alpha = tho*alpha
   end do

   !On met a jour les poids APRES LE PASSAGE DE TOUS LES EXEMPLES
   !("batch")
```

```
     wtab = wtab - alpha*grad_w
     b = b - alpha*grad_b(couche(2):)

     print *, erreur
  end do

  call CPU_TIME( end )
  print *, end-start

END SUBROUTINE grad_steepest_descent

!-------------------------------------------------------------------------------

FUNCTION error(INDL,INDPC,wtab,b,couche,training_set)

!Evalue l'erreur gloable commise sur un ensemble d'exemples

  double precision,dimension(:,:),intent(in)          :: training_set
  integer, dimension(:)                               :: INDL, INDPC,couche
  double precision, dimension(:)                      :: b,wtab
  double precision, dimension(couche(size(couche))-1) :: y
  integer                                             :: l
  double precision                                    :: error

  error = 0

  do l=1,size(training_set,2)

     y = evaluation(wtab,b,indl,indpc,training_set(:,l),couche)
     error = error + norme2(training_set(couche(2):,l)-y(couche(size(couche)-1):))/2.

  end do


END FUNCTION error

!-------------------------------------------------------------------------------

!Les fonctions suivantes définissent la structure des graphes en initialisant
!la matrice des poids W sous format CSC et le vecteur ligne des poids-seuils b
!On initialise également le pointeur couche qui permet de naviguer plus aisément
!dans le réseau.

SUBROUTINE iris(INDL,INDPC,wtab,b,couche)
  !Agencement du graphe complet 4-10-3

  integer, dimension(:),allocatable, intent(inout) :: INDL,INDPC,couche
  double precision, dimension(:),allocatable, intent(inout) :: b,wtab
  integer :: i

  allocate(INDPC(18))
```

```
  allocate(INDL(70))
  allocate(wtab(70))
  allocate(b(13))
  allocate(couche(4))

  !Initialisation des poids selon les règles données par Yann LeCun et al.

  CALL gen_norm(40,0.,sqrt(1.),wtab(:40))
  CALL gen_norm(30,0.,sqrt(4.),wtab(41:70))
  CALL gen_norm(10,0.,sqrt(4.),b(:10))
  CALL gen_norm(3,0.,sqrt(10.),b(11:13))

  !Initialisation des tableaux INDL, INDPC et COUCHE sachant la structure du graphe

  couche(1) = 1
  couche(2) = 5
  couche(3) = 15
  couche(4) = 18

  INDPC = (/ 1,1,1,1,1,5,9,13,17,21,25,29,33,37,41,51,61,71 /)

  do i=1,10
     INDL(4*(i-1)+1:4*i) = (/ 1,2,3,4 /)
  end do

  do i=1,3
     INDL(41+10*(i-1):40+10*i) = (/ 5,6,7,8,9,10,11,12,13,14 /)
  end do

END SUBROUTINE iris

!-------------------------------------------------------------------------------

SUBROUTINE binaire(INDL,INDPC,wtab,b,couche)
  !Agencement du graphe complet 6-15-4

  integer, dimension(:),allocatable, intent(inout) :: INDL,INDPC,couche
  double precision, dimension(:),allocatable, intent(inout) :: b,wtab
  integer :: i

  allocate(INDPC(26))
  allocate(INDL(150))
  allocate(wtab(150))
  allocate(b(19))
  allocate(couche(4))

  !Initialisation des poids selon les règles données par Yann LeCun et al.

  CALL gen_norm(90,0.,sqrt(1.),wtab(:90))
  CALL gen_norm(60,0.,sqrt(6.),wtab(91:150))
  CALL gen_norm(15,0.,sqrt(6.),b(:15))
  CALL gen_norm(4,0.,sqrt(15.),b(16:19))
```

```
  !Initialisation des tableaux INDL, INDPC et COUCHE sachant la structure du graphe

  couche(1) = 1
  couche(2) = 7
  couche(3) = 22
  couche(4) = 26

  INDPC = (/ 1,1,1,1,1,1,1,7,13,19,25,31,37,43,49,55,61,67,73,79,85,91,106,121,136,
  151 /)

  do i=1,15
     INDL(6*(i-1)+1:6*i) = (/ 1,2,3,4,5,6 /)
  end do

  do i=1,4
     INDL(91+15*(i-1):90+15*i) = (/ 7,8,9,10,11,12,13,14,15,16,17,18,19,20,21 /)
  end do

END SUBROUTINE binaire

!-------------------------------------------------------------------------------

!Tentative d'implémentation de la structure complexe étudiée du graphe de l'addition
!de deux nombres binaires codés sur 3 bits

SUBROUTINE binaire2(INDL,INDPC,wtab,b,couche)
  !Agencement du graphe non complet 6-4-4-2-4

  integer, dimension(:),allocatable, intent(inout) :: INDL,INDPC,couche
  double precision, dimension(:),allocatable, intent(inout) :: b,wtab
  integer :: i

  allocate(INDPC(21))
  allocate(INDL(28))
  allocate(wtab(28))
  allocate(b(14))
  allocate(couche(6))

  !Initialisation des poids

  CALL gen_norm(28,0.,10.,wtab)
  CALL gen_norm(14,0.,5.,b)

  !Initialisation des tableaux INDL, INDPC et COUCHE sachant la structure du graphe

  couche(1) = 1
  couche(2) = 7
  couche(3) = 11
  couche(4) = 15
  couche(5) = 17
  couche(6) = 21
```

```
    INDPC = (/1,1,1,1,1,1,1,3,5,7,9,11,13,15,17,19,21,23,25,27,29 /)
    INDL =  (/1,4,2,5,2,5,3,6,1,4,7,8,7,8,9,10,11,12,13,14,15,16,13,14,9,10,3,6 /)



  END SUBROUTINE binaire2


  !Fonctions annexes-----------------------------------------------------------------------

  !Les fonctions suivantes sont :
  !    - la sigmoide logistique appliquee a un reel ou a un vecteur
  !    - la derivee de la sigmoide logistique appliquee a un reel ou a un vecteur

  FUNCTION logistique_reel(x)
    double precision :: logistique_reel,x

    logistique_reel = 1/(1+exp(-x))
  END FUNCTION logistique_reel

  FUNCTION logistique_vect(x)
    double precision,dimension(:) :: x
    double precision,dimension(size(x)) :: logistique_vect

    logistique_vect = 1/(1+exp(-x))
  END FUNCTION logistique_vect

  FUNCTION Dlogistique_reel(x)
    double precision :: x,Dlogistique_reel

    Dlogistique_reel = x*(1-x)
  END FUNCTION Dlogistique_reel

  FUNCTION Dlogistique_vect(x)
    double precision,dimension(:) :: x
    double precision,dimension(size(x)) :: Dlogistique_vect,ones

    ones(:) = 1.

    !Ici, on multiplie les vecteurs par * volontairement, pour avoir une multiplication
    !terme a terme (produit d'Hadamard)

    Dlogistique_vect = x*(ones-x)
  END FUNCTION Dlogistique_vect

  !-------------------------------------------------------------------------------------

END MODULE ter
```

Programme addition de nombres binaires

```
!POTTIEZ Aurélien
!DUQUESNOY Marc
!Master MAS - Université de Lille
!Travail Encadré de Recherche
!Intelligence artificielle et apprentissage de réseaux connexionnistes

!Ce programme Fortran a pour but de réaliser l'apprentissage de
!l'addition de deux nombres binaires codés sur 3 bits, en utilisant
! l'algorithme gradient stochastique et gradient steepest descent.

PROGRAM addition_binaire
  USE TER

  IMPLICIT NONE

  double precision, dimension(:,:), allocatable :: training_set,evalfinal,evalfinal2
  integer, dimension(:),allocatable             :: INDL,INDPC,couche
  double precision, dimension(:),allocatable    :: b,wtab,stock,copie1,copie2,stock2
  integer :: cpt,i,cpt2

  !Tout d'abord, on lit les données d'un fichier texte et on les stocke dans training_se

  CALL lire("addition.txt",training_set)

  !On fait appel a la fonction binaire pour avoir la structure du graphe

  CALL binaire(INDL,INDPC,wtab,b,couche)

  !On copie wtab et b pour les utiliser dans gradient steepest descent
  !indépendamment des résultats de gradient stochastique

  copie1 = wtab
  copie2 = b

  !On lance nos deux algorithmes d'apprentissage (avec 60 exemples d'apprentissage)

  CALL gradstoch(INDL,INDPC,wtab,b,couche,training_set(:,1:60),cpt)
  CALL grad_steepest_descent(INDL,INDPC,copie1,copie2,couche,training_set(:,1:60),cpt2)

  !On affiche le nombre d'itérations nécessaires

  print*, "Nb iterations : gradstoch = ", cpt, "gradsteepest = ", cpt2


  allocate(evalfinal(size(training_set,2),8))
  allocate(stock(couche(size(couche))-1))
  allocate(evalfinal2(size(training_set,2),8))
  allocate(stock2(couche(size(couche))-1))

  !On stocke dans une matrice les resultats prévisionnels et les résultats théoriques,
  !et ce pour chaque algorithme
```

```
  do i=1,size(training_set,2)
     stock = evaluation(wtab,b,indl,indpc,training_set(:,i),couche)
     evalfinal(i,:4) = stock(couche(size(couche)-1):)
     evalfinal(i,5:) = training_set(7:,i)

 stock2 = evaluation(copie1,copie2,indl,indpc,training_set(:,i),couche)
     evalfinal2(i,:) = stock2(couche(size(couche)-1):)
     evalfinal2(i,4:) = training_set(7:,i)
  end do

  !On affiche les résultats

  CALL afficher(evalfinal)
  CALL afficher(evalfinal2)

END PROGRAM addition_binaire
```

Programme classification des iris

```
    !POTTIEZ Aurélien
!DUQUESNOY Marc
!Master MAS - Université de Lille
!Travail Encadré de Recherche
!Intelligence artificielle et apprentissage de réseaux connexionnistes

!Ce programme Fortran a pour but de réaliser l'apprentissage de
!la classification des iris de Fisher en utilisant
! l'algorithme gradient stochastique et gradient steepest descent.

PROGRAM reseau_iris
  USE TER

  IMPLICIT NONE

  double precision, dimension(:,:), allocatable :: training_set,evalfinal,evalfinal2
  integer, dimension(:),allocatable             :: INDL,INDPC,couche
  double precision, dimension(:),allocatable    :: b,wtab,stock,copie1,copie2,stock2
  integer :: cpt,i,cpt2

  !Tout d'abord, on lit les données d'un fichier texte et on les stocke dans training_se

  CALL lire("iris.txt",training_set)

  !On fait appel a la fonction iris pour avoir la structure du graphe

  CALL iris(INDL,INDPC,wtab,b,couche)

  !On copie wtab et b pour les utiliser dans gradient steepest descent
  !indépendamment des résultats de gradient stochastique

  copie1 = wtab
  copie2 = b
```

```
  !On lance nos deux algorithmes d'apprentissage (avec 80 exemples d'apprentissage)

  CALL gradstoch(INDL,INDPC,wtab,b,couche,training_set(:,1:80),cpt)
  CALL grad_steepest_descent(INDL,INDPC,copie1,copie2,couche,training_set(:,1:80),cpt2)

  !On affiche le nombre d'itérations nécessaires

  print*, "Nb iterations : gradstoch = ", cpt, "gradsteepest = ", cpt2

  allocate(evalfinal(size(training_set,2),6))
  allocate(evalfinal2(size(training_set,2),6))
  allocate(stock(couche(size(couche))-1))
  allocate(stock2(couche(size(couche))-1))

  !On stocke dans une matrice les resultats prévisionnels et les résultats théoriques,
  !et ce pour chaque algorithme

  do i=1,size(training_set,2)
     stock = evaluation(wtab,b,indl,indpc,training_set(:,i),couche)
     evalfinal(i,:) = stock(couche(size(couche)-1):)
     evalfinal(i,4:) = training_set(5:,i)

     stock2 = evaluation(copie1,copie2,indl,indpc,training_set(:,i),couche)
     evalfinal2(i,:) = stock2(couche(size(couche)-1):)
     evalfinal2(i,4:) = training_set(5:,i)
  end do

  !On affiche les résultats

  CALL afficher(evalfinal)
  CALL afficher(evalfinal2)



END PROGRAM reseau_iris
```