Implémentation de l'algorithme de rétropropagation sur un réseau neuronal

Rapport de Projet

Raboude Gérard Jean-Maurice
Picasarri-Arrieta Lucas
Coulet Gaspard
Himeur Areski
Guyard Félix

Département Informatique
Université de Montpellier
Mai 2018

Nous tenons à remercier toute personne qui a d'une manière ou d'une autre contribué au bon aboutissement de ce projet. Nous tenons tout particulièrement à témoigner notre reconnaissance à nos deux encadrants, Messieurs Federico Ulliana et Dino Ienco qui nous ont guidés tout au long de ce projet, ont été à l'écoute, ont pris le temps de relire ce rapport et ont toujours été de très bons conseils.

Table des matières

Chapitre 1

Introduction

L'intelligence artificielle, l'ensemble de théories et de techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence, n'a jamais été autant présente dans nos vies quotidiennes qu'aujourd'hui. Elles sont développées souvent d'après les travaux d'un précurseur de ce domaine, Alan Turing. Il déclare en 1950 : « Plutôt que de tenter de simuler un cerveau adulte, pourquoi ne pas simuler un cerveau d'enfant ? En le confrontant à une éducation appropriée, nous obtiendrons notre cerveau adulte. » [1].

Ce sont depuis les années 2000 que les travaux sur cet apprentissage automatisé voient leurs résultats dépasser toutes les attentes. Ce dossier étudiera ce qu'est l'apprentissage supervisé, un exemple d'implantation et les résultats obtenus par cette implémentation.

1.1 Problématique

L'une des problématiques de l'intelligence artificielle est la prise de décision. Pour une situation donnée, il est demandé à l'intelligence artificielle de savoir prendre une décision en adéquation avec l'environnement. Prenons pour exemple une intelligence capable de trier nos messages entre courriel indésirable et légitime. L'objectif de notre projet est de pouvoir répondre à cette problématique et à toutes les problématiques semblables de classification de données.

1.2 Apprentissage supervisé

Ce projet doit permettre une classification d'un ensemble de données ayant des propriétés communes selon des règles inconnues. Pour cela, nous allons étudier le principe d'apprentissage supervisé. L'appren-

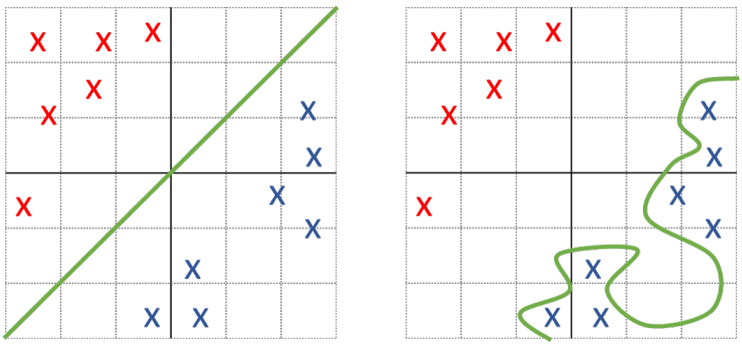

FIGURE 1.1 – Généralisation et Surinterprétation - Schéma d'exemple simplifié de séparation des données [croix colorées] par une ligne

tissage supervisé est une technique d'apprentissage qui extrait automatiquement des règles à partir d'une base de données d'apprentissage contenant des exemples déjà validés.

Formellement, un algorithme ou une structure d'apprentissage supervisé doit pouvoir déterminer une fonction de prédiction permettant pour des entrées données en exemple de retourner la sortie qui lui est associée dans le but de généraliser l'utilisation de cette fonction à de nouvelles observations. Lorsque l'ensemble des valeurs de sortie est fini, on parle d'un problème de classification, qui revient à attribuer une étiquette à chaque entrée. La fonction de prédiction est alors appelée un classificateur. Cette fonction ne doit donc pas correspondre trop étroitement aux données en exemple pour éviter une surinterprétation [3] des exemples empêchant l'extension de cette fonction à des observations futures.

Dans la figure 1.1, à gauche se trouve une fonction affine simple permettant de séparer les données en deux dimensions "rouges" des données "bleues". Cette fonction permet de généraliser simplement à de nouvelles données. À droite, la fonction de prédiction est complexe et surinterprète les exemples, une généralisation à de nouvelles observations sera erronée.

1.3 État de l'art

La structure d'apprentissage supervisé la plus connue est le réseau neuronal artificiel dont la conception est schématiquement inspirée du fonctionnement du cerveau humain. Cette technique souvent nommée « Deep Learning » pour son apprentissage automatisé complexe et pourtant sa structure simple, a connu une recrudescence d'intérêt en 2012. En effet, lors de l'édition 2012 de l'« ImageNet Large Scale Visual Recognition Challenge », une compétition annuelle de reconnaissance visuelle à grande échelle internationale, la technique des réseaux de neurones artificielle a dominé toutes les autres techniques et continue à ce jour à le faire.

De nombreuses bibliothèques permettent aujourd'hui à n'importe quel passionné ou professionnel d'utiliser simplement un réseau sur ses propres données. L'un des plus connus est TensorFlow, un outil open source d'apprentissage automatique développé par Google.

FIGURE 1.2 – Logo de TensorFlow : plus d'informations sur **tensorflow.org**

Il existe aussi d'autres bibliothèques et « APIs » permettant de créer et manipuler des réseaux neuronaux. Les plus connues et plus utilisées sont : Caffe , deeplearn.js , openNN et Theano.

1.4 Le cahier des charges

Le but de ce projet est d'implémenter en C++ une structure de données et un ensemble d'algorithmes modélisant un réseau de neurones artificiel afin de réaliser un apprentissage supervisé pour la classification de données. Il est à noter que ce projet utilise les bases théoriques de la statistique, mais n'utilise pas de bibliothèques spécialisées pour les réseaux neuronaux tels que TensorFlow. L'exercice étant de créer intégralement le réseau avec pour cahier des charges :

- La création d'un Neurone simple à n entrées implémentant au minimum :
  - Une fonction d'activation
  - Un coefficient d'apprentissage
  - La rétropropagation.
- Tester sur un neurone l'apprentissage de portes logiques pour observer les résultats.

— L'implémentation d'un biais et de poids aléatoires sur le Neurone et comparer avec les résultats précédents.
— L'implémentation d'un réseau neuronal utilisant l'algorithme de rétropropagation configurable selon plusieurs paramètres :
  — Fonctions d'activation
  — Coefficient d'apprentissage
  — Gestion de l'erreur

Pour finir, le projet doit avoir un code clair, commenté et respectant les conventions d'écritures.

1.5 Organisation

Ce projet a été réalisé avec une coopération étroite entre chacun des auteurs de ce dossier. Les tâches étaient réparties équitablement selon les motivations de chacun puis effectuées en parallèle. Pour nous aider dans l'organisation, nous avons mis en place :

— Pour travailler simultanément sur les mêmes fichiers, un serveur basé sur le logiciel libre Gitlab hébergé à domicile. Ce logiciel permet de tirer profit de l'outil libre Git et ainsi de pouvoir modifier simultanément les mêmes fichiers avant de partager les modifications effectuées. Le lien vers notre projet est **git.coulet.xyz/Gaspard/projet_s4.git**
— En dehors de l'université, pour faciliter la communication, un serveur Discord, un logiciel de communication textuelle et vocale de groupe.
— Afin que nos tuteurs puissent voir l'avancée de notre travail au propre, nous avons publié l'avancement de notre travail sur un Github public. **github.com/k1nd0ne/Neural_Networks.git**
— Pour générer une documentation de notre code, nous avons utilisé Docxygen et publié la documentation sur un de nos sites personnels. **k1nd0ne.com/docxygen**

Nous utilisons le compilateur g++ ainsi que Linux comme système d'exploitation pour la programmation. Les éditeurs de textes utilisés sont Atom, Vim et Xcode.

Chapitre 2

Préalable mathématique

2.1 Le Perceptron

2.1.1 Classificateur linéaire

Un perceptron est un classificateur linéaire. C'est-à-dire un classificateur qui calcule une sortie par combinaison linéaire des entrées. Un classificateur linéaire binaire peut se voir comme un séparateur dans un espace de grande dimension [les entrées] par un hyperplan défini par deux vecteurs : tous les points d'un côté de l'hyperplan sont dans la première classe, les autres sont dans la seconde. Cet hyperplan est appelé hyperplan séparateur. On peut alors étendre aux classificateurs linéaires n-aire définis par n vecteurs.
L'apprentissage supervisé aura alors pour objectif de déterminer la valeur des poids w_i.

2.1.2 Structure

Un perceptron ou neurone artificiel est une représentation simplifiée d'un neurone biologique. Il peut être vu comme un réseau de neurones ne possédant qu'un neurone. Un neurone artificiel à k entrés, tel

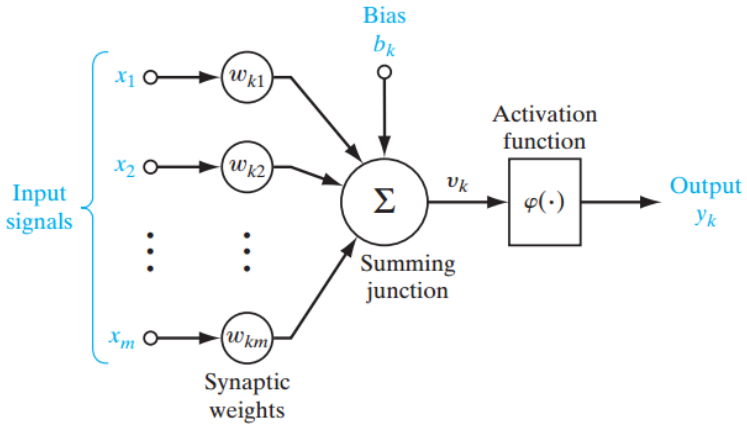

FIGURE 2.1 – Schéma d'un perceptron

qu'illustré dans la figure, 2.1 peut-être vu comme une fonction $f(x_1, ..., x_k; w_1, ..., w_k) = y$ où les x_i représentent les entrées et w_i les poids associés à chaque entrée (pour i de 1 à k). Le neurone reçoit un signal s proportionnel aux entrées $x_1, ..., x_k$ et aux paramètres de poids $w_1, ..., w_k$:

$$\sum_i x_i \cdot w_i$$

Il émet ensuite un signal entre 0 [éteint] et 1 [actif] qui est déterminé par une fonction appelée fonction d'activation.

$$y = activation(s)$$

2.1.3 Fonction d'activation

L'important pour une fonction d'activation est qu'elle soit dérivable en tout point. La fonction d'activation que nous allons principalement utiliser et la fonction sigmoïde de la figure 2.2.

$$Sigm(x) = \frac{1}{1 + e^{-kx}}$$

$Sigm'(x) = kx(1 - x)$ ce qui nous donne pour $\frac{\partial}{\partial_s} act(s) = ks(1 - s)$

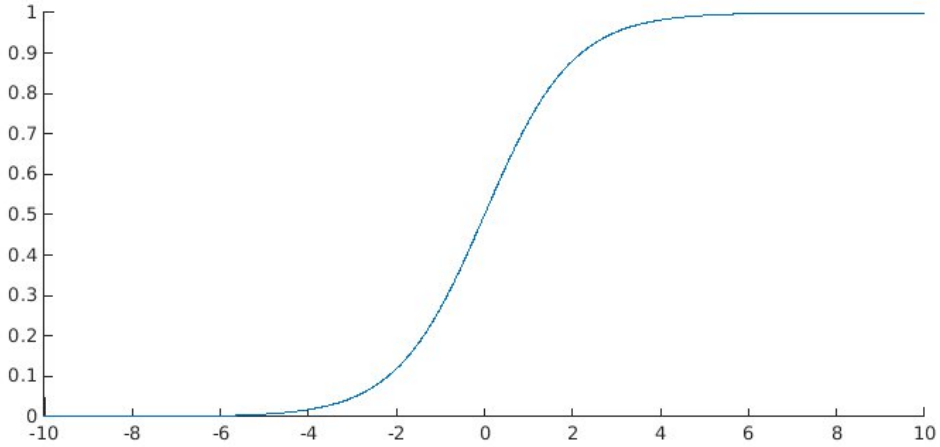

FIGURE 2.2 – Représentation de la fonction sigmoïde

2.1.4 Rétropropagation

Un neurone artificiel est initialisé avec des poids aléatoires. L'algorithme de rétropropagation détermine la valeur optimale des poids avec un apprentissage supervisé.
Il prend donc en paramètre un vecteur d'entrée I $(x_1, ..., x_i)$, et un vecteur de valeurs de sortie souhaitée pour I noté o. Il fait alors évoluer les poids afin que y se rapproche de la sortie souhaitée. Le facteur de cette avancée dépend du taux d'apprentissage (η). Lors de l'apprentissage supervisé, il est nécessaire d'effectuer plusieurs rétropropagations. La fonction err calcule l'erreur quadratique du neurone artificiel sur un exemple (I,o) et est définie ainsi :

$$(o - y)^2$$

L'algorithme de rétropropagation consiste à modifier les poids en fonction du gradient de la fonction erreur en I. En effet si le gradient est positif, la fonction d'erreur est strictement croissante en I et inversement.

$\nabla_{err} = (\frac{\partial}{\partial w_1}_{err}, ..., \frac{\partial}{\partial w_k}_{err})$ où chaque dérivée partielle décrit l'erreur commise par le neurone à cause du poids w_i .

Enfin, la quantité $\eta \cdot \frac{\partial err}{\partial w_i}$ est soustraite à w_i une fois que la valeur y a été calculée pour l'exemple (I, o), où η est le taux d'apprentissage.

La dérivée partielle $\frac{\partial err}{\partial w_i}$ correspond à la formule suivante :

$$x_i \cdot s(1 - s)(o - y)$$

Pour comprendre d'où vient cette quantité, il suffit d'écrire l'erreur comme une composition de fonctions

$$err(a), \quad a = act(s), \quad s = sig(x_1, ..., x_k \quad ; \quad w_1, ..., w_k)$$

et observer que pour dériver la fonction, il suffit d'appliquer la règle de dérivation en chaîne ce qui nous donne

$$\frac{\partial}{\partial w_i} err(a) = \frac{\partial}{\partial a} err(a) \cdot \frac{\partial}{\partial s} act(s) \cdot \frac{\partial}{\partial w_j} sig(x_1, ..., x_k \quad ; \quad w_1, ..., w_k)$$

et ainsi résoudre chaque terme séparément.

2.2 Réseau neuronal

2.2.1 Présentation

Un réseau neuronal est organisé en couches ou « layer » en anglais. Une couche est un groupe de neurones uniformes sans connexion les uns avec les autres. Une couche réalise une transformation vectorielle à partir d'un vecteur d'entrée. Ainsi une couche ne peut utiliser que les sorties des couches précédentes. Les dimensions d'entrée et de sortie peuvent être différentes.
On différencie la couche d'entrée, des couches dites "cachées" permettant la transformation vectorielle et la couche de sortie telle que dans la figure 2.3. Les entrées d'une couche cachée sont les sorties de la couche précédente.

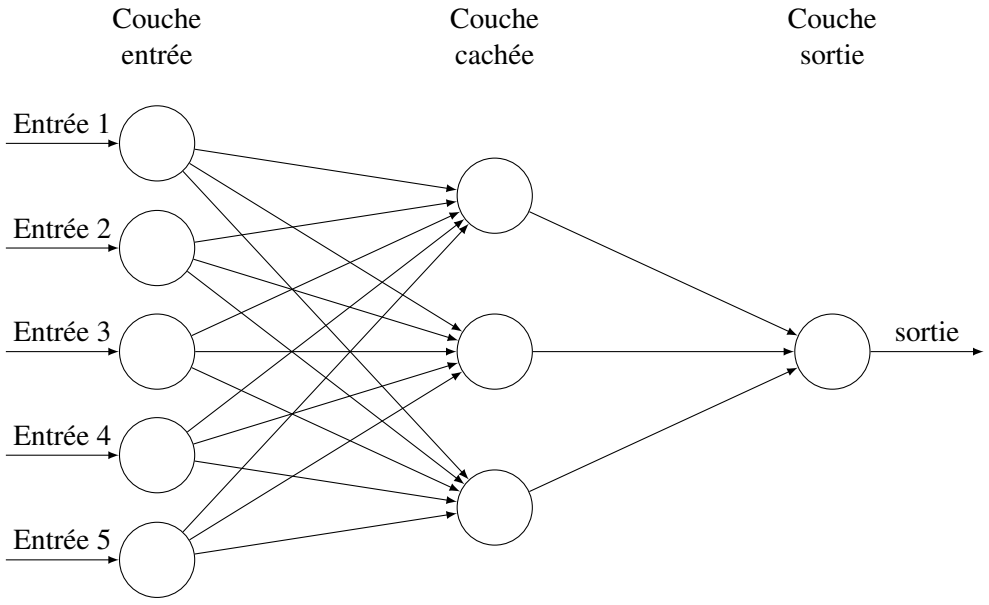

FIGURE 2.3 – Schéma d'un réseau neuronal possédant une seule couche cachée composée de trois neurones.

2.2.2 Rétropropagation au sein d'un réseau neuronal

La méthode de rétropropagation est la même que celle du perceptron. Il s'agit de calculer la dérivée partielle $\frac{\partial err}{\partial w_i}$. Seulement, il n'est plus possible de calculer pour chaque neurone la dérivée partielle, $\frac{\partial}{\partial w_i} err(a)$ car l'issu d'un neurone n'est plus forcément une sortie, mais peut être une entrée d'un autre neurone. On peut alors calculer une formule récursive de la dérivée partielle de l'erreur :

$$\begin{cases} \left(\frac{\partial err}{\partial w_i}\right)_{l,n} = x_i.s(1-s)(o-y) \text{ si le neurone est sur la dernière couche} \\ \left(\frac{\partial err}{\partial w_i}\right)_{l,n} = x_i.s(1-s)\sum_{j=1}^{m}\left(\frac{\partial err}{\partial w_n}\right)_{l+1,j} \text{ sinon} \end{cases}$$

où $\left(\frac{\partial err}{\partial w_i}\right)_{l,n}$ désigne la dérivée partielle de l'erreur par rapport au poids numéro i, du neurone n sur la couche l. De plus, m désigne le nombre de neurones sur la couche suivante.

2.3 Vocabulaire important

— **Une époque :** Représente une passe avant et une passe arrière de tous les exemples d'entraînement.
— **Taux d'apprentissage :** Une variable qui permet de modifier la longueur de chaque pas en direction du gradient noté η.

Chapitre 3

Implémentation

3.1 Neurone simple

3.1.1 Une première classe Neurone

Pour mettre en place cette classe, il est nécessaire de bien comprendre les différents processus d'apprentissage d'un neurone décrits ci-dessus. Dans le cadre de l'exercice, le neurone est une classe purement fonctionnelle. Les seuls attributs que l'on devrait y trouver sont certains paramètres comme le poids et le nombre d'entrées. Dans le cas d'un test effectif, on fera « tirer » un neurone pour les paramètres donnés, avec les poids déjà paramétrés. Dans le cas d'un apprentissage, on appellera successivement un « tir » puis des fonctions de modification de poids. Nous proposons une nomination de la forme suivante :

- *fire()* : pour la fonction mère qui déclenche le neurone, et produit sa réponse
- *learn()* : pour la fonction mère, qui s'occupe de modifier les poids en fonction des résultats
- *fw func()* :pour une fonction interne allant dans le sens « forward » c'est-à-dire dans le sens « normal »

3.1.2 Premiers résultats

Dans le cadre du projet, le mieux est de commencer par des apprentissages sur des jeux de données simples. Pour cela,on peut utiliser l'exemple des portes logiques. Nous avons donc entraîné notre neurone pour des entrées booléennes d'une porte OR et AND et la sortie indiquée. Par exemple, on veut pouvoir faire « apprendre » au neurone que le couple (0,1) d'une porte AND doit impliquer une sortie 0 et 1 pour une porte OR. Voici les premiers résultats à la fin de l'apprentissage et après une phase de test pour une porte AND : Les résultats sont en effet totalement incorrect par rapport à la sortie attendue. Cela est dû

TABLE 3.1 – Apprentissage de la porte AND avec un perceptron

	Résultat	Attendu
0 0	0.5	0
0 1	0.62855	0
1 0	0.663735	0
1 1	0.769588	1

à plusieurs problèmes. Le jeu de donnée est toujours dans le même ordre. En effet, il faudrait à chaque époque mélanger les entrées du jeu de donnée aléatoirement. Par ailleurs, il faudrait donner une valeur aléatoire aux poids en début d'apprentissage. Enfin, l'implémentation d'un biais semble nécessaire à la vu des valeurs de sortie proche de 0.5.

3.2 Le Biais

3.2.1 Définition

Le biais peut être défini simplement comme le potentiel d'activation d'un neurone. Celui-ci est soustrait à la somme des entrées pondérées par les poids, c'est lui qui contrôle si un neurone est actif ou non (supérieur à 0,5, ou 0 en fonction de ce qu'on utilise). Plus celui-ci est grand, moins il a de chance d'être actif, et inversement.

3.2.2 Héritage sur Neurone

L'ajout de l'héritage est traduit par un nouveau poids ajouté à l'entrée du neurone ayant comme valeur 1. Afin de l'intégrer au Neurone déjà écrit en C, un héritage est une bonne approche. En effet, un Neurone sera désormais soit un Neurone simple (sans biais) soit avec biais par spécialisation de celui-ci.

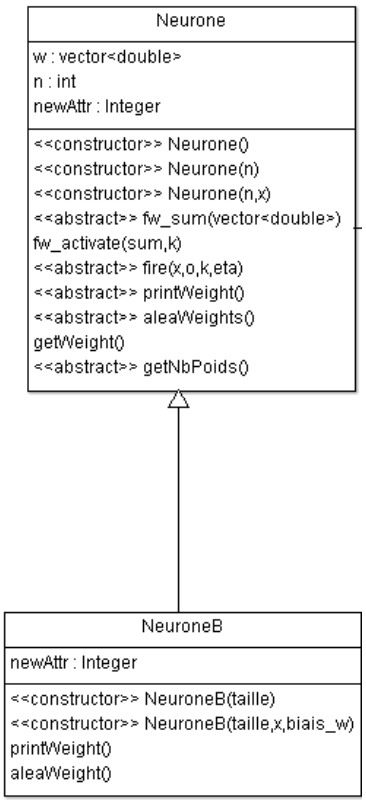

FIGURE 3.1 – Schéma UML d'un neurone

3.2.3 Nouveaux résultats

Voici après ajout du biais, des poids aléatoires, et entrées mélangées les nouveaux résultats.

TABLE 3.2 – Apprentissage de la porte AND avec un perceptron avec biais

	Résultat	Attendu
0 0	0.01e-12	0
0 1	0.000786	0
1 0	0.0000784	0
1 1	0.999987	1

Les résultats sont cohérents et valides.

3.3 Programmation d'un réseau de neurones

Après création d'un neurone fonctionnant sur des jeux de tests simples, il est maintenant temps de fabriquer un réseau neuronal afin de pouvoir prendre plus de deux décisions en fonction d'un jeu de tests plus complexe.

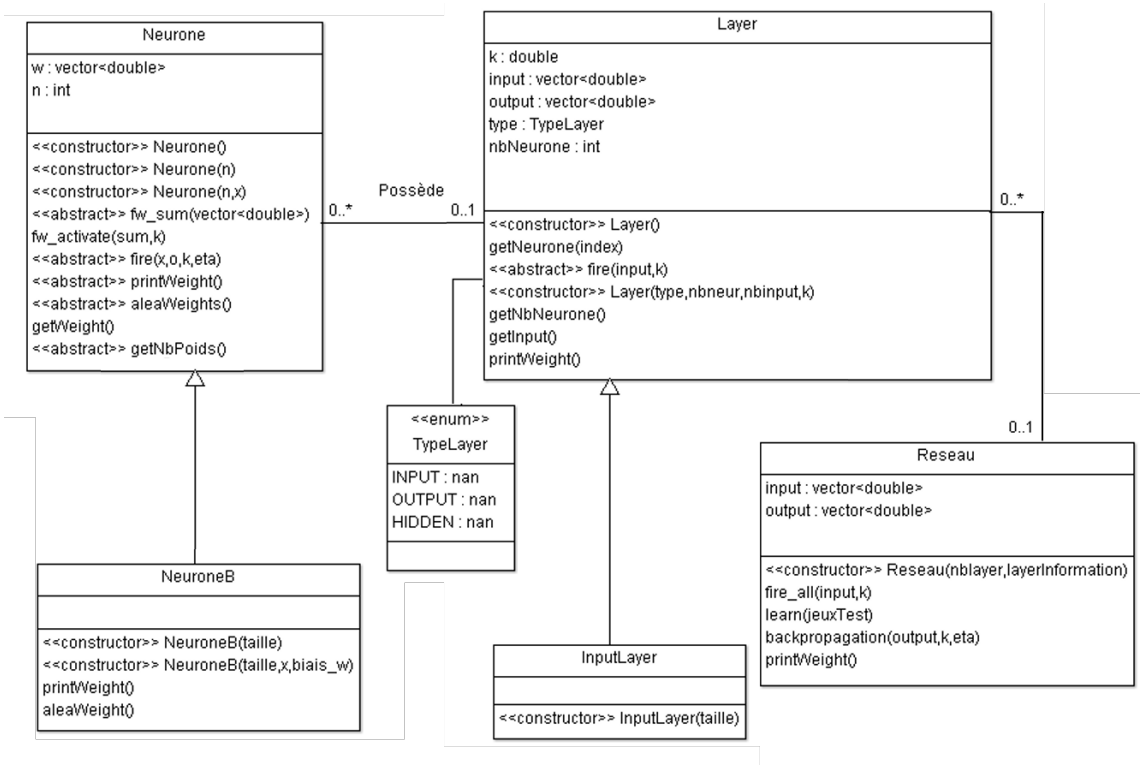

FIGURE 3.2 – Schéma UML du réseau neuronal

3.4 L'algorithme de rétropropagation

Les méthodes de « tir » et d'apprentissage des neurones restent les mêmes, mais dans une plus grande échelle. Il est cependant important de décrire l'algorithme de rétropropagation qui change dans le réseau neuronal.

En effet, il faut considérer que chaque neurone d'une couche est relié à tous les neurones de la couche suivante et de la couche précédente. De plus, selon le lieu dans le réseau, les calculs d'erreur ne sont pas les mêmes.

Algorithm 1 Rétropropagation(vect<double> output, double k, double etat) ;

$nbNeuroneMax \leftarrow max(nbNeurone)$;
$derrdact, dsig, dact : double[nbLayers][nbNeuroneMax][nbNeuroneMax]$;

for $l = nbLayer - 1$ to 0 **do**
  $Layer \leftarrow reseau[l]$;
  for $n = 0$ to $Layer \rightarrow getNbNeurones()$ **do**
    $Neurone \leftarrow l[n]$;
    $s \leftarrow \sum_{k=0}^{NbPoids} x_k \cdot w_k$
    for $i = 0$ to $Neurone \rightarrow getNbPoids()$ **do**
      $dact[l][n][i] \leftarrow x_i$;
      $dsig[l][n][i] \leftarrow s(1 - s)$;
      if $l == nbLayers - 1$ **then**
        $derrdact[l][n][i] \leftarrow -2(output[n] - n \rightarrow activate())$;
      else
        $derrdact[l][n][i] \leftarrow \sum_{k=0}^{reseau[l+1]} \rightarrow getNbNeurone()w_{k,n} \cdot derrdact[l+1][n][i] \cdot dsig[l+1][n][i]$
      end if
    end for
  end for
end for

for $l = nbLayer - 1$ to 0 **do**
  for $n = 0$ to $Layer \rightarrow getNbNeurones()$ **do**
    for $i = 0$ to $Neurone \rightarrow getNbPoids()$ **do**
      $w_{l,n,i} = w_{l,n,i} - \eta \cdot derrdact[l][n][i] \cdot dsig[l][n][i] \cdot dact[l][n][i]$
    end for
  end for
end for

Il est possible de minimiser le nombre d'appels à l'algorithme de rétropropagation en utilisant la moyenne de plusieurs erreurs observées, on parle alors de rétropropagation de l'erreur moyenne. Cette technique peut être utilisée en option dans le programme.

Chapitre 4

Expérimentation

Pour pouvoir tester notre implémentation, nous avons effectué un grand nombre d'expériences. Cette partie présente des expériences représentatives des possibilités de notre programme. Pour réaliser nos expériences, nous avons pris aléatoirement 80% des données de la source pour l'apprentissage supervisé. Les données restantes sont utilisées pour tester le réseau sur des données lui étant inconnues. Les tests sont réalisés sur un processeur Intel Core i5-7400. Cependant, l'objectif est souvent de comparer les résultats entre eux, il est donc plus important de préciser qu'ils ont tous étaient réalisés sur le même processeur.

Les paramètres ont été modifiés récursivement pour obtenir la configuration possédant le moins de neurones et de couches pour accélérer le processus tout en ayant des résultats pertinents.

4.1 Sources

Pour un apprentissage supervisé, il faut un grand nombre de données d'exemples. Il existe un grand nombre de sites internet proposant des collections libres de ces données. On peut citer parmi ces sites :

— **Open-Source, Distributed, Deep Learning Library : deeplearning4j.org**
— **Open ML Data set : openml.org**
— **Center for Machine Learning and Intelligent Systems : archive.ics.uci.edu**

Ces sites seront les sources de nos apprentissages pour les expériences suivantes.

4.2 Un apprentissage sur perceptron

Nous avons d'abord testé notre perceptron avec un jeu de données correspondant au ET et OU logique. Les premiers tests, sans biais, nous renvoyaient des résultats où l'on pouvait observer que le perceptron avait appris, car les résultats étaient relativement corrects, on pouvait voir qu'il discernait le ET du OU.

TABLE 4.1 – Résultats du perceptron sans biais

Entrée 1	Entrée 2	Résultat ET	Attendu	Résultat OU	Attendu
0	0	0.5	0	0.5	0
0	1	0.62855	0	0.731059	1
1	0	0.663735	0	0.731059	1
1	1	0.769588	1	0.875452	1

Nous avons ensuite présenté ces résultats à nos encadrants qui nous ont conseillé d'intégrer un biais à notre perceptron afin de voir la différence. Cette fois-ci, les résultats étaient beaucoup plus proches de ceux attendus et nous en avons donc conclus qu'un biais serait nécessaire pour la suite.

TABLE 4.2 – Résultats du perceptron avec biais

Entrée 1	Entrée 2	Résultat ET	Attendu	Résultat OU	Attendu
0	0	4.99e-05	0	0.0334286	0
0	1	0.0335797	0	0.978888	1
1	0	0.0335912	0	0.978883	1
1	1	0.960297	1	0.999982	1

4.3 Un apprentissage basique sur un réseau

4.3.1 Résultats pertinents

Nous allons dans un premier temps classer trois espèces d'Iris à l'aide d'une base de données de R.A. Fisher ["The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936).

Cette base de données contient 150 exemples possédant quatre attributs :

— La longueur du sépale en cm
— La largeur du sépale en cm
— La longueur du pétale en cm
— La largeur du pétale en cm

L'objectif est de classer de nouveaux exemples dans l'une de ces trois espèces :

— Iris Setosa
— Iris Versicolour
— Iris Virginica

Avec une configuration simple [1 couche cachée possédant 2 neurones], notre réseau réussira en quelques secondes à classer toutes les données de test sans erreurs dont le résultat lui était inconnu. Nous avons ici un exemple du bon fonctionnement d'une classification tertiaire dans une dimension 4.

4.3.2 Nombre de neurones et de couches

Pour pouvoir configurer notre architecture, les tests effectués dans la Table 4.3 nous permettent de conjecturer que le nombre de couche et de neurones par couches ne doit pas être trop faible ni trop élevé. Cela s'explique par le nombre de modifications vectorielles nécessaires et le temps de calcul que nécessitent ces modifications.

TABLE 4.3 – Importance du nombre de neurones et du nombre de couches

Nombre de neurones	Nombre de couches cachées	Résultat en 5 secondes de calcul
0	0	13% d'erreurs
1	1	6% d'erreurs
2	1	0% d'erreurs
3	1	0% d'erreurs
4	2	3% d'erreurs
5	2	6% d'erreurs
2	2	6% d'erreurs
1	3	70% d'erreurs
2	3	70% d'erreurs

4.4 Retour à la problématique - Courriel indésirable

Dans l'introduction de ce dossier, il était question d'une intelligence capable de trier nos messages entre courriel indésirable et légitime. Il est donc temps de mettre à l'épreuve notre réseau neuronal. Pour entraîner notre réseau, nous allons utiliser une base de données créée par Mark Hopkins, Erik Reeber, George Forman et Jaap Suermondt.

Cette base de données contient 4600 données possédant 58 attributs :

— 48 attributs décrivant la fréquence d'un mot

— 6 attributs décrivant la fréquence d'un caractère

— 4 statistiques sur les majuscules dans le texte

La sortie est binaire avec « courriel indésirable » ou « courriel légitime ». Avec une configuration simple [2 couches cachées possédant 4 neurones chacune], notre réseau peut obtenir un taux de réussite supérieur à 95% pour classer toutes les données de test sans erreurs . Cet exemple dans une dimension 58 permet de voir la rapidité de traitement du réseau neuronal. Il faudrait plusieurs heures à un humain pour trouver une telle corrélation.

4.5 Un exemple plus complexe

Pour finir, nous allons étudier un cas plus complexe avec des images de la base de données « Genuine and forged banknotes Data Set » de Volker Lohweg. Cette base correspond à des données extraites d'image de papier officiel d'une banque : l'entropie, la variance, l'asymétrie et la concentration de chaque image. L'objectif est de classer ces images selon leur authenticité ou non.

Il faut une configuration avec 5 couches et 5 neurones par couche pour obtenir des résultats satisfaisants en une trentaine de secondes. Sans ce nombre important de couches cachées, le taux d'erreur ne peut descendre sous 3%. C'est donc un bon exemple de classification de données dont la séparation nécessite de nombreuses modifications vectorielles. Cette tache était, il y a encore quelques années en arrières impossibles en un temps restreint comme celui-ci.

Chapitre 5

Compléments

5.1 Configuration du réseau

Il est possible de configurer le réseau neuronal selon de nombreux paramètres :
— –input (-i) pour configurer un nouveau fichier d'entrée.
— –autooff (-o) pour préciser un pourcentage d'erreur qui arrêtera l'apprentissage. Cette option permet aussi d'afficher le pourcentage d'erreur en continu.
— –archi (-a) pour afficher la visualisation de l'architecture en python comme décrit plus bas.
— –epoch (-e) pour configurer le nombre maximum d'époques à effectuer.
— –eta (-lr) pour préciser la valeur d'êta (learning rate).
— –gradiant (-gr) pour préciser le nombre de calculs d'erreurs avant la rétropropagation de l'erreur moyenne.
— –function (-f) pour configurer la fonction d'activation à utiliser parmi une liste.

5.2 Documentation

L'un des points importants du cahier des charges était d'avoir un code propre et documenté. Pour cela, le groupe a commenté tout au long de l'implémentation du réseau les différentes fonctions, méthodes, classes et variables. Ces commentaires sont tous formatés selon les règles de générateur de documentation Doxygen. Ce générateur sous licence libre permet de générer une documentation sous de nombreux formats. Nous avons décidé de privilégier la génération d'un site internet de documentation et un dossier PDF. Ceux-ci sont disponibles à l'adresse suivante : **k1nd0ne.com/docxygen**. En complément, ce dossier comprend des schémas UML de l'architecture du code.

5.3 Visualisation

Le cahier des charges comprenait aussi la visualisation des résultats et de l'apprentissage du réseau neuronal. L'objectif principal est d'exhiber l'évolution des différentes caractéristiques durant la phase d'apprentissage. Pour faciliter cette visualisation, plusieurs compléments sont intégrés au code principal.

5.3.1 Architecture avec Python

La première est une visualisation de l'architecture du réseau avant l'exécution de l'apprentissage. Cette visualisation s'appuie sur Matplotlib, une bibliothèque du langage de programmation Python destinée à tracer et visualiser des données sous forme de graphiques. Si l'argument *-a* ou *–archi* est précisé au lancement du programme, une fenêtre apparaîtra et exhibera l'architecture configurée comme dans la figure 5.1

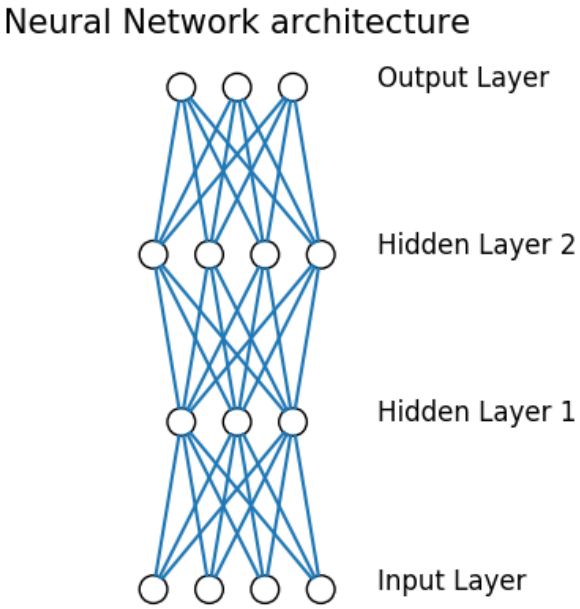

FIGURE 5.1 – Visualisation de l'architecture en Python

5.3.2 Résultats avec JavaScript et PHP

Pour permettre de configurer plus facilement et à distance un réseau neuronal ou un perceptron, une surcouche PHP a été ajoutée. Cette surcouche permet sur un serveur de lancer le programme précompilé selon les options configurées à travers une interface graphique en HTML et CSS. Par la suite, un code JavaScript permet de visualiser l'évolution du réseau neuronal "en temps réel". Ce code JavaScript utilise la bibliothèque p5.js qui permet de générer des graphiques ayant pour support de rendu le navigateur. Ce choix de bibliothèque est motivé par la possibilité de rendre visibles les résultats directement sur le navigateur sans outils complémentaires à installer comme dans les figures 5.2 et 5.3.

La figure 5.2 représente une interface web où on peut voir l'évolution d'un perceptron avec le temps. Cette interface permet de visualiser l'évolution du perceptron sur 2 jeux de données : le ET et le OU logique. Le rouge représente un 0, et le vert un 1. Lien : **neural.areski.info**

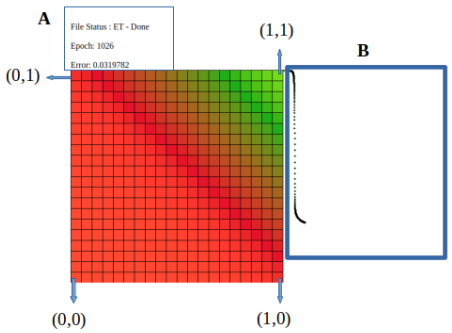

FIGURE 5.2 – Visualisation du perceptron
A - Informations sur l'apprentissage. B - Courbe représentant l'erreur par rapport au temps.

La figure 5.3 représente une autre interface web où l'on peut voir l'évolution d'un réseau neuronal avec le temps. On peut voir, « en temps réel » , les poids de chaque arc et aussi la sortie. Lien : **server.areski.info/network/**

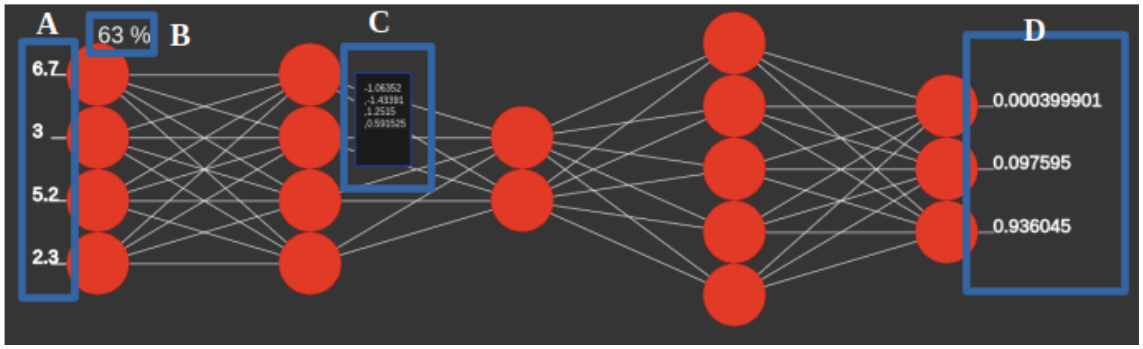

FIGURE 5.3 – Visualisation du réseau
A - Entrées. B - Pourcentage de l'apprentissage. C - Détails sur les poids de chaque arc. D - Sorties.

Chapitre 6

Conclusion

Lors de ce projet, le cahier des charges était complexe à tenir. Il a donc fallu organiser notre travail. En effet, il fallait construire une structure simple de neurone puis y ajouter de nombreuses fonctionnalités et finalement l'intégrer dans un réseau neuronal. À ce jour, nous pouvons apprendre à notre réseau de neurones divers jeux de tests (reconnaissance de spams, vin, fleurs ...) avec des résultats concluants.

Les perspectives d'amélioration sont nombreuses. En effet, il serait possible d'utiliser le réseau de neurones pour la reconnaissance d'image. Dans le cas d'images en noir et blanc, il faut passer le niveau de gris de chaque pixel de l'image en entrée. On peut donc entraîner le réseau sur un jeu de test d'image puis l'interroger sur des nouvelles.

Ce projet ambitieux nous a permis d'en apprendre davantage sur le domaine de l'intelligence artificielle. La partie théorique de ce projet est un apport important pour maîtriser l'informatique dans son ensemble. D'autre part, l'implémentation de cette théorie nous a permis de progresser et d'apprendre à nous organiser et à gérer un projet.

Bibliographie

[1] *« Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's ? If this were then subjected to an appropriate course of education, one would obtain the adult brain. »*
Alan Turing
Computing Machinery and Artificial Intelligence
1950.

[2] **Yohann Mansiaux**
Analyse d'un grand jeu de données en épidémiologie : problématiques et perspectives méthodologiques
Université Pierre et Marie Curie - Paris VI
2014.

[3] *Surinterprétation est un synonyme de surapprentissage, cependant le terme de « surinterprétation » rend mieux compte de l'interprétation excessive que signifie le terme anglais « overfitting ».*
Babyak Michael A. PhD
What You See May Not Be What You Get : A Brief, Nontechnical Introduction to Overfitting in Regression-Type Models
Duke University Medical Center, Durham, NC
2004.

[4] **Rafael Geraldeli Rossi, Solange Oliveira Rezende**
Generating Features from Textual Documents Through Association Rules
Instituto de Ciencias Matematicase de Computacao - Universidade de Sao Paulo
Sao Carlos, Brazil
2011.

[5] **Yoav Freund, Robert E. Schapire**
Large Margin Classification Using the Perceptron Algorithm
Shannon Laboratory
1999.

[6] **Jun Han, Claudio Moraga**
The influence of the sigmoid function parameters on the speed of backpropagation learning
University of Dortmund, Dortmund, Germany
1995.

[7] **Tangente Sup**
Le Perceptron
N 44-45 Septembre 2008.

Table des figures