Executive summary

This graduation work deals with the topic of feed-forward neural networks and their application in the imaging field. The purposes are understanding their the founding mathematical principles and the way they "learn" to recognize shapes in images. Furthermore, a C/C++ neural networks library called **NeuroLib** is also developed within the scope of this work.

The first chapter talks about the context of the internship and machine learning in general. This introduction develops the background of the workplace, the specifications to reach and also some common words that often appear in any type of publication discussing about the subject.

After a short introduction, an explanation of the theory of the perceptron comes with its implementation. Then, two learning algorithms for the perceptron are detailed with their implementation. Finally, the results of some experimentations are discussed as their limits.

The second chapter leads us to describ an improvement of the perceptron : the multi-layer perceptron along its implementation. Just as the perceptron learning algorithms were explained, a mathematical explanation of the back-propagation algorithm followed by it's C/C++ transcription is also provided. In his case, several simple examples are used to analyze the impacts of each of the numerous parameters required by a neural network. Again, the results and the limits of the multi-layer perceptron are discussed. More over, the theory of universal approximators is approached and another example is supplied.

Afterwards, feed-forward neural networks to character recognition on french license plate is applied. This chapter provides information about a software developed to make experiment easier, the datasets used to train and validate the network and some pre-processing that can be done to improve the results. Initially, the pixels will serve as the input data of neural networks that are differently configured for comparison purposes. Secondly, the geometrical features of the characters are used to distinguish them from another. Finally, the results are compared in order to find the best way to recognize shape in images.

The fifth chapter compares several existing deep learning libraries and **NeuroLib**. The experimentations are made in Python except for **NeuroLib**. The performances obtained by the other library are compared with **NeuroLib** in order to see the how well it performs and how it can be improved.

The final chapter talks about the respect of the specification provided at the beginning of the internship. A recipe that describes how to tune the neural network to make it efficient is also explained. Finally, the perspective of neural networks study is also discussed by mentioning other features that can also be studied.

Table des matières

Chapitre 1

Introduction

Mon stage s'effectue dans un contexte ERASMUS et a été supervisé par Monsieur Patrick BONNIN, un professeur et chercheur français spécialisé dans le traitement d'images à l'Université de Versailles Saint-Quentin-en-Yvelines. Dans cette université, il dispense les cours de C/C++, d'Operating System ainsi que de traitement d'images aux étudiants des filières Mécatronique et Systèmes Embarqués Électroniques dont il est directeur. Plus spécifiquement, Monsieur BONNIN est spécialisé dans le traitement d'images dans la vision par ordinateur en *temps réel* dans les applications telles que la robotique, les drones ou encore la vision industrielle.

Le but de ce stage est d'évaluer les possibilités de reconnaissance automatique d'objets, en *temps réel*, grâce à la méthode de machine learning des *réseaux de neurones*. Les réseaux de neurones sont capables de classifier une entrée quelconque qui lui est fournie à condition d'avoir été entraîné à cette tâche. L'entraînement d'un réseau de neurone correspond à l'analyse d'exemples qu'il est susceptible de devoir analyser en production à l'exception que les réponses sont également disponibles. Il peut ainsi se corriger et "apprendre" à reconnaître, ou *classifier*, correctement les entrées qui lui sont soumises. Cette étude est utile au département d'ingénieur, également appelé Institut des Sciences et Techniques des Yvelines (ISTY), de l'Université Saint-Quentin-en-Yvelines. Elle souhaite participer à la *RoboCup@Home Social*, où un robot est programmé pour servir l'Homme comme, par exemple, trouver une personne dans un appartement. Dans ce contexte, la reconnaissance automatique de formes est intéressante car elle permet, par exemple, à un robot d'identifier une personne spécifique. Ce dernier analysera alors les visages des personnes présentes afin de repérer sa cible. Le cadre de mon stage se situe donc en amont du contexte de la compétition *RoboCup@Home Social*.

Le réseau de neurones n'est pas une méthode récente. Depuis son apparition dans les années 40, elle a subi diverses vagues de popularité. Son engouement actuel est notamment du à la puissance de calcul des ordinateurs contemporains, du parallélisme des GPU, des bases de données riches en information. Ces trois ressources sont, en effet, essentielles aux réseaux de neurones et manquaient dans les décennies précédentes.

1.1 Contexte de travail

1.1.1 L'Université de Versailles Saint-Quentin-en-Yvelines

L'Université de Versailles Saint-Quentin-en-Yvelines (UVSQ) est une université publique située dans le département des Yvelines (78) en banlieue parisienne (France) qui a été officiellement créée le 22 juillet 1991. Elle accueille près de 19 000 étudiants répartis sur quatre campus. L'UVSQ donne accès à presque 200 formations différentes.

Tout d'abord, l'Unité de Formation et de Recherche (UFR) des Sciences accueille les étudiants de physique, chimie, mathématiques et informatique sur le site de Versailles et Le Chesnay. Ensuite, l'UFR des Sciences Sociales délivre les formations d'économies, d'administration économique et sociale ainsi que les sciences géographiques et sociales sur le site de Saint-Quentin-en-Yvelines. Ce dernier accueille également les étudiants de la faculté de Droit et des Sciences Politiques. L'UFR des Sciences de la Santé de Simone Veil regroupe les formations des médecins et sages-femmes, les étudiants en histoire, lettres de l'Institut d'Études Culturelles et Internationales et ceux en sciences de gestion, de l'information, de communication et de l'éducation de l'Institut Supérieur de Management.

Comme mentionné plus haut, l'UVSQ dispose d'une école d'ingénieurs appelée l'Institut des Sciences et Techniques des Yvelines (ISTY). Cette dernière forme des ingénieurs en Informatique, en Mécatronique ainsi qu'en Systèmes Électroniques Embarqués répartis sur les deux sites restants : celui de Vélizy-Villacoublay et celui de Mantes-en-Yvelines. Cet institut est couplé avec les Instituts Universitaires de Technologies, abrégé en IUT, qui forment différents bacheliers diplômés en génies chimique, électrique, administratif et commercial, informatique, réseaux et télécommunications sur le site de Vélizy avec son annexe de Rambouillet. Le second IUT, situé à Mantes-En-Yvelines, offre des formations en génies civil, industriel, maintenance, mécanique et en gestion.

En plus de ces formations, l'ISTY possède un cycle préparatoire intégré, une formation scientifique de deux années donnant l'accès aux filières ingénieures en France. Pour certains de ces cycles, les classes préparatoires ne sont pas obligatoires. Les étudiants ont alors accès à des formations d'ingénieurs en trois ans, avec une spécialisation précoce, dès la première année.

1.1.2 La compétition *RoboCup@Home Social* Standard Edition

Mon stage se déroule dans le cadre de la compétition *RoboCup*, et plus particulièrement, dans la catégorie *RoboCup@Home Social*. *RoboCup* est un projet international créé afin de promouvoir l'intelligence artificielle, la robotique et les autres domaines connexes. Ce projet est formé d'un ensemble de concours, divisés en catégories. Chaque catégorie fournit des challenges se basant sur des problèmes courants où les participants proposent une ou plusieurs solutions.

Les différentes catégories sont :

- *RoboCup Soccer* : deux équipes composées de robots Nao s'affrontent dans un match de football anglais. Les robots ont chacun une fonction, telle qu'attaquant ou défenseur. Ils doivent également se diriger vers la balle et la suivre dans le but d'inscrire des goals.
- *RoboCup Rescue* : le robot doit porter secours à une ou plusieurs personnes en danger. Cela peut être une mission de déminage ou d'extraction d'une personne d'un endroit exigu.
- *RoboCup@Home* : elle est composée de trois sous-catégories :
  - *Domestic* : elle utilise le robot *Toyota Human Support Robot* (HSR). Le but de cette compétition est d'assister des personnes, âgées principalement, souffrant de maladies ou d'handicaps divers.
  - *Social* : les robots ont pour objectif d'assister les humains pour leur permettre d'évoluer dans un confort optimal. Les robots peuvent, par exemple, jouer le rôle de garçon de salle ou d'hôte dans un magasin ou dans un musée.
  - *Open-Platform* : il s'agit d'une ligue où les compétiteurs souhaitent simplement effectuer des tests divers avec leur configuration. Les restrictions des autres compétition de la *RoboCup@Home Social*ne sont pas d'application.

— *RoboCup Industrial* : elle concerne l'utilisation de robots dans un milieu de travail. À nouveau, il existe plusieurs sous-catégories :
  — *RoboCup@Work* : le robot doit effectuer une tâche telle que : charger, décharger un conteneur rempli d'objets divers ; effectuer certaines opérations sur des produits ; réaliser un transport d'objets coopératif ; etc.
  — *RoboCup* Logistics : concerne la logistique interne à une usine. Le robot doit, par exemple, pouvoir prendre des matériaux dans le stock et les transporter jusqu'à la machine adéquate, puis reprendre le produit fini et le transporter vers le camion.

1.1.3 Cahier des charges

La compétition *RoboCup@Home Social*, dans laquelle le stage s'intègre, vise à développer les technologies assistant l'Homme dans un contexte domestique. L'intérêt principal est concentré sur l'interaction et la coopération robot-humain, sur la navigation du robot dans un appartement et sur la cartographie de ce dernier. Cependant, afin de simuler un environnement réel dans lequel les objets peuvent être déplacés, ces cartographie et navigation doivent pouvoir s'adapter à un environnement dynamique. Pour ce faire, le robot doit pouvoir reconnaître des objets, les manipuler et s'adapter à toute situation.

Le stage concerne principalement la reconnaissance temps réelle de formes dans une image. Les multiples objectifs de ce stage sont :
— Réaliser une étude théorique des réseaux de neurones afin de comprendre leur comportement et leurs limites ;
— Évaluer les capacités des réseaux de neurones concernant la reconnaissance de formes dans des images en temps réel ;
— Implémentation d'une librairie de réseaux de neurones complète temps réelle, portable entre les différentes plateformes existantes (Windows, Linux, etc.), servant à la reconnaissance de formes dans des images.
— Comparaison des performances de la librairie de réseaux de neurones développée avec d'autres librairies existantes. Les points de comparaison sont divers : le temps, l'efficacité, la modularité, les possibilités, etc. De plus, en fonction des résultats obtenus par les autres librairies, une liste d'amélioration pouvant être apportée à la bibliothèque développée dans le cadre de ce stage pourra être établie.
— Déterminer si les réseaux de neurones sont applicables dans le contexte de la *RoboCup*.

1.2 Le machine learning

Avant d'aborder l'étude théorique des réseaux de neurones, nous allons, tout d'abord, décrire brièvement le machine learning afin d'avoir une vue globale du domaine. Les réseaux de neurones ne sont qu'une des nombreuses techniques qui constituent le vaste ensemble qu'est le machine learning. Ensuite, les différents types d'apprentissage statistiques seront expliqués. Cette introduction se terminera par un bref historique sur les réseaux de neurones.

1.2.1 Introduction au machine learning

Tout d'abord, le machine learning est un domaine issu de la branche statistique des mathématiques. Sa popularité a explosé ces dernières années et beaucoup d'articles issus de la recherche, de livres d'initiation et divers tutoriaux sur Internet sont apparus. Cette section présente quelques définitions et explications sur les divers types de machine learning qui existent afin de recentrer les idées et les diverses applications de cette technique.

Intelligence artificielle (IA): désigne les systèmes capables d'accomplir des tâches qui relèvent de l'intelligence humaine. Ces tâches sont assez générales : planifier, comprendre un langage, reconnaître des objets ou des sons, apprendre, résoudre un problème, etc.
Deux catégories d'intelligence artificielle se distinguent. La première est l'IA générale ayant des caractéristiques similaires à celles de l'humain. Elle est la plus difficile à concevoir : l'Homme n'en n'est encore qu'à ses premiers balbutiements. À l'inverse, la seconde est plus restrictive. En effet, elle n'exhibe qu'une ou l'autre facette de l'intelligence humaine. Citons, comme exemple, la reconnaissance d'un mot dans une phrase, sans pour autant en comprendre le sens.

Machine learning: ensemble de techniques d'apprentissage statistiques dans lesquelles un modèle implicite ou explicite peut être établi. Grâce à ce modèle, il est possible de classifier, prédire ou déceler une structure interne à des données, et de fournir une IA "étroite". Une combinaison de techniques de machine learning est nécessaire afin de fournir une IA moins "étroite".

Neurone artificiel: créé par McCulloch et Pitts en 1949, il s'agit d'un modèle mathématique qui se base sur le neurone biologique du cerveau humain. Les entrées du neurone biologique sont appelées dendrites et correspondent à celles du neurone artificiel. La sommation pondérée du neurone agit comme le soma du neurone biologique. Enfin, l'axone reçoit son signal du soma. Une fois que le soma atteint un certain potentiel, l'axone s'activera et transmettra un signal. La fonction d'activation du neurone artificiel copie le fonctionnement de l'axone. En statistiques, le neurone artificiel peut être utilisé pour faire de la régression ou de la classification linéaire.

Réseaux de neurones: ils sont composés de couches de neurones, dont les sorties d'une couche correspondent aux entrées de la couche suivante. Ils permettent alors de faire de la régression ou de la classification non-linéaire.

Deep learning: il s'agit d'un sous-domaine du machine learning dont la structure se base sur les neurones artificiels. En réalité, un réseau de neurones artificiels comporte plusieurs couches de neurones. Plus il y a de couches, plus le réseau sera "profond".

Apprentissage supervisé (Figure 1.1) [Ree17]

L'apprentissage supervisé utilise des données d'apprentissage *étiquetées*. L'étiquette donne l'information que le modèle devra, par la suite, prédire. Ces données étiquetées, ou labeled data en anglais, servent d'ensemble de données d'apprentissage (Learning Dataset) et permettent de trouver la relation entre les variables d'entrée X et de sortie Y qui sera de forme $Y = f(X)$. Il existe deux types d'apprentissage supervisés :

- La classification : ce type d'apprentissage prédit la valeur d'un échantillon dans lequel la variable Y est une *classe* ou encore *catégorie* telle qu'une lettre, le sexe d'une personne, son état de santé (bon, mauvais, moyen), etc. Un exemple est visible sur la gauche de la Figure 1.1 où deux classes sont bien séparées.
- La régression : elle prédit la valeur d'un échantillon sous forme de réel ou de vecteur de réels. Par exemple, le poids d'une personne, une quantité d'énergie, le prix d'un produit, etc. Un exemple est illustré sur la droite de la Figure 1.1 dont la droite de régression est en rouge.

Exemples d'algorithmes d'apprentissage de machine learning supervisés : régression linéaire, régression logistique, Arbres de Classification et Régression (CART / C&RT), la Classification Naïve Bayésienne ou encore la méthode des k plus proches voisins (KNN).

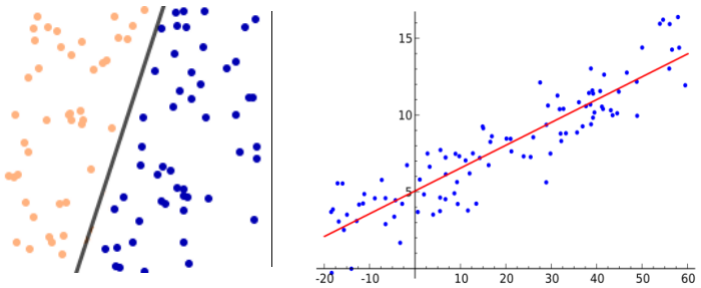


FIGURE 1.1 – À gauche, un exemple de classification et à droite, un exemple de régression.

Apprentissage non-supervisé (Figure 1.2) [Ree17]

L'apprentissage non-supervisé ne possède que les variables d'entrée X. Les données ne sont donc pas étiquetées comme dans l'apprentissage supervisé. Trois types d'apprentissage non-supervisé existent :

- L'association : elle détermine la corrélation entre deux phénomènes. À titre d'exemple : si un client achète un pain, la probabilité qu'il achète de la confiture est de 80%. Un exemple d'association est visible sur la Figure 1.2, en bas à gauche, une association est réalisée entre les consommateurs des services basiques et ceux des services privilégiés (premium).
- Le groupement (clustering) : il regroupe en *classes* des observations qui se ressemblent. Un exemple est la division de la clientèle en petits groupes homogènes où les clients achetant du pain seront groupés avec ceux achetant la confiture. Ces derniers seront séparés des clients qui achètent du lait et des céréales. Sur la Figure 1.2, le graphique en haut à gauche montre trois classes bien distinctes.
- La réduction de dimensions : dans un dataset, le nombre de variables est souvent élevé et il devient difficile de les visualiser toutes en même temps. Par ailleurs, certaines informations peuvent s'avérer insignifiantes lors de l'analyse. Les variables peu importantes sont alors élaguées afin de simplifier l'ensemble. Dès lors, la mise en évidence des structures de données éventuelles est plus aisée. À titre d'exemple, dans les groupes de clients qui achètent du lait et des céréales d'une part et ceux qui achètent du pain et de la confiture d'autre part, les variables détergent et maquillage, que l'on suppose *non-discriminantes*, peuvent alors être supprimées du modèle. Sur la Figure 1.2, le graphique, en haut à droite, présente plusieurs caractéristiques flèchées. Les caractéristiques éloignées du bord du cercle des variables, autrement dit proches de l'origine, ne seront pas interprétées et peuvent être enlevées de ce modèle. En effet, celles-ci sont mal projetées, ne sont pas corrélées correctement avec les deux facteurs constituant le plan factoriel. Un modèle simplifié peut alors être déduit.

Exemples d'algorithmes d'apprentissage de machine learning non-supervisés : K-means, Analyse en Composantes Principales (ACP).

Apprentissage par renforcement [Ree17]

L'apprentissage par renforcement se base sur un apprentissage des comportements fondés sur des expériences. L'état de l'algorithme va conditionner ses choix. Afin d'illustrer l'utilité d'un tel type d'apprentissage, l'exemple choisi concerne la robotique. Un robot doit se déplacer dans un espace rempli d'obstacles. L'expérience concerne alors le phénomène de collision. Si un robot reçoit un feedback négatif d'un de ses capteurs, indiquant que celui-ci vient de percuter un objet, le robot apprend qu'à cet endroit, un obstacle est présent. Par la suite, si le robot revient à la même position, il sera conscient que ce chemin contient un obstacle obstruant le passage.

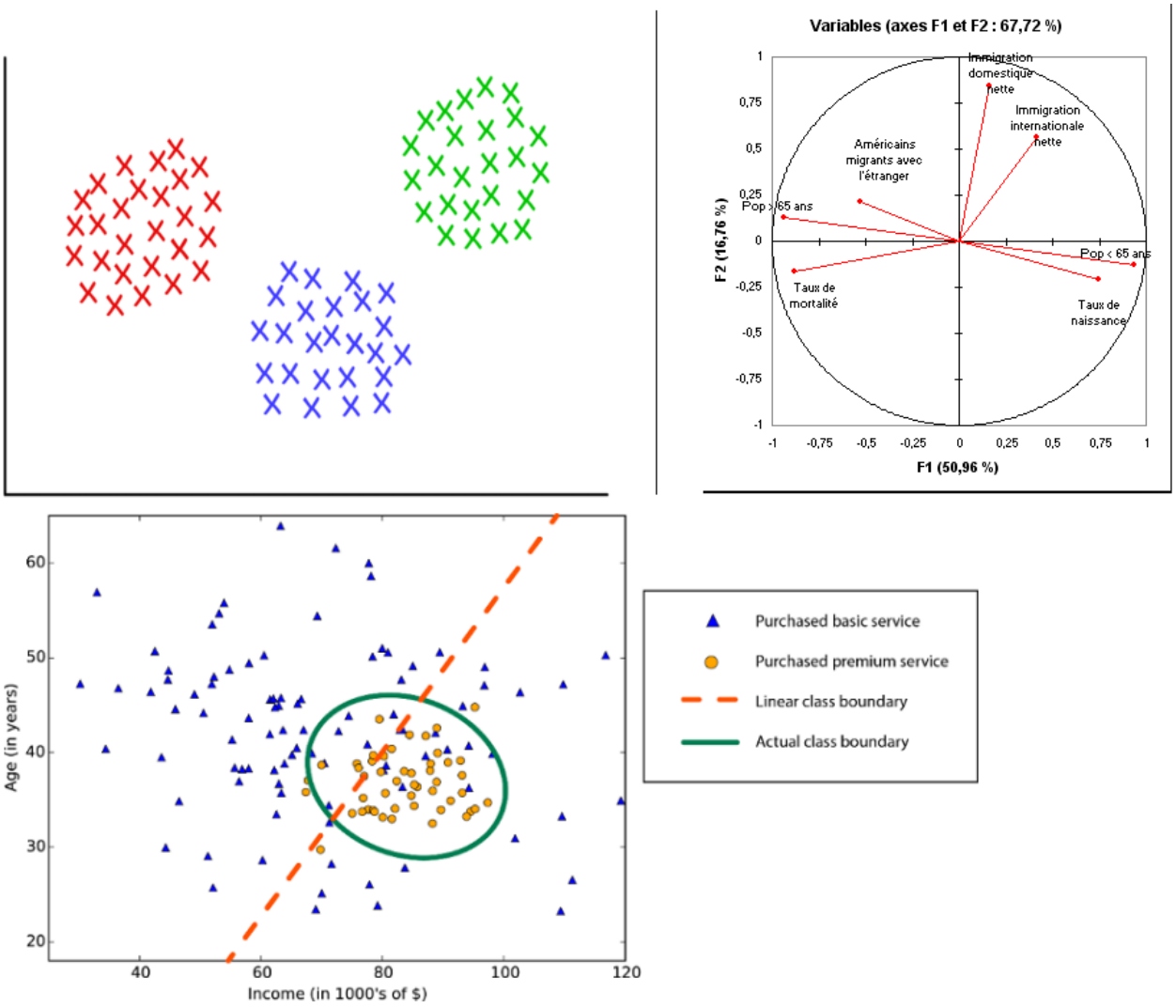


FIGURE 1.2 – Une représentation des trois types d'apprentissage non-supervisés.

L'apprentissage par renforcement dérive donc de l'apprentissage supervisé où les données ne sont pas préalablement disponibles.

Exemples d'algorithmes d'apprentissage de machine learning par renforcement : forêts d'arbres décisionnels (Random Forest), Adaptive Boosting (Adaboost).

1.3 Historique des neurones artificiels

— 1943 : W. MCCULLOCH (neuropsychiatre) et W. PITTS (logicien) inventent le premier neurone artificiel basé sur leurs travaux portant sur les neurones biologiques [KS96a]. Ce dernier étant très rudimentaire, il n'effectue qu'une somme des entrées pondérées par des coefficients, qui est ensuite seuillé.

— 1958 : ROSENBLATT s'appuie sur les travaux de MCCULLOCH et PITTS et crée le principe du perceptron [KS96a]. Il est identique au neurone précédemment établi et il sert à la *classification* binaire. Par extension, le nom perceptron peut désigner plusieurs neurones destinés à classifier des entrées communes, c'est un réseau de neurones artificiels dit mono-couche.

— 1960 : l'automaticien B. WIDOW développe avec son étudiant T. HOFF le modèle ADALINE (ADAptative LINear Element), une variante de l'algorithme de descente du gradient qui est à la base de l'algorithme de Rétro-propagation du Gradient utilisé dans la

phase d'apprentissage des réseaux de neurones multicouches [KS96a].

— 1969 : les scientifiques M. Minsky & S. Papert établissent les limites du perceptron [KS96a]. En effet, ce dernier peut apprendre des poids adaptés (processus d'apprentissage) si et seulement si les données sont linéairement séparables dans un plan à N dimensions, N étant le nombre de poids du neurone. Cela signifie qu'un ou plusieurs hyper-plan(s), correspondant à un ou plusieurs neurone(s), permettent de séparer dans l'espace les différentes classes. Après la publication de ces deux scientifiques, les réseaux de neurones artificiels n'ont plus suscité autant d'intérêt si bien que peu de scientifiques ont continué de les étudier.
— 1974 : le scientifique Werbos invente la méthode d'apprentissage par rétro-propagation du gradient qui permet aux perceptrons multicouches d'effectuer un processus d'apprentissage plus efficace qu'un simple gradient [KS96a].
— 1982 : le physicien Hopfield invente des neurones qui portent son nom et relance une vague d'engouement sur les réseaux de neurones qui s'était estompée après l'article de M. Minsky & S. Papert en 1969.
— 1986 : Rumelhart et Hinton créent les réseaux de neurones multi-couches et remettent au goût du jour une version adaptée de l'algorithme développé par Werbos concernant la rétro-propagation de l'erreur du gradient [KS96a].
— 1989 : G. Cybenko énonce le théorème d'approximation universel [KS96a] selon lequel un réseau de neurones n'ayant qu'une seule couche cachée avec un nombre fini de neurones peut approximer n'importe quelle fonction continue avec une fonction d'activation sigmoïde.
— 1991 : K. Hornik complète le théorème de Cybenko en spécifiant que cela est valable pour n'importe quelle fonction d'activation [KS96a]. C'est en réalité, selon lui, une propriété propre aux réseaux, permettant ainsi aux neurones artificiels d'être des approximateurs universels.
— 2006 : Hinton et ses chercheurs créent un algorithme de rétro-propagation pour les *Deep Belief Network*, des réseaux de neurones particuliers qui reconstruisent les entrées qui leur ont été soumises. Cela qui relance l'engouement quelques peu estompé des recherches sur les réseaux de neurones.

Chapitre 2

Le neurone artificiel

2.1 Le perceptron

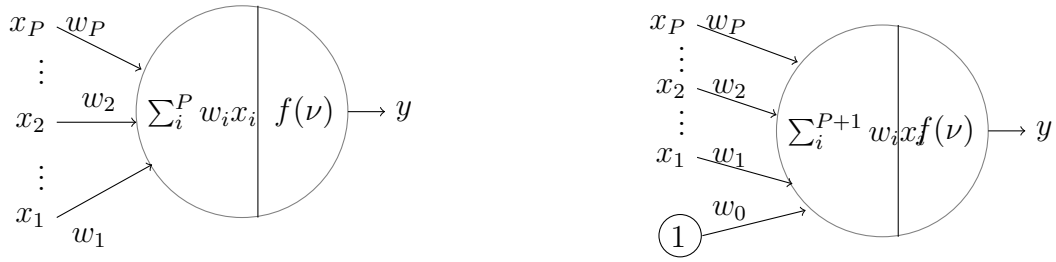
2.1.1 Théorie du perceptron

Le premier neurone artificiel, schématisé sur la Figure 2.1a, a été inventé par McCulloch et Pitts et consiste en une somme de ses entrées pondérées par des *poids*. Ce résultat, parfois appelé *potentiel* ν, est ensuite seuillé à un seuil prédéfini noté θ. La sortie y vaut 1 si le potentiel est supérieur au seuil, sinon 0. Le potentiel et la sortie se calculent selon [Fra00] :

$$\nu = \sum_{p=1}^{P} w_p x_p \text{ , avec } P \text{ le nombre de poids et donc d'entrées.}$$

$$y = \begin{cases} 1 & \text{Si } \nu > \theta \\ 0 & \text{Sinon} \end{cases}$$

Comme mentionné plus tôt, le perceptron a été établi par Rosenblatt en 1958. Son schéma, présent sur la Figure 2.1b, est pratiquement identique à celui de McCulloch et Pitts. L'apport consiste en un biais $w_0 = \theta$ pouvant être perçu comme un poids si on considère qu'il est multiplié par une variable d'entrée $x_0 = 1$. Remarquons que même si le biais peut être confondu avec un poids, il n'est pas considéré comme une dimension. Ce détail est expliqué dans la Section 2.2.3. Les poids $\mathbf{w}$ et l'exemple d'entrée $\mathbf{x}$ sont tous deux des vecteurs dans un espace à P dimensions, P étant le nombre de variables d'entrée, biais exclu, et par conséquent le nombre



(a) Principe d'un neurone artificiel selon W. McCulloch et W. Pitts.

(b) Principe d'un perceptron selon Rosenblatt .

Figure 2.1 – Les premiers neurones artificiels créés.

de poids. Le produit scalaire entre ces deux vecteurs donne le potentiel ν du neurone.

$$\nu = w_0 + \sum_{p=1}^{P} w_p x_p \text{ , si le potentiel est exprimé selon le modèle algébrique.}$$
$$\nu = \sum_{p=0}^{P+1} w_p x_p \ = \ \mathbf{x}^T \cdot \mathbf{w} \text{ , notée sous forme produit scalaire.} \tag{2.1}$$

Le scalaire y est la sortie du perceptron. Elle se calcule par la fonction d'Heaviside, notée $H(\cdot)$, dont la variable est le potentiel :

$$y \ = \ H(\nu) \ = \ H(\sum_{i=0}^{P} w_i x_i) \ = \ H(\mathbf{x}^T \cdot \mathbf{w}) \tag{2.2}$$

Le second apport, le plus important, est celui de l'apprentissage, dont l'algorithme porte le nom de *Règle du perceptron* [Fra00] [KS96a] et est plutôt simple :

1. Initialiser le perceptron avec des poids aléatoires et de préférence petits, par exemple des poids compris entre $[-2, 2]$.
2. Présenter un exemple d'entrée $\mathbf{x}$ au neurone et calculer sa sortie y.
3. Corriger les poids en leur retirant une quantité :

$$\Delta w_p = \eta(t - y)x_p \tag{2.3}$$

   Où
   — t : est la sortie réelle, qu'aurait dû avoir le neurone.
   — y : est la sortie prédite calculée par le neurone.
   — η : est un coefficient appelé *taux d'apprentissage* compris dans l'intervalle $]0, 1]$ qu'il faut déterminer empiriquement.
   — x_p : est l'entrée correspondante au poids p du neurone.
4. Recommencer à l'étape 2 pour chacun des vecteurs d'entrées disponibles.

Le but de cet algorithme est de *converger* vers un vecteur de poids de telle sorte que le neurone ne ferait plus aucune erreur de classification ou de prédiction. On parle également de *convergence absolue* lorsque le nombre d'erreurs est nul.

Il s'agit d'une procédure de correction d'erreur : les poids ne sont pas modifiés lorsque la sortie calculée est égale à la sortie attendue. On remarque que si l'entrée correspondante à un poids est nulle, aucune correction ne sera appliquée à ce poids. De moins en moins de perceptrons utilisent la notion de seuil, car les résultats sont parfois assez décevants, la convergence vers une solution robuste n'est pas assurée en pratique : la correction, lorsqu'elle a lieu, est toujours la même pour chaque poids. C'est pourquoi cette procédure ne sera pas expliquée en détail.

2.1.2 Implémentation

Un neurone, perceptron ou classique, possède deux paramètres : ses poids et sa fonction d'activation. Il contient également un potentiel, un biais et une sortie tous dépendants des entrées fournies lors d'une évaluation. Les poids peuvent être initialisés à des valeurs prédéfinies ou générées aléatoirement et dans tous les cas, les poids proches de zéros sont préférables car plus facilement corrigeables [KJ18b]. La définition de la classe **Neurone**, dans le fichier Neurone/neurone.cpp, est disponible dans le Listing 2.1.

Par ailleurs, même si le langage utilisé est le C++, la librairie développée n'utilise pas pour autant les paradigmes orientés objet. Ceux-ci compliqueraient la compréhension du lecteur non-expérimenté en programmation orienté objet. En outre, la librairie est destinée à être utilisée dans des systèmes embarqués, avec une capacité de mémoire vive et un processeur ayant des performances limitées. Le C++ possède des mécanismes internes plutôt gourmands en terme de mémoire vive ou de stockage. L'avantage de ce langage est de posséder une librairie standard suffisamment complète pour ne pas à avoir recourt à des librairies externes. Aussi, quelques mécanismes tels que les références ou le garbage collector sont utiles afin de simplifier la programmation de cette bibliothèque. En bref, l'utilisation du C++ se limite aux quelques facilités qu'il apporte.

```
class Neurone
{
    public:
        Neurone();
        Neurone(vector<double> &poids, double (*fonctionActivation)(double),
    double fonctionActivationDerivee(double)=LogistiqueDerivee);
        Neurone(int nbPoids, double (*fonctionActivation)(double), double
    fonctionActivationDerivee(double)=LogistiqueDerivee);
        ~Neurone();

        double Evaluer(const vector<double> &entrees);
        double CalculerPotentiel(const vector<double> &entrees);

        string PoidsToString();

        vector<double> poids; //paramètres du neurones
        double potentiel, y, biais; //potentiel et sortie du neurone

        double (*fonctionActivation)(double);
        double (*fonctionActivationDerivee)(double);

    private:
        Neurone(double (*fonctionActivation)(double), double (*
    fonctionActivationDerivee) (double)=LogistiqueDerivee);
        void InitPoidsGaussien(size_t nbPoids, double moyenne, double ecarttype);
};
```

Listing 2.1 – La classe Neurone définie dans Neurone/neurone.h.

```
Neurone::Neurone()
{
    biais=1.0f;
    potentiel=0.0f;
    y=0.0f;
}

Neurone::Neurone(double (*fonctionActivation)(double), double (*
    fonctionActivationDerivee)(double))
: Neurone()
{
    this->fonctionActivation=fonctionActivation;
    this->fonctionActivationDerivee=fonctionActivationDerivee;
}

Neurone::Neurone(vector<double> &poids, double (*fonctionActivation)(double),
    double fonctionActivationDerivee(double))
: Neurone(fonctionActivation, fonctionActivationDerivee)
{
```

```
    this->poids=poids;
}

Neurone::Neurone(int nbPoids, double (*fonctionActivation)(double), double
    fonctionActivationDerivee(double))
: Neurone(fonctionActivation, fonctionActivationDerivee)
{
    InitPoidsGaussien(nbPoids, 0, 1);
}
```

Listing 2.2 – Les différents constructeurs permettant d'initialiser un neurone.

Les différents constructeurs disponibles dans la librairie développée se trouvent dans le Listing 2.2. Le premier constructeur sans paramètre `Neurone();`, permet d'initialiser les variables. Il n'est pas destiné à être appelé dans du code client, mais par les autres constructeurs.

Le constructeur `Neurone(double (*fonctionActivation)(double), double (*fonctionActivationDerivee) (double)=LogistiqueDerivee);` permet de spécifier la fonction d'activation et sa dérivée. Cependant, celui-ci est privé, car la notion de poids n'est pas présente et il faut obligatoirement définir les poids d'une manière ou d'une autre ! La dérivée de la fonction d'activation, contenue dans le *pointeur de fonction* **fonctionActivationDerivee**, sera utile lors de l'implémentation d'un réseau de neurones multi-couches qui apprennent avec la rétro-propagation du gradient de l'erreur. Cet algorithme est utilisé dans l'apprentissage de perceptrons multi-couches, détaillé dans le chapitre 3. Si le neurone est destiné à être utilisé sans réseau, tel que dans le cas présent, alors la dérivée de la fonction d'activation peut-être omise : celle-ci possède une valeur par défaut, n'ayant aucune importance.

Le constructeur suivant `Neurone(int nbPoids, double (*fonctionActivation)(double), double fonctionActivationDerivee(double)=LogistiqueDerivee);` voit son utilité principalement dans des objectifs de *debug*, lorsque le programmeur souhaite reproduire un certain comportement afin de l'analyser. Par conséquent, ce constructeur permet de spécifier les poids un à un ainsi que la fonction d'activation.

Enfin, le dernier constructeur `Neurone(vector<double> &poids, double (*fonctionActivation)(double), double fonctionActivationDerivee(double)=LogistiqueDerivee);`, sera sans nul doute le plus fréquent. Celui-ci permet d'initialiser un nombre **nbPoids** de poids pseudo-aléatoirement selon une distribution Gaussienne. La fonction initialisant les poids est présentée dans le Listing 2.3.

```
void Neurone::InitPoidsGaussien(size_t nbPoids, double moyenne, double ecarttype
    )
{
    poids=vector<double>(nbPoids);
    random_device rd;
    mt19937 e2(rd());
    normal_distribution<double> dist(moyenne, ecarttype);
    for(size_t i=0; i<nbPoids; i++)
        poids[i]=dist(e2);
}
```

Listing 2.3 – La fonction qui permet d'initialiser les poids avec des valeurs selon une loi Gaussienne, dont les paramètres sont la moyenne et l'écart-type de la distribution.

Une fonction utile à des fins de débug est **PoidsToString()**, disponible dans le Listing 2.4. Elle insère les poids dans un flux qui sera ensuite transformé en un string. De cette manière, les poids sont affichables dans un terminal ou même dans une interface graphique.

```
// Utile à des fins de debugs
string Neurone::PoidsToString()
{
    ostringstream strs;
    for(double p: poids) // Concatène les poids
        strs << p << '\t';

    strs << "biais: " << biais; // Ajoute le biais à la fin
    return strs.str(); // Renvoie le contenu du flux sous forme de string
}
```

Listing 2.4 – La fonction **PoidsToString** permettant de récupérer les poids sous forme de string

Les fonctions d'activation sont définies dans l'en-tête *Fonction/fonctionactivation.cpp*, dont le code source se trouve dans le Listing 2.5. Ces fonctions sont définies de façon identiques afin de pouvoir être utilisée sans distinction par la classe **Neurone** et son pointeur **fonctionActivation**.

```
#include "fonctionactivation.h"

double Logistique(double x)
{
    return 1/(1+exp(-x));
}

double LogistiqueDerivee(double x)
{
    double fx = Logistique(x);
    return fx*(1-fx);
}

double TangenteHyperbolique(double x)
{
    return tanh(x);
}

double TangenteHyperboliqueDerivee(double x)
{
    return 1 - tanh(x) * tanh(x);
}

double Heaviside(double x)
{
    if(x<0)
        return 0;
    else
        return 1;
}

double Signe(double x)
{
    if(x>0.0f)
        return 1;
    else
```

```
        return -1;
38  }
```

Listing 2.5 – Les différentes fonctions d'activation dans Fonction/fonctionactivation.cpp.

Afin de calculer la sortie d'un neurone, un vecteur d'entrée est requis. La somme pondérée est effectuée par la fonction **CalculerPotentiel** selon la formule (2.1) et la sortie du neurone est ensuite retournée par l'appel de la fonction d'activation.

```
double Neurone::Evaluer(const vector<double> &entrees)
{
    potentiel=CalculerPotentiel(entrees);
    y=fonctionActivation(potentiel);
    return y;
}

double Neurone::CalculerPotentiel(const vector<double> &entrees)
{
    double tmp=0.0f;
    for(size_t i=0; i<entrees.size(); i++)
        tmp+=entrees[i]*poids[i];

    return tmp+biais;
}
```

Listing 2.6 – Le code pour calculer le potentiel ainsi que la sortie du neurone sur base d'un vecteur d'entrée.

2.2 La Descente de Gradient

2.2.1 Théorie de la Descente de Gradient

La Descente de Gradient est un algorithme d'apprentissage pratiquement identique à la Règle du Perceptron. La différence est que la sortie du neurone considérée par l'algorithme n'est plus issue de la fonction d'activation Heaviside mais d'une fonction continue. Ainsi, plus la sortie du neurone est éloigné de la sortie attendue, plus la correction appliquée aux poids sera importante [Fra00].

De manière générale, la méthode du gradient est une méthode d'optimisation servant à trouver itérativement un minimum $\mathbf{x}_0$ d'une fonction $f(\mathbf{x})$ dérivable en calculant itérativement le gradient de cette fonction dépendante de la variable vecteur $\mathbf{x}$. L'algorithme suppose que f possède effectivement le minimum recheché en $\mathbf{x}_0$. Afin de se déplacer vers ce minimum, le déplacement se fait selon une ligne droite dans la direction de $-\nabla f(\mathbf{x})$, une quantité aussi appelée la *direction de la descente la plus raide* ou *steepest descent* en anglais [Alf00]. De telle sorte que :

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla f(\mathbf{x}_n) \text{ , où } n \text{ désigne le numéro de l'itération et } \eta \text{ le pas.}$$

Où $f(\mathbf{x}_n) \geq f(\mathbf{x}_{n+1})$ si η est suffisamment petit. Au fil des itérations, $f(\mathbf{x}_n)$ va donc décroître et une suite sera obtenue, de telle sorte que : $f(\mathbf{x}_n) \geq f(\mathbf{x}_{n+1}) \geq f(\mathbf{x}_{n+2}) \geq f(\mathbf{x}_{n+3}) \ \ldots$ [Alf00].

Dans le cas présent, la fonction d'erreur $E(\mathbf{w}, \mathbf{x})$ joue le rôle de $f(\mathbf{x})$ et $\nabla f(\mathbf{x})$ est jouée par le vecteur $\nabla E(\mathbf{w}, \mathbf{x})$. Quant à $\mathbf{w}$, il correspond à la variable $\mathbf{x}$. Ce qui revient à :

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E(\mathbf{w}, \mathbf{x}) \text{ , où } t \text{ symbolise le fait que ce soit itératif.}$$

En outre, cette méthode calcule le gradient de la fonction d'erreur. Cela implique que la fonction d'activation du perceptron doit être continue et dérivable. la fonction d'activation choisie avec cet algorithme est la fonction **Linéaire** $f(x) = x$ au détriment des fonctions d'Heaviside et **Signe**.

Tout d'abord, le choix d'une fonction d'erreur pour la correction du perceptron s'impose. La fonction d'erreur préférée est la Quadratique [KS96a] [Gé07] :

$$E = \frac{1}{2} \sum_{k=1}^{K} (t_k - y_k)^2$$

Avec K le nombre d'exemples et t_k la sortie souhaitée pour le $k$ème exemple et y_k la sortie prédite par le neurone pour ce même exemple k.

Si la fonction d'erreur est telle que $E = 0$, alors le perceptron classifie correctement tous les échantillons. De plus, cette fonction est extensible à plusieurs perceptrons situés dans la même couche, il suffirait de faire la somme des erreurs de chacun des neurones de la couche pour chacun des exemples pour obtenir l'erreur totale. Le but de l'algorithme est de minimiser l'erreur du perceptron. Dans le cas présent, la fonction d'erreur est dépendante de y lui même dépendant des poids et des entrées. La formulation de l'erreur quadratique devient :

$$E(\mathbf{w}, \mathbf{x}) = \frac{1}{2} \sum_{k=1}^{K} [t_k - y_k(\mathbf{w}, \mathbf{x}_k)]^2 \tag{2.4}$$

Où $\mathbf{w}$ désigne les poids du neurone considéré, K le nombre d'exemples et $\mathbf{x}_k$ l'exemple k analysé.

Le but de la Descente de Gradient est d'appliquer une variation $\Delta\mathbf{w}$ au vecteur de poids $\mathbf{w}$ proportionnelle à l'opposé de la dérivée de l'erreur mesurée, afin de descendre la pente de la fonction et d'arriver au minimum de la fonction [Gé07]. Il est donc nécessaire de calculer cette valeur de la pente, qui n'est autre que le gradient de la fonction. Pour un seul neurone de sortie :

$$\Delta w_p = \nabla E(\mathbf{w}, \mathbf{x}) = \frac{\partial E(\mathbf{w}, \mathbf{x})}{\partial w_p} \text{ , la correction à appliquer pour le poids } p.$$

Grâce à règle de la dérivée d'une fonction composée, on peut également écrire :

$$\frac{\partial E(\mathbf{w}, \mathbf{x})}{\partial w_p} = \frac{\partial E(\mathbf{w}, \mathbf{x})}{\partial \boldsymbol{\nu}} \frac{\partial \boldsymbol{\nu}}{\partial w_p}$$

Où $\boldsymbol{\nu}$ est le vecteur de K éléments contenant le potentiel pour chaque exemple d'entrée $\mathbf{x}_k$ avec k allant de 1 à K.

Étant donné que le potentiel est une fonction linéaire, il advient, pour le $k$ème exemple de l'ensemble d'apprentissage :

$$\frac{\partial \nu_k}{\partial w_p} = x_{kp} \text{ , la valeur de l'entrée correspondant au poids } p \text{ de l'exemple } k.$$

Et :

$$\frac{\partial E(\mathbf{w}, \mathbf{x})}{\partial \boldsymbol{\nu}} = -\sum_{k=1}^{K} (t_k - y_k)$$

On obtient finalement une expression aisément calculable :

$$\Delta w_p = -\sum_{k=1}^{K}(t_k - y_k)x_{kp} \tag{2.5}$$

Le poids du neurone i à l'itération $t+1$ est :

$$w_p(t+1) = w_p(t) - \eta\left(-\sum_{k=1}^{K}(t_k - y_k)x_{kp}\right) = w_p(t) + \eta\left(\sum_{k=1}^{K}(t_k - y_k)x_{kp}\right) \tag{2.6}$$

La correction Δw_p pour un poids w_p est donc dépendante de l'ensemble des entrées x_k du jeu de données disponible. Evidemment, cette correction doit être appliquée pour tous les poids du neurone considéré. De plus, tout comme pour la correction du perceptron, un taux d'apprentissage η est utilisé :

- Le terme correcteur appliqué aux poids est : $w_p(t+1) = w_p(t) + \eta\Delta w_p$ [Fra00].
- $\eta \in]0, 1]$, qu'il faut déterminer empiriquement.
- Un trop grand taux d'apprentissage risque de faire osciller l'erreur autour du minimum sans jamais converger.
- Un trop petit taux d'apprentissage risque un nombre d'itérations très élevé et donc une convergence de l'erreur très lente voir impossible.

2.2.2 Implémentation

Traditionnellement, la Descente de Gradient est réglée avec un seuil de tolérance et il s'arrête lorsque l'erreur atteint ce seuil [Fra00]. L'adoption d'un nombre d'erreurs nul à approcher à été préférée, où la sortie du neurone seuillée par la fonction d'activation d'Heaviside, est comparée à la sortie attendue. L'avantage principal que procure cette implémentation est que ces deux sorties sont différentes en cas d'erreur, mais elles sont strictement identiques en cas de bonne prédiction. Tandis que la fonction **Linéaire** retourne un résultat se *rapprochant* de la sortie attendue, mais n'atteint jamais ou rarement la réponse attendue.

Par ailleurs, de façon empirique, la classification avec la Descente de Gradient est plus efficace avec deux classes -1 et 1 qu'avec les classes traditionnelles 0 et 1. Cela nécessite un remplacement de la fonction d'activation d'Heaviside par la fonction **Signe**.

Algorithme 1 : Algorithme de la descente du gradient [Fra00].

Entrées : neurone : le perceptron à corriger ayant P poids,
$\mathbf{x} = (\mathbf{x}_1, ..., \mathbf{x}_K)$: la matrice des exemples d'apprentissage, dont chaque élément est un vecteur d'entrée,
$\mathbf{y} = (y_1, ..., y_K)$: la matrice des réponses aux exemples d'apprentissage dont chaque élément est la valeur de la réponse attendue,
nbIterations : le nombre d'itérations maximum que l'algorithme doit faire,
tauxApprentissage : le taux d'apprentissage.
Output : nbErreurs : la quantité d'erreurs résiduelle dans l'évaluation des exemples d'apprentissage post-apprentissage.

```
1  nbErreurs = ∞
2  pour Nb itérations > nbIterations ET nbErreurs > 0 faire
3  |  nbErreurs = 0
4  |  deltaBiais = 0
5  |  deltaPoids[Nombre Poids] = {0}
6  |  pour Chacun des exemples k de 1 à K faire
7  |  |  sortie = Évaluer sortie pour l'exemple x[k] selon la l'équation (2.1)
8  |  |  delta = (y[k] - potentiel du neurone ) * tauxApprentissage
   |  |  // Accumuler les variations à appliquer aux poids et au biais
9  |  |  pour Chacun des poids p du neurone de 1 à P faire
   |  |  |  // Calculer les corrections à appliquer aux poids grâce à la
   |  |  |     formule (2.5)
10 |  |  |  deltaPoids[p] += delta * x[k][p]
11 |  |  |  deltaBiais += delta
12 |  |  fin
13 |  |  si sortie != y[k] alors
14 |  |  |  nbErreurs += 1
15 |  |  fin
16 |  fin
   |  // Mettre à jour les poids après avoir parcouru l'ensemble des exemples
   |     en utilisant la formule (2.6)
17 |  pour Chacun des poids p du neurone de 1 à P faire
18 |  |  poids[p] += deltaPoids[p]
19 |  fin
20 |  biais += deltabiais
21 fin
22 retourner nbErreurs
```

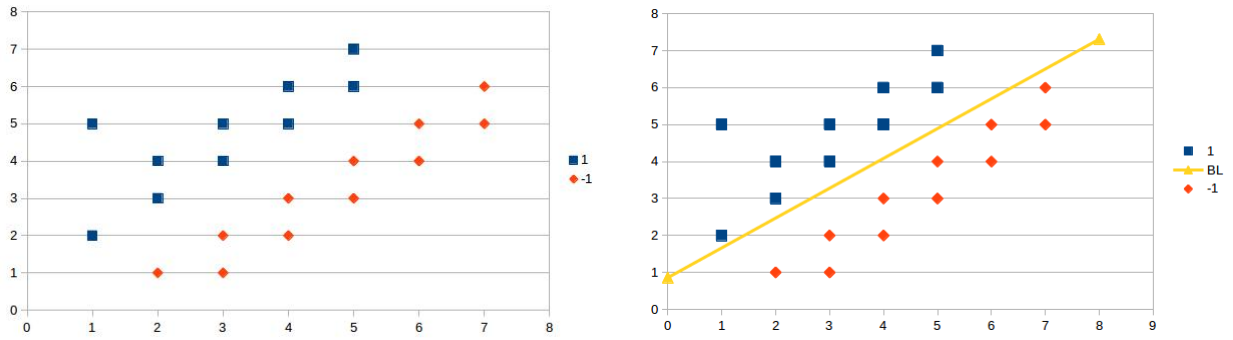
Le code C/C++ de la fonction **DescenteGradient** définie dans Apprentissage/apprentissage.cpp :

```
int DescenteGradient(Neurone &n, vector<vector<double>> &entrees, vector<double>
    &sortiesDesirees, size_t nbIterations, double tauxApprentissage)
{
   size_t nbErreurs=(size_t) INFINITY;
   double erreur=0.0f;

   for(size_t i=0; i<nbIterations && nbErreurs>0; i++)
   {
      nbErreurs=0;
      erreur=0.0f;
      vector<double> deltaPoids(n.poids.size()); // init à 0
      double deltaB=0.0f;

      //Présenter un vecteur d'entrée X et évaluer la sorties du neurone
      for(size_t j=0; j<entrees.size(); j++)
      {
         vector<double> exemple=entrees[j];
         //pour chaque neurone
         double sortieTmp = n.Evaluer(exemple), //1. calculer ma sortie

         // Calculer les corrections avec la variation du poids
         //   MAJ les poids: W(l+1) = w(l) + \eta Somme((t-y)x_i) : t=target
   output, l= indice d'itération, eta: gain
         diff = (sortiesDesirees[j] - n.potentiel);
         erreur += 0.5*pow(diff,2);
         deltaB += tauxApprentissage * diff;

         for(size_t k=0; k<deltaPoids.size(); k++)
            deltaPoids[k] += tauxApprentissage * diff * exemple[k];

         if(sortiesDesirees[j]!=sortieTmp)
            nbErreurs++;
      }

      //MAJ poids après l'analyse de tous les exemples
      for(size_t k=0; k<n.poids.size(); k++)
         n.poids[k] += deltaPoids[k];

      n.biais += deltaB; // MAJ biais

      cout << "Iteration " << i << "\tPoids: " << n.PoidsToString()   <<
      "\tNombre d'erreurs: " << nbErreurs << "\tErreur: " << erreur << endl;
   }

   return nbErreurs;
}
```

Listing 2.7 – L'implémentation C de l'algorithme de Descente du Gradient dans Apprentissage/apprentissage.cpp

2.2.3 Résultats et limites de la Descente de Gradient

De façon générale, le résultat produit par un perceptron est un hyper-plan, appelé *limite de décision*, de dimensions dans $\mathbb{R}^P$, avec P le nombre de poids et donc d'entrées, séparant deux classes dans l'hyper-espace. Chaque espace de part et d'autre



(a) (b)

FIGURE 2.2 – À gauche, les données que l'on souhaite séparer. À droite, les données séparées par une droite, aussi appelée limite de décision.

de la limite de décision correspondra à une classe. Une explication sur le traçage de limite de décision est fournie en Annexe A.1.

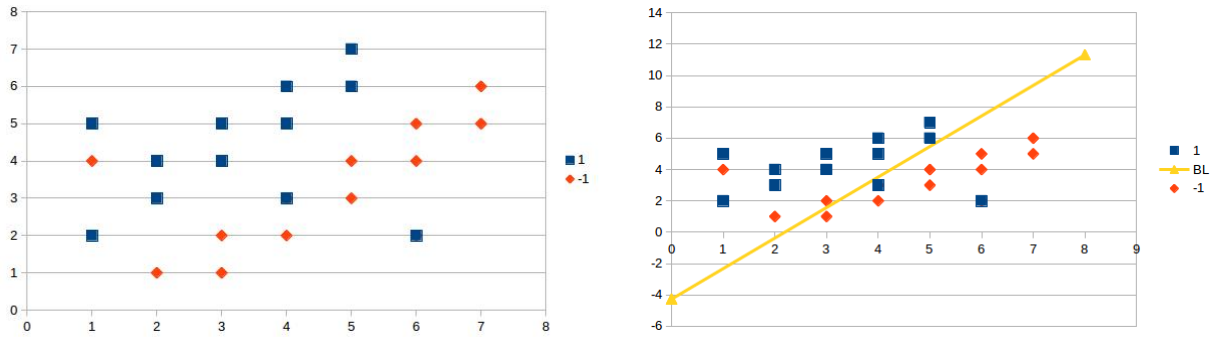
Imaginons un cas simple à deux entrées et par conséquent contenant deux poids, l'hyperplan est alors une droite qui va séparer deux classes quelconques dans $\mathbb{R}^2$. Les données sont représentées sur le graphique en Figure 2.2a. Sur cette image, les données sont réparties en deux classes. Celles-ci sont peu nombreuses et disposent d'un espace étroit où la limite de décision devrait normalement pouvoir s'insérer. Le code faisant référence à cet essai se trouve dans le Listing 2.8.

```
#include <Neurone/neurone.h>
#include <Apprentissage/apprentissage.h>

int main()
{
  vector<vector<double>> x;
  vector<double> y;
  size_t nbEntrees=2;
  double (*ptrfa)(double)=Signe;

  /* 2 = le nombre de poids et d'entrées
   * ptrfa = le pointeur de la fonction d'activation, Signe dans ce cas */
  Neurone n(nbEntrees, ptrfa);

  /* "/home/julien/workspace/NeuroLibTest/separation_simple_adaline.csv" =
   Chemin du fichier contenant les données
   * x = matrice contenant les variables d'entrées des différents exemples
   * nbEntree = nombre de variables d'entrées pour chacun des exemples contenus
   dans le fichier, 2 dans ce cas
   * y = matrice où une ligne va contenir l'étiquette correspondant à l'exemple.
    Il y a autant de colonnes que de classes différentes contenues dans le jeu
   de données */
  LireFichierCSV("/home/julien/workspace/NeuroLibTest/separation_simple_adaline.
   csv", x, nbEntrees, y);

  /* n = le neurone qu'il faut entrainer
   * x = le vecteur deux dimensions contenant les exemples
   * y = les étiquettes du vecteur d'entrées
   * 50 = le nombre maximum d'itérations
   * 0.01 = le taux d'apprentissage */
   DescenteGradient(n, x, y, 50, 0.0001);
```

```
28    cout << "Poids après apprentissage: " << endl << n.PoidsToString() << endl;
}
30
// Iteration 0  Poids: -0.617079  0.142073  biais: 1.00184  Nombre d'erreurs: 6
32 ...
// Iteration 49 Poids: -0.523186  0.274738  biais: 1.01407  Nombre d'erreurs: 0
34 // Poids après apprentissage:
// -0.523186  0.274738  biais: 1.01407
```

Listing 2.8 – Apprentissage sur des données quelconques comportant deux classes.

Le Listing 2.8 donne les poids, obtenus après 50 itérations avec l'algorithme de la Descente de Gradient, permettant de tracer la droite, représentée sur la Figure 2.2b, séparant de justesse les deux classes.



(a) (b)

FIGURE 2.3 – À gauche, les données falsifiées que l'on souhaite à nouveau séparer. À droite, la limite de décision.

De même, en introduisant quelques erreurs dans le jeu de données, disponible sur la Figure 2.3a, l'algorithme converge également vers une solution séparant un maximum de points dans les différentes classes, la limite de décision obtenue est visible sur la Figure 2.3b. En regardant la Figure 2.2b, il est plus aisé de comprendre pourquoi le biais n'est pas considéré comme une dimension. Le biais est un poids particulier autorisant la limite de décision à prendre n'importe quel pente par rapport à l'origine.

Par ailleurs, en pratique, trouver un taux d'apprentissage adapté n'est pas toujours évident. La technique consiste à commencer avec un taux choisi au hasard et d'augmenter ou diminuer l'ordre de grandeur en fonction du nombre d'erreur. Si celui-ci ne fait qu'augmenter, le taux d'apprentissage est trop élevé et son ordre de grandeur doit être diminué. Au contraire, si le nombre d'erreurs est stationnaire ou diminue, mais converge lentement, il est alors nécessaire d'augmenter son ordre de grandeur. Une fois qu'un ordre de grandeur acceptable a été trouvé, la méthode de la dichotomie peut être utilisée afin de trouver le taux d'apprentissage optimal.

Le Listing 2.9 présente des sorties de programme avec des taux d'apprentissage différents. Le premier taux est $1E-4$ et le nombre d'erreur diminue au fil des itérations sans pour autant atteindre 0. Ceci est dû au fait que le taux d'apprentissage est trop faible et l'algorithme ne converge pas assez rapidement. Le taux d'apprentissage suivant est $1E-3$ et converge après 46 itérations. Ensuite, le taux d'apprentissage est augmenté à $1E-2$. Ce dernier ne converge pas réellement, les poids deviennent très grands et le nombre d'erreurs n'a diminué que d'une

seule unité. Ceci est simplement dû à un taux d'apprentissage trop élevé et n'apporte pas une correction efficace aux poids du neurone.

```cpp
#include <Neurone/neurone.h>
#include <Apprentissage/apprentissage.h>

void Test(double tauxApprentissage, vector<vector<double>> &x, vector<double> &y
    )
{
  size_t nbEntrees=2;
  double (*ptrfa)(double)=Signe;//pointeur fonction activation

  /* 2 = le nombre de poids et d'entrées
   * ptrfa = le pointeur de la fonction d'activation, Heaviside dans ce cas */
  Neurone n(nbEntrees, ptrfa);

  cout << "Taux d'apprentissage  = 1E-4" << endl;
  /* n = le neurone qu'il faut entrainer
   * x = le vecteur deux dimensions contenant les exemples
   * y = les étiquettes du vecteur d'entrées
   * 50 = le nombre maximum d'itérations
   * tauxApprentissage = le taux d'apprentissage */
  DescenteGradient(n, x, y, 50, tauxApprentissage);
}

int main()
{
  vector<vector<double>> x;
  vector<double> y;

  /* "/home/julien/workspace/NeuroLibTest/separation_simple_adaline.csv" =
    Chemin du fichier contenant les données
   * x = matrice contenant les variables d'entrées des différents exemples
   * nbEntree = nombre de variables d'entrées pour chacun des exemples contenus
    dans le fichier, 2 dans ce cas
   * y = matrice où chaque ligne va contenir l'étiquette correspondant à l'
    exemple. Il y aura autant de colonnes que de classes différentes contenues
    dans le jeu de données */
  LireFichierCSV("/home/julien/workspace/NeuroLibTest/separation_simple_adaline.
    csv", x, nbEntrees, y);

  cout << "Taux d'apprentissage  = 1E-4" << endl;
  Test(0.0001, x, y);

  cout << "Taux d'apprentissage  = 1E-3" << endl;
  Test(0.01, x, y);

  cout << "Taux d'apprentissage  = 1E-2" << endl;
  Test(0.1, x, y);
}

// Taux d'apprentissage  = 1E-4
//Iteration 0 Poids: -1.81132 -0.598924 biais: 1.01905  Nombre d'erreurs: 10
...
// Iteration 49 Poids: -0.76412 0.424944  biais: 1.23421  Nombre d'erreurs: 2

// Taux d'apprentissage  = 1E-3
// Iteration 0  Poids: -0.207743  -0.144692 biais: 0.95585  Nombre d'erreurs: 11
...
// Iteration 46 Poids: -0.585495  0.397047  biais: 0.766248 Nombre d'erreurs: 0
```

```
53 // Taux d'apprentissage  = 1E-2
   // Iteration 0  Poids: -3.59708 -3.01482  biais: 0.152847 Nombre d'erreurs: 11
55 ...
   // Iteration 49 Poids: 3.94918e+40  3.96674e+40 biais: 8.86153e+39  Nombre d'
      erreurs: 10
```

Listing 2.9 – Tests de différents taux d'apprentissage

2.3 ADALINE

2.3.1 La théorie d'ADALINE

Une amélioration de l'algorithme de Descente de Gradient a été proposée par Widrow et Hoff début des années 60 et se nomme *ADALINE* pour ADAptive LINear Element. Au lieu de calculer la variation des poids pour l'ensemble des exemples disponibles, la modification est appliquée après chaque exemple analysé [Fra00] [KS96a].

2.3.2 Implémentation

L'impémentation de l'algorithme ADALINE est très semblable à la Descente de Gradient. La seule différence est que les poids sont directement mis à jour après le calcul de $\nabla E(\mathbf{x}, \mathbf{w})$ selon l'Équation (2.6).

La mise à jour des poids dans la Descente de Gradient qui s'effectuait une fois que tous les exemples analysés est maintenant exécutée pour chacun des exemples, de la ligne 7 à 10.

```
1  int DescenteGradientWidrowHoff(Neurone &n, vector<vector<double>> &entrees,
       vector<double> &sortiesDesirees, size_t nbIterations, double
       tauxApprentissage)
   {
3    size_t nbErreurs=(size_t) INFINITY;

5    for(size_t i=0; i<nbIterations && nbErreurs>0 ; i++)
     {
7      nbErreurs = 0;
       double erreur = 0.0f;
9
       //Présenter un vecteur d'entrée X et évaluer la sorties du neurone
11     for(size_t j=0; j<entrees.size(); j++)
       {
13       vector<double> exemple=entrees[j];
         //pour chaque neurone
15       double sortieTmp=n.Evaluer(exemple), //1. calculer ma sortie
         diff = tauxApprentissage * (sortiesDesirees[j] - n.potentiel) ; //2.
       calculer l'erreur entre prédit et non prédit
17       erreur += 0.5 * pow(diff,2);
         n.biais += diff;
19
         //  MAJ les poids: W(l+1) = w(l) + \eta Somme((t-y)x_i) : t=target output,
       l= indice d'itération, eta: gain/learning rate
21       for(size_t k=0; k<n.poids.size(); k++)
           n.poids[k] += diff * exemple[k];
23
         if(sortiesDesirees[j]!=sortieTmp)
25         nbErreurs++;
       }
```

Algorithme 2 : Algorithme de la descente du gradient de Widrow et Hoff (ADALINE) [Fra00].

Entrées : neurone : le perceptron à corriger ayant P poids,
$\mathbf{x} = (\mathbf{x}_1, ..., \mathbf{x}_K)$: la matrice des exemples d'apprentissage, dont chaque élément est un vecteur d'entrée,
$\mathbf{y} = (y_1, ..., y_K)$: la matrice des réponses aux exemples d'apprentissage dont chaque élément est la valeur de la réponse attendue,
nbIterations : le nombre d'itérations maximum que l'algorithme doit faire,
tauxApprentissage : le taux d'apprentissage.
Output : nbErreurs : la quantité d'erreurs résiduelle dans l'évaluation post-apprentissage des exemples

```
1  nbErreurs = ∞
2  pour Nb itérations < nbIterations ET nbErreurs > 0 faire
3  |  nbErreurs = 0
4  |  pour Chacun des exemples k de 1 à K faire
5  |  |  sortie = Evaluer sortie pour l'exemple x[k] selon la l'équation (2.1)
   |  |  // Calculer les corrections à appliquer aux poids grâce à la
   |  |     formule (2.5)
6  |  |  delta = (y[k] - potentiel du neurone) * tauxApprentissage
7  |  |  pour Chacun des poids p du neurone de 1 à P faire
   |  |  |  // Mettre à jour les poids via la formule (2.6)
8  |  |  |  poids[p] += delta * x[k][p]
9  |  |  fin
10 |  |  biais += delta
11 |  |  si sortie != y[k] alors
12 |  |  |  nbErreurs += 1
13 |  |  fin
14 |  fin
15 fin
16 retourner nbErreurs
```

```
27      cout << "Iteration " << i << "\tPoids: " << n.PoidsToString()  <<
          "\tNombre d'erreurs: " << nbErreurs << "\tErreur: " << erreur << endl;
29    }

31    cout << "Poids finaux : " << n.PoidsToString() << endl;

33    return nbErreurs;
  }
```

Listing 2.10 – L'implémentation C de l'algorithme de Descente du Gradient ADALINE

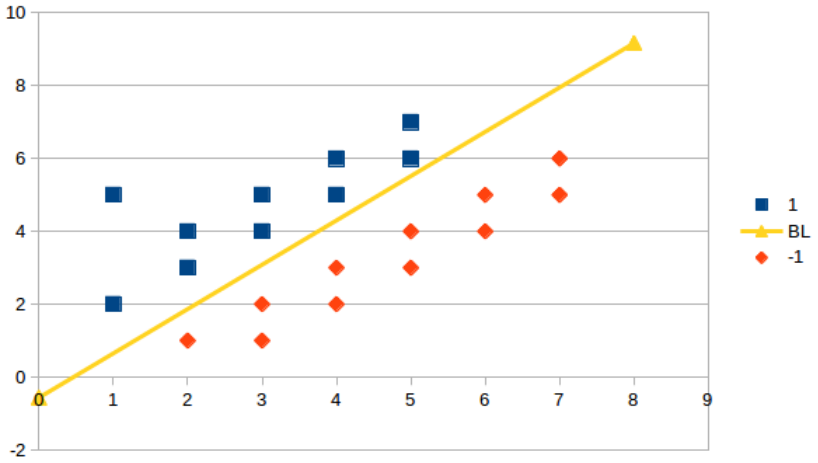


FIGURE 2.4 – Limite de décision obtenue avec l'exemple d'utilisation d'ADALINE.

2.3.3 Résultats et limites d'ADALINE

```
#include <Neurone/neurone.h>
#include <Apprentissage/apprentissage.h>

int main()
{
  vector<vector<double>> x;
  vector<double> y;
  size_t nbEntrees=2;
  double (*ptrfa)(double)=Signe;

  /* 2 = le nombre de poids et d'entrées
   * ptrfa = le pointeur de la fonction d'activation, Signe dans ce cas */
  Neurone n(nbEntrees, ptrfa);

  /* "/home/julien/workspace/NeuroLibTest/separation_simple_adaline.csv" =
   Chemin du fichier contenant les données
   * x = matrice qui va contenir les variables d'entrées des différents exemples
   * nbEntree = nombre de variables d'entrées pour chacun des exemples contenus
   dans le fichier, 2 dans ce cas
   * y = matrice où chaque ligne va contenir l'étiquette correspondant à l'
   exemple. Il y aura autant de colonnes que de classes différentes contenues
   dans le jeu de données */
   LireFichierCSV("/home/julien/workspace/NeuroLibTest/separation_simple_adaline
   .csv", x, nbEntrees, y);

  /* n = le neurone qu'il faut entrainer
   * x = le vecteur deux dimensions contenant les exemples
   * y = les étiquettes du vecteur d'entrées
   * 50 = le nombre maximum d'itérations
   * 0.01 = le taux d'apprentissage */
   DescenteGradient(n, x, y, 50, 0.01);

   cout << "Poids après apprentissage: " << endl << n.PoidsToString() << endl;
}

// Iteration 0 Poids: -0.00345609 -0.438309 biais: 1.05562 Nombre d'erreurs:
   0 Erreur Moyenne : 1.50717
...
// Iteration 21 Poids: -0.459216 0.314429 biais: 0.566731 Nombre d'erreurs: 0
   Erreur Moyenne : 0.122743
```

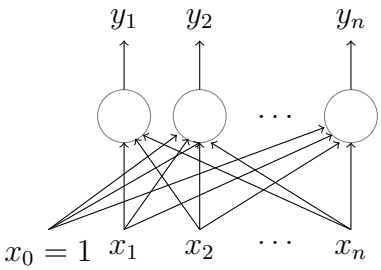


FIGURE 2.5 – Un perceptron mono-couche à n neurones de sortie.

```
34 // Poids après apprentissage:
   // -0.432623  0.355637  biais: 0.200844
```

Listing 2.11 – Apprentissage sur des données quelconques comportant deux classes.

Cet algorithme a nécessité moins d'itérations que la Descente de Gradient. La classification se fait sans erreur après apprentissage et l'erreur reste toutefois sembable à celle de la Descente de Gradient. Cet algorthime a rendu la Descente de Gradient du perceptron obsolète. Cependant, les exemples analysés concernent une classification binaire. S'il y a plus de deux classes, il faut alors recourir au perceptron mono-couche.

2.4 Le perceptron mono-couche

2.4.1 La théorie du perceptron mono-couche

La notion de perceptron s'étend aussi à une seule couche de neurones suivant l'algorithme du perceptron tel que représenté sur la figure 2.5. Toutes les entrées sont communes aux neurones et les neurones ne sont pas connectés entre eux : ils sont totalement indépendants. La sortie d'un neurone vaudra 1 si l'exemple analysé appartient à la classe que représente le neurone, sinon -1. De plus, chaque neurone fournira une limite de décision entre sa classe 1 et le reste des classes. L'apprentissage d'un neurone parmi ceux présents dans la couche, sera également indépendant des autres neurones.

2.4.2 Implémentation

Pour implémenter un perceptron mono-couche, il suffit d'instancier autant de perceptron qu'il y a de classes à distinguer et de les entraîner sur le même jeu de données sans oublier d'adapter le contenu des vecteurs de sorties désirées. Par ailleurs, la fonction **LireFichierCSV**, présente dans le Listing 2.12, permet d'extraire hors d'un fichier CSV les vecteurs d'entrées et les sorties attendues et de les assigner aux paramètres **x** et **y**. Les fichiers sont organisés en lignes, où une ligne représente un vecteur d'entrée, symbolisée par x suivi de la ou les sorties souhaitées, symbolisées par y. Exemple des entrées contenues dans le fichier *iris-3classes-adaline.csv* utilisé dans le Listing 2.12 :

x_1	x_2	y_1	y_2	y_3
3.1	2.3	-1	-1	1
2.8	2.4	-1	-1	1
-1.2	1.5	1	1	-1
...	...	...	...	...

```
#include <Neurone/neurone.h>
#include <Apprentissage/apprentissage.h>

void main(
{
    vector<vector<double>> x;
    vector<double> y;
    double (*ptrfa)(double)=Signe;
    /* 2 = nombres de poids et par conséquent d'entrées
     * ptrfa = pointeur fonction d'activation, Signe dans le cas présent */
     Neurone n1(2, ptrfa), n2(2, ptrfa), n3(2, ptrfa);

     /* "/home/julien/workspace/NeuroLibTest/iris-3classes-adaline.csv" = Chemin
    du fichier contenant les données
      * x = matrice contenant les variables d'entrées des différents exemples
      * 2 = nombre de variables d'entrées pour chacun des exemples contenus dans
    le fichier
      * y = matrice où chaque ligne va contenir l'étiquette correspondant à l'
    exemple. Il y aura autant de colonnes que de classes différentes contenues
    dans le jeu de données
      * 3 = Nombre de classes différentes */
     LireFichierCSV("/home/julien/workspace/NeuroLibTest/iris-3classes-adaline.
    csv", x, 2, y, 3);

    // transposer matrice y (150x3) en (3 (=neurones) x 150 (=exemples) )! :
    vector<vector<double>> y1(3);
    for(size_t i=0; i<y1.size(); i++)
        y1[i]=vector<double>(y.size());

    for(size_t i=0; i<y.size(); i++)
        for(size_t j=0; j<3; j++)
            y1[j][i] = y[i][j];

    /* n = le neurone qu'il faut entrainer
     * x = le vecteur deux dimensions contenant les exemples
     * y1[i] = les étiquettes du vecteur d'entrées correspondant au neurone i
     * 50 = le nombre maximum d'itérations
     * 0.01 = le taux d'apprentissage */
    DescenteGradientWidrowHoff(n1, x, y1[0], 50, 0.01);
    DescenteGradientWidrowHoff(n2, x, y1[1], 50, 0.01);
    DescenteGradientWidrowHoff(n3, x, y1[2], 50, 0.01);
}
```

Listing 2.12 – Test de la descente du gradient avec un perceptron mono-couche.

La matrice contenant les sorties souhaitées doit être transposée afin de pouvoir récupérer plus facilement toutes les réponses pour un même neurone. Par exemple, si le fichier contient 150 exemples et 3 populations différentes, alors la matrice des réponses est de dimensions 150×3. Il est alors difficile d'obtenir toutes les réponses pour un même neurone. Tandis que la transposée de cette matrice est une 3×150, où chaque ligne accédée contient l'ensemble des réponses souhaitées.

Le perceptron mono-couche, représenté sur la Figure 2.6 correspondant à cet exemple est donc composé de trois neurones car il y a trois classes à distinguer, ainsi que de deux entrées par neurone et donc deux poids, sans compter les biais.

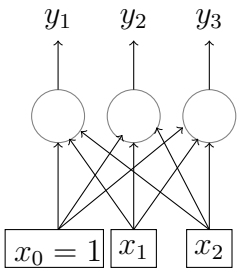


FIGURE 2.6 – Le perceptron mono-couche à 3 neurones de sortie correspondant à l'exemple du Listing 2.12.

2.4.3 Résultats et limites du perceptron mono-couche

Tous les neurones ont rapidement convergé et lors de l'évaluation post-apprentissage, aucun d'entre eux n'a commit d'erreur. La convergence du perceptron mono-couche est donc *absolue*. Sur la Figure 2.7, trois limites de décision sont visibles, séparant les trois types d'individus. Cependant la limite de classification du troisième neurone ne dissocie pas parfaitement la classe une de la trois. En revanche, cela ne l'empêche pas de tout de même bien classifier ses entrées.

```
int main()
{
    vector<vector<double>> x;
    vector<double> y;
    ptrfa=Signe;

    /* "/home/julien/workspace/NeuroLibTest/iris-3classes-adaline.csv" = Chemin du fichier contenant les données
     * x = vecteur deux dimensions qui va contenir les variables d'entrées des différents exemples
     * nbEntrees = nombre de variables d'entrées pour chacun des exemples contenus dans le fichier
     * y = matrice où chaque ligne va contenir l'étiquette correspondant à l'exemple. Il y aura autant de colonnes que de classes différentes contenues dans le jeu de données
     * 3 = Nombre de classes différentes
     */
    LireFichierCSV("/home/julien/workspace/NeuroLibTest/iris-3classes-adaline.csv", x, nbEntrees, y, 3);

    /* 2 = nombres de poids et par conséquent d'entrées
     * ptrfa = pointeur fonction d'activation, Signe dans le cas présent */
    Neurone n1=Neurone(nbEntrees,ptrfa), n2=Neurone(nbEntrees,ptrfa), n3=Neurone(nbEntrees,ptrfa);

    // transposer matrice y (150x3) en (3 (=neurones) x 150 (=exemples) )! :
    vector<vector<double>> y1(3);
    for(size_t i=0; i<y1.size(); i++)
        y1[i]=vector<double>(y.size());

    for(size_t i=0; i<y.size(); i++)
        for(size_t j=0; j<3; j++)
            y1[j][i] = y[i][j];

    /* n = le neurone qu'il faut entrainer
     * x = le vecteur deux dimensions contenant les exemples
     * y1[i] = les étiquettes du vecteur d'entrées correspondant au neurone i
     * 50 = le nombre maximum d'itérations
     * 0.01 = le taux d'apprentissage */
```

```
33      DescenteGradientWidrowHoff(n1, x, y1[0], 50, 0.01);
        DescenteGradientWidrowHoff(n2, x, y1[1], 50, 0.01);
35      DescenteGradientWidrowHoff(n3, x, y1[2], 50, 0.01);

37      /* ni = le neurone "i" dont on souhaite tester les performances
         * x = le vecteur deux dimensions contenant les exemples à évaluer
39       * y1[i] = les réponses aux entrées à évaluer afin de comparer les résultats
        concernant le neurone i */
        Evaluation(&n1, &x, &y1[0]);
41      Evaluation(&n2, &x, &y1[1]);
        Evaluation(&n3, &x, &y1[2]);
43  }

45  // Apprentissage:
    // Neurone 1
47  // Iteration 1  Poids: -0.446381  -0.185191 biais: 0.668215 Nombre d'erreurs: 0
        Erreur: 0.000171302
    // Poids finaux : -0.446381 -0.185191 biais: 0.668215
49
    // Neurone 2
51  // Iteration 2  Poids: -0.0582039 0.309829  biais: -1.01068 Nombre d'erreurs: 0
        Erreur: 0.00219012
    // Poids finaux : -0.0582039  0.309829  biais: -1.01068
53
    // Neurone 3
55  // Iteration 1  Poids: 0.50343  -0.352715 biais: 0.365333 Nombre d'erreurs: 0
        Erreur: 0.00047071
    // Poids finaux : 0.50343 -0.352715 biais: 0.365333
57
    // Evaluation:
59  // Neurone 1
    // Nombre d erreur(s): 0 sur 150 valeurs
61
    // Neurone 2
63  //Nombre d erreur(s): 0 sur 150 valeurs

65  // Neurone 3
    // Nombre d erreur(s): 0 sur 150 valeurs
```

Listing 2.13 – Test de l'algorithme ADALINE avec un perceptron mono-couche.

Une seconde exécution a été effectuée avec l'algorithme de Descente de Gradient, afin de comparer les résultats avec un ensemble de données un peu plus conséquent. Le code et la sortie du programme sont présentés dans le Listing 2.14 ainsi que la limite de décision tracée sur la Figure 2.8.

```
    #include <Neurone/neurone.h>
2   #include <Apprentissage/apprentissage.h>

4   int main()
    {
6       vector<vector<double>> x;
        vector<double> y;
8       ptrfa=Signe;

10      /* "/home/julien/workspace/NeuroLibTest/iris-3classes-adaline.csv" = Chemin
        du fichier contenant les données
         * x = vecteur deux dimensions qui contiendra les variables d'entrées des
        différents exemples
```

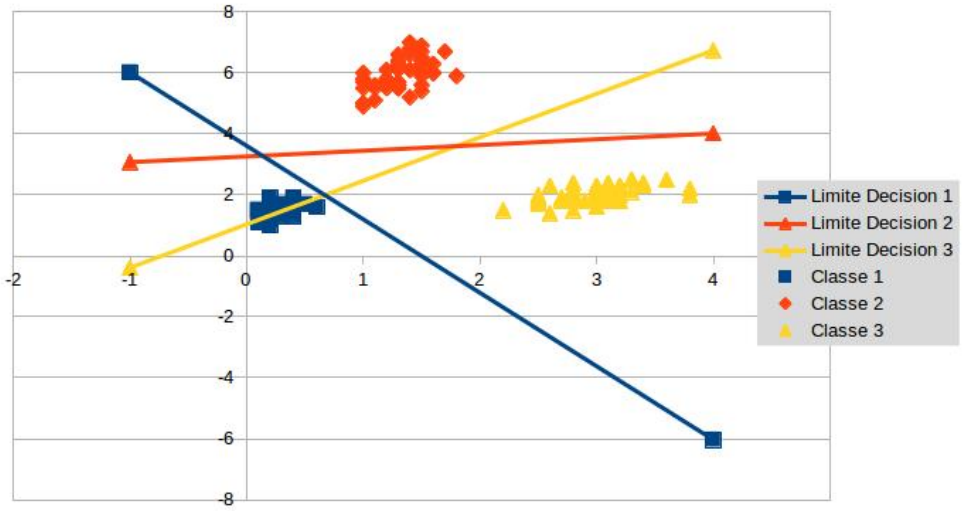


FIGURE 2.7 – Classification de 3 types d'individus différents avec l'utilisation de l'algorithme ADALINE et un perceptron mono-couche à trois neurones.

```
12      * nbEntrees = nombre de variables d'entrées pour chacun des exemples
        contenus dans le fichier
        * y = matrice où chaque ligne va contenir l'étiquette correspondant à l'
        exemple. Il y aura autant de colonnes que de classes différentes contenues
        dans le jeu de données
14      * 3 = Nombre de classes différentes
        */
16      LireFichierCSV("/home/julien/workspace/NeuroLibTest/iris-3classes-adaline.csv
        ", x, nbEntrees, y, 3);

18      /* 2 = nombres de poids et par conséquent d'entrées
        * ptrfa = pointeur fonction d'activation, Signe dans le cas présent */
20      Neurone n1=Neurone(nbEntrees,ptrfa), n2=Neurone(nbEntrees,ptrfa), n3=Neurone(
        nbEntrees,ptrfa);

22      // transposer matrice y (150x3) en (3 (=neurones) x 150 (=exemples) )! :
        vector<vector<double>> y1(3);
24      for(size_t i=0; i<y1.size(); i++)
            y1[i]=vector<double>(y.size());
26
        for(size_t i=0; i<y.size(); i++)
28          for(size_t j=0; j<3; j++)
                y1[j][i] = y[i][j];
30
        /* n = le neurone qu'il faut entrainer
32      * x = le vecteur deux dimensions contenant les exemples
        * y1[i] = les étiquettes du vecteur d'entrées correspondant au neurone i
34      * 50 = le nombre maximum d'itérations
        * 0.01 = le taux d'apprentissage */
36      DescenteGradient(n1, x, y1[0], 50, 0.0001);
        DescenteGradient(n2, x, y1[1], 50, 0.0001);
38      DescenteGradient(n3, x, y1[2], 50, 0.0001);

40      /* ni = le neurone dont on souhaite tester les performances
        * x = le vecteur deux dimensions contenant les exemples à évaluer
42      * y[i] = les réponses aux entrées à évaluer afin de comparer les résultats
        du neurone i */
        Evaluation(&n1, &x, &y1[0]);
```

```
44      Evaluation(&n2, &x, &y1[1]);
        Evaluation(&n3, &x, &y1[2]);
46  }

48  // Apprentissage:
    // Neurone 1
50  // Iteration 46 Poids: -0.252726  -0.502849 biais: 1.21018  Nombre d'erreurs: 0
        Erreur: 44.3167
    // Poids finaux (+ biais) : -0.252726 -0.502849 biais: 1.21018
52
    // Neurone 2
54  // Iteration 39 Poids: 0.0651141  0.416019  biais: -2.06145 Nombre d'erreurs: 0
        Erreur: 15.4059
    // Poids finaux (+ biais) : 0.0651141 0.416019  biais: -2.06145
56
    // Neurone 3
58  // Iteration 4  Poids: 0.416768 -0.316202 biais: 0.688686 Nombre d'erreurs: 0
        Erreur: 33.3238
    // Poids finaux (+ biais) : 0.503302  -0.354729 biais: 0.374531
60
    // Evaluation:
62  // Neurone 1
    // Nombre d'erreur(s): 0 sur 150 valeurs
64
    // Neurone 2
66  // Nombre d'erreur(s): 0 sur 150 valeurs

68  // Neurone 3
    // Nombre d'erreur(s): 0 sur 150 valeurs
```

Listing 2.14 – Test de la descente du gradient avec un perceptron mono-couche.

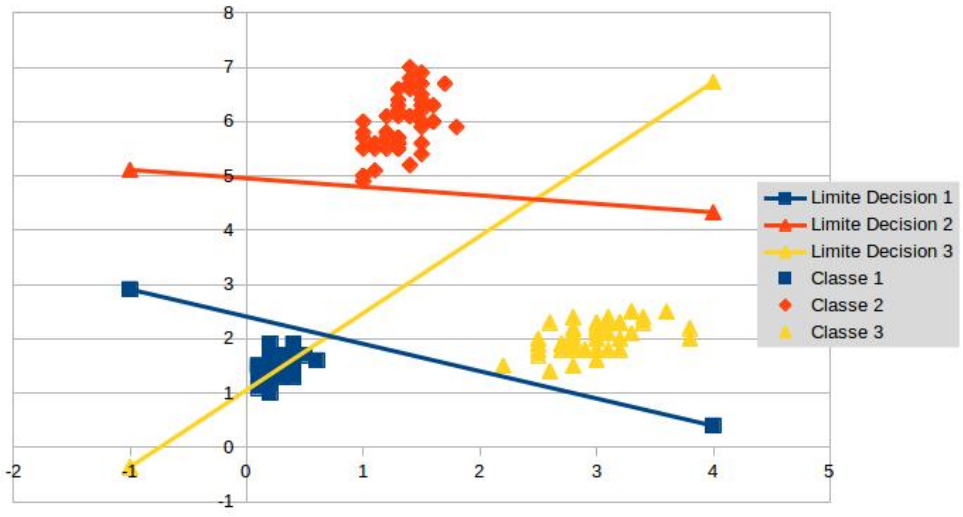


FIGURE 2.8 – Classification de 3 types d'individus différents avec l'utilisation de l'algorithme de Descente de Gradient et un perceptron mono-couche à trois neurones.

En premier, ADALINE converge systématiquement plus rapidement que la Descente de Gradient. Aussi, l'erreur résiduelle est nettement inférieure dans le cas d'ADALINE. La dernière remarque concerne les limites de décision. Pour l'ADALINE, la limite de décision du troisième perceptron coupait déjà le jeu de données. Dans le cas de la descente du gradient, cet effet est

toujours présent. Les résultats donnés pas la Descente de Gradient ne sont donc pas meilleurs que ceux d'ADALINE.

2.5 Limites du perceptron

En 1969, les scientifiques M. Minsky & S. Papert ont défini les limites du perceptron. Le perceptron est en fait capable de classifier des données linéairement séparables, c'est-à-dire que ces données sont séparables, dans l'hyper-espace, par un hyper-plan. Si le but est de classifier des données non-linéairement séparables, les réseaux de neurones sont nécessaires [KS96a].

```
#include <Neurone/neurone.h>
#include <Apprentissage/apprentissage.h>

void main(
{
    double (*ptrfa)(double)=Signe;
    vector<double> y = {-1.0, 1.0, 1.0, -1.0};
    vector<vector<double>> x = {{-1.0, -1.0}, {-1.0, 1.0}, {1.0, -1.0}, {1.0,
    1.0}}

    Neurone n(2, ptrfa), n1(2, ptrfa);

    /* n = le neurone qu'il faut entrainer
     * x = le vecteur deux dimensions contenant les exemples
     * y1[i] = les étiquettes du vecteur d'entrées correspondant au neurone i
     * 50 = le nombre maximum d'itérations
     * Le taux d'apprentissage par défaut est utilisé (=0.01) */
    DescenteGradientWidrowHoff(n, x, y, 50);

    /* n = le neurone dont on souhaite tester les performances
     * x = le vecteur deux dimensions contenant les exemples à évaluer
     * y = les réponses aux entrées à évaluer afin de comparer les résultats */
    Evaluation(&n, &x, &y);
}

// Iteration 49 Poids: -0.0942069 -0.0738512  biais: 0.131909 Nombre d erreurs:
    1   Erreur: 0.000212917
// Poids finaux : -0.0942069  -0.0738512  biais: 0.131909
// Sortie obtenue: 1 Sortie attendue: -1
// Sortie obtenue: 1 Sortie attendue: 1
// Sortie obtenue: 1 Sortie attendue: 1
// Sortie obtenue: -1 Sortie attendue: -1
// Nombre d erreur(s): 1 sur 4 valeurs
```

Listing 2.15 – Test de la descente du gradient avec le XOR.

Un exemple classique, présent dans le Listing 2.15, est la fonction XOR qui retourne 0 si les bits d'entrées sont tous égaux et 1 sinon. Tout comme la fonction OR, la fonction XOR est également souvent citée dans les références telles que [KS96a] afin de désigner l'incapacité des perceptrons à séparer des classes non-séparables linéairement. Même si l'erreur générale est très faible, le perceptron se trompe tout de même dans la classification de deux exemples. La limite de décision de ce perceptron est visible sur la Figure 2.9. Ce résultat était déjà visible sur la Figure 2.3b, où l'ensemble des données n'était pas linéairement séparable.

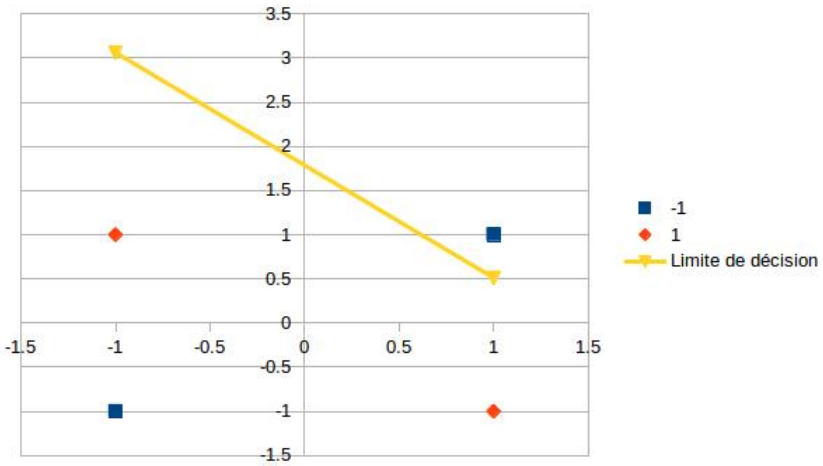


FIGURE 2.9 – Tentative de classification de la fonction XOR avec un perceptron entraîné avec l'algorithme ADALINE.

Chapitre 3

Les réseaux de neurones multi-couches

L'intérêt des neurones réside dans les propriétés découlant de l'association en réseaux : ce dernier peut approximer une fonction non-linéaire, ce qui constituait la limite des réseaux mono-couche. Un réseau de neurones possède alors au moins deux couches. Une première couche de neurones, dite *cachée*, appliquant une fonction non-linéaire à partir des différentes entrées. L'adjectif "caché" des neurones est dû à leur sortie invisible depuis l'extérieur du réseau. La seconde couche est celle des neurones de *sorties* donnant le résultat de l'analyse des entrées par le réseau [Gé07]. Un exemple d'un tel réseau est représenté sur la Figure 3.1.

Il existe beaucoup de réseaux de neurones différents mais l'étude dans ce document se limite aux perceptrons multi-couches. Tout d'abord, les réseaux feed-forward doivent être distingués des récurrents, dont les schémas se trouvent sur la Figure 3.2. Un réseau est qualifié de *feed-forward* ou *non-bouclé* ou encore *statique* si aucune sortie d'un neurone n'est redirigée vers un autre neurone de la même couche où d'une couche antérieure. Contrairement aux réseaux feed-forward, un réseau *récurrent* ou *bouclé* ou même *dynamique* possède cette caractéristique. En outre, l'adjectif *fully-connected* ou *complètement connecté* en français désigne les réseaux de neurones dont chaque unité (neurone ou entrée) sert d'entrée à la couche de neurones suivante. L'exemple du réseau de neurones présent sur la Figure 3.1 est *fully-connected*. Il est, en effet, possible d'imaginer des réseaux de neurones dont certains neurones ne fourniraient pas leur sortie comme entrée de certains neurones [Gé07].

Par ailleurs, il est également possible d'avoir plusieurs couches cachées. Par exemple, sur la Figure 3.1, le réseau de neurones est feed-forward et possède une seule couche cachée. Le nombre

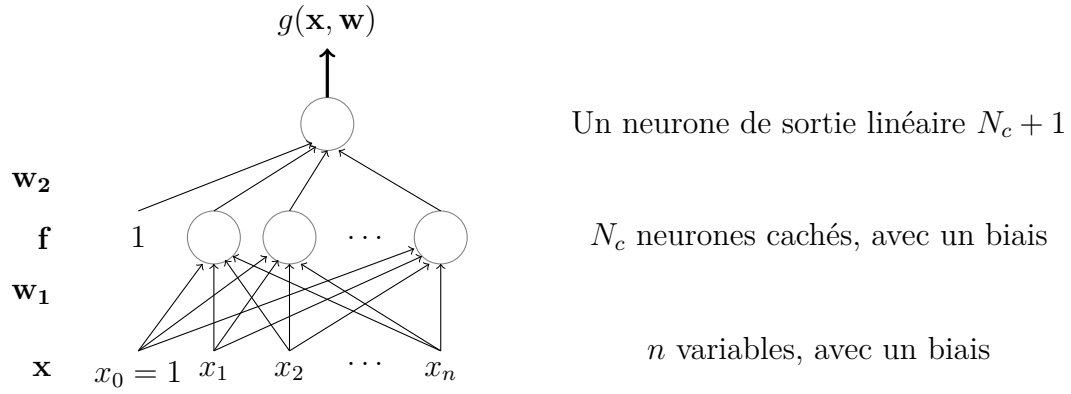


FIGURE 3.1 – Un réseau de neurones feed-forward à une couche cachée et à un seul neurone de sortie selon [Gé07].

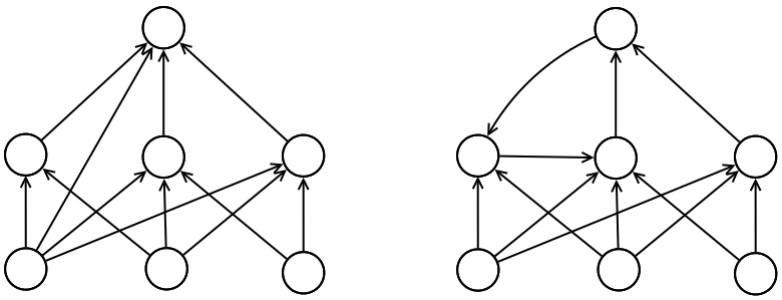


FIGURE 3.2 – À gauche un réseau de neurones feed-forward. À droite, un réseau de neurones bouclé.

de couches d'un réseau désigne sa *profondeur*, plus un réseau possède de couches cachées, plus il sera "*deep*" (profond).

3.1 Les perceptrons multi-couches

3.1.1 Théorie des perceptrons multi-couches

Les Perceptrons Multi-Couches (PMC ou MLP en anglais, pour Multi-Layer Perceptron) sont des réseaux de neurones feed-forward et complètement connectés utilisant des perceptrons, c'est-à-dire pas de Hopfield, Boltzmann, etc. Cependant ces perceptrons ne sont pas tels que ceux de ROSENBLATT , ceux-ci possèdent une fonction d'activation continue [Gos96] et dérivable : le terme "perceptron" est donc plutôt mal choisi.

La Figure 3.3 présente les fonctions d'activation le plus souvent utilisées pour les PMC. Celles-ci sont toutes d'allures sigmoïdes (en "S") et possèdent une sortie bornée dans un certain intervalle. De plus, elles sont également strictement croissantes, elles ne possèdent donc pas de dérivée qui s'annule. Cela aurait comme effet néfaste de bloquer l'apprentissage par un gradient nul. Enfin, si un bruit est introduit dans une composante de l'entrée, le réseau sera également peu perturbé grâce au comportement asymptotique des fonctions. Les bornes de ces fonctions représenteront les deux classes auxquelles les entrées peuvent appartenir [Gé07].

Voici un tableau contenant les fonctions d'activation et leur dérivée les plus efficaces avec les réseaux de neurones selon [Gé07] :

Fonction	Formule	Dérivée	Bornes
Logistique (ou Sigmoïde)	$f(x) = 1/(1+e^{-x})$	$f'(x) = f(x)(1-f(x))$	$]\ 0,\ 1\ [$
Tangente Hyperbolique (tanh)	$f(x) = tanh(x)$	$f'(x) = 1-(f(x))^2$	$]\ -1,\ 1\ [$

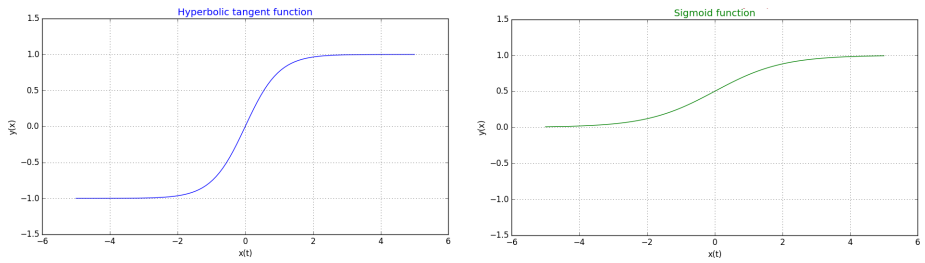


FIGURE 3.3 – Les fonctions d'activations les plus populaires pour les PMC.

3.1.2 Implémentation

Tout d'abord, un PMC n'est qu'une accumulation en couche de neurones, les paramètres de la classe **ReseauNeurone** sont alors le nombre d'entrées, le nombre de couches cachées ainsi que la quantité de neurones par couche cachées et enfin le nombre de neurones de sortie. Toutes les couches de neurones auront donc le même nombre de neurones cachés. Cela peut sembler restrictif à première vue, mais si un utilisateur souhaite ne pas avoir un réseau complètement connecté, il lui suffit de mettre le poids de la connexion indésirable à 0. De même, afin de supprimer un certain neurone du réseau, les poids de ses connexions doivent être mis à 0. Aussi, pour chacun des neurones, une fonction d'activation ainsi que sa dérivée doivent être définis. Par défaut, les fonctions **Logistique** et **LogistiqueDerivee** seront attribuées.

```
class ReseauNeurone
{
    public:
        ReseauNeurone(size_t nbPoidsEntree, double (*fonctionActivation)(double) =
            Logistique, double (*fonctionActivationDerivee)(double) =
            LogistiqueDerivee, size_t nbNeuroneParCouche=1,
            size_t nbCouchesCachees=1, size_t nbNeuroneSorties=1);

        ReseauNeurone(double (*fonctionActivation)(double),
            vector<vector<vector<double>>> &poidsCouches);

        ~ReseauNeurone();

        vector<double>& Evaluer(vector<double> &entrees);

        size_t getIndiceCoucheSortie();
        vector<Neurone>& getCoucheSortie();

        vector<vector<Neurone>> couches;
        vector<vector<double>>sortiesReseau;

    private:
        ReseauNeurone();
};
```

Listing 3.1 – Définition de la classe ReseauNeurone dans Neurone/reseauneurone.cpp.

La variable membre **couches** est un tableau en deux dimensions de **Neurone**, dont une ligne contiendra une couche de neurones cachés et aura autant de lignes que de couches cachées. La dernière couche de neurones correspondra à la couche de sortie.

La variable membre **sortiesReseau** contient l'ensemble des sorties des neurones des différentes couches. De cette façon, les sorties d'une couche de neurones peuvent être fournies sous forme de vecteur d'entrée à la couche suivante. Ce tableau à deux dimensions est aussi large qu'il y a de neurones dans une couche et aussi long qu'il y a de couches dans le réseau.

Cette fois, seulement deux constructeurs sont disponibles, le constructeur par défaut `ReseauNeurone();` étant *privé* car il ne spécifie aucun paramètre du réseau. Le constructeur `ReseauNeurone(size_t nbPoidsEntree, double (*fonctionActivation)(double) = Logistique, double (*fonctionActivationDerivee)(double) = LogistiqueDerivee, size_t nbNeuroneParCouche=1, size_t nbCouchesCachees=1, size_t nbNeuroneSorties=1);` est le plus propice à être utilisé dans la vie réelle. Celui-ci crée un réseau de neurones avec un certain nombre de neurones, de couches,

d'entrées et de sorties et initialise les poids de façon aléatoires suivant une loi gaussienne en utilisant le constructeur de la classe **Neurone** requiérant la quantité de poids désirée.

Le constructeur `ReseauNeurone(double (*fonctionActivation)(double), vector< vector< vector< double > > > &poidsCouches);` est analogue au constructeur de la classe **Neurone** et permet de spécifier les poids des unités du réseau ainsi que sa fonction d'activation et la dérivée. Ce constructeur sert donc à reproduire un comportement bien spécifique. Par exemple, un comportement que le développeur souhaite débugger.

```
ReseauNeurone::ReseauNeurone() { }

ReseauNeurone::ReseauNeurone(size_t nbPoidsEntree, double (*fonctionActivation)(
    double), double (*fonctionActivationDerivee)(double), size_t
    nbNeuroneParCouche, size_t nbCouchesCachees, size_t nbNeuroneSorties)
:ReseauNeurone()
{
    //Créer couche d'entrée
    vector<Neurone> neuronesTmp;
    for(size_t i=0; i<nbNeuroneParCouche; i++)
        neuronesTmp.push_back(Neurone(nbPoidsEntree, fonctionActivation,
    fonctionActivationDerivee));

    if(neuronesTmp.size()>0) // Si réseau d'un seul neurone
        couches.push_back(neuronesTmp);

    // Créer toutes les couches cachées
    for(int i=0; i<((int)nbCouchesCachees)-1; i++) // Conversion en int
    obligatoire: si nbCouchesCachees vaut 0, alors nbCouchesCachees-1 vaut -1 ->
    pas dans size_t
    {
        // Créer tous les neurones pour chaque couche cachée
        vector<Neurone> neuronesTmp;
        for(size_t j=0; j<nbNeuroneParCouche; j++)
            neuronesTmp.push_back(Neurone(nbNeuroneParCouche, fonctionActivation,
                fonctionActivationDerivee));

        couches.push_back(neuronesTmp);
    }

    // Créer les neurones de sorties
    vector<Neurone> neuronesSorties(0);
    for(size_t i=0; i<nbNeuroneSorties; i++)
        //Entrées neurones de sortie = neurones cachés donc autant d'entrées que
    neurones cachés
        neuronesSorties.push_back(Neurone(nbNeuroneParCouche, fonctionActivation,
            fonctionActivationDerivee));

    if(nbNeuroneSorties>0) // Si réseau d'un seul neurone
        couches.push_back(neuronesSorties);

    //Créer les tableaux de sorties des couches cachées
    size_t nbCouches=nbCouchesCachees+1;
    sortiesReseau=vector<vector<double>>(nbCouches);//nb de couches cachees + 1
    seule couche de sortie
    for(size_t i=0; i<nbCouches; i++)
        sortiesReseau[i]=vector<double>(nbNeuroneParCouche);

    if(nbNeuroneSorties>0) // Si réseau d'un seul neurone
        sortiesReseau[nbCouches-1]=vector<double>(nbNeuroneSorties);
```

```
44 }

46 ReseauNeurone::ReseauNeurone(double (*fonctionActivation)(double), vector<vector
       <vector<double>>> &poidsCouches)
   {
48     //poidsCouches[i][j][k]: kème poids du jème neurone de la couche i
       size_t nbCouches=poidsCouches.size();
50
       // Créer les couches du réseau
52     size_t i;
       for(i=0; i<nbCouches; i++)
54     {
           // Créer tous les neurones pour chaque couche cachée
56         vector<Neurone> neuronesTmp;
           size_t nbNeuronesCouche=poidsCouches[i].size(); // Un vecteur<double> par
       neurone et autant de vecteur que de neurone
58         for(size_t j=0; j<nbNeuronesCouche; j++)
               neuronesTmp.push_back(Neurone(poidsCouches[i][j], fonctionActivation));
60
           couches.push_back(neuronesTmp);
62     }

64     //Créer les tableaux de sorties des couches cachées
       sortiesReseau=vector<vector<double>>(nbCouches);//nb de couches cachees + 1
       seule couche de sortie
66     for(size_t i=0; i<nbCouches; i++)
           sortiesReseau[i]=vector<double>(poidsCouches[i].size());
68
       sortiesReseau[nbCouches-1]=vector<double>(poidsCouches[nbCouches-1].size());
70 }
```

Listing 3.2 – Définition des constructeurs de ReseauNeurone dans Neurone/reseauneurone.cpp.

La propagation des entrées dans le réseau de neurones se fait grâce à la fonction **Evaluer** homonyme à celle de la classe **Neurone**. Chaque évaluation de neurone remplira avec la sortie de ce neurone, la case correspondante dans le tableau **sortieReseau** de la classe **ReseauNeurone**.

```
   vector<double>& ReseauNeurone::Evaluer(vector<double> &entrees)
2  {
       // couche d'entrée
4      for(size_t j=0; j<couches[0].size(); j++)
           sortiesReseau[0][j]=couches[0][j].Evaluer(entrees);
6
       // couches cachées
8      for(size_t i=1; i<couches.size()-1; i++)
           for(size_t j=0; j<couches[i].size(); j++)
10             sortiesReseau[i][j]= couches[i][j].Evaluer(sortiesReseau[i-1]);

12     return sortiesReseau[getIndiceCoucheSortie()];
   }
```

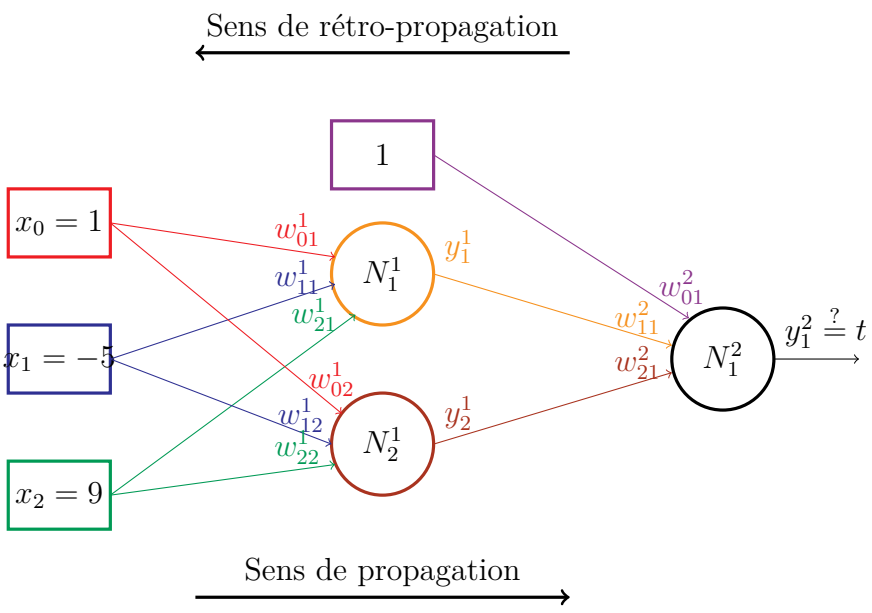
Listing 3.3 – Evaluation des couches d'un réseau de neurones.

Voici un exemple de création et d'utilisation de réseau de neurones :

```
1  #include <Neurone/reseauneurone.h>

3  void main()
   {
```

```
5     size_t nbEntrees=2; // Nombre variables d'entrées
      vector<vector<double>> x, y;
7
      /* Charger les exemples dans x et les labels correspondants dans y :
9      * "/home/julien/workspace/NeuroLibTest/moon_class0.csv" = Chemin du fichier
      contenant les données
       * x = matrice contenant les variables d'entrées des différents exemples
11     * nbEntree = nombre de variables d'entrées pour chacun des exemples contenus
       dans le fichier, 2 dans ce cas
       * y = matrice où chaque ligne va contenir l'étiquette correspondant à l'
      exemple. Il y aura autant de colonnes que de classes différentes contenues
      dans le jeu de données
13     * 1 = Nombre de sorties contenues dans le fichier */
      LireFichierCSV("/home/julien/workspace/NeuroLibTest/moon_class0.csv",
15       x, nbEntrees, y, 1);


17
      /* Création d'un RDN à 5 neurones/couches cachées, 2 couches et 1 neurone de
      sortie :
19     * nbEntrees = nombre de poids et d'entrées pour la couche d'entrée du RDN
       * Logistique = fonction d'activation des neurones du réseau
21     * LogistiqueDerivee = dérivée de la fonction d'activation des neurones du ré
      seau
       * 5 = nombre de neurones dans les couches cachées
23     * 2 = nombre de couches cachées
       * 1 = nombre de neurones de sortie */
25    ReseauNeurone rdn(nbEntrees, Logistique, LogistiqueDerivee, 5, 2, 1);

27    /* Autre fonction d'activation :
       * ReseauNeurone rdn(nbEntrees, TangenteHyperbolique,
29           TangenteHyperboliqueDerivee, 5, 2, 1); */

31    size_t nbErreurs=0;
      for(size_t i=0; i<x.size(); i++)
33    {
         vector<double> exemple=x[i];
35       // exemple : un exemple parmi ceux disponibles qui sera analysé
         rdn.Evaluer(exemple);
37       //Le réseau ne possède qu'un seul neurone de sortie, dont l'indice
      commence à 0
         double sortie=rdn.sortiesReseau[rdn.getIndiceCoucheSortie()][0];
39
         //Equivalent:
41       //double sortie=rdn.Evaluer(exemple)[0];

43    /* Seuiller la sortie et vérifier que la réponse soit correcte
       * Remarque:
45     *  - Si fonction tangente hyperbolique, sortie = -1 ou 1.
       *  - Si fonction logistique, sortie = 0 ou 1 */
47       double output=0.0f;
         if(sortie>0.5)
49          output=1.0f;

51       if(output!=y1[i][0])
            nbErreurs++;
53    }
      cout << "Nombre d'erreurs (seuil =" << seuil << "): " << nbErreurs << endl;
55 }
```

Listing 3.4 – Exemple : analyse des entrées chargée d'un fichier et compte le nombre d'erreur de classification.



Notation :

- N_i^L : Désigne le $i$ème neurone de la couche L.
- w_{ji}^L : Désigne le poids j du $i$ème neurone de la couche L.
- y_i^L : Désigne la sortie du $i$ème neurone de la couche L.
- x_i : Désigne la $i$ème entrée du réseau de neurones.
- t : La réponse attendue pour cet exemple.

Les poids sont au préalablement initialisés à des valeurs aléatoires :

Neurone (N_i^L)	Poids (w_{0i}^L) 0	Poids 1 (w_{1i}^L)	Poids 2(w_{2i}^L)
N_1^1 (i=1, L=1)	1	2	4
N_2^1 (i=2, L=1)	1	-3	8
N_1^2 (i=1, L=2)	1	6	-8

FIGURE 3.4 – Exemple d'un réseau de neurone sur lequel la rétro-propagation du gradient va être appliqué.

3.2 La Rétro-propagation du Gradient de l'Erreur

3.2.1 Théorie de la Rétro-propagation du Gradient de l'Erreur

Cet algorithme créé par WERBOS en 1974 permet l'apprentissage des réseaux de neurones. La méthode se base toujours sur le Gradient de l'Erreur précédemment décrite. Plus précisément, l'erreur est calculée *sur la couche de sortie* et se verra propagée dans le sens inverse de la propagation du réseau de neurones, c'est-à-dire de la couche de sortie vers la couche d'entrée, afin d'appliquer une correction aux poids.

Considérons le réseau présenté sur la Figure 3.4, ne possédant qu'un seul neurone de sortie ainsi que d'une seule couche cachée composée de deux neurones. Ce réseau comporte deux entrées et un biais. Dans le cadre de l'apprentissage, lorsqu'un exemple est présenté au réseau, l'erreur se mesure sur la couche de sortie avec la fonction quadratique [KS96b] :

$$E(\mathbf{w}, \mathbf{x}) = \frac{1}{2}(t - y)^2$$

Où :
— y : la sortie obtenue par le réseau de neurones pour l'exemple choisi ;
— t : la sortie attendue pour l'exemple choisi ;
— E : désigne l'erreur totale de la couche de sortie.

L'entropie croisée, parfois appelée entropie relative [Gos96], est une fonction alternative à l'erreur quadratique. Elle s'utilise avec une fonction d'activation produisant des *probabilités* telles que la fonction **Logistique**. Une seconde fonction d'activation produisant des probabilités est la fonction Softmax. Elle est particulière car elle ne dépend pas du potentiel d'un neurone mais des potentiels des neurones *appartenant à une même couche*. Ces deux fonctions sont présentées dans l'Annexe D.

Plus généralement, l'erreur de la couche de sortie d'un réseau de neurones sur un exemple k ayant N_s neurones de sorties se calcule en sommant l'erreur $E_{ki}x(\mathbf{w}, \mathbf{x})$ d'un neurone i de sortie :

$$E_k(\mathbf{w}, \mathbf{x}) = \sum_{i=1}^{N_s} E_{ki}(\mathbf{w}, \mathbf{x}) = \frac{1}{2} \sum_{i=1}^{N_s} (t_{ki} - y_{ki}(\mathbf{w}, \mathbf{x}))^2$$

Où :
— y_{ki} : la sortie obtenue pour le neurone i pour l'exemple k ;
— t_{ki} : la sortie attendue pour le neurone i pour l'exemple k ;
— K : le nombre d'exemples dans l'ensemble des données d'apprentissage ;
— E_{ki} : correspond à l'erreur du neurone i pour l'exemple k,
— E_k : désigne l'erreur totale de la couche de sortie pour l'exemple k.

Et donc l'erreur totale, pour un réseau quelconque est, pour un ensemble de K exemples :

$$E(\mathbf{w}, \mathbf{x}) = \sum_{k=1}^{K} \sum_{i=1}^{N_s} (t_{ki} - y_{ki}(\mathbf{w}, \mathbf{x}_k))^2 \tag{3.1}$$

Quant à l'erreur moyenne $\overline{E}(\mathbf{w}, \mathbf{x})$, elle vaut :

$$\overline{E}(\mathbf{w}, \mathbf{x}) = \frac{1}{2} \frac{1}{K} E(\mathbf{w}, \mathbf{x}) \tag{3.2}$$

En premier lieu, vient l'étape de la *Propagation en avant*. Un seul exemple sera considéré afin de simplifier l'explication. Pour chaque neurone j d'une couche L, en commençant par la couche la plus proche du vecteur d'entrée, il faut calculer successivement le potentiel puis la sortie des neurones de cette couche. Cette étape doit être répétée pour les couches suivantes jusqu'à la couche de sortie. Cependant, les entrées des couches ultérieures ne seront plus celles fournies par les exemples, mais les sorties des neurones de la couche précédente. Le potentiel d'un neurone i quelconque de la couche L se calcule selon [Bis06] :

$$\nu_i^L(\mathbf{w}, \mathbf{x}) = \sum_{p=0}^{P} w_{pi}^L \; x_{pi}^L \tag{3.3}$$

La sortie de ce neurone i de la couche L, où $\phi_i^L(\cdot)$ est la fonction d'activation du neurone [Bis06] :

$$y_i^L(\mathbf{w}, \mathbf{x})) = \phi_i^L(\mathbf{w}, \mathbf{x})) \tag{3.4}$$

Le but de l'algorithme est de minimiser la fonction d'erreur $E(\mathbf{w}, \mathbf{x}) = \frac{1}{2}(y(\mathbf{w}, \mathbf{x}) - t)^2$ du neurone de sortie pour un exemple quelconque. Le théorème de la dérivation des fonctions composées dit que si une variable a dépend d'une variable b, elle-même dépendante d'une variable c, de telle façon que a dépend de c par l'intermédiaire de b, alors la dérivée d'une fonction composée s'écrit :

$$\frac{\partial a}{\partial c} = \frac{\partial a}{\partial b}\frac{\partial b}{\partial c}$$

Calcul du gradient de l'erreur pour la couche de sortie

Dans le cas de la fonction d'erreur $E(\mathbf{w}, \mathbf{x})$, E est dépendante des poids $\mathbf{w}$ par l'intermédiaire du potentiel $\nu(\mathbf{w}, \mathbf{x})$. Le réseau de neurones considéré possède une sortie et une seule couche cachée composée de deux neurones. Le gradient de l'erreur s'énonce pour **la couche de sortie**, indexée par l'indice S :

$$\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pi}} = \frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}\frac{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pi}} \text{ , où } p \text{ désigne l'entrée et le poids concernés.}$$

Où la quantité $\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pi}}$ désigne le gradient de la fonction d'erreur $E(\mathbf{w}, \mathbf{x})$ pour le neurone i de la couche de sortie S, dépendante des poids p de ce neurone. Cette quantité permet à la correction appliquée aux poids p du neurone i de descendre la fonction d'erreur afin d'arriver à un minimum local [KKN10a]. Le premier terme de cette expression vaut :

$$\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} = -\frac{2}{2}(t_i - y_i^S(\mathbf{w}, \mathbf{x}))(y_i^S(\mathbf{w}, \mathbf{x}))'$$

Étant donné que le but de l'algorithme est de retirer la quantité de la descente la plus raide aux poids, on peut insérer un "-" devant. De plus, si on injecte (3.4), on obtient :

$$\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S} = -(y_i^S(\mathbf{w}, \mathbf{x}) - t_i)\ (\phi_i^S(\nu_i^S))'$$
$$\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S} = (t_i - y_i^S(\mathbf{w}, \mathbf{x}))\ (\phi_i^S(\nu_i^S))'$$

Cette erreur est appelée *erreur locale* car propre au neurone i. Elle se note généralement δ_i.

$$\delta_i^S = (t_i - y_i^S(\mathbf{w}, \mathbf{x})(\phi_i^S(\nu_i^S))' \tag{3.5}$$

En ce qui concerne le second terme :

$$\frac{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pi}^S} = \frac{\partial}{\partial w_{pi}^S}(\sum_{h=0}^{P} w_{hi}^S x_{hi}^S)')$$

Où h parcourt les P poids du neurone de sortie i, parmi lesquels se trouve w_{pi}^S. Or, tous les termes où $h \neq p$ s'annulent car ils sont considérés comme constants. L'erreur est évaluée pour une entrée spécifique p dans le neurone. Seul le terme où $p = h$ subsiste :

$$\frac{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pi}^S} = x_{pi}^S$$

Si les deux termes sont regroupés selon leur nouvelle formulation, l'expression du gradient de l'erreur, pour un neurone i de la couche de sortie S fonction d'un certain poids p prend une nouvelle forme. Le gradient de l'erreur doit être calculé pour chacun des poids du neurone de sortie grâce à la formule [Bis06] [KS96b] :

$$\frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial w_{pi}^S} = \delta_i^S \; x_{pi}^S \tag{3.6}$$

Si la couche de sortie comporte plusieurs neurones, alors le gradient de l'erreur doit être calculé pour chacun des poids de chacun de ces neurones. Aussi, pour des raisons de lisibilité, le gradient de l'erreur est aussi noté, pour le poids p du neurone i appartenant à une couche L :

$$\frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial w_{pi}^S} = \Delta w_{pi}^L$$

Calcul du gradient de l'erreur pour une couche cachée

Grâce à ces formules, l'évaluation de l'erreur locale pour un poids p du neurone j de la couche cachée H précédant la couche de sortie $S = H+1$, en fonction de sa contribution au neurone de sortie i est aisée. L'erreur totale mesurée par rapport aux variables fournies par un neurone j de la couche L est dépendante du potentiel du neurone de sortie i, lui-même fonction de la sortie du neurone caché j dont on souhaite justement mesurer l'erreur. La contribution à l'erreur du neurone caché j à un neurone de sortie i est calculable en appliquant à nouveau le théorème de la dérivation d'une fonction composée [KS96b] :

$$\frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial y_j^H(\mathbf{w},\mathbf{x})} = \frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial \nu_i^S(\mathbf{w},\mathbf{x})} \frac{\partial \nu_i^S(\mathbf{w},\mathbf{x})}{\partial y_j^H(\mathbf{w},\mathbf{x})}$$

Dont le premier terme a déjà été évalué et est égal à :

$$\frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial \nu_i^S(\mathbf{w},\mathbf{x})} = \delta_i^S$$

Ensuite, le potentiel du neurone de sortie i se calcule selon (3.3). Or les entrées considérées dans le potentiel sont les sorties des neurones cachés de la couche précédente. Sa dérivée par rapport à la sortie d'un neurone caché j s'annule pour tout $p \neq j$:

$$\frac{\partial \nu_i^S(\mathbf{w},\mathbf{x})}{\partial y_j^H(\mathbf{w},\mathbf{x})} = w_{ji}^H$$

Où j désigne le poids pondérant la sortie du neurone j de la couche cachée H. C'est également la $j$ème entrée du neurone de sortie i. De plus, la sortie d'un neurone caché j est dépendante du potentiel du neurone caché j, lui-même fonction des poids p du neurone j. Le gradient de l'erreur prend donc une forme semblable à celle du gradient de l'erreur pour une unité de sortie :

$$\begin{aligned}
\frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial w_{pj}^H} &= \frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial \nu_i^S(\mathbf{w},\mathbf{x})} \frac{\partial \nu_i^S(\mathbf{w},\mathbf{x})}{\partial y_j^H(\mathbf{w},\mathbf{x})} \frac{\partial y_j^H(\mathbf{w},\mathbf{x})}{\partial w_{pj}^H} \\
&= \delta_i^S \; w_{ji}^S \frac{\partial y_j^H(\mathbf{w},\mathbf{x})}{\nu_j^H(\mathbf{w},\mathbf{x})} \frac{\nu_j^H(\mathbf{w},\mathbf{x})}{\partial w_{pj}^H} \\
&= \delta_i^S \; w_{ji} \; (\phi_j^H(\nu_j^H))' \; x_{pj}^H
\end{aligned}$$

Avec $\phi_j^H(\cdot)$ désigne la fonction d'activation du neurone j de la couche H. L'erreur locale pour un neurone caché j, contribuant à un neurone de sortie i est notée δ_{ji} se calcule comme suit :

$$\delta_{ji}^H = \delta_i^S \; w_{ji}^H \; (\phi_j^H(\nu_j^H))' \tag{3.7}$$

Et l'erreur spécifique à un poids p du neurone j de la couche cachée H pour la contribution à un neurone de sortie i vaut :

$$\frac{\partial E_i^S}{\partial w_{pj}^H} = \delta_{ji}^H \; x_{pj}^H = \Delta w_{pj}^H \text{ [Bis06] [KS96b]}$$

Les formules, ci-dessus, calculent le gradient de l'erreur pour un neurone caché en fonction de sa contribution à l'erreur d'un neurone de sortie. En pratique, un neurone caché contribue à plusieurs neurones de sortie. La fonction d'erreur de la couche étant une somme, la dérivée d'une somme devient comme la somme des dérivées. Il suffit donc d'effectuer la somme sur les N_s neurones de sorties afin d'obtenir la contribution totale d'un neurone caché à l'erreur du réseau. Ainsi, le gradient d'un neurone caché d'une couche cachée H d'un réseau à N_s neurones de sortie vaut :

$$\begin{aligned}
\Delta w_{pj}^H &= \sum_{i=1}^{N_s} \delta_i^S \; x_{pj}^H \\
&= \sum_{i=1}^{N_s} \delta_i^S \; w_{ji}^S \; (\phi_j^H(\nu_j))' \; x_{pj}^H \\
\Delta w_{pj}^H &= (\phi_j^H(\nu_j^H))' \;\; x_{pj}^H \; \sum_{i=1}^{N_s} \delta_i^S \; w_{ij}^H
\end{aligned}$$

Pour un réseau de neurones avec plusieurs couches cachées, la formule *générale* du gradient de l'erreur pour une *couche cachée* H est :

$$\Delta w_{pj}^H = \sum_{i=1}^{N} \frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} \frac{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}{\partial y_j^H(\mathbf{w}, \mathbf{x})} \frac{\partial y_j^H(\mathbf{w}, \mathbf{x})}{\partial w_{pj}^H} \tag{3.8}$$

$$\leftrightarrow \; \Delta w_{pj}^H = (\phi_j^H(\nu_j^H))' \;\; x_{pj}^H \; \sum_{i=1}^{N_s} \delta_i^{H+1} \; w_{ij}^H \text{ [Bis06] [KS96b]} \tag{3.9}$$

Cette relation signifie que la valeur de δ_j pour un neurone caché j en particulier est obtenue en *propageant* les erreurs de la sortie vers l'entrée du réseau : de $H+1$ vers H. Par ailleurs, la relation (3.9) est valable pour n'importe quelle couche cachée de neurones et la couche de sortie est remplacée par la couche suivante. De plus, l'erreur d'un neurone caché est fonction de l'erreur des neurones auxquels il fournit une entrée (sa sortie) et du poids correspondant à cette connexion.

Enfin, les gradients de l'erreur sont appliqués aux poids d'un neurone j *quelconque* d'une couche L *quelconque* comme terme correctif, à nouveau pondéré par un taux d'apprentissage. Les poids au temps $t+1$ sont donné par la formule η [Bis06] [KS96b] :

$$w_{pj}^L(t+1) = w_{pj}^L(t) - \eta \Delta w_{pj}^L \tag{3.10}$$

Application de la rétro-propagation à un exemple

Voici un récapitulatif des étapes de la rétro-propagation appliquées à l'exemple mentionné sur la Figure 3.4 :

0. Les poids sont au préalablement initialisé à des valeurs aléatoires :

Neurone (N_i^L)	Poids (w_{0i}^L) 0	Poids 1 (w_{1i}^L)	Poids 2(w_{2i}^L)
N_1^1 (i=1, L=1)	1	2	4
N_2^1 (i=2, L=1)	1	-3	8
N_1^2 (i=1, L=2)	1	6	-8

1. La première étape est celle de la propagation en avant des neurones de la couche d'entrée jusqu'à la couche de sortie. Dans le cas de l'exemple décrit sur la Figure 3.4 et en considérant que les neurones de cet exemple utilisent la fonction d'activation **Logistique**les sorties sont, pour l'exemple considéré :

   Pour le premier neurone de la couche cachée N_1^1 :

$$\begin{aligned} \nu_1^1 &= 1 \times 1 \ + \ (-5 \times 2) \ + \ 9 \times 4 = 27 \\ y_1^1 &= 1/(1 + e^{-27}) = -0.013813374 \end{aligned}$$

   Pour le second neurone de la couche cachée N_2^1 :

$$\begin{aligned} \nu_2^1 &= 1 \times 1 \ + \ (-5) \times (-3) \ + \ 9 \times 8 = 89 \\ y_1^2 &= 1/(1 + e^{-89}) = 1 \end{aligned}$$

   Pour le premier et unique neurone de la couche de sortie N_1^2 :

$$\begin{aligned} \nu_1^2 &= 1 \times 1 \ - \ 6 \times 0.013813374 - 8 \times 1 = -7.08288025 \\ y_1^2 &= 1/(1 + e^{-7.08288025}) = 0.285545128 \end{aligned}$$

   En résumé, les sorties des neurones sont :

Neurone (N_i^L)	Potentiel (ν_i^L)	Sorties (y_i^L)
N_1^1 (i=1, L=1)	27	-0.013813374
N_2^1 (i=2, L=1)	89	1
N_1^2 (i=1, L=2)	-7.08288025	0.285545128

2. Ensuite, l'erreur locale du neurone de sortie doit être calculée selon l'équation (3.5) afin d'être rétro-propagée. Pour ce faire, il est tout d'abord impératif de calculer l'erreur globale du réseau. Dans l'exemple sur la Figure 3.4, il n'y a qu'un seul neurone de sortie. Si $t = 1$, l'erreur totale du réseau est calculable grâce à la formule 3.5 :

$$E(\mathbf{w}, \mathbf{x}) = \frac{1}{2}(1 - 0.285545128)^2 = 0.255222882$$

   Alors l'erreur locale δ de ce neurone vaut, pour une fonction d'activation **Logistique** :

$$\begin{aligned} \delta_1^2 &= (1 - y_1^2) \times \phi_1^2(\nu_1^2)(1 - \phi_1^2(\nu_1^2)) \\ &= (1 - 0.285545128) \times (-7.08288025 \ \times \ (1 - (-7.08288025))) \\ \delta_1^2 &= -40.902593496 \end{aligned}$$

L'erreur du neurone de sortie peut être rétro-propagée dans le réseau et les erreurs locales des neurones cachés sont calculés avec la formule (3.7), en supposant toujours que les neurones utilisent la fonction **Logistique** :

$$\delta_{11}^1 = -40.902593496 \times 6 \times -0.013813374(1 - (-0.013813374)) = 3.436844501$$
$$\delta_{12}^1 = -40.902593496 \times 1 \times 1(1 - 1) = -40.902593496$$

Neurone (N_i^L)	Erreur locale (δ_j^L)
N_1^1 (j=1, L=1)	$\delta_1^1 = 3.436844501$
N_2^1 (j=2, L=1)	$\delta_2^1 = -40.902593496$

3. Il est dès lors possible de calculer les gradients des erreurs servant à appliquer les corrections sur les poids des neurones calculés grâce aux équations 3.6 pour la couche de sortie et l'équation 3.9 pour les couches cachées :
   Pour le premier neurone caché de la première couche, les gradients de l'erreur sont :

$$\Delta w_{01}^1 = 3.436844501 \times 1 = 3.436844501$$
$$\Delta w_{11}^1 = 3.436844501 \times (-5) = -17.18422505$$
$$\Delta w_{21}^1 = 3.436844501 \times 9 = 30.931600509$$

   Pour le second neurone caché de la première couche, les gradients de l'erreur sont :

$$\Delta w_{02}^1 = -40.902593496 \times 1 = -40.902593496$$
$$\Delta w_{12}^1 = -40.902593496 \times (-5) = 204.51296748$$
$$\Delta w_{12}^1 = -40.902593496 \times 9 = -368.123341464$$

   En utilisant l'équation (3.6), les trois poids du neurone de sortie à corriger, les gradients de l'erreur à appliquer sont :

$$\Delta w_{01}^2 = -40.902593496 \times 1 = -40.902593496$$
$$\Delta w_{11}^2 = 40.902593496 \times (-0.013813374) = +0.565002822$$
$$\Delta w_{21}^2 = -40.902593496 \times 1 = -40.902593496$$

Neurone (N_i^L)	Δw_{0i}	Δw_{1i}	Δw_{2i}
N_1^2	-40.902593496	0.565002822	-40.902593496
N_1^1	3.436844501	-17.184222505	30.931600509
N_2^1	40.902593496	204.51296748	-368.123341464

4. Enfin, en supposant un $\eta = 0.1$, on peut appliquer la correction aux poids.

   Pour le neurone de sortie N_1^2 :

$$w_{01}^2(t+1) = w_{01}^2(t) + \eta\Delta w_{01}^2 = 1 + 0.1 \times (-40.902593496) = -3.902593496$$
$$w_{11}^2(t+1) = 6 + 0.1 \times 0.56500288 = 6.0565002822$$
$$w_{21}^2(t+1) = -8 + 0.1 \times (-40.902593496) = -12.0902593496$$

   Pour le premier neurone de la couche cachée N_1^1 :

$$w_{01}^1(t+1) = 1 + 0.1 \times 3.436844501 = 1.3436844501$$
$$w_{11}^1(t+1) = 2 + 0.1 \times (-17.18422) = 0.28157749$$
$$w_{21}^1(t+1) = 4 + 0.1 \times 30.91600509 = 7.0931600509$$

Enfin, pour le second neurone de la couche cachée N_2^1 :

$$w_{01}^2(t+1) = 1 + 0.1 \times 40.902593496 = 5.0902593496$$
$$w_{11}^2(t+1) = -3 + 0.1 \times 204.51296748 = 17.451296748$$
$$w_{21}^2(t+1) = 8 + 0.1 \times (-368.123341464) = -28.8123341464$$

Neurone (N_i^L)	$\eta\ \Delta w_{0i}$	$\eta\ \Delta w_{1i}$	$\eta\ \Delta w_{2i}$
N_1^2	-4.0902593496	0.0565002822	-4.0902593496
N_1^1	0.3436844501	-1.7184222505	3.0931600509
N_2^1	4.0902593496	20.451296748	-36.8123341464

Les nouveaux poids au temps $t+1$ obtenus grâce la formule (3.10) sont :

Neurone (N_i^L)	$w_{0i}(t+1)$	$w_{1i}(t+1)$	$w_{2i}(t+1)$
N_1^2	-3.0902593496	6.0565002822	-12.0902593496
N_1^1	1.3436844501	0.281577749	7.0931600509
N_2^1	5.0902593496	17.451296748	-28.8123341464

3.2.2 Implémentation

L'algorithme peut se résumer en quatre étapes, qu'il faut nécessairement remettre dans l'ordre [Bis06] :

1. Effectuer la Forward Propagation : appliquer un exemple au réseau et calculer les potentiels et sorties de tous les neurones grâce aux formules (3.3) et (3.4).
2. Évaluer l'ensemble des erreurs δ_i de tous les neurones de sorties i en utilisant la formule (3.6).
3. Rétro-propager les erreurs grâce à la formule (3.9) afin d'obtenir les δ_j pour chaque neurone caché dans le réseau.
4. Une fois les erreurs rétro-propagées pour l'ensemble des exemples calculées, les poids sont modifiés grâce à la formule 3.10.

L'implémentation traditionnelle de la Descente du Gradient avec rétro-propagation comprend une mise à jour des poids du réseau après l'analyse de *l'ensemble des exemples disponibles* ainsi qu'un calcul de l'erreur *moyenne* de la couche de sortie sur l'ensemble des exemples.

Toutefois, plusieurs stratégies existent afin d'itérer les exemples d'un jeu de données dans l'algorithme de la descente de gradient. Les trois plus populaires sont :

— Full-Batch : Les corrections à appliquer aux poids sont calculées pour chacun des exemples avant d'être appliquées. Cette version est décrite dans l'Algorithme 3 et implémentée dans le Listing 3.5.
— Stochastique : A chaque itération, un exemple est choisi au hasard. Les corrections sont calculées uniquement pour cet exemple et sont directement appliquées.
— Mini-Batch : Cette stratégie se trouve entre la descente de gradient full-batch et stochastique. Un sous-ensemble des exemples disponibles sont choisis et mélangés à chaque itération. Une fois que les corrections sont calculées à partir de ce sous-ensemble, les poids sont mis-à-jour.

Une explication plus approfondie des alternatives à la Descente de Gradient Full-Batch se trouve en Annexe B.

L'implémentation de la rétro-propagation du gradient peut poser quelques difficultés. Étant donné qu'il s'agit de formules obtenues en appliquant des dérivations, les gradients peuvent également être calculés avec des méthodes numériques classiques. En pratique, ces méthodes numériques sont utilisées afin de vérifier que les gradients calculés avec la rétro-propagation sont correctes. Une présentation ainsi qu'une implémentation aux méthodes numériques de dérivation dans le cas des réseaux de neurones est exposée en Annexe C.

Algorithme 3 : Algorithme de la descente du gradient [Bis06] [KS96b].

Entrées : rdn : le réseau de neurones à corriger comportant L couches cachées de N neurones cachés,
$\mathbf{x} = (\mathbf{x}_1, ..., \mathbf{x}_K)$: la matrice des exemples d'apprentissage, dont chaque élément $\mathbf{x}_k$ est un vecteur d'entrée,
$\mathbf{t} = (\mathbf{t}_1, ..., \mathbf{t}_K)$: la matrice contenant les réponses aux K exemples d'apprentissage, dont chaque vecteur $\mathbf{t}_k$ contient N_s éléments qui correspondent aux sorties des N_s neurones de sortie du réseau,
nbIterations : le nombre d'itérations maximum que l'algorithme peut faire,
tauxApprentissage : le taux d'apprentissage,
seuilTolérance : le seuil que l'erreur moyenne du réseau doit atteindre.
Output : erreurMoyenne : l'erreur moyenne de la couche de sortie.

```
1  erreurMoyenne = ∞
2  pour Nb itérations < nbIterations ET erreurMoyenne > seuilTolérance faire
3      erreurGlobale = 0.0
4      deltaW[Nb de couche] [Nb de neurones] = {{0}}
5      deltaBiais[Nb de couches] [Nb de neurones] = {{0}}
6      deltaPoids[Nb de couches] [Nb de neurones] [Nb Poids] = {{{0}}}
7      pour Chacun des exemples k de 1 à K faire
8          Evaluer sortiez pour l'exemple k // FORWARD PROPAGATION
           // BACK PROPAGATION
9          pour Chacun des neurones de sortie i de 1 à N_s faire
               // Accumuler deltas couche de sortie selon (3.5)
10             erreurGlobale += 0.5 * (y[k][i] - sortie[i]) // Calculer erreur globale selon (3.1)
11             deltaW[L+1][i] = 0.5 * (y[k][i] - sortie[i]) // L + 1 = Indice couche de sortie
12             deltaBiais[L+1][i] = deltaW[L+1][i]
               // Calculer correction des poids p de la couche de sortie (équation (3.6))
13             pour Chacun des poids p allant de 1 à P du neurone de sortie i faire
14                 deltaPoids[L+1][i][p] += deltaW[L+1][i] * neurone[L][p].sortie
15             fin
16         fin
17         h = L // L = Indice dernière couche cachée
18         pour Chacune des couches cachées h allant de L à 1 faire
               // Accumuler deltas couches cachées avec la formule (3.7)
19             pour Chacun des neurones j de 1 à N_c de la couche cachée h faire
20                 deltaBiais[h][j] = deltaW[h+1][j] * neurone[h][j].derivee
                   // Calculer correction des poids p des couches cachées selon formule (3.9)
21                 pour Chacun des poids p de 1 à N du neurone j faire
22                     deltaPoids[h][j][p] += deltaW[h][j] * neurone[h-1][p].sortie
23                 fin
24             fin
25             h = h - 1
26         fin
27     fin
28     h = 1
29     pour Chacune des couches cachées h de 1 à L // MAJ poids grâce à la formule (3.10)
30      faire
31         pour Chacun des neurones j de la couche cachée h faire
32             neurone[h][j].biais = deltaBiais[h][j] * tauxApprentissage
33             pour Chacun des poids p du neurone j faire
34                 neurone[h][j].poids[p] += deltaPoids[h][j][p] * tauxApprentissage
35             fin
36         fin
37         h += 1
38     fin
39     erreurMoyenne = erreurGlobale/(taille entrées)
40 fin
```

Le code C/C++ de cet algorithme se situe dans le fichier Apprentissage/apprentissage.cpp :

```
void RetropropagationGradient(ReseauNeurone &rdn, vector<vector<double>> &
    entrees, vector<vector<double>> &sortiesDesirees, size_t nbIterations, double
     tauxApprentissage, double seuilTolerance)
{
    vector<vector<double>> deltasW, deltasB;
    vector<vector<vector<double>>> deltasWij;
    double erreurMoyenne=INFINITY;
    size_t nbCouches=rdn.couches.size();

    /* Allocation de l'espace mémoire du tableau de deltas W et B et Wij
     * deltasW = tableaux contenant les erreurs locales de chacun des neurones
     * deltasB = tableau contenant la correction a appliquer au biais de chacun
    des neurones
     * deltasWij = tableau contenant la correction a appliquer à chacun des biais
     de chacun des neurones
     */
    CreerTableauxDeltas(deltasW, deltasB, deltasWij, rdn);

    for(size_t i=0; i<nbIterations && abs(erreurMoyenne)>seuilTolerance; i++)
    {
        /* Remise à zéro du tableau de deltas W et B et Wij
         * deltasW = tableaux contenant les erreurs locales de chacun des neurones
         * deltasB = tableau contenant la correction a appliquer au biais de
    chacun des neurones
         * deltasWij = tableau contenant la correction a appliquer à chacun des
    biais de chacun des neurones
         */
        InitialiserTableauxDeltas(deltasW, deltasB, deltasWij, nbCouches);
        erreurMoyenne=0.0f;
        for(size_t j=0; j<entrees.size(); j++)
        {
            vector<double> exemple=entrees[j];
            /* FORWARD PROPAGATION
             * Présenter un vecteur d'entrée X et évaluer la sortie du neurone
             * exemple = l'exemple que l'on souhaite analyser avec le RDN
             */
            rdn.Evaluer(exemple);

            /* BACK PROPAGATION
             * rdn : le réseau de neurones auquel on souhaite appliquer l'
    algorithme de rétro-propagation
             * exemple = l'exemple que l'on souhaite analyser avec le RDN
             * sortiesDesirees[j] = un vecteur contenant un 1 pour la bonne réponse
     et 0 pour les mauvaises
             * deltasW = tableaux contenant les erreurs locales de chacun des
    neurones
             * deltasWij = tableau contenant la correction a appliquer à chacun des
     biais de chacun des neurones
             */
            erreurMoyenne += Retropropager(rdn, exemple, sortiesDesirees[j],
                  deltasW, deltasWij);

            // Copier les Deltas W dans Deltas B
            for(size_t k=0; k<deltasW.size(); k++)
                for(size_t l=0; l<deltasW[k].size(); l++)
                    deltasB[k][l] += deltasW[k][l];
        }

        if(entrees.size()>0)
```

```
            erreurMoyenne /= entrees.size();
51
        /* MAJ poids après avoir parcouru l'ensemble des exemples
53       * rdn : le réseau de neurones que l'on souhaite corriger
         * deltasB = tableau contenant la correction a appliquer au biais de
    chacun des neurones
55       * deltasWij = tableau contenant la correction a appliquer à chacun des
    biais de chacun des neurones
         * Taux d'apprentissage : le taux d'apprentissage a utiliser
57       */
        MAJPoids(rdn, deltasB, deltasWij, tauxApprentissage);
59
        cout << "Iteration " << i << '\t' << "Erreur Moyenne=" <<
61          erreurMoyenne << endl;
    }
63 }
```

Listing 3.5 – Code correspondant à la rétro-propagation du gradient de l'erreur dans Apprentissage/apprentissage.cpp

Tout d'abord, la fonction **CreerTableauxDeltas** alloue l'espace nécessaire aux tableaux deux et trois dimensions prévus pour contenir les termes correcteurs Δw_i de la formule 3.10. Quant à la fonction **InitialiserTableauxDeltas** réinitialise ces tableaux : elle remet à 0 les valeurs contenues dans ces tableaux.

En second lieu, la fonction **Evaluer** membre de la classe **ReseauNeurone** permet l'étape de *propagation en avant* de l'entrée. Ensuite, la fonction **Retropropager** permet de calculer, grâce à la formule 3.9 dite de rétro-propagation du gradient de l'erreur, les corrections à appliquer aux différents poids et ce pour chaque neurone de chaque couche.

```
1  double Retropropager(ReseauNeurone &rdn, vector<double> &exemple, vector<double>
       &sortiesDesirees, vector<vector<double>> &deltasW, vector<vector<vector<
       double>>> &deltasWij)
   {
3      size_t nbCouches=rdn.couches.size(), indiceCoucheSortie=nbCouches-1;

5      double erreurGlobale = CalculerErreursCoucheSortie(
           rdn.couches[indiceCoucheSortie], rdn.sortiesReseau[indiceCoucheSortie-1],
7          sortiesDesirees, deltasW[indiceCoucheSortie], deltasWij[indiceCoucheSortie
       ]);

9      for(int l=indiceCoucheSortie-1; l>0; l--)
           CalculerErreursCoucheCachee(rdn.couches[l], rdn.couches[l+1],
11             rdn.sortiesReseau[l-1], deltasW[l+1], deltasW[l], deltasWij[l]);

13     CalculerErreursCoucheCachee(rdn.couches[0], rdn.couches[1], exemple,
           deltasW[1], deltasW[0], deltasWij[0]);
15
       return erreurGlobale;
17 }

19 double CalculerErreursCoucheSortie(vector<Neurone> &src, vector<double> &entrees
       , vector<double> sortieDesiree, vector<double> &deltasW, vector<vector<double
       >> &deltasWij)
   {
21     double erreurGlobale=0.0f;
       for(size_t i=0; i<deltasW.size(); i++) // Pour chaque neurone de la couche
23     {
```

```
        erreurGlobale += CorrectionPerteQuadratique(src[i], sortieDesiree[i],
25          deltasW[i]);

27      for(size_t k=0; k<src[i].poids.size(); k++)
            deltasWij[i][k] += deltasW[i] * entrees[k];
29  }

31  return erreurGlobale;
}
33
void CalculerErreursCoucheCachee(vector<Neurone> &src, vector<Neurone> &
    coucheUlterieure, vector<double> &entrees, vector<double> &
    deltasCoucheUlterieure, vector<double> &deltasWDst, vector<vector<double>> &
    deltasWij)
35 {
    // Pour chaque neurone de la couche actuelle
37  for(size_t i=0; i<src.size(); i++)
    {
39      double errtmp = 0.0f;

41      for(size_t j=0; j<coucheUlterieure.size(); j++)
            errtmp += deltasCoucheUlterieure[j] * coucheUlterieure[j].poids[i];
43
        deltasWDst[i] = errtmp * src[i].getDerivee();
45
        for(size_t k=0; k<src[i].poids.size(); k++)
47          deltasWij[i][k] += deltasWDst[i] * entrees[k];
 }
49 }
```

Listing 3.6 – Code correspondant à la fonction Retropropager

La fonction **getDerivee** de la classe Neurone n'a pas encore été explicitée car inutile jusqu'à alors. Cette fonction permet simplement de calculer la dérivée de la fonction d'activation grâce à sa variable membre **fonctionActivationDerivee**, un pointeur de fonction devant être initialisé avec l'une des fonctions dérivées contenues dans le fichier Fonction/fonctionactivation.cpp, présent dans le Listing 3.8. Par exemple : la fonction **LogistiqueDerivee** ou**TangenteHyperboliqueDerivee**. Le code source de ces fonctions est disponible dans le Listing 3.7.

```
1 double Neurone::getDerivee()
{
3   return fonctionActivationDerivee(potentiel);
}
```

Listing 3.7 – Calcul de la dérivée de la fonction d'activation d'un Neurone, dans Neurone/neurone.cpp

```
double Logistique(double x)
2 {
    //f(x) = 1 / ( 1 + exp(-x) )
4   return 1/(1+exp(-x));
}
6
double LogistiqueDerivee(double x)
8 {
    //f'(x) = f(x) x ( 1 - f(x) )
10  double fx = Logistique(x);
    return fx*(1-fx);
```

```
}

double TangenteHyperbolique(double x)
{
    //f(x) = tanh(x)
    return tanh(x);
}

double TangenteHyperboliqueDerivee(double x)
{
    //f(x) = 1-tanh(x)^2
    return 1 - pow(tanh(x), 2);
}

double Lineaire(double x)
{
    //f(x) = x
    return x;
}

double LineaireDerivee(double x)
{
    //f'(x) = (x)' = 1
    return 1;
}
```

Listing 3.8 – Rappels des différentes fonctions d'activation disponible pour les réseaux de neurones et de leur dérivée présentent dans Fonction/fonction.cpp.

Une fois que l'entièreté des exemples a été parcouru, les poids peuvent enfin être mis à jour grâce à la fonction **MAJPoids**. Cette fonction prend les facteurs correcteurs ainsi que le taux d'apprentissage comme paramètre.

```
void MAJPoids(ReseauNeurone &rdn, vector<vector<double>> &deltasW, vector<vector
    <vector<double>>> &deltasWij, double tauxApprentissage)
{
    size_t nbCouches=rdn.couches.size();

    // MAJ neurones couche cachée
    for(size_t l=0; l<nbCouches; l++) // l: indice couche (layer)
        MAJCoucheNeurones(rdn.couches[l], tauxApprentissage, deltasW[l],
            deltasWij[l]);
}

void MAJCoucheNeurones(vector<Neurone> &neurones, double tauxApprentissage,
    vector<double> &deltasB, vector<vector<double>> &deltasWij)
{
    // Pour chaque neurone de la couche
    for(size_t i=0; i<neurones.size(); i++)
    {
        // Pour chaque poids du neurone
        for(size_t j=0; j<neurones[i].poids.size(); j++)
            // MAJ POIDS
            neurones[i].poids[j] += tauxApprentissage * deltasWij[i][j];

        neurones[i].biais += tauxApprentissage * deltasB[i]; // MAJ biais
    }
}
```

Listing 3.9 – Application des corrections aux poids des neurones du réseau.

3.3 Résultats du PMC à une couche cachée

La porte logique XOR est un cas récurrent dans la littérature car elle n'est pas linéairement séparable. Par conséquent, un perceptron mono-couche ne pourrait pas classifier correctement les exemples de cette fonction. Cependant, avec un réseau de neurones à une seule couche cachée avec 2 neurones cela devient possible. La fonction d'activation utilisée sera la sigmoïde, bornée entre $]0, 1[$ et les classes choisies en conséquences sont 0 et 1.

```
#include <Neurone/reseauneurone.h>
#include <Apprentissage/apprentissage.h>

void main()
{
    vector<vector<double>> x={{0,0}, {1,0}, {0,1}, {1,1 }};
    vector<vector<double>> y={ {0}, {1}, {1}, {0} };
    ReseauNeurone rdn(2, Logistique, 2, 1, 1);

 /* Application de l'algorithme de rétro-propagation
    * rdn : le réseau de neurones que l'on souhaite corriger
    * x = le vecteur deux dimensions contenant les exemples à évaluer
    * y = les réponses aux entrées à évaluer afin de comparer les résultats
    * 10000 = le nombre d'itérations maximum que l'algorithme peut faire
    * Taux d'apprentissage : le taux d'apprentissage a utiliser
    */
    RetropropagationGradient(rdn, x, y, 10000, 0.7f, 0.01);

    cout << "Poids après backprop: " << endl << rdn.PoidsToString() << endl;

    for(vector<double> exemple : x)
    {
        rdn.Evaluer(exemple);
        cout << "Résultat: " << rdn.sortiesReseau[rdn.sortiesReseau.size()-1][0]
           << endl;
    }
}

// Iteration 681  Erreur Moyenne=0.00994237
// Poids apres backprop:
// Couche 1
// Neurone1: 2.86005  2.86054 biais: -4.3319    Neurone 2: 5.02929  5.03192
    biais: -1.85772
// Couche 2
// -6.13623 5.81114 biais: -2.58525

// Resultat: 0.132328
// Resultat: 0.863727
// Resultat: 0.863743
// Resultat: 0.156166
```

Listing 3.10 – Tentative de création d'un RDN capable de classifier comme le fait la fonction XOR

Sur la Figure 3.5, la limite de décision est tracée pour le réseau de neurones entraîné dans le Listing 3.10. La classification se déroule correctement. En bleu clair, tous les points pour lequel le RDN considère comme appartenant à la classe 0. En saumon, tous les points pour lesquels le RDN donne la classe 1 comme résultat.

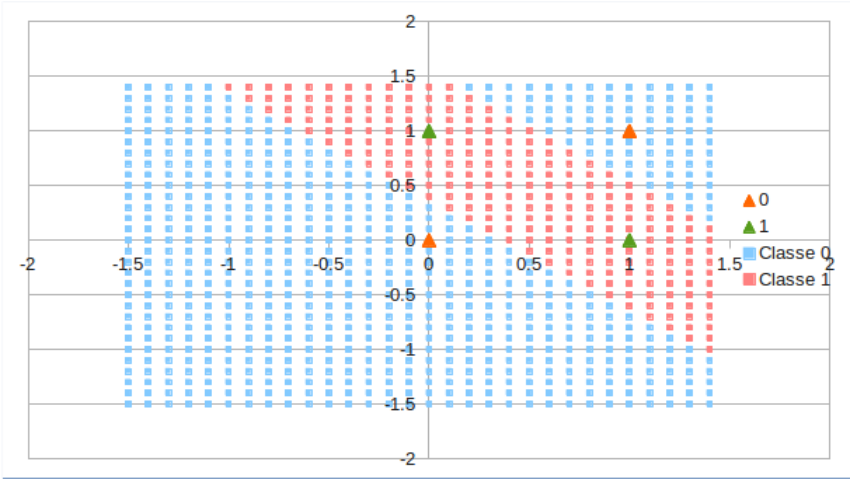


FIGURE 3.5 – Limite de décision du réseau de neurones pour la fonction XOR issue du Listing 3.10.

3.4 Théorème d'approximation universel

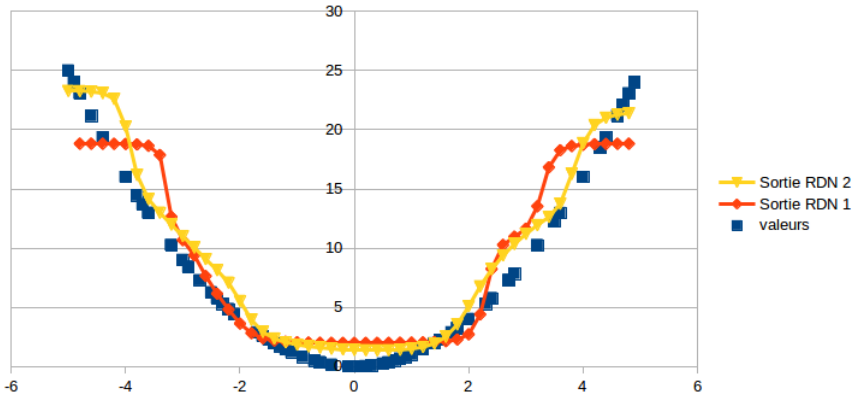


FIGURE 3.6 – La fonction $y(x) = x^2$ approximée par deux réseaux de neurones.

En 1989, le mathématicien CYBENKO, a démontré qu'une fonction continue de n variables peut être approchée avec une précision arbitraire par une combinaison linéaire de la fonction sigmoïde. Ce qui signifie qu'un réseau de neurones à fonction d'activation sigmoïdale ne possédant qu'une seule couche cachée et composée d'au minimum de deux neurones cachés peut approcher n'importe quelle fonction continue [Gé07]. En 1991, le statisticien HORNIK a complété ce théorème en prouvant que l'approximation universelle n'est pas une propriété de la fonction sigmoïdale mais est valable pour n'importe qu'elle fonction continue, bornée et non-constante [KS96b] [Gé07].

Les travaux de CYBENKO et HORNIK ont démontré que les réseaux de neurones sont des approximateurs universels. Ce théorème n'est qu'un théorème d'existence et ne donne en aucun cas une méthode permettant de trouver les paramètres du réseau [Gé07]. Dans le cas des réseaux de neurones, ce n'est pas la précision du réseau qui prime mais sa capacité à se généraliser à d'autres cas [Gé07].

En guise d'illustration du théorème, la Figure 3.6 où deux réseaux de neurones sont entraînés à reproduire la fonction $y(x) = x^2$. Le réseau de neurones 1 , en rouge, a été entraîné avec 1000

itérations et s'approche de la forme de la courbe formée par les points bleus, représentant la fonction x^2. Le second réseau, en jaune, a effectué 2000 itérations et s'adapte d'autant mieux. Dans un réseau de neurones conçu à des fins de régression, le neurone de sortie doit utiliser la fonction d'activation **Linéaire** . S'il utilise une autre fonction d'activation, sa sortie sera bornée et il ne sera pas apte à prédire n'importe quelle valeur.

Chapitre 4

Application à la reconnaissance des caractères de plaques d'immatriculation

4.1 Le programme *RDN Maker*

Afin de rendre les tests plus conviviaux, un programme avec une interface graphique, présentée sur la Figure 4.1, à été développé en C++. Ce programme utilise la librairie **NeuroLib** de réseaux de neurones, élaborée dans le cadre de ce stage ainsi que **Qt** pour l'interface graphique. L'interface présente quatres zones, visibles sur la Figure 4.1 :

1. La barre d'outil : elle permet d'avoir accès à des fonctionnalités d'enregistrement et de lecture de fichiers stockant des perceptrons mono/multi-couches, de conversion ou de traitement des fichiers CSV utilisés durant les manipulations ou encore d'avoir accès à certaines applications telle que la reconnaissance de caractères sur les plaques d'immatriculation.
2. La seconde zone concerne les fichiers contenant les données destinées à être utilisées pour l'entraînement et pour les tests.
3. La troisième zone contient l'ensemble des paramètres d'entraînement d'un perceptron mono/multi-couches ainsi qu'une zone de log qui affichera des informations complémentaires.
4. Enfin, la dernière zone qui contient les résultats de l'entraînement ainsi que des tests. Étant donné que l'objectif de ce stage est d'appliquer les réseaux de neurones sur des images, l'image est également affichée lorsque le test est effectué sur un exemple bien précis, qu'il est possible de choisir : paramètre "*Indexe de la donnée à afficher*", puis cliquer sur "*Tester élément*".

4.1.1 Les fichiers CSV

Dans le cadre des applications, les fichiers Comma-Separated Values, CSV en abrégé, sont abondamment utilisés en raison de leur simplicité. Ceux-ci stockent les jeux de données d'apprentissage et de test ainsi que des perceptrons mono/multi-couches sauvegardés pour une utilisation ultérieure. La structure d'un fichier CSV permet de contenir n'importe quel type de données. Le fichier est formatté en lignes, séparées par un *carriage return* et en colonnes, délimitées par une virgule, point-virgule ou une tabulation. Ce format permet à un éditeur de texte de lire aisément le fichier et permettent aux exemples d'être visualisées sur des graphiques avec des logiciels tableurs tels qu'*Excel* ou *LibreOffice-Calc*. Si les données sont en deux ou trois

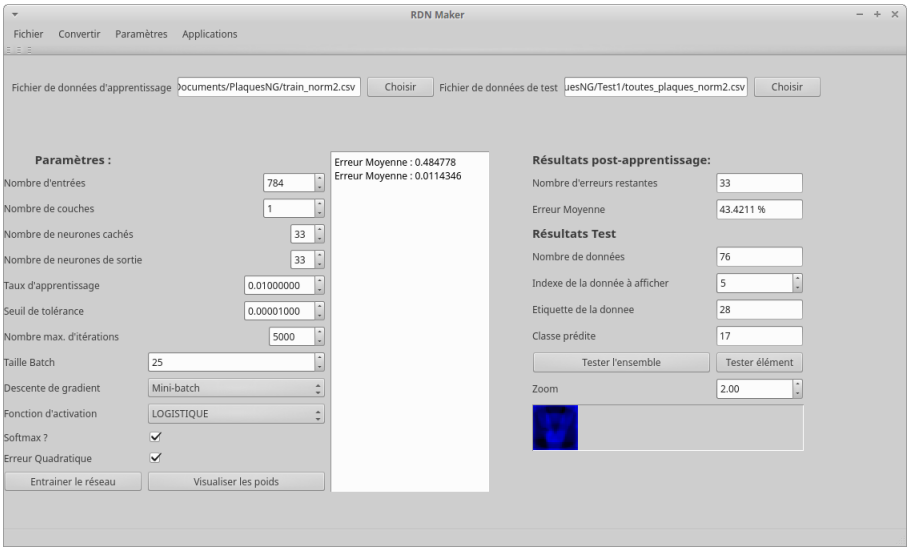


FIGURE 4.1 – *RDN Maker*, l'application développée dans le cadre des tests.

dimensions alors elles peuvent être visualisées dans des graphique. Les tableurs permettent également d'appliquer de nombreuses formules pré-encodées afin de comparer ou encore de vérifier lors des différentes phases de l'élaboration d'un jeu de données ou d'implémentation de fonctionnalités. De plus, si un pré-traitement sur les données est requis, les nouvelles données issues de ce processus seront enregistrées dans un second fichier de façon à ne pas devoir répéter cette manœuvre.

Dans le cadre de ce stage, les exemples contenus dans les fichiers CSV sont des images dont chaque colonne représente un pixel de l'image et chaque ligne contient une image. De plus, les images doivent pouvoir être reformée à partir des fichiers. Par conséquent, le nombre de lignes et de colonnes doivent nécesserairement être répertoriés. En outre, la sortie souhaitée pour cette image doit également pouvoir être retrouvée. Dès lors, les sorties sont ajoutées après les entrées. Ces sorties sont encodées sous forme de "*one hot encoded vector*", un vecteur où chaque élément correspond à une sortie possible. La valeur "1" est attribuée à la bonne réponse parmi celles disponibles tandis que les autres sont à 0. C'est pourquoi les fichiers CSV utilisés par le programme *RDN Maker* respectent une syntaxe spécifique, en plus du format CSV. La première ligne doit contenir les paramètres suivant :

- Un magic number : nécessaire afin de connaître le type de données contenues dans le fichier CSV : 0 pour des jeux de données ; 1 pour un perceptron mono-couche ; 2 pour un perceptron multi-couches.
- La quantité de données contenue dans le fichier.
- Le nombre de lignes de l'image.
- Le nombre de colonnes de l'image.
- Le nombre de sorties possibles pour le jeu de données.

En guise d'exemple, supposons qu'un fichier contienne les 26 lettres de l'alphabet sous forme d'images binarisées de dimensions 28×28. Le fichier contiendrait donc 26 exemples, en considérant que chaque lettre n'apparaît qu'une seule fois *dans l'ordre alphabétique* dans le fichier. Chaque image du jeu de données correspond à une ligne dans le fichier et les pixels de ces images sont transformés en un vecteur de $H \times L = 28 \times 28 = 784$ entrées. S'ensuit un one hot encoded vector de 26 sorties. Le principe de l'encodage d'images dans des fichiers CSV décrit dans cette section est synthétisé dans la Table 4.1 et un exemple réél de fichier CSV est fournit dans le Listing 4.1.

En-tête				
Magic number	*Quantité*	*Nb Lignes*	*Nb Colonnes*	*Nb Sorties*
0	26	28	28	26
Données				
Entrées (784)				*Sorties (26)*
0, 255, ..., 0, ..., 255				1, 0, 0, ...
255, 255, ..., 0, ..., 255				0, 1, 0, ...
...				

TABLE 4.1 – Principe des images encodées dans des fichier CSV.

```
0,33,28,28,33,
255,255,255,0,0,0,0,255,0,...,1,0,0,0,0,0,0,0,0,...,0,0,0,0
255,255,0,0,0,0,255,255,0,...,0,1,0,0,0,0,0,0,0,...,0,0,0,0
255,255,255,0,0,0,255,0,0,...,0,0,1,0,0,0,0,0,0,...,0,0,0,0
```

Listing 4.1 – Exemple réel d'un fichier CSV content l'en-tête et les lettres "A", "B" et "C".

4.1.2 Pré-traitement des données

Selon [KJ18b], les données doivent être également être pré-traitées avant d'être apprises par un réseau de neurones ou avant d'être évaluées par ce dernier. C'est pour cela que le premier pré-traitement est normaliser tous les exemples selon une taille fixe. La taille choisie dans le cadre de ces expérimentations sera 28×28, comme pour la base des caractères manuscrits MNIST, utilisée comme base commune des comparaisons des différentes librairies de machine learning au Chapitre 5.

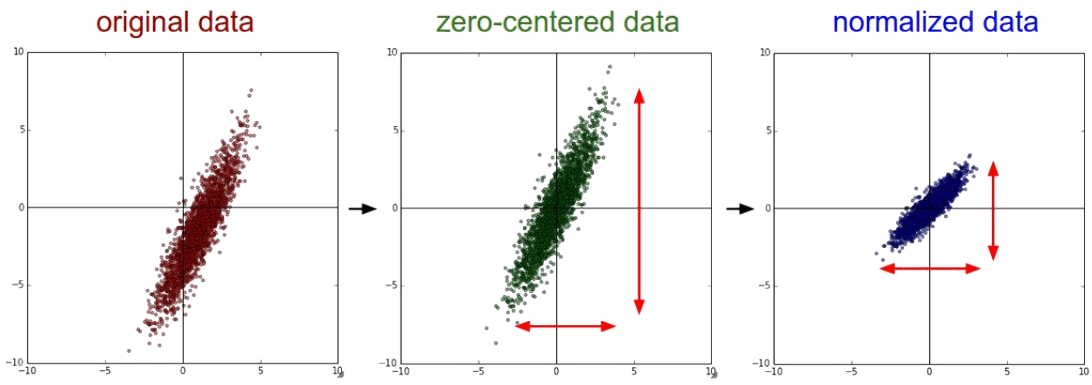


FIGURE 4.2 – Exemple de données centrées-réduites provenant de la référence [KJ18b].

Un second pré-traitement consiste à calculer la moyenne et l'écart-type des exemples d'apprentissage, par *variables*, et d'appliquer la formule de *Normalisation* de l'équation (4.1). Dans le cas présent, chaque variable d'entrée du perceptron consiste en un pixel et il est donc nécessaire de calculer la moyenne et l'écart-type pour chacun des pixels grâce aux formules (4.2) et (4.3). Ces pré-traitements sont également appelés *centrage* et *réduction* des données. Parce qu'après ce pré-traitement, les données sont de moyenne nulle et d'écart-type unitaire. La Fi-

gure 4.2 présente un exemple de données successivement centrées puis réduites.

$$x'_{ki} = \frac{x_{ki} - \overline{x}_i}{s_i} \tag{4.1}$$

$$\overline{x}_i = \frac{1}{K} \sum_{k=1}^{K} x_{ki} \tag{4.2}$$

$$s_i = \sqrt{\frac{1}{K} \sum_{k=1}^{K} (x_{ki} - \overline{x}_i)^2} \tag{4.3}$$

Où :
— x'_{ki} désigne la $i$ème variable d'entrée *normalisée* du $k$ème exemple. La matrice $\mathbf{x}'$ est donc la matrice normalisée.
— x_{ki} désigne la $i$ème variable d'entrée du $k$ème exemple, $\mathbf{x}$ est une matrice de dimensions $K \times N$, où N est le nombre de variables d'entrées. Par exemple, $N = 28 \times 28 = 784$ pour l'exemple de la sous-section 4.1.1.
— $\overline{x}_i$ et s_i sont respectivement la moyenne et l'écart-type de la $i$ème variable d'entrée.
— K est le nombre d'exemples présents dans le jeu d'apprentissage.

Lorsque le jeu d'apprentissage est centré et réduit, les résultats sont enregistés dans un fichier CSV afin de pouvoir entraîner le réseau de neurones avec ce jeu de données. De plus, lors de l'enregistrement les moyennes et les écart-types de chacune des variables seront ajoutées à la fin du fichier. Ces deux informations supplémentaires sont essentielles afin de centrer et réduire les données lors de la phase de test ou durant l'évaluation d'exemples en phase de production. Le fichier CSV contenant des données centrées et réduites de l'exemple de la sous-section 4.1.1 ressemble donc à :

En-tête				
Magic number	*Quantité*	*Nb Lignes*	*Nb Colonnes*	*Nb Sorties*
0	26	28	28	26
Données				
Entrées (784)				*Sorties (26)*
0.65884, 0.918236, ..., 0.887528, ..., -0.159907				1, 0, 0, ...
0.922893, 0.901422, ..., -0.350211, ..., 0.815952				0, 1, 0, ...
...				
Moyennes				
226.333, 211, ..., 190.121, ..., 187.667				Vide
Écart-types				
43.5108, 53.9248, ..., 71.9738, ..., 70.9529				Vide

4.2 Les jeux de données

4.2.1 Le jeu d'apprentissage

FIGURE 4.3 – L'image dont les caractères du jeu d'apprentissage sont issus.

Les données d'apprentissage viennent d'une image prise sur Internet, disponible sur la Figure 4.3. Les caractères des plaques d'immatriculation étant assez similaires d'une plaque à l'autre, très peu de données d'apprentissage sont supposées nécessaires. En l'occurrence, un seul exemple de chaque caractère est présent dans le jeu d'apprentissage. Le but ici n'est pas de faire une chaîne de segmentation complète avec détection de la plaque d'immatriculation, mais d'analyser la capacité d'un réseau de neurones à classifier et à se généraliser. Les lettres ont été extraites hors de cette image grâce à **OpenCV** dont le code est listé dans le Listing 4.2. Ensuite, les moyennes et écart-types des pixels ont été calculés sur ces lettres qui ont été successivement centrées puis réduites.

Toutefois, même si la Figure 4.3 présente tous les caractères alphanumériques dans une certaine police de plaque d'immatriculation, les caractères "I", "O" et "U" ne sont *pas* présent sur les plaques d'immatriculation *françaises*. Celles-ci peuvent être confondues avec respectivement le "1", le "0" et le "V" [Wik18]. Le jeu de données présente alors 33 caractères différents au lieu de 36 : 10 chiffres (de 0 à 9) et 23 lettres de l'alphabet.

4.2.2 Le jeu de test

FIGURE 4.4 – Exemple de plaque choisie pour le jeu de test.

Le jeu de test comporte des images de caractères issues de plaques réelles prises en photo par une caméra de smartphone dans une ruelle. Un exemple de plaque photographiée dans la rue est visible sur la Figure 4.4, sur laquelle les imperfections sont évidentes. L'intérêt de telles données est de pouvoir mesurer l'efficacité des réseaux de neurones à se généraliser malgré les défauts que les caractères rééels peuvent présenter, alors que ce dernier s'est entraîné sur des données dignes d'un cas d'école. À nouveau, les caractères ont été extraits grâce au code présent dans le Listing 4.2.

Le code contenu dans le Listing 4.2 permet d'extraire les caractères contenus dans les plaques d'immatriculation. La première opération après avoir chargé l'image en mémoire est un seuillage automatique dont la valeur de seuil a été calculé avec l'algorithme d'OTSU. L'image étant désormais binaire, deux régions existent : le fond en blanc et les lettres en noir. Lors de la détection des contours des objets qui suit, les objets sont supposés être en blanc. C'est pourquoi, après le suillage, les couleurs de l'images sont inversées. Ensuite, les contours sont détectés. Cependant, la détection des contours peut contenir des contours qui ne sont que du bruit, c'est pourquoi un filtrage est effectué, qui se basent sur des paramètres des contours normalisés par rapport à la taille de l'image dont les seuils ont été déterminés empiriquement.

Lorsque les contours restant ne correspondent qu'à des lettres, les caractères peuvent être extraits. Les images des plaques ont été prises sous des angles parfois peu avantageux. Á cause de cela, une projection de perspective est effectuée grâce à la fonction **warpPerspective** prenant notamment en paramètre une matrice de transformation crée par la fonction **getPerspectiveTransform**. Les coins d'une lettre sont trouvés grâce à la fonction **TrouverCoins** recherchant,

parmi les points constituant le contour, les extrémités du contour. Dès lors, les caractères apparaissent bien droits sur les images ce qui permettra de faciliter la reconnaissance ultérieurement. Les caractères de la plaque présentée dans la Figure 4.4 sont visibles sur la Figure 4.5.

```
void ExtraireCaracteresPlaques(string &fichierSrc, string &fichierDst, string &extension)
{
  // Lire l'image
    cv::Mat fichier = cv::imread(fichierSrc, cv::IMREAD_GRAYSCALE),
          fichierSeuille = cv::Mat::zeros(fichier.rows, fichier.cols, CV_8UC1),
          fichierInverse = cv::Mat::zeros(fichier.rows, fichier.cols, CV_8UC1);
    int newH=28, newW=28; // Taille des images des caractères extraits

  // Seuillage d'OTSU
    cv::threshold(fichier, fichierSeuille, 0, 255, cv::THRESH_OTSU | cv::THRESH_BINARY);

  // Inversion des couleurs
    cv::bitwise_not(fichierSeuille, fichierInverse);

    cv::Mat element = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(3, 3));
    cv::morphologyEx(fichierInverse, fichierInverse, cv::MORPH_CLOSE, element);

  // Détection des contours des caractères
    vector<vector<cv::Point>> contours, contoursCaracteres, coinsCaracteres;
    vector<cv::Vec4i> hierarchy;

    cv::findContours(fichierInverse, contours, hierarchy, cv::RETR_CCOMP, cv::CHAIN_APPROX_SIMPLE);

  // Filtrer contours
    for(size_t i=0; i<contours.size(); i++)
    {
        // Diviser par périmètre pour normaliser vis à vis des dim. de l'image
        double aire = cv::contourArea(contours[i])/(fichierInverse.rows * fichierInverse.cols),
               perimetre = cv::arcLength(contours[i], FALSE)/(fichierInverse.rows * fichierInverse.cols);

        if((aire > 0.03 || perimetre>0.0017))
            contoursCaracteres.push_back(contours[i]);
    }

    // Les 4 coins de chacun des contours
    for(vector<cv::Point> &contour : contoursCaracteres)
        coinsCaracteres.push_back(TrouverCoins(contour));

    // Extraire les caractères + transformatio n
    cv::Point2f orderedCorners[4], dstCorners[4];

    for(size_t i=0; i<.coinsCaracteres.size(); i++)
    {
        for(size_t j=0; j<coinsCaracteres[i].size(); j++) // conversion vector en tableau
        {
            orderedCorners[j].x = coinsCaracteres[i][j].x;
            orderedCorners[j].y = coinsCaracteres[i][j].y;
        }

    // Calculer dimensions image:  [0] = Hauteur, [1] = largeur
        vector<int> dimensions = CaculerDimensions(coinsCaracteres[i]);
```

```
52          dstCorners[0]=cv::Point2f(0.0f, 0.0f);
            dstCorners[1]=cv::Point2f(dimensions[1]-1, 0.0f);
54          dstCorners[2]=cv::Point2f(dimensions[1]-1, dimensions[0]-1);
            dstCorners[3]=cv::Point2f(0.0f, dimensions[0]-1);
56
            cv::Mat caractereExtrait = cv::Mat(dimensions[0], dimensions[1], CV_8UC1
    , cv::Scalar(0));
58          cv::Mat transform_matrix = cv::getPerspectiveTransform(orderedCorners,
    dstCorners);
            cv::warpPerspective(fichier, caractereExtrait, transform_matrix,
    caractereExtrait.size());
60
            cv::Mat caractereRetaille=cv::Mat(newH, newW, CV_8UC1, cv::Scalar(255));
62          cv::resize(caractereExtrait, caractereRetaille, caractereRetaille.size()
    );

64      // Créer le nom du fichier destination
            std::ostringstream stringStream;
66          stringStream << fichierDst << "_" << i << extension;

68          // Enregistrer sur le disque
            cv::imwrite(stringStream.str(), caractereRetaille);
70      }
}
72
vector<int> CaculerDimensions(vector<cv::Point> &coins)
74{
    vector<int> res(2); // Hauteur et Largeur
76  int largeurH, largeurB, hauteurH, hauteurB;

78  largeurH = coins[1].x - coins[0].x, // largeur en haut
    largeurB = coins[3].x - coins[2].x; // largeur en bas
80  res[1] = max(largeurH, largeurB);

82  hauteurH = coins[3].y - coins[0].y, // hauteur à gauche
    hauteurB = coins[2].y - coins[1].y; // hauteur à droite
84  res[0] = max(hauteurH, hauteurB);

86  return res;
}
88
vector<cv::Point> TrouverCoins(vector<cv::Point>pts)
90{
    vector<cv::Point> res(4);
92
    int xmax=pts[0].x, xmin=pts[0].x,
94      ymax=pts[0].y, ymin=pts[0].y;
    res[0].x = res[3].x = xmin; res[1].x = res[2].x = xmax;
96  res[0].y = res[1].y = ymin; res[2].y = res[3].y = ymax;

98  for(size_t i=1; i<pts.size(); i++)
    {
100     xmin = min(xmin, pts[i].x);
        xmax = max(xmax, pts[i].x);
102     ymin = min(ymin, pts[i].y);
        ymax = max(ymax, pts[i].y);
104 }

106 res[0].x = xmin; res[0].y = ymin;
    res[1].x = xmax; res[1].y = ymin;
```

```
108     res[2].x = xmax; res[2].y = ymax;
        res[3].x = xmin; res[3].y = ymax;
110     return res;
}
```

Listing 4.2 – Code permettant d'extraire les caractères des plaque d'immatriculation contenus dans une image.

FIGURE 4.5 – Les caractères extraits avec le code du Listing 4.2.

4.3 Perceptron mono-couche

Pour rappel, le perceptron mono-couche présente une limite de décision *linéaire* et permet donc de séparer parfaitement plusieurs catégories de données si et seulement si un hyper-plan dans $\mathbb{R}^N$ permet de toutes les données dans l'hyper-espace de dimensions N. L'exemple actuel se situe dans un hyper-espace de $N = 784$ dimensions, chaque pixel correspondant à une dimension. Par conséquent, le nombre d'erreur durant l'apprentissage des données par le perceptron mono-couche descendra jusqu'à zéro si et seulement si les données sont *linéairement séparables*.

Les fichiers CSV contenant les jeux d'apprentissage et de test doivent être préalablement chargé en mémoire. Ce chargement configure les paramètres "*Nombre d'entrées*" et "*Nombre de neurones de sortie*" en conséquence du contenus grâce aux en-têtes du fichier décrit dans la section 4.1.1.

Ensuite, l'utilisateur doit configurer les paramètres "*Nombre de couches*", "*Nombre de neurones cachés*", "*Taux d'apprentissage*", "*Descente de gradient*" et "*Fonction d'activation*". Les deux derniers paramètres sont choisis sans tests préalables. En effet, la descente de gradient d'ADALINE oscillait moins que la Descente de Gradient classique du perceptron autour du minimum local de la fonction d'erreur et convergeait plus rapidement. De plus, ADALINE fonctionne avec la fonction d'activation *Linéaire*, les fonctions d'activation *Signe* et HEAVISIDE étant abandonnée car non-dérivable. Le neurone choisi comme étant la sortie prédite par le perceptron mono-couche sera celui avec la sortie la plus grande. Le Listing 4.3 présente une tentative d'entraînement d'un perceptron mono-couche.

```
Chargement du fichier /home/julien/Documents/PlaquesNG/Test2/train_norm_adaline.
    csv terminé en 10ms 203us 648ns
Chargement du fichier /home/julien/Documents/PlaquesNG/Test2/train_norm_adaline.
    csv terminé en 8ms 688us 640ns
Temps mis pour entrainer: 2ms 205us 440ns Nombre d'erreur neurone: 0 = inf
Temps mis pour entrainer: 2ms 241us 792ns Nombre d'erreur neurone: 1 = inf
Temps mis pour entrainer: 2ms 136us 64ns Nombre d'erreur neurone: 2 = inf
...
Temps mis pour entrainer: 1ms 937us 408ns Nombre d'erreur neurone: 30 = inf
Temps mis pour entrainer: 1ms 930us 752ns Nombre d'erreur neurone: 31 = inf
Temps mis pour entrainer: 1ms 940us 992ns Nombre d'erreur neurone: 32 = inf
```

Listing 4.3 – Entrainement d'un perceptron mono-couche avec *RDN Maker* avec un taux d'apprentissage non-adapté.

En outre, le taux d'apprentissage doit être choisi en fonction de l'erreur retournée par le perceptron. Sur le Listing 4.3, l'erreur atteint la limite de précision de type **double** et affiche "*inf*". Le taux d'apprentissage est par conséquent trop grand et l'erreur ne converge pas. Tant que l'erreur moyenne diminue, le taux peut être diminué. Cependant, le fais qu'un taux trop faible fera converger l'erreur trop lentement doit être gardé à l'esprit. L'erreur est avoisine son minimum avec un taux d'apprentissage $\eta \leq 10^{-3}$. Ceci est établi avec seulement 5 itérations, afin de voir comment le perceptron se comporte avec un certain taux. Le meilleur taux est ensuite choisi en augmentant le nombre d'itérations, par exemple 100 itérations comme sur le Listing 4.4.

```
Temps mis pour entrainer: 30ms 377us 728ns Nombre d'erreur neurone: 0 =
    0.000918159
Temps mis pour entrainer: 28ms 289us 24ns Nombre d'erreur neurone: 1 =
    0.00099772
```

```
Temps mis pour entrainer: 28ms 249us 600ns Nombre d'erreur neurone: 2 =
    0.000994057
...
Temps mis pour entrainer: 30ms 210us 816ns Nombre d'erreur neurone: 30 =
    0.000892618
Temps mis pour entrainer: 29ms 923us 840ns Nombre d'erreur neurone: 31 =
    0.000976475
Temps mis pour entrainer: 29ms 864us 960ns Nombre d'erreur neurone: 32 =
    0.000993383
Test du jeu exécuté en 24ms 531us 712ns Erreur : 0 (0 %) # Erreur sur le jeu d'
    apprentissage
```

Listing 4.4 – Entrainement d'un perceptron mono-couche avec *RDN Maker* avec un taux d'apprentissage optimal.

L'erreur atteinte sur le Listing 4.4 est de l'ordre de 10^{-4}, le jeu d'apprentissage est linéairement séparable puisque l'évaluation post-apprentissage ne commet aucune erreur. Le Listing 4.5 affiche les résultats post-apprentissage sur le jeu de test. Le perceptron classifie correctement 64/77 (86.37%) des exemples de test et se trompe donc sur 13/77 (13.63%) des cas.

```
Test du jeu exécuté en 24ms 169us 728ns Erreur : 13 (16.8831 %) # Erreur sur le
    jeu de test
```

Listing 4.5 – Évaluation des performances du perceptron mono-couche sur le jeu de test après 100 itérations.

Malgré l"erreur très faible, le perceptron mono-couche ne peine à se généraliser. Augmenter le nombre d'itération jusqu'à 1000 itérations n'améliore cependant pas les résultats, comme le montre le Listing 4.6.

```
Temps mis pour entrainer: 36ms 814us 848ns Nombre d'erreur neurone: 0 =
    0.000918164
Temps mis pour entrainer: 29ms 7us 360ns Nombre d'erreur neurone: 1 =
    0.000997773
Temps mis pour entrainer: 29ms 217us 792ns Nombre d'erreur neurone: 2 =
    0.000994051
...
Temps mis pour entrainer: 29ms 460us 992ns Nombre d'erreur neurone: 30 =
    0.000892604
Temps mis pour entrainer: 28ms 994us 816ns Nombre d'erreur neurone: 31 =
    0.000976454
Temps mis pour entrainer: 29ms 107us 456ns Nombre d'erreur neurone: 32 =
    0.000993393
Test du jeu d'apprentissage exécuté en 23ms 343us 872ns Erreur : 13 (16.8831 %)
```

Listing 4.6 – Évaluation des performances du perceptron mono-couche sur le jeu de test après 1000 itérations.

4.4 Perceptron multi-couches avec l'Erreur Quadratique

Le jeu de test étant linéairement séparable, le perceptron mono-couche peine quelque peu à diminuer son erreur en dessous de 13/77 exemples mal classifiés. Voyons si les résultats sont meilleurs avec un perceptron multi-couches. Un réseau de neurones possède de nombreux paramètres qui doivent être testés et adaptés :

- Le taux d'apprentissage qui dépend de la fonction d'erreur et d'activation utilisée,
- La fonction d'activation (**Logistique** ou **Tangente Hyperbolique**),
- Le nombre de neurones par couches cachées,
- Le nombre de couche cachées,
- La fonction d'erreur (quadratique ou entropie relative),

Afin de rechercher les paramètres optimaux, le point de départ est un réseau de neurones ne comportant qu'une seule couche cachée de 10 neurones ayant pour fonction d'activation la fonction **Logistique**. La correction du réseau se fait grâce à l'algorithme de la rétro-propagation du gradient *Full-Batch* utilisant l'erreur quadratique et itérant maximum 50 fois.

4.4.1 Avec la fonction d'activation Logistique

Trouver le taux d'apprentissage

Le premier paramètre dont il faut se soucier est le taux d'apprentissage. Celui-ci permet une convergence plus ou moins rapide. Un taux d'apprentissage trop grand fera osciller l'erreur autour d'un minimum et un taux d'apprentissage trop faible fera converger l'erreur vers le minimum beaucoup trop lentement. Typiquement, les premiers taux à essayer sont $\eta_1 = 10^{-1}$ et $\eta_2 = 10^{-2}$. Si η_2 donne une erreur moyenne plus faible que η_1, alors le taux d'apprentissage voit son ordre de grandeur réduire. Par ailleurs, il se peut que le taux d'apprentissage doit être adapté lorsque le nombre d'itération augmente. En effet, à partir d'un certain nombre d'itération, il se peut que le taux d'apprentissage ne fasse plus converger l'erreur vers un minimum.

Le Listing 4.7 montre quatre essais avec un taux d'apprentissage décroissant valant respectivement 0.1, 0.01, 0.001 et 0.0001. L'erreur moyenne du réseau ne fait qu'augmenter avec les taux $\eta \leq 0.001$ et nous pouvons ainsi conclure que le meilleur taux d'apprentissage est $\eta = 0.01$.

```
# Taux d'apprentissage = 0.1
Temps mis pour entrainer: 328ms 563us 712ns Erreur Moyenne : 0.553646
# Taux d'apprentissage = 0.01
Temps mis pour entrainer: 333ms 181us 184ns Erreur Moyenne : 0.521164
# Taux d'apprentissage = 0.001
Temps mis pour entrainer: 354ms 860us 32ns Erreur Moyenne : 2.94189
# Taux d'apprentissage = 0.0001
Temps mis pour entrainer: 318ms 424us 576ns Erreur Moyenne : 7.93025
```

Listing 4.7 – Comparaison de l'erreur moyenne donnée par un réseau de neurones à 1 couche cachée de 10 neurones, après 50 itérations et différents taux d'apprentissage décroissant

Trouver le bon nombre de neurones cachés

Afin de trouver le nombre de neurones par couche cachées, il est nécessaire de tester avec quelques itérations et un certain taux d'apprentissage. Reprenons donc nos 50 itérations et le taux précédemment trouvé : $\eta = 0.01$. Le nombre de neurones par couche N_h, sans considérer la

couche de sortie, commencera avec $N_h = 10$ et sera augmenté de 10 neurones à chaque itération. La quantité *optimale* sera celle avec une erreur moyenne minimale.

Le Listing 4.8 présente l'erreur moyenne d'un réseau ayant de $N_h = 10$ à $N_h = 60$ neurones cachés. L'erreur moyenne ne fait que chuter avec une quantité de neurones cachés croissante. La quantité $N_h = 40$ sera gardée comme quantité de neurones par couche cachées car la diminution de l'erreur moyenne est très faible avec un N_h croissant. Le but étant de ne pas complexifier le modèle sans apport utile.

```
# 10 neurones cachés
Temps mis pour entrainer: 319ms 762us 944ns Erreur Moyenne : 0.526908
# 20 neurones cachés
Temps mis pour entrainer: 631ms 385us 856ns Erreur Moyenne : 0.505814
# 30 neurones cachés
Temps mis pour entrainer: 969ms 221us 632ns Erreur Moyenne : 0.496606
# 40 neurones cachés
Temps mis pour entrainer: 1s 267ms 615us 232ns Erreur Moyenne : 0.492233
# 50 neurones cachés
Temps mis pour entrainer: 1s 544ms 131us 72ns Erreur Moyenne : 0.489807
# 60 neurones cachés
Temps mis pour entrainer: 1s 887ms 258us 368ns Erreur Moyenne : 0.488333
```

Listing 4.8 – Comparaison de l'erreur moyenne donnée par un réseau de neurones à 1 couche cachée après 50 itérations et un taux d'apprentissage $\eta = 0.01$ avec une quantité de neurones cachés croissant.

L'impact du nombre de couches

La quantité de couche cachées permet d'établir une fonction plus complexe [Gé07]. Le Listing 4.9 affiche l'erreur moyenne après un apprentissage du réseau de neurones avec un nombre de couches cachées croissant. L'apport d'une couche fait diminuer très lentement l'erreur moyenne avec le nombre de couches. À nouveau, le modèle le plus simple sera conservé car il semble inutile de rendre le modèle trop complexes sans apport utile.

```
# 1 couche cachée
Temps mis pour entrainer: 1s 330ms 804us 224ns Erreur Moyenne : 0.492233
# 2 couches cachées
Temps mis pour entrainer: 1s 402ms 602us 240ns Erreur Moyenne : 0.488253
# 3 couches cachées
Temps mis pour entrainer: 1s 469ms 522us 688ns Erreur Moyenne : 0.488194
# 4 couches cachées
Temps mis pour entrainer: 1s 543ms 786us 752ns Erreur Moyenne : 0.488194
```

Listing 4.9 – Comparaison de l'erreur moyenne donnée par un réseau de neurones de 40 neurones après 50 itérations, un taux d'apprentissage $\eta = 0.01$ avec une quantité couches cachées croissante.

Choisir le bon algorithme de descente de gradient

À présent, l'objectif est de comparer l'impact du choix de l'algorithme de descente de gradient. L'algorithme Full-Batch a été présenté au Chapitre 3 mais deux alternatives également très populaires sont présentées en Annexe B. Pour rappel, les paramètres fixés sont :

- Le réseau comporte $N_h = 40$ unités cachées,
- Le réseau comporte 33 neurones de sorties,
- Le réseau comporte 784 entrées,

— Le réseau comporte 1 couche cachée,
— Les neurones du réseau utilisent la fonction d'activation logistique ainsi que sa dérivée,
— Le taux d'apprentissage est de $\eta = 0.01$.

Les résultats après 200 itérations sont disponibles sur le Listing 4.10. Le nombre d'itérations a été augmenté afin de visualiser l'impact sur le long terme du taux d'apprentissage. Le premier exemple concerne un réseau de neurones entraîné avec la rétro-propagation Full-Batch. Le second avec la Stochastique et les trois derniers avec un Mini-Batch respectivement de taille 33, soit la taille du jeu d'apprentissage, 20 et 15 exemples. Le meilleur résultat est obtenu par le réseau ayant été entraîné par la méthode Stochastique 41 erreurs sur les 77 éléments présents dans le jeu de test ansi que l'erreur moyenne la plus faible. Remarquons toutefois que l'erreur moyenne la plus faible n'est pas forcément celle du réseau donnant les meilleurs résultats lors de la phase de test. Lorsque l'erreur moyenne est très faible mais que le réseau ne donne pas de bons résultats sur le jeu de test, cela signifie que le réseau ne se *généralise* pas correctement. Autrement dit, il a trop bien appris les données du jeu d'apprentissage, ce phénomène est aussi appelé *sur-apprentissage* ou overfitting en anglais.

Par ailleurs, la descente de gradient la plus longue est également la Stochastique. La Descente de Gradient Stochastique est à la rétro-propagation du gradient ce qu'ADALINE est à la Descente de Gradient d'un perceptron : la mise à jour se fait à chaque itération, en plus de mélanger les données dans le cas de la Descente de Gradient Stochastique. En termes d'efficacité, mettre à jour les poids fois est plus couteux que de le faire après quelques accumulation de gradient [KJ18c].

```
# Full-Batch
Temps mis pour entrainer: 5s 45ms 654us 16ns Erreur Moyenne : 0.49989
Test du jeu d'apprentissage exécuté en 18ms 536us 192ns Erreur : 71 (92.2078 %)
# Stochastique
Temps mis pour entrainer: 10s 721ms 897us 472ns Erreur Moyenne : 0.342325
Test du jeu d'apprentissage exécuté en 18ms 440us 448ns Erreur : 41 (53.2468 %)
# Mini-Batch 33
Temps mis pour entrainer: 5s 36ms 128us 768ns Erreur Moyenne : 0.499828
Test du jeu d'apprentissage exécuté en 18ms 759us 680ns Erreur : 74 (96.1039 %)
# Mini-Batch 20
Temps mis pour entrainer: 3s 180ms 326us 912ns Erreur Moyenne : 0.479579
Test du jeu d'apprentissage exécuté en 20ms 97us 536ns Erreur : 73 (94.8052 %)
# Mini-Batch 15
Temps mis pour entrainer: 2s 396ms 470us 16ns Erreur Moyenne : 0.484924
Test du jeu d'apprentissage exécuté en 19ms 304us 448ns Erreur : 71 (92.2078 %)
```

Listing 4.10 – Comparaison de l'erreur moyenne donnée par un réseau de neurones de 30 neurones après 50 itérations, un taux d'apprentissage $\eta = 0.1$ avec une quantité couches cachées croissante.

Cependant, les descentes de gradients Stochastiques et Mini-Batch effectuent le calcul du gradient sur un sous-ensemble des exemples du jeu d'apprentissage. De ce fait, les corrections apportées par la rétro-propagation sont moins importantes que pour le Full-Batch. Dès lors, essayer un taux d'apprentissage un peu plus important que celui utilisé actuellement permettra sans doute de palier ces "petites" corrections. Cet essai est effectué dans le Listing 4.11. Pour tous les algorthimes de descente de gradient excepté le Stochastique, cette idée à eu l'effet inverse. Un dernier essai a été ajouté, celui du Mini-Batch avec une mise à jour des poids tous les quatres exemples. Même si le résultat est moins bon que celui du Stochastique, l'effet escompté est bel et bien présent.

```
# Full-Batch
Temps mis pour entrainer: 5s 14ms 950us 656ns Erreur Moyenne : 0.5
Test du jeu d'apprentissage exécuté en 26ms 149us 120ns Erreur : 71 (92.2078 %)
# Stochastique
Temps mis pour entrainer: 10s 820ms 90us 880ns Erreur Moyenne : 0.22347
Test du jeu d'apprentissage exécuté en 18ms 729us 728ns Erreur : 31 (40.2597 %)
# Mini-Batch 33
Temps mis pour entrainer: 5s 38ms 210us 48ns Erreur Moyenne : 0.5
Test du jeu d'apprentissage exécuté en 19ms 377us 664ns Erreur : 76 (98.7013 %)
# Mini-Batch 20
Temps mis pour entrainer: 3s 86ms 733us 56ns Erreur Moyenne : 0.499977
Test du jeu d'apprentissage exécuté en 18ms 863us 872ns Erreur : 75 (97.4026 %)
# Mini-Batch 15
Temps mis pour entrainer: 2s 414ms 229us 760ns Erreur Moyenne : 0.456905
Test du jeu d'apprentissage exécuté en 27ms 744us 512ns Erreur : 73 (94.8052 %)
# Mini-Batch 4
Temps mis pour entrainer: 770ms 899us 456ns Erreur Moyenne : 0.441292
Test du jeu d'apprentissage exécuté en 27ms 846us 656ns Erreur : 54 (70.1299 %)
```

Listing 4.11 – Comparaison de l'erreur moyenne donnée par un réseau de neurones de 30 neurones après 50 itérations, un taux d'apprentissage $\eta = 0.1$ avec une quantité couches cachées croissante.

L'idée de garder un taux d'apprentissage plus élevé pour la Descente de Gradient Stochastique a été gardée dans le Listing 4.12 qui présente des apprentissages de réseaux de neurones avec un nombre d'itérations croissant. Le meilleur résultat est atteint par celui ayant effectué 600 itérations. Les réseaux suivants ont une erreur moyenne qui décroît avec un nombre d'itérations croissant mais sans aucun impact bénéfique sur l'erreur du jeu de test.

```
# 500 itérations
Temps mis pour entrainer: 27s 420ms 786us 432ns Erreur Moyenne : 0.00972404
Test du jeu exécuté en 18ms 719us 488ns Erreur : 3 (3.8961 %)
# 600 itérations
Temps mis pour entrainer: 32s 656ms 960us 768ns Erreur Moyenne : 0.00740723
Test du jeu exécuté en 21ms 625us 344ns Erreur : 2 (2.5974 %)
# 700 itérations
Temps mis pour entrainer: 39s 62ms 790us 400ns Erreur Moyenne : 0.00561103
Test du jeu exécuté en 20ms 468us 480ns Erreur : 3 (3.8961 %)
# 800 itérations
Temps mis pour entrainer: 43s 149ms 11us 712ns Erreur Moyenne : 0.00501266
Test du jeu exécuté en 19ms 358us 464ns Erreur : 2 (2.5974 %)
# 1000 itérations
Temps mis pour entrainer: 53s 836ms 986us 624ns Erreur Moyenne : 0.00338554
Test du jeu exécuté en 19ms 543us 40ns Erreur : 2 (2.5974 %)
# 1500 itérations
Temps mis pour entrainer: 1m 24s 804ms 663us 552ns Erreur Moyenne : 0.00216345
Test du jeu exécuté en 21ms 512us 960ns Erreur : 4 (5.19481 %)
```

Listing 4.12 – Comparaison de l'erreur moyenne donnée par un réseau de neurones de 30 neurones après 50 itérations, un taux d'apprentissage $\eta = 0.1$ avec une quantité couches cachées croissante.

4.4.2 Avec la fonction d'activation Tangente Hyperbolique

Dans le cas du réseau de neurones utilisant la fonction d'activation **Tangente Hyperbolique** ne se transforment pas en [-1, 1] et restent [0, 1]. En recherchant les paramètres de la même façon que dans la sous-section 4.4.1, les paramètres trouvés sont visibles sur le Listing 4.13 et sont :

— $N_h = 40$ neurones cachés,
— 1 seule couche cachée,
— un taux d'apprentissage $\eta = 0.01$,

```
# RECHERCHE DU MEILLEUR TAUX D'APPRENTISAGE
# Taux apprentissage = 0.1
Temps mis pour entrainer: 692ms 159us 744ns Erreur Moyenne : 0.871786
Test du jeu exécuté en 15ms 74us 560ns Erreur : 73 (94.8052 %)
# Taux apprentissage = 0.01
Temps mis pour entrainer: 679ms 136us Erreur Moyenne : 0.443398
Test du jeu exécuté en 19ms 389us 184ns Erreur : 64 (83.1169 %)
# Taux apprentissage = 0.001
Temps mis pour entrainer: 698ms 39us 296ns Erreur Moyenne : 0.488293
Test du jeu exécuté en 9ms 480us 704ns Erreur : 73 (94.8052 %)
# Taux apprentissage = 0.0001
Temps mis pour entrainer: 719ms 602us 176ns Erreur Moyenne : 4.89557
Test du jeu exécuté en 20ms 264us 960ns Erreur : 74 (96.1039 %)

# RECHERCHE DU NOMBRE DE NEURONES PAR COUCHE
# Nombre de neurones par couche = 10
Temps mis pour entrainer: 406ms 745us 88ns Erreur Moyenne : 0.375483
Test du jeu exécuté en 19ms 485us 696ns Erreur : 41 (53.2468 %)
# Nombre de neurones par couche = 20
Temps mis pour entrainer: 725ms 127us 936ns Erreur Moyenne : 2.91557
Test du jeu exécuté en 18ms 513us 664ns Erreur : 52 (67.5325 %)
# Nombre de neurones par couche = 30
Temps mis pour entrainer: 929ms 400us 576ns Erreur Moyenne : 0.594936
Test du jeu exécuté en 15ms 213us 824ns Erreur : 22 (28.5714 %)
# Nombre de neurones par couche = 40
Temps mis pour entrainer: 1s 272ms 440us 832ns Erreur Moyenne : 0.335119
Test du jeu exécuté en 22ms 233us 344ns Erreur : 15 (19.4805 %)
# Nombre de neurones par couche = 50
Temps mis pour entrainer: 1s 562ms 543us 360ns Erreur Moyenne : 2.59369
Test du jeu exécuté en 23ms 817us 472ns Erreur : 43 (55.8442 %)

# AUGMENTATION DU NOMBRE DE COUCHES
# Nombre de couche = 1
Temps mis pour entrainer: 318ms 625us 280ns Erreur Moyenne : 1.54903
Test du jeu exécuté en 9ms 475us 72ns Erreur : 61 (79.2208 %)
# Nombre de couche = 2
Temps mis pour entrainer: 340ms 22us 16ns Erreur Moyenne : 1.93941
Test du jeu exécuté en 10ms 903us 40ns Erreur : 71 (92.2078 %)
# Nombre de couche = 3
Temps mis pour entrainer: 348ms 284us 160ns Erreur Moyenne : 1.93941
Test du jeu exécuté en 19ms 579us 392ns Erreur : 76 (98.7013 %)

# TEST DES DIFFERENTES DESCENTES DE GRADIENTS (200 itérations)
# Full-Batch
Temps mis pour entrainer: 5s 3ms 269us 120ns Erreur Moyenne : 0.148083
Test du jeu exécuté en 19ms 217us 920ns Erreur : 8 (10.3896 %)
# Stochastique
Temps mis pour entrainer: 11s 27ms 89us 152ns Erreur Moyenne : 0.0101789
Test du jeu exécuté en 28ms 246us 16ns Erreur : 4 (5.19481 %)
# Mini-Batch de taille 33
```

```
Temps mis pour entrainer: 5s 30ms 494us 208ns Erreur Moyenne : 1.86698
Test du jeu exécuté en 36ms 201us 472ns Erreur : 17 (22.0779 %)
# Mini-Batch de taille 20
Temps mis pour entrainer: 3s 157ms 958us 656ns Erreur Moyenne : 0.119414
Test du jeu exécuté en 55ms 805us 440ns Erreur : 15 (19.4805 %)
# Mini-Batch de taille 15
Temps mis pour entrainer: 2s 431ms 526us 400ns Erreur Moyenne : 0.0683527
Test du jeu exécuté en 19ms 597us 56ns Erreur : 12 (15.5844 %)
```

Listing 4.13 – Recherche des paramètres du réseau de neurones avec la fonction **Tangente Hyperbolique**.

Le Listing 4.14 présente une augmentation du nombre d'itérations avec la Descente de Gradient Stochastique . L'erreur diminue entre 200 et 500 itérations mais augmente après. Un des résultats pouvant être acquis sans perdre trop de temps sur les paramètres est 5/77 erreurs, c'est-à-dire un taux de reconnaissance de 93.5% du jeu de test.

```
# 200 itérations
Temps mis pour entrainer: 10s 892ms 251us 136ns Erreur Moyenne : 0.00847005
Test du jeu exécuté en 21ms 902us 592ns Erreur : 7 (9.09091 %)
# 300 itérations
Temps mis pour entrainer: 16s 293ms 778us 432ns Erreur Moyenne : 0.00389034
Test du jeu exécuté en 20ms 828us 416ns Erreur : 7 (9.09091 %)
# 400 itérations
Temps mis pour entrainer: 21s 583ms 169us 24ns Erreur Moyenne : 0.00596786
Test du jeu exécuté en 22ms 23us 680ns Erreur : 7 (9.09091 %)
# 450 itérations
Temps mis pour entrainer: 21s 491ms 421us 440ns Erreur Moyenne : 0.00375775
Test du jeu exécuté en 64ms 367us 360ns Erreur : 5 (6.49351 %)
# 500 itérations
Temps mis pour entrainer: 27s 402ms 306us 304ns Erreur Moyenne : 0.0046154
Test du jeu exécuté en 19ms 4us 928ns Erreur : 6 (7.79221 %)
# 600 itérations
Temps mis pour entrainer: 32s 581ms 878us 528ns Erreur Moyenne : 0.00238486
Test du jeu exécuté en 58ms 10us 112ns Erreur : 7 (9.09091 %)
# 700 itérations
Temps mis pour entrainer: 37s 552ms 816us 640ns Erreur Moyenne : 0.00261696
Test du jeu exécuté en 42ms 787us 840ns Erreur : 7 (9.09091 %)
```

Listing 4.14 – Résultats obtenus avec la fonction d'activaton **Tangente Hyperbolique** divers quantités d'itérations avec la Descente de Gradient Stochastique.

4.5 Perceptron multi-couches avec l'Entropie Relative

Perceptron multi-couches avec l'Entropie Relative sans Softmax

La fonction d'erreur de l'Entropie Relative se base sur un réseau de neurones utilisant une fonction d'activation lui fournissant en entrée des valeurs comprises dans l'intervalle [0,1], telle que la fonction **Logistique**. Cette dernière est censée fournir une erreur "plus adaptée" afin de permettre une meilleure correction dans la rétro-propagation [Nie20]. Elle peut également s'utiliser conjointement avec la fonction d'activation **Softmax**, une fonction d'activation qui s'applique sur la couche de *sortie* du réseau de neurones. Cette dernière transforme les sorties des RDN en probabilité. Ces probabilités permettent ensuite d'effectuer un classement des prédictions du réseau, du plus probable au moins probable.

Le Listing 4.15 montre le résultats d'entrainements de réseaux de neurones utilisant la fonction d'activation **Logistique**.

```
# RECHERCHE DU MEILLEUR TAUX D'APPRENTISAGE
# Taux apprentissage = 0.1
Temps mis pour entrainer: 321ms 703us 680ns Erreur Moyenne : 0.480266
Test du jeu exécuté en 5ms 168us 128ns Erreur : 66 (85.7143 %)
# Taux apprentissage = 0.01
Temps mis pour entrainer: 318ms 718us 464ns Erreur Moyenne : 0.532968
Test du jeu exécuté en 6ms 317us 56ns Erreur : 75 (97.4026 %)
# Taux apprentissage = 0.001
Temps mis pour entrainer: 319ms 569us 408ns Erreur Moyenne : 2.94158
Test du jeu exécuté en 5ms 295us 360ns Erreur : 73 (94.8052 %)

# RECHERCHE DU NOMBRE DE NEURONES PAR COUCHE
# Nombre de neurones par couche = 10
Temps mis pour entrainer: 325ms 294us 80ns Erreur Moyenne : 0.554934
Test du jeu exécuté en 5ms 370us 368ns Erreur : 76 (98.7013 %)
# Nombre de neurones par couche = 20
Temps mis pour entrainer: 631ms 824us 384ns Erreur Moyenne : 0.483471
Test du jeu exécuté en 10ms 508us 32ns Erreur : 69 (89.6104 %)
# Nombre de neurones par couche = 30
Temps mis pour entrainer: 973ms 92us 96ns Erreur Moyenne : 0.49549
Test du jeu exécuté en 16ms 184us 832ns Erreur : 75 (97.4026 %)
# Nombre de neurones par couche = 40
Temps mis pour entrainer: 1s 269ms 379us 72ns Erreur Moyenne : 0.499895
Test du jeu exécuté en 19ms 405us 568ns Erreur : 75 (97.4026 %)
# Nombre de neurones par couche = 50
Temps mis pour entrainer: 1s 584ms 274us 688ns Erreur Moyenne : 0.499991
Test du jeu exécuté en 24ms 962us 304ns Erreur : 75 (97.4026 %)

# TEST DES DIFFERENTES DESCENTES DE GRADIENTS (200 itérations)
# Full-Batch
Temps mis pour entrainer: 2s 580ms 101us 888ns Erreur Moyenne : 0.369173
Test du jeu exécuté en 10ms 393us 600ns Erreur : 36 (46.7532 %)
# Stochastique
Temps mis pour entrainer: 5s 399ms 44us 864ns Erreur Moyenne : 0.406061
Test du jeu exécuté en 9ms 405us 184ns Erreur : 60 (77.9221 %)
# Mini-Batch de taille 33
Temps mis pour entrainer: 2s 571ms 421us 952ns Erreur Moyenne : 0.38001
Test du jeu exécuté en 9ms 868us 800ns Erreur : 37 (48.0519 %)
# Mini-Batch de taille 20
Temps mis pour entrainer: 1s 585ms 629us 184ns Erreur Moyenne : 0.441162
Test du jeu exécuté en 9ms 618us 944ns Erreur : 64 (83.1169 %)
# Mini-Batch de taille 15
```

```
Temps mis pour entrainer: 1s 199ms 822us 592ns Erreur Moyenne : 0.405392
Test du jeu exécuté en 9ms 944us 576ns Erreur : 56 (72.7273 %)
```

Listing 4.15 – Résultats obtenus avec la fonction d'activaton **Tangente Hyperbolique** divers quantités d'itérations avec la Descente de Gradient Stochastique.

Les paramètres retenus des divers essais du Listing 4.15 sont : un taux d'apprentissage $\eta = 0.1$, $N_h = 20$ neurones cachés et la Descente de Gradient Full-Batch . En augmantant le nombre d'itérations, le réseau de neurones atteints au mieux 4/77 erreurs avec 600 itérations.

```
# 500 itérations
Temps mis pour entrainer: 6s 303ms 599us 872ns Erreur Moyenne : 0.0376604
Test du jeu exécuté en 11ms 495us 168ns Erreur : 6 (7.79221 %)
# 600 itérations
Temps mis pour entrainer: 7s 576ms 784us 384ns Erreur Moyenne : 0.025455
Test du jeu exécuté en 11ms 413us 760ns Erreur : 4 (5.19481 %)
# 700 itérations
Temps mis pour entrainer: 8s 701ms 422us 80ns Erreur Moyenne : 0.0196959
Test du jeu exécuté en 18ms 313us 728ns Erreur : 11 (14.2857 %)
```

Listing 4.16 – Résultats obtenus avec la fonction d'activaton **Tangente Hyperbolique** divers en utilisant la Descente de Gradient Full-Batch et un nombre d'itération croissant.

Perceptron multi-couches avec l'entropie croisée avec Softmax

Le Listing 4.17 contient les résultats dans le cas de réseaux de neurones avec l'option **Softmax** activée et l'utilisation de la fonction d'entropie croisée. Des résultats assez performants sur le jeu de test sont atteints avec peu d'itérations. Notamment, un réseau de neurones atteint un score de 1/77 erreur avec la Descente de Gradient Mini-Batch avec un batch de taille 33, soit de l'entiereté du jeu de données d'apprentissage.

```
# TAUX D'APPRENTISSAGE
# Taux apprentissage = 0.1
Temps mis pour entrainer: 340ms 892us 160ns Erreur Moyenne : 1.11652
Test du jeu exécuté en 5ms 750us 272ns Erreur : 23 (29.8701 %)
# Taux apprentissage = 0.01
Temps mis pour entrainer: 340ms 742us 656ns Erreur Moyenne : 3.32231
Test du jeu exécuté en 21ms 432us 576ns Erreur : 60 (77.9221 %)
# Taux apprentissage = 0.001
Temps mis pour entrainer: 338ms 949us 632ns Erreur Moyenne : 3.49647
Test du jeu exécuté en 23ms 415us 296ns Erreur : 69 (89.6104 %)

# NEURONES CACHES PAR COUCHE
# 10 neurones
Temps mis pour entrainer: 342ms 705us 408ns Erreur Moyenne : 0.954994
Test du jeu exécuté en 22ms 976us Erreur : 28 (36.3636 %)
# 20 neurones
Temps mis pour entrainer: 661ms 752us 576ns Erreur Moyenne : 0.450336
Test du jeu exécuté en 19ms 888us 128ns Erreur : 14 (18.1818 %)
# 30 neurones
Temps mis pour entrainer: 978ms 248us 960ns Erreur Moyenne : 0.180397
Test du jeu exécuté en 18ms 926us 336ns Erreur : 8 (10.3896 %)
# 40 neurones
Temps mis pour entrainer: 1s 292ms 452us 96ns Erreur Moyenne : 0.123305
Test du jeu exécuté en 20ms 892us 416ns Erreur : 6 (7.79221 %)
# 50 neurones
Temps mis pour entrainer: 1s 602ms 193us 408ns Erreur Moyenne : 0.0814576
Test du jeu exécuté en 33ms 98us 752ns Erreur : 4 (5.19481 %)
```

```
# 60 neurones
Temps mis pour entrainer: 1s 913ms 293us 824ns Erreur Moyenne : 0.0618655
Test du jeu exécuté en 34ms 126us 336ns Erreur : 4 (5.19481 %)

# NOMBRE DE COUCHES CACHEES
# 1 couche cachée
Temps mis pour entrainer: 1s 600ms 670us 208ns Erreur Moyenne : 0.0817067
Test du jeu exécuté en 31ms 990us 272ns Erreur : 3 (3.8961 %)
# 2 couches cachées
Temps mis pour entrainer: 1s 760ms 393us 728ns Erreur Moyenne : 3.473
Test du jeu exécuté en 27ms 65us 88ns Erreur : 76 (98.7013 %)
# 3 couches cachées
Temps mis pour entrainer: 1s 898ms 775us 808ns Erreur Moyenne : 3.49651
Test du jeu exécuté en 61ms 665us 536ns Erreur : 73 (94.8052 %

# DESCENTES DE GRADIENT (200 itérations)
# Full-Batch
Temps mis pour entrainer: 6s 431ms 568us 384ns Erreur Moyenne : 0.0141642
Test du jeu exécuté en 26ms 154us 752ns Erreur : 3 (3.8961 %)
# Stochastique
Temps mis pour entrainer: 13s 899ms 859us 968ns Erreur Moyenne : 0.023263
Test du jeu exécuté en 25ms 426us 688ns Erreur : 7 (9.09091 %)
# Mini-Batch de taille  33
Temps mis pour entrainer: 6s 392ms 731us 648ns Erreur Moyenne : 0.0122461
Test du jeu exécuté en 24ms 795us 136ns Erreur : 1 (1.2987 %)
# Mini-Batch de taille 20
Temps mis pour entrainer: 3s 961ms 238us 528ns Erreur Moyenne : 0.0290012
Test du jeu exécuté en 26ms 318us 848ns Erreur : 6 (7.79221 %)
# Mini-Batch de taille 15
Temps mis pour entrainer: 3s 35ms 355us 136ns Erreur Moyenne : 0.0327774
Test du jeu exécuté en 30ms 587us 136ns Erreur : 3 (3.8961 %)
```

Listing 4.17 – Résultats obtenus avec la fonction d'activaton **Tangente Hyperbolique** divers quantités d'itérations avec la Descente de Gradient Stochastique.

4.6 Utilisation de caractéristiques des caractères au lieu des pixels

Une seconde approche pour reconnaître les caractères des plaques d'immatriculation françaises est envisageable. Au lieu d'utiliser les pixels de l'image, des caractéristiques des caractères des plaques seront données comme entrées au réseau de neurones afin de déterminer à quel caractère l'entrée correspond. Les caractéristiques choisies sont :

- La surface du caractère,
- Le périmètre du contour du caractère,
- Le nombre de trous contenus dans le caractère (par exemple, la lettre "B" possède 2 trous),
- Le centre de gravité,
- Les sept moments invariants de Hu.

Ces caractéristiques sont non-exhaustives, il ne s'agit, dans le cas présent, que d'un test simple afin de comparer des performances de deux réseaux de neurones ayant respectivement comme entrées des pixels et des caractéristiques. Afin d'extraire les caractéristiques susmentionnées, les caractères sont au préalables détectés, extraits et réduits à une image de 28×28 pixels.

4.6.1 Les moments de Hu

Les moments en statistiques

En statistiques, selon [KKN10b], le moment E d'ordre n (où $n = 1, 2, \dots$) d'une variable aléatoire X d'une fonction de probabilité f s'écrit :

$$E(X^n) = \sum_j x_j^n f(x_j) \text{ dans le cas où } f(X) \text{ est discrète et :}$$
$$E(X^n) = \int_{-\infty}^{+\infty} x^n f(x)\, dx \text{ dans le cas où } f(X) \text{ est continue.}$$

Tandis que le moment centré d'ordre n (où $n = 1, 2, \dots$) de X est :

$$E([X-\mu]^n) = \sum_j [x_j - \mu]^n f(x_j) \text{ dans le cas où } f(X) \text{ est discrète et :}$$
$$E([X-\mu]^n) = \int_{-\infty}^{+\infty} [x-\mu]^n f(x)\, dx \text{ dans le cas où } f(X) \text{ est continue.}$$

Ceci inclut le moment de premier ordre de X ($n = 1$) appelé *espérance* et noté μ :

$$\mu = E(X)$$

Enfin, le moment centré-réduit est un moment où la moyenne est retirée à la variable aléatoire X qui se voit ensuite divisée par son écart-type σ :

$$E\left(\left[\frac{X-\mu}{\sigma}\right]^n\right) = \sum_j \left[\frac{x_j - \mu}{\sigma}\right]^n f(x_j) \text{ dans le cas où } f(X) \text{ est discrète et :}$$
$$E\left(\left[\frac{X-\mu}{\sigma}\right]^n\right) = \int_{-\infty}^{+\infty} \left[\frac{x-\mu}{\sigma}\right]^n f(x)\, dx \text{ dans le cas où } f(X) \text{ est continue}$$

Parmi les moments les plus fréquents, la variance est le moment centré d'ordre deux ($n = 2$) et se note $V(X)$:

$$V(X) = E([X - \mu]^2)$$

Enfin, un dernier moment, toutefois moins courant, est le *coefficient d'asymétrie* γ, le moment centré-réduit d'ordre trois ($n = 3$) :

$$\gamma = E\left(\left[\frac{X - \mu}{\sigma}\right]^3\right)$$

Les moments d'une image

En traitement d'images, la notion de moment est également applicable à des fins diverses. Celle qui nous intéresse est la reconnaissance de forme. Tout comme en statistiques ou en physique, la notion de moment appliqué à l'image permet d'extraire des caractéristiques du contenu de l'image. Une image est une fonction de deux variables : tout d'abord x, sa dimension horizontale ainsi que y, sa dimension verticale. Pour une fonction à deux dimensions continues, le scientifique Hu a énoncé un moment m_{pq} d'ordre $(p + q)$ de la fonction de probabilité f dépendant cette fois-ci des variables aléatoires X et Y comme étant [Shu02] :

$$m_{pq} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} x^p \; y^q \; f(x, y) \; dx \; dy$$

Dans le cas d'une fonction discrète telle qu'une image, la relation devient :

$$m_{pq} = \sum_{x=1}^{M} \sum_{y=1}^{N} x^p \; y^q \; P(x, y) \tag{4.4}$$

Où :
- M désigne le nombre de colonnes de l'image,
- N désigne le nombre de lignes de l'image,
- $P(x, y)$ désigne la valeur du pixel aux coordonnées (x, y).

Grâce à la formule (4.4), la "masse totale" de l'image peut être calculée, il s'agit simplement de la somme de la valeur de tous les pixels. Les paramètres prennent les valeurs $p = 0$ et $q = 0$ [Shu02] :

$$m_{00} = \sum_{i=1}^{M} \sum_{j=1}^{N} x_i^0 \; y_j^0 \; P(x_i, y_j) = \sum_{i=1}^{M} \sum_{j=1}^{N} \; P(x_i, y_j) \tag{4.5}$$

Le moment du premier ordre des variables x et y avec les paramètres ayant respectivement les valeurs $p = 1$, $q = 0$ et $p = 0$, $q = 1$ sont utilisé pour calculer le centre de gravité d'une image *binaire* conjointement avec l'équation (4.5) :

$$x_G = \frac{m_{10}}{m_{00}} \qquad\qquad y_G = \frac{m_{01}}{m_{00}}$$

Avec :

$$m_{10} = \sum_{i=1}^{M} \sum_{j=1}^{N} x_i^1 \; y_j^0 \; P(x_i, y_j) = \sum_{i=1}^{M} \sum_{j=1}^{N} x_i \; P(x_i, y_j)$$

$$m_{01} = \sum_{i=1}^{M} \sum_{j=1}^{N} x_i^0 \; y_j^1 \; P(x_i, y_j) = \sum_{i=1}^{M} \sum_{j=1}^{N} y_j \; P(x_i, y_j)$$

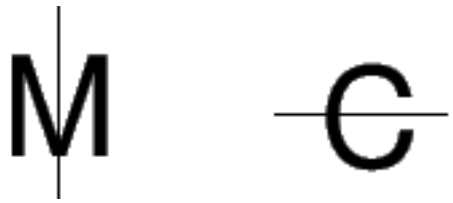


FIGURE 4.6 – Deux lettres binarisées. La lettre de gauche est symmétrique selon l'axe vertical tandis que celle de droite est symétrique selon l'axe horizontal. L'exemple est issu de la référence [Shu02].

Lettre	η_{11}	η_{20}	η_{02}	η_{21}	η_{12}	η_{30}	η_{03}
M	0	+	+	-	0	0	-
C	0	+	+	0	+	+	0

TABLE 4.2 – Les signes des différents moments pour les caractère de la Figure 4.6.

HU a aussi défini les *moments centrés* en deux dimensions de façon analogue aux moments centrés en statistiques [Shu02]. Centrer un moment rend ce dernier insensible à toute translation.

$$\mu_{pq} = \sum_{i=1}^{M}\sum_{j=1}^{N}(x_i - x_G)^p \ (y_j - y_G)^q \ P(x_i, y_j) \tag{4.6}$$

Ensuite, HU a également rendu le moment de l'image *invariant* à l'échelle en divisant le moment centré par la masse totale de l'image exposant un facteur γ [Shu02]. En reprenant l'équation (4.6), on obtient le moment *normalisé*, qui s'apparente à un moment centré-réduit :

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}} \tag{4.7}$$

Avec :

$$\gamma = \frac{p+q}{2} + 1 \qquad\qquad \forall \ (p+q) \geq 2$$

La Figure 4.6 présente deux exemples de lettres binarisées et symmétrique selon l'axe vertical, pour celle de gauche, tandis que celle de droite est symmétrique selon l'axe horizontal. La Table 4.2 présente les valeurs des moments pour les exemples de la Figure 4.6.

Propriétés de symétrie

Le moment statistique centré-réduit de troisième ordre décrit le coefficient d'asymétrie d'une distribution d'une variable aléatoire. Ce coefficient est :

- Nul si la distribution de la variable aléatoire est parfaitement symétrique.
- Positif si la distribution de la variable aléatoire est étalée sur la droite.
- Négatif si la distribution de la variable aléatoire est étalée sur la gauche.

La Figure 4.7, présente deux exemples de la loi de distribution Beta de paramètres α et β. À gauche, la loi possède les paramètres $\alpha = 2$ et $\beta = 5$ et son coefficient d'asymétrie est négatif. Tandis que sur la droite, $\alpha = 5$ et $\beta = 2$ et son un coefficient d'asymétrie est positif.

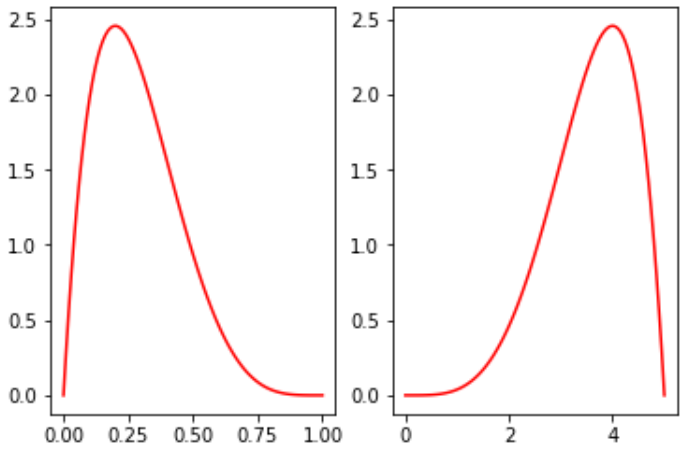


FIGURE 4.7 – La loi Beta, fonction de deux paramètres α et β. À gauche, un coefficient d'asymétrie négatif et à droite un coefficient d'asymétrie positif.

Dans le cas d'une image, les moments centrés du troisième ordre décrivant l'asymétrie de l'image sont μ_{30} dans le cas de l'axe horizontal représenté par x et μ_{03} dans le cas de l'axe vertical, symbolisé par y [Shu02]. La symétrie d'un objet dans une image *binaire* est relative à son centre de gravité, ce qui justifie l'utilisation de moments centrés. Sept moments normalisés sont utilisés dans l'étude des caractéristiques de caractères, soumises comme des images binaires : η_{11}, η_{20}, η_{02}, η_{21}, η_{12}, η_{30}, η_{03}.

En résumé, les différents rôles des moments sont :

- $\eta_{11} = 0$ si la forme présente une symétrie quelconque, c'est-à-dire une symétrie selon x ou selon y.
- Pour des formes symétriques selon l'axe y, $\eta_{12} = 0$ et $\eta_{30} = 0$.
- Pour des formes symétriques selon l'axe x, $\eta_{12} \geq 0$ et $\eta_{03} = 0$.
- Pour des formes asymétriques selon l'axe x, $\eta_{20} \geq 0$, $\eta_{21} \leq 0$ et $\eta_{20} \geq 0$.

Les moments invariants de Hu

Pour l'instant, les moments utilisés sont invariants à l'échelle ainsi qu'à la translation. HU a reformulé les moments normalisés afin de rendre ces nouveaux moments invariants à la *rotation*. Les différents moments de HU, notés ϕ_i avec i l'indexe du moment, sont [Shu02] :

$$
\begin{aligned}
\phi_1 &= \eta_{20} + \eta_{02} \\
\phi_2 &= (\eta_{20} - \eta_{02)}^2 + 4\eta_{11}^2 \\
\phi_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
\phi_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
\phi_5 &= (\eta_{30} - 3\eta_{12})\ (\eta_{30} + \eta_{12})\ \left[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2\right] \ + \\
&\quad (3\eta_{21} - \eta_{03})\ (\eta_{21} + \eta_{03})\ \left[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2\right] \\
\phi_6 &= (\eta_{20} - \eta_{02})\ \left[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2 + 4\eta_{11}\ (\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})\right] \\
\phi_7 &= (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})\ \left[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2\right] \\
&\quad + (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})\ \left[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2\right]
\end{aligned}
$$

4.6.2 Analyse en Composantes Principales des caractéristiques des caractères

Réalisation

Les caractéristiques sont collectées *sur le jeu d'apprentissage*. En effet, il est supposé que seul le jeu d'apprentissage est connu et c'est le réseau de neurones qui doit se généraliser. La capacité d'un réseau neuronale à se généraliser se mesure sur le jeu de test. Le rôle de l'Analyse en Composantes Principales (ACP) dans notre cas est de pouvoir identifier les variables, c'est-à-dire nos caractéristiques, afin de voir celles qui apportent une certaine redondance d'information dont le réseau de neurones pourrait se passer. L'objectif recherché est d'optimiser notre classifieur en allégeant son modèle afin de diminuer les temps de calcul sans perte importante d'efficacité. Un rappel sur l'ACP est présent en Annexe E.

L'ACP a été réalisé avec le logiciel de statistiques **R** et ainsi que des librairies **FactoMineR** et **factoextra** disponibles sur le CRAN [1]. La librairie **FactoMineR** possède une fonction **PCA** permettant de réaliser l'ACP en elle-même tandis que la librairie **factoextra** permet de dessiner des graphiques personnalisables. Le code permettant de faire une ACP sur **R** et d'enregistrer les graphiques ainsi que le contenu des variables est présent dans le Listing 4.18.

```
# Charger librairie qui va permettre de faire l'ACP
library(FactoMineR)
# Charger librairie permettant de faire des graphiques personnalisables
library(factoextra)

# Lecture du fichier contenant les carac. du jeu de d'apprentissage
donneesCaracteristiques <- read.table("/home/julien/Documents/PlaquesNG/ACP/
    plaques_carac_train_acp.csv", sep=",", header = TRUE)

# Chaque ligne possède le nom du fichier comme "identifiant"
rownames(donneesCaracteristiques) <- donneesCaracteristiques$Fichier

# Enlever la colonne nom du fichier car inutile
donneesCaracteristiques <- donneesCaracteristiques[,-1]

# Faire l'ACP
pcaCaracteres <- PCA(donneesCaracteristiques, scale.unit = TRUE, ncp = 5)

# Afficher le graph. des individus
fviz_pca_ind(pcaCaracteres, col.ind = "cos2",
    gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
    repel = TRUE ) # Evite le chevauchement de texte

# Enregistrer le graph. des individus dans un fichier PDF
dev.print(pdf, "/home/julien/Documents/PlaquesNG/ACP/
    pca_train_caracteres_individus.pdf")

# Afficher graph. des variables
fviz_pca_var(pcaCaracteres, col.var = "contrib",
alpha.var="cos2",
gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
repel = TRUE ) # Evite le chevauchement de texte

# Enregistrer le graph. des variables dans un fichier PDF
dev.print(pdf, "/home/julien/Documents/PlaquesNG/ACP/
    pca_train_caracteres_variables.pdf")
```

1. Le "Comprehensive R Archive Network", un dépôt de packages **R**.

```
34
# Sauvegarder les résultats dans un fichier CSV
36 write.infile (pcaCaracteres, "/home/julien/Documents/PlaquesNG/pca_train.csv",
    sep = ";")
```

Listing 4.18 – Le code R permettant de faire lÁCP.

Analyse du graphique des individus

Les graphiques fournis par l'ACP sont visibles sur les Figures 4.8 et 4.9 et se nomment respectivement graphique des individus (ou observations) et graphique des variables. Tout d'abord, le graphique des individus représente les observations dans un plan où les deux dimensions rassemblent une *dispersion maximale*. Le Listing 4.19 présente les valeurs propres de l'ACP qui permet de choisir le nombre de composantes principales à prendre en considération. En choisissant les quatre premières composantes, 86% de la variance est expliquée.

```
eig
2      eigenvalue  percentage of variance  cumulative percentage of variance
comp 1   5.3163       44.3021                44.3021
4 comp 2   2.3081       19.2345                63.5366
comp 3   1.5961       13.301                 76.8376
6 comp 4   1.216        10.1332                86.9708
comp 5   0.6671       5.5591                 92.5299
8 comp 6   0.3924       3.2697                 95.7996
comp 7   0.2054       1.7114                 97.511
10 comp 8   0.1386       1.1547                 98.6657
comp 9   0.0852       0.7104                 99.3761
12 comp 10  0.0473       0.394                  99.77
comp 11  0.0206       0.1713                 99.9413
14 comp 12  0.007        0.0587                 100
```

Listing 4.19 – Les valeurs propres des différentes composantes principales.

Le graphique des individus de la Figure 4.8 regroupe les caractères présentant des similitudes. Par exemple, les caractères "8" et "B" sont très proches, car, intuitivement, ils se ressemblent. Pareil pour le "0", le "D" et le "Q", le "W" et le "M" ou encore le "2", le "S" et le "5". Cependant, certaines relations, à priori inexistantes, apparaissent. Par exemple, le "R" entre le "6" et le "9", le "J"et le "L" sont apparemment deux caractères très singuliers. D'ailleurs, le "L" a la plus grande contribution aux deux premières composantes principales, comme le montre le Listing 4.20. La contribution du "J" est également notable, mais pas exceptionnelle.

```
ind
2 contrib
                  Dim.1     Dim.2    Dim.3    Dim.4    Dim.5
4 L_Train_N.pgm  45.4623  13.8946  27.337   0.1945   5.047
J_Train_N.pgm  27.3569  2.4267   47.4079  4.9919   1.568
6 B_Train_N.pgm  3.3379   11.7635  0.0103   0.1897   1.9145
8_Train_N.pgm  3.2643   10.5653  0.1266   0.2757   1.1707
8 0_Train_N.pgm  2.6621   6.1389   0.1135   0.3218   0.076
Q_Train_N.pgm  2.5509   4.9688   0.0672   0.5738   0.0295
10 D_Train_N.pgm  2.3932   6.1208   0.0082   0.094    0.2649
C_Train_N.pgm  1.9197   4.1064   3.8726   0.1881   64.4185
12 R_Train_N.pgm  1.7666   0.9256   0.1082   0.7612   0.0131
9_Train_N.pgm  1.5273   0.352    0.0049   0.3918   0.0239
14 ...
```

Listing 4.20 – Les contributions des individus aux cinq premières composantes principales.

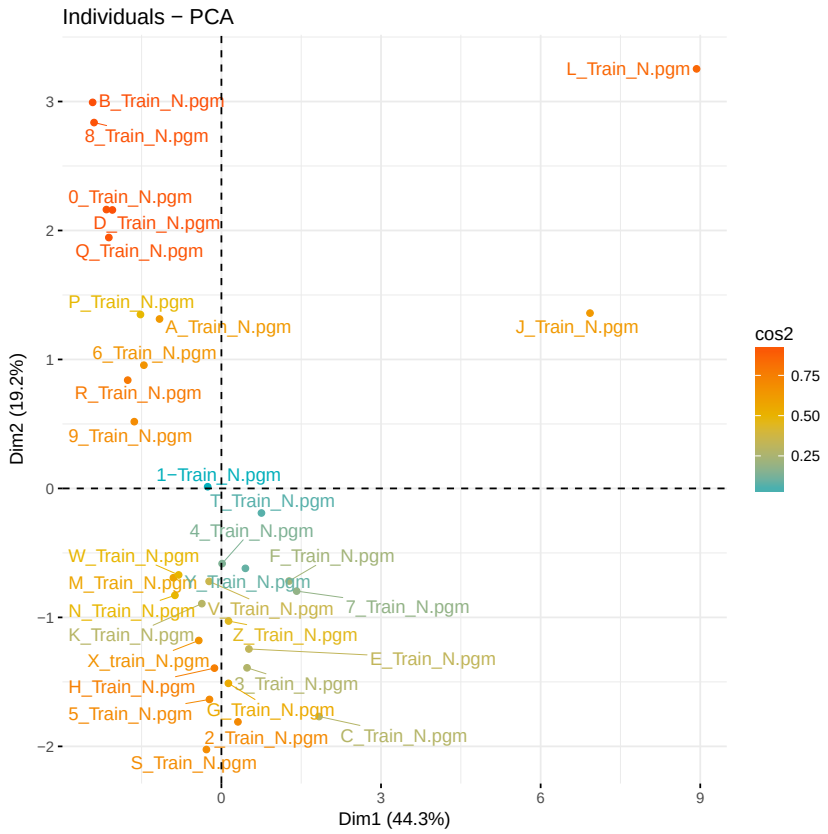


FIGURE 4.8 – Le graphique des individus de l'ACP sur les caractéristiques de plaques d'immatriculation.

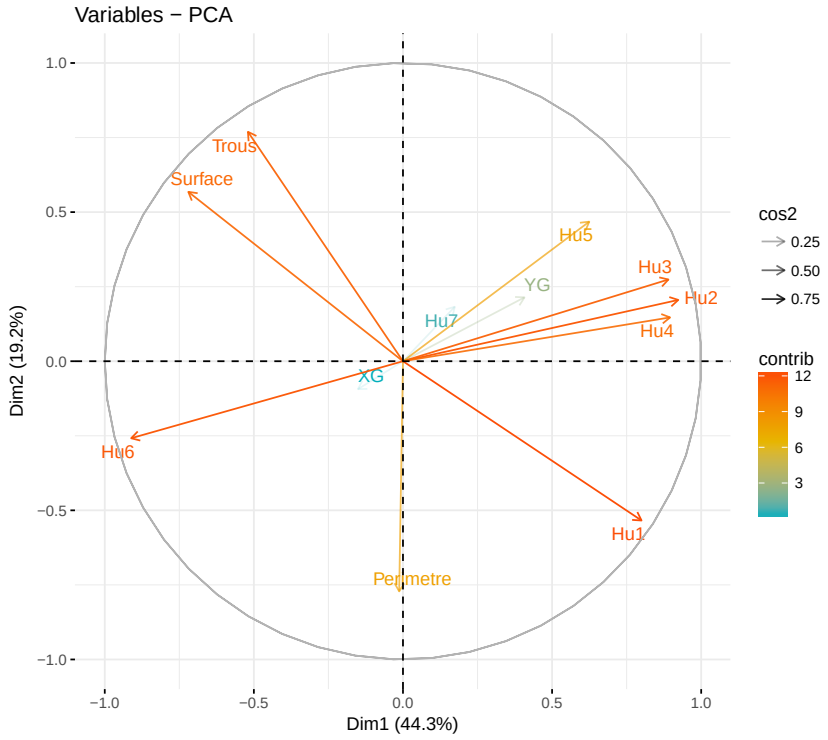


FIGURE 4.9 – Le graphique des variables de l'ACP sur les caractéristiques de plaques d'immatriculation.

Analyse du graphique des variables

Le graphique présent sur la Figure 4.9 est appelé *graphique des variables* et représente un cercle de corrélation. Sur cette figure, les variables **Hu7**, **XG** et **YG** présentent un faible cos2 et par conséquent elles sont mal représentées et aucun jugement ne leur sera appliqué.

En revanche, les autres variables semblent être suffisamment bien représentées que pour être interprétées. Deux variables présentant un angle aigu entre leur flèche sont d'autant plus correlées que l'angle est petit. On peut en déduire que les variables **Hu2**, **Hu3** et **Hu4** sont fortement corrélées entre elles et que le modèle peut être simplifié en ne conservant que **Hu2** car cette dernière est la variable se trouvant au milieu des deux autres et possède la plus grande contribution (cf. Listing 4.21) et c'est également celle qui est la mieux représentée. Le même raisonnement peut être appliqué avec les variables **Surface** et **Trous**. Par ailleurs, on remarque que les variables **Surface** et **Hu1** sont corrélées négativement, ce qui est aussi valable pour **Hu6** et **Hu3**.

Le Listing 4.21 présente les contributions des variables aux cinq axes principaux. En ce qui concerne le premier axe principal, les variables qui y contribuent le plus sont **Hu2**, **Hu6**, **Hu4**, **Hu3** et **Hu1**. Les autres variables, exceptée le **Perimetre**, ont tout de même une certaine contribution, mais moins importante. Concernant le second axe principal, ce dernier explique sa variance notamment grâce au **Périmètre** et aux nombres de **Trous**.

```
var
contrib
            Dim.1     Dim.2     Dim.3     Dim.4     Dim.5
Hu2       16.0352    1.8299    2.0173    1.9443    1.7965
Hu6       15.6288    2.889     0.3648    0.9606    8.8146
Hu4       15.0933    0.935     0.0003    0.6087   20.7491
Hu3       14.9065    3.2529    1.5518    5.0072    1.4436
Hu1       12.039    12.358     0.0363    0.1188    2.7052
Surface    9.7462   13.9755    0.1954    5.3982    0.5551
Hu5        7.342     9.4902    0.0376    3.5407   41.8583
Trous      5.0839   25.6397    0.0011    0.2209    5.9469
YG         3.124     1.988     0.0009   58.5738    0.3046
Hu7        0.5683    1.4455   43.9236    1.096     9.1857
XG         0.4297    0.3803   48.2344    1.3361    4.952
Perimetre  0.003    25.8158    3.6364   21.1948    1.6884
```

Listing 4.21 – Les contributions des variables aux cinq premières composantes principales.

Pour conclure sur l'ACP, peu de variables sont à supprimer dans ce modèle. Tout d'abord, la variable **Hu2** sera conservée au détriment de **Hu3** et **Hu4**. Le premier axe principal trouve la majorité de sa variance dans les moments 2,6,4,3,1 de Hu, ainsi que dans la **Surface**. De même, la variable **Périmètre** sera préférée à la variable **Trous** car elle apporte un rien de variance en plus au second axe principal.

4.6.3 Entraînement d'un réseau de neurones sur les caractéristiques sans ACP

Le réseau de neurones ayant les meilleurs résultats obtenus avec la recheche de paramètres précédemment utilisée est présenté dans le Listing 4.22. Les paramètres retenus sont : $N_h = 50$ neurones cachés, une couche, un taux d'apprentissage $\eta = 0.1$, la Descente de Gradient Stochastique et 300 itérations, la fonction Logistique pour le réseau ainsi que la fonction **Soft-**

maxutilisée sur la couche de sortie. Le réseau se trompe alors sur 15 parmi les 77 exemples du jeu de test, soit une précision atteinte de 79.22%.

```
Temps mis pour entrainer: 1s 645ms 232us 896ns Erreur Moyenne : 0.347694
Test du jeu exécuté en 9ms 313us 280ns Erreur : 15 (19.4805 %)
```

Listing 4.22 – Résultats obtenus avec un réseau de neurones où les caractéristiques des plaques sont ses entrées.

4.6.4 Entraînement d'un réseau de neurones sur les caractéristiques avec ACP

Le rôle de l'ACP étant d'éliminer les variables non-discriminantes de la liste des caractéristiques établie afin d'effectuer une rétro-propagation tout aussi performante avec moins de calculs. Á l'issue de cette dernière, le modèle est *allégé*. Par conséquent, *le modèle n'est pas supposé être plus performant mais fournir les mêmes résultats avec moins d'entrées*. Pour rappel, les variables d'entrée éliminées du modèle sont : **Hu3**, **Hu4**, **Trous**. Le modèle passe alors de 12 à 9 variables. Le Listing 4.23 montre les résultats du réseau de neurones le plus performant ayant été obtenu avec la configuration suivante : $N_h = 20$ neurones cachés, 1 couche cachée, un taux d'apprentissage $\eta = 0.0001$, 200 itérations et une Descente de Gradient Mini-Batch avec un batch d'une taille de 20 exemples, suivit d'une seconde exécution avec ces mêmes paramètres.

```
# 1er apprentissage
Temps mis pour entrainer: 88ms 843us 264ns Erreur Moyenne : 0.993318
Test du jeu exécuté en 1ms 7us 360ns Erreur : 35 (45.4545 %)
```

Listing 4.23 – Résultats obtenus avec un réseau de neurones où les caractéristiques des plaques sont ses entrées.

4.7 Conclusions

La Table 4.3 contient les meilleurs résultats parmi tous les tests effectués dans ce chapitre. Tout d'abord, le meilleur résultat obtenu parmi tous est la combinaison de : pixels, d'entropie croisée, de la fonction d'activation **Logistique** conjointement avec la fonction **Softmax**sur la couche de sortie, d'une descente de gradient Mini-Batch d'une taille de 33 exemples après 200 itérations, ce dernier ne laissait qu'une seule erreur après analyse du jeu de test. En général, l'utlisation de la fonction **Softmax**avec la fonction **Logistique**et l'entropie croisée donne de bons résultats, bien que ceux donnés par la **Tangente Hyperbolique**sont très proches. De plus, dans la majorité des cas,la Descente de Gradient Mini-Batch fournit également de bons résultats, la taille du Mini-Batch étant variable. Dans des grands jeux de données, la taille du batch n'est pas un paramètre fixé du réseau de neurone et n'est alors pas sujet à l'expérimentation [Nie20]. Par ailleurs, le taux d'apprentissage est établi est propre à chaque jeu de données ainsi qu'aux autres paramètres du réseaux, tout comme le nombre de neurones et le nombre d'itérations. Enfin, l'erreur moyenne doit être analysée en fonction des paramètres et des données. Une erreur moyenne pour un réseau ayant été entraîné avec une fonction d'activation et d'erreur spécifique n'a pas la même signification avec une autre fonction d'activation ou d'erreur.

En outre, les résultats obtenus par les réseaux de neurones analysant les pixels plutôt que les caractéristiques des caractères sont nettement meilleurs. De plus, l'ACP effectuée sur ces caractéristiques n'a pas avoir les effets attendus car les résultats avec le modèle allégé sont très

peu performants. À mes yeux, les mauvaises performances du modèle allégé par l'ACP est dû au faible nombre de variables d'entrées, même si variables élaguées du modèle n'apportent que peu de variance.

Entrées	**Fonction erreur**	**Fonction activation**	**Neurones cachés**	**Taux Apprentissage**	**Descente Gradient**	**Softmax ?**	**Erreur moyenne** (%)	**Précision** (/77)	**Itérations**	**Temps** (sec)
Pixels	Quadratique	Logistique	40	0.01	Stochastique	Non	0.00740723	2	600	32.656
Pixels	Quadratique	Tanh	40	0.1	Full-Batch	Non	0.025455	5	450	21.491
Pixels	Entropie Croisée	Logistique	20	0.1	Full-Batch	Non	0.025455	4	600	7.576
Pixels	Entropie Croisée	Logistique	20	0.1	Mini-Batch (33)	Oui	0.0122461	1	200	6.962
Caractéristiques (12)	Entropie Croisée	Logistique	20	0.0001	Mini-Batch (20)	Oui	0.347694	16	300	1.645
Caractéristiques (9)	Entropie Croisée	Logistique	20	0.0001	Mini-Batch (20)	Oui	0.993318	35	200	0.088

TABLE 4.3 – Regroupement des meilleurs résultats obtenus pour chacun des tests exécutés.

Chapitre 5

Comparaison de différentes librairies de machine learning

Dans ce chapitre, la librairie **NeuroLib**, développée dans le cadre de ce stage, sera comparée avec des autres librairies de réseaux de neurones : **Keras**,**OpenCV**et **Mxnet**. Ces trois utilisées en Python version 3.5. Les bibliothèques **Keras** et **Mxnet** ont été choisies de part leur popularité et leur facilité d'utilisation. Quant à **OpenCV**, il s'agit principalement d'une librairie de traitement d'images qui dispose d'un module de machine learning, deux sujets très présents dans ce stage.

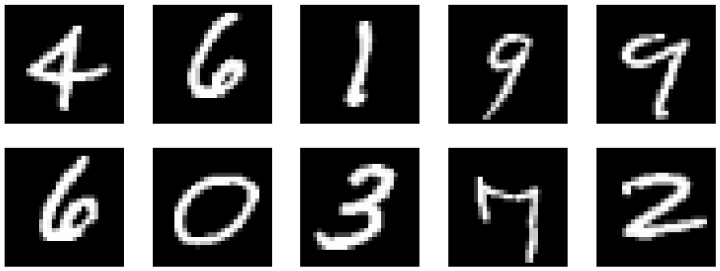


Figure 5.1 – Exemples des données MNIST.

Cette comparaison se fera sur la base de données de caractères manuscrits *MNIST* élaborée par Yann Lecun, Corinna Cortes et Christopher Burges. Cette base de données dispose de 60 000 caractères dans le jeu d'apprentissage et 10 000 caractères dans le jeu de test. Chacun des caractères a été normalisé à une taille de 28×28 et centré au milieu de l'image [LCB98]. Un aperçu des caractères présents dans la base de données est visible sur la Figure 5.1.

La comparaison des libraires a plusieurs objectifs. Le premier est de comparer la facilité d'utilisation, même dans un langage aussi haut niveau que le Python. Le second est d'identifier la librairie la plus complète, celle permettant de paramétrer au degré le plus fin un réseau de neurones et son apprentissage. Il est également important de savoir quelles sont les informations récupérables dans les librairies, pour une analyse ultérieure par exemple. Grâce à toutes ces informations, une liste des améliorations pouvant être apportée à **NeuroLib** pourra alors être établie.

Les modèles entraînés dans ce chapitre seront tous les trois identiques. Les paramètres ont été choisis de façon à ne pas avoir un modèle trop complexe et difficile à entraîner et fournissant

des résultats acceptables.

5.1 Keras

5.1.1 Présentation de Keras

Keras[1] est une librairie de réseaux de neurones open-source haut-niveau, écrite en Python par François Chollet. En réalité, elle interface une librairie au choix, qui doit être préalablement installée. La première bibliothèque est **TensorFlow**[2], développée par Google, est celle choisie par défaut. Deux autres librairies alternatives sont également supportées : **CNTK**[3] développée librement par Microsoft ou **Theano**[4] crée par l'Université de Montréal, également open-source. La librairie interfacée, dans notre cas, ou *backend*, sera **TensorFlow**. L'idée de **Keras** est de se concentrer sur une mise en œuvre rapide et aisée d'un réseau de neurones quelconque grâce à son API user-friendly. De plus, c'est une bibliothèque très complète en fonctionnalité et en documentation[5]. Elle supporte les réseaux de neurones : feed-forward, récurrents ou un mélange des deux. En outre, si le backend est installé avec le support du GPU, alors **Keras** supportera également l'exécution sur un GPU.

Keras fournit deux façons de définir un réseau de neurones, aussi appelé *modèle* dans sa terminologie. Le premier est un modèle "**Sequential**", qui empile les couches. La propagation en avant se fait séquentiellement, c'est-à-dire d'une couche à la suivante. Le second est un modèle provenant de l'API **Functional** et permet de définir des modèles plus complexes disposant, par exemple, de plusieurs entrées dans différentes couches du réseau ou encore de réseaux partageant des couches.

5.1.2 Application

Importation des librairies

Afin de faire fonctionner un réseau de neurones **Keras**, plusieurs importations sont nécessaires. Les modèles séquentiels de cette librairie se trouvent dans le package **keras.models**. De plus, les couches complètement connectées portent le nom de **Dense** et se trouvent dans **keras.layers**. La librairie possède également des packages permettant de charger directement des bases de données couramment utilisées comme références pour comparer les performances des classifieurs. Parmi ces banques de données, il y a MNIST qui se trouve dans le package **keras.datasets**.

Par ailleurs, **numpy** est une librairie mathématique Python destinée à des usages scientifiques. La fonctionnalité utilisée dans notre cas sera les tableaux multi-dimensionnels. Enfin, **time** est une librairie standard Python permettant d'utiliser des fonctions en rapport avec les dates et le temps. Elle sera utilisée afin d'effectuer des mesures de temps d'exécution.

```
1 from keras.models import Sequential
  from keras.layers import Dense
3 from keras.datasets import mnist
  from keras import initializers, optimizers
5
```

1. Le site officiel de **Keras** : `https://keras.io/`
2. Site officiel de **TensorFlow** : https ://www.tensorflow.org/
3. **The Microsoft Cognitive Toolkit** : `https://docs.microsoft.com/en-us/cognitive-toolkit/`
4. Site officiel de **Theano** : `http://deeplearning.net/software/theano/`
5. Voir le site officiel : `https://keras.io/`

```
import numpy as np
import time
```

Listing 5.1 – L'importation des packages nécessaire à Keras permettant de créer un réseau de neurones.

Importer les données MNIST

La première ligne du Listing 5.2 charge les images MNIST d'apprentissage (60 000) et de test (10 000) respectivement dans les variables **x_train x_test**. Si la base de donnée MNIST n'est pas encore présente sur le disque, cette fonction la télécharge également du cloud d'Amazon. Les images sont contenues dans des tableaux tridimensionnels de dimensions 60 000 × 28 × 28, où 28 × 28 sont la hauteur et la largeur de l'image. Pour qu'elles puissent être analysées par un réseau de neurones, les données doivent être converties en un vecteur à deux dimensions de 60 000 × 784 en vue d'être fournie comme entrées au réseau sous forme de vecteur. Cette étape est réalisée grâce à la fonction **reshape**. Les sorties souhaitées correspondantes, sont contenues dans les variables **y_train** pour le jeu d'apprentissage et **y_test** pour le jeu de test.

```
#%% Importer les données MNIST
(xtrain, ytrain), (xtest, ytest) = mnist.load_data()

## Applatir les entrées: 28x28 -> 1x784
dimData = np.prod(xtrain.shape[1:])
train_data = xtrain.reshape(xtrain.shape[0], dimData)
test_data = xtest.reshape(xtest.shape[0], dimData)

# Récupérer le nombre de classes possibles: c'est-à-dire 10
classes = np.unique(ytrain)
nClasses = len(classes)
```

Listing 5.2 – L'importation de la base de données MNIST sur laquelle va s'effectuer les tests.

Création du réseau de neurones

Keras fournit également diverses possibilités de personnalisation en terme d'initialisation, contenues dans la classe **initializers**. Souvent, les poids souhaités suivent une distribution gaussienne de moyenne nulle et d'écart-type très faible, 10^{-3} par exemple et le biais sont initialisés à 1.

L'algorithme d'apprentissage est symbolisé par la classe **optimizers** dans **Keras**. Dans le cas présent, nous utiliserons la Descente de Gradient Stochastique, mais d'autres algorithmes sont disponibles. La Descente de Gradient fonctionne avec un taux d'apprentissage qu'il faut préciser. Ce dernier correspond au paramètre **lr** qui signifie "learning rate" en anglais.

Une fois les données chargées, le réseau de neurones est créé dans le Listing 5.3. Celui-ci est un modèle dans la terminologie de **Keras** et sera séquentiel. La fonction **add** permet d'ajouter une couche complètement connectée, représentée par la classe **Dense**. Pour ajouter une couche, il faut spécifier la quantité de neurones en précisant le paramètre **units**, **activation** pour la fonction d'activation, la forme dans laquelle les entrées sont fournies à la couche en cours de création via le paramètre **input_shape**. Les paramètres optionnels sont **kernel_initialize** et **bias_initializer** correspondent à des objets **initializers** et laissent le choix de la méthode

d'initialisations des poids et des biais de cette couche. Pour rappel, dans la librairie **NeuroLib** dont les poids sont initialisés par une distribution gaussienne de moyenne nulle et d'écart-type $\sigma = 10^{-3}$ et dont les biais sont unitaires. Les fonctions correspondantes en Python sont **RandomNormal** et **Ones** de la classe **initializers**.

Ainsi, le modèle créé comporte deux couches : une première utilisant la fonction **Logistique**, composée de 40 neurones et de 784 entrées et une seconde utilisant la fonction Softmax, comportant autant de neurones que de sorties possibles, autrement dit 10 et 40 entrées.

Une fois le modèle créé, celui-ci doit être compilé pour pouvoir être utilisé. La méthode **compile** de la classe **model** requiert l'algorithme de Descente de Gradient et la fonction d'erreur. Le nom de l'entropie croisée utilisée dans la classification multi-classe est **sparse_ categorical_ crossentropy**. Le dernier paramètre de la fonction **compile** est **metrics**. Celui-ci permet d'afficher d'autres informations lors de l'entraînement du modèle. Dans le cas présent, un paramètre n'ayant pas encore été mentionné est le taux de bonne reconnaissance, symbolisé par *accuracy*, des exemples du jeu de test.

```
#%% Créer les Kernel et Biais Initializers et Optimizers
# Initializers des poids: suivent une loi normale de moyenne=0, sigma=1E-3
weights_init = initializers.RandomNormal(mean=0.0, stddev=0.001)
# Initializers des biais: tous égaux à 1
bias_init = initializers.Ones()

# Optimizer: Stochastic Gradient Descent (SGD)
sgd = optimizers.SGD(lr=0.01)

# Instancier le modèle1
model = Sequential()
# Première couche
model.add(Dense(units=40, activation='sigmoid', input_shape=(dimData,),
kernel_initializer=weights_init,
bias_initializer=bias_init))
# Seconde couche
model.add(Dense(nClasses, activation='softmax',
kernel_initializer=weights_init,
bias_initializer=bias_init))

# Compiler le modèle
model.compile(optimizer=sgd, loss='sparse_categorical_crossentropy', metrics=['
    accuracy'])
```

Listing 5.3 – Création du modèle.

Entraîner le modèle

Le code correspondant à l'apprentissage du réseau de neurones se trouve dans le Listing 5.4. Étant donné que la Descente de Gradient utilisée est la Stochastique, la taille du batch est à définir :

- Si le paramètre **batch_size** = 1 : alors l'algorithme utilisé le SGD classique,
- En revanche, si **batch_size** > 1, alors c'est l'algorithme de Descente de Gradient Mini-Batch qui est utilisée. Cette dernière est présentée en Annexe B.

De même, le nombre d'itérations à faire doit également être défini grâce au paramètre **epochs**. La fonction **fit** permet l'apprentissage même du réseau de neurones. En plus de prendre le jeu d'apprentissage en paramètre, elle prend également le jeu de test ou de *validation* qui

permet de tester la performance du modèle. Le dernier paramètre à préciser est **verbose**. Ce dernier demande à **Keras** d'afficher des informations lors de la descente de gradient. Une fois l'apprentissage terminé, le code affiche en secondes le temps mis par l'algorithme.

```
#%% Entrainer le modèle
tailleMB = 256
nbIteration = 10

start = time.time() # Temps depuis epoch AVANT exécution
history = model.fit(train_data, ytrain, batch_size=tailleMB, epochs=nbIteration,
    verbose=1, validation_data=(test_data, ytest))
end = time.time() # Temps depuis epoch APRES exécution
# Affiche le temps mis par l'algorithme pour effectuer l'apprentissage
print("Temps écoulé: {} secondes".format(end - start))
```

Listing 5.4 – Apprentissage du réseau de neurones.

Résulats

La sortie du programme est dans le Listing 5.5. Nous pouvons aussi constater que **Keras** affiche énormément d'informations, et ce, à chaque itération grâce au paramètre **verbose** expliqué précédemment. On y retrouve, dans l'ordre :

- une barre de progression indiquant où le programme en est dans l'itération en cours,
- le temps mis pour faire l'itération,
- *loss* l'erreur moyenne de la couche de sortie pour le jeu d'apprentissage,
- *acc* : le taux de bonne classification du réseau sur le jeu d'apprentissage,
- *val_loss* : l'erreur moyenne de la couche de sortie pour le jeu de validation.
- *val_acc* : le taux de bonne classification du réseau sur le jeu de validation.

Toutes ces informations sont stockées dans un objet **History** du package **History** permettant une analyse ultérieure. Imaginons que le nombre d'itérations de la Descente de Gradient soit de 100 000. Si à un moment le taux de bonnes classifications du jeu de validation cesse de diminuer et augmente, il est possible de retrouver le nombre d'itérations donnant le meilleur taux.

Après entraînement, la valeur de la variable **val_acc** vaut 0.9154. C'est-à-dire que 91.54% des exemples de validation sont classifiés correctement.

```
Using TensorFlow backend.
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [==============================] - 4s 0us/step
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 1s 13us/step - loss: 1.8299 - acc
    : 0.5990 - val_loss: 1.4288 - val_acc: 0.8141
Epoch 2/10
60000/60000 [==============================] - 1s 12us/step - loss: 1.2000 - acc
    : 0.8458 - val_loss: 1.0431 - val_acc: 0.8702
Epoch 3/10
60000/60000 [==============================] - 1s 11us/step - loss: 0.8905 - acc
    : 0.8798 - val_loss: 0.8398 - val_acc: 0.8782
Epoch 4/10
60000/60000 [==============================] - 1s 11us/step - loss: 0.7162 - acc
    : 0.8941 - val_loss: 0.6628 - val_acc: 0.9003
Epoch 5/10
60000/60000 [==============================] - 1s 13us/step - loss: 0.6079 - acc
    : 0.9019 - val_loss: 0.6106 - val_acc: 0.8998
Epoch 6/10
```

```
60000/60000 [==============================] - 1s 11us/step - loss: 0.5366 - acc
    : 0.9058 - val_loss: 0.5286 - val_acc: 0.9096
Epoch 7/10
60000/60000 [==============================] - 1s 12us/step - loss: 0.4849 - acc
    : 0.9098 - val_loss: 0.4938 - val_acc: 0.9038
Epoch 8/10
60000/60000 [==============================] - 1s 12us/step - loss: 0.4466 - acc
    : 0.9131 - val_loss: 0.4348 - val_acc: 0.9122
Epoch 9/10
60000/60000 [==============================] - 1s 12us/step - loss: 0.4158 - acc
    : 0.9155 - val_loss: 0.4393 - val_acc: 0.9152
Epoch 10/10
60000/60000 [==============================] - 1s 12us/step - loss: 0.3930 - acc
    : 0.9181 - val_loss: 0.3912 - val_acc: 0.9154
Temps écoulé: 7.198914289474487 secondes
```

Listing 5.5 – Résultat de l'apprentissage d'un réseau de neurone **Keras** sur la base de données MNIST."

5.2 Mxnet

5.2.1 Présentation de Mxnet

Mxnet[6] est une bibliothèque open-source de deep learning développée et entretenue par la communauté Apache Software Foundation[7]. Écrite en C++, elle fournit également un support pour plusieurs languages : Python, R, Julia et Scala. En outre, un support multi-GPU et multi-CPU est installé par défaut. Contrairement à **Keras**, **Mxnet** n'a pas besoin d'autres importation pour fonctionner telle que **numpy** : elle fournit un support complet des tableaux multi-dimmensionnels et des fonctions mathématiques. Et comme **Keras**, elle possède déjà des datasets, tel que celui du MNIST, prêt à être testé.

De plus, un double support de programmation est fourni en Python : la programmation classique impérative au travers de son module **Gluon**[8] et la programmation symbolique. Sans entrer dans les détails, le module **Gluon** fournit le support classique des tableaux multi-dimmensionnels, des fonctionnalités mathématiques ainsi qu'une flexibilité importante. Le principe de la programmation symbolique est de créer un ensemble de variables et d'opérations à l'avance. Afin d'exécuter les opérations définies, des valeurs doivent être attribuées aux variables. À ce moment seulement, les opérations pourront être exécutées. Le mécanisme de programmation symbolique de **Mxnet** réalise des optimisations avant d'exécuter les opérations. Par exemple, il élimine les opérations non-nécessaires ou compresse les actions répétitives. Cette méthode tend à fournir des exécution plus rapide que ceux de la programmation impérative. Dans le cadre des expérimentations avec **Mxnet**, la programmation symbolique sera préférée.

5.2.2 Application

Importation des librairies

Le Listing 5.6 contient les importations nécessaires à **Mxnet**pour fonctionner. Comme précédemment mentionné, la librairie **Mxnet** se suffit à elle même. Cependant à des fin d'analyse

6. Site officiel du projet **Mxnet** : `https://mxnet.incubator.apache.org/`
7. `http://apache.org/`
8. Site de Gluon : `https://gluon.mxnet.io/index.html`

de cette dernière et des résultats obtenus, deux autres importations sont nécessaires : celle du module **logging** permettant à des fonctions de callbacks de **Mxnet** d'afficher des informations complémentaires. Le second module, comme pour l'essai de **Keras**, est **time** afin de réaliser des mesures de temps.

```
import mxnet as mx
import logging
import time
```

Listing 5.6 – Importation nécessaire à **Mxnet**.

Importation des données MNIST

De la même façon que **Keras** contient un package avec les jeux de données les plus courant, **Mxnet** possède la fonction **test_utils.get_mnist()** intégrée dans son API afin de télécharger si nécessaire et ensuite de charger les jeux de données d'apprentissage et de validation. Dès lors, la variable **mnist** contient les deux jeux de données, ceux-ci seront séparés ultérieurement. Ensuite, la variable symbolique **data** est créée et "applatira" les chiffres manuscrits MNIST lorsque des données lui seront passées.

```
#%% Charger les données & initialiser contexte
mnist = mx.test_utils.get_mnist()
data = mx.sym.var('data')
# Flatten transforme les données 3-D (canal, hauteur largeur) en des données 1-D
data = mx.sym.flatten(data=data)
```

Listing 5.7 – Chargement du dataset MNIST avec la librairie **Mxnet**.

Création du modèle

Dans un premier temps le modèle est créé dans le Listing 5.8 avec des couches symboliques. Le chaînage des couches s'obtient en précisant la variable d'entrée de la couche. Remarquons que le calcul du potentiel et de la sortie d'un neurone, habituellement regroupé dans une même couche est séparé dans le cas de **Mxnet**. En effet, la couche calculant le potentiel doit être définie comme étant l'entrée de la couche appliquant la fonction d'activation.

Ensuite, les données d'apprentissage et de validation sont séparées. Pour chacun des jeux, un itérateur est crée : **train_iter** pour le jeu d'apprentissage et **val_iter** pour le jeu de validation. Cette méthode permet de ne pas charger la base de données MNIST en entier en mémoire mais d'utiliser un itérateur qui va, à chaque itération, charger en mémoire autant de données que le spécifie la variable **batch_size**. Enfin, un objet **Context** est instancié dans la variable **ctx**. Ce dernier sert à préciser si l'utilisateur souhaite exécuter l'apprentissage du modèle et ses futures prédictions sur le GPU ou le CPU. Dans le cas présent, le CPU sera préféré. **Mxnet** est multi-thread et l'utilisation d'un réseaux de neurones s'exécutera simultanément sur tous les coeurs du CPU.

Finalement, le modèle est crée grâce à la fonction **Module**, prenant en paramètre la dernière couche du réseau de neurones et le contexte. Les autres couches seront retrouvées par le chaînage existant entre les différentes couches. Ensuite, les données sont liées via la l'instruction **bind** affectant les données pointée par **train_iter** via son membre **provide_data** et **provide_label**, dans le cas des réponses souhaitées.

```
#%% Créer le réseau
fc1  = mx.sym.FullyConnected(data=data, num_hidden=40)
```

```
act1 = mx.sym.Activation(data=fc1, act_type='sigmoid')
# 10 classes dans MNIST
fc2  = mx.sym.FullyConnected(data=act1, num_hidden=10)
# Softmax utilise automatiquement la fonction de perte d'entropie croisée
mlp  = mx.sym.SoftmaxOutput(data=fc2, name='softmax')

#%% Définir les paramètres du réseau
batch_size = 256
train_iter = mx.io.NDArrayIter(mnist['train_data'], mnist['train_label'],
    batch_size, shuffle=True)
val_iter = mx.io.NDArrayIter(mnist['test_data'], mnist['test_label'], batch_size
    )
ctx = mx.cpu(0)

#%% Créer modèle
mlp_model = mx.mod.Module(symbol=mlp, context=ctx)
mlp_model.bind(data_shapes=train_iter.provide_data, label_shapes=train_iter.
    provide_label)
mlp_model.init_params(initializer=mx.init.Normal(sigma=0.001))
```

Listing 5.8 – Création d'un modèle .

Résultats

Enfin, l'apprentissage peut être examiné, le code correspondant se trouve dans le Listing 5.9. Tout d'abord, la **logger** de **Mxnet** est définit pour fournir des informations de debug. Ensuite, la fonction **fit**, dont l'utilité est identique à celle de **Keras**, est appelée et requiert plusieurs paramètres en plus des itérateurs des données d'apprentissage et de validation. Tout d'abord, l'algorithme de descente de gradient, Stochastique dans notre cas; la taille du batch étant configurée auparavant avec les itérateurs de données, elle n'a pas besoin d'être spécifiée à nouveau. Tout comme **Keras**, le taux d'apprentissage, le nombre d'itérations et le métrique d'évaluations doivent être indiqués. Durant cet essai, seulement deux métriques sont demandés, bien que d'autres existent : la précision sur les jeux de données ainsi que l'erreur moyenne de la fonction d'entropie relative. **Mxnet** donne également la possibilité d'appeler une fonction de *callback* à chaque élément du batch analysé par le réseau. Plusieurs fonctions de callback existent mais celle qui sera préférée dans le cadre de cette expérimentation sera **Speedometer**. Cette dernière prend en paramètre la taille du batch du réseau, 256 dans notre cas et à quel intervalle elle doit afficher les informations. Celui-ci est définit à 256, soit un batch complet.

```
logging.getLogger().setLevel(logging.DEBUG)  # logging dans stdout
#%% Apprentissage
# Plusieurs metrics à afficher
eval_metrics = mx.metric.CompositeEvalMetric()
eval_metrics.add(mx.metric.Accuracy())
eval_metrics.add(mx.metric.CrossEntropy())

# Lancer l'apprentissage
start = time.time() # Temps depuis epoch AVANT exécution
mlp_model.fit(train_iter,  # train data
              eval_data=val_iter,  # validation data
              optimizer='sgd',  # use SGD to train
              optimizer_params={'learning_rate':0.1},  # use fixed learning rate
              eval_metric= eval_metrics,  # report accuracy during training
              batch_end_callback = mx.callback.Speedometer(batch_size, 256), #
    output progress for each 100 data batches
              num_epoch=10)  # train for at most 10 dataset passes
end = time.time() # Temps depuis epoch APRES exécution
print("Temps écoulé: {} secondes".format(end - start)) # Affiche le temps
```

Listing 5.9 – **Mxnet**.

Les résultats obtenus avec **Mxnet** sont visibles dans le Listing 5.10. Pour chaque itération, la durée, l'erreur moyenne sur les deux jeux de données, la précision sur le jeu d'apprentissage et de validation sont affichés. Contrairement à **Keras**, **Mxnet** ne les engrange pas dans une structure semblable à la classe **History**. Toutefois, une fonction de callback peut-être définie par un programmeur s'il le souhaite. Finalement, la précision atteinte sur le jeu de test est de 90.5%, soit un peu moins que **Keras**.

```
INFO:root:train-labels-idx1-ubyte.gz exists, skipping download
INFO:root:train-images-idx3-ubyte.gz exists, skipping download
INFO:root:t10k-labels-idx1-ubyte.gz exists, skipping download
INFO:root:t10k-images-idx3-ubyte.gz exists, skipping download
INFO:root:Epoch[0] Train-accuracy=0.111353
INFO:root:Epoch[0] Train-cross-entropy=2.301114
INFO:root:Epoch[0] Time cost=0.439
INFO:root:Epoch[0] Validation-accuracy=0.114160
INFO:root:Epoch[0] Validation-cross-entropy=2.296405
INFO:root:Epoch[1] Train-accuracy=0.231566
INFO:root:Epoch[1] Train-cross-entropy=2.169508
INFO:root:Epoch[1] Time cost=0.385
INFO:root:Epoch[1] Validation-accuracy=0.426953
INFO:root:Epoch[1] Validation-cross-entropy=1.840675
INFO:root:Epoch[2] Train-accuracy=0.547922
INFO:root:Epoch[2] Train-cross-entropy=1.460758
INFO:root:Epoch[2] Time cost=0.375
INFO:root:Epoch[2] Validation-accuracy=0.669336
INFO:root:Epoch[2] Validation-cross-entropy=1.131869
INFO:root:Epoch[3] Train-accuracy=0.727709
INFO:root:Epoch[3] Train-cross-entropy=0.965378
INFO:root:Epoch[3] Time cost=0.387
INFO:root:Epoch[3] Validation-accuracy=0.785547
INFO:root:Epoch[3] Validation-cross-entropy=0.804433
INFO:root:Epoch[4] Train-accuracy=0.808826
INFO:root:Epoch[4] Train-cross-entropy=0.721473
INFO:root:Epoch[4] Time cost=0.383
INFO:root:Epoch[4] Validation-accuracy=0.836328
INFO:root:Epoch[4] Validation-cross-entropy=0.624261
INFO:root:Epoch[5] Train-accuracy=0.846908
INFO:root:Epoch[5] Train-cross-entropy=0.583373
INFO:root:Epoch[5] Time cost=0.391
INFO:root:Epoch[5] Validation-accuracy=0.867090
INFO:root:Epoch[5] Validation-cross-entropy=0.519897
INFO:root:Epoch[6] Train-accuracy=0.869282
INFO:root:Epoch[6] Train-cross-entropy=0.499523
INFO:root:Epoch[6] Time cost=0.375
INFO:root:Epoch[6] Validation-accuracy=0.883203
INFO:root:Epoch[6] Validation-cross-entropy=0.453242
INFO:root:Epoch[7] Train-accuracy=0.883743
INFO:root:Epoch[7] Train-cross-entropy=0.443983
INFO:root:Epoch[7] Time cost=0.382
INFO:root:Epoch[7] Validation-accuracy=0.895312
INFO:root:Epoch[7] Validation-cross-entropy=0.407960
INFO:root:Epoch[8] Train-accuracy=0.892354
INFO:root:Epoch[8] Train-cross-entropy=0.405379
INFO:root:Epoch[8] Time cost=0.392
INFO:root:Epoch[8] Validation-accuracy=0.901465
```

```
INFO:root:Epoch[8] Validation-cross-entropy=0.376087
INFO:root:Epoch[9] Train-accuracy=0.898172
INFO:root:Epoch[9] Train-cross-entropy=0.377534
INFO:root:Epoch[9] Time cost=0.385
INFO:root:Epoch[9] Validation-accuracy=0.905078
INFO:root:Epoch[9] Validation-cross-entropy=0.352774
Temps écoulé: 4.365509271621704 secondes
```

Listing 5.10 – Résultat de l'apprentissage d'un réseau de neurones avec **Mxnet** sur la base de données MNIST.

5.3 Le module Machine Learning d'OpenCV

5.3.1 Présentation du module de Machine Learning

De base, **OpenCV**est une librairie de traitement d'image. Elle possède également un module **ml** de machine learning. Ce dernier contient des fonctionnalités d'apprentissage supervisé ou non supervisé tels que des réseaux de neurones, des machines à vecteurs de support (Support Vector Machine, SVM en anglais), d'analyse en composantes principales (ACP), des K Plus Proches Voisins (K-Nearest Neighbours, KNN en anglais), etc. Contrairement à **Keras**, le module **ml** d'**OpenCV** n'est pas spécialisé dans les réseaux de neurones. L'utilisation d'**OpenCV** en Python cache en réalité des bindings vers les modules C++. Par ailleurs, une difficulté s'impose dans **OpenCV**, la documentation du module est très peu complète. Cette section est référencée en partie grâce à la documentation, mais est completée avec la source [Bey17].

De plus, les réseaux de neurones sont bien moins pourvus que ceux de **Keras**. Beaucoup de fonctionnalités manquent à l'appel : la fonction d'erreur d'entropie croisée, la fonction d'activation Softmax, les Descente de Gradient Stochastique et Mini-Batch, réseaux de neurones convolutifs (Convolutional Neural Network, CNN en anglais), etc. Une fonctionnalité particulière est toutefois présente : le chargement de couche à partir d'un fichier **Caffe**, une autre librairie Open-Source initiée à l'université de Berkeley, dans le module Deep Neural Networks **dnn** de la librairie **OpenCV**. En réalité, **OpenCV** n'est pas assez développée pour entraîner des réseaux de neurones profonds, mais pour utiliser des modèles pré-entraînés par d'autres bibliothèques telles que **TensorFlow**, **Caffe**, **Torch**, etc.

5.3.2 Application

Importation des librairies

Afin d'utiliser le module **ml** de Machine Learning d'OpenCV, le package Python **cv2** d'**OpenCV** est importé dans le Listing 5.11. À nouveau, **numpy** et **Keras** seront également utilisés, cependant ce dernier ne servira qu'à importer le jeu de données MNIST. L'objet **OneHotEncoder** appartenant au package **preprocessing** de la librairie statistique Python **sci-kit**. Celui-ci permet de transformer les labels des réponses en one hot encoded vector requis par le réseau de neurones. L'objet **accuracy_score** de cette même librairie est utilisé afin d'évaluer le pourcentage de bonne classification du réseau après l'apprentissage.

```
import cv2
import numpy as np
from keras.datasets import mnist
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import accuracy_score
```

```
import time
```

Listing 5.11 – Importations nécessaires pour créer un réseau de neurones avec OpenCV.

Importer les données MNIST

À nouveau, les caractères MNIST sont chargés en mémoire et sont transformés en tableaux tri-dimensionnels utilisables par **OpenCV**. Pour rappel, le jeu de données possède 10 classes : les chiffres de 0 à 9 et 60 000 exemples d'apprentissage. L'objet **OneHotEncoder** permet de transformer la variable **y_train** de dimensions 60 000 × 1 en un objet 60 000 × 10. Ce dernier est ensuite redimensionné pour avoir un seul exemple par ligne et une sortie souhaitée par colonne, c'est-à-dire en un tableau de dimensions 60000 × 1 × 10. Le code correspondant est disponible dans le Listing 5.12.

```
#%% Importer les données MNIST
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Transformer les labels des données en one hot encoded vectors
enc = OneHotEncoder(sparse=False, dtype=np.float32)
Y_train_pre = enc.fit_transform(y_train.reshape(-1, 1))
Y_test_pre = enc.fit_transform(y_test.reshape(-1, 1))

# Applatir les entrées: 28x28 -> 1x784
X_train_pre = X_train.astype(np.float32)
X_train_pre = X_train_pre.reshape((X_train.shape[0], -1))

X_test_pre = X_test.astype(np.float32)
X_test_pre = X_test_pre.reshape((X_test.shape[0], -1))
```

Listing 5.12 – Chargement et manipulation des caractères MNIST.

Création et apprentissage du modèle

Dans le Listing 5.13, un réseau de neurones identique à celui de la Section 5.1 est instancié grâce à la fonction **ANN_MLP_create** du package **ml**. Cette dernière prend en paramètre un vecteur d'entiers spécifiant, dans l'ordre, le nombre d'entrées, le nombre de neurones dans la couche et le nombre de neurones de sortie. Ce vecteur est aussi long qu'il y a de couches dans le réseau.

Comme pour **Keras**, la fonction d'activation peut être choisie. Toutefois, la fonction **Logistique** classique n'existe pas telle qu'elle. Celle qui est disponible est la fonction :

$$\sigma_{\alpha,\beta}(x) = \beta \, \frac{1 - e^{-\alpha x}}{1 + e^{-\alpha x}} \ , \ \sigma(x) \ \in \ [0, \beta]$$

Où α et β sont des paramètres. Les paramètres par défaut de la méthode **setActivationFunction** avec la fonction **ANN_MLP_SIGMOID_SYM** sélectionnée sont $\alpha = \beta = 0$, la fonction qui sera réellement utilisée est :

$$y(x) = 1.7159 * tanh(2/3 * x) \ , \ y(x) \ \in \ [-1.7159, 1.7159]$$

```
#%% Fixer les paramètres du réseau
n_input = 784 # Quantité d'entrées
n_hidden = 40 # Quantité de neurones dans la première couche
n_output = 10 # Quantité de neurones de sortie
```

```
alpha= 2.5
beta = 1.0
eta = 0.01

# Créer le réseau de neurones
mlp = cv2.ml.ANN_MLP_create()

# Créer les couches
mlp.setLayerSizes(np.array([n_input, n_hidden, n_output]))
mlp.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, alpha, beta)
mlp.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP)
mlp.setBackpropWeightScale(eta)
```

Listing 5.13 – Création du réseau de neurones.

Une fois le réseau paramétré, vient l'apprentissage, dont le code est disponible dans le Listing 5.14. **OpenCV** permet de personnaliser l'arrêt de la descente de gradient de plusieurs façon. La première est l'arrêt classique car le nombre d'itérations spécifié est dépassé grâce à la macro **TERM_CRITERIA_MAX_ITER**. Le second autorise l'arrêt de l'apprentissage si l'erreur du réseau ne décroît plus de manière significative et s'active avec la macro **TERM_CRITERIA_EPS**.

```
#%% Entraîner le RDN
term_mode = cv2.TERM_CRITERIA_MAX_ITER #+ cv2.TERM_CRITERIA_EPS
term_max_iter = 10
term_eps = 0.01
mlp.setTermCriteria((term_mode, term_max_iter, term_eps))

start = time.time() # Temps depuis epoch AVANT exécution
mlp.train(X_train_pre, cv2.ml.ROW_SAMPLE, Y_train_pre)
end = time.time() # Temps depuis epoch APRES exécution
print("Temps écoulé: {} secondes".format(end - start)) # Affiche le temps

# Calculer la précision sur le jeu d'APPRENTISSAGE
_, y_hat_train = mlp.predict(X_train_pre)
accuracy_score(y_hat_train.round(), Y_train_pre)

# Calculer la précision sur le jeu de VALIDATION
y_hat_test, y_hat_test = mlp.predict(X_test_pre)
accuracy_score(y_hat_test.round(), Y_test_pre)
```

Listing 5.14 – Apprentissage du réseau de neurones avec **OpenCV**.

Résultats

Même si **OpenCV** ne dispose pas de la fonction d'erreur de l'entropie croisée et de la Descente de Gradient Stochastique, le résultat obtenu est comparable à celui de Keras. En effet, le taux de reconnaissance obtenu sur le jeu de validation est de 88.49%. Cependant, on remarque que le temps mis par **OpenCV** pour effectuer l'apprentissage est nettement supérieur à celui de **Keras**. Par ailleurs, contrairement à **Keras**, **OpenCV** ne dispose pas d'une fonction permettant, durant la descente de gradient, de tester le taux de reconnaissance sur le jeu de validation qui doit être évalué manuellement. De plus, aucun affichage n'est possible durant cette phase et les informations ne peuvent être récupérées. Par ailleurs, le temps moyen par itération avec **OpenCV** est de 3.7 seconde, ce qui est beaucoup plus que **Keras**.

```
Temps écoulé: 37.012752294540405 secondes
0.8849
```

Listing 5.15 – Résultat de l'apprentissage d'un réseau de neurones **OpenCV** sur la base de données MNIST.

5.4 Application avec NeuroLib

Importer les données MNIST

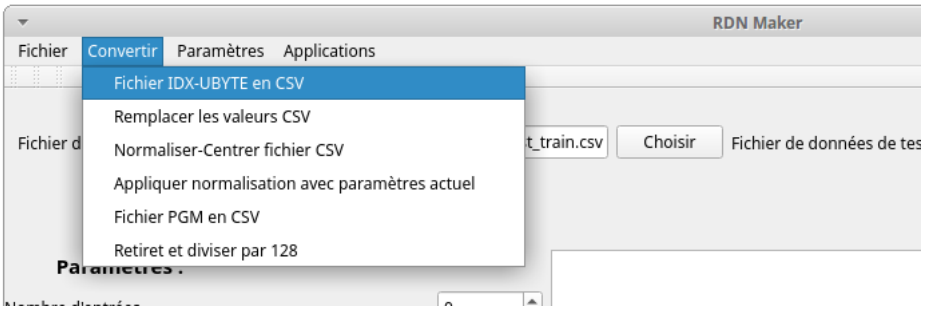


FIGURE 5.2 – L'option permettant de convertir un fichier idx-ubyte en CSV.

Le logiciel *RDN Maker* possède une fonctionnalité pour convertir les fichiers *.udx3-ubyte* contenant les caractères MNIST dans des fichiers CSV. Comme le montre la Figure 5.2, il suffit d'aller dans les options "Conversion" > "Fichier UDX-UBYTE en CSV". Dans un premier temps, l'utilisateur doit sélectionner le fichier contenant les images dont l'extension est "*.udx3-ubyte*". Dans un second temps, l'utilisateur doit sélectionner le fichier correspondant à celui des images, contenant les labels de ces dernières, dont l'extension est "*.udx1-ubyte*". Dans un troisième temps, l'utilisateur doit sélectionner le dossier et le nom du fichier CSV dans lequel il souhaite enregistrer les résultats. Enfin, les fichiers convertis, ils peuvent être chargés en mémoire en les choisissant comme fichier d'apprentissage et de test.

Création et apprentissage du modèle

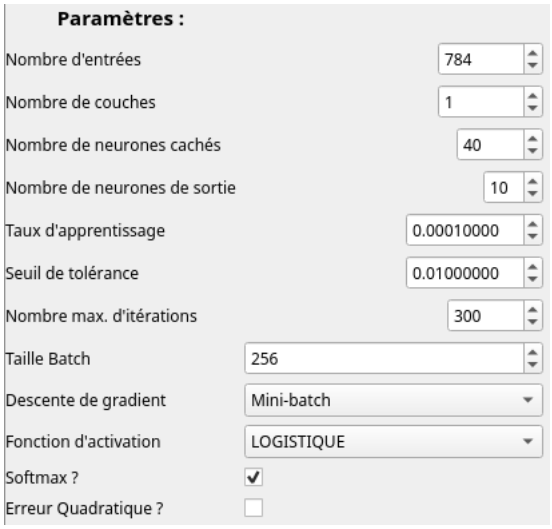


FIGURE 5.3 – Paramétrage du réseau de neurones dans *RDN Maker*.

Une fois les données chargées en mémoire, le modèle peut être créé. Cette étape est montrée sur la Figure 5.3. Ensuite, en cliquant sur "Entraîner le réseau", l'apprentissage du modèle sera lancé. Nous pouvons remarquer que le taux d'apprentissage et le nombre d'itérations ont dû être adapté aux capacités de **NeuroLib** afin d'avoir des résultats comparables à ceux obtenus par les autres librairies.

Résultats

Le Listing 5.16 contient la sortie du programme *RDN Maker* pour le réseau de neurones créé. Tout d'abord, les résultats obtenus sont nettement inférieurs à ceux de **Keras** et **OpenCV**. En effet, le taux d'apprentissage et le nombre d'itérations ont dû être adaptés. Ensuite, le temps mis pour entraîner le réseau de neurones avec 300 itérations est d'approximativement 38.750 secondes, soit une moyenne de 0.19 seconde par itération. Cette vitesse est dûe au C et au C++ qui sont des langages compilés tandis le Python est interprété.

```
Le fichier /home/julien/Documents/MNIST/mnist_train.csv ne contient pas de ligne
    avec les écarts-types
Chargement du fichier /home/julien/Documents/MNIST/mnist_train.csv terminé en 7s
    169ms 875us 456ns
Chargement du fichier /home/julien/Documents/MNIST/mnist_test.csv terminé en 1s
   205ms 278us 208ns
Temps mis pour entrainer: 56s 899ms 378us 176ns Erreur Moyenne : 0.889575
Test du jeu exécuté en 2s 384ms 144us 640ns Erreur : 1287 (12.87 %)
```

Listing 5.16 – Résultats obtenus avec un réseau de neurones de la bibliothèque **NeuroLib** sur la base de données MNIST.

5.5 Conclusions

La Table 5.1 synthéthise les performances des réseaux de neurones entraînés avec les différentes librairies. Tout d'abord, les meilleures performances sont atteintes avec le réseau de **Keras**avec 91.54% de précision sur le jeu de test en seulement 10 itérations. Ensuite viennent **Mxnet** avec 90.5% puis **NeuroLib** et enfin **OpenCV** . De plus, le temps moyen par itération d'**OpenCV** est nettement supérieur à celui de **Keras**. Cela est dû au fait qu'**OpenCV** n'implémente pas la Descente de Gradient Mini-Batch mais uniquement la Full-Batch. **OpenCV** doit alors parcourir l'ensemble des exemples d'apprentissage avant de faire une mise à jour des poids, tandis que les autres librairies n'en parcourent que 256. De ce fait, elles sont plus rapides. Par ailleurs, le meilleurs temps moyen par itération, pour les librairies utilisées en Python, est celui de **Mxnet** grâce à son implémentation multi-CPU.

En termes de fonctionnalités, un résumé est disponible dans la Table 5.2. **Keras** surpasse **OpenCV** et **NeuroLib**. En effet, à chaque itération et pour chaque jeu de données, d'apprentissage ou de validation, l'erreur et la précision sont enregistrées dans une structure de données appelée **History**. Dans un cas plus complexe, ou plusieurs modèles avec des paramètres différents sont testés, un graphique, tel que sur la Figure 5.4, affichant la précision des modèles construits. D'autant plus qu'en Python, ce genre de graphique est très facile à réaliser grâce à la bibliothèque **matplotlib**. Le code exécuté pour obtenir la Figure 5.4 se trouve dans le Listing 5.17. Une fonctionnalité intéressante de **Mxnet** est la possibilité d'appeler des fonctions de callbacks pendant l'apprentissage des données. Ce type de fonction permet, à une fréquence régulière, d'utiliser le réseau de neurones pour qu'il affiche ou enregistre des informations par exemple. En général, les réseaux de **Mxnet** sont plus personnalisables et plus flexible que ceux de **Keras**.

Librairie	Descente de Gradient	Fonction d'erreur	Itérations	Temps moyen par itération (sec)	Précision % (jeu de test)
Keras	Mini-Batch (256)	Entropie croisée	10	1	91.54
Mxnet	Mini-Batch (256)	Entropie croisée	10	0.43	90.5
OpenCV	Full-Batch	Quadratique	10	3.7	88.49
NeuroLib	Mini-Batch (256)	Entropie croisée	300	0.19	87.13

TABLE 5.1 – Récapitulatif des performances des réseaux de neurones entraînés avec les différentes librairies.

Librairie	Full-Batch	Stochastique	Mini-Batch	Quadratique	Entropie croisée	Structure **History**	Évaluation automatique de l'erreur sur le jeu de test
Keras	Oui	Oui	Oui	Oui	Oui	Oui	Oui
Mxnet	Oui	Oui	Oui	Oui	Oui	Non mais dispose de fonctions callbacks	Oui
OpenCV	Oui	Non	Non	Oui	Non	Non	Non
NeuroLib	Oui	Oui	Oui	Oui	Oui	Non	Non

TABLE 5.2 – Récapitulatif des performances des réseaux de neurones entraînés avec les différentes librairies.

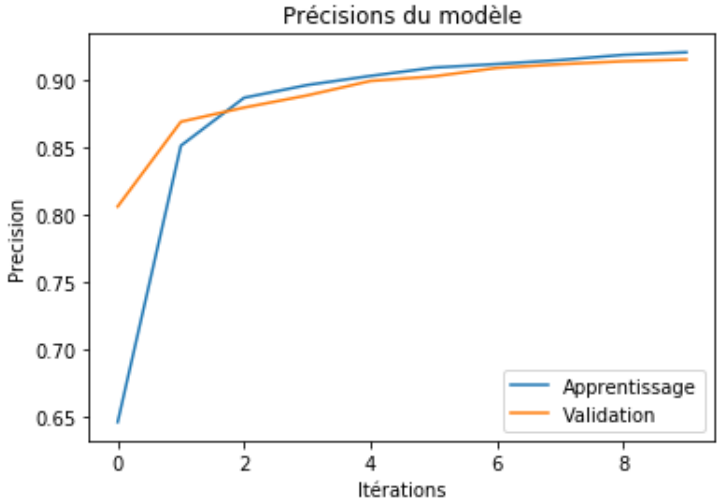


FIGURE 5.4 – La précision du modèle obtenu avec **Keras**.

```
import matplotlib.pyplot as plt

fig = plt.figure() # Créer une nouvelle figure
plt.plot(history.history['acc']) # Précision sur le jeu d'apprentissage
plt.plot(history.history['val_acc']) # Précision sur le jeu de validation

plt.title('Précisions du modèle')
plt.ylabel('Precision')
plt.xlabel('Itérations')
plt.legend(['Apprentissage', 'Validation'], loc='lower right')

# Enregistrement dans un fichier
fig.savefig('/home/julien/Documents/RapportsStage/images/keras_accuracy_plot.png', bbox_inches='tight')
plt.show()
```

Listing 5.17 – Code utilisé pour dessiner le graphique de la Figure 5.4.

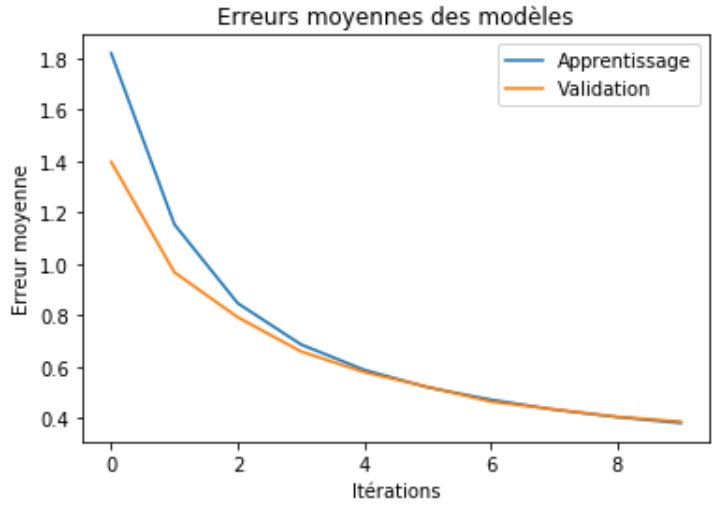


FIGURE 5.5 – La précision du modèle obtenu avec **Keras**.

Concernant **OpenCV**, le module **ml** est peu documenté et également peu étoffé, même s'il fournit d'excellents résultats. L'inconvénient majeur est qu'il fournit très peu d'informations

sur l'apprentissage du réseau de neurones contrairement à **Keras**. Effectivement, **OpenCV**ne remplit aucune structure similaire à l'objet **History**. À cause de cela, des figures telles que 5.4 ou 5.5 ne peuvent être tracées.

Enfin, la librairie **NeuroLib** dispose tout de même de résultats comparables aux autres librairies. Malgré un nombre d'itérations plus élevé et un taux d'apprentissage différent, les résultats obtenus sont légèrement meilleurs que ceux obtenus avec **OpenCV**. Cette comparaison de librairies a également permis de déceler fonctionnalités manquantes :

1. La première est une structure de données semblable à **History** de **Keras**. Celle-ci contiendrait les différentes valeurs de l'erreur moyenne et la précision sur le jeu d'apprentissage et les mêmes informations sur le jeu de validation, la quantité de données contenues dans les deux jeux, etc.
2. La seconde est l'équivalent de la fonction **fit** de **Keras** qui demanderait le jeu d'apprentissage et de test (ou de validation) et permettrait d'obtenir un objet **History** comme valeur de retour.
3. La troisième est une intégration des fonctions de callback tel que dans **Mxnet**. Celle-ci serait utile, par exemple, d'afficher ou de regrouper certaines informations, aux choix, sur l'apprentissage dans une structure identique à **History**.
4. Une quatrième amélioration, moins importante pour l'instant, consiste à intégrer le support multi-CPU et multi-GPU à **NeuroLib**.
5. Enfin, la dernière amélioration se trouve au niveau du logiciel *RDN Maker*. Grâce à la classe **History** obtenue après un entraînement, plusieurs graphiques peuvent être utiles tel qu'un graphique traçant les précisions des réseaux de neurones entraînés. Un second traçant les erreurs moyennes des réseaux. La Figure 5.5 contient un graphique des erreurs moyennes au fil des itérations et le code du Listing 5.18 contient le code correspondant.

```
import matplotlib.pyplot as plt

fig = plt.figure() # Créer une nouvelle figure

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])

plt.title('Erreurs moyennes des modèles')
plt.ylabel('Erreur moyenne')
plt.xlabel('Itérations')
plt.legend(['Apprentissage', 'Validation'], loc='upper right')

# Enregistrement dans un fichier
fig.savefig('/home/julien/Documents/RapportsStage/images/keras_loss_plot.png',
    bbox_inches='tight')
plt.show()
```

Listing 5.18 – Code utilisé pour dessiner le graphique de la Figure 5.5.

Chapitre 6

Conclusions générales

L'objectif premier de ce stage est de réaliser une étude théorique des réseaux de neurones feed-forward afin de comprendre leur fonctionnement et leurs limites. Ce travail commence l'étude des réseaux de neurones par le perceptron, le neurone le plus simple. Le principe de ce dernier est expliqué et accompagné d'une implémentation. Deux algorithmes d'apprentissage existent dans le cas du perceptron. Le premier est la Descente de Gradient, un algorithme d'optimisation numérique de minimisation de l'erreur. Ensuite, vient ADALINE, une optimisation de la Descente de Gradient. Ces deux algorithmes sont testés et comparés. D'entre eux, ADALINE semble converger plus rapide vers un minimum local grâce à une mise à jour des poids plus fréquente que la Descente de Gradient. De plus, il est montré que la limite du perceptron est une classification ou régression linéaire, en fonction de l'utilisation. Dans le cas de la classification, elle se limite à deux classes distinctes. C'est pourquoi le modèle du perceptron seul est étendu à une couche de perceptrons permettant de distinguer autant de classes qu'il y a de perceptrons. Cependant, les limites de décisions qui en découlent restent linéaires.

Le perceptron ne fournissant que des limites de décisions linéaires, il est étendu au modèle de perceptron multi-couches utilisant des fonctions d'activation non-linéaires. Cette amélioration transforme la limite de décision en une limite non-linéaire. À nouveau, une explication du principe et une implémentation sont fournis. Les mécanismes mathématiques de rétro-propagation du gradient de l'erreur, une généralisation de la Descente de Gradient du perceptron, est expliqué suivit d'une description de l'algorithme ainsi que d'une implémentation. Comme pour le perceptron, des tests sur des données en deux dimensions sont effectués afin de visualiser l'effet des neurones mis en réseau. Ce chapitre se termine ensuite par une brève explication du théorème d'approximation universel. Il s'agit d'un théorème s'appliquant aux réseaux de neurones qui stipule qu'ils peuvent approximer, avec une précision arbitraire, n'importe quelle fonction continue. Ce théorème est purement théorique et ne fournit aucune explication quant à la méthode à utiliser pour arriver à paramétrer le réseau de neurones afin d'approximer une fonction.

Consécutivement, une application à la reconnaissance des caractères issus de plaques d'immatriculation française a été réalisée selon deux méthodes. La première méthode consiste en l'utilisation brute des pixels comme entrées des réseaux de neurones tandis que la seconde demande l'extraction de caractéristiques géométriques des caractères utilisées par le réseau comme entrées. Une Analyse en Composantes Principales a également été pratiquée sur ces caractéristiques afin d'en discerner les plus discriminantes et alléger le calcul du réseau. Malheureusement, cette méthode s'est montrée peu performante. Hors de cette expérimentation, une méthodologie a pu être relevée afin de trouver des paramètres efficaces d'un réseau de

neurones. De plus, ce chapitre aborde également l'importance de la qualité des données et des pré-traitements appliqués à ces données avant l'apprentissage et la validation.

Le chapitre suivant aborde une comparaison de trois librairies de deep learning avec **NeuroLib**, la librairie développée dans le cadre de ce stage sur l'exemple connu qu'est la base de données des caractères manuscrits MNIST. Les librairies sus-mentionnées sont : **Keras**, **Mxnet**et **OpenCV**. Hors de cette analyse, un classement des librairies en fonction des résultats obtenus et des fonctionnalités dont elles disposent a pu être effectué. D'une part, les résultats obtenus par **NeuroLib** sont comparables à ceux des autres librairies. D'autre part, grâce à ce classement, une liste d'améliorations à apporter à **NeuroLib** a pu être rédigée.

Le développement d'une librairie de réseaux de neurones a permis de faire la liaison entre la théorie et la programmation de ces derniers. Une problématique abordée est le dépassement de capacité des types primitifs. Les bases de données de machine learning sont souvent lourdement peuplées et de ce fait, l'accumulation des gradients conduisent fréquemment à des valeurs importantes voir des **NaN** [1] ou **Inf**. Ce phénomène se répercute ensuite dans les corrections apportées aux poids et sur l'erreur moyenne, la librairie a alors du être modifiée afin de devenir plus robuste. De plus, cette bibliothèque est la preuve de la compréhension des différents points abordés dans ce travail. Enfin, cette compréhension a également permis la rédaction d'un "How To" des réseaux de neurones dans la section suivante.

Pour conclure, les différentes spécifications du cahier des charges ont été respectées. L'étude théorique des réseaux de neurones a été effectuée. Les capacités dans la reconnaissance de formes ont également été analysées et appliquées au travers des caractères des plaques d'immatriculation et les caractères MNIST. En outre, la librairie demandée a été développée et testée avec succès sur les exemples mentionnés, sur un système Windows et un Linux. Des résultats sont obtenus rapidement et ces derniers sont concluants vis-à-vis des autres librairies. De ce fait, les réseaux de neurones semblent être applicables dans le contexte de la *RoboCup*.

6.1 "How To" des réseaux de neurones

Le premier paramètre, un des plus importants, est le taux d'apprentissage. Souvent, celui-ci est fixé à une valeur de 0.1 et il est ensuite testé sur le jeu d'apprentissage. Un taux d'apprentissage trop élevé renverra un **NaN** ou un **inf** s'il est vraiment trop grand. Il est coutume de diminuer le taux d'apprentissage tant que l'erreur moyenne et le taux d'erreurs sur le jeu de validation continuent de diminuer. De la même façon, le taux d'apprentissage peut-être augmenté tant qu'il continue à fournir de meilleurs résultats au fur et à mesure de tests. Par ailleurs, le taux d'apprentissage est un paramètre propre aux données d'apprentissage et de validation.

Un second paramètre important est le nombre de neurones dans une couche. L'augmentation de la quantité de neurones permet au réseau de s'adapter aux différentes singularités que la limite de décision peut prendre, mais peuvent également être plus difficile à entraîner. Par ailleurs, plus un réseau comporte de neurones plus ce dernier est susceptible de faire du sur-apprentissage et donc de mal se généraliser aux exemples de validation. Parfois, un modèle très complexe peut donner des résultats très proches de la perfection, mais il est obtenu au cours de longs calculs. Ces modèles seront, en fonction des besoins, laissé au détriment de modèles plus simples et pratiquement tout aussi performants. Par conséquent, le nombre de neurones peut

1. Not A Number

être augmenté, avec un taux d'apprentissage fixe tant que la précision sur le jeu de validation diminue.

Lors des différents tests, l'augmentation du nombre de couches n'a donné qu'une erreur moyenne plus importante et par conséquent, le taux de bonne classification ne s'en voyait guère amélioré. En effet, dans le cas de la classification, le nombre de couches est un paramètre qui renforce la difficulté à entraîner un réseau de neurones. De ce fait, ce paramètre sera très souvent laissé à l'unité.

Les réseaux ayant les meilleurs résultats ont été établi avec la fonction d'activation **Logistique**dans les couches cachées et la fonction **Softmax**dans la couche de sortie conjointement avec la fonction d'erreur d'entropie croisée. Ces fonctions seront alors souvent privilégiées, mais la fonction **Tangente Hyperbolique**donne de très bons résultats également, c'est pourquoi elle reste toujours intéressante à tester.

Enfin, la Descente de Gradient privilégiée sera la Stochastique ou la Mini-Batch. Dans le cas de petits jeux de données, la Stochastique est intéressante. Mais sur les gros volumes de données, la Descente de Gradient Mini-Batch d'une taille relativement petites par rapport aux données disponibles est très efficace également et permet d'obtenir un modèle performant et plus rapidement qu'avec la Descente de Gradient Stochastique . Quant à la Descente de Gradient Full-Batch sur des bases de données importantes, l'erreur moyenne tendra à converger plus lentement voir à ne pas converger du tout.

6.2 Perspectives

Dans ce document, la thématique générale des réseaux de neurones a été abordée. Il s'agit cependant d'un sujet très populaire actuellement et de nombreux algorithmes, études et articles ont été établis. Par conséquent, d'autres sujets peuvent être abordés afin de compléter ce travail :

- La régulation : le but de la régulation est d'éviter le sur-apprentissage des données. Des techniques couramment rencontrées dans la littérature telles que le *Dropout* ou la régulation L1 et L2 n'ont pas été étudiées durant ce stage.
- Des optimisations de Descente de Gradient existent : le terme de moment, Adagrad, RMSProp, etc [2].
- Un type de réseau de neurones feed-forward n'a pas été abordé dans ce travail bien qu'il donne des résultats très performant est le réseau de neurones convolutif ou Convolutional Neural Network (CNN) en anglais.
- Un pré-traitement mentionné dans la référence [KJ18b] propose une autre utilisée de l'Analyse en Composantes Principales qui est le *PCA Whitening* ou blanchiment par l'ACP en français. Cette technique permet de décorreler les données.

2. Une liste non-exhaustive est disponible à l'URL : `http://ruder.io/optimizing-gradient-descent/` .

Bibliographie

[Bey17] Michael Beyeler. *Machine Learning for OpenCV*. Packt, 2017. Chap. 9, p. 230–242.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Gos96] Bernard Gosselin. "Application de réseaux de neurones artificiels à la reconnaissance automatique de caractères manuscrits". Thèse. Faculté Polytechnique de Mons, 1996. URL : http://www.tcts.fpms.ac.be/publications/phds/gosselin/GosselinB_Phd.pdf.

[KJ18a] Andrej Karpathy et Justin Johnson. *Linear classification : Support Vector Machine, Softmax*. 2018. URL : http://cs231n.github.io/linear-classify/.

[KJ18b] Andrej Karpathy et Justin Johnson. *Neural Networks Part 2 : Setting up the Data and the Loss*. 2018. URL : http://cs231n.github.io/neural-networks-2/.

[KJ18c] Andrej Karpathy et Justin Johnson. *Optimization : Stochastic Gradient Descent*. 2018. URL : http://cs231n.github.io/optimization-1/.

[KKN10a] Erwin Kreyszig, Herbert Kreyszig et Edward J. Norminton. *Advanced Engineering Mathematics, 10th Edition*. Wiley, 2010. Chap. 22, p. 950–1008.

[KKN10b] Erwin Kreyszig, Herbert Kreyszig et Edward J. Norminton. *Advanced Engineering Mathematics, 10th Edition*. Wiley, 2010. Chap. 24, p. 1037–1038.

[KS96a] Ben Kröse et Patrick Van der Smagt. *An introduction to Neural Networks*. The University of Amsterdam, 1996, p. 23–32. URL : https://www.infor.uva.es/~teodoro/neuro-intro.pdf.

[KS96b] Ben Kröse et Patrick Van der Smagt. *An introduction to Neural Networks*. The University of Amsterdam, 1996, p. 33–46. URL : https://www.infor.uva.es/~teodoro/neuro-intro.pdf.

[LCB98] Yann Lecun, Corinna Cortes et Christopher J.C. Burges. *The MNIST database of handwritten digits*. 1998. URL : http://yann.lecun.com/exdb/mnist/.

[Ng+13] Andrew Ng et al. *Gradient checking and advanced optimization*. 2013. URL : http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization.

[Nie20] Michael Nielsen. *Neural Networks and Deep Learning*. 12/2017. URL : http://neuralnetworksanddeeplearning.com/chap3.html#the_cross-entropy_cost_function.

[Pre01] Franck Preiswerk. *Shannon entropy in the context of machine learning and AI*. 4/01/2018. URL : https://medium.com/swlh/shannon-entropy-in-the-context-of-machine-learning-and-ai-24aee2709e32.

[Ren14] Steve Renals. *Multi-Layer Neural Networks*. 2014. URL : https://www.inf.ed.ac.uk/teaching/courses/asr/2013-14/asr08a-nnDetails.pdf.

[Shu02] Jamie Shutler. *Statistical Moments*. 2002. URL : https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/SHUTLER3/CVonline_moments.html.

[Vil17] Claude Vilvens. *Data mining avec R*. Haute École de la Province de Liège, 2017. Chap. 3, p. 139–164.

[Wag16] Jean-Marc Wagner. *Mathématiques appliquées à l'informatique et à l'imagerie 3D*. Haute École de la Province de Liège, 2016.

[Wik18] Wikipedia. *Plaque Immatriculation Françaises*. 2018. URL : https://fr.wikipedia.org/wiki/Plaque_d%27immatriculation_fran%C3%A7aise#Num%C3%A9rotation.

[Alf00] Alfio Quarteroni and Riccardo Sacco and Fausto Saleri. *Numerical Mathematics*. Springer, 2000. Chap. 7, p. 300–302.

[Fra00] François Denis and Rémi Gilleron. *Apprentissage à partir d'exemples*. 2000. URL : http://www.grappa.univ-lille3.fr/polys/apprentissage/sortie005.html.

[Gé07] Gérard Dreyfus and Jean-Marc Martinez and Manuel Samuelides and Mirta B. Gordon and Fouad Badran and Sylvie Thiria. *Apprentissage statistique*. Livre prêté par M. Vilvens. Édition Eyrolles, 2007.

[Ree17] Reena Shaw. *Top 10 Machine Learning Algorithms for Beginners*. 2017. URL : https://www.kdnuggets.com/2017/10/top-10-machine-learning-algorithms-beginners.html.

Annexe A

Tracer une limite de décision

A.1 Perceptron

Dans ce document, beaucoup de limites de décision de perceptrons mono et multi-couches ont été montrées. Les perceptrons possèdent une limite de décision linéaire. Celle-ci est représentée dans un graphique où les entrées constituent les dimensions. S'il y a P entrées et donc P poids, alors l'hyper-espace est $\mathbb{R}^P$ [KS96a]. Dans les cas étudiés, les exemples se situaient toujours dans $\mathbb{R}^2$ dans le but de simplifier l'étude des perceptrons mono-couche et de leurs résultats. Afin de tracer cette limite de décision, il suffit d'éliminer la notion de seuil et de résoudre l'équation de l'hyper-plan :

$$w_0 + \sum_{i=0}^{P} w_i x_i = 0$$

Où x_i est la ième variable de l'équation et w_i le poids i correspondant.

Par exemple, un perceptron à 2 entrées x et y et de poids w_0, w_1, w_2. Sa limite de décision est une droite dans le plan. Par conséquent, elle se trace dans l'espace à deux dimensions $\mathbb{R}^2$. L'équation de la droite de est [KS96a] :

$$\begin{aligned} w_1 x + w_2 y + w_0 &= 0 \\ y &= -\frac{w_1}{w_2} x - \frac{w_0}{w_2} \end{aligned}$$

A.2 Réseau de neurones

Déduire l'équation de la limite de décision d'un réseau de neurones est plus complexe. Une méthode simple est, pour un réseau ayant P entrées et dont le plan de décision est dans $\mathbb{R}^P$, de constituer tous les exemples possibles dans un intervalle. Puis faire analyser chacun des exemples dans le réseau et récupérer ses prédictions. Ainsi, pour un certain intervalle, les régions des différentes classes seront déterminées.

Prenons un exemple d'un perceptron multi-couches à 2 entrées. Par conséquent, les zones de décision sont dans $\mathbb{R}^2$. Si l'ensemble des entrées de la première dimension, notée x est compris entre $[-1, 1]$ et l'ensemble des entrées de la seconde dimension, notée y est comprise entre $[-0.5, 0.5]$. La sortie du réseau de neurones doit être déduite pour chaque entrée possibe (x, y). Le code dans le Listing A.1 présente une fonction enregistrant la limite de décision, dans un certain intervalle, d'un réseau de neurones à deux entrées.

```
void EnregistrerLimiteDecision(const string &nomFichier, double xmin, double
    xmax, double ymin, double ymax, double pas, ReseauNeurone &rdn)
{
    size_t quantite_x = (size_t) (xmax-xmin)/pas,
        quantite_y = (size_t) (ymax-ymin)/pas;

    // Allouer l'espace mémoire au tableau
    double** z = new double*[quantite_y];
    for(size_t i = 0; i < quantite_y; i++)
        z[i] = new double[quantite_x];

    ostringstream strs;

    double ybase=ymin;
    for(size_t i=0; i<quantite_y; i++)
    {
        double xbase=xmin;
        for(size_t j=0; j<quantite_x; j++)
        {
            // Evaluer la classe à laquelle appartient l'exemple
            vector<double> x={xbase,ybase};
            rdn.Evaluer(x);
            z[i][j]=rdn.sortiesReseau[rdn.sortiesReseau.size()-1][0];

            // Ecrire dans le flux
            strs << xbase << ',' << ybase << ',' << z[i][j] << endl;

            xbase+=pas;
        }
        ybase+=pas;
    }

    // Ecrire dans le fichier
    ofstream outfile (nomFichier);
    outfile << strs.str();
    outfile.close();

    // Libérer la mémoire allouée
    for(size_t i = 0; i < quantite_y; i++)
        delete [] z[i];

    delete [] z;
}
```

Listing A.1 – Code permettant de générer la limite de décision d'un réseau de neurones et de l'enregistrer dans un fichier CSV.

Annexe B

Algorithmes alternatifs à la Descente de Gradient Full-Batch

L'algorithme de rétro-propagation du gradient décrit dans la Section 3.2 suit implicitement un algorithme de Descente de Gradient nommé *Batch Gradient Descent* ou encore *Full-Batch Gradient Descent*. Ce dernier analyse d'abord tous les exemples avant d'effectuer la correction des poids. Deux variantes sont également couramment rencontrées.

B.1 La Descente de Gradient Stochastique

Principe

Le premier est la Descente de Gradient Stochastique, ou SGD pour *Stochastic Gradient Descent* en anglais. Le premier apport de SGD est le mélange des données. En effet, à chaque début d'itération, l'algorithme mélange les données afin de ne pas retrouver les mêmes séquences de correction. De plus, la correction des poids est exécutée après chaque exemple analysé, comme le fait ADALINE contrairement à la Descente de Gradient classique. Cet algorithme est en réalité peu fréquent car il diminue l'efficacité temporelle de l'algorithme. Une mise à jour des poids après chaque exemple est effectivement moins efficace qu'une mise à jour des poids après avoir itéré cents exemples [KJ18c].

Implémentation

L'Algorithme 4 présente l'algorithme général de la Descente de Gradient Stochastique. Le Listing B.1 présente l'implémentation C/C++ de la librairie **NeuroLib**. Concernant le Listing B.1, le paramètre **utiliserQuadratique** est un booléen qui, comme son nom l'indique, indique à l'algorithme quelle fonction d'erreur utiliser : la Quadratique ou celle de l'entropie croisée. Cette dernière fonction d'erreur est expliquée dans l'Annexe D. De plus, le mélange des données est pratiqué via les indexes de ces données. Les données sont représentées par leur numéro dans la file. Ce numéro est inséré dans le vecteur **indexes** dont les éléments seront mélangés à chaque début d'itération par la fonction **MelangerIndexes**. Cette méthode est efficace car elle permet de ne pas devoir manipuler les structures de données contenant les exemples et les sorties souhaitées.

```cpp
double DescenteGradientStochastique(ReseauNeurone &rdn, vector<vector<double>> &
    entrees, vector<vector<double>> &sortiesDesirees, size_t nbIterations, double
     tauxApprentissage, double seuilTolerance, bool utiliserQuadratique)
{
    vector<vector<double>> deltasW, deltasB;
    vector<vector<vector<double>>> deltasWij;
    vector<size_t> indexes;
    double erreurMoyenne=INFINITY;
    size_t nbCouches=rdn.couches.size();

    // Initialisation du tableau de deltas W et B et Wij
    CreerTableauxDeltas(deltasW, deltasB, deltasWij, rdn);

    for(size_t i=0; i<nbIterations && abs(erreurMoyenne)>seuilTolerance; i++)
    {
        MelangerIndexes(entrees.size(), indexes); // Mélanger les données
        erreurMoyenne=0.0f;

        for(size_t j=0; j<entrees.size(); j++)
        {
            // Reinitialiser les accumulateurs pour chaque exemple
            InitialiserTableauxDeltas(deltasW, deltasB, deltasWij, nbCouches);

            // FORWARD PROPAGATION
            vector<double> exemple=entrees[indexes[j]];
            rdn.Evaluer(exemple); // Présenter un vecteur d'entrée X et évaluer la
    sortie du neurone

            // BACK PROPAGATION
            erreurMoyenne += Retropropager(rdn, exemple, sortiesDesirees[indexes[j
    ]], deltasW, deltasWij, utiliserQuadratique);

            // MAJ poids à chaque exemple
            MAJPoids(rdn, deltasW, deltasWij, tauxApprentissage);
        }

        erreurMoyenne = erreurMoyenne / entrees.size();
    }

    return erreurMoyenne;
}

void MelangerIndexes(size_t nombreDonnees, vector<size_t> &indexesDst)
{
    random_device rng;
    mt19937 urng(rng());

    vector<size_t> indexes(nombreDonnees);     // Vecteur contenant les indexes
    des deux vecteurs entrees et sortieDesirees
    for(size_t i=0; i<indexes.size(); i++)
        indexes[i]=i;

    shuffle(indexes.begin(), indexes.end(), urng);     //Mélanger les indexes

    indexesDst=indexes;
}
```

Listing B.1 – L'algorithme de Descente de Gradient stochastique dans Apprentissage/apprentissage.cpp.

Algorithme 4 : Descente de Gradient Stochastique selon [KJ18c].

Entrées : rdn : le réseau de neurones à corriger comportant L couches cachées de N neurones cachés,
$\mathbf{x} = (\mathbf{x}_1, ..., \mathbf{x}_K)$: la matrice des exemples d'apprentissage, dont chaque élément $\mathbf{x}_k$ est un vecteur d'entrée,
$\mathbf{t} = (\mathbf{t}_1, ..., \mathbf{t}_K)$: la matrice contenant les réponses aux K exemples d'apprentissage, dont chaque vecteur $\mathbf{t}_k$ contient N_s éléments qui correspondent aux sorties des N_s neurones de sortie du réseau,
nbIterations : le nombre d'itérations maximum que l'algorithme peut faire,
tauxApprentissage : le taux d'apprentissage,
seuilTolérance : le seuil que l'erreur moyenne du réseau doit atteindre.
Output : erreurMoyenne : l'erreur moyenne de la couche de sortie.

```
1  erreurMoyenne = ∞
2  pour Nb itérations < nbIterations ET erreurMoyenne > seuilTolérance faire
3      erreurGlobale = 0.0
4      deltaW[Nb de couche] [Nb de neurones] = {{0}}
5      deltaBiais[Nb de couches] [Nb de neurones] = {{0}}
6      deltaPoids[Nb de couches] [Nb de neurones] [Nb Poids] = {{{0}}}
7      pour Chacun des exemples k de 1 à K faire
8          Evaluer sortiez pour l'exemple k // FORWARD PROPAGATION
           // BACK PROPAGATION
9          pour Chacun des neurones de sortie i de 1 à N_s faire
               // Accumuler deltas couche de sortie selon (3.5)
10             erreurGlobale += 0.5 * (y[k][i] - sortie[i]) // Calculer erreur globale
                   selon (3.1)
11             deltaW[L+1][i] = 0.5 * (y[k][i] - sortie[i]) // L + 1 = Indice couche de
                   sortie
12             deltaBiais[L+1][i] = deltaW[L+1][i]
               // Calculer correction des poids p de la couche de sortie
                   (équation (3.6))
13             pour Chacun des poids p allant de 1 à P du neurone de sortie i faire
14                 deltaPoids[L+1][i][p] += deltaW[L+1][i] * neurone[L][p].sortie
15             fin
16         fin
17         h = L // L = Indice dernière couche cachée
18         pour Chacune des couches cachées h allant de L à 1 faire
               // Mise à jour des poids grâce aux formules (3.7) et (3.10)
19             pour Chacun des neurones j de la couche cachée h faire
20                 neurone[h][j].biais = deltaW[h+1][j] * neurone[h][j].derivee *
                    tauxApprentissage
21                 pour Chacun des poids p du neurone j faire
22                     neurone[h][j].poids[p] += deltaW[h][j] * neurone[h-1][p].sortie *
                        tauxApprentissage
23                 fin
24             fin
25             h = h - 1
26         fin
27     fin
28     h = 1
29     erreurMoyenne = erreurGlobale/(taille entrées)
30 fin
```

B.2 La Descente de Gradient "Mini-Batch"

Principe

La Descente de Gradient Stochastique n'est qu'un cas particulier de la Descente de Gradient "Mini-Batch" où la taille du lot est unitaire. La Descente de Gradient Mini-Batch permet de spécifier à quel intervalle les poids sont corrigés, ce paramètre porte aussi le nom de *lot*. Tout comme la Descente de Gradient Stochastique, la Descente de Gradient Mini-Batch mélange les exemples à chaque début d'itération. Comme mentionné plus tôt, il est plus efficace de mettre à jour les poids après cents itérations que de faire cents mises à jour de poids consécutives. Cet algorithme est utilisé notamment afin de sauvegarder de la mémoire. Si le jeu d'apprentissage contient énormément de données, elles ne pourront sans doute pas toutes être chargées en mémoire. Cependant, en chargeant un sous-ensemble des exemples disponibles en mémoire variant à chaque itération, tous les exemples pourront probablement être parcourus dans un ordre aléatoire.

Implémentation

L'Algorithme 5 contient l'algorithme général d'une Descente de Gradient Mini-batch. Quant au Listing B.2, il présente l'implémentation de la librairie **NeuroLib**. L'implémentation fournie définit un nombre d'exemples, le lot, contenu dans la variable **tailleMiniBatch**, à parcourir avant de mettre à jour les poids. Les exemples choisis par l'algorithme, une fois les données mélangées, se fera de part et d'autre du milieu du jeu de données. Par conséquent, **tailleMiniBatch**/2 exemples seront ceux précédant le milieu du jeu de données et **tailleMiniBatch**/2 exemples seront ceux suivant le milieu du jeu de données. Ce mécanisme est représenté sur la Figure B.1.

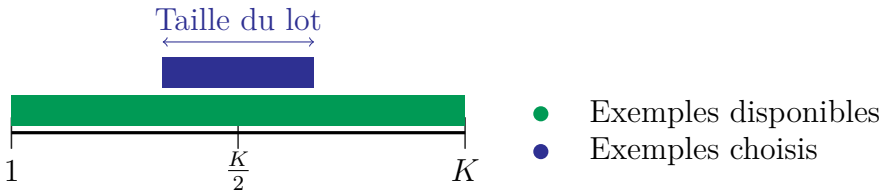


FIGURE B.1 – Mécanisme de sélection des exemples dans la Descente de Gradient Mini-Batch implémenté dans le Listing B.2.

```cpp
double DescenteGradientStochastiqueMB(ReseauNeurone &rdn, vector<vector<double>> &entrees, vector<vector<double>> &sortiesDesirees, size_t nbIterations, size_t tailleMiniBatch, double tauxApprentissage, double seuilTolerance, bool utiliserQuadratique)
{
    vector<vector<double>> deltasW, deltasB;
    vector<vector<vector<double>>> deltasWij;
    vector<size_t> indexes;
    double erreurMoyenne=INFINITY;
    size_t nbCouches=rdn.couches.size(),
    /* Centrer l'apprentissage sur le milieu du dataset. L'apprentissage va aller de :
    (Taille Dataset /2) - (Taille Minibatch /2 )   ; (Taille Dataset /2) + ( Taille Minibatch /2 )] */
    milieuDataset=(entrees.size()/2)-(tailleMiniBatch/2);

    // Initialisation du tableau de deltas W et B et Wij
    CreerTableauxDeltas(deltasW, deltasB, deltasWij, rdn);

    for(size_t i=0; i<nbIterations && abs(erreurMoyenne)>seuilTolerance; i++)
    {
        MelangerIndexes(entrees.size(), indexes); // Mélanger les données

        // Reinitialiser les accumulateurs pour chaque MiniBatch
        InitialiserTableauxDeltas(deltasW, deltasB, deltasWij, nbCouches);
        erreurMoyenne=0.0f;

        for(size_t j=0; j<tailleMiniBatch; j++)
        {
            size_t indexeActuel=indexes[milieuDataset+j];
            // FORWARD PROPAGATION
            vector<double> exemple=entrees[indexeActuel];
            rdn.Evaluer(exemple); // Présenter un vecteur d'entrée X et évaluer la sortie du neurone

            // BACK PROPAGATION
            erreurMoyenne += Retropropager(rdn, exemple, sortiesDesirees[indexeActuel], deltasW, deltasWij, utiliserQuadratique);
        }

        // MAJ poids à chaque fin de Mini-Batch
        MAJPoids(rdn, deltasW, deltasWij, tauxApprentissage);

        erreurMoyenne = (double) (erreurMoyenne / tailleMiniBatch);
    }

    return erreurMoyenne;
}
```

Listing B.2 – L'algorithme de Descente de Gradient Stochastique version Mini-Batch dans Apprentissage/apprentissage.cpp.

Algorithme 5 : Descente de Gradient Mini-Batch selon [KJ18c]

Entrées : rdn : le réseau de neurones à corriger comportant L couches cachées de N neurones cachés,
$\mathbf{x} = (\mathbf{x}_1, ..., \mathbf{x}_K)$: la matrice des exemples d'apprentissage, dont chaque élément $\mathbf{x}_k$ est un vecteur d'entrée,
$\mathbf{t} = (\mathbf{t}_1, ..., \mathbf{t}_K)$: la matrice contenant les réponses aux K exemples d'apprentissage, dont chaque vecteur $\mathbf{t}_k$ contient N_s éléments qui correspondent aux sorties des N_s neurones de sortie du réseau,
nbIterations : le nombre d'itérations maximum que l'algorithme peut faire,
tauxApprentissage : le taux d'apprentissage,
seuilTolérance : le seuil que l'erreur moyenne du réseau doit atteindre,
M : la taille du mini-lot.
Output : erreurMoyenne : l'erreur moyenne de la couche de sortie.

```
1  erreurMoyenne = ∞
2  pour Nb itérations < nbIterations ET erreurMoyenne > seuilTolérance faire
3      erreurGlobale = 0.0
4      deltaW[Nb de couche] [Nb de neurones] = {{0}}
5      deltaBiais[Nb de couches] [Nb de neurones] = {{0}}
6      deltaPoids[Nb de couches] [Nb de neurones] [Nb Poids] = {{{0}}}
7      Melanger les exemples
8      pour Chacun des exemples m de 1 à M du mini-lot faire
9          Evaluer sortiez pour l'exemple m // FORWARD PROPAGATION
           // BACK PROPAGATION
10         pour Chacun des neurones de sortie i de 1 à N_s faire
               // Accumuler deltas couche de sortie selon (3.5)
11             erreurGlobale += 0.5 * (y[m][i] - sortie[i]) // Calculer erreur globale selon (3.1)
12             deltaW[L+1][i] = 0.5 * (y[m][i] - sortie[i]) // L + 1 = Indice couche de sortie
13             deltaBiais[L+1][i] = deltaW[L+1][i]
               // Calculer correction des poids p de la couche de sortie (équation (3.6))
14             pour Chacun des poids p allant de 1 à P du neurone de sortie i faire
15                 deltaPoids[L+1][i][p] += deltaW[L+1][i] * neurone[L][p].sortie
16             fin
17         fin
18         h = L // L = Indice dernière couche cachée
19         pour Chacune des couches cachées h allant de L à 1 faire
               // Accumuler deltas couches cachées avec la formule (3.7)
20             pour Chacun des neurones j de 1 à N_c de la couche cachée h faire
21                 deltaBiais[h][j] = deltaW[h+1][j] * neurone[h][j].derivee
                   // Calculer correction des poids p des couches cachées selon formule (3.9)
22                 pour Chacun des poids p de 1 à N du neurone j faire
23                     deltaPoids[h][j][p] += deltaW[h][j] * neurone[h-1][p].sortie
24                 fin
25             fin
26             h = h - 1
27         fin
28     fin
29     h = 1
30     pour Chacune des couches cachées h de 1 à L // MAJ poids grâce à la formule (3.10)
31      faire
32         pour Chacun des neurones j de la couche cachée h faire
33             neurone[h][j].biais = deltaBiais[h][j] * tauxApprentissage
34             pour Chacun des poids p du neurone j faire
35                 neurone[h][j].poids[p] += deltaPoids[h][j][p] * tauxApprentissage
36             fin
37         fin
38         h += 1
39     fin
40     erreurMoyenne = erreurGlobale/(taille entrées)
41 fin
```

Annexe C

Vérification du gradient de l'erreur

L'algorithme de rétro-propagation du gradient de l'erreur est en fait un algorithme optimisé permettant de calculer le gradient de la fonction d'erreur à tous les niveaux. Cela signifie qu'il est donc possible de calculer ce gradient avec les méthodes numériques traditionnelles. La dérivée partielle d'une fonction f par rapport à un paramètre x est définie par le quotient différentiel suivant :

$$\frac{\partial}{\partial x} f(x) \;=\; lim_{\epsilon \to 0} \; \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

En analyse numérique, ϵ se traduit par un nombre décimal très petit, par exemple : 10^{-4}. Si on souhaite calculer numériquement la dérivée de la fonction $f(x) = x^2$ où $x = 3$, il suffit de faire :

$$\frac{\partial}{\partial x}(x^2) \;=\; \frac{(3+0.0001)^2 - 3^2}{0.0001} = 6.0001$$

Sachant que la dérivée est $(x^2)' = 2x$. Si $x = 3$, la réponse souhaitée est 6. La réponse obtenue est donc proche de celle attendue.

Afin d'améliorer ce résultat, une autre formule est très populaire, car elle fournit la stabilité numérique désirée. Il s'agit de la formule de la *différence centrale* [KJ18c] :

$$f'(x) \;\approx\; lim_{\epsilon \to 0} \; \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

Et essayons à nouveau avec la fonction $f(x) = x^2$, $x = 3$ et $\epsilon = 0.0001$:

$$f'(x) \;\approx\; \frac{f(3+0.0001)^2 - f(3-0.0001)}{2 \times 0.0001} = 6$$

Cette méthode peut donc nous permettre, *avec une précision intéressante*, de vérifier que l'algorithme de rétro-propagation implémenté, calcule correctement les gradients. Notons toutefois que le paramètre ici est un vecteur de poids. Pour calculer la dérivée, il faudra alors perturber les poids un à un avec ϵ. La dérivée calculée avec le poids perturbé correspondra alors au gradient de ce poids [Ng+13]. La méthode porte le nom des *différences finies*. [KJ18c]

Le booléen **utiliserQuadratique** permet de choisir la fonction d'erreur utilisée. En effet, deux fonctions sont susceptibles d'être employées : la fonction d'erreur quadratique initialement choisie avec la rétro-propagation du gradient dans la Section 3.2 ainsi que la fonction d'entropie croisée décrite comme une optimisation, pour les fonctions d'activation logistique, de la fonction d'erreur quadratique.

Algorithme 6 : Algorithme de l'évaluation des différences finies.

Entrées : rdn : réseau de neurones sur lequel calculer les différences finies,
epsilon : la perturbation à ajouter aux poids,
entree : un exemple sur lequel se baser pour évaluer le gradient de l'erreur,
utiliserQuadratique : spécifie la fonction d'erreur à utiliser : si **true**, la quadratique sinon l'entropie croisée.

Output : deriveesBiais[Nb de couches][Nb de neurones] : contenant les corrections à appliquer aux biais,
deriveesPoids[Nb de couches] [Nb de neurones] [Nb de poids] : contenant les corrections à appliquer aux poids.

```
1  pour Chacune des couches h faire
2      pour Chacun des neurones n faire
3          pour Chacun des poids p faire
4              Calculer + epsilon rdn.couches[h][n][p] + epsilon // Ajouter la
                   perturbation
5              Evaluer exemple dans le réseau de neurones
6              grad2 = Evaluer l'erreur avec fonction correspondante à
                utiliserQuadratique
7              rdn.couches[h][n][p] - epsilon // Rétablir le poids
8              rdn.couches[h][n][p] - epsilon // Retirer la perturbation
9              Evaluer exemple dans le réseau de neurones
10             grad1 = Evaluer l'erreur avec fonction correspondante à
                utiliserQuadratique
11             rdn.couches[h][n][p] + epsilon // Rétablir le poids
12             deriveePoids[h][n][p] = (grad2-grad1)/(2*epsilon)
13         fin
14         rdn.couches[h][n].biais + epsilon // Ajouter la perturbation au biais
15         Evaluer exemple dans le réseau de neurones
16         grad2 = Evaluer l'erreur avec fonction correspondante à utiliserQuadratique
17         rdn.couches[h][n].biais - epsilon // Rétablir le poids
18         rdn.couches[h][n].biais - epsilon // Retirer la perturbation
19         Evaluer exemple dans le réseau de neurones
20         grad1 = Evaluer l'erreur avec fonction correspondante à utiliserQuadratique
21         rdn.couches[h][n].biais + epsilon // Rétablir le poids
22         deriveeBiais[h][n] = (grad2-grad1)/(2*epsilon)
23     fin
24 fin
```

La fonction **VerifierGradient**, présentée dans le Listing C.1, calcule les corrections des poids selon les deux méthodes disponibles. Tout d'abord avec la fonction **Retropropager** présente dans le Listing 3.6 utilisée dans les implémentation classique de rétro-propagation du gradient. Mais aussi avec la méthode des différences finies utile lors des phases de *debug*. La fonction **VerifierGradient** compare ensuite les gradients obtenus afin de déterminer si ces corrections sont identiques. C'est cette méthode qui permet de valider l'implémentation de la rétro-propagation du gradient.

```
void VerifierGradient(ReseauNeurone &rdn, vector<vector<double>> &entrees,
    vector<vector<double>> &sortiesDesirees, vector<vector<double>> &deltasB,
    vector<vector<vector<double>>> &deltasWij, vector<vector<double>> &
    deriveesBiais, vector<vector<vector<double>>> &deriveesPoids, double epsilon,
     bool utiliserQuadratique)
{
    double seuilPrecision = 1e-7;
    // Initialisation des tableaux des deltas (Backprop) et derivée (Difference
    Finies)
    CreerTableauxDeltas(deriveesBiais, deltasB, deltasWij, rdn);
    CreerTableauxDeltas(deriveesBiais, deltasB, deriveesPoids, rdn);

    EvaluerDifferencesFinies(rdn, entrees[0], sortiesDesirees[0], deriveesBiais,
    deriveesPoids, epsilon, utiliserQuadratique);

    Retropropager(rdn, entrees[0], sortiesDesirees[0], deltasB, deltasWij,
    utiliserQuadratique);

    //Pour chaque poids de chaque neurone de chaque couche
    for(size_t i=0; i<rdn.couches.size(); i++)
    {
        for(size_t j=0; j<rdn.couches[i].size(); j++)
        {
            // Dérivé des poids
            for(size_t k=0; k<rdn.couches[i][j].poids.size(); k++)
            {
                cout << "C: " << i << " N: " << j << " P: " << k << " :: Delta: "
                    << deltasWij[i][j][k] << "\t Derivee: "
                    << deriveesPoids[i][j][k] << endl;

                if(abs(deriveesPoids[i][j][k]-deltasWij[i][j][k])>seuilPrecision)
                    cout << "!!! La derivee calculee et le delta du Poids " +
                        "ne respectent pas le seuil " << seuilPrecision << " !!!" <<
    endl;
            }
            cout << "C: " << i << " N: " << j << " BIAIS :: Delta: " << deltasB[i][
    j]
                << "\t Derivee: " << deriveesBiais[i][j] << endl;

            if(abs(deriveesBiais[i][j]-deltasB[i][j])>seuilPrecision)
                cout << "!!! La derivee calculee et le delta du Biais " +
                    "ne respectent pas le seuil " << seuilPrecision << " !!!" << endl
    ;
        }
    }
}

void EvaluerDifferencesFinies(ReseauNeurone &rdn, vector<double> &exemple,
    vector<double> &sortiesDesirees,
vector<vector<double>> &deriveesBiais, vector<vector<vector<double>>> &
    deriveesPoids, double epsilon, bool utiliserQuadratique)
{
```

```
41      //Pour chaque poids de chaque neurone de chaque couche
        for(size_t i=0; i<rdn.couches.size(); i++)
43      {
            for(size_t j=0; j<rdn.couches[i].size(); j++)
45          {
                // Dérivé des poids
47              for(size_t k=0; k<rdn.couches[i][j].poids.size(); k++)
                {
49                  double grad2=CalculerErreurPerturbation(rdn,
                        &rdn.couches[i][j].poids[k], epsilon, exemple,
51                      sortiesDesirees, utiliserQuadratique);

53                  double grad1=CalculerErreurPerturbation(rdn,
                        &rdn.couches[i][j].poids[k], -epsilon, exemple,
55                      sortiesDesirees, utiliserQuadratique);

57                  deriveesPoids[i][j][k] = FonctionDifferenceCentrale(grad1,
                        grad2, epsilon);
59              }

61              // Dérivée du biais
                double grad2=CalculerErreurPerturbation(rdn, &rdn.couches[i][j].biais,
63                  epsilon, exemple, sortiesDesirees, utiliserQuadratique);

65              double grad1=CalculerErreurPerturbation(rdn, &rdn.couches[i][j].biais,
                    -epsilon, exemple, sortiesDesirees, utiliserQuadratique);
67
                deriveesBiais[i][j] = FonctionDifferenceCentrale(grad1, grad2, epsilon)
    ;
69          }
        }
71  }

73  double CalculerErreurPerturbation(ReseauNeurone &rdn, double *poidsAModifier,
        double epsilon, vector<double> &exemple, vector<double> &sorties, bool
        utiliserQuadratique)
    {
75      size_t indiceCoucheSortie=rdn.couches.size()-1;
        vector<Neurone> &neuroneSortie=rdn.couches[indiceCoucheSortie];
77
        *poidsAModifier += epsilon; // Ajouter la perturbation au poids
79
        rdn.Evaluer(exemple); // Evaluer le RDN pour l'exemple
81      // Calcule la dérivée de l'erreur
        double df=CalculerErreurGlobale(neuroneSortie, sorties,
83          utiliserQuadratique, rdn.isSoftmax);

85      *poidsAModifier -= epsilon; // Enlever la perturbation au poids

87      return df;
    }
89
    double FonctionDifferenceCentrale(double grad1, double grad2, double epsilon)
91  {
        return -(grad2-grad1)/(2*epsilon); // "-" car dans le GD, on RETIRE la pente
        .
93  }
```

Listing C.1 – Code permettant calculer avec la méthode classique de l'analyse numérique, les dérivées de l'algorithme de rétro-propagation.

La précision atteinte par le calcul numérique de la dérivée est de l'ordre du ϵ utilisé. Ainsi, si $\epsilon = 10^{-4}$, la précision sera de 10^{-4}. Dans le cas de l'exemple présent dans le Listing C.2, $\epsilon = 10^{-7}$ et le réseau de neurones possède 6 couches, dont 5 couches cachées, chacune contenant 3 neurones cachés, ainsi qu'un neurone de sortie. Les gradients calculés selon les méthodes de rétro-propagation et des différences finies correspondent parfaitement.

```
void EssaiGradientCheck()
{
    vector<vector<double>> deltasB, deriveesBiais;
    vector<vector<vector<double>>> deltasWij, deriveesPoids;
    vector<vector<double>> x,y,y1;
    size_t nbEntrees=2;

    LireFichierCSV("/home/julien/workspace/NeuroLibTest/moon.csv",
       x, nbEntrees, y, 1);

    for(double yi : y[0])
    {
       vector tmp;
       tmp.push_back(yi);
       y1.push_back(tmp);
    }

    ReseauNeurone rdn(2, Logistique, LogistiqueDerivee, 3, 5, 1);
    VerifierGradient(rdn, x, y, deltasB, deltasWij, deriveesBiais,
       deriveesPoids, 1e-7, true);
}

// C: pour couche, N: pour neurone, P: pour poids et BIAIS pour le biais
// C: 0 N: 0 P: 0 :: Delta: 3.21878e-06 Derivee: 3.21965e-06
// C: 0 N: 0 P: 1 :: Delta: 7.83049e-06 Derivee: 7.83151e-06
// C: 0 N: 0 BIAIS :: Delta: 7.14553e-06  Derivee: 7.14762e-06
// C: 0 N: 1 P: 0 :: Delta: 2.81871e-06 Derivee: 2.81775e-06
// C: 0 N: 1 P: 1 :: Delta: 6.85722e-06 Derivee: 6.85674e-06
// C: 0 N: 1 BIAIS :: Delta: 6.25739e-06  Derivee: 6.25722e-06
// C: 0 N: 2 P: 0 :: Delta: -2.47435e-06  Derivee: -2.4758e-06
// C: 0 N: 2 P: 1 :: Delta: -6.01947e-06  Derivee: -6.01741e-06
// C: 0 N: 2 BIAIS :: Delta: -5.49293e-06 Derivee: -5.49227e-06
[ ... ]
// C: 5 N: 0 P: 0 :: Delta: -0.0434242  Derivee: -0.0434242
// C: 5 N: 0 P: 1 :: Delta: -0.0489789  Derivee: -0.0489789
// C: 5 N: 0 P: 2 :: Delta: -0.0317435  Derivee: -0.0317435
// C: 5 N: 0 BIAIS :: Delta: -0.0498676 Derivee: -0.0498676
```

Listing C.2 – Essai de la vérification de gradient

Annexe D

La fonction d'activation Softmax et la fonction d'erreur d'entropie croisée

D.1 La fonction Softmax

Présentation

Contrairement aux fonctions d'activation classiques qui s'utilisent sur un seul neurone, la fonction Softmax s'applique à une *couche* de neurones. Sa formule générale est [KJ18a] :

$$S_i(\mathbf{x}) = \frac{e^{x_i}}{\sum_{k=1}^{N} e^{x_k}} \text{ , où } \mathbf{x} \text{ dénote un vecteur d'entrées.}$$

Où $\mathbf{x}$ désigne un vecteur de N valeurs. La fonction exponentielle transforme l'entrée en une valeur qui sera toujours positive. La fonction Softmax divise donc cette exponentielle par la somme des exponentielles des valeurs de ce vecteur. Cela a pour impact de normaliser les sorties dans un intervalle $[0, 1]$. Aussi, plus la valeur x_i est importante, plus la sortie de la fonction Softmax sera proche de l'unité. Plus généralement, la fonction Softmax permet d'interpréter les différentes du vecteur comme des *probabilités*. Dès lors, la somme de chacune de ces probabilités, correspondant à l'application de la fonction Softmax sur chacun des éléments du vecteur $\mathbf{x}$ vaut 1 [KJ18a] :

$$\sum_{i=0}^{N} S_i(\mathbf{x}) = 1$$

Dans le cadre de la classification avec des réseaux de neurones, cette fonction d'activation présente un attrait évident. Tout d'abord, elle ne sera utilisée que sur la couche de sortie d'un réseau. Grâce à elle, le réseau de neurones fournit un classement des classes les plus probables correspondant à l'entrée qui lui est présenté. Appliquée à un réseau de neurone, la variable vecteur $\mathbf{x}$ de la fonction Softmax correspond aux potentiels ν des neurones de la couche de sortie. La fonction Softmax peut alors s'écrire :

$$y_i(\boldsymbol{\nu}) = \frac{e^{\nu_i}}{\sum_{k=1}^{N} e^{\nu_k}}$$

Où :

- y_i désigne la sortie du $i$ème neurone de la couche de sortie.
- ν désigne les potentiels des neurones de la couche de sortie sous forme de vecteur.
- ν_i correspond au potentiel du $i$ème neurone de sortie.
- La couche de sortie comporte N neurones.
- $\sum_{k=1}^{N} e^{\nu_k}$ est la somme des potentiels des neurones appartenant à la couche de sortie.

Dérivation de la fonction Softmax

Un réseau de neurones utilisant la fonction Softmax doit connaître la dérivée de cette fonction d'activation pour pouvoir utiliser la rétro-propagation du gradient de l'erreur. Celle-ci se calcule, pour un neurone i en fonction du potentiel d'un neurone j, tous deux issus de la couche de sortie S. Cette dépendance est due à la normalisation par le dénominateur de la fonction Softmax [Ren14]. Considérons pour l'instant le cas $i = j$:

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = \frac{\partial}{\partial \nu_j}\left[\frac{e^{\nu_i}}{\sum_{k=1}^{N} e^{\nu_k}}\right]$$

La formule de la dérivée d'un quotient est :

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x) \ - \ f(x)g'(x)}{(g(x))^2}$$

Or les dérivées des termes de la somme sont nuls pour tout $k \neq j$ car considérés comme constants.

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = \frac{e^{\nu_i}\sum_{k=1}^{N} e^{\nu_k} \ - \ e^{\nu_i}e^{\nu_j}}{(\sum_{k=1}^{N} e^{\nu_k})^2} \tag{D.1}$$

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = \frac{e^{\nu_j}}{\sum_{k=1}^{N} e^{\nu_k}}\left(\frac{\sum_{k=1}^{N} e^{\nu_k}}{\sum_{k=1}^{N} e^{\nu_k}} \ - \ \frac{e^{\nu_j}}{\sum_{k=1}^{N} e^{\nu_k}}\right)$$

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = S_i(\nu_j)(1 - S_j(\nu_j)) \tag{D.2}$$

Si $i = j$, l'expression de la dérivée de Softmax est similaire à celle de la fonction **Logistique**. Cependant, si $i \neq j$, cette relation n'est plus vraie [Ren14]. Reprenons à partir de l'Équation (D.1) de la dérivée pour justifier cela :

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = \frac{e^{\nu_i}\sum_{k=1}^{N} e^{\nu_k} \ - \ e^{\nu_i}e^{\nu_j}}{(\sum_{k=1}^{N} e^{\nu_k})^2}$$

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = \frac{0 \ - \ e^{\nu_i}e^{\nu_j}}{(\sum_{k=1}^{N} e^{\nu_k})^2}$$

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = - \ \frac{e^{\nu_i}}{\sum_{k=1}^{N} e^{\nu_k}} \ \frac{e^{\nu_j}}{\sum_{k=1}^{N} e^{\nu_k}}$$

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = -S_i \ S_j \tag{D.3}$$

Ces deux cas sont couremment regroupés ensemble dans la littérature en introduisant la notation *Delta* de Kronecker δ_{ij} où [Ren14] :

— $\delta_{ij} = 1$ si $i = j$.
— $\delta_{ij} = 0$ si $i \neq j$.

La dérivée de la fonction Softmax s'écrit [Ren14] :

$$\frac{\partial S_i'(\boldsymbol{\nu})}{\partial \nu_j} = S_i(\delta_{ij} - S_j) \tag{D.4}$$

D.2 L'entropie croisée comme fonction d'erreur

D.2.1 Rappels sur l'entropie de Shannon

L'entropie de Shannon désigne la quantité d'information inconnue et donc intéressante obtenue dans une expérience quelconque (discussion, pile ou face ...). Soit I, la quantité d'information est une fonction ayant les propriétés suivantes [Wag16] :

1. I est une fonction continue de la probabilité p_k, d'avoir le symbole x_k.
2. I croit si p_k décroit : plus un symbole est fréquent, moins il apporte d'information.
3. $I(p_i, p_j) = I(p_i) + I(p_j)$ où $I(p_i, p_j)$ est la quantité d'information des deux symboles successifs x_i et x_j.
4. Un symbole certain, s'il existe, possède une quantité d'information nulle : $I(pk) = 0$ si$pk = P(X = x_k) = 1$.

La relation $I(x_k) = -log(p_k) = log_b(\frac{1}{p_k})$ satisfait cette relation et définit la quantité d'information qu'apporte un symbole x_k de probabilité p_k. La base du logarithme b est le nombre de symboles dans l'alphabet.

L'entropie d'une source se définit comme étant l'espérance de la quantité d'information [Wag16] :

$$H(S) = E[I(X)] = \sum_{k=1}^{K} p_k I(x_k) = -\sum_{k=1}^{K} p_k log(p_k)$$

D.2.2 L'entropie croisée

L'entropie croisée ou entropie croisée de deux distributions de probabilités p et q concernant une variable aléatoire X se caractérise par la relation [Pre01] :

$$H(p, q) = -\sum_{x} p(x) log[q(x)]$$

Elle mesure le nombre de bits moyen nécessaire pour coder un événement issu de q si on utilise le "mauvais" codage q au lieu de p sur un ensemble d'évènements x. Cette mesure est utile en machine learning pour mesurer la similarité de deux distributions. La première distribution est empirique et correspond aux réponses souhaitées du dataset d'apprentissage. La distribution est estimée et correspond aux sorties du réseau de neurones [Pre01].

Par exemple, la distribution de l'ensemble d'apprentissage correspond à la distribution p. Dans le cas d'une classification, la distribution de l'ensemble d'apprentissage doit être vue comme un *one hot vector*, c'est-à-dire un vecteur contenant autant de valeurs que de classes et où chaque valeur représente la probabilité qu'un exemple appartienne à une classe. Dans le cas présent, nous considérons que les classes sont mutuellement exclusives : un exemple appartient à une et une seule classe. Ceci se traduit par une valeur de 100% pour cette classe et de 0% pour les autres. Quant à la seconde distribution q, elle correspond à l'ensemble des valeurs prédites par le réseau de neurones pour chacun des exemples.

Soient deux classes étiquetées 0 et 1. Le modèle assigne alors les probabilités complémentaires [Pre01] :

— $q(y = 1) = \hat{y}$ avec y la sortie désirée pour cet exemple et $\hat{y}$ la sortie prédite par la fonction d'activation.
— $q(y = 0) = (1 - \hat{y})$

La même notation peut être utilisée pour p, dont les paramètres sont bien définis et non estimés comme pour la distribution q [Pre01] :

— $p(\hat{y} = 1) = y$

— $p(\hat{y} = 0) = (1 - y)$

Pour un exemple quelconque parmi ceux disponibles, l'entropie croisée de ce dernier est :

$$H(p, q) = -\sum_i p_i \ log(q_i) = -y \ log(\hat{y}) - (1 - y) \ log(1 - \hat{y})$$

Remarquons que :

— Si la probabilité que la classe correspondante est nulle, alors le premier terme s'annule et l'entropie croisée vaut : $H(p, q) = -log(1 - \hat{y})$.

— De même, si la probabilité de la classe correspondante vaut 1, alors le second terme s'annule et l'entropie croisée vaut : $H(p, q) = -log(\hat{y})$.

Deux propriétés en particulier permettent de l'interpréter comme une fonction d'erreur [Nie20].

1. H est toujours strictement positive : $H > 0$. Tous les termes sont négatifs étant donné que les valeurs injectées dans les logarithmes sont comprises entre 0 et 1, il s'agit en effet de probabilités. De plus, le résultat d'un logarithme d'une valeur négative, est positif.
2. Plus le résultat émis par le RDN est proche du résultat attendu, plus l'erreur sera petite. Et a contrario, plus le résultat émis par le RDN est erroné, plus l'erreur sera grande.

Tout comme la fonction d'erreur quadratique, l'erreur globale et l'erreur moyenne de la couche de sortie d'un réseau de neurones sont également définies.

$$H(\mathbf{w}, \mathbf{x}) = \sum_{k=0}^{K} \sum_{i=0}^{N} \ t_{ki} \ log(y_{ki}) \tag{D.5}$$

Dans le cas d'un réseau de neurones, la formule générale de l'erreur moyenne d'un réseau de neurone s'écrit :

$$\overline{H}(\mathbf{w}, \mathbf{x}) = -\frac{1}{K} \sum_{k=0}^{K} \sum_{i=0}^{N} \ t_{ki} \ log(y_{ki}) \tag{D.6}$$

Pour un ensemble d'exemples de taille K, et d'un réseau n'ayant que deux neurones de sortie, l'erreur moyenne s'énonce comme [Nie20] :

$$\overline{H}(\mathbf{w}, \mathbf{x}) = -\frac{1}{K} \sum_{k=0}^{K} \ t_k \ log(y_k) + (1 - t_k) \ log(1 - y_k) \tag{D.7}$$

Où :

— N définit le nombre de neurones de sortie.

— k est l'indexe des exemples. Il y a K exemples considérés.

— t_{ki} : la sortie désirée correspondante à l'exemple k analysé pour le $i$ème neurone de sortie ;

— y_k : la sortie prédite par le réseau de neurones pour l'exemple k.

L'exemple consiéré ne possède que deux classes, ce qui est le minimum. Une bonne pratique est d'avoir tout de même deux neurones de sorties bien distinct. Chaque neurone de sortie correspondra alors à une classe. Toutefois, même si le réseau ne comporte qu'un seul neurone de sortie, l'Équation D.7 reste d'application puisqu'il y a tout de même deux classes.

D.2.3 La rétro-propagation du gradient avec l'entropie croisée

Comme mentionné dans la section précédente, l'entropie croisée concerne deux distributions de probabilités. C'est pourquoi cette fonction d'erreur est couremment conjointement avec la fonction Softmax ou la fonction **Logistique**. Cependant, afin de pouvoir l'utiliser, le calcul du gradient de l'erreur E doit être adapté. Nous considérons un simple exemple afin de simplifier.

Soit $y(\mathbf{w}, \mathbf{x})$ la sortie d'un réseau de neurones à un seul neurone de sortie correspondant à un exemple quelconque et t, la sortie souhaitée pour cet exemple. La fonction d'erreur de ce réseau pour une classification s'écrit [Nie20]

$$E(\mathbf{w}, \mathbf{x}) = -t \; log[y(\mathbf{w}, \mathbf{x})] \; - \; (1-t) \; log[1 - y(\mathbf{w}, \mathbf{x})] \tag{D.8}$$

Calcul du gradient de l'erreur pour la couche de sortie

L'expression générale du gradient de l'erreur Δw_{pi}^S d'un poids p appartenant au neurone i de la couche de sortie S est donné par [Nie20]

$$\Delta w_{pi}^S = \frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pi}^S} = \frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} \frac{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pi}^S}$$

Le premier terme devient :

$$\begin{aligned}
\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} &= \frac{\partial}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} \left[-t_i \; log(y_i^S(\mathbf{w}, \mathbf{x})) \right] - \frac{\partial}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} \left[(1-t_i) \; log(1 - y_i^S(\mathbf{w}, \mathbf{x})) \right] \\
&= \left(\phi_i^S(\nu_i^S) \right)' \; \frac{-t_i}{y_i^S(\mathbf{w}, \mathbf{x})} \; + \; \left(\phi_i^S(\nu_i^S) \right)' \; \frac{1 - t_i}{1 - y_i^S(\mathbf{w}, \mathbf{x})} \\
&= \left(\phi_i^S(\nu_i^S) \right)' \; \frac{-t_i + \cancel{t_i \, y_i^S(\mathbf{w}, \mathbf{x})} - \cancel{t_i \, y_i^S(\mathbf{w}, \mathbf{x})} + y_i^S(\mathbf{w}, \mathbf{x})}{y_i^S(\mathbf{w}, \mathbf{x})(1 - y_i^S(\mathbf{w}, \mathbf{x}))}
\end{aligned}$$

L'expression du premier terme du gradient de l'entropie croisée est [Nie20] :

$$\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} = \left(\phi_i^S(\nu_i^S) \right)' \; \frac{y_i^S(\mathbf{w}, \mathbf{x}) - t_i}{y_i^S(\mathbf{w}, \mathbf{x})(1 - y_i^S(\mathbf{w}, \mathbf{x}))}$$

Si $\phi(\nu)$ désigne la fonction **Logistique** $\sigma(x)$, dont la dérivée, présentée dans le Chapitre 3, se simplifie [Ren14]

$$\sigma'(x) = \sigma(x) \, (1 - \sigma(x))$$

Alors l'expression du gradient de l'erreur devient :

$$\Delta w_{pi}^S = \cancel{y_i^S(\mathbf{w}, \mathbf{x})(1 - y_i^S(\mathbf{w}, \mathbf{x}))} \frac{y_i^S(\mathbf{w}, \mathbf{x}) - t_i}{\cancel{y_i^S(\mathbf{w}, \mathbf{x})(1 - y_i^S(\mathbf{w}, \mathbf{x}))}} = y_i^S(\mathbf{w}, \mathbf{x}) - t_i$$

À nouveau, la notation de l'erreur local est introduite. Pour rappel, elle s'écrit δ_i^S pour le $i$ème neurone de la couche S :

$$\delta_i^S = (y_i^S(\mathbf{w}, \mathbf{x}) - t_i) \tag{D.9}$$

La façon de calculer la somme pondérée des entrées n'ayant guère changé, le second terme est identique à celui de la fonction d'erreur quadratique qui est [Ren14]

$$\frac{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pi}^S} = x_{pi}^S$$

Le calcul du gradient de l'erreur de la fonction d'entropie croisée appliquée à la couche de sortie d'un réseau de neurones utilisant la fonction **Logistique** se résume à :

$$\Delta w_{pi}^{S} = (y_i^S(\mathbf{w}, \mathbf{x}) - t) x_{pi}^{S} = \delta_i^S \; x_{pi}^{S}$$

Remarquons que pour l'expression du gradient de l'erreur des nœuds de sortie, la dérivée de la fonction d'activation n'est plus requise. L'entropie croisée s'affranchit de ce terme qui ralentit la convergence et devient directement proportionnelle à la différence entre la sortie souhaitée et prédite [Nie20].

Calcul du gradient de l'erreur pour une couche cachée

Dans le Chapitre 3, le gradient d'un poids p d'un neurone i appartenant la couche cachée H précédant la couche de sortie $S = H + 1$ est énoncé comme étant [Ren14]

$$\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial w_{pj}^H} = \sum_{i=1}^{N} \frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} \frac{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}{\partial y_j^H(\mathbf{w}, \mathbf{x})} \frac{\partial y_j^H(\mathbf{w}, \mathbf{x})}{\partial w_{pj}^H}$$

Le premier terme de la somme a déjà été calculé dans l'Équation D.9 et vaut :

$$\frac{\partial E_i^S(\mathbf{w}, \mathbf{x})}{\partial \nu_i^S(\mathbf{w}, \mathbf{x})} = \delta_i^S$$

Afin de calculer le second terme de la somme, il ne faut pas oublier que la *sortie* y_j du $j$ème neurone de la couche H correspond à la $j$ème *entrée* du neurone i de la couche de sortie S. Le potentiel n'étant qu'une somme, tous les termes où $i \neq j$ sont considérés comme constants et il ne reste que le terme où $i = j$ [Ren14] :

$$\frac{\partial \nu_i^S(\mathbf{w}, \mathbf{x})}{\partial y_j^H(\mathbf{w}, \mathbf{x})} = w_{ji}^S$$

Enfin, le dernier terme de la somme correspond à la sortie du neurone caché j de la couche H avec son $p$ème poids.

$$\frac{\partial y_j^H(\mathbf{w}, \mathbf{x})}{\partial w_{pj}^H} = \left(\phi_j^H(\nu_j^H)\right)' \; x_{pj}$$

La formulation finale est :

$$\begin{aligned} \Delta w_{pi}^H &= \sum_{i=1}^{N} \delta_i^S \; w_{ji}^S \; \left(\phi_j^H(\nu_j^H)\right)' \; x_{pj} \\ \leftrightarrow \; \Delta w_{pi}^H &= \left(\phi_j^H(\nu_j^H)\right)' \; x_{pj} \; \sum_{i=1}^{N} \delta_i^S \; w_{ji}^S \end{aligned}$$

Le calcul du gradient de l'erreur des couches cachées avec la fonction d'entropie croisée est identique à celui de la fonction quadratique ! Seule la formule de δ_i^S change et ce, uniquement pour la couche de sortie.

D.2.4 Calcul du gradient de l'erreur avec la fonction Softmax

L'équation du gradient de l'erreur de l'entropie croisée avec un réseau de neurones à deux neurones de sorties utilisant la fonction Softmax sur sa couche de sortie s'écrit :

$$\Delta w_{pi}^S = \frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial w_{pi}^S} = \frac{\partial E_i^S(\mathbf{w},\mathbf{x})}{\partial \nu_i^S(\mathbf{w},\mathbf{x})} \frac{\partial \nu_i^S(\mathbf{w},\mathbf{x})}{\partial w_{pi}^S}$$

Le réseau présente donc deux sorties prédites y_1^S et y_2^S qu'il compare à deux sorties souhaitées, respectivement t_1 et t_2. Pour rappel, ces sorties souhaitées sont les valeurs d'un *one hot encoded vector* $\mathbf{t}$ ayant la valeur 1 pour la classe représentée par l'exemple et 0 pour toutes les autres. Le premier terme devient :

$$\begin{aligned}\frac{\partial E^S(\mathbf{w},\mathbf{x})}{\partial \nu^S(\mathbf{w},\mathbf{x})} &= \frac{\partial}{\partial \nu^S(\mathbf{w},\mathbf{x})} \left(-t_1\ log[y_1^S(\mathbf{w},\mathbf{x})]\ -\ t_2\ log[y_2^S(\mathbf{w},\mathbf{x})]\right) \\ &= -\frac{t_1}{y_1^S(\mathbf{w},\mathbf{x})}(\phi_1^S(\nu_1^S))' - \frac{t_2}{y_2^S(\mathbf{w},\mathbf{x})}(\phi_2^S(\nu_2^S))'\end{aligned}$$

Où $(\phi_i^S(\nu_i^S))'$ désigne, dans ce cas, la dérivée de la fonction d'activation du neurone i considéré, appartenant à la couche de sortie S dont la formule se trouve dans l'Équation (D.4). L'expression du premier terme du gradient du neurone 1 devient :

$$\frac{\partial E^S(\mathbf{w},\mathbf{x})}{\partial \nu^S(\mathbf{w},\mathbf{x})} = -\frac{t_1}{y_1^S(\mathbf{w},\mathbf{x})} S_1(\boldsymbol{\nu})\left[1 - S_1(\boldsymbol{\nu})\right]\ - \frac{t_2}{y_2^S(\mathbf{w},\mathbf{x})}\left[-S_1(\boldsymbol{\nu})S_2(\boldsymbol{\nu})\right]$$

Dans l'équation ci-dessus, il est important de se rappeler que $S_1 = y_1^S$ et $S_2 = y_2^S$. L'équation peut être simplifiée :

$$\begin{aligned}\frac{\partial E^S(\mathbf{w},\mathbf{x})}{\partial \nu^S(\mathbf{w},\mathbf{x})} &= -\ \left[t_1\ (1 - S_1(\boldsymbol{\nu}))\right]\ +\ \left[t_2\ S_1(\boldsymbol{\nu})\right] \\ &= -t_1\ +\ t_1 S_1(\boldsymbol{\nu})\ +\ t_2 S_1(\boldsymbol{\nu}) \\ &= -t_1 + S_1(\boldsymbol{\nu})(t_1 + t_2)\ \ \text{[Ren14]}\end{aligned}$$

Étant donné que $\mathbf{t}$ est un one encoded vector, la somme de ses éléments vaut 1. En considérant, dans ce cas, que $t_1 = 1$ alors $t_2 = 0$ et l'équation devient [Ren14] :

$$\frac{\partial E^S(\mathbf{w},\mathbf{x})}{\partial \nu^S(\mathbf{w},\mathbf{x})} = S_1(\boldsymbol{\nu}) - t_1$$

Quant au second terme, il reste identique. L'équation générale pour calculer le gradient du poids p du neurone i la couche de sortie S s'énonce comme suit [Ren14] :

$$\Delta w_{pi}^S = (y_i^S - t_i) x_p^S$$

D.2.5 Implémentation

Implémenter l'entropie croisée pour la fonction logistique ne demande pas beaucoup de modification puisque seule l'expression des nœuds de sortie est modifiée. Il est également nécessaire

de prendre en compte si la fonction Softmax est utilisée dans la couche de sortie, car sa dérivée est particulière.

Algorithme 7 : Calcule des gradients des poids avec la fonction d'erreur de l'entropie croisée.

Entrées : coucheNeurones = $(Neurone_1, ..., Neurone_N)$: une couche de neurones, sous forme de vecteur, contenant les N neurones de sortie du réseau,
$\mathbf{x} = (x_1, ..., x_K)$: le vecteur d'entrées de la couche de sortie, c'est-à-dire les sorties de la dernière couche cachée,
$\mathbf{t} = (t_1, ..., t_N)$: la matrice contenant les réponses des N neurones de sorties du réseau,
Output : $\delta\mathbf{B} = (\delta B_1, ..., \delta B_N$: vecteur contenant les corrections à apporter aux biais des N neurones de sortie,
$\delta W_{np} = ((\delta W_{10}, ..., \delta W_{1K}), ..., (\delta W_{N0}, ..., \delta W_{NK}))$: tableau deux dimensions contenant les corrections à apporter aux K poids des N neurones de la couche de sortie,
erreurGlobale : l'erreur globale (de tous les neurones de sortie) calculée avec la fonction d'entropie croisée.

```
1 erreurGlobale=0.0
2 pour Chacun des neurones n allant de 1 à N faire
3 |   δ = (t[n]-coucheNeurones[n].y) ;        // Calculer l'erreur locale grâce à la
  |    formule (D.9)
4 |   δB[n] = δ
5 |   pour Chacun des poids k allant de 1 à K du neurone n faire
6 |   |   δW[n][k] = δ * x[k]
7 |   fin
8 |   erreurGlobale += - t[n]*log(coucheNeurones[n].y) // Calculer l'erreur globale
  |      grâce à la formule (D.5)
9 fin
```

Le code C/C++ de la rétro-propagation du gradient de l'erreur a dû être adapté avec la fonction d'erreur d'entropie croisée disponible dans le fichier Apprentissage/apprentissage.cpp. Pour ce faire, un booléen **utiliserQuadratique** a été défini et indique si la fonction d'erreur à choisir est la quadratique (**true**) ou celle de l'entropie croisée (**false**) :

```
1  double CalculerErreursCoucheSortie(vector<Neurone> &src, vector<double> &entrees
       , vector<double> &sortieDesiree, vector<double> &deltasW, vector<vector<
       double>> &deltasWij, bool utiliserQuadratique)
   {
3    double erreurGlobale=0.0f;

5    for(size_t i=0; i<deltasW.size(); i++) // Pour chaque neurone de la couche
     {
7      if(utiliserQuadratique)
         erreurGlobale += CorrectionPerteQuadratique(src[i], sortieDesiree[i],
       deltasW[i]);
9      else //sinon utiliser Entropie Relative
         erreurGlobale += CorrectionPerteEntropieCroisee(src[i], sortieDesiree[i],
       deltasW[i]);
11
       for(size_t k=0; k<src[i].poids.size(); k++)
13       deltasWij[i][k] += deltasW[i] * entrees[k];
     }
15
     return erreurGlobale;
17 }
```

Listing D.1 – La fonction calculant l'erreur des neurones de sortie avec la cross-entropy dans Apprentissage/apprentissage.cpp.

```
double  CorrectionPerteEntropieCroisee(Neurone &n, double sortieDesiree, double
    &deltasW)
{
  deltasW = (sortieDesiree-n.y);
  return FonctionPerteEntropieCroisee(n.y, sortieDesiree);
}

double FonctionPerteEntropieCroisee(double sortieReelle, double sortieDesiree)
{
  return -(sortieDesiree * log(sortieReelle) + (1-sortieDesiree) * log(1-
    sortieReelle));
}
```

Listing D.2 – Les fonctions d'erreurs permettant de calculer l'erreur ainsi que la corresction. Ces fonctions sont définies dans Fonction/fonctionerreur.cpp.

```
#include <Neurone/ReseauNeurone.cpp>
#include <Apprentissage/apprentissage>

void main()
{
 cout << "EssaiRDN_Moon_CrossEntropy" << endl;
 size_t nbEntrees=2;
 vector<vector<double>> x, y, y1;
 LireFichierCSV("/home/julien/workspace/NeuroLibTest/moon_class0.csv", x,
    nbEntrees, y, 1);

 ReseauNeurone rdn(nbEntrees, Logistique, 9, 1, 1);
 RetropropagationGradient(rdn, x, y, 50, 0.009, 0.01);

 size_t nbErreurs=0;
 double seuil=0.1;
 for(size_t i=0; i<x.size(); i++)
 {
 vector<double> exemple=x[i];
 rdn.Evaluer(exemple);
 double sortie=rdn.sortiesReseau[rdn.sortiesReseau.size()-1][0],
  output=0.0f;
 cout << i << " Resultat: " << rdn.sortiesReseau[rdn.sortiesReseau.size()-1][0]
    << endl;

 if(sortie>0.5)
 output=1.0f;

 if(output!=y1[i][0])
 nbErreurs++;
 }
 cout << "Nombre d erreurs (seuil =" << seuil << "): " << nbErreurs << endl;

 //Pour pouvoir tracer les points dans excel/calc
 double xmin = -3, xmax=3,
 ymin=-3, ymax=3,
 pas=0.1;

 EnregistrerLimiteDecision("moon_BL_CE.csv", xmin, xmax, ymin, ymax, pas, rdn);
}
```

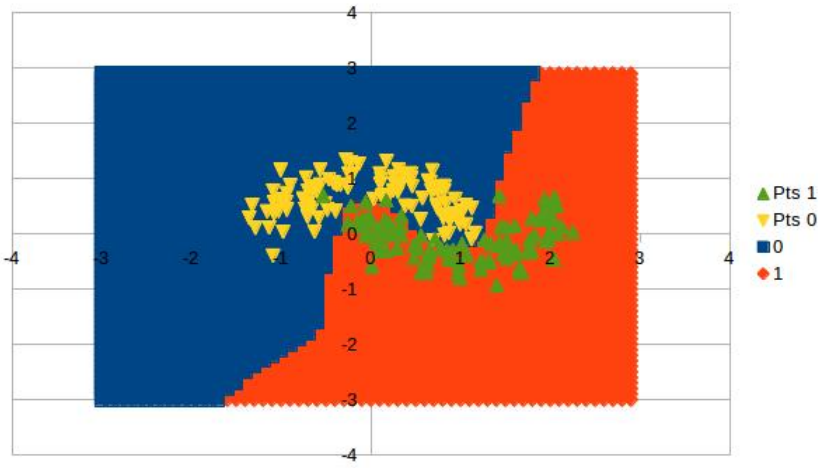


FIGURE D.1 – La limite de décision du RDN entraîné avec la cross-entropy.

```
40 // Iteration 4999 Erreur Moyenne=0.0548779
   // Nombre d erreurs (seuil =0.1): 3
```

Listing D.3 – Entrainement d'un RDN avec la fonction d'entropie croisée.

Les tests fait avec la fonction d'erreur d'entropie croisée ne sont pas concluant. En effet, l'erreur décroît moins vite qu'avec la fonction quadratique, parfois d'un facteur 10, cela dépend des cas et du taux d'apprentissage. De plus, le seuil de 0.01 n'est jamais atteint, même avec 5 000 itérations sur l'ensemble du learning set. Toutefois, le nombre d'erreurs dans la classification post-apprentissage est très bon. De plus, sur la limite de décision tracée sur la Figure D.1, on voit qu'elle s'adapte déjà très, voir trop, bien à la forme que prennent les données. Un seuil aussi bas n'est peut-être donc pas nécessaire également. Enfin, la fonction d'entropie croisée est aussi destinée à être utilisée avec la fonction Softmax, que je n'ai pas encore eu le temps de terminer. Les tests fait avec la fonction d'erreur d'entropie croisée ne sont pas concluant. En effet, l'erreur décroît moins vite qu'avec la fonction quadratique, parfois d'un facteur 10, cela dépend des cas et du taux d'apprentissage. De plus, le seuil de 0.01 n'est jamais atteint, même avec 5 000 itérations sur l'ensemble du learning set. Toutefois, le nombre d'erreurs dans la classification post-apprentissage est très bon. De plus, sur la limite de décision tracée sur la Figure D.1, on voit qu'elle s'adapte déjà très, voir trop, bien à la forme que prennent les données. Un seuil aussi bas n'est peut-être donc pas nécessaire également. Enfin, la fonction d'entropie croisée est aussi destinée à être utilisée avec la fonction Softmax, que je n'ai pas encore eu le temps de terminer.

Annexe E

Rappels sur l'Analyse en Composantes Principales

L'Analyse en Composantes Principales, *PCA* pour Principal Component Analysis en anglais, trouve son utilité pour des données quantitatives. L'idée de cette technique de data mining est de proposer une représentation dans un espace de dimensions réduites permettant de mettre en évidence d'éventuelles structures internes aux données qui lui sont soumises. La projection des données sur un plan à deux dimensions où le premier axe est celui ayant la plus grande inertie et le second celui ayant la seconde plus grande inertie [Vil17].

Soit un tableau de donnée $\mathbf{X}$, possédant P variables quantitatives et N observations, ce tableau est représentaté sous forme de matrice de dimensions $N \times P$ [Vil17].

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & ... & x_{1P} \\ x_{21} & x_{22} & ... & x_{2P} \\ ... & ... & ... & ... \\ x_{N1} & x_{N2} & ... & x_{NP} \end{bmatrix}$$

Dans un premier temps, la matrice $\mathbf{X}$ est centrée puis réduite. Pour chacune des observations, les variables $p = 1, \ldots, P$ sont successivement centrées en retirant les moyennes $\overline{x}_p$ et centrées en divisant par $\overline{s}_p$. Sans réduction, une variable à grande variance attirera toute l'ACP. Tandis qu'avec réduction, une variable à faible variance se hissera au même niveau qu'une variable à grande variance [Vil17].

$$\overline{\mathbf{X}} = \begin{bmatrix} x_{11} - \overline{x}_1 & x_{12} - \overline{x}_2 & ... & x_{1P} - \overline{x}_P \\ x_{21} - \overline{x}_1 & x_{22} - \overline{x}_2 & ... & x_{2P} - \overline{x}_P \\ ... & ... & ... & ... \\ x_{N1} - \overline{x}_1 & x_{N2} - \overline{x}_2 & ... & x_{NP} - \overline{x}_P \end{bmatrix}$$

$$\overline{\mathbf{X}}_r = \begin{bmatrix} \frac{x_{11} - \overline{x}_1}{s_1} & \frac{x_{12} - \overline{x}_2}{s_2} & ... & \frac{x_{1P} - \overline{x}_P}{s_P} \\ \frac{x_{21} - \overline{x}_1}{s_1} & \frac{x_{22} - \overline{x}_2}{s_2} & ... & \frac{x_{2P} - \overline{x}_P}{s_P} \\ ... & ... & ... & ... \\ \frac{x_{N1} - \overline{x}_1}{s_1} & \frac{x_{N2} - \overline{x}_2}{s_2} & ... & \frac{x_{NP} - \overline{x}_P}{s_P} \end{bmatrix}$$

Deux matrices peuvent ensuite être obtenues. La première est la matrice de covariance $\mathbf{V}$ [Vil17] :

$$\mathbf{V} = \frac{1}{N} \overline{\mathbf{X}} \mathbf{X}$$

Et la seconde est la matrice de corrélation $\mathbf{R}$ [Vil17] :

$$\mathbf{R} = \frac{1}{N}\overline{\mathbf{X}}_r\mathbf{X}_r$$

L'ACP projète ensuite le nuage de points de son axe de base v sur l'axe u ayant une variance maximale. La projection d'un vecteur v sur un axe de vecteur u est un vecteur v' tel que [Vil17] :

$$\|v'\| = \overrightarrow{u} \cdot \overrightarrow{v} = \|u\| \cdot \|v\| \cdot cos(\theta) = x_u \cdot x_v + y_u \cdot y_v$$

Où θ est l'angle entre les deux vecteurs $\overrightarrow{u}$ et $\overrightarrow{v}$ et par conséquent, $\|v'\|$ est proportionnel au cosinus de cet angle. Les vecteurs recherchés pour cette projection sont les vecteurs propres de la matrice $\mathbf{R}$ [Vil17] :

$$\mathbf{R} \cdot u - \lambda\ u = (R - \lambda I) \cdot\ u = 0$$

Les valeurs et vecteurs propres sont ensuite calculés afin de trouver les axes ayant le maximum de variance grâce à la décomposition en valeur singulière [1]. Cette projection fournit deux graphiques à deux dimensions : le graphique des individus et le graphique des variables.

Le graphique des individus représente les observations dans un plan à deux dimensions. Les individus ayant chacun P variables, l'hyper-espace dans lequels ils se trouvaient à été réduit de $\mathbb{R}^P$ à $\mathbb{R}^2$. Le premier et second axe sont respectivement celui avec la plus grande variance et celui avec la seconde plus grande variance. Des groupements peuvent être décelés : des observations très proches dans le plan se ressemblent et possèdent donc des caractéristiques similaires. Au contraire, des variables très éloignées les unes des autres possèdent des caractéristiques très différentes. Enfin, les variables proches de l'origine sont très susceptibles d'être mal projetées et possède une faible contribution à la variance des composantes principales [Vil17].

Ensuite, le graphique des variables représente les données dans une hyper-sphère à N dimensions réduite à un cercle 2 dimensions de corrélation. Les coordonnées des P points sont les corrélations de la variable avec l'axe correspondant. Dès lors, deux variables présentant un angle faible entre-elles sont correlées positivement. Dans le cas contraire, si l'angle entre ces deux variables se rapproche de 180 degrés, les variables sont correlées négativement. Enfin, un angle rectangle entre ces deux variables signifie qu'elle ne sont pas ou très peu corrlées [Vil17].

1. Singular Value Decomposition ou SVD, en anglais