Dessine-moi un mouton

Compte-Rendu de projet long

James Barrios, Antoine Cordelle, Benjamin Mouscadet, Emmanuel Tran

encadrés par Arpad Rimmel et Joanna Tomasik

2017 – 2018

CentraleSupélec

Table des matières

Introduction

Ce compte-rendu présente le Projet Long Supélec *Dessine-moi un mouton*, mené par 8 élèves de Supélec - séparés en deux équipes de 4 - et encadré par deux enseignants-chercheurs, Monsieur Arpad Rimmel et Madame Joanna Tomasik. Plus particulièrement, c'est le travail de l'équipe **Couleuvre** qui est rapporté ici.

L'objectif de ce projet était d'étudier les réseaux de neurones et de les utiliser pour générer des images de synthèses. L'étude des réseaux de neurones comportait une dimension de recherche afin d'appréhender la théorie de ces objets, et une dimension de développement et de programmation dans l'implémentation de ces réseaux et leur utilisation. Le projet s'est donc naturellement décomposé en plusieurs étapes.

Dans un premier temps, il a fallu étudier le fonctionnement des réseaux de neurones, et établir les équations régissant le comportement des réseaux de type perceptron. Puis nous avons implémenté un perceptron afin de résoudre un problème de décision simple au début (**XOR**), puis plus complexe (application à la base de données de chiffres manuscrits MNIST).

Dans un second temps, nous avons commencé à utiliser la structure qui représentait le coeur du projet : les réseaux semi-supervisés en compétition mutuelle, selon la méthode du **Generative Adversarial Network**. Nous l'avons d'abord appliqué à la génération d'images synthétiques de chiffres, puis nous l'avons utilisé pour faire de la synthèse d'images plus complexes, comme des images de moutons.

Afin de mener à bien ce projet long, il a été nécessaire de mettre en place plusieurs outils de gestion de projet. Nous avons créé une organisation **Github** (https://github.com/supelec-GAN) afin de faciliter le partage du code; nous avons utilisé **Trello** pour s'organiser en tâches et optimiser le travail; une bibliographie commune a été mise en place grâce à **Zotero**; la documentation du projet a été faite avec **Doxygen**; enfin, une réunion hebdomadaire réunissant les deux équipes et les enseignants a été imposée, permettant de rendre compte des avancées, de recevoir les directives sur le travail à effectuer, et de manger du chocolat.

Enfin, nous avons choisi le langage C++ pour ce projet. En effet, la structure de réseaux de neurones et de computational graphs (qui sera détaillée plus loin) s'accorde naturellement avec la représentation objet; de plus les réseaux de neurones nécessitent par définition de mener des calculs en grandes quantités. La rapidité de calcul du C++ (héritée du C) ainsi que son approche objet robuste et particulièrement puissante en faisaient le candidat idéal. De plus, un seul membre du groupe n'était pas familier avec le C++ et a dû apprendre en cours de route. **Nous déconseillons vivement aux futurs membres de ce projet long d'utiliser un langage qui ne soit pas maîtrisé par au moins la moitié de l'équipe**.

Chapitre 1

Réseaux de neurones

1.1 Problème de classification

Dans la théorie de l'apprentissage statistique, la classification a pour objectif de déduire d'un nombre fini d'observations indépendantes une partition de l'espace en un ensemble a priori inconnu de domaines de l'espace appelés classes.

1.2 Apprentissage supervisé

1.3 Structure d'un neurone

L'extraordinaire capacité d'apprentissage et d'adaptation des réseaux de neurones biologiques a poussé les scientifiques à tenter de modéliser informatiquement leur fonctionnement afin d'exploiter ces capacités.

Les observations biologiques ont mené au premier modèle du neurone, divisé en 3 parties. Le neurone reçoit des signaux chimio-électriques par ses dendrites; ces signaux sont traités dans le corps cellulaire, où un effet de seuil est appliqué; enfin l'axone permet la transmission du signal chimio-électrique de sortie. De nouvelles découvertes sur les neurones ont permis de complexifier grandement ce modèle, mais c'est celui qui sert de base aux neurones artificiels.

Définition 1 (Neurone)**.** Un neurone à n entrées permet de modéliser une fonction de $\mathbb{R}^n$ dans $\mathbb{R}$. Un neurone est défini par la données de trois paramètres :

- un vecteur de poids $\omega \in \mathbb{R}^n$
- un biais $b \in \mathbb{R}$
- une fonction d'activation $g \in \mathbb{R}^{\mathbb{R}}$

La fonction f modélisée par le neurone s'écrit alors :

$$\forall x \in \mathbb{R}^n, f(x) = g(\omega^T x - b) = g(\sum_{i=0}^{n-1} \omega_i x_i - b) \tag{1.1}$$

Remarque 1 (Biais). Il est possible de rajouter une composante $x_n = -1$ à tous les vecteurs d'entrées afin de pouvoir considérer le biais b comme la composante ω_n du vecteur de poids. L'équation (1.1) devient alors

$$\forall x \in \mathbb{R}^n, f(x) = g(\omega^T x) = g(\sum_{i=0}^{n} \omega_i x_i) \tag{1.2}$$

Cependant, considérer le biais comme un poids pose des problèmes lorsqu'on utilise l'approche des graphes de calculs, qui sera explicitée dans la suite de ce rapport. Nous n'avons donc pas appliqué cette simplification pour nos propres réseaux.

1.4 Réseaux de neurones

Les possibilités limitées d'un neurone seul ont donc poussé l'informatique à agencer les neurones en réseaux afin de créer des modèles plus complexes.

Définition 2 (Réseau de neurone - Perceptrons)**.** Un réseau de neurones est défini par un graphe orienté $\mathcal{G}(V,A)$ où les nœuds sont des neurones, les arêtes des liens entre les neurones et par un ensemble de neurones d'entrée $V_{in} \subset V$ et de neurones de sorties $V_{out} \subset V$. Une arête partant d'un neurone i vers un neurone j signifie que la sortie du neurone i est une entrée pour le neurone j. Notons f_j la fonction représentant le neurone j. Finalement, nous noterons un réseau $\mathcal{N}(V,A,V_{in},V_{out},f)$ où f est l'ensemble des fonctions des neurones.

Définition 3 (Perceptron)**.** Un réseau de neurones est de type perceptron lorsque le graphe qui le représente est acyclique (réseau feedforward) et qu'il peut être représenté comme une suite finie de couches de neurones telle que les sorties d'une couche sont exactement les entrées de la couche suivante.

1.5 Propagation dans un perceptron

On cherche à obtenir les équations qui régissent la propagation des entrées dans un réseau perceptron. Comme les neurones sont agencées par couches, on peut indicer chaque neurone par son numéro de couche et sa position dans la couche. De plus, on parlera dans la suite indistinctement du neurone ou de la fonction qu'il modélise.

Alors, soit un réseau perceptron à N couches. Pour $i \in [\![1,\ N]\!]$ soit n_i le nombre de neurones de la couche i. Pour tout i,j tel que $i \in [\![1,\ N]\!]$,$j \in [\![1,\ n_i]\!]$, le neurone (i,j) est défini par

- sa fonction d'activation $f_{i,j}$
- son vecteur de poids $\omega_{i,j}$
- son biais $b_{i,j}$

Comme on est dans un réseau perceptron, tous les neurones de la couche $i-1$ sont connectés au neurone (i,j). Par conséquent le vecteur d'entrée de ce neurone est un vecteur X_n de dimension $n_{i-1} \times 1$. Alors la sortie du neurone (i,j) est égale à $f_{i,j}(\omega_{i,j}X_{i-1}^T - b_{i,j})$. En écrivant $X_n = \begin{pmatrix} X_0^n & ... & X_{n_i-1}^n \end{pmatrix}^T$, on a alors la relation de récurrence

$$X_i^{j+1} = f_{j,i}(\omega_{j,i}X_j^T - b_{j,i}) \tag{1.3}$$

Dans un réseau perceptron, on suppose de plus que toutes les fonctions d'activations des neurones d'une même couche sont identiques. On a alors $\forall i \in [\![1,\ N]\!]$, $\forall j \in [\![1,\ n_i]\!]$, $f_{i,j} = f_i$ On peut alors réécrire l'équation (1.3) sous forme matricielle. On pose

$$\forall i \in [\![1,\, N]\!],\, F_i = \begin{pmatrix} f_i & ... & f_i \end{pmatrix} \quad \text{(Fonction d'activation de la couche)}$$
$$W_i = \begin{pmatrix} \omega_{i,0} \mid & ... & \mid \omega_{i,n_i-1} \end{pmatrix}^T \quad \text{(Matrice des poids de la couche)}$$
$$B_i = \begin{pmatrix} b_{i,n_i-1} & ... & b_{i,n_i-1} \end{pmatrix}^T$$

En nommant X_0 le vecteur en entrée du perceptron, on a la nouvelle équation matricielle :

$$\forall i \in [\![1,\, N]\!],\, X_i = F_i(W_i X_{i-1} - B_i) \tag{1.4}$$

Chapitre 2

Implémentation des premiers réseaux

Ce chapitre décrit l'implémentation de nos réseaux de neurones sous forme de perceptrons, que nous utiliserons par la suite dans les différentes applications. Nous verrons ici cette implémentation et le premier exemple avec le XOR.

2.1 Implémentation du perceptron

2.1.1 Présentation et diagramme UML

Comme décrit dans la partie théorique sur les perceptrons (1.5), nous avons choisi de représenter nos réseaux de neurones sous forme matricielle. Les neurones ne seront donc pas représentés individuellement, mais par couche de neurones (neuronLayer). Cela permet d'effectuer les calculs sous forme matricielle par rapport à des propagations et rétro-propagations neurone par neurone, ce qui fait gagner beaucoup de temps de calcul. Cela est possible en gardant les mêmes fonctions d'activation pour chaque neurone d'une même couche, ainsi que les mêmes entrées et sorties. Dans le cadre du perceptron, cela convient.

Le diagramme UML comporte une classe principale, Application, qui va gérer les expériences et réseaux de neurones. Cette classe contient un réseau de neurones (NeuralNetwork) composé de plusieurs couches de neurones (NeuronLayer). Application possède aussi un collecteur qui est utilisé pour récupérer les données des expériences, les traiter et les exporter dans un fichier .csv. Enfin, la classe Teacher a pour rôle de gérer la rétropropagation et l'apprentissage du réseau de neurones.

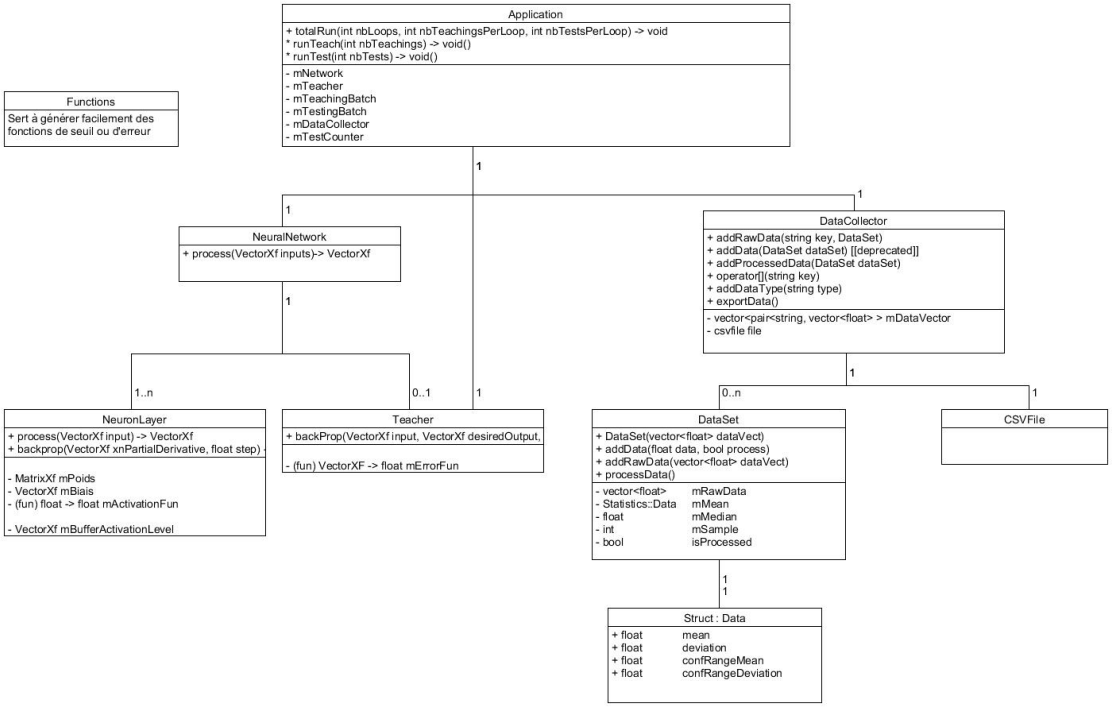

FIG. 2.1 – *Diagramme UML*

2.1.2 NeuronLayer

Une instance de la classe **NeuronLayer** représente une couche de neurones. Comme nous utilisons la représentation matricielle des réseaux perceptrons, une couche de neurones n'est constituée que d'une matrice de poids, d'un vecteur de biais et d'une fonction d'activation. Si *input* et *output* représentent respectivement la dimension du vecteur d'entrée et de sortie de la couche, alors on a les dimensions suivantes (on travaille en ligne) :

- La matrice de poids est de dimension $input \times output$
- Le vecteur de biais est un vecteur ligne de dimension *output*

2.1.3 NeuralNetwork

Une instance de la classe **NeuralNetwork** représente un réseau perceptron, c'est à dire plusieurs couches de neurones (NeuronLayer) mises les unes à la suite des autres. Chaque couche de neurones reçoit en entrée la sortie de la couche précédente et transmet sa sortie à la couche suivante. Pour représenter cet objet, nous avons utilisé la structure de liste simplement chaînée. La class **NeuralNetwork** hérite donc de la classe **std::list<NeuronLayer>**. Cette structure légère et robuste permet d'itérer facilement sur toutes les couches du réseau.

2.1.4 DataCollector

Afin de collecter les données du réseau et de pouvoir effectuer un traitement statistique dessus (calcul de moyenne, ecart-type, intervalle de confiance...) nous avons écrit la classe

DataCollector. Elle se charge de récupérer les sorties du réseau, d'effectuer des calculs statistiques standards et de les exporter au format **.csv** afin de permettre une visualisation facile de toutes les données.

2.2 Etude du XOR

Le problème du XOR est un problème de classification standard, auquel toutes les méthodes de classification sont confrontées. Nous avons donc naturellement commencé par le traitement de ce problème pour nous exercer à la manipulation des perceptrons.

2.2.1 Definition du problème

Le problème du XOR est un problème en deux dimensions : les entrées sont des paires (x_1, x_2) et la sortie est un booléen. Deux approches différentes existent, selon le choix du domaine de définition :

- 1^{er} choix : on veut obtenir un XOR purement booléen, dont les 4 entrées possibles sont les paires (0,0) (0,1), (1,0), (1,1). Les résultats attendus sont respectivements 0, 1, 1, 0.
- 2^{nd} choix : on veut obtenir un XOR défini sur le carré unité du plan : $(x,y) \in [-1,1]^2$. Dans ce cas, on attend du XOR qu'il renvoie 1 lorsque $x \times y < 0$ et 0 sinon. On fait donc un XOR sur le signe des coordonnées.

Selon l'approche, le réseau requis sera différent. Dans le premier cas, il est possible de classifier les 4 points selon leur appartenance à un ensemble convexe (qui prend la forme d'une bande). Par conséquent, le problème peut se résoudre avec deux couches de neurones.

Dans le second cas, il faut séparer le plan en 4 quadrants, et regrouper les quadrants par deux. On obtient des classes non convexes, et le problème requiert 3 couches de neurones pour être résolu.

Nous avons choisi de travailler sur la deuxième situation, le but est donc d'obtenir un réseau qui découpe le carré unité en quatre quadrants bien distincts, qui correspondent aux quatre quadrants délimités par les axes des abscisses et des ordonnées.

2.2.2 Résolution

Il s'agit d'un problème à deux dimensions, renvoyant une seule valeur, et nécessitant 3 couches cachées. Nous optons donc pour un réseau 2-2-1.

Qualitativement, on peut dire que

- les deux neurones de la première couche vont se charger de tracer les deux droites correspondant aux abscisses et aux ordonnées
- les deux neurones de la seconde couche vont se charger de réaliser les fonctions öuët ëtsur les quadrants ainsi tracés, afin de d'obtenir deux quadrants correspondants au résultat 1 et deux quadrants correspondants aux résultat 0.
- la troisième couche se charge de réaliser la fonction öusur les résultats renvoyés par la couche 2, afin que le résultat final soit 1 si le point se trouve dans l'un des deux quadrants correspondant au résultat 1.

De plus, nous avons choisi d'utiliser des fonctions d'activations sigmoides en $\frac{1}{1+exp(-\lambda x}$. Une telle fonction d'activation est nécessaire pour pouvoir permettre un apprentissage par rétro-propagation (continue et dériavable), cependant elle empêche d'avoir un résultat booléen à la fin, puisque la valeur de sortie varie continument entre 0 et 1. Le résultat est donc une densité de probabilité.

Enfin, nous utilisons un pas d'apprentissage de $\eta <= 0.01$. Bien que la littérature suggère souvent le pas empirique $\eta = 0.2$, la valeur de 0.01 est la valeur maximale que nous ayons trouvée qui permette au XOR de converger. Au delà, le réseau se bloque presque systématiquement dans un état non désiré.

Avec une dizaine de milliers d'apprentissages, on obtient le résultat suivant :

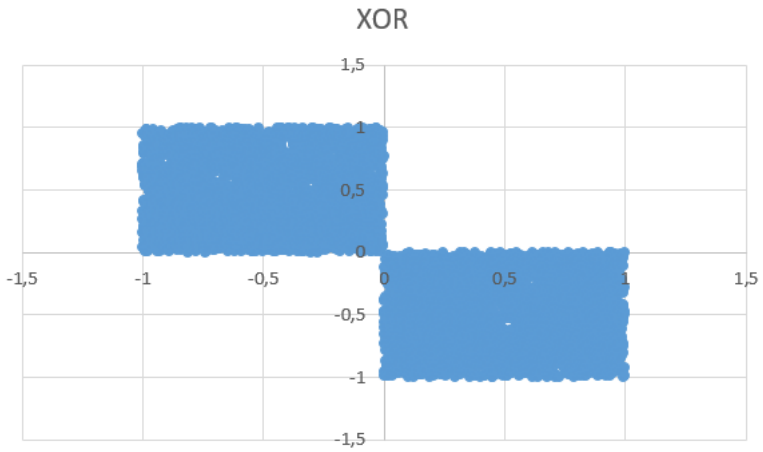

FIG. 2.2 – *Résultat d'un XOR sur* 1000 *tests après* 10000 *apprentissages pour un réseau 2-2-1*

Chapitre 3

Reconnaissance des chiffres manuscrits

3.1 La base de données MNIST

3.1.1 Présentation de MNIST

La base de données MNIST (ou Mixed National Institute of Standards and Technology) est un regroupement de 70 000 chiffres manuscrits. 60 000 d'entre eux servent à l'apprentissage et les 10 000 autres permettent de tester le réseau de neurones après l'apprentissage. Ce sont des images normalisées en noir et blanc, de 28 pixels de côté chacun codé sur un octet. Nous nous appuierons sur cette base de données pour faire apprendre à notre réseau de neurones la reconnaissance de chiffres.

3.1.2 Extraction de MNIST

Les données sont dans le format idx1 et idx3 tel que :

offset	**type**	**valeur**	**description**
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of items
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
...	...	...	...
xxxx	unsigned byte	??	pixel

On peut distinguer :

- Le *magic number* qui permet d'identifier le format de la base de données
- Les *number of items, number of rows, number of colums* donne des informations sur les données, ce qui nous permettra d'extraire les images
- Les *pixels* qui sont les octets en niveau de gris des images

L'extraction se fait donc en lisant successivement les octets et en les stockant dans des *vector<Eigen:Matrix>* grâce aux données récupérées. En faisant de même avec les labels dont le format est très sensiblement le même, on obtient une liste de vecteur colonne

d'*Eigen::Matrix* de taille 784 (28*28) ainsi que leur label sur un autre *Eigen::Matrix*. On fait de même avec les échantillons de test et on est fin prêt pour l'apprentissage.

3.2 Apprentissage de la base de données

3.2.1 Paramétrage du réseau de neurones

Bien que le format de l'entrée soit le même (vecteur colonne si ce n'est la taille qui change), un réseau à 3 ou 4 neurones ne suffit pas. Il a fallu adapter de façon empirique les différents paramètres. On peut jouer sur :

- Le nombre de neurones
- Le nombre de couches
- Le pas d'apprentissage
- Les fonctions d'activations par couches
- Le nombre d'apprentissages

Le nombre de couches et de neurones

Le pas d'apprentissage

Les fonctions d'activations par couches

Le nombre d'apprentissages

3.2.2 Résultat

Chapitre 4

Implémentation en C++ et problèmes rencontrés

Ce chapitre a pour but d'aider les prochains qui souhaiteraient coder en C++ pour qu'ils ne se heurtent pas aux même ecueils que nous, et qu'ils puissent passer plus de temps sur la conception du projet plutôt que sur le debug et la recherche d'outils efficaces.

4.1 Outils utilisés

4.1.1 IDE

Il est important que chaque membre développe dans un IDE qui lui soit familier, et qui offre un minimum d'ergonomie et d'efficacité. Dans le projet, les IDE utilisés ont été :

- XCode, pour développer sous MAC.
- QtCreator, disponible sous Linux et Windows (bien qu'il soit un peu plus pénible à installer sous windows).
- Vim, déconseillé à ceux qui ne maîtrisent pas l'outil, mais qui peut se révéler très puissant avec les bons plugins.

Chaque IDE venant avec ses propres fichiers de configuration, nous conseillons de nommer ceux-ci du nom de leur utilisateur (typiquement john_doe.pro.user) ou de les ajouter au .gitignore afin d'éviter les conflits.

4.1.2 Eigen

Les réseaux de neurones de type perceptron reposent largement sur les produits matriciels. Afin d'effectuer ces calculs le plus efficacement possible, notre code utilise le framework C++ Eigen. Ce framework permet de manipuler l'algèbre linéaire et les matrices de manière la plus optimale possible. De plus, Eigen utilise openmp ce qui permet de paralléliser les calculs sur l'ensemble des coeurs.

De ce que nous avons pu voir, il n'existe aucune alternative à Eigen qui offre autant de possibilités et d'optimisation. Cependant Eigen n'est pas facile à manipuler au premier abord. Sa construction entièrement à base de template et organisée de façon à pouvoir effectuer le plus de calculs possible en compile-time rend les types des objets et les messages d'erreur difficilement déchiffrables. Debugguer un problème lié à Eigen nécessite un bon débuggueur et de la patience. De plus, la méthode de parallélisation d'Eigen reste assez obscure. Le framework semble paralléliser les calculs en colonnes, c'est à dire qu'il calcule

simultanément les colonnes de la matrices résultante d'un calcul. Autrement dit, si le résultat est une ligne, le calcul sera parallélisé, mais si c'est une colonne il ne tournera que dans un seul thread. Ceci implique de n'utiliser que des matrices lignes dans le code, bien que tous les calculs menés théoriquement soient faits avec des colonnes. Attention donc à transposer correctement.