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

Un neurone permet, en prenant n entrées, de renvoyer en sortie une valeur réelle entre 0 (ou -1) et 1. Au vu des fonctions d'activation utilisées, les réponses renvoyées peuvent être assimilées à des booléens (0 : 1). La fonction de transfert du réseau de neurone étant une combinaison linéaire à qui on applique le créneau (fonction d'activation), un neurone seul peut séparer un plan en deux. Par exemple, en prenant un réseau à deux entrées, une sortie, dans un problème de classification classique, ce réseau permet de séparer le plan (x,y) en deux parties séparées par une droite, comme montré ci-dessous :

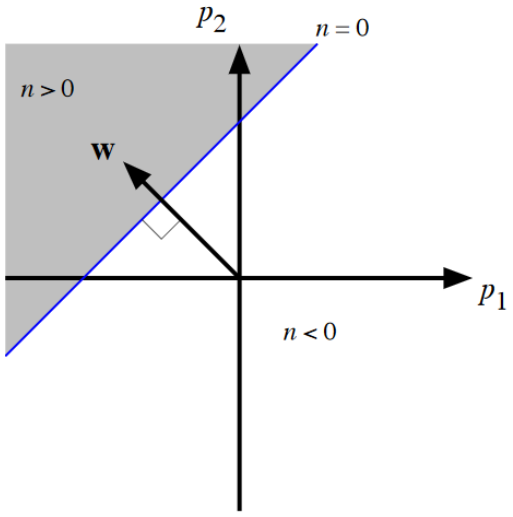

FIG. 1.1 – *Classification pour un neurone seul à deux entrées, une sortie*

Un neurone seul ne peut donc que séparer un plan en deux parties. 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. Ainsi, un réseau à deux couches peut (à condition d'avoir suffisamment de neurones dans ces couches) identifier tout zone convexe d'un espace vectoriel, alors qu'un réseau à 3 couches peut séparer toute partie de cet espace. On agence alors 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

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

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.

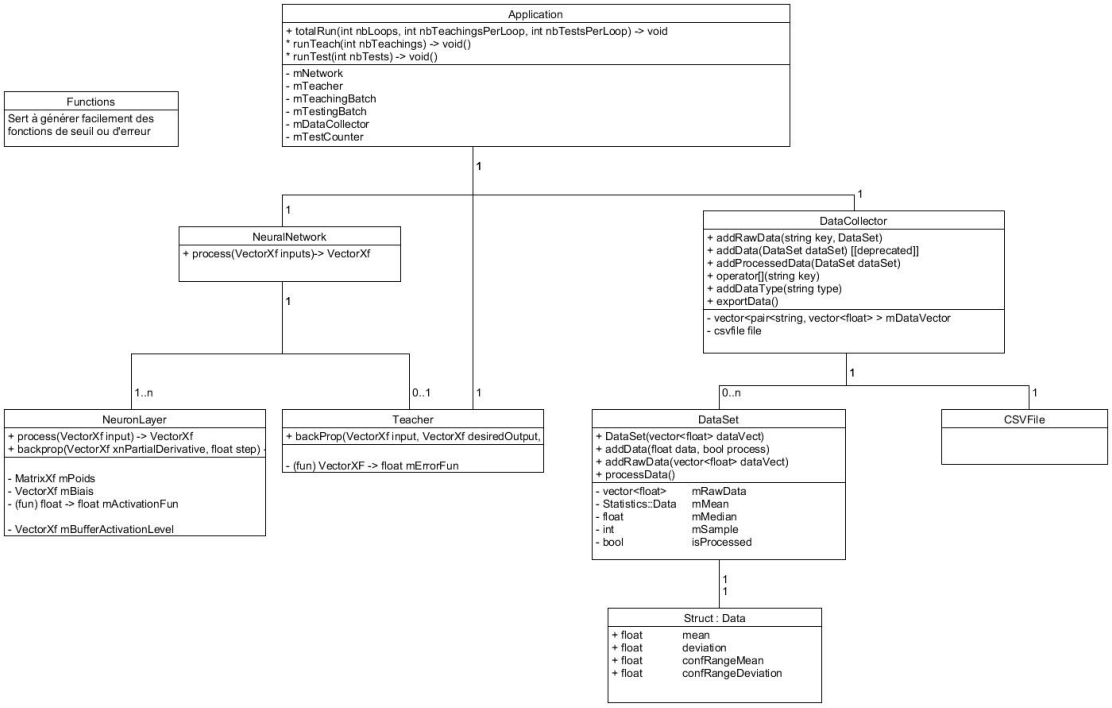

FIG. 2.1 – *Diagramme UML*

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 :

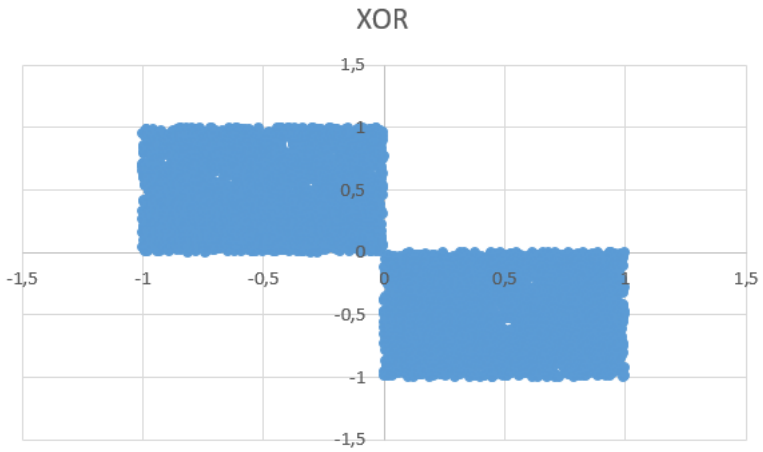

FIG. 2.2 – *Résultat d'un XOR sur* 1000 *tests après* 10000 *apprentissages pour un réseau 2-2-1*

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.

Chapitre 5

Generative Adversarial Networks - Partie théorique

Après avoir appris à manipuler correctement les réseaux perceptrons simples, nous nous sommes intéressés à la structure de **Generative Adversarial Networks**, qui a constitué le coeur du projet, et son enjeu majeur.

La naissance du GAN se place dans un contexte de recherche de moyens innovants et efficaces de génération de données arbitraires. Dans l'ère de l'information, les données constituent une ressource fondamentale, et la capacité d'en générer facilement de nouvelles représente un pan entier de la recherche, si ce n'est plusieurs. Le GAN sont une solution à base de réseaux à apprentissage semi-supervisés, dont l'objectif est de générer des données arbitraires en extrapolant à partir de données initialement fournies.

Dans ce chapitre, nous décrirons qualitativement le fonctionnement attendu des GANs et leur théorie, puis nous détaillerons les aspects plus techniques et mathématiques du problème.

5.1 Description du fonctionnement des *Generative Adversarial Networks*

Le GAN est une structure qui fait intervenir deux réseaux placés en compétition, qui vont s'émuler mutuellement afin d'atteindre un optimum, ceci en partant de rien. Les GAN représentent un domaine de recherche qui s'est ouvert en 2014 avec la publication d'un article de Ian Goodfellow.

En 2014,l'équipe de Ian Goodfellow publie dans NIPS un article intitulé **Generative Adversarial Nets**, dans lequel il décrit une nouvelle structure censée répondre au problème de génération de donnée. Cette structure est fondée sur l'entraînement simultané de deux réseaux neuronaux en compétition, respectivement nommés **Générateur** et **Discriminateur**.

5.1.1 GAN - Idée générale

Pour commencer, on souhaite obtenir un réseau qui soit capable de générer sur commande des données

- Pertinentes par rapport aux contraintes qu'on lui impose (exemple : on souhaite pouvoir obtenir un réseau qui dessine spécifiquement des moutons, et non un réseau

qui dessine n'importe quoi comme bon lui semble)
- Variées, c'est à dire que les données générées sont différentes deux a deux
- Réalistes, voire indiscernables de données réelles aux yeux d'un humain

En résumé on souhaite construire un réseau (nommé dans toute la suite **Générateur**) qui modélise une projection d'un ensemble d'entrée vers l'ensemble des données réalistes et pertinentes vis-à-vis des contraintes imposées, et dont l'image ne soit pas réduite à un point de cet ensemble.

Résumée de cette façon, la tâche semble extraordinairement ardue, c'est pourquoi la structure fait appel à un second réseau (nommé **Discriminateur** dans toute la suite). En effet, la difficulté principale liée aux réseaux de neurones est l'apprentissage. Dans ce cas en particulier, la difficulté majeure est de caractériser ce qu'est une donnée de synthèse ¨pertinente et réaliste, voire indiscernable d'une donnée réelle¨ car s'il est très facile de distinguer à l'oeil une image de mouton de n'importe quoi d'autre, il est extrêmement difficile de construire un critère mathématique permettant de distinguer les images de mouton des autres. Or c'est ce critère dont nous avons besoin pour faire apprendre correctement à notre **Générateur**.

Cependant, nous avons vus avec les perceptrons et la base de données MNIST qu'il était parfaitement possible d'obtenir un réseau de neurones modélisant avec une grande précision des critères visuels, que l'on serait bien incapables d'exprimer mathématiquement. Il est par exemple impossible d'écrire analytiquement ce qu'est un chiffre manuscrit, mais il est possible de générer un réseau de neurones qui sache discriminer les chiffres manuscrits des autres images. Ainsi, en déléguant à un second réseau la responsabilité de modéliser le critère selon lequel nous voulons que nos données soient générées (exemple: on ne veut générer que des images de mouton, on va faire apprendre au **Discriminateur** à différencier un mouton de toutes les autres images possibles), on résout le (premier) problème majeur qui se pose.

5.1.2 Des réseaux en compétition

Le GAN va donc faire intervenir deux réseaux, dont l'un sert seulement à rendre possible l'apprentissage de l'autre. Le **Discriminateur** a pour unique rôle de modéliser le critère que l'on souhaite utiliser; son apprentissage se fait donc de la même manière que tout ce qui a été fait jusqu'ici, en apprentissage supervisé. C'est un réseau dont l'entrée est de la dimension des données que l'on souhaite construire, et dont la sortie est un réel entre 0 et 1, caractérisant la certitude du réseau que l'entrée qui lui été donnée est une donnée réelle ou non. Plus précisément, pour une entrée donnée, plus la sortie est proche de 1, plus le **Discriminateur** est convaincu que l'entrée est une donnée réelle, et non de synthèse. Son objectif est donc de sortir 1 pour toute entrée réelle, et 0 pour toute donnée de sythèse.

Le **Générateur** de son côté est le réseau qui doit générer des données synthétiques suffisamment convaincantes pour que le **Discriminateur** ne puisse les différencier des données réelles. Sa sortie est donc naturellement de la dimension des données qu'on souhaite construire. L'entrée du **Générateur** est un bruit (blanc) à partir duquel le réseau doit construire une donnée convaincante. Le **Générateur** est donc un réseau qui réalise une projection de l'ensemble d'entrée vers l'ensemble des données ¨suffisamment convaincantes pour tromper un humain¨ Passer un bruit en entrée permet d'obtenir des sorties variées. On comprend cependant que, le réseau réalisant un calcul déterministe, la dimension de la

sorties est nécessairement inférieure à la dimension de l'entrée. Pour être certain d'obtenir un ensemble de sortie aussi grand que possible, on s'assurera toujours d'avoir un ensemble d'entrée de dimension suffisamment grande.

Exemple 1. Cas des images de mouton

On souhaite obtenir un réseau qui génère des images 28×28 de moutons, suffisamment réalistes pour ne pouvoir être distinguées de réelles photos de moutons. On souhaite également observer de la variation dans cesimages (avoir plein de moutons différents).

L'ensemble de sortie du **Générateur** est donc un ensemble inclu dans l'ensemble des images 28×28 et qui contient toutes les images que le **Générateur** considère comme étant des moutons réalistes. Pour être certain que l'ensemble d'entrée soit de dimension au moins aussi importante que l'ensemble de sortie, on mettra en entrée du réseau un bruit blanc de dimension 28×28. On est certain de cette façon que la taille de l'ensemble d'entrée n'impose pas de restriction sur les sorties.

5.2 Discriminateur et générateur

5.2.1 Théorie des jeux et équilibre de Nash

$D(G(z)) = D(x) = \frac{1}{2}$: un premier critère d'optimalité

5.2.2 Les différentes distributions statistiques impliquées

5.3 es fonctions de coût, l'apprentissage

5.3.1 Les différentes fonctions de coût

5.3.2 Apprentissage par récompense ou par punition ?

5.3.3 Quand un réseau prend le pas sur l'autre

Chapitre 6

Generative Adversarial Networks - Implémentation

La structure théorique des **Generative Adversarial Networks** étant maintenant définie, nous devons implémenter ces réseaux. Pour cela, nous nous sommes lancés dans la continuité de MNIST, avec la génération de chiffres manuscrits par le **Générateur**, qui doit alors convaincre le **Discriminateur**. Ce cas de figure est idéal pour une première implémentation des GAN, puisque nous restons avec des données relativement simple (Dimension 28*28, en niveau de gris), avec un format normalisé et une importante base de données à disposition (La base de données MNIST : 60000 images d'apprentissages, 10000 données de test).

Le **Discriminateur** s'entraine donc à reconnaître les chiffres provenant de la base de données de MNIST en les différenciant des chiffres factices créés par le **Générateur**. Dans un premier temps, nous travaillons sur des chiffres uniques, en apprenant au Discriminateur à reconnaitre uniquement de 3 par exemple.

Nous verrons dans ce chapitre tout d'abord l'implémentation de notre GAN, puis les premiers résultats obtenus. Ces résultats nous ont confronté à divers problèmes, que nous aborderons alors avec les différentes solutions envisagées.

6.1 Implémentation des *Generative Adversarial Networks*

6.1.1 Présentation de la structure

6.1.2 Ratio d'apprentissage

6.2 Premiers résultats

6.3 Difficultés rencontrées et solutions émises

6.3.1 Mode Collapse du *Generateur* vers un unique point

Présentation du problème

Un problème récurrent que l'on peut observer dans les GAN est le *mode collapse*, ou *missing modes*. Prenons un GAN, le générateur doit tenter de créer des images les plus réalistes possible par rapport à une distribution de données réelles, par l'intermédiaire du discriminateur. Cette distribution peut présenter plusieurs modes, plusieurs zones en

quelque sorte (En reprenant l'exemple des chiffres de MNIST, on peut vouloir générer des 3, ou des 4). Cependant, on observe que le générateur a souvent tendance à apprendre à ne générer qu'un seul mode. On appelle cela le mode collapse : le générateur apprendra à aller vers une des zones de la distribution voulue, se rendra compte que cette zone marche, et y restera. Sur l'exemple simplifié ci-dessous, la distribution d'origine présente 4 zones, et le GAN n'en couvre qu'une seule.

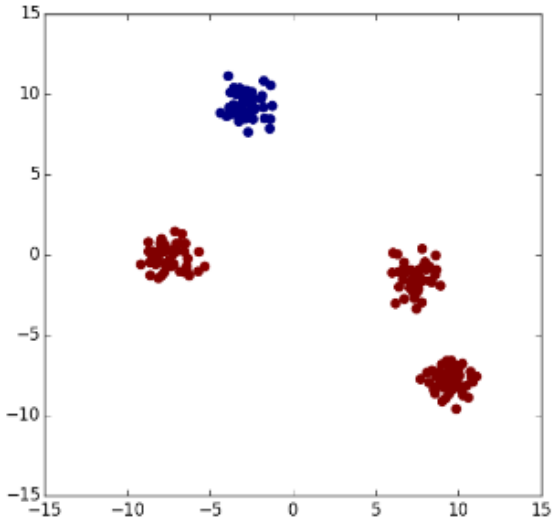

FIG. 6.1 – *Missing modes : les points répresentent les éléments de la distribution de donnéees réelles. En bleu l'ensemble de sortie du GAN appartenant à la distribution réelle, et en rouge les zones de la distribution initiale non couverte par le GAN.*

Ce phénomène récurrent est observable à différents degrés : on peut avoir des collapses partiels, les données générées seront donc variables mais toute la distribution ne sera pas représentée par le générateur, ou des collapses totaux, où le générateur ne sera qu'une projection vers un point. Les données générées seront alors très proches les unes des autres. Nous avons été confrontés à ce problème de collapse total en générant des chiffres à partir de la base de données MNIST. Comme montré ci-dessous, en essayant de générer des 3, nous obtenons pour un réseau donné, des images générées qui sont toutes très proches, avec la même forme de 3, les mêmes pixels singuliers.

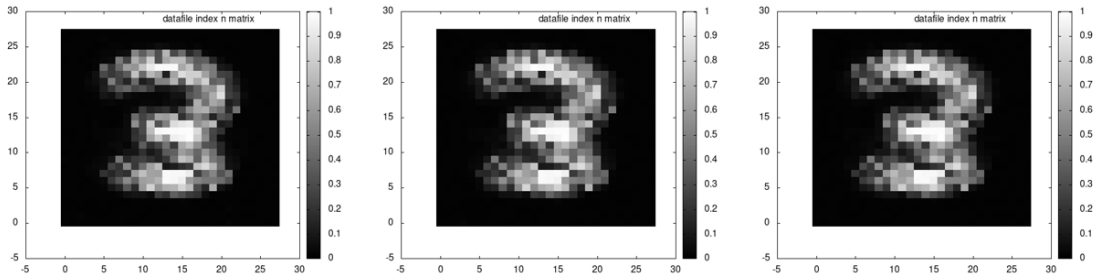

FIG. 6.2 – *Missing modes : On obtient pour un même réseau, 3 échantillons générés qui sont sensiblement les mêmes.*

Solution envisagée : Retrait du biais dans les réseaux (non fructueuse)

Une première hypothèse concernant ce problème de Mode Collapse était la suivante : l'unique image produite par le générateur n'était en réalité qu'une image du masque de biais qui figeait l'image, par exemple en plaçant certains biais trop haut ou trop bas, annulant les entrées bruitées en début de réseau. On aurait alors une image déterminée uniquement par les biais, ce qui expliquerait un collapse vers une unique image. De plus, en observant précisément le masque de biais sous forme matricielle, on remarque que ses poids forment effectivement l'image d'un nombre, comme on peut le voir ci-dessous :

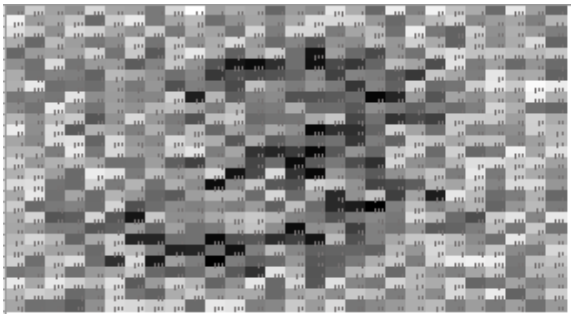

FIG. 6.3 – *Masque du biais : on peut observer que le biais forme lui même les bases d'une image, un 3 ici.*

On peut vérifier cette supposition en supprimant les biais de nos réseaux de neurones. On aurait alors plus de valeurs constantes qui figeraient l'image. Cependant, en implémentant ces réseaux, on n'observe qu'un ralentissement de l'apprentissage vers un état de Collapse similaire : les résultats sont moins intéressants (chiffres moins beaux), l'apprentissage est plus lent, et on converge toujours vers un Mode Collapse.

La suppression du biais ne permet donc pas de résoudre ce problème, c'est bien la fonction de transfert globale du réseau de neurones qui converge vers cet unique point.

Solution envisagée : Injection de bruit dans chaque couche du Générateur (fructueuse)