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, Couleuvre et Salamandre - et encadré par deux enseignants-chercheurs du Laboratoire de Recherche Informatique, 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**.

Nous avons de plus pu présenter les résultats de nos expériences sur le GAN au CS^2, le

Colloque Scientifique de CentraleSupélec, le 9 avril 2018. Nous avons rencontré et discuté avec Yann LeCun des avancées sur le GAN à l'occasion d'une de ses conférences à la Bibliothèque Nationale de France, et rencontré un membre du Laboratoire de Signaux et Systèmes de CentraleSupélec, Giuseppe Valenzise, pour discuter de l'évaluation de la qualité d'image.

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

L'apprentissage supervisé est une branche de l'apprentissage automatique qui consiste à reproduire automatiquement des règles à partir d'une base de données d'apprentissage labellisée. A partir de ces exemples, on apprend donc à reconnaitre chaque élément et à le labelliser correctement. L'objectif final est alors de pouvoir généraliser cet apprentissage. Ainsi, si l'on prend l'exemple (que l'on appliquera plus tard avec MNIST) d'une base de données de chiffres manuscrits, on apprend à reconnaitre les chiffres parmi cette base, en espérant pouvoir obtenir une généralisation qui nous permettra de reconnaitre par la suite n'importe quel chiffre manuscrit.

On peut distinguer l'apprentissage supervisé de l'apprentissage non-supervisé, qui lui consiste à créer des règles à partir d'une base de données fournie, qui n'aura alors pas de labels pré-définis. Le système devra alors trouver des patterns, des caractéristiques communes entre les éléments pour les regrouper en de nouveaux labels.

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 :

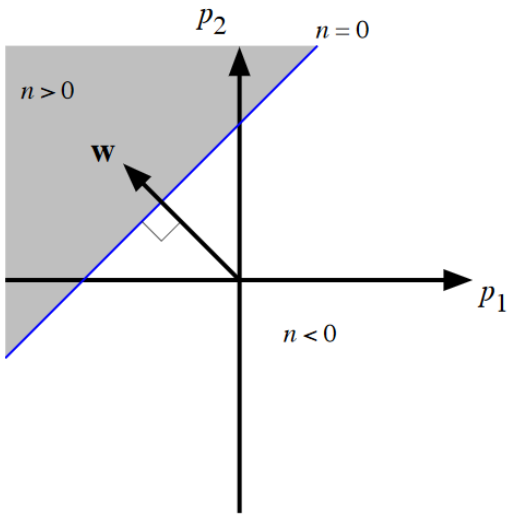


FIG. 1.1 – *Classification pour un neurone seul à deux entrées, une sortie*

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 :

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

1.6 Apprentissage : Rétropropagation

Remarque 2. On a dans toutes nos implémentations pris un pas de rétropropagation de $dx = 0{,}01$.

La propagation des entrées dans un perceptron permet d'obtenir la sortie de ce réseau. Or, tout l'objectif des réseaux neuronaux est de les améliorer afin d'obtenir les sorties voulues. Ainsi, l'apprentissage du perceptron est nécessaire. Celui-si se fait à travers la rétro-propagation des erreurs. Il s'agit en fait d'effectuer une descente de gradient, qui consiste à déterminer pour chaque paramètre du réseau (coefficient de poids ou de biais) le gradient par rapport à la sortie et de modifier ce paramètre en conséquence pour minimiser la sortie. La rétropropagation consiste donc à calculer l'influence de chaque paramètre sur la sortie et à les mettre à jour en fonction de cette influence

La formule de mise à jour du réseau à chaque apprentissage est la suivante :

$$W(t+1) = W(t) + \eta \frac{\partial E}{\partial W}$$

avec η le pas de convergence et $\frac{\partial E}{\partial W}$ la matrice de descente du gradient .

Remarque 3. Dans l'article de LeCun 98 [1], les équations matricielles de la rétropropagation données par l'article sont fausses. En effet, un abus de notation fait penser à un produit matriciel. Il s'agit en fait des matrices de produits (coefficients à coefficients). Les équations présentées ci-dessous sont corrigées.

A la couche k l'influence des poids est donnée par :

$$\frac{\partial E^p}{\partial W_k} = \frac{\partial F}{\partial W}(W_k, X_{k-1}) \frac{\partial E^p}{\partial X_k}$$

Avec $\frac{\partial F}{\partial W}(W_k, X_{k-1})$ la matrice jacobienne de F par rapport à la variable W_k.

Pour pouvoir calculer l'influence des poids de toutes les couches, il faut donc calculer $\frac{\partial E^p}{\partial W_k}$

On peut calculer par récurrence cette valeur pour toutes les couches.

$$\frac{\partial E^p}{\partial X_{k-1}} = \frac{\partial F}{\partial X}(W_k, X_{k-1}) \frac{\partial E^p}{\partial X_k}$$

Avec $\frac{\partial F}{\partial X}(W_k, X_{k-1})$ la matrice jacobienne de F par rapport à la variable X_k. De plus, dans un perceptron on peut noter la sortie de la couche k :

$$Y_k = W_k X_k$$
$$X_k = F(Y_k)$$

On obtient donc ces 3 équations :

$$\frac{\partial E^p}{\partial y_k^i} = f'(x_k^i) \frac{\partial E^p}{\partial x_k^i}$$
$$\frac{\partial E^p}{\partial w_k^{i,j}} = x_{k-1}^j \frac{\partial E^p}{\partial y_k^i}$$
$$\frac{\partial E^p}{\partial x_{k-1}^m} = \sum_i (w_k^{im} \frac{\partial E^p}{\partial X_k})$$

En forme matricielle, ces équations donnent :

$$\frac{\partial E^p}{\partial Y_k} = Diag(f'(x_k^i))\frac{\partial E^p}{\partial x_k^i}$$
$$\frac{\partial E^p}{\partial W_k} = X_{k-1}^T \frac{\partial E^p}{\partial Y_k}$$
$$\frac{\partial E^p}{\partial X_{k-1}} = W_k^T \frac{\partial E^p}{\partial X_k}$$

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 rétro-propagation et l'apprentissage du réseau de neurones.

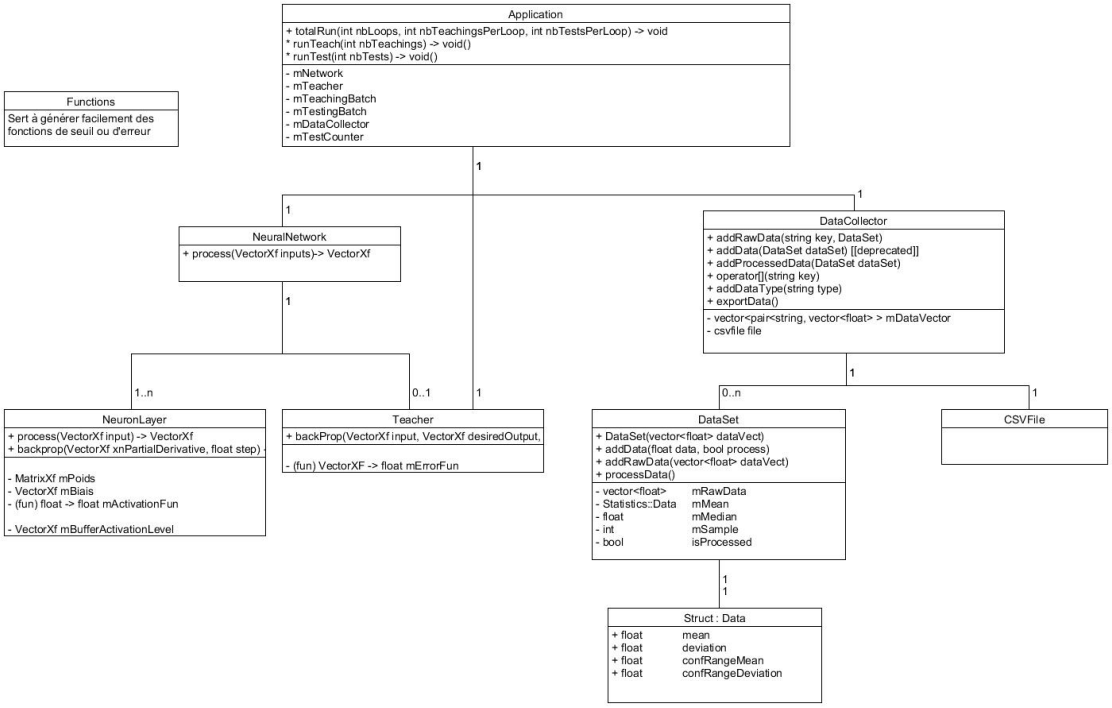


FIG. 2.1 – *Diagramme UML*

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, écart-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 Étude 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 Définition 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 respectivement 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.

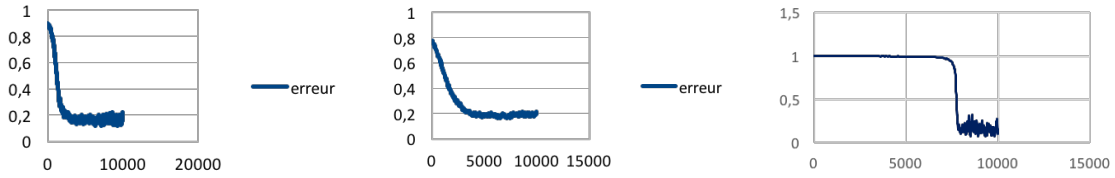
2.2.2 Résolution

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

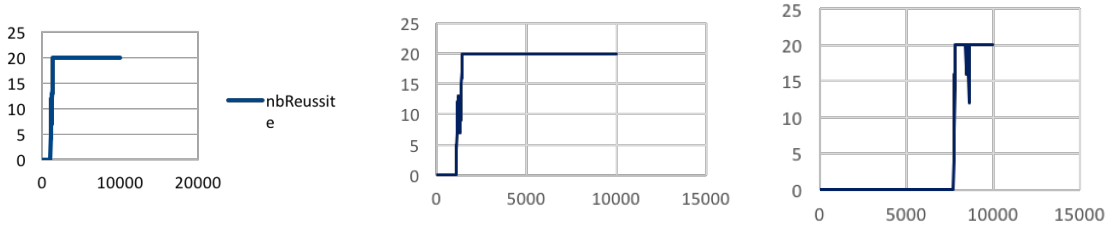
- les deux neurones de la première couche vont se charger de tracer les deux droites correspondant aux abscisses et aux ordonnées
- les deux neurones de la seconde couche vont se charger de réaliser les fonctions öuët ëtsur les quadrants ainsi tracés, afin de d'obtenir deux quadrants correspondants au 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 sigmoïdes en $\frac{1}{1+exp(-\lambda x)}$. Une telle fonction d'activation est nécessaire pour pouvoir permettre un apprentissage par rétro-propagation (continue et dérivable), cependant elle empêche d'avoir un résultat booléen à la fin, puisque la valeur de sortie varie continûment entre 0 et 1. Le résultat est donc une densité de probabilité. Le paramètre de fonction d'activation a été fixé à $\lambda = 0.4$ après différents tests affichés aux figures 2.2 et suivante.



$\lambda = 2$ $\lambda = 4$ $\lambda = 8$

FIG. 2.2 – *taux d'erreur selon la norme 2*



$\lambda = 2$ $\lambda = 4$ $\lambda = 8$

FIG. 2.3 – *nombre de réussites (chiffre identifié correctement) pour 20 tentatives*

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 :

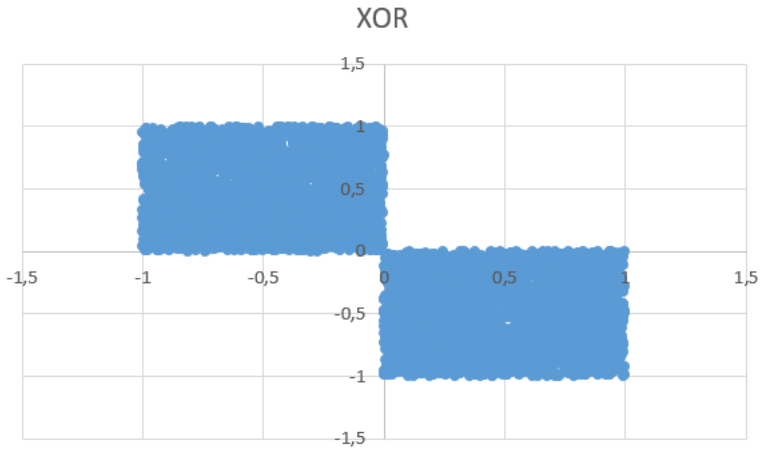


FIG. 2.4 – *Résultat d'un XOR sur* 1000 *tests après* 10000 *apprentissages pour un réseau 2-2-1*

L'étude du XOR semble être un problème qui peut poser des problèmes de convergence, c'est ainsi qu'une partie de nos simulations ont convergé vers un système qui n'a pas bien appris, restant bloqué dans un minimum local.

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.

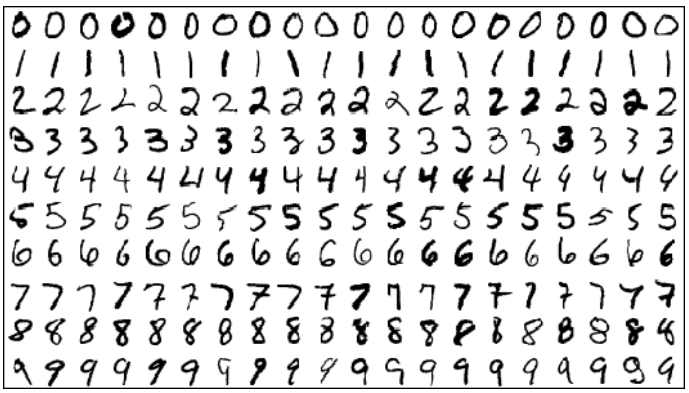


FIG. 3.1 – *Exemples d'images tirées de la base de données MNIST. Chaque chiffre fait 28 pixels de côté*

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 que pour le XOR (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 par couche
- 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

Dans le cas d'un perceptron, nous avons réussi à obtenir un réseau classificateur de chiffre à 90% de succès avec uniquement une couche composée de 10 neurones. Néanmoins, afin d'obtenir un score de réussite supérieur, il a été nécessaire de d'augmenter la nombre de couche à 3, selon une configuration $784 \times 300 \times 100 \times 10$ où 784 représente le nombre d'entrée.

Le pas d'apprentissage

Le pas d'apprentissage par défaut utilisé avec succès sur le XOR était de $\eta = 0{,}01$. Or, pour MNIST, un pas de $\eta = 0{,}01$ ne permet pas au réseau de sortir d'un minimum local. Un pas fonctionnel était de $\eta = 0{,}2$

Les fonctions d'activations par couches

Nous avons, comme dans le cas du XOR (cf. 2.2.2), continué à utiliser un fonction d'activations sigmoïde pour toutes les couches : $\frac{1}{1+exp(-\lambda x)}$, avec cette fois-ci $\lambda = 0{,}1$

Le nombre d'apprentissages

Le nombre d'apprentissage nécessaire pour obtenir un taux d'erreur inférieur à 10% est relativement important, notamment par rapport au nombre d'apprentissages nécessaire au GAN, aux alentours de 250 000 apprentissages. Néanmoins, passé ce nombre d'apprentissage, la courbe d'apprentissage varie peu. Il est nécessaire d'effectuer environ un million d'apprentissages pour avoir un taux d'erreur de 5%. Ces résultats ont été effectués avec des descentes de gradients classiques sur le réseau à 3 couches cité plus haut.

3.2.2 Résultat

On peut voir le résultat sur la figure 3.2

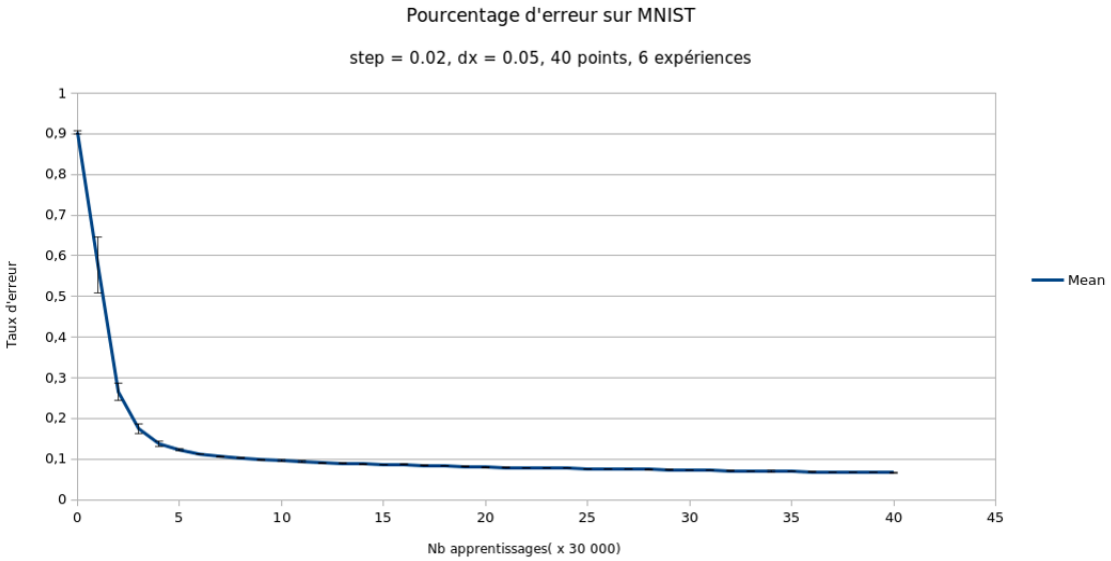


FIG. 3.2 – *Courbe du taux d'erreur de MNIST*

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 écueils 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 (et utilisation de Clang au lieu de LLVM pour les parallélisations OpenMP [2]).
- 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 Conseils à de futurs groupes

Language de programmation Outre le fait que 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, nous avons proposé au corps encadrant, après un passage obligatoire par un langage de programmation afin de comprendre la structure des réseaux neuronaux, de faire passer à des framework de *machine learning*, tels que PyTorch ou Keras+TensorFlow, afin de réduire les risques de problèmes de codes survenant dans l'implémentation des réseaux neuronaux plus complexes.

Bonnes pratiques Nous conseillons aux membres du groupe d'adopter dès le début de bonnes pratiques, qui ont pu être source de conflit les années précédentes ou simplement

afin d'éviter de produire du code pouvant larguer d'autres membres du groupe. Quelques unes sont listées ci-dessous :

- Adopter une convention de syntaxe de programmation par défaut.
- Mettre en place une documentation générée par Doxygen (on conseille le module plantUML pour la génération automatique de parties du diagramme UML).
- Faire usage des merge/pull requests validé par autrui

4.1.3 Affichage des résultats

Les images et courbes étaient initialement manuellement obtenues depuis des modèles de fichiers Excel. Afin de ne pas avoir à coder une seule ligne du language hérétique qu'est Python et son module GnuPlot, nous avons écrit un simple script bash (disponible sur le dépôt github) exploitant GnuPlot pour l'affichage des résultats.

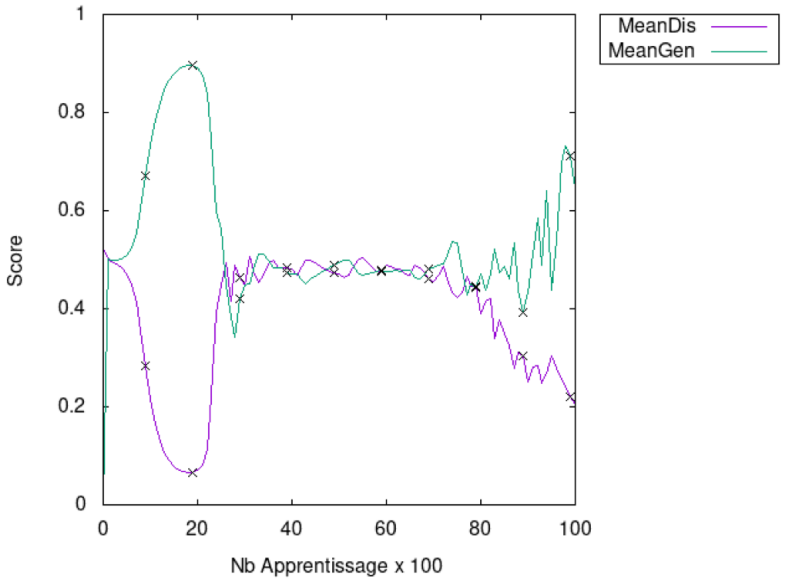


FIG. 4.1 – *Un exemple d'une exportation GnuPlot*

4.1.4 Rapport

Le rapport est fait sous LaTeX, avec des éditeurs et compilateurs en local ou via Overleaf (et son module git) ou Sharelatex (et son module Gitlab). La bibliographie BibTex est générée à partir de l'export de la base de donnée de documents Zotero. La compilation vers PDF en CI n'a pas été faite car Travis-CI ne le prend pas en charge facilement (nécessité de réinstaller latex sur les images docker à chaque compilation, ce qui est beaucoup trop lent). La compilation aurait été faisable, en utilisant une image gitlab-ci latex, si le projet avait été hosté sur Gitlab.com.

4.2 C++ et Problèmes rencontrés

4.2.1 C++11

Un des groupes précédents [3] ayant codé en C++ ont perdu une bonne partie de leur année à débugger des fuites de mémoire, dû à la gestion de pointeurs sous C++98. Nous sommes donc parti sur C++11 afin de faire un usage extensif des *smart pointers* N.B. Ce même groupe a perdu une autre bonne partie de l'année à vouloir recoder la multiplication marticielle et à la débugger. C'est la raison de la section ci-dessous

4.2.2 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 cœurs.

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 est entièrement à base de template et est organisée de façon à pouvoir effectuer le plus de calculs possible en compile-time, ce qui rend les types des objets et les messages d'erreur difficilement déchiffrables. Débugger un problème lié à Eigen nécessite un bon débogueur et de la patience.

Particularités La classe Eigen::MatrixXf ne sait pas faire du coefficient par coefficient, il faut passer par la classe Eigen::ArrayXf par une conversion soit : (matrice1.array() * matrice2.array()).matrix()...

Parallélisation des calculs 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.

Convolution Contrairement à Numpy, Eigen ne traite pas la convolution, il est nécessaire de la coder à la main. Cela nous a ralenti dans l'avancement du projet et dans l'obtention de résultats, notamment pour CIFAR qui nécessite des étapes de convolution.

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 cœur 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 [4].

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'œil 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 synthèse.

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 ces images (avoir plein de moutons différents).

L'ensemble de sortie du **Générateur** est donc un ensemble inclus 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.

Exemple 2. Génération du chiffre 8

On peut schématiser de la manière suivante un réseau générateur entraîné à générer des 8.

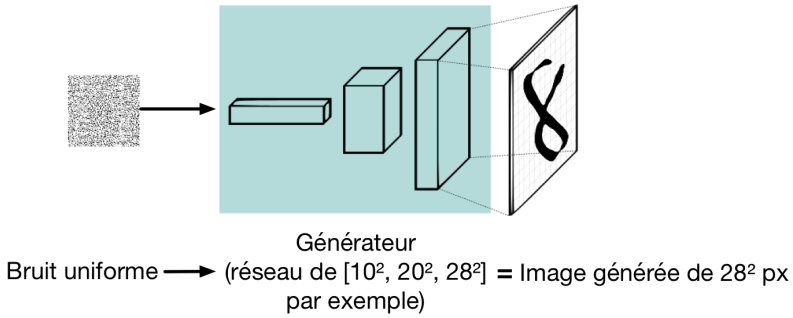


FIG. 5.1 – *Génération du chiffre 8 par un générateur* $100 \times 400 \times 784$

5.2 Discriminateur et Générateur

5.2.1 Notations

On note $\theta^{(D)}$ et $\theta^{(G)}$ les paramètres des réseaux, représentant les valeurs de la matrice de poids et de biais de chacun des réseaux.

On dispose de deux fonctions de coûts, notées $J^{(D)}(\theta^{(D)},\theta^{(G)})$ et $J^{(G)}(\theta^{(D)},\theta^{(G)})$, représentant respectivement les fonctions de coûts du Discriminateur et du Générateur. Dans leur notation, on montre bien que chacune d'elle dépendent des paramètres de chacun des deux réseaux. En effet, le Discriminateur apprend à discerner les fausses images issues du générateur des vrais images, tandis que le générateur apprend grâce aux indications fournies par le Discriminateur.

On note z l'entrée du générateur (du bruit blanc multidimensionnel), x une image réelle (issue d'une banque d'image), $G(z)$ la sortie du générateur (une image générée) et enfin $D(x)$ et $D(G(z))$ la sortie du discriminateur comportant en entrée respectivement une image réel et une image générée (le score donné respectivement à une image réelle et

à une image générée. x est alors une variable d'observation tandis que z est une variable *indirecte*.

D et G représentent alors des fonctions (différentiables) de paramètres $\theta^{(D)}$ et $\theta^{(G)}$. On appellera dans toute la suite $D(x)$ **et** $D(G(z))$ **les scores du discriminateur et du générateur**. On considérera souvent, à des intervalles d'apprentissages données, le score cumulé, appelé plus simplement score, défini comme la norme euclidienne d'un nombre donné de scores d'images (par exemple 20 images).

Remarque 4. (Bruit) On notera que le bruit n'est pas nécessairement présent uniquement sur la première couche, il peut être présent quel que soit la couche. Il a été implémenté avec succès (voir *NoisyLayer* dans [5]), et utilisé avec succès par le groupe Salamandre [6].

5.2.2 Théorie des jeux et équilibre de Nash

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

Comme toute fonction de coût, le discriminateur cherchera au cours de l'apprentissage à réduire la fonction de coût $J^{(D)}(\theta^{(D)},\theta^{(G)})$, en n'ayant de contrôle que sur $\theta^{(D)}$. De même, le générateur cherchera à réduire la fonction de coût $J^{(G)}(\theta^{(D)},\theta^{(G)})$, en n'ayant de contrôle que sur $\theta^{(G)}$. La solution à ce problème peut se traduire en théorie des jeux. On obtient alors un équilibre de Nash. On s'attend alors à ce que $D(G(z)) = D(x) = \frac{1}{2}$.

5.2.3 Les différentes distributions statistiques impliquées

Les distributions que l'on considère sont p_{data}, p_{model} et p_G. p_G correspond à la distribution représentée par le générateur, tandis que p_{model} est la distribution vers lequel on souhaiterait tendre. Néanmoins, cette distribution p_{model} est inatteignable car l'ensemble de la base d'apprentissage ne représente qu'une approximation de celle-ci, que nous noterons p_{data}. On cherche donc, par l'apprentissage semi-supervisé, à faire tendre p_G vers p_{data}.

Remarque 5. On pourrait introduire de plus une 4ème distribution p_D, qui correspondrait à l'interprétation du discriminateur de p_{data}. Ainsi, plus précisément, on chercherait au cours de l'apprentissage à faire tendre p_G vers p_D et p_D vers p_{data}. On a affiché cette distribution dans les graphiques relatifs au GAN 1D (en bleu) (section **??**). Néanmoins, p_D se construit simultanément (voir section 5.3.2) à partir de p_G et p_{data}, ce qui rend la réflexion peu aisée. On s'en tiendra donc à p_G et p_{data}.

5.3 Les fonctions de coût, l'apprentissage

5.3.1 Les différentes fonctions de coût

Discriminateur Le discriminateur n'a, dans le cas du GAN classique, qu'une fonction de coût qui soit utilisée :

$$J^{(D)}(\theta^{(D)},\theta^{(G)}) = -\frac{1}{2}\mathbb{E}_{x\ p_{data}} \log D(x) - \frac{1}{2}\mathbb{E}_z \log\left(1 - D(G(z))\right)$$

$-\frac{1}{2}\mathbb{E}_{x\ p_{data}} \log D(x)$ correspond à la capacité de D à reconnaître les vraies images, tandis que $-\frac{1}{2}\mathbb{E}_z \log\left(1 - D(G(z))\right)$ correspond à la capacité de D à reconnaître les résultats de

G.

Remarque 6. Dans l'implémentation, par équivalence de développement limité, nous avons cherché à minimiser $-\log(D(G(z))$ plutôt que $\log(1 - D(G(z))$.

Générateur Différentes fonctions de coût peuvent être utilisées pour le Générateur. Nous avons implémenté le MinMax, l'heuristique non saturante et le maximum de vraisemblance (aussi appelée KL-Divergence). La définition mathématique de chacune peut être trouvée dans [7].

5.3.2 Apprentissage par récompense ou par punition ?

La figure 5.2 illustre les 3 apprentissages successifs possibles pour le GAN. Habituellement, les 2 apprentissages du discriminateur se font un pour un, et on joue sur le rapport entre le nombre de d'apprentissages du générateur et celui du générateur (voir *nbGenTeach* et *nbDisTeach* dans le fichier de configuration [5]).

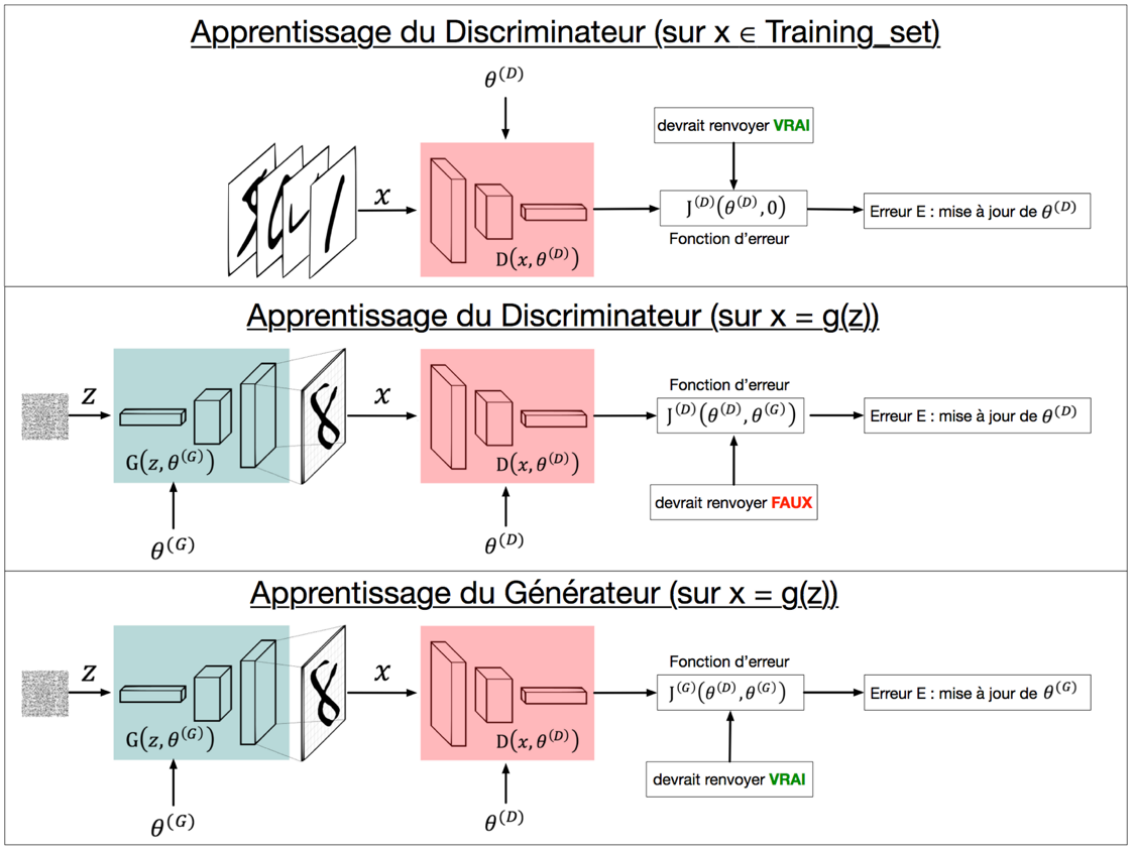


FIG. 5.2 – *Les 3 types d'apprentissages successifs appliqués au GAN*

5.3.3 Quand un réseau prend le pas sur l'autre

Lorsque le rapport d'apprentissage est mal paramétré, on assiste à une déroute de l'un ou l'autre des réseaux, et la génération de contenu est alors non-fonctionnelle. En effet, si le discriminateur est trop performant, il indiquera à chaque apprentissage du

générateur que l'image est une fausse image; le générateur n'aura alors aucune indication de la d̈irectionv̈ers laquelle s'orienter, l'apprentissage ne lui procure aucune information si ce n'est que la distribution p_G ne colle pas parfaitement à p_{data}; le générateur ne peut donc pas s'améliorer. A contrario, un générateur beaucoup plus performant que le discriminateur d'un point de vue fonction d'erreur (ce qui est beaucoup plus rare et moins problématique) est plus difficile à interpréter; cela indique que le discriminateur n'a pas assez appris et qu'il sur-apprendra par la suite. Dans les 2 cas, on observe un apprentissage moins efficace.

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'entraîne 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 reconnaître 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

La structure de notre *Generative Adversarial Networks* diffère assez peu de ce que nous avons implémenté pour MNIST : on suit le même principe, la classe *Application* ne possédant plus un réseau mais les deux réseaux de neurones nécessaires au GAN. Ainsi, une classe *Teacher* se charge d'apprendre aux deux réseaux en fonction des paramètres que l'on définit dans un fichier de configuration json. Enfin, pour une plus grande souplesse dans la création de nos réseaux, nous avons implémenté divers types de couches de neurones, avec NeuronLayer classe abstraite mère, dont découlent les couches bruitées (NoisyLayer), les couches de perceptrons classiques (FullConnectedLayer), et les différentes couches à convolution (traitées dans le chapitre suivant).

6.1.2 Taille des réseaux - Influence

Nombre de couches Les réseaux de nombre de couches complètement connectées inférieur ou égal à 2 sont majoritairement inefficaces. Des réseaux à 3 couches ont souvent été suffisants. Rajouter davantage de couches a peu d'influence, si ce n'est que le temps de calcul de la rétropropagation est grandement augmenté. Pour modifier davantage le comportement du réseau, il est nécessaire de recourir aux DCGANs (voir 7.3).

nombre de neurone par couche Des couches avec plus de 1000 neurones dans une couche ont souvent été peu efficaces. On a utilisé des couches avec souvent entre 50 et 800 neurones par couche.

6.1.3 Ratio d'apprentissage- Influence

En théorie : On a vu dans la partie précédente l'importance théorique de l'équilibre entre les deux réseaux : en effet, l'état idéal correspond à l'équilibre de Nash, où le générateur et le discriminateur sont tous deux au même niveau, et si l'un progresse, cela sera au détriment de l'autre (voir partie théorique sur les GAN). Cela correspondrait au niveau des scores que le générateur répondrait 0.5 à toutes les demandes : il ne saurait pas si l'image est réelle ou fictive. Le générateur ne serait pas encore assez fort pour entièrement tromper le discriminateur, mais il ne serait pas non plus complètement démasqué par le discriminateur.
Cet état d'équilibre devrait permettre aux deux réseaux d'apprendre simultanément. Pour équilibrer le système et atteindre cet état, un paramètre sur lequel il serait intéressant de jouer est le ratio d'apprentissage : lors d'un cycle d'apprentissage, combien de fois devons-nous faire apprendre le générateur puis combien de fois devons-nous entrainer le discriminateur ? Il y a deux aspects à prendre en compte pour ce paramètre : tout d'abord, adapter cette cadence d'apprentissage permet de rééquilibrer les deux réseaux. De plus, il peut être intéressant d'entrainer le générateur un grand nombre de fois par rapport à un discriminateur fixé, permettant alors un apprentissage plus précis. Le générateur essaiera alors de viser une cible fixe, et apprendra de manière plus efficace.

En pratique : En pratique, ce paramètre a été très difficile à adapter : en effet, l'équilibre de Nash a été en pratique impossible à obtenir, le générateur étant la majorité du temps largement au dessus du générateur. De plus, le système est trop instable pour permettre un tel équilibre. Ainsi, la cadence d'apprentissage ne nous permettait que d'aider le générateur, lui permettant de garder le rythme un peu plus longtemps dans la course contre le discriminateur.

Nous verrons d'autres méthodes qui jouent directement sur ces apprentissages, et qui permettent alors d'aider le générateur à chaque itération, dans la partie suivante, ce problème étant récurrent dans notre mise en place des GANs.

6.1.4 Taille des entrées - Influence

Les entrées, qui consistent en du bruit $\in [-1,1]$ doivent être suffisamment nombreuses. Bien qu'une entrée de bruit soit réelle (double flottant dans l'implémentation) et qu'elle devrait en théorie correspondre à une infinité d'entrées, les résultats [ref. nécessaire] montrent que le nombre d'entrées doit être idéalement au moins aussi grande que le nombre de

sorties. Une solution couramment adoptée est l'utilisation d'entrées, donc de bruit, sur les autres couches du générateur. On aura alors une entrée sur la première couche du réseau de neurones, et les autres couches peuvent alors être bruitées [7]. Nous traiterons cette méthode dans ce chapitre sur le point du *Mode Collapse* du Générateur.

6.2 Premiers résultats

Les résultats initiaux pour les GANs furent très encourageants. Les courbes de score du générateur (voir 5.2.1 pour la définition du score) avaient toujours la même forme, tandis que la courbe du discriminateur était parfois symétrique par rapport à la droite 0,5, mais pas dans tous les cas. On fait figurer ici quelques-uns de nos résultats initiaux

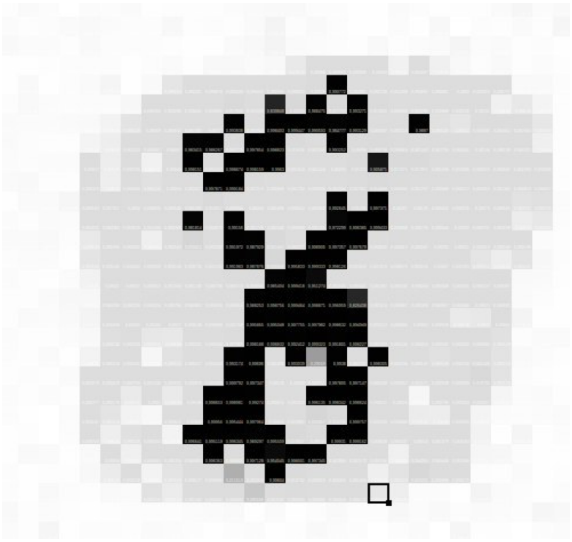


FIG. 6.1 – *Chiffre 8 généré un mois après l'implémentation effective du GAN - le temps de trouver des paramètres de réseau qui fonctionnent*

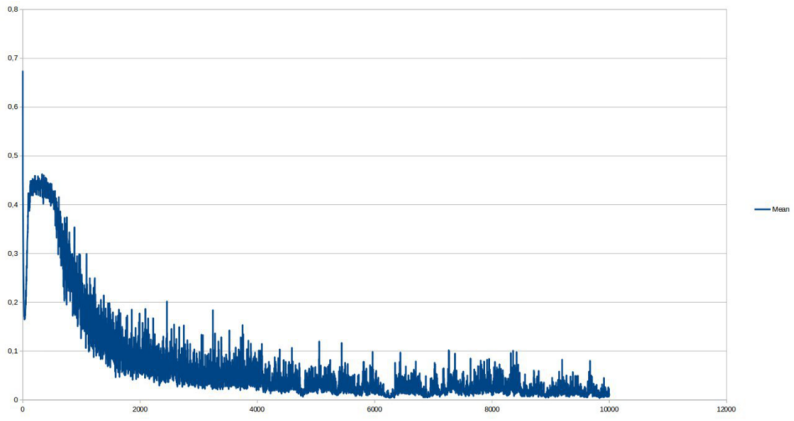
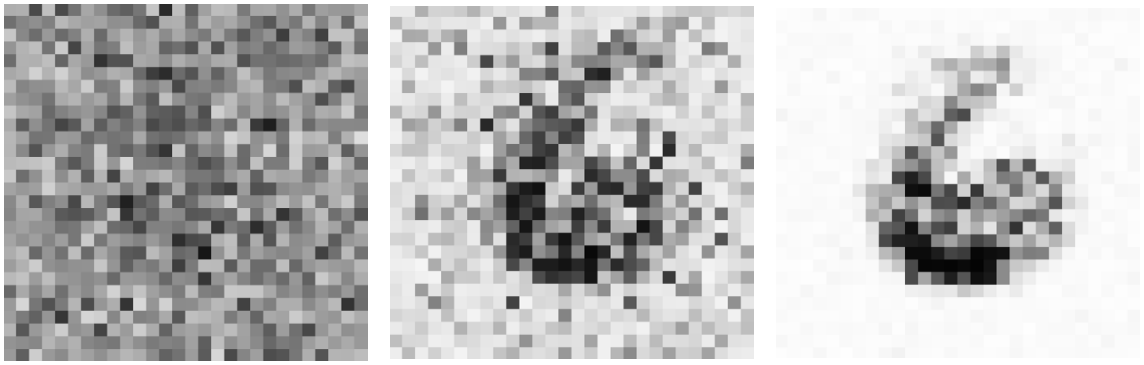


FIG. 6.2 – *Apprentissage du Générateur associée au chiffre 8 - on affiche le score du générateur* $D(G(z))$



5000 apprentissages | 10000 apprentissages | 15000 apprentissages

50000 apprentissages | 90000 apprentissages | 95000 apprentissages

FIG. 6.3 – *affichage successif du chiffre 6 à différents moments de l'apprentissage*

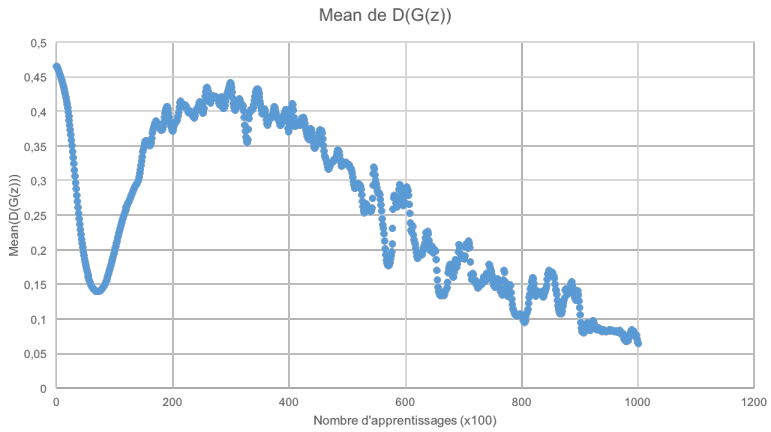


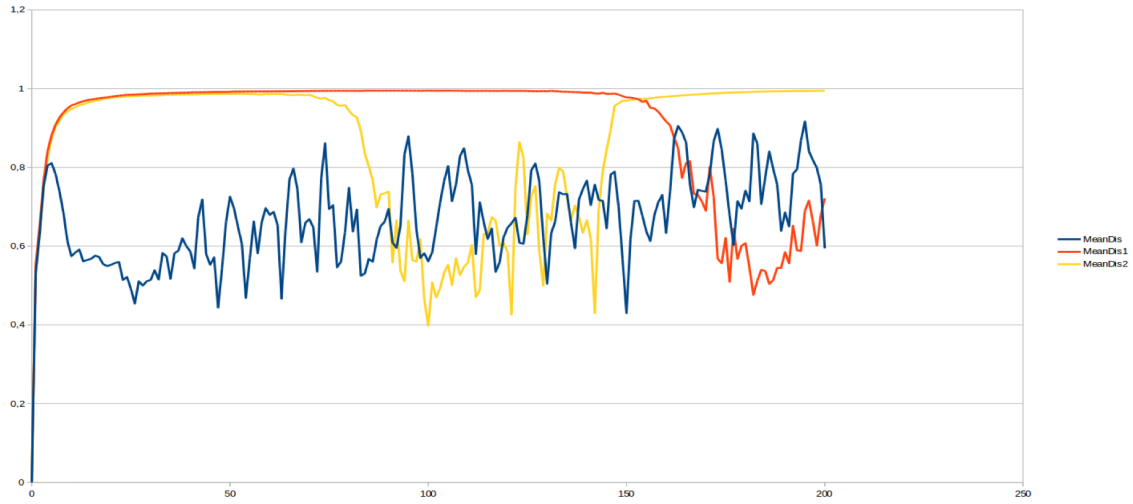
FIG. 6.4 – *Apprentissage du Générateur associée au chiffre 6 - on affiche le score du générateur $D(G(z))$*

affichage successif d'un chiffre à différents moments de l'apprentissage

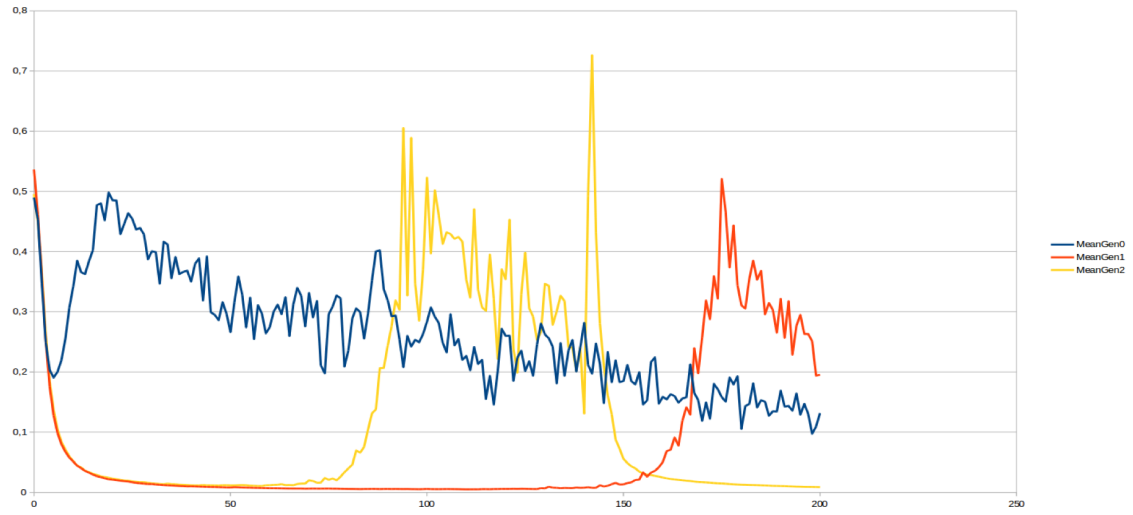
6.2.1 Comparaison des fonctions de coût du générateur

Différentes fonctions de coût peuvent être utilisées pour le Générateur. Nous avons implémenté le MinMax, l'heuristique non saturante et le maximum de vraisemblance

(aussi appelée KL-Divergence), et les avons comparées. On affiche en figure 6.5 le score du discriminateur et générateur pour chacune des fonctions de coût du générateur. On montre à titre de comparaison le meilleur 3 obtenu pour chaque fonction de coût à la figure 6.6. Le meilleur 3 a à chaque fois été obtenu dans la zone de plus forte variation. Avant le réseau n'avait pas appris, après le réseau était passé en mode collapse (voir pour plus d'explications). Dans le cas de l'heuristique non-saturante, l'image est prise proche au premier pic, qui est un maximum global du score du générateur.



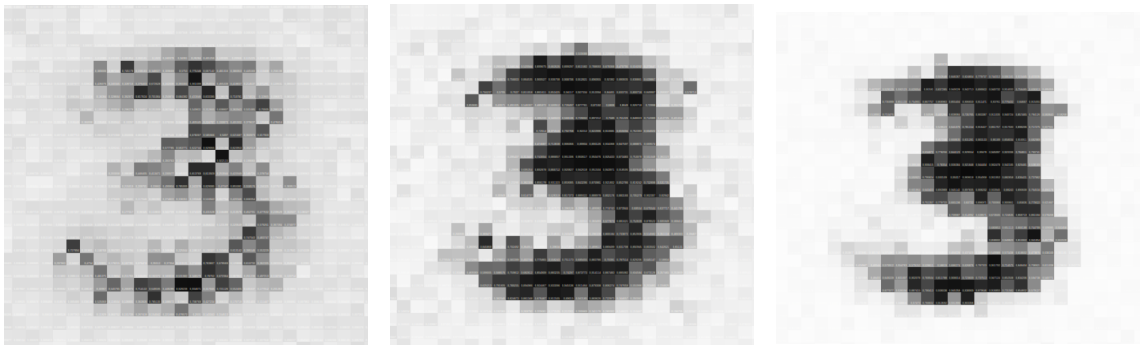
Score du discriminateur $D(x)$



Score du générateur $D(G(z))$

FIG. 6.5 – *Heuristique non saturante en bleue, KL en rouge, MinMax en jaune*

On remarque que la KL-divergence et le MinMax prennent beaucoup de temps à se mettre en place et collapse rapidement, tandis que l'heuristique non-saturante permet d'obtenir un résultat satisfaisant très tôt dans l'apprentissage. Nous avons effectué pour ces raisons par la suite la majorités des tests avec l'heuristique non-saturante.



Heuristique non saturante | KL-Divergence | MinMax

FIG. 6.6 – *Comparaison de la qualité d'image pour 3 fonctions de coût du générateur différentes*

6.3 Difficultés rencontrées et solutions émises

Au cours du développement des GANs, nous avons été confrontés à de très nombreuses difficultés, que nous avons tenté de répertorier avec les solutions envisagées. À l'heure actuelle, certains problèmes sont toujours sans solution probante et restent des domaines de recherches ouverts : c'est le cas en particulier du **mode collapse** que nous détaillons ci-après.

6.3.1 Le problème d'évaluation des performances

Le premier et plus grand problème que nous avons rencontré est **l'absence de méthode systématique d'évaluation des performances d'un réseau donné.**

En effet, pour qu'un protocole expérimental puisse être mené à bien, il faut comparer un résultat témoin avec les résultats pour lesquels des paramètres ont varié, et déduire de la différence des performances si le paramètre testé est signifiant, et dans quelle mesure.

Ce protocole est **fondamental** en recherche expérimentale, et il est **inapplicable** ici. En effet, nous n'avons trouvé aucun moyen de mesurer la qualité d'une image, à l'exception de la mesure à l'œil par les membres de l'équipe (ce qui ne peut être qualifié de méthode rigoureuse et systématique d'évaluation).

Les données à disposition pour mesurer les performances

Afin de bien cerner le problème, résumons ce que nous avons à disposition à la fin d'une expérience. Si dans certains cas nous avons fait des essais nous permettant de mesurer d'autres grandeurs que les suivantes (comme la valeur des poids etc), seules les deux ci-après sont accessibles systématiquement, pour chaque expérience. Nous disposons donc de :

- Un ensemble d'images de synthèse générées à différents instants d'apprentissage; par exemple nous pouvons demander à produire 100 images tous les 10 apprentissages, ce qui permet de suivre assez finement l'évolution du GAN au fur et à mesure de son apprentissage
- Les courbes de score des deux réseaux, générées à la fin de l'expérience. La courbe de score d'un réseau représente la proportion d'erreurs dans toutes les réponses qu'il

donne. Ainsi la courbe de score du discriminateur compte le nombre de bonnes réponses du réseau à des entrées aussi bien de synthèse (réponse attendue : 0) que réelles (réponse attendue : 1); la courbe du générateur est calculée en fonction des réponses du discriminateur aux images de synthèse. Ces courbes sont graduées en abscisse par rapport à nombre d'apprentissages effectués.

Sur l'utilisation des courbes de score À première vue, ces courbes paraissent être un indicateur pertinent de la qualité des réseaux. Elles permettent **en théorie** :

- d'observer l'équilibre de Nash (qui, en pratique, n'a jamais été observé)
- d'observer les déséquilibres entre les deux réseaux afin d'affiner les paramètre d'équilibre sur les expériences suivantes
- de comparer les images produites avec les scores des réseaux au moment de la génération de l'image

En pratique, nous n'avons jamais réussi à interpréter efficacement ces courbes. Leur étude représente une perte pure de temps, temps qui aurait dû être alloué à la recherche d'une méthode d'évaluation efficace. Les courbes paraissent extrêmement instables, on peut observer des formes récurrentes mais très difficiles à expliquer (chute brutale de l'une des deux malgré une bonne qualité d'image, etc), et par dessus tout nous n'avons pas réussi à mettre en lumière des corrélations entre les paramètres choisis et les courbes résultantes.

Enfin, nous avons été incapables de trouver des corrélations pertinentes entre la qualité des images et les courbes de score. Nous avons au mieux été capables de trouver des explications à posteriori sur les liens qui peuvent exister; c'est à dire que nous n'avons jamais pu prévoir (de manière reproductible et indépendante du hasard) la qualité des images à partir de la seule analyse des courbes.

Sur l'analyse des échantillons d'images Deux problèmes principaux s'opposent à l'analyse

Évaluation par réseau(x) tiers

Une première idée pour l'évaluation, serait de passer les images de synthèses dans un $3^{\text{ème}}$ réseau (appelons le évaluateur, mais il se comporte exactement comme le discriminateur) qui aurait appris à part, en dehors de la structure de GAN et qui serait un juge objectif sur les résultats du GAN. Ce réseau aurait été entraîné sur les données réelles, et resterait immuable une fois son apprentissage terminé.

Cette méthode est en pratique inapplicable (et inappliquée, pour ce que nous en savons), notamment à cause de la difficulté à équilibrer ce $3^{\text{ème}}$ réseau avec les deux autres; équilibrer un GAN est déjà une tâche particulièrement ardue (nous y reviendrons), ajouter cet évaluateur la rend insurmontable. Car si l'évaluateur est très performant, il risque de donner une performance nulle à tous les générateurs qu'il testera; à l'inverse si on l'a délibérément fait peu apprendre, il ne permettra pas de discriminer efficacement les générateurs qu'il testera.

On pourrait envisager de créer une échelle d'évaluateurs, constituée de plusieurs évaluateurs ayant chacun arrêté leur apprentissage à un moment différent, afin de pouvoir graduer les résultats obtenus par les générateurs sur une échelle de performance. Nous n'avons pas nous même implémenté cette technique et n'avons pas trouvé d'articles faisant mention de l'application de cette méthode, mais elle a au moins l'avantage majeur d'être rigoureuse, systématique et indépendante du jugement humain.

6.3.2 Mode Collapse du *Générateur* 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. Le discriminateur apprendra alors à se méfier de cette zone, puisque, même si elle est très proche de la distribution d'origine, il apprend au fur et à mesure à reconnaître les images du générateur. Ainsi, cette zone est peu à peu considérée comme artificielle pour le discriminateur, et le générateur va collapse sur une autre zone considérée comme valide. On aura alors un jeu de rebonds, le générateur collapse sur des zones qui seront alors désapprouvées par le discriminateur, et se déplacera vers une autre zone. On a donc un cercle qui fera que le générateur sera et restera dans un état de collapse. Il faut alors trouver des méthodes pour empêcher le générateur de ne générer qu'une distribution très restreinte (un mode).

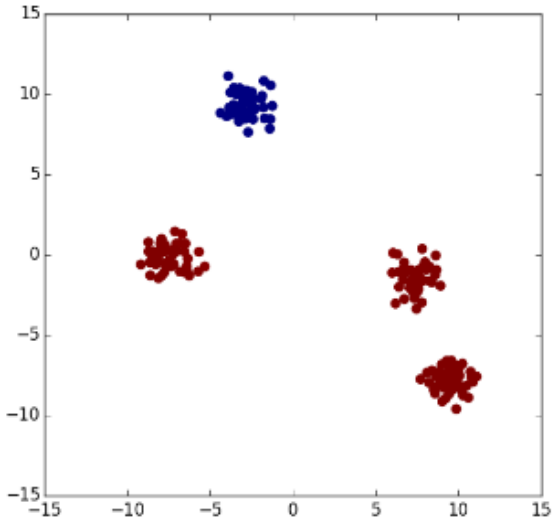


FIG. 6.7 – *Missing modes : les points représentent les éléments de la distribution de données réelles. En bleu l'ensemble de sortie du GAN appartenant à la distribution réelle, et en rouge les zones de la distribution initiale non couverte par le GAN.*

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.

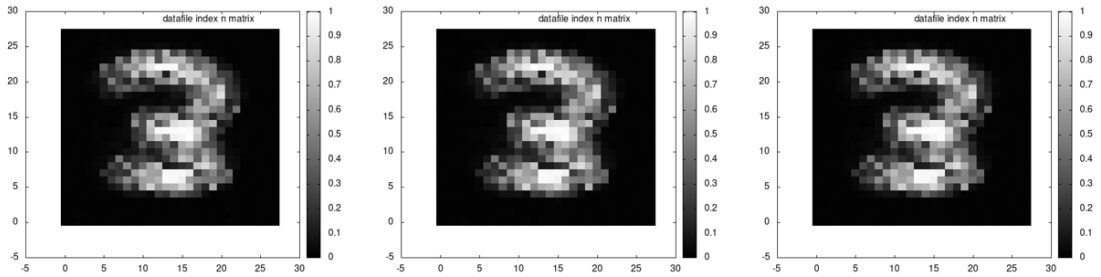


FIG. 6.8 – *Missing modes : On obtient pour un même réseau, 3 échantillons générés qui sont sensiblement les mêmes.*

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 :

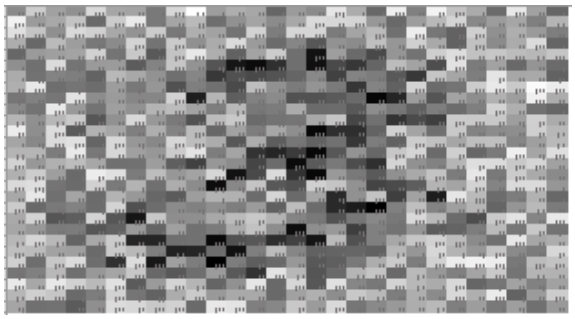


FIG. 6.9 – *Masque du biais : on peut observer que le biais forme lui même les bases d'une image, un 3 ici.*

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)

Une solution est proposée par Ian Goodfellow dans son article dans NIPS 2016 intitulé **Tutorial : Generative Adversarial Networks** [7]. La structure du générateur est assez libre dans l'architecture des GAN, ainsi, on peut placer des entrées (du bruit) au début du réseau, mais aussi sur n'importe quelle couche de celui-ci notamment, sans réellement compromettre les performances. En effet, injecter des entrées de bruit dans des couches intermédiaires revient à faire des entrées classiques qui traverseront moins de couches de neurones. Cependant, le bruit au niveau de la dernière couche semble perturber le réseau et l'empêche de tomber dans un mode collapse.

Cela peut s'expliquer par le fait que les entrées bruitées perturbent la dernière couche du réseau, l'empêchant de se stabiliser sur une unique sortie trompant le discriminateur. Cette perturbation va forcer le générateur à diversifier ses sorties. Une autre hypothèse serait que la dernière couche ne peut pas différencier les entrées provenant des couches antérieures des entrées de bruit si celles-ci sont suffisamment nombreuses. Ainsi, il sera obligé de toutes les prendre en compte (il ne peut pas juste annuler les poids correspondant) ce qui générera de la variabilité dans la sortie, puisqu'une partie non négligeable des entrées n'est que du bruit.

Cependant on peut observer dans des réseaux bruités, pour un même nombre d'apprentissages, des patterns généraux dans chaque image, ce qui nous montre un mode collapse partiel mais très léger comparé au précédent. De tels réseaux ont de plus besoin de beaucoup plus d'apprentissages pour donner des résultats convaincants. Cette solution nous permet donc de limiter partiellement ce problème de mode collapse.

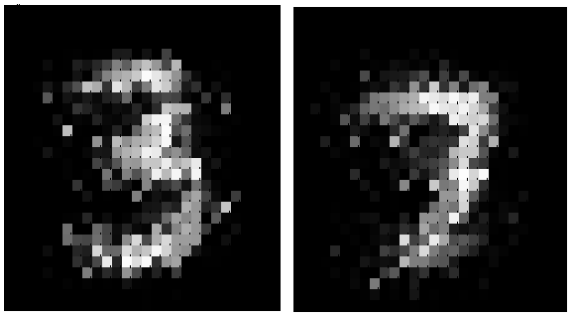


FIG. 6.10 – *Mode collapse éliminé : pour un même nombre d'apprentissages (8500 ici) on obtient un 3 et un 7 à l'aide d'un réseau bruité*

Chapitre 7

Generative Adversarial Networks - Améliorations

7.1 Apprentissage par Mini-Batch

7.1.1 Principe

L'apprentissage par Batch consiste à apprendre sur l'ensemble du batch d'apprentissage avant de mettre à jour chaque réseau, et de réitérer l'apprentissage à chaque itération sur l'ensemble du batch d'apprentissage. Il s'oppose à l'apprentissage habituel, dit *stochastique* (mais qui n'a rien de particulièrement stochastique), qui consiste à rétropropager l'erreur et mettre à jour les poids image par image.

On calcule successivement la rétropropagation pour chaque image, on somme l'erreur sur l'ensemble des images et on applique la mise à jour du réseau avec la somme des erreurs. On obtient donc une moyenne des directions de propagation afin d'effectuer la mise à jour, afin de converger de manière plus précise.

Certaines bases d'apprentissage étant très grandes - soixante mille avec MNIST par exemple -, on va préférer à l'apprentissage par batch, l'apprentissage par minibatch. On sélectionne aléatoirement parmi la base d'apprentissage et de tests un nombre plus petit d'éléments. Les valeurs sont habituellement de l'ordre de la dizaine d'éléments.

7.1.2 Implémentation

Mise à jour des poids Afin de pouvoir effectuer la rétro-propagation uniquement à la fin de l'itération du minibatch, nous avons simplement

Sets d'apprentissage et de tests. plus petits Dans le cas de MNIST, les chiffres du début de la base d'apprentissage étant plus simple, nous avons, afin d'obtenir de meilleurs résultats, parfois effectué une sélection aléatoire sur uniquement une première partie des 60000 images d'apprentissage (cf argument *labelTrainSize* du fichier de configuration [5]) et des 10000 images de tests (cf argument *labelTestSize*.

7.1.3 Résultats

Notre équipe a obtenu des résultats corrects mais peu intéressant, et très différents des résultats relativement exotiques du groupe Salamandre [6]. Globalement, l'apprentissage s'effectue légèrement plus rapidement pour un même nombre d'itérations, au prix d'une

augmentation du temps de calcul par itération (proportionnel selon la taille des batch). Le temps de calcul était donc quasiment multiplié par la taille du batch pour un résultat équivalent en stochastique (voire pire dans le cas du minibatch 1, l'apprentissage stochastique ayant par ailleurs été optimisé, comme on peut le remarquer sur la figure ci-dessous en comparant la courbe orange de la jaune).

La technique de minibatch, bien qu'intéressante dans des cas précis, tel que l'apprentissage simultané de plusieurs chiffres au même temps, a donc été par la suite abandonnée.

FIG. 7.1 – *Apprentissage par minibatch 100 (marron et gris) et minibatch 10 (vert et cyan) d'un GAN sur MNIST comparé à un apprentissage stochastique (orange et bleu clair)*

7.2 Algorithmes de pas d'apprentissage adaptatif

7.2.1 Principe

L'objectif de ces algorithmes est de paramétrer au mieux la convergence de la distribution représentée par notre réseau vers la distribution p_data. Le pas adaptatif n'est alors plus constant; il varie en fonction du gradient des étapes d'apprentissage précédentes, de manière plus ou moins évoluée selon les algorithmes.

L'objectif est ainsi de faire évoluer le pas en fonction des erreurs, selon diverses formules. Deux méthodes de pas adaptatifs ont été utilisées : RMSprop et Adam. Nous en présentons ici d'autres.

7.2.2 Adagrad

Cette première méthode de descente adaptative suit les équations suivantes :

$$\begin{aligned} g_{t+1} &= g_t + (\frac{\partial J}{\partial W})^2 \\ W_{t+1} &= W_t - \frac{\eta}{\sqrt{g_{t+1}} + \epsilon} \frac{\partial J}{\partial W} \end{aligned} \tag{7.1}$$

Adagrad n'a pas été implémenté en raison des piètres performances annoncées[ref nécessaire]. En effet, g est strictement croissante, étant incrémenté à chaque itération de $(\frac{\partial J}{\partial W})^2$. Les réseaux apprenant sur un grand nombre d'itération, le coefficient g_t devient rapidement très important, et le pas en $\frac{1}{\sqrt{g_{t+1}}}$ devient rapidement négligeable. Les performances de cet algorithme sont donc assez faibles.

7.2.3 RMSprop

La RMSProp permet de résoudre ce problème. En effet, on a un décroissance exponentielle de l'influence de l'erreur dans le temps grâce au coefficient γ correspondant au moment d'ordre 2. Ainsi, l'influence des anciennes valeurs de $(\frac{\partial J}{\partial W})^2$ décroit exponentiellement, les erreurs les plus récentes sont donc les plus influentes sur le pas.

$$\begin{aligned} g_{t+1} &= \gamma g_t + (1-\gamma)(\frac{\partial J}{\partial W})^2 \\ W_{t+1} &= W_t - \frac{\eta}{\sqrt{g_{t+1}} + \epsilon}\frac{\partial J}{\partial W} \end{aligned} \tag{7.2}$$

7.2.4 Adam

$$\begin{aligned} g_{t+1} &= \gamma g_t + (1-\gamma)(\frac{\partial J}{\partial W})^2 \\ \hat{g}_t &= \frac{g_t}{1-\gamma^t} \\ m_{t+1} &= \delta g_t + (1-\delta)\frac{\partial J}{\partial W} \\ \hat{m}_t &= \frac{m_t}{1-\delta^t} \\ W_{t+1} &= W_t - \frac{\eta}{\sqrt{\hat{g}_{t+1}} + \epsilon}\hat{m}_t \end{aligned} \tag{7.3}$$

La méthode de descente de gradient AdamProp comprend deux aspects importants : le moment d'ordre 2 est conservé par rapport à la RMSProp, on a donc de nouveau cette décroissance exponentielle de l'influence des anciennes erreurs, qui vont pondérer le pas. De plus, elle prend en compte directement les anciennes valeurs de l'erreur, en remplaçant le terme d'erreur $\frac{\partial J}{\partial W}$ par m_t qui correspond à un moment d'ordre 1 (moyenne). On garde donc en mémoire la tendance récente de l'erreur, ce qui permet d'adapter de façon plus précise le pas. $\hat{g}_t$ et $\hat{m}_t$ sont des estimateurs corrigeant des biais si les pas sont trop faibles. Adam est considéré comme plus performante que RMSProp.

7.2.5 Utilisation

La plupart de nos réseaux semblaient présenter une faiblesse au niveau du générateur, dont le score était trop souvent très faible, par rapport à un discriminateur très performant. Une meilleure descente pourrait donc accélérer et rééquilibrer ces apprentissages.

Utiliser la RMSProp sur les deux réseaux permet d'apprendre des chiffres de belle qualité très rapidement (convergence jusqu'à 10 fois plus rapide que sous une descente de gradient normale), cependant on observe toujours ce déséquilibre entre les deux compétiteurs. Ce système semblait étrangement relativement instable, les paramètres amenant à la convergence étant assez restreints.

L'utilisation d'Adam pour le générateur et de RMSProp pour le discriminateur (ayant comme but d'entraîner mieux le générateur) semble bien fonctionner. La qualité des images est plus variable (le générateur produit des images laides et belles). En revanche, l'utilisation de RMSProp pour le générateur et d'Adam pour le discriminateur ne fonctionne pas. Cela corrobore les résultats présentés par d'autres [8][ref nécessaire : autre ref].

Enfin, si cela nous permet d'avoir un apprentissage plus performant, avec des images relativement belles en permettant au générateur de rester dans la course, cette solution ne permet pas de traiter le mode collapse : au contraire, l'apprentissage étant plus précis et rapide, on tombe très rapidement dans des états de mode collapse.

Ces apprentissages plus performants sont importants dans le sens où ils permettent un apprentissage plus performant des réseaux : celui-ci devrait être alors à la fois plus profond et plus précis. Cela permettrait alors de travailler correctement avec des réseaux plus grands, et des bases de données plus complexes avec des couches convolutives (CIFAR...).

7.3 Réseaux de convolution

7.3.1 Implémentation

Comme indiqué au 4.2.2, nous avons dû coder la convolution à la main. Afin de mettre en place un réseau à convolution, nous avons implémenté les différentes couches nécessaires : couche à convolution, couche de *zero-padding* et couche de *max-pooling*. Néanmoins, l'implémentation en C++ du GAN à convolution ne s'est jamais comportée comme souhaitée. En effet, quel que soit le paramétrage, le réseau produit des résultats comparables à ceux obtenus avec un perceptron simple. Cela a donc eu des répercussions sur nos avancées (par exemple les tests sur la banque d'image CIFAR10 n'ont pas donné de résultats concluants - nous ne les présenterons donc pas).

7.3.2 Résultats

Un réseau de configuration $784 \times 625 \times 100 \times 10$ avec uniquement une couche à convolution donne une erreur de 10% comme classificateur de MNIST. [Manu : développer résultats du 25 mai (voir EdL et slides)]

7.4 Échanges avec Giuseppe Valenzise

Dans le domaine d'évaluation de la qualité des images, on distingue l'estimation avec référence (comparaison par rapport à une image) de l'estimation sans référence. Maître de conférences au laboratoire L2S de CentraleSupélec, Giuseppe Valenzise est spécialiste en évaluation de qualité d'image (*Visual Quality Assessment*) sans référence. Les conclusions auxquelles nous avons pu aboutir au cours de l'échange étaient les mêmes que celles que nous avions pu formuler par le passé. Cela a néanmoins permis de les clarifier, et de nous confirmer l'intérêt du Wasserstein GAN.

– L'évaluation d'une qualité d'image standard passe très souvent par l'usage de métriques (i.e. fonctions) appliquées sur les valeurs des pixels. Une métrique souvent proposée est NIQE (*Naturalness Image Quality Evaluator*), qui consiste en la recherche de statistiques gaussiennes locales avec des coefficients mscn (*mean subtracted and contrast normalized*). Néanmoins, les images utilisées dans notre étude (MNIST, CIFAR) sont beaucoup plus petites que des images standard (28x28 et 32x32 au lieu

de plusieurs millions de pixels par image). L'estimation des gaussiennes est fortement bruitée, les métriques ne sont donc pas efficaces.
- Les méthodes avancées d'évaluation de qualité d'image consistent en l'utilisation de classificateurs. On cherche à extraire des caractéristiques des images (vecteurs, symétries, ...) et on crée des classificateurs qui décident sur la base de ces caractéristiques. Le travail consiste alors à trouver les meilleures caractéristiques d'image.
- Dans le cas des réseaux neuronaux, la distance entre les poids du réseau constitue une des caractéristiques les plus exploitées. G. Valenzise considère que dans le cas du GAN, la fonction de coût du discriminateur devrait constituer un des meilleurs classificateurs. Or dans le cas d'un GAN classique, nous avons pu remarquer qu'elle ne suit pas toujours la qualité des images.
- Il nous a suggéré de réfléchir à un « réseau de classification » qui permettrait de s'affranchir de ces problèmes. Or, un tel réseau de classification existe déjà dans la littérature : Le WGAN.
- Le WGAN permet d'obtenir un meilleur lien entre qualité d'image et fonction de coût du discriminateur, le classificateur *fonction de coût du réseau discriminant* est amélioré, en modifiant profondément sa structure.

7.5 Pistes de recherche

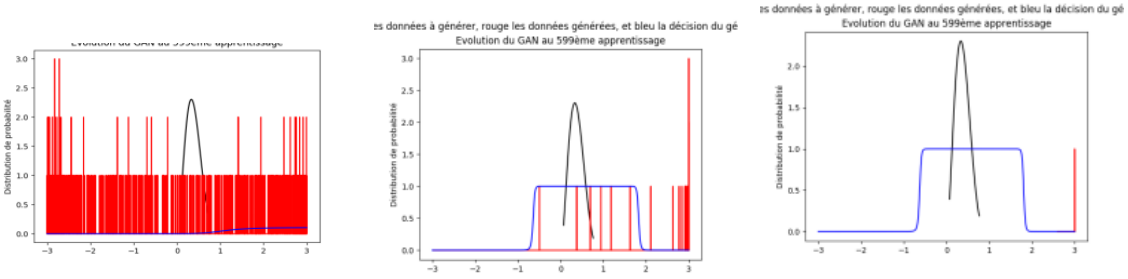
Cette section regroupe les pistes de recherche que nous avons pu rencontrer mais qui n'ont pas été approfondie faute de temps. Certaines pistes de recherches ne bénéficient pas de papiers à leur sujet.

- Étudier l'impact du nombre d'images disponibles dans la base d'entraînement. A priori, s'il est petit, l'effet d'apprentissage « par cœur » peut apparaître. Proposé par la chef de l'équipe de TaO du LRI.

7.5.1 Generative Adversarial Networks 1D : Visualisation en une dimension

Afin de mieux comprendre le fonctionnement du GAN, nous nous sommes intéressés à l'étude du GAN à une dimension. Le principe consiste à remplacer la banque d'image, de dimension 784, par des entrées à une dimension. On peut l'interpréter comme un scalaire, ou comme un pixel. La distribution représentée p_{data} est alors une distribution unidimensionnelle, que nous allons pouvoir représenter graphiquement, plutôt qu'une distribution à 784 dimensions.

Les résultats ne furent pas concluants. En effet, la distribution du générateur ne s'approche pas du tout de la représentation p_{data} comme on peut le voir à la figure 7.2. On fait figurer en noir la distribution cible p_{data}, en bleu le score attribué par le discriminateur, et en rouge un histogramme des résultats du Générateur.



Instant initial.

Apprentissage en cours

après un certain nombre d'apprentissage.

FIG. 7.2 – *Graphique du GAN 1D. La distribution prend ses valeurs entre -3 et 3* Initialement la distribution p_G est une distribution uniforme, ce qui est conforme aux attentes considérant que les poids du réseau sont initialisés suivant une loi uniforme. Après apprentissage, le discriminateur cherche à faire apprendre une distribution proche de p_{data}. Néanmoins, après un nombre supérieur d'apprentissage, le générateur est perdu. Dans notre cas, l'interprétation est la suivante : le générateur est allé dans un minimum local qui ne correspond pas au minimum que l'on cherche à atteindre.

Chapitre 8

Generative Adversarial Networks - Wasserstein GAN

Avertissement : certains éléments présentés ici ont pu être incompris. On conseille au lecteur le plus grand sens critique.

8.1 Objectif

L'objectif du passage à un Wasserstein GAN est de résoudre deux problèmes majeurs. Le premier est que les scores du générateur et discriminateur ne sont pas reliés à la qualité de l'image. Le WGAN permet [9] l'établissement d'un lien entre la qualité d'image et le score du discriminateur qui joue le rôle de classificateur. Le second problème concerne la stabilité de la génération d'image, en fonction du nombre d'apprentissage ou en fonction des paramètres.

8.2 Théorie

L'idée globale consiste à remplacer le discriminateur par un *critique*. La différence majeure réside dans la descente de gradient du critique. La descente du gradient du Discriminateur s'écrit [4]: $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))]$. Le second terme de la somme dépend de l'évaluation de la fonction de coût du discriminateur appliqué à une image $G(z^{(i)})$ issue du générateur. L'évolution du discriminateur dépend donc de l'état d'apprentissage du Discriminateur. Le réseau est purement concurrentiel. La théorie des jeux s'applique. Dans le cas du critique, on cherche à calculer la divergence entre la distribution p_r qu'on cherche à représenter et la distribution p_θ décrite par le critique. On l'écrit sous la forme suivante, appelée métrique de Wasserstein à l'ordre 1 :

$$W(\mathbb{P}_r,\mathbb{P}_g) = \inf_{\gamma \in (\mathbb{P}_r,\mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma} [||x - y||]$$

Cette métrique n'est néanmoins pas exploitable par un réseau neuronal. On fait alors appel à un théorème qui n'a rien à voir avec les réseaux neuronaux, celui de la dualité Kantorovich-Rubinstein (développé dans [10]), pour aboutir à une divergence équivalente (démonstration originale dans le papier [9], des démonstrations plus abordables sont présentes dans [11] et [12]), sous la forme suivante:

$$W(\mathbb{P}_r,\mathbb{P}_g) = \sup_{||f||_L<1} \mathbb{E}_{x\sim\mathbb{P}_r} [f(x)] - \mathbb{E}_{x\sim\mathbb{P}_g} [f(x)]$$

Le critique a alors pour objectif la recherche de cette borne supérieure. L'estimation de la divergence de Wasserstein ne dépend alors que de la valeur des poids du critique, et non du Générateur.

8.3 Implémentation et résultats

Pour des raisons de structure de code et d'avancement dans le projet, le groupe Couleuvre n'a pas réussi à faire fonctionner un GAN à Critique. Nous conseillons de se référer au rapport du groupe Salamandre.

Bibliographie

[1] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag.

[2] Anton Menshov. Clang + OpenMP Setup in Xcode. `http://antonmenshov.com/2017/09/09/clang-openmp-setup-in-xcode/`, 2017.

[3] Maxime Amossé, Julien Hemery, Hugo Hervieux, Sylvain Pascou, Arpad Rimmel, and Joanna Tomasik. PinaPL Réseaux de neurones & LSTM. `https://github.com/supelec-lstm/PinaPL-report`, 2017.

[4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[5] James Barrios, Antoine Cordelle, Benjamin Mouscadet, and Emmanuel Tran. GAN C++ Implementation. `https://github.com/Supelec-GAN/Couleuvre-GAN`, 2018.

[6] François Bouvier d'Yvoire, Matthieu Delmas, Romain Poirot, and Paul Witz. Dessine-moi un mouton. `https://github.com/Supelec-GAN/Salamandre-Rapport/blob/master/main.pdf`, 2018.

[7] Ian Goodfellow. NIPS 2016 tutorial: Generative adversarial networks. In *arXiv preprint arXiv:1701.00160*, 2016.

[8] Vincent Auriau, Laurent Beaughon, Marc Belicard, Yaqine Hechaichi, Thaïs Rahoul, Pierre Vigier, Joanna Tomasik, and Arpad Rimmel. Apprentissage Automatique de séquences. `https://github.com/supelec-lstm/appartement-report/raw/master/compte-rendu.pdf`, 2017.

[9] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[10] Cédric Villani. Optimal Transport: Old and New. *Springer*, 2008.

[11] Alexander Irpan. Read-through: Wasserstein GAN, 2017.

[12] Vincent Herrmann. Wasserstein GAN and the Kantorovich-Rubinstein Duality, 2017.

[13] Andrej Karpathy. Convolutional Neural Networks (CNNs / ConvNets), 2015.

[14] Robert Cornish, Hongseok Yang, and Franck Wood. Towards a Testable Notion of Generalization for Generative Adversarial Networks. 2018.

[15] Javier Portilla and Eero P. Simoncelli. A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients. *International Journal of Computer Vision*, 40(1):49–70, 2000.

[16] Lucas Theis, Aäron Van Der Oord, and Matthias Bethge. A note on the evaluation of generative models. 2016.

[17] Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Chen, Greg S. Corrado, Jeff Dean, and Andrew Y. Ng. Building High-level Features Using Large Scale Unsupervised Learning. 2012.

[18] C++ - Eigen Doc.

[19] Christopher Olah. Backpropagation, 2015.

[20] Marius-Constantin Popescu, Valentina E. Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

[21] Scott Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran, Bernt Schiele, and Honglak Lee. Generative adversarial text to image synthesis. *arXiv preprint arXiv:1605.05396*, 2016.

[22] Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. Generating videos with scene dynamics. In *Advances In Neural Information Processing Systems*, pages 613–621, 2016.

[23] Karol Gregor, Frederic Besse, Danilo Jimenez Rezende, Ivo Danihelka, and Daan Wierstra. Towards conceptual compression. In *Advances In Neural Information Processing Systems*, pages 3549–3557, 2016.

[24] Scott E. Reed, Zeynep Akata, Santosh Mohan, Samuel Tenka, Bernt Schiele, and Honglak Lee. Learning what and where to draw. In *Advances in Neural Information Processing Systems*, pages 217–225, 2016.

[25] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and others. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv preprint arXiv:1609.04802*, 2016.

[26] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016.

[27] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *arXiv preprint arXiv:1511.06434*, 2015.

[28] Ferenc Huszár. Thanksgiving Special: GANs are Being Fixed in More than One Way, 2017.

[29] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled Generative Adverarial Networks. 2017.

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

[31] Marc Parizeau. *Réseaux de Neurones*. 2004.

[32] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Internal Representation by Error Propagation. In David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[33] XuanLong Nguyen. Wasserstein distances for discrete measures and convergence in nonparametric mixture models. 2011.

[34] S. Albanie, S. Ehrhardt, and J. F. Henriques. Stopping GAN Violence: Generative Unadversarial Networks. *ArXiv e-prints*, 2017.