CentraleSupélec
Projet long du cursus Supélec
Encadré par : J. Tomasik et A. Rimmel

Dessine-moi un mouton
Generative Adversarial Network

François Bouvier d'Yvoire
Matthieu Delmas
Romain Poirot
Paul Witz

Étude des réseaux de neurones en perceptrons
avec application au concept des Generative Adversarial Network

Années 2017-2018

Dessine-moi un mouton
Generative Adversarial Network

Résumé

Ce document est le rapport d'un projet long effectué dans l'année 2017-2018 à Supélec. Il aborde les GANs (Generative Adversarial Networks) et différents axes de recherche pour en améliorer les résultats (WGAN, Mini-Batch, Algorithmes de descente de gradient à pas adaptatif...) et propose une implémentation d'une librairie Python pour les manipuler.

Table des matières

Chapitre 1

Introduction aux réseaux de neurones et premières applications

Le première partie de ce projet a pour but de comprendre le fonctionnement des réseaux de neurones et leurs applications à la classification. Une structure informatique en python pour les utiliser sera également mise en place.

1.1 Outils utilisés pour le projet

Ce projet possède une dimension de conception logicielle. Il s'agit de programmer des réseaux de neurones efficaces et performants adaptés aux problèmes que nous souhaitons résoudre sans utiliser de bibliothèques existantes.
Étant donné l'évolution prévue de notre code (du perceptron simple pour les problèmes de XOR ou du MNIST, puis la mise en place du GAN, et enfin toute sortes d'améliorations utiles), nous devons être particulièrement vigilants sur la souplesse de notre code. La programmation en équipe, sur une longue durée et avec de telles contraintes, nécessite la mise en place d'outils et certains choix techniques.

1.2 Réseaux de neurones et fonctionnement

Les réseaux de neurones font partie des piliers de l'intelligence artificielle. Leur fonctionnement est basé sur une interprétation sommaire du cerveau humain. Des neurones seuls reçoivent des signaux, les traitent et renvoient un signal de sortie. Les neurones sont alors agrégés en un réseau avec des entrées et des sorties globales. On modélise la plasticité du cerveaux par des paramètres variables qui changent au cours de l'apprentissage. Celui-ci se fait en comparant les sorties de notre réseau aux résultats attendus.

Les réseaux ainsi obtenus et entrainés sont donc des fonctions. L'apprentissage permet aux réseaux d'approcher les fonctions que nous souhaitons. L'objectif est d'approcher des fonctions qui ne sont pas facilement réalisables avec les moyens usuels. Par exemple, il est très difficile de définir la fonction indicatrice des chiffres manuscrits alors qu'un réseau neuronal est capable de le faire.

1.2.1 Le neurone

L'unité de base du réseau est le neurone, on peut l'imaginer comme une fonction mathématique F. Il possède n entrées (ou plutôt un vecteur X de dimension n), chacune affectée d'un poids w_i (on a donc un vecteur de poids w, et une fonction mathématique de $\mathbb{R}$ dans $\mathbb{R}$ non linéaire dite fonction d'activation σ. Le rôle du neurone sera de renvoyer le résultat de la fonction d'activation appliquée à la somme des entrées pondérées par leurs poids respectifs : on a $F(x) = \sigma(w^T x)$. On peut ajouter un biais b comme paramètre de notre neurone, ce qui permet de donner un aspect affine au calcul : $F(x) = \sigma(w^T x + b)$

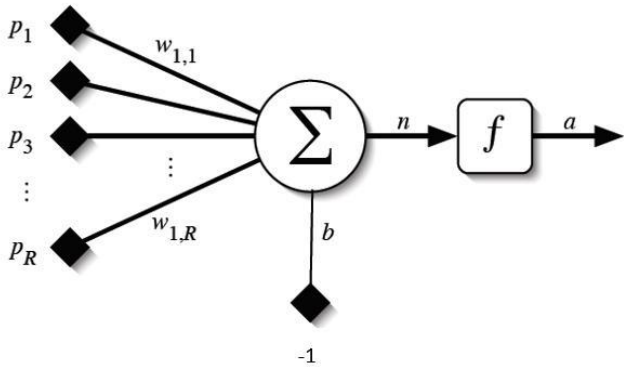


FIGURE 1.1 – Schéma d'un neurone

Un exemple simple du réseau de neurones est la séparation d'un plan en deux.
Imaginons un neurone à deux entrées e_1 et e_2, chacune attribuée d'un poids w_1 et w_2. On affecte au neurone un biais b et une fonction d'activation seuil (Heavyside par exemple).
Notre neurone renverra : 1 si $(e_1 w_1 + e_2 w_2) - b > 0$ et 0 sinon.

Si les e_1 et e_2 représentent les abscisses et ordonnées d'un point du plan, on reconnaît dans l'argument de la fonction d'activation l'équation d'une droite affine. Notre neurone pourra donc distinguer les points du plan selon le côté de la droite où ils se trouvent.

On peut déjà voir qu'une modification des poids entraînera une délimitation différente du plan. On peut donc imaginer faire « apprendre » au réseau une délimitation choisi en modifiant ses poids. Nous reviendrons sur ce concept par la suite.
Cependant, les applications d'un neurone seul sont vite limitées. C'est pourquoi on va s'intéresser à en connecter plusieurs entre eux.

1.2.2 Réseau de neurones et perceptron

On a déjà vu que le neurone se prêtait bien à une séparation binaire des données. On va voir que l'organisation de neurones en réseau permet de meilleures classifications.
L'organisation du réseau se fera au moyen de couches de neurones. Une couche est un ensemble de neurones possédant les mêmes entrées. Cette couche aura alors plus d'une sortie (une par neurone dans la couche), ce qui permet de généraliser aux sorties vectorielles le concept du neurone seul. En ayant ainsi une sortie vectorielle, on peut utiliser les sorties d'une couche en tant qu'entrée d'une autre couche, ce qui complexifie encore les fonctions possiblement décrites par les neurones. On peut imaginer des réseaux plus complexes où la sortie n'est pas réutilisée dans la couche suivante mais plusieurs couches plus loin ou dans des couches antérieures. On peut en réalité construire le graphe de neurones que l'on veut, mais, dans la pratique, seules certaines structures sont utilisés.

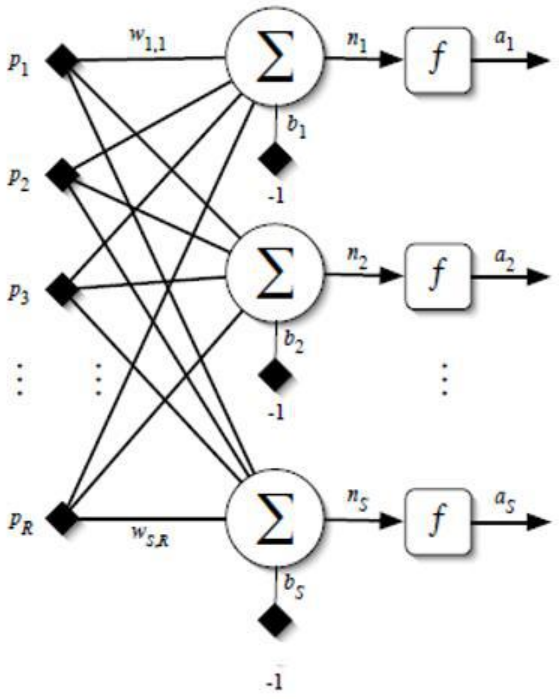


FIGURE 1.2 – Exemple d'une couche de neurones

Le perceptron est un modèle de réseau de neurones auquel on va s'intéresser particulièrement. Il s'agit d'un réseau linéaire où chaque couche est entièrement connectée à la suivante, c'est-à-dire que chaque neurone d'une couche prend en entrée toutes les sorties de la couche précédente. On ne trouve aucune boucle dans le graphe d'un perceptron, c'est donc une propagation vers l'avant. L'utilité d'avoir plusieurs couches se comprend facilement. Si on reprend notre exemple du problème de classification des points, on peut imaginer, par exemple, quatre neurones qui enverront leur sortie sur un neurone à quatre entrées. Chacun des neurones réalisera la séparation du plan en deux selon le principe déjà évoqué précédemment. Le neurone de la couche de sortie pourra réaliser facilement le rôle d'un ET logique. On vient de sélectionner un carré dans le plan. En étendant le raisonnement, on voit qu'un réseau à deux couches permet de sélectionner n'importe quelle zone convexe de l'espace des entrées (ici du plan). De même, un réseau à trois

couches pourra sélectionner n'importe quelle zone concave de l'espace des entrées.

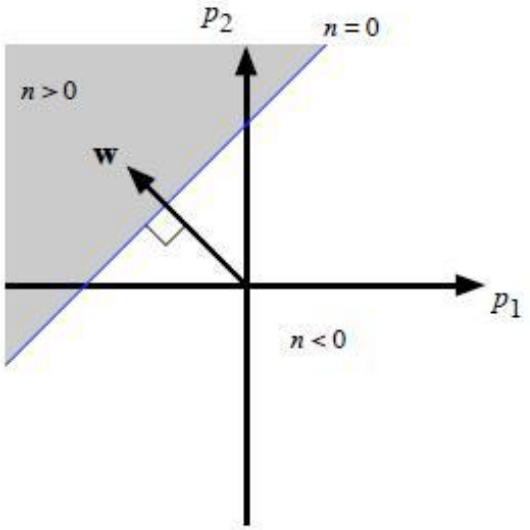


FIGURE 1.3 – Frontière de décision pour un perceptron simple à 1 neurone et 2 entrées

On appelle la dernière couche, celle qui donne le résultat du réseau de neurones, la couche de sortie ; et la première couche où l'on donne les entrées est appelée couche d'entrée. Les autres couches sont appelées des couches cachées. Cette appellation vient du fait qu'a priori, nous n'avons aucun moyen de voir ou de corriger les comportements des neurones cachés. En effet, avec la seule donnée de la sortie, l'influence des poids des couches cachées sur celle-ci n'est pas évidente. C'est ce que nous allons étudier dans la partie suivante.

1.2.3 Apprentissage par rétro-propagation

Il existe plusieurs types d'apprentissages du réseau. Les deux grandes catégories sont l'apprentissage supervisé et l'apprentissage non supervisé. Un apprentissage supervisé nécessite une base d'apprentissage à enseigner au réseau. Elle est composée d'associations entre entrées et sorties voulues. Le réseau déduira de cette base les autres cas qu'on ne lui aura pas appris. Une apprentissage non supervisé ne nécessite pas une telle base d'apprentissage.

Dans le cadre du perceptron, nous utilisons un apprentissage supervisé, la rétro-propagation des erreurs. Il s'agit en fait d'effectuer une descente de gradient. Nous avons notre réseau qui représente F une fonction, cette fonction dépend de nombreux paramètres qui sont les poids W de tous les neurones de toutes les couches (ainsi que les biais). Lorsque l'on veut faire tendre F vers une fonction G, et que l'on peut avoir une distance entre ces deux fonctions, on obtient alors un problème de minimisation, avec pour paramètres d'optimisation les poids W. L'algorithme de descente de gradient consiste à calculer pour chaque paramètre le gradient par rapport à la sortie (c'est une descente de Newton quand on est à une dimension) afin de déplacer ce paramètre dans la bonne direction pour minimiser la sortie.

Comme dit précédemment, on n'a pas facilement accès aux couches cachées, et il parait très coûteux d'opter pour des calculs de dérivées discrètes pour obtenir les gradients voulus pour tous

les paramètres. La rétro-propagation consiste à calculer l'influence de chaque paramètre sur la sortie et à les mettre à jour en fonction de cette influence. On a vu que les couches de neurones avaient leurs propres sorties, on peut donc calculer l'influence des poids d'une couche sur sa sortie propre, puis s'en servir pour calculer l'influence sur les entrées, qui sont la sortie d'une couche précédente, on procède comme suit :

Les paramètres que l'on fait évoluer sont les poids et les biais. La formule de mise à jour est la suivante :

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

avec η le pas de convergence, $\frac{\partial E}{\partial W}$ la matrice de terme général $\frac{\partial E}{\partial W_{i,j}}$.
Pour pouvoir mettre à jour les poids, il faut donc calculer les $\frac{\partial E}{\partial W_{i,j}}$.
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 :

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

En forme matricielle, ces équations donnent :

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

On obtient donc une formule de récurrence que l'on doit propager de la dernière couche à la première couche (d'où le terme rétro-propagation). On sait donc maintenant mathématiquement faire l'apprentissage d'un perceptron : il nous faut des données connues, une fonction d'erreur (appelée Loss Function dans la littérature), et appliquer cet algorithme.

1.3 Conception logicielle des réseaux de neurones

L'un des objectifs du projet est la conception d'une librairie permettant l'implémentation de réseaux de neurones. Notre démarche est la suivante, nous cherchons à mettre en place la structure la plus simple possible mais également la plus souple possible. Ainsi nous ne cherchons pas l'exhaustivité de notre librairie, mais nous pouvons facilement la compléter dès lors que nous avons besoin de fonctionnalité supplémentaire.

Notre code est structuré autour de 3 types de classes, les classes permettant la création et le fonctionnement d'un ou plusieurs réseaux de neurones (ce sont les classes qui font l'intelligence du programme, noté *brain*), les classes apportant des outils de compréhension et de travail sur les réseaux (affichage de résultats, chargement et sauvegarde de paramètres, etc) et les classes permettant de lancer une expérience (classes *main*, instanciant les objets et les expériences).

Le projet est découpé en 2 répertoire Github, le premier correspondant au code le plus simple, fonctionnel sur le problème du XOR, le second correspondant au développement suivant. Ces derniers développement correspondent à la généralisation à tout types de problèmes d'apprentissage de perception simple, puis à la mise en place du GAN et toutes les évolutions que nous avons mis en place.

Vous pouvez retrouver les codes sur https ://github.com/Supelec-GAN/Salamandre-XOR.git et sur https ://github.com/Supelec-GAN/Salamandre-Code.git .

1.3.1 Structure du code

Afin de pouvoir implémenter facilement les calculs matriciels obtenus plus tôt, nous définissons notre plus bas niveau d'intelligence par une classe NeuronLayer représentant une couche de neurone.

Une couche de neurones est définie par sa matrice de poids $weights$, son vecteur de biais $biais$ ainsi que sa fonction d'activation $activation\_function$, nécessairement commune à tous les neurones dans cette structure.

Présentation des méthodes

- *compute* : Propagation d'une entrée au sein de cette couche.
- *backprop* : Rétro-Propagation de l'influence de l'erreur de la sortie de la couche avec mise à jour des poids et calculs de l'influence de l'erreur sur la sortie.
- $derivate\_error$: Calcul intermédiaire dans la rétro-propagation.

— *init_derivate_error* : Calcul de la dérivée de l'erreur pour la couche de sortie

Pour faire nos réseaux de Neurones, nous utilisons une classe Network qui permet de relier les différents NetworkLayer. Ainsi elle possède *self_layer_list* qui est une liste de NetworkLayer. Les méthodes principales sont :

— *compute* : Propagation depuis l'entrée au sein de toutes les couches.

— *backprop* : Rétro-Propagation de l'influence de l'erreur de la sortie du réseau et itération de *backprop* sur chaque NetworkLayer.

— *reset* : Remet tout les NetworkLayer à l'état initial.

Afin de simplifier l'utilisation des fonctions au sein de nos réseaux, nous créons une classe Function. Elle comprend les fonctions d'activations, les fonctions d'erreur, mais également des fonctions d'apprentissage i.e. les sorties attendues pour une entrée lors de l'apprentissage (ex : XorTest renvoie le résultat du XOR, MnistTest renvoie le label pour une image labellisée donnée).

Finalement pour utiliser et interpréter des réseaux, nous avons une classe Interface, celle si comporte plusieurs méthodes importantes. *learning_manager* permet d'organiser l'apprentissage et la récupération de données. Elle contient également des méthodes pour afficher des résultats.

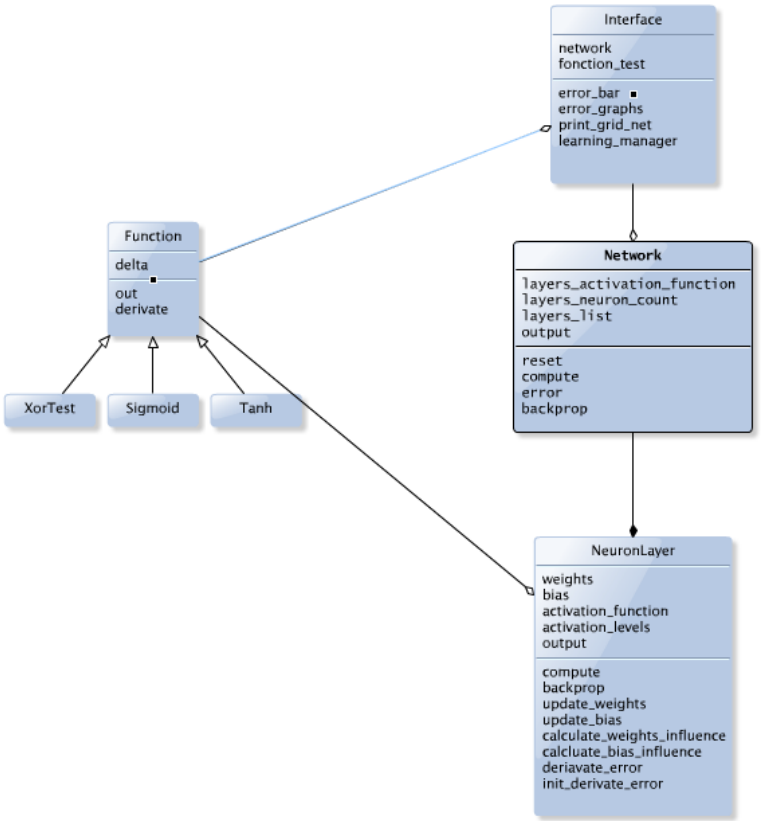


FIGURE 1.4 – Diagramme UML du code minimal et initial

Après évolution du code, on trouvera dans DataProcessor des méthodes pour effectuer des calculs

sur les données, tels que la moyenne sur plusieurs séries d'expérience et l'écart-type. La classe DataInterface sert à la sauvegarde des résultats et des paramètres ainsi qu'à leur chargement pour utilisation ultérieure.

Finalement les scripts run.py et tests.py servent à lancer des expériences en utilisant la librairie mise en place.

1.4 Application au problème du XOR

Lorsque l'on souhaite travailler sur des algorithmes d'apprentissage par ordinateur, il est recommandé de les essayer sur des problèmes connus afin d'en vérifier les performances.
Le problème du XOR est l'un des plus classiques car il apporte de nombreuses difficultés.

L'objectif du XOR est de séparer le plan complexe en quatre cadrants, $(x > 0, y > 0)$, $(x > 0, y < 0)$, $(x < 0, y > 0)$ et $(x > 0, y < 0)$. Pour l'expérimentation, on restreint le plan à $[-1; 1]^2$. Les sorties attendues par le réseau de neurones sont alors 1 pour les points tel que $x * y > 0$ et -1 pour les points tels que $x * y < 0$.
Le premier intérêt de ce problème est qu'il est non linéaire. Cela se traduit par le fait qu'une droite séparant le plan en 2 ne répond pas du tout au problème.

C'est en se basant sur la résolution du XOR que nous avons construit notre structure de réseau et vérifié la cohérence de notre code. La littérature propose comme réseau le plus simple pour ce problème une couche cachée de 2 neurones, avec 2 entrées (x et y) et 1 sortie dans $[-1, 1]$. Nous avons étudié également quelques autres formes de réseaux pour comparer les résultats.

Notion de résultats La notion de résultats nécessite d'être correctement définie afin de pouvoir être interprétée correctement, en particulier pour la comparaison à d'autres résultats obtenus par nous-même ou par d'autres personnes.
La structure de perceptron sous cette forme classe les objets que l'on donne en entrée. Généralement, le résultat est défini par rapport à un pourcentage de succès dans cette classification. Pour l'obtenir, on commence par définir une erreur relative, c'est-à-dire une distance entre la sortie cible et la sortie obtenue. Un seuil est alors appliqué afin de définir une sortie booléenne de la classification de l'entrée.
Dans le cas du XOR, on met en place un seuil de 0.5, c'est à dire que, si l'erreur est inférieure à 50%, le réseau a raison. Cela peut s'interpréter comme suit : le réseau donne un résultat qui indique sa confiance dans la sortie. 1 ou 0 si il est certain que la sortie doit être 1 ou 0, 0.5 si il ne peut départager l'un ou l'autre, le seuil consiste à dire que sa réponse est celle en qui il a le plus confiance. On cherche également à évaluer la vitesse d'apprentissage. Ainsi, on calcule le pourcentage de succès du réseau à intervalles réguliers au cours de l'apprentissage. Les réseaux étant soumis à une forte composante aléatoire (l'ordre d'apprentissage, ainsi que l'initialisation des poids), on effectue des apprentissages dans les mêmes conditions plusieurs fois afin d'obtenir des courbes moyennes, et des intervalles de confiances justifiant nos résultats.

Réseau en $2 \to 2 \to 1$ Les résultats obtenus au début sur ce réseau extrêmement simple semblaient tout à fait aléatoires et nous ont permis de détecter des erreurs de traduction des

équations de rétro-propagation en code Python. Nous avons finalement pu obtenir des résultats satisfaisants, comme le montre la figure 1.5.

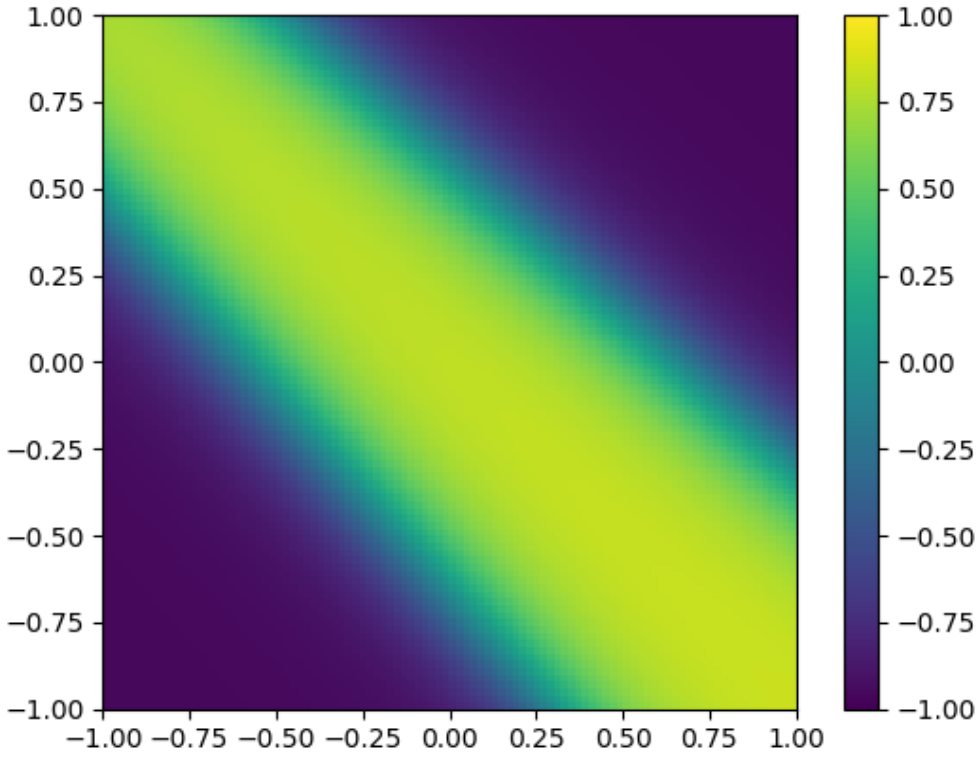


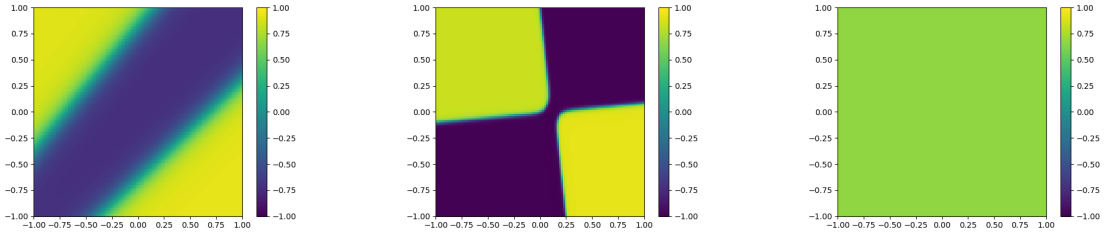
FIGURE 1.5 – Application d'un réseau 2-2-1 au plan pour xor

Importance du pas d'apprentissage η L'un des paramètres de la descente de gradient est le pas d'apprentissage η. Il intervient dans le formule de mise à jour des poids :

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

Plus il est petit, plus l'algorithme avancera lentement. Cependant, s'il est trop grand, l'algorithme ne se rapprochera pas du minimum mais divergera. Il faut donc le choisir suffisamment petit pour que cela converge mais suffisamment grand pour que ce ne soit pas trop lent. La figure 1.6 montre en 10000 apprentissages le résultat d'un réseau entraîné sur XOR selon le pas d'apprentissage. On remarque notamment que, lors qu'il est trop élevé, le réseau n'arrive pas à découper l'espace.

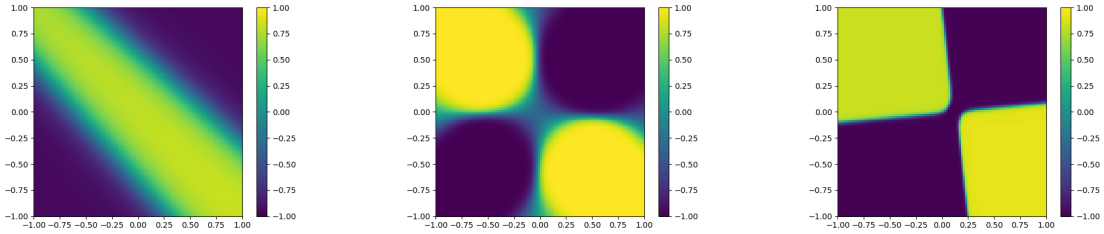
Structure du réseau La structure du réseau choisie est également importante. Comme expliqué plus haut, si le réseau est 2-2-1, il ne peut pas délimiter l'espace comme on le souhaite. Cela signifie que, pour la fonction XOR avec laquelle on travaille (à savoir qu'on représente VRAI par 1 et FAUX par -1), il ne peut pas découper le plan selon les axes des abscisses et ordonnées. En revanche, il va essayer de le délimiter de façon à accorder aux points (1,1) et (-1, -1) la valeur



(a) eta =0.1 (b) eta =0.5 (c) eta =0.9

FIGURE 1.6 – Xor en 2-2-2-1 avec différents pas d'apprentissages au bout de 10000 apprentissages

VRAIE et (1,-1) et (-1,1) la valeur FAUX. Pour cela, il trace une bande comme sur la figure 1.7a. En effet, il y a priorité sur ces valeurs puisque la base d'apprentissage de notre réseau est constituée de couples de -1 et de 1.



(a) Structure 2-2-1 (b) Structure 2-4-1 (c) Structure 2-2-2-1

FIGURE 1.7 – Xor avec différentes structures

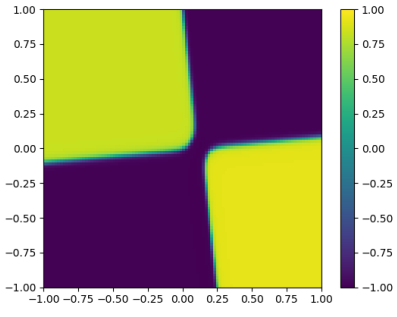
Cependant, lorsqu'on lui accorde plus de neurones, 4 neurones sur la première couche comme sur la figure 1.7b, ou 3 couches par exemple comme sur la figure 1.7c, le réseau découpe l'espace comme souhaité. En revanche, le XOR avec la structure 2-2-2-1 est assez instable : pour les mêmes paramètres, on peut obtenir un réseau qui converge et un autre qui se plante complètement comme le montre la figure 1.8

Conclusion sur le XOR Pas d'apprentissage très petit par rapport à la littérature
Influence des fonctions d'activation

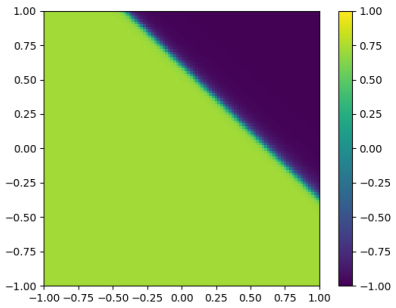
1.5 Application à la base de données MNIST

Description du problème

Pour le problème du MNIST qui consiste à apprendre à reconnaitre des chiffres manuscrits, les données étaient les suivantes :



(a) XOR qui converge



(b) XOR planté

FIGURE 1.8 – XOR en 2-2-2-1 avec eta =0.5

— 60000 images pour l'apprentissage, avec leurs étiquettes
— 10000 images de test

Toutes les images ont une dimension de 28*28 pixels en noir et blanc. Ces sets d'images sont récupérables sur le site http ://yann.lecun.com/exdb/mnist/ sous le format IDX. L'extraction de ce format vers une liste python est faite grâce au module python-mnist.

Paramètres généraux utilisés :

Les fonctions d'activation utilisées pour toutes les expériences ici sont des sigmoïdes de paramètre μ : $\sigma_\mu(x) = \frac{1}{1+e^{-\mu x}}$
On pourra étudier l'influence de μ sur la vitesse de convergence. Puisque les fonctions d'activation utilisées sont des sigmoïdes dont la sortie est dans [0, 1], les valeurs d'entrées situées entre 0 et 255 sont normalisées entre 0 et 1.
L'erreur utilisée sur la couche de sortie est l'erreur quadratique.
Les poids sont initialisés avec une répartition gaussienne centrée réduite. Les biais sont initialisés à 0.
De bons résultats ont été obtenus avec le réseau suivant, conformément à la littérature :

— Eta 0.2
— Sigmoïde 0.1
— Réseau à une couche cachée de 300 neurones et 10 neurones de sortie (784-300-10)
— Apprentissage stochastique

Avec ce type de réseau, on obtient rapidement des taux de succès proche de 95% après 10 passes de l'ensemble du set d'apprentissages, et un écart-type final de 0.001223411. Nous allons maintenant voir l'influence des différents paramètres.

Variation d'êta :

Sur le réseau 300-10 précédent, une augmentation du êta de 0.2 à 10 ne semble qu'améliorer la vitesse de convergence comme le montre la figure 1.10. L'écart-type n'augmente pas et on obtient

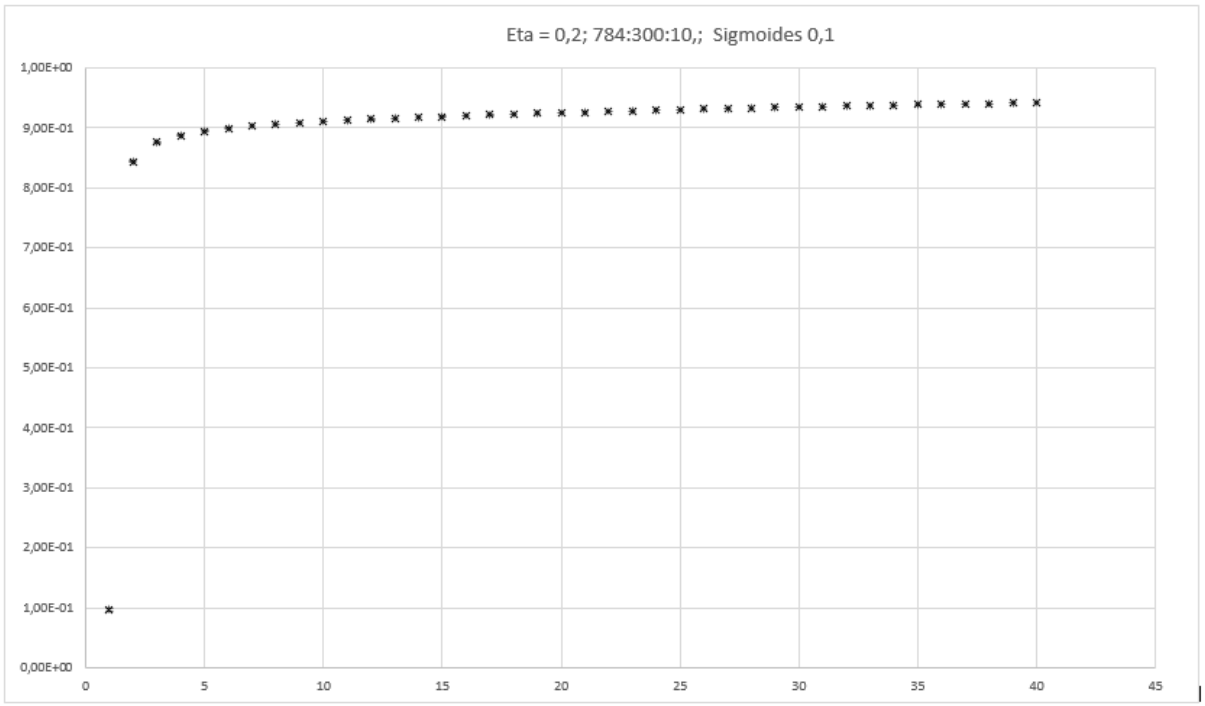


FIGURE 1.9 – Courbe de réussite du réseau sur MNIST

une meilleure précision à la fin.

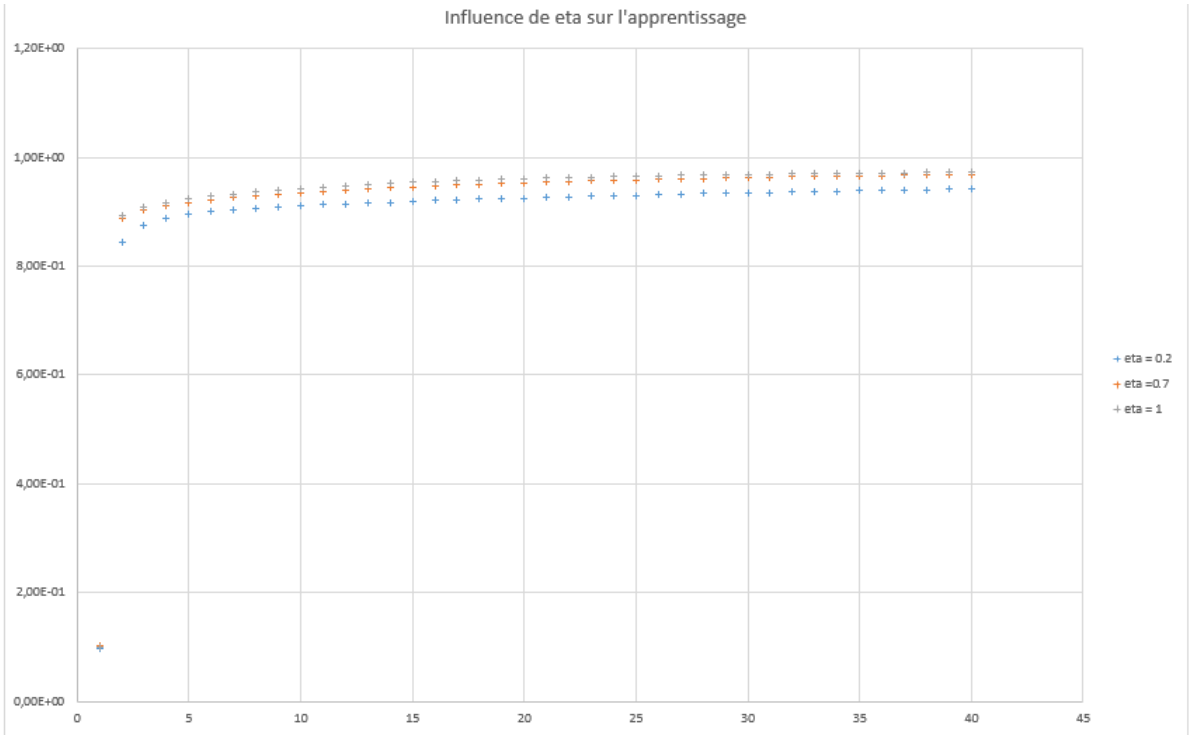
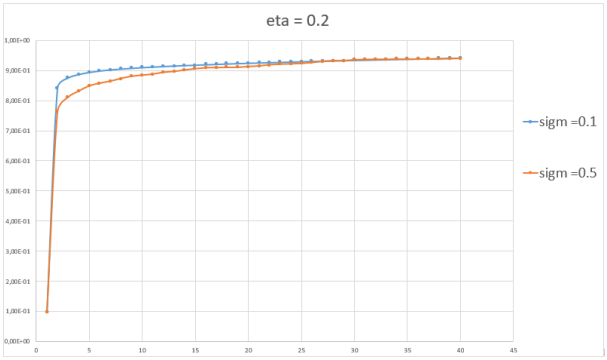


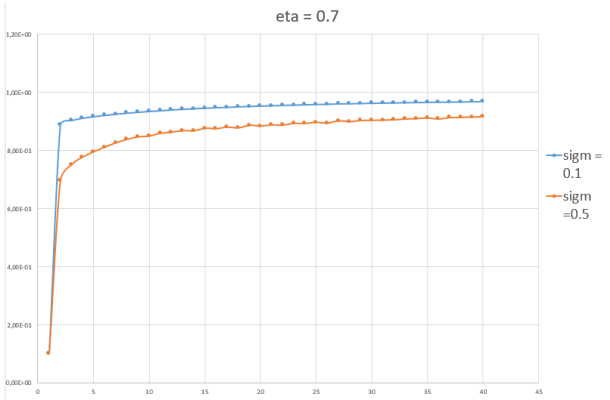
FIGURE 1.10 – Courbe de réussite du réseau sur MNIST avec différents η

Choix du paramètre de la sigmoïde :

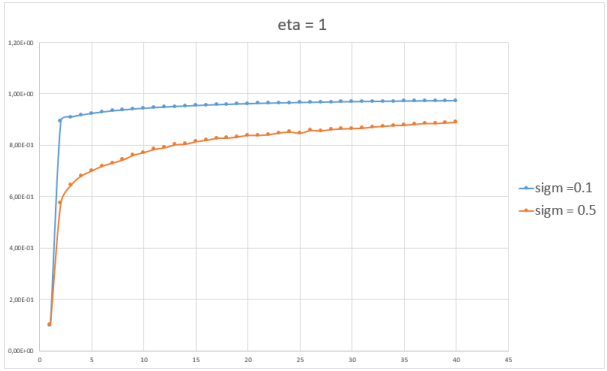
Ici, l'influence du choix de la sigmoïde est observée. Les tests ont été effectués avec $\mu = 0.1$ et $\mu = 0.5$ comme paramètre de sigmoïdes. [[insert courbe (cf 17/01/18)]] On remarque qu'en tout point, choisir 0.1 en paramètre à la place de 0.5 est mieux : vitesse de convergence, précision finale, écart-type. Ce résultat empire avec un êta plus élevé, au point de ne plus réussir à apprendre. On restera donc sur une sigmoïde de paramètre 0.1 pour la suite des expériences.



(a) $\eta = 0.2$



(b) $\eta = 0.7$



(c) $\eta = 1$

FIGURE 1.11 – Comparaison de deux sigmoïdes pour plusieurs η

Différents réseaux :

Intuitivement, un réseau avec plus de couches cachées devrait obtenir une meilleure précision, mais devrait avoir un temps d'apprentissage plus long. Ces résultats se confirment avec des expériences sur le réseau à une couche cachée (300 neurones) précédent, et un réseau sans couche cachée. Ce dernier converge très vite aussi bien vis-à-vis du nombre d'apprentissages nécessaire que du temps de calcul. Cependant, il est difficile de dépasser les 90% de succès. Alors que sur le réseau avec couche cachée, on arrive à obtenir moins de 5% d'erreur. En revanche, les temps de calculs sont plus élevés. Le réseau avec deux couches cachées, 1000 puis 300, a aussi été testé. Les résultats ici sont satisfaisants, cependant l'amélioration des résultats n'est pas très importante, alors que les temps de calculs augmentent fortement. [[insert courbe (cf 17/01/18)]] Ce problème est bien connu dans l'étude des réseaux de neurones, et a justifié l'apparition de l'apprentissage profond, que nous évoquerons plus tard. Temps de calcul approximatifs pour une passe du set d'apprentissage, et un test tous les 1000 apprentissages :

— 5-6 minutes pour le 784-1000-300-10
— 4 minutes pour le 784-300-10

Ces temps peuvent être améliorés avec l'introduction du batch learning qui permet de calculer les résultats des tests plus rapidement.

Chapitre 2

Generative Adversarial Networks

La première partie de cette étude nous a permis de maîtriser l'utilisation de réseaux en perceptron et de structurer une architecture logicielle efficace et souple pour l'étude des GAN.
Après avoir obtenu des résultats satisfaisants dans la classification de motifs sur la base MNIST, nous étudions la génération de données à l'aide de réseaux de neurones en nous basant sur le concept de GAN, introduit par I. Goodfellow en 2016 [4].
Notre objectif dans cette partie est d'appréhender le concept de GAN et de l'appliquer sur notre programme afin d'étudier les différents paramètres.

2.1 Principe

Le GAN s'inscrit dans les problèmes de générations de données par ordinateur. Ses modèles cherchent à produire de données nouvelles respectant un certain nombre de contraintes. Les applications possibles sont très nombreuses tant au niveau scientifique qu'industriel, avec, par exemple, la modélisation de nouvelles protéines, le dessin de circuits intégrés, etc. Le but de nos GAN sera de générer des images que l'on ne pourra distinguer de « vraies » images, prises avec un appareil photo.

Le principe général est le suivant :
un GAN est constitué de deux réseaux de neurones, le Générateur (G) et le Discriminateur (D). Le Générateur a pour but de créer les images et le Discriminateur de déterminer si les images qu'on lui donne sont de « vraies » images ou ont été créées par le Générateur. Ces deux réseaux sont mis en compétition : le Générateur a pour but de tromper le Discriminateur tandis que le Discriminateur doit détecter les « fausses » images.
L'apprentissage du Discriminateur se fait à la fois sur des images générées par le Générateur et de « vraies » images, issues d'une banque d'images afin de continuellement améliorer sa capacité de discernement.
L'apprentissage du Générateur dépend de la réponse du Discriminateur : lorsqu'il génère une image, on la donne au Discriminateur pour voir si le Générateur a réussi à le tromper. Le

discriminateur sert donc de fonction d'erreur au Générateur.

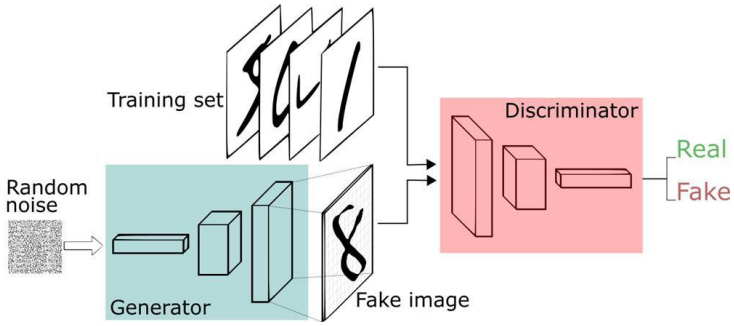


FIGURE 2.1 – Fonctionnement du GAN

De façon plus formelle, on travaille avec 3 distributions : p_x, la distribution idéale des vraies images, p_{data}, la distribution de l'échantillon des vraies images et p_{model} la distribution réalisée par les images issues du Générateur. Le but de l'apprentissage est de rapprocher p_{model} de p_x. Comme p_x nous est inconnu, on va plutôt s'approcher de p_{data}.
On dispose de deux fonctions de coûts $J_D(\theta_G, \theta_D)$ et $J_G(\theta_G, \theta_D)$, représentant respectivement les fonctions de coûts du Discriminateur et du Générateur. On note θ_G et θ_D les paramètres des réseaux. Les fonctions de coûts dépendent bien des paramètres des deux réseaux car le Discriminateur apprend à discerner les vraies images des fausses, dépendantes du générateur, et le générateur apprend via le résultat du Discriminateur.
C'est un problème d'optimisation simultanée. Il peut également être décrit comme un problème de jeux à informations complètes. G a accès aux données de D, mais ne peut influer que sur θ_G et D a accès aux données de G, mais ne peut influer que sur θ_D. Cette vision permet de déduire un algorithme où chaque joueur va faire un mouvement de manière optimal, afin de tendre vers un équilibre de Nash.

2.2 Apprentissage

L'apprentissage consiste à appliquer cette méthode de jeu à l'apprentissage des réseaux de neurones. Nous fournissons au Discriminateur des images x (de la BDD MNIST par exemple) et lui demandons de nous renvoyer un réel entre 0 et 1, qui représente son degré de confiance sur le fait que l'image fournie ait été tirée d'une banque de données authentiques ou du Générateur. Les réponses attendues sont respectivement $(D(x) = 1)$ et $(D(x) = 0)$ ce qui nous permet de calculer des erreurs pour la descente de gradient du Discriminateur.
Le Générateur, quant à lui, génère une image à partir d'un vecteur de bruit z. Cette image est ensuite jugée par le Discriminateur : $D(G(z)) = 1$ si le Générateur a dupé le Discriminateur et 0 sinon. L'objectif du Générateur est d'être le plus proche possible de la première situation, l'erreur pour la descente du gradient du Générateur en est déduite.
L'apprentissage complet se fait en alternant les 2 phases successivement, chaque réseau jouant tour à tour. On parle de réseau concurrent car le Discriminateur cherche à obtenir $D(G(z)) = 0$ pour tout z et le Générateur $D(G(z)) = 1$.

Problèmes liés à la convergence L'apprentissage des GANs n'est pas simple à maîtriser car ils ne fonctionnent pas comme un seul réseau qui apprend avec une algorithme de descente de gradients. Il s'agit en réalité d'une descente de gradient simultanée (J_G et J_D) qui n'est pas un cas particulier, mais une généralisation du problème classique d'optimisation. La résolution mathématique de ce problème n'est pas triviale, et les méthodes ne s'adaptent pas facilement aux réseaux de neurones. La méthode proposée ci-dessous est donc une heuristique que l'on peut fortement adapter. Elle a fait ses preuves dans de nombreux cas, malgré son manque d'appui mathématique. Cependant d'autres façons de voir les choses permettent d'obtenir des résultats toujours corrects mais avec une plus grande rigueur. Les questionnements mathématiques seront abordés dans les axes de recherches.

2.3 Paramètres des GANS

Voici une description des paramètres principaux sur lesquels on peut jouer pour l'implémentation d'un GAN. Ils sont nombreux car la description précédente est en réalité peu restrictive.

- Fonctions de coûts
- Ratios d'apprentissage
- Paramètres classiques des réseaux de neurones (Structures des réseaux, pas d'apprentissage, etc.)

2.4 Structure et utilisation du code

Dans cette section nous allons aborder la structure de notre code pour l'implémentation des GANs ainsi que les différentes améliorations successives.

L'UML 2.2 précédent décrit le code présent dans le répertoire GitHub Salamandre-Code cité précédemment. On y retrouve les éléments de l'UML utilisé pour le XOR, à ceci près que nous avons fait évoluer l'Interface pour devenir la classe Engine, afin de bien séparer les actions d'apprentissages et de traitements de données.

Pour l'implémentation des GANs, il nous suffit de créer l'interaction entre 2 réseaux. Nous avons donc mis en place une classe GanGame qui instancie une partie entre 2 réseaux, l'un jouant le rôle du discriminateur, l'autre du générateur.

Les méthodes principales sont les suivantes :

- $Play\_And\_Learn$: Orchestre les joueurs et leurs apprentissages en fonction des ratios.
- $discriminator\_learning\_real$: apprentissage du discriminateur sur des vrais images.
- $discriminator\_learning\_virt$: apprentissage du discriminateur sur des fausses images.
- $generator\_learning$: apprentissage du générateur.

Au fur et à mesure des améliorations de notre code, nous avons obtenus la structure de code suivante 2.3 (le fichier pdf non compressé permet sa lisibilité). On notera l'ajout de différentes couches de réseaux de neurones, permettant les réseaux à convolutions et les WGAN, la diffé-

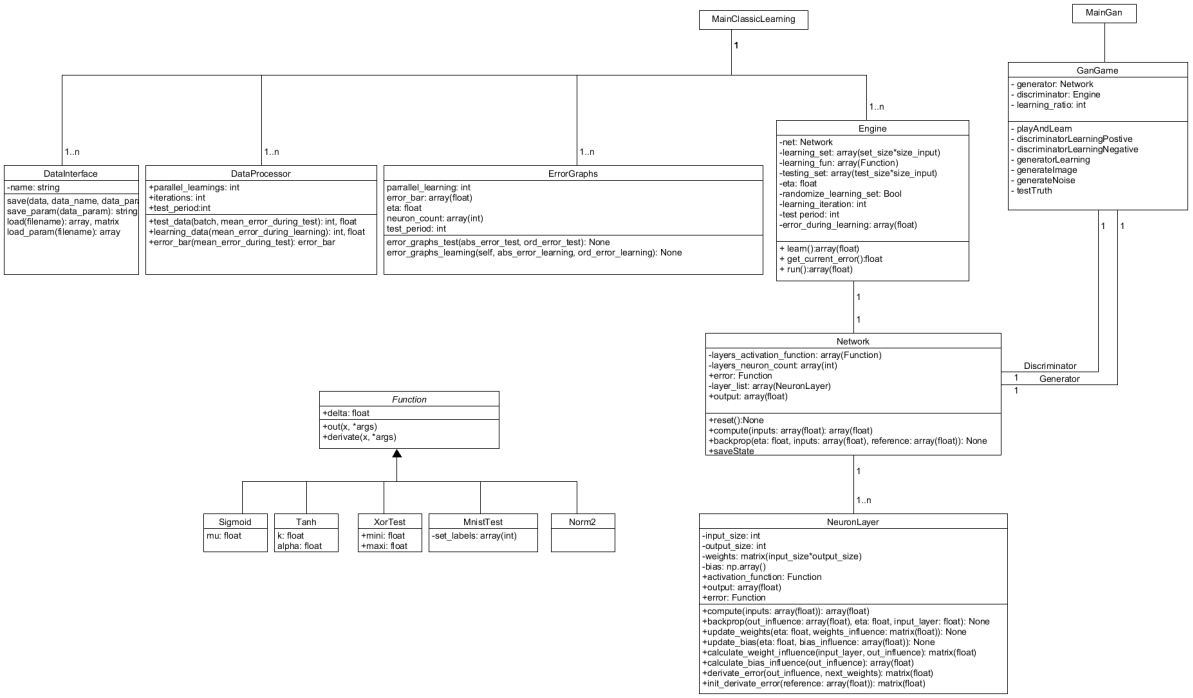


FIGURE 2.2 – Diagramme UML du code minimal et initial

rentiation des Function en fonction de leur utilité et l'ajout de quelques classes pour le dessin de données.

2.5 Premiers résultats pour des GANs simples

Afin d'appréhender correctement le fonctionnement des GANs, il est nécessaire de tester de nombreux paramètres. Les résultats, fructueux ou non, permettent d'évaluer l'intérêt des paramètres et de comprendre le sens "physique" de leur influence.
Ces essais se feront sur la base MNIST, c'est-à-dire que notre objectif sera d'obtenir des chiffres manuscrits dessinés par le générateur. Plus précisément, on attend du générateur après apprentissage que chaque entrée génère un chiffre entre 0 et 9 qu'un être humain ne peut distinguer d'un chiffre manuscrit écrit par un humain (donc de la base MNIST).

2.5.1 Méthodologie initiale

L'un des objectifs de ce projet est de décortiquer au mieux la méthode GAN. Pour cela, nous utilisons les réseaux de neurones les plus simples, ils seront complexifiés par la suite. La plupart des articles sur le sujet présentent des réseaux utilisant des structures avancées (couche convolutive, optimiseur de descente, etc.). Il n'est donc pas possible de se référer aux paramètres de ces articles pour obtenir immédiatement des résultats et s'assurer que notre programme tourne correctement.

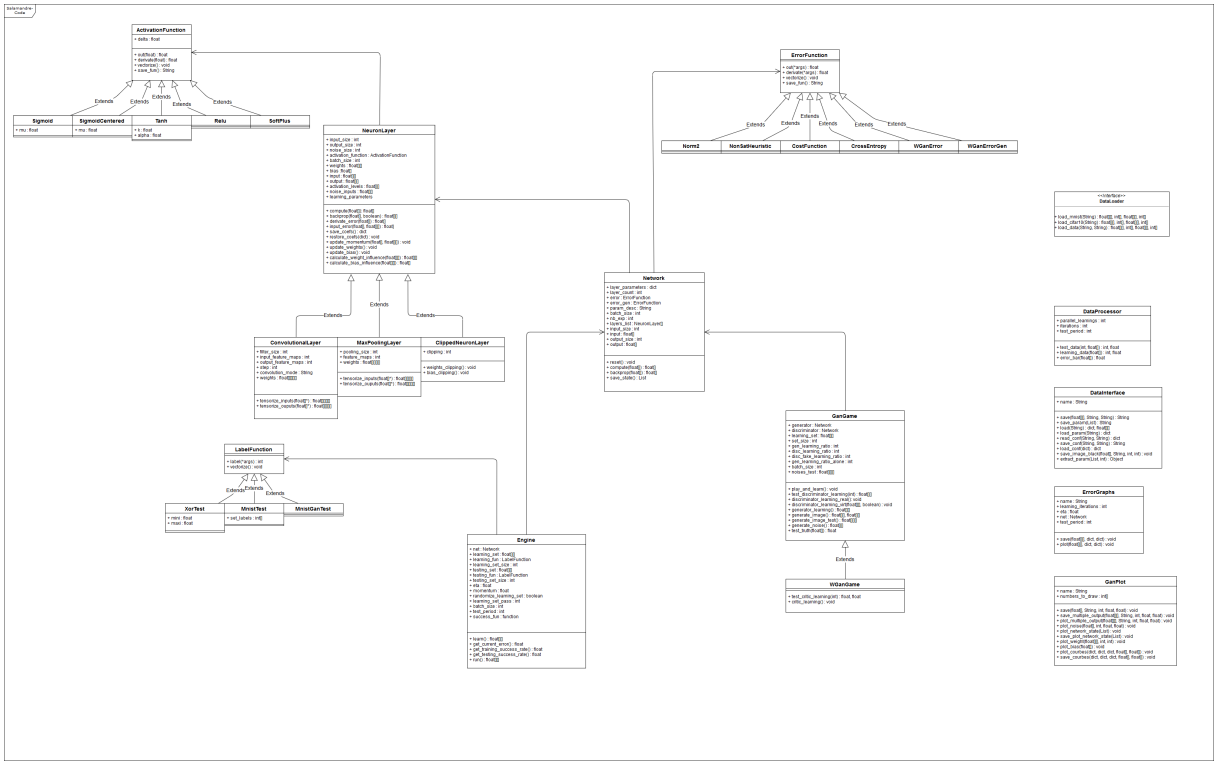


FIGURE 2.3 – Diagramme UML du code minimal et initial

Pour le choix du réseau Discriminateur, le choix se porte sur des structures ayant eu de bonnes performances pour le MNIST, c'est-à-dire soit un très bon taux final, soit une vitesse de convergence élevée. En effet, le Discriminateur n'est qu'un simple classificateur sur la base MNIST. Le choix du générateur est plus complexe car il faut que la structure soit assez puissante pour dessiner des chiffres. Cela semble bien plus difficile que simplement les reconnaitre, car ils peuvent être reconnus à l'aide de features très particuliers, tandis que la génération exige une information complète. La première idée pour dimensionner le réseau est d'avoir une entrée suffisante pour générer la diversité de sortie souhaitée, mais pas nécessairement plus pour ne pas avoir un réseau trop lourd en calcul.
Pour les autres paramètres, nous n'avons a priori pas d'idée précise de leurs ordres de grandeur nécessaires, en particulier pour le pas d'apprentissage (très différents pour MNIST et XOR par exemple).
Nous avons donc balayé les paramètres possibles afin d'obtenir le maximum d'information avec des GANs simples.
Le principal souci que nous avons eu avec cette approche fut l'évaluation des performances de nos GANs, au début de l'étude. En effet, nous n'arrivions pas encore à former des chiffres satisfaisants et, par conséquent, n'avions aucun moyen pour savoir si un paramètre était meilleur qu'un autre.
Nous avons considéré l'étude de l'évolution des scores que le Discriminateur attribuait aux images de Mnist et à celles du Générateur, mais elles se sont révélées inexploitables : le score brut ne permettait pas de rendre compte de l'apprentissage effectué jusqu'alors. Et le score ne révélait en rien la qualité de l'image. ici des exemples de courbes sur lesquelles on travaillait au début

2.6 Mode Collapse

2.6.1 Présentation du problème

Le problème du Mode Collapse est bien connu dans la confections de GANs. Il se traduit par un générateur qui ne produit plus qu'un type de données particulier, sans diversité. Imaginons un générateur qui doit générer deux types de données : X et Y. Après quelques apprentissages, il réussit à générer des X suffisamment convaincants pour tromper le Discriminateur. A ce moment, le Générateur est "récompensé" pour tromper Le Discriminateur, et ce dernier n'arrive plus à distinguer les X de la BBD de ceux générés artificiellement. Du coup le Discriminateur ne fait plus confiance aux X qu'il reçoit et se concentre sur les Y, sur lesquels il se fait davantage récompenser pour son travail. Après quelques apprentissages à nouveau, le générateur oublie les X et se met à générer des Y trompeurs. Ainsi, notre GAN fournit alternativement des X et des Y médiocres sur une longue période, alors qu'au contraire on souhaiterai obtenir des données diverses. Si ce problème est dû au fait que les données à générer sont différentes, on voit que paradoxalement, plus il y a de diversité dans les données à générer , moins le Collapse est important.

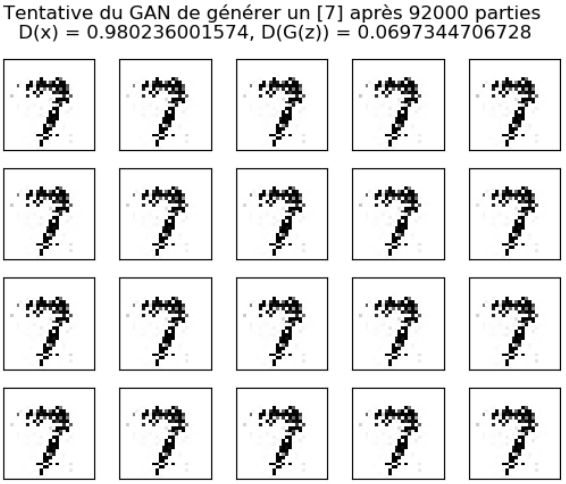


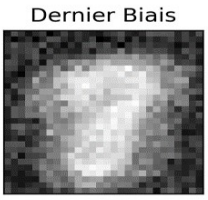
FIGURE 2.4 – Exemple de Collapse

2.6.2 Une première idée de résolution du problème

Une première étape pour éviter le problème est décrite dans l'article de NIPS 2014 de I.Goodfellow [4]. Il s'agit d'ajouter du bruit sur une couche intermédiaire du Générateur. En forçant ainsi une part d'aléatoire dans le Générateur, on peut espérer qu'il ne se "verrouille" pas et ne produise qu'une donnée particulière. Nous avons implémenté cette solution lors de notre travail sur MNIST. Goodfellow suggère plusieurs techniques d'implémentations : ajouter un bruit à la sortie d'une couche, multiplier le vecteur de sortie terme à terme avec un vecteur de bruit, ou encore concaténer le vecteur de sortie avec un vecteur de bruit gaussien. Nous avons choisi d'implémenter cette dernière solution. On pourrait toujours craindre que, dans le pire des cas, le Générateur

décide d'attribuer au bruit des poids nuls ou négligeables. Mais lors de l'implémentation, on a néanmoins remarqué une baisse significative du taux de Mode Collapse dans nos expériences.

En figure 2.5, on voit les résultats d'un GAN avec du bruit et un biais non nul en dernière couche. Les chiffres sont peu satisfaisants mais le réseau ne tombe pas en mode Collapse. En revanche, le dernier biais est très influent : il dessine presque à lui tout seul les 7 du réseau.



(a) Résultats d'un GAN en mettant du bruit et du biais en dernière couche

(b) Biais en dernière couche

FIGURE 2.5 – Résultats d'un GAN apprenant sur des 7 et des 8 uniquement

En retirant le biais, on obtient des résultats comme à la figure 2.6. Le réseau arrive à tracer des 8 différents et même des 7 que l'on reconnaît clairement.

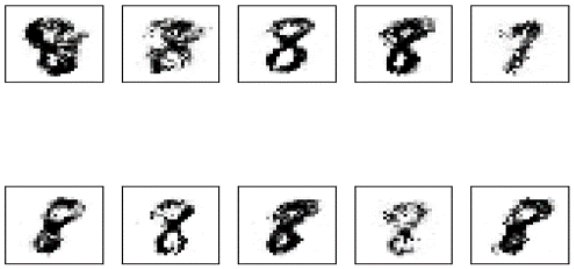


FIGURE 2.6 – Résultats d'un GAN en mettant du bruit mais pas de biais en dernière couche

Enfin, pour un GAN sans biais ni bruit sur la dernière couche, on obtient des résultats semblables à la figure 2.7. Sans bruit, ni biais, on retombe dans le monde collapse. Comme première solution, rajouter du bruit sur la dernière couche est efficace !

2.6.3 Perturber le GAN avec des "secousses"

Une approche pour empêcher le GAN de tomber en Mode Collapse fut suggérée par T. Salimans. Elle consiste à ajouter une part d'aléatoire dans le processus.

(a) Le GAN apprend sur des 7

(b) Le GAN apprend sur des 8

FIGURE 2.7 – Résultats d'un GAN sans bruit ni biais sur la dernière couche

Plutôt que de s'attendre à ce que le Discriminateur renvoie 1 sur une image de la base de données, on calcule l'erreur à partir d'un réel proche de 1 (entre 0.7 et 0.99) afin de plus facilement le récompenser.
En parallèle, de temps en temps (un cas sur deux cent, grand maximum) on trompe le Discriminateur en lui disant qu'on attendait une vraie image lorsqu'il en traite une fausse et inversement.
Ces deux techniques permettraient, en théorie, de renforcer le générateur de manière subtile (pour la première) et de le perturber suffisamment pour sortir des situations de Collapse (deuxième technique).
Cependant, les expériences menées par nos soins sur ces techniques ne se sont pas montrées concluantes, le Collapse apparaissait parfois plus vite que sans ces techniques .
Cela est sans doute dû au fait que nos GANs, contrairement à ceux de Goodfellow et Salimans, n'étaient pas assez robustes pour qu'on puisse se permettre d'ajouter cette part d'aléatoire.

2.6.4 Une autre piste : le minibatch

Une autre méthode destinée à empêcher le Mode Collapse et à forcer la diversité des données générées fut proposée par T.Salilmans et I.Goodfellow dans Improved techniques for training GANs [9].
L'idée consiste en proposant au Discriminateur non plus une seule donnée du Générateur mais bien tout un petit paquet (minibatch). Une distance mathématique entre chacune des données de ce minibatch est calculée et traitée (passée à l'exponentielle négative) puis ajoutée à la sortie d'une couche intermédiaire du Discriminateur.
Ainsi, on envoie n images dans notre Discriminateur, qui possèdera sur une couche intermédiaire n neurones supplémentaires, l'activation du neurone i correspond à un score de ressemblance entre la donnée traitée actuellement et la donnée i. Le Discriminateur a donc un moyen de déterminer si les données générées par le Générateur sont toutes semblables entre elles ou non. Il pourra ainsi facilement punir le Discriminateur si celui tombait en Mode Collapse .

2.7 Résultat sans collapse

En figure 2.8, des résultats obtenus sans Collapse, la solution choisie fut celle qui consistait à mettre du bruit dans les couches intermédiaires. De plus, comme pour prouver que plus de diversité signifiait moins de Collapse, on s'est aperçus que le réseau avait plus de mal à produire des chiffres différents lorsqu'on lui demandait d'en produire 3 que 10. Les chiffres produits sont cependant de meilleure qualité. En figure 2.9, se trouvent les résultats d'un GAN ayant appris sur les 10 chiffres.

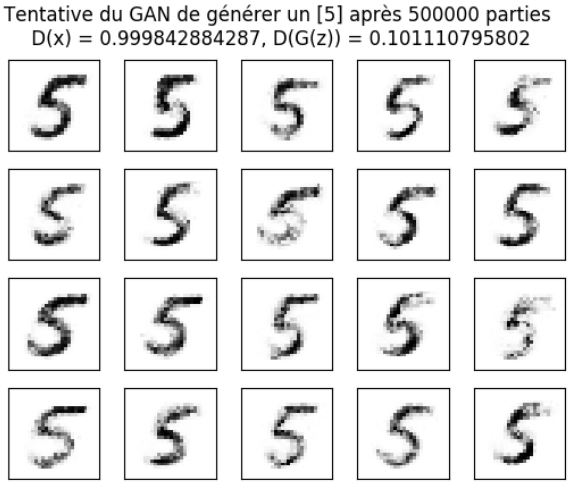


FIGURE 2.8 – Réseau ayant appris sur des 5

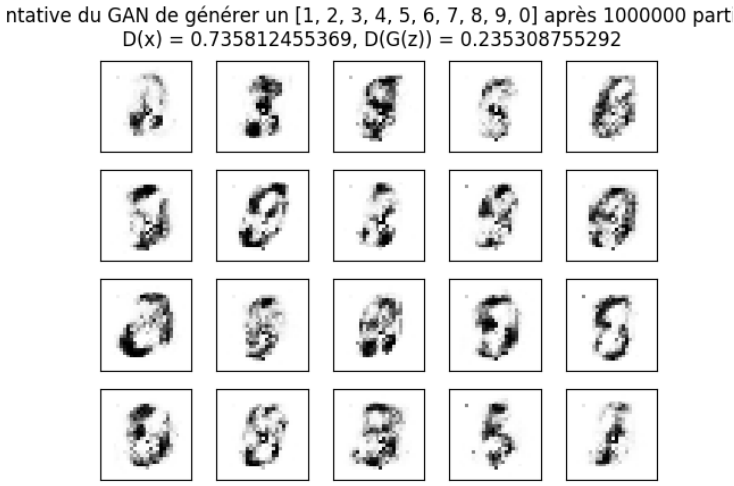


FIGURE 2.9 – Réseau ayant appris sur tous les chiffres

2.8 Autres problèmes rencontrés

Comme mentionné plus haut, il nous a été très difficile de savoir dans quel direction progresser. En effet, il n'existe pour l'instant aucune méthode fiable pour attribuer un "score de vraisemblance" aux donnés générées par GAN. En l'état, la meilleure méthode suggérée par l'état de l'art consiste à demander à des humains si les chiffres qu'on leur présentait leur paraissaient vraisemblables ou non. L'expérience fut tentée par Goodfellow avec l'aide du captcha d'Amazon, mais même de tels moyens n'ont pas donné de résultats très satisfaisants (il se trouve que le sujet humain, à l'instar d'un Discriminateur, apprenait de ses erreurs et devenait trop bon trop vite) .
On a donc recherché un moyen d'évaluer nos résultats. Après qu'on nous ait suggéré des pistes sur le traitement d'images, nous nous sommes tournés vers le WGAN, dont on parlera plus tard , qui ne souffrait pas de telles difficultés.

De plus, après un bref entretien avec Ian LeCun (au cours duquel nous avons constaté que nous butions sur des problèmes biens établis de la théorie du GAN), une autre piste d'amélioration a été proposée.
D'après LeCun, la base de donnés Mnist était "trop binaire" (comprenez que l'on passe du blanc au noir trop rapidement dans l'image). Ainsi, le réseau n'était pas très enclin à fournir de la diversité (l'information se trouve dans ce pixel est pas dans celui juste à côté).
Il nous a conseillé de travailler sur des bases de données plus diverses (CIFAR 10 par exemple), cependant, le traitement de cette base de données ne pouvait plus se faire avec le perceptron : le sujet de la photo n'est pas toujours pris selon le même angle, à la même distance etc. D'où notre idée d'étudier les réseaux à Convolution, dont on parlera plus tard .

Chapitre 3

Améliorations classiques des Réseaux de neurones appliqués à GAN

Les réseaux de neurones que nous utilisons pour le GAN sont de simples perceptrons. De nombreuses méthodes pour améliorer les résultats et/ou la convergence ont été proposés pour ces types de réseaux. Nous avons étudié en particulier les algorithmes de descente de gradient avec pas adaptatif et les réseaux de neurones à convolution.

3.1 Algorithmes de descente de gradient à pas adaptatif

En utilisant la descente de gradient classique, nous avons constaté que, dans le générateur, presque seule la couche de sortie travaillait. Le GAN ne générait alors pas des images d'assez bonne qualité ni assez diverses. Nous nous sommes donc intéressés à d'autres algorithmes de descente, dans l'espoir qu'ils soient plus efficaces et atteignent plus "en profondeur" les réseaux.

Les algorithmes de descente auxquels nous nous sommes intéressés sont notamment des algorithmes à pas adaptatif. En effet, dans ces algorithmes, le pas change au fur et à mesure de l'apprentissage. Il peut être grand au début, pour aller dans la bonne direction, et petit à la fin pour plus de précision.

3.1.1 Momentum

Dans une descente de gradient classique, la formule de mise à jour des poids est la suivante.

$$W_{k+1} = W_k - \eta * \frac{\partial J}{\partial W}$$

On peut également le noter :

$$\Delta W_k = -\eta * \frac{\partial J}{\partial W}$$

Une première méthode qui est compatible avec tous les algorithmes suivants est de rajouter une inertie au gradient, ou momentum. Le but est de limiter les oscillations "inutiles" qui peuvent arriver lors d'une descente de gradient. On a alors :

$$\Delta W_k = \mu * W_k - \eta * \frac{\partial J}{\partial W}$$

3.1.2 AdaGrad

AdaGrad (qui signifie Adaptative Gradient) est un algorithme où le pas change en fonction de l'erreur. On calcule la somme des carrés des gradients :

$$g_{k+1} = g_k + (\frac{\partial J}{\partial W})^2$$

La formule de mise à jour des poids est alors :

$$\Delta W_k = -\frac{\partial J}{\partial W} * \frac{\eta}{\sqrt{g_{k+1}} + \epsilon}$$

ϵ est une valeur arbitrairement faible pour éviter une division par zéro et pour initialiser l'algorithme. L'inconvénient majeur de AdaGrad est que la quantité g_k ne peut qu'augmenter dans le temps, ce qui implique que le pas devient de plus en plus faible. Si l'apprentissage dure trop longtemps, les poids ne bougeront presque plus à cause du faible pas.

3.1.3 RMSProp

RMSProp est sensiblement identique à Adagrad mais avec une amélioration : au lieu de considérer la somme des carrés des gradients, on considère une pondération de cette somme. Cela permet de donner plus d'importance aux derniers gradients. On calcule donc :

$$g_{k+1} = \gamma * g_k + (1 - \gamma) * (\frac{\partial J}{\partial W})^2$$

La formule de mise à jour des poids est donc identique à celle d'Adagrad :

$$\Delta W_k = -\frac{\partial J}{\partial W} * \frac{\eta}{\sqrt{g_{k+1}} + \epsilon}$$

Dans le calcul de g_k, il y a un terme quadratique. On appelle donc g_k **moment d'ordre 2**. Il existe également un moment d'ordre 1 qui se calcule par :

$$g_{k+1} = \gamma * g_k + (1 - \gamma) * \frac{\partial J}{\partial W}$$

Cela donne comme équation de mise à jour des poids :

$$\Delta W_k = -\frac{\partial J}{\partial W} * \frac{\eta}{\sqrt{g_{t+1} - (m_{t+1})^2} + \epsilon}$$

3.1.4 Adadelta

Cet algorithme est similaire à RMSProp et utilise également une somme mobile pour calculer le moment d'ordre 2 du gradient, g_k. Cependant, au lieu d'avoir un η fixe, on introduit x_k, le moment d'ordre 2 de ΔW_k.

$$g_{k+1} = \gamma * g_k + (1-\gamma) * (\frac{\partial J}{\partial W})^2$$

$$x_{k+1} = \gamma * x_k + (1-\gamma) * (\Delta W_k)^2$$

On obtient donc :

$$\Delta W_k = -\frac{\partial J}{\partial W} * \frac{\sqrt{x_k + \epsilon}}{\sqrt{g_{t+1} - (m_{t+1})^2 + \epsilon}}$$

3.1.5 Adam

Adam (pour Adaptive Moment Estimation) adapte le pas en fonction des moments d'ordre 1 et 2 du gradient. Notons m_k le moment d'ordre 1 et v_k le moment d'ordre 2. On les calcule par :

$$m_{k+1} = \beta_1 * m_k + (1-\beta_1) * g_k$$

$$v_{k+1} = \beta_2 * v_k + (1-\beta_2) * g_k^2$$

Quand m_k et v_k sont initialisés à 0, ils sont biaisés vers 0. Pour pallier à cela, on considère $\widehat{m_k}$ et $\widehat{v_k}$:

$$\widehat{m_k} = \frac{m_k}{1-\beta_1^k}$$

$$\widehat{v_k} = \frac{v_k}{1-\beta_2^k}$$

On met à jour les poids avec :

$$\Delta W_k = -\frac{\eta}{\sqrt{\widehat{v_k}} + \epsilon} * \widehat{m_k}$$

3.1.6 Comparaison des algorithmes sur MNIST

Sur la figure 3.1, on voit que ces algorithmes améliorent le résultat sur MNIST autant en précision qu'en vitesse de convergence par rapport à l'algorithme de descente de gradient classique. Le meilleur algorithme à utiliser est Adam avec $\eta = 0.01$, $\gamma_1 = 0.9$ et $\gamma_2 = 0.999$ comme paramètres. Cependant, comme l'a étudié le groupe Couleuvre, il peut être avantageux d'utiliser deux algorithmes différents sur le générateur et le discriminateur. En effet, utiliser Adam sur le générateur et RMSProp sur le discriminateur permet de "ralentir" le discriminateur qui a tendance à devenir meilleur que le générateur.

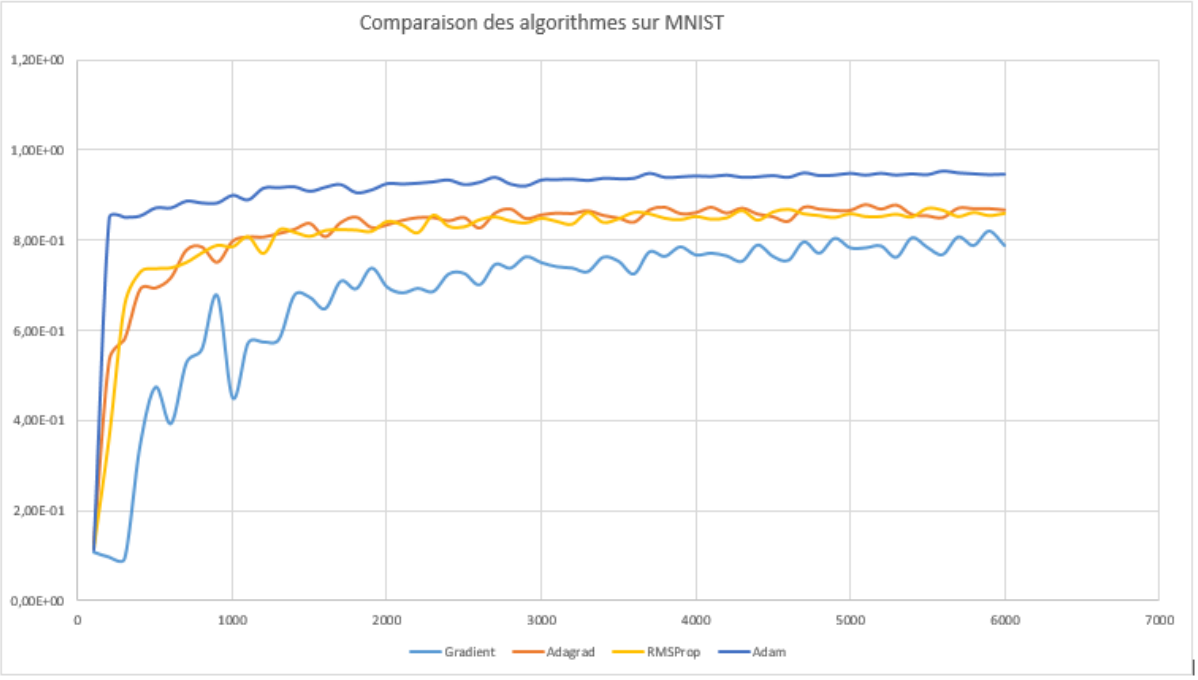


FIGURE 3.1 – Comparaison des algorithmes de descentes

3.2 Réseaux à convolution : DCGAN

Cette section sera remplie quand Romain finira par mettre sa partie du rapport en ligne.

Chapitre 4

Axes de Recherches : WGAN

Avec les différents résultats obtenus par nos premiers GAN, nous avons pu tirer, entre autres, deux conclusions importantes. Le GAN manque cruellement de stabilité (par exemple un petit changement de paramètre l'empêche de converger correctement) et de métriques pertinentes, c'est à dire que les scores des générateurs et des discriminateurs n'ont pas d'interprétations en termes de progrès de la qualité d'image perçue.
Les chercheurs se sont beaucoup attardés depuis 2016 sur la première question, en comparant par exemple les différents optimiseurs possibles [1], le deuxième point est lui moins souvent abordé. L'article de 2017 Wasserstein GAN [8] propose une méthode qui, en s'éloignant légèrement de la philosophie originale du papier de Goodfellow [3], tente d'apporter une réponse à ces deux questions, avec, en particulier, une métrique pertinente.

4.1 Problématique de la descente de gradient simultanée

L'article de Goodfellow semble démontrer la convergence du système GAN, cependant la mise en œuvre montre que cette convergence n'est pas aussi évidente à obtenir. En effet, il semblerait que la stratégie de descente de gradient de l'algorithme de GAN ne permette pas d'assurer cette convergence. Le blog inFERENCe [6] décrit une partie du problème en se basant sur l'article The Numerics of GANs [7].

Ces articles montrent que la descente de gradient simultanée n'est pas simplement une double descente de gradient, mais une descente de gradient dans un champ vectoriel. L'algorithme de GAN effectue l'optimisation suivante :

$$x_{t+1} \leftarrow x_t + hv(x_t) \text{ avec } v(x) = \begin{pmatrix} \frac{\partial}{\partial\theta} f(\theta, \phi) \\ \frac{\partial}{\partial\phi} g(\theta, \phi) \end{pmatrix}.$$

f et g étant respectivement les fonctions de coût du Discriminateur et du Générateur. Cependant on constate 2 problèmes. D'une part, l'algorithme basé sur la théorie des jeux, qui consiste à

faire "jouer" tour à tour le Discriminateur et le Générateur pour optimiser ses paramètres ne consiste qu'en une approximation de la simultanéité de la descente. D'autre part, il n'y a aucune preuve que le champ vectoriel x dans lequel l'on se déplace possède des propriétés conservatives. En particulier, rien ne garantit que le rotationnel soit nul, ce qui implique que la descente de gradient ne soit pas garantie d'aller vers un minimum, même local (Figure : 4.1) !

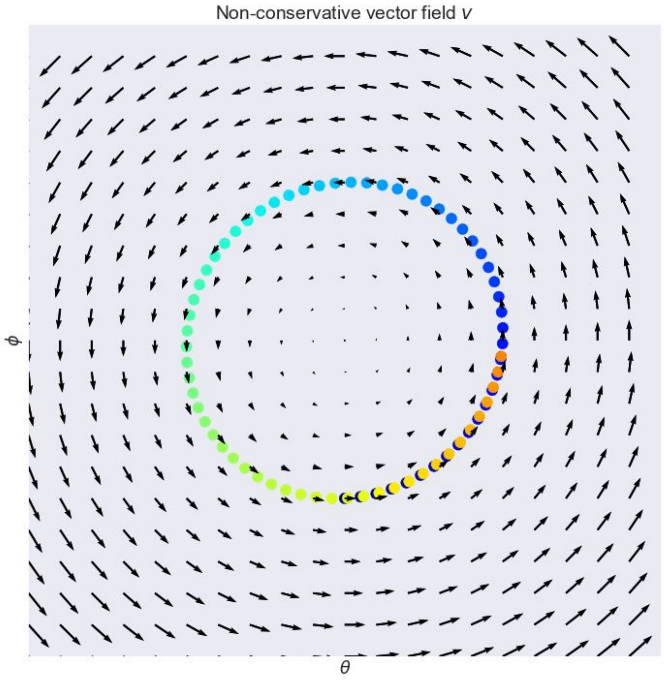


FIGURE 4.1 – champ vectoriel non conservatif : exemple de descente de gradient

On est donc à la recherche d'une autre approche qui contournerait ce problème.

4.2 L'approche Wasserstein GAN

Le papier Wasserstein GAN [2], propose une autre approche que celle de la théorie des jeux.

Il s'agit de calculer une divergence entre distributions afin de se servir de cette métrique pour faire l'apprentissage de l'une sur l'autre. Au premier abord, cette approche ne semble pas si éloignée de l'approche de Goodfellow, mais elle en est sensiblement différente. Dans l'approche précédente, on tente de minimiser la divergence entre deux distributions par un algorithme utilisant 2 réseaux de neurones. Cependant, nous n'avons jamais accès à cette divergence (que l'on utilise des fonctions d'erreur pour approcher la KL-divergence ou une autre), et comme nous l'avons montré précédemment, la convergence n'est pas réellement assurée.

L'idée du papier Wasserstein GAN est de calculer explicitement une divergence entre deux ensembles. Cela n'est pas une question simple, comme nous avons pu le voir au chapitre 1. En effet, nous n'avons généralement pas accès à la distribution $p_{\text{réel}}$ mais uniquement à un échantillon tiré

de cette distribution.
Cependant il apparait possible de calculer une divergence entre deux ensembles à l'aide des réseaux de neurones. Cela est possible en tout cas avec la divergence de Wasserstein, comme le montre ce papier, nous allons revenir sur les étapes de raisonnement.

L'objectif est de rapprocher 2 distributions en utilisant la métrique de Wasserstein à l'ordre 1. Celle-ci s'écrit :

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

On peut la nommer également distance Earth-Mover, c'est à dire distance du déplacement de terre. En effet cette métrique calcule l'effort à faire pour passer d'une distribution à l'autre (Figure : 4.2). Par exemple, si l'on a deux terrains contenant des tas de terre, la hauteur de terre en un point représente la densité de probabilité à cette endroit, le volume complet de terre étant le même (cela représente l'intégrale sur le terrain), alors la distance EM entre les deux terrains est l'effort minimal que l'on peut faire pour déplacer la terre de l'un des terrains pour le faire ressembler à l'autre. Il y a une infinité de façons de déplacer la terre, avec des efforts différents. La distance de Wasserstein est le coût en utilisant le plan de transport optimal. Cela se traduit également en termes de probabilités jointes, pour une approche plus mathématique.

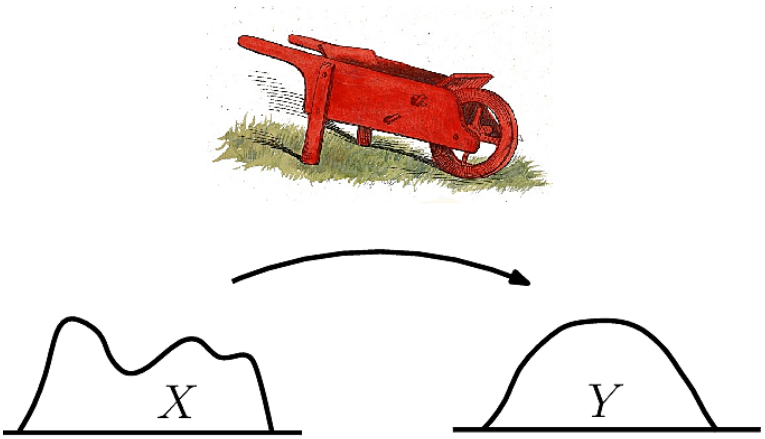


FIGURE 4.2 – Illustration de la divergence Earth-Mover

L'idée d'utiliser cette divergence de Wasserstein provient des propriétés mathématiques qui lui sont propres. Cette divergence est théoriquement plus pertinente pour l'apprentissage des GANs. Le détail se trouve dans ce papier, mais cela se peut se résumer ainsi : Wassertein donne plus d'informations que la KL-Divergence et ses dérivées (Jensen-Shannon, etc) en étant, entre autres, bien définie lorsque les supports de distribution sont disjoints et en étant moins souvent constante, avec donc des gradients non nuls et plus adaptés à l'apprentissage.

On ne peut toujours pas se servir de cette divergence, mais un théorème (la dualité Kantorovich-Rubinstein [10]) permet d'obtenir une nouvelle forme de cette divergence :

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

Vous pouvez en voir une preuve simplifiée sur le blog de Vincent Hermann [5]. On se retrouve alors à calculer un sup sur un ensemble de fonctions (les fonctions 1-Lipschitziennes), et cela est pratique car c'est justement ce que permet de faire un réseau de neurones : Simuler des

fonctions que l'on optimise par rapport à un paramètre ! On construit un réseau qui joue le rôle des fonctions f, on a donc des paramètres W à optimiser pour trouver la valeur maximum. Il ne reste qu'à s'assurer que les réseaux de neurones peuvent garantir le caractère 1-Lipschitzien.

Une méthode pour s'assurer de cette propriété est de restreindre les poids dans un intervalle [-c, c] (on parle de weight-clipping). Cependant on obtient un caractère K-Lipschitzien, avec un K inconnu. (En terme de preuve, il suffit de voir que des matrices avec ses propriétés sont toutes K-Lipschitziennes avec un même K.) Cela nous assure que l'on peut calculer non pas $W(\mathbb{P}_r, \mathbb{P}_g)$ mais $K * W(\mathbb{P}_r, \mathbb{P}_g)$, mais le K étant fixe tout au long de l'apprentissage cela reste une métrique pertinente.

Nous avons donc la possibilité avec un réseau de neurones de calculer la métrique de Wasserstein et nous allons pouvoir nous en servir pour l'apprentissage d'un générateur.

4.3 Mise en œuvre

Fort de cette métrique que l'on peut calculer, nous allons pouvoir mettre en place un algorithme d'apprentissage pour la génération d'image.

Tout d'abord, nous avons toujours besoin de deux réseaux, l'un pour la génération, dont le rôle est strictement identique aux générateurs qui ont pu être vus avant, et l'autre pour le calcul de la divergence de Wasserstein. Ce second réseau est appelé critique (plutôt que Discriminateur) par la littérature. Attention, le critique simule une fonction f qui est telle que $\mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_g}[f(x)]$ soit la sortie d'un batch contenant de vraies images et des images de synthèses, c'est à dire que $f(x)$ et $f(G(z))$ n'ont pas de sens contrairement aux $D(x)$ et $D(G(z))$ vus précédemment. Afin que le critique calcule effectivement $W(\mathbb{P}_r, \mathbb{P}_g)$, il est nécessaire de faire converger les paramètres du critique pour obtenir $f_{\max}$. Une fois $W(\mathbb{P}_r, \mathbb{P}_g)$ obtenu, on peut chercher à le minimiser pour faire progresser le générateur, on effectue donc une descente de gradient à partir de ce coût. Cependant on a en fait $f_{\max}(\mathbb{P}_r, \mathbb{P}_g, x)$, c'est à dire que $f_{\max}$ dépend des distributions à un instant donné, par conséquent après une itération on obtient :

$$\mathbb{E}_{x\sim\mathbb{P}_r}[f_{\max}((\mathbb{P}_r, \mathbb{P}_g), x)] - \mathbb{E}_{x\sim(\mathbb{P}_g+\Delta\mathbb{P}_g)}[f_{\max}((\mathbb{P}_r, \mathbb{P}_g), x)]$$

Ce qui est différent de :

$$\mathbb{E}_{x\sim\mathbb{P}_r}[f_{\max}((\mathbb{P}_r, (\mathbb{P}_g + \Delta\mathbb{P}_g)), x)] - \mathbb{E}_{x\sim\mathbb{P}_g}[f_{\max}((\mathbb{P}_r, (\mathbb{P}_g + \Delta\mathbb{P}_g)), x)] = W(\mathbb{P}_r, \mathbb{P}_g + \Delta\mathbb{P}_g)$$

Il faut donc à chaque itération refaire converger le critique pour garantir que l'on a bien la fonction $f_{\max}$ correspondant à la distribution $\mathbb{P}_g$ courante (on a en effet $\mathbb{P}_r$ fixe tout du long).

L'algorithme se dessine alors simplement :

— Faire converger le critique en maximisant la quantité $\mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_g}[f(x)]$
  - Récupérer un batch d'images réelles
  - Générer une batch d'images virtuelles
  - Calculer $\mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_g}[f(x)]$
  - Rétro-propager dans le critique
  - Recommencer jusqu'à convergence

— Faire évoluer le générateur en minimisant la quantité $\mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_g}[f(x)]$
- Générer un batch d'images virtuelles
- Calculer $-\mathbb{E}_{x\sim\mathbb{P}_g}[f(x)]$
- Il n'est pas nécessaire d'effectuer le calcul pour de vraies images, car elles ne servent pas dans la rétro-propagation dans le générateur
- Rétro-propager dans le critique sans le modifier
- Rétro-propager dans le générateur

— Recommencer les 2 étapes jusqu'à convergence du générateur

On notera dans cet algorithme deux zones floues qui sont les ré-itérations "jusqu'à la convergence". En effet il est particulièrement difficile de s'assurer que l'on a bien convergé, ou plutôt de savoir que l'on a convergé suffisamment. Comme dans tous les algorithmes de ce genre, il y a un trade-off entre le temps de calcul et la précision.

Pour la convergence du critique, l'article Wasserstein [2] suggère de le faire converger de façon certaine au début, i.e de s'assurer que l'on calcule bien $W(\mathbb{P}_r, \mathbb{P}_g)$ avec $\mathbb{P}_g$ étant la sortie du générateur avant tout apprentissage, c'est-à-dire à priori un bruit. Pour cela, on effectue longuement la première étape avant toutes choses. Puis on peut estimer qu'une itération de descente de gradient sur le générateur change très peu ce dernier, et par une hypothèse de continuité sur la distance de Wasserstein (qui semble justifiée si l'on considère cette distance comme la distance Earth-Mover) on peut estimer qu'il faudra peu d'itérations de montée de gradient pour faire de nouveau converger "suffisamment" le critique.

Pour la convergence du générateur, on en revient aux problèmes classiques du GAN qui consiste à se demander à quel moment le générateur est suffisamment performant. Nous y reviendrons dans les parties suivantes.

4.4 Essais pratiques du WGAN

Après l'étude théorique des Wassertstein GAN, nous avons mis en pratique dans notre bibliothèque python l'algorithme correspondant.

4.4.1 Modification logicielle

Comme cela a été observé, si la réflexion est très différente du GAN classique, l'algorithme des WGANs reste très proche de l'algorithme original.
Nous avons dû cependant rajouter plusieurs features à notre librairie pour pouvoir en faire l'implémentation correcte.

— Nous avons créé une nouvelle classe *WGanGame* héritant de la classe *GanGame* afin de redéfinir quelques fonctions :

play_and_learn – Il n'y a que 2 types d'apprentissages possibles, le critique et le générateur.

critic_learning – Remplace discriminator_learning afin que l'entrée soit un batch d'images réelles et virtuelles, il faut également effectuer une montée de gradient.

generator_learning – Il est nécessaire de vérifier que le critique est en mode descente

de gradient désormais.
— Il a fallu mettre en place le choix de descente ou de montée de gradient au sein des réseaux. En effet on ne peut pas simplement choisir un pas négatif pour le critique, car il y a une rétro-propagation décroissante dans le discriminateur lors de l'apprentissage du générateur. Cela a impliqué plusieurs gros changements dans les paramètres à passer aux fonctions de rétro-propagation.
— Afin de garantir le caractère lipschitzien, il a fallu ajouter une possibilité de weight-clipping. Cela a été effectué en utilisant la méthode *clamp* de numpy, et en rajoutant un paramètre au constructeur des couches FCs.
— Il a fallu ajouter également 2 fonctions d'erreurs pour le critique et le générateur.

4.4.2 Méthodologie et résultats du WGAN

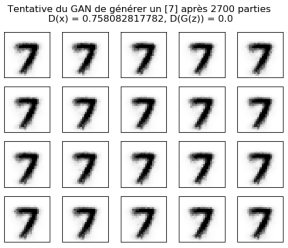
Afin de vérifier si la méthode fonctionne, nous avons eu 2 approches : Reproduire les réseaux présents dans la littérature sur le WGAN et tester avec les mêmes paramètres ou appliquer le WGAN à des structures proches des meilleurs résultats que nous avons obtenus avec des GANs simples.

Pour essayer de reproduire la littérature, nous nous sommes heurtés au problème suivant, elle utilise quasi-systématiquement des réseaux convolués. C'est logique car ce sont ceux qui offrent les meilleures performances lorsque les réseaux traitent des images. Cependant notre implémentation des réseaux à convolutions n'a jamais été suffisamment satisfaisante pour pouvoir s'en servir sur les GANs.

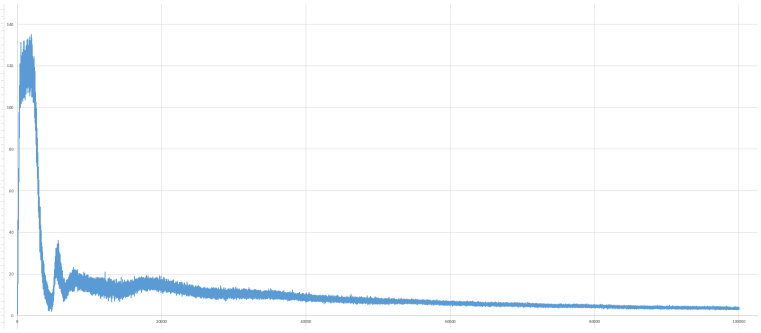
La deuxième méthode a eu plus de succès. Il n'est pas possible de reprendre les structures telles quelles. En effet, nous utilisions que des Sigmoids pour fonctions d'activations, restreignant les sorties. Or, pour le Wasserstein GAN, il est important de calculer librement la distance EM. C'est pourquoi le critique utilise désormais des fonctions ReLU d'activation. On a donc un critique de la forme [784-20-1], avec des poids restreints à [-0.1, 0.1]. Pour le générateur, la même structure peut être utilisée, soit un [100-300-784] pour le moment sans bruit. Il n'est pas nécessaire (ni même conseillé) de restreindre les poids du générateur.

Pour le reste des paramètres, nous avons suivi le papier [2], c'est à dire un ratio de 5 pour le critique, des pas de 0.05 (au lieu de 0.0005), en utilisant RMSprop.

La figure 4.3a montre le résultat de la structure précédente. La première conclusion est que l'algorithme peut fonctionner et donner des résultats, ce qui est très encourageant. On observe que les 7 sont biens formés, que la délimitation avec la partie blanche n'est pas nette, et qu'il ont tous sensiblement la même forme. On est donc dans un mode collapse qui se confirme lorsque l'on regarde l'évolution. Un seul 7 est visible à chaque instant, mais sa forme évolue constamment entre plusieurs 7 différents.
La figure 4.3b montre l'évolution du score au cours de ce même apprentissage. On y observe bien la pertinence entre la distance de Wasserstein (le score si les convergence sont bien faites) et la qualité visuelle des chiffres.
La forte augmentation au début correspond à la phase d'initialisation, où l'on ne fait apprendre que le critique afin de calculer la divergence initiale. On observe au cours de l'apprentissage l'amélioration de l'image de façon synchrone avec la chute du score, avec une très rapide progression au début (pour les contours grossier), puis une progression plus limitée (pour les contours



(a) Résultat du WGAN pour 27 000 parties, D(x) = 7.58)



(b) Distance de Wasserstein pendant l'apprentissage

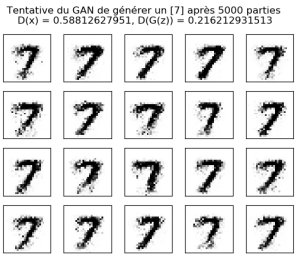
FIGURE 4.3 – Apprentissage d'un 7 sans bruit en sortie (100 000 parties)

fins), et enfin une stagnation visuelle à partir de 25 000 apprentissages.

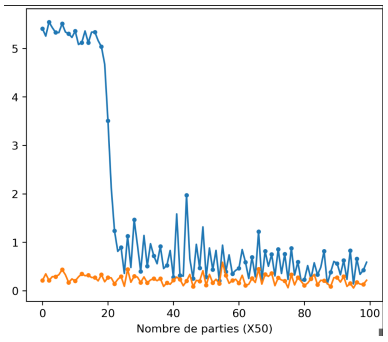
L'étape suivante consiste à tenter d'obtenir de la diversité, soit en s'éloignant du mode Collapse, soit en produisant plusieurs chiffres (objectifs en réalité très similaires).

Nous avons introduit du bruit en couche de sortie (comme vu au chapitre 2.7) afin d'observer de la diversité. En tentant de produire plusieurs chiffres, nous n'avons eu aucun résultat concluant, seul une tache centrale persiste, que l'on peut peut être interpréter comme une moyenne des chiffres existants. Avec un seul chiffre, on obtient le résultat de la figure 4.4a, on parvient donc à sortir du Collapse, mais on perd en qualité visuelle, le tracé n'étant plus lisse, mais fragmenté.

Attention, ces derniers résultats sont à prendre avec beaucoup de précaution, ils ont été faits rapidement en fin de projet et n'ont donc pas pu être affinés. Ainsi les conclusions sont que le WGAN fonctionne, que les métriques qu'ils proposent semblent pertinentes vis à vis de ce qui est présenté dans le papier, mais qu'il est soumis à une part importante des problèmes des GANs, une intolérance vis à vis des paramètres approximatifs et le mode Collapse.



(a) Résultat du WGAN pour 50 000 parties, D(x) = 0.58)



(b) Distance de Wasserstein pendant l'apprentissage

FIGURE 4.4 – Apprentissage d'un 7 avec bruit en sortie (100 000 parties)

4.5 Réflexion sur l'approche

4.5.1 Utilisation de la distance de Wasserstein pour évaluer un générateur

4.5.2 Problème du caractère Lipschitziens des réseaux de neurones

4.5.3 Peut-on adapter la logique de cette algorithme à d'autre méthode ?

Bibliographie

[1] Comparaison des optimiser pour le gan.

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

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[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] Vincent Hermann. Wasserstein gan and the kantorovich-rubinstein duality. `https://vincentherrmann.github.io/blog/wasserstein/`.

[6] Ferenc Huszár. Gans are being fixed in more than a way. `http://www.inference.vc/gans-are-being-fixed-in-more-than-one-way/`.

[7] Sebastian Nowozin Lars Mescheder and Andreas Geiger. The numerics of gan.

[8] Soutmith Chintala Martin Arjovsky and Léon Bottou. Wasserstein gan. 2017.

[9] 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.

[10] Cédric Villani. Optimal transport : Old and new. 2009.