

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

Etude 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

Résumé

Mots-clefs

Mots-clefs

Table des matières

1	Introduction aux réseaux de neurones et premières applications	1
1.1	Outils utilisés pour le projet	1
1.2	Réseaux de neurones et fonctionnement	1
1.2.1	Le neurone	2
1.2.2	Réseau de neurones et perceptron	2
1.2.3	Apprentissage par rétro-propagation	3
1.3	Conception logicielle des réseaux de neurones	4
1.3.1	Structure du code	5
1.4	Application au problème du XOR	5
1.5	Application à la base de données MNIST	7
2	Generative Adversarial Networks	9
2.1	Principe	9
2.2	Apprentissage	10
2.3	Paramètres des GANS	11
2.4	Structure et utilisation du code	11
2.5	Premier résultats pour des GANs simples	11
2.6	Mode Collapse et Bruit en entrée	11
2.7	Résultat sans collapse	11
3	Améliorations classiques des Réseaux de neurones appliqués à GAN	12
4	Améliorations spécifiques au GAN	13
5	Axes de Recherches : WGAN	14

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. On mettra également en place une structure informatique en python pour les utiliser.

1.1 Outils utilisés pour le projet

Ce projet possède une dimension de conception logiciel. Il s'agit de programmer sans utiliser de bibliothèques existantes des réseaux de neurones efficaces et performants adaptés aux problèmes que nous souhaitons résoudre.

Étant donné l'évolution prévue de notre code, perceptron simple pour XOR ou MNIST, puis mise en place du GAN, et enfin toute sortes d'améliorations utiles, nous devons être particulièrement vigilant sur la souplesse de notre code. La programmation en équipe, sur une longue durée et avec de telles contraintes nécessitent la mise en place d'outils et certains choix techniques.

1.2 Réseaux de neurones et fonctionnement

Les réseaux de neurones font partis des piliers de l'intelligence artificielle. Leur fonctionnement est basé sur une interprétation sommaire du cerveau humain. Des neurones seules reçoivent des signaux, les traitent et renvoient un signal de sortie. Les neurones sont alors agrégés en réseaux avec des entrées du réseaux et des sorties. On modélise la plasticité du cerveaux par des

paramètres variables qui changent au cours de l'apprentissage, ce dernier se faisant en comparant des sorties attendues aux sorties obtenues.

1.2.1 Le neurone

L'unité de base du réseau est le neurone, on peut l'imaginer comme une fonction mathématique. Lui sont attribuées n entrées, chacune affectée d'un poids w_i et une fonction mathématique de \mathbb{R} dans \mathbb{R} . Le rôle du neurone sera de renvoyer le résultat de la fonction, appliquée à la somme pondérée par le poids des entrées. On pourrait ajouter un biais comme paramètre de notre neurone afin d'ajuster notre résultat (choisir quand une fonction seuil renvoie 1 par exemple).

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 différente délimitation du plan. On peut donc imaginer faire « apprendre » au réseau quels points délimiter 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. Dans les structures les plus basiques, la sortie d'une couche est utilisée comme l'entrée de la couche suivante. 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 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.

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.

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. Dans le cadre du perceptron, nous utilisons seulement un apprentissage supervisé, la rétro-propagation des erreurs. 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.

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.

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 :

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

En forme matricielle, ces équations donnent :

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

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 développements 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 défini par sa matrice de poids *weights*, son vecteur de biais *biais* ainsi que sa fonction d'activation *activationfunction*, nécessairement commune à tout les neurones dans cette structure.

Présentation des méthodes

— *compute* : Propagation d'une entrée au sein de cette couche

—

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 $2 \rightarrow 2 \rightarrow 1$ 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 ???. Cependant, ce résultat n'était pas obtenu dans l'intégralité des apprentissages, nous fournissant des résultats très différents, comme sur la figure ??. La littérature, et en particulier les rapports des années précédentes [2] et [3], nous ont montré que le XOR n'était effectivement pas juste dans 100% des cas.

Nous avons donc soumis le réseau à de nombreux apprentissages, en faisant varier les paramètres

ainsi que la forme du réseau. Voici les résultats les plus intéressants :

Conclusion sur le XOR Instabilité du réseau $2 \rightarrow 2 \rightarrow 1$ et comparaison avec le $2 \rightarrow 4 \rightarrow 1$ et le $2 \rightarrow 2 \rightarrow 2 \rightarrow 1$

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 reconnaître des chiffres manuscrits, les données étaient les suivantes :

- 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.

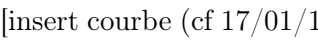
Les poids sont initialisés avec une répartition gaussienne centrée réduite. Les biais sont initialisés à 0.

De bons résultats ont été obtenus avec le réseau suivant, conformément à la littérature :

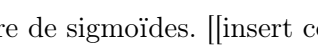
- Eta 0.2
- Sigmoïde 0.1
- Réseau à une couche cachée de 300 neurones et 10 neurones de sortie (784-300-10)
- Apprentissage stochastique

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 ????. Nous allons maintenant voir l'influence des différents paramètres.

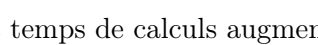
Variation d'êta :

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

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.  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.

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'apprentissage 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.  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 motif 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 [1].

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 ordinateurs. Ses modèles cherchent à produire de données nouvelles respectant un certain nombre de contraintes. Les applications possible sont très nombreuses, tant au niveau scientifiques qu'industrielles, avec par exemple la modélisation de nouvelles protéines, le dessin de circuit 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.

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 influencer que sur θ_D et D a accès aux données de G, mais ne peut influencer que sur θ_G . 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 $D(G(z)) = 1$.

2.3 Paramètres des GANS

Voici une description des paramètres principaux sur lequel 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

description des modifications importantes pour le GAN (or optimisation du code source)

2.5 Premier résultats pour des GANs simples

présentation des paramètres principaux (bruit seulement à l'entrée, etc) description des résultats avec des GANs simple, sans optimisations, et évaluation des différences de paramètres

2.6 Mode Collapse et Bruit en entrée

Le mode collapse aura été présenté à la section d'avant, tentative d'explication de pourquoi on en sort avec le bruit

2.7 Résultat sans collapse

même chose que la section premiers résultats, mais avec le bruit en entrée

Chapitre 3

Améliorations classiques des Réseaux de neurones appliqués à GAN

Applications des optimizers (RMS, ...), DCgan, etc

Chapitre 4

Améliorations spécifiques au GAN

MiniBatch, etc

Chapitre 5

Axes de Recherches : WGAN

Bibliographie

- [1] Intro au gan - 2014.
- [2] Rapport appartement.
- [3] Rapport pinaple.