

Réseaux adverses génératifs (GAN)  
pour la génération d'images de microstructures

NAJLAA SRIFI  
LOIC IMBERT  
ANTOINE SCHOONAERT  
ARCADY RUVIDIC

2023  
Janvier

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Le concept des GAN</b>	<b>3</b>
<b>3</b>	<b>Les réseaux de neurones</b>	<b>3</b>
3.1	Les fondamentaux d'un réseau de neurones . . . . .	3
3.1.1	La structure des réseaux de neurones . . . . .	3
3.1.2	L'information dans un réseau de neurones . . . . .	5
3.1.3	Les neurones sensitifs . . . . .	5
3.1.4	Les neurones des couches intermédiaires . . . . .	5
3.1.5	L'apprentissage des réseaux de neurones . . . . .	5
3.2	Les réseaux de neurones de convolution . . . . .	6
3.2.1	La convolution . . . . .	6
3.2.2	Le max pooling . . . . .	7
3.2.3	Le padding . . . . .	8
<b>4</b>	<b>Les concepts essentiels à l'apprentissage des réseaux de neurones</b>	<b>8</b>
4.1	Les fonctions de perte . . . . .	8
4.1.1	Les fonctions de régression . . . . .	8
4.1.2	Les fonctions de classification . . . . .	9
4.1.3	Une fonction de perte originale . . . . .	9
4.2	Rétropropagation et descente du gradient . . . . .	10
4.2.1	La descente du gradient : la fondation le l'optimisation . . . . .	10
4.2.2	Application aux réseaux de neurones . . . . .	10
4.2.3	Les algorithmes d'optimisation . . . . .	12
4.2.4	La descente du gradient avec momentum/mémoire . . . . .	12
<b>5</b>	<b>Etude du comportement des réseaux de neurones</b>	<b>15</b>
5.1	Les réseaux de neurones standards . . . . .	15
5.2	L'organisation des couches . . . . .	15
5.3	Expérimentation avec un réseau de neurone sans librairies . . . . .	16
5.3.1	La base de données du MNIST . . . . .	16
5.3.2	La structure du réseau . . . . .	16
5.3.3	Les paramètres du réseau . . . . .	16
5.3.4	Les tests d'efficacité . . . . .	17
5.3.5	L'influence du learning rate . . . . .	17
5.3.6	L'influence du nombre de neurones de la couche intermédiaire . . . . .	18
5.3.7	L'influence de la valeurs des poids initiaux . . . . .	18
5.3.8	L'influence de la fonction d'activation de la couche cachée . . . . .	19

5.3.9	L'influence de la taille du batch	19
5.3.10	Résultat avec des paramètres corrects	20
5.4	Les réseaux de neurones convolutifs	20
5.4.1	L'influence des paramètres d'entraînement	21
5.4.2	L'influence du learning rate	21
5.4.3	L'influence du nombre d'epochs	22
5.4.4	L'influence du type de Pooling	22
5.4.5	L'influence de type de la fonction perte	23
5.4.6	L'influence du type de la fonction d'optimisation	23
5.4.7	L'influence de type de la fonction d'activation	23
5.4.8	L'influence du type de modèle	24
5.4.9	Comparaison avec un réseau non-convolutif	25
<b>6</b>	<b>Les fondamentaux des GAN</b>	<b>26</b>
6.1	Le discriminateur	26
6.2	Le générateur	26
6.3	Une étape d'entraînement	27
6.4	La fonction de perte et algorithme d'optimisation	27
6.5	Choix des fonctions de perte appliqué aux GAN	27
6.5.1	La fonction minimax	27
6.5.2	La fonction non saturante	28
6.5.3	La fonction des moindres carrés	28
6.6	Implémentation d'un GAN	28
6.6.1	Une première implémentation	28
6.6.2	Modification des hyperparamètres du modèle GAN	29
6.7	Implémentation d'un DCGAN	35
<b>7</b>	<b>Les GAN appliqués à notre problème</b>	<b>36</b>
7.1	La structure du GAN	36
7.2	Le choix de la fonction de perte	36
7.3	Les premiers essais avec un DCGAN	37
7.4	Le GAN de Wasserstein	37
7.4.1	Introduction au GAN de Wasserstein	37
7.4.2	Implémentation du GAN de Wasserstein	39
7.5	Autres pistes	40
<b>8</b>	<b>Conclusion</b>	<b>41</b>
<b>9</b>	<b>Annexe</b>	<b>42</b>
9.1	Vecteur gaussien	42
9.2	Divergence de Kullback-Leibler	42
9.3	Divergence de Jensen-Shannon	42
9.4	La matrice de confusion	43

# 1 Introduction

Les propriétés physiques d'un matériau sont essentielles dans la prévision de la réponse de celui-ci. Dans notre situation, il est nécessaire d'employer la tomographie par rayon X pour visualiser une tranche de notre matériau. Cette méthode étant très coûteuse, il est difficile de produire un nombre suffisant de données pour appliquer des calculs extensifs sur ces images. Notre approche pour pallier ce problème est d'employer une intelligence artificielle pour générer des images de microstructures fidèles à la réalité, de manière artificielle. L'objectif de notre Projet Tech est donc d'explorer la validité et la capacité de cette option, en s'intéressant particulièrement à un modèle de Deep Learning, le réseau adverse génératif ou GAN (Generative Adversarial Network). La figure 1 ci-dessous est un exemple d'échantillon à disposition dans notre base de données.

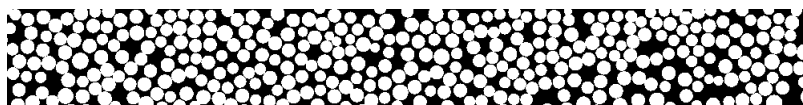


FIGURE 1 – Exemple d'image par tomographie

## 2 Le concept des GAN

Les GAN sont une classe d'algorithmes de Deep Learning non supervisés, c'est-à-dire, un algorithme où l'objectif est d'entraîner le modèle à analyser et regrouper les informations des données en entrée. Cette idée a initialement été introduite par Ian J. Goodfellow et al [3]. Essentiellement, il s'agit de l'ensemble de deux réseaux de neurones : le générateur et le discriminateur. Le générateur est le réseau de neurone qu'on souhaite réellement entraîner afin de générer des images proches de celles de la base de données. Le discriminateur est un réseau de neurone qui permet d'entraîner le générateur, son rôle étant d'apprendre à différencier les images réelles et celles produites par le générateur. Il renvoie une valeur entre 0 et 1, représentant la probabilité pour chaque image d'être réelle. L'erreur commise par le discriminateur est ce qu'on exploite pour mettre à jour chacun des réseaux.

Il s'agit donc d'un jeu à deux joueurs, le générateur essayant d'apprendre à tromper le discriminateur, et le discriminateur apprenant à différencier les images. On souhaite ainsi que l'entraînement améliore petit à petit les deux réseaux en même temps pour qu'on puisse finalement générer des images identiques à celles de la base de données.

La figure 2 est une représentation schématique du fonctionnement d'un GAN. Les notations introduites dans le schéma seront conservées tout le long du document.

## 3 Les réseaux de neurones

### 3.1 Les fondamentaux d'un réseau de neurones

Les réseaux de neurones, aussi appelés parfois avec les préfixes artificiels ou stimulés, sont un sous-ensemble de méthodes de Machine Learning qui se situe au cœur du Deep Learning.

#### 3.1.1 La structure des réseaux de neurones

La conception du réseau de neurones est à l'origine schématiquement inspirée du fonctionnement des neurones biologiques. Comme représenté par la figure 3, un réseau est constitué d'un ensemble de neurones, répartis en plusieurs couches, et communiquant au travers de liaisons appelées synapses.

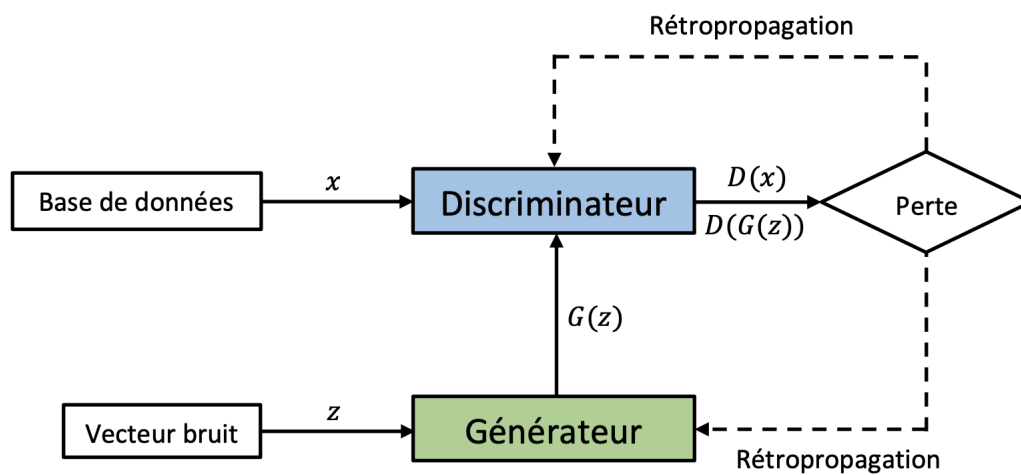


FIGURE 2 – Schéma de fonctionnement d'un GAN

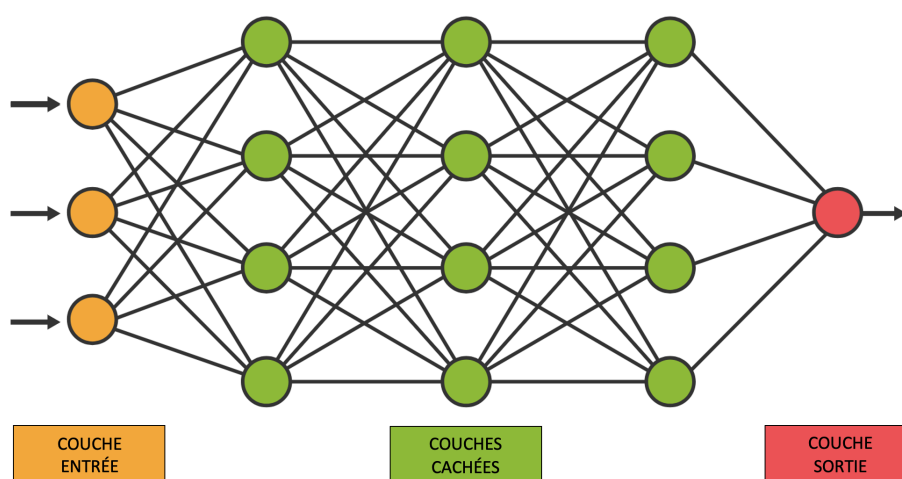


FIGURE 3 – Schéma de la structure d'un réseau de neurones

### 3.1.2 L'information dans un réseau de neurones

Les neurones de la couche d'entrée captent un scalaire, appelé intensité. Chaque neurone va effectuer une opération sur les scalaires pour obtenir une nouvelle intensité qui sera transmise à la couche suivante. Ce processus est répété de couche en couche jusqu'à ce que l'information nous est retransmise par la couche de sortie.

### 3.1.3 Les neurones sensitifs

Les neurones de la couche d'entrée ont pour fonction de percevoir une information de notre donnée en entrée, ces neurones sont appelés neurones sensitifs. Chacun de ces neurones sera sensible à un paramètre de notre entrée. Par exemple, si on souhaite passer un vecteur au réseau de neurone, on aura autant de neurones dans cette première couche que d'éléments dans notre vecteur. Ces intensités perçues sont ensuite transmises à la couche suivante où démarre les opérations.

### 3.1.4 Les neurones des couches intermédiaires

Le schéma 4 représente l'ensemble des opérations appliquées sur une information passant à travers un neurone. Chaque synapse va pondérer son intensité transmise par un poids  $w$ . La synapse est activatrice si  $w > 0$ , inhibitrice si  $w < 0$ . Ainsi, chaque neurone intermédiaire va percevoir  $n$  intensités pondérées par  $(w_i)_{1 \leq i \leq n}$ . L'intensité totale perçue va alors être la somme des ces intensités pondérées, à laquelle le neurone va ajouter une valeur  $b$  appelée le biais, soit  $w_0b + \sum_{i=1}^n w_i x_i$  avec  $(x_i)_{1 \leq i \leq n}$  les intensités non pondérées traversant les synapses.

Dès lors, notre neurone va appliquer une fonction d'activation  $\sigma$  ou  $f$  à l'intensité perçue, afin de calculer l'intensité qui sera transmise aux couches de neurones suivantes. Il existe 4 fonctions d'activation courantes : la fonction tangente hyperbolique, la fonction sigmoïde, la fonction ReLU et la fonction LeakyReLU. La fonction ReLU est une fonction rampe sur  $[0, +\infty[$  et vaut 0 sur  $] - \infty, 0]$ . La fonction LeakyReLU présente une première rampe de coefficient  $a$  sur  $] - \infty, 0]$  et une seconde rampe de coefficient  $b$  sur  $[0, +\infty[$ . La fonction sigmoïde est un quotient d'exponentielle. En général, pour le traitement d'image, on préfère la fonction ReLU car le coût de calcul d'une rampe est plus faible qu'une exponentielle.

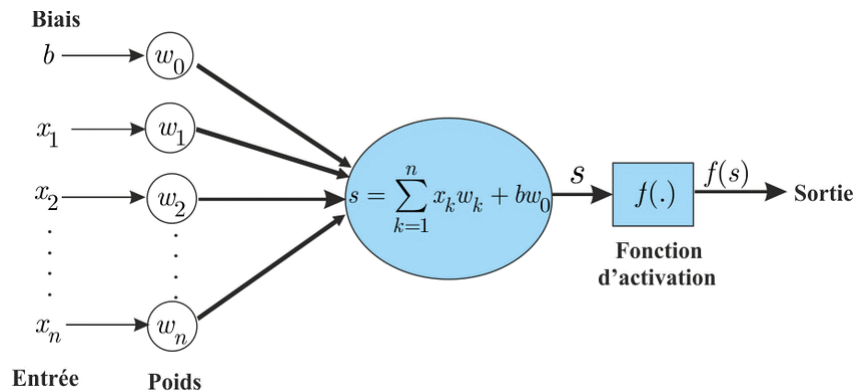


FIGURE 4 – L'opération lors de la transmission d'une intensité

### 3.1.5 L'apprentissage des réseaux de neurones

Lorsqu'on souhaite entraîner un réseau de neurones, l'objectif est de mettre à jour pas à pas les poids et biais de chaque couche afin de calibrer la sortie graduellement vers une information souhaitée et

exploitable. La qualité de l'information produite va être qualifiée de manière quantitative par la fonction de perte (développée dans les paragraphes suivants). Plus la fonction de perte est faible, plus l'information en sortie est de qualité. L'entraînement sera donc fondé sur cette fonction de perte que l'on cherchera à minimiser.

Pour cela, nous allons effectuer un processus de descente du gradient stochastique, s'appuyant sur la rétropropagation. Ce processus consiste à calculer le gradient par rapport à la fonction de perte déterminé après chaque batch afin de parcourir le réseau de neurones en sens inverse pour mettre à jour les poids et biais couche par couche suivant un algorithme d'optimisation. Notons que le gradient dans cette situation représente l'ensemble des dérivées partielles de la fonction de perte par rapport à chacun des poids et biais présent dans le réseau de neurones. Le calcul de celui-ci peut s'avérer être très coûteux selon l'échelle du réseau entraîné.

## 3.2 Les réseaux de neurones de convolution

En apprentissage automatique, un réseau de neurones convolutif ou réseau de neurones à convolution est un type de réseau de neurones artificiel dans lequel le motif de connexion entre les neurones est inspiré par le cortex visuel des animaux. Les neurones de cette région du cerveau sont arrangés de sorte à correspondre à des régions qui se chevauchent lors du pavage du champ visuel. Leur fonctionnement est inspiré par les processus biologiques, consistant en un empilage multicouche de perceptrons, dont le but est de prétraiter des petites quantités d'informations.

### 3.2.1 La convolution

Un réseau de neurones convolutif est basé principalement sur le produit de convolution qui consiste à appliquer un filtre sur une image. Le filtre est une petite matrice carrée de taille variable (3x3, 5x5, 9x9, ...) appelée kernel ou noyaux de convolution. Le fonctionnement est représenté sur le schéma 5 : nous appliquons un kernel sur chaque portion de l'image, et combinons les valeurs de ces pixels pondérées par la matrice afin d'obtenir une information de taille réduite. Cette opération est en fait un simple produit scalaire entre la matrice des pixels sélectionnés et la matrice du kernel.

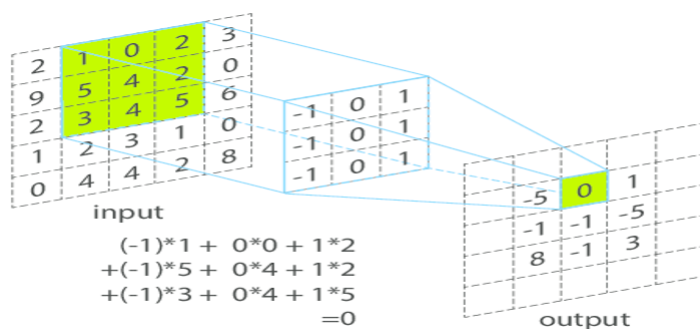


FIGURE 5 – Produit de convolution

Dans les réseaux de neurones de type classificateur (comme le discriminateur), le réseau prend en entrée une image et lui applique un certain nombre de kernels ou noyaux de convolution. Chacun produit une carte de convolutions qui met en évidence une caractéristique particulière de l'image et ainsi de suite pour les couches suivantes, avec un processus d'augmentation de la complexité des caractéristiques interprétées. Cependant il est possible et même fortement recommandé dans un réseau de convolutions d'ajouter une opération intermédiaire nommée Pooling, permettant de réduire la quantité d'information tout en conservant son sens.

Supposons que nous avons notre entrée  $x$ , ainsi que notre noyau de convolution  $w$ . La convolution en 1D ( $x * w$ ) est définie comme suit :

$$a_t = \sum_{i=-inf}^{inf} x_i w_{t-i} \quad (1)$$

où  $a_t$  est le résultat après la convolution à l'emplacement  $t$  et correspond à la pré-activation dans le cas des réseaux de neurones. Puisque la convolution est commutative et qu'en pratique  $w \in R_H$  est de taille finie, nous pouvons obtenir la formule plus intuitive suivante :

$$a_t = \sum_{i=0}^H x_{i-t} w_i \quad (2)$$

$H$  est la taille du filtre.

Traditionnellement plusieurs noyaux  $w$  sont utilisés simultanément, afin d'extraire différentes caractéristiques. On peut constater que  $a_i$  (appelé Feature Map en Anglais) n'est calculé qu'à partir d'informations locales, dont la taille est déterminée par  $H$ . Lors du traitement des langues naturelles,  $H$  correspond à la taille des N-grams utilisés pour extraire l'information. Afin d'apprendre des dépendances à long terme tout en gardant le partage de paramètres au maximum, nous pouvons empiler des couches cachées, ou encore faire du sous-échantillonnage en effectuant du Pooling, ou du Striding.

### 3.2.2 Le max pooling

Le Max Pooling est un processus de discrétisation sur des échantillons. Son objectif est de sous-échantillonner l'image d'entrée en réduisant sa dimension. De plus, il permet de réduire le coût de calcul en diminuant le nombre de paramètres à apprendre et fournit une invariance par petites translations (si une petite translation ne modifie pas le maximum de la région balayée, le maximum de cette région restera le même et donc la nouvelle matrice créée restera identique). Le schéma 6 montre une opération de Max Pooling avec un kernel 2x2 et une stride de 2. Notons que le paramètre de stride influe beaucoup sur l'information finale. Si celui-ci valait 1 sur cet exemple, on obtiendrait un 8 sur la case bleue, ce qui élimine cette grande différence d'intensité.

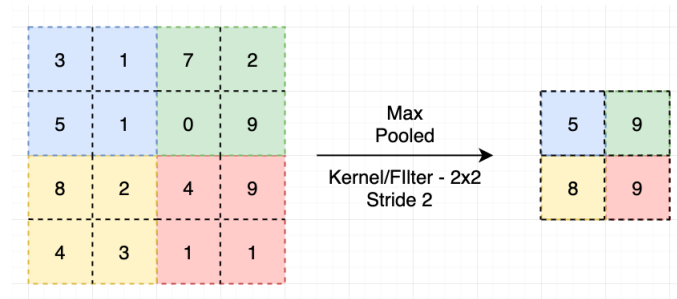


FIGURE 6 – Opération de Max Pooling



### 3.2.3 Le padding

La convolution réduisant la taille de l'image, la dimension de l'image finale est, de façon générale :

$$m = n - p + 1 \quad (3)$$

où  $n$  est la dimension de l'image initiale et  $p$  celle du noyau.

Ainsi, si on veut obtenir une image de même dimension que l'image initiale, il faut rajouter des zéros autour de la matrice initiale avant la convolution, on dit qu'on fait du Padding ou Zero Padding.

Le Padding n'est pas obligatoire et, il arrive généralement qu'on fasse sans. Il peut donc sembler anecdotique mais il a une réelle utilité. Analysons les pixels dans les coins de la matrice en guise d'exemple. Ces pixels verront passer le noyau de convolution une seule fois, alors que la plupart des autres pixels le rencontreront plus de deux fois. Les pixels situés au bord auront donc moins d'influence sur ceux de l'image de sortie que les autres. C'est pour limiter cet effet de bord qu'on élargit la matrice initiale avec des zéros, de sorte à ce que les pixels aux extrémités ne soient pas sous-représentés.

## 4 Les concepts essentiels à l'apprentissage des réseaux de neurones

### 4.1 Les fonctions de perte

Nous recherchons généralement la convergence d'un modèle sur un ensemble de données d'apprentissage observé comme la minimisation de la fonction de perte choisie sur l'ensemble des données d'apprentissage. Si la fonction de perte ne varie plus, on peut considérer que le gradient a atteint un minimum local de cette fonction de perte et que l'entraînement est terminé. Explorons donc les diverses fonctions de pertes qui sont mises à notre disposition.

#### 4.1.1 Les fonctions de régression

Dans un modèle de régression, on cherche à prédire la valeur d'une variation continue (par exemple le prix d'une voiture, la température, etc). Les fonctions de perte pour ce type de problème dépendent tous sur une opération fondamentale, l'écart (soustraction) entre la valeur prédite et la valeur réelle. On dispose donc d'une multitude de fonctions de perte fondées sur ce principe.

Notons  $p_i$  la valeur prédite par le modèle,  $y_i$  la valeur réelle et  $n$  le nombre de données dans un batch.  $P$  et  $Y$  représentent des vecteurs avec l'ensemble des valeurs pour un batch d'entraînement.

##### a. La fonction écarts moyens

La forme la plus simple de fonction de coût est donc la moyenne de tous ces écarts sur les données d'entraînement.

$$L(P, Y) = \frac{1}{n} \sum_{i=1}^n p_i - y_i \quad (4)$$

##### b. La fonction moindres carrés

Le désavantage de la fonction précédente et d'avoir des termes pouvant s'annuler en additionnant des valeurs positives et négatives. On emploie donc la différence au carré.

De plus, cette fonction permet d'augmenter l'influence des grandes erreurs et de minimiser l'influence des faibles valeurs.

$$MSE(P, Y) = \frac{1}{n} \sum_{i=1}^n (p_i - y_i)^2 \quad (5)$$

Cette fonction est souvent notée MSE en Anglais (Mean Square Error), ou encore L2 lorsqu'on la cherche dans les bibliothèques de programmation.

### c. La fonction moyenne quadratique

Afin d'obtenir des erreurs du même ordre de grandeur que nos valeurs, on utilise parfois la racine carrée de la fonction précédente.

$$RMSE(P, Y) = \sqrt{\frac{1}{n} \sum_{i=1}^n (p_i - y_i)^2} \quad (6)$$

Cette fonction est notée RMSE (Root Mean Square Error).

## 4.1.2 Les fonctions de classification

La deuxième catégorie de fonctions de perte est utilisée pour les problèmes de classification. La perte par entropie croisée (Cross Entropy en Anglais) est la méthode la plus utilisée dans ce domaine.

$$CE(p, y) = -p \cdot \log(y) \quad (7)$$

Cette fonction est utilisée lorsqu'il existe plusieurs classes de données, et que l'entrée correspond l'une d'entre elles. Cette formule mesure seulement l'entropie croisée pour une seule observation.

### a. Entropie croisée catégorique

Pour exploiter la formule 7, on calcule généralement la moyenne de celle-ci, ce qui donne la fonction d'entropie croisée catégorique.

$$CCE(P, Y) = \frac{1}{n} \cdot -P^T \cdot \log(Y) \quad (8)$$

### b. Entropie croisée binaire

Un cas particulier qui nous est intéressant est la cas d'une classification binaire (par exemple vrai ou faux, chat ou chien, etc). Dans ce cas-ci, on a  $y = 0$  ou  $y = 1$ , et on parle d'entropie croisée binaire.

$$BCE(p, y) = \begin{cases} -\log(p) & \text{si } y = 1 \\ -\log(1 - p) & \text{si } y = 0 \end{cases} \quad (9)$$

## 4.1.3 Une fonction de perte originale

Il peut arriver que le problème que l'on cherche à résoudre ne permette pas l'utilisation de fonctions de perte pré-codées correspondant exactement à ce qu'on souhaite mesurer. Il n'est alors pas plus difficile d'en créer une personnelle, la seule condition à respecter étant la forme  $L = f(P, Y)$ . Parfois, certains paramètres devront être fixés au préalable, en particulier pour certains coefficients ou seuils. Il est ainsi possible de pondérer, multiplier ou additionner des termes supplémentaires avec des objectifs autres que la simple convergence du modèle.

## 4.2 Rétropropagation et descente du gradient

On emploie souvent les termes de rétropropagation et d'optimisation de manière mélangée, cependant ces deux termes désignent deux choses différentes. La rétropropagation est le processus calculant le gradient pour chaque poids dans le réseau de neurones, tandis que l'optimisation est l'algorithme de descente du gradient qui met à jour ces poids en utilisant la valeur du gradient obtenu par rétropropagation. Dans cette partie on donnera quelques exemples d'algorithme de descente du gradient du premier ordre.

### 4.2.1 La descente du gradient : la fondation de l'optimisation

L'optimisation de la fonction de perte impliquera d'optimiser des fonctions de perte convexes et dérivables. L'algorithme classique de minimisation de telles fonctions est la descente du gradient. Cet algorithme est itératif :

1. On fixe un point initial  $x_0$ , un pas  $\eta$  et une condition d'arrêt  $\epsilon$ .
2. On calcule la direction de descente  $-\Delta f(x_i)$  pour  $i = 0$ .
3. Ensuite, on met à jour notre point selon la formule  $x_{i+1} = x_i - \eta \Delta f(x)$  avec  $i = 0$ .
4. On affecte à  $i$  la valeur  $i+1$ .
5. On réitère jusqu'à obtenir  $|x_{i+1} - x_i| < \epsilon$ .
6. On obtient notre résultat  $\operatorname{argmin}(f) = x_{i+1}$ .

### 4.2.2 Application aux réseaux de neurones

La rétropropagation consiste à comprendre comment la modification des poids et des biais dans un réseau modifie la fonction de perte [6]. On peut considérer que notre modèle est bon s'il est capable de trouver la bonne valeur de sortie  $y$  (le rendement vrai) associé aux données d'entrées correspondantes, et le rendement que l'on prédit avec notre modèle noté  $\phi(z)$ . On peut donc construire une fonction d'erreur simple (qu'on peut aussi appeler coût du modèle)  $C_x$  pour chacun des sites d'étude  $x \in X$ .

$$C_x = [\phi(z(x)) - y(x)]^2 \quad (10)$$

Cette fonction de coût pénalise les sites pour lesquels le rendement a été mal prédit. L'objectif, sans trop de surprise, est de chercher à minimiser cette fonction de coût, ce qui revient à dire que l'on cherche à minimiser les erreurs de prédiction sur les sites d'étude. A partir de cette fonction de coût par site d'étude, on peut construire une fonction de coût pour l'ensemble du modèle qui est la moyenne du coût associé à chaque site d'étude.

$$C = \frac{1}{|X|} \sum C_x \quad (11)$$

avec  $n$  le nombre de sites d'étude.

Introduisons des notations pour écrire nos formules avec le schéma 7.

- $w_{jk}^l$  est le poids de la connexion entre le neurone  $k$  de la couche  $l-1$  et le neurone  $j$  de la couche  $l$ .
- $b_j^l$  est le biais du neurone  $j$  de la couche  $l$ .
- $a_j^l$  est l'activation du neurone  $j$  de la couche  $l$ .
- $L$  est le nombre de couches dans le réseau de neurones donc le numéro de la dernière couche car la couche d'entrée 0 n'est pas prise en compte.

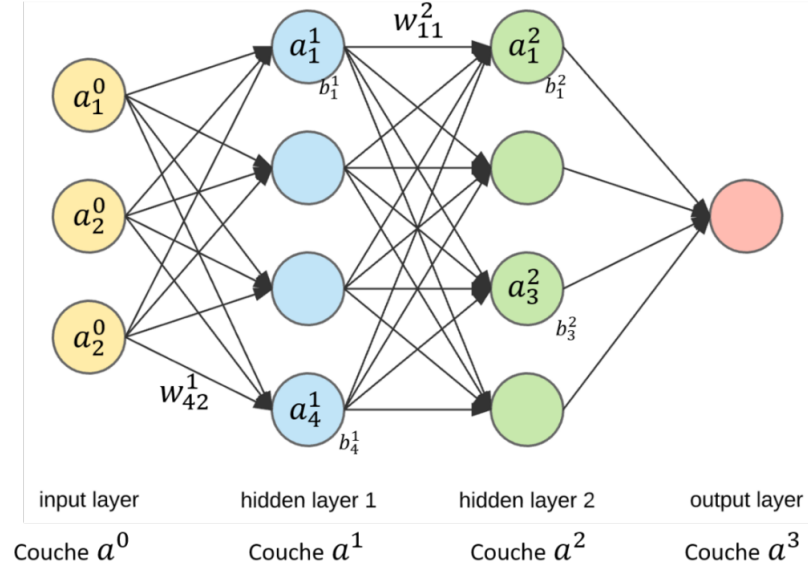


FIGURE 7 – Notations réseau de neurones

Dans ce cas la fonction coût est égale à :

$$C_x = [a^l(x) - y(x)]^2 \quad (12)$$

avec

$$a^l = \phi(w^l a^{l-1} + b^l) = \phi(z^l) \quad (13)$$

La fonction de coût seule ne suffit pas pour entraîner le modèle, car celle-ci ne fait que calculer une valeur représentant la performance du modèle. Entre autres, il nous faut une méthode pour interpréter cette valeur, et un algorithme pour mettre à jour les poids  $w_1, w_2, \dots, w_k$  et le biais  $b$  ou  $w_0$  afin de rendre les prédictions meilleures. A l'état initial, les poids sont choisis de manière aléatoire mais au fur et à mesure que le modèle apprend, il faut lui faire indiquer dans quel direction le modèle doit modifier ses poids et biais pour obtenir de meilleurs résultats. Pour ce faire, on calcule les dérivées de la fonction coût par rapport aux poids et aux biais, autrement dit, un gradient.

$$\frac{\partial C_x}{\partial w^l} = \frac{\partial z^l(x)}{\partial w^l} + \frac{\partial a^l}{\partial z^l(x)} + \frac{\partial C_x}{\partial a^l} \quad (14)$$

$$\frac{\partial C_x}{\partial b^l} = \frac{\partial z^l(x)}{\partial b^l} + \frac{\partial a^l}{\partial z^l(x)} + \frac{\partial C_x}{\partial a^l} \quad (15)$$

Puis on met à jour les paramètres du réseau de neurones avec les formules suivantes :

$$w^{l+1} = w^l - \eta \frac{\partial C}{\partial w^l} \quad (16)$$

$$b^{l+1} = b^l - \eta \frac{\partial C}{\partial b^l} \quad (17)$$

avec  $\eta$  la vitesse d'apprentissage qu'on fixe comme hyperparamètre du modèle dans certains cas. Sa valeur peut influencer sur le déroulement de l'entraînement. Augmenter cette valeur de manière modérée peut accélérer le processus de l'entraînement. Une valeur trop petite ne ferait que retarder le processus. En revanche,

une valeur trop grande peut engendrer des instabilités ou "rater" le minimum local, ce qui n'est pas non plus souhaitable. Il faut donc trouver un juste milieu pour avoir un algorithme le plus rapide possible. Notons que parmi les algorithmes d'optimisation, il en existe plusieurs qui résolvent ce problème par un pas adaptatif, c'est-à-dire, modifié par l'algorithme selon l'amplitude du gradient.

### 4.2.3 Les algorithmes d'optimisation

La descente du gradient reste une méthode très simple et se révèle être inefficace dans certains cas. On dispose donc de plusieurs variantes de cette méthode qu'on utilise dans des cas plus particuliers ou si on souhaite avoir une meilleure performance.

### 4.2.4 La descente du gradient avec momentum/mémoire

On rappelle la formule de la descente du gradient classique appliquée à un poids  $w$ .

$$w^{i+1} = w^i - \alpha \nabla f(w^i) \quad (18)$$

Avec  $\alpha$  le taux d'apprentissage,  $f$  la fonction de perte et  $w^i$  la valeur du poids considéré à l'instant  $t$ .

On pose  $s^i = -\alpha \nabla f(w^i)$ . On a alors pour équation  $w^{i+1} = w^i - s^i$ . Ajouter un momentum/une mémoire à la descente du gradient consiste à utiliser  $s^i = -\alpha \nabla f(w^i) + \beta s^{i-1}$  avec  $\beta$  le coefficient de mémoire.

#### a. La descente du gradient stochastique (SGD)

La descente du gradient stochastique est une variante de la descente du gradient. Cette méthode s'adapte à une fonction de coût adaptée sous forme d'une moyenne  $f = \frac{1}{n} \sum_{i=1}^n f_i$ . Pour un gros jeu de données tel que le réseau de neurones, possédant un grand nombre de variables (ici des poids) sur plusieurs batchs différents, on veut pouvoir trouver un gradient qui sera représentatif du gradient sur un lot de valeur. Par exemple,  $f_i$  peut représenter la fonction de perte calculée sur les batchs précédents pour un neurone donné. Le gradient utilisé dans la descente sera alors  $\Delta f = \frac{1}{n} \sum_{i=1}^n n \Delta f_i$ . On effectue ainsi une moyenne temporelle du gradient. On peut également effectuer un moyennage du gradient par neurone : on sépare les neurones en plusieurs régions neuronales, à chaque batch, on calcule la moyenne du gradient des neurones  $\Delta f = \frac{1}{n} \sum_{i=1}^n n \Delta_{w_i} f_i$ , les  $w_i$  représentant les neurones du lot et  $n$  le nombre de neurone du lots. On parle alors de descente du gradient stochastique "mini-lot".

L'implémentation de cet algorithme dans pytorch est **torch.optim.Adagrad** qui prend jusqu'à 8 paramètres :

- **params** (iterable) - itérable des paramètres à optimiser ou dicte la définition des groupes de paramètres
- **lr** (float) - taux d'apprentissage
- **momentum** (float, optional) - le terme du moment (par défaut : 0)
- **weight\_decay** (float, optional) - déclin du poids (pénalité L2) (par défaut : 0)
- **dampening** (float, optional) - déclin du moment (par défaut : 0)
- **nesterov** (bool, optional) - utiliser le moment de Nesterov (par défaut : False)
- **maximize** (bool, optional) - maximiser les paramètres en fonction de l'objectif, au lieu de les minimiser (par défaut : False)
- **foreach** (bool, optional) - utiliser ou non l'implémentation foreach de l'optimisateur (par défaut : None)

### b. Le gradient au pas adaptatif (AdaGrad)

AdaGrad est un algorithme d'optimisation basé sur le gradient qui adapte le taux d'apprentissage aux paramètres, en effectuant des mises à jour plus petites (c'est-à-dire des taux d'apprentissage faibles) pour les paramètres associés à des fonctionnalités fréquentes, et des mises à jour plus importantes pour les paramètres associés à des caractéristiques peu fréquentes. Le pas est calculé comme suit :

$$\eta'_t = \frac{\eta}{\sqrt{(\alpha_t + \epsilon)}} \quad (19)$$

avec

$$\alpha_t = \sum_{i=1}^t \left( \frac{\partial C}{\partial w^i} \right)^2 \quad (20)$$

La principale faiblesse d'AdaGrad est son accumulation des gradients au carré dans le dénominateur : puisque chaque terme ajouté est positif, la somme accumulée ne cesse de croître pendant l'entraînement. Cela entraîne à son tour une diminution du taux d'apprentissage et finit par devenir infiniment petit, à quel point l'algorithme n'est plus en mesure d'acquérir des connaissances supplémentaires.

L'implémentation de cet algorithme se fait sous pytorch en utilisant la fonction : **torch.optim.Adagrad** qui prend 7 paramètres :

- **params** (iterable) – itérable des paramètres à optimiser ou dicte la définition des groupes de paramètres
- **lr** (float, optionnel) – taux d'apprentissage (par défaut : 1e-2)
- **lr\_decay** (float, optionnel) – déclin du taux d'apprentissage (par défaut : 0)
- **weight\_decay** (float, optionnel) – déclin du poids (pénalité L2) (par défaut : 0)
- **eps** (float, optionnel) – terme ajouté au dénominateur pour améliorer la stabilité numérique (par défaut : 1e-10)
- **foreach** (bool, optionnel) – si l'implémentation foreach de l'optimiseur est utilisée (par défaut : None)
- **maximize** (bool, optionnel) – maximiser les paramètres en fonction de l'objectif, au lieu de minimiser (par défaut : False)

### c. La propagation de la moyenne quadratique (RMSProp)

La propagation de la moyenne quadratique est une extension de la descente du gradient AdaGrad employant un déclin de la moyenne des gradients partiels dans l'adaptation du pas du gradient pour chaque paramètre. Utiliser ce déclin signifie oublier les valeurs les plus anciennes du gradient pour se concentrer sur les valeurs récentes pour contrôler l'entraînement.

L'implémentation de cet algorithme se fait sous pytorch en utilisant la fonction : **torch.optim.RMSProp** qui prend 9 paramètres :

- **params** (iterable) - itérable des paramètres à optimiser ou dicte la définition des groupes de paramètres
- **lr** (float) - taux d'apprentissage
- **momentum** (float, optional) - le terme du moment (par défaut : 0)
- **alpha** (float, optional) – constante de lissage (par défaut : 0.99)
- **eps** (float, optionnel) – terme ajouté au dénominateur pour améliorer la stabilité numérique (par défaut : 1e-8)
- **centered** (bool, optional) – calculer ou non le RMSProp centré, le gradient étant normalisé par une estimation de sa variance
- **weight\_decay** (float, optionnel) – déclin du poids (pénalité L2) (par défaut : 0)

- **foreach** (bool, optionnel) – si l’implémentation foreach de l’optimiseur est utilisée (par défaut : Aucun)
- **maximize**(bool, optionnel) – maximiser les paramètres en fonction de l’objectif, au lieu de minimiser (par défaut : False)

#### d. L’estimation adaptative du moment (Adam)

L’estimation adaptative du moment (Adam) est une autre méthode qui calcule les taux d’apprentissage adaptatifs pour chaque paramètre. Adam réalise notamment les avantages d’AdaGrad et de RMSProp. Au lieu d’adapter les taux d’apprentissage des paramètres en fonction du premier moment moyen (la moyenne) comme dans RMSProp, Adam utilise également la moyenne des seconds moments des gradients (la variance non centrée). Plus précisément, l’algorithme calcule une moyenne mobile exponentielle du gradient et du gradient au carré, et les paramètres  $\beta_1$  et  $\beta_2$  contrôlent les taux de décroissance de ces moyennes mobiles. La valeur initiale des moyennes mobiles et les valeurs  $\beta_1$  et  $\beta_2$  proches de 1,0 (recommandé) entraînent un biais des estimations de moment vers zéro. Ce biais est surmonté en calculant d’abord les estimations biaisées avant de calculer ensuite les estimations corrigées du biais.

Tout d’abord, nous devons maintenir un vecteur de moment et une norme à l’infini pondérée exponentiellement pour chaque paramètre optimisé dans le cadre de la recherche, appelés respectivement  $m$  et  $v$ . Ils sont initialisés à 0.0 au début de la recherche.

L’algorithme est exécuté de manière itérative sur le temps  $t$  à partir de  $t=1$ , et chaque itération consiste à calculer un nouveau jeu de valeurs de paramètres  $x$ .

Ensuite, le premier moment est mis à jour à l’aide du gradient et d’un hyperparamètre  $\beta_1$  et le deuxième moment est mis à jour à l’aide du carré du gradient et d’un hyperparamètre  $\beta_2$  comme suit :

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial C}{\partial w^l} \quad (21)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \frac{\partial C^2}{\partial w^l} \quad (22)$$

On contrecarre ces biais en calculant les estimations des premier et deuxième moments corrigées du biais :

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad (23)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (24)$$

$$w^l = w^l - \frac{\eta}{\sqrt{(\hat{v}_t + \epsilon)}} \hat{m}_t \quad (25)$$

L’implémentation de cet algorithme se fait sous pytorch en utilisant la fonction **torch.optim.Adam** qui prend 10 paramètres :

- **params** (iterable) – itérable des paramètres à optimiser ou dicte la définition des groupes de paramètres
- **lr** (float, optionnel) – taux d’apprentissage (par défaut : 1e-3)
- **betas** (Tuple[float, float], optionnel) – coefficients utilisés pour calculer les moyennes mobiles du gradient et son carré (par défaut : (0.9, 0.999))
- **eps** (float, optionnel) – terme ajouté au dénominateur pour améliorer la stabilité numérique (par défaut : 1e-8)
- **weight\_decay** (float, optionnel) – déclin du poids (pénalité L2) (par défaut : 0)
- **amsgrad**(bool, optionnel) – s’il faut utiliser la variante AMSGrad de cet algorithme de l’article Sur la convergence d’Adam et au-delà (par défaut : False)
- **foreach** (bool, optionnel) – si l’implémentation foreach de l’optimiseur est utilisée (par défaut : Aucun)
- **maximize**(bool, optionnel) – maximiser les paramètres en fonction de l’objectif, au lieu de minimiser (par défaut : False)

- **capturable** (bool, optionnel) – si cette instance peut être capturée en toute sécurité dans un graphe CUDA. Passer True peut altérer les performances sans graphique, donc si vous n’avez pas l’intention de capturer graphiquement cette instance, laissez False (par défaut : False)
- **fused** (bool, optionnel) – si l’implémentation fusionnée de l’optimiseur est utilisée

## 5 Etude du comportement des réseaux de neurones

L’objectif de cette section est d’étudier les paramètres pouvant influencer sur le comportement des réseaux de neurones. Ces éléments peuvent être internes au réseau, donc au niveau structurel, ou externe au réseau, particulièrement les hyperparamètres de l’entraînement. Il sera aussi question de comparer un réseau de neurone standard à un réseau de neurones à convolution pour un problème de classification.

### 5.1 Les réseaux de neurones standards

Ce qu’on définit comme réseau de neurones standard est un réseau dont les couches cachées principales sont constituées de couches non-convolutives, en particulier les couches linéaires. Une couche linéaire applique simplement une transformation linéaire à l’information entrante.

### 5.2 L’organisation des couches

Le nombre de couches est un des premiers éléments représentatifs de la structure d’un réseau de neurones. Le choix de celui-ci dépend de plusieurs paramètres : le type de problème que l’on souhaite résoudre, la précision voulue, l’efficacité en temps et encore d’autres pour certains cas particuliers.

Par définition, un réseau de neurones contient au moins deux couches : la couche d’entrée et la couche de sortie. Ainsi, la forme la plus simple de réseau de neurones contient seulement ces deux couches. Cela implique qu’on ne peut qu’appliquer des fonctions élémentaires sur notre entrée. En général, on considère qu’un tel réseau est capable de résoudre les problèmes de séparation linéaire. Par exemple, il peut s’agir de la séparation d’un nuage de points en deux ensembles de taille égale. En revanche, il existe un problème typique montrant la limite de cette structure ; il s’agit du XOR. Si on souhaite séparer par une droite les points (0,0) et (1,1) de (0,1) et (1,0), alors on n’obtiendra jamais de réussite car il s’agit tout simplement d’une situation insoluble.

La solution au problème du XOR peut être apporté par l’ajout d’une couche cachée dans le réseau. On pourrait alors calculer deux droites pour effectivement séparer les points. Plus généralement, lorsqu’on a une couche cachée, on peut approximer une fonction associant un espace fini à un autre. Par exemple, un ensemble d’images à un ensemble de classe pour un classificateur.

A partir de deux couches et au-delà, on considère qu’augmenter le nombre de couches permet d’apprendre des informations plus complexes, au coût du temps d’entraînement [2]. Par exemple, si on entraîne un réseau de neurones pour la reconnaissance de visages, la première couche cachée pourrait percevoir une image globale. La couche suivante pourrait reconnaître quelques formes géométriques, et plus on avance, plus les couches peuvent identifier des motifs précis tel des yeux, une bouche, etc. Cependant, rajouter trop de couches ne pose pas seulement un problème de temps, mais peut engendrer une situation de surapprentissage, c’est-à-dire que le réseau de neurones peut trouver une solution trop particulière par rapport à une donnée. Ainsi, il faut essayer plusieurs configurations pour trouver un juste milieu entre les capacités de notre réseau de neurone et le coût d’entraînement.



Le nombre de neurones par couche dépend souvent du type de couche, mais il n'y a généralement pas de règle absolue à suivre. Il est tout de même conseillé à maintenir la somme du nombre de neurones des couches cachées en-dessous de la somme des neurones d'entrée et de sortie. Pour la couche d'entrée, il n'y a pas vraiment de choix, le nombre de neurones dépendra directement de la taille de l'information. Si on passe en entrée une image rgb de taille 32x32, alors le nombre de neurones en entrée sera 32x32x3. La nombre de neurones en sorties dépendra de l'information sortante. Pour un problème de classification binaire, on aura seulement un neurone. De même pour un problème de régression. Pour un problème de classification multi-classes, le nombre de neurones sera égal au nombre de classes possibles.

### 5.3 Expérimentation avec un réseau de neurone sans librairies

Dans cette partie nous allons mettre en place un réseau de neurones sans employer de librairies pré-faites afin de pouvoir comprendre le fonctionnement des réseaux de neurones au plus bas niveau. Ceci devrait permettre d'éviter l'image classique de la "boîte noire" que sont les intelligences artificielles. Ce réseau sera conçu pour reconnaître les chiffres 0 et 1 parmi les dix de la base de données du MNIST représenté sur la figure 8.



FIGURE 8 – Extrait de la base de données MNIST

#### 5.3.1 La base de données du MNIST

La base de données MNIST contient 70000 images des nombres allant de 0 à 9 manuscrits. Nous isolons les images de 0 et 1 afin de réduire la dimension du problème pour ce premier entraînement de réseau de neurones. Les images sont de dimension 1x28x28. Nous prenons 80% des données pour entraîner le réseau et 20% pour tester le réseau sur des données qu'il n'aura jamais rencontrées.

#### 5.3.2 La structure du réseau

Ce réseau prendra en entrée des images 28x28. Ces images créent donc des vecteurs d'entrées de 784 coordonnées. Nous ne mettons qu'une seule couche cachée et 1 neurone de sortie qui donnera la probabilité que l'image soit une image contenant un 1. Et inversement, la probabilité que l'image soit un 0 sera égale à 1-la probabilité que l'image soit un 1. Notons que cette structure est discutable après coup, puisqu'il aurait été plus intelligent de formuler le problème sous forme de reconnaissance de 1, car avec une unique sortie, on ne peut pas obtenir de probabilité que l'image soit un 0, mais seulement non-1.

#### 5.3.3 Les paramètres du réseau

Nous allons étudier l'impact des différents paramètres. Les paramètres sont :

- le nombre de neurones dans la couche intermédiaire
- le nombre d'epochs
- les poids initiaux

- le type de fonction d'activation
- la valeur du pas du gradient aussi appelé learning rate
- taille de batch

Nous fixons la taille d'un batch et nous utilisons la descente du gradient à pas constant.

#### 5.3.4 Les tests d'efficacité

Nous allons étudier l'impact des paramètres sur l'efficacité du réseau. Nous allons également le comparer à un réseau de neurones convolutif. Le score est le score dit RMSE pour Root Mean Square Error. Le score est calculé sur les données tests sur lesquelles le réseau ne s'est jamais entraîné. Pour rappel, l'expression du score RMSE est donnée au paragraphe 4.1.1.

#### 5.3.5 L'influence du learning rate

Les paramètres sont :

- 5 neurones sur la couche intermédiaire
- nombre d'epochs = 1000
- poids initiaux = 0
- sigmoïde pour tous les neurones
- taille de batch = 1

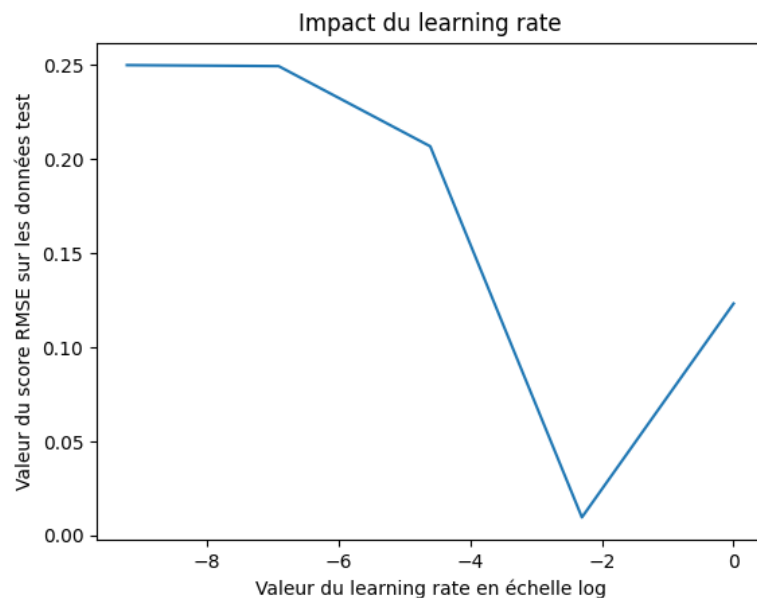


FIGURE 9 – Impact du learning rate sur le score RMSE

Ainsi on constate que trouver une valeur de learning rate correcte est obligatoire pour avoir un résultat satisfaisant. C'est pourquoi il existe des algorithmes de descente de gradient à pas optimal et plusieurs autres méthodes d'optimisation.

### 5.3.6 L'influence du nombre de neurones de la couche intermédiaire

Les paramètres sont :

- learning rate =  $1e-1$
- nombre d'epochs = 1000
- poids initiaux = 0
- sigmoïde pour tous les neurones
- taille de batch = 1

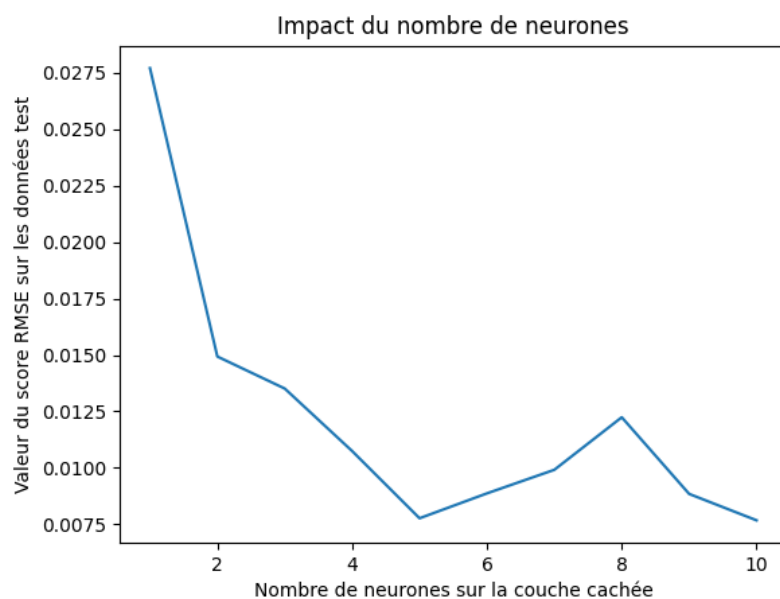


FIGURE 10 – Impact du nombre de neurones sur le score RMSE

Grâce à ces résultats on constate que le nombre de neurones peut effectivement diviser l'erreur par 5. Ainsi, rajouter des neurones sur la couche intermédiaire est très utile. Cependant l'idée de rajouter des neurones sans compter a un impact sur le temps de calcul. En effet, à chaque neurone ajouté, il y a dans notre cas  $784(28*28) + 1$  (le biais) = 785 nouveaux poids. Par exemple pour un seul neurone, le temps de calcul est de 1 seconde sur un ordinateur classique contre 9,6 secondes pour 10 neurones cachées. De plus, on remarque qu'après un certain seuil de nombre de neurones, le score ne s'améliore pas tant que ça. Ainsi il faut trouver un bon compromis entre temps de calcul et nombre de neurones.

### 5.3.7 L'influence de la valeurs des poids initiaux

Les paramètres sont :

- learning rate =  $1e-1$
- nombre d'epochs = 1000
- nombre de neurones sur la couche intermédiaire = 5
- sigmoïde pour tous les neurones
- taille de batch = 1

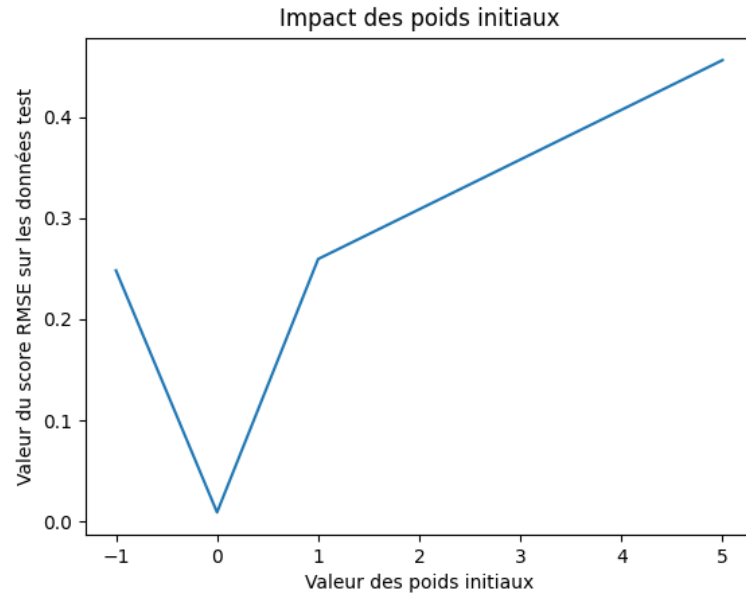


FIGURE 11 – Influence des poids initiaux sur le score RMSE

Sur le graphique, on voit le rôle des poids initiaux. Les poids initiaux correspondent au point initial pour la descente du gradient. Ainsi un mauvais choix de point initial pour faire tomber le réseau de neurones dans un optimal local. De plus il faut ne pas les prendre trop grands. Par exemple lors du test avec les poids initiaux à -5, le code a planté à cause d'un résultat trop grand. En effet, lors du calcul de la propagation avant, le facteur -5 dans l'exponentielle a fait l'équivalent d'une division par 0 et donc une erreur.

### 5.3.8 L'influence de la fonction d'activation de la couche cachée

Les paramètres sont :

- learning rate =  $1e-1$
- nombre d'epochs = 1000
- nombre de neurones sur la couche intermédiaire = 5
- poids initiaux = 0
- taille de batch = 1

Influence de la fonction d'activation de la couche cachée	
Type de fonction d'activation	Score RMSE
Sigmoïde	$9.3e-3$
Relu	$5.4e-1$

On voit que la fonction ReLu est relativement peu efficace. En cherchant l'explication, on apprend que la fonction relu est souvent utilisée pour les couches cachées et lorsque les couches cachées sont nombreuses. Donc dans notre cas, il est cohérent d'avoir un résultat si mauvais avec cette fonction d'activation.

### 5.3.9 L'influence de la taille du batch

Les paramètres sont :

- learning rate =  $1e-1$
- nombre d'épochs = 1000 / taille de batch (afin de toujours garder un budget total = 1000)
- nombre de neurones sur la couche intermédiaire = 5
- poids initiaux = 0

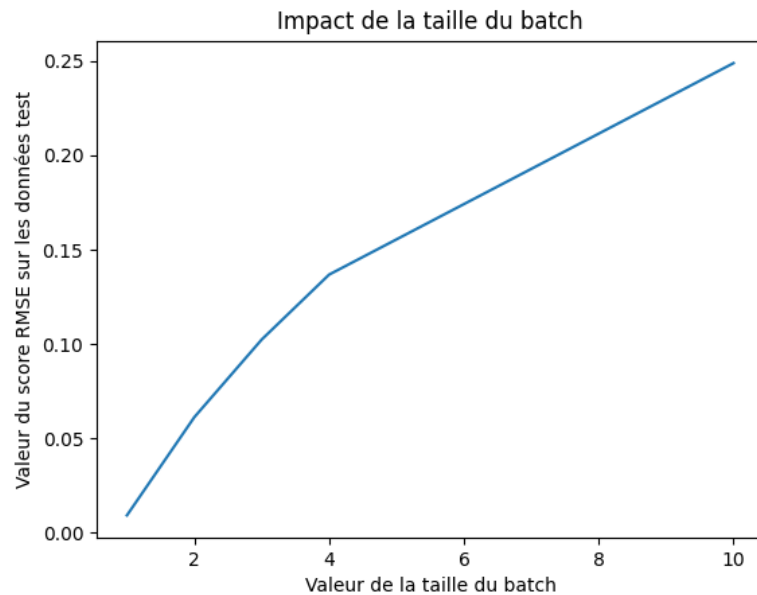


FIGURE 12 – Influence de la taille du batch sur le score RMSE

On remarque que l'augmentation de la taille du batch a un effet négatif sur le score RMSE. Cette remarque n'est pas en accord avec ce qu'on a lu dans certains cours. L'explication est sûrement que l'on a pris un problème très simple et qu'augmenter la taille du batch ne fait que ralentir l'apprentissage du réseau et ne lui apporte rien d'avantageux.

#### 5.3.10 Résultat avec des paramètres corrects

On essaie désormais de prédire la classe des images donc soit 0 soit 1. Pour faire la prédiction on considère qu'une image est un 1 si la sortie du réseau donne une valeur plus grande que 0,5 et 0 si la valeur est inférieure à 0,5. On obtient cette matrice de confusion (la définition d'une matrice de confusion est donnée en annexe 9.4) :

Réalité \ Prédiction	Prédiction	
	0	1
0	1396	1
1	1	1558

Nous obtenons une matrice de confusion quasiment parfaite!

## 5.4 Les réseaux de neurones convolutifs

Les réseaux de neurones convolutifs sont particulièrement préféré pour le traitement d'informations représentable sous forme de matrice, ce qui est exactement le cas d'une image. On suppose donc pouvoir obtenir de meilleurs résultats sur un problème de classification.

### 5.4.1 L'influence des paramètres d'entraînement

Comme dans la partie des RNN simple, nous allons prendre en entrée des images de niveau gris de 28 par 28, et nous allons étudier l'impact des différents paramètres. L'objectif de ce réseau sera de prédire le nombre qui existe dans une image tout en utilisant la base de données de MNIST qui contient des images contenant des nombres allant de 0 à 9. Les paramètres sont :

- le nombre d'epochs
- la valeur du pas du gradient aussi appelé learning rate
- le type de Pooling
- le type de fonction d'activation
- le type de la fonction de perte
- le nombre de couche de convolution
- le nombre de features détectés
- le type de fonction d'optimisation
- la taille du batch sera fixé à 64

L'étude de l'influence des différents paramètres se fait en se basant sur le calcul de l'accuracy ou précision.

Avant de faire test. On doit d'abord comprendre le fonctionnement des couches de convolution dans PyTorch. En effet, lorsque nous définissons une couche de convolution, nous fournissons le nombre de canaux d'entrée, le nombre de canaux de sortie et la taille de noyaux (Kernel). Dans notre cas, la base de données MNIST contient des images de niveaux de Gris, pour cette raison le nombre de canaux d'entrée de la première couche est égale à 1. Ainsi, chaque couche possède un certain nombre de canaux pour détecter des fonctionnalités spécifiques dans les images, et après chaque couche que ce soit de convolution ou de Pooling, on calcule la dimension des images de sortie par la relation suivante :

$$\frac{Input - Kernel + 2padding}{Stride} + 1 \quad (26)$$

L'ensemble des opérations sont bien montrés dans l'image ci-dessous :

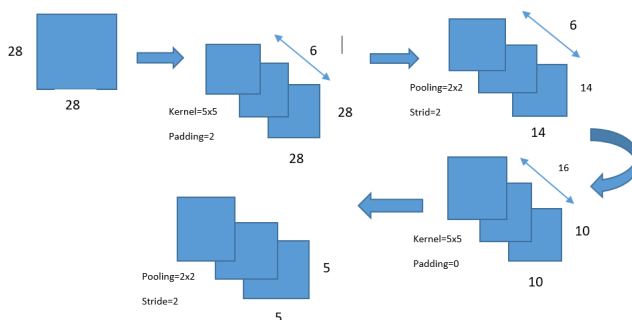


FIGURE 13 – Couche de convolution

### 5.4.2 L'influence du learning rate

Les paramètres sont :

- nombre d'epochs= 45
- fonction de perte= CrossEntropyLoss

— fonction d'optimisation = RMSProp

Ces paramètres sont utilisés dans le modèle suivant :

```
def creat_lenet():
    model=nn.Sequential(
        nn.Conv2d(1,6,5,padding=2),
        nn.LeakyReLU(),
        nn.AvgPool2d(2, stride=2),
        nn.Conv2d(6,16,5,padding=0),
        nn.LeakyReLU(),
        nn.AvgPool2d(2, stride=2),
        nn.Flatten(),
        nn.Linear(400, 120),
        nn.LeakyReLU(),
        nn.Linear(120, 84),
        nn.Tanh(),
        nn.Linear(84, 10)
    )
    return model
```

Impact du learning rate		
Valeur du learning rate	Accuracy	RMSE
10	29.26%	3.559
1	23.85%	4.046
1e-1	10.10%	3.788
1e-2	10.31%	3.788
1e-3	98.87%	0.449
1e-4	97.65%	0.723

On remarque qu'il faut choisir une valeur de learning rate ni trop grande ni trop petite. En effet, Si le Learning Rate est trop grand, alors nous ferons de trop grands pas dans la descente de gradient. Cela a l'avantage de descendre rapidement vers le minimum de la fonction coût, mais nous risquons de louper ce minimum en oscillant autour à l'infini, et s'il est trop faible, alors nous risquons de mettre un temps infini avant de converger vers le minimum de la fonction coût

#### 5.4.3 L'influence du nombre d'epochs

On utilise le même modèle que la partie précédente et avec les mêmes paramètres tout en fixant le learning rate à la valeur 1e-3 et en variant le nombre d'epoch :

Impact du nombre d'epoch		
Valeur d'epoch	Accuracy	RMSE
1	96.93%	0.713
50	98.91%	0.395
70	99.20%	0.376
100	99.25%	0.391

#### 5.4.4 L'influence du type de Pooling

En utilisant le même modèle avec une learning rate= 1e-3 et un nombre d'epoch=45, on compare entre le Maxpooling et le Averagpooling

Impact de type de pooling		
type de pooling	Accuracy	RMSE
Maxpooling	98.97%	0.440
Averagpooling	98.91%	0.449

Le type de pooling n'affecte pas vraiment la performance de notre modèle

#### 5.4.5 L'influence de type de la fonction perte

On teste différents types de fonction de perte, en utilisant le modèle précédent avec un learning rate=1e-3 et un nombre d'épochs égale à 10

Influence du type de la fonction de perte		
type de la fonction de perte	Accuracy	RMSE
CrossEntropyLoss	98.97%	0.449
NLLoss	11.35%	4.499

La fonction d'entropie croisée est exprimée comme suit :

$$l(x, y) = l_1, \dots, l_N^T, l_n = -w_n \log\left(\frac{\exp(x_{n,y_n})}{\sum_{c=1} \exp(x_{n,c})}\right) \quad (27)$$

Où x est l'entrée, y est la cible, w est le poids, C est le nombre de classes et N couvre la dimension du mini-lot. Alors que la fonction NLL est exprimée comme suit :

$$l(x, y) = l_1, \dots, l_N^T, l_n = -w_n x_{n,y_n} \quad (28)$$

On voit bien que la fonction d'entropie croisée est beaucoup plus performante.

#### 5.4.6 L'influence du type de la fonction d'optimisation

On varie cette fois-ci la fonction d'optimisation

Influence du type de la fonction d'optimisation		
type de la fonction d'optimisation	Accuracy	RMSE
Adam	98.79%	0.486
RMSprop	98.87%	0.449
SGD	56.20%	4.498
Adagrad	91.58%	1.196

On voit bien que le type de la fonction d'optimisation joue un rôle très important. Dans notre cas les deux fonctions Adam et RMSprop performe très bien contrairement aux descente de gradient stochastique et l'AdaGrad qui donne de moins bons résultats.

#### 5.4.7 L'influence de type de la fonction d'activation

On teste différents type de fonctions d'activation

Influence du type de la fonction d'activation		
type de la fonction d'activation	Accuracy	RMSE
ReLU	98.65%	0.439
LeakyReLU	98.69%	0.487
Sigmoid	97.87%	0.620
Tanh	98.38%	0.544
LeakyReLU avec ReLU dans la dernière couche	99.08%	0.478
LeakyReLU avec Tanh dans la dernière couche	98.91%	0.449



### 5.4.8 L'influence du type de modèle

Dans cette partie, on va essayer plusieurs modèles dont on va faire varier le nombre de couches de convolution, le nombre des features à extraire dans chaque couche, ainsi qu'on va enlever les couches de Pooling afin de voir leurs influence.

#### Modèle 1 : Sans Pooling

On test un modèle avec deux couches de convolution non suivie des couches de pooling et on trouve une **accuracy** de 98.65% et une **RMSE** de 0.451

```
def creat_lenet():
    model=nn.Sequential(
        nn.Conv2d(1,6,5,padding=2),
        nn.ReLU(),
        nn.Conv2d(6,16,5,padding=0),
        nn.ReLU(),
        nn.Flatten(),
        nn.Linear(9216, 120),
        nn.ReLU(),
        nn.Linear(120, 84),
        nn.ReLU(),
        nn.Linear(84, 10)
    )
    return model
```

#### Modèle 2 : Le nombre de features

Dans ce modèle, on augmente le nombre des features à extraire de 6 à 12 dans la première couche et de 16 à 24 dans la deuxième couches. On trouve une **accuracy** de 98.90% et un **RMSE** de 0.435

```
def creat_lenet():
    model=nn.Sequential(
        nn.Conv2d(1,12,3,padding=2),
        nn.ReLU(),
        nn.MaxPool2d(2, stride=2),
        nn.Conv2d(12,24,3,padding=0),
        nn.ReLU(),
        nn.MaxPool2d(2, stride=1),
        nn.Conv2d(24,32,3,padding=2),
        nn.ReLU(),
        nn.MaxPool2d(2, stride=2),
        nn.Flatten(),
        nn.Linear(1568, 120),
        nn.ReLU(),
        nn.Linear(120, 84),
        nn.ReLU(),
        nn.Linear(84, 10)
    )
    return model
```

#### Modèle 3 : Un seul couche de convolution

Dans ce modèle, on diminue le nombre des couches de convolution à une seule couche. On trouve une accuracy de 98.58% et un **RMSE** de 0.454

```
def creat_lenet():
```

```

model=nn.Sequential(
    nn.Conv2d(1,20,5,padding=2),
    nn.ReLU(),
    nn.MaxPool2d(2, stride=2),
    nn.Flatten(),
    nn.Linear(3920, 120),
    nn.ReLU(),
    nn.Linear(120, 84),
    nn.ReLU(),
    nn.Linear(84, 10)
)
return model

```

#### 5.4.9 Comparaison avec un réseau non-convolutif

Afin de comparer convenablement la classification par réseau de neurones convolutif et par réseau de neurones simple, on refait un code afin de faire une classification à 10 classes. Ce sera exactement le même que pour le réseau simple. La seule différence c'est qu'au lieu d'avoir une unique sortie, il en aura 10 ce qui complique considérablement le code. Chaque sortie sera la probabilité d'être le numéro de sortie associée. Par exemple, la sortie numéro 3 donnera la probabilité que l'image en entrée soit un 3. Pour passer de cette sortie à 10 composantes à une sortie qui donne un chiffre entre 0 et 9, il suffit de considérer la composante qui a la plus haute probabilité.

En résultat, nous obtenons des choses étonnantes. Après plusieurs essais avec plusieurs paramètres différents, on se rends compte que la fonction de perte converge effectivement vers 0, et que lorsqu'on test sur les données test on obtient un score RMSE correct sans pour autant être aussi bon que dans la classification à 2 classes. Cependant, lorsqu'on applique ce réseau de neurones aux données tests et qu'on regarde la matrice de confusion on se rend compte que le réseau ne classifie pas du correctement. Voici un exemple de matrice de confusion obtenu pour un score RMSE de 9.1e-2 :

Prédiction \ Réalité	0	1	2	3	4	5	6	7	8	9
0	0	1	1	2	0	1	2	16	0	1
1	1347	1560	1306	1361	1268	1174	1344	1362	1362	1359
2	1	21	12	12	11	14	3	74	2	20
3	3	0	6	4	0	2	1	1	0	1
4	0	0	0	0	0	0	0	0	0	0
5	21	12	46	44	32	66	10	15	1	10
6	0	0	0	0	0	0	0	0	0	0
7	8	31	7	2	8	4	6	11	1	9
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

Ce problème peut s'expliquer de plusieurs manières. Tout d'abord, étant donné que la fonction de perte converge mais que le résultat n'est clairement pas cohérent, le problème vient sûrement d'elle. En effet nous avons utilisé la fonction de perte RMSE. Il serait intéressant d'essayer avec une fonction de perte plus orientée vers la classification à plus de 2 classes. Ensuite, un second problème est le qu'on ne laisse peut-être pas assez le temps à la fonction de converger. Le résultat précédent a été obtenu avec 50000 epochs ce qui bien plus que lors des précédents tests. Il pourrait être judicieux de tester sur encore plus de budget cependant nous sommes limités avec nos ordinateurs. Ensuite une dernière possibilité que nous n'avons pas trouvé la bonne combinaison de paramètres.

## 6 Les fondamentaux des GAN

Pour rappel, un GAN est l'ensemble de deux réseaux de neurones, le générateur et le discriminateur. Regardons de plus près chacun de ces deux réseaux.

### 6.1 Le discriminateur

Pour rappel, le discriminateur a le rôle de classifier les images qui lui sont passées en entrée avec une “probabilité” en sortie que celles-ci soient réelles. Le discriminateur va prendre un vecteur de taille  $n$  (où  $n$  est la dimension d'un élément à reproduire) pour en donner la probabilité qu'il soit réel. On définit la fonction  $D$  de cette manière :

$$D : \mathbb{R}^n \rightarrow [0; 1]$$

Dans le cas d'un GAN simple, autrement appelé Vanilla GAN, le discriminateur est principalement constitué de trois types de couches. Une couche linéaire réduisant la taille de l'image pour qu'elle soit plus facile à traiter, suivis d'une couche d'activation ReLU, et enfin des filtres qui extraient certaines propriétés de l'image. Le discriminateur applique des couches comme celles-ci plusieurs fois pour finir avec la fonction sigmoid qui transforme la sortie en une valeur entre 0 et 1, qu'on considère comme la “probabilité” que l'image venant d'être traitée soit réelle (dispose de fortes similitudes avec les images d'entraînement). Le label prédit dépend très fortement du choix des couches de neurones, en particulier les filtres qui sélectionnent certaines informations de l'image en entrée.

Un GAN simple pourrait fonctionner pour générer certaines images, cependant on peut constater que la plupart des chercheurs manipulant les GAN sont plutôt d'avis à utiliser un GAN à convolution profonde (DCGAN) qui est beaucoup plus robuste que le GAN simple. Dans le cas d'un GAN à convolution, au lieu des couches linéaires, on dispose de couches de convolution. En théorie, c'est la seule différence entre ces GAN, cependant, en pratique, on retire également les filtres car la convolution permet également de filtrer l'image selon ses paramètres, en particulier la stride. On peut par exemple implémenter les couches de convolutions avec une stride (le pas de déplacement du kernel) de 2 et un zero-padding de 1, ce qui se comportera comme un filtre. L'avantage de cette approche est de pouvoir faire apprendre au réseau ses propres paramètres pour le filtre au lieu de les définir arbitrairement au préalable. L'opération de convolution nécessite ensuite l'intervention d'une couche de normalisation. Notons également qu'on utilise l'activation par leaky ReLU assez fréquemment, la différence par rapport au ReLU standard étant que la fonction possède une faible pente dans les valeurs négatives au lieu d'être nulle, ce qui permet de tenir compte de ses valeurs qui seront annulées par ReLU standard.

### 6.2 Le générateur

Le générateur peut être vu comme le processus inverse du discriminateur. Dès lors que le discriminateur réduit la taille de l'information, le générateur l'agrandit. C'est du upsampling, qui peut être linéaire pour un vanilla GAN, ou des couches de convolutions transposées dans un DCGAN. Ces couches vont transformer le vecteur bruit en entrée pour atteindre la taille de l'image à générer. Le GAN part d'une densité  $\gamma$  définie sur  $\mathbb{R}^d$  où  $d \leq n$ . Habituellement  $\gamma \sim N(0, I_d)$ . Nous allons donc obtenir plusieurs vecteurs de  $\mathbb{R}^d$  et le générateur va modifier ces vecteurs pour les faire passer d'une distribution  $\gamma$  à une distribution  $\mu$  où  $\mu$  est la distribution des données initiales notées  $\chi$ . On définit donc la fonction  $G$  de cette manière :

$$G : \mathbb{R}^d \rightarrow \mathbb{R}^n$$

Similairement au discriminateur, le générateur dispose également de couches de normalisation et d'activation ReLU. La dernière opération appliquée à l'information est une fonction de tangente hyperbolique,

ce qui génère donc des valeurs entre -1 et 1.

### 6.3 Une étape d'entraînement

Une étape d'entraînement peut être différente d'un code à l'autre. Par exemple, pour entraîner un DCGAN, il est conseillé d'entraîner le discriminateur une fois sur un batch constitué uniquement d'images réelles, et d'un batch avec seulement des images générées. Un batch est un ensemble de plusieurs images, car on entraîne toujours le discriminateur avec plusieurs images à la fois. La perte pour le discriminateur est calculée sur chacune des deux étapes, et est ensuite rétropropagée. Pour un cycle d'entraînement du générateur, celui-ci génère un batch d'images qui est ensuite passé au discriminateur, dont la sortie est utilisée pour calculer la perte. Cette perte est également rétropropagée par la suite.

### 6.4 La fonction de perte et algorithme d'optimisation

La fonction de perte est une fonction quantifiant l'écart entre la sortie du discriminateur, et les labels des images en entrée. Les labels valent soit 0 pour les images générées, soit 1 pour les images réelles. Le discriminateur a donc pour objectif de prédire correctement ces labels.

Pour le générateur, il n'y a pas de fonction de perte à proprement parler ; les images qu'il génère ne sont pas directement employés pour le calcul d'une perte. En quelque sorte, la fonction de perte pour le générateur peut être considérée comme étant  $C \circ D$  avec  $C$  une fonction de perte simple et  $D$  le discriminateur. Notons d'ailleurs qu'il est possible que la fonction de perte utilisée pour la perte du générateur peut être différente de celle du discriminateur, tout comme l'algorithme d'optimisation peut être différent pour les deux.

De manière générale, on prend l'algorithme d'Adam pour entraîner les GAN. L'algorithme d'Adam est en fait une combinaison de deux autres algorithmes très connus, AdaGrad et RMSProp. AdaGrad est un algorithme avec gradient adaptatif qui consiste à modifier le pas selon l'amplitude de la valeur du gradient, ce qui présente des avantages pour des problèmes computer vision. RMSProp prend en compte les variations récentes des poids ce qui permet un apprentissage unique par neurone individuel. Un autre avantage de cet algorithme est sa vitesse comparé aux autres algorithmes employés fréquemment. De plus, les exemples d'implémentation montrent que cet algorithme converge de manière plus stable, ce qui en fait donc un choix très naturel.

### 6.5 Choix des fonctions de perte appliqué aux GAN

Tandis qu'on utilise seulement quelques algorithmes présélectionnés pour d'optimisation, les choix de fonction de pertes sont multiples.

#### 6.5.1 La fonction minimax

La fonction minimax fait référence à l'optimisation simultanée minimax des modèles de discriminateur et de générateur. Elle fait référence notamment à une stratégie d'optimisation dans les jeux au tour par tour à deux joueurs pour minimiser la perte ou le coût pour le pire des cas de l'autre joueur.

Le discriminateur cherche à maximiser la probabilité attribuée aux images réelles et fausses, toute en maximisant la moyenne du log de probabilité pour les images réelles et le log des probabilités inversées des fausses images :

$$\max(\log D(x) + \log(1 - D(G(z)))) \quad (29)$$

Pourtant, le générateur apprend à générer des échantillons qui ont une faible probabilité d'être faux, toute en minimisant le logarithme de la probabilité inverse prédite par le discriminateur pour les fausses images :

$$\min(\log(1 - D(G(z)))) \quad (30)$$

### 6.5.2 La fonction non saturante

La perte GAN non saturante est une modification de la perte du générateur pour surmonter le problème de saturation. Il s'agit d'un changement subtil qui implique que le générateur maximise le logarithme des probabilités de discriminateur pour les images générées au lieu de minimiser le logarithme des probabilités de discriminateur inversé pour les images générées :

$$\max(\log(D(G(z)))) \quad (31)$$

et du coup le générateur cherche à maximiser la probabilité que des images soient prédites comme réelles au lieu de minimiser la probabilité que des images soient prédites comme fausses.

### 6.5.3 La fonction des moindres carrés

Cette fonction de perte est une approche alternative. Son avantage est qu'elle pénalise davantage les erreurs plus importantes, ce qui entraîne à son tour une correction importante plutôt qu'un gradient de fuite et aucune mise à jour du modèle.

Le discriminateur cherche à minimiser la somme des différences au carré entre les valeurs prédites et attendues pour les images réelles et fausses :

$$\min((D(x) - 1)^2 + (D(G(z)))^2) \quad (32)$$

Le générateur cherche à minimiser la somme des différences au carré entre les valeurs prédites et attendues comme si les images générées étaient réelles :

$$\min((D(G(z)) - 1)^2) \quad (33)$$

## 6.6 Implémentation d'un GAN

Avant de se lancer dans le cœur du problème, étudions d'abord le comportement d'un GAN sur des exemples proposés en ligne. On choisit la base de données du MNIST par raison de simplicité, mais aussi de similitude par rapport à nos images à générer.

### 6.6.1 Une première implémentation

On élabore une première implémentation du GAN. On choisit les paramètres de références/initiaux suivants : - Un générateur à 3 couches 'LeakyRelu' ayant respectivement 256, 512, et 1024 neurones et une sortie 'tanh'. - Un discriminateur à 2 couches de réseaux de neurones 'ReLU' de 512 et 256 neurones et une sortie de type 'Sigmoid'. - On gardera une fonction de perte de type Binary Cross Entropy. - On choisit un optimiseur de type Adam.

Nous allons modifier ces paramètres et comparer les performances des réseaux sur deux critères : - la vitesse de convergence de la fonction de perte. - la qualité des images obtenues après convergence.

Pour ce premier modèle, nous obtenons le résultat suivant.

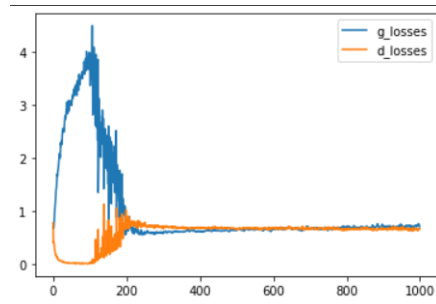


FIGURE 14 – Convergence de référence

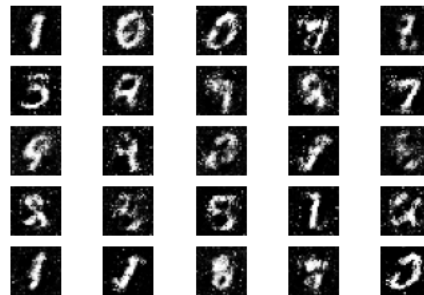


FIGURE 15 – Qualités de l'image générée en 2000 époques avec le modèle de référence

## 6.6.2 Modification des hyperparamètres du modèle GAN

### Modification de l'optimiseur

On remplace l'optimiseur Adam par un optimiseur de type SGD (Stochastic Gradient Descent).

On obtient que les fonctions de perte du générateur et du discriminateur ne convergent pas. L'image générée reste alors un bruit.

### Suppression de la dernière couche du générateur

A partir du modèle de générateur, on supprime la dernière couche du générateur.

On obtient le résultat suivant :

On observe que les fonctions de pertes ne convergent pas totalement. Cependant, le modèle sans la dernière couche du générateur semble générer des images de qualité légèrement plus rapidement que le modèle initial.

### Suppression de la dernière couche du générateur et de la première couche du discriminateur

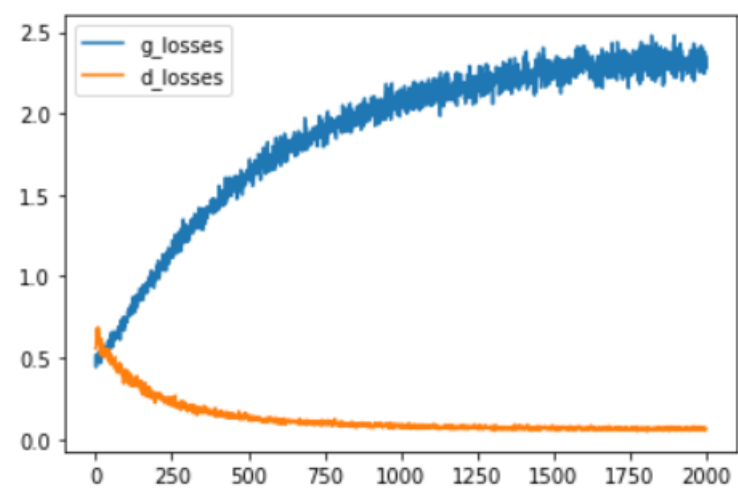


FIGURE 16 – Convergence SGD

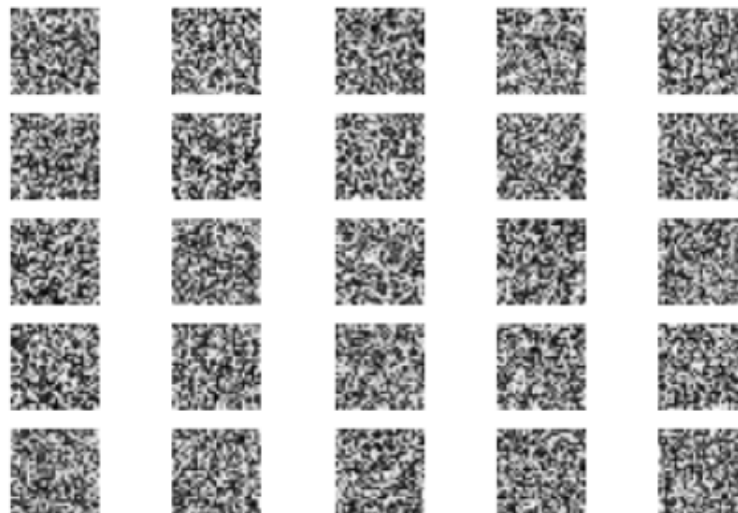


FIGURE 17 – Qualité des images générées par SGD pour 4000 époques

A présent, nous allons supprimer la dernière couche du générateur et la première couche du discriminateur du modèle de référence.

Pour ce modèle, on observe que les fonctions de perte semblent converger, malgré la présence d'un bruit. On a alors une convergence de moins bonne qualité. Les images générées par le modèle semblent quant à elles avoir une qualité similaire au modèle de référence à nombre d'époques égaux.

### Modification du nombre de neurones des couches du générateur

Le générateur de notre modèle de référence est constitué de 3 couches de neurones possédant respectivement 256, 512, et 1024 neurones. Nous avons fait varier ces différentes valeurs une à une et avons comparé le résultat obtenu avec le modèle de référence.

Nous avons obtenu quasiment les mêmes performances que le modèle de référence pour les variations

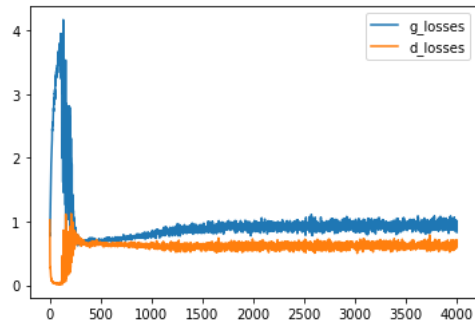


FIGURE 18 – Convergence sans la dernière couche du générateur

Pour 2000 époques sans dernière couche	Pour 2000 époques initialement	Pour 3800 époques sans dernière couche	3800 époques initialement

FIGURE 19 – Comparaison de la qualité des images du modèle de référence avec le générateur sans dernière couche

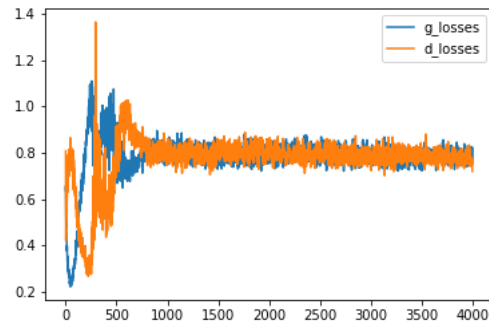


FIGURE 20 – Convergence sans la dernière couche du générateur et la première couche du discriminateur

2000 époques initialement	2000 époques après suppression des 2 couches	3800 époques initialement	3800 époques après suppression des deux couches

FIGURE 21 – Comparaison de la qualité des images du modèle de référence avec le modèle sans sans dernière couche du générateur et sans première couche du discriminateur



suivantes : - Diminution du nombre de neurones de la première couche (20 neurones) - Augmentation du nombre de neurones de la première couche (700 neurones) - Diminution du nombre de neurones de la deuxième couche (300 neurones) - Augmentation du nombre de neurones de la deuxième couche (800 neurones) - Augmentation du nombre de neurones de la dernière couche (1600 neurones)

La seule variation impliquant des résultats différents du modèle de référence est la diminution du nombre de neurones de la dernière couche à 500 neurones. Voici les résultats obtenus.

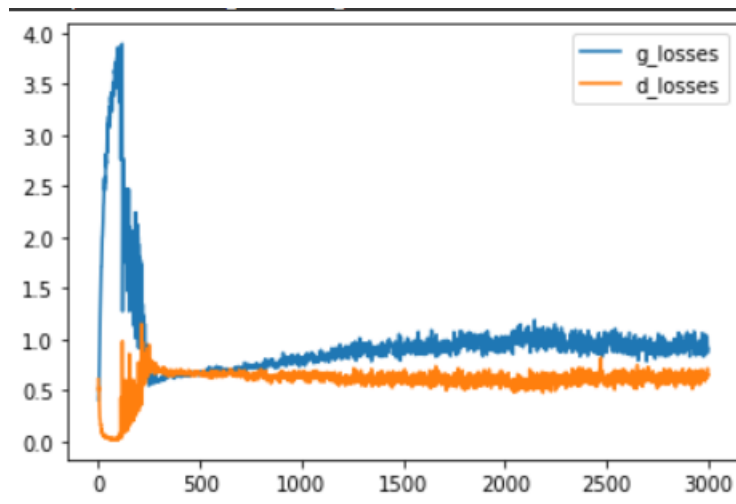


FIGURE 22 – Convergence en diminuant la dernière couche du générateur

2000 époques sur le modèle de référence	2000 époques en diminuant la dernière couche à 500 neurones

FIGURE 23 – Comparaison de la qualité des images du modèle de référence avec le modèle pour lequel la dernière couche du générateur est diminuée

On observe que les deux fonctions de perte ne semblent pas adjacentes, on a donc une moins bonne convergence des fonctions de perte. On observe cependant que les images du nouveau modèle sont presque d'aussi bonne qualité que le modèle de référence.

### Modification du nombre de neurones des couches du discriminateur

Le discriminateur de notre modèle de référence est constitué de 2 couches de neurones possédant respectivement 512 et 256 neurones. A l'instar des tests précédents, nous avons fait varier ces différentes valeurs une à une et avons comparé le résultat obtenu avec le modèle de référence.

Nous avons obtenu quasiment les mêmes performances que le modèle de référence pour les variations suivantes : - Augmentation du nombre de neurones de la première couche (800 neurones) - Diminution du nombre

de neurones de la deuxième couche (100 neurones) - Augmentation du nombre de neurones de la deuxième couche (550 neurones)

La seule variation impliquant des résultats différents du modèle de référence est la diminution du nombre de neurones de la première couche à 200 neurones. Voici les résultats obtenus.

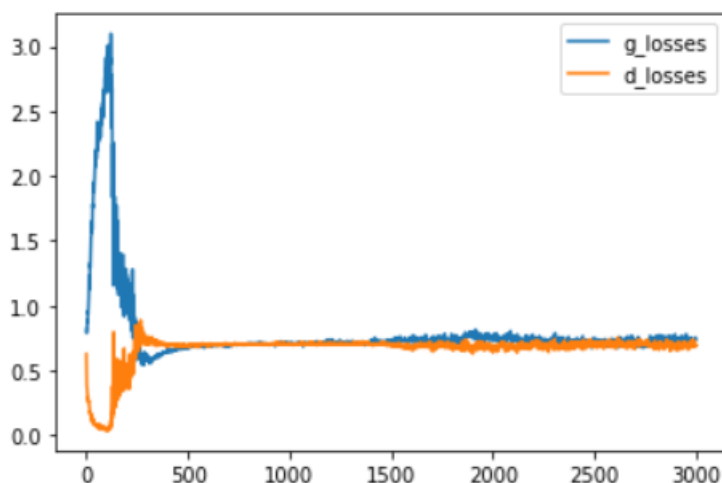


FIGURE 24 – Convergence en diminuant la première couche du discriminateur

2000 époques initialement	2000 époques en diminuant la première couche du discriminateur

FIGURE 25 – Comparaison de la qualité des images du modèle de référence avec le modèle pour lequel la première couche du discriminateur est diminuée

### Modification du type des fonctions d'activation

Le modèle de référence utilise des fonction LeakyReLU. Nous avons décidé d'évaluer le modèle en remplaçant les LeakyReLU par des ReLU, des Tangentes hyperboliques Tanh et des Sigmoides.

Voici le résultat obtenu pour les fonctions d'activation ReLU.

On observe des performances similaires au modèle de référence, quoique les fonctions de pertes ne sont pas parfaitement adjacentes et que les images générées sont de qualité légèrement moins bonne.

Voici le résultat obtenu pour les fonctions d'activation Tanh.

On observe que convergence est beaucoup moins rapide pour les fonctions d'activation Tanh. En 2000 époques, les images générées sont tout juste légèrement de moins bonne qualité que le modèle de référence.

Voici le résultat obtenu pour les fonctions d'activation Sigmoid.

On observe une convergence très rapide du modèle en 250 époques. En 2000 époques, les images semblent de qualité similaire au modèle LeakyReLU de référence. Cependant vers 2400 époques, les fonctions de pertes semblent exploser avant de converger de nouveau. Ce phénomène s'observe également sur la banque d'image

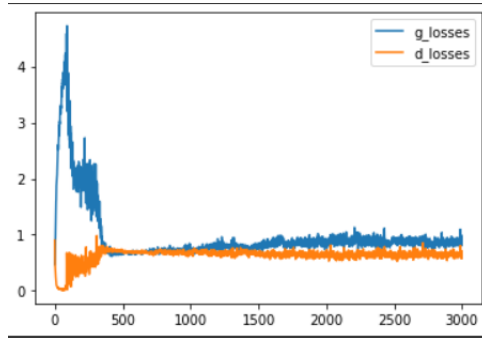


FIGURE 26 – Convergence pour les fonctions d'activation ReLU

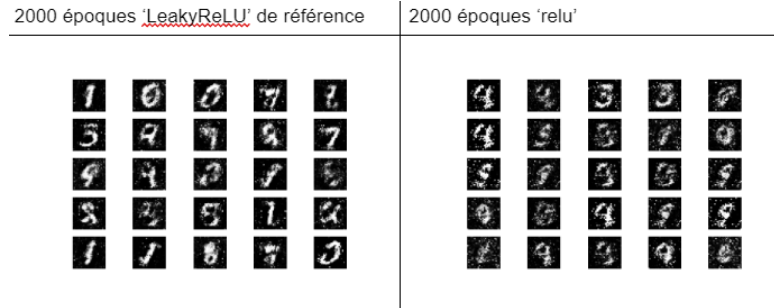


FIGURE 27 – Comparaison de la qualité des images du modèle de référence LeakyReLU avec le modèle ReLU

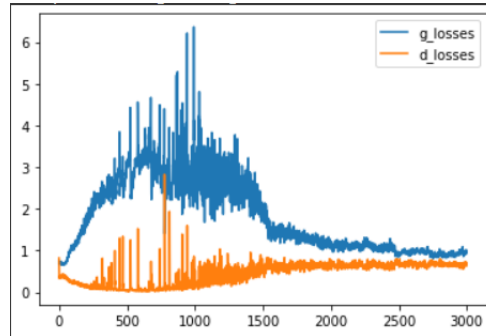


FIGURE 28 – Convergence pour les fonctions d'activation Tanh

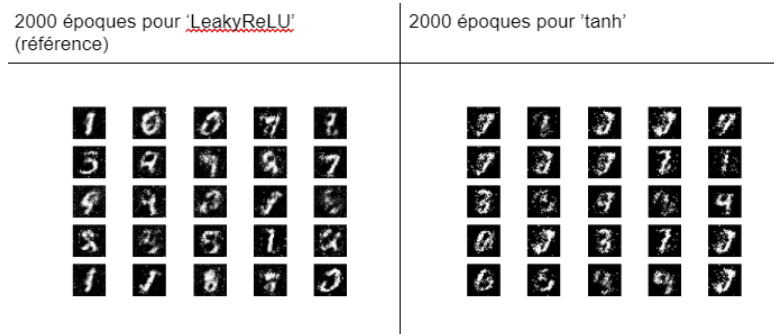


FIGURE 29 – Comparaison de la qualité des images du modèle de référence LeakyReLU avec le modèle Tanh

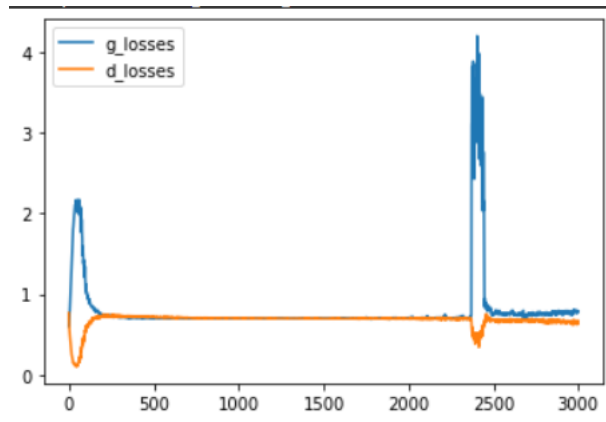


FIGURE 30 – Convergence pour les fonctions d’activation Sigmoid




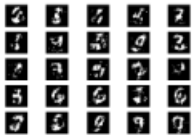
2000 époques Leaky ReLU de référence	2000 époques Sigmoid	2400 époques Sigmoid	2800 époques Sigmoid
			

FIGURE 31 – Comparaison de la qualité des images du modèle de référence LeakyReLU avec le modèle Sigmoid et affichage des époques suivantes

pour lequel une partie des images semblent inverser le noir et le blanc vers 2400 avant de retourner vers des images de bonnes qualités vers 2800 époques.

## 6.7 Implémentation d’un DCGAN

Le DCGAN modifie non seulement le discriminateur avec des couches de convolution, mais aussi le générateur qui appliquera une opération de convolution transposée (opération assimilable à l’inverse de la convolution).

En termes de paramètres, on constate que l’impact de ceux-ci sur le réseau de neurones est très similaire au GAN classique. Focalisons-nous donc à la comparaison entre ces deux réseaux. La plus grande différence se déduit de l’observation des images générées après un certain nombre d’époques.

La figure 32 nous apporte deux informations. Tout d’abord, le DCGAN demande moins d’époques pour atteindre des résultats satisfaisants. De plus, les images générées sont moins floues pour le DCGAN. Cette différence peut s’expliquer par le fonctionnement de la convolution. La convolution permet d’extraire des caractéristiques d’une image. Ceci permet de raffiner les détails tandis que le GAN classique n’est capable que de compter sur une image globale. Cependant, nous devons noter que le DCGAN prend environ 7 à 8 fois plus de temps par époque que le GAN classique. Dans le cadre d’une démonstration comme notre cas avec des images à faible résolution, la différence n’est pas gênante, en revanche, si l’on souhaite générer une image haute définition, il est évident qu’utiliser le réseau à convolution est un choix meilleur.

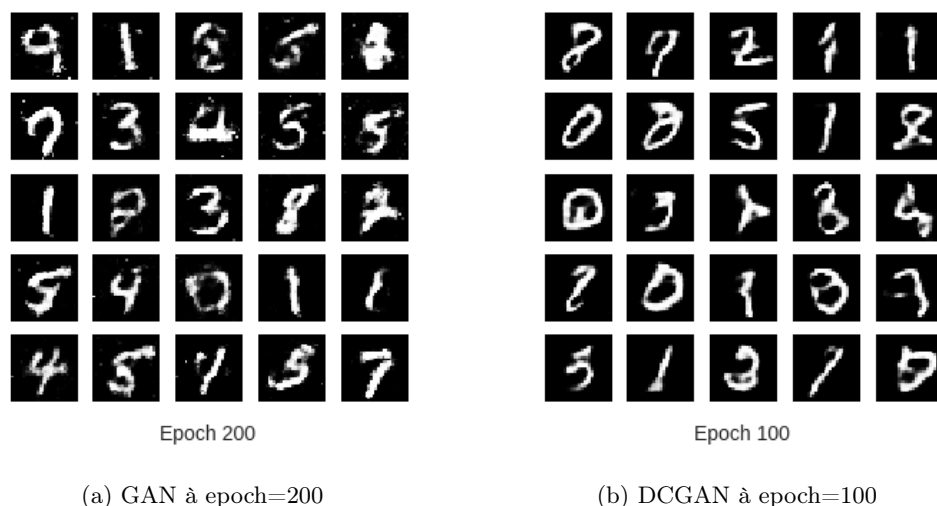


FIGURE 32 – Comparaison de la qualité des images générées

## 7 Les GAN appliqués à notre problème

Maintenant qu'on dispose des connaissances du fonctionnement des réseaux de neurones et des GAN, il faut choisir le modèle qu'on souhaite entraîner.

### 7.1 La structure du GAN

Tout d'abord, il faut noter que pour décider de la structure d'un réseau de neurones, c'est-à-dire le type de couches cachées, leur nombre et le nombre de neurones par couche, il n'existe pas de méthode purement analytique et déterministe. Toutes ces options se présentent comme des hyperparamètres du problème, ce qui signifie qu'ils sont choisis par un humain et c'est à lui d'apprendre à trouver les paramètres qui produisent les meilleurs résultats. L'approche la plus sage est de bien maîtriser le problème qui est traité et la spécificité des données afin de partir d'un modèle qui présente déjà un comportement souhaitable. Par exemple, si on souhaite différencier des images sur des petites caractéristiques d'une image, on choisirait des filtres plus petits que si on souhaitait étudier des éléments plus grands. Autrement dit, il existe des conseils mais pas de règles.

### 7.2 Le choix de la fonction de perte

Dans un cas général, on pourrait tout simplement employer les fonctions de pertes de classification classiques afin de faire converger notre modèle. Cependant, notre problème comporte d'autres dimensions qu'une simple ressemblance des images : les aspects physiques. Ainsi il nous est indispensable de d'abord identifier les paramètres à suivre sur les images, et d'en créer une fonction de perte globale permettant de tenir compte des caractéristiques physiques du matériau. Notons également que notre fonction de perte ne pourra pas être conventionnelle puisque les effets des caractéristiques physiques devront être directement calculées à partir des images.

En premier temps, il serait sage de trouver ou forger une fonction de perte de classification binaire qui soit bien adaptée à notre problème, et ensuite la retravailler pour modifier la trajectoire de l'entraînement.

## 7.3 Les premiers essais avec un DCGAN

Il a été montré dans la section précédente que les GAN à convolution performant généralement mieux pour l'entraînement avec des images. La première approche est alors de reprendre ce code fonctionnel et de simplement remplacer les paramètres de la base de données et la taille d'image pour observer le comportement du réseau.

Cependant, les premiers essais n'ont pas abouti. La raison étant simplement que les paramètres n'étaient pas adaptés pour notre cas. Il a donc été nécessaire de modifier plusieurs paramètres, en particulier les paramètres d'optimisation afin d'obtenir quelque chose qui n'est pas un bruit. Malheureusement, c'est à ce moment que nous avons heurté un problème récurrent chez les GAN, le mode collapse. Nous avons cherché plusieurs solutions à ce problème, en utilisant [ce lien](#) qui propose plusieurs pistes pour résoudre ce problème. Cependant, il s'est avéré être impossible de résoudre notre problème sans grand changement à notre modèle. Ainsi, il a été temps de modifier la variante de GAN à nouveau.

## 7.4 Le GAN de Wasserstein

Le GAN de Wasserstein ou WGAN est une extension du GAN qui implémente une manière plus efficace d'entraîner le modèle de générateur pour mieux se rapprocher de la distribution des données réelles.

### 7.4.1 Introduction au GAN de Wasserstein

Pour comprendre le fonctionnement de cette variante, il faut introduire une première notion : la distance de Wasserstein. Pour une distribution réelle de données  $\mathbb{P}_r$  et une distribution générée de données  $\mathbb{P}_g$ , la distance de Wasserstein [5] est définie comme la borne inférieure de tous les chemins de transports :

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|] \quad (34)$$

avec  $\Pi(\mathbb{P}_r, \mathbb{P}_g)$  désignant l'ensemble des distributions conjointes  $\gamma(x, y)$  donc  $\mathbb{P}_r$  et  $\mathbb{P}_g$  en sont les lois marginales.

Cette nouvelle distance nous permettra d'étudier la perte d'une manière différente. Jusqu'à présent, les fonctions de pertes reposaient sur un principe fondamental qui est la distance de Kullback-Leibler, ou une version plus sophistiquée de celle-ci, la distance de Jensen-Shannon (voir annexe mathématique). La définition de la distance influe directement sur la fonction de perte et le gradient. Une courte réflexion mathématique montre que dans le cas classique, le gradient devient quasi-nul si le générateur produit des distributions très éloignées, ce qui l'empêche de s'améliorer par la suite. En revanche, la distance de Wasserstein ne présente pas ce défaut, et au contraire, l'amplitude du gradient augmente également si on s'éloigne trop de la distribution réelle.

La distance de Wasserstein est également appelée distance de Kantorovich-Rubinstein pour avoir été démontrée pouvoir prendre la forme suivante :

$$W(\mathbb{P}_r, \mathbb{P}_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r} [f(x)] - \mathbb{E}_{x \sim \mathbb{P}_g} [f(x)] \quad (35)$$

Le plus grand intérêt de cette forme est de démontrer que le problème revient à trouver une fonction 1-Lipschitzienne, qu'on peut déterminer à travers un réseau de neurones. Ceci peut se faire par une simple modification du discriminateur : on retire la sigmoïde de la dernière couche pour obtenir un score scalaire au lieu d'une probabilité. Notons que cette modification induit une modification du vocabulaire. On désignera désormais le discriminateur par le terme "critique".

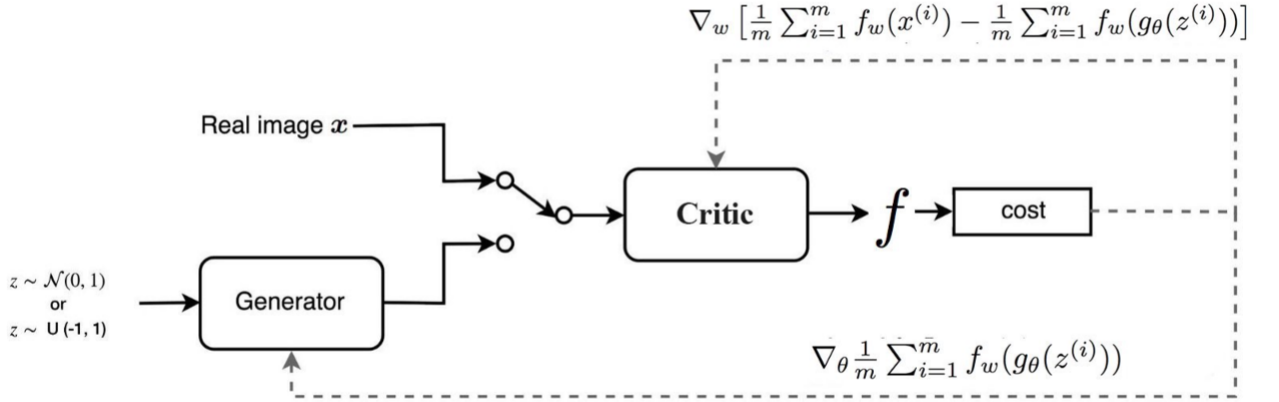


FIGURE 33 – Schéma du GAN de Wasserstein

Le schéma 33 représente le nouveau principe d'entraînement du GAN. Cependant, ce schéma n'indique pas toutes les subtilités, en particulier la condition que  $f$  soit 1-Lipschitzienne. Dans la publication originale du GAN de Wasserstein [1], on force cette condition par weight-clipping, c'est-à-dire, en restreignant les valeurs que peuvent prendre les poids à un intervalle choisis arbitrairement. Mais il a été signalé plus tard que le weight-clipping n'est pas une méthode efficace [4].

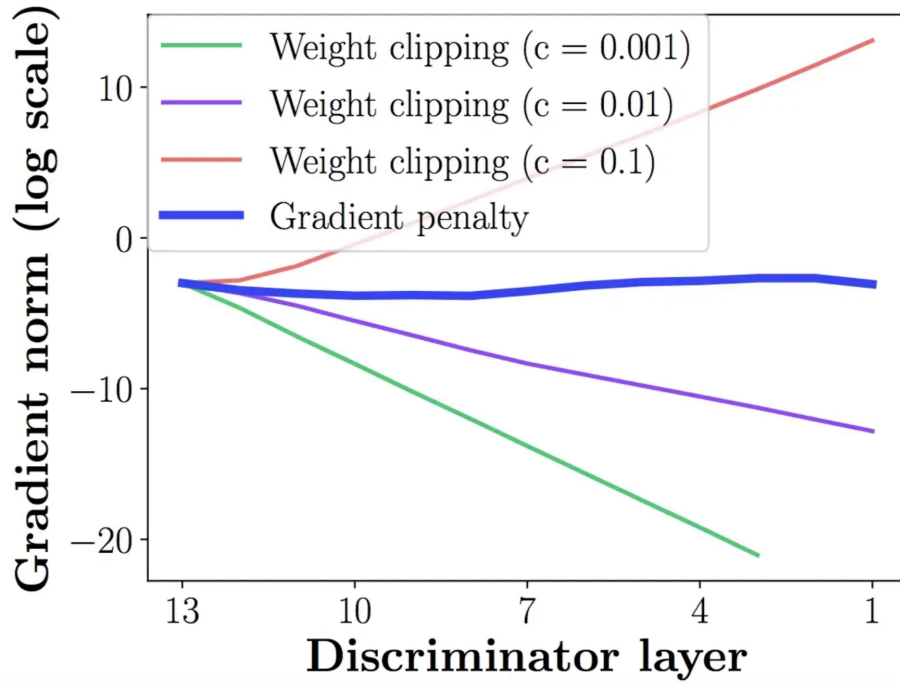


FIGURE 34 – Inefficacité du weight clipping

On peut constater sur le schéma 34 qu'il est très facile de se retrouver dans une situation de gradient quasi-nul ou infini en faisant varier les bornes du weight clipping. Une alternative à cette méthode est donc le gradient penalty, ou correction du gradient. On prend en compte cette correction dans la fonction de perte comme suit :

$$C(\mathbb{P}_r, \mathbb{P}_g) = \underbrace{\mathbb{E}_{\tilde{x} \sim \mathbb{P}_g} [D(\tilde{x})] - \mathbb{E}_{x \sim \mathbb{P}_r} [D(x)]}_{\text{Perte initiale}} + \lambda \underbrace{\mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]}_{\text{Correction du gradient}} \quad (36)$$

avec  $\lambda$  un coefficient choisis arbitrairement, mais fixé à 10 par choix empirique.

Il a été constaté que le GAN de Wasserstein est plus lent en général par rapport au GAN standard ou GAN à convolution. Cependant, il présente un réel avantage. De tous les essais jusqu'à présent, il n'y a jamais eu de signe de mode collapse, ce qui était un de nos problèmes observés. De plus, le générateur peut apprendre quel que soit sa performance.

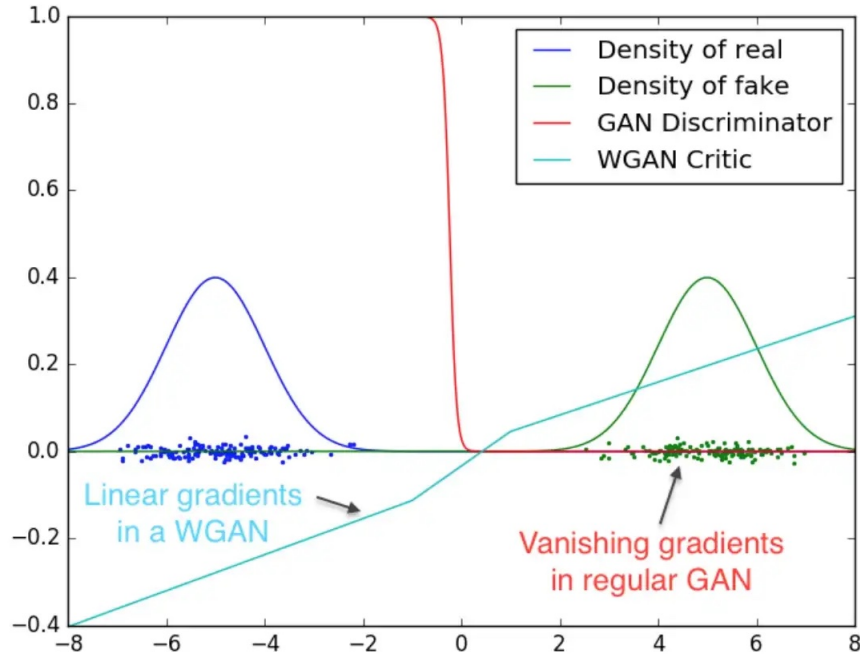


FIGURE 35 – Discriminateur et critique optimal lors de l'apprentissage de la distribution de deux gaussiennes

On constate effectivement sur la figure 35 que, lorsque la distribution du générateur est très éloignée de la distribution réelle, on se retrouve avec un gradient quasi-nul pour le discriminateur. En revanche, le critique utilise efficacement cette information avec, au contraire, un gradient de grande magnitude pour réorienter la direction d'évolution pour la rétropropagation.

#### 7.4.2 Implémentation du GAN de Wasserstein

Cette version du GAN a donc été implémentée, dont le code est accessible sur [GitLab](#). Le résultat en figure 36 est un batch généré à la centième époque après environ 50 minutes d'entraînement sur une RTX 3060, sachant que le processus aurait pu être plus rapide s'il n'était pas limité par la vitesse du processeur.

Toutes les images ne sont pas parfaites, mais certaines sont très convaincantes. Il a été difficile de rendre le GAN capable de générer uniquement des images valides. Nous avons essayé d'intégrer la transformée de Hough pour utiliser le critère d'intersection des cercles détectés comme une nouvelle mesure pour mettre à jour le réseau. Ceci revient à calculer un terme de la perte du générateur à sa sortie, et un second à la sortie du



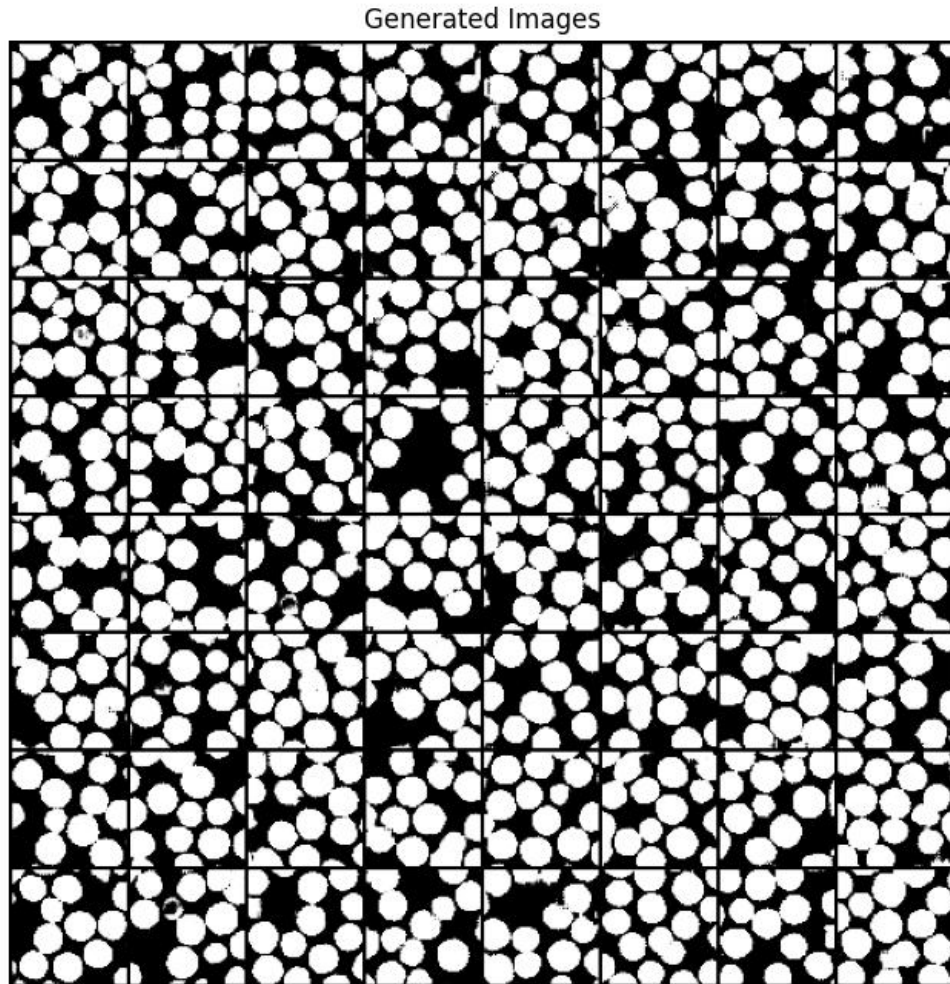


FIGURE 36 – Exemple de sortie du GAN de Wasserstein

discriminateur. Par simplicité, ce terme n'intervient qu'à partir d'une certaine époque où chercher des cercles prend sens. Mais par difficulté de bien équilibrer cette valeur et le manque de temps pour faire des essais, cette tentative a été mise de côté.

## 7.5 Autres pistes

Tenant compte des résultats que l'on peut obtenir à partir du GAN de Wasserstein, il semblerait également pertinent de prendre une autre approche, par exemple, utiliser la transformée de Hough non pas dans l'apprentissage mais pour valider les images générées lors de l'utilisation. Certaines images venant du générateur sont déjà quasiment parfaites. Il suffirait alors de seulement garder ces images pour les calculs qui suivent. On pourrait d'ailleurs rajouter d'autres critères programmables pour sélectionner de manière plus stricte les bonnes

images.

Autrement, il serait aussi sage d'explorer d'autres types de réseaux de neurones si le résultat du GAN est insatisfaisant. Il existe par exemple les VAE (Variational Auto-Encoder) qui sont des réseaux très puissants pour créer des variations d'images. Il existe également une sous-famille de GAN qui combine ces deux concepts en un VAE-GAN. Et enfin, il existe également DRAW (Deep Recurrent Attentive Writer) qui est une variante des auto-encodeurs conçue par Google Deepmind dont l'idée est de faire apprendre aux réseaux des caractéristiques de l'image les unes après les autres.

## 8 Conclusion

Dans l'optique de création d'images de microstructures, il semblerait que les GAN aient du potentiel. Malgré les difficultés rencontrées, il a été possible de concevoir un modèle permettant de générer des images très proches des attentes, et même si elles ne sont pas parfaites, nous avons pu mettre en avant des chemins à suivre pour explorer le sujet au-delà des GANs. Notons également que ce projet a permis de clarifier multiples points sur les réseaux de neurones ce qui a permis de démystifier certains aspects de ces intelligences artificielles. Enfin, ce projet démontre que la complexité des réseaux de neurones ne vient pas seulement de la théorie, mais de l'approche pragmatique qu'il faut avoir pour s'attaquer à un problème avec des méthodes bien définies afin de ne pas entraîner un réseau défectueux aveuglement.

Tout le code utilisé pour ce projet est disponible sur le [GitLab](#) de l'EMSE.

## 9 Annexe

### 9.1 Vecteur gaussien

**Définition :** La matrice de covariance d'un vecteur  $X \in \mathbb{R}^d$  est la matrice  $\Gamma$  où :

$$\Gamma = E((X - \xi)(X - \xi)^T) = \begin{bmatrix} Var(X_1) & cov(X_1, X_2) & \cdots & cov(X_1, X_d) \\ cov(X_1, X_2) & Var(X_2) & \cdots & cov(X_2, X_d) \\ \vdots & \vdots & \ddots & \vdots \\ cov(X_1, X_d) & cov(X_2, X_d) & \cdots & Var(X_d) \end{bmatrix} \quad (37)$$

où  $\xi$  est la moyenne de  $X$ .

**Définition :** Un vecteur aléatoire réel  $X = (X_1, \dots, X_d) \in \mathbb{R}^d$  est dit gaussien non dégénéré si :

- Sa matrice de covariance  $\Gamma$  est inversible
- Il admet la densité (multi) gaussienne ou normale suivante :

$$x \rightarrow \frac{1}{\sqrt{\det(\Gamma)}\sqrt{2\pi}} \exp\left(-\frac{1}{2} \langle x - \xi | \Gamma^{-1}(x - \xi) \rangle\right) \quad (38)$$

où  $\xi$  est la moyenne de  $X$  et  $\langle | \rangle$  un produit scalaire.

### 9.2 Divergence de Kullback-Leibler

Une manière de mesurer l'écart entre 2 distributions de probabilité est la divergence de Kullback-Leibler. Soit  $p(x)$  et  $q(x)$  deux densités de probabilité définies sur  $\mathbb{R}^n$ . On définit la divergence de Kullback-Leibler de la manière suivante :

$$D_{KL}(p||q) = \int_{\mathbb{R}^n} \log\left(\frac{p(x)}{q(x)}\right) p(x) dx \quad (39)$$

L'utilisation de cette divergence est souvent utilisée pour comparer une distribution connue et une distribution approchée.

### 9.3 Divergence de Jensen-Shannon

Un autre type de divergence est la divergence de Jensen-Shannon. En reprenant les mêmes notations, cette divergence est définies par :

$$D_{JS}(p||q) = \frac{1}{2} D_{KL}(p||M) + \frac{1}{2} D_{KL}(q||M) \quad (40)$$

où  $M = \frac{p(x)+q(x)}{2}$ .

On remarque que cette divergence se définit par la divergence de Kullback-Leibler. Elle permet également de mesurer l'écart entre 2 distributions de probabilité. Cette divergence est symétrique  $D_{JS}(p||q) = D_{JS}(q||p)$

## 9.4 La matrice de confusion

Une matrice de confusion est une matrice qui compare les prédictions et les valeurs réelles pour un problème de classification. Par exemple pour un problème à 2 classes, positif ou négatif, on obtient ceci :

Réalité \ Prédiction	Positif	Négatif
	Positif	Négatif
Positif	vrai positif (VP)	faux négatif (FN)
Négatif	faux positif (FP)	vrai négatif (VN)

Grâce à cette matrice, nous pouvons définir la précision, la spécificité et la sensibilité par :

$$precision = \frac{VP + VN}{VP + FN + FP + VN} \quad (41)$$

$$specificite = \frac{VP}{VP + FP} \quad (42)$$

$$sensibilite = \frac{VP}{VP + FN} \quad (43)$$

## Références

- [1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017. 7.4.1
- [2] Yoshua Bengio. Springtime for ai : The rise of deep learning, 2016. 5.2
- [3] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. 2
- [4] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans, 2017. 7.4.1
- [5] Jonathan Hui. Wasserstein gan and wgan-gp. <https://jonathan-hui.medium.com/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490>, 2018. 7.4.1
- [6] Nielsen. Michael. How the backpropagation algorithm works, 2019. 4.2.2