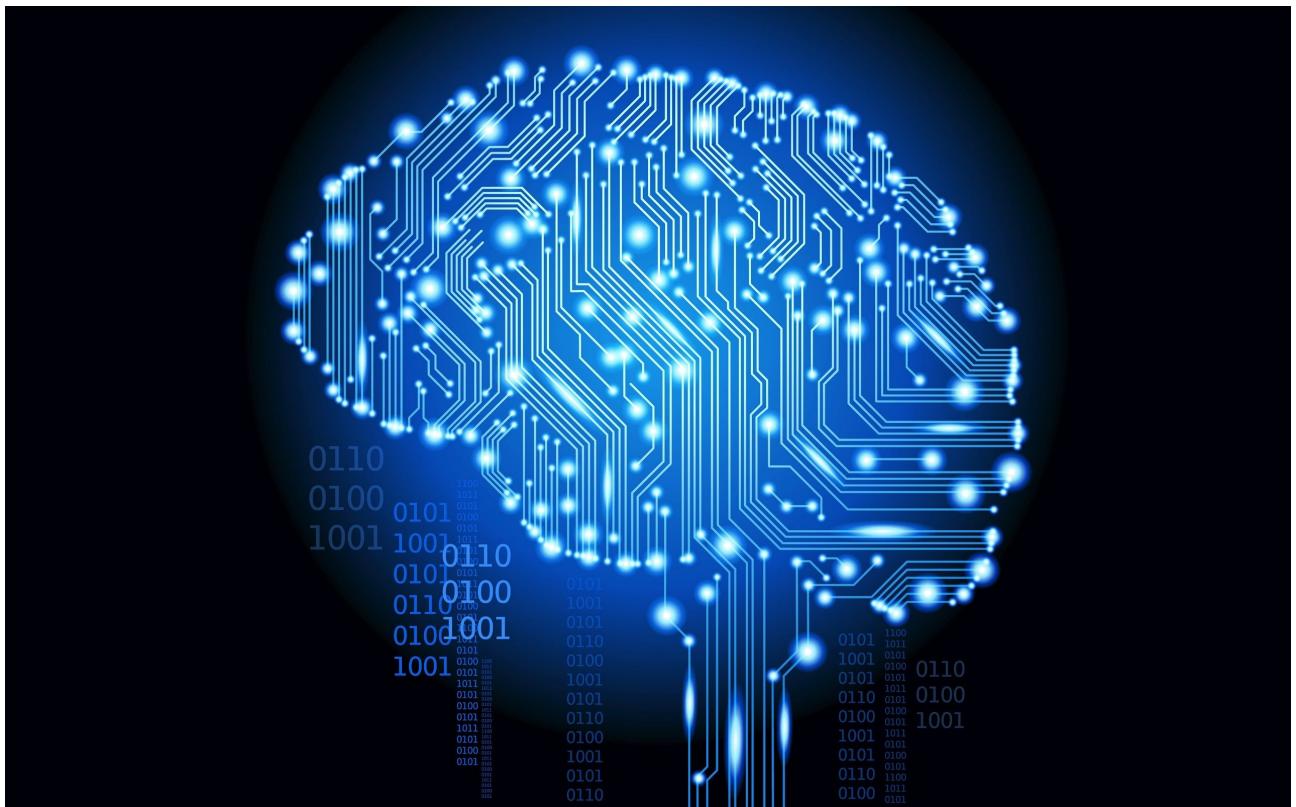


Université de Bordeaux

Modèles Génératifs en Intelligence Artificielle



Dorian Hervé, Marie-Mathilde Garcia, Wael Ben Hadj Yahia

Enseignant : Nicolas Papadakis

Projet réalisé dans le cadre du TER
2019-2020

Avant-Propos

Dans le cadre de l'UE TER nous avons choisi de travailler sur le sujet "Modèles Génératifs en Intelligence Artificielle" (GAN). L'objectif de ce projet est de comprendre les différents mécanismes qui sont impliqués dans les processus qui interviennent dans ce modèle ainsi que d'en réaliser une implémentation fonctionnelle (en Python).

Ce rapport présentera des notions de base nécessaires à la compréhension du modèle GAN avant d'en détailler le fonctionnement. Ensuite, nous exposerons nos résultats et nous terminerons en présentant les limites de ce modèle. L'implémentation informatique a été possible à travers l'outil collaboratif Google Colaboratory.

Ce rapport a été réalisé par Wael Ben Hadj Yahia, Marie-Mathilde Garcia, et Dorian Hervé dans le cadre de l'UE "TER".

Nous tenons à remercier M. Nicolas Papadakis pour nous avoir encadré lors de la réalisation de ce projet.

Table des matières

Introduction	1
1 Notions de base	2
1.1 Machine Learning	2
1.2 Réseaux de neurones	2
1.2.1 Fonctionnement	2
1.2.2 La rétropropagation	4
2 Le modèle GAN	6
2.1 Le concept	6
2.2 Le générateur : D'un bruit à une sortie	6
2.3 Le discriminateur : D'une entrée à une probabilité	7
2.4 L'algorithme	7
3 Implémentation	9
3.1 GAN sur un exemple simple : loi normale	9
3.1.1 Générateur	9
3.1.2 Discriminateur	9
3.1.3 Résultats	10
3.2 GAN sur la librairie MNIST	13
3.2.1 Code	13
3.2.2 Résultats	18
3.3 GAN sur la librairie FASHION MNIST	21
3.3.1 Résultats	21
4 Les limites du modèle GAN	24
Conclusion	25
Bibliographie	25

Introduction

Depuis leur introduction en 2014 par Ian Goodfellow, les modèles GAN ont suscité un grand intérêt car ils ont repoussé les limites de l'Intelligence Artificielle. L'IA ne se contente plus de reconnaître des images, elle est désormais capable d'en générer.

L'article publié en 2014 par Ian Goodfellow et une équipe de chercheurs au sein du département informatique et recherche opérationnelle de l'Université de Montréal ([1]), très novateur dans le domaine, a servi de base pour tous les travaux et les avancées qui ont suivis.

Notre sujet de TER s'appuie essentiellement sur cet article détaillant le fonctionnement de l'algorithme du modèle GAN (qui repose fondamentalement sur la recherche d'un point selle dans le cadre d'un problème min-max à 2 joueurs).

Chapitre 1

Notions de base

1.1 Machine Learning

Le Machine Learning est une technologie d'Intelligence Artificielle qui permet aux machines d'apprendre à classifier des données de manière automatique à partir d'importants jeux de données. Le Machine Learning se base sur ces jeux de données pour fournir des analyses prédictives. Celui-ci se révèle être le plus efficace parmi les autres méthodes d'analyse de données massives, que ce soit en rapidité ou en précision. Plus le nombre de données grandit, plus le modèle affinera son analyse prédictive et donc sa précision. L'objectif d'une analyse prédictive est d'exploiter des données et les traiter afin de prédire des probabilités en se basant sur le passé (i.e les données utilisées lors de l'apprentissage).

Nous parlons d'apprentissage supervisé lorsque les données d'entrée sont déjà labellisées. Les algorithmes se servent alors de ces données pour être en mesure d'en catégoriser de nouvelles qui ne seront pas labellisées, et cela avec un taux d'erreur le plus faible possible. L'apprentissage non supervisé reçoit un jeu de données qui n'est pas catégorisé, l'algorithme doit alors détecter des similarités dans les données afin de les organiser.

Les modèles GAN sont des algorithmes d'apprentissage non supervisé utilisant des réseaux de neurones, c'est-à-dire du deep learning.

1.2 Réseaux de neurones

1.2.1 Fonctionnement

Les réseaux de neurones artificiels sont inspirés du cerveau humain : dans notre cerveau nous avons des neurones, qui sont reliés entre eux et qui se transmettent des informations grâce à des impulsions électriques. En fonction de l'intensité de la pulsion électrique les neurones suivants s'activeront (ou pas si la pulsion électrique est trop faible). C'est selon ce même principe que les mathématiciens ont conçu les réseaux de neurones artificiels.

Warren McCulloch un neurologue américain et Walter Harry Pitts un mathématicien américain, sont les premiers à introduire les neurones artificiels en 1943. Un neurone artifi-

ciel est donc un modèle mathématique représentant un neurone biologique. En 1958, Frank Rosenblatt a introduit le perceptron, c'est le premier classifieur binaire utilisant l'apprentissage supervisé et les neurones artificiels. La figure 1 nous donne une représentation de ce perceptron.

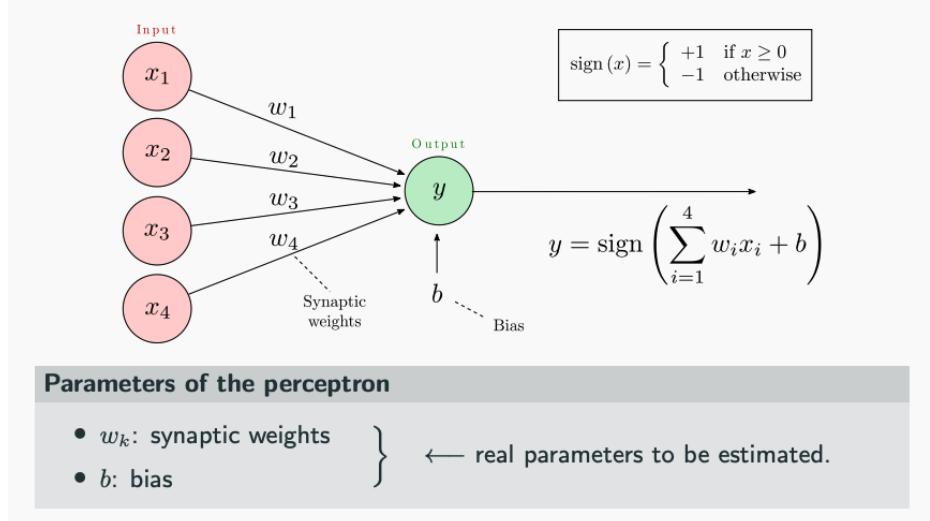


Figure 1: Illustration du fonctionnement du perceptron [2]

Il comporte plusieurs entrées, avec des poids sur chacune des arêtes. Le neurone effectue une sommation pondérée des entrées via les poids et déclenche un signal de sortie en fonction d'un certain seuil (le biais), comme nous pouvons le voir sur le schéma ci-dessus. Par exemple, si à la fin du perceptron $y = 1$, nous pouvons considérer que le signal est déclenché, alors que si $y = -1$ le signal n'est pas déclenché. L'entraînement du modèle consiste alors à ajuster les poids et le biais pour optimiser les résultats en sortie. Le problème de l'algorithme du perceptron est qu'il ne converge que pour des problèmes linéairement séparables.

Pour résoudre ce problème, les réseaux de neurones artificiels sont introduits. Ces réseaux sont aussi désignés par perceptron multicouche car ce type de réseau est le plus répandu.

Un réseau de neurones artificiels est un ensemble de neurones artificiels connectés. Ils vont être répartis en trois familles : les entrées (inputs), les sorties (outputs), et les "couches cachées" (hidden layers). La figure 2 est un exemple d'un tel réseau.

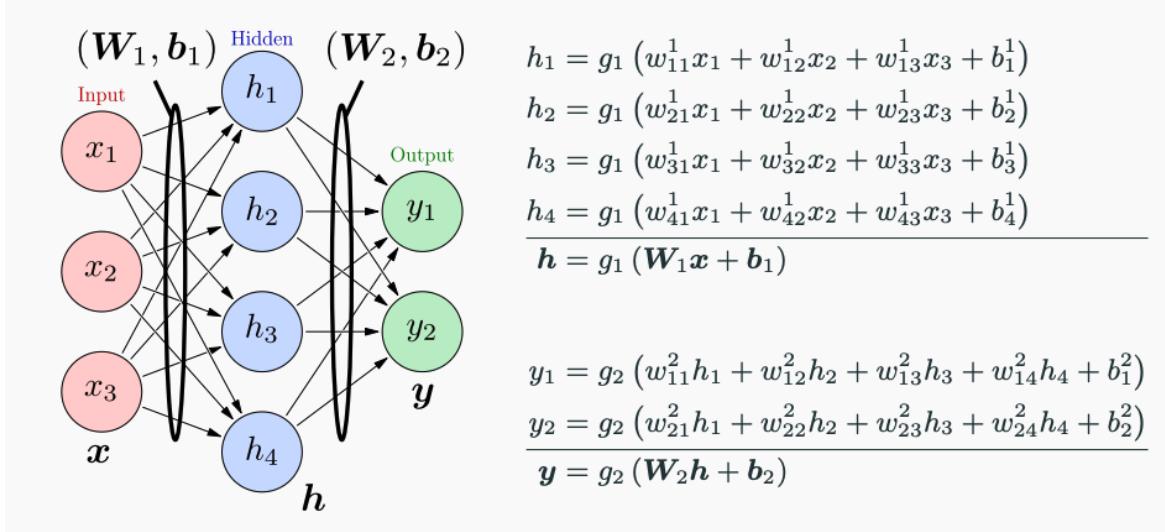


Figure 2: Illustration du fonctionnement du perceptron multicouche [2]

Nous noterons, x le vecteur des entrées, h_1, \dots, h_n les vecteurs d'une couche cachée (car dans cet exemple nous n'en avons qu'une), y le vecteur des sorties et W la matrice des poids de chacune des connexions entre les neurones. Par rapport aux calculs ci dessus (figure 2) nous avons aussi :

- w_{ij}^k qui représente le poids sur l'arête entre le noeud j et le noeud i à la k -ième couche cachée
- g_k la fonction d'activation appliquée aux x d'entrée.

Dans l'exemple de la figure 2, nous pouvons voir les calculs à chaque étape de ce réseau de neurones, le but est d'optimiser les valeurs des matrices W_1 , W_2 et des vecteurs b_1 et b_2 .

Dans cet exemple, nous avions une unique couche cachée mais dans ce que nous appelons les réseaux de neurones profonds, il peut y en avoir beaucoup plus.

L'une des applications des réseaux de neurones profonds est la classification d'images. Dans ce cas, en entrée sont donnés les pixels d'une image, et en sortie nous allons avoir 1 si l'image est un chien ou 0 si ce n'en est pas un (si par exemple nous cherchons à déterminer si l'image donnée en entrée est un chien).

Il existe plusieurs fonctions d'activation, ce sont ces fonctions qui conditionnent les sorties de notre réseau. Nous verrons plus tard laquelle nous déciderons d'utiliser pour notre cas.

1.2.2 La rétropropagation

Comme nous venons de l'expliquer, le but d'un réseau de neurones artificiel profond est d'apprendre de ce qu'on lui donne en entrée pour optimiser ses sorties. Pour les sorties, il va

optimiser son biais et sa matrice des poids.

Pour cela, le perceptron multicouche utilise la méthode de rétropropagation du gradient. Cette méthode consiste à, une fois que l'on a notre sortie y , calculer l'écart entre la sortie attendue et la sortie obtenue. Ensuite, il faut propager cette erreur vers l'arrière du réseau, en mettant à jour les poids et le biais en utilisant les dérivées en chaînes (Chain rule).

Prenons un exemple : nous donnons une image de chat en entrée de notre réseau de neurones. En sortie nous obtenons une probabilité de 0,6 que cette image soit un chat, nous voulons qu'avec l'apprentissage cette probabilité s'approche de 1. Nous calculons donc le taux d'erreur, qui à cette étape est égal à : $(1 - 0,6)^2 = 0,16$. L'étape suivante est de minimiser cette erreur, pour cela nous regardons l'effet qu'un changement de poids, ou du biais, a sur notre sortie. Nous modifierons donc ces valeurs en utilisant les dérivées en chaînes, jusqu'à ce que la probabilité obtenue en sortie soit le plus proche possible de 1.

Chapitre 2

Le modèle GAN

2.1 Le concept

Nous commençons par donner une vulgarisation du principe du GAN afin de donner au lecteur un aperçu global du modèle.

Le modèle GAN peut être assimilé à un scénario à deux acteurs : un bandit et un policier. Le bandit cherche à produire des faux billets par imitation en se basant sur des vrais billets. Le rôle du policier est de détecter ces faux billets. Au tout début du scénario, le bandit est inexpérimenté, et les billets qu'il produit sont clairement falsifiés (i.e. le policier est capable de les discerner des vrais billets de manière aisée). Le bandit va donc apprendre de ses erreurs, et produira des faux billets plus convaincants au deuxième tour. Le policier sera confronté à des faux billets un peu plus convaincants que la fois précédente, et augmentera donc sa vigilance et son attention aux détails, et ainsi de suite ... Le jeu se termine lorsque le bandit s'est amélioré au point d'être capable de produire des faux billets tellement réalistes que le policier ne sera plus en mesure de détecter la fraude. A ce stade, nous aurons maximisé le nombre de faux billets non détectés du bandit et minimisé la certitude avec laquelle le policier décide si le billet auquel il est confronté est vrai ou frauduleux, d'où la fameuse appellation "Two player Min-Max game".

Rapidement, le problème revient à se rapprocher du point selle décrit par l'équation :

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log (1 - D(G(z)))] [3]$$

où G est le générateur (bandit dans l'exemple précédent), D le discriminateur (policier), $p_{data}(x)$ est la distribution du générateur sur les données x , $p_z(z)$ est l'a priori sur les variables de bruit en entrée. Ceci est l'écriture duale d'une différence entre deux distributions (celle de D et celle de G). L'objectif est d'obtenir $D(x) = \frac{1}{2}$, c'est-à-dire que le discriminateur n'est pas capable de faire la différence entre les deux distributions.

2.2 Le générateur : D'un bruit à une sortie

Le générateur est un réseau de neurones qui a pour objectif de produire un nouvel objet assez réaliste afin que le discriminateur le confonde avec les objets de la base de données. Un

bruit aléatoirement généré (comme une loi uniforme ou une loi normale) est alors donné en entrée du réseau de neurones.

2.3 Le discriminateur : D'une entrée à une probabilité

Le discriminateur est un réseau de neurones qui prend une entrée x (dont la nature dépend du problème : cela peut être un scalaire, une image ...) et a pour but de renvoyer le scalaire $D(x)$ qui correspond à la probabilité que x provienne de la base de données.

2.4 L'algorithme

Le modèle GAN est donc -principalement- l'agencement des deux éléments précédents : un générateur dont la sortie sera l'entrée du discriminateur. Nous avons aussi bien entendu une base de données dont les éléments peuvent être donnés en entrée du discriminateur. La figure 3 nous donne l'algorithme permettant l'approximation de la solution optimale telle que décrite dans l'article de référence [3].

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```

for number of training iterations do
    for  $k$  steps do
        • Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
        • Sample minibatch of  $m$  examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .
        • Update the discriminator by ascending its stochastic gradient:
            
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)}))) \right].$$

    end for
    • Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
    • Update the generator by descending its stochastic gradient:
            
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(\mathbf{z}^{(i)}))).$$

end for
```

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 3: Algorithme d'optimisation du modèle GAN [3]

Le schéma de l'architecture globale du GAN permettant de visualiser l'application de la rétropropagation est présenté dans la figure 4.

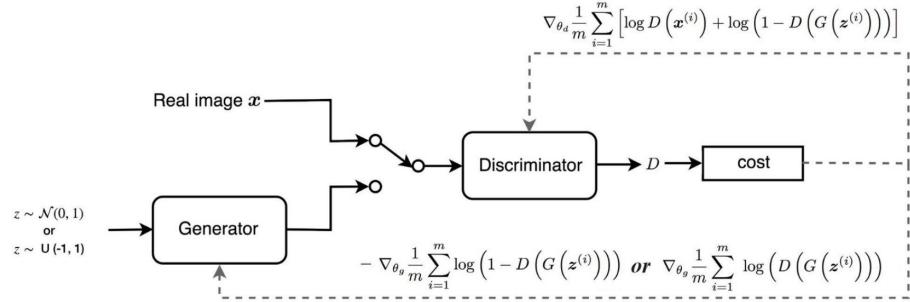


Figure 4: Architecture de l'algorithme [4]

Chapitre 3

Implémentation

3.1 GAN sur un exemple simple : loi normale

Pour une première implémentation des modèles GAN en python, nous avons cherché un code qui génère des fonctions usuelles. Nous avons trouvé un code ([5]) qui génère des lois normales.

3.1.1 Générateur

Le générateur prend en entrée des échantillons aléatoires. Or il serait trop simple pour le générateur d'apprendre à dessiner une loi normale à partir d'un échantillon d'entrée suivant une loi normale, nous lui donnons donc un échantillon de loi uniforme. Dans cet exemple, le générateur est composé de deux couches cachées. Un opérateur linéaire est appliqué sur chaque couche et la fonction d'activation utilisée est la fonction tangente hyperbolique. Le générateur reçoit donc les échantillons de données uniformément distribués et va tenter d'imiter la courbe de la normale.

3.1.2 Discriminateur

L'idée est proche de celle du générateur. En revanche, la fonction d'activation utilisée pour le réseau du discriminateur est la fonction sigmoïde. Le discriminateur reçoit des échantillons générés et d'autres provenant de la base de données. La base de données contient donc des échantillons suivant la fonction usuelle que nous allons vouloir estimer. Nous obtenons en sortie un scalaire entre 0 et 1 que nous pouvons interpréter comme un "real" vs "fake".

Pour résumer, le générateur va envoyer un bruit uniforme au discriminateur qui va essayer de différencier ce bruit d'un échantillon de la base de données. Ces échantillons suivent la loi usuelle que nous souhaitons estimer. Au fur et à mesure des epochs le générateur va améliorer sa sortie pour qu'elle se rapproche le plus possible d'un échantillon suivant la loi de nos échantillons de la base de données. Au final le discriminateur n'arrivera plus à faire la différence entre les deux.

3.1.3 Résultats

Nous avons voulu ensuite regarder l'efficacité du modèle GAN pour estimer une loi usuelle. Nous avons choisi de lancer le code ([5]) sur une $\mathcal{N}(4, 1.25^2)$.

Nous avons exécuté le code pour 1000 epoch. On dit qu'une epoch a été parcourue quand tous les éléments de la base de données d'entraînement ont été exploitées une fois (par sous-échantillonnages successifs). Voici les résultats :

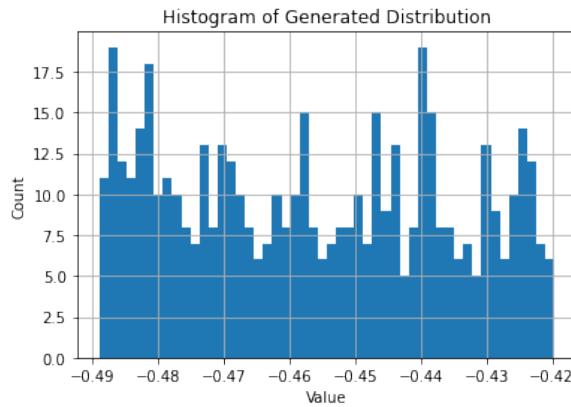


Figure 5: Distribution générée aléatoirement suivant une loi uniforme

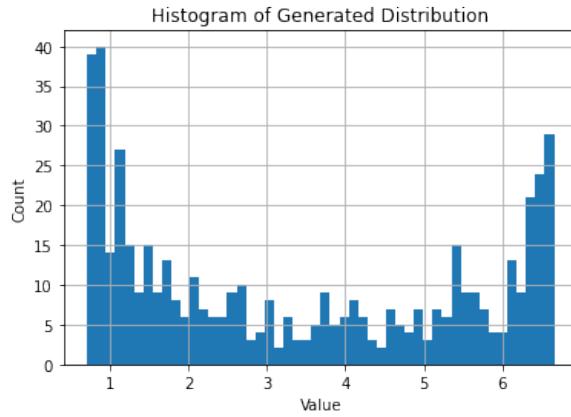


Figure 6: Distribution générée à l'epoch 200

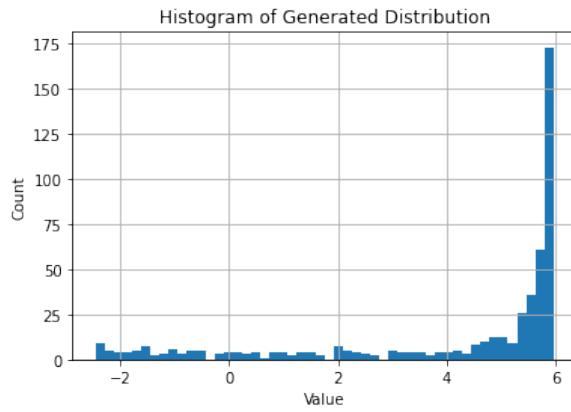


Figure 7: Distribution générée à l'epoch 400

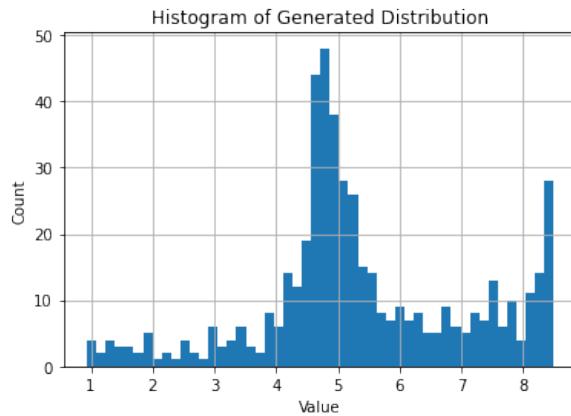


Figure 8: Distribution générée à l'epoch 600

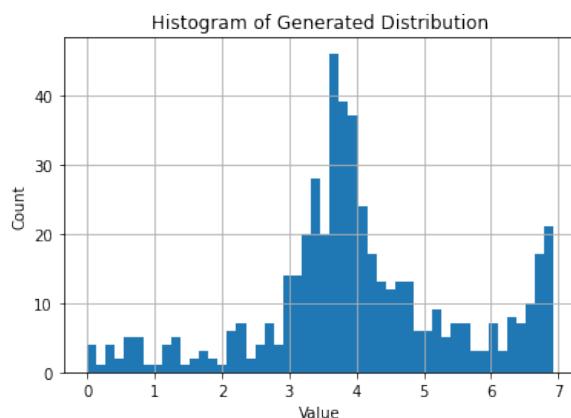


Figure 9: Distribution générée à l'epoch 800

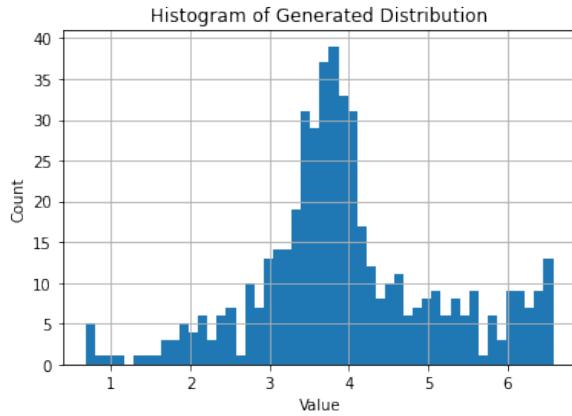


Figure 10: Distribution générée à l'epoch 1000

A 1000 epoch, nous avons déjà un histogramme prenant l'allure d'une loi normale d'espérance 4.

Dans nos cours de cette année et même des années précédentes nous avons vu plusieurs manières d'estimer une loi inconnue : en estimant sa densité, sa fonction de répartition... En travaillant sur ces modèles GAN, nous pouvons imaginer de les utiliser pour estimer une loi inconnue. Nous pourrions par exemple mettre dans une base de données des échantillons suivant la même loi inconnue et ensuite lancer le code du modèle GAN pour que le générateur arrive à créer des échantillons de cette loi inconnue et donc l'afficher sur un graphique comme ci-dessus pour la loi normale.

3.2 GAN sur la librairie MNIST

Nous allons maintenant appliquer le modèle GAN sur la base d'images MNIST qui contient environ 70 000 éléments. MNIST est une base d'images de chiffres manuscrits de taille 28×28 . Les images considérées sont en niveaux de gris.



Figure 11: Base de données MNIST ([6])

Cette fois-ci notre générateur va essayer de recréer des images les plus ressemblantes possibles à celles de la base d'images pour que le discriminateur n'arrive pas à les différencier.

3.2.1 Code

Nous avons réadapté le code ([7]) pour qu'il fonctionne sur Google Colaboratory. Vous trouverez le lien amenant à notre Google Colaboratory dans la bibliographie ([8]). Dans cette partie, nous allons expliciter les parties fondamentales de l'implémentation du modèle obtenu avec Python en utilisant une approche Programmation-Objet.

Ce code fait appel aux libraries Tensorflow et Keras pour construire les réseaux de neurones artificiels profonds. Nous commençons par expliquer la construction du générateur. Les codes qui seront commentés dans la suite font partie de la définition d'une classe appelée GAN.

Générateur

```

def build_generator(self):

    noise_shape = (100,)

    model = Sequential()

    model.add(Dense(256, input_shape=noise_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(np.prod(self.img_shape), activation='tanh'))
    model.add(Reshape(self.img_shape))

    model.summary()

    noise = Input(shape=noise_shape)
    img = model(noise)

    return Model(noise, img)

```

Figure 12: Code de la construction du générateur

Notre générateur est défini de telle sorte qu'il contienne 4 couches denses, c'est-à-dire que tous les neurones d'une couche sont reliés à tous les neurones des couches adjacentes. Les 3 premières couches utilisent la fonction d'activation LeakyReLU. Par opposition à la fonction ReLU définie comme ceci :

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{sinon} \end{cases}$$

La fonction LeakyReLU est définie de cette manière :

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha x & \text{sinon} \end{cases}$$

Dans notre cas, nous avons pris $\alpha = 0.2$.

L'avantage du LeakyReLU par rapport au ReLU classique est qu'il permet d'éviter le problème du "dying ReLU" : en effet, en utilisant la fonction d'activation ReLU, lorsqu'un neurone est dans la partie négative, il renvoie toujours un signal 0. De plus, puisque la pente de la fonction dans la partie négative est constante à 0, il y a peu de chances que ce neurone puisse envoyer un signal non nul par la suite (il y est "coincé"). De tels neurones ne jouent donc pas dans le modèle et perdent leur utilité. Par ailleurs, il semblerait que cela accélère la phase d' entraînement.

Voici le graphe des deux fonctions :

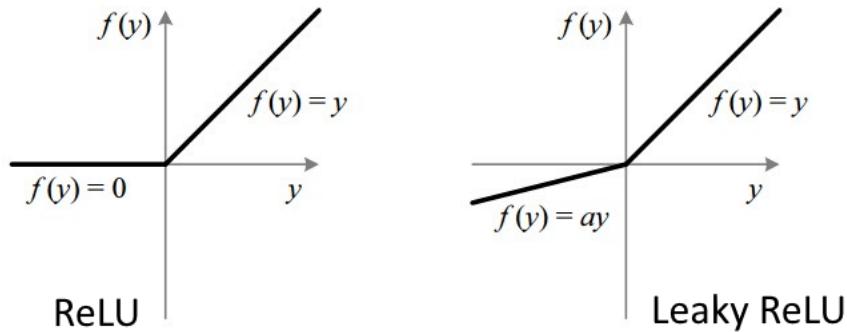


Figure 13: Graphique des fonctions *ReLU* et *LeakyReLU* ([9])

Enfin, la dernière couche utilise la fonction tangente hyperbolique comme fonction d’activation. La sortie de ce réseau est reconstituée en une image qui sera alors retournée.

Discriminateur

```
def build_discriminator(self):

    img_shape = (self.img_rows, self.img_cols, self.channels)

    model = Sequential()

    model.add(Flatten(input_shape=img_shape))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()

    img = Input(shape=img_shape)
    validity = model(img)

    return Model(img, validity)
```

Figure 14: Code de la construction du discriminateur

Notre discriminateur prend en entrée une image qu'il réduit progressivement sur une étendue de 3 couches. Il renvoie en sortie un scalaire qui correspond à la probabilité que l'image donnée en entrée provienne de la base de données. La première couche a pour rôle de transformer

l'image en entrée (une matrice de pixels) en un vecteur unidimensionnel grâce à une opération d'aplatissement (flatenning) comme illustré sur la figure suivante.

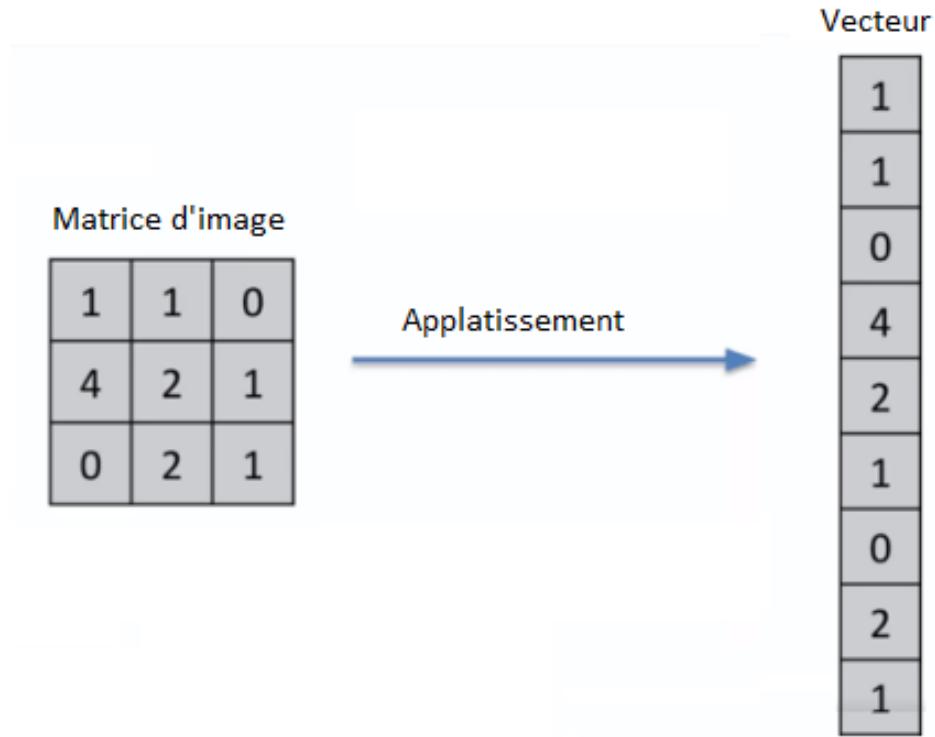


Figure 15: Opération d'aplatissement ([10])

La dernière couche utilise la fonction sigmoïde comme fonction d'activation.

Entraînement

La partie d'entraînement se résume en les étapes suivantes : les données sont découpées en échantillons train/test, et ensuite, en itérant sur les epochs, le modèle est optimisé en utilisant l'optimiseur implicite défini avec ADAM . ADAM est un algorithme de descente de gradient stochastique basé sur l'estimation adaptée de moments, souvent utilisé en apprentissage profond pour son efficacité et sa complexité ([11]). Cet optimiseur a été spécifié au début de la classe GAN. De plus, le bruit qui démarre le processus est de loi normale centrée réduite.

```
def train(self, epochs, batch_size=128, save_interval=50):

    # Load the dataset
    #(X_train, _), (_, _) = fashion_mnist.load_data()
    (X_train, _), (_, _) = mnist.load_data()

    # Rescale -1 to 1
    X_train = (X_train.astype(np.float32) - 127.5) / 127.5
    X_train = np.expand_dims(X_train, axis=3)

    half_batch = int(batch_size / 2)

    for epoch in range(epochs):
        idx = np.random.randint(0, X_train.shape[0], half_batch)
        imgs = X_train[idx]

        noise = np.random.normal(0, 1, (half_batch, 100))
        gen_imgs = self.generator.predict(noise)

        # Train the discriminator
        d_loss_real = self.discriminator.train_on_batch(imgs, np.ones((half_batch, 1)))
        d_loss_fake = self.discriminator.train_on_batch(gen_imgs, np.zeros((half_batch, 1)))
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # -----
        # Train Generator
        # -----

        noise = np.random.normal(0, 1, (batch_size, 100))

        valid_y = np.array([1] * batch_size)
        g_loss = self.combined.train_on_batch(noise, valid_y)

        #print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch, d_loss[0], 100*d_loss[1], g_loss))

        if epoch % save_interval == 0:
            self.save_imgs(epoch)
```

Figure 16: Code de l'entraînement du modèle GAN

3.2.2 Résultats

Voici les résultats obtenus en exécutant notre code avec 30 000 epochs et avec des sous-échantillons de 32 images.

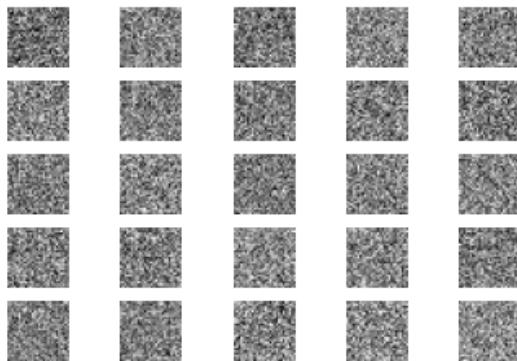


Figure 17: *Images générées aléatoirement (bruit) à l'epoch 0*

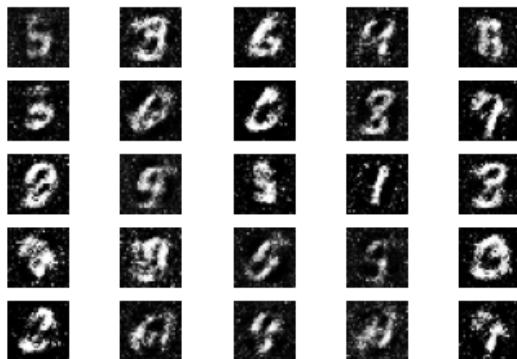


Figure 18: *Images générées à l'epoch 2000*

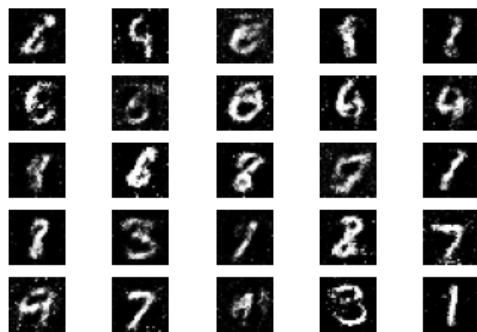


Figure 19: Images générées à l'epoch 5000



Figure 20: Images générées à l'epoch 10000

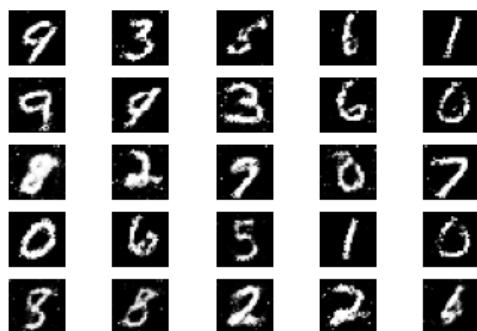


Figure 21: Images générées à l'epoch 30000

Nous observons qu'à partir de 10 000 epochs, nous pouvons déjà distinguer la majorité des chiffres générés. Pour indication, le code a pris 30 minutes pour arriver au bout des 30 000 epochs.

3.3 GAN sur la librairie FASHION MNIST

Nous allons maintenant appliquer le modèle GAN sur la base d'images FASHION MNIST. Les images de cette base montrent des vêtements, d'articles de Zalando, en basse résolution (28×28 pixels) et en niveaux de gris. Cette base de données contient 70 000 images.



Figure 22: Base de donnée FASHION MNIST ([12])

Nous allons utiliser le même code que pour la base MNIST, nous modifions seulement les lignes de chargement de la base de données. Le générateur et le discriminateur restent donc construits de la même manière.

3.3.1 Résultats

Voici les résultats obtenus en exécutant notre code avec 100 000 epochs et avec des sous-échantillons de 32 images. Dans cet exemple, nous avons du exécuter le code sur plus d'epochs pour obtenir un résultat satisfaisant. Cela s'explique par le fait que ces images ont plus de détails et sont plus variées, donc pour les distinguer clairement il faut plus d'apprentissage.

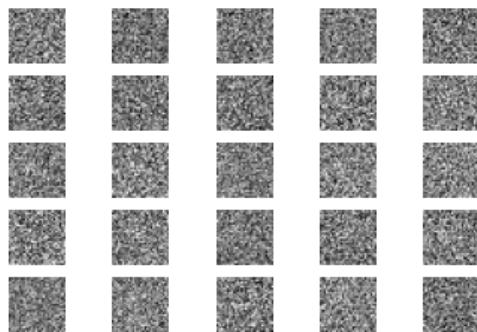


Figure 23: *Images générées aléatoirement (bruit) à l'epoch 0*

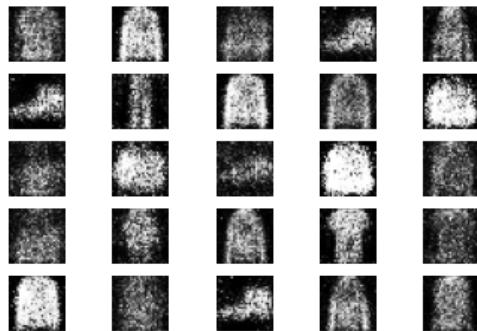


Figure 24: *Image générée à l'epoch 1000*

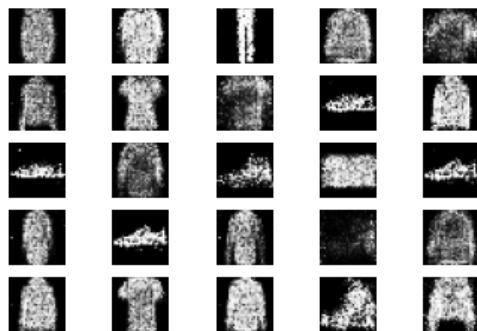


Figure 25: *Image générée à l'epoch 10 000*



Figure 26: Image générée à l'epoch 50 000

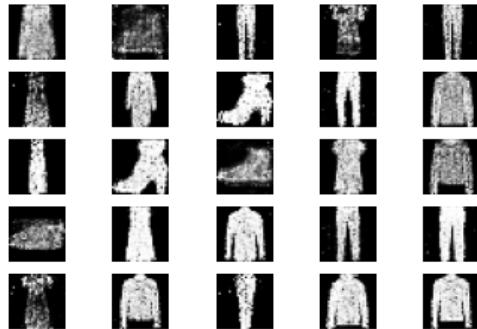


Figure 27: Image générée à l'epoch 100 000

Cette fois-ci, l'entraînement doit se faire sur plus d'epochs afin d'obtenir des images visuellement satisfaisantes. A partir de l'epoch 50 000, nous commençons à distinguer assez facilement la nature des habits (par exemple on peut différencier une botte d'un manteau). Par ailleurs, à l'epoch 100 000, nous distinguons de plus en plus des subtilités entre vêtements de même nature (bottes à talon vs. chaussures). Pour atteindre les 100 000 epochs, notre code a mis environ 1h30 à s'exécuter.

Chapitre 4

Les limites du modèle GAN

Nous allons présenter dans cette partie les limites de ces modèles sous différents aspects.

Un premier point qui est plus généralement vrai pour les réseaux de neurones artificiels, est qu'il est difficile de connaître les critères de classification des réseaux de neurones. Par exemple, pour l'être humain un chat est un animal à 4 pattes qui fait "miaou", mais cela peut être différent pour le réseau de neurones.

Par ailleurs, les modèles GAN sont efficaces lorsque nous avons une base données suffisamment riche. Cela peut être problématique dans le traitement de maladies orphelines par exemple. En effet, nous n'aurions pas suffisamment de sujets atteints de ces maladies pour permettre au modèle d'apprendre efficacement sur les données et d'obtenir des résultats fiables.

De plus, les modèles sont bien adaptés aux images mais beaucoup moins à d'autres types de données comme les données discrètes, comme du texte.

D'autre part, nous n'avons pas de garantie absolue sur la convergence de l'algorithme, pour des raisons mathématiques. Ceci est du au fait que le problème "min-max" n'est pas convexe. Finalement lorsque l'algorithme converge nous obtenons des résultats très satisfaisants, mais si ce n'est pas le cas, toute modification que nous pourrions apporter ne nous garantit pas au préalable la convergence de l'algorithme vers une solution plus satisfaisante.

Un autre point à considérer est l'aspect écologique de l'utilisation de ces modèles. En effet, pour des petites images (comme celles que nous avons utilisées) l'impact est moindre, mais pour des images de haute définition cela n'est pas la même chose. Si nous prenons l'exemple des modèles GAN qui génèrent des images réalistes de personnes qui n'existent pas, la phase d'entraînement peut prendre des jours car la qualité et donc le nombre de pixels de ces images est très élevée. La puissance de calcul requise est très conséquente, tout autant que son impact énergétique. En reprenant les mots d'un expert de ce sujet, cela pourrait à terme "faire fondre la banquise".

Finalement, un problème d'ordre éthique s'accompagne de ces avancées. En effet, l'usage des résultats qu'offrent les dérivées du GAN peut être mal-intentionné, comme créer des discours falsifiés en calquant le visage et la voix d'une figure politique importante ([13] : durée \approx 1m30), et/ou falsifier (de manière très réaliste) des preuves appuyant un procès juridique.

Conclusion

En conclusion de ce rapport, le modèle GAN consiste en l’interaction entre deux réseaux de neurones (le générateur et le discriminateur) ainsi qu’une phase d’optimisation (par rétropropagation) dans l’optique de résoudre un problème de min-max.

Il est souvent utilisé pour synthétiser des images, comme nous l’avons fait avec les bases de données MNIST et FASHION MNIST. Cependant, bien que les résultats obtenus sont généralement très satisfaisants, son utilisation n’est pas adaptée à tous les problèmes (et son utilisation peut être controversée).

Ceci a motivé le développement de nouvelles dérivées du GAN qui cherchent à l’améliorer et diversifier ses champs d’applications ([14]).

Enfin, les modèles GAN offrent des perspectives d’avenir très intéressantes et notamment dans l’industrie pharmaceutique. En effet, des chercheurs les ont déjà utilisés afin de trouver de nouvelles molécules avec des propriétés anti-cancer. Ils ont pour cela utilisé une autre version du modèle GAN, une version appelée AAE pour Adversarial Autoencoder. Cette version leur a permis de générer des empreintes moléculaires.([15])

Bibliographie

- [1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. in advances in neural information processing systems, (pp. 2672-2680). 2014.
- [2] Cours : https://www.charles-deledalle.fr/pages/files/ucsd_kpuw/2_predeep.pdf.
- [3] I. Goodfellow. NIPS 2016 tutorial : Generative adversarial networks. 2017.
- [4] Blog : https://medium.com/@jonathan_hui/gan-whats-generative-adversarial-networks-and-its-application-f39ed278ef09.
- [5] Blog : <https://medium.com/@devnag/generative-adversarial-networks-gans-in-50-lines-of-code-pytorch-e81b79659e3f>.
- [6] Blog : https://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es_MNIST.
- [7] GitHub : <https://github.com/eriklindernoren/Keras-GAN/blob/master/gan/gan.py>.
- [8] Code Google Colaboratory : https://colab.research.google.com/drive/1_0yKSY5BGRjaTymbJa_qIJgC7bRGFqcA?usp=sharing#scrollTo=h8MkcfqNL6uM.
- [9] Blog : <https://www.desigeek.com/blog/amit/2018/06/12/neural-network-basics-activation-functions/>.
- [10] Blog : <https://www.supinfo.com/articles/single/8037-deep-learning-reseau-convolution>.
- [11] Blog : <https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c>.
- [12] Blog : <https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/>.
- [13] Vidéo : <https://www.youtube.com/watch?v=ttGUIwfTYvg>.
- [14] Zhiting Hu, Zichao Yang, Ruslan Salakhutdinov, and Eric Xing. On unifying deep generative models. 2018.

- [15] Artur Kadurin, Sergey Nikolenko, Kuzma Khrabrov, Alex Aliper, and Alex Zhavoronkov. druGAN : An advanced generative adversarial autoencoder model for de novo generation of new molecules with desired molecular properties in silico. 2017.
- [16] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks (pp. 4401-4410). 2019.
- [17] Blog : <https://www.livosphere.com/2017/10/23/les-limites-de-l-ia-intelligence-artificielle-et-des-solutions-pour-y-faire-face/>.
- [18] Blog : <https://www.heidi.news/sciences/les-gan-nouvelle-technologie-star-en-intelligence-artificielle>.