Faculté des Sciences et Technologies
Département de Mathématiques

Master : Ingénierie, Statistique et Numérique - Data Science

Projet de fin d'année

Génération d'images artificielles de saulaies

Réalisé par :
Rayen Zaghouani, Loïc Thiery et Victor Vannobel

Année universitaire : 2023-2024

Table des matières

CHAPITRE 1

Introduction aux GANs

I Contexte

Le saule est un arbre typique des zones humides qui pousse naturellement en colonies denses appelées saulaies. Ces formations végétales procurent de nombreux services écosystémiques comme la régulation des eaux, l'épuration naturelle ou encore un habitat pour la faune et la flore. Leur esthétique unique en fait également des sujets prisés pour la peinture de paysage.
C'est dans ce contexte que s'inscrit notre projet de recherche visant à générer artificiellement des images de saulaies à l'aide de GANs.

Les GANs, introduits par Goodfellow et al. en 2014, représentent une classe de modèles d'apprentissage profond qui ont démontré des capacités impressionnantes pour générer des données artificielles qui ressemblent étonnamment aux données réelles. Cette technologie a été appliquée avec succès dans de nombreux domaines, allant de la génération d'images et la synthèse de voix à la création d'œuvres d'art. De plus, l'ajout du concept de transfert d'apprentissage à cette technologie a permis d'améliorer encore davantage la qualité et l'efficacité des GANs.

L'objet de ce rapport est de présenter notre travail sur l'utilisation des GANs pour générer des images artificielles de saulaies. Ce travail, réalisé dans le cadre de notre projet de TER, a été effectué en collaboration avec l'artiste Anourak Visouth. L'objectif est d'utiliser les GANs pour créer des images qui serviront d'inspiration à Madame Visouth pour la production de ses tableaux.

Dans ce contexte, nous avons constitué une base de données d'images de saules issues de photographies libres de droits. Nous avons ensuite conçu et entraîné un GAN en appliquant des techniques favorisant la variété des images générées.
Les GANs sont composés de deux réseaux distincts, un générateur et un discriminateur. Le générateur est chargé de produire des données qui tentent de reproduire la distribution des données réelles. Le discriminateur, quant à lui, est formé pour distinguer les images réelles des images générées. Ces deux réseaux sont formés simultanément dans un jeu de minimax, où le générateur tente de tromper le discriminateur et le discriminateur s'efforce de ne pas se laisser tromper.

Le transfert d'apprentissage, quant à lui, est une technique qui permet de profiter des connaissances acquises lors de la résolution d'un problème pour en résoudre un autre, généralement connexe. Dans le contexte des GANs, cela signifie utiliser les paramètres (poids) appris par un

GAN formé sur un ensemble de données (par exemple, des images génériques) pour initialiser un autre GAN destiné à générer des images d'un type spécifique (dans notre cas, des images de saulaies).

Toutefois, plutôt que d'adopter une approche de transfert d'apprentissage, nous avons choisi de concevoir et d'entraîner notre GAN spécifiquement pour la tâche de génération d'images de saulaies. Cette décision repose sur plusieurs motivations clés, notamment le désir de capturer les caractéristiques uniques et les nuances subtiles des saulaies, l'opportunité d'expérimenter avec des architectures de GANs personnalisées, et la nécessité de maintenir une efficacité computationnelle adaptée à nos ressources disponibles.

Dans ce rapport, nous détaillerons la structure et le fonctionnement des GANs, le processus de collecte et de préparation des données, notre approche spécifique pour l'entraînement des réseaux sans recourir au transfert d'apprentissage, et les méthodes employées pour évaluer les résultats obtenus. Nous discuterons également des défis rencontrés au cours de ce projet et des stratégies développées pour les surmonter.

Alors que le cœur de notre projet se concentre sur la génération d'images de saulaies, notre démarche vise également à contribuer à une compréhension plus large des capacités et des limitations des GANs, ainsi que des implications pratiques et théoriques du choix entre le transfert d'apprentissage et l'entraînement from scratch dans le contexte de la génération d'images.

Ce rapport se veut ainsi un exposé détaillé de notre exploration des GANs, illustrant non seulement notre travail spécifique sur la génération d'images de saulaies mais aussi offrant une vision sur l'utilisation de ces technologies avancées en intelligence artificielle pour la création d'images. Nous espérons que vous le trouverez à la fois informatif et inspirant.

(Figure 1) Saule pleureur.

II Théorie des GANs

II.1 Générateur et discriminateur

Un GAN comprend deux éléments principaux : un générateur G et un discriminateur D. Mathématiquement, le générateur G est une fonction qui transforme un vecteur de bruit z issu d'une distribution p_z (souvent une distribution normale) en une donnée qui imite la distribution des données réelles p_{data}. En d'autres termes, si z est tiré de p_z, alors $G(z)$ cherche à reproduire une donnée issue de p_{data}.

Le discriminateur D est une fonction de classification qui évalue la probabilité que l'échantillon x soit une donnée réelle plutôt que générée par G. Ainsi, $D(x)$ est estimé pour être proche de 1 si x est réel et proche de 0 si x est généré.
Le générateur G essaye quant à lui de minimiser la probabilité que le discriminateur D prédise qu'une image générée le soit, c'est à dire que $D(G(z))$ ait une probabilité proche de 1.

II.2 Fonction objectif et théorie des jeux

La dynamique des GANs est encadrée par un jeu minimax, où les deux joueurs (générateur et discriminateur) ont des objectifs opposés. La fonction objectif, ou valeur du jeu, est donnée par :

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

Le discriminateur D cherche à maximiser cette fonction en identifiant correctement les vraies données et les données générées. Simultanément, le générateur G vise à minimiser cette même fonction, en trompant le discriminateur pour qu'il classe ses sorties, $G(z)$, comme de vraies données.

II.3 Équilibre de Nash et convergence

Dans le contexte des GANs, un équilibre de Nash est atteint lorsque le générateur produit des données tellement convaincantes que le discriminateur ne peut mieux faire que de deviner au hasard l'authenticité des données (probabilité de 0.5 de se tromper/avoir raison). Mathématiquement, cet équilibre est atteint lorsque la fonction objectif atteint son point-selle, où la stratégie de G ne peut plus réduire la valeur de $V(D,G)$ et D ne peut plus l'augmenter.

II.4 Enjeux mathématiques de l'entraînement

L'entraînement des GANs peut être délicat en raison de la nature de la fonction objectif. Des problèmes tels que la disparition du gradient peuvent survenir, en particulier lorsque le discriminateur surpasse significativement le générateur. De plus, trouver l'équilibre de Nash dans des espaces de fonctions complexes peut s'avérer difficile.

II.5 Techniques d'optimisation

Pour pallier ces problèmes, diverses techniques mathématiques ont été proposées. Le gradient penalty, par exemple, ajoute un terme de régularisation à la fonction objectif pour assurer un comportement plus doux et régulier du gradient. La normalisation spectrale est une autre technique qui vise à restreindre la complexité du discriminateur en normalisant ses poids, évitant ainsi une domination excessive sur le générateur.

CHAPITRE 2

Génération d'images réalistes

I DCGAN

I.1 Présentation

Les Deep Convolutional Generative Adversarial Networks (DCGAN) représentent une avancée significative dans la conception des GANs, en intégrant des architectures de réseaux de neurones convolutifs dans le cadre des GANs.

Introduit en 2016 par Alec Radford, le DCGAN utilise des architectures convolutionnelles profondes pour le générateur et le discriminateur, qui se distinguent par leur capacité à apprendre des hiérarchies de représentations spatiales à partir d'images. Contrairement aux couches entièrement connectées, les couches convolutives exploitent la nature spatiale des images, permettant au modèle de reconnaître et de générer des motifs avec une efficacité computationnelle supérieure (notamment avec l'ajout d'un procédé de normalisation des couches). La compréhension de ces concepts se situe dans le domaine du traitement d'images.

Le générateur de DCGAN transforme un vecteur latent z, tiré d'une distribution aléatoire, en une image à l'aide de couches de convolution transposée. Le générateur du modèle peut être représenté mathématiquement comme suit :

$$G(z) = f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1(z)$$

$$f_i = \begin{cases} \text{ReLU}(\text{BatchNorm2d}(\text{ConvTranspose2d}(z, W_1) + b_i)) & \text{pour } i = 1 \\ \text{ReLU}(\text{BatchNorm2d}(\text{ConvTranspose2d}(f_{i-1}, W_i) + b_i)) & \text{pour } i = 2, 3, ..., n-1 \\ \text{Tanh}(\text{ConvTranspose2d}(f_{n-1}, W_n)) & \text{pour } i = n \end{cases}$$

Où W_i et b_i sont respectivement les poids de la couche de convolution et le terme de biais pour la normalisation des couches (plus précisemment : normalisation des images de sortie pour chaque canal de sortie).

Dans cette représentation, $G(z)$ est le générateur prenant en entrée un vecteur de bruit z et produisant une image générée.

Le discriminateur applique une série de convolutions pour réduire progressivement la dimensionnalité de l'image d'entrée, évaluant finalement la probabilité qu'une image soit réelle ou générée. Le discriminateur dans le modèle peut être représenté mathématiquement comme suit :

$$D(x) = f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1(x)$$

$$f_i = \begin{cases} \text{LeakyReLU}(\text{Conv2d}(x, W_1)) & \text{pour } i = 1 \\ \text{LeakyReLU}(\text{BatchNorm2d}(\text{Conv2d}(f_{i-1}, W_i) + b_i)) & \text{pour } i = 2, 3, ..., n-1 \\ \text{Sigmoid}(\text{Conv2d}(f_{n-1}, W_n)) & \text{pour } i = n \end{cases}$$

Où W_i et b_i sont respectivement les poids de la couche de convolution et le terme de biais pour la normalisation des couches.

Le discriminateur, représenté par $D(x)$, prend en entrée une image x et produit une sortie qui représente la probabilité que x soit une image réelle.

Cependant, malgré leurs avantages, les DCGANs peuvent encore souffrir de certains problèmes tels que le mode collapse et le problème de convergence instable.

Dans le cadre de notre recherche, un script informatique spécifique a été développé et employé. Ce script est écrit en Python, un choix motivé par la richesse de son écosystème dans le domaine de l'apprentissage automatique, et utilise PyTorch comme bibliothèque de backend pour la modélisation des GANs. PyTorch offre une flexibilité et une efficacité optimales dans la définition et l'entraînement des modèles complexes, rendant possible l'implémentation de nos GANs personnalisés pour la génération d'images.

I.2 Prétraitement des images

Le segment initial du script est dédié au prétraitement des images. Cette étape est nécessaire pour normaliser les données d'entrée, assurant ainsi que le réseau peut les traiter efficacement. Les opérations comprennent le redimensionnement des images à une taille 64x64 et la normalisation des valeurs de pixels.

On importe les 1000 images de notre base de données. Les images sont en couleur, de taille 512x512.

```
dataroot = "repertoire/contenant/les/images"
nb_channels = 3 # For RGB images, but if you use grayscale images, ToTensor()
    ↪ will replicate the single channel into three channels, so you should not
    ↪ have to modify anything
image_resize = 64

batch_size = 64
nb_gpu = 0
nb_workers = 0 # based on system resources

# GPU or CPU (Not having at least 1 GPU can prevent code from working)
device = torch.device("cuda:0" if (torch.cuda.is_available() and nb_gpu > 0)
    ↪ else "cpu")
print(device)
```

Les explications des paramètres sont les suivantes :

- **dataroot** : Chemin vers le répertoire contenant les images.
- **nb_channels** : Nombre de canaux pour les images. Pour les images en niveaux de gris, ce paramètre vaut 1.
- **image_resize** : Taille de redimensionnement des images.
- **batch_size** : Taille des mini-lots pour l'entraînement.
- **nb_gpu** : Nombre de GPU à utiliser.
- **nb_workers** : Nombre de travailleurs pour le chargement des données, basé sur les ressources du système.
- **device** : Dispositif à utiliser (GPU ou CPU) en fonction de la disponibilité des GPU sur le système.

L'implémentation par défaut est faite pour des images de taille 64x64, c'est pourquoi il faut redimensionner les images. Si une autre taille était voulue, il aurait fallu modifier les architectures

de G et D.

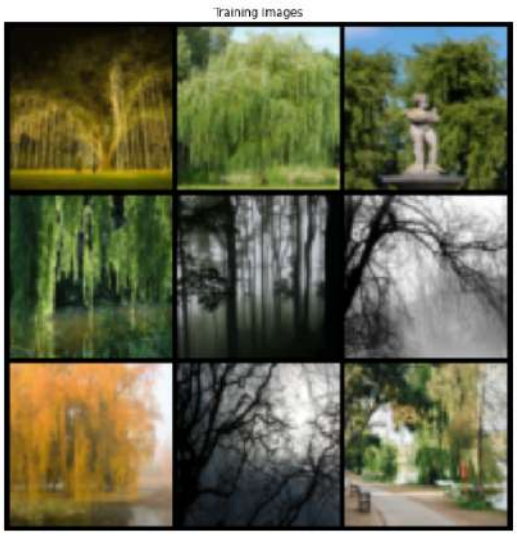
Voici le procédé pour obtenir nos images en 3x64x64, les dimensions attendues par notre GAN.

```
# Create the dataset by applying transformation to our images
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_resize),
                               transforms.CenterCrop(image_resize),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
                                   ↪ 0.5)),
                           ]))

print(f'Number of downloaded images: {len(dataset)}')
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=nb_workers)
```

Les images sont converties en tenseurs PyTorch et les valeurs des pixels sont normalisées afin d'améliorer les performances d'entraînement.

Après la préparation du dataset, le script initialise un dataloader pour gérer efficacement le chargement des données pendant l'entraînement. Le dataloader sépare efficacement nos données en mini-lots de taille **batch_size**.



(Figure 2) Aperçu de 9 images d'entraînement tirées d'un mini-lot du dataloader.

I.3 Hyperparamètres

Le script définit les paramètres clés pour le modèle, ces paramètres sont alignés sur ceux recommandés dans le papier DCGAN

```
# Size of z latent vector (i.e. size of generator input)
nz = 100
# Size of feature maps in generator
ngf = 64
```

```
# Size of feature maps in discriminator
ndf = 64
# Number of training epochs
num_epochs = 300
# Learning rate for optimizers, same value as described in the DCGAN paper
lr = 0.0002
# Beta1 hyperparameter for Adam optimizers, same value as described in the
    ↪ DCGAN paper
beta1 = 0.5
```

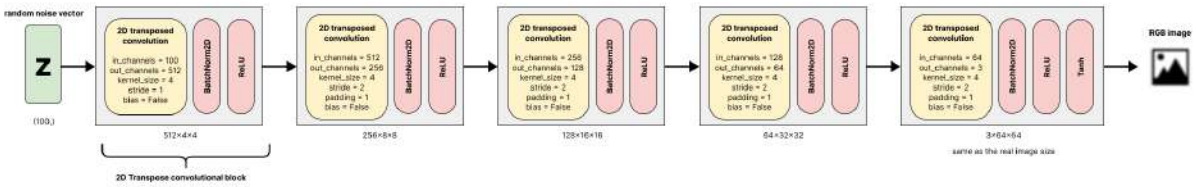
La variable **nz** est la taille du vecteur de bruit qui correspond à l'entrée du générateur.
La variable **num_epoch** correspond au nombre d'épisodes souhaité pour entraîner le réseau de neurones.
La variable **ngf** contrôle la taille des couches de déconvolution (convolution transposée) du générateur tandis que la variable **ndf** contrôle la taille des couches de convolution du discriminateur.

Nous utilisons l'optimisateur Adam. Dans ce contexte, **lr** contrôle la taille des pas de mise à jour des poids lors de l'entraînement, tandis que **beta1** est un paramètre spécifique à l'optimiseur Adam qui influe sur la façon dont les gradients sont combinés pour mettre à jour les paramètres du modèle.

I.4 Structure du réseau de neurones

Générateur

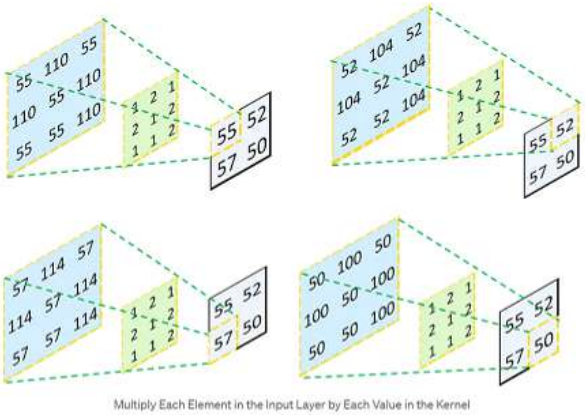
On commence par présenter l'architecture du générateur. On définit une classe `Generator` qui hérite de `nn.Module` de PyTorch. L'architecture du générateur est construite en séquence (`nn.Sequential`) de couches de convolution transposée (`nn.ConvTranspose2d`), chacune suivie par une normalisation par lots (`nn.BatchNorm2d`) et une fonction d'activation non-linéaire (`nn.ReLU` pour toutes les couches sauf la dernière qui utilise `nn.Tanh`).



(Figure 3) Structure du générateur.

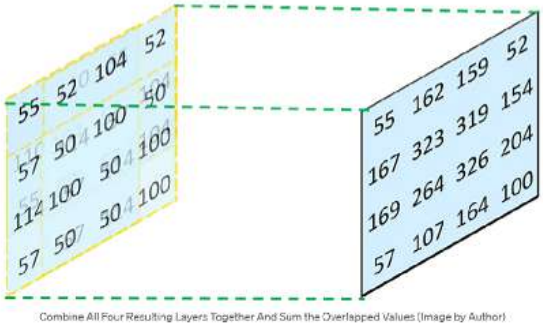
Couches du réseau de neurones

1. **Convolution transposée (`nn.ConvTranspose2d`)** : Cette opération réalise le principe opposé de la convolution. Au lieu de réduire les dimensions de l'entrée, elle les augmente. C'est ce qui permet au générateur de transformer un petit vecteur de bruit en une image de plus grande dimension.



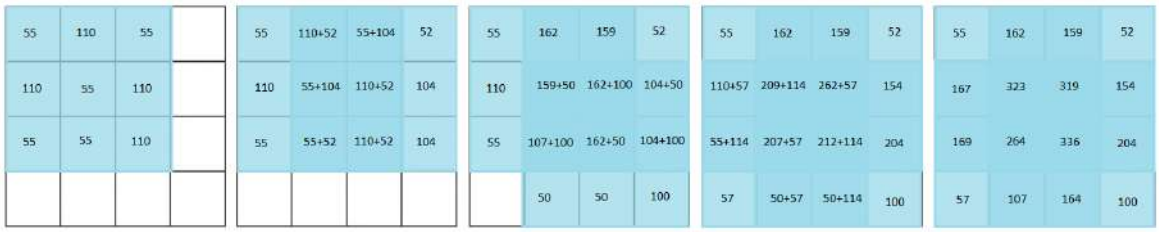
(Figure 4) Fonctionnement d'une couche de convolution transposée avec un stride de 1 et un padding de 0.

Le graphique ci-dessus montre une convolution transposée avec un stride de 1 et sans padding.
On multiplie chaque valeur de la matrice d'entrée (blanche) par la matrice du noyau de convolution (verte) afin d'obtenir des matrices temporaires (bleue).



(Figure 6) Assemblage de la matrice de sortie.

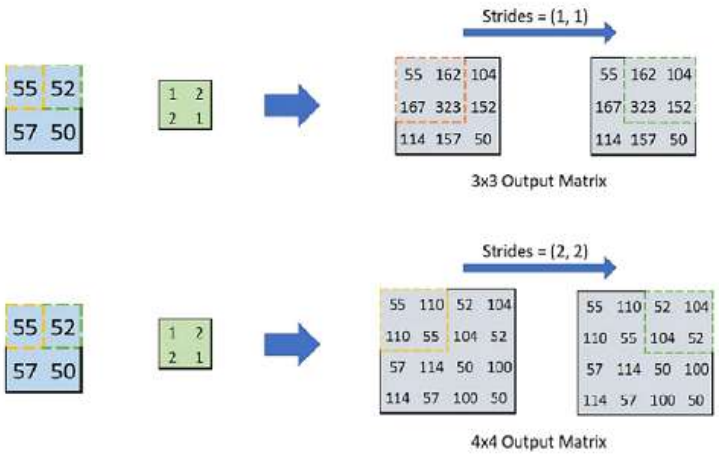
Ces matrices temporaires sont ensuite combinées en fonction de leurs positions initiales dans la matrice d'entrée. Pour obtenir la valeur finale de chaque pixel dans la matrice de sortie (matrice de droite dans la figure 6), on additionne les valeurs superposées.



(Figure 7) Visualisation des calculs.

Vu de cette manière, cela ressemble à une "déconvolution" (mais ce n'en est pas réellement une) : on fait le chemin inverse d'une convolution (que l'on verra plus tard).

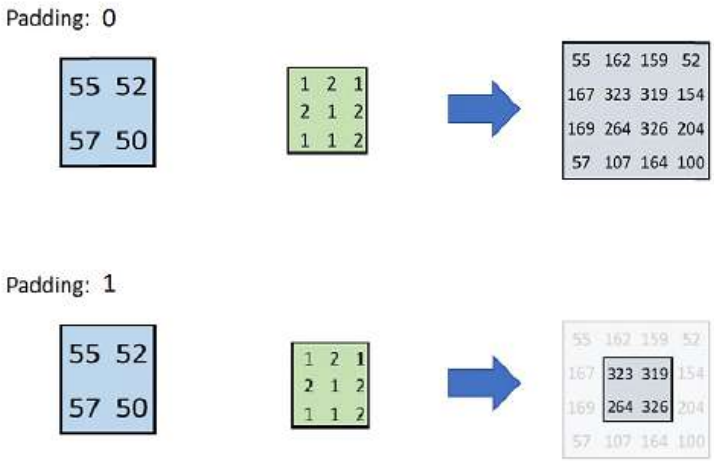
Strides :



(Figure 8) Fonctionnement d'une couche de convolution transposée avec un stride.

Le paramètre "strides" détermine la vitesse de déplacement du noyau sur la couche de sortie, comme illustré dans l'image ci-dessus. Il est important de noter que le noyau se déplace toujours d'un seul pas à la fois sur la couche d'entrée. Par conséquent, des strides plus grands entraînent une matrice de sortie plus grande.

Padding :



(Figure 9) Fonctionnement d'une couche de convolution transposée avec un padding.

L'objectif du padding dans une convolution transposée est de contrôler la taille de la sortie. Cependant, il est important de noter que seule la partie centrale de la sortie sera conservée, ce qui permet de conserver les informations les plus importantes de l'entrée.

Ces opérations de convolution transposée sont indispensables dans le processus de création d'images détaillées par le générateur de DCGAN à partir de données bruitées.

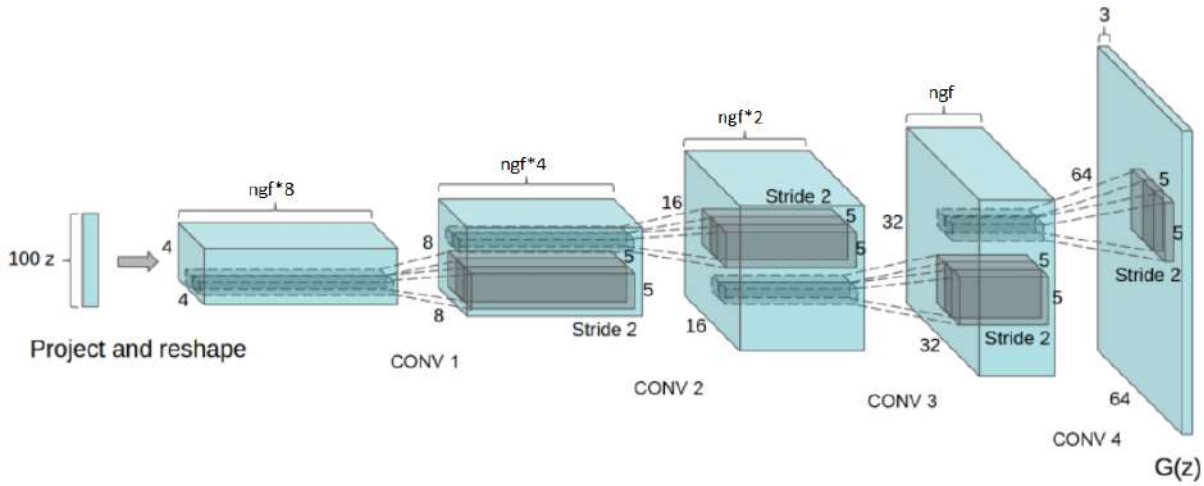
2. **Normalisation par lots (`nn.BatchNorm2d`)** : Chaque couche de convolution transposée est suivie d'une normalisation par lots qui stabilise l'apprentissage en normalisant la sortie de chaque couche. Cela permet au générateur d'apprendre plus rapidement et plus efficacement.
3. **Fonction d'activation ReLU (`nn.ReLU`)** : La Rectified Linear Unit (ReLU) est utilisée pour introduire de la non-linéarité dans le processus d'apprentissage, permettant au modèle de générer des données plus complexes.
   Pour $x \in \mathbb{R}$, $\text{ReLU}(x) = \max(0, x)$
4. **Dernière couche (`nn.Tanh`)** : La dernière couche de convolution transposée utilise une fonction d'activation hyperbolique tangente (`nn.Tanh`) pour normaliser les valeurs de sortie entre -1 et 1, ce qui correspond à la normalisation des pixels des images, aide ainsi à stabiliser l'entraînement.
   Pour $x \in \mathbb{R}$, $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

Processus de génération d'image

En commençant par un vecteur de bruit aléatoire z, le générateur effectue les opérations suivantes :

1. Le vecteur z est d'abord transformé par une convolution transposée en un tenseur de dimension 512x4x4.
2. Ce tenseur est ensuite agrandi et transformé à travers une série de convolutions transposées pour atteindre progressivement la taille d'une image réelle. À chaque étape, la hauteur et la largeur sont doublées, tandis que la profondeur est divisée par deux (sauf pour la dernière couche qui convertit en canaux RGB).
3. La dernière couche de convolution transpose le tenseur en une image RGB de taille 64x64, qui est la même taille que les images réelles utilisées pour l'entraînement du GAN.

Chaque bloc de convolution consiste en une convolution transposée, une normalisation par lots, et une activation ReLU, avec l'exception du dernier bloc qui remplace ReLU par Tanh.



(Figure 10) Visualition du processus de convolution transposée.

On remarque à quel point les valeurs de **nz**, **ngf** et **ndf** influent l'architecture du réseau de neurones.
Une formule s'obtient pour trouver la taille de sortie d'une couche de convolution transposée :

$$\text{output\_size} = (I - 1) * S + K - 2P$$

Où :

— I est la taille en entrée.

— K est la taille du noyau.

— S est stride.

— P est le padding.

BatchNorm2d étant une couche de normalisation, elle ne modifie pas la taille de sortie.
On applique la formule pour le premier bloc : $I = 1$, $K = 4$, $S = 1$ et $P = 0$, on a donc que output_size $= 4$. De plus, on avait défini **ngf***8 filtres (ou canaux) en sortie et en rentrant un seul bruit dans la couche, on obtient en sortie de couche un tenseur de taille (1, **ngf***8, 4, 4).
Pour le deuxième bloc : $I = 4$, $K = 4$, $S = 2$ et $P = 1$, on a donc que output_size $= 8$. On obtient en sortie de couche un tenseur de taille (1, **ngf***4, 8, 8). Et ainsi de suite.

L'implémation du générateur est la suivante :

```
# Generator Code
class Generator(nn.Module):
    def __init__(self, nb_gpu):
        super(Generator, self).__init__()
        self.nb_gpu = nb_gpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution

            nn.ConvTranspose2d(nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. ``(ngf*8) x 4 x 4``

            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. ``(ngf*4) x 8 x 8``

            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. ``(ngf*2) x 16 x 16``

            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. ``(ngf) x 32 x 32``

            nn.ConvTranspose2d(ngf, nb_channels, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. ``(nb_channels) x 64 x 64``
        )

```

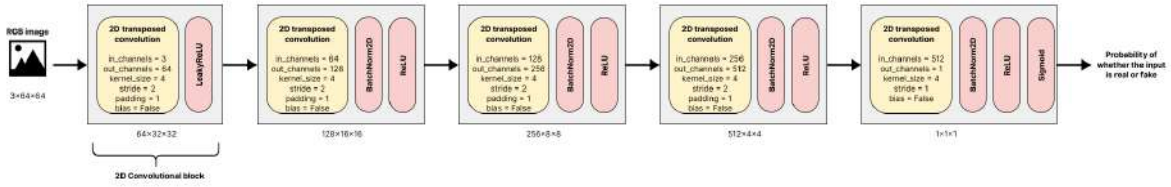
```
    def forward(self, input):
        return self.main(input)

# Create the generator
netG = Generator(nb_gpu).to(device)
# Apply the ''weights_init'' funb_channelstion to randomly initialize all
    ↪ weights
netG.apply(weights_init)
```

On crée une instance de la classe Generator pour obtenir un générateur que l'on appelle netG. Ses poids sont initialisés avec la fonction weights_init() qui est détaillée dans le code. On rappelle son principe : les poids d'une couche de convolution/convolution transposée sont initialisés suivant une $\mathcal{N}(0, 0.2)$ et les poids d'une couche de normalisation par lots suivant une $\mathcal{N}(1, 0.2)$.

Discriminateur

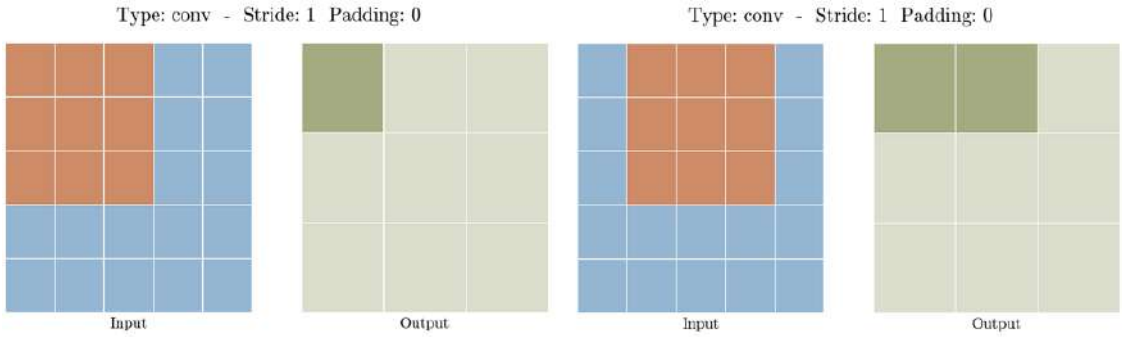
Le code définit une classe `Discriminator` qui hérite de `nn.Module` de PyTorch. L'architecture du discriminateur est composée d'une séquence de couches convolutives (`nn.Conv2d`), chacune suivie par une fonction d'activation LeakyReLU (`nn.LeakyReLU`) et, pour certaines couches, une normalisation par lots (`nn.BatchNorm2d`).



(Figure 11) Structure du discriminateur.

Couches du réseau de neurones

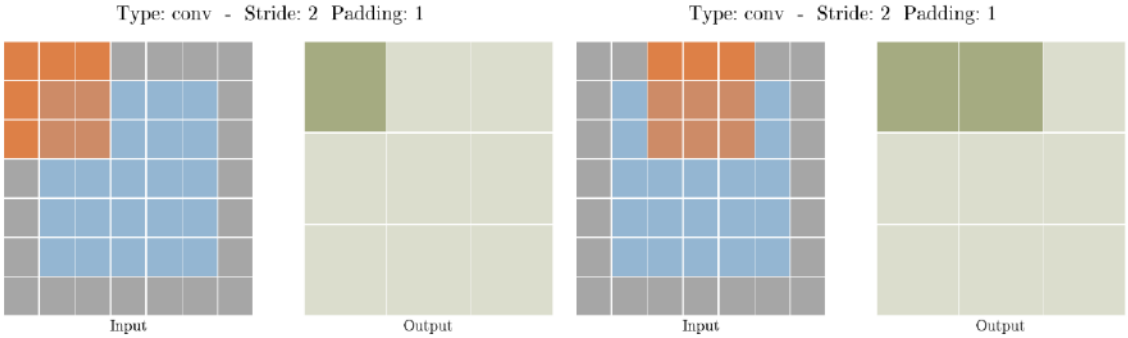
1. **Convolution (`nn.Conv2d`)** : Contrairement au générateur, le discriminateur utilise des convolutions (pas transposées) pour réduire progressivement la dimension spatiale de l'image tout en augmentant la profondeur (nombre de canaux). Les convolutions permettent au discriminateur d'extraire et d'apprendre des caractéristiques de différentes échelles spatiales.



(Figure 12) Fonctionnement d'une couche de convolution avec un stride de 1 et un padding de 0.

La figure 12 montre le début d'un procédé de de convolution avec un stride (il s'agit du pas de déplacement du noyau) de 1 et sans padding (pas de remplissage autour avec des 0). Voici les détails clés :

Input : Une grille 5x5. Les cases colorées représentent le noyau.
La transformation mathématique associée débute par un produit matriciel d'Hadamard entre la matrice du noyau et la sous matrice (de même taille) des valeurs de la grille que le noyau "regarde" puis la somme des 9 valeurs résultantes, ce qui donne la première valeur en output.
Output : L'output est une grille réduite à 3x3 après application de la convolution.
Ceci est dû à l'absence de padding et à un stride de 1, qui fait que le filtre de convolution se déplace d'un pas à chaque fois sans étendre les bords de l'entrée.
Cette configuration est typiquement utilisée pour maintenir la résolution spatiale proche de celle de l'entrée tout en capturant des caractéristiques locales.



(Figure 13) Fonctionnement d'une couche de convolution avec un stride de 2 et un padding de 1.

La figure 13 montre une couche de convolution avec un stride de 2 et un padding de 1. Les implications de cette configuration sont :
Input : Semblable au premier graphique, une grille 5x5 est utilisée comme entrée, mais cette fois avec un padding qui étend légèrement l'espace autour de l'entrée avec des 0.
Output : L'output est une grille 3x3, résultant de l'application d'un filtre de convolution qui se déplace de deux unités à chaque pas. Le padding permet de conserver une partie de la dimension spatiale qui serait autrement perdue en raison du stride plus grand.
Cette configuration est couramment utilisée pour réduire la dimensionnalité de l'entrée, ce qui est utile pour le discriminateur dans les GANs afin de progressivement condenser l'information spatiale en une décision binaire réelle ou fausse.

La convolution avec stride 1 et sans padding est utilisée pour préserver les détails spatiaux tout en capturant des caractéristiques à l'échelle locale. D'autre part, la convolution avec stride 2 et padding 1 est stratégiquement utilisée pour réduire la dimension spatiale, permettant ainsi au discriminateur de faire des généralisations plus larges à mesure qu'il classifie les images en tant que réelles ou générées.

2. **Activation LeakyReLU (`nn.LeakyReLU`)** : La fonction d'activation LeakyReLU est utilisée pour permettre un petit gradient même lorsque la valeur est inférieure à zéro (fuite), ce qui peut aider à maintenir l'activité des neurones et éviter le problème des neurones morts. La pente de la partie 'leaky' est définie à 0.2.

$$\text{Pour } x \in \mathbb{R},\ \text{LeakyReLU}(x) = \begin{cases} x & \text{si } x > 0 \\ 0.2x & \text{sinon} \end{cases}$$

3. **Normalisation par lots (`nn.BatchNorm2d`)** : Après certaines couches convolutives, une normalisation par lots est appliquée pour stabiliser l'apprentissage en normalisant les canaux, ce qui permet un entraînement plus rapide et plus stable.
4. **Dernière couche (`nn.Sigmoid`)** : La dernière couche convolutive est suivie par une fonction d'activation sigmoid pour ramener la sortie à une probabilité entre 0 et 1, indiquant si l'entrée est considérée comme réelle (sortie proche de 1) ou fausse (sortie proche de 0). La fonction est la suivante :
   Pour $x \in \mathbb{R}$, $\sigma(x) = \dfrac{1}{1 + e^{-x}}$

Processus de discrimination

L'architecture commence par prendre une image RGB de taille 64x64 et la passe à travers une série de convolutions qui réduisent sa taille tout en augmentant le nombre de caractéristiques détectées. À la fin, le discriminateur produit une seule valeur de probabilité, qui indique la confiance du discriminateur quant au fait que l'image analysée soit réelle.
Une formule s'obtient pour trouver la taille de sortie d'une couche de convolution :

$$\text{output\_size} = (I - K + 2P)/S + 1$$

Où I, K, S et P sont identiques à ceux définis pour le calcul de output_size d'une couche de convolution transposée.

On applique la formule pour le premier bloc : $I = 64$, $K = 4$, $S = 2$ et $P = 1$, on a donc que output_size = 32. De plus, on avait défini **ndf** filtres (ou canaux) en sortie et en rentrant une seule image dans la couche, on obtient en sortie de couche un tenseur de taille (1, **ndf**, 32, 32). Pour le deuxième bloc : $I = 32$, $K = 4$, $S = 2$ et $P = 1$, on a donc que output_size = 16. On obtient en sortie de couche un tenseur de taille (1, **ndf***2, 16, 16). Et ainsi de suite.

L'implémation du discriminateur est la suivante :

```
# Discriminator Code
class Discriminator(nn.Module):
    def __init__(self, nb_gpu):
        super(Discriminator, self).__init__()
        self.nb_gpu = nb_gpu
        self.main = nn.Sequential(
            # input is ``(nb_channels) x 64 x 64``

            nn.Conv2d(nb_channels, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``

            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*2) x 16 x 16``

            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*4) x 8 x 8``

            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
```

```
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``

            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)

# Create the discriminator
netD = Discriminator(nb_gpu).to(device)
netD.apply(weights_init)
```

On crée une instance de la classe Discriminator pour obtenir un générateur que l'on appelle netD et ses poids sont initialisés avec la fonction weights_init() de la même manière que pour le générateur.

I.5 Optimisateurs

```
# Define loss function
criterion = nn.BCELoss()
# Setup optimizers for both G and D, according to the DCGAN paper
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

Nous utilisons l'optimisateur Adam et la fonction de perte BCE conformément aux conventions du papier GAN original. En utilisant cette fonction de perte, le discriminateur est encouragé à attribuer des probabilités proches de 1 aux données réelles et des probabilités proches de 0 aux données générées. Elle est définie dans ce contexte comme suit :

$$\text{BCE}(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Où :

— y est la vraie étiquette (1 pour les données réelles, 0 pour les données générées).

— $\hat{y}$ est la sortie du discriminateur (prédiction) pour cette observation.

Le but est de minimiser la perte BCE.

I.6 Boucle d'apprentissage

Cette section du script explique la boucle d'entraînement pour le GAN, mettant en évidence les mécanismes clés pour l'actualisation des réseaux du générateur (G) et du discriminateur (D).

Mise à jour du réseau du discriminateur (D)

Le discriminateur est d'abord mis à jour avec un lot d'images réelles. Chaque image réelle est évaluée par D, qui tente de maximiser la probabilité de correctement identifier les images réelles comme telles (avoir $\log(D(x))\approx 1$).

Ensuite, un lot d'images fictives générées par le générateur est évalué par D. Dans ce cas, D vise à maximiser la probabilité d'identifier correctement ces images fictives comme étant fausses (avoir $\log(1 - D(G(z)))\approx 0$).

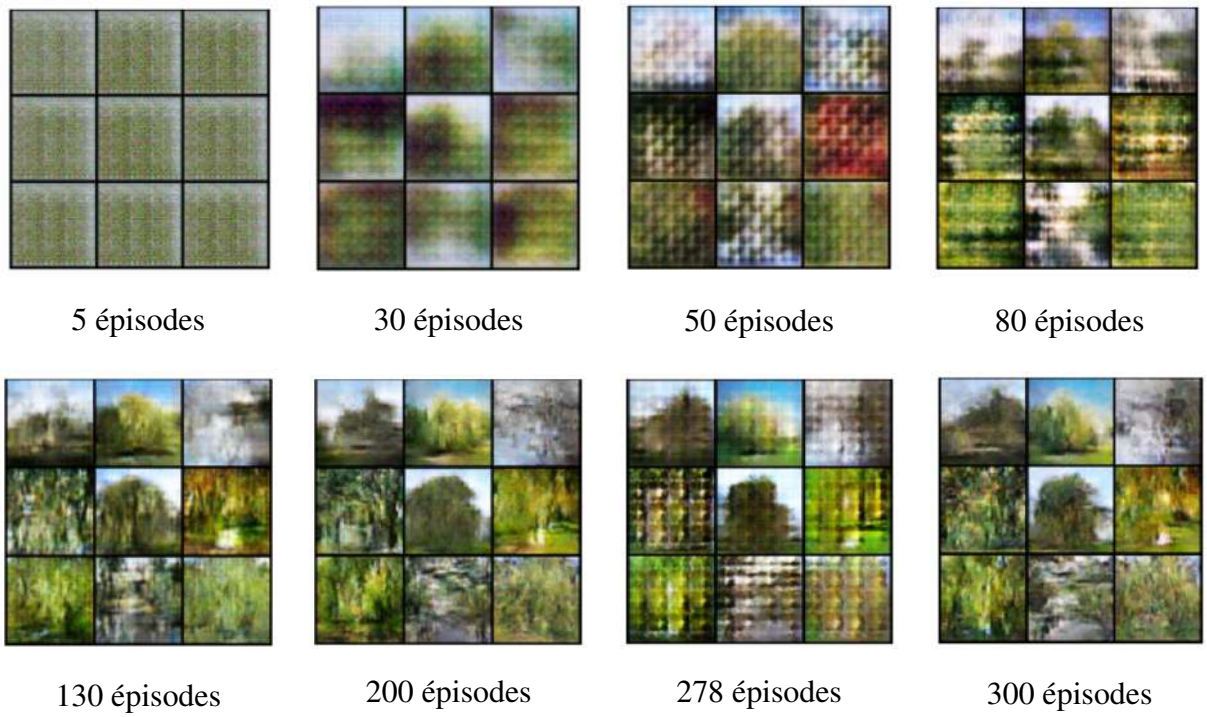
Mise à jour du réseau du générateur (G)

Le générateur G tente de minimiser $\log(1 - D(G(z)))$ en générant des images qui sont indiscernables des vraies pour le discriminateur. Cette étape est essentielle pour améliorer la capacité de G à créer des images réalistes.

II Entraînement

II.1 Évolution de la qualité des images au cours de l'entraînement

Dans la partie précédente, on a montré le code permettant la mise en place et l'entraînement de notre réseau GAN. Dans cette partie on va présenter les résultats et expliquer quelques pistes que nous avons explorées afin d'améliorer nos résultats.
On rappelle que l'on a entraîné le modèle sur 300 épisodes avec **batch_size**=64.



(Figure 14) Visualisation de l'évolution des images pendant l'entraînement.

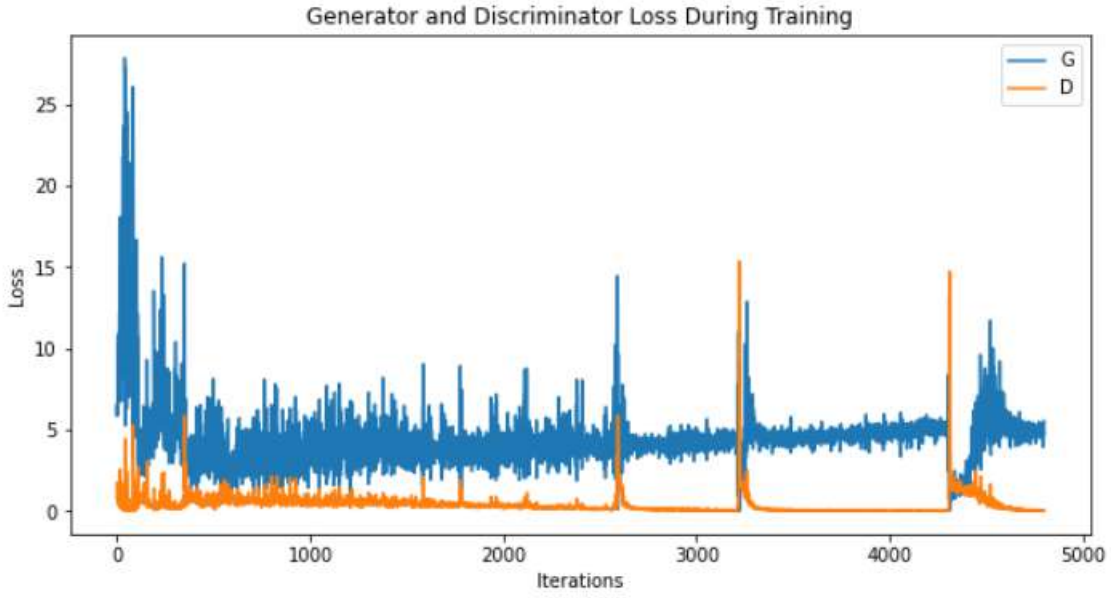
Début de l'entraînement (épisodes 1 à 5) : Les premières images peuvent être considérées comme du bruit. Cela est dû au processus d'apprentissage initial où le générateur produit des sorties aléatoires et non informatives, nécessaires pour comprendre les structures sous-jacentes des données.

Phase intermédiaire (épisodes 30 à 80) : Au fur et à mesure que le modèle continue d'apprendre, on observe une augmentation de la définition et de la diversité des textures et des couleurs. Les images générées commencent à évoquer les caractéristiques des saulaies, telles que la présence d'arbres et de végétation, bien qu'un quadrillage régulier de flou soit observable.

Approche de la convergence du modèle (épisodes 130 et plus) : Nous commençons à voir nos images converger vers un résultat acceptable à partir de l'épisode 130 de l'entraînement de notre modèle GAN. Cependant, nous remarquons également quelques moments d'instabilité, comme celui à l'épisode 278, où le processus d'entraînement a été momentanément perturbé. Ces variations sont courantes lors de l'apprentissage des GAN en raison de leur complexité et instabilité.

La progression observée démontre la capacité du modèle GAN (DCGAN) à apprendre et à simuler la complexité des saulaies plutôt rapidement. L'augmentation de la qualité visuelle au fil des épisodes indique que le modèle pourrait être utilisé pour générer des images qui servent d'inspiration ou de base pour des œuvres artistiques, conformément aux objectifs initiaux du projet.

II.2 Analyse des pertes du générateur et du discriminateur



(Figure 15) Évolution des pertes du générateur/discriminateur.

Une composante cruciale de l'évaluation de l'efficacité d'un modèle GAN réside dans l'analyse des courbes de perte durant l'entraînement. Dans le cadre de notre recherche, l'observation des courbes de pertes du générateur (G) et du discriminateur (D) au fil des itérations offre une perspective quantifiable de la dynamique d'apprentissage du modèle DCGAN.
Mathématiquement, les tendances observées en figure 15 sont cohérentes avec l'objectif de minimax décrit dans la formulation des GANs, où G minimise la probabilité que D identifie correctement les images générées comme fausses, tandis que D maximise cette probabilité. L'oscillation des pertes, particulièrement notable dans les premières itérations, est typique de la nature du jeu minimax où chaque réseau affine ses paramètres en réponse à l'autre.

La phase initiale de l'entraînement est marquée par une volatilité élevée, traduisant une phase d'adaptation où G et D s'efforcent de trouver une stratégie viable. Les pics importants dans la perte du générateur signalent des tentatives de création d'images qui parviennent à tromper temporairement le discriminateur.
Avec la poursuite de l'entraînement, une tendance à la stabilisation des pertes est observable. Cette stabilisation reflète l'ajustement progressif des paramètres du modèle, permettant au discriminateur d'améliorer sa précision et au générateur d'augmenter la qualité des images produites.

Les pics occasionnels dans les pertes, en particulier pour le générateur, indiquent des mo-

ments où le générateur a trouvé des astuces pour tromper le discriminateur temporairement, ce qui entraîne une augmentation de la perte du discriminateur jusqu'à ce qu'il s'ajuste à nouveau. Cela ne signifie pas qu'il produit de meilleures images au sens visuel.

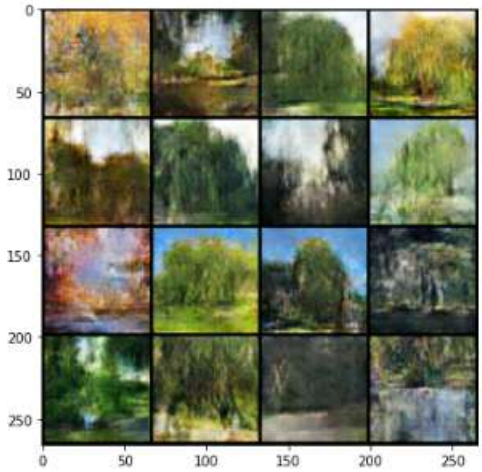
On constate que le modèle a bien appris aux alentours de 150 épisodes et que continuer d'entraîner le modèle ne fournira pas forcément de meilleures images : il y'aura peut-être des instabilités qui seront parfois corrigées par le modèle et d'autres fois pas, ce qui impliquerait dans ce dernier cas une génération d'images de moins bonne qualité voire vraiment mauvaise (phénomène observé en expérimentation).
Il est important de noter que l'équilibre entre le générateur et le discriminateur ne signifie pas nécessairement que les deux pertes convergeront à un point identique. Un équilibre sain dans un GAN est atteint lorsque les deux réseaux sont suffisamment bons pour se défier continuellement, ce qui peut se refléter par des pertes fluctuantes mais relativement stables. C'est en particulier le cas ici où à partir d'un certain stade, la perte du générateur oscille autour de 5 et celle du discriminateur avoisine 0.

N.B : On lit un peu moins de 5000 itérations alors qu'on avait réalisé 300 épisodes, c'est normal car avec un **batch_size** de 64 et 1000 images dans la base de données, on réalise 16 itérations (une itération avec un mini lot de 40 pour chaque épisode) par épisode ce qui donne 4800 itérations.

III Résultats

III.1 Images générées par le DCGAN

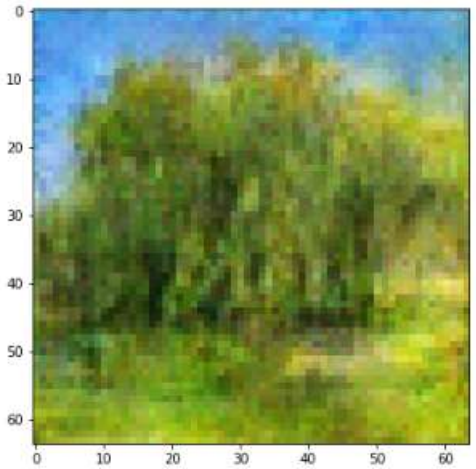
Voici une génération d'images possible par notre DCGAN entraîné :



(Figure 16) Génération de 9 images artificielles par notre DCGAN.

Les images produites sont en couleur de dimension 64x64. De loin, et en sachant que le but est de générer des saules, on peut parfois arriver à visualiser des saules. On constate aussi une diversité parmis les images générées : même si les saules ne sont pas toujours reconnaissables , on peut distinguer un ciel nuageux, un ciel bleu, les couleurs spécifiques des différentes saisons (été, printemps, hiver et autonme), des lacs et plus encore...

Affichons l'image située dans le carré central en bas à gauche :



(Figure 17) Image artificielle de saule générée par notre DCGAN.

L'image générée peut effectivement servir de base pour une œuvre mais le manque de détails est assez préoccupant et il est naturel de vouloir améliorer encore nos résultats. La faible résolution (64x64) nous pose problème, c'est ce qui empêche l'image de contenir des détails.

III.2 Augmentation de la résolution

Une première piste a été d'utiliser une méthode de super-résolution pour augmenter la résolution de nos images. Il s'agit de méthodes d'interpolation, de deep learning et plus encore pour créer de l'information dans nos images pour augmenter la résolution de ces dernières.
Notre choix s'est tourné vers le GAN ESRGAN, un GAN de super-résolution disponible sur internet et dont les poids ont déjà été entraînés par les créateurs.
Cependant nos images manquaient tellement d'information qu'on obtenait en sortie une image de meilleure résolution, mais sans gain d'information : pour une valeur (a, b, c) de pixel dans l'image originale, un carré de taille ixi (i raisonnablement petit, dépend de la taille de la nouvelle image) de pixels de valeur (a, b, c) était créé dans la nouvelle image et ce pour chaque pixel de l'image originale, ce qui est donc inutile.

(Figure 18) Comparaison sortie DCGAN/ESRGAN.

On voit que l'image de droite (produite par le ESRGAN) est plus nette (et également de meilleure résolution par définition) mais il n y a pas plus d'information.

On a donc considéré par la suite les sorties de notre DCGAN comme des images de saules avec un floutage de type pixelisé et le but est maintenant de les dépixeliser.

Dépixelisation

On a opté pour **letsenhance.io** pour dépixeliser nos images. Une méthode par deep learning est utilisée pour dépixeliser les images en laissant une IA les modifier, ce qui crée de l'information. Le facteur de "créativité" est contrôlable et on l'a laissé petit pour ne pas laisser l'IA créer des détails trop fantaisistes sur nos images.
Selon les développeurs, la technologie de super-résolution à été utilisée. Le réseau de neurones, formé sur une large base de photographies réelles, apprend à restituer les détails et à conserver des lignes et des contours clairs, en s'appuyant sur sa connaissance des objets et textures typiques qui existent dans le monde réel.

Prenons l'image suivante :

(Figure 19) Image en sortie de DCGAN.

Que nous donnons à l'IA de dépixelisation pour obtenir cette image :

(Figure 20) Image en sortie de dépixelisation.

Le résultat est très satisfaisant, la nouvelle taille d'image est 924x924. Les modifications s'approchent de la réalité et l'image produite est assez bien détaillée.

Avec ces images, on peut désormais utiliser ESRGAN pour encore augmenter la résolution et les rendre plus réalistes étant donné qu'il y a suffisament de détails dans ces nouvelles images.

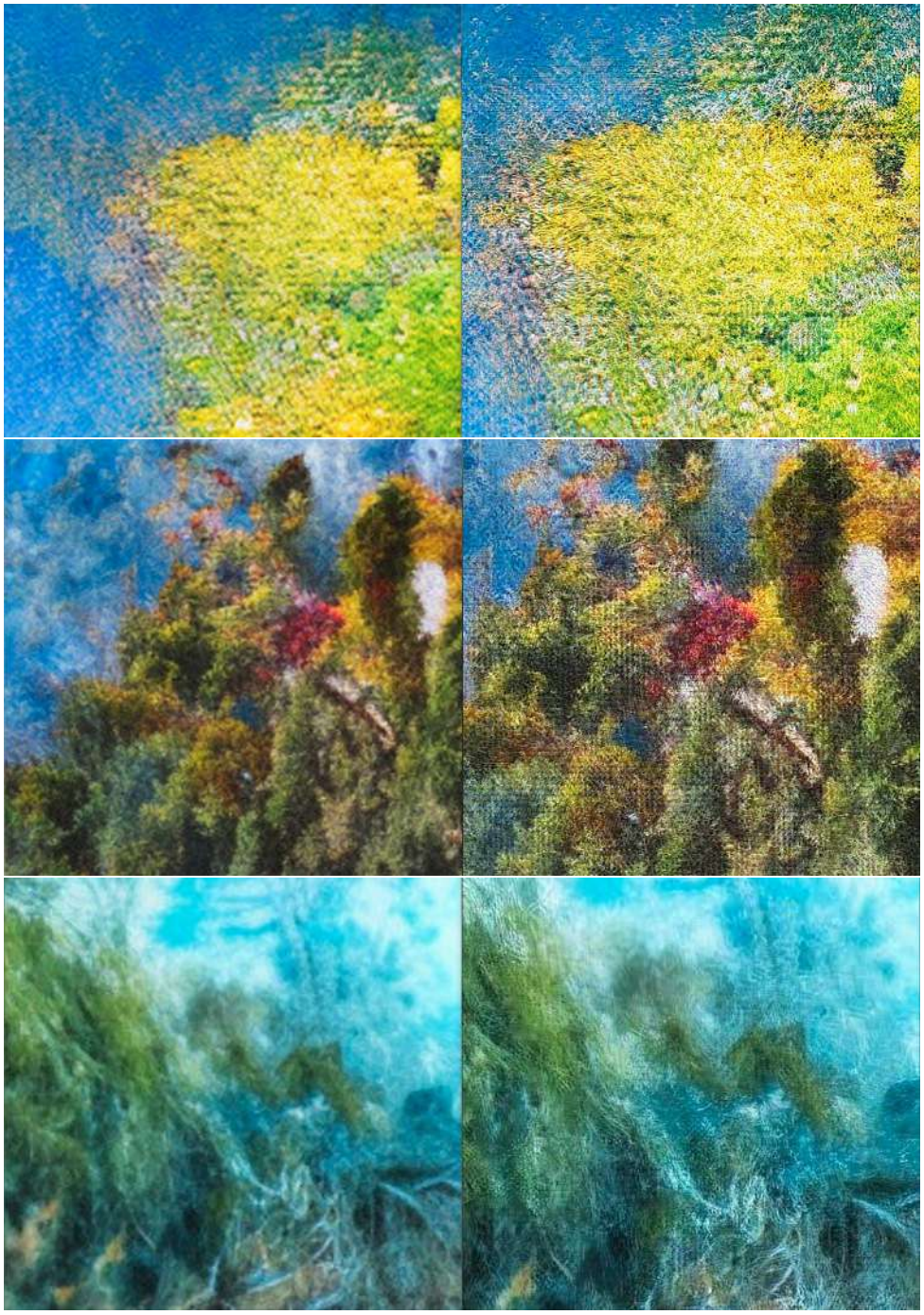
ESRGAN pour augenter la résolution

Nous donnons cette même image dépixelisée au réseau de neurones ESRGAN.

(Figure 21) Image dépixelisée en sortie de ESRGAN.

Il faut zoomer manuellement sur l'image pour apercevoir plus de détails.
Nous allons zoomer sur la partie gauche supérieure des deux images pour mettre en évidence la différence de netteté.

Voici quelques zoom sur nos images pour comparer les résultats :



(Figure 22) Zoom de différentes images dépixelisée avant/après ESRGAN.

En sortie de ESRGAN, les images sont en très haute définition (2608x2608 ou 3696x3696) et de meilleure qualité.

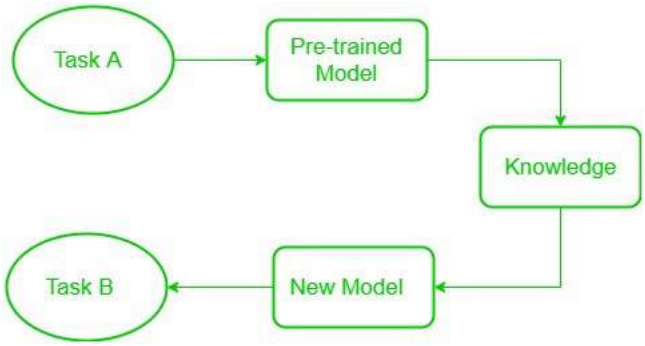
CHAPITRE 3

Pour aller plus loin

I Transfert d'apprentissage

I.1 Qu'est-ce que le transfert d'apprentissage ?

Le transfert d'apprentissage est une technique d'apprentissage automatique où un modèle pré-entraîné sur une tâche est utilisé comme point de départ pour un autre modèle sur une deuxième tâche. C'est particulièrement utile dans les situations où les données disponibles pour la deuxième tâche sont limitées.



(Figure 24) Illustration du transfert par apprentissage.

I.2 Théorie mathématique du transfert d'apprentissage

Supposons que nous ayons deux tâches d'apprentissage, une tâche source S et une tâche cible T. Chaque tâche est définie par une distribution de probabilité sur un ensemble de données. Pour la tâche source, nous avons une distribution $P_S(X_S, Y_S)$ et pour la tâche cible, nous avons une distribution $P_T(X_T, Y_T)$.
Le but du transfert d'apprentissage est de tirer profit de la distribution source P_S pour améliorer l'apprentissage sur la distribution cible P_T. Dans la pratique, cela se fait généralement en entraînant un modèle sur la tâche source et en utilisant ensuite les paramètres de ce modèle comme initialisation pour un modèle sur la tâche cible.
Mathématiquement, on peut représenter un modèle d'apprentissage automatique comme une fonction f_θ, où θ est un ensemble de paramètres. L'objectif de l'apprentissage est de trouver les paramètres θ qui minimisent une certaine fonction de perte L sur l'ensemble de données. Pour la tâche source, cela peut être écrit comme :

$$\theta_S = \arg\min_{\theta} \mathbb{E}_{(x,y)\sim P_S}[L(f_\theta(x), y)]$$

Et pour la tâche cible :

$$\theta_T = \arg\min_{\theta} \mathbb{E}_{(x,y)\sim P_T}[L(f_\theta(x), y)]$$

Dans une approche d'apprentissage traditionnelle, les tâches source et cible sont apprises indépendamment, et donc θ_S et θ_T sont optimisés séparément. Dans le transfert d'apprentissage, cependant, on commence par trouver θ_S et on utilise ensuite θ_S comme initialisation pour trouver θ_T.
Cela peut être écrit comme :

$$\theta_T = \arg\min_{\theta} \mathbb{E}_{(x,y)\sim P_T}[L(f_{\theta_S+\Delta\theta}(x), y)]$$

où $\Delta\theta$ est une mise à jour des paramètres à partir de θ_S basée sur les données de la tâche cible.

I.3 Importance du transfert de caractéristiques

Un aspect clé du transfert d'apprentissage est le transfert de caractéristiques (feature transfer). Dans les réseaux de neurones profonds, les couches inférieures apprennent souvent à détecter des caractéristiques de bas niveau (comme les bords dans les images), tandis que les couches supérieures apprennent à détecter des caractéristiques de plus haut niveau (comme les formes ou les objets). Lors du transfert d'apprentissage, les paramètres des couches inférieures, qui ont appris des caractéristiques de bas niveau sur la tâche source, sont souvent transférés à la tâche cible, et seulement les paramètres des couches supérieures sont ajustés pour la tâche cible. Cela permet de bénéficier des caractéristiques de bas niveau apprises sur la tâche source, même lorsque les données de la tâche cible sont limitées.
On note que le transfert d'apprentissage est particulièrement efficace lorsque les tâches source et cible sont similaires, c'est-à-dire lorsque P_S et P_T ont des supports qui se chevauchent ou sont proches. Si les tâches sont très différentes, le transfert d'apprentissage peut même être nuisible, car les caractéristiques apprises sur la tâche source peuvent ne pas être pertinentes pour la tâche cible.

I.4 Choix du modèle pré-entraîné

Pour ce projet, il est possible aussi d'utiliser un GAN pré-entraîné sur un large ensemble de données d'images. Comme nous voulons générer des images de saulaies, un modèle pré-entraîné sur des images de plantes ou de paysages naturels serait un excellent choix.
Un exemple de tel modèle pourrait être le styleGAN2, qui a été formé sur un grand ensemble de données d'images variées et a démontré une excellente performance dans la génération d'images de haute qualité.

I.5 Adaptation des données du projet

Aprés le choix d'un modèle pré-entraîné, la prochaine étape est de l'adapter à nos données spécifiques. Cela se fait généralement en utilisant une technique appelée "fine-tuning", où le modèle est d'abord formé sur les données pré-existantes, puis il est légèrement ajusté sur les nouvelles données.
Dans notre cas, l'utilisation de StyleGAN2 pré-entraîné, puis l'ajustement sur notre ensemble de données spécifiques de saulaies permettrait au modèle d'apprendre les caractéristiques spécifiques à partir des données collectées.

I.6 Limites du transfer learning

Le transfer learning, bien que bénéfique dans des contextes où les données d'entraînement sont limitées ou lorsque le temps d'entraînement doit être réduit, présente des défis significatifs pour notre objectif de générer des images de saulaies :

Disparité des données : Les modèles pré-entraînés sont souvent optimisés pour des ensembles de données larges et génériques. Cependant, les saulaies présentent des caractéristiques uniques qui peuvent ne pas être bien représentées dans ces ensembles de données standards, conduisant à des transferts de connaissances inefficaces.

Adaptabilité du modèle : L'utilisation de modèles pré-entraînés impose des contraintes architecturales qui peuvent limiter la capacité du modèle à s'adapter aux particularités des images de saulaies, entravant la créativité et la qualité de la génération d'images.

Surapprentissage : Le transfer learning peut conduire à un surapprentissage si le modèle pré-entraîné est trop spécifique à son ensemble de données d'origine, ce qui réduit sa capacité à généraliser à de nouvelles tâches telles que la génération d'images de saulaies.

Complexité computationnelle : Les modèles pré-entraînés sont souvent volumineux et complexes, nécessitant des ressources computationnelles considérables qui dépassent les capacités disponibles pour notre projet. Il en est de le même pour le stockage mémoire où les poids d'un tel réseau sont très couteux en mémoire. Pour information, les poids de notre DCGAN pesaient 25 Mo, les poids d'ESRGAN pesaient 65 Mo et après une tentative infructueuse de DCGAN pour générer des images de taille 128x128, les poids de ce réseau pesaient déjà 350 Mo.
De plus, sans GPU, les calculs prenaient environ 3-4h pour entraîner notre DCGAN, avec GPU c'était au moins 2 fois plus rapide mais un seul membre du groupe en possédait un. Les temps de calcul ne seront pas raisonnables avec le matériel que l'on possède pour des infrastructures plus grosses.

Face à ces défis, DCGAN s'est avéré être l'alternative la plus appropriée.

II Synthèse et conclusions de l'étude

Cette recherche a exploré les frontières de la génération d'images artificielles, en se concentrant spécifiquement sur les saulaies, à travers l'implémentation et l'entraînement de Réseaux Génératifs Adversariaux (GANs). En optant pour une approche sans transfer learning, le projet visait à produire des images qui capturent non seulement l'esthétique mais aussi la complexité écologique des saulaies, tout en s'engageant dans une étude approfondie des capacités et des défis associés à cette technologie d'IA.

Développement du modèle GAN

Le modèle GAN a été construit avec une architecture soigneusement calibrée, inspirée par les meilleures pratiques établies dans la littérature, mais adaptée aux exigences uniques de la tâche. Les choix techniques, y compris l'initialisation des poids et les stratégies d'optimisation, ont été méticuleusement considérés pour optimiser la performance et l'efficacité.

Progression de l'apprentissage

L'entraînement du modèle a été documenté à travers des époques significatives, révélant une progression tangible dans la qualité des images générées. Les premières époques ont montré des images primitives, alors que les époques intermédiaires ont commencé à refléter une meilleure définition et diversité. Finalement, les dernières époques, notamment l'époque 300, ont produit des images avec des détails remarquablement affinés et une ressemblance accrue avec les paysages réels de saulaies.

Analyse qualitative des résultats

La qualité des images générées a été évaluée qualitativement, notant des améliorations significatives dans la précision des textures, la cohérence de la composition, et la variabilité des scènes. Ces images démontrent la capacité du modèle à apprendre des caractéristiques clés des saulaies et à les réinterpréter de manière créative, fournissant ainsi une nouvelle fenêtre sur la représentation numérique de ces environnements naturels.

Réflexions sur le transfer learning

Bien que le transfer learning offre des avantages dans certains contextes, cette étude a révélé que la construction d'un modèle spécifique peut être plus avantageuse pour des tâches hautement spécialisées et artistiques. Cette approche a permis un contrôle plus précis sur le processus d'apprentissage et a évité les problèmes potentiels de surapprentissage ou de biais introduits par des données non pertinentes. De plus, cette décision a été renforcée par la nécessité de concorder avec les ressources computationnelles disponibles, qui sont limitées dans notre contexte de recherche académique.

Fermeture

Dans notre étude, nous avons suivi une approche méthodique, depuis la conception initiale du modèle jusqu'à l'évaluation de sa performance en passant par l'entraînement et l'optimisation

détaillés, en présentant un cas d'usage unique où les GANs sont appliqués avec succès pour générer des représentations de saulaies qui allient fidélité artistique et richesse écologique. Les résultats obtenus ouvrent la voie à de futures recherches et applications potentielles, allant de l'aide à la conception paysagère à la sensibilisation environnementale.

Liens utiles

- **GitHub ESRGAN :**
  https://github.com/xinntao/ESRGAN?tab=readme-ov-file
- **letsenhance.io :**
  https://letsenhance.io/
- **Explication convolution1 :**
  https://towardsdatascience.com/understand-transposed-convolutions-and-build-your-own-transposed-convolution-layer-from-scratch-4f5d97b2967
- **Explication convolution2 :**
  https://towardsdatascience.com/what-is-transposed-convolutional-layer-40e5e6e31c11
- **Explication convolution3 :**
  https://www.youtube.com/watch?v=581X9wsnWJs&ab_channel=CNRS-FormationFIDLE
- **Explication convolution4 :**
  https://www.youtube.com/watch?v=96_oGE8WyPg&ab_channel=JorisvanLienen
- **Explication convolution5 :**
  https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8
- **Notes DCGAN :**
  https://arxiv.org/pdf/1511.06434.pdf

Bibliographie

[1] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative Adversarial Nets. Dans *Advances in Neural Information Processing Systems (NIPS 2014)*. Ce papier introduit les GANs, proposant un nouveau cadre pour estimer des modèles génératifs via un processus adversarial.

[2] Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. Pré-imprimé arXiv :1511.06434. Ce travail, souvent cité sous le nom de DCGAN, a introduit des améliorations architecturales clés pour les GANs, améliorant la stabilité de l'entraînement des GANs profonds.

[3] Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. Pré-imprimé arXiv :1701.07875. Ce papier présente le WGAN, une amélioration des GANs qui utilise la distance de Wasserstein comme fonction de coût pour améliorer la stabilité de l'entraînement.

[4] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2018). Progressive Growing of GANs for Improved Quality, Stability, and Variation. Pré-imprimé arXiv :1710.10196. Ce travail propose une méthode pour entraîner les GANs en augmentant progressivement la taille du générateur et du discriminateur, ce qui permet de produire des images de haute résolution.

[5] Brock, A., Donahue, J., & Simonyan, K. (2019). Large Scale GAN Training for High Fidelity Natural Image Synthesis. Pré-imprimé arXiv :1809.11096. Ce papier, connu pour BigGAN, explore l'entraînement des GANs à grande échelle pour améliorer la qualité et la diversité des images générées.

[6] Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. Dans *IEEE International Conference on Computer Vision (ICCV 2017)*. Ce travail introduit CycleGAN, une méthode pour la traduction d'image à image sans appariement d'images, démontrant l'applicabilité des GANs dans des tâches de traduction d'image complexes.