

Rapport sur le Projet CycleGan

Paulin Brissonneau Thomas Kebaili Ilyas Moutawwakil
Valentin Laurent Lilian Lecomte

XX mai 2020

Table des matières

Introduction	3
1 Le Multiperceptron	4
1.1 Le perceptron simple	4
1.2 Le multiperceptron	4
1.3 Implémentation et résultats	4
2 Les réseaux à convolutions	5
2.1 Présentation générale	5
2.1.1 Introduction	5
2.1.2 Structure d'un CNN	5
2.1.3 Principe général	6
2.2 Formalisation d'un CNN	6
2.2.1 Le produit de convolution : un outil indispensable	6
2.2.2 Le balayage de l'image : une histoire de Padding et de Stride	7
2.2.3 Généralisation en 3D	8
2.2.4 Couche de pooling	8
2.2.5 Structure complète d'un CNN	8
2.2.6 Apprentissage d'un CNN	9
2.3 Implémentation et résultats	9
3 Les GAN (Réseaux Adverses Génératifs)	10
3.1 Principe général des GAN	10
3.2 Le DCGAN (Deep Convolutionnal Adversarial Network)	10
3.3 Étude de la convergence des GAN	11
3.3.1 L'effondrement des modes	11
3.3.2 Perte de l'équilibre	13
3.4 Cadre théorique et WGAN	13
3.4.1 Approche bayésienne des GAN	14
3.4.2 DCGAN	14
3.4.3 WGAN	14
3.5 Implémentation et résultats	14
4 Le cycleGAN	16
4.1 Présentation de la problématique	16
4.2 Principe général du cycleGAN	18
4.3 Les fonctions de coûts	20
4.4 Les métriques d'évaluations	21

4.5	Implémentation et résultats	22
4.5.1	Détails d'implémentation	22
4.5.2	Quelques résultats	24
4.5.3	Limitations et ouverture	25
	Conclusion	27

Introduction

L'objectif de ce projet est de comprendre, maîtriser, et utiliser la technologie des cycleGAN, proposée pour la première fois par Zhu et al. [1]. Un cycleGAN est un algorithme particulier de traitement des images, qui prend la forme d'un réseau de neurones. La problématique à laquelle répond le cycleGAN est celle du transfert de style, cela signifie que l'on cherche à travailler une donnée structurée pour en modifier l'apparence globale. Par exemple, la transformation des objets d'une image, le changement de style pictural ainsi que le changement de style musical sont des transferts de style, et peuvent être abordés par l'utilisation d'un cycleGAN. C'est un problème particulièrement difficile pour un algorithme, en particulier lorsque les données de sont pas appairées d'un style à un autre.

Le projet s'articule autour des cycleGAN, cependant ceux-ci reposent grandement sur la famille d'algorithmes des GAN introduite par Ian Goodfellow [2], qui reposent eux-même sur beaucoup d'autres concepts de *machine learning*. C'est pourquoi, pour comprendre les cycleGAN, nous devons d'abord passer par plusieurs autres étapes importantes. Nous poserons d'abord les bases du *machine learning*, en partant du simple **perceptron multicouches** (**Chapitre 1**). Ensuite nous étudierons la spécificité des **couches à convolutions, ou CNN** (**Chapitre 2**) indispensables au traitement des données structurées compositionnelles telles que les images, et développées notamment par Yann LeCun [3]. Puis nous nous intéresserons aux **GAN ou Réseaux Adverses Génératifs** (**Chapitre 3**). Il existe plusieurs architectures de GAN, nous en verrons les deux types principaux : les DCGAN [4] et les W-GAN [5]. Enfin, grâce à tous ces outils, nous pourrons comprendre le fonctionnement des **cycleGAN** (**Chapitre 4**). Tous ces points sont accompagnés de l'implémentation des algorithmes sur TensorFlow 2.0 [6].

Ce rapport constitue un résumé des connaissances que nous avons acquises, et sur lesquelles nous nous reposerons pour mener à bien la suite du projet : appliquer la technologie des cycleGAN à un problème à notre choix.

Chapitre 1

Le Multiperceptron

1.1 Le perceptron simple

Le principe du perceptron est bla bla bla

1.2 Le multiperceptron

Le principe du multiperceptron est bla bla bla

1.3 Implémentation et résultats

Le principe du multiperceptron est bla bla bla

Chapitre 2

Les réseaux à convolutions

2.1 Présentation générale

2.1.1 Introduction

L'architecture d'un multiperceptron se prête bien à des applications simplistes comme de la reconnaissance de chiffre. Cependant, dès que la complexité des données augmente ne serait-ce que légèrement, le MLP devient innefficace en un temps raisonnable pour pouvoir reconnaître des objets . Heureusement, il existe une structure bien plus efficace qui est parfaitement adaptée à la reconnaissance d'image : le **CNN (convolutional neural network)**. L'idée de cette architecture a été introduite en 1980 [ref doc1] et a ensuite été améliorée pour aujourd'hui donner naissance à des réseaux comme AlexNet ou Inception. [ref doc]

2.1.2 Structure d'un CNN

La structure d'un CNN se rapproche de celle d'un multiperceptron dans la mesure où celui-ci est aussi organisé sous forme de couches comme le montre la figure 2.1. Cependant, certaines couches, nommées couche de convolution, ont un fonctionnement radicalement différent de celui d'une couche dense.

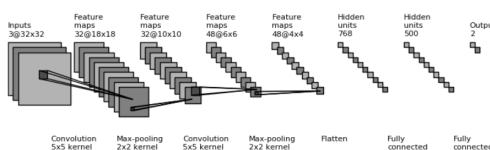


FIGURE 2.1 – Structure simplifiée d'un CNN.

Ainsi, l'image en entrée est d'abord traitée par une succession de **couches de convolutions** suivies d'une **couche de pooling**. Ensuite ce processus se répète un certain nombre de fois et finalement le résultat passe par une couche dense(plus rarement plusieurs) afin de donner la bonne classification.

2.1.3 Principe général

Le principe directeur se cachant derrière les couches de convolution se base sur le fonctionnement même de notre vision : on dispose d'un noyau(pattern) capable de reconnaître un motif en particulier. Il suffit alors de balayer l'image pixel par pixel(notion expliquée plus en détails ultérieurement) pour savoir où se trouvent précisément certains motifs sur l'image. En répétant ce procédé avec un grand nombre de noyaux différents, nous sommes dans la possibilité de pouvoir identifier la présence ou non de certains motifs caractéristiques de l'objet à reconnaître.

Nous disposons ainsi d'une "carte" des différents motifs qu'il nous est possible de réduire en taille(étape de pooling) pour réduire le nombre de calculs à effectuer. Le processus se répète un certains nombre de fois jusqu'à ce que l'on considère que les données soient suffisamment bien réduites pour qu'un MLP puisse s'en charger.

2.2 Formalisation d'un CNN

2.2.1 Le produit de convolution : un outil indispensable

Le coeur de l'efficacité d'un CNN repose sur le produit de convolution. On assimile l'image à une matrice notée I (nous la supposons pour l'instant en niveau de gris, de telle sorte que la matrice est en 2D mais la généralisation en 3D se fait aisément). De même, le noyau sera noté sous la forme d'une matrice K . L'opération de convolution s'écrit alors :

$$\forall(x, y) \in [|0, W_I|] \times [|0, H_I|], (N \otimes I)_{x,y} = \sum_{i=0}^{W_K} \sum_{j=0}^{H_K} K_{i,j} \times I_{x-i, y-j}$$

Où l'on a posé W_I, H_I la taille de l'image et W_K, H_K la taille du noyau.

Voici quelques exemples de l'effet du produit de convolution avec plusieurs noyaux :

Noyau	Résultat
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	

FIGURE 2.2 – Effets de la convolution avec quelques noyaux.

Ainsi, cette opération nous permet de mettre en évidence des motifs élémentaires en activant la case correspondante si le motif est présent au niveau de celle-ci.

2.2.2 Le balayage de l'image : une histoire de Padding et de Stride

Pour pouvoir évaluer toute l'image, il va falloir la balayer avec le noyau. De manière assez logique, on commence par le placer sur le coin supérieur gauche de l'image, puis on effectue un produit de convolution. Ensuite, il suffit de faire glisser latéralement le noyau d'un pixel et de répéter l'opération. Une fois au bout de la ligne, on descend d'un pixel et on repart à gauche.

Voici un exemple de balayage d'un noyau sur une image :

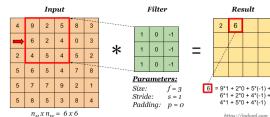


FIGURE 2.3 – Fonctionnement de la couche de convolution sur un exemple simple.

On remarque alors que le résultat à une dimension plus petite que l'image d'origine : on a ici effectué ce que l'on appelle un **simple padding**. Cependant, on pourrait souhaiter que le résultat ait la même dimension : cela est rendu possible grâce au **same padding**. Pour cela, nous pouvons rajouter des gardes autour de l'image d'origine : on rajoute des zéros autour puis on effectue l'opération de convolution comme présenté sur cette image :

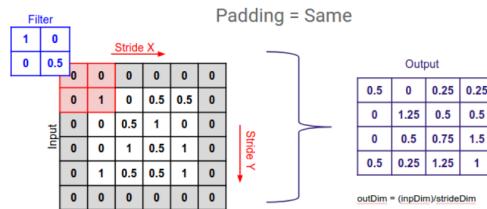


FIGURE 2.4 – Fonctionnement de la couche de convolution avec du same padding.

Il existe encore un paramètre permettant de personnaliser l'opération : le **stride**. Celui correspond au décalage à utiliser pour le noyau :

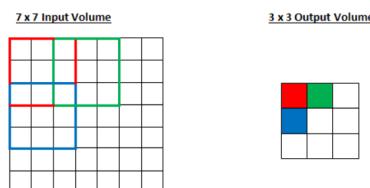


FIGURE 2.5 – Fonctionnement de la couche de convolution avec un stride valant 2.

L'avantage d'avoir un stride plus grand que 1 est aussi son désavantage : en effet il va permettre d'avoir un résultat en sortie de plus petite dimension mais en contrepartie, il y a une perte d'information.

2.2.3 Généralisation en 3D

Les images en couleurs peuvent être représentées par une matrice 3D de profondeur 3. Pour pouvoir gérer ce cas, nous prenons des noyaux de la même profondeur que l'image, puis nous appliquons séparément le produit de convolution sur les profondeurs correspondantes, le résultat final n'étant que la somme des résultats sur chaque profondeur.

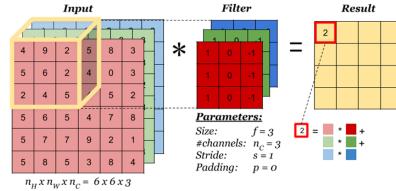


FIGURE 2.6 – Structure simplifiée d'un CNN

Notons cependant que ce résultat est très important car même si l'on considère que l'image n'a qu'une seule profondeur (image en niveau de gris par exemple), les données en entrée sur les autres couches de convolution sont dans l'immense majorité des cas en 3 dimensions. De manière intuitive, il est important de comprendre que la généralisation en 3D permet de détecter des motifs tridimensionnels au même titre qu'une opération de convolution simple permet d'identifier la présence d'un motif bidimensionnel.

2.2.4 Couche de pooling

Les couches de pooling permettent de réduire la taille de la sortie au prix d'une perte d'information. Elles sont nécessaires dans la mesure où le nombre de calcul à effectuer sans elles rendrait le CNN totalement inexploitable. On considère généralement deux types de couches de pooling :

- max pooling : on applique un équivalent d'un noyau d'une opération de convolution qui ne retient que la valeur maximale de la zone sur laquelle il effectue les calculs. C'est la couche de pooling la plus utilisée.
- average pooling : le fonctionnement est le même que la couche de max pooling à l'exception que l'on applique la fonction moyenne au lieu de la fonction maximum.

Il est à noter que la taille des noyaux est une variable et que plus celle-ci est grande, plus la taille de la sortie sera petite.

2.2.5 Structure complète d'un CNN

La reconnaissance d'objet est le plus souvent complexe et nécessite de nombreux motifs à déceler pour pouvoir être efficace. Ainsi, si l'on avait un motif par couche, la taille du CNN serait gigantesque. Pour éviter cela, il suffit d'associer à chaque couche plusieurs noyaux (généralement une puissance de 2). En appliquant les processus précédant pour chacun des noyaux, on se retrouve avec un nombre de matrice en sortie égal au nombre de noyaux. On se contente alors de les empiler en rajoutant une dimension supplémentaire : on obtient alors en sortie de couche de convolution un bloc de profondeur le nombre de noyau

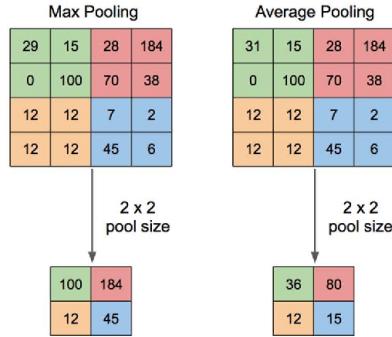


FIGURE 2.7 – Comparaison sur un exemple simple entre la couche de max pooling et la couche avg pooling.

contenant dans chacune des couches le résultat du produit de convolution pour un noyau.

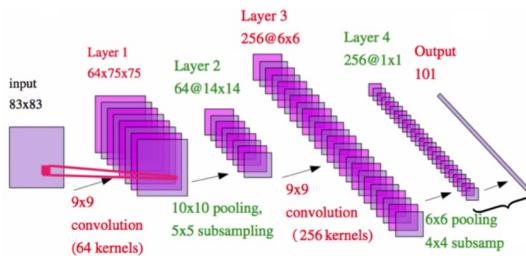


FIGURE 2.8 – Structure complète d'un CNN : la première couche de convolution contient 64 noyaux et la deuxième 256.

On alterne ainsi entre couches de convolution pour prélever des motifs de plus en plus complexes et couches de pooling pour réduire la quantité de calculs. Les données étant ainsi pré-traitées, il suffit de les mettre en entrée d'une couche dense pour finalement avoir le résultat escompté.

2.2.6 Apprentissage d'un CNN

Le CNN, de par sa structure analogue à celle d'un MLP, a besoin d'une phase d'apprentissage dans le but d'apprendre à reconnaître les motifs intéressants. Cela se fait par un processus de backpropagation. Cette partie étant essentiellement calculatoire, nous ne présenterons pas les opérations exactes à effectuer dans cette section mais nous vous invitons fortement à consulter le site suivant [<https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/>] pour de plus amples informations.

2.3 Implémentation et résultats

Blablabla

<https://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf>

Chapitre 3

Les GAN (Réseaux Adverses Génératifs)

3.1 Principe général des GAN

Le principe général des GAN repose sur l'utilisation de deux réseaux, ayant des objectifs contraires, on dit qu'ils sont **adversaires**. Le premier réseau transforme du bruit en image, c'est le **générateur** (G). Le deuxième réseau prend en entrées des images et les classe en deux catégories, en leur associant leur probabilité d'être issues de la base de donnée : c'est donc un classifieur binaire, il est appelé **discriminateur** (D). Le plus souvent, le discriminateur sera alimenté par des images de deux sortes : celles provenant de la base de donnée (images réelles), et celles générées par le générateur, on rôle sera donc de dire si une image est réelle ou générée. Il s'agit ensuite d'entraîner G afin qu'il maximise la probabilité que D fasse une erreur, et D la justesse de sa classification.

L'architecture des GAN a été introduite pour la première fois par Ian Goodfellow [ref] en 2014. Cet article innovant montrait déjà un gain de performance pour la génération d'images suivant une base de donnée. Mais l'atout majeur des GAN sont leur adaptabilité à tous types de données.

3.2 Le DCGAN (Deep Convolutionnal Adversarial Network)

Le DCGAN est la première architecture de GAN qui a été proposée [ref GoodFellow]. Le générateur et le discriminateur sont tous les deux des **réseaux à convolutions** [ref].

Cette architecture est caractérisée par les fonctions de coût de G et D. La fonction de coût de G à minimiser est la suivante :

$$\mathcal{L}_{DCGAN}(G, D, p_{\text{data}}, p_{\text{bruit}}) = \mathbb{E}_{x \sim p_{\text{data}}}(\log(D(x))) + \mathbb{E}_{z \sim \text{bruit}}(\log(1 - D(G(z))))$$

Pareillement, on donne comme fonction de coût pour D l'opposé de celle de G. L'architecture du DCGAN correspond alors à un jeu à somme nulle. La théorie des équilibres de Nash donne un unique état stable. Il correspond à un coût égal à $-\log 4$ pour G et $\log 4$ pour D. Dans cette configuration, le

discriminateur est forc e d'associer une probabilit e de 0,5 pour chaque image donn e e en entr ee, le g n rateur  tant devenu trop fort.

Une variation int  essante sur le DCGAN est de poser $\mathbb{E}_{x \sim p_{\text{data}}}(\log(D(x))) + \mathbb{E}_{z \sim \text{bruit}}(\log(D(G(z)))$ comme fonction de co t en d but d'apprentissage. L'int   t de cette modification r  ulte d'un probl me : le discriminateur a tendance   facilement distinguer les images g n r es par G de celles de la base de donn e e en d but d'apprentissage. Dans ce cas, le terme $\log(1 - D(G(z))$ sature vers 0. Le remplacer par $\log(1 - D(G(z))$ r  oud ce probl me.

3.3 tude de la convergence des GAN

De par leur caract re adversaires, les GAN requi rent un quilibre fin entre la g n rateur et le discriminateur, ils sont donc par nature **instables**. L'tude de la convergence des GAN est un domaine encore tr s actif de la recherche. Nous allons discuter de deux ph nom nes tr s communs qui peuvent g ner ou ruiner l'apprentissage des GAN : l'**effondrement des modes** (*mode collapse*), et la **non-convergence** due   la perte d'quilibre du syst me.

3.3.1 L'effondrement des modes

L'effondrement des modes survient quand le r  seau g n rateur ne g n re pas des images conformement   l'ensemble de la distribution des images r  elles, mais seulement   une petite partie. L'effondrement des modes est tr s visible lorsque la distribution des images r  elles forme des zones bien s par es, c'est   dire quand celle-ci comporte des classes bien d finies. La manifestation de ce ph nom ne se traduit par des images g n r es qui se ressemblent toutes. Les figures [ref figure] montrent des exemples du ph nom ne sur la base de donn ees MNIST et CelebA.

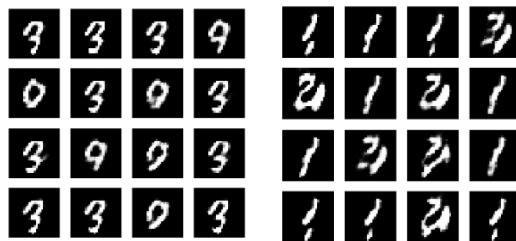


FIGURE 3.1  Exemples d'effondrement des modes sur la banque de chiffres MNIST.

Pour mieux comprendre le ph nom ne, il est int   tissant de regarder la distribution des images de MNIST dans son ensemble, cela est possible gr  ce   des algorithmes de r  duction de dimension. Attention, la r  duction de dimension se fait dans l'espace des pixels, et non pas dans un espace s mantique, la visualisation ne permet donc pas de s parer efficacement les diff  entes classes, elle permet seulement un aper u de la distribution dans l'espace s mantique. La figure [ref figure] pr  ente une visualisation de MNIST par transformation t-sne [ref tsne].



FIGURE 3.2 – À gauche, un exemple d’effondrement des modes sur la banque d’image CelebA. À droite, une génération sans effondrement pour comparaison. On observe que sur l’image de gauche, tous les personnages ont la même tête.

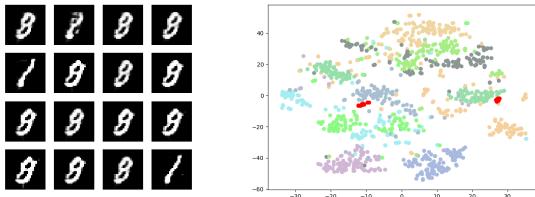


FIGURE 3.3 – mettre une legende (effondrement 2 modes (2 chiffres))

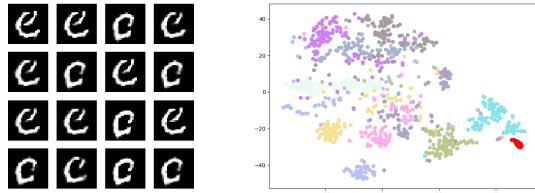


FIGURE 3.4 – mette une legende (effondrement 1 mode (pas un chiffre))

L’ensemble de points rouges correspond à un ensemble d’images générées par le réseau générateur lors de l’effondrement des modes. Sur les données MNIST, on observe différents groupes de points (des *clusters*), ce sont les **modes** inhérents à la base de donnée MNIST : les chiffres de 1 à 9. Ce qu’il est intéressant de noter, c’est que les points générés sont rassemblés autour de un ou plusieurs pôles denses très localisés, qui ne sont pas répartis dans tout l’espace. Cela traduit l’effondrement des modes : les images générées ne couvrent qu’une petit partie de la distribution de la base de donnée d’entraînement.

Il n’y a pas de solution simple, directe et universelle pour lutter contre l’effondrement des modes, mais quelques solutions ont été proposées :

- La pénalisation de la similarité des images en sortie de générateur *minibatch discrimination*. Cela consiste à ajouter un terme à la fonction de coût pour traduire la similarité (il peut s’agir de calculer une similaire pixel à pixel, ou d’estimer la similarité sémantique avec un autre réseau de neurones).

- Le *one-side label smoothing*. Cela consiste à changer l'objectif du discriminateur : son objectif ne sera plus de discriminer les fausses images avec une probabilité de 1, mais une probabilité plus faible, par exemple 0.9. Cela permet d'éviter la sur-confiance, et permet de laisser le générateur explorer tous l'espace des images réelles.
- Certaines architectures sont plus résistantes que d'autres à l'effondrement des modes. Par exemple, les GAN de Wasserstein ne présentent ce problème.

3.3.2 Perte de l'équilibre

Comme expliqué plus haut, l'apprentissage des GAN repose sur un équilibre fin entre le discriminateur et le générateur. Cet équilibre est parfois difficile à atteindre et est souvent instable, c'est pourquoi parfois le système s'effondre complètement. Cet effondrement vient souvent du fait que le discriminateur est devenu "trop fort" (sa fonction de perte tombe à zéro), et le générateur ne peut plus s'améliorer. Lorsque cela arrive, l'entraînement peut être arrêté : les images générées ne s'amélioreront plus. Un exemple de ce phénomène est illustré [ref figure], où l'on voit qu'à partir d'un cycle d'entraînement, la fonction de perte du discriminateur s'écroule et celle du générateur diverge.

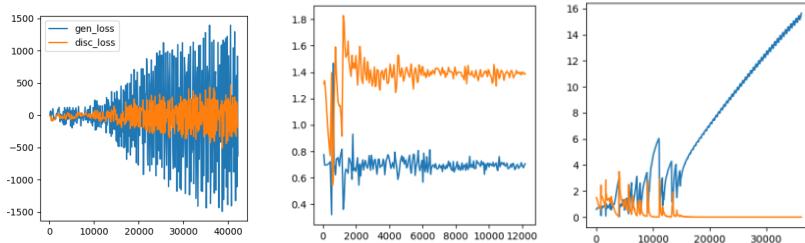


FIGURE 3.5 – legende

Il existe des solutions pour lutter contre ce problème, et cela consiste souvent à rééquilibrer les puissances ou les vitesses de convergence des différents réseaux. On peut par exemple diminuer la complexité du discriminateur, diminuer le taux d'apprentissage du discriminateur, ou mettre à jouer plus souvent le générateur que le discriminateur. Ajouter du bruit sur les images de la base de donnée permet aussi de renforcer la stabilité de l'apprentissage. Par ailleurs, on peut noter que les GAN de Wasserstein sont plus stables que les DCGAN, mais ne sont pas totalement immunisés aux problèmes de convergence.

3.4 Cadre théorique et WGAN

Nous allons essayer dans cette section de donner un cadre probabiliste et statistique rigoureux permettant d'expliquer le fonctionnement des GAN. Cette approche permettra de justifier l'algorithme WGAN qui permet de significativement réduire les problèmes d'apprentissage des GAN.

3.4.1 Approche bayésienne des GAN

L'objectif d'un GAN - la génération d'images suivant un dataset - peut être formalisé comme un problème d'optimisation bayésienne. Nous cherchons à approcher la distribution p_{data} d'une variable aléatoire $X : \Omega \rightarrow \mathcal{X}$. Pour ce faire, on se donne une famille paramétrique de distributions $\mathcal{M}_{\mathbb{R}^d} = \{p_\theta, \theta \in \mathbb{R}^d\}$, ainsi qu'un prior $p_{\text{bruit}}(z)$ relatif à une variable aléatoire $Z : \Omega \rightarrow \mathcal{Z}$. On détermine ensuite la distribution souhaitée à l'aide de la formule de Bayes.

$$p(\theta|\text{data}) \propto p(z)p(\text{data}|z)$$

Comme il est impossible de résoudre directement la formule de Bayes, il nous faut définir une fonction de coût qui mesure la distance entre p_θ et p_{data} , puis employer des algorithmes de descente du gradient. La famille $\mathcal{M}_{\mathbb{R}^d}$ prend alors naturellement la forme d'un réseau de neurones, que l'on écrit $g : \mathcal{Z} \times \mathbb{R}^d \rightarrow \mathcal{X}$, ou en notation condensée $g_\theta(z)$.

3.4.2 DCGAN

Le DCGAN utilise la métrique δ pour mesurer l'écart entre deux distributions p_{data} et p_θ , définie par la relation suivante.

$$\delta(p_{\text{data}}, p_\theta) = -\log 4 + 2\text{DJS}(p_{\text{data}}||p_\theta)$$

où DJS est la divergence de Jensen-Shanon.

On peut montrer [ref DCGAN] la relation suivante, en posant \mathcal{F} l'ensemble des fonctions continues $\mathcal{X} \rightarrow (0, 1)$.

$$\delta(p_{\text{data}}, p_\theta) = \sup_{f \in \mathcal{F}} \mathbb{E}_{x \sim p_{\text{data}}} (\log(f(x))) + \mathbb{E}_{x \sim p_\theta} (\log(1 - f(x)))$$

On remarque alors si $f : \mathcal{X} \rightarrow (0, 1)$ est solution de ce problème on peut calculer $\nabla_\theta \delta$, dans l'optique d'optimiser $g_\theta(z)$.

$$\nabla_\theta \delta(p_{\text{data}}, p_\theta) = \mathbb{E}_{z \sim p_{\text{bruit}}(z)} \left(\frac{\nabla_\theta f(g_\theta(z))}{f(g_\theta(z)) - 1} \right)$$

Il nous reste encore à déterminer f . Il est intuitif de chercher à calculer f comme un réseau de neurones $\{f_w, w \in \mathcal{W}\}$, qui peut être optimisé par rétro-propagation à partir de $\mathbb{E}_{x \sim p_{\text{data}}} \left(\frac{\nabla_w f_w(x)}{f_w(x)} \right) + \mathbb{E}_{z \sim p_{\text{bruit}}} \left(\frac{\nabla_\theta f(g_\theta(z))}{f(g_\theta(z)) - 1} \right)$.

Ce formalisme nous renvoie donc à la définition du DCGAN par sa fonction de coût pour G (ici g_θ) et D (ici f_w).

3.4.3 WGAN

Les Wasserstein GAN, ou WGAN, ont été introduits en 2017 par Martin Arjovsky et al. [ref]

3.5 Implémentation et résultats

blablablabla



FIGURE 3.6 – mettre une légende nos résultats

Chapitre 4

Le cycleGAN

4.1 Présentation de la problématique

Les cycleGAN sont des architectures de GAN, proposées par Zhu et al. [1], qui permettent de répondre à une problématique bien spécifique : le **transfert de style non appairé** [7]. Pour comprendre L'intérêt du cycleGAN, il faut bien comprendre le problème auquel il répond. Le transfert de style consiste à transformer des données d'un *style à un autre*. Le terme de *style* est à prendre au sens large et les données que l'on manipule peuvent être de natures diverses. Il peut s'agir par exemple de transformer des images de pommes en images d'orange, de transformer un paysage d'été en un paysage d'hiver, de transformer une musique classique en rock, ou encore de modifier l'expression les expressions faciales d'individus présents sur une image. Quelques exemples sont présentés sur la figure 4.1.

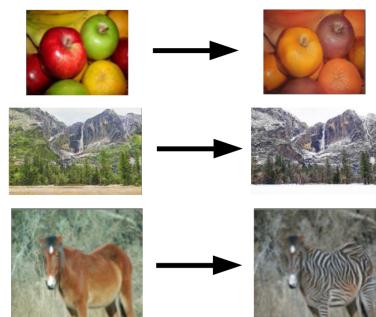


FIGURE 4.1 – Exemples de transfert de style effectués par un cycleGAN. De haut en bas : transformation *pommes ↔ oranges*, transformation *paysages d'été ↔ paysages d'hiver* et transformation *chevaux ↔ zèbres*. Ces exemples sont tirés de l'article de Zhu et al. [1]

Le transfert de style peut s'effectuer entre plus que seulement deux *classes de styles*, mais nous allons ici nous concentrer dans le cas binaire où l'on considère deux styles. La problématique est donc de transformer des images d'un style à l'autre, et ceci dans les deux sens.

Le transfert de style (à deux classes), repose sur deux banques de données, que l'on notera A et B. Suivant les données auxquelles nous avons accès, il existe deux cas différents :

- Dans le cas où nous connaissons un appairage entre les images de A et de B, le problème est un **transfert de style appairé**. Le but est donc d'apprendre et de généraliser le transfert d'une donnée de A à une donnée de B à partir d'exemples de paires déjà existantes.

Par exemple, si A représente des bâtiments de jour, et B représente des bâtiments de nuit, il est possible de prendre la même photo de jour et de de nuit. Ces deux photos constituent une paire dont chaque élément est d'un style différent.

- Dans le cas où chaque élément de A n'a pas de lien direct avec un élément de B en particulier, le problème est un **transfert de style non appairé**. Le but n'est plus d'apprendre et de généraliser le transfert d'une donnée de A à une donnée de B à partir d'exemples de paires déjà existantes, mais d'apprendre le transfert entre le style de A et le style de B, sans avoir d'exemple d'une telle transformation. Il faut donc *comprendre* à un niveau sémantique les styles de A et B.

Par exemple, si vous voulez transformer une image de votre chien en image de chat, vous ne pouvez pas obtenir une banque d'image de chiens déguisés en chats. Vous devez donc travailler avec d'une part des images de chiens (A), d'autre part des images de chats (B), sans pouvoir former de paires entre A et B.

La différence entre ces deux cas est illustrée par la figure 4.2.

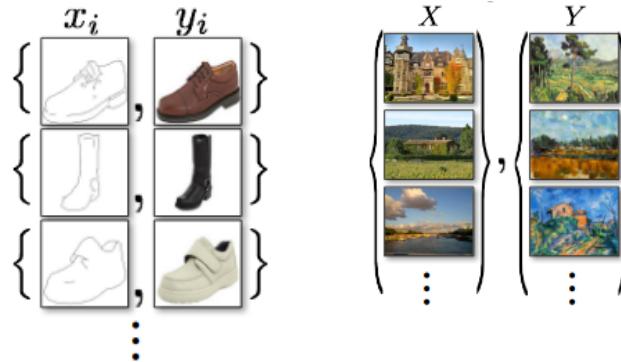


FIGURE 4.2 – Les deux types de transfert de style. À gauche, le transfert de style appairé où chaque donnée est associée à son équivalent dans un autre style. À droite, le transfert de style non appairé où les images n'ont pas d'équivalent dans l'autre style. Ces images sont tirés de l'article de Zhu et al. [1]

Ces deux types de transfert de style se traitent différemment. Pour le transfert de style appairé, une structure de GAN classique suffit puisque le discriminateur peut aisément comparer l'image générée avec l'image *idéale*. Ce problème, que nous ne développerons pas ici, est traité et manié efficacement par différents algorithmes, dont **Pix2Pix**. Le transfert de style non appairé ne permet pas la comparaison à l'image-cible puisqu'il n'existe pas de paires. **Il faut donc utiliser d'autres architectures, comme par exemple le cycleGAN.**

4.2 Principe général du cycleGAN

En vertu des explications présentées au paragraphes précédent, le problème se présente ainsi : nous avons une banque de données structurées A, et une banque de données structurées B, de même nature, dont les styles sont différents. Dans la suite, nous nous placerons dans le cas où s'est données sont des images. Le but est de transformer les images de A pour leur donner le style des images de B, et inversement. Le cycleGAN, peut aussi résoudre des problèmes de segmentation d'images, en considérant la segmentation comme un style pour l'image.

Le cycleGAN repose sur deux GAN, tête-bêche, l'un permettant de passer du style A au style B, l'autre du style B au style A. Plus précisément, il y a deux générateurs, un générateur qui prend des images de la banque A et doit générer des images du style de B (noté G), l'autre qui prend des images de la banque B et doit générer des images du style de A (noté F). Il y a aussi deux discriminateurs, notés D_A et D_B , qui respectivement discriminent des images du style A et celles du style B. L'architecture est présentée par la figure 4.3.

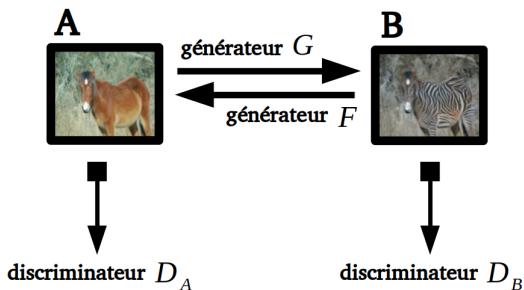


FIGURE 4.3 – Structure globale du cycleGAN. Le générateur G créer des images du style de B à partir d'images du style de A. Le générateur F créer des images du style de A à partir d'images du style de B. Chaque banque d'images, associée à un style, possède son discriminateur.

Comme on l'a entrevu dans le paragraphe précédent, une difficulté est que les données ne sont pas appairées, la fonction de coût ne peut donc pas venir de la comparaison directe de l'image générée à l'image souhaitée. Pour pallier à ce manque, deux fonctions de coûts principales et indépendantes sont utilisées.

La première est celle d'un GAN classique : pour une transformation $A \rightarrow B$ (resp. $B \rightarrow A$), le discriminateur D_B (resp. D_A) prédit si l'image est une image qui appartient réellement à la banque B (resp. A). Le coût associé au GAN ainsi défini est appelé *Adversarial Loss* ou *GAN Loss*. La figure 4.4 montre la décomposition du cycleGAN en deux GAN.

Comme on peut s'y attendre, cela ne suffit pas. En effet, si l'on considère seulement ce coût, comment peut-on s'assurer que l'image obtenue a encore un lien avec l'image de départ ? Pour garantir cela, il faut s'assurer de pouvoir reconstruire l'image de départ après lui avoir fait subir la transformation $A \rightarrow B$ suivie de $B \rightarrow A$. En d'autres termes, cela revient à ajouter des conditions sur les générateurs G et F telles que :

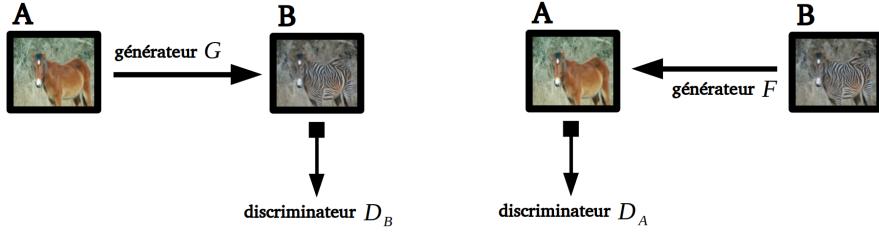


FIGURE 4.4 – Décomposition du cycleGAN en deux GAN distincts.

$$\begin{aligned} \forall a \in A, F(G(a)) &\approx a \\ \forall b \in B, G(F(b)) &\approx b \end{aligned} \quad (4.1)$$

Le coût qui en découle (et qui sera détaillé dans la suite), est appelé *Cycle Consistency Loss*. Les deux égalités ci-dessus consiste en réalité à parcourir le cycle respectivement en avant et en arrière, ceci est décrite de manière schématique par la figure 4.5.

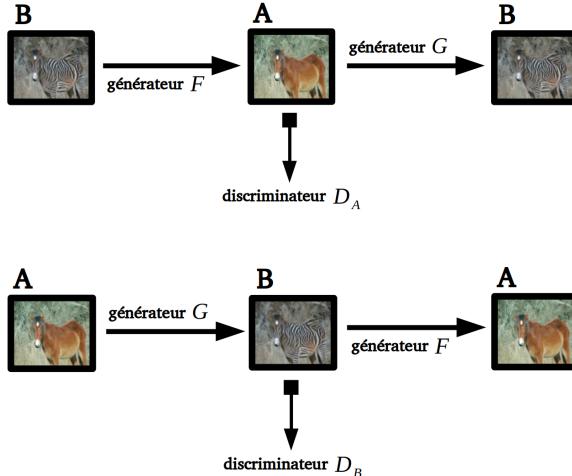


FIGURE 4.5 – Décomposition du cycleGAN en deux GAN distincts et ajout des contraintes de consistance cyclique. En haut, la consistance cyclique arrière (*backward cycle consistency*). En bas, la consistance cyclique avant (*forward cycle consistency*).

Pour résumer le fonctionnement global du cycleGAN. Le générateur G (qui assure la transformation $A \rightarrow B$) est optimisé pour tromper le discriminateur D_B comme dans un GAN classique, mais aussi pour que à F fixé, $F \circ G = \text{id}$. Et symétriquement, il en est de même pour le générateur F (qui assure la transformation $B \rightarrow A$). Les discriminateurs, quant à eux, sont mis à jour selon la même fonction de coût qu'un discriminateur de GAN classique. Les fonctions de coûts utilisées sont détaillées dans la partie suivante.

4.3 Les fonctions de coûts

Coût adversaire : *GAN Loss*

Comme précisé dans la partie précédente, le coût associé au caractère adversaire de l'apprentissage est celui d'un GAN classique [8]. Avec les mêmes notations que dans le paragraphe précédent, en considérant le générateur G et son discriminateur associé D_B associé, on a :

$$\mathcal{L}_{\text{GAN}}(G, D_B, A, B) = \mathbb{E}_{b \sim p_{\text{data}}(b)} [\log D_B(b)] + \mathbb{E}_{a \sim p_{\text{data}}(a)} [\log (1 - D_B(G(a)))]$$

Comme dans le cas d'un GAN classique, le générateur tend à minimiser ce coût et le discriminateur tend à la minimiser.

Pour l'autre GAN, c'est à dire le générateur F et son discriminateur D_A , on a de même :

$$\mathcal{L}_{\text{GAN}}(F, D_A, B, A) = \mathbb{E}_{a \sim p_{\text{data}}(a)} [\log D_A(a)] + \mathbb{E}_{b \sim p_{\text{data}}(b)} [\log (1 - D_A(F(b)))]$$

Coût du cycle : *Cycle Consistency Loss*

Conformément aux explications données dans le paragraphe précédent, on cherche une fonction de coût qui assure que : $F \circ G = \mathbb{1}$ et $G \circ F = \mathbb{1}$. Ces deux égalités sont appelées respectivement *backward cycle consistency* et *forward cycle consistency* et ont été utilisées depuis longtemps dans le suivi d'objets [9]. Il est important de noter que l'on veut un coût qui n'intervient pas à une hauteur sémantique. On considère donc deux simples comparaisons pixel à pixel, une pour la *backward cycle consistency* et une pour la *forward cycle consistency*, que l'on somme. La fonction de coût qui en découle est donc :

$$\mathcal{L}_{\text{cyc}}(G, F) = \mathbb{E}_{a \sim p_{\text{data}}(a)} [\|F(G(a)) - a\|_1] + \mathbb{E}_{b \sim p_{\text{data}}(b)} [\|G(F(b)) - b\|_1]$$

Fonction de coût globale

Les deux fonctions de coûts adversaires jouent des rôles symétriques, elles ont la même importance dans la forme de la fonction de coût globale. Cependant, rien ne laisse penser que l'importance de la fonction de coût du cycle leur est aussi équivalente. Il est donc nécessaire d'introduire un $\lambda \in \mathbb{R}$ tel que :

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{GAN}}(G, D_B, A, B) + \mathcal{L}_{\text{GAN}}(F, D_A, B, A) + \lambda \cdot \mathcal{L}_{\text{cyc}}(G, F)$$

λ est un hyper-paramètre. D'après Zhu et al. [1], $\lambda \approx 10$ donne les meilleurs résultats.

Préservation de la couleur

Pour certaines applications particulières, notamment pour le traitement de paysages, il est nécessaire de rajouter un autre terme à la fonction de coût. En effet, comme on l'observe sur la figure 4.6 les couleurs globales des photos en entrée de sont pas retrouvées en sortie. Les images sont par exemple bleuies ou jaunies. Dans l'article de Zhu et al. [1], l'équipe propose de contraindre encore

plus l'espace dans lequel évolue les générateurs du cycleGAN, une technique introduite par Taigman et al. [10]. L'idée consiste à ajouter un coût demi-cyclique qui tend à ce que $F \approx \mathbb{1}$ et $G \approx \mathbb{1}$. On rajoute donc un coût $\mathcal{L}_{\text{identity}}$ défini comme :

$$\mathcal{L}_{\text{identity}}(G, F) = \mathbb{E}_{b \sim p_{\text{data}}(b)} [\|G(b) - b\|_1] + \mathbb{E}_{a \sim p_{\text{data}}(a)} [\|F(a) - a\|_1]$$

On comprend bien que c'est une limitation très forte, qui ne convient qu'à certains problèmes pour lesquels les images de sortie sont très proches des images d'entrée et pour lesquels la couleur ne doit pas beaucoup changer. Sous ces conditions, il se trouve que cette méthode conserve efficacement la composition des couleurs, comme peut l'attester la figure 4.6.

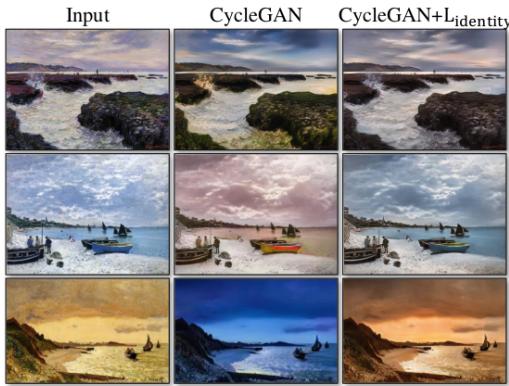


FIGURE 4.6 – Mise en évidence de la dégradation de la composition des couleurs, par Zhu et al. [1]. À droite, l'effet de la fonction de coût $\mathcal{L}_{\text{identity}}$, qui améliore la cohérence des couleurs.

4.4 Les métriques d'évaluations

Comme dans le cas d'un GAN classique, évaluer la qualité de la sortie d'un cycleGAN n'est pas une chose facile. En effet, nous n'avons de métrique simple et universelle qui permettrait de juger de la crédibilité ou du réalisme d'une image. Pour tenter d'évaluer au mieux la qualité d'un cycleGAN, il existe plusieurs solutions.

La première, sans grande surprise, c'est de faire une étude de réalisme basée sur une enquête auprès de personnes chargées de noter la qualité des images fournies, c'est ce que l'on appelle des études de perceptions (*perceptual studies*). On comprend vite que ce n'est une très bonne solution : ces études restent subjectives, elles ne sont pas toujours reproductibles, et elles coûtent cher. Comme pour les GAN, on ne peut pas donc s'en servir pour poser une métrique universelle pour comparer différents algorithmes.

Pour quelques problèmes particuliers, on peut trouver des métriques convenables. C'est le cas par exemple si l'on considère un problème de segmentation

et si les données sont accompagnées de leurs segmentations réelles, appelée aussi *ground truth*. Dans ce cas particulier, évaluer le cycleGAN revient simplement à évaluer de résultat de la segmentation par rapport au *ground truth*. Il existe plusieurs métriques classiques pour évaluer les algorithmes de segmentation comme la précision par pixel à pixel ou la précision classes à classes, mais la métrique la plus courante pour cela est l'indice de Jaccard (ou *IoU : Intersection over Union*). Cette métrique consiste à calculer, pour chaque classe de la segmentation, l'intersection de la zone prédite par l'algorithme avec la zone réelle, avant de normaliser par l'union des deux zones. C'est une métrique classique utilisée en segmentation, elle est définie ci-dessous.

$$\text{Indice de Jacard} : J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Cependant, dans le cas général le problème n'est pas un problème de segmentation, mais un problème de génération d'images réalistes suivant un style, et la métrique précédente n'est pas utilisable. Il en existe d'autres, par exemple le **score FCN**. Le score FCN consiste à évaluer ininterprétable du résultat par un algorithme classique de segmentation sémantique (ici le FCN, pour *Fully Convolutional Networks for Semantic Segmentation [?]*). Sur une image générée par le cycleGAN, le FCN prédit une carte de segmentation. Cette carte de segmentation est ensuite comparée à l'image d'entrée avec des métriques classiques que l'on a évoquées au-dessus, en particulier l'indice de Jaccard. Notons que le score FCN ne permet pas de vérifier que le style de l'image est correct, mais seulement d'évaluer grossièrement la caractère réaliste de l'image, à travers l'interprétabilité de l'image par un autre algorithme. Il n'existe aucune métrique idéale.

4.5 Implémentation et résultats

4.5.1 Détails d'implémentation

Notre implémentation, comme pour les autres algorithmes, utilise TensorFlow 2.0. Nous avons globalement respecté la structure préconisée dans [1], qui a été proposée par Johnson et al. [11] mais nous avons adapté l'architecture à chacune de nos banques de données. L'architecture de base, comme décrite dans l'article est la suivante :

Pour le discriminateur, nous avons utilisé un PatchGAN [12]. Avec les notations utilisées par TensorFlow :

```

Conv2D(64, (4, 4), strides = (2, 2))
LeakyReLU(alpha = 0.2)
Conv2D(128, (4, 4), strides = (2, 2))
InstanceNormalization
LeakyReLU(alpha = 0.2)
Conv2D(256, (4, 4), strides = (2, 2))
InstanceNormalization
LeakyReLU(alpha = 0.2)
Conv2D(512, (4, 4), strides = (2, 2))
InstanceNormalization
LeakyReLU(alpha = 0.2)
Conv2D(512, (4, 4), strides = (2, 2))
InstanceNormalization
LeakyReLU(alpha = 0.2)
Conv2D(1, (4, 4))

```

À noter que toutes les convolutions ont un *padding* défini sur *same*. La couche *Instance Normalization* fait référence à la normalisation présentée dans [13].

Pour le générateur, toujours comme dans l'article de Zhu et al. [1], nous avons utilisé un réseau résiduel. Avec les notations utilisées par TensorFlow :

Soit le bloc résiduel (*ResBlock*) de paramètre $n_{filters}$ défini par :

```

Conv2D( $n_{filters}$ , (3, 3))
InstanceNormalization
Activation(relu)
Conv2D( $n_{filters}$ , (3, 3))
 $g = InstanceNormalization$ 
Concatenate()([ $g, inputlayer$ ])

```

Le générateur complet s'écrit :

```

 $Conv2D(64, (7, 7))$ 
 $Activation(relu)$ 
 $Conv2D(128, (3, 3))$ 
 $InstanceNormalization$ 
 $Activation(relu)$ 
 $Conv2D(256, (3, 3))$ 
 $InstanceNormalization$ 
 $Activation(relu)$ 

 $N \times [ResBlock(n_{filters})]$ 

 $Conv2DTranspose(128, (3, 3))$ 
 $InstanceNormalization$ 
 $Activation(relu)$ 
 $Conv2DTranspose(64, (3, 3))$ 
 $InstanceNormalization$ 
 $Activation(relu)$ 
 $Conv2D(3, (7, 7))$ 
 $InstanceNormalization$ 
 $Activation(tanh)$ 

```

À noter que toutes les convolutions ont un *padding* défini sur *same* et des *strides* de (2, 2).

Le bloc résiduel été proposé par He et al. [14]. Notons que $n_{filters}$ et N sont des hyper-paramètres. Leur valeur dépend de la taille des images et de la puissance de calcul disponible. Sur les conseils de Zhu et al., nous utilisons $n_{filters} = 256$ et $N \in [5, 10]$. Dans notre implémentation, tous les hyper-paramètres du modèle sont facilement modifiable depuis un unique fichier.

Pour les paramètres d'apprentissage, nous avons suivi les conseils de l'article nous avons adapté les valeurs à chaque banque d'image en testant différentes valeurs. Nous avons, de manière générale, les valeurs nominales suivantes :

- Nombre de passes : 150
- Taille des batch : 1
- Optimiseur : Adam
- α_{Adam} : 0.0002 puis linéairement décroissant à partir de la passe 100
- β_1_{Adam} : 0.5

Comme pour les GAN, la stabilité du modèle peut être améliorée en entraînant le discriminateur sur un historique des images générées. Cette technique a été proposée par Shrivastava et al. [15] et reprise par Zhu et al. pour les cycles-GAN. Nous l'avons aussi implanté. La taille du *buffer* contenant l'historique des images générées est un nouvel hyper-paramètre. Nous prenons, comme proposé dans l'article de Zhu et al., $buffer_{max} = 50$.

4.5.2 Quelques résultats

Quelques exemples de nos résultats sont présentés sur les figures 4.7 à 4.9



FIGURE 4.7 – Exemples de sorties de notre cycleGAN sur la banque d’image CelebA. La première ligne correspond aux images de la banque, la deuxième ligne correspond à la sortie du générateur. À gauche, il s’agit de la transformation *portrait sans sourire* vers *portrait avec sourire*. À droite, il s’agit de la transformation inverse.



FIGURE 4.8 – Exemples de sorties de notre cycleGAN sur la banque d’image *pommes ↔ oranges*. À gauche, les images d’orange de la banque. À droite, les mêmes images dans le style des pommes en sortie de cycleGAN.



FIGURE 4.9 – Exemple de sortie de notre cycleGAN sur la banque d’image *chevaux ↔ zèbres*. À gauche, une image de chevaux de la banque. À droite, la même images dans le style des images zèbres en sortie de cycleGAN.

4.5.3 Limitations et ouverture

Les résultats que nous obtenons pour l’instant sont corrects mais restent mitigés. En effet, ils sont convenables sur des images de petites dimensions qui ne demandent que peu de ressources. Sur les images de hautes dimension, les résultats pourraient être améliorés si nous pouvions exécuter nos scripts plus

longtemps. Pour l'instant, nous sommes ralenti par le fait d'enregistrer correctement nos modèles pour pouvoir continuer l'apprentissage. Actuellement, la reprise de l'apprentissage ne se passe pas au mieux, ce qui fausse nos résultats.

Cependant, étant donné que sur les premières passes, nous obtenons des résultats cohérent avec les implémentations de références, nous sommes plutôt confiants quand à la qualité de notre implémentation. Son caractère modulaire et sa paramétrabilité très facile permet de l'adapter à beaucoup de banque d'images différentes. C'est ce qui nous permet d'utiliser notre script pour des problèmes non abordés dans l'article de Zhu et al. [1], comme celebA. De plus, on peut l'utiliser de manière équivalente sur nos CPU personnels et sur le cluster de GPU Fusion.

Une fois ces derniers détails réglés, dans peu de temps, nous pourrons utiliser notre programme pour étudier un nouveau problème, que nous devons définir.

Conclusion

conclusion conclusion conclusion conclusion conclusion conclusion
conclusion conclusion conclusion conclusion conclusion conclusion
conclusion conclusion conclusion conclusion conclusion conclusion
conclusion conclusion conclusion conclusion conclusion conclusion
conclusion conclusion conclusion conclusion conclusion conclusion

Bibliographie

- [1] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, “Unpaired image-to-image translation using cycle-consistent adversarial networks,”
- [2] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,”
- [3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” in *Proceedings of the IEEE*, vol. 86, pp. 2278–2324.
- [4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Advances in neural information processing systems*, pp. 2672–2680.
- [5] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein gan,”
- [6] P. Goldsborough, “A tour of TensorFlow,”
- [7] L. A. Gatys, A. S. Ecker, and M. Bethge, “Image style transfer using convolutional neural networks,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2414–2423, IEEE.
- [8] I. Goodfellow, “NIPS 2016 tutorial : Generative adversarial networks,” in *arXiv preprint arXiv :1701.00160*.
- [9] Z. Kalal, K. Mikolajczyk, and J. Matas, “Forward-backward error : Automatic detection of tracking failures,” in *Proceedings of the 2010 20th International Conference on Pattern Recognition, ICPR ’10*, pp. 2756–2759, IEEE Computer Society.
- [10] Y. Taigman, A. Polyak, and L. Wolf, “Unsupervised cross-domain image generation,”
- [11] J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual losses for real-time style transfer and super-resolution,”
- [12] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,”
- [13] D. Ulyanov, A. Vedaldi, and V. Lempitsky, “Instance normalization : The missing ingredient for fast stylization,”
- [14] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,”
- [15] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb, “Learning from simulated and unsupervised images through adversarial training,”