

# Rapport sur le Projet CycleGan

Paulin Brissonneau      Thomas Kebaili      Ilyas Moutawwakil  
Valentin Laurent      Lilian Lecomte

XX mai 2020

# Table des matières

<b>1 Le Multiperceptron</b>	<b>2</b>
1.1 Le perceptron simple . . . . .	2
1.2 Le multiperceptron . . . . .	2
1.3 Implémentation et résultats . . . . .	2
<b>2 Les réseaux à convolutions</b>	<b>3</b>
2.1 Présentation générale . . . . .	3
2.1.1 Introduction . . . . .	3
2.1.2 Structure d'un CNN . . . . .	3
2.1.3 Principe général . . . . .	3
2.2 Formalisation d'un CNN . . . . .	4
2.2.1 Le produit de convolution : un outil indispensable . . . . .	4
2.2.2 Le balayage de l'image : une histoire de Padding et de Stride	4
2.2.3 Généralisation en 3D . . . . .	5
2.3 Implémentation et résultats . . . . .	5
<b>3 Les GAN (Réseaux Adverses Génératifs)</b>	<b>6</b>
3.1 Principe général des GAN . . . . .	6
3.2 Le DCGAN (Deep Convolutionnal Adversarial Network) . . . . .	6
3.3 Le W-GAN (GAN de Wasserstein) . . . . .	7
3.4 Étude de la convergence des GAN . . . . .	7
3.4.1 L'effondrement des modes . . . . .	7
3.4.2 Perte de l'équilibre . . . . .	9
3.5 Implémentation et résultats . . . . .	9
<b>4 Le cycleGAN</b>	<b>11</b>
4.1 Présentation de la problématique . . . . .	11
4.2 Principe général du cycleGAN . . . . .	13
4.3 Les fonctions de coûts . . . . .	14
4.4 Les métriques d'évaluations . . . . .	16
4.5 Implémentation et résultats . . . . .	17
4.5.1 Détails d'implémentation . . . . .	17
4.5.2 Quelques résultats . . . . .	19
4.5.3 Limitations et ouverture . . . . .	19

# 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 inefficace 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). Dans cette partie, nous allons nous efforcer d'expliquer le fonctionnement de celui-ci ainsi que ses avantages.

#### 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. Cependant, certaines couches, nommées couche de convolution, ont un fonctionnement radicalement différent de celui d'une couche dense.

Voici l'architecture d'un CNN :

insérer image architecture CNN

Comme on peut le voir, 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 est assez intuitif : 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 effectués. 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

Une fois le principe directeur mis en évidence, nous pouvons nous atteler à la formalisation du 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 :

insérer image chien de prairie

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 :

insérer image balayage image

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 : c'est du **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 :

insérer image same padding

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

insérer image stride

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. Puisqu'un schéma vaut bien plus qu'un texte :

insérer image 3D

## 2.3 Implémentation et résultats

Blablabla

## 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 :

$$\mathbb{E}_{x \in dataset}(\log(D(x))) + \mathbb{E}_z(\log(1 - D(G(z)))) \quad (3.1)$$

Pareillement, on donne comme fonction de coût pour D l'opposé de celle de G. L'architecture du DCGAN correspond alors à un jeu minmax. La théorie des équilibres de Nash donnent un unique état stable. Il correspond à un coût égal à  $-\log 4$  pour G et  $\log 4$  pour D. Cette situation représente un discriminateur

forcé d'associer une probabilité de 0,5 pour chaque image donnée en entrée, le générateur étant devenu trop fort.

Une variation intéressante sur le DCGAN est de poser  $\mathbb{E}_{x \in \text{dataset}}(\log(D(x))) + \mathbb{E}_z(\log(D(G(z))))$  comme fonction de coût en début d'apprentissage. L'intérêt de cette modification résulte d'un problème : le discriminateur a tendance à facilement distinguer les images générées par  $G$  de celles de la base de donnée en début d'apprentissage. Dans ce cas, le terme  $\log(1 - D(G(z)))$  sature vers 0. Le remplacer par  $\log(1 - D(G(z)))$  résoud ce problème.

### 3.3 Le W-GAN (GAN de Wasserstein)

blablabla

## 3.4 Étude de la convergence des GAN

De par leur caractère adversaires, les GAN requièrent un équilibre fin entre la générateur et le discriminateur, ils sont donc par nature **instables**. L'étude de la convergence des GAN est un domaine encore très actif de la recherche. Nous allons discuter de deux phénomènes très communs qui peuvent gêner ou ruiner l'apprentissage des GAN : l'**effondrement des modes** (*mode collapse*), et la **non-convergence** due à la perte d'équilibre du système.

### 3.4.1 L'effondrement des modes

L'effondrement des modes survient quand le réseau générateur ne génère pas des images conformes à l'ensemble de la distribution des images réelles, mais seulement à une petite partie. L'effondrement des modes est très visible lorsque la distribution des images réelles forme des zones bien séparées, c'est à dire quand celle-ci comporte des classes bien définies. La manifestation de ce phénomène se traduit par des images générées qui se ressemblent toutes. Les figures [ref figure] montrent des exemples du phénomène sur la base de données MNIST et CelebA.

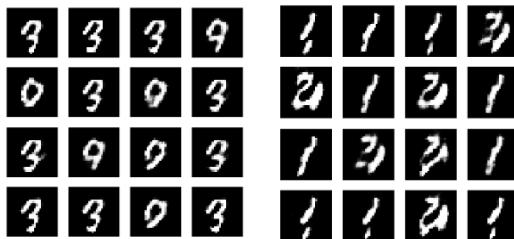


FIGURE 3.1 – Exemples d'effondrement des modes sur la banque de chiffres MNIST.

Pour mieux comprendre le phénomène, il est intéressant de regarder la distribution des images de MNIST dans son ensemble, cela est possible grâce à des algorithmes de réduction de dimension. Attention, la réduction de dimension se fait dans l'espace des pixels, et non pas dans un espace sémantique, la



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.

visualisation ne permet donc pas de séparer efficacement les différentes classes, elle permet seulement un aperçu de la distribution dans l’espace sémantique. La figure [ref figure] présente une visualisation de MNIST par transformation t-sne [ref tsne].

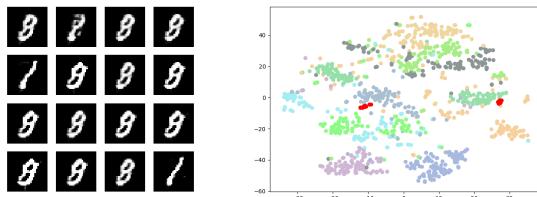


FIGURE 3.3 – mettre une legende (effondrement 2 modes (2 chiffres))

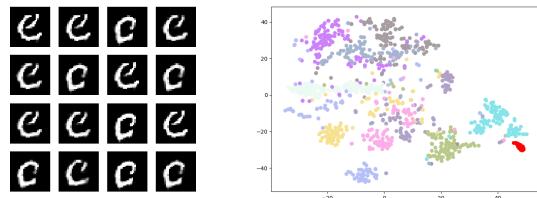


FIGURE 3.4 – mette une legende (effondrement 1 mode (pas un chiffre))

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.

### 3.4.2 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.

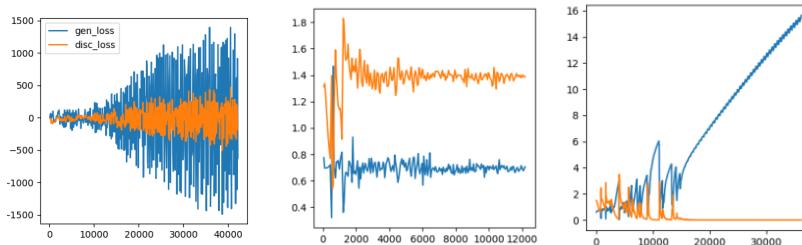


FIGURE 3.5 – légende

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.

## 3.5 Implémentation et résultats

blablablabla



FIGURE 3.6 – mettre une légende nos résultats

# Chapitre 4

## Le cycleGAN

### 4.1 Présentation de la problématique

[AJOUTER DES REFS]

Les cycleGAN sont des architectures de GAN qui permettent de répondre à une problématique bien spécifique : le **transfert de style non appairé**, que nous expliciterons.

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. Le cycleGAN peut aussi résoudre des problèmes de segmentation d'images, en considérant la segmentation comme un style pour l'image. Quelques exemples sont présentés sur la figure 4.1.

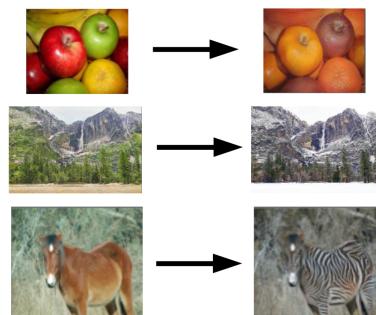


FIGURE 4.1 – légende

Le transfert de style peut s'effectuer entre plusieurs *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 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'images 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.

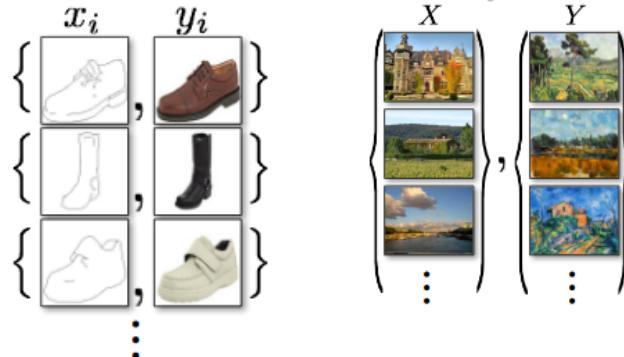


FIGURE 4.2 – légende (pas beau!!)

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 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.

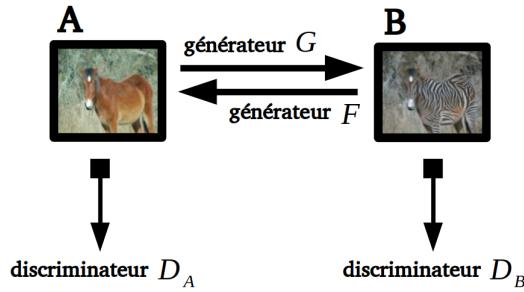


FIGURE 4.3 – legende (figure à changer)

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.



FIGURE 4.4 – legende (figure à changer)

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 :

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

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.

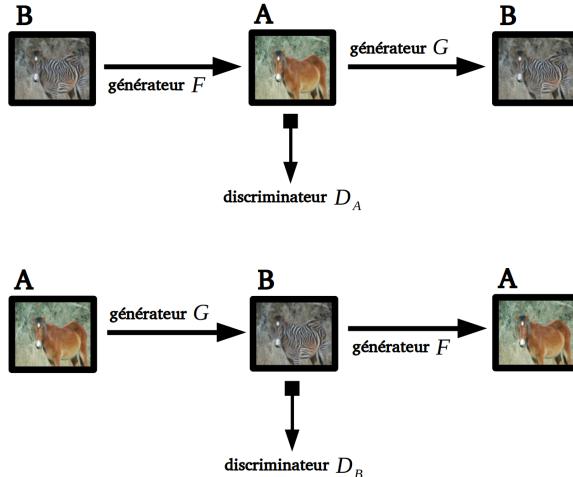


FIGURE 4.5 – légende (figure à changer)

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 aussi pour que à  $F$  fixé,  $F \circ G = \mathbb{1}$ . 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*

[REF]

Comme précisé dans la partie précédente, le coût associé au caractère adversaire de l'apprentissage est celui d'un GAN classique. 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*

[REF]

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*. Il est important de noter que l'on veut un coût qui n'interviennent 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 :

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

$\lambda$  est un hyper-paramètre. D'après [REF],  $\lambda \approx 10$  donne les meilleurs résultats.

### Préservation de la couleur

[REF]

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 ??, les couleurs globales des photos en entrée de sont pas retrouvées en sortie. Les images sont par exemple bleuies ou jaunies. Dans [REF], l'équipe de recherche propose de contraindre encore plus l'espace dans lequel évolue les générateurs du cycleGAN, une technique introduite par [Taigman et al. [49]]. 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.

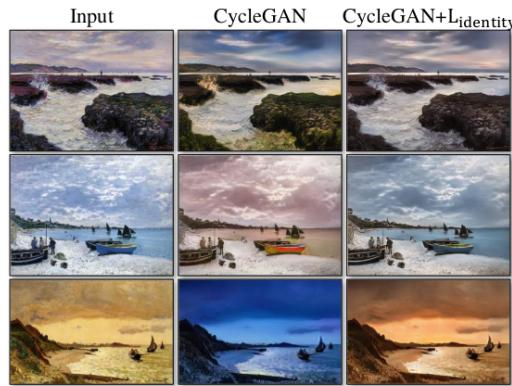


FIGURE 4.6 – legende (ca vient de l'article)

#### 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 pas 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 le 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 prédictive 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.

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

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* [REF]). 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 [REF], qui a été proposée par [Johnson et al.] mais 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 [22, 30, 29]. 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 [REF]

53].

**Pour le générateur**, toujours comme dans l'article [], 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))  
 $g = \text{InstanceNormalization}$   
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 [\text{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 [REF 18]. 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 [REF], 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
- $\alpha_{Adam}$  : 0.0002 puis linéairement décroissant à partir de la passe 100
- $\beta_1 Adam$  : 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.'s strategy [REF 46] et reprise par [REF] pour les cyclesGAN. Nous l'avons aussi implémenté. La taille du *buffer* contenant l'historique des images générées est un nouvel hyperparamètre. Nous prenons, comme proposé dans [REF],  $buffer_{max} = 50$ .

#### 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 du 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 du cycleGAN sur la banque d'image [legende]

#### 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



FIGURE 4.9 – Exemples de sorties du cycleGAN sur la banque d'image [legende]

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 [REF], 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.  
[1]

# Bibliographie

- [1] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient BackProp,” in *Neural Networks : Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pp. 9–50, Springer-Verlag.