

CENTRALESUPÉLEC

## RAPPORT DE PROJET

---

# CYCLEGAN

## TRANSFORMER LES CHEVAUX EN ZÈBRES

---

PAULIN BRISSONNEAU  
THOMAS KEBAILI  
VALENTIN LAURENT  
LILIAN LECOMTE  
ILYAS MOUTAWWAKIL

PROFESSEURS ENCADRANTS :  
JOANNA TOMASIK  
ARPAD RIMMEL

8 JUIN 2020

# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1 Multi-Layer Perceptron</b>	<b>4</b>
1.1 Les neurones . . . . .	4
1.2 Le perceptron . . . . .	5
1.3 Perceptron multicouche . . . . .	5
1.3.1 Quelques fonctions d'activation . . . . .	6
1.4 Rétropropagation . . . . .	8
1.4.1 Principe . . . . .	8
1.4.2 Les fonctions de coûts . . . . .	9
1.4.3 Algorithmes d'optimisation . . . . .	9
1.4.4 Les batchs . . . . .	11
1.5 Implémentation et résultats . . . . .	12
1.5.1 Avant-propos . . . . .	12
1.5.2 Effet du nombre de neurones par couche . . . . .	12
1.5.3 Effet du nombre de couches de neurones . . . . .	12
1.5.4 Taille des batchs . . . . .	13
1.5.5 Initialisation des poids . . . . .	15
1.5.6 Taux d'apprentissage . . . . .	16
<b>2 Les réseaux à convolutions</b>	<b>17</b>
2.1 Présentation générale . . . . .	17
2.1.1 Introduction . . . . .	17
2.1.2 Structure d'un CNN . . . . .	17
2.1.3 Principe général . . . . .	18
2.2 Formalisation d'un CNN . . . . .	18
2.2.1 Le produit de convolution : un outil indispensable . . . . .	18
2.2.2 Le balayage de l'image : une histoire de Padding et de Stride	19
2.2.3 Généralisation en 3D . . . . .	20
2.2.4 Couche de pooling . . . . .	20
2.2.5 Structure complète d'un CNN . . . . .	21
2.2.6 Apprentissage d'un CNN . . . . .	22
2.2.7 Dropout . . . . .	22
2.3 Implémentation et résultats . . . . .	23
2.3.1 Implémentation . . . . .	23
2.3.2 Résultats . . . . .	24

<b>3 Les GAN (Réseaux Adverses Génératifs)</b>	<b>26</b>
3.1 Principe général des GAN . . . . .	26
3.2 Le DCGAN (Deep Convolutionnal Adversarial Network) . . . . .	26
3.3 Étude de la convergence des GAN . . . . .	27
3.3.1 L'effondrement des modes . . . . .	27
3.3.2 Perte de l'équilibre . . . . .	29
3.4 Cadre théorique et WGAN . . . . .	29
3.4.1 Approche bayésienne des GAN . . . . .	30
3.4.2 DCGAN . . . . .	30
3.4.3 WGAN . . . . .	31
3.4.4 Avantages comparatif du WGAN par rapport au DCGAN	31
3.5 Implémentation et résultats . . . . .	31
3.5.1 Détails de l'implémentation . . . . .	31
3.5.2 MNIST . . . . .	32
3.5.3 CelebA . . . . .	33
<b>4 Le cycleGAN</b>	<b>34</b>
4.1 Présentation de la problématique . . . . .	34
4.2 Principe général du cycleGAN . . . . .	36
4.3 Les fonctions de coûts . . . . .	37
4.4 Les métriques d'évaluations . . . . .	39
4.5 Implémentation et résultats . . . . .	40
4.5.1 Détails d'implémentation . . . . .	40
4.5.2 Quelques résultats . . . . .	42
4.5.3 Limitations et ouverture . . . . .	43
<b>Conclusion</b>	<b>45</b>

# Introduction

L'objectif de ce projet est de comprendre, maîtriser, et utiliser la technologie des cycleGAN, proposée par Zhu et al. [1] en 2017. 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 ne 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 proposés par Radford et al. en 2016 [4] et les W-GAN proposés par Arjovsky et al. en 2017 [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 baserons pour mener à bien la suite du projet : appliquer la technologie des cycleGAN à un problème de notre choix.

# Chapitre 1

## Multi-Layer Perceptron

### 1.1 Les neurones

Les technologies qui vont être présentées dans la suite se basent toutes sur l'idée de neurone artificiel, aussi appelé neurone formel. McCulloh et Pitts le formalisent en 1943 [7]. Un neurone formel est composé de deux parties.

- La première consiste à faire la somme pondérée par des poids des valeurs d'entrée du neurone auxquelles on peut éventuellement ajouter un biais. Les poids sont propres au neurone et il y a un poids par entrée.
- La deuxième partie du neurone est la fonction d'activation. Cette fonction va s'appliquer sur le résultat de la somme pondérée. On choisit quasi-exclusivement des fonctions non-linéaires pour deux raisons : briser la linéarité (car dans le cas contraire le réseau serait assimilable à une seule matrice) et obtenir un résultat d'une certaine forme (par exemple une probabilité entre 0 et 1).

McCulloh et Pitts dans leur première ébauche du neurone formel considèrent des neurones au résultat binaire à l'aide d'une fonction d'activation de Heavyside. Un tel neurone formel est représenté par la figure 1.1.

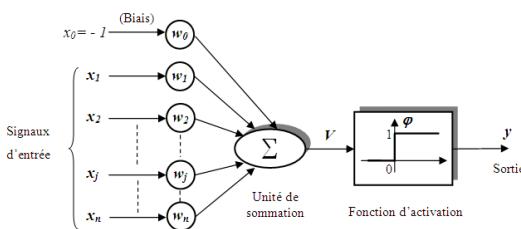


FIGURE 1.1 – Exemple de neurone dont les poids sont les coefficients  $w_1$  à  $w_2$ , et le biais est  $w_0$ . La fonction de Heavyside s'applique à la somme pour donner une sortie du neurone dans  $\{0, 1\}$ . Cette image provient d'un article de Benharir et al. [8].

Ainsi formellement en utilisant les notant  $X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$  les entrées du neurone,

$W = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$  les poids correspondants et  $\varphi$  la fonction d'activation,  $b$  le biais, un neurone correspond à une fonction  $N$  telle que :

$$N(x_1, \dots, x_n, w_1, \dots, w_n) = \varphi(b + \sum_i x_i w_i) = \varphi(W^T X + b)$$

## 1.2 Le perceptron

En 1958, Frank Rosenblatt utilise l'idée des neurones artificiels pour inventer le perceptron [9]. Son idée est d'utiliser les neurones pour reconnaître des formes simples dans des images. Cependant dans cette forme simple, il conserve les neurones tels que définis en 1943 avec la fonction de Heavyside et une seule couche de neurones (c'est-à-dire que les différents neurones ne sont pas reliés entre eux). Cela limite l'intérêt du perceptron : il ne peut apprendre que des *pattern* linéairement séparables. Minsky démontre par exemple que le perceptron est incapable d'effectuer un XOR [10]. L'idée est d'ajuster les poids pour que la sortie du neurone soit 1 si et seulement si l'entrée est dans l'ensemble que l'on cherche à reconnaître. Le schéma d'un perceptron est donné sur la figure 1.2.

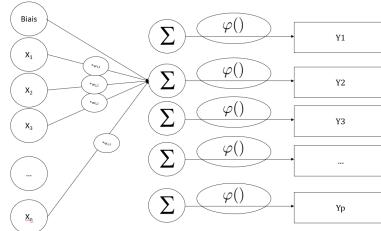


FIGURE 1.2 – Exemple de perceptron à  $n$  entrées et  $p$  sorties. Il peut donc servir de classificateur à  $p$  classes.

## 1.3 Perceptron multicouche

Un **perceptron multicouche**, **multiperceptron**, ou **MLP** (pour *Multi-Layer Perceptron*) est composé de différentes couches :

- La couche d'entrée qui correspond aux valeurs des données.
- Une couche de sortie qui correspond aux valeurs renvoyées par le MLP.
- Une ou plusieurs couches cachées (*hidden layers*) qui sont des couches de neurones reliées entre elles. À chaque couche, les sorties correspondent aux entrées de la couche suivante.

Écrivons formellement ceci. On définit  $N$  la sortie de la  $i^e$  couche :

$$N^{(i)} = \begin{pmatrix} 1 \\ n_1^{(i)} \\ \vdots \\ n_{p_i}^{(i)} \end{pmatrix}$$

Le 1 permet la présence d'un biais.

Définissons également  $W^{(i)}$  la matrice des poids de la couche i.

$$W^{(i)} = \begin{pmatrix} w_{1,1}^{(i)} & \dots & w_{1,p_i}^{(i)} \\ \vdots & \ddots & \vdots \\ w_{p_i,1}^{(i)} & \dots & w_{p_i,p_i}^{(i)} \end{pmatrix}$$

Soit  $\varphi_k^{(i)}$  la fonction d'activation de ce même neurone et  $\Phi^{(i)}((x_1, \dots, x_{p_i})) \rightarrow (\varphi_1^{(i)}(x_1), \dots, \varphi_{p_i}^{(i)}(x_{p_i}))$ .

On notera que si le réseau possède  $L + 1$  couches (de 0 à  $L$ ),  $N^{(0)} = X$  et  $N^{(L)} = Y$ . Dès lors,

$$\forall i \in [1, L], N^{(i)} = \Phi^{(i)}(W^{(i)}N^{(i-1)}).$$

### 1.3.1 Quelques fonctions d'activation

Il existe de nombreuses fonctions d'activation avec chacune leurs avantages et leurs défauts. En voici quelques-unes.

#### Sigmoïde

Cette fonction permet d'avoir des valeurs de sortie entre 0 et 1 mais souffre d'un problème de *gradient vanishing*, c'est-à-dire que proche de sa zone de saturation le gradient sera très vite trop faible. Cela portera défaut à l'apprentissage lors de l'application de la rétropropagation du gradient. La fonction introduit un hyperparamètre  $\lambda$ . Son allure est donnée par la figure 1.3.

$$\varphi_\lambda(x) = \frac{1}{1 + e^{-\lambda x}}$$

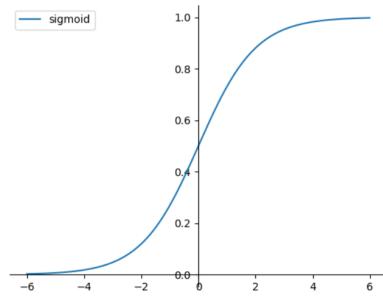


FIGURE 1.3 – Fonction sigmoïde

#### Tangente hyperbolique

Cette fonction permet d'avoir des valeurs de sortie entre -1 et 1. Elle souffre du même problème de *gradient vanishing* que la sigmoïde. Son allure est donnée par la figure 1.4.

$$\varphi(x) = \tanh(x)$$

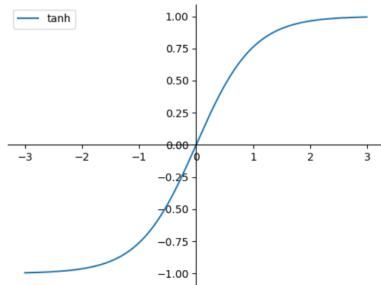


FIGURE 1.4 – Fonction tangente hyperbolique

### Softmax

Cette fonction s'applique sur un vecteur, elle est souvent utilisée en sortie car elle permet d'obtenir des sorties homogènes à des probabilités (entre 0 et 1 et dont la somme sur les sorties vaut 1).

$$\varphi_\lambda(X)_j = \frac{e^{-x_j}}{\sum_k e^{-x_k}}$$

### ReLU

La ReLU (pour *Rectified Linear Unit*) définie ci-dessous permet d'éviter le *gradient vanishing* puisqu'elle ne possède pas de zone de saturation. Son inconvénient réside dans sa partie de gradient nul. En effet, si sa pré-activation (la somme pondérée de ses entrées) est négative, l'effet du neurone sur les couches suivantes sera nul. Autrement, le gradient de l'erreur commise par ce neurone sera nul, il ne pourra donc pas mettre à jour ses poids et restera dans cette position pendant tout l'apprentissage. On dit que le neurone *meurt*. L'allure de la ReLU est donnée par la figure 1.5.

$$\varphi(x) = \max(0, x)$$

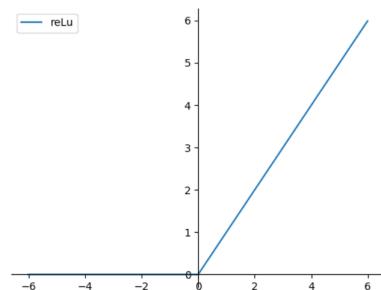


FIGURE 1.5 – Fonction reLu

### Et bien d'autres...

Il existe beaucoup d'autres fonctions d'activation ayant divers effets. Ce champ de recherche est très actif : de nouvelles fonctions sont découvertes chaque année dont l'effet n'est pas toujours bien compris. On peut par exemple citer la fonction Mish (2019) [11] ou la softexponentielle(2016) [12].

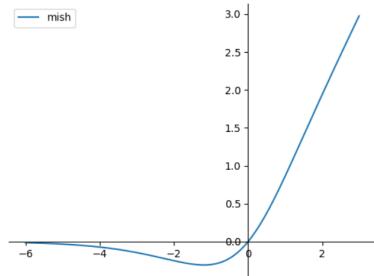


FIGURE 1.6 – Fonction Mish

## 1.4 Rétropropagation

### 1.4.1 Principe

On définit une fonction de coût  $J$ . Cette fonction mesure l'écart entre ce que renvoie notre MLP et ce qu'il devrait renvoyer. Notre objectif est donc de la minimiser afin d'obtenir la plus grande précision possible. Pour cela on va agir sur les seuls paramètres modifiables : les poids. On va ainsi calculer le gradient  $\nabla J$  à l'aide des dérivées partielles par rapport aux poids de la dernière couche de neurones. Dès lors, par la règle de la chaîne, on peut calculer les dérivées partielles par rapport aux poids de la couche précédente et ainsi de suite. Ces dérivées partielles seront utilisées par les algorithmes d'optimisation pour minimiser  $J$  en mettant à jour les poids par itérations successives. Le principe de la rétropropagation de l'erreur est illustré par la figure 1.7.

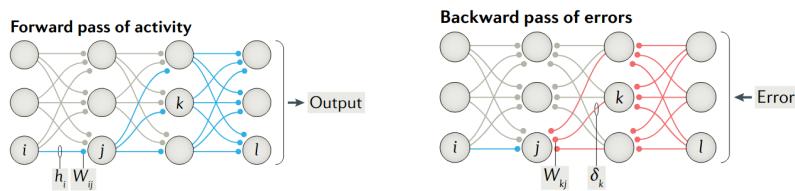


FIGURE 1.7 – Principe de la rétropropagation du gradient. À gauche, la propagation avant, qui permet de calculer la sortie du réseau en fonction de l'entrée. À droite, la rétropropagation de l'erreur, qui permet de calculer l'influence de chaque poids sur l'erreur en sortie. Ces deux schémas proviennent d'un article de Liliacrap et al. [13]

### 1.4.2 Les fonctions de coûts

Il en existe plusieurs, voici les principales.

#### Mean Absolute Error

Cette fonction compte les erreurs proportionnellement à leur importance.

$$MAE = \frac{1}{n} \sum_{i=0}^n |y_{th} - y_{mes}|$$

où  $y_{th}$  est la sortie théorique, et  $y_{mes}$  est la sortie prédite par l'algorithme.

#### Mean Squared Error

Parfois, on est prêt à tolérer de faibles erreurs mais on refuse catégoriquement les aberrations. C'est exactement ce que fait l'erreur quadratique moyenne. Elle attribue un poids beaucoup plus important aux grandes déviations.

$$MSE = \frac{1}{n} \sum_{i=0}^n (y_{th} - y_{mes})^2$$

où  $y_{th}$  est la sortie théorique, et  $y_{mes}$  est la sortie prédite par l'algorithme.

#### Entropie croisée

L'entropie croisée, introduite à l'origine pour mesurer la distance entre deux distributions de probabilités est très efficace pour les réseaux neuronaux dans le cas de la classification. En effet, dans ce cas il s'agit d'approximer une densité de probabilité.

$$CrossEntropy = - \sum_{i=0}^n y_{th} \log(y_{mes})$$

où  $y_{th}$  est la classe théorique, et  $y_{mes}$  est la classe prédite par l'algorithme.

### 1.4.3 Algorithmes d'optimisation

La fonction de coût utilisée dans toute cette partie est  $J = \sum_k (y_{th} - y_{mes})^2$

#### Descente de gradient stochastique

Il s'agit de l'algorithme classique.  $w_{jk}^{(i)} \leftarrow w_{jk}^{(i)} - \alpha \frac{\partial J}{\partial w_{jk}^{(i)}}$ . Un coefficient  $\alpha$  appelé **taux d'apprentissage** définit l'ampleur de la modification du poids. Plus  $\alpha$  est grand, plus le pas est important. Ainsi un taux d'apprentissage trop faible entraînera une convergence trop lente ou un blocage dans un minimum local mais un taux trop élevé risque de ne jamais converger. Un exemple d'application de la descente de gradient stochastique est donné par la figure 1.8.

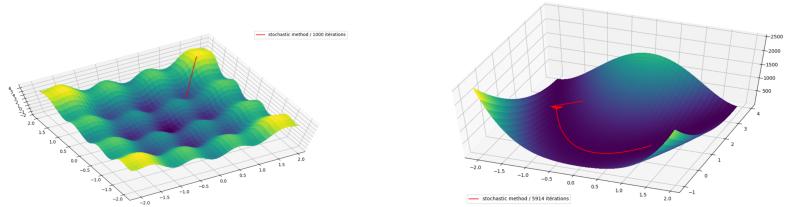


FIGURE 1.8 – Exemple d’application de la descente de gradient stochastique. La figure représente le cheminement de la valeur des deux poids. La hauteur représente l’erreur. On voit bien ici que l’algorithme peut entraîner un blocage dans un minimum local.

### Descente de gradient newtonienne

Ici, on ajoute un terme d’inertie à la modification du poids cela entraîne l’apparition d’un nouvel hyperparamètre  $\beta_1$  qui sert à doser cette inertie. On note  $\nu$  le terme de descente au pas précédent :

$$w_{jk}^{(i)} \leftarrow w_{jk}^{(i)} + \beta_1 \nu - \frac{\partial J}{\partial w_{jk}^{(i)}}$$

Un exemple d’application de la descente de gradient stochastique est donné par la figure 1.9.

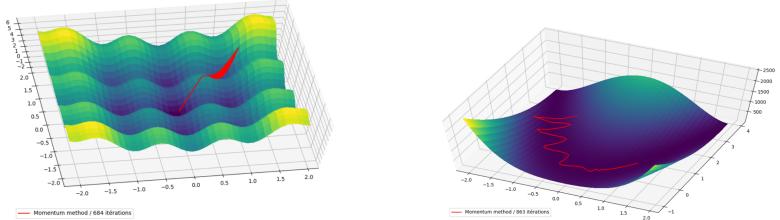


FIGURE 1.9 – Exemple d’application de la descente de gradient newtonienne. La figure représente le cheminement de la valeur des deux poids. La hauteur représente l’erreur. On remarque que l’algorithme est bien plus instable mais explore plus et ne reste pas forcément bloqué dans les minimum locaux.

### RMSProp

L’algorithme, dont un exemple d’application est donné sur la figure 1.10, est le suivant : on choisit les hyper-paramètres  $\beta_2$ ,  $\alpha$ , et  $\varepsilon$  tel que  $\varepsilon$  soit petit mais non nul, on initialise  $\nu$  à 0. A chaque pas on calcule :

$$\begin{aligned} s &\leftarrow \beta_2 s + (1 - \beta_2) \left( \frac{\partial J}{\partial w_{jk}^{(i)}} \right)^2 \\ w_{jk}^{(i)} &\leftarrow w_{jk}^{(i)} - \frac{\alpha}{\sqrt{s + \varepsilon}} \frac{\partial J}{\partial w_{jk}^{(i)}} \end{aligned}$$

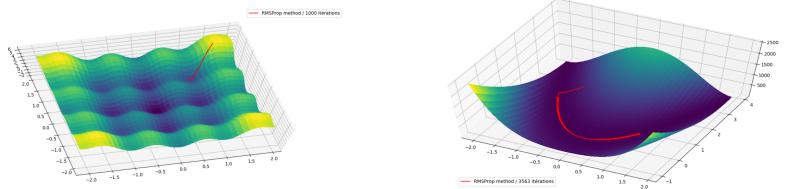


FIGURE 1.10 – Exemple d’application de RMSProp. La figure représente le cheminement de la valeur des deux poids. La hauteur représente l’erreur. L’algorithme est sensible aux minima locaux mais peu aux grands gradients.

## ADAM

Il s’agit d’une combinaison de RMSProp et de la descente de gradient newtonienne. L’algorithme est le suivant : on choisit les hyperparamètre  $\beta_1$ ,  $\beta_2$ ,  $\alpha$  et  $\varepsilon$  tel que epsilon soit petit mais non nul, on initialise  $\nu$  et  $s$  à 0. A chaque pas on calcule :

$$\begin{aligned}\nu &\leftarrow \beta_1\nu + (1 - \beta_1)\frac{\partial J}{\partial w_{jk}^{(i)}} \\ s &\leftarrow \beta_2s + (1 - \beta_2)(\frac{\partial J}{\partial w_{jk}^{(i)}})^2 \\ w_{jk}^{(i)} &\leftarrow w_{jk}^{(i)} - \frac{\alpha}{\sqrt{s + \varepsilon}}\frac{\partial J}{\partial w_{jk}^{(i)}}\end{aligned}$$

Un exemple d’application est donné sur la figure 1.11.

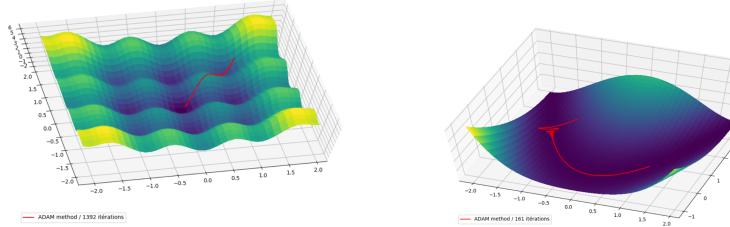


FIGURE 1.11 – Exemple d’application de ADAM. La figure représente le cheminement de la valeur des deux poids. La hauteur représente l’erreur. L’algorithme est peu sensible aux minima locaux et beaucoup plus stable que la méthode de descente de gradient newtonienne.

## Comparatif des algorithmes

Pour comparer les différents algorithmes, nous pouvons représenter sur un seul dessin leurs convergences. Le résultat est présenté par la figure 1.12.

### 1.4.4 Les batchs

Souvent, la dérivée  $\frac{\partial J}{\partial w_{jk}^{(i)}}$  n’est pas calculée sur une seule entrée. Afin d’obtenir une estimation plus précise du gradient de  $J$ , on le moyenne sur  $n$  entrées.

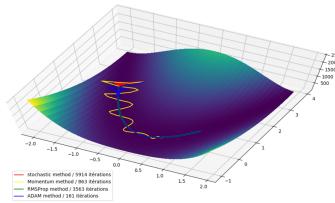


FIGURE 1.12 – Comparaison des 4 algorithmes présentés. La figure représente le cheminement de la valeur des deux poids. La hauteur représente l'erreur. On observe que ADAM est clairement meilleur que les autres.

On appelle cela la méthode par batchs. Il est bon de remarquer que le choix de  $n$  n'est pas anodin. Il s'agit ici encore d'un hyper-paramètre à régler qui influe sur la convergence du réseau. L'influence de la taille des batchs sera étudiée plus loin.

## 1.5 Implémentation et résultats

### 1.5.1 Avant-propos

Nous avons étudié l'impact de différents hyperparamètres sur les résultats du MLP. Sauf mention contraire, le jeu de données utilisé est MNIST. MNIST est un ensemble d'images en niveau de gris de chiffres écrits à la main. Le MLP est alors utilisé en tant que classificateur. Il doit reconnaître, à partir du niveau de gris des pixels, quel est le chiffre dessiné. La tâche typique de l'algorithme est présentée par une interface simple sur la figure 1.13.

Les résultats de précision sont calculés sur un jeu de données de test composé de données choisies aléatoirement avant l'entraînement et n'étant pas utilisées pour celui-ci. Afin de mesurer l'effet des hyperparamètres, on trace toujours les courbes avec tous les hyperparamètres égaux sauf celui qui varient. Les MLP utilisés pour les mesures ont été réalisés *from scratch*. La configuration à partir de laquelle on fait varier les hyper-paramètres a été obtenue par ajustement par essais-erreurs et ne prétend pas être optimale. Toutefois, les valeurs observées étaient de l'ordre de celles trouvées dans la littérature. Les données sont moyennées sur 50 lancers de l'algorithme et l'intervalle de confiance à 95% est indiqué.

### 1.5.2 Effet du nombre de neurones par couche

On s'intéresse ici à la largeur du réseau, c'est-à-dire-dire le nombre de neurones par couche. Les résultats sont présentés sur la figure 1.14.

### 1.5.3 Effet du nombre de couches de neurones

On s'intéresse ici à la profondeur du réseau. Les résultats sont présentés sur la figure 1.15.

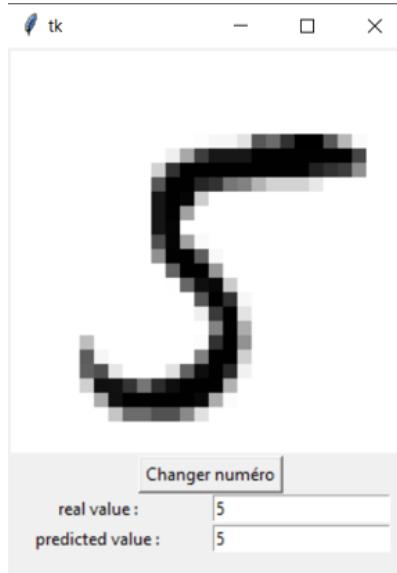


FIGURE 1.13 – Exemple de résultat de classification avec le MLP

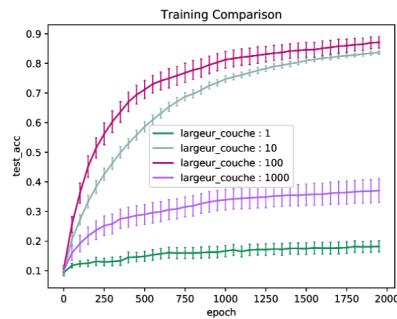


FIGURE 1.14 – Courbes de précision sur les données de test en fonction de l'avancement de l'apprentissage. On observe à la fois que prendre trop peu de neurones ne permet pas une classification satisfaisante, et à la fois que prendre trop de neurones est contre productif. Il s'agit alors de trouver le juste milieu.

#### 1.5.4 Taille des batchs

Comme vu précédemment la taille des batchs influe grandement sur l'algorithme.

##### Méthode stochastique

Il s'agit du cas où le batch est de taille 1. On met à jour les poids pour chaque exemple parcouru. On observe alors sur la figure 1.16 une forte instabilité liée au fait que le gradient peut fortement varier d'un exemple à l'autre.

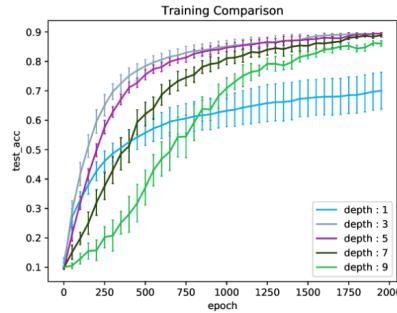


FIGURE 1.15 – Courbes de précision sur les données de test en fonction de l'avancement de l'apprentissage. Une seule couche correspond donc à un perceptron simple. On observe que le MLP est plus performant. Cependant augmenter la profondeur du réseau réduit sa vitesse de convergence.

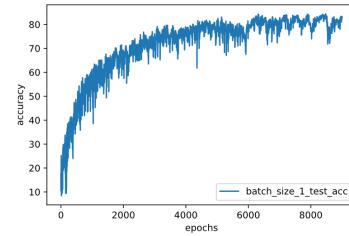


FIGURE 1.16 – Courbes de précision sur les données de test en fonction de l'avancement de l'apprentissage avec  $batch\_size = 1$ . Un epoch correspond au passage d'un batch dans l'algorithme.

### Méthode par batch

On illustre ici avec de très grands batch. On moyenne le gradient dans cet exemple sur 2048 images. Sur la figure 1.17, on n'observe plus l'instabilité initiale. Cependant de grands batch augmentent la sensibilité aux minima locaux et mène parfois à une saturation de la fonction de coût. De plus, une passe correspond au passage d'un batch dans l'algorithme. Ainsi, avec de très grands batch, le calcul de chaque pas peut devenir très long. Souvent, une grande taille de batch est associée à un fort taux d'apprentissage (car on estime mieux le gradient).

### Méthode par mini-batch

On choisit des batch de taille plus restreinte (ici 32 images). On réduit l'instabilité pour qu'elle ne gêne pas trop la convergence, de plus le coût de calcul reste raisonnable. Enfin on garde une légère instabilité qui rend l'algorithme plus résistant aux minima locaux. Le résultat est présenté sur la figure 1.18.

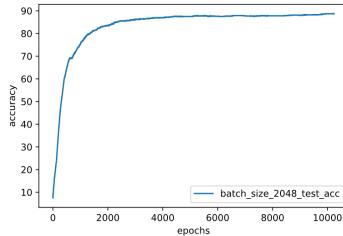


FIGURE 1.17 – Courbes de précision sur les données de test en fonction de l'avancement de l'apprentissage avec  $batch\_size = 2048$ . Un epoch correspond au passage d'un batch dans l'algorithme.

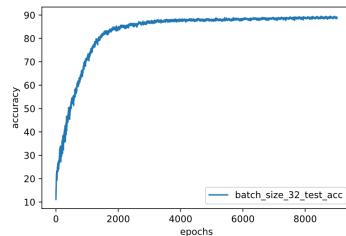


FIGURE 1.18 – Courbes de précision sur les données de test en fonction de l'avancement de l'apprentissage avec  $batch\_size = 32$ . Un epoch correspond au passage d'un batch dans l'algorithme.

### 1.5.5 Initialisation des poids

Si l'on veut faire évoluer les poids du réseau, il faut bien les initialiser. Ainsi, on peut s'intéresser à l'effet de leur initialisation sur l'apprentissage. On commence par s'intéresser à une initialisation par loi normale. La moyenne et l'écart-type sont alors deux hyper-paramètres. Grâce aux résultats présentés par la figure 1.19, on remarque bien qu'il est important de choisir des valeurs raisonnables pour obtenir une convergence.

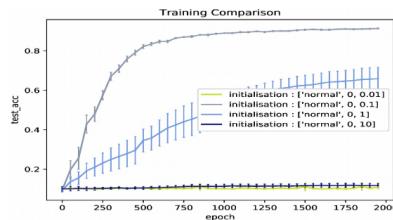


FIGURE 1.19 – Courbes de précision sur les données de test en fonction de l'avancement de l'apprentissage avec initialisation des poids selon une loi normale.

On s'intéresse ensuite au cas d'une initialisation selon une loi uniforme. Les bornes sont alors deux hyperparamètres. Ici encore, la figure 1.20 montre qu'il est important de choisir des valeurs judicieuses.

On remarque d'ailleurs que dans le cas optimal la courbe d'apprentissage est

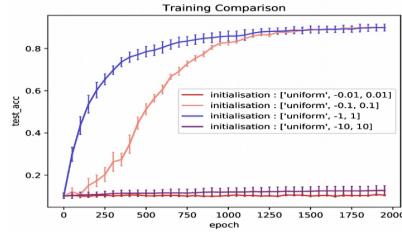


FIGURE 1.20 – Courbes de précision sur les données de test en fonction de l'avancement de l'apprentissage avec initialisation des poids selon une loi uniforme.

similaire pour les deux lois.

### 1.5.6 Taux d'apprentissage

Le taux d'apprentissage est un hyper-paramètre très important, il va déterminer la vitesse d'apprentissage et conditionner sa stabilité. Un taux d'apprentissage trop important risque de provoquer des oscillations dans la descente de gradient, voire même de faire diverger l'apprentissage. Alors qu'un taux d'apprentissage trop faible donnera un apprentissage lent. L'influence du taux d'apprentissage est donnée par la figure 1.21.

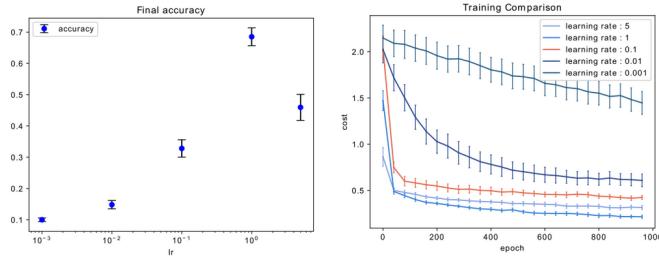


FIGURE 1.21 – Influence du taux d'apprentissage sur la convergence. À gauche, la précision atteinte sur les premiers batchs en fonction du taux d'apprentissage. À droite, l'évolution temporelle du coût en fonction du taux d'apprentissage. On observe l'existence d'une valeur optimale pour le taux d'apprentissage, celle-ci dépend des autres hyper-paramètres. Les essais sont moyennés sur 50 essais, sur les premiers batchs seulement, ce qui explique que l'on n'atteint pas une bonne précision.

# 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)**. L'idée de cette architecture a été introduite en 1980 [14] et a ensuite été développée par Yann LeCun [3] pour donner naissance plus récemment à des réseaux comme AlexNet [15] ou Inception [16]. Les couches à convolutions sont maintenant indispensables à la plupart des réseaux traitant des informations structurées compositionnelles.

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

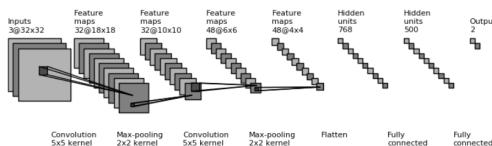


FIGURE 2.1 – Structure simplifiée d'un CNN. L'image est tirée de [17]

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 ou plusieurs couches denses qui jouent le rôle d'un classifieur classique.

### 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 (*kernel*) capable de reconnaître un motif en particulier. Il suffit alors de balayer l'image pixel par pixel (le balayage sera expliqué 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 capables 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. Il nous est possible de la réduire en taille (étape de *pooling*) pour réduire le nombre de calculs à effectuer. Le processus se répète un certain 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 cœur 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 :

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

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

La figure 2.2 donne 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. L'image provient de Wikipedia.

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 se 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. La figure 2.3 donne un exemple de balayage d'un noyau sur une image.

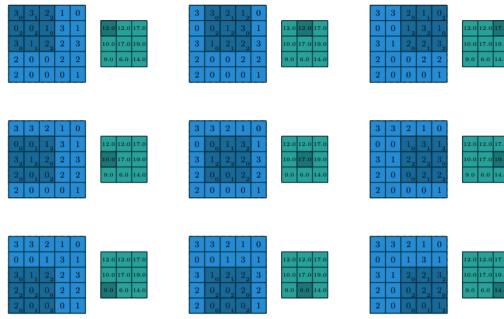


FIGURE 2.3 – Fonctionnement de la couche de convolution sur un exemple simple. L'image est tirée de l'article de Dumoulin et al. [18].

On remarque alors que le résultat a 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 présentée sur la figure 2.4

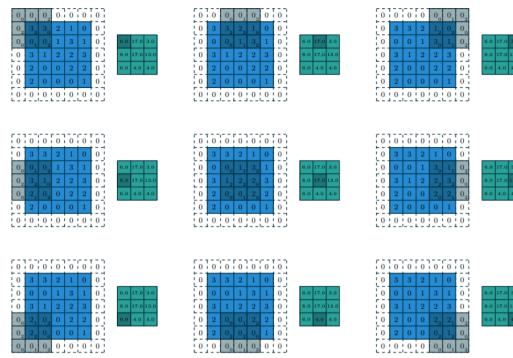


FIGURE 2.4 – Fonctionnement de la couche de convolution avec du same padding. L'image est tirée de l'article de Dumoulin et al. [18].

Il existe encore un paramètre permettant de personnaliser l'opération : le

**stride.** Celui correspond au décalage à utiliser pour le noyau. Ce processus est représenté sur la figure 2.5.

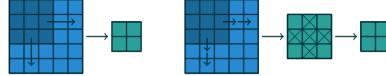


FIGURE 2.5 – Fonctionnement de la couche de convolution avec un stride valant 2. À gauche, le résultat de l'opération. À droite, une façon différente de voir les choses : le noyau balaye toute la matrice avec un pas de 1 mais ne garde les valeurs que pour les déplacements pairs. L'image est tirée de l'article de Dumoulin et al. [18].

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 provoque 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. Un exemple d'une telle opération est représenté par la figure 2.6.

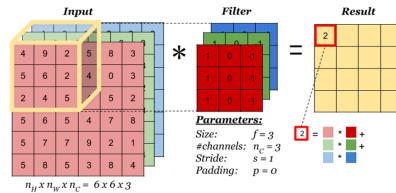


FIGURE 2.6 – Exemple de convolution en 3D. L'image est tirée de l'article *1D and 3D Convolution* de S.Verma [19].

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 pour compresser l'information. On considère généralement deux types de couches de pooling :

- max pooling : on applique l'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. Un exemple de fonctionnement est présenté par la figure 2.7.

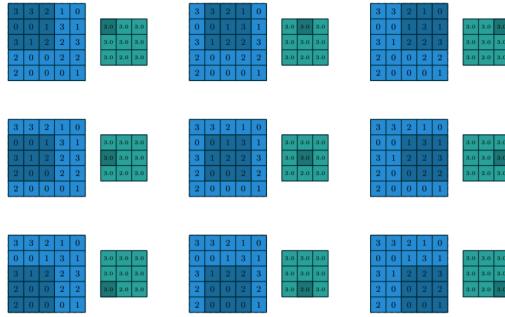


FIGURE 2.7 – Exemple simple d'application de la couche max pooling. L'image est tirée de l'article de Dumoulin et al. [18].

- average pooling : le fonctionnement est le même que la couche de max pooling à sauf que l'on applique la fonction moyenne au lieu de la fonction maximum. Un exemple de fonctionnement est présenté par la figure 2.8.

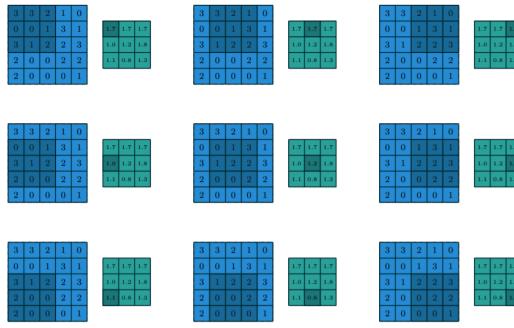


FIGURE 2.8 – Exemple simple d'application de la couche avg pooling. L'image est tirée de l'article de Dumoulin et al. [18].

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 précédents 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 égale au nombre de noyaux, contenant dans chacune des couches le résultat du produit de convolution pour un noyau. La figure 2.9 illustre ce principe.

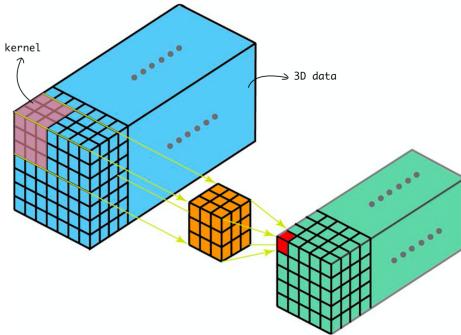


FIGURE 2.9 – Exemple du résultat à la sortie d'une couche de convolution d'un CNN avec une structure complète. L'image est tirée de l'article *1D and 3D Convolution* de S.Verma [19].

On alterne ainsi entre couches de convolution pour prélever des motifs de plus en plus complexes et couches de pooling pour compresser les données. 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. Les calculs sont essentiellement une généralisation de la rétropropagation présenté au chapitre 1. Pour plus d'informations, nous invitons le lecteur à consulter la littérature scientifique à ce sujet.

### 2.2.7 Dropout

Le principal problème de ce type d'architecture est le phénomène de surapprentissage. Pour pallier ce problème, des améliorations ont été proposées, notamment en 2014 par Srivastava et al. [20]. Les auteurs proposent en effet d'éteindre de manière aléatoire une certaine proportion de neurones (qui est un hyper-paramètre du dropout) pendant la phase de propagation avant (*forward propagation*). Ils ont nommé cette technique le **dropout**. Cela permet de réduire la co-adaptation entre les neurones, évitant ainsi qu'un neurone ne prenne trop d'importance dans le processus d'apprentissage, ce qui pourrait amener à du sur-apprentissage. Une explication visuelle est proposée par la figure 2.10.

La figure 2.11 présente les résultats de ce même article. Les résultats sont plus que positifs dans la mesure où la qualité prédictive du réseau est grandement améliorée par le dropout.

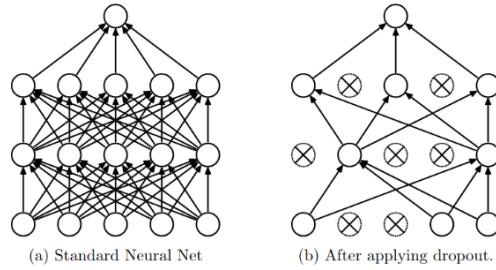


FIGURE 2.10 – A gauche, une structure MLP classique. A droite, un exemple de l’effet du dropout (paramètre ajusté à 50%). Cette image vient de l’article de Srivastava et al. [20].

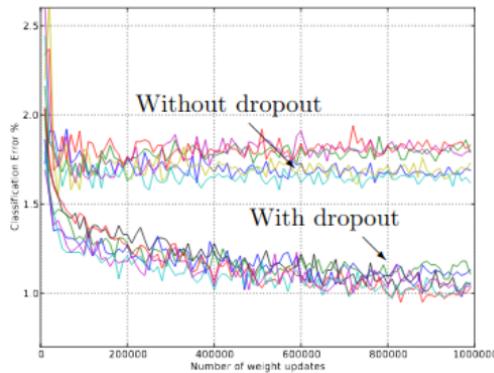


FIGURE 2.11 – Résultats de l’article sur l’efficacité du dropout sur un réseau neuronal. Cette image vient de l’article de Srivastava et al. [20].

## 2.3 Implémentation et résultats

### 2.3.1 Implémentation

Nous avons une nouvelle fois utilisé TensorFlow 2.0 qui permet l’utilisation rapide de couches telles que celles de convolution, de pooling mais aussi de *flattening*. La couche de *flattening* est une couche permettant de faire la transition entre la dernière couche de convolution et la première couche dense. En d’autres termes, elle transforme les données en 3 dimensions en un tableau en 1 dimension.

La structure du réseau simplifié avec lequel nous avons fait les tests est la suivante :

```

Conv2D(64, 3, activation = relu)
Conv2D(32, 3, activation = relu)
Flatten()
Dense(128, activation = relu)
Dense(10, activation = softmax)

```

Nous avons de plus utilisé l’optimiseur ADAM pour effectuer la descente de gradient.

### 2.3.2 Résultats

Nous avons tout d'abord voulu tester la taille des noyaux pour voir quelle influence celle-ci a sur les performances de notre CNN. La figure 2.12 présente les résultats sur la base de données MNIST.

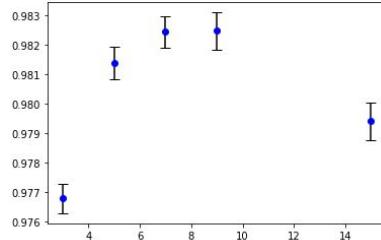


FIGURE 2.12 – Comparaison de la précision de notre CNN en fonction de la taille des noyaux. En ordonnée la précision, en abscisse la taille des noyaux. La base de données utilisée est MNIST.

Nous pouvons en déduire que des noyaux de taille trop grande par rapport aux images d'origine risquent de nuire à l'efficacité du CNN.

Nous nous sommes, de plus, intéressés à l'influence du **stride** et du **padding** sur les performances du CNN. La figure 2.13 présente les résultats de notre analyse sur MNIST.

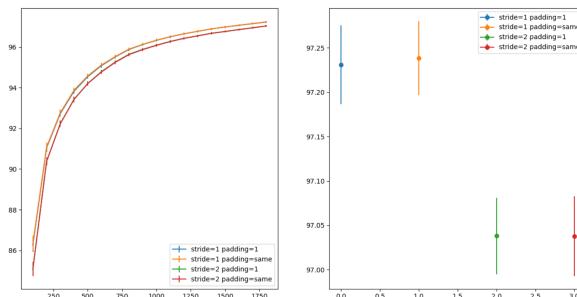


FIGURE 2.13 – Comparaison de la précision de notre CNN en fonction des paramètres stride et padding. En ordonnée la précision, en abscisse le nombre de batchs d'entraînement. La base de données utilisée est MNIST.

Nous pouvons conclure que, sur cette base de données, l'utilisation d'un stride supérieur à 1 a comme effet de réduire les capacités prédictives du CNN. En effet, lorsque ce paramètre est trop élevé, la perte d'information est trop importante, nuisant aux performances de l'algorithme. Nous remarquons de plus que le padding n'a ici aucune influence. Ce résultat n'est pas surprenant car les images de MNIST sont centrées avec des bords sombres. Ainsi il n'y a aucune information sur les bords, rendant le changement de padding quasiment inutile.

De même, nous avons testé l'influence du **dropout**. Les résultats sont présentés sur la figure 2.14. Les résultats sont beaucoup moins significatifs que ce

que laisse présager l'article de Srivastava et al. [20]. Il semble donc que le dropout n'a aucune influence sur les résultats de notre CNN. Ceci est probablement dû à la banque d'images MNIST et à la petite taille de notre réseau, qui ne sont pas propices au sur-apprentissage .

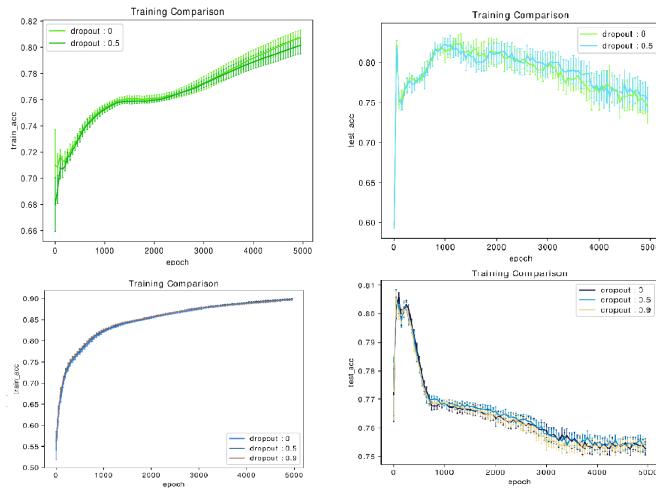


FIGURE 2.14 – Influence du dropout sur l'efficacité de nos réseaux. En haut, un CNN sur la base de données MNIST. En bas, un MLP sur la base de données Moon. Nous remarquons que les courbes sont toutes confondues, quel que soit le paramètre de dropout choisi.

## 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 entrée des images et les classe en deux catégories en leur associant leur probabilité d'être issues de la base de données 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 données (images réelles), et celles générées par le générateur. Son 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'optimiser D afin qu'il améliore la justesse de sa classification.

L'architecture des GAN a été introduite pour la première fois par Ian Goodfellow [21] en 2014. Cet article innovant montrait déjà un gain de performance pour la génération d'images suivant une base de données. Mais l'atout majeur des GAN sont leur adaptabilité à tous types de données.

### 3.2 Le DCGAN (Deep Convolutionnal Adversarial Network)

Le DCGAN utilise la fonction de coût proposée par Goodfellow [21], mais le générateur et le discriminateur sont tous les deux des **réseaux à convolutions** [22]. La fonction de coût de G à minimiser est la suivante :

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

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

Une variation int ressante 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 r t de cette modification r sulte 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 es 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 sout ce probl me.

### 3.3 tude de la convergence des GAN

De par leur caract re d'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

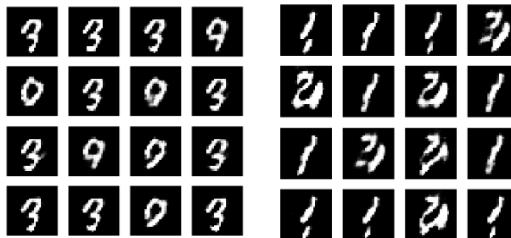


FIGURE 3.1  Exemples d'effondrement des modes sur la banque de chiffres MNIST.



FIGURE 3.2  A gauche, un exemple d'effondrement des modes sur la banque d'images CelebA. A droite, une g n ration sans effondrement pour comparaison. On observe que sur l'image de gauche, tous les personnage ont la m me t te.

L'effondrement des modes survient quand le r seau g n rateur ne g n re pas des images conformes   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 définies. La manifestation de ce phénomène se traduit par des images générées qui se ressemblent toutes. Les figures 3.1 et 3.2 montrent des exemples du phénomène sur la base de données MNIST et CelebA.

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 visualisation ne permet donc pas de séparer efficacement les différentes classes, elle permet seulement un aperçu de la distribution des images de la banque. Les figures 3.3 et 3.4 présentent une visualisation de MNIST par transformation t-SNE [23].

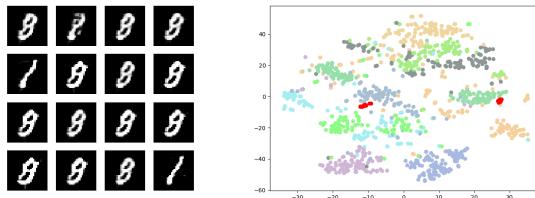


FIGURE 3.3 – On observe un effondrement à deux modes. Le GAN ne génère que des chiffres 1 et des chiffres 8, correspondant aux point rouges sur la représentation t-SNE.

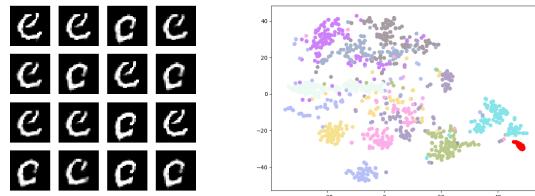


FIGURE 3.4 – On observe un effondrement à un mode, donc tous les points rouges sont regroupés. De plus le GAN génère un symbole non présent dans la base MNIST, donc les points rouges ne correspondent à aucun nuage de points de la banque d'images.

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 données 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 petite partie de la distribution de la base de données 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, c'est la

*minibatch discrimination.* Cela consiste à ajouter un terme à la fonction de coût pour traduire la similarité (il peut s'agir de calculer une similarité pixel à pixel, ou d'estimer la similarité sémantique grâce 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 tout l'espace des images réelles.
- Certaines architectures sont plus résistantes que d'autres à l'effondrement des modes. Par exemple, les GAN de Wasserstein [3.4.3] ne présentent pas 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 dans la figure 3.5, 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.

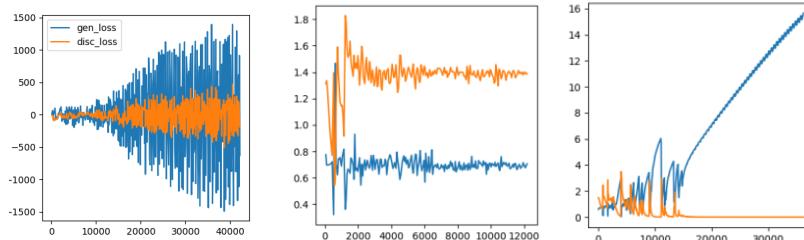


FIGURE 3.5

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 données 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  $p_{\text{data}}$  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.

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

Comme il est impossible de résoudre directement la formule de Bayes, il nous faut définir une fonction de coût qui mesure la distance entre  $p_\theta$  et  $p_{\text{data}}$ , 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  $\delta$  pour mesurer l'écart entre deux distributions  $p_{\text{data}}$  et  $p_\theta$ , 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 [21] 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  $g_\theta$ ) et D (ici  $f_w$ ). En effet, on a la relation suivante :

$$\delta(p_{\text{data}}, p_\theta) = \sup_{f \in \mathcal{F}} \mathcal{L}_{\text{DCGAN}}(g_\theta, f, p_{\text{data}}, p_{\text{bruit}})$$

### 3.4.3 WGAN

Les Wasserstein GAN, ou WGAN, ont été introduits en 2017 par Arjovsky et al. [5]. Les auteurs y introduisent une nouvelle distance, la distance 1-Wasserstein (qu'on appellera ici distance Wassertein), définie par la relation suivante :

$$W(p_{\text{data}}, p_{\theta}) = \inf_{\gamma \in \Pi(p_{\text{data}}, p_{\theta})} \mathbb{E}_{(x,y) \sim \gamma} (\|x - y\|)$$

Avec  $\Pi(p_{\text{data}}, p_{\theta})$  l'ensemble des densités de distributions jointes  $\gamma(x, y)$  de lois marginales respectivement  $p_{\text{data}}$  et  $p_{\theta}$ . On peut réécrire ce résultat à l'aide de la formulation duale du théorème de Kantorovich [24].

$$W(p_{\text{data}}, p_{\theta}) = \sup_{f \in L_1} (\mathbb{E}_{x \sim p_{\text{data}}} [f(x)] - \mathbb{E}_{x \sim p_{\theta}} [f(x)])$$

Avec  $L_1$  l'ensemble des fonctions 1-lipschitziennes  $\mathcal{X} \rightarrow \mathbb{R}$ . On obtient donc une équation assez similaire à celle du paragraphe 3.4.2. Nous allons estimer  $f$  par un réseau de neurones  $\{f_w, w \in \mathcal{W}\}$  qui sera optimisé à l'aide du gradient  $\mathbb{E}_{x \sim p_{\text{data}}} [\nabla_w f_w(x)] - \mathbb{E}_{z \sim p_{\text{bruit}}} [\nabla_w f_w(g_{\theta}(z))]$ . De même, à  $f$  fixé,  $g_{\theta}(z)$  s'optimise par rétropropagation du gradient selon  $\theta$ .

### 3.4.4 Avantages comparatif du WGAN par rapport au DCGAN

Maintenant que nous savons ce qu'est un WGAN, il s'agit de comprendre son avantage par rapport au DCGAN. D'abord, la distance  $W$  est topologiquement plus faible que la divergence de Jensen-Shanon, et donc par extension que  $\delta$ . Ceci signifie qu'il est plus facile en pratique de faire converger un WGAN qu'un DCGAN.

Ensuite, à chaque boucle d'apprentissage, le WGAN peut mieux optimiser le discriminateur avant de faire apprendre le générateur. En effet, pour un WGAN, meilleur est le discriminateur, meilleur est le gradient utilisé pour l'apprentissage du générateur. Ceci permet de résoudre le problème d'instabilité au début de l'apprentissage rencontré dans les DCGAN, pour lesquels un discriminateur trop bon fait saturer le gradient du générateur à 0.

Finalement, la capacité du WGAN d'entraîner d'abord le discriminateur empêche le phénomène du mode collapse. En effet, la cause du mode collapse est que pour un discriminateur fixé, le meilleur générateur est celui qui ne génère que les points de  $\mathcal{X}$  de plus grande valeur pour le discriminateur.

## 3.5 Implémentation et résultats

Nous avons implémenté avec succès l'algorithme WGAN pour la base de donnée MNIST, et DCGAN pour les banques de données MNIST et CelebA.

### 3.5.1 Détails de l'implémentation

Nous avons utilisé la même structure de réseau pour le DCGAN et le WGAN, en changeant uniquement la fonction de coût.

Pour le discriminateur :,

```

Conv2D(64, (5, 5), strides = (2, 2))
LeakyReLU()
Dropout
Conv2D(128, (5, 5), strides = (2, 2))
BatchNormalization
LeakyReLU()
Dropout
Conv2D(256, (5, 5), strides = (2, 2))
BatchNormalization
LeakyReLU()
Dropout
Flatten()
Dense(1)
LeakyReLU()

```

Pour le générateur :

```

Dense(240)
LeakyRelu()
Reshape((10, 8, 3))
Conv2DTranspose(256, (5, 5), strides = (2, 2))
BatchNormalization
LeakyReLU()
Conv2DTranspose(64, (5, 5), strides = (2, 2))
BatchNormalization
LeakyReLU()
Conv2DTranspose(3, (5, 5), strides = (2, 2))

```

### 3.5.2 MNIST

Nous présentons ici les résultats obtenus après 50 passes sur la base de données MNIST, à l'aide des algorithmes DCGAN et WGAN. On observe que nous avons eu plus de mal dans l'implémentation du WGAN, et n'arrivons pas à obtenir le gain de performance prédit par l'article [5].

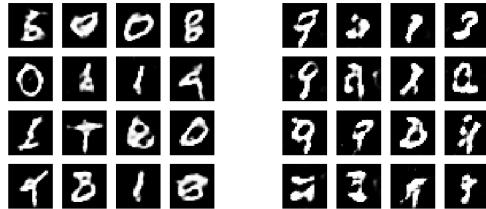


FIGURE 3.6 – A gauche, 16 images générées par le DCGAN. A droite, 16 images générées par le WGAN

### 3.5.3 CelebA

Nous présentons ici les résultats obtenus après 400 passes sur la base de données CelebA. Nous avons donc réussi à construire avec succès un générateur de visages réalistes sans mode collapse.



FIGURE 3.7 – 39 images obtenues par DCGAN sur la base CelebA

# Chapitre 4

## Le cycleGAN

### 4.1 Présentation de la problématique

Les cycleGAN, proposées par Zhu et al. [1], sont des architectures dérivées des GAN qui permettent de répondre à une problématique bien spécifique : le **transfert de style non appairé** [25]. 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'oranges, de transformer un paysage d'été en un paysage d'hiver, de transformer une musique classique en musique rock, ou encore de modifier les expressions faciales d'individus présents sur une image. Quelques exemples sont présentés sur la figure 4.1.

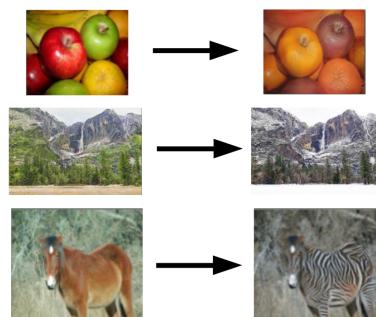


FIGURE 4.1 – Exemples de transfert de style effectués par un cycleGAN. De haut en bas : transformation *pommes ↔ oranges*, transformation *paysages d'été ↔ paysages d'hiver* et transformation *chevaux ↔ zèbres*. Ces exemples sont tirés de l'article de Zhu et al. [1].

Le transfert de style peut se faire entre deux *classes de styles* ou plus, mais nous allons ici nous concentrer sur 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 ici A et B, et par extension nous parlerons du style A et du style B pour faire référence aux styles de ces banques de données. 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.

*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.

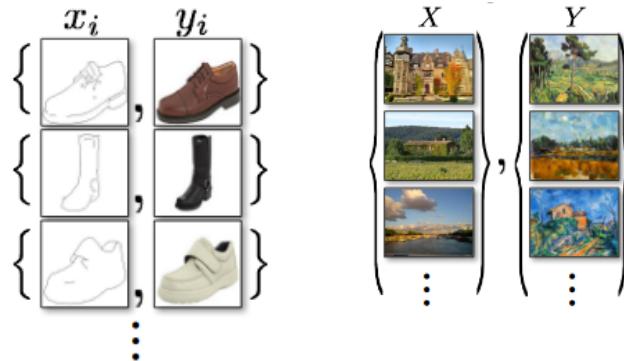


FIGURE 4.2 – Les deux types de transferts de style. À gauche, le transfert de style appairé où chaque donnée est associée à son équivalent dans un autre style. À droite, le transfert de style non appairé où les images n'ont pas d'équivalent dans l'autre style. Ces images sont tirées de l'article de Zhu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

Ces deux types de transferts 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 manièrée efficace par différents algorithmes, dont **Pix2Pix** proposé par Isola et al. en 2016 [26]. Le transfert de style non appairé quant à lui ne permet pas la comparaison à l'image-cible

puisque'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 paragraphe 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 ces données sont des images. Le but est de transformer les images de A pour leur donner le style des images de B, et inversement.

Par ailleurs, on remarque qu'en considérant la segmentation comme un style pour l'image, le cycleGAN peut aussi résoudre des problèmes de segmentation, même si il existe des algorithmes beaucoup plus efficaces spécialisés sur ce problème.

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 les images du style A et celles du style B. L'architecture est présentée par la figure 4.3.

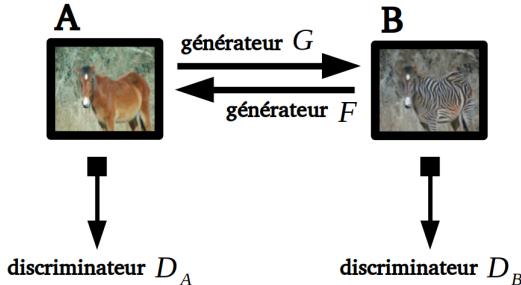


FIGURE 4.3 – Structure globale du cycleGAN. Le générateur G crée des images du style de B à partir d'images du style de A. Le générateur F crée des images du style de A à partir d'images du style de B. Chaque banque d'images, associée à un style, possède son discriminateur.

Comme on l'a entrevu dans le paragraphe précédent, une difficulté est que les données ne sont pas appariées, la fonction de coût ne peut donc pas venir de la comparaison directe de l'image générée à l'image souhaitée. Pour pallier 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 à chacun des deux GAN ainsi définis est appelé *Adversarial Loss* ou *GAN Loss*. La figure 4.4

montre la décomposition du cycleGAN en deux GAN.

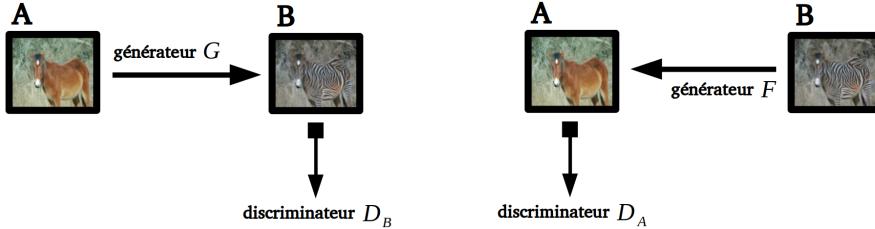


FIGURE 4.4 – Décomposition du cycleGAN en deux GAN distincts.

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 :

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

Le coût qui en découle (et qui sera détaillé dans la suite), est appelé *Cycle Consistency Loss*. Les deux égalités ci-dessus consistent en réalité à parcourir le cycle respectivement en avant et en arrière, elles sont appelées respectivement *backward cycle consistency* et *forward cycle consistency* et sont utilisées depuis longtemps dans le suivi d'objets [27]. Ceci est décrit de manière schématique par la figure 4.5.

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,  $F \circ G = \mathbb{1}$ . 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 [28]. Avec les mêmes notations que dans le paragraphe précédent, en considérant le générateur G et son discriminateur associé  $D_B$ , on a :

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

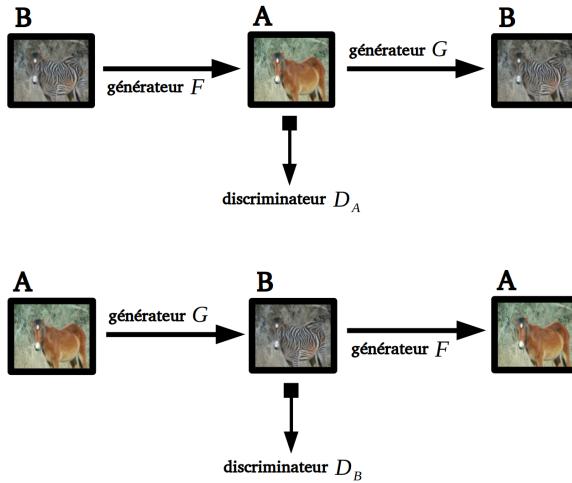


FIGURE 4.5 – Décomposition du cycleGAN en deux GAN distincts et ajout des contraintes de consistance cyclique. En haut, la consistance cyclique arrière (*backward cycle consistency*). En bas, la consistance cyclique avant (*forward cycle consistency*).

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

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

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

### 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  $G \circ F = \mathbb{1}$ . Il est important de noter que l'on veut un coût qui n'intervienne pas à une hauteur sémantique. On considère donc deux 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 donc 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 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 ne sont pas inchangées 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 évoluent les générateurs du cycleGAN, par une technique introduite par Taigman et al. [29]. L'idée consiste à ajouter un coût demi-cyclique qui tend à ce que  $F \approx 1$  et  $G \approx 1$ . 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 pas 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, appelées 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 par exemple la précision 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.

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

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, donc la métrique précédente n'est pas utilisable. Il en existe d'autres métriques, par exemple le **score FCN**. Le score FCN consiste à évaluer l'interprétabilité des images de sortie par un algorithme classique de segmentation sémantique (ici le FCN, pour *Fully Convolutional Networks for Semantic Segmentation* [30]). 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 les 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 le caractère réaliste de l'image, à travers l'interprétabilité de l'image par un autre algorithme. En somme, il n'existe aucune métrique idéale pour évaluer les cycleGAN.

## 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 des GAN préconisée dans l'article de Zhu et al. [1], qui a été proposée par Johnson et al. [31] 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 [32]. 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 par Ulyanov et al. [33].

**Pour le générateur**, toujours comme dans l'article de Zhu et al. [1], nous avons utilisé un réseau résiduel (proposé par He et al. [34]). 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 × [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).

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 modifiables depuis un unique fichier.

Pour les paramètres d'apprentissage, nous avons aussi suivi les conseils de l'article puis nous avons adapté les valeurs à chaque banque d'images en testant différentes valeurs possibles. 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. [35] et reprise par Zhu et al. pour les cyclesGAN. Nous l'avons aussi implémentée. Il en découle un nouvel hyper-paramètre : la taille du *buffer* contenant l'historique des images générées. 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’images 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’images *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 image dans le style des 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 grandes dimensions, les résultats pourraient être améliorés si nous pouvions exécuter nos scripts plus

longtemps. Pour l'instant, nous sommes ralenties 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 cohérents avec les implémentations de référence, nous sommes plutôt confiants quant à la qualité de notre implémentation. Son caractère modulaire et sa paramétrabilité très facile permettent de l'adapter à beaucoup de banques d'images différentes très rapidement. C'est ce qui nous permet d'utiliser notre script pour des problèmes non abordés dans l'article de Zhu et al. [1], comme les sourires sur CelebA. De plus, on peut l'utiliser de manière équivalente sur nos CPU personnels et sur le cluster de GPU Fusion.

Une fois les derniers détails d'implémentations réglés (dans peu de temps) nous pourrons utiliser notre programme pour étudier un nouveau problème, que nous devons définir.

# Conclusion

Tout au long du rapport, nous avons exploré des concepts et outils du *machine learning*. Le découpage du rapport suit le déroulement temporel du projet tout au long de cette année, qui nous a emmené des bases du *machine learning* avec le perceptron multicouches, jusqu'au cycleGAN, en passant par les réseaux à convolutions et les GAN. À chacune des étapes, nous avons pu comprendre en profondeur les outils que l'on manipule et nous avons pu apprendre à les expliquer. Chaque étape s'est soldée d'une implémentation fonctionnelle des algorithmes.

Nous avons pu arriver au bout du cheminement et implémenter un cycleGAN de qualité, capable d'être exécuté sur le mésocentre, mais encore perfectible sur quelques points. C'est un ingrédient indispensable pour la poursuite du projet, puisqu'il nous permettra maintenant de nous attaquer à un problème concret et novateur, que nous sommes - pour l'instant - encore en train définir. Nous pouvons entrer avec confiance dans la deuxième phase du projet grâce à tous ces outils nécessaires à sa réussite.

# 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] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,”
- [5] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein gan,”
- [6] P. Goldsborough, “A tour of TensorFlow,”
- [7] W. S. McCulloch and W. Pitts, “A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY,” vol. 5, pp. 115–133.
- [8] N. Benharir, M. Zerikat, S. Chekroun, and A. Mechernene, “Approche adaptative d’une commande neuronale sans capteur d’un moteur asynchrone associée à un observateur par mode glissant,”
- [9] F. Rosenblatt, “The perceptron : A probabilistic model for information storage and organization in the brain.,” vol. 65, no. 6, pp. 386–408. Place : US Publisher : American Psychological Association.
- [10] M. Minsky and S. A. Papert, *Perceptrons : An introduction to computational geometry*. MIT press.
- [11] D. Misra, “Mish : A self regularized non-monotonic neural activation function,”
- [12] L. B. Godfrey and M. S. Gashler, “A continuum among logarithmic, linear, and exponential functions, and its potential to improve generalization in neural networks,”
- [13] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, “Backpropagation and the brain,” vol. 21, no. 6, pp. 335–346.
- [14] K. Fukushima, “Neocognitron : A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” vol. 36, no. 4, pp. 193–202.
- [15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc.

- [16] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,”
- [17] Jefkine, “Backpropagation in convolutional neural networks.” Library Catalog : [www.jefkine.com](http://www.jefkine.com).
- [18] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,”
- [19] S. Verma, “Understanding 1d and 3d convolution neural network | keras,” Library Catalog : [towardsdatascience.com](http://towardsdatascience.com).
- [20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout : A simple way to prevent neural networks from overfitting,” p. 30.
- [21] 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.
- [22] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,”
- [23] L. van der Maaten and G. Hinton, “Visualizing data using t-SNE,”
- [24] C. Villani, *Optimal Transport : Old and New*.
- [25] 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.
- [26] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,”
- [27] 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.
- [28] I. Goodfellow, “NIPS 2016 tutorial : Generative adversarial networks,” in *arXiv preprint arXiv :1701.00160*.
- [29] Y. Taigman, A. Polyak, and L. Wolf, “Unsupervised cross-domain image generation,”
- [30] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,”
- [31] J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual losses for real-time style transfer and super-resolution,”
- [32] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,”
- [33] D. Ulyanov, A. Vedaldi, and V. Lempitsky, “Instance normalization : The missing ingredient for fast stylization,”
- [34] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,”
- [35] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb, “Learning from simulated and unsupervised images through adversarial training,”