

Faculté Polytechnique



Sujet TCTS-15 Apprendre à collaborer par le langage et la vision

Travail de fin d'études

Michaël ROMBAUX

Master ingénieur civil en informatique et gestion, à finalité spécialisée en web et stratégies d'entreprises
Charleroi



Sous la direction de :
Monsieur le Professeur Thierry DUTOIT (promoteur)
Monsieur Jean-Benoit DELBROUCK... (copromoteur)
Monsieur ... (rapporteur)
Septembre 2019

Résumé

Ce travail se base sur des méthodes d'intelligence artificielle et de Deep Learning.

Ce document propose un outil dans la compréhension des réseaux de neurones et plus particulièrement les CGANs (Conditional Generative Adversarial network).

Les GANs sont des réseaux entraînés pour faire des imitations, les CGANs en sont une évolution « labélisée ».

Les différents algorithmes ont été développés en langage python sur l'IDE Visual Code Studio. Les codes sources et la documentation se retrouvent sur un github : <https://github.com/rombaux/PyTorch-GAN>

Les algorithmes ont été exécutés sur GPU sur la plateforme Google Colaboratory car il est nécessaire d'avoir des machines puissantes effectuant de nombreux calculs vectoriels. Un simple CPU ne serait pas assez puissant et le temps d'apprentissage serait extrêmement long.

Un bref descriptif de ces réseaux CGAN : deux réseaux sont en concurrence, un générateur et un discriminateur, ce document va expliquer plus en profondeur le fonctionnement.

Cette méthode a été appliquée à différentes sources d'images :

- Noir et blanc
- Niveau de gris
- Couleurs

Les résultats d'imitation seront présentés. Des vidéos ont aussi été créées pour voir l'évolution de l'apprentissage et placées sur github.

1 Table des matières

2	Introduction	7
2.1	Objectif demandé.....	7
2.2	Les outils informatiques pour le développement.	7
2.2.1	L'environnement de développement « Visual Studio code.	7
2.2.2	Le langage de programmation Python.	7
2.2.3	L'hébergement et le versioning du code.....	7
2.2.4	L'environnement Google Colaboratory.	8
2.2.5	L'espace de stockage.	8
2.2.6	Les datasets utilisés.....	9
2.3	Les neurones et le perceptron.....	12
2.4	Les réseaux de neurones.....	14
2.5	Fonction d'activation.....	15
2.5.1	Fonction linéaire.	15
2.5.2	Fonction non linéaire.....	15
2.6	Apprentissage.....	17
2.6.1	L'apprentissage supervisé.....	18
2.6.2	L'apprentissage non supervisé.....	18
2.6.3	Introduction au Epoch – Batch size.	18
2.7	Les réseaux antagonistes génératifs (GAN).....	18
2.7.1	Principe de la Rétropropagation.	19
2.8	Les réseaux antagonistes génératifs conditionnels.	20
2.8.1	Random noise z.....	21
2.9	Représentation schématique des GAN.....	22
2.10	Représentation schématique des CGAN	23
3	Les différents algorithmes produits	24
3.1	L'apprentissage (cgan_train.py)	24
3.1.1	La mise en place.....	24
3.1.2	L'installation	24
3.1.3	L'espace latent.	26
3.1.4	La « class Generator(nn.Module) »	26
3.1.5	Les datasets.....	30
3.1.6	Les tenseurs	30
3.1.7	Les données et les blocs	31

3.1.8	Extraction et sauvegarde des modèles en fichier *.pth	32
3.2	Le Générateur (cgan_generate.py)	33
4	L'exécution du code	34
4.1	L'apprentissage.....	34
4.2	Le test des modèles.....	35
5	Les résultats d'images générées.	37
5.1	Dataset n°0 – MNIST.....	37
5.2	Dataset n°1 – Cifar 10 (2019-08-15_10-36).....	38
5.2.1	Analyse par classe :	38
5.3	Dataset n°2 – Cifar 100 (2019-08-14_17-02)	40
5.4	Dataset n°3 – STL 10	41
5.5	Dataset n°4 – Fashion -MNIST	41
5.6	Dataset n°5 – EMNIST	42
5.7	L'exploitation de LOSS du générateur et discriminateur.....	42
5.7.1	Dataset n°1	43
6	Conclusion	49
7	Bibliographie	50

Remerciements

Je tiens à remercier toutes les personnes qui m'ont soutenu pendant mon cursus scolaire et aussi pendant la rédaction de ce travail de fin d'études.

Je voudrais remercier Monsieur Thierry DUTOIT, mon promoteur et professeur à l'UMONS ainsi que Monsieur Jean-Benoit Delbrouck, mon co-promoteur qui m'ont assisté, aidé et proposé le thème de ce travail.

Je tiens également à témoigner ma reconnaissance envers tous les professeurs et assistants qui m'ont enseigné et partagé leurs connaissances pendant ce cycle de deux années à l'Université de MONS (Site de Charleroi).

2 Introduction

2.1 Objectif demandé.

L'objectif de ce travail de fin d'études est la compréhension et l'utilisation des réseaux antagonistes génératifs conditionnels.

Il a été demandé de tester plusieurs sources de données et ensuite de pouvoir générer à la demande, après entraînement, une classe définie.

Le code doit aussi pouvoir être repris et poursuivi par la suite, un outil doit donc être mis en place et doit être documenter.

2.2 Les outils informatiques pour le développement.

2.2.1 L'environnement de développement « Visual Studio code.

Il s'agit d'un éditeur multiplateforme. Le choix s'est porté sur cet outil car sa prise en main est simple et c'est un outil est très complet. Il facilite le codage avec la coloration syntaxique. Il intègre un outil Git ; en 2 clics la mise à jour sur le Github peut être effectuée.

Plus d'informations disponibles sur : <https://code.visualstudio.com/>

2.2.2 Le langage de programmation Python.

C'est un langage de programmation interprété. Celui-ci a été choisi car il se prête très bien au calcul matriciel, vectoriel, et à la manipulation d'images, etc...

De plus, il existe de nombreuses library spécialisées pour le calcul matriciel, les plus connue étant NumPy et SciPy. Ce langage possède une grande communauté, et donc les informations sont faciles à trouver.

Ce langage est, pour moi, un nouvel apprentissage et a donc nécessité une prise en main.

2.2.3 L'hébergement et le versioning du code.

J'ai choisi Github car il propose un service gratuit qui satisfait à ma demande. De plus, des plug-ins permettent de pousser le code et les documents directement à partir de Visual Code Studio.

Plus d'information disponibles sur : <https://github.com/>

Sur ce site, l'ensemble des codes python est hébergé ainsi que les fichiers ipyn (Google Colab – Ipython Notebook).

2.2.4 L'environnement Google Colaboratory.

Cette plateforme propose un service gratuit mais limité dans le temps pour le Machine Learning.

Etant donné que l'apprentissage demande beaucoup de ressource, Google met à disposition, une machine avec les caractéristiques suivantes :

- 2 x Intel(R) Xeon(R) CPU @ 2.30GHz
- 12 Gb de Ram
- 50 Gb de disque dur.

Néanmoins, le plus important est leur GPU qui permet les calculs. Il propose une carte GPU Tesla K80 avec les performances suivantes :

```
!nvidia-smi

Sun Aug 11 15:07:49 2019

+-----+
| NVIDIA-SMI 418.67      Driver Version: 410.79      CUDA Version: 10.0      |
+-----+-----+
| GPU  Name            Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
+-----+-----+
| 0   Tesla K80         Off          | 00000000:00:04:0 | Off              |
| N/A   29C    P8      27W / 149W |      0MiB / 11441MiB |      0%      Default |
+-----+-----+

+-----+
| Processes:                                                       GPU Memory |
|  GPU       PID    Type    Process name                     Usage    |
+-----+-----+
| No running processes found                                     |
+-----+
```

- ✓ 4992 cœurs CUDA avec conception bi-GPU
- ✓ Jusqu'à 2,91 TFlops de performances en double précision avec NVIDIA GPU Boost
- ✓ Jusqu'à 8,73 TFlops de performances en simple précision avec NVIDIA GPU Boost
- ✓ 24 Go de mémoire GDDR5
- ✓ 480 Go/s de bande passante globale

(source <https://www.nvidia.com/fr-fr/data-center/tesla-k80/>)

Cet environnement supporte python 2 ou 3 et permet les commandes Linux en natif par l'ajout d'un ! devant la commande.

2.2.5 L'espace de stockage.

Comme l'application nécessite un grand volume de stockage, Google Drive a été choisi avec un espace de 2 Tb. Certains modèles fournis par l'apprentissage produisent des fichiers de modèle de 111 Mb (Dataset 2).

Il est possible de monter un gdrive sur Colab afin de sauvegarder les différents fichiers. Cela permet d'exploiter les données par la suite.

2.2.6 Les datasets utilisés.

Comme jeu de données, j'ai utilisé un panel de 5 dataset :

- 2 datasets en noir et blanc (MNIST et EMNIST) ;
- 1 dataset en niveau de gris (Fashion-MNIST)
- 3 datasets couleurs (Cifar 10, Cifar 100 et STL-10)

2.2.6.1 Dataset n°0 - MNIST

Ce dataset est une base de données de chiffres manuscrits qui est devenu un standard. Il comprend 60.000 images d'apprentissage et 10.000 de tests.

Les images sont en noir et blanc et normalisées centrées, d'une taille de 28px x 28px. Disponible au téléchargement sur : <http://yann.lecun.com/exdb/mnist/>

Exemple d'images :



Il comprend 10 classes reprenant les 10 chiffres de 0 à 9.

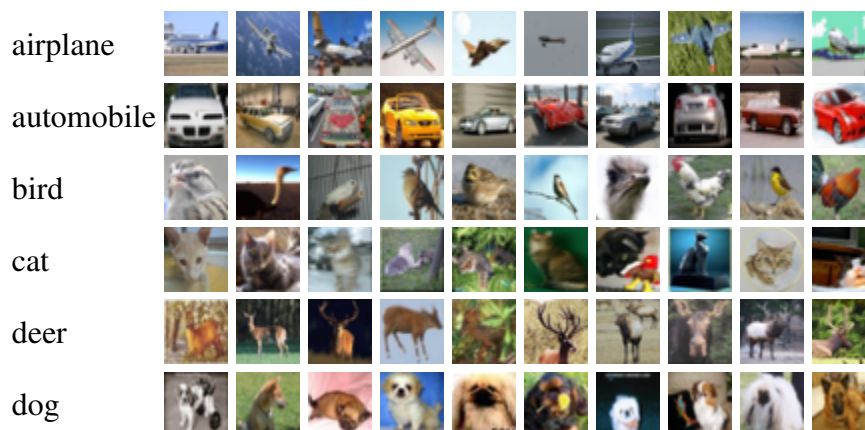
(MNIST : Mixed National Institute of Standards and Technology)

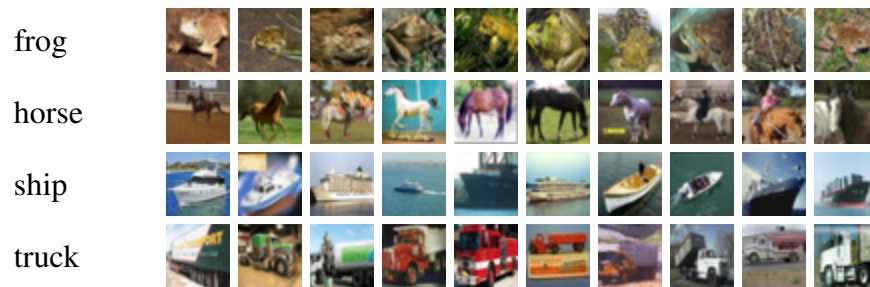
2.2.6.2 Dataset n°1 – Cifar 10

Ce dataset comprend 60.000 images (50.000 Training/10.000 Test). Les images sont de taille 32x32 et composées de 10 classes avec 6.000 images chacune.

Les classes reprises sont : airplane, automobile, bird, cat, deer, dog, frog, horse, ship et truck. Ce sont des images couleur (RGB)

Exemple du dataset :





Disponible au téléchargement sur : <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz>

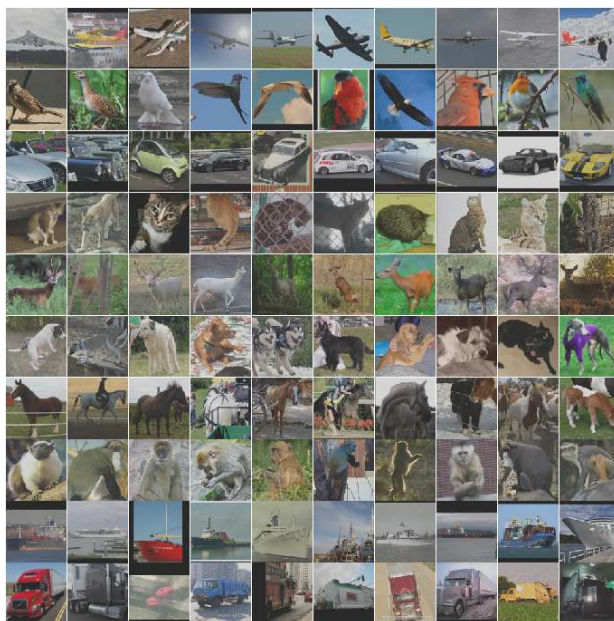
2.2.6.3 Dataset n°2 – Cifar 100

Le dataset est semblable au Cifar 10 mais il est composé de 100 classes.

2.2.6.4 Dataset n°3 – STL 10

Ce dataset est inspiré par Cifar 10 mais les images ont une résolution de 96x96.

Il n'est pas très populaire car il possède peu d'images (+- 5.000) et donc trop petit pour un bon apprentissage.



2.2.6.5 Dataset n°4 – Fashion MNIST

Ce dataset comporte 10 classes représentant chacune des types de vêtements.

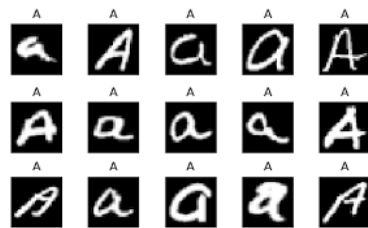


Site : <https://cs.stanford.edu/~acoates/stl10/>

2.2.6.6 Dataset n°5 – EMNIST

Ce dataset est une extension de MNIST. Il peut être splitter (by class, letters, digits, mnist, ...) et peut compter jusqu'à 62 classes. (10 pour les chiffres et 2 x 26 pour les lettres minuscules et majuscules)

Exemple pour la lettre « a ».



Plus d'informations disponibles sur <https://www.nist.gov/node/1298471/emnist-dataset>

2.3 Les neurones et le perceptron

Ils se basent sur la biologie du cerveau humain. Il s'agit d'une modélisation du fonctionnement du cerveau et particulièrement de ses unités de traitement d'information, appelées neurones (Figure 1 - Neurones).

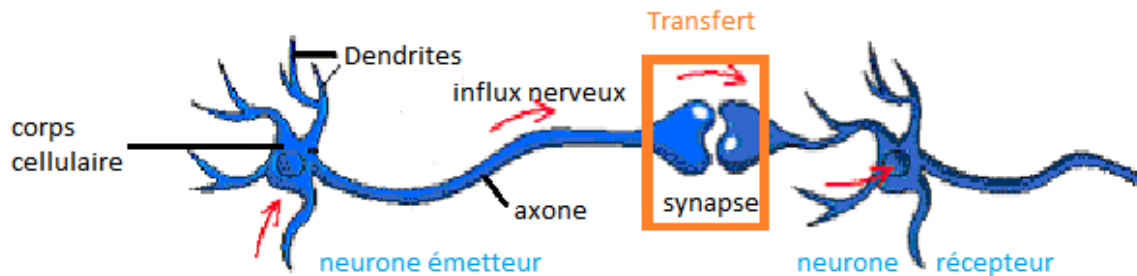


FIGURE 1 - NEURONES

Image source : <https://www.supinfo.com/articles/single/7923-deep-learning-fonctions-activation>
<https://fr.wikipedia.org/wiki/Neurone>

Ce sont des approximateurs de fonctions universelles. Leur entrée peut avoir différents formats comme un vecteur ou une matrice. Ces réseaux sont l'interconnexion de l'élément unitaire que l'on appelle Perceptron (Figure 2 - Perceptron).

Tous les liens sont pondérés avec des biais et des poids W_i (Weights) afin de donner l'intensité de la connexion. Chaque neurone a une seule sortie pour les connexions multiples, c'est la même valeur de sortie qui est propagée.

Le perceptron est le réseau de neurones le plus simple. Il ne contient aucun cycle. Il a n entrée (X_1, X_2, \dots, X_n) et une seule couche de sortie qui est binaire.

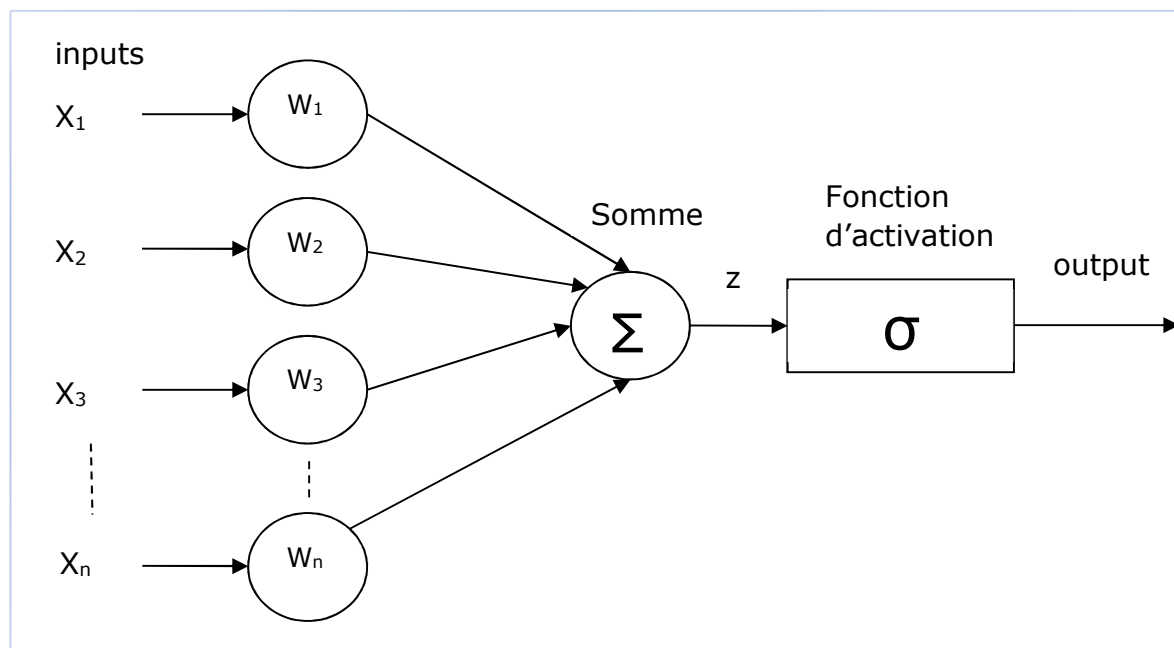


FIGURE 2 - PERCEPTRON

La sortie $z = \sum_{i=1}^n w_i x_i + b$ avec $f(z) = \begin{cases} 0 & \text{si } z \leq 0 \\ 1 & \text{si } z > 0 \end{cases}$

La sortie est donc la somme pondérée des inputs.

On peut aussi leur appliquer un biais, c'est-à-dire une fonction d'activation, un seuil.

Les fonctions d'activation peuvent être linéaires ou non.

Il est aussi possible de réaliser des portes logiques avec le perceptron.

Voici un exemple pour la fonction « AND »

Symbole

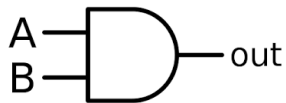


Table de vérité

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Quelle sera le poids de x_1 et x_2 ainsi que le biais

Pour la première rangée, on a $Out = x_1 \cdot w_1 + x_2 \cdot w_2 + b$

Out doit être égal à 0 avec $x_1 = 0$ et $x_2 = 0$

On choisit arbitrairement : $w_1 = w_2 = 1$ et $b = -1$

$x_1 (1) + x_2 (1) - 1 = 0 + 0 - 1 = -1$ comme $Wx + b < 0$ alors $Out = 0$.

Pour 2^{ème} rangé,

Out doit être égal à 0 avec $x_1 = 0$ et $x_2 = 1$

$x_1 (1) + x_2 (1) - 1 = 0 + 1 - 1 = 0$ comme $Wx + b \geq 0$ alors Out passe à 1.

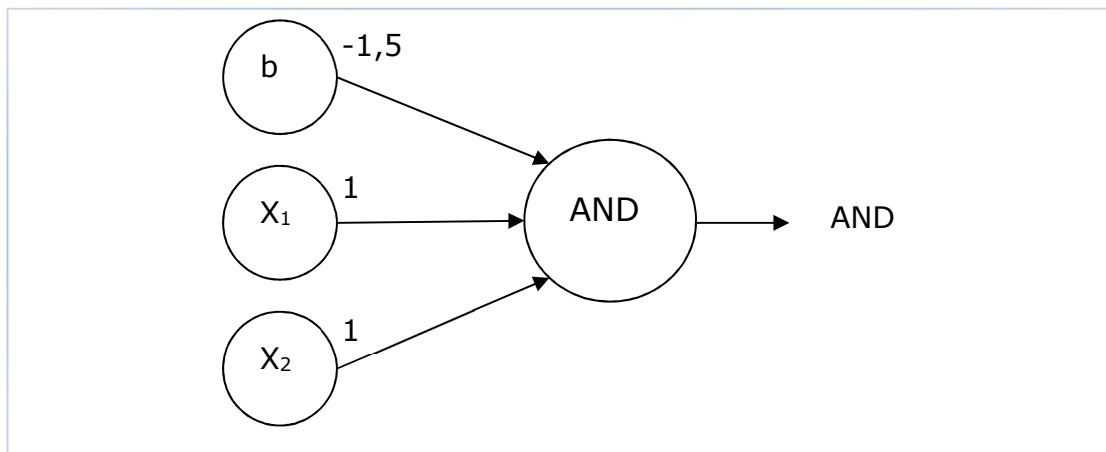
On a donc une mauvaise combinaison, il faut donc réduire b à -1,5 et l'équation devient $0 + 1 - 1,5 = -0,5 \Rightarrow Out = 0$

La 3^{ème} rangée prend ses mêmes valeurs

Pour la 4^{ème} rangée,

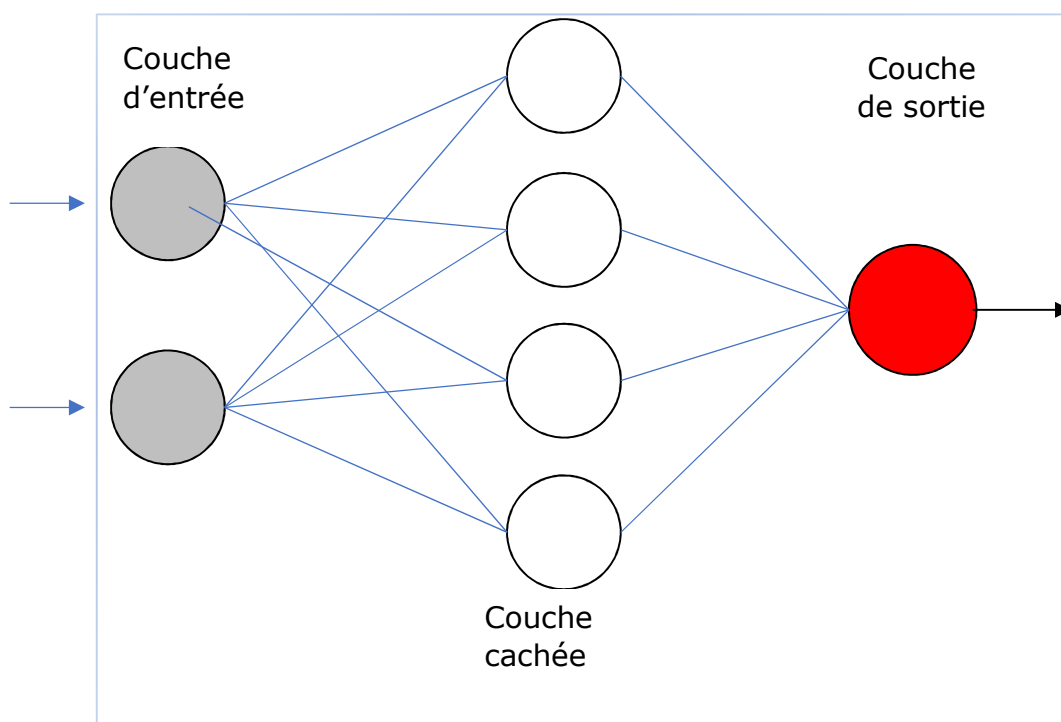
$x_1 (1) + x_2 (1) - 1,5 = 1 + 1 - 1,5 = 0,5$, Out est bien égal à 1

Le perceptron « fonction AND » se présente comme suit :



Le même principe peut être appliqué pour représenter les autres fonctions telles que OR, NOT, XOR, XNOR.

2.4 Les réseaux de neurones.



Les réseaux de neurones sont composés d'un graphe plus ou moins complexe de neurones formels. Il peut y avoir des boucles de rétroaction.

Le réseau peut être feedforward, c'est-à-dire unidirectionnelle, les signaux vont uniquement de l'entrée vers la sortie.

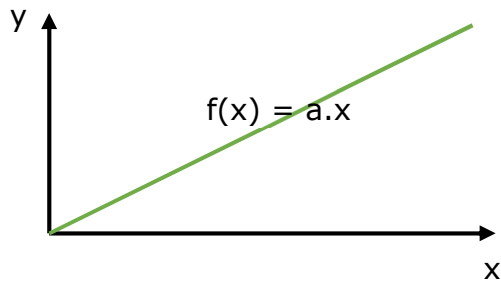
On peut aussi avoir des signaux qui bouclent, on parle alors de réseau récurrent (RNN).

La densité des liaisons est aussi une information. Si les neurones sont totalement interconnectés, on parle de couche « fully connected ».

2.5 Fonction d'activation

Il existe différentes façons pour un neurone de s'activer, on applique alors ce qu'on appelle les fonctions d'activation qui peuvent prendre différentes formes.

2.5.1 Fonction linéaire.



La sortie est donc proportionnelle à l'entrée. C'est un modèle de régression linéaire.

Pour aller plus loin que ces simples régressions, on introduit des fonctions non linéaires.

2.5.2 Fonction non linéaire.

Ces fonctions sont plus utilisées car elle impacte beaucoup plus le résultat pour résoudre des problèmes complexes.

2.5.2.1.1 La sigmoïde

Le but de cette fonction (Figure 3 - Sigmoid) d'avoir une valeur qui est comprise entre 0 et 1 et d'exprimer les valeurs sous formes de probabilité. Si l'entrée est très grande, on aura une valeur de 1 en sortie et une valeur très négative prendra la valeur 0.

Elle a quelques défauts :

- Comme elle n'est pas centrée sur « 0 », certaines entrées négatives peuvent fournir des sorties positives.
- Elle est assez plate, le résultat est donc souvent soit proche de 0 ou de 1.
- Elle est d'ordre élevé car c'est une opération exponentielle.

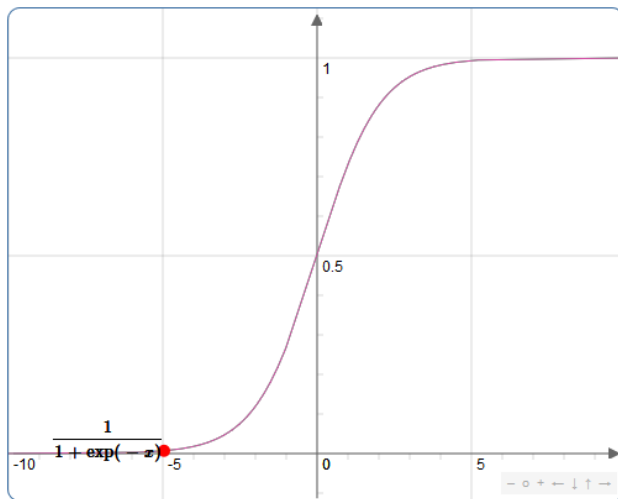
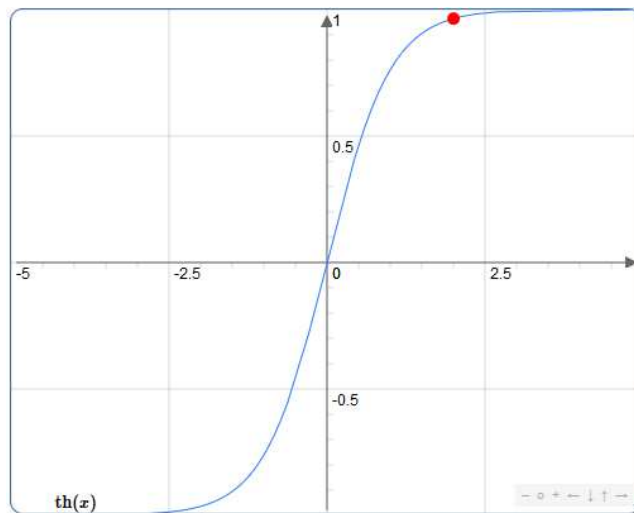


FIGURE 3 - SIGMOÏDE

2.5.2.1.2 La Tanh

Elle ressemble fortement à la sigmoïde mais sa sortie peut varier entre -1 et 1.

Elle est centrée sur 0 et elle est privilégiée par rapport à la sigmoïde. Les grands entrées positives donneront 1 et les négatives donneront -1. Elle a les mêmes inconvénients que la sigmoïde.



2.5.2.1.3 La ReLU

Cette fonction (Figure 4 - ReLU) est modélisée par $f(x) = \max(0, x)$. Elle permet d'augmenter la convergence du réseau sans saturation.

Si l'entrée est négative, la sortie sera 0. Le neurone ne s'activera pas, son poids ne sera pas mis à jour et donc le réseau n'apprendra pas. Si elle est positive, la sortie sera proportionnelle à l'entrée.

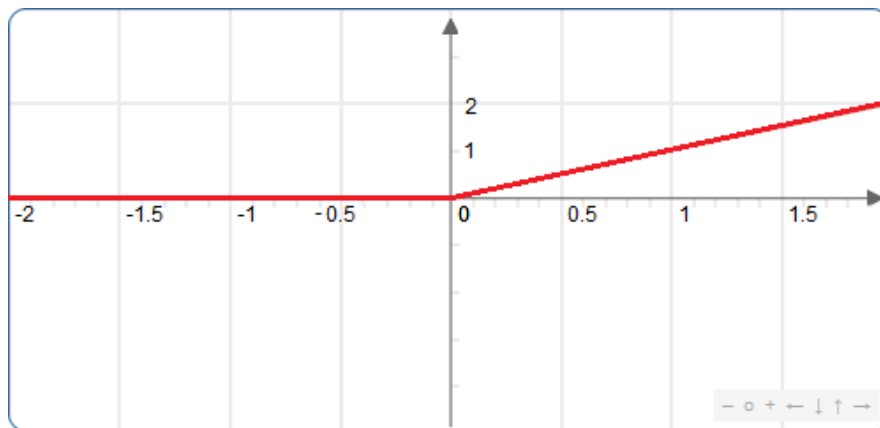
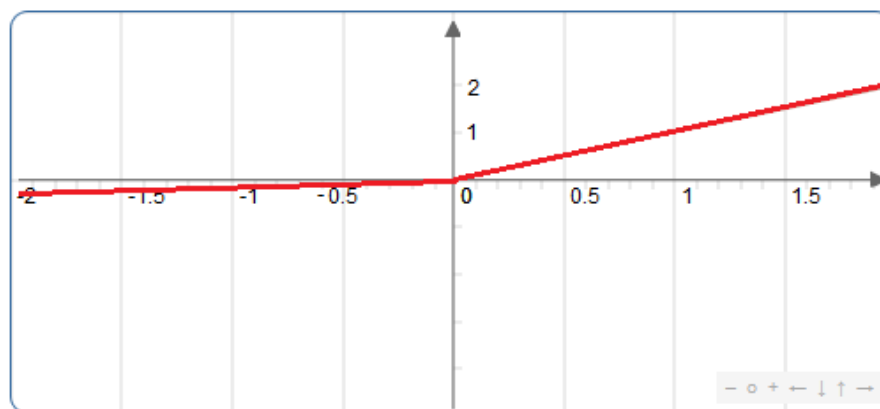


FIGURE 4 - RELU

2.5.2.1.4 Leaky ReLU

$$f(x) = \max(0.1x, x)$$

Avec cette fonction, on essaie de corriger la fonction d'activation ReLU pour les entrées négatives. Le réseau est fonctionnel pour les petites valeurs négatives.



2.6 Apprentissage.

L'apprentissage est la période pendant laquelle le réseau se construit jusqu'à obtenir le comportement souhaité.

Il existe deux types d'apprentissage :

- L'apprentissage supervisé.
- L'apprentissage non supervisé.

Pendant l'entraînement, les poids sont adaptés en permanence afin de minimiser l'erreur. L'algorithme produit utilise des modèles qu'il sauvegarde périodiquement à chaque fin d'Epoch (format de fichier *.pth).

2.6.1 L'apprentissage supervisé

On parle d'apprentissage supervisé lorsque les données d'entrées sont annotées par des labels représentant des classes. Le réseau sait ainsi à quelle classe appartient l'entrée et ce qu'il doit donc sortir.

Il est souvent utilisé pour faire de la reconnaissance d'image, de la segmentation ou encore des prédictions.

Exemple : une image x « cat.png » est associé à un label y « cat »



2.6.2 L'apprentissage non supervisé

Cet apprentissage se base sur des jeux de données non labellisés. L'algorithme ne peut donner par exemple un score d'adéquation au résultat. L'algorithme va lui-même trouver des différences entre les objets. Il va lui-même créer ses classes, ses features. L'extraction des données est donc descriptive.

2.6.3 Introduction au Epoch – Batch size.

L'Epoch est l'unité de mesure d'un jeu de données. Lors de l'entraînement, on dit que lorsque tout le jeu de données a été passé dans le réseau, il s'agit d'un Epoch. Un grand nombre d'Epoch est nécessaire pour un bon apprentissage.

Il ne faut confondre batch et batch size. Le Batch size correspond au nombre d'images comprises dans un lot d'images. Il est nécessaire de diviser le set de training car il n'est pas concevable de fournir l'ensemble des données en une seule fois au réseau.

Par exemple pour un dataset de 60.000 images, on choisit un Batch size de 16, on aura donc 3.750 lots de 16 images par Epoch.

2.7 Les réseaux antagonistes génératifs (GAN).

Ian Goodfellow est le créateur de la technologie GAN, cette architecture de réseaux de neurones est conçue spécialement pour imiter des données (images, sons, vidéos, ...).

Le GAN est composé de deux réseaux profonds, le générateur et le discriminateur.

- Un générateur G tente de produire des données au plus proche du dataset.
- Un discriminateur D estime la probabilité qu'une donnée soit vraie (issue du dataset) ou fausse (générée).

Le principe est d'introduire un adversaire appelé « discriminateur » en relation avec G. Ce dernier crée de fausses images et D lui revoit son feedback sur ce qu'il observe. L'antagoniste n'a pas vraiment un adversaire, son rôle est plutôt d'enseigner ce qu'il fait bien ou ce qu'il fait mal.

Cet adversaire doit s'entraîner sur des images réelles pour donner un avis concret. Ce réseau se base sur le principe de rétropropagation, le discriminateur donne une note au générateur sur la fausse image quant à la ressemblance avec une réelle. Il remonte les gradients calculés au générateur qui modifie par la suite ses paramètres.

2.7.1 Principe de la Rétropropagation.

Le discriminateur D émet une valeur $D(x)$ qui indique la probabilité que x soit une image réelle. L'objectif est de maximiser la probabilité de savoir qu'une image est vraie pour une image réelle et que les images générées sont fausses. Pour cela, nous utilisons la cross-entropy.

Pour un vrai label d'image réelle, P est égale à 1.

L'objectif pour le discriminateur est donc (1) :

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

z : entrée

p_{data} : exemples d'apprentissage du dataset

x : image issue du dataset

L'objectif de G est qu'il génère des images avec la plus grande valeur possible de $D(x)$ pour duper le discriminateur.

L'équation du générateur G est (2) :

$$\min_G V(G) = \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

Le Gan est souvent défini comme un jeu minimax dans lequel le générateur veut minimiser V alors que le discriminateur veut le maximiser.

En fusionnant (1) et (2), on obtient :

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

On cherche ensuite à optimiser cette dernière équation.

Au fur et mesure que le générateur s'améliore, le discriminateur doit lui aussi réapprendre.

Le GAN alterne deux phases. Une phase où l'adversaire apprend à distinguer les images réelles des fausses du générateur et une seconde où le générateur améliore sa performance grâce aux feedbacks. Il peut ainsi faire des images de plus en plus indiscernables des réelles.

Les applications sont déjà nombreuses. Elles sont capables d'augmenter la résolution d'images très pixelisées, de retoucher des images en reconstituant des parties de l'image supprimée ou encore en simulant des conditions climatiques sur une image.

Récemment des fausses vidéos issues des GANs ont aminé des visages de célébrités pour leur faire dire n'importe quoi.

Les GANs font partie des réseaux non supervisés, on leur donne juste des données brutes non étiquetées.

En résumé, l'objectif du GAN est de générer une image x à partir d'une entrée z qui suit une distribution prédéfinie, et ce quel que soit le z choisi parmi la distribution :
 $\tilde{x} = G(z), z \sim P(z)$

2.8 Les réseaux antagonistes génératifs conditionnels.

Les CGAN utilisent le même principe que le GAN mais on ajoute une information supplémentaire au réseau, le label y .

On définit le générateur conditionnel $cG(z, y)$ qui génère une image à partir d'un vecteur aléatoire z et d'un vecteur de label y associé à l'image $x^* \in Data$.

L'équation devient : $\tilde{x} = cG(z, y), z \sim P(z), y = label(x^*)$

Le discriminateur conditionnel $cD(x, y)$ qui prédit si l'image x de label y est une image réelle x^* ou une image générée \tilde{x}

On a l'équation $cD(x, y) \in [0, 1]$, idéalement, $\begin{cases} D(\tilde{x}, y) = 0, \tilde{x} = G(z, y) \\ D(x^*, y) = 0, x^* \in Data \end{cases}$

L'équation du jeu minimax devient pour le CGAN :

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

2.8.1 Random noise z

Ce bruit a une distribution gaussienne (Figure 5 - Bruit blanc Gaussien).

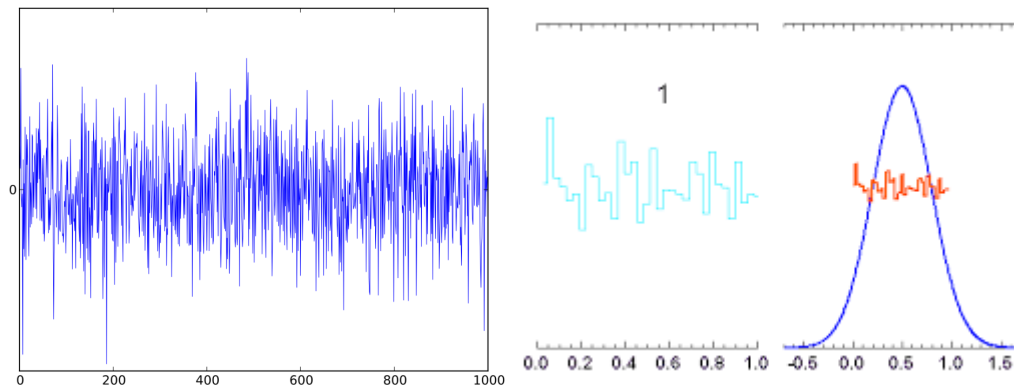
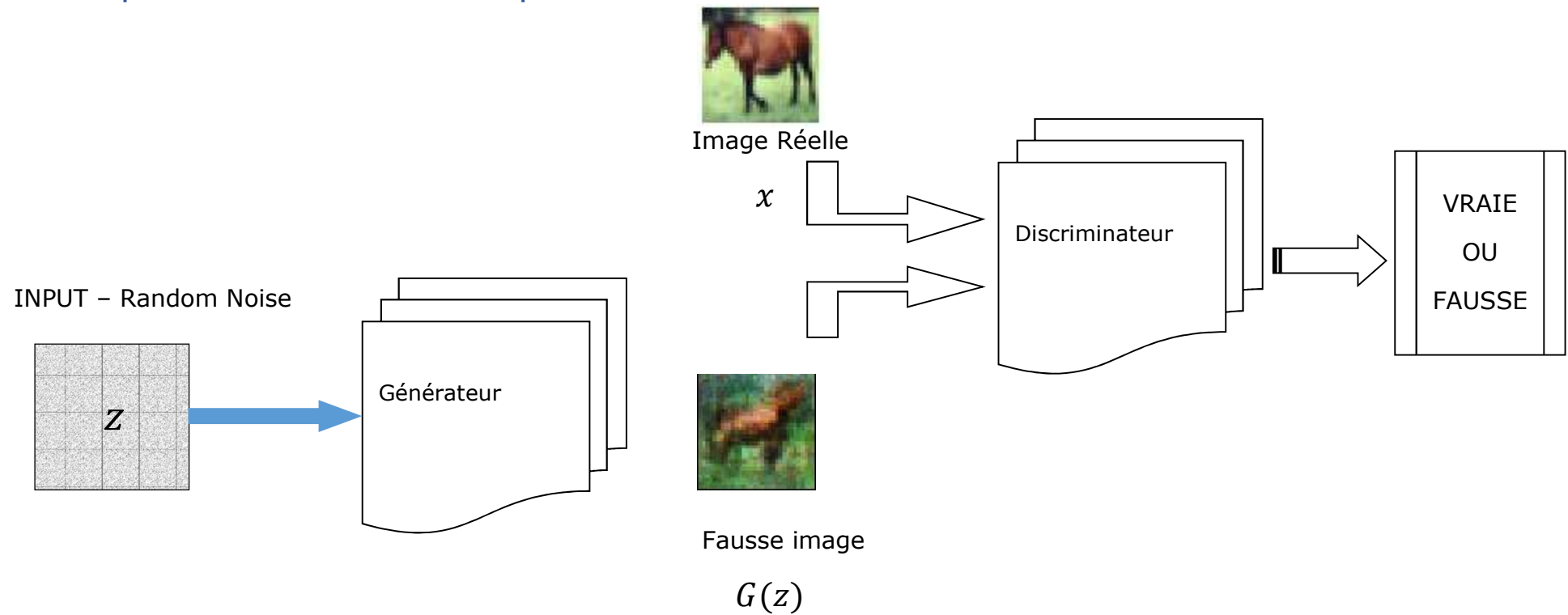


FIGURE 5 - BRUIT BLANC GAUSSIEN

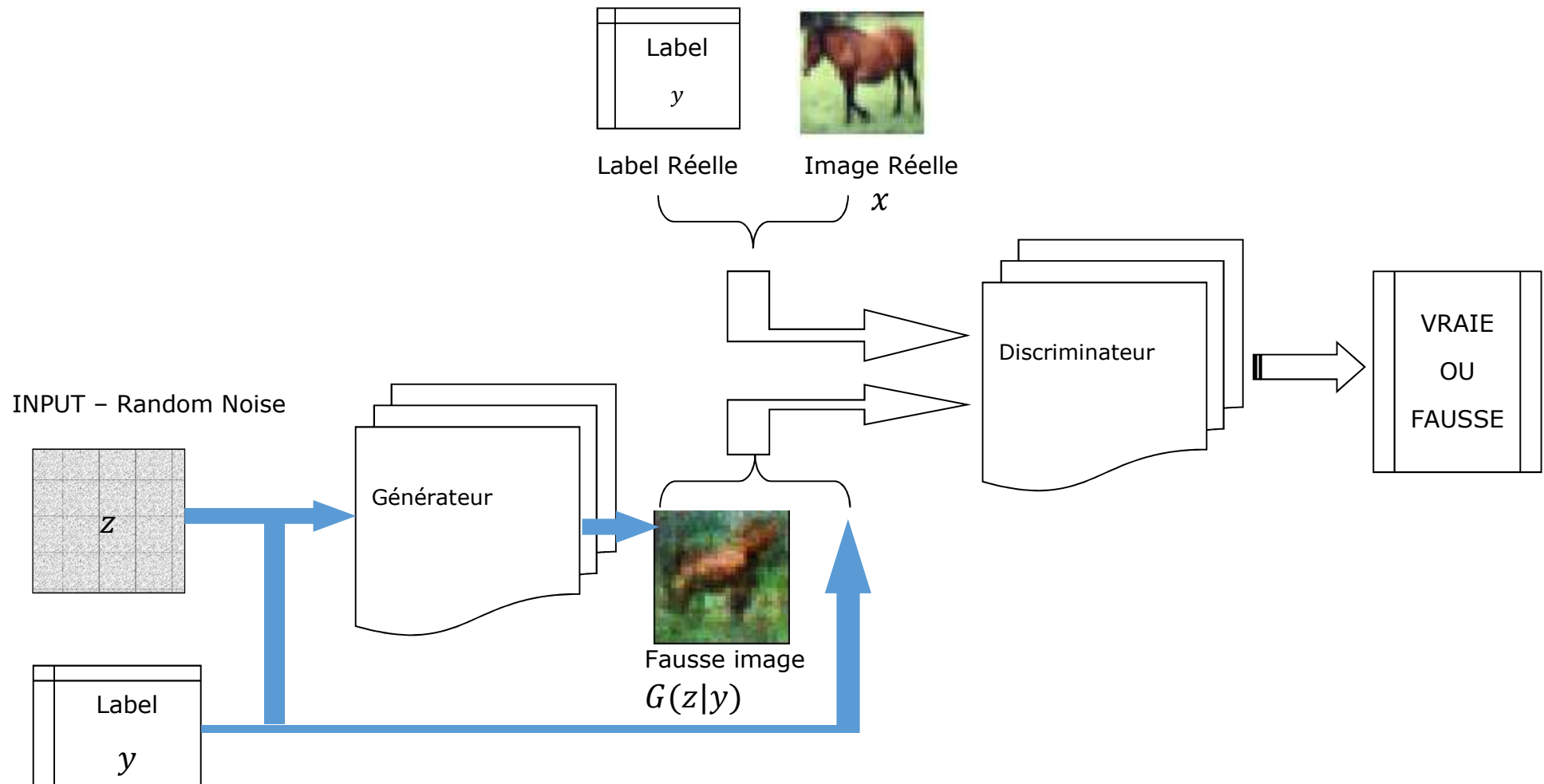
Ce bruit aléatoire est nécessaire car il est bénéfique aux réseaux. Il permet de ne pas biaiser le réseau d'entrée afin que le réseau apprenne plus efficacement

Ce bruit permet d'avoir une entrée aléatoire structurée. Grâce à sa structure imposée, on peut prendre un point dans l'espace de la distribution et on est sûr d'avoir un résultat qui ressemble à quelque chose. Ça permet aussi d'avoir un générateur non déterministe, vu que les points dans la distribution sont quasi infinis on est certain que G aura toujours un input différent et donc une sortie tjrs différente.

2.9 Représentation schématique des GAN



2.10 Représentation schématique des CGAN



3 Les différents algorithmes produits

3.1 L'apprentissage (cgan_train.py)

Le code peut prendre plusieurs paramètres pour configurer l'entraînement en une seule commande. La version Google Colab propose un menu pour la configuration.

3.1.1 La mise en place

3.1.2 L'installation

Il faut premièrement sélectionner le dataset

L'utilisation d'un parser permet cette action.

Les différents choix sont :

- Le nombre d'Epochs ;
- La Batch size ;
- Le nombre de classes ;
- La taille de l'image à traiter ;
- La période de sauvegarde des images générées ;
- Le choix du dataset ;
- L'espace latent ;
- Le nombre canaux (RGB, N/B, Niveau de gris) ;
- Paramètre de l'optimisateur ADAM (learning rate, β_1 et β_2)

Les images (img_shape) sont représentées comme une matrice en 3 dimensions : (C x H x W)

La première dimension est ici, le nombre de canaux. La 2^{ème} et 3^{ème} reprennent les coordonnées en x,y.

Cuda (Compute Unified Device Architecture – développé par NVIDIA) est un utilitaire qui permet d'exécuter le code sur le GPU et non pas sur le CPU. Les GPU permettent, grâce à leur architecture, d'effectuer un plus grand nombre de calculs que les processeurs traditionnels et surtout sur des vecteurs. Le GPU Tesla K80 de Google Colab permet jusqu'à 8.73 TFlops (floating-point operations per second).

L'outil Torch est utilisé. C'est un logiciel Open-source spécialisé pour le calcul scientifique. Il est compatible avec Cuda et offre un large panel de packages comme le traitement d'image, vidéo, vision par ordinateur et surtout l'apprentissage automatique.

Il intègre des réseaux de neurones et des bibliothèques d'optimisation.

L'algorithme de ce présent travail engendre 3 grandes familles de données tout au long de l'apprentissage :

- 1) Les modèles ;
- 2) Les images générées ;
- 3) Les données des « loss » ;

Un fichier de configuration est aussi généré afin de connaître les paramètres utilisés lors du « training »

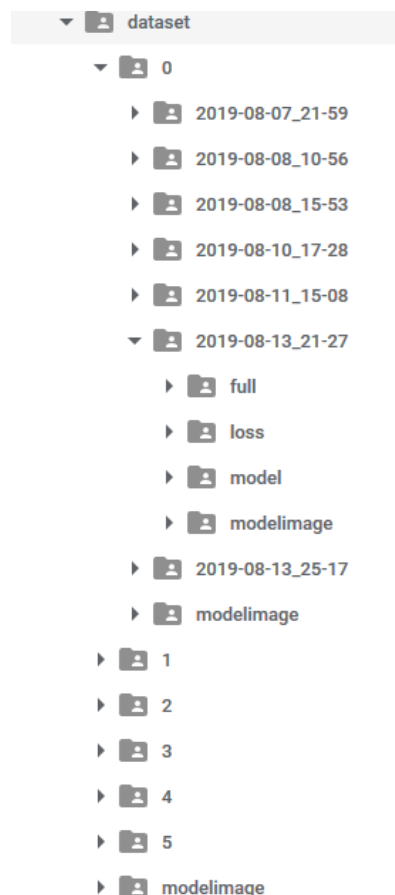
Exemple du fichier config.txt

```
Namespace(b1=0.9, b2=0.999, batch_size=64, channels=1, dataset=0,
img_size=32, latent_dim=100, lr=0.0004, n_classes=10, n_cpu=8,
n_epochs=100, sample_interval=938)
```

On conserve ici les différents paramètres des apprentissages.

Représentation de l'arborescence de travaux

Le code crée automatiquement différents dossiers pour son espace de travail dans le gdrive. Elle présente comme ci-dessous par date de création sur le drive.



3.1.3 L'espace latent.

Il représente l'espace des caractéristiques (features) que le modèle peut extraire.

Les features sont des zones d'intérêt d'une image. Elles peuvent représenter un contour, des points, lignes, ...

3.1.4 La « class Generator(nn.Module) »

Voici la définition de la classe.

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.label_emb = nn.Embedding(opt.n_classes, opt.n_classes)

        def block(in_feat, out_feat, normalize=True):
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat, 0.8))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *block(opt.latent_dim + opt.n_classes, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )
```

FIGURE 6 - CLASS GENERATEUR

Dans cette classe, les différentes couches formant le réseau sont définies.

Les couches créées se composent de blocs :

- linéaire appliquant une transformation linéaire aux données input : $y = x \cdot A^T + b$
- un bloc de normalisation peut être ajouté. Il permet d'initier le réseau grâce à une distribution gaussienne forçant les activations.

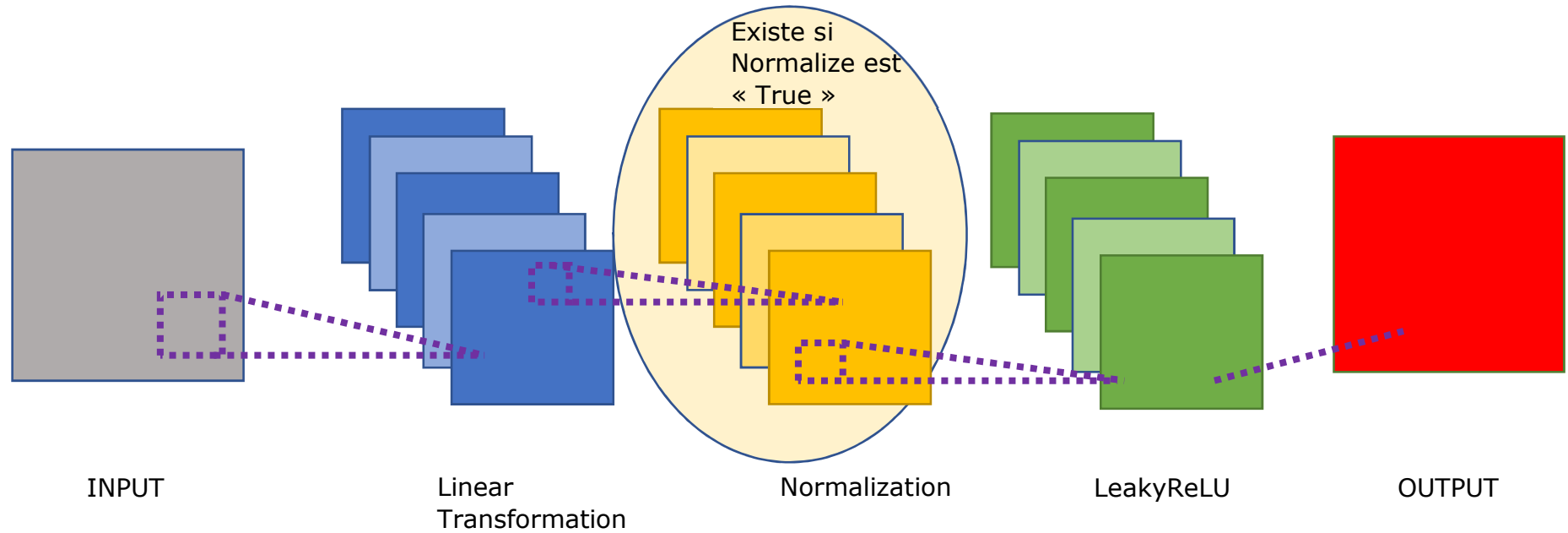
Ensuite le bloc reçoit une fonction d'activation LeakyReLU :

$$\text{LeakyReLU}(x) = \max(0, x) + \text{pente négative} * \min(0, x)$$

$$\text{Ou } \text{LeakyReLU}(x) = \begin{cases} x & \text{si } x \geq 0 \\ \text{pente} * x & \text{si } x < 0 \end{cases}$$

Le modèle se compose de ces 4 blocs suivi d'un autre bloc de transformation linéaire. La fonction d'activation finale est la tangente hyperbolique.

Bloc initial du générateur



Couches du générateur

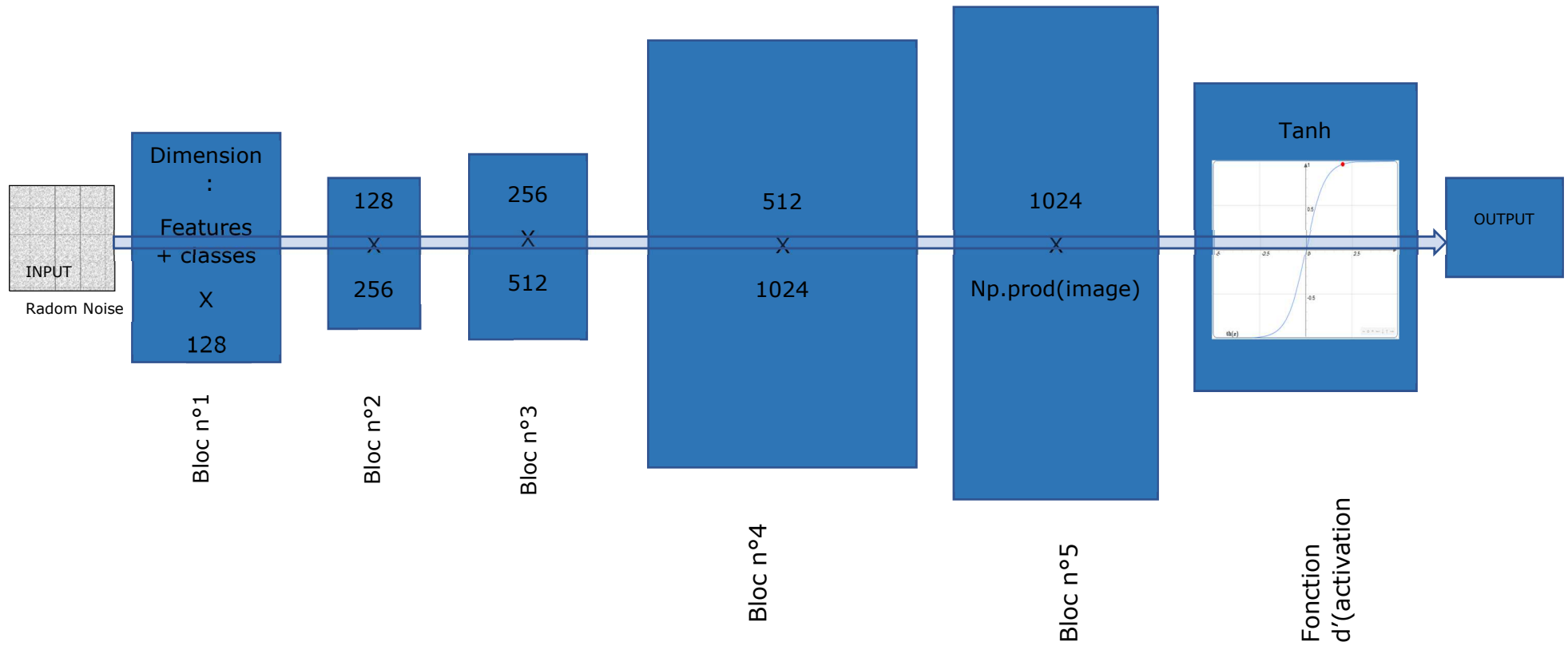
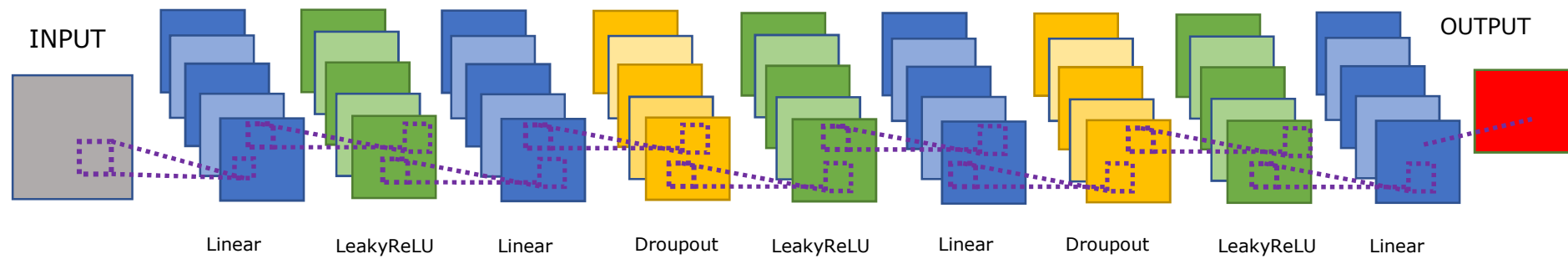


FIGURE 7 - SCHÉMA RÉSEAU GÉNÉRATEUR

Couches du discriminateur



3.1.5 Les datasets

L'utilitaire Torch propose des méthodes qui permettent de télécharger automatiquement le dataset et d'effectuer des compositions de transformations.

Parmi ces transformations, il est possible de :

- Redimensionner l'image
- Transformer les images en tenseurs (C x H x W)
- Normaliser les tenseurs

Cette méthode normalise un tenseur représentant l'image avec deux paramètres et ce indépendamment pour chaque canal (RGB) :

- Moyenne
- Ecart type

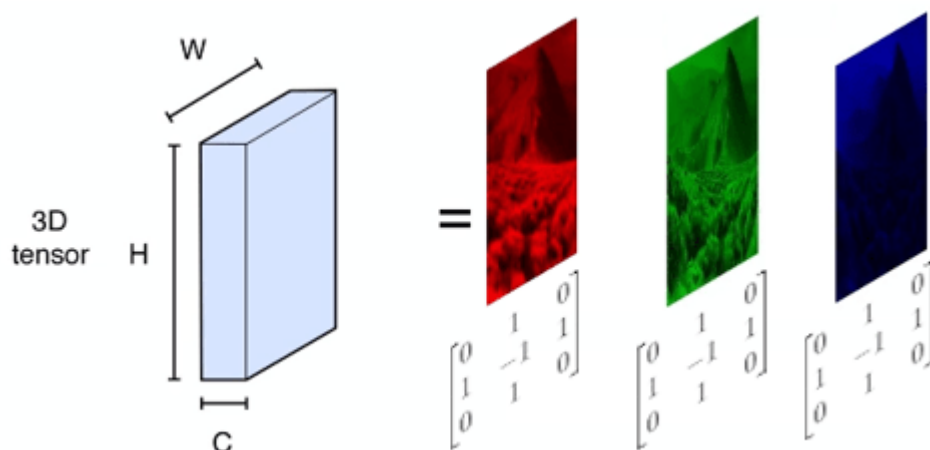
3.1.6 Les tenseurs

Un tenseur est une table de données multi-dimensionnelles. On pourrait penser qu'un tenseur à 2 dimensions peut être représenté comme une matrice et un tenseur à 1 dimension serait un vecteur. Ce n'est pas tout à fait le cas.

Pour des matrices, on sait que l'on peut effectuer des opérations mathématiques comme l'addition, la soustraction ou la multiplication mais il faut que leur taille soit compatible. Une matrice représente un simple tableau de nombres.

En revanche, un tenseur s'exprime dans un espace vectoriel. Un tenseur peut être représenté par un tableau multidimensionnel organisé de valeurs numériques par rapport à une base spécifique. Le changement de base transforme les valeurs du tableau en une caractéristique qui permet de définir les tenseurs en tant qu'objets adhérant à ce comportement transformationnel.

Ils sont utilisés pour représenter une image car notre réseau de neurones ne peut pas prendre d'image au format png, bmp ou autre.



Source : <https://deeplylearning.fr/cours-pratiques-deep-learning/composition-et-conversion-dune-image/>

Une image est composée de 3 composants RGB qui représente les canaux. Un tenseur est donc la représentation de l'image dans l'espace vectoriel.

Les couleurs d'une image PIL (Python Imaging Library) se distingue par des valeurs allant de 0 à 255. Le tenseur normalise ses valeurs entre 0 et 1.

3.1.7 Les données et les blocs

Les différentes données (dataset) ne doivent pas être redimensionnées ou normalisées. Le réseau accepte des tailles différentes mais elles doivent être passées en paramètre à l'algorithme afin qu'il crée ses couches. Voir les blocs 1 et 5 de l'image (Figure 7 - schéma réseau générateur) que prennent les dimensions de paramètres.

Exemple :

1) Pour un jeu de données de 28 x 28 pixels, un canal et 62 classes,

Impression du premier bloc généré par l'algorithme

```
0.weight      torch.Size([128, 162])
0.bias        torch.Size([128])
=> Nous avons bien 128 (fixé) et (100 + #classes) soit 162.
```

Impression du dernier bloc (Linear)

```
11.weight     torch.Size([784, 1024])
11.bias       torch.Size([784])
Le bloc 5 « Linear » s'adaptent : 28 x 28 = 784 points sur 1024 (fixé)
```

Pour une image (3, 32, 32) et 100 classes.

```
0.weight      torch.Size([128, 200])
0.bias        torch.Size([128])
=> Nous avons bien 128 (fixé) et (100 + #classes) soit 200.
```

```
11.weight     torch.Size([3072, 1024])
11.bias       torch.Size([3072])
=> Nous avons bien 32x32x3 soit 3072
```

Les paramètres n'interviennent que sur ces 2 couches. Cela est logique car il s'agit des couches d'entrée et de sortie. Le reste du réseau est invariable.

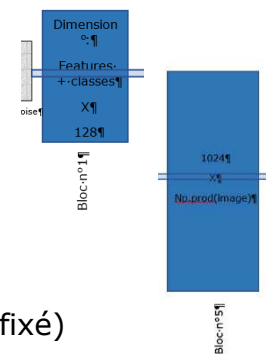
Les différentes données passent par un optimisateur Adam. Il est l'un des plus efficace pour l'optimisation par descente de gradient. Il adapte automatiquement le taux d'apprentissage pour chaque paramètre.

Il calcule des estimations adaptatives des premier et second moment.

$$\forall i, (m_{t+1})_i \leftarrow \beta_1 \cdot (m_t)_i + (1 - \beta_1) \cdot (\nabla J(\theta_t))_i,$$

$$\forall i, (v_{t+1})_i \leftarrow \beta_2 \cdot (v_t)_i + (1 - \beta_2) \cdot ((\nabla J(\theta_t))^2)_i,$$

$$\forall i, (\theta_{t+1})_i \leftarrow (\theta_t)_i - \alpha \frac{\sqrt{1 - \beta_2}}{1 - \beta_1} \cdot \frac{(m_t)_i}{\sqrt{(v_t)_i + \epsilon}},$$



$$\alpha, \epsilon > 0 \text{ et } \beta_1, \beta_2 \in]0,1[$$

m_t est le premier moment du gradient (la moyenne)

v_t est le second (variance non-centrée)

ϵ est un paramètre de précision

β_1 et β_2 sont des paramètres pour réaliser des moyennes d'exécution sur les moments m_t et v_t respectivement. Dans l'algorithme, ils sont notés b1 et b2

3.1.8 Extraction et sauvegarde des modèles en fichier *.pth

A la fin de chaque Epoch, une procédure a été mise en place pour que le modèle soit sauvegardé. Cela se fait via la méthode torch.save().

Il reprend l'ensemble des poids et biais du réseau. Il se représente comme ci-dessous.

```
Model's state_dict :
0.weight      torch.Size([128, 162])
0.bias        torch.Size([128])
2.weight      torch.Size([256, 128])
2.bias        torch.Size([256])
3.weight      torch.Size([256])
3.bias        torch.Size([256])
3.running_mean torch.Size([256])
3.running_var  torch.Size([256])
3.num_batches_tracked torch.Size([1])
5.weight      torch.Size([512, 256])
5.bias        torch.Size([512])
6.weight      torch.Size([512])
6.bias        torch.Size([512])
6.running_mean torch.Size([512])
6.running_var  torch.Size([512])
6.num_batches_tracked torch.Size([1])
8.weight      torch.Size([1024, 512])
8.bias        torch.Size([1024])
9.weight      torch.Size([1024])
9.bias        torch.Size([1024])
9.running_mean torch.Size([1024])
9.running_var  torch.Size([1024])
9.num_batches_tracked torch.Size([1])
11.weight     torch.Size([784, 1024])
11.bias       torch.Size([784])
```

Voici un extrait du contenu des tenseurs à un moment de l'apprentissage. Il représente les 128 biais du premier bloc. Tous les autres blocs ont été extraits.

'model.0.bias', Parameter containing:

```
tensor([-0.0062, -0.0464,  0.0088, -0.0794, -0.0873, -0.0786,  0.0639, -0.0402,
        0.0199,  0.0685, -0.0082, -0.0011,  0.0623, -0.0828, -0.0412, -0.0196,
        0.0759, -0.0928,  0.0717, -0.0305, -0.0370,  0.0769,  0.0591,  0.0793,
       -0.0153,  0.0136, -0.0166,  0.0176, -0.0279, -0.0502,  0.0817, -0.0445,
       -0.0085, -0.0839, -0.0419,  0.0135,  0.0909, -0.0423,  0.0028,  0.0446,
        0.0792,  0.0748, -0.0629,  0.0425,  0.0650,  0.0940,  0.0652, -0.0574,
```



```

0.0748, 0.0146, 0.0755, 0.0821, 0.0668, -0.0235, -0.0560, 0.0075,
0.0744, -0.0799, 0.0615, 0.0195, -0.0627, 0.0095, -0.0328, -0.0526,
0.0099, -0.0333, -0.0425, 0.0616, -0.0479, 0.0218, -0.0127, -0.0537,
0.0288, -0.0131, 0.0234, -0.0844, 0.0759, 0.0450, -0.0327, 0.0148,
-0.0724, -0.0526, -0.0103, -0.0337, 0.0131, 0.0606, -0.0687, 0.0259,
-0.0314, 0.0808, 0.0530, -0.0523, 0.0266, -0.0487, 0.0161, 0.0070,
0.0391, -0.0876, 0.0045, -0.0102, -0.0006, -0.0877, -0.0208, 0.0750,
-0.0321, -0.0663, -0.0442, -0.0390, 0.0657, -0.0357, 0.0565, 0.0284,
0.0162, 0.0096, 0.0416, 0.0831, -0.0558, -0.0861, 0.0384, -0.0407,
-0.0594, 0.0626, 0.0900, 0.0688, 0.0947, -0.0891, -0.0786, -0.0773],
requires_grad=True)),

```

L'ensemble de l'extraction du modèle est disponible sur le github <https://github.com/rombaux/PyTorch-GAN/assets/model.txt>

Ces valeurs sont ensuite utilisées lors de la génération du label défini. A ce moment, il ne faut que quelques secondes pour générer une image pré-entraînée.

Elle représente l'ensemble des poids et biais que le réseau a calculés.

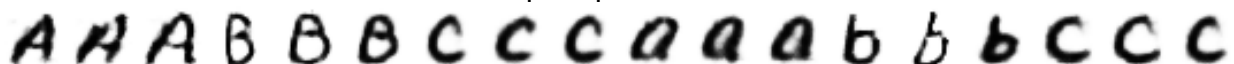
3.2 Le Générateur (cgan_generate.py)

L'algorithme du générateur d'image utilise la même classe « Generator » reprenant la construction des différentes couches du réseau.

Contrairement à l'apprentissage, le générateur d'image peut se faire sur CPU, une version CPU est disponible : cgan_generate_on_cpu.py

Il utilise donc la même méthode pour générer les images, on introduit toujours un random noise en entrée mais on charge le modèle optimal qui a donné des bons résultats. Cela se fait avec la méthode torch.load().

On peut remarquer que le bruit intervient dans la création. Ici deux images produites dont une avec un bruit fixe (non renouvelé dans l'algorithme) et un autre avec un bruit actualisé. On remarque qu'avec le bruit actualisé la sortie donne



Et pour le bruit unique :



Pour ce dernier, on voit que les mêmes images (lettres) sont identiques tandis que pour l'autre, elles diffèrent légèrement.

L'écriture d'un robot avec un bruit actualisé ressemble plus au comportement humain où chaque « frappe » diffère légèrement. On sait écrire un 'A' mais on ne le fera jamais deux fois identiquement au 'pixel' près.

4 L'exécution du code

L'exécution se fait en plusieurs étapes :

- ✓ Montage du gdrive
- ✓ Git Donwload
- ✓ Installation des dépendances comme par exemple (torch, matplot, numpy,...)
- ✓ La mise à jour du Git est disponible, cela évite de charger le git complet en cas de modification et en cours de développement
- ✓ Le bloc « GENERATEUR »
 - Procédure d'apprentissage avec son menu et le print des Loss
 - Le test du modèle, le menu et l'affichage des images

L'installation sera détaillée dans le Github, ici nous verrons uniquement le code « *Procédure d'apprentissage du CGAN* »

4.1 L'apprentissage

La méthode est simple, il faut suivre les instructions à l'écran.

1) Le menu apparait et propose donc le dataset à utiliser.

```
[0] Dataset 0 - MNIST
[1] Dataset 1 - CIFAR 10
[2] Dataset 2 - CIFAR 100
[3] Dataset 3 - STL 10
[4] Dataset 4 - Fashion MNIST
[5] Dataset 5 - EMNIST
Choisissez le dataset à entrainer [0-5]:
```

2) Ensuite, il faut choisir la taille sur laquelle le réseau va apprendre. Il propose la taille par défaut du dataset.

Entrer la taille de l'image (Défaut : 32px x 32px) :

3) Il propose ensuite différents batch size et il informe aussi du nombre de batches qu'il y aura par Epoch.

```
[0]      2          soit 30000 Batches par Epoch
[1]      4          soit 15000 Batches par Epoch
[2]      8          soit 7500 Batches par Epoch
[3]     16          soit 3750 Batches par Epoch
[4]     32          soit 1875 Batches par Epoch
[5]     64          soit 938 Batches par Epoch
[6]    128          soit 469 Batches par Epoch
[7]    256          soit 234 Batches par Epoch
[8]    512          soit 117 Batches par Epoch
[9]   1024          soit 59 Batches par Epoch
[10]  2048          soit 29 Batches par Epoch
```

Choisissez le batch size à tester [0-10] (Défaut = 64) :

- 4) Il proposera ensuite l'intervalle de génération et de sauvegarde locale des images d'apprentissage.

Entrer l'intervalle de génération d'image (Défaut = 938) :

Dans le code, le modèle est, quant à lui, sauvegardé à chaque Epoch.

- 5) Il faut, enfin pour terminer, choisir le nombre d'Epoch

Choisissez le nombre d'Epoch (Défaut = 100) :

- 6) Pour chaque dataset, il va télécharger automatiquement le dataset sur internet. Exemple pour le dataset n°0 :

```
. . . . .
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
../../data/mnist/MNIST/raw/train-images-idx3-ubyte.gz
. . . . .
```

- 7) Voici un extrait du retour de l'apprentissage, on peut suivre en temps réel l'évolution

```
2019-08-16_18-29 [Epoch 5/100] [Batch 0/938] [D loss: 0.072978] [G loss:
0.861564]
Les images pour l'intervalle n° : 0006 et batches_done = 4690
sauvée dans /content/gdrive/My Drive/TFE/dataset/0/2019-08-16_18-27/
Model saved in /content/gdrive/My Drive/TFE/dataset/0/2019-08-16_18-
27/model/model_from_epoch_0005.pth
image redimensionné à 32
2019-08-16_18-30 [Epoch 6/100] [Batch 0/938] [D loss: 0.138594] [G loss:
1.213389]
Les images pour l'intervalle n° : 0007 et batches_done = 5628
sauvée dans /content/gdrive/My Drive/TFE/dataset/0/2019-08-16_18-27/
Model saved in /content/gdrive/My Drive/TFE/dataset/0/2019-08-16_18-
27/model/model_from_epoch_0006.pth
```

4.2 Le test des modèles

La méthode est assez ressemblante à l'apprentissage mais simplifiée car certains paramètres ne sont plus nécessaires.

- 1) Le menu apparait et propose donc le dataset à utiliser.

```
[0] Dataset 0 - MNIST
[1] Dataset 1 - CIFAR 10
[2] Dataset 2 - CIFAR 100
[3] Dataset 3 - STL 10
[4] Dataset 4 - Fashion MNIST
[5] Dataset 5 - EMNIST
Choisissez le dataset à tester [0-5]:
```

- 2) Il faut ensuite entrer le mot que l'on veut générer

Entrer un mot à générer : (Tapez "Entrer" pour le mot par défaut)

Dans ce cas on tape « Enter pour la valeur par défaut » ou ici 135792468

3) Il propose ensuite la liste des 10 derniers modèles des différents apprentissages.

Dataset n: 0 selected and 10 classes used
Recherche dans : /content/gdrive/My Drive/TFE/dataset/0

```
[0] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_99.pth
[1] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_98.pth
[2] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_97.pth
[3] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_96.pth
[4] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_95.pth
[5] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_94.pth
[6] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_93.pth
[7] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_92.pth
[8] /content/gdrive/My Drive/TFE/dataset/0/2019-08-07_21-59/model/model_from_epoch_91.pth
[9] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/modelg.pth
[10] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/model_from_epoch_0999.pth
[11] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/model_from_epoch_0998.pth
[12] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/model_from_epoch_0997.pth
[13] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/model_from_epoch_0996.pth
[14] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/model_from_epoch_0995.pth
[15] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/model_from_epoch_0994.pth
[16] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/model_from_epoch_0993.pth
[17] /content/gdrive/My Drive/TFE/dataset/0/2019-08-08_10-56/model/model_from_epoch_0992.pth
```

Choisissez le modèle à tester [0-143]:

4) L'output sort la grille de x labels (classes) et le mot demandé 135792468

Grille de tous les labels du dataset



Mot Choisi :



5 Les résultats d'images générées.

5.1 Dataset n°0 – MNIST

Paramètres de l'apprentissage (2019-08-08_10-56) :

```
b1=0.5, b2=0.999, batch_size=32, channels=1, dataset=0, genidlabel=4,  
gennumber=1234567890, img_size=32, latent_dim=100, lr=0.0002, n_classes=10,  
n_cpu=8, n_epochs=1000, sample_interval=1875)
```



On peut voir que la qualité de l'imitation est très bonne, c'est ce qui est recherché.

On remarque aussi que le bruit autour des caractères disparaît pour avoir des images nettes au final.

[Images en début d'apprentissage :](#)



5.2 Dataset n°1 – Cifar 10 (2019-08-15_10-36)

```
Namespace(b1=0.5, b2=0.999, batch_size=16, channels=3, dataset=1,
img_size=32, latent_dim=200, lr=0.0002, n_classes=10, n_cpu=8,
n_epochs=2000, sample_interval=3125)
```



Ce dataset imité donne des résultats intéressants. On voit que l'algorithme arrive à imiter les classes.

5.2.1 Analyse par classe :

Avec beaucoup d'imagination, de réserve et des visionnages des différentes générations d'images, je vais tenter d'expliquer ce que je vois.

5.2.1.1 La classe « airplane ».

On voit une dominance d'images sur fond bleu, cela fait penser au ciel. On observe aussi des images coupées en deux (fond vert/bleu) qui seraient des images d'avion au sol.

5.2.1.2 La classe «Automobile »

On voit que dans cette classe, les images semblent être présentées sous différents angles. Les images de voiture de ce dataset le confirment.



5.2.1.3 La classe «bird »

On remarque aussi une dominance de vert et de bleu ainsi que des couleurs foncées ou brunâtres qui font penser à des oiseaux dans des arbres ou en vol.

5.2.1.4 La classe «cat »

Pour ces images, je n'arrive pas à sortir des features.

5.2.1.5 La classe «deer »

On voit aussi pour cette classe, une colorisation verte et brune avec un objet au centre qui ferait penser à l'animal dans son milieu naturel

5.2.1.6 La classe «dog »

Cette classe ne m'inspire pas non plus. (Les IA n'en ont-elles pas marre de voir des chats et des chiens ?)

5.2.1.7 La classe «frog »

Les images sont très vertes avec des lignes qui font penser à la silhouette de la grenouille.

5.2.1.8 La classe «horse »

Cette classe est pour moi la meilleure car dans de nombreuses images sur différent 'learning', des silhouettes de cheval sont reconnaissables.

5.2.1.9 La classe «ship »

Ces images sur fond bleu avec un objet au centre fait penser à un bateau.

5.2.1.10 La classe «truck »

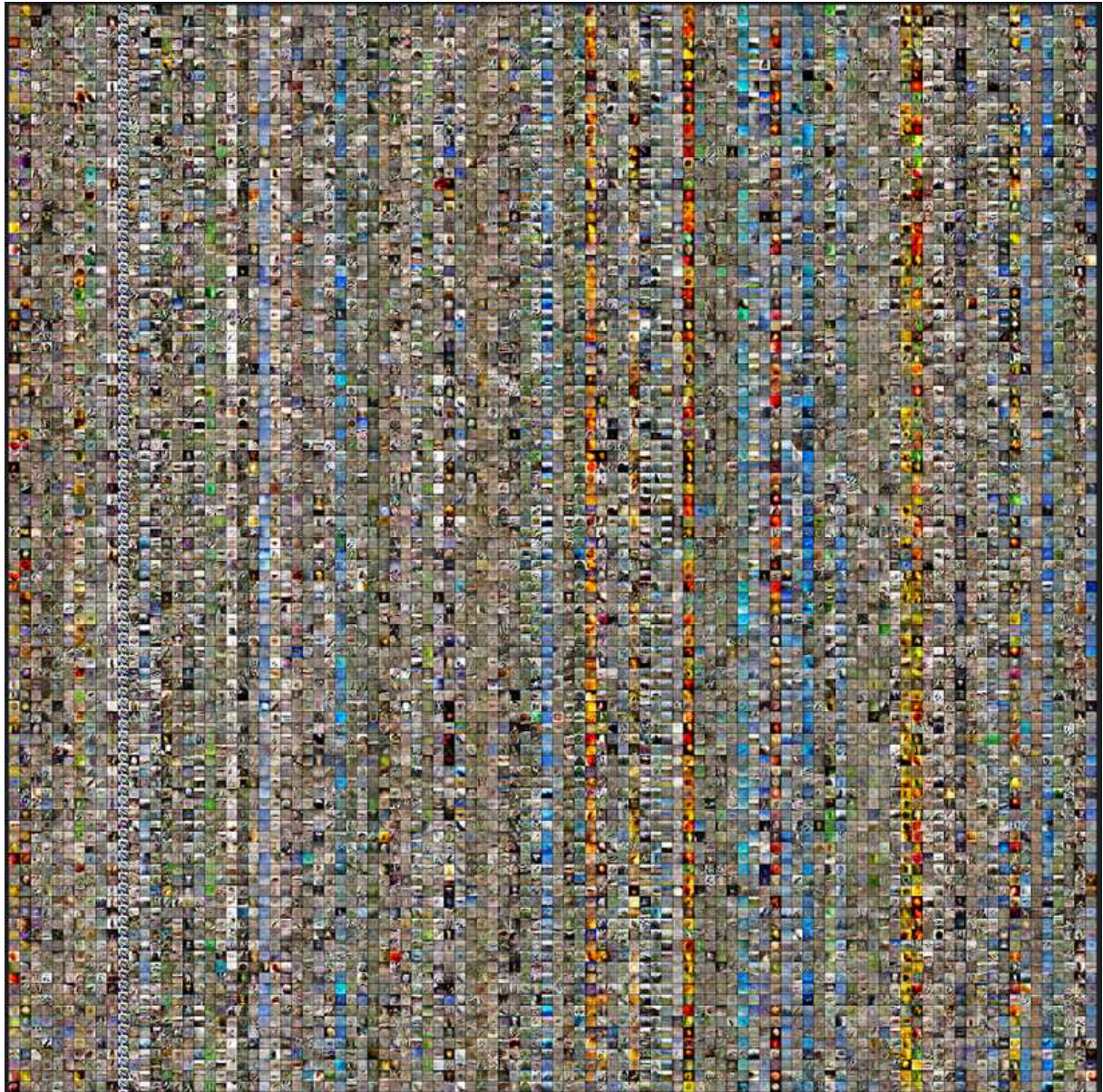
Cette classe génère des images assez semblables que la classe 'car' mais l'objet semble plus volumineux.

Il aurait été intéressant de travailler avec des images en plus haute résolution mais les différents entraînements sont déjà longs et gourmands en stockage. Il ralentit le déroulement de ce travail de fin d'études au contraire d'autres matières ne nécessitant pas de deep learning ou encore IA.

5.3 Dataset n°2 – Cifar 100 (2019-08-14_17-02)

```
Namespace(b1=0.5, b2=0.999, batch_size=16, channels=3, dataset=2,  
img_size=32, latent_dim=100, lr=0.0002, n_classes=100, n_cpu=8,  
n_epochs=3000, sample_interval=3125)
```

Ce dataset comprend 100 classes, le tableau générique est presque illisible en version papier.



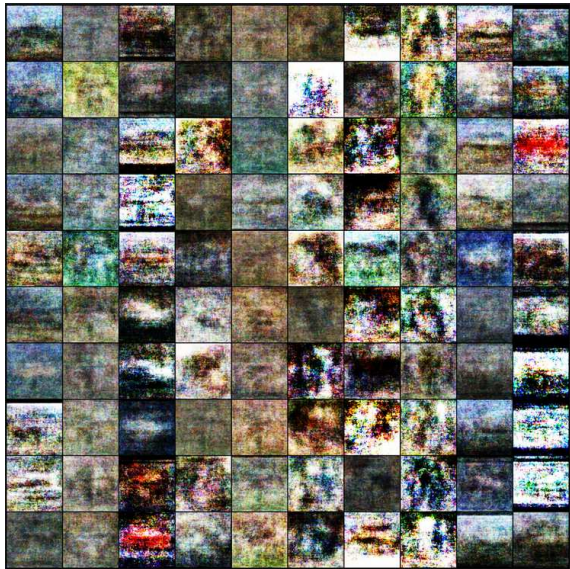
Comme le dataset précédent, on voit que le réseau génère des familles.

Des poissons seraient dans les fonds bleus, des fleurs colorées, etc...

5.4 Dataset n°3 – STL 10

Ce dataset n'a pas été beaucoup exploité à cause de son faible volume de data.

De plus, pour ce dataset le modèle standard calculé fait plus de 100Mb, l'entraînement génère trop de volume pour le peu d'intérêt. Néanmoins, on peut apercevoir des silhouettes mais avec ici encore plus d'imagination.



5.5 Dataset n°4 – Fashion -MNIST

Voici les paramètres utilisés pour cet apprentissage (2019-08-08_10-55) :

```
b1=0.5, b2=0.999, batch_size=32, channels=1, img_size=28, latent_dim=100,  
lr=0.0002, _epochs=1000, sample_interval=1875
```



Ce dataset donne de très bons résultats bien que l'image soit en niveau de gris et représente donc une image plus complexe que MNIST à entraîner.

5.6 Dataset n°5 – EMNIST

Voici les paramètres utilisés pour cet apprentissage (2019-08-10_19-47)

```
Namespace(b1=0.5, b2=0.999, batch_size=8, channels=1, dataset=6,
genidlabel=45, gennumber=1234567890, img_size=28, latent_dim=100,
lr=0.0002, n_classes=62, n_cpu=8, n_epochs=2000, sample_interval=43621)
```



Cet apprentissage a sorti un output de qualité moyenne. On remarque ici, que le modèle a du mal à apprendre des lettres présentant de caractéristiques similaires comme le m, M, w et M (et G)

D'autres entrainements ont pu quand même générer ces lettres.

5.7 L'exploitation de LOSS du générateur et discriminateur.

Le discriminateur calcule les pertes (loss) par la méthode de l'erreur quadratique moyenne.

$$l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = (x_n - y_n)^2$$

Avec N : le batch size et x, y , des tenseurs.

Deux valeurs de loss sont retournées par le discriminateur, le `d_real_loss` et `d_fake_loss`. Il représente la perte pour la détection d'une vraie et d'une fausse.

Les 2 pertes sont ensuite additionnées pour faire la moyenne de la perte du D :

$$d_loss = (d_real_loss + d_fake_loss) / 2$$

Tandis que le générateur ne calcule qu'une perte `g_loss` par la même méthode.

La suite du document va montrer le déroulement de plusieurs apprentissages et leurs paramètres.

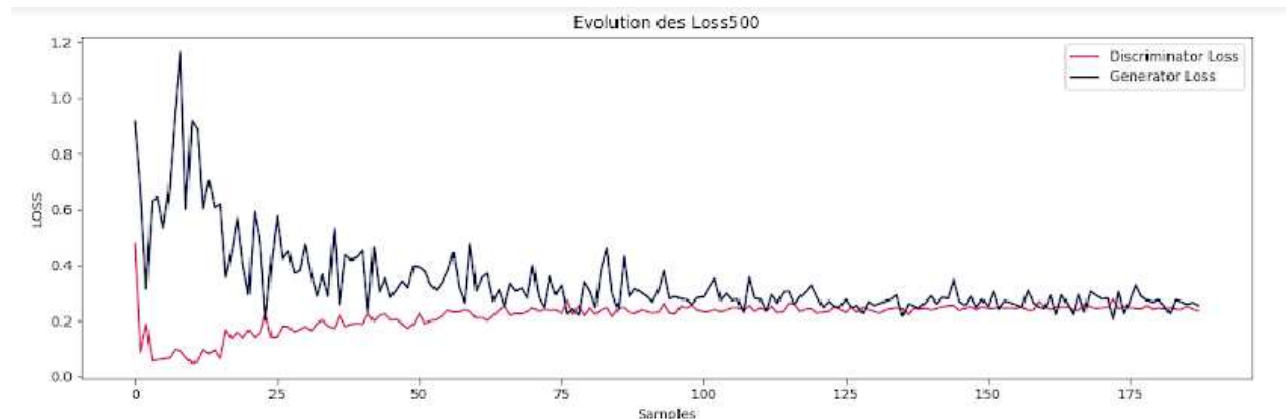
5.7.1 Dataset n°1

5.7.1.1 Graphe 1

Ce graphe des loss a été produit avec les paramètres suivants :

batch_size=64 | latent_dim=100 | lr=0.0002 | Epoch : 100 | img_size=96

du 2019_08_07-21_59



On peut remarquer qu'au début de l'apprentissage, D apprend mieux que le G et qu'il y a un minima pour D et un maxima pour G. On voit par la suite les g loss et d loss tend à se confondre. On peut considérer que D est aussi bon que G.

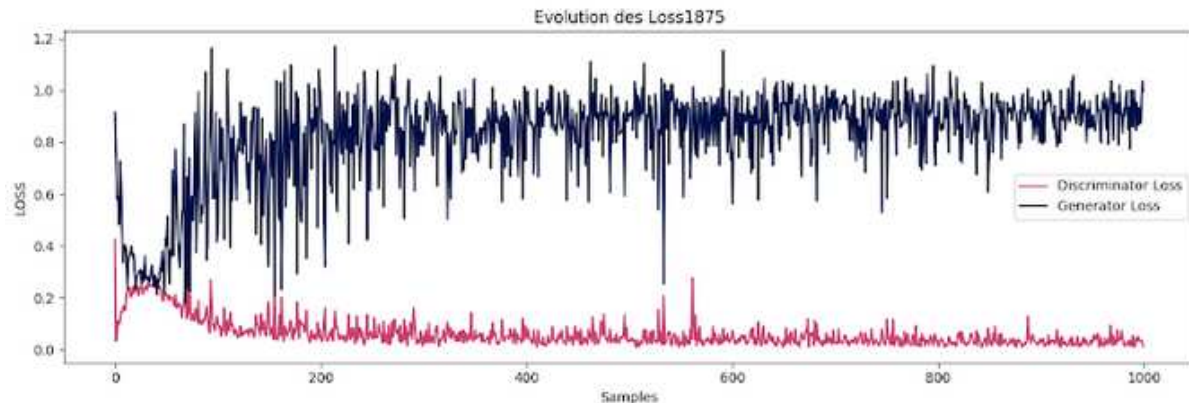
Résultat à 100 Epoch.



FIGURE 8 - 100 EPOCH - BS 64

5.7.1.2 Graphe 2

batch_size=32 | latent_dim=100 | lr=0.0002 | n_epochs=1000 du 2019-08-08_10-56



On voit ici un comportement inverse, on a changé la taille du batch en passant de 64 à 32 et la taille de l'image de 96 à 32 pixels.
Le D est meilleur que le G.

Résultat à 100 et 1.000 Epoch.



FIGURE 9 - RÉSULTAT À 100 EPOCH - BATCHSIZE 32



FIGURE 10 - RÉSULTAT À 1.000 EPOCH - BS 32

Pour les images générées à 100 et 1.000 Epoch, on constate que l'image est meilleure que celle du graphe 1 (Figure 8 - 100 Epoch - Bs 64

On pourrait se demander pourquoi la Figure 8 - 100 Epoch - Bs 64 n'est pas meilleure que la Figure 9 - Résultat à 100 Epoch - Batchsize 32 vu que D et G sont très bons.

Regardons de plus près aux valeurs des 2 figures : Pour la Figure 8 - 100 Epoch - Bs 64 et Figure 9 - Résultat à 100 Epoch - Batchsize 32, les loss valent respectivement $D \text{ loss}_1 = 0.2366362$ et $G \text{ loss}_1 = 0.2558707$.

Et pour le batch de 32, on a $D \text{ loss}_2 = 0.07406757772$ et $G \text{ loss}_2 = 0.585297644$.

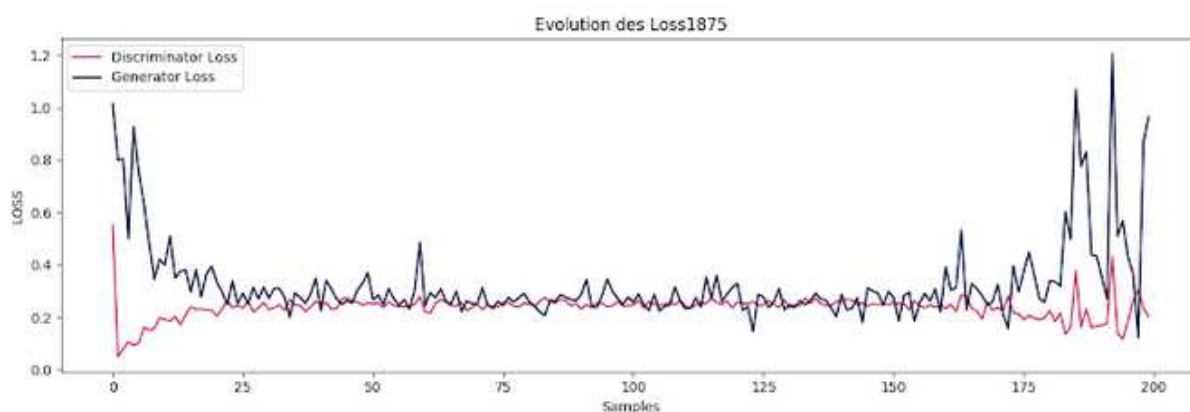
On peut en conclure que pour cette configurateur $D \text{ loss}_2$ est 3,2 x meilleur que $D \text{ loss}_2$. Proportionnellement, on conclut que le G est meilleur pour un batch de 32.

On remarque aussi que les courbes se stabilisent.

5.7.1.3 Graphe 3

batch_size=32 | img_size=96 | latent_dim=100 | n_epochs=200

du 2019-08-08_15-53



Résultat à 200 Epoch.



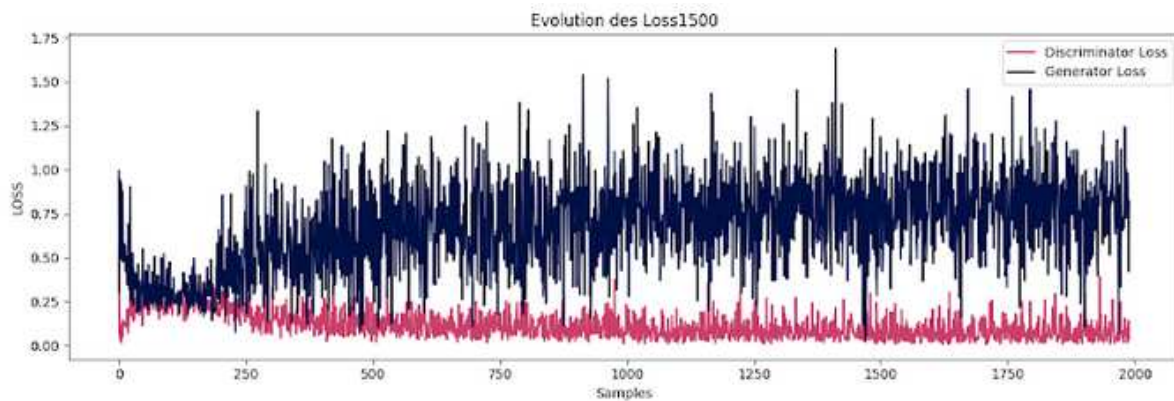
On remarque ici que le batch size utilisé est le même que le précédent mais l'apprentissage s'est fait sur des images de 96 pixels.

On remarque aussi qu'après 150 Epoch, D devient meilleur que G, ce qui diminue la performance de G. La moyenne des loss situe aux alentours de 0,25 pour arriver à $D \text{ loss } 0.1163332984$ et $G \text{ loss } = 0.5682783127$ vers 200 Epoch.

5.7.1.4 Graphe 4

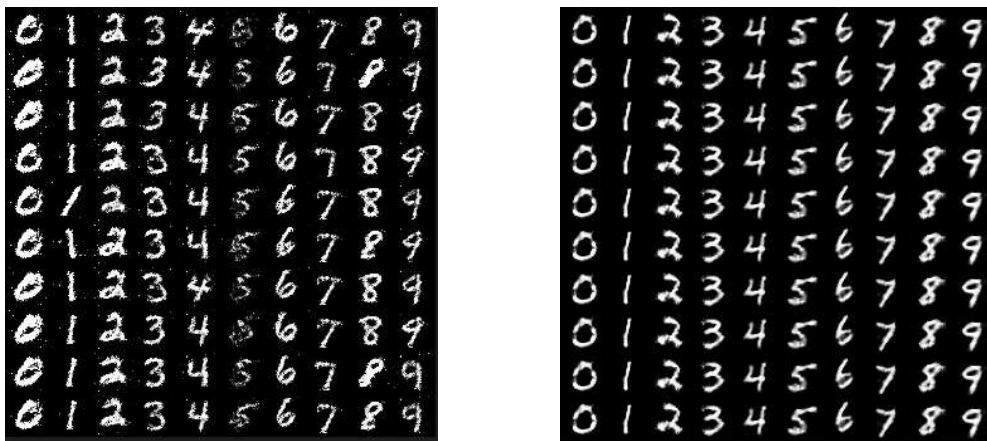
Ici un batch_size de 4 sur une img_size=32 donne des résultats semblables à un batch de 32. Les loss ont une plus grande amplitude

du 2019-08-10_17-28



En fin d'apprentissage, on obtient les valeurs suivantes (0.04068 ; 0.728588)

A 100 et 1000 Epoch, on a le résultat suivant :

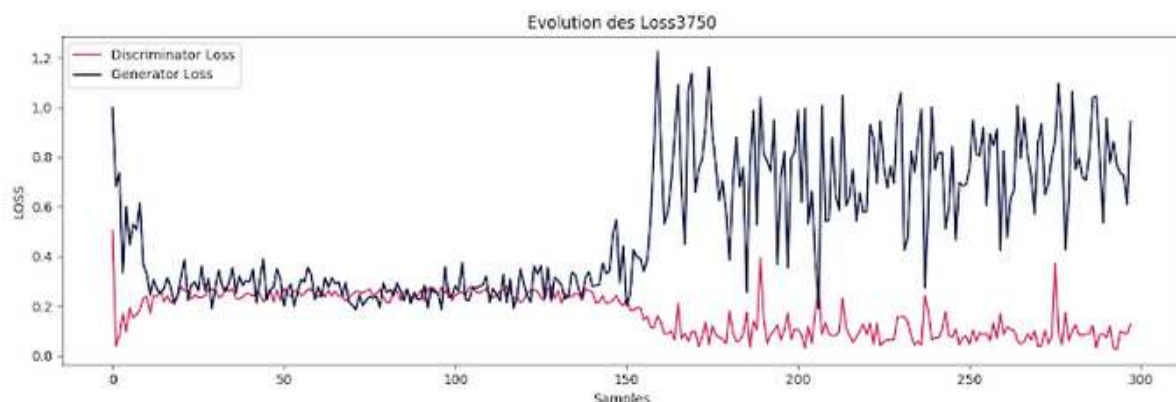


On remarque que le modèle à moins bien appris avec un Batch size de 4 à 100 Epoch qu'avec un de 32 mais on remarque qu'en fin d'apprentissage, le modèle génère des images très semblables même avec le bruit aléatoire z.

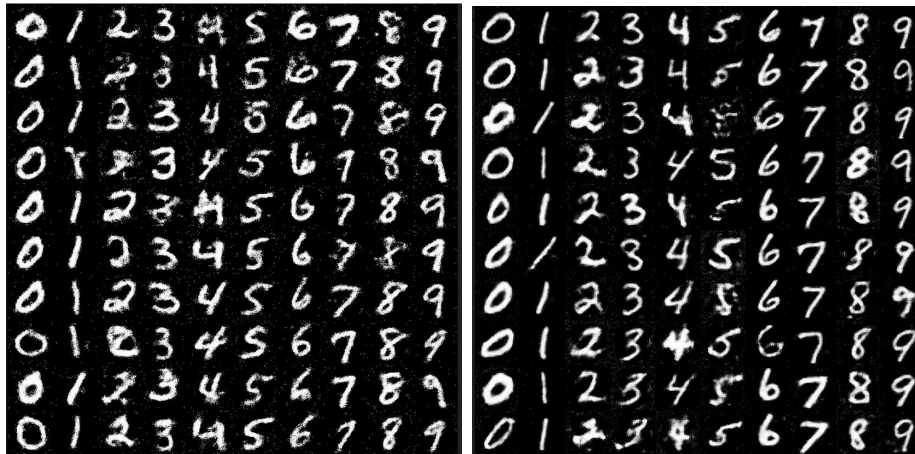
5.7.1.5 Graphe 5

```
Namespace(b1=0.5, b2=0.999, batch_size=16, channels=1, dataset=0,
genidlabel=5, gennumber=1234567890, img_size=128, latent_dim=100,
lr=0.0002, n_classes=10, n_cpu=8, n_epochs=1000, sample_interval=3750)
```

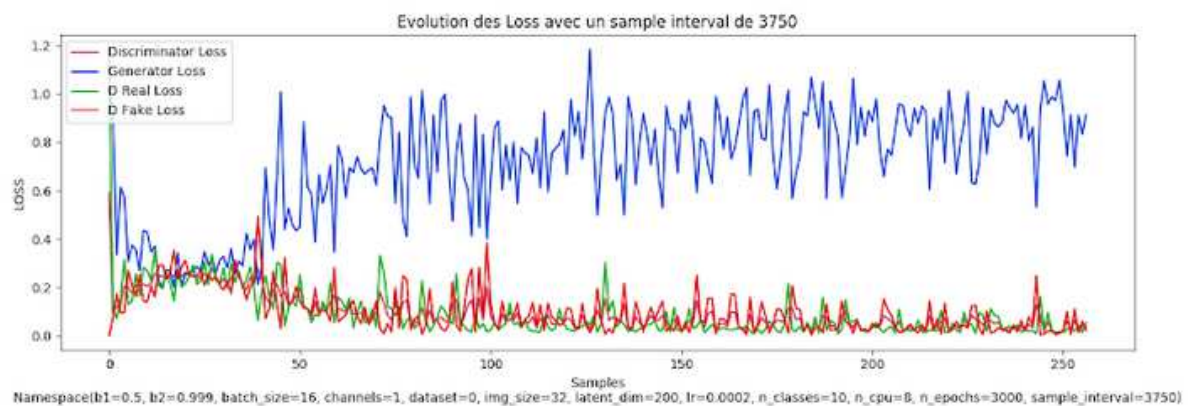
du 2019-08-11_15-08



On remarque pour les images générées au Epochs 130 et 300, l'apprentissage n'a pas évolué. On a une tendance qui augmente pour le loss du G



5.7.1.6 Graphe 5



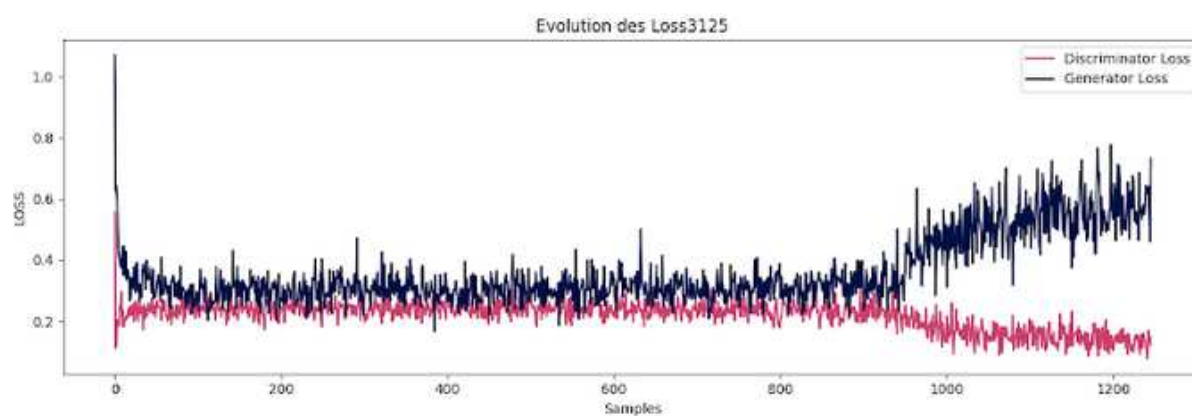
Ce graphe reprend non seulement les D loss et G loss mais aussi les D Real loss et D fake loss, on remarque que ces 2 derniers ne se distinguent pas vraiment. On peut dire que le D apprend autant à reconnaître de vraies que des fausses images.

Pour ce graphe (2019-08-15_17-18), on a paramétré latent_dim = 200 pour un batch size de 16. On a encore amélioré le D loss à 0,02053831331

Image à 100 Epoch,



Dataset 5 (2019-08-09_09-57)



6 Conclusion

Pour conclure, ce travail m'a passionné au tant au bas et moyen niveau que pour le (très) haut niveau : le deep learning. Pour moi, ce projet est un mix de programmations et d'IA. Les réseaux de neurones étant une matière très vaste, il est très facile de s'égarer, j'ai voulu ici faire un travail de premier contact avec les méthodes de deep learning.

Je parle de bas et moyen niveau, la programmation en python qui pour moi était une nouvelle matière. J'appris à manipuler, parcourir, créer, rechercher des fichiers et dossiers comme avec des commandes (`os.makedirs`, `os.walk`, `os.path`,...)

J'ai aussi appris à faire des conversions, transformation, affichage, générateur d'images, vidéos.

J'ai aussi fait l'utilisation simple de Github avec un travail sur la branch 'master'

Au niveau IA, j'ai pu étendre mes connaissances.

Je pense avoir fait un bon compromis entre programmation et deep learning.

Les outils dans ce domaine sont très nombreux, avoir fait ce travail me permettra de faciliter mes futurs algorithmes avec d'autres outils ou réseaux.

En espérant que vous avez apprécié la lecture de ce document.

7 Bibliographie

<https://www.solumaths.com/fr/graphique-logiciel-traceur-courbe/tracer>

<https://fr.wikipedia.org>

https://fr.wikipedia.org/wiki/Compute_Unified_Device_Architecture

<http://torch.ch/>

https://medium.com/@jonathan_hui/gan-whats-generative-adversarial-networks-and-its-application-f39ed278ef09

<http://webia.lip6.fr/~robert/cours/rdfia/tme10-11.pdf>