Rapport Projet Python Avancé

Entrainement d'un Generative Adversarial Network (GAN) pour générer des images de fraises

Ambroise WARNERY

Introduction

Les GAN, pour General Adversarial Networks, ou réseaux antagonistes génératifs ont été développés en 2014 par Ian Goodfellow et son équipe. Ces modèles reposent sur un principe simple. Deux modèles sont mis en concurrences. Le premier génère des images synthétiques ressemblant le plus possible aux données d'entrainement. Le second doit détecter si les images qu'on lui présentent sont de vrais image issues de la base d'entrainement ou ont été générée par le modèle génératif. En entrainant ces deux modèles l'un contre l'autre, on améliore ainsi leurs performances jusqu'à obtenir des images très ressemblantes des données d'entrainements.

La capacité des GAN à générer des images synthétiques mais réalistes est particulièrement intéressante pour agrandir des jeux de données, qui seront ensuite utilisés pour entrainer de nouveaux modèles.

Dans ce projet, nous allons réutiliser le modèle utiliser durant le cours de Python Avancé pour générer des images de fraises.

Toute la difficulté se trouve dans le calage des hyper-paramètres. En effets les modèles GAN sont très sensibles aux différents hyper-paramètres et peuvent donc donner des résultats très variables.

Note importante : Grâce à mon alternance, j'ai pu avoir accès à des GPU pour entrainer mon modèle GAN. Ces modèles sont très gourmands en puissance de calculs et cela m'aurait couté cher de faire les mêmes calculs sur Google Collab. A cause de ce problème, mon code ne peut pas être lancé sur Google Collab.

I – Construction du code

1) Préparation de l'espace de travail

Pour commencer la première cellule de mon Jupyter Notebook contient les imports de toutes les librairies python dont je vais me servir dans la suite du code.

J'utilise notamment opendatasets pour l'import de la base de données. Matplotlib, PIL et IPython pour l'afffichage des images. Keras et Tensorflow pour la création et l'entrainement des réseaux de neurones, tqdm pour avoir des barres de chargements et voir la progression des entrainements.

Dans un second temps, j'importe ma base de données. Comme celle-ci se trouve sur Kaggle, je dois rentrer mes identifiants Kaggle permettant d'utiliser leur API.

2) Prétraitement des données

Pour pouvoir utiliser les images de fraises que j'ai télécharger pour entrainer des réseaux de neurones, il faut que je les prépare. Le prétraitement choisit est simple. Je convertis toutes les images en format RGB. Ce format permet de coder la couleur de chaque pixel sous la forme d'un triplet (_, _, _). Ces trois valeurs évoluent entre 0 et 255 et représente la proportion de rouge vert et bleu. Une fois ces 3 couleurs « mélangées » on obtient une couleur spécifique. Ces couleurs sont les couleurs de base du fonctionnement des pixels des écrans d'ordinateurs, elles ne sont donc pas choisies au hasard. Afin de normaliser les valeurs de ces pixels, on les divise par 255 pour obtenir des valeurs comprises entre 0 et 1.

Ensuite, on réduit la taille en pixel des images. On passe de 300*300 pixels à 128*128.

Finalement, on stock nos images sous la formes d'un Numpy Array, qui est le format utilisés pour entrainer nos modèles.

```
#Création du np.array qui va stocker les données d'entrainement

X_train = []
for file in names_imgs:
        img = Image.open(dir_data+'/'+file)
        img = img.convert('RGB') # On convertit les images au format RGB
        img = img.resize((128,128)) # on réduit la taille des images à 128 * 128 pixels
        img = np.asarray(img)/255  # Ici on normalise les pixel.
                                   # En RGB, on code la couleur d'un pixel avec un triplet (_, _, _) dont les valeurs évoluent de 0 à 255.
                                   # On normalise ces valeurs en divisant par 255, pour qu'elles oscillent entre 0 et 1
        X_train.append(img)


X_train = np.array(X_train)
print(X_train.shape)
```

3) Construction des réseaux de neurones

Pour la construction des réseaux de neurones, comme ce n'est pas ma spécialité et que je ne sais pas quelles sont les informations qui doivent guider vers l'ajout ou le retrait d'une couche, j'ai décidé de reprendre les réseaux utilisés en cours.

Le réseau discriminant est un réseau séquentiel, car toutes les couches de neurones sont ajoutées les unes après les autres. Par opposition aux couches qui permettent d'avoir différents inputs, différents output, ou qui permettent de réutiliser l'information passées comme les RNN.

Le modèle est constitué d'une couche d'entrée, trois couches cachées et une couche de sortie. Toutes ces couches sont denses, c'est-à-dire que tous les neurones sont connectés entre eux. Ils reçoivent en entrée une valeur égale à la somme pondérée de l'ensemble des sorties des neurones de la couche précédente. Toutes ces couches utilisent la fonction d'activation LeakyReLU.

Cette fonction d'activation permet d'appliquer un filtre en sortie de couche, en amplifiant les valeurs positives tout en laissant survire les valeurs négatives. Cette fonction d'activation permet d'introduire un traitement non linéaire des informations et elle résout le problème connu sous le nom de « Dying ReLU » en laissant passer les valeurs négatives.

Des couches de « Dropout » sont aussi utilisés pour empêcher le modèle de faire du sur-apprentissage. En mettant 50% des input à 0 de manière aléatoire, on empêche le modèle de se reposer de manière trop importante sur un petit ensemble de données et on l'oblige à apprendre de manière robuste et à généraliser les données d'entrainements.

En sortie, on utilise une fonction d'activation Sigmoide pour donner une valeur comprise entre 0 et 1. Si le réseau renvoie une valeur proche de 0, il estime que l'image est probablement fausse, s'il renvoie une valeur proche de 1, il estime que l'image est probablement une originale.

Le modèle générateur est construit en utilisant les mêmes blocs de base et ajoute des couches de Batch Normalization. Celles-ci permettent de stabiliser l'entrainement en normalisant les activations de la couche précédente. Cela permet de réduire le temps d'apprentissage nécessaire. La couche de sortie est un vecteur (128, 128, 3) pour le nombre de pixels et la couleur.

4) Entrainement et test des réseaux de neurones

Après avoir définis nos réseaux de neurones et les avoir compilés, on doit les entrainer.

Pour les entrainer, un réseau de neurones, on effectue plusieurs étapes.

La première étape consiste à passer tous les exemples d'entrainement dans le modèle et à réaliser des prédictions, ensuite on fait un calcul de la perte, c'est-à-dire qu'on calcule la différence entre la sortie prédite et la sortie réelle à l'aide d'une fonction de perte. Ici la fonction de perte utilisée est la fonction binaire d'entropie croisée. Ensuite on calcule les gradients de la fonction de perte par rétropopagation. Et finalement on met à jour les paramètres du modèle pour minimiser la fonction de perte. Souvent, ces étapes sont effectuées sur des batch, c'est-à-dire des sous ensemble du data set d'entrainement. Cela permet de ne pas charger tous le dataset d'entrainement dans la mémoire d'un coup et donc d'accélérer les calculs.

Toutes ces étapes représentent une epoch, et on peut décider du nombre d'epoch ainsi que de la taille des batch.

Dans la pratique, pour un GAN, comme on a deux réseau à entrainer, les étapes sont effectuées dans un sens particulier.

On commence par générer des premières images fausses avec le générateur, ensuite on entraine le discriminateur à faire la différence entre les données d'entrainements et les fausses images du générateur.

Ensuite, on fixe les poids du modèle discriminateur et on entraine le modèle générateur avec les images classifier comme vrai par le discriminateur. Ainsi, à chaque fois qu'une image du générateur est considéré comme vrai par le discriminateur, elle va être ajouter à la base d'entrainement et va permettre d'encourager le modèle générateur vers ce types d'image.

Finalement, on affiche 10 images du générateur toutes les 10 epoch pour pouvoir suivre l'évolution de l'entrainement.

Après avoir entrainer notre modèle, on génère des images avec notre modèle générateur pour évaluer ses performances.

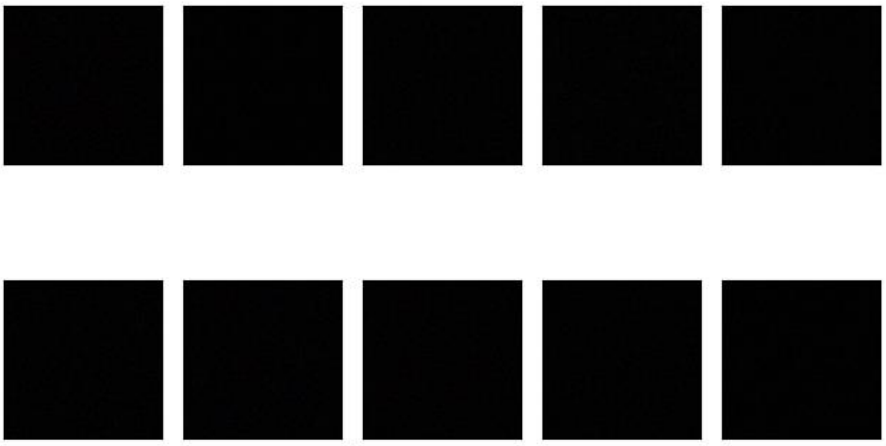
II – Résultat obtenus

Les meilleurs résultats obtenus l'ont été avec 500 epoch d'entrainement et des batch de taille 100.

Nous discuterons du choix de ces paramètres dans la dernière partie.

On peut voir que le modèle a fourni des résultats très différents au cours de son apprentissage.

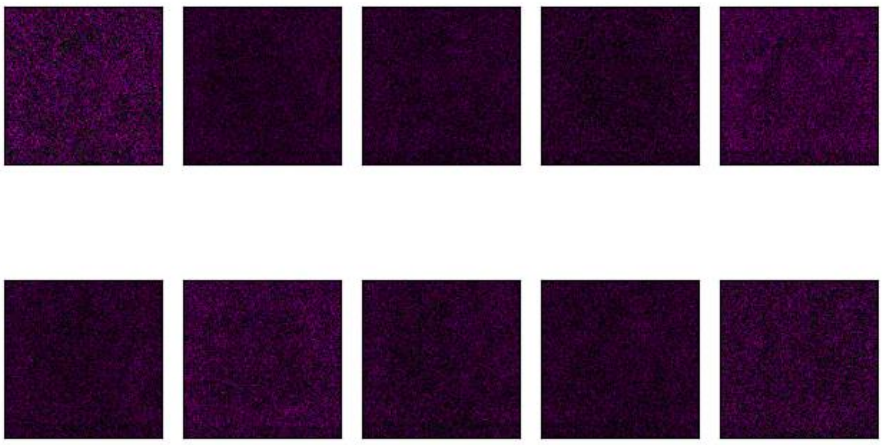
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



```
batchs in epoch 1: 100%|██████████| 2/2 [00:00<00:00,  3.64it/s]
```

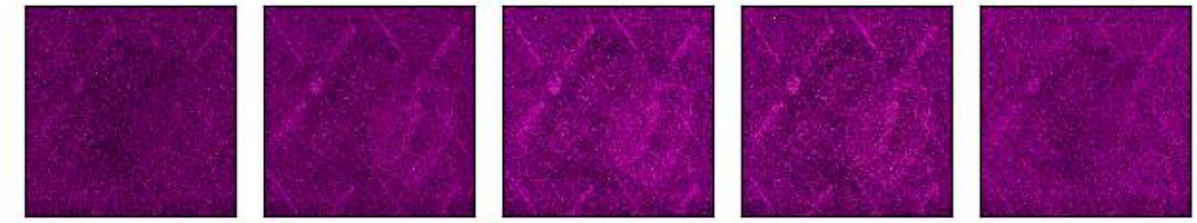
Avant son entrainement, le modèle génère des images entièrement noirs.

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



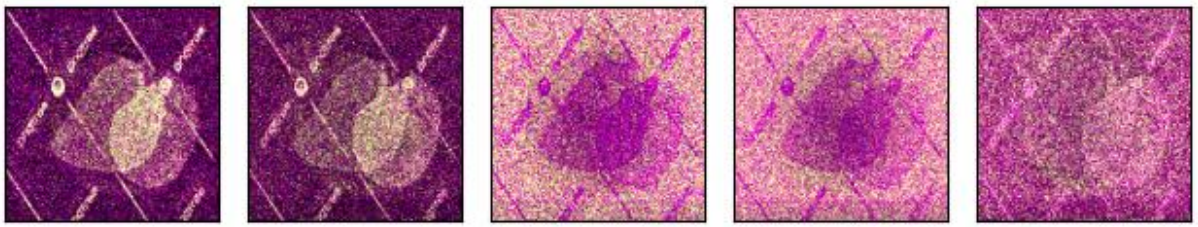
```
batchs in epoch 111: 100%|██████████| 2/2 [00:00<00:00,  3.64it/s]
```

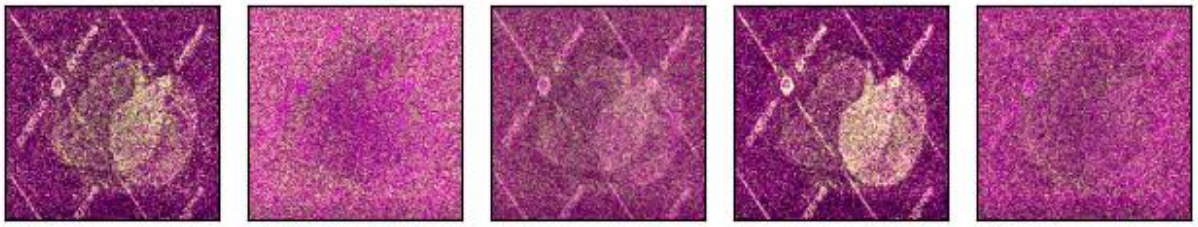
Après une centaine d'epoch, on peut voir que le modèle commence à générer des pixels plus claires.



batchs in epoch 321: 100%|██████████| 2/2 [00:00<00:00, 3.57it/s]

Au bout de 300 epoch, on voit que des formes apparaissent au centre de l'image. En effet, les fraises sont concentrées au centre de l'image dans les images d'entrainements. On voit également un quadrillage plus clair apparaitre, qui ne ressemble pas aux fraises. N discutera de ce phénomène plus tard.



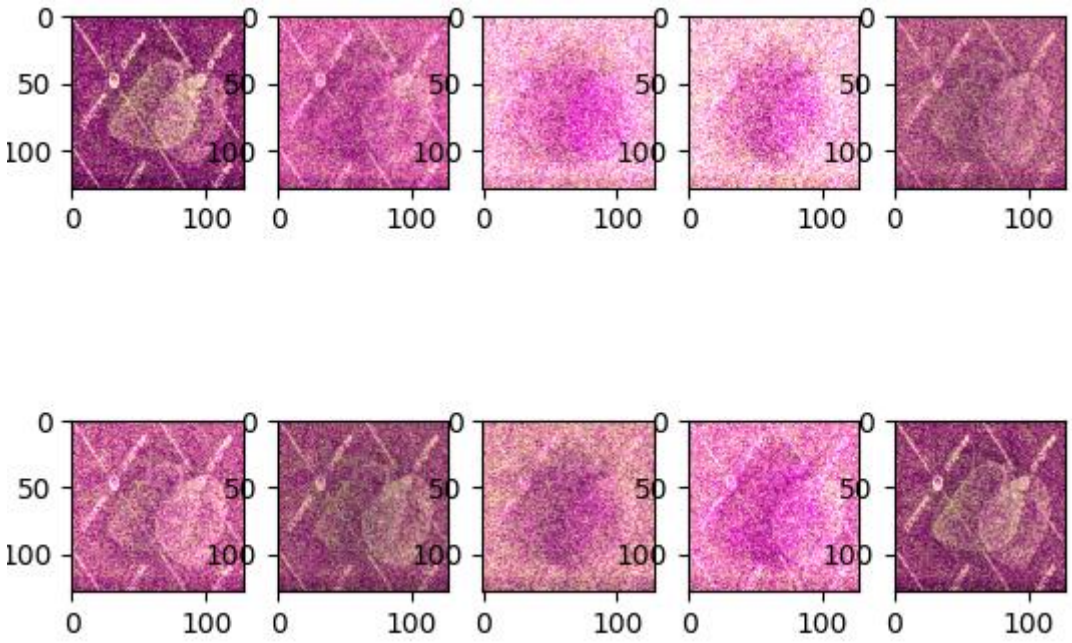


Vers la fin de l'apprentissage, on remarque trois choses :

- Un quadrillage ressemblant à des écritures est maintenant très présent sur les images.

- Les formes au centre de l’image sont presque nettes et clairement délimités mais elles sont parfois plus clair que leurs entourages et parfois plus sombres.
- Des pixels verts apparaissent de manière un peu aléatoire autour de la bordure des formes centrales.

Finalement, les résultats obtenus à la fin de l’entrainement sont les suivants :

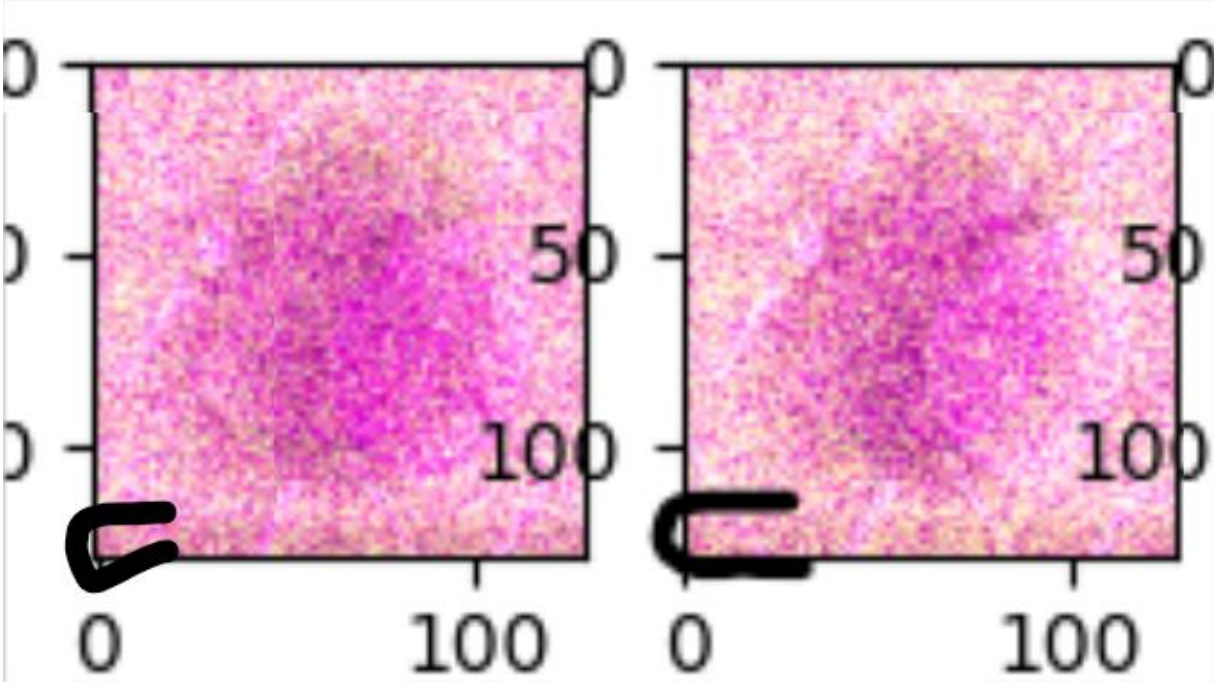


III – Analyse des résultats et des difficultés rencontrées

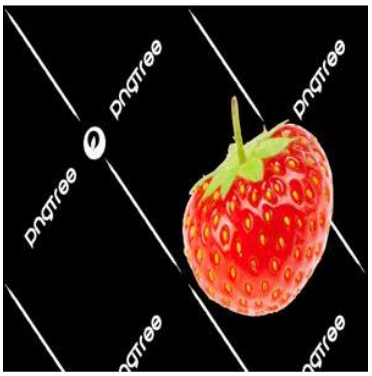
1) Analyse des résultats

Tout d'abord, on peut affirmer que les résultats ne sont pas excellents. Il est très difficile de reconnaitre une fraise dans les images générées. Les images générées sont très différentes les unes des autres, avec parfois des centres roses et des bordures plus claires mais parfois des centres clairs avec des bordures roses, cela est dû aux 4 photos sur fonds noirs qui ont dû prendre beaucoup d'importance pendant l'apprentissage. Le modèle a également beaucoup de mal à prédire ou sera la « queue » de la fraise et place des pixels verts de manière un peu aléatoirement.

On peut cependant voir que le modèle a bien appris des informations de son ensemble d'entrainement. Tout d'abord, la couleur majoritaire tourne autour du rose/rouge, ce qui est le cas dans nos données d'entrainements. Ensuite, la forme et l'endroit où se trouve les « fraises » correspond à ce qu'on peut voir dans les données d'entrainements. Sur beaucoup de photos de la base d'entrainements, il y a une petite bande noire en bas. On peut retrouver une bande légèrement sombre dans les images générées :



On retrouve également le filigrane (« watermark ») qui est présent sur beaucoup de photos de la base d'entrainement.



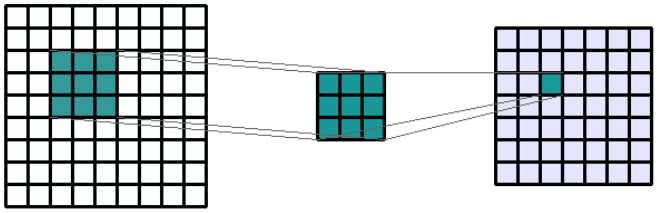
2) Choix des hyper paramètres et du modèle

Le choix des hyper paramètres a été la partie la plus complexe de ce projet. En effet, les modèles GAN sont très sensibles aux choix des hyper paramètres et finissent souvent par divergés si ceux-ci sont mal choisies. L'exemple le plus simple est l'exemple du batch_size. En effet, si celui-ci est trop grand, le modèle discriminateur aura beaucoup d'image lors de son premier entrainement et deviendra bien meilleur que le modèle générateur. Cela peut également arriver quand le batch_size est trop petit. On peut voir ci-dessous un avec batch_size = 1 ou le discriminateur est devenu bien meilleur que le générateur dès la deuxième epoch :

```
Epoch [2/20] Batch 900/1655                 Loss D: 0.0000, loss G: 13.6520
Epoch [2/20] Batch 1000/1655                 Loss D: 0.0000, loss G: 12.3711
Epoch [2/20] Batch 1100/1655                 Loss D: 0.0000, loss G: 12.8518
Epoch [2/20] Batch 1200/1655                 Loss D: 50.0000, loss G: 0.0000
Epoch [2/20] Batch 1300/1655                 Loss D: 50.0000, loss G: 0.0000
```

Figure 1 : Victoire du modèle discriminant avec une taille de batch de 1

Le modèle que j'ai utilisé un simple GAN avec des neurones Dense() donc complétement connectés. Pour la génération d'image, j'aurais pu choisir d'utiliser un modèle Deep Convolutional GAN. En effet, ceux-ci utilisent des réseaux de convolutions qui « résument » une matrice de pixel en une seule valeur. Cette capacité permet au réseau de capturer les informations de hiérarchie et de dépendance spatiale. La « queue » verte des fraises aurait sans doute était mieux capturées avec ce modèle.



Je n’ai également pas pu utiliser un nombre très grand d’epoch qui aurait peut-être permis à mon modèle d’apprendre plus précisément la distribution de pixel ressemblant à mes images d’entrainements. En effet, la puissance de calcul à laquelle j’ai eu accès n’est pas illimitées et l’entrainements sur un seul GPU prend beaucoup de temps.

3) Défauts de la base d’entrainements choisies

La base que j’ai utilisée n’était pas non plus de très grande qualité. La présence des watermark en grands nombre et les images sur fonds noirs ont par exemple affecté l’entrainement et le résultat. Le faible nombre de photos est également un problème qui a certainement affectée les capacités de mon modèle à « comprendre » à quoi ressemble statistiquement une fraise.