TP2 : implémentation d'un réseau profond

Dans ce TP, nous allons utiliser Tensorflow et Keras pour implémenter des réseaux avec un grand nombre de couches. Pour des raisons pratiques, nous travaillerons avec Google Colab.

I Petit échauffement avec TensorFlow et Keras

Nous allons dans un premier temps reproduire notre réseau multi-couches avec ces librairies.

➔ Il est vivement conseillé de séparer les parties du code dans des cellules séparées pour éviter de devoir ré-exécuter certaines parties ou re-télécharger un dataset.

On commence par importer les librairies :

```
import tensorflow as tf
from tensorflow import keras

import numpy as np
import matplotlib.pyplot as plt
```

Puis on charge le dataset MNIST :

```
# import du dataset
 (img_train, label_train),(img_test,label_test) = keras.datasets.mnist.load_data()

# conversion des images en float
img_train=img_train/255.0
img_test=img_test/255.0

# conversion des labels en vecteurs de sortie avec to_categorical (avec 10 classes)
output_train=keras.utils.to_categorical(label_train, num_classes=10)
output_test=keras.utils.to_categorical(label_test, num_classes=10)

# on vérifie la taille de notre dataset
print(img_train.shape)
print(output_train.shape)

# on affiche un échantillon d'images
for i in range(10):
  plt.subplot(2,5,i+1)
  plt.imshow(img_train[i], cmap='gray')

plt.show()
```

On construit le réseau. Notons que la première couche d'entrée est de type Flatten pour convertir les images d'entrée en vecteurs. On ajoute ensuite deux couches Dense avec respectivement 20 et 10 neurones, avec une fonction d'activation sigmoïde :

```
model = keras.Sequential( )

model.add(keras.layers.Flatten(input_shape=(28,28)))
model.add(keras.layers.Dense(20, activation='sigmoid'))
model.add(keras.layers.Dense(10, activation='sigmoid'))

# on affiche la structure
model.summary()
```

Observez la structure du réseau et le nombre de paramètres modifiables

On compile le réseau après avoir choisi la fonction de coût MeansSquaredError et la descente de gradient avec un taux d'apprentissage à 0.05 :

```
loss=keras.losses.MeanSquaredError()
optim=keras.optimizers.SGD(learning_rate=0.05)

model.compile(loss=loss, optimizer=optim, metrics=["accuracy"])
```

Il ne reste plus qu'à entraîner le réseau à l'aide de la fonction fit. Nous effectuerons 20 epochs et un batch de 4 :

```
model.fit(img_train, output_train, batch_size=4, epochs=20, shuffle=True, verbose=2)
```

On notera que le résultat est un peu plus faible que sur notre implémentation précédente en Java : l'utilisation de plusieurs exemples simultanément (batch) permet d'accélérer l'apprentissage au détriment de l'efficacité du réseau.

On peut évaluer le réseau sur le dataset de test :

```
model.evaluate(img_test,output_test, batch_size=4, verbose=2)
```

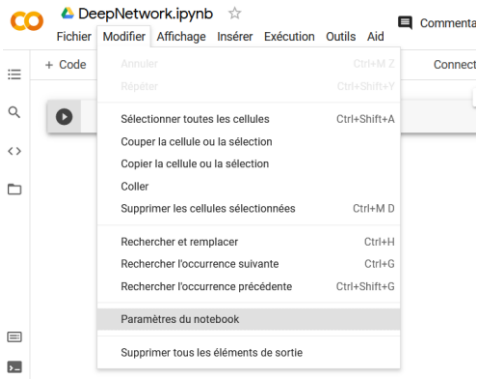
Puis l'utiliser pour faire des prédictions :

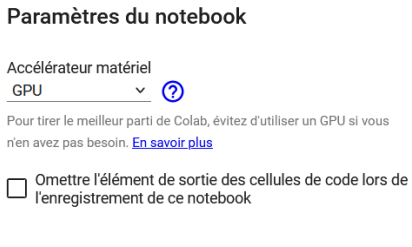
```
for i in range(50):
  output=model.predict(img_test[i:i+1,:,:])
  print(label_test[i]," , ",output.argmax())
```

II Un premier réseau profond

Passons maintenant aux choses sérieuses : nous allons définir un réseau profond convolutif pour améliorer la reconnaissance des images du dataset *MNIST fashion*. Créez un nouveau document Colab que vous appellerez *DeepNetwork1*.

➔ Afin de permettre l'apprentissage de nos réseaux en un temps raisonnable, nous allons utiliser les GPU fourni par Google Colab : Modifier -> Paramètres du notebook -> Accélérateur matériel sur GPU





➔ Pensez à désactiver le GPU et arrêter les sessions actives quand vous terminerez pour être sûr de libérer les ressources.

Exécutez le code suivant pour tester le GPU :

```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

Si le message apparait, le GPU est fonctionnel.

Ajoutez dans cette même cellule Colab les librairies à importer :

```
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
```

Nous allons maintenant charger le dataset et convertir les formats.

```
import tensorflow_datasets as tfds
from tensorflow.keras.utils import to_categorical

dataset = keras.datasets.fashion_mnist
(img_train, label_train),(img_test,label_test) = dataset.load_data()

# conversion des images en float
img_train=img_train/255.0
img_test=img_test/255.0

output_train=keras.utils.to_categorical(label_train, num_classes=10)
output_test=keras.utils.to_categorical(label_test, num_classes=10)

print(img_train.shape)
print(img_test.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
```

Nous obtenons respectivement 60 000 et 10 000 images 2D de 28x28 pixels.
Problème : les images pour un réseau débutant avec une couche de convolution doivent avoir une profondeur. Les images couleur ont déjà 3 canaux, mais ce n'est pas le cas d'une image en niveau de gris. On ajoute donc les instructions suivantes pour ajouter une dimension à nos images (puis, affichez la forme de ces matrices) :

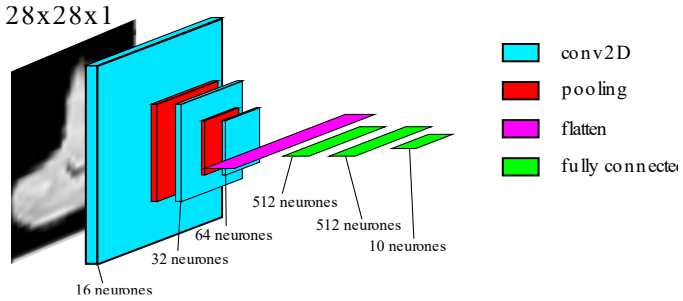
```
img_train2 = np.expand_dims(img_train, axis=-1)
img_test2  = np.expand_dims(img_test, axis=-1)
```

```
(60000, 28, 28, 1)
(10000, 28, 28, 1)
```

On affiche quelques-unes des images (en utilisant les images à 2 dimensions) pour vérifier :

```
for i in range(10):
  plt.subplot(2,5,i+1)
  plt.imshow(img_train[i], cmap='gray')

plt.show()
```

Construisez maintenant un réseau séquentiel suivant cette architecture :



Architecture à reproduire

Observez le nombre de paramètres entraînables. La taille de ce réseau reste raisonnable pour un apprentissage.

Compilez votre réseau avec une fonction de coût de type *CategoricalCrossEntropy* et une fonction d'optimisation *Adam* avec un taux d'apprentissage de 0.001 . Vous enregistrerez l'efficacité ('accuracy') pour suivre l'évolution pendant l'apprentissage

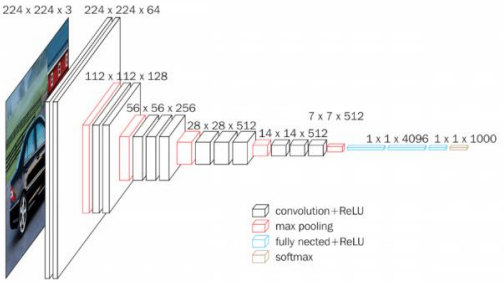
Effectuez l'apprentissage de votre réseau sur le dataset d'entraînement. Effectuez 20 epochs avec une taille de batch de 16. L'apprentissage dure environ 5 minutes, profitez-en pour souffler un peu…

Mesurez enfin les performances de votre réseau avec le dataset de test. Notez le taux de réussite du réseau.

```
model.evaluate(img_test2, output_test, batch_size=16, verbose=2)
```

III Continuons avec le transfer learning

Le transfer learning va nous permettre de construire et entraîner un réseau doté d'un grand nombre de couches, avec un dataset limité et en un temps raisonnable. Nous nous baserons sur une architecture très utilisée dans la reconnaissance d'image : VGG16
Créez un nouveau document Colab que vous appellerez *DeepNetwork2* (n'oubliez pas d'arrêter le précédent)



L'architecture VGG16

Ce réseau a été proposé à l'ILSVRC 2014, et a été entraîné sur le dataset ImageNet, regroupant 14 millions d'images réparties en 1000 classes (entraînement sur des GPU NVIDIA Titan Black pendant plusieurs jours).

Nous allons pour cela récupérer les couches de ce réseau (avec leur poids entraînés), figer ces couches et remplacer la partie fully-connected pour la remplacer par un nouveau réseau. Ce réseau aura pour but de reconnaitre des images de chats et de chiens.

Nous utiliserons cette fois un dataset disponible en ligne, que nous pouvons, sou Colab, récupérer avec la commande suivante (utilisez une cellule à part pour ne pas la recharger) :

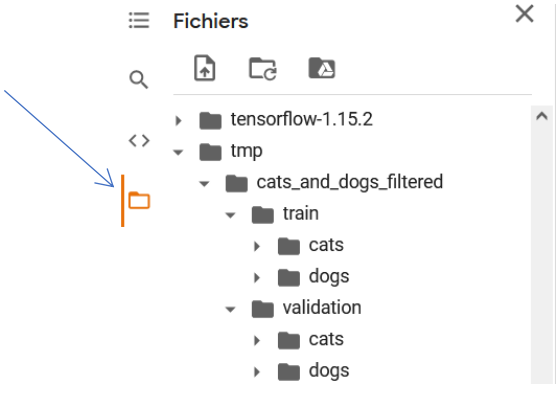
```
!wget --no-check-certificate \
    https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip \
    -O /tmp/cats_and_dogs_filtered.zip
```

Décompressons maintenant l'archive et notons les chemins vers les dossiers (à adapter si vous travaillez en local). Les fichiers sont visibles dans l'explorateur, disponible à gauche de l'interface Colab.

```
import os
import zipfile

local_zip = '/tmp/cats_and_dogs_filtered.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp')
zip_ref.close()

train_dir = '/tmp/cats_and_dogs_filtered/train'
test_dir = '/tmp/cats_and_dogs_filtered/validation'
```

L'arborescence, après décompression, ressemble à l'arbre suivant :

```
--- cats_and_dogs_filtered
                    |
                    |----------train
                    |              |
                    |              |---------- cats
                    |              |              |--------liste d'images .jpg
                    |              |
                    |              |----------dogs
                    |                             |--------liste d'images .jpg
                    |----------validation
                                   |
                                   |---------- cats
                                   |              |--------liste d'images .jpg
                                   |
                                   |----------dogs
                                                  |--------liste d'images .jpg
```



Ce type d'arborescence n'est pas anodin : nous allons pouvoir l'exploiter pour générer le dataset de test et d'apprentissage à l'aide d'une classe particulière de Keras : le ImageDataGenerator

Cette classe va parcourir cette arborescence pour alimenter le réseau en exemples d'apprentissage ou de test, en tenant compte des sous-dossiers pour définir le label de chaque image. Dans une nouvelle cellule, créez deux générateur (pour l'apprentissage et pour l'évaluation)

```
trainDataGenerator = keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
trainData = trainDataGenerator.flow_from_directory(directory=train_dir, target_size=(224,224))

testDataGenerator = keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
testData = testDataGenerator.flow_from_directory(directory=test_dir, target_size=(224,224))
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

Nous allons maintenant charger le réseau VGG16 pré-entraîné, avec les poids obtenus après entrainement sur le dataset ImageNet. Keras propose des fonctions pour charger les réseaux les plus connus et les plus efficaces (liste disponible sur https://keras.io/api/applications/). Observez le nombre de paramètres entraînables.

```
VGG16= keras.applications.VGG16(weights="imagenet")

VGG16.summary()
```

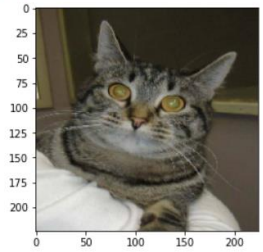
Il est évident que nous ne pourrons pas entraîner un réseau de cette taille nous-même. Nous pouvons tout de même exploiter ce réseau pour faire des prédictions sur nos images. Récupérez la liste des images disponibles :

```
train_cats_files = os.listdir(train_dir+"/cats")
print(train_cats_files[:10])

train_dogs_files = os.listdir(train_dir+"/dogs")
print(train_dogs_files[:10])
```

Puis récupérez une image (en changeant l'index et le dossier 'dogs' ou 'cats') et effectuez une prédiction

```
index=3
img = keras.preprocessing.image.load_img(
                                    test_dir+"/cats/"+test_cats_files[index],target_size=(224,224))
img = np.asarray(img)
plt.imshow(img)
img = np.expand_dims(img, axis=0)

prediction=VGG16.predict(img)
prediction.argmax()
```

```
281
```



La liste des 1000 classes d'ImageNet est disponible sur https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a (la classe 281 correspond à 'tabby cat').

Le réseau a donc déjà été entraîné avec des images de chats et de chiens, ce qui va faciliter le transfer learning pour notre problème de reconnaissance chat VS chien.

Keras permet de charger uniquement les couches convolutives d'un réseau avec le paramètre 'include_top' que nous mettrons à False. Chargez à nouveau le réseau VGG16, mais sans la partie fully-connected :

```
base_VGG16= keras.applications.VGG16(weights="imagenet", include_top=False, input_shape=(224,224,3))

base_VGG16.summary()
```

Notez où le réseau est tronqué. A noter également, avec la version tronquée du réseau, il devient possible d'utiliser des images de taille 150x150px (nous resterons pour la suite du TP avec les images de 224x224).

Nous allons faire de l'extraction de feature, et donc, geler toutes les couches de convolution, en mettant le paramètre 'trainable' du réseau à False :

```
base_VGG16.trainable=False

base_VGG16.summary()
```

Vous pourrez noter que tous les paramètres sont désormais passés en 'Non-trainable', ce qui nous arrange…

À noter : les couches du réseau ont aussi un paramètre 'trainable', il est donc possible de ne figer qu'une partie du réseau pour faire un fine-tuning partiel.

Il faut maintenant remplacer la partie fully-connected manquante. Nous proposons d'ajouter une couche cachée de 50 neurones avec une fonction d'activation ReLU et une couche de sortie avec deux neurones avec une fonction d'activation SoftMax.

Les deux approches, séquentielle et fonctionnelle, sont possibles pour former le réseau complet. Nous utiliserons l'approche séquentielle.

Commencez par créer un nouveau réseau global, puis ajoutez-y le réseau VGG16 tronqué.

```
model=keras.models.Sequential()
model.add(base_VGG16)
```

On ajoute ensuite une à une les couches nécessaires pour compléter le réseau.

Affichez la structure du réseau (summary). Vous aurez noté comment des réseaux peuvent être construits à partir d'autres réseau, ce qui est très pratique pour créer des réseaux plus complexes comme les GANs.

Compilez ensuite le réseau global, en utilisant la fonction de coût CategoricalCrossentropy et la fonction d'apprentissage Adam (avec un learning rate à 0.001)

On peut ensuite entraîner et évaluer le réseau avec nos générateurs de dataset (les labels sont fournis par ces générateurs)

```
model.fit(trainData, epochs=5, batch_size=32)
```

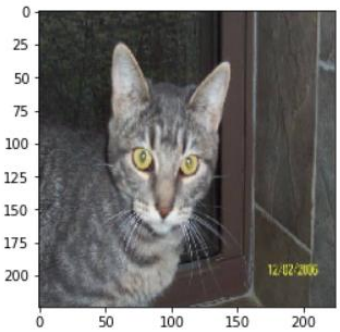
```
model.evaluate(testData, batch_size=32, verbose=2)
```

Notez les performances de notre réseau. Pas si mal pour un réseau entraîné avec un dataset de 2000 images !

Vous pourrez ensuite utiliser votre réseau pour reconnaitre des images (comme précédemment). Le code ci-dessous permet d'écrire en clair le label retourné par le réseau.

```
prediction=model.predict(img)
if prediction.argmax()==0:
  print("cat")
else :
  print("dog")
```

```
cat
```

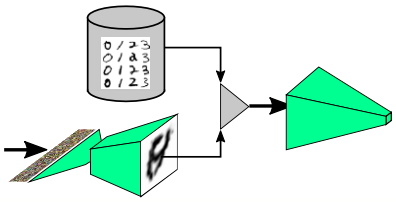


IV Construisons un GAN générateur d'images

Nous allons réaliser un petit réseau générateur qui devra générer des images de chiffres manuscrits, du même type que ceux de la database MNIST. (Cet exercice est adapté du tutoriel suivant : https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/ -> très bon site pour aller plus loin dans le domaine du deep learning)

Notre GAN sera constitué de :

- La base d'image MNIST, qui servira de base comparative
- Un réseau convolutif générateur, qui devra générer les images
- Un générateur de vecteurs aléatoires, pour alimenter le réseau générateur
- Un réseau discriminateur, qui devra apprendre à différentier les images générées et réelles



Commençons par préparer l'environnement de travail : créez un nouveau document Colab que vous appellerez GAN1 (n'oubliez pas d'arrêter les autres sessions) et importez les librairies nécessaires :

```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

from tensorflow import keras

import numpy as np
import matplotlib.pyplot as plt
```

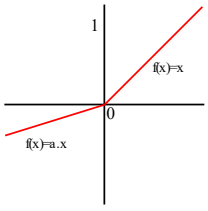
Chargez ensuite la base MNIST. Nous utiliserons uniquement la base d'images d'entraînement :

```
# on charge le dataset
(img_train, _), (_,_) = keras.datasets.mnist.load_data()
img_train = img_train / 255.0
dataset = np.expand_dims(img_train, axis=-1)

for i in range(10):
  plt.subplot(2,5,i+1)
  plt.imshow(img_train[i], cmap='gray')
plt.show()
```

On définit ensuite nos deux réseaux, en commençant par le discriminateur.

Ce réseau comportera deux couches de convolution avec 64 neurones consécutives, avec une fonction d'activation LeakyReLU, une couche Flatten et enfin une couche fully connected avec une sortie unique (un seul résultat : vrai ou fake) et une fonction d'activation sigmoïde.

La fonction d'activation LeakyReLU ne pouvant pas être utilisée en argument d'une couche de neurone (car elle nécessite ses propres paramètres), on utilisera une couche dédiée. à noter, toutes les fonctions d'activation disposent d'une couche Keras dédiée.



On compilera ensuite le réseau avec la fonction de coût BinaryCrossEntropy et la fonction d'optimisation Adam

```
discriminator_model = keras.Sequential()

discriminator_model.add(keras.layers.Conv2D(64, (3,3),input_shape=(28,28,1),
   strides=(2, 2), padding='same'))
discriminator_model.add(keras.layers.LeakyReLU(alpha=0.2))
```

```
discriminator_model.add(keras.layers.Conv2D(64, (3,3), strides=(2, 2), padding='same'))
discriminator_model.add(keras.layers.LeakyReLU(alpha=0.2))
discriminator_model.add(keras.layers.Flatten())
discriminator_model.add(keras.layers.Dense(1, activation='sigmoid'))

discriminator_model.summary()

loss=keras.losses.BinaryCrossentropy()
optim = keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
discriminator_model.compile(loss=loss, optimizer=optim, metrics=['accuracy'])
```

Puis on crée le réseau générateur. Pour passer d'un vecteur, dont nous définirons la taille à 100, à une image de 28x28 pixels, nous allons procéder en plusieurs étapes :

- On commence par générer une image de 7x7 pixels avec une profondeur de 128. Pour cela, on utilise une couche fully-connected avec 128x7x7 sorties, que l'on va organiser en 128 images (profondeurs) à l'aide d'une couche de type Reshape.
- On effectue ensuite deux étapes de convolution transposée/unpooling pour 'agrandir' les images jusqu'à la taille de 28x28. A noter, pour faire un unpooling de type 'bed of nails', on utilisera un stride de (2,2) dans la convolution transposée (rappel : sur la convolution transposée, le stride 'écarte' les pixels. Avec un stride de (2,2), l'image de sortie sera deux fois plus haute et large)
- Une dernière étape avec un unique neurone convolutif va générer une image de profondeur 1.

A noter : on ne compilera pas ce réseau (du moins, pas directement).

```
# espace latent à 100 dimensions
latent_dim=100

generator_model = keras.Sequential()

# on génère une image de 7x7 pixels avec une profondeur de 128
n_nodes = 128 * 7 * 7
generator_model.add(keras.layers.Dense(n_nodes, input_dim=100))
generator_model.add(keras.layers.LeakyReLU(alpha=0.2))
generator_model.add(keras.layers.Reshape((7, 7, 128)))

# convolution transposée et unpooling (7x7 -> 14x14)                    ↓ unpooling
generator_model.add(keras.layers.Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
generator_model.add(keras.layers.LeakyReLU(alpha=0.2))

# convolution transposée et unpooling (14x14 -> 28x28)                  ↓ unpooling
generator_model.add(keras.layers.Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
generator_model.add(keras.layers.LeakyReLU(alpha=0.2))

# convolution unique pour réduire la profondeur à 1
generator_model.add(keras.layers.Conv2D(1, (7,7), activation='sigmoid', padding='same'))

generator_model.summary()
```

On assemble ensuite le réseau générateur et le réseau discriminateur pour former le GAN. On compilera ensuite le réseau complet.

```
# connect them
gan_model = keras.Sequential()
# add generator
gan_model.add(generator_model)
# add the discriminator
gan_model.add(discriminator_model)

loss=keras.losses.BinaryCrossentropy()
optim = keras.optimizers.Adam(lr=0.0002, beta_1=0.5)
gan_model.compile(loss=loss, optimizer=optim)

gan_model.summary()
```

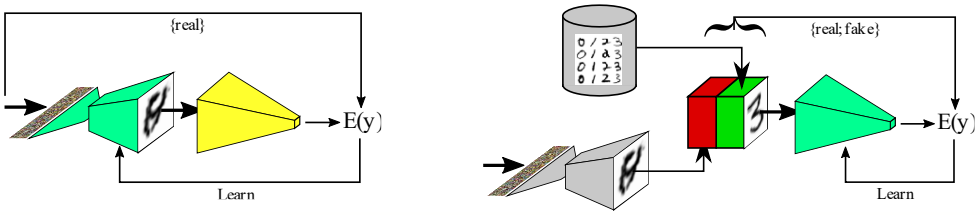
Notez la structure hiérarchique des réseaux : il est toujours possible d'entraîner le discriminateur indépendamment du reste du GAN (pas le générateur puisque non compilé).

L'entraînement du GAN est particulier : il faut entraîner, de façon alternée, le réseau discriminateur sur la base d'images réelles et générées, et le réseau générateur, par le biais du GAN entier, sur la base de vecteurs aléatoires.

Nous allons gérer nous-même les étapes de l'apprentissage : on va se servir d'une fonction d'apprentissage de Keras : *train_on_batch*. Cette fonction permet un apprentissage sur un unique paquet d'exemples (ou *batch*), ce qui nous permettra d'entraîner de façon alterné le réseau discriminateur et le réseau générateur.

Il faut donc pouvoir générer des batchs d'apprentissage :

- Une liste de vecteurs aléatoire pour alimenter le GAN
- Une liste d'images réelles issues du dataset et des images générées par le générateur pour entrainer le réseau discriminateur



Apprentissage du générateur (à gauche) et du discriminateur (à droite)

On définit une fonction pour générer une série de vecteurs aléatoires d'une taille donnée (ici, le réseau générateur utilise des vecteurs de taille 100) :

```
# échantillon de vecteurs de l'espace latent
def generate_latent_vectors(latent_dim, n_samples):
  input_vectors = np.random.randn(latent_dim * n_samples)
  input_vectors = input_vectors.reshape(n_samples, latent_dim)
  return input_vectors
```

On définit ensuite une fonction pour récupérer un ensemble aléatoire d'images du dataset. Nous utiliserons le label '1' pour désigner une image réelle :

```
# échantillon d'images réelles
def generate_real_samples(dataset, n_samples):
  rand = np.random.randint(0, dataset.shape[0], n_samples)
  X = dataset[rand]
  # vecteur de 1 pour désigner les vraies images
  y = np.ones((n_samples, 1))
  return X, y
```

Enfin, on définit une fonction pour générer un ensemble de 'fausses' images. Pour cela, on génère un ensemble de vecteurs aléatoires qui sont envoyés dans le réseau générateur. Nous utiliserons le label '0' pour désigner une image générée :

```
# échantillon d'images générées par le réseau générateur
def generate_fake_samples(g_model, latent_dim, n_samples):
  input_vectors = generate_latent_vectors(latent_dim, n_samples)
  X = g_model.predict(input_vectors)
  y = np.zeros((n_samples, 1))
  return X, y
```

Afin de conserver une trace des images générées par notre GAN, voici une fonction pour enregistrer un échantillon de 100 images sous forme d'un fichier png. On génère un vecteur de 100 échantillons de l'espace latent qui servira à générer les images, afin de voir l'évolution, epoch après epoch, des images générées par ces mêmes 100 points.

```
# un échantillon de 100 vecteurs de l'espace latent
fake_sample_latent=generate_latent_vectors(latent_dim, 100)
```

```
# fonction pour enregistrer un échantillon d'images générées au format png
def save_image(epoch, n=10):

  # on génère les images depuis l'échantillon de vecteur
  sample_fakes=generator_model.predict(fake_sample_latent)

  # on assemble l'image
  for i in range(n * n):
    plt.subplot(n, n, 1 + i)
    plt.axis('off')
    plt.imshow(sample_fakes[i, :, :, 0], cmap='gray_r')
  # on enregistre l'image
  filename = '/tmp/generated_plot_e%03d.png' % (epoch+1)
  plt.savefig(filename)
  plt.close()
```

On effectue enfin l'apprentissage du réseau. Pour l'apprentissage du discriminateur, on construit un batch constitué à moitié d'images réelles et à moitié d'images issues du générateur. Pour l'apprentissage du générateur, on fige les poids du discriminateur, et on construit un batch de vecteurs aléatoires avec le label '1' (le résultat que l'on veut obtenir 'côté générateur') pour effectuer un entraînement sur le GAN.

La séquence suivante marche plutôt bien : deux batchs pour le discriminateur, un batch pour le générateur.

```
n_epochs=10
n_batch=256
batch_per_epoch = int(dataset.shape[0] / n_batch)
# pour chaque epoch
for i in range(n_epochs):
  # pour chaque batch
  for j in range(batch_per_epoch):

    # étape 1 : on entraine le réseau discriminateur

    # on créé un batch d'échantillon à partir d'images réelles et générées
    X_real, y_real = generate_real_samples(dataset, int(n_batch/2))
    X_fake, y_fake = generate_fake_samples(generator_model, latent_dim, int(n_batch/2))
    X, y = np.vstack((X_real, X_fake)), np.vstack((y_real, y_fake))

    # on entraine le réseau discriminateur
    discriminator_model.trainable=True
    discriminator_model.train_on_batch(X, y)

    # second batch pour entrainer le discriminateur
    X_real, y_real = generate_real_samples(dataset, int(n_batch/2))
    X_fake, y_fake = generate_fake_samples(generator_model, latent_dim, int(n_batch/2))
    X, y = np.vstack((X_real, X_fake)), np.vstack((y_real, y_fake))
    d_loss, _ = discriminator_model.train_on_batch(X, y)

    # étape 2 : on entraine le GAN entier (discriminateur figé)

    # on créé un batch d'échantillons à partir d'images générées
    X_gan = generate_latent_vectors(latent_dim, n_batch)
    y_gan = np.ones((n_batch, 1))

    # on fige le discriminateur et on effectue l'apprentissage
    discriminator_model.trainable=False
    g_loss = gan_model.train_on_batch(X_gan, y_gan)

    print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, batch_per_epoch, d_loss, g_loss))

  # à chaque fin d'epoch, on enregistre une image d'échantillons
  save_image(i)
```

à la fin de chaque batch, vous pourrez observer les images générées dans le répertoire /tmp. Vous pourrez ensuite augmenter le nombre d'epoch (comptez environ 1min par epoch).

N'oubliez pas d'arrêter vos sessions Colab après usage !