```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#%matplotlib inline
import keras
from keras.layers import Dense, Dropout, Input
from keras.models import Model,Sequential
from keras.datasets import mnist
from tqdm import tqdm
from keras.layers.advanced_activations import LeakyReLU
from tensorflow.keras.optimizers import Adam
```

```
def load_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = (x_train.astype(np.float32) - 127.5)/127.5

    # convert shape of x_train from (60000, 28, 28) to (60000, 784)
    # 784 columns per row
    x_train = x_train.reshape(60000, 784)
    return (x_train, y_train, x_test, y_test)
(X_train, y_train,X_test, y_test)=load_data()
print(X_train.shape)
```

```
(60000, 784)
```

We will use Adam optimizer as it is computationally efficient and has very little memory requirement. Adam is a combination of Adagrad and RMSprop.

```
def adam_optimizer():
    return Adam(lr=0.0002, beta_1=0.5)
```

We create Generator which uses MLP using simple dense layers activated by tanh

```
def create_generator():
    generator=Sequential()
    generator.add(Dense(units=256,input_dim=100))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=512))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=1024))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=784, activation='tanh'))
```

```
    generator.compile(loss='binary_crossentropy', optimizer=adam_optimizer())
    return generator
g=create_generator()
g.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 256)               25856
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               131584
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              525312
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 1024)              0
_________________________________________________________________
dense_3 (Dense)              (None, 784)               803600
=================================================================
Total params: 1,486,352
Trainable params: 1,486,352
Non-trainable params: 0
_________________________________________________________________
C:\anaconda3\envs\aaic\lib\site-packages\keras\optimizer_v2\optimizer_v2.py:356: User
  "The `lr` argument is deprecated, use `learning_rate` instead.")
```

We now create the Discriminator which is also MLP. Discriminator will take the input from real data which is of the size 784 and also the images generated from Generator.

```
def create_discriminator():
    discriminator=Sequential()
    discriminator.add(Dense(units=1024,input_dim=784))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))


    discriminator.add(Dense(units=512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(units=256))
    discriminator.add(LeakyReLU(0.2))

    discriminator.add(Dense(units=1, activation='sigmoid'))

    discriminator.compile(loss='binary_crossentropy', optimizer=adam_optimizer())
```

```
    discriminator.compile(loss='binary_crossentropy', optimizer=adam_optimizer())
    return discriminator
d =create_discriminator()
d.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_4 (Dense)              (None, 1024)              803840    
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 1024)              0         
_________________________________________________________________
dropout (Dropout)            (None, 1024)              0         
_________________________________________________________________
dense_5 (Dense)              (None, 512)               524800    
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 512)               0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_6 (Dense)              (None, 256)               131328    
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 256)               0         
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 257       
=================================================================
Total params: 1,460,225
Trainable params: 1,460,225
Non-trainable params: 0
_________________________________________________________________
```

We now create the GAN where we combine the Generator and Discriminator. When we train the generator we will freeze the Discriminator.

We will input the noised image of shape 100 units to the Generator. The output generated from the Generator will be fed to the Discriminator.

```
def create_gan(discriminator, generator):
    discriminator.trainable=False
    gan_input = Input(shape=(100,))
    x = generator(gan_input)
    gan_output= discriminator(x)
    gan= Model(inputs=gan_input, outputs=gan_output)
    gan.compile(loss='binary_crossentropy', optimizer='adam')
    return gan
gan = create_gan(d,g)
gan.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 100)]             0
_________________________________________________________________
sequential (Sequential)      (None, 784)               1486352
_________________________________________________________________
sequential_1 (Sequential)    (None, 1)                 1460225
=================================================================
Total params: 2,946,577
Trainable params: 1,486,352
Non-trainable params: 1,460,225
_________________________________________________________________
```

Before we start training the model, we will write a function plot_generated_images to plot the generated images. This way we can see how the images are generated. We save the generated images to file that we can view later

```
def plot_generated_images(epoch, generator, examples=100, dim=(10,10), figsize=(10,10)):
    noise= np.random.normal(loc=0, scale=1, size=[examples, 100])
    generated_images = generator.predict(noise)
    generated_images = generated_images.reshape(100,28,28)
    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig('gan_generated_image %d.png' %epoch)
```

We finally start to train GAN. We will first have the full code for training GAN and then break it step by step for understanding how the training happens

```
def training(epochs=1, batch_size=128):

    #Loading the data
    (X_train, y_train, X_test, y_test) = load_data()
    batch_count = X_train.shape[0] / batch_size

    # Creating GAN
    generator= create_generator()
    discriminator= create_discriminator()
    gan = create_gan(discriminator, generator)

    for e in range(1,epochs+1 ):
        print("Epoch %d" %e)
        for _ in tqdm(range(batch_size)):
        #generate  random noise as an input  to  initialize the  generator
            noise= np.random.normal(0,1, [batch_size, 100])
```

```
            # Generate fake MNIST images from noised input
            generated_images = generator.predict(noise)

            # Get a random set of  real images
            image_batch =X_train[np.random.randint(low=0,high=X_train.shape[0],size=batch_s

            #Construct different batches of  real and fake data
            X= np.concatenate([image_batch, generated_images])

            # Labels for generated and real data
            y_dis=np.zeros(2*batch_size)
            y_dis[:batch_size]=0.9

            #Pre train discriminator on  fake and real data  before starting the gan.
            discriminator.trainable=True
            discriminator.train_on_batch(X, y_dis)

            #Tricking the noised input of the Generator as real data
            noise= np.random.normal(0,1, [batch_size, 100])
            y_gen = np.ones(batch_size)

            # During the training of gan,
            # the weights of discriminator should be fixed.
            #We can enforce that by setting the trainable flag
            discriminator.trainable=False

            #training  the GAN by alternating the training of the Discriminator
            #and training the chained GAN model with Discriminator’s weights freezed.
            gan.train_on_batch(noise, y_gen)

        if e == 1 or e % 20 == 0:

            plot_generated_images(e, generator)
training(400,128)
```

```
Epoch 1
100%|██████████████████████████████████████████████████
Epoch 2
100%|██████████████████████████████████████████████████
Epoch 3
100%|██████████████████████████████████████████████████
Epoch 4
100%|██████████████████████████████████████████████████
Epoch 5
100%|██████████████████████████████████████████████████
Epoch 6
100%|██████████████████████████████████████████████████
Epoch 7
100%|██████████████████████████████████████████████████
Epoch 8
100%|██████████████████████████████████████████████████
```

```
100%|
Epoch 9
100%|
Epoch 10
100%|
Epoch 11
100%|
Epoch 12
100%|
Epoch 13
100%|
Epoch 14
100%|
Epoch 15
100%|
Epoch 16
100%|
Epoch 17
100%|
Epoch 18
100%|
Epoch 19
100%|
Epoch 20
100%|
Epoch 21
100%|
Epoch 22
100%|
Epoch 23
100%|
Epoch 24
100%|
Epoch 25
100%|
Epoch 26
100%|
Epoch 27
100%|
Epoch 28
100%|
```