Generative Adversarial Networks

Praneeth Vonteddu and Ankit Kumar Gupta

***Abstract*— Generative Adversarial Networks(GAN) have became popular because of their abilty to generate photo-realsitc images.In this project we have implemented some of the GAN architectures starting from as simple as using GAN to generate datapoints from a function to generating MNIST handwritten digits using a Convolutional GAN.**

I. INTRODUCTION

Generative Adversarial Networks are a class of generative models which use an adversarial process to train both of it's networks *Discriminator*(D) and *Generator*(G) simultaneously.Discriminator and Generator compete against each other in a mini-max zero sum game and hence the name 'adversarial'.We can model these two networks as multilayer perceptrons.

These two networks have opposite goals.The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistinguishable from the genuine articles

II. THEORY

Let p_g be the generator's distribution over data x,and $p_z(z)$ be the noise variable's distribution and let $G(z;\theta_g)$ be the multilayer perceptron(Generator) with parameters θ_g that maps noise space to data space outputting a fake sample and let $D(x;\theta_d)$ be the other multilayer perceptron(Discriminator) with parameters θ_d that takes real sample from $p_{data}(x)$ and fake samples(generated by the generator) as inputs and outputs the probability that a sample is real.

We train D to maximize the probability of assigning the correct label to both the real samples and fake samples. We simultaneously train G to minimize $\log(1 - D(G(z)))$: We can think of D and G playing a two-player mini-max game with value function $V(G, D)$:

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

Early in the learning phase when G is poor D rejects fake samples with high confidence , this means $\log(1 - D(G(z)))$ saturates.Hence we train G to maximize $\log(D(G(z)))$ instead of minimizing $\log(1$ - $D(G(z)))$ for better learning. In an ideal scenario we have at the Nash Equilibrium of this game p_g = p_{data} and $D(x) = 1/2$ which means that the discriminator can no longer differentiate between real and fake samples and the generator has perfectly learnt how to generate samples from the data distribution.

III. ALGORITHM

In each iteration we do,

1) Sample minibatch of m noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from noise distribution $p_z(z)$.
2) Sample minibatch of m examples $\{x^{(1)}, ..., x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
3) Update the discriminator by stochastic gradient ascent:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

4) Again sample minibatch of m noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from noise distribution $p_z(z)$.
5) Update the discriminator by stochastic gradient descent:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z^{(i)})))]$$

6) Repeat the above steps until convergence

IV. Experiments

A. Univariate GAN

In this section we demonstrate that a GAN can generate data from a single variable function efficiently.

consider 2 univariate functions $y = \sin(x)$ and $y = sinc(x)$

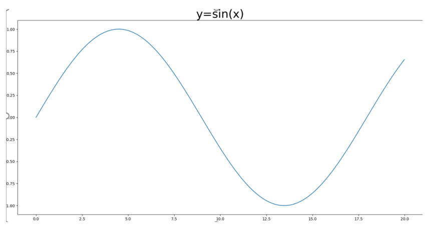

Fig. 1. Sin function

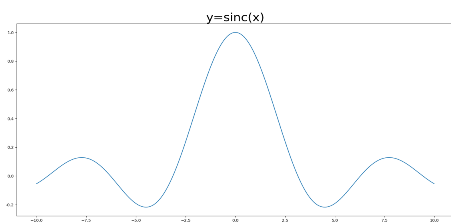

Fig. 2. Sinc function

Our goal is to sample points from this function.

1) Discriminator Architecture: The discriminator is a Multi Layer Perceptron with 1 hidden layer having 25 hidden nodes. It takes a point in 2-d space and gives out the probability that the point is real or fake.

2) Generator Architecture: The generator is also a Multi Layer Perceptron with 1 hidden layer having 15 hidden nodes. It takes a point in n-d space and generates a point in 2-d space which is the fake sample that is fed as an input to the discriminator.We experimented with the value of n and took it as 5.

3) Results: After training the GAN we visualize the output ,We can see that GAN can sample from univariate functions pretty well.

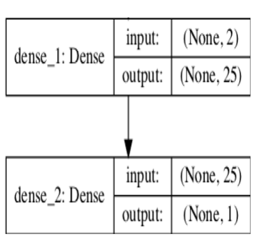

Fig. 3. Discriminator

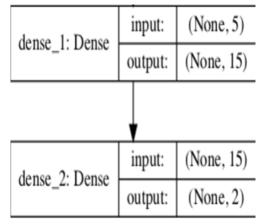

Fig. 4. Generator

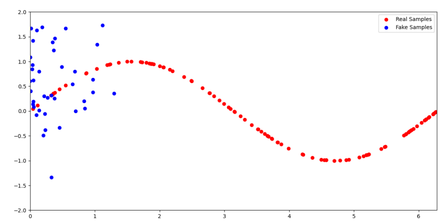

Fig. 5. Learning Sin Function Before Training

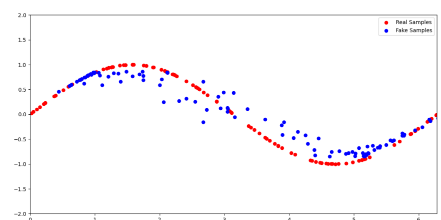

Fig. 6. Learning Sin Function After Training

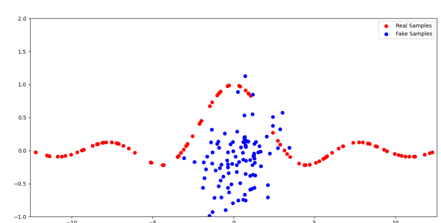

Fig. 7. Learning Sinc Function Before Training

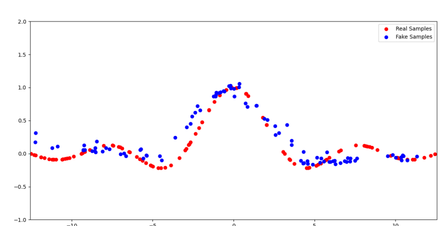

Fig. 8. Learning Sinc Function After Training

B. GAN for genrating images from MNIST dataset

In this section we use slightly complicated networks using convolutional layers as our models as we are dealing with images.

1) MNIST dataset: The MNIST handwritten digits dataset contains 60000 gray images each of size 28×28.

Fig. 9. MNIST Handwritten digits dataset

2) Discriminator Architecture: The discriminator contains 2 convolutional layers with leaky-ReLU activation function and a dense layer at the output connected to a single output node.It takes an image of size 28×28 and outputs a number which can be interpreted as the probability that an image is real rather than generated.

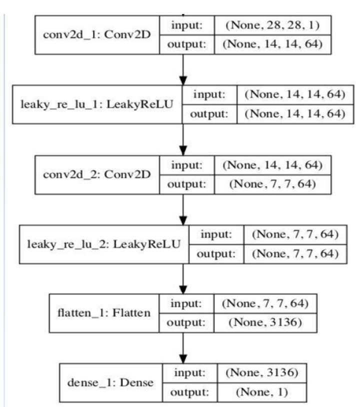

Fig. 10. Discrminator for MNIST GAN

3) Generator Architecture: The generator contains a dense layer and 2 transposed convolutional layers with leaky-ReLU activation function and a convolutional layer at the output.It takes a 100 dimensional normal random vector as input and generates a fake image which is fed to the discriminator as an input.

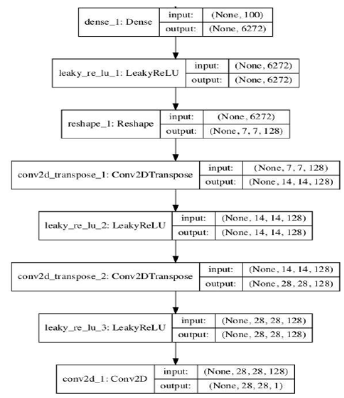

Fig. 11. Generator for MNIST GAN

4) Results: The ouput of the MNIST GAN is visualized below , we can see after 100 epochs of training the output resembling the real MNIST dataset.

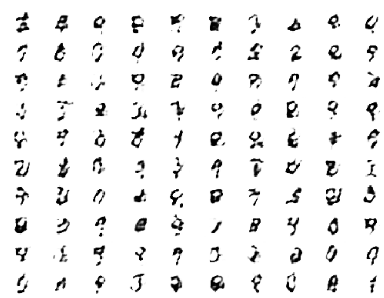

Fig. 12. Output after 10 epochs

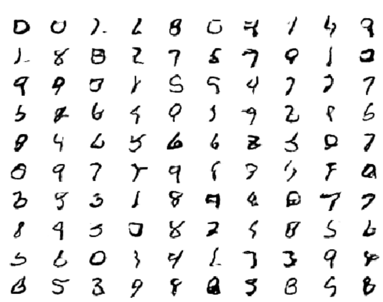

Fig. 13. Output after 100 epochs

V. EVALUATION OF GAN OUTPUT

We observe, as the random points generated in every iteration(for training the generator) are not correlated with each other.As a result of which,the output generated at every iteration does not have similarity with previous ones.

Further,unlike other deep learning neural network models that are trained with a loss function until convergence, a GAN generator model is trained using a discriminator model that learns to classify images as real or generated. Both are trained to maintain an equilibrium.

Hence,as such,there is no way to objectively assess the progress of the training and the relative or absolute quality of the model.This still remains a topic to be actively researched upon.

So,Manual inspection of generated images still remains a good starting point, we can have qualitative studies to judge the perceptual quality of generated images or quantitative measusres like SSIM or PSNR for image quality.

However,nowadays,a parameter called " Inception Score ", is generally looked upon to evaluate GANs, but it requires a pre-trained deep learning neural network model for image classification to classify the generated images, which is beyond the scope of this project.

VI. CONCLUSIONS AND EXTENSIONS

Conditional GAN: We must observe that a trained generator will generate random data from real data distribution,Instead we can create a conditional generative model by conditioning on the class labels.For example we can generate a particular digit from MNIST digit unlike a vanilla GAN which generates a random MNIST digit.
We conclude that this method of adversarial learning can be really useful for generating fake but perceptually close to real images or audio.

REFERENCES

[1] Ian J. Goodfellow, Jean Pouget-Abadie , Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair , Aaron Courville, Yoshua Bengio. Generative Adversarial Nets.

[2] Alec Radford, Luke Metz, Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

[3] https://machinelearningmastery.com/how-to-evaluate-generative-adversarial-networks/

APPENDIX

VII. PYTHON CODES

The training process and the python codes can also be viewed at this link.

Univariate GAN

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import
    train_test_split
from keras.utils import to_categorical #for one hot
     encoding

def sigmoid(x):
    return(1/(1+np.exp(-x)))

def sigmoiddash(a):
    return a*(1-a)

def relu(x):

    x[x<0]=0
    return x

def reludash(x):

    x[x>0]=1
    x[x<0]=0
    return x

def gen_layers_size(Z,X,h):
    n_x=np.shape(Z)[0]
    n_h=h
    n_y=np.shape(X)[0]
    return(n_x,n_h,n_y)


def dis_layers_size(X,ou,h):
    n_x=np.shape(X)[0]
    n_h=h
    n_y=ou
    return(n_x,n_h,n_y)


def init_gen_params(n_x,n_h,n_y):

    GW1=np.random.randn(n_h,n_x)*0.01
    Gb1=np.zeros((n_h,1))
    GW2=np.random.randn(n_y,n_h)*0.01
    Gb2=np.zeros((n_y,1))

    gen_params={'GW1':GW1,'Gb1':Gb1,'GW2':GW2,'Gb2'
    :Gb2}
    return gen_params


def init_dis_params(n_x,n_h,n_y):

    DW1=np.random.randn(n_h,n_x)*0.01
    Db1=np.zeros((n_h,1))
    DW2=np.random.randn(n_y,n_h)*0.01
    Db2=np.zeros((n_y,1))

    dis_params={'DW1':DW1,'Db1':Db1,'DW2':DW2,'Db2'
    :Db2}
    return dis_params


def gen_feed_forward(Gen_params,Z):

    W1=Gen_params['GW1']
    b1=Gen_params['Gb1']
```

```python
    W2=Gen_params['GW2']
    b2=Gen_params['Gb2']

    Z1=np.dot(W1,Z).reshape(np.shape(b1))+b1
    A1=relu(Z1)
    Z2=np.dot(W2,A1).reshape(np.shape(b2))+b2
    A2=relu(Z2)
    gen_cache={'Z1':Z1,'Z2':Z2,'A1':A1,'A2':A2}

    return A2,gen_cache


def dis_feed_forward(Dis_params,X):

    W3=Dis_params['DW1']
    b3=Dis_params['Db1']
    W4=Dis_params['DW2']
    b4=Dis_params['Db2']

    Z3=np.dot(W3,X).reshape(np.shape(b3))+b3
    A3=relu(Z3)
    Z4=np.dot(W4,A3).reshape(np.shape(b4))+b4
    A4=sigmoid(Z4)
    dis_cache={'Z3':Z3,'Z4':Z4,'A3':A3,'A4':A4}

    return A4,dis_cache

def dis_backprop(dis_params,dis_cache,Y,X):

    W3=dis_params['DW1']
    b3=dis_params['Db1']
    W4=dis_params['DW2']
    b4=dis_params['Db2']

    Z3=dis_cache['Z3']
    A3=dis_cache['A3']
    Z4=dis_cache['Z4']
    A4=dis_cache['A4']

    dA4=A4-Y.reshape(np.shape(A4))
    dW4=dA4*A3.T
    db4=dA4

    r3=reludash(Z3)
    local=r3*W4.T

    dW3=dA4*local*X.T
    db3=dA4*local


    dis_dparams={'dDW1':dW3,'dDb1':db3,'dDW2':dW4,'dDb2':db4}
    return dis_dparams

def update_dis_params(dis_params,dis_dparams,lr):

    W3=dis_params['DW1']
    b3=dis_params['Db1']
    W4=dis_params['DW2']
    b4=dis_params['Db2']

    dW3=dis_dparams['dDW1']
    db3=dis_dparams['dDb1']
    dW4=dis_dparams['dDW2']
    db4=dis_dparams['dDb2']

    dis_params['DW1']=W3-(lr)*dW3
    dis_params['Db1']=b3-(lr)*db3.reshape(np.shape(b3))
    dis_params['DW2']=W4-(lr)*dW4
    dis_params['Db2']=b4-(lr)*db4.reshape(np.shape(b4))

    return dis_params


def gen_predict(gen_params,Z,n):
    P=np.zeros((2,n))
    for j in range(n):
        A2,cache=gen_feed_forward(gen_params,Z[:,j])
        P[:,j]=A2.T

    return P

def dis_predict(dis_params,X):
    P=np.random.rand(np.shape(X)[1])
    for j in range(np.shape(X)[1]):
        A2,cache=dis_feed_forward(dis_params,X[:,j])
        p=1 if A2>0.5 else 0
        P[j]=p

    return P


def gen_backprop(dis_params,dis_cache,gen_params,gen_cache,Y,X):


    W3=dis_params['DW1']
    b3=dis_params['Db1']
    W4=dis_params['DW2']
    b4=dis_params['Db2']

    Z3=dis_cache['Z3']
    A3=dis_cache['A3']
    Z4=dis_cache['Z4']
    A4=dis_cache['A4']


    W1=gen_params['GW1']
    b1=gen_params['Gb1']
    W2=gen_params['GW2']
    b2=gen_params['Gb2']

    Z1=gen_cache['Z1']
    A1=gen_cache['A1']
    Z2=gen_cache['Z2']
    A2=gen_cache['A2']

    dZ4=A4-Y.reshape(np.shape(A4))
    dW4=dZ4*A3.T
    db4=dZ4

    r3=reludash(Z3)
    local=r3*W4.T

    dW3=dZ4*local*A2.T
    db3=dZ4*local

    r2=reludash(Z2)
    local2=r2*W3.T

    dW2=dZ4*local*local2*A1.T
    db2=dZ4*local*local2

    r1=reludash(Z1)
    local3=r1*W2.T

    dW1=dZ4*local*local2*local3*X.T
    db1=dZ4*local*local2*local3


    gen_dparams={'dGW1':dW1,'dGb1':db1,'dGW2':dW2,'dGb2':db2}
    return gen_dparams

def update_gen_params(gen_params,gen_dparams,lr):

```

```
    W1=gen_params['GW1']
    b1=gen_params['Gb1']
    W2=gen_params['GW2']
    b2=gen_params['Gb2']

    dW1=gen_dparams['dGW1']
    db1=gen_dparams['dGb1']
    dW2=gen_dparams['dGW2']
    db2=gen_dparams['dGb2']

    gen_params['GW1']=W1-(lr)*dW1
    gen_params['Gb1']=b1-(lr)*db1.reshape(np.shape(b1))
    gen_params['GW2']=W2-(lr)*dW2
    gen_params['Gb2']=b2-(lr)*db2.reshape(np.shape(b2))

    return gen_params

def real_samples(n):

    X1 = 2*np.pi*np.random.rand(n) - np.pi
    X2 = np.sin(X1)
    X1 = X1.reshape(n, 1)
    X2 = X2.reshape(n, 1)
    X = np.hstack((X1, X2)).T
    y = np.ones((n, 1)).T
    return X, y

def latent_points(latent_dim, n):

    x_input = np.random.randn(latent_dim * n)
    x_input = x_input.reshape(n, latent_dim).T
    return x_input

def fake_samples(gen_params, latent_dim, n):

    Z = latent_points(latent_dim, n)
    X = gen_predict(gen_params,Z,n)
    y = np.zeros((n, 1)).T
    return X, y




def main():

    n=100
    latent_dim=5
    lr=0.1

    x_real,y_real=real_samples(n)
    Z=latent_points(latent_dim,n)

    n_1,n_2,n_3=gen_layers_size(Z,x_real,25)
    n_4,n_5,n_6=dis_layers_size(x_real,1,15)

    gen_params=init_gen_params(n_1,n_2,n_3)
    dis_params=init_dis_params(n_4,n_5,n_6)


    for i in range(1):

        x_real,y_real=real_samples(n)
        real=[x_real[0,:],x_real[1,:]]

        Z=latent_points(latent_dim,n)

        x_fake,y_fake=fake_samples(gen_params,latent_dim, n)
        fake=[x_fake[0,:],x_fake[1,:]]

        X_dis=np.concatenate((real,fake),axis=1)
        y_dis=np.hstack((y_real,y_fake))

        for j in range(np.shape(X_dis)[1]):

            A4,dis_cache=dis_feed_forward(dis_params,X_dis[:,j])

            dis_dparams=dis_backprop(dis_params,dis_cache,y_dis[:,j],X_dis[:,j])
            dW1=dis_dparams['dDW1']
            db1=dis_dparams['dDb1']
            dW2=dis_dparams['dDW2']
            db2=dis_dparams['dDb2']

            dis_params=update_dis_params(dis_params,dis_dparams,lr)


        x_gan=latent_points(latent_dim,n)
        y_gan=np.ones((1,n))

        for j in range(np.shape(x_gan)[1]):

            A2,gen_cache=gen_feed_forward(gen_params,x_gan[:,j])
            A4,dis_cache=dis_feed_forward(dis_params,X_dis[:,j])

            gen_dparams=gen_backprop(dis_params,dis_cache,gen_params,gen_cache,y_gan[:,j],x_gan[:,j])
            gen_params=update_gen_params(gen_params,gen_dparams,lr)







    print(dis_params)

    x_real,y_real=real_samples(n)

    p=dis_predict(dis_params,x_real)


    print(y_real)
    print(p)

    Z=latent_points(latent_dim,n)

    x_fake,y_fake=fake_samples(gen_params,latent_dim, n)

    p=dis_predict(dis_params,x_fake)
    print(y_fake)
    print(p)
```

Univariate GAN using Keras

```
import numpy as np
from numpy.random import *
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense



def Discriminator(n_inputs=2):

    model =Sequential()
    model.add(Dense(25,activation='relu', kernel_initializer='he_uniform',input_dim=n_inputs))
    model.add(Dense(1,activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam',metrics=['accuracy'])

    return model



def Generator(latent_dim,n_outputs=2):

    model =Sequential()
    model.add(Dense(15,activation='relu', kernel_initializer='he_uniform',input_dim=latent_dim))
    model.add(Dense(n_outputs,activation='linear'))
    return model


def GAN(generator,discriminator):

    discriminator.trainable=False

    model=Sequential()
    model.add(generator)
    model.add(discriminator)
    model.compile(loss='binary_crossentropy', optimizer='adam')
    return model


def real_samples(n):


    X1 = 2*np.pi*rand(n)
    X2 = np.sin(X1)
    X1 = X1.reshape(n, 1)
    X2 = X2.reshape(n, 1)
    X = np.hstack((X1, X2))
    y = np.ones((n, 1))
    return X, y

def latent_points(latent_dim, n):

    x_input = np.pi+randn(latent_dim * n)
    x_input = x_input.reshape(n, latent_dim)
    return x_input

def fake_samples(generator, latent_dim, n):

    x_input = latent_points(latent_dim, n)
    X = generator.predict(x_input)
    y = np.zeros((n, 1))
    return X, y


def summarize_performance(epoch, generator, discriminator, latent_dim, n=100):

    x_real, y_real = real_samples(n)
    _, acc_real = discriminator.evaluate(x_real, y_real, verbose=0)

    x_fake, y_fake = fake_samples(generator, latent_dim, n)
    _, acc_fake = discriminator.evaluate(x_fake, y_fake, verbose=0)

    print(epoch, acc_real, acc_fake)
    plt.scatter(x_real[:, 0], x_real[:, 1], color='red')
    plt.scatter(x_fake[:, 0], x_fake[:, 1], color='blue')
    plt.legend(['Real Samples','Fake Samples'])
    plt.xlim([0,2*np.pi])
    plt.ylim([-2,2])
    plt.show()


def train(g_model, d_model, gan_model, latent_dim, n_epochs=50000, n_batch=128,
          n_eval=2000):
    half_batch = int(n_batch / 2)

    for i in range(n_epochs):

        x_real, y_real = real_samples(half_batch)
        x_fake, y_fake = fake_samples(g_model, latent_dim, half_batch)
        d_model.train_on_batch(x_real, y_real)
        d_model.train_on_batch(x_fake, y_fake)
        x_gan = latent_points(latent_dim, n_batch)
        y_gan = np.ones((n_batch, 1))

        gan_model.train_on_batch(x_gan, y_gan)
        if (i+1) % n_eval == 0:
                summarize_performance(i, g_model, d_model, latent_dim)


latent_dim = 5

discriminator = Discriminator()
generator = Generator(latent_dim)
gan_model = GAN(generator, discriminator)
train(generator, discriminator, gan_model, latent_dim)
```

MNIST GAN

```
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import vstack
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from matplotlib import pyplot

# Discriminator model
def Discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding=’same’, input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding=’same’))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation=’sigmoid’))
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=’binary_crossentropy’, optimizer=opt, metrics=[’accuracy’])
  return model

# Generator model
def Generator(latent_dim):
  model = Sequential()
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding=’same’))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding=’same’))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation=’sigmoid’, padding=’same’))
  return model

# Combining generator and discriminator model
def GAN(generator, discriminator):
  discriminator.trainable = False
  model = Sequential()
  model.add(generator)
  model.add(generator)
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=’binary_crossentropy’, optimizer=opt)
  return model

def load_real_samples():
  (trainX, _), (_, _) = load_data()
  X = expand_dims(trainX, axis=-1)
  X = X.astype(’float32’)
  X = X / 255.0
  return X

def generate_real_samples(dataset, n_samples):
  ix = randint(0, dataset.shape[0], n_samples)
  X = dataset[ix]
  y = ones((n_samples, 1))
  return X, y

def generate_latent_points(latent_dim, n_samples):
  x_input = randn(latent_dim * n_samples)
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

def generate_fake_samples(generator, latent_dim, n_samples):
  x_input = generate_latent_points(latent_dim, n_samples)
  X = generator.predict(x_input)
  y = zeros((n_samples, 1))
  return X, y

def save_plot(examples, epoch, n=10):
  for i in range(n * n):
    pyplot.subplot(n, n, 1 + i)
    pyplot.axis(’off’)
    pyplot.imshow(examples[i, :, :, 0], cmap=’gray_r’)
  filename = ’generated_plot_e%03d.png’ % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()

def summarize_performance(epoch, g_model, discriminator, dataset, latent_dim, n_samples=100):
  X_real, y_real = generate_real_samples(dataset, n_samples)
  _, acc_real = discriminator.evaluate(X_real, y_real, verbose=0)
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  _, acc_fake = discriminator.evaluate(x_fake, y_fake, verbose=0)
  print(’>Accuracy real: %.0f%%, fake: %.0f%%’ % (acc_real*100, acc_fake*100))
  save_plot(x_fake, epoch)
  filename = ’generator_model_%03d.h5’ % (epoch + 1)
  g_model.save(filename)

def train(generator, discriminator, gan, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)

  for i in range(n_epochs):
    for j in range(bat_per_epo):
      X_real, y_real = generate_real_samples(dataset, half_batch)
      X_fake, y_fake = generate_fake_samples(generator, latent_dim, half_batch)
      X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))

      d_loss, _ = discriminator.train_on_batch(X, y)

      X_gan = generate_latent_points(latent_dim, n_batch)
      y_gan = ones((n_batch, 1))
      g_loss = gan.train_on_batch(X_gan, y_gan)

      print(’>%d, %d/%d, d=%.3f, g=%.3f’ % (i+1, j+1, bat_per_epo, d_loss, g_loss))
    if (i+1) % 10 == 0:
      summarize_performance(i, generator, discriminator, dataset, latent_dim)


latent_dim = 100
```

```
d_model = Discriminator()
g_model = Generator(latent_dim)
gan_model = GAN(g_model, d_model)
dataset = load_real_samples()
train(g_model, d_model, gan_model, dataset,
    latent_dim)
```

Visualizing the output of a trained MNIST GAN

```
#After Training for visualizing output

from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot

def generate_latent_points(latent_dim, n_samples):
  x_input = randn(latent_dim * n_samples)
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

def save_plot(examples, n):
  for i in range(n * n):
    pyplot.subplot(n, n, 1 + i)
    pyplot.axis('off')
    pyplot.imshow(examples[i, :, :, 0], cmap='
    gray_r')
  pyplot.show()

model = load_model('generator_model_100.h5')
latent_points = generate_latent_points(100, 25)
X = model.predict(latent_points)
save_plot(X, 5)
```