

In [9]:

*#Importing required libraries.*

**from keras.datasets import** mnist

**from keras.utils import** np\_utils

**from keras.models import** Sequential, Model

**from keras.layers import** Input, Dense, Activation, Flatten, Reshape

**from keras.layers.convolutional import** Conv2D, Conv2DTranspose, UpSampling2D, Convolution2D

**from keras.layers.normalization import** BatchNormalization **from keras.layers.advanced\_activations import** LeakyReLU **from keras.optimizers import** Adam **import numpy as np**

**import matplotlib.pyplot as plt**

**import random**

**from tqdm import** tqdm\_notebook

(60000, 28, 28)

In [ ]:

* *Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10, 000 images.*

(X\_train, Y\_train), (X\_test, Y\_test) = mnist.load\_data() print(X\_train.shape)

In [10]:

* *Preprocessing : This step is crucial for this type of GAN , because here the images fed to discriminator must have a size with channels included.*

*#Therefore we reshape the b&w images with 1 channel to (28,28,1).*

X\_train = X\_train.reshape(60000, 28, 28, 1)

X\_test = X\_test.reshape(10000, 28, 28, 1)

X\_train = X\_train.astype('float32')/255

X\_test = X\_test.astype('float32')/255

In [17]:

*#Using images of 0 only so to make datset smaller and training time faster.*

X\_train= X\_train[Y\_train==0]

In [11]:

* *Set the dimensions of the noise* z\_dim = 100

In [34]:

nch = 200

In [32]:

* *Here we define the architecture of GAN which consists of both "generator" and "discrimi nator".*

*#We define architecture of Generator which is fully-convolutional network. It accepts noi se as input and generates am image out of it.*

*#Generator learns when gradients are backpropagated through the network just after discri minator clasifies any image real or fake.*

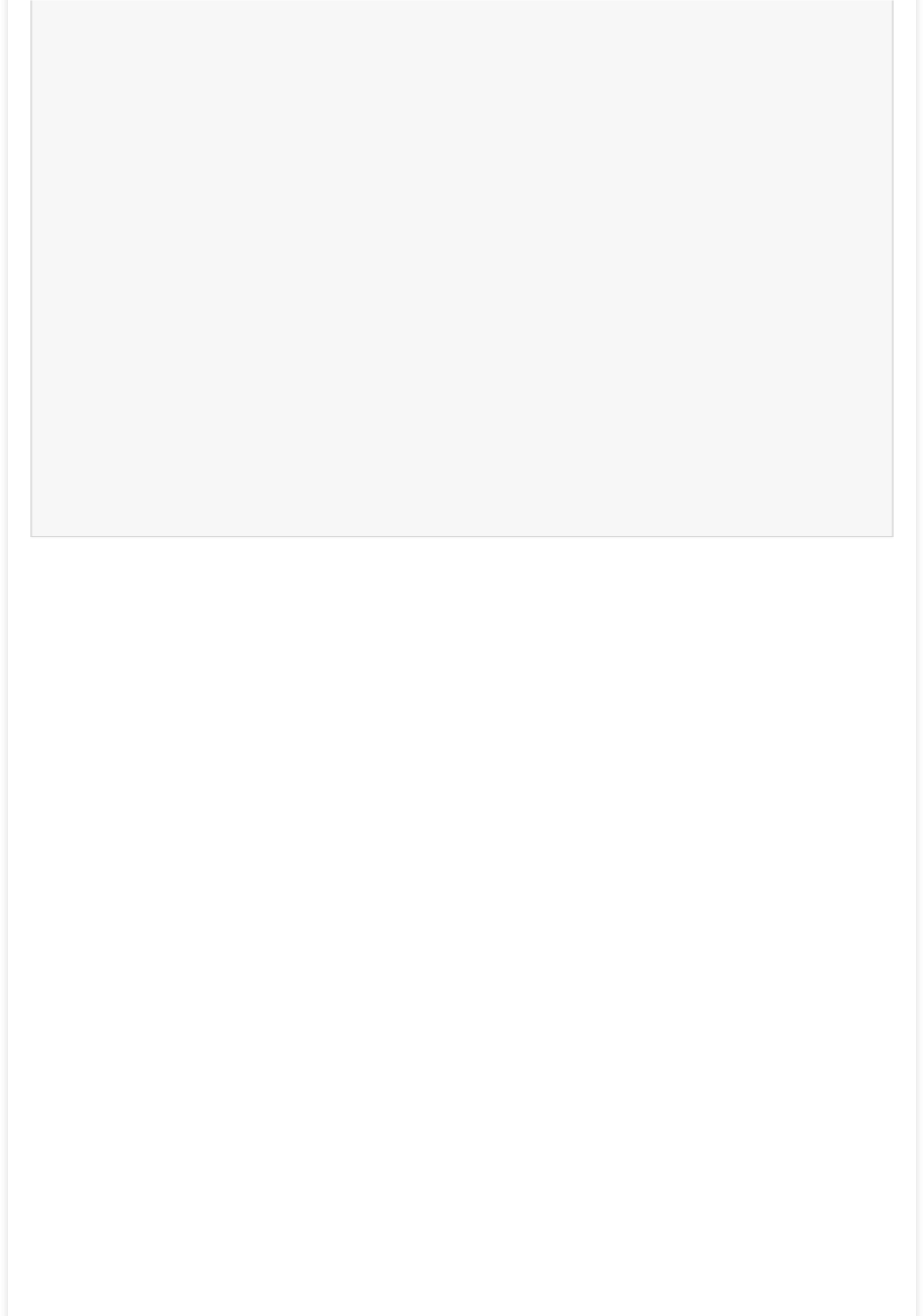
*#A special type of convolutional layer is being used here which increases the dimension o f input image/array by using transpose convolution notion.*

*#Because network discriminates between real and generated image , therefore this uses "bi nary\_crossentropy" as loss.*

*#First encoding is fed to generator which generates an image from it and then subsequentl y the generated image is fed into discriminator for it to classify*

* *image as real or fake.*

*#Thus in the whole architecture , they both fight against each other. And thus make each other learn more everytime.*



adam = Adam(lr=0.0002, beta\_1=0.5)

g = Sequential()

g.add(Dense(7\*7\*112, input\_dim=z\_dim))

g.add(Reshape((7, 7, 112)))

g.add(BatchNormalization())

g.add(Activation(LeakyReLU(alpha=0.2)))

g.add(Conv2DTranspose(56, 5, strides=2, padding='same'))

g.add(BatchNormalization())

g.add(Activation(LeakyReLU(alpha=0.2)))

g.add(Conv2DTranspose(1, 5, strides=2, padding='same', activation='sigmoid'))

g.compile(loss='binary\_crossentropy', optimizer=adam, metrics=['accuracy'])

g.summary()

d = Sequential()

d.add(Conv2D(56, 5, strides=2, padding='same', input\_shape=(28, 28, 1), activation=Leaky ReLU(alpha=0.2)))

d.add(Conv2D(112, 5, strides=2, padding='same'))

g.add(BatchNormalization())

g.add(Activation(LeakyReLU(alpha=0.2)))

d.add(Conv2D(224, 5, strides=2, padding='same'))

g.add(Activation(LeakyReLU(alpha=0.2)))

d.add(Flatten())

d.add(Dense(112, activation=LeakyReLU(alpha=0.2)))

d.add(Dense(1, activation='sigmoid'))

d.compile(loss='binary\_crossentropy', optimizer=adam, metrics=['accuracy'])

d.summary()

Model: "sequential\_2"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_5 (Dense) (None, 5488) 554288

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

reshape\_3 (Reshape) (None, 7, 7, 112) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| batch\_normalization\_7 (Batch (None, 7, 7, 112) | 448 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_9 (Activation) (None, 7, 7, 112) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| conv2d\_transpose\_2 (Conv2DTr (None, 14, 14, 56) | 156856 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| batch\_normalization\_8 (Batch (None, 14, 14, 56) | 224 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_10 (Activation) (None, 14, 14, 56) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| conv2d\_transpose\_3 (Conv2DTr (None, 28, 28, 1) | 1401 |

=================================================================

Total params: 713,217

Trainable params: 712,881

Non-trainable params: 336

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Model: "sequential\_3"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_6 (Conv2D) (None, 14, 14, 56) 1456

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_7 (Conv2D) (None, 7, 7, 112) 156912

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_8 (Conv2D) (None, 4, 4, 224) 627424

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_1 (Flatten) (None, 3584) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_6 (Dense) (None, 112) 401520

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

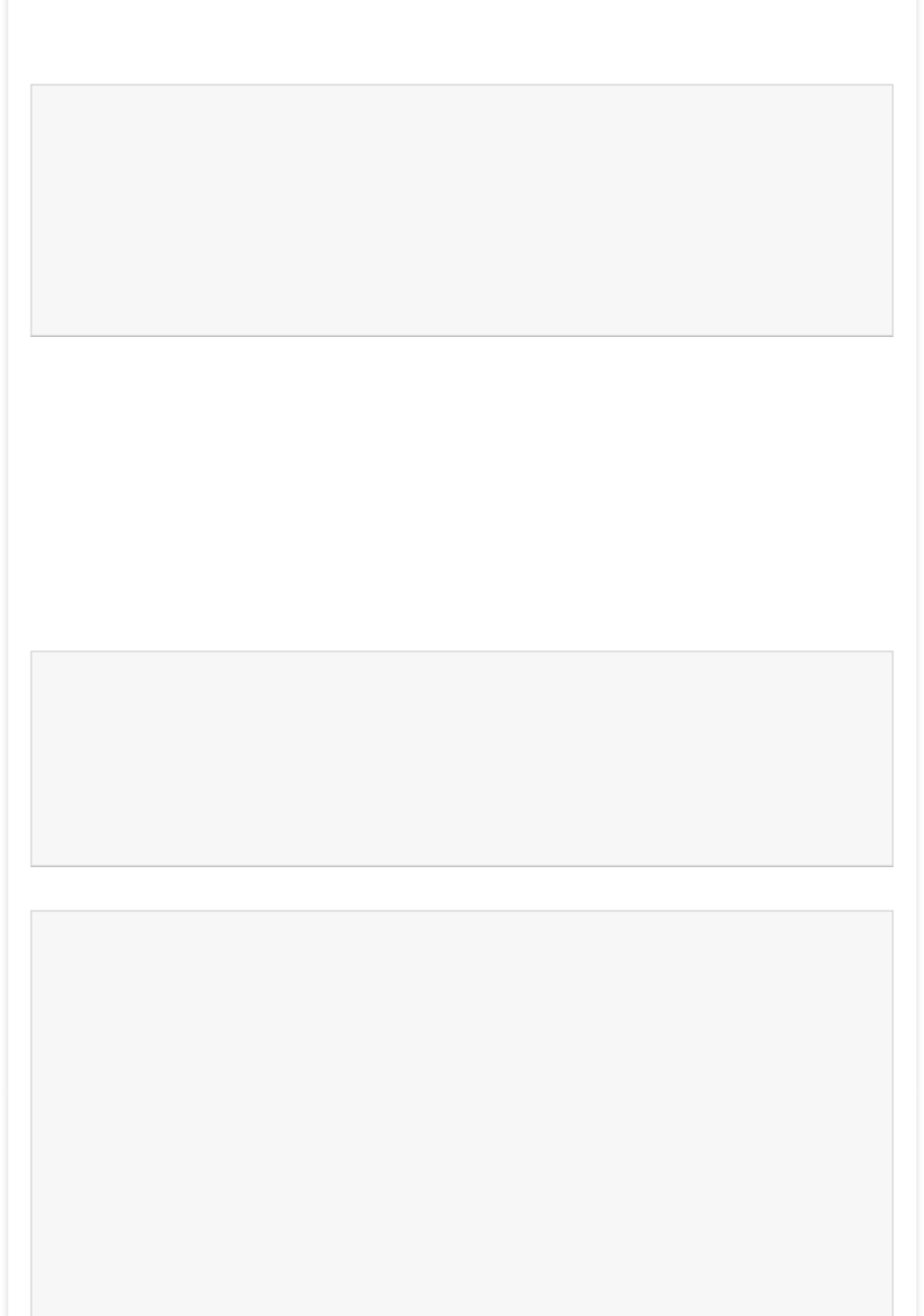
dense\_7 (Dense) (None, 1) 113

=================================================================

Total params: 1,187,425

Trainable params: 1,187,425

Non-trainable params: 0



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [33]:

*#We first make discriminator non-trainable so it just classify images simply and do not l earn anything from the data fed to it.*

*#And we set the complete loss-fnunction and optimizer. Advanced architectures uses variou*

*s kind of loss so to make it more*

*# efficient.*

d.trainable = **False**

inputs = Input(shape=(z\_dim, ))

hidden = g(inputs)

output = d(hidden)

gan = Model(inputs, output)

gan.compile(loss='binary\_crossentropy', optimizer=adam, metrics=['accuracy'])

gan.summary()

Model: "functional\_5"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_5 (InputLayer) [(None, 100)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

sequential\_2 (Sequential) (None, 28, 28, 1) 713221

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

sequential\_3 (Sequential) (None, 1) 1187425

=================================================================

Total params: 1,900,646

Trainable params: 712,883

Non-trainable params: 1,187,763

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [13]:

*#This is train function which trains the whole architecture in one.*

*#First an random noise is generated and is fed to generator "g" which generates some nois y image from it as the generator is not yet trained.*

*#The generated noisy image is then concatenated with real images and all of them are fed into discriminator. Discriminator is tricked to believe that they*

*#real images as we give target as "0". It then learns somehting about the real images. #Then comes the generator's training. We simply generate some more random noise, set targ et as "1" so to make the "GAN" believe that they are real images.*

*#Just then we make discriminator non-trainable and feed this data into whole "GAN" archit ecture. Thus the generator learns when the loop completes and*

*# the gradients flow back assuming that all of the images given were real images.*

In [18]:

* *Set up a vector (dict) to store the losses* samples = []

**def** train(epochs=1, plt\_frq=1, BATCH\_ SIZE=128):batchCount = int(X\_train.shape[0] / BATCH\_SIZE) print('Epochs:', epochs)

print('Batch size:', BATCH\_SIZE)

print('Batches per epoch:', batchCount)

**for** e **in** tqdm\_notebook(range(1, epochs+1)):

**if** e== 1 **or** e%plt\_frq== 0:

print('-'\*15, 'Epoch **%d**' % e, '-'\*15)

**for** \_ **in** range(batchCount): *# tqdm\_notebook(range(batchCount), leave=False):*

*# Create a batch by drawing random index numbers from the training set*

image\_batch = X\_train[np.random.randint(0, X\_train.shape[0], size=BATCH\_SIZE

)]

image\_batch = image\_batch.reshape(image\_batch.shape[0], image\_batch.shape[1]

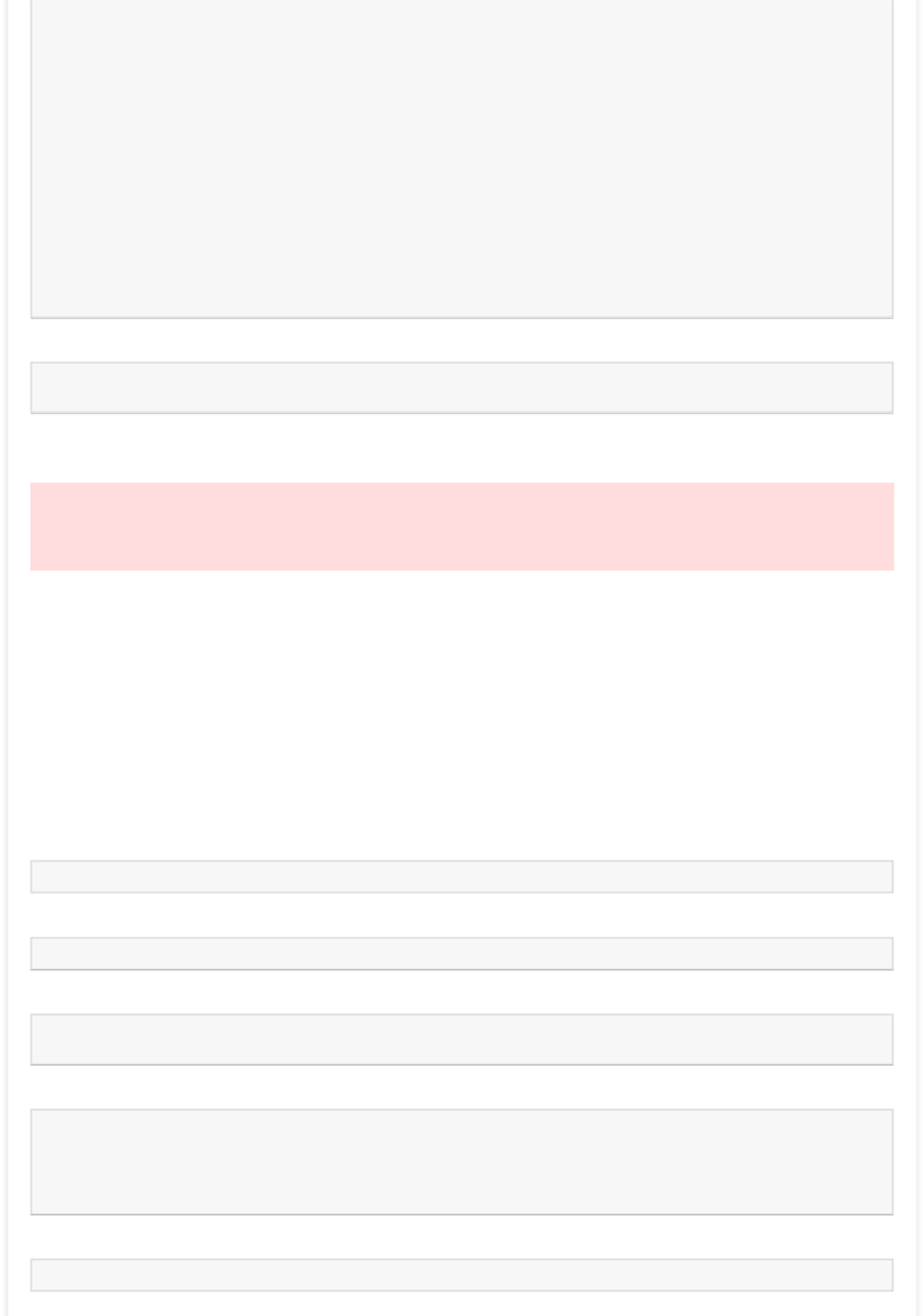
* image\_batch.shape[2], 1)
  + *Create noise vectors for the generator*

noise = np.random.normal(0, 1, size=(BATCH\_SIZE, z\_dim))

* *Generate the images from the noise* generated\_images = g.predict(noise) samples.append(generated\_images)

X = np.concatenate((image\_batch, generated\_images))

* *Create labels*



* = np.zeros(2\*BATCH\_SIZE)

y[:BATCH\_SIZE] = 0.9 *# One-sided label smoothing*

* *Train discriminator on generated images* d.trainable = **True**

d\_loss = d.train\_on\_batch(X, y)

* *Train generator*

noise = np.random.normal(0, 1, size=(BATCH\_SIZE, z\_dim))

y2 = np.ones(BATCH\_SIZE)

d.trainable = **False**

g\_loss = gan.train\_on\_batch(noise, y2)

In [21]:

*#Training 50 epochs.*

train(epochs=200, plt\_frq=20, BATCH\_SIZE=128)

Epochs: 200

Batch size: 128

Batches per epoch: 46

/usr/local/lib/python3.6/dist-packages/ipykernel\_launcher.py:10: TqdmDeprecationWarning:

This function will be removed in tqdm==5.0.0

Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm\_notebook`

# Remove the CWD from sys.path while we load stuff.

--------------- Epoch 1 ---------------

--------------- Epoch 20 ---------------

--------------- Epoch 40 ---------------

--------------- Epoch 60 ---------------

--------------- Epoch 80 ---------------

--------------- Epoch 100 ---------------

--------------- Epoch 120 ---------------

--------------- Epoch 140 ---------------

--------------- Epoch 160 ---------------

--------------- Epoch 180 ---------------

--------------- Epoch 200 ---------------

In [34]:

In [33]:

In [33]:

*#generating random noise and then feeding it to generator which generates some image/arra y from it .*

In [29]:

n\_ex=10

dim=(1, 10)

noise = np.random.normal(0, 1, size=(n\_ex, z\_dim))

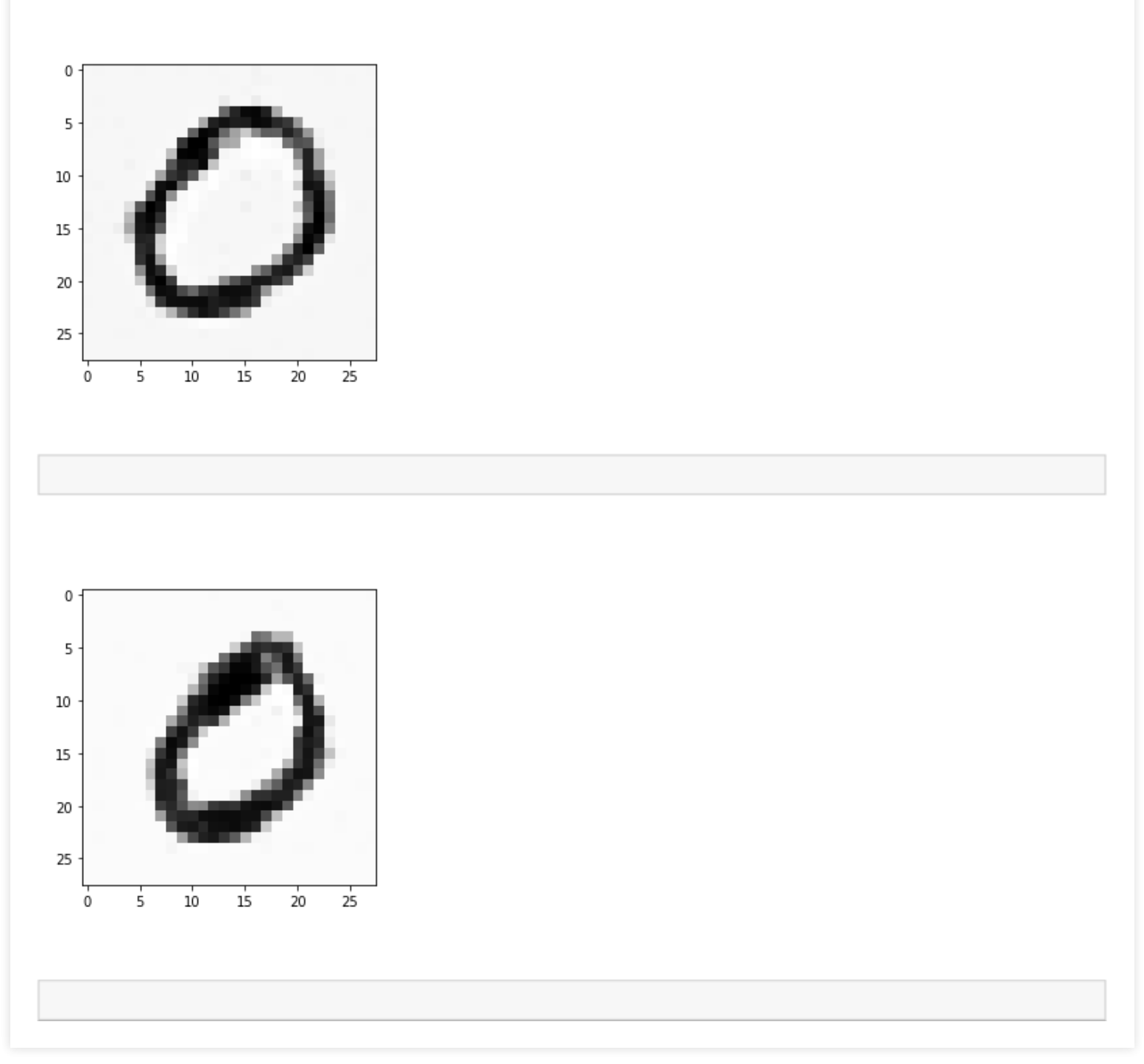
generated\_images = g.predict(noise)

generated\_images = generated\_images.reshape(generated\_images.shape[0], 28, 28)

In [30]:

plt.imshow(generated\_images[0, :, :], interpolation='nearest', cmap='gray\_r')

Out[30]:

<matplotlib.image.AxesImage at 0x7f37ba574828>

In [31]:

plt.imshow(generated\_images[1, :, :], interpolation='nearest', cmap='gray\_r')

Out[31]:

<matplotlib.image.AxesImage at 0x7f37ba62c7f0>

In [31]: