

# Digit Generation with a G.A.N

---

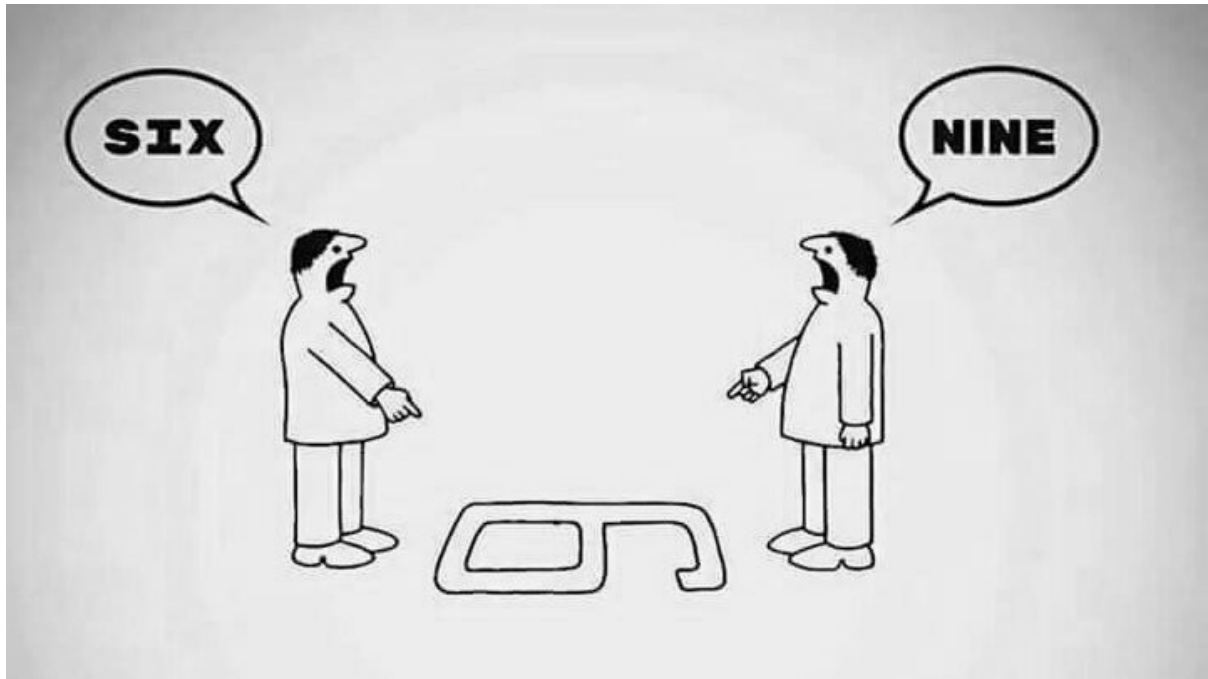


Photo by [Eridy Lukau](https://towardsdatascience.com/@EridyLukau) (<https://towardsdatascience.com/@EridyLukau>).

---

## Goal

This is not the 1st ambition of this [challenge](https://www.kaggle.com/c/digit-recognizer/) (<https://www.kaggle.com/c/digit-recognizer/>), anyway generation of new digits can also be interesting !

Here i'm going to create a GAN model, to train this model, and then use it to generate new "handwritten" digits...

In [60]:

```
import random
import numpy as np
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

#from keras.datasets import mnist
```

In [61]:

```
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping, TensorBoard
from tensorflow.keras.preprocessing.image import ImageDataGenerator, array_to_img,
```

In [62]:

```
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Dropout, MaxPool2D
from tensorflow.keras.layers import BatchNormalization, Activation, ZeroPadding2D
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
```

## Credits

Thanks to [Antoine Meicler \(https://github.com/ameicler\)](https://github.com/ameicler) and [Vincent Vandenbussche \(https://github.com/vincent-vdb\)](https://github.com/vincent-vdb) for all the things you teach me !

## Data preparation

We'll use the famous MNIST data intended to learn computer vision fundamentals.

In [63]:

```
df = pd.read_csv('../input/train.csv')
df.head()
```

Out[63]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775
0	1	0	0	0	0	0	0	0	0	0	...	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0
3	4	0	0	0	0	0	0	0	0	0	...	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0

5 rows × 785 columns

In [64]:

```
df['label'].unique()
```

Out[64]:

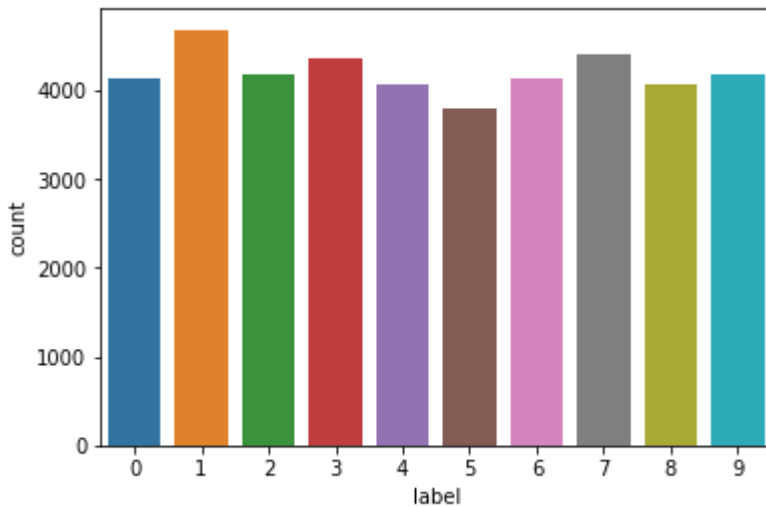
```
array([1, 0, 4, 7, 3, 5, 8, 9, 2, 6])
```

In [65]:

```
sns.countplot(df['label'])
```

Out[65]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f5ec4138048>



They are 10 different classes which seem to be balanced.

In [66]:

```
X_train, y_train = np.array(df.iloc[:, 1:]), np.array(df['label'])  
X_train.shape, y_train.shape
```

Out[66]:

((42000, 784), (42000,))

In [68]:

```
img_width, img_height, channels = 28, 28, 1  
X_train = X_train.reshape(42000, img_width, img_height, channels)  
X_train.shape
```

Out[68]:

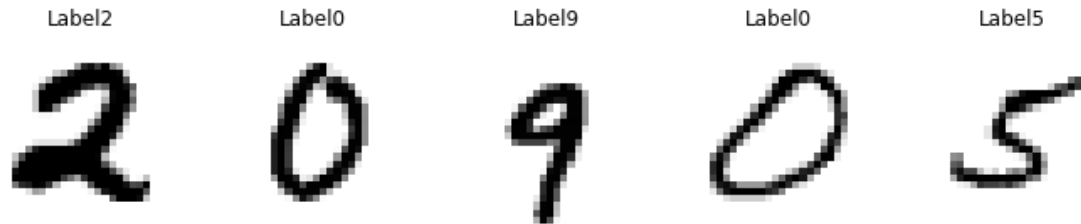
(42000, 28, 28, 1)

In [70]:

```
# display 5 randomly choosen images
plt.figure(figsize=(12, 5))

for i in range(1, 6):
    plt.subplot(1, 5, i)
    num = random.randint(0, X_train.shape[0])
    plt.imshow(X_train[num].reshape(img_width, img_height), cmap="gray_r")
    plt.axis('off')
    label = 'Label' + str(y_train[num])
    plt.title(label)

plt.show()
```



In [73]:

```
type(X_train[0, 0, 0, 0])
```

Out[73]:

numpy.int64

In [74]:

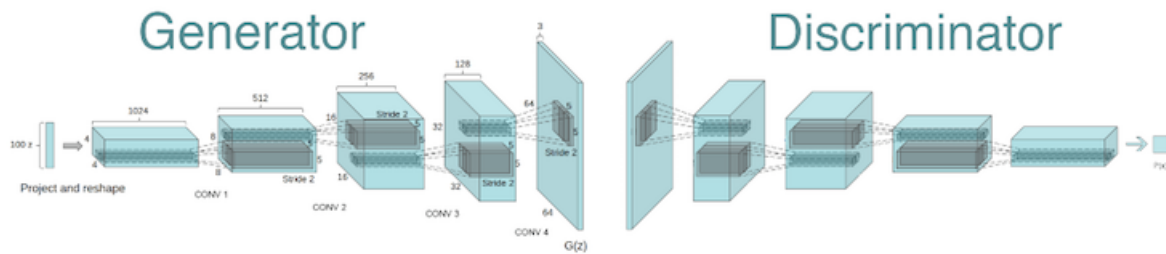
```
# Rescale -1 to 1 and format the X_train dataset
X_train = (X_train.astype(np.float32) - 127.5) / 127.5
#X_train = np.expand_dims(X_train, axis=3)
```

Don't forget that the MNIST dataset is grayscale so it contains only one channel.

Keras expects input images to have 3 dimensions even if there is only one channel.

## The GAN architecture

### Theory



Summary form [Towardsdatascience \(https://towardsdatascience.com/image-generator-drawing-cartoons-with-generative-adversarial-networks-45e814ca9b6b\)](https://towardsdatascience.com/image-generator-drawing-cartoons-with-generative-adversarial-networks-45e814ca9b6b)

We would like to provide a set of images as an input, and generate samples based on them as an output.

Input Images -> GAN -> Output Samples

With the following problem definition, GANs fall into the Unsupervised Learning bucket because we are not going to feed the model with labeled data.

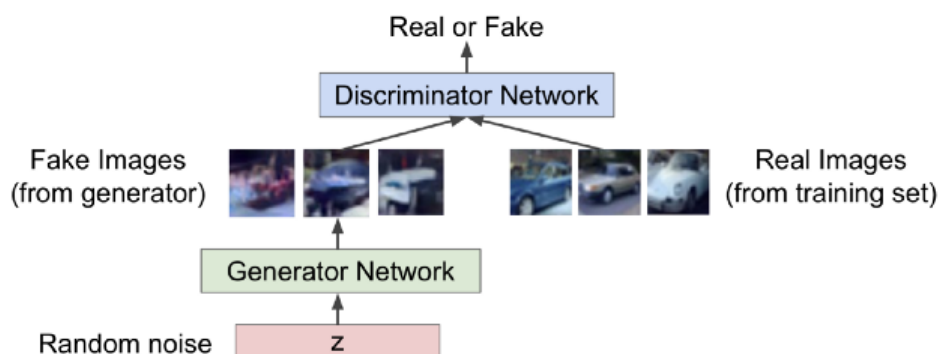
*The underlying idea behind GAN is that it contains two neural networks that compete against each other in a zero-sum game framework, i.e. generator and a discriminator.*

## Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

**Generator network:** try to fool the discriminator by generating real-looking images

**Discriminator network:** try to distinguish between real and fake images



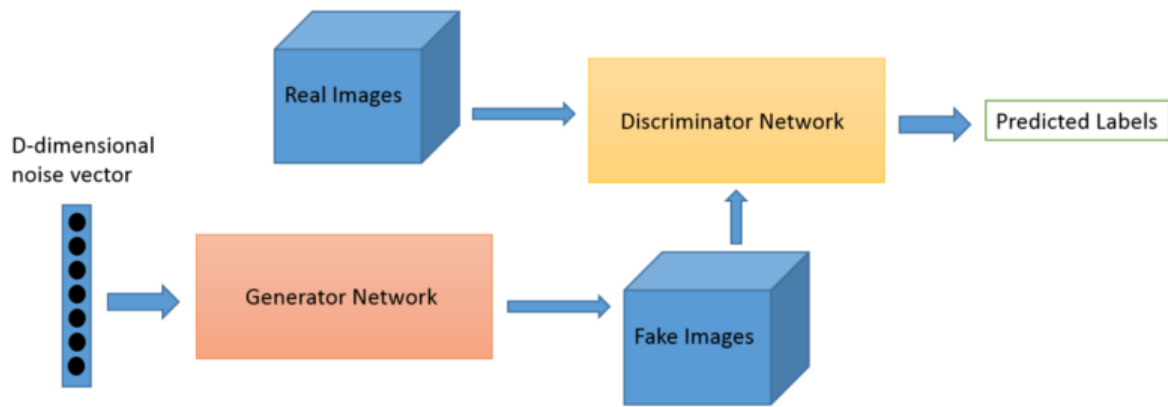
Fake and real images copyright Emily Denton et al. 2015. Reproduced with permission.

**Generator** The Generator takes random noise as an input and generates samples as an output. It's goal is to generate such samples that will fool the Discriminator to think that it is seeing real images while actually seeing fakes. We can think of the Generator as a counterfeit.

**Discriminator** Discriminator takes both real images from the input dataset and fake images from the Generator and outputs a verdict whether a given image is legit or not. We can think of the Discriminator as a policeman trying to catch the bad guys while letting the good guys free.

**Minimax Representation** If we think once again about Discriminator's and Generator's goals, we can see that they are opposing each other. Discriminator's success is a Generator's failure and vice-versa. That is why we can represent GANs framework more like Minimax game framework rather than an optimization problem.

GAN data flow can be represented as in the following diagram.



## The Generator

The first step is to build a generator. We start with an **input noise shape of size 100**. Then, we create a sequential model to increase the size of the data up to 1024, before reshaping the data back to the input image shape.

Each layer will be made of:

- A **Dense layer** (sizes 256, 512, 1024 in order)
- A **LeakyRelu activation** with alpha = 0.2
- A **Batch normalization** (momentum = 0.8)

In [76]:

```
img_shape = (img_width, img_height, channels)

def build_generator():
    # Input Data
    noise_shape = (100,)
    noise = Input(shape=noise_shape)

    # Create the sequential model
    model = Sequential()

    # Build the first layer
    model.add(Dense(256, input_shape=noise_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    # Second layer
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    # Third layer
    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    # Flatten and reshape
    model.add(Dense(np.prod(img_shape), activation='tanh'))
    model.add(Reshape(img_shape))

    # Get model summary
    img = model(noise)
    model.summary()

    return Model(noise, img)
```

Compilation of the Generator and add an Adam optimizer as advised.

In [77]:

```
optimizer = Adam(0.0002, 0.5)
generator = build_generator()
generator.compile(loss='binary_crossentropy', optimizer=optimizer)
```

Layer (type)	Output Shape	Param #
dense_14 (Dense)	(None, 256)	25856
leaky_re_lu_11 (LeakyReLU)	(None, 256)	0
batch_normalization_v1_9 (Ba	(None, 256)	1024
dense_15 (Dense)	(None, 512)	131584
leaky_re_lu_12 (LeakyReLU)	(None, 512)	0
batch_normalization_v1_10 (B	(None, 512)	2048
dense_16 (Dense)	(None, 1024)	525312
leaky_re_lu_13 (LeakyReLU)	(None, 1024)	0
batch_normalization_v1_11 (B	(None, 1024)	4096
dense_17 (Dense)	(None, 784)	803600
reshape_1 (Reshape)	(None, 28, 28, 1)	0
Total params: 1,493,520		
Trainable params: 1,489,936		
Non-trainable params: 3,584		

## The Discriminator

Now let's build the discriminator. It takes an input that has the shape of the image. The steps are the following :

- Declaration of the **Sequential** model
- **Flatten** the images (with input shape = image shape)
- Addition of a **Dense layer** of 512 and a **Leaky Relu** (0.2)
- Addition of a **Dense layer** of 256 and a **Leaky Relu** (0.2)
- Addition of a **Dense layer** of size 1. What activation function would you use ?



In [78]:

```
def build_discriminator():
    img = Input(shape=img_shape)

    # Create the sequential model
    model = Sequential()

    # Flatten the images taken as inputs
    model.add(Flatten(input_shape=img_shape))

    # First layer
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))

    # Second layer
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.2))

    # Last layer, return either 0 or 1
    model.add(Dense(1, activation='sigmoid'))

    # Get model summary
    validity = model(img)
    model.summary()

    return Model(img, validity)
```

Compilation of the discriminator. (Observe the metric we are using)

In [79]:

```
discriminator = build_discriminator()
discriminator.compile(
    loss='binary_crossentropy',
    optimizer=optimizer,
    metrics=['accuracy'])
```

Layer (type)	Output Shape	Param #
=====		
flatten_1 (Flatten)	(None, 784)	0
dense_18 (Dense)	(None, 512)	401920
leaky_re_lu_14 (LeakyReLU)	(None, 512)	0
dense_19 (Dense)	(None, 256)	131328
leaky_re_lu_15 (LeakyReLU)	(None, 256)	0
dense_20 (Dense)	(None, 1)	257
=====		
Total params: 533,505		
Trainable params: 533,505		
Non-trainable params: 0		

# Build the whole GAN model

It is time to build the entire **GAN model**. This operation can be achieved in 4 major steps :

- Declare the input
- Set the image as the result of the generator of the input
- Set the output as the result of the discriminator of the generated image
- Define and compile the model

In [80]:

```
# 1. Declare input of size (100, )
z = Input(shape=(100,))

# 2. Define the generated image from the input - Use the generator model compiled above
img = generator(z)

# 3. Define the output from the image - Use the discriminator model compiled above
valid = discriminator(img)

# For the combined model, only train the generator
discriminator.trainable = False

# 4. Combined model - by defining the input and the output
combined = Model(z, valid)

# Once created, compilation of the whole model
combined.compile(loss='binary_crossentropy', optimizer=optimizer)
```

Summary of the new model created.

In [81]:

```
combined.summary()
```

Layer (type)	Output Shape	Param #
=====		
input_9 (InputLayer)	(None, 100)	0
-----		
model_3 (Model)	(None, 28, 28, 1)	1493520
-----		
model_4 (Model)	(None, 1)	533505
=====		
Total params: 2,027,025		
Trainable params: 1,489,936		
Non-trainable params: 537,089		
-----		

Function that is used to save generated images once in a while.

In [85]:

```
def save_imgs(epoch):  
  
    # Predict from input noise  
    r, c = 5, 5  
    noise = np.random.normal(0, 1, (r * c, 100))  
    gen_imgs = generator.predict(noise)  
  
    # Rescale images 0 - 1  
    gen_imgs = 0.5 * gen_imgs + 0.5  
  
    # Subplots  
    fig, axs = plt.subplots(r, c)  
    cnt = 0  
  
    for i in range(r):  
        for j in range(c):  
            axs[i,j].imshow(gen_imgs[cnt, :,:,0], cmap='gray')  
            axs[i,j].axis('off')  
            cnt += 1  
  
    fig.savefig("../output/mnist_%d.png" % epoch)  
    plt.close()
```

## Model Training

First of all, we set :

- the number of epochs the model will train to 15'000
- the batch size to 64
- the interval at which we save the images to 1000

In [86]:

```
epochs = 15000  
batch_size = 64  
save_interval = 1000  
half_batch = int(batch_size / 2)
```

The following code is complete. Try to understand the different steps, debug potential errors from your previous code and compile it.

In [87]:

```

d_loss_hist = []
g_loss_hist = []
d_acc = []

for epoch in range(epochs):

    # -----
    # Train Discriminator
    # -----

    # Pick 50% of sample images
    idx = np.random.randint(0, X_train.shape[0], half_batch)
    imgs = X_train[idx]

    # Generate 50% of new images
    noise = np.random.normal(0, 1, (half_batch, 100))
    gen_imgs = generator.predict(noise)

    # Train discriminator on real images with label 1
    d_loss_real = discriminator.train_on_batch(imgs, np.ones((half_batch, 1)))

    # Train discriminator on fake images with label 0
    d_loss_fake = discriminator.train_on_batch(gen_imgs, np.zeros((half_batch, 1)))

    # Loss of discriminator = Mean of Real and Fake loss
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    d_loss_hist.append(d_loss[0])
    d_acc.append(d_loss[1])

    # -----
    # Train Generator
    # -----

    # The generator wants the discriminator to label the generated samples as valid
    noise = np.random.normal(0, 1, (batch_size, 100))
    valid_y = np.array([1] * batch_size)

    # Train the generator
    g_loss = combined.train_on_batch(noise, valid_y)
    g_loss_hist.append(g_loss)

    # Print the progress
    print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch, d_loss[0], 100*d_loss[1], g_loss))

    if epoch % save_interval == 0:
        save_imgs(epoch)

```

WARNING:tensorflow:Discrepancy between trainable weights and collected trainable weights, did you set `model.trainable` without calling `model.compile` after ?

WARNING:tensorflow:Discrepancy between trainable weights and collected trainable weights, did you set `model.trainable` without calling `model.compile` after ?

```

0 [D loss: 0.374034, acc.: 81.25%] [G loss: 0.722144]
1 [D loss: 0.339896, acc.: 81.25%] [G loss: 0.761038]
2 [D loss: 0.347759, acc.: 81.25%] [G loss: 0.847006]
3 [D loss: 0.334043, acc.: 81.25%] [G loss: 0.955707]

```

```
4 [D loss: 0.314673, acc.: 87.50%] [G loss: 1.075714]
5 [D loss: 0.272925, acc.: 90.62%] [G loss: 1.225585]
6 [D loss: 0.217444, acc.: 98.44%] [G loss: 1.306365]
7 [D loss: 0.205875, acc.: 100.00%] [G loss: 1.505471]
8 [D loss: 0.172943, acc.: 100.00%] [G loss: 1.633218]
9 [D loss: 0.144670, acc.: 100.00%] [G loss: 1.828782]
10 [D loss: 0.122131, acc.: 100.00%] [G loss: 1.955405]
11 [D loss: 0.097202, acc.: 100.00%] [G loss: 2.050833]
12 [D loss: 0.094377, acc.: 100.00%] [G loss: 2.157885]
```

## Creation of new digits

We now have all the elements required to generate new samples. What are according to you :

- the steps to generate new samples ?
- the part of the network we re-use ?

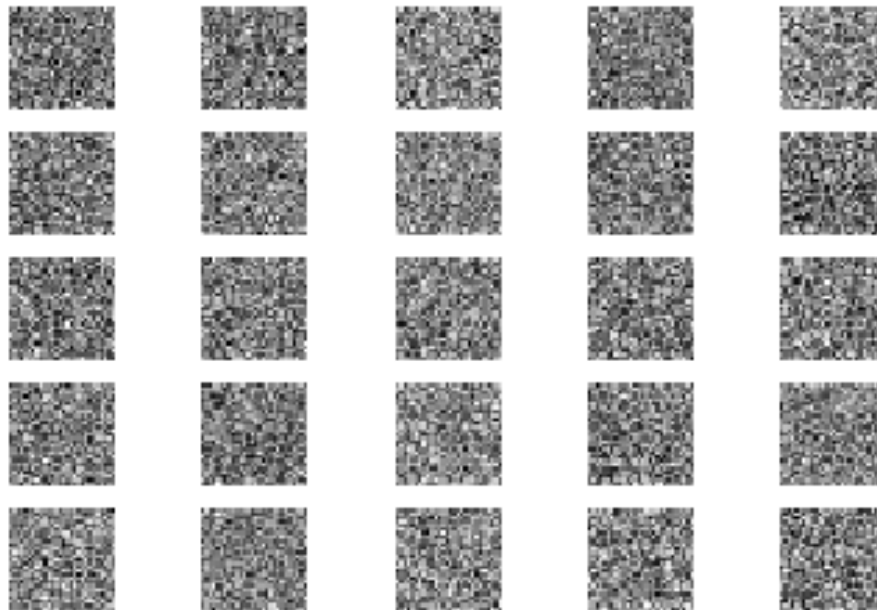
You are now asked to generate and visualize new samples from the steps you defined above. Pay attention when plotting generated images to :

- rescale the images between 0 and 1 (as done previously)
- reshape the generated image to 28\*28

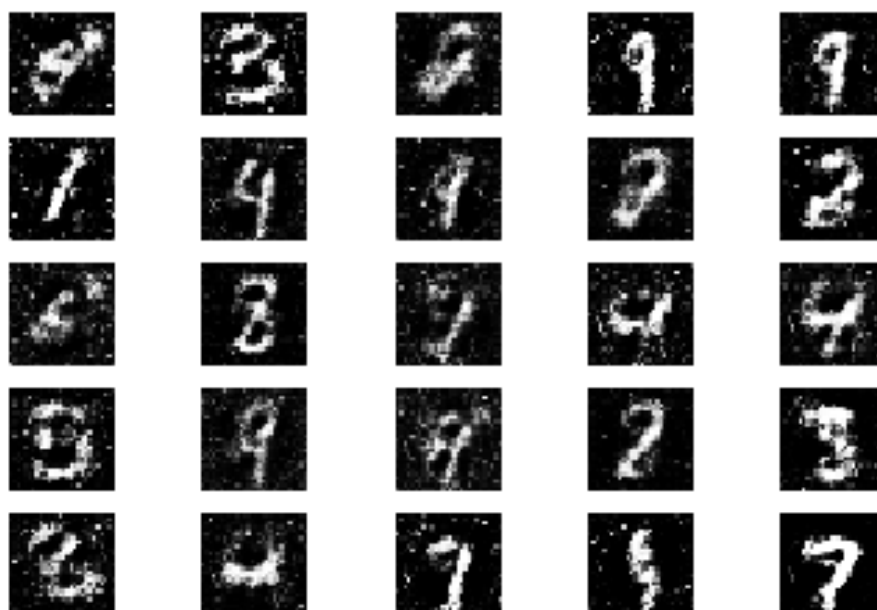
In [ ]:

```
noise = np.random.normal(0, 1, (1, 100))
gen_imgs = generator.predict(noise)
gen_imgs = 0.5 * gen_imgs + 0.5
plt.imshow(gen_imgs.reshape(28,28), cmap="gray_r")
plt.axis("off")
plt.show()
```

Here are the 1st digits created, it's only noise...



After 3000 iterations...the shape is here



And after 6000 iterations, it's much better

