# LAB 11 : Generative Adversarial Networks

Name : Vaishnavi, Joshitha

Roll Number : 180020039, 180020009

References :

1. https://towardsdatascience.com/gans-from-scratch-8f5da17b3fb4

## **Problem 1** : Understanding and Training Vanilla GANs

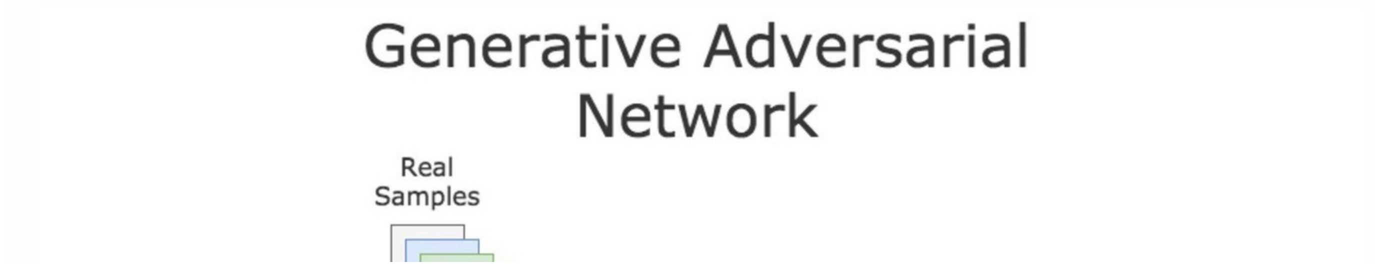
1. Consider MNIST Digit Dataset
2. Create a Generator and a Discriminator Network and train the network. Follow this article : https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f
3. Explain briefly about your understanding of GANs

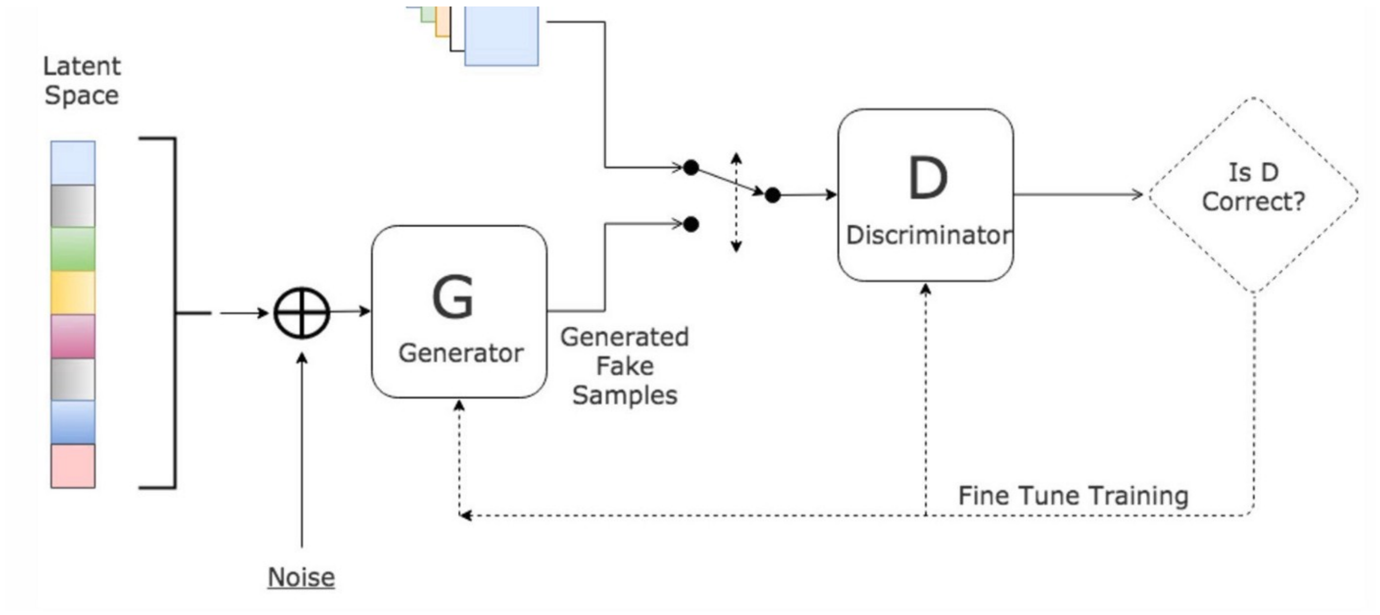
## Write down the Objectives, Hypothesis and Experimental description for the above problem

Objective : Train a GAN with a generator and discriminator on mnist digit dataset.

Experimental Description:

1. Sample a noise set and a real-data set, each with size m.
2. Train the Discriminator on this data.
3. Sample a different noise subset with size m.
4. Train the Generator on this data.
5. Repeat from Step 1.





Generative Adversarial Networks are composed of two models:

1. The first model is called a Generator and it aims to generate new data similar to the expected one. The Generator could be asimilated to a human art forger, which creates fake works of art.
2. The second model is named the Discriminator. This model's goal is to recognize if an input data is 'real' — belongs to the original dataset — or if it is 'fake' — generated by a forger. In this scenario, a Discriminator is analogous to an art expert, which tries to detect artworks as truthful or fraud.

## Programming :

Please write a program to demonstrate the same

```
## Write your code here
!pip install torchvision tensorboardx jupyter matplotlib numpy

```

```
import torch
from torch import nn, optim
from torch.autograd.variable import Variable
from torchvision import transforms, datasets

```

LeCunn's MNIST dataset, consisting of about 60,000 black and white images of handwritten digits, each with size 28x28 pixels²

Preprocessing :

Specifically, the input values which range in between [0, 255] will be normalized between -1 and 1. This means the value 0 will be mapped to -1, the value 255 to 1, and similarly all values in between will get a value in the range [-1, 1].

```
def mnist_data():
    compose = transform = transforms.Compose([transforms.ToTensor(), transforms.
    out_dir = './dataset'
    return datasets.MNIST(root=out_dir, train=True, transform=compose, download=
# Load data
data = mnist_data()
# Create loader with data, so that we can iterate over it
data_loader = torch.utils.data.DataLoader(data, batch_size=100, shuffle=True)
# Num batches
num_batches = len(data_loader)

```

## Discriminator

Discriminator will take a flattened image as its input, and return the probability of it belonging to the real dataset, or the synthetic dataset. The input size for each image will be 28x28=784.

Structure :
3 hidden layers, each followed by a Leaky-ReLU nonlinearity and a Dropout layer to prevent overfitting.

A Sigmoid/Logistic function is applied to the real-valued output to obtain a value in the open-range (0, 1):

```
class DiscriminatorNet(torch.nn.Module):
    """
    A three hidden-layer discriminative neural network
    """
    def __init__(self):
        super(DiscriminatorNet, self).__init__()
        n_features = 784
        n_out = 1

        self.hidden0 = nn.Sequential(
            nn.Linear(n_features, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )
        self.hidden1 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )
        self.hidden2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )
        self.out = nn.Sequential(
            torch.nn.Linear(256, n_out),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.hidden0(x)
        x = self.hidden1(x)
        x = self.hidden2(x)
        x = self.out(x)
        return x


discriminator = DiscriminatorNet()

```

Additional functionality that allows us to convert a flattened image into its 2-dimensional representation, and another one that does the opposite.

```
def images_to_vectors(images):
    return images.view(images.size(0), 784)

def vectors_to_images(vectors):
    return vectors.view(vectors.size(0), 1, 28, 28)

```

## Generative Network :

Takes a latent variable vector as input, and returns a 784 valued vector, which corresponds to a flattened 28x28 image. The purpose of this network is to learn how to create undistinguishable images of hand-written digits, which is why its output is itself a new image.

Structure :

1. three hidden layers, each followed by a Leaky-ReLU nonlinearity.
2. The output layer will have a TanH activation function, which maps the resulting values into

the (-1, 1) range, which is the same range in which our preprocessed MNIST images is bounded.

```
class GeneratorNet(torch.nn.Module):
    """
    A three hidden-layer generative neural network
    """
    def __init__(self):
        super(GeneratorNet, self).__init__()
        n_features = 100
        n_out = 784

        self.hidden0 = nn.Sequential(
            nn.Linear(n_features, 256),
            nn.LeakyReLU(0.2)
        )
        self.hidden1 = nn.Sequential(
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2)
        )
        self.hidden2 = nn.Sequential(
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2)
        )

        self.out = nn.Sequential(
            nn.Linear(1024, n_out),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.hidden0(x)
        x = self.hidden1(x)
        x = self.hidden2(x)
        x = self.out(x)
        return x
generator = GeneratorNet()

```

This function allows us to create the random noise. The random noise will be sampled from a normal distribution with mean 0 and variance 1.

```
def noise(size):
    '''
    Generates a 1-d vector of gaussian sampled random values
    '''
    n = Variable(torch.randn(size, 100))
    return n

```

we’ll use Adam as the optimization algorithm for both neural networks, with a learning rate of 0.0002

```
d_optimizer = optim.Adam(discriminator.parameters(), lr=0.0002)
g_optimizer = optim.Adam(generator.parameters(), lr=0.0002)

```

Loss function : Binary Cross Entopy Loss (BCE Loss) as it resembles the log-loss for both the Generator and Discriminator

```
loss = nn.BCELoss()

```

$$L = \{l_1, ..., l_N\}^T, l_i = -w_i[y_i \cdot log(v_i) + (1-y) \cdot log(1-v_i)]$$

Binary Cross Entropy Log. Mean is calculated by computing sum(L) / N. In this formula the values y are named targets, v are the inputs, and w are the weights. Since we don’t need the weight at all, it’ll be set to $w_i$=1 for all i.

Discriminator Loss:

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

If we replace $v_i$ = D($x_i$) and $y_i$=1 ∀ i (for all i) in the BCE-Loss definition, we obtain the loss related to the real-images. Conversely if we set $v_i$ = D(G($z_i$)) and $y_i$=0 ∀ i, we obtain the loss related to the fake-images

Since maximizing a function is equivalent to minimizing it’s negative, and the BCE-Loss term has a minus sign, we don’t need to worry about the sign.

We can observe that the real-images targets are always ones, while the fake-images targets are zero, so it would be helpful to define the following functions:

```
def ones_target(size):
    '''
    Tensor containing ones, with shape = size
    '''
    data = Variable(torch.ones(size, 1))
    return data

def zeros_target(size):
    '''
    Tensor containing zeros, with shape = size
    '''
    data = Variable(torch.zeros(size, 1))
    return data

```

By summing up these two discriminator losses we obtain the total mini-batch loss for the Discriminator. We will calculate the gradients separately, and then update them together.

```
def train_discriminator(optimizer, real_data, fake_data):
    N = real_data.size(0)
    # Reset gradients
    optimizer.zero_grad()

    # 1.1 Train on Real Data
    prediction_real = discriminator(real_data)
    # Calculate error and backpropagate
    error_real = loss(prediction_real, ones_target(N) )
    error_real.backward()

    # 1.2 Train on Fake Data
    prediction_fake = discriminator(fake_data)
    # Calculate error and backpropagate
    error_fake = loss(prediction_fake, zeros_target(N))
    error_fake.backward()

    # 1.3 Update weights with gradients
    optimizer.step()

    # Return error and predictions for real and fake inputs
    return error_real + error_fake, prediction_real, prediction_fake

```

Generator Loss:

$$\frac{1}{m}\sum^{m}\log\left(1-D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

$i=1$

Rather than minimizing log(1- D(G(z))), training the Generator to maximize log D(G(z)) will provide much stronger gradients early in training. Both losses may be swapped interchangeably since they result in the same dynamics for the Generator and Discriminator.

Maximizing log D(G(z)) is equivalent to minimizing it’s negative and since the BCE-Loss definition has a minus sign, we don’t need to take care of the sign. Similarly to the Discriminator, if we set $v_i$ = $D(G(z_i))$ and $y_i$=1 ∀ i, we obtain the desired loss to be minimized.

```
def train_generator(optimizer, fake_data):
    N = fake_data.size(0)
    # Reset gradients
    optimizer.zero_grad()
    # Sample noise and generate fake data
    prediction = discriminator(fake_data)
    # Calculate error and backpropagate
    error = loss(prediction, ones_target(N))
    error.backward()
    # Update weights with gradients
    optimizer.step()
    # Return error
    return error

```

```
!wget https://raw.githubusercontent.com/diegoalejogm/gans/master/utils.py
```

```
from utils1 import Logger

```

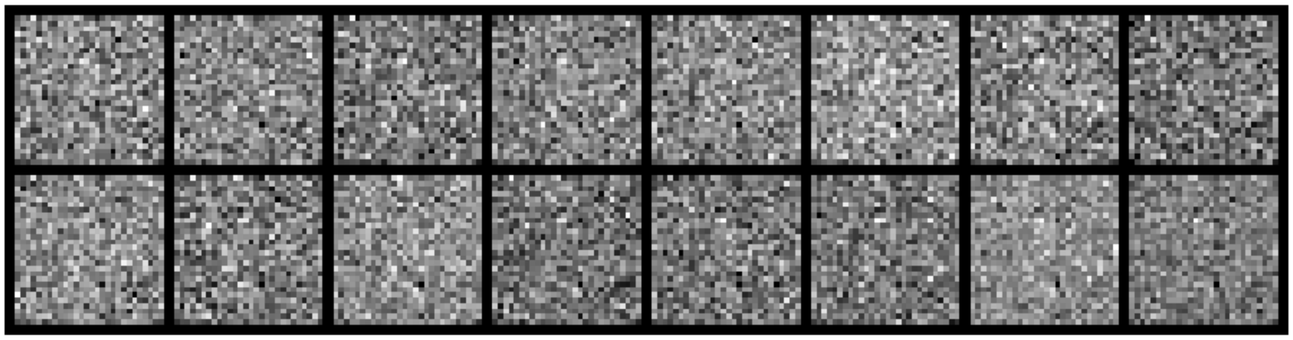
```
num_test_samples = 16
test_noise = noise(num_test_samples)

```

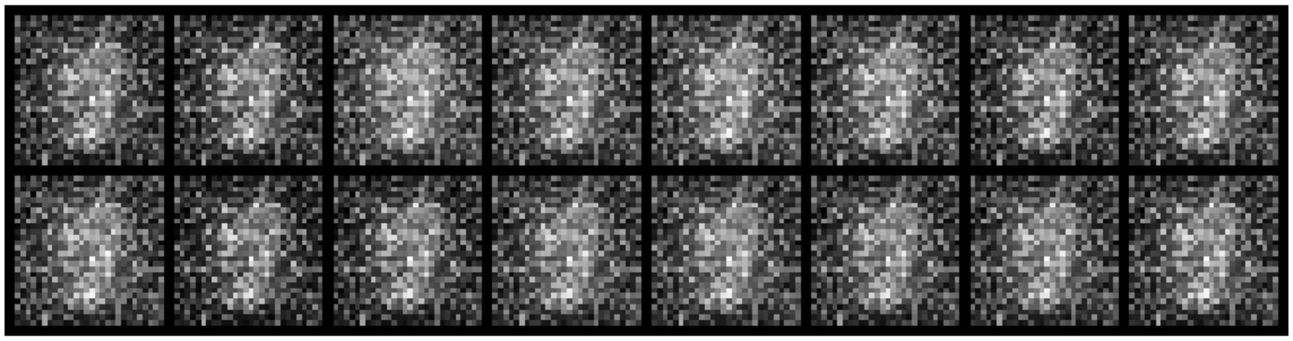
```
# Create logger instance
logger = Logger(model_name='VGAN', data_name='MNIST')
# Total number of epochs to train
num_epochs = 200
for epoch in range(num_epochs):
    for n_batch, (real_batch,_) in enumerate(data_loader):
        N = real_batch.size(0)
        # 1. Train Discriminator
        real_data = Variable(images_to_vectors(real_batch))
        # Generate fake data and detach
        # (so gradients are not calculated for generator)
        fake_data = generator(noise(N)).detach()
        # Train D
        d_error, d_pred_real, d_pred_fake = \
              train_discriminator(d_optimizer, real_data, fake_data)

        # 2. Train Generator
        # Generate fake data
        fake_data = generator(noise(N))
        # Train G
        g_error = train_generator(g_optimizer, fake_data)
        # Log batch error
        logger.log(d_error, g_error, epoch, n_batch, num_batches)
        # Display Progress every few batches
        if (n_batch) % 100 == 0:
            test_images = vectors_to_images(generator(test_noise))
            test_images = test_images.data
            logger.log_images(
                test_images, num_test_samples,
                epoch, n_batch, num_batches
            );
            # Display status Logs
            logger.display_status(
                epoch, num_epochs, n_batch, num_batches,
                d_error, g_error, d_pred_real, d_pred_fake
            )

```

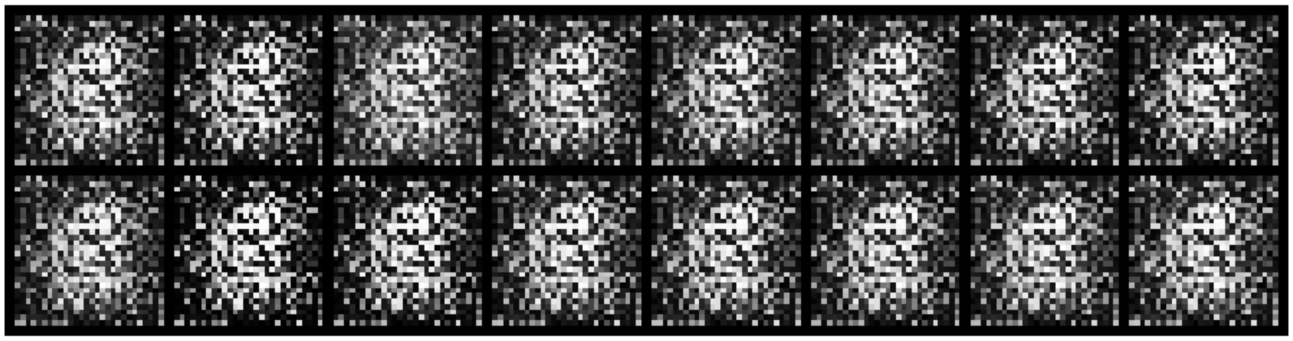


```
Epoch: [0/200], Batch Num: [0/600]
Discriminator Loss: 1.3903, Generator Loss: 0.6739
D(x): 0.5079, D(G(z)): 0.5095
```

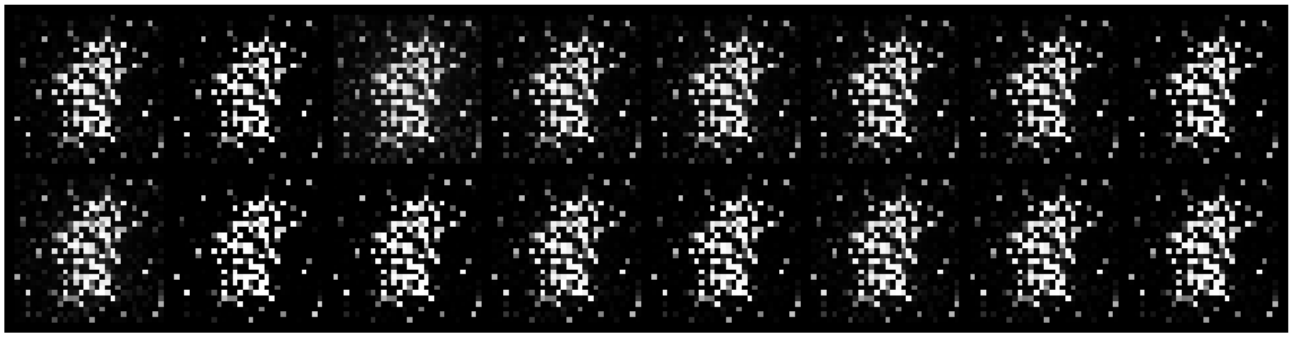


```
Epoch: [0/200], Batch Num: [100/600]
```

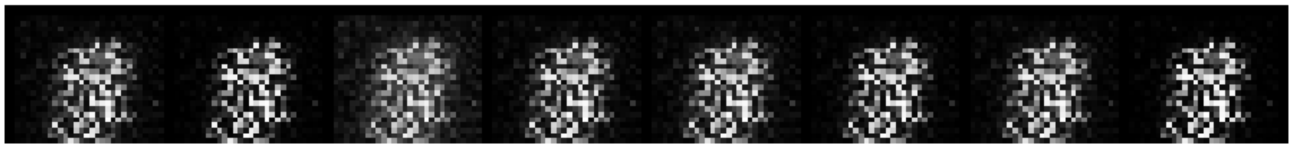
```
Discriminator Loss: 1.3107, Generator Loss: 1.5540
D(x): 0.6453, D(G(z)): 0.4654
```



```
Epoch: [0/200], Batch Num: [200/600]
Discriminator Loss: 0.4103, Generator Loss: 3.8607
D(x): 0.8561, D(G(z)): 0.1803
```



```
Epoch: [0/200], Batch Num: [300/600]
Discriminator Loss: 0.2172, Generator Loss: 5.2012
D(x): 0.9144, D(G(z)): 0.0706
```



## Inferences and Conclusion : State all the key observations and conclusion

1. The discriminator error is very high (Aroung 1.4 - 1.3 in the first few epochs) in the beginning, as it doesn't know how to classify correctly images as being either real or fake. After 1 epoch :

   ```
   Discriminator Loss: 1.3903, Generator Loss: 0.6739
   ```

2. As the discriminator becomes better and its error decreases to about .5, the generator error increases, proving that the discriminator outperforms the generator and it can correctly classify the fake samples.

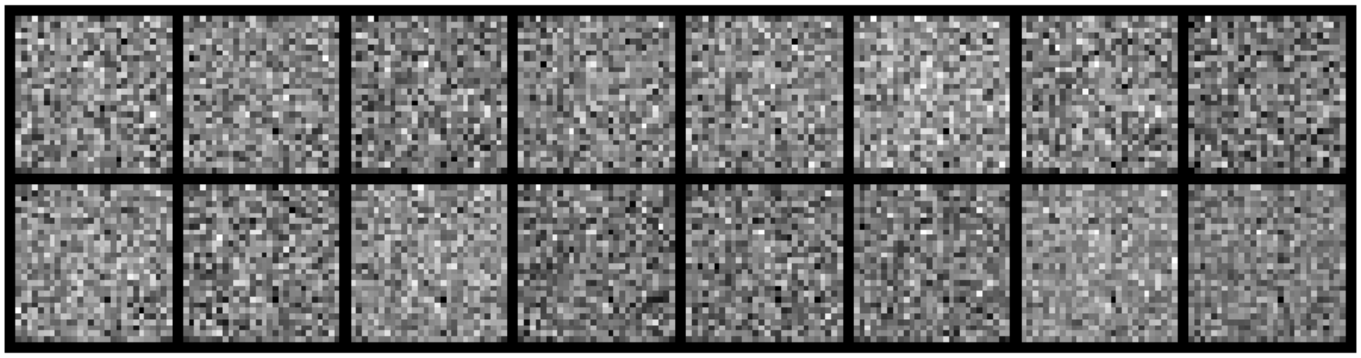
```
Discriminator Loss: 0.4254, Generator Loss: 5.7648
```

3. As time passes and training continues, the generator error lowers, implying that the images it generates are better and better. While the generator improves, the discriminator’s error increases, because the synthetic images are becoming more realistic each time.

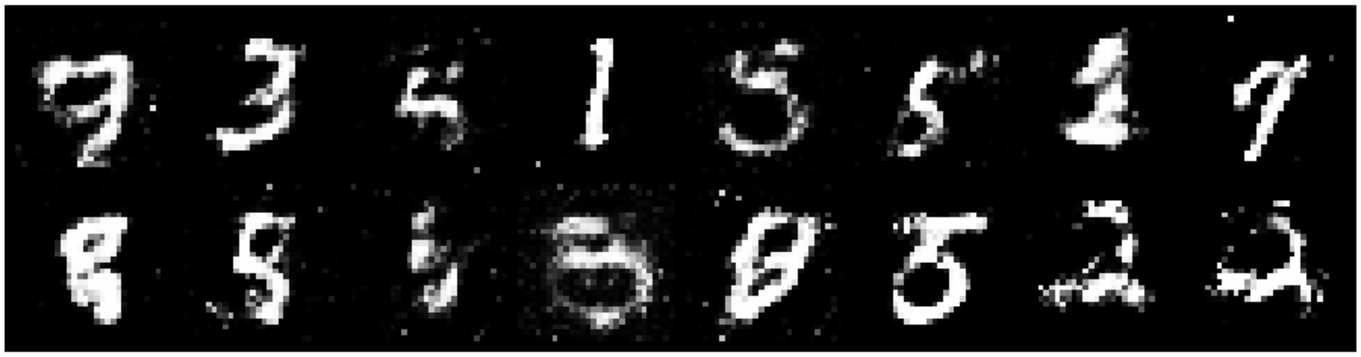
Final epoch :

```
Discriminator Loss: 1.3020, Generator Loss: 0.8640
```

Generated Images are :



AFter some 50 epochs :



After 200 epochs: