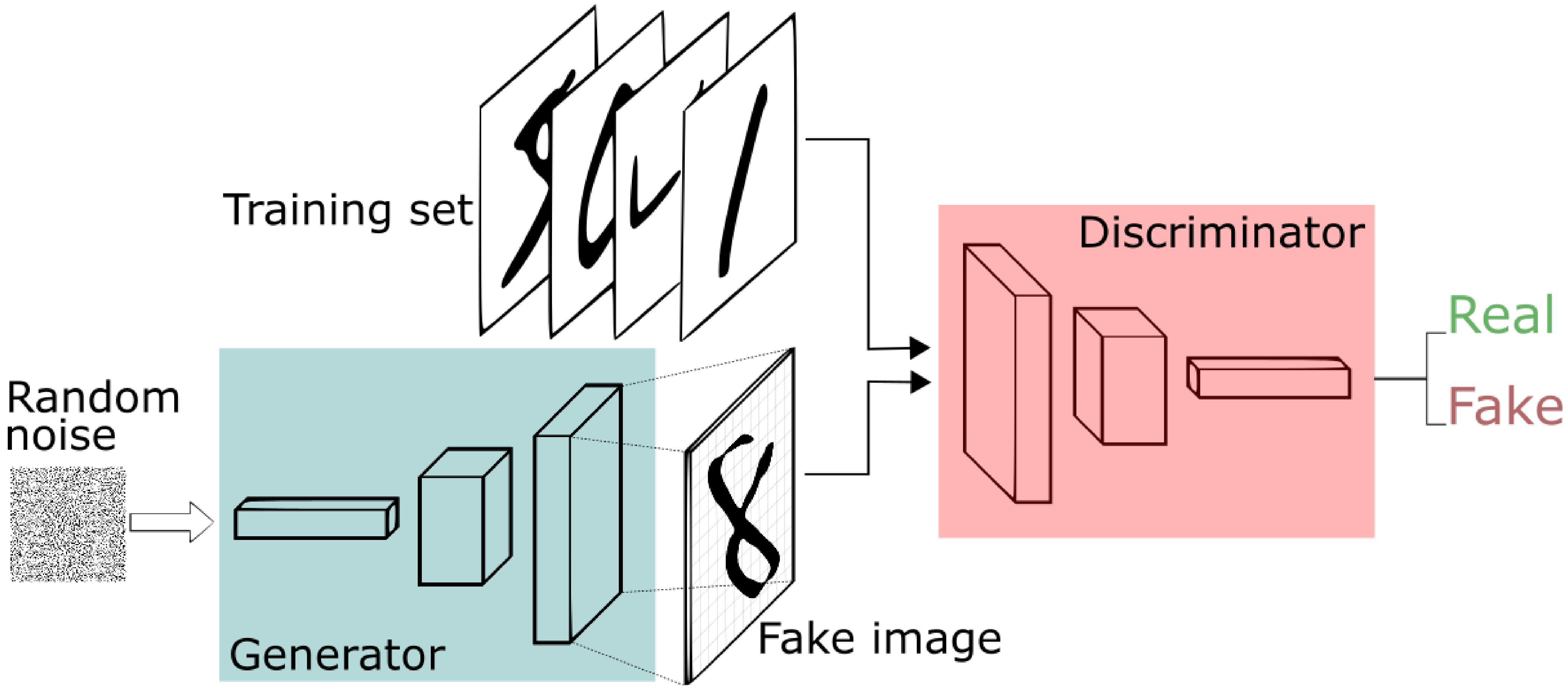
Project: Deep learning with PyTorch

Build a Generative Adversarial Network

Basic process structure of Deep learning



Import Libraries

4초

[1]
```
import torch
torch.manual_seed(42)
import numpy as np
import matplotlib.pyplot as plt

from tqdm.notebook import tqdm
```

Configurations

[2]
```
device = 'cuda' # image = image.to(device) FOR GPU

batch_size = 128 # trainloader, training loop

noise_dim = 64 # generator model

# optimizers parameters
lr = 0.0002 # learning rate
beta_1 = 0.5
beta_2 = 0.99

#Trainning variables
epochs = 20
#epochs: an event or a time that begins a new period or development, in this case, the number of training of all the dataset.
```

Load MNIST Dataset

4초 [3]

```python
from torchvision import datasets, transforms as T
# pytorch에서 제공하는 데이터셋들이 모여있는 패키지
```

[4]

```python
train_augs = T.Compose([
                        T.RandomRotation((-20,+20)),
                        #image를 random으로 degree 각도를 회전함
                        T.ToTensor() # (h, w, c) -> (c, h, w)
                        # 데이터를 0에서 255까지 있는 값을 0에서 1사이 값으로 변환
])
```

[5]

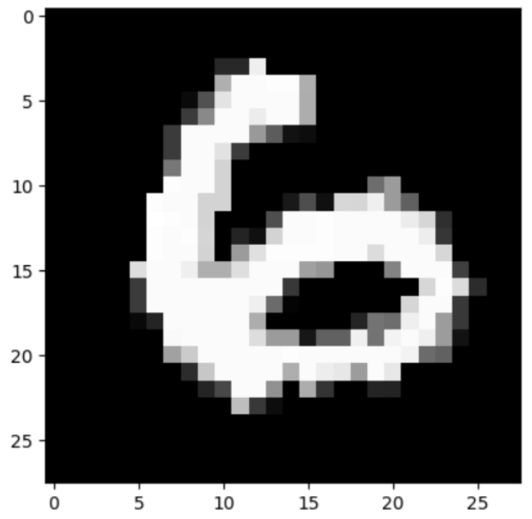
```python
trainset = datasets.MNIST('MNIST/', download = True, train = True, transform = train_augs)
#The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems.
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to MNIST/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:00<00:00, 140694717.97it/s]Extracting MNIST/MNIST/raw/train-images-idx3-ubyte.gz to MNIST/MNIST/raw
```

0초 [34]

```python
image, label = trainset[9000]
plt.imshow(image.squeeze(), cmap = 'gray')
```

<matplotlib.image.AxesImage at 0x7e89c46e6a10>



0초 [7]

```python
print("total images present in trainset are : ", len(trainset))
```

```
total images present in trainset are :  60000
```

Load Dataset Into Batches

```
[8] #import DataLoader and make_grid
    from torch.utils.data import DataLoader
    from torchvision.utils import make_grid
```

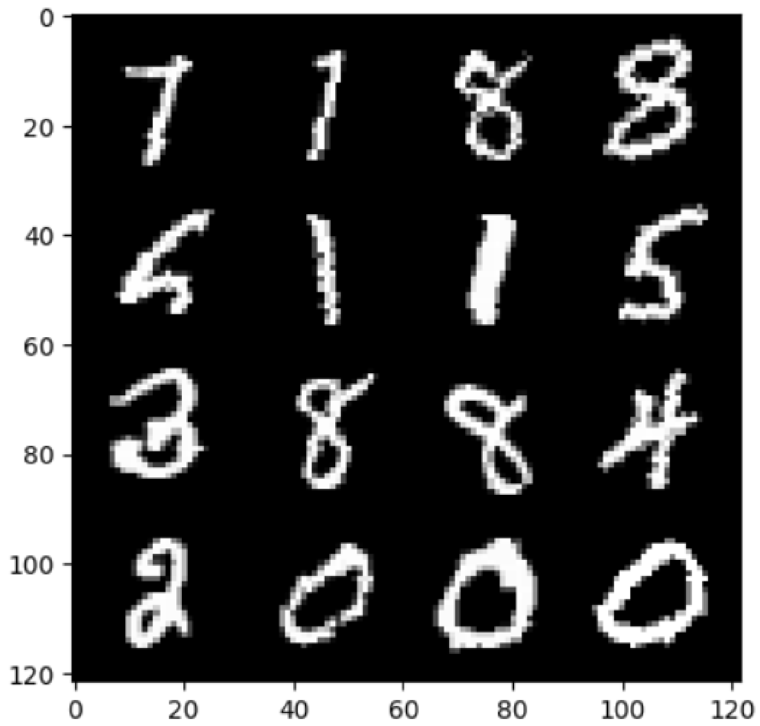
```
[9] #make an object of Dataloader
    trainloader = DataLoader(trainset, batch_size = batch_size, shuffle = True)
```

```
[10] print("Total no. of batches in trainloader: ", len(trainloader))

     Total no. of batches in trainloader:  469
```

```
[11] #create an iterator for trainloader
     dataiter = iter(trainloader)
     images, _ = next(dataiter)
     print(images.shape)

     torch.Size([128, 1, 28, 28])
```

```
[12] # 'show_tensor_images' : function is used to plot some of images from the batch

     def show_tensor_images(tensor_img, num_images = 16, size=(1, 28, 28)):
         unflat_img = tensor_img.detach().cpu()
         img_grid = make_grid(unflat_img[:num_images], nrow=4)
         plt.imshow(img_grid.permute(1, 2, 0).squeeze())
         plt.show()
```

```
[13] show_tensor_images(images, num_images = 16)
```



Create Discriminator Network

0초 [14]

```
#In case if torch summary is not installed
from torch import nn
from torchsummary import summary
#!pip install torchsummary
```

0초 [15]

```
'''

Network : Discriminator

input : (bs, 1, 28, 28) #batch size, #of channels, #height, #width
          |                                                                            ---- SUMMARY ----
          V
Conv2d( in_channels = 1, out_channels = 16, kernel_size = (3,3), stride = 2)            #(bs, 16, 13, 13)
BatchNorm2d()                                                                          #(bs, 16, 13, 13)
LeakyReLU()                                                                            #(bs, 16, 13, 13)
          |
          V
Conv2d( in_channels = 16, out_channels = 32, kernel_size = (5,5), stride = 2)           #(bs, 32, 5, 5)
BatchNorm2d()                                                                          #(bs, 32, 5, 5)
LeakyReLU()                                                                            #(bs, 32, 5, 5)
          |
          V
Conv2d( in_channels = 32, out_channels = 64, kernel_size = (5,5), stride = 2)           #(bs, 64, 1, 1)
BatchNorm2d()                                                                          #(bs, 64, 1, 1)
LeakyReLU()                                                                            #(bs, 64, 1, 1)
          |
          V
Flatten()                                                                              #(bs, 64)
Linear(in_features = 64, out_features = 1)                                             #(bs, 1)

'''
```

[16]
```python
def get_disc_block(in_channels, out_channels, kernel_size, stride):
    return nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size, stride),
        #Conv2d: Applies a 2D convolution over an input signal composed of several input planes.
        nn.BatchNorm2d(out_channels),
        #BatchNorm2d: Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension)
        nn.LeakyReLU(0.2)
        # LeakyReLU: Applies the element-wise function
    )
```

0초 [19]
```python
class Discriminator(nn.Module):

    def __init__(self):
        super(Discriminator, self).__init__()

        self.block1 = get_disc_block(1, 16, (3,3), 2)
        self.block2 = get_disc_block(16, 32, (5,5), 2)
        self.block3 = get_disc_block(32, 64, (5,5), 2)

        self.flatten = nn.Flatten()
        self.linear = nn.Linear(in_features = 64, out_features = 1)

    def forward(self,images):

        x1 = self.block1(images)
        x2 = self.block2(x1)
        x3 = self.block3(x2)

        x4 = self.flatten(x3)
        x5 = self.linear(x4)

        return x5
```

0초

```python
D = Discriminator()
D.to(device)

summary(D, input_size = (1, 28, 28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 13, 13]             160
       BatchNorm2d-2           [-1, 16, 13, 13]              32
         LeakyReLU-3           [-1, 16, 13, 13]               0
            Conv2d-4             [-1, 32, 5, 5]          12,832
       BatchNorm2d-5             [-1, 32, 5, 5]              64
         LeakyReLU-6             [-1, 32, 5, 5]               0
            Conv2d-7             [-1, 64, 1, 1]          51,264
       BatchNorm2d-8             [-1, 64, 1, 1]             128
         LeakyReLU-9             [-1, 64, 1, 1]               0
          Flatten-10                   [-1, 64]               0
           Linear-11                    [-1, 1]              65
================================================================
Total params: 64,545
Trainable params: 64,545
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.08
Params size (MB): 0.25
Estimated Total Size (MB): 0.33
----------------------------------------------------------------
```

Create Generator Network

```
[ ] '''

    Network : Generator

    z_dim = 64
    input : (bs,z_dim)

          |
          | Reshape
          V

    input : (bs, channel, height, width) -> (bs, z_dim , 1 , 1)
          |                                                                    ---- SUMMARY ----
          V
    ConvTranspose2d( in_channels = z_dim, out_channels = 256, kernel_size = (3,3), stride = 2)        #(bs, 256, 3, 3)
    BatchNorm2d()                                                                                     #(bs, 256, 3, 3)
    ReLU()                                                                                            #(bs, 256, 3, 3)
          |
          V
    ConvTranspose2d( in_channels = 256, out_channels = 128, kernel_size = (4,4), stride = 1)          #(bs, 128, 6, 6)
    BatchNorm2d()                                                                                     #(bs, 128, 6, 6)
    ReLU()                                                                                            #(bs, 128, 6, 6)
          |
          V
    ConvTranspose2d( in_channels = 128, out_channels = 64, kernel_size = (3,3), stride = 2)           #(bs, 64, 13, 13)
    BatchNorm2d()                                                                                     #(bs, 64, 13, 13)
    ReLU()                                                                                            #(bs, 64, 13, 13)
          |
          V
    ConvTranspose2d( in_channels = 64, out_channels = 1, kernel_size = (4,4), stride = 2)             #(bs, 1, 28, 28)
    Tanh()                                                                                            #(bs, 1, 28, 28)

    '''
```

```
[ ] def get_gen_block(in_channels, out_channels, kernel_size, stride, final_block = False):
      if final_block == True:
        return nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride),
            nn.Tanh()
        )
      return nn.Sequential(
          nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride),
          nn.BatchNorm2d(out_channels),
          nn.ReLU()
      )
```

```
[ ] class Generator(nn.Module):

      def __init__(self, noise_dim):
        super(Generator,self).__init__()

        self.noise_dim = noise_dim
        self.block_1 = get_gen_block(noise_dim, 256, (3,3), 2)
        self.block_2 = get_gen_block(256, 128, (4,4), 1)
        self.block_3 = get_gen_block(128, 64, (3,3), 2)

        self.block_4 = get_gen_block(64, 1, (4,4), 2, final_block = True)

      def forward(self, r_noise_vec):

        #(bs, noise_dim) -> (bs, noise_dim, 1, 1)
        x = r_noise_vec.view(-1, self.noise_dim, 1, 1)

        x1 = self.block_1(x)
        x2 = self.block_2(x1)
        x3 = self.block_3(x2)
        x4 = self.block_4(x3)

        return x4
```

```
[ ] G = Generator(noise_dim)
    G.to(device)

    summary(G, input_size = (1,noise_dim))
```

```python
G = Generator(noise_dim)
G.to(device)

summary(G, input_size = (1,noise_dim))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
   ConvTranspose2d-1            [-1, 256, 3, 3]         147,712
       BatchNorm2d-2            [-1, 256, 3, 3]             512
              ReLU-3            [-1, 256, 3, 3]               0
   ConvTranspose2d-4            [-1, 128, 6, 6]         524,416
       BatchNorm2d-5            [-1, 128, 6, 6]             256
              ReLU-6            [-1, 128, 6, 6]               0
   ConvTranspose2d-7           [-1, 64, 13, 13]          73,792
       BatchNorm2d-8           [-1, 64, 13, 13]             128
              ReLU-9           [-1, 64, 13, 13]               0
  ConvTranspose2d-10            [-1, 1, 28, 28]           1,025
             Tanh-11            [-1, 1, 28, 28]               0
================================================================
Total params: 747,841
Trainable params: 747,841
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.42
Params size (MB): 2.85
Estimated Total Size (MB): 3.27
----------------------------------------------------------------
```

```python
# Replace Random initialized weights to Normal weights

def weights_init(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
        nn.init.normal_(m.weight, 0.0, 0.02)
    if isinstance(m, nn.BatchNorm2d):
        nn.init.normal_(m.weight, 0.0, 0.02)
        nn.init.constant_(m.bias, 0)
```

```python
# D = Discriminator , G = Generator
D = D.apply(weights_init)
G = G.apply(weights_init)
```

Create Loss Function and Load Optimizer

```
[ ]  def real_loss(disc_pred):
       criterion = nn.BCEWithLogitsLoss()
       ground_truth = torch.ones_like(disc_pred)
       loss = criterion(disc_pred, ground_truth)
       return loss

     def fake_loss(disc_pred):
       criterion = nn.BCEWithLogitsLoss()
       ground_truth = torch.zeros_like(disc_pred)
       loss = criterion(disc_pred, ground_truth)
       return loss
```

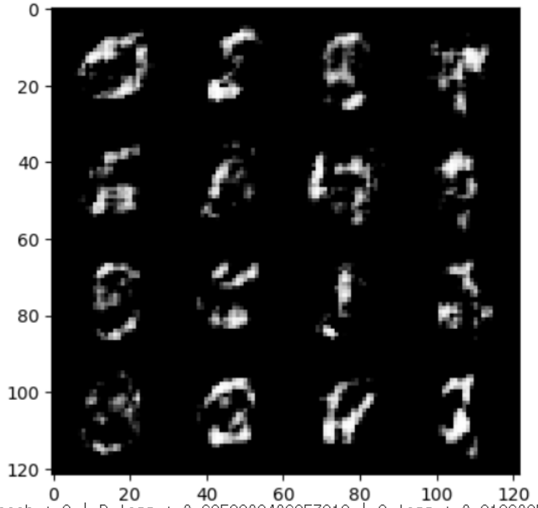
```
[ ]  D_opt = torch.optim.Adam(D.parameters(), lr = lr, betas = (beta_1, beta_2))
     G_opt = torch.optim.Adam(G.parameters(), lr = lr, betas = (beta_1, beta_2))
```

Training Loop

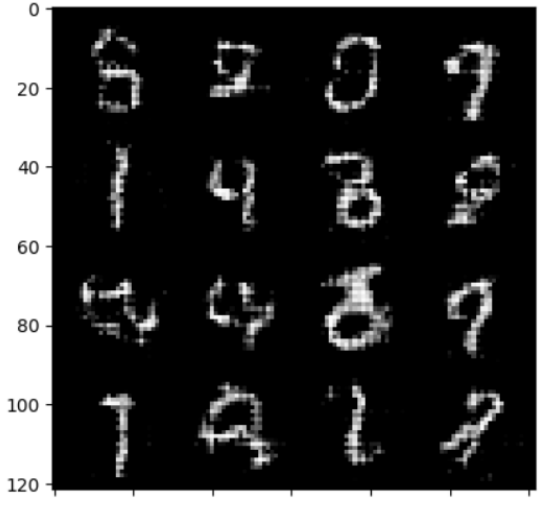
```
for i in range(epochs):

  total_d_loss = 0.0
  total_g_loss = 0.0

  for real_img, _ in tqdm(trainloader):

    real_img = real_img.to(device)
    noise = torch.randn(batch_size, noise_dim, device = device)

    #find loss and update weight for D

    D_opt.zero_grad()

    fake_img = G(noise)
    D_pred = D(fake_img)
    D_fake_loss = fake_loss(D_pred)

    D_pred = D(real_img)
    D_real_loss = real_loss(D_pred)

    D_loss = (D_fake_loss + D_real_loss) / 2

    total_d_loss += D_loss.item()

    D_loss.backward()
    D_opt.step()

    #find loss and update weights for G
    G_opt.zero_grad()

    noise = torch.randn(batch_size, noise_dim, device=device)

    fake_img = G(noise)
    D_pred = D(fake_img)
    G_loss = real_loss(D_pred)

    total_g_loss += G_loss.item()
```

```python
        G_loss.backward()
        G_opt.step()

    avg_d_loss = total_d_loss / len(trainloader)
    avg_g_loss = total_g_loss / len(trainloader)

    print("Epoch : {} | D_loss : {} | G_loss : {}".format(i+1, avg_d_loss, avg_g_loss))

    show_tensor_images(fake_img)
```

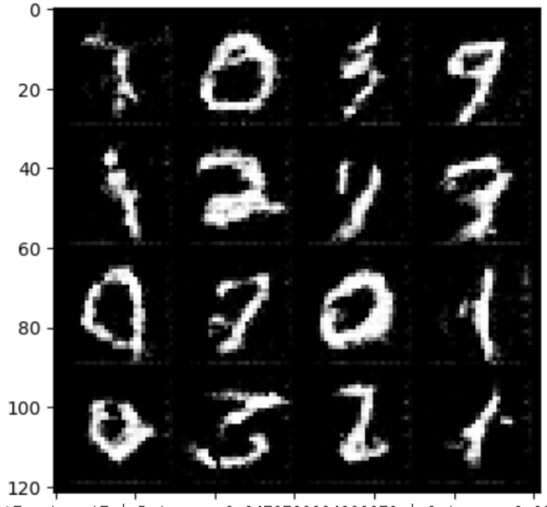
Epoch : 1 | D_loss : 0.6836742739687597 | G_loss : 0.6847326618267



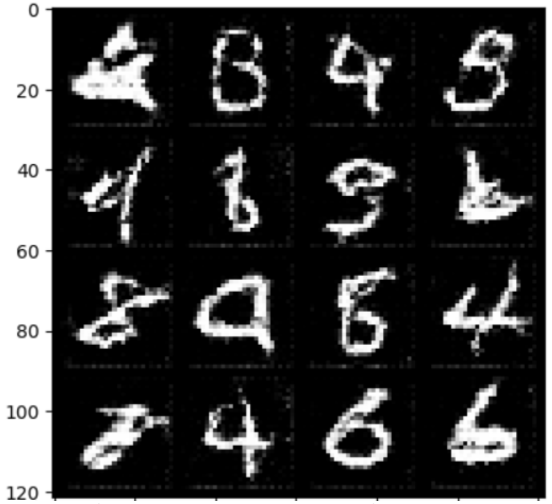
Epoch : 3 | D_loss : 0.634563537040499 | G_loss : 0.7607166955



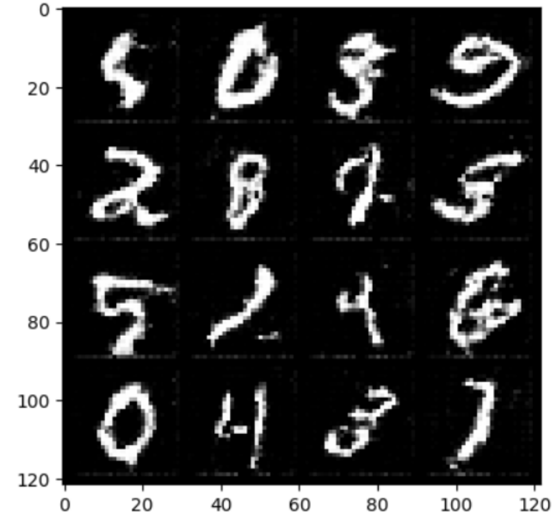
Epoch : 5 | D_loss : 0.6101203947179099 | G_loss : 0.80450796623473



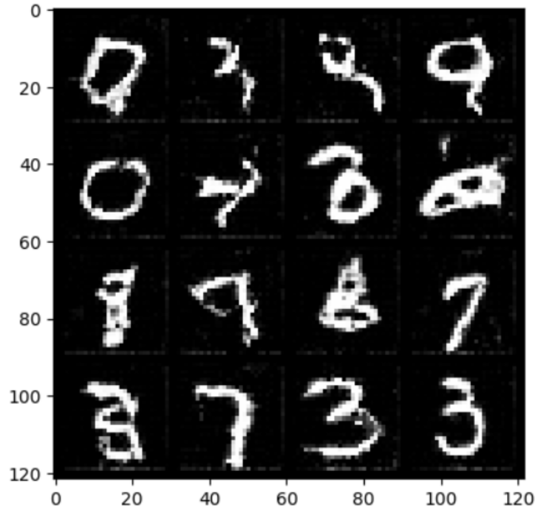
Epoch : 7 | D_loss : 0.6363723483929502 | G_loss : 0.7902455578989057



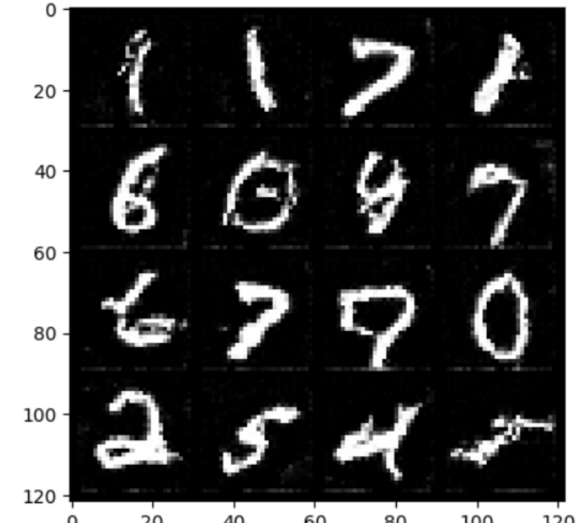
Epoch : 9 | D_loss : 0.6259902406857212 | G_loss : 0.812608504473



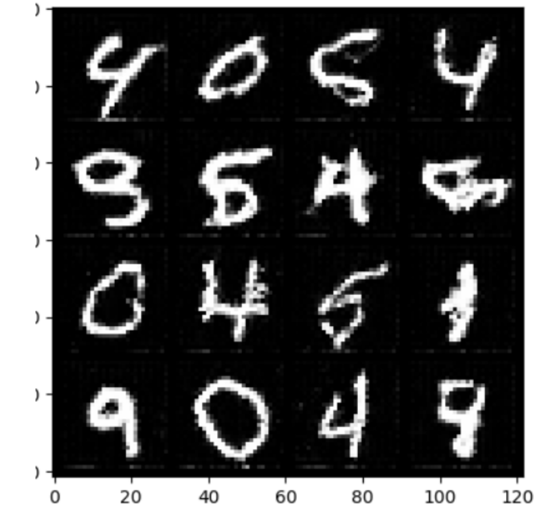
Epoch : 11 | D_loss : 0.6403779754760677 | G_loss : 0.80264810699



Epoch : 17 | D_loss : 0.6459560324388273 | G_loss : 0.801471571551203

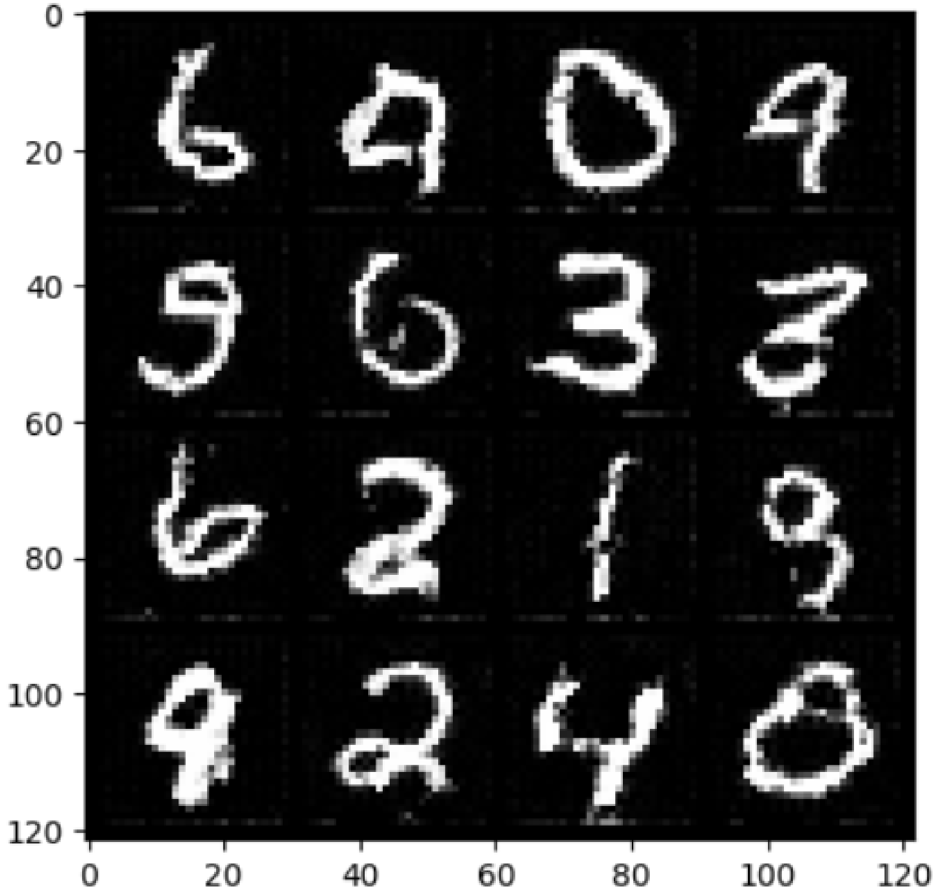


: 20 | D_loss : 0.6421407586984289 | G_loss : 0.8087291237133652



```python
# Run after training is completed.
# Now you can use Generator Network to generate handwritten images

noise = torch.randn(batch_size, noise_dim, device = device)
generated_image = G(noise)

show_tensor_images(generated_image)
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



Conclusion

- **Definition of Generator and Discriminator.**
- Generator: generating fake image from random noise.
- Discriminator: distinguishing whether it is fake or real image from random noise or training set.

- **As they get trained…**
- As the model trained, Generator can generate better quality image which seems more real.
- As the model trained, Discriminator can distinguish fake and real images more elaborately.

- The Colab link of this project.
- https://colab.research.google.com/drive/1qh6cV_gmwyMOTl6imm_dQiUqM2ZYL4HR?usp=sharing