CV2_HW4_dh3027

April 18, 2022

0.1 GANs : Generative Adversarial Networks

Image from here

A generative adversarial network (GAN) is a generative model composed of two neural networks: a generator and a discriminator. These two networks are trained in unsupervised way via competition. The generator creates ”realistic” fake images from random noise to fool the discriminator, while the discriminator evaluates the given image for authenticity. The loss function that the generator wants to minimize and the discriminator to maximize is as follows:

min G max D L(D, G) = Ex pdata(x)[log D(x)] + Ez pz(z)[log(1 − D(G(z)))]

Here, G and D are the generator and the discriminator. The first and second term of the loss represent the correct prediction of the discriminator on the real images and on the fake images respectively.

0.2 DCGAN

- You will implement deep convolutional GAN model on the MNIST dataset with Pytorch. The input image size is 28 x 28.
- The details of the generator of DCGAN is described below.
- You will start with batch size of 128, input noise of 100 dimension and Adam optimizer with learning rate of 2e-4. You may vary these hyperparameters for better performance.

0.3 Architectures

Generator:

The goal for the generator is to use layers such as convolution, maybe also upsampling layer/transposedConvolution to produce image from the given input noise vector. As this is DC-GAN (deep convolutional GAN), we expect you to use convolution in the generator. You will get full credit if you can produce `[batchsize, 1, 28, 28]` vector (image) from the given `[batchsize, 100, 1, 1]` vector (noise).

Linear Layers that you may use:

- torch.nn.Conv2d
- torch.nn.UpsamplingBilinear2d
- torch.nn.ConvTranspose2d

Non-linear layer:

- torch.nn.LeakyReLU with slope=0.2 between all linear layers.

- torch.nn.Tanh for the last layer's activation. Can you explain why do we need this in the code comment?

You may use `view` to change the vector size: https://pytorch.org/docs/stable/generated/torch.Tensor.view.html

We recommend to use 2 Conv/TransposedConv layers. When you are increasing the feature map size, considering upsample the feature by a factor of 2 each time. If you have width of 7 in one of your feature map, to get output with width of 28, you can do upsampling with factor of 2 and upsampling 2 times.

Discriminator:

You will get full credit if you can produce an output of `[batchsize, 1]` vector (image) from the given input `[batchsize, 1, 28, 28]` vector (noise).

Linear Layers that you may use:

- torch.nn.Conv2d

- torch.nn.Linear

Non-linear Layers:

- torch.nn.LeakyReLU with slope=0.2 between all linear layers.

- torch.nn.Sigmoid for the last layer's activation. Can you explain why do we need this in the code comment?

Use Leaky ReLu as the activation function between all layers, except after the last layer use Sigmoid.

You may use `view` to change the vector size: https://pytorch.org/docs/stable/generated/torch.Tensor.view.html

As an example, you may use 2 convolution layer and one linear layer in the discriminator, you can also use other setup. Note that instead of using pooling to downsampling, you may also use stride=2 in convolution to downsample the feature.

[19]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
from torchvision.utils import save_image
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
import numpy as np
from torch.optim.lr_scheduler import StepLR
import torchvision.utils as vutils
from torch.utils.data import DataLoader, TensorDataset
from scipy import linalg
from scipy.stats import entropy
```

```python
import tqdm
import cv2
# image input size
image_size=28

# Setting up transforms to resize and normalize
transform=transforms.Compose([
                                transforms.ToTensor(),
                                ])
# batchsize of dataset
batch_size = 100

# Load MNIST Dataset
gan_train_dataset = datasets.MNIST(root='./MNIST/', train=True,␣
 ↪transform=transform, download=True)
gan_train_loader = torch.utils.data.DataLoader(dataset=gan_train_dataset,␣
 ↪batch_size=batch_size, shuffle=True)
```

0.4 Model Definition (TODO)

[20]:
```python
class DCGAN_Generator(nn.Module):
    def __init__(self):
        super(DCGAN_Generator,self).__init__()

        ################################
        # Please fill in your code here:
        ################################

        self.layers = nn.Sequential(
            nn.ConvTranspose2d(100, 128, 4, 1, 0, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(128, 256, 4, 2, 0, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(256, 128, 4, 2, 2, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(128, 64, 2, 2, 2, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(64, 1, 1, 1, 2, bias=False),
            nn.Tanh()
        )
```

```python
    def forward(self, input):

        ################################
        # Please fill in your code here:
        ################################
        out = self.layers(input)

        # Explain why Tanh is needed for the last layer
        # The step is to normalize the input from large range to [-1,1] in case␣
↪of
        # minimal information cannot be captured by the next discriminator. And␣
↪use
        # Tanh rather than sigmoid can prevent gradients vanishing.

        return out


class DCGAN_Discriminator(nn.Module):
    def __init__(self):
        super(DCGAN_Discriminator, self).__init__()
        ################################
        # Please fill in your code here:
        ################################
        # Reference from pytorch tutorial
        self.layers = nn.Sequential(
            nn.Conv2d(1, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.Conv2d(128, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2),
            nn.Conv2d(512, 1, 4, 2, 1, bias=False),
            nn.Flatten(),
            nn.Linear(1,1),
            nn.Sigmoid()
        )

    def forward(self, input):


        ################################
        # Please fill in your code here:
        ################################
```

```
        out = self.layers(input)
        # Explain why Sigmoid is needed for the last layer
        # The step is to normalize the input to [0,1] in case of minimal␣
 ↪information
        # cannot be make use of decision function with outputs 'fake' and␣
 ↪'Real'.
        # In classification, we use Sigmoid.

        return out


# Code that check size
g=DCGAN_Generator()
batchsize=2
z=torch.zeros((batchsize, 100, 1, 1))
out = g(z)
print(out.size()) # You should expect size [batchsize, 1, 28, 28]



d=DCGAN_Discriminator()
x=torch.zeros((batchsize, 1, 28, 28))
out = d(x)
print(out.size()) # You should expect size [batchsize, 1]
```

```
torch.Size([2, 1, 28, 28])
torch.Size([2, 1])
```

GAN loss (TODO)

[21]:
```
import torch

def loss_discriminator(D, real, G, noise, Valid_label, Fake_label, criterion,␣
 ↪optimizerD):

    '''
    1. Forward real images into the discriminator
    2. Compute loss between Valid_label and dicriminator output on real images
    3. Forward noise into the generator to get fake images
    4. Forward fake images to the discriminator
    5. Compute loss between Fake_label and discriminator output on fake images␣
 ↪(and remember to detach the gradient from the fake images using detach()!)
    6. sum real loss and fake loss as the loss_D
    7. we also need to output fake images generate by G(noise) for␣
 ↪loss_generator computation
    '''
    ################################
```

```
    # Please fill in your code here:
    outputs = D(real).view(-1)
    real_loss = criterion(outputs, Valid_label)
    real_loss.backward()
    fake_imgs = G(noise)
    out = D(fake_imgs.detach()).view(-1)
    fake_loss = criterion(out, Fake_label)
    fake_loss.backward()
    loss_D = real_loss + fake_loss
    ################################


    return loss_D, fake_imgs

def loss_generator(netD, netG, fake, Valid_label, criterion, optimizerG):
    '''
    1. Forward fake images to the discriminator
    2. Compute loss between valid labels and discriminator output on fake images
    '''

    ################################
    # Please fill in your code here:
    outputs = netD(fake).view(-1)
    loss_G = criterion(outputs, Valid_label)
    loss_G.backward()
    ################################

    return loss_G
```

[22]:

```
import torchvision.utils as vutils
from torch.optim.lr_scheduler import StepLR
import pdb

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Number of channels
nc = 3
# Size of z latent vector (i.e. size of generator input)
nz = 100

netG = DCGAN_Generator().to(device)
netD = DCGAN_Discriminator().to(device)

from torchsummary import summary
print(summary(netG,(100,1,1)))
print(summary(netD,(1, 28, 28)))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
   ConvTranspose2d-1            [-1, 128, 4, 4]         204,800
       BatchNorm2d-2            [-1, 128, 4, 4]             256
         LeakyReLU-3            [-1, 128, 4, 4]               0
   ConvTranspose2d-4          [-1, 256, 10, 10]         524,288
       BatchNorm2d-5          [-1, 256, 10, 10]             512
         LeakyReLU-6          [-1, 256, 10, 10]               0
   ConvTranspose2d-7          [-1, 128, 18, 18]         524,288
       BatchNorm2d-8          [-1, 128, 18, 18]             256
         LeakyReLU-9          [-1, 128, 18, 18]               0
  ConvTranspose2d-10           [-1, 64, 32, 32]          32,768
      BatchNorm2d-11           [-1, 64, 32, 32]             128
        LeakyReLU-12           [-1, 64, 32, 32]               0
  ConvTranspose2d-13            [-1, 1, 28, 28]              64
             Tanh-14            [-1, 1, 28, 28]               0
================================================================
Total params: 1,287,360
Trainable params: 1,287,360
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 3.09
Params size (MB): 4.91
Estimated Total Size (MB): 8.01
----------------------------------------------------------------
None
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 14, 14]           1,024
       BatchNorm2d-2           [-1, 64, 14, 14]             128
         LeakyReLU-3           [-1, 64, 14, 14]               0
            Conv2d-4            [-1, 128, 7, 7]         131,072
       BatchNorm2d-5            [-1, 128, 7, 7]             256
         LeakyReLU-6            [-1, 128, 7, 7]               0
            Conv2d-7            [-1, 512, 3, 3]       1,048,576
       BatchNorm2d-8            [-1, 512, 3, 3]           1,024
         LeakyReLU-9            [-1, 512, 3, 3]               0
           Conv2d-10              [-1, 1, 1, 1]           8,192
          Flatten-11                    [-1, 1]               0
           Linear-12                    [-1, 1]               2
          Sigmoid-13                    [-1, 1]               0
================================================================
Total params: 1,190,274
Trainable params: 1,190,274
Non-trainable params: 0
```

```
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.54
Params size (MB): 4.54
Estimated Total Size (MB): 5.08
----------------------------------------------------------------
None
```

TRAINING

[23]:
```python
import torchvision.utils as vutils
from torch.optim.lr_scheduler import StepLR
import pdb

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Number of channels
nc = 3
# Size of z latent vector (i.e. size of generator input)
nz = 100

# Create the generator and discriminator
netG = DCGAN_Generator().to(device)
netD = DCGAN_Discriminator().to(device)

# Initialize BCELoss function
criterion = nn.BCELoss()

# Create latent vector to test the generator performance
fixed_noise = torch.randn(36, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1
fake_label = 0

learning_rate = 0.0002
beta1 = 0.5

# Setup Adam optimizers for both G and D

#################################
# Please fill in your code here:

optimizerD = optim.Adam(netD.parameters(), lr=learning_rate, betas=(beta1, 0.
↪999))
optimizerG = optim.Adam(netG.parameters(), lr=learning_rate, betas=(beta1, 0.
↪999))
```

```
################################



img_list = []
real_img_list = []
G_losses = []
D_losses = []
iters = 0
num_epochs = 10


def load_param(num_eps):
  model_saved = torch.load('/content/gan_{}.pt'.format(num_eps))
  netG.load_state_dict(model_saved['netG'])
  netD.load_state_dict(model_saved['netD'])

# GAN Training Loop
for epoch in range(num_epochs):
    for i, data in enumerate(gan_train_loader, 0):
        real = data[0].to(device)
        b_size = real.size(0)
        noise = torch.randn(b_size, nz, 1, 1, device=device)

        Valid_label = torch.full((b_size,), real_label, dtype=torch.float,␣
 ↪device=device)
        Fake_label = torch.full((b_size,), fake_label, dtype=torch.float,␣
 ↪device=device)

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################

        ################################
        # Please fill in your code here:
        netD.zero_grad()
        loss_D, fake_imgs = loss_discriminator(netD, real, netG, noise,␣
 ↪Valid_label, Fake_label, criterion, optimizerD)
        optimizerD.step()
        ################################

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################

        ################################
        # Please fill in your code here:
```

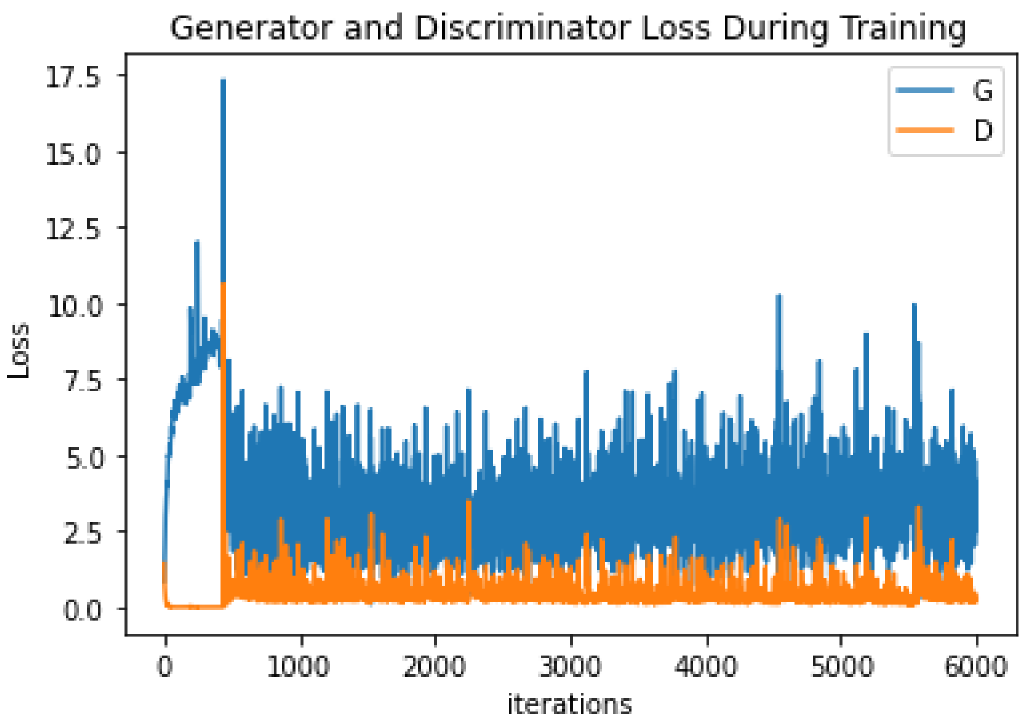
```
        netG.zero_grad()
        loss_G = loss_generator(netD, netG, fake_imgs, Valid_label, criterion,␣
↪optimizerG)
        optimizerG.step()
        ################################

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\t'
                  % (epoch, num_epochs, i, len(gan_train_loader),
                     loss_D.item(), loss_G.item()))

        # Save Losses for plotting later
        G_losses.append(loss_G.item())
        D_losses.append(loss_D.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==␣
↪len(gan_train_loader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1



plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()


checkpoint = {'netG': netG.state_dict(),
              'netD': netD.state_dict()}
torch.save(checkpoint, 'gan_{}.pt'.format(num_epochs))
```

```
[0/10][0/600]   Loss_D: 1.4395  Loss_G: 0.8541
[0/10][50/600]  Loss_D: 0.0224  Loss_G: 5.5533
[0/10][100/600] Loss_D: 0.0036  Loss_G: 6.5800
[0/10][150/600] Loss_D: 0.0019  Loss_G: 7.1339
[0/10][200/600] Loss_D: 0.0157  Loss_G: 8.5549
[0/10][250/600] Loss_D: 0.0021  Loss_G: 9.4847
[0/10][300/600] Loss_D: 0.0037  Loss_G: 9.3039
```

```
[0/10][350/600] Loss_D: 0.0007  Loss_G: 8.7520
[0/10][400/600] Loss_D: 0.0009  Loss_G: 8.6666
[0/10][450/600] Loss_D: 1.0989  Loss_G: 5.1204
[0/10][500/600] Loss_D: 0.3220  Loss_G: 3.9213
[0/10][550/600] Loss_D: 0.6191  Loss_G: 2.2830
[1/10][0/600]   Loss_D: 0.3472  Loss_G: 2.8101
[1/10][50/600]  Loss_D: 0.2995  Loss_G: 2.0540
[1/10][100/600] Loss_D: 0.3558  Loss_G: 2.5007
[1/10][150/600] Loss_D: 1.0873  Loss_G: 6.6616
[1/10][200/600] Loss_D: 0.3044  Loss_G: 3.8561
[1/10][250/600] Loss_D: 0.2065  Loss_G: 2.8666
[1/10][300/600] Loss_D: 0.7712  Loss_G: 6.0092
[1/10][350/600] Loss_D: 0.2105  Loss_G: 2.6980
[1/10][400/600] Loss_D: 0.3247  Loss_G: 3.1079
[1/10][450/600] Loss_D: 0.4064  Loss_G: 2.2099
[1/10][500/600] Loss_D: 0.3653  Loss_G: 3.6658
[1/10][550/600] Loss_D: 0.7750  Loss_G: 3.5491
[2/10][0/600]   Loss_D: 0.2083  Loss_G: 2.7046
[2/10][50/600]  Loss_D: 0.4085  Loss_G: 1.9068
[2/10][100/600] Loss_D: 0.3106  Loss_G: 2.8062
[2/10][150/600] Loss_D: 0.4981  Loss_G: 4.3313
[2/10][200/600] Loss_D: 0.4082  Loss_G: 2.2721
[2/10][250/600] Loss_D: 0.5657  Loss_G: 1.9689
[2/10][300/600] Loss_D: 0.3133  Loss_G: 2.9004
[2/10][350/600] Loss_D: 0.4807  Loss_G: 2.7801
[2/10][400/600] Loss_D: 0.5428  Loss_G: 1.6733
[2/10][500/600] Loss_D: 0.1836  Loss_G: 2.8979
[2/10][550/600] Loss_D: 0.6466  Loss_G: 2.8735
[3/10][0/600]   Loss_D: 0.3426  Loss_G: 2.5990
[3/10][50/600]  Loss_D: 1.1325  Loss_G: 5.9731
[3/10][100/600] Loss_D: 0.3000  Loss_G: 2.0274
[3/10][150/600] Loss_D: 0.6136  Loss_G: 3.0581
[3/10][200/600] Loss_D: 1.2000  Loss_G: 4.4493
[3/10][250/600] Loss_D: 0.5336  Loss_G: 2.2393
[3/10][300/600] Loss_D: 0.2679  Loss_G: 2.5504
[3/10][350/600] Loss_D: 0.2076  Loss_G: 2.8943
[3/10][400/600] Loss_D: 0.2621  Loss_G: 2.7931
[3/10][450/600] Loss_D: 0.5845  Loss_G: 2.0853
[3/10][500/600] Loss_D: 1.1475  Loss_G: 3.7529
[3/10][550/600] Loss_D: 0.4137  Loss_G: 3.5502
[4/10][0/600]   Loss_D: 0.4417  Loss_G: 1.5140
[4/10][50/600]  Loss_D: 0.3532  Loss_G: 2.8051
[4/10][100/600] Loss_D: 0.2610  Loss_G: 2.4036
[4/10][150/600] Loss_D: 0.8951  Loss_G: 3.4137
[4/10][200/600] Loss_D: 0.3779  Loss_G: 2.5432
[4/10][250/600] Loss_D: 0.3277  Loss_G: 2.8436
[4/10][300/600] Loss_D: 0.2879  Loss_G: 2.5572
[4/10][350/600] Loss_D: 0.2485  Loss_G: 3.4861
```

```
[4/10][400/600] Loss_D: 0.6790  Loss_G: 1.5138
[4/10][450/600] Loss_D: 0.2840  Loss_G: 3.3710
[4/10][500/600] Loss_D: 0.4320  Loss_G: 2.0600
[4/10][550/600] Loss_D: 0.3722  Loss_G: 1.7186
[5/10][0/600]   Loss_D: 0.3225  Loss_G: 3.4825
[5/10][50/600]  Loss_D: 0.4497  Loss_G: 3.5879
[5/10][100/600] Loss_D: 0.2294  Loss_G: 3.1065
[5/10][150/600] Loss_D: 0.4005  Loss_G: 3.5037
[5/10][200/600] Loss_D: 0.4449  Loss_G: 1.8653
[5/10][250/600] Loss_D: 0.5173  Loss_G: 3.6596
[5/10][300/600] Loss_D: 0.5166  Loss_G: 1.4815
[5/10][350/600] Loss_D: 0.2562  Loss_G: 2.3746
[5/10][400/600] Loss_D: 0.2232  Loss_G: 2.8050
[5/10][450/600] Loss_D: 0.3722  Loss_G: 3.1250
[5/10][500/600] Loss_D: 0.2260  Loss_G: 2.9418
[5/10][550/600] Loss_D: 0.2482  Loss_G: 2.2199
[6/10][0/600]   Loss_D: 0.3549  Loss_G: 2.3177
[6/10][50/600]  Loss_D: 0.6400  Loss_G: 3.6243
[6/10][100/600] Loss_D: 0.4446  Loss_G: 2.6129
[6/10][150/600] Loss_D: 0.2803  Loss_G: 3.7779
[6/10][200/600] Loss_D: 0.6343  Loss_G: 2.9019
[6/10][250/600] Loss_D: 0.7729  Loss_G: 2.2620
[6/10][300/600] Loss_D: 0.2823  Loss_G: 3.3544
[6/10][350/600] Loss_D: 0.3648  Loss_G: 2.5363
[6/10][400/600] Loss_D: 0.4429  Loss_G: 2.9459
[6/10][450/600] Loss_D: 0.3532  Loss_G: 3.1216
[6/10][500/600] Loss_D: 0.2870  Loss_G: 2.1404
[6/10][550/600] Loss_D: 0.3004  Loss_G: 2.5827
[7/10][0/600]   Loss_D: 0.3168  Loss_G: 3.2149
[7/10][50/600]  Loss_D: 0.3003  Loss_G: 3.0593
[7/10][100/600] Loss_D: 0.1506  Loss_G: 5.0919
[7/10][150/600] Loss_D: 0.2250  Loss_G: 4.0048
[7/10][200/600] Loss_D: 0.2623  Loss_G: 3.2852
[7/10][250/600] Loss_D: 0.4737  Loss_G: 3.9448
[7/10][300/600] Loss_D: 0.3490  Loss_G: 4.1689
[7/10][350/600] Loss_D: 0.9398  Loss_G: 4.0711
[7/10][400/600] Loss_D: 0.5688  Loss_G: 1.7997
[7/10][450/600] Loss_D: 0.3657  Loss_G: 4.6120
[7/10][500/600] Loss_D: 0.1903  Loss_G: 2.9295
[7/10][550/600] Loss_D: 0.2824  Loss_G: 3.8851
[8/10][0/600]   Loss_D: 0.2974  Loss_G: 3.6058
[8/10][50/600]  Loss_D: 0.5625  Loss_G: 3.7612
[8/10][100/600] Loss_D: 0.2362  Loss_G: 2.7925
[8/10][150/600] Loss_D: 0.4387  Loss_G: 3.5062
[8/10][200/600] Loss_D: 0.2416  Loss_G: 2.5670
[8/10][250/600] Loss_D: 0.2513  Loss_G: 3.4173
[8/10][300/600] Loss_D: 0.2815  Loss_G: 3.3367
[8/10][350/600] Loss_D: 0.9523  Loss_G: 2.4099
```

```
[8/10][400/600] Loss_D: 0.8361  Loss_G: 2.3761
[8/10][450/600] Loss_D: 0.3963  Loss_G: 2.2499
[8/10][500/600] Loss_D: 0.3237  Loss_G: 2.5145
[8/10][550/600] Loss_D: 0.2122  Loss_G: 3.1745
[9/10][0/600]   Loss_D: 0.2427  Loss_G: 2.8601
[9/10][50/600]  Loss_D: 0.1423  Loss_G: 4.2237
[9/10][100/600] Loss_D: 0.1019  Loss_G: 4.2023
[9/10][150/600] Loss_D: 0.3132  Loss_G: 2.4446
[9/10][200/600] Loss_D: 1.0305  Loss_G: 3.1943
[9/10][250/600] Loss_D: 0.3721  Loss_G: 2.8814
[9/10][300/600] Loss_D: 0.5153  Loss_G: 3.1722
[9/10][350/600] Loss_D: 0.3613  Loss_G: 3.5833
[9/10][400/600] Loss_D: 0.1698  Loss_G: 3.7078
[9/10][450/600] Loss_D: 0.4006  Loss_G: 3.3135
[9/10][500/600] Loss_D: 0.2402  Loss_G: 3.2695
[9/10][550/600] Loss_D: 0.1962  Loss_G: 2.6098
```



0.5 Qualitative Visualisations

[24]:
```
# Test GAN on a random sample and display on 6X6 grid
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8,8))
```

```python
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000,␣
 ↪blit=True)

HTML(ani.to_jshtml())
```

[24]: <IPython.core.display.HTML object>