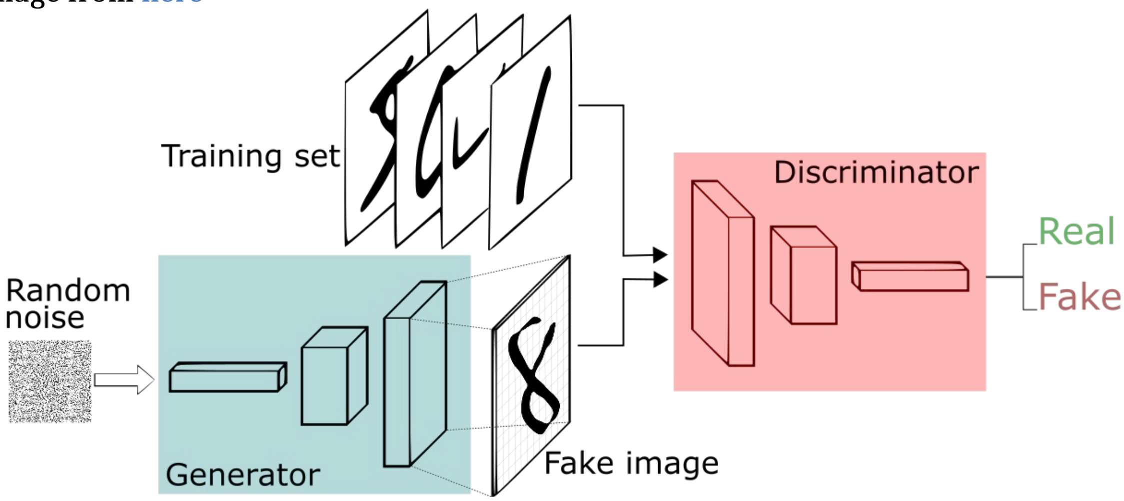
GANs : Generative Adversarial Networks

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.

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.

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 DCGAN (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.

```python
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
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)
```

Model Definition (TODO)

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

        ################################
        # Please fill in your code here:
        ################################
        self.nc = 1
        self.nz = 100
        self.ngf = 32
        self.network = nn.Sequential(
            nn.ConvTranspose2d(in_channels=self.nz,
out_channels=self.ngf * 8, kernel_size=11, stride=1, padding=0,
bias=False),
```

```
            nn.BatchNorm2d(self.ngf * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(in_channels=self.ngf*8,
out_channels=self.ngf * 4, kernel_size=9, stride=1, padding=0,
bias=False),
            nn.BatchNorm2d(self.ngf * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d(in_channels=self.ngf * 4,
out_channels=self.ngf*2, kernel_size=7, stride=1, padding=0,
bias=False),
            nn.BatchNorm2d(self.ngf*2),
            nn.ReLU(True),
            nn.ConvTranspose2d(in_channels=self.ngf*2 ,
out_channels=1, kernel_size=4, stride=1, padding=0, bias=False),
            nn.Tanh()
        )

    def forward(self, input):

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

        out = self.network(input)

        # Explain why Tanh is needed for the last layer
        '''
        Using a bounded activation function such as Tanh or Sigmoid
would allow the model to learn more quickly and have a
        better color saturation of the training distribution for
genrated images i.e the genrated images would
        appear more realistic.
        Source: UNSUPERVISED REPRESENTATION LEARNING WITH DEEP
CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS
        https://arxiv.org/pdf/1511.06434.pdf]
        '''
        return out


class DCGAN_Discriminator(nn.Module):
    def __init__(self):
        super(DCGAN_Discriminator, self).__init__()
        ################################
        # Please fill in your code here:
        ################################
        self.ndf = 32
        self.nc = 1
        self.network = nn.Sequential(
                nn.Conv2d(in_channels=self.nc,
out_channels=self.ndf*8, kernel_size=4, stride=1, padding=0,
```

```
bias=False),
                nn.BatchNorm2d(self.ndf*8),
                nn.Conv2d(in_channels=self.ndf*8,
out_channels=self.ndf*4, kernel_size=7, stride=1, padding=0,
bias=False),
                nn.BatchNorm2d(self.ndf*4),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Conv2d(in_channels=self.ndf*4,
out_channels=self.ndf * 2, kernel_size=9, stride=1, padding=0,
bias=False),
                nn.BatchNorm2d(self.ndf * 2),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Conv2d(in_channels=self.ndf * 2,
out_channels=self.nc, kernel_size=11, stride=1, padding=0,
bias=False),
                nn.Sigmoid()
            )

    def forward(self, input):

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

        out = self.network(input)
        out = out.view(-1, 1)

        # Explain why Sigmoid is needed for the last layer
        """
        As the output of the discriminator needs to be probablities of
each class we use a sigmod function which
        brings the value of its input between 0 and 1.
        """
        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)

```python
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:
    ################################

    pred_real = D(real)
    loss_real = criterion(pred_real.squeeze(1), Valid_label)
    fake_imgs = G(noise)
    pred_fake = D(fake_imgs.detach())
    loss_fake = criterion(pred_fake.squeeze(1), Fake_label)
    loss_D = loss_real + loss_fake

    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:
    ################################

    pred = netD(fake)
    loss_G = criterion(pred.squeeze(1), Valid_label)
```

```
    return loss_G

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:depth-idx)                   Output Shape
Param #
======================================================================
====================
├─Sequential: 1-1                        [-1, 1, 28, 28]           --
|    └─ConvTranspose2d: 2-1              [-1, 256, 11, 11]
3,097,600
|    └─BatchNorm2d: 2-2                  [-1, 256, 11, 11]         512
|    └─ReLU: 2-3                         [-1, 256, 11, 11]         --
|    └─ConvTranspose2d: 2-4              [-1, 128, 19, 19]
2,654,208
|    └─BatchNorm2d: 2-5                  [-1, 128, 19, 19]         256
|    └─ReLU: 2-6                         [-1, 128, 19, 19]         --
|    └─ConvTranspose2d: 2-7              [-1, 64, 25, 25]
401,408
|    └─BatchNorm2d: 2-8                  [-1, 64, 25, 25]          128
|    └─ReLU: 2-9                         [-1, 64, 25, 25]          --
|    └─ConvTranspose2d: 2-10             [-1, 1, 28, 28]
1,024
|    └─Tanh: 2-11                        [-1, 1, 28, 28]           --
======================================================================
====================
Total params: 6,155,136
Trainable params: 6,155,136
Non-trainable params: 0
Total mult-adds (G): 1.59
======================================================================
```

```
====================
Input size (MB): 0.00
Forward/backward pass size (MB): 1.79
Params size (MB): 23.48
Estimated Total Size (MB): 25.27
======================================================================
====================
======================================================================
====================
Layer (type:depth-idx)                   Output Shape
Param #
======================================================================
====================
├─Sequential: 1-1                        [-1, 1, 28, 28]           --
|    └─ConvTranspose2d: 2-1              [-1, 256, 11, 11]
3,097,600
|    └─BatchNorm2d: 2-2                  [-1, 256, 11, 11]         512
|    └─ReLU: 2-3                         [-1, 256, 11, 11]         --
|    └─ConvTranspose2d: 2-4              [-1, 128, 19, 19]
2,654,208
|    └─BatchNorm2d: 2-5                  [-1, 128, 19, 19]         256
|    └─ReLU: 2-6                         [-1, 128, 19, 19]         --
|    └─ConvTranspose2d: 2-7              [-1, 64, 25, 25]
401,408
|    └─BatchNorm2d: 2-8                  [-1, 64, 25, 25]          128
|    └─ReLU: 2-9                         [-1, 64, 25, 25]          --
|    └─ConvTranspose2d: 2-10             [-1, 1, 28, 28]
1,024
|    └─Tanh: 2-11                        [-1, 1, 28, 28]           --
======================================================================
====================
Total params: 6,155,136
Trainable params: 6,155,136
Non-trainable params: 0
Total mult-adds (G): 1.59
======================================================================
====================
Input size (MB): 0.00
Forward/backward pass size (MB): 1.79
Params size (MB): 23.48
Estimated Total Size (MB): 25.27
======================================================================
====================
======================================================================
====================
Layer (type:depth-idx)                   Output Shape
Param #
======================================================================
====================
├─Sequential: 1-1                        [-1, 1, 1, 1]             --
```

```
|    └─Conv2d: 2-1                        [-1, 256, 25, 25]
4,096
|    └─BatchNorm2d: 2-2                   [-1, 256, 25, 25]         512
|    └─Conv2d: 2-3                        [-1, 128, 19, 19]
1,605,632
|    └─BatchNorm2d: 2-4                   [-1, 128, 19, 19]         256
|    └─LeakyReLU: 2-5                     [-1, 128, 19, 19]         --
|    └─Conv2d: 2-6                        [-1, 64, 11, 11]
663,552
|    └─BatchNorm2d: 2-7                   [-1, 64, 11, 11]          128
|    └─LeakyReLU: 2-8                     [-1, 64, 11, 11]          --
|    └─Conv2d: 2-9                        [-1, 1, 1, 1]
7,744
|    └─Sigmoid: 2-10                      [-1, 1, 1, 1]             --
======================================================================
====================
Total params: 2,281,920
Trainable params: 2,281,920
Non-trainable params: 0
Total mult-adds (M): 664.77
======================================================================
====================
Input size (MB): 0.00
Forward/backward pass size (MB): 3.26
Params size (MB): 8.70
Estimated Total Size (MB): 11.97
======================================================================
====================
======================================================================
====================
Layer (type:depth-idx)                    Output Shape
Param #
======================================================================
====================
├─Sequential: 1-1                         [-1, 1, 1, 1]             --
|    └─Conv2d: 2-1                        [-1, 256, 25, 25]
4,096
|    └─BatchNorm2d: 2-2                   [-1, 256, 25, 25]         512
|    └─Conv2d: 2-3                        [-1, 128, 19, 19]
1,605,632
|    └─BatchNorm2d: 2-4                   [-1, 128, 19, 19]         256
|    └─LeakyReLU: 2-5                     [-1, 128, 19, 19]         --
|    └─Conv2d: 2-6                        [-1, 64, 11, 11]
663,552
|    └─BatchNorm2d: 2-7                   [-1, 64, 11, 11]          128
|    └─LeakyReLU: 2-8                     [-1, 64, 11, 11]          --
|    └─Conv2d: 2-9                        [-1, 1, 1, 1]
7,744
|    └─Sigmoid: 2-10                      [-1, 1, 1, 1]             --
======================================================================
```

```
====================
Total params: 2,281,920
Trainable params: 2,281,920
Non-trainable params: 0
Total mult-adds (M): 664.77
======================================================================
====================
Input size (MB): 0.00
Forward/backward pass size (MB): 3.26
Params size (MB): 8.70
Estimated Total Size (MB): 11.97
======================================================================
====================
```

TRAINING

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

start_time = time.time()
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:
        ################################

        optimizerD.zero_grad()
        loss_D, fake_imgs = loss_discriminator(netD, real, netG,
noise, Valid_label, Fake_label, criterion, optimizerD)
        loss_D.backward()
        optimizerD.step()
        ############################
        # (2) Update G network: maximize log(D(G(z)))
```

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

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

        optimizerG.zero_grad()
        loss_G = loss_generator(netD, netG, fake_imgs, Valid_label,
criterion, optimizerG)
        loss_G.backward()
        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))

end_time = time.time()
```

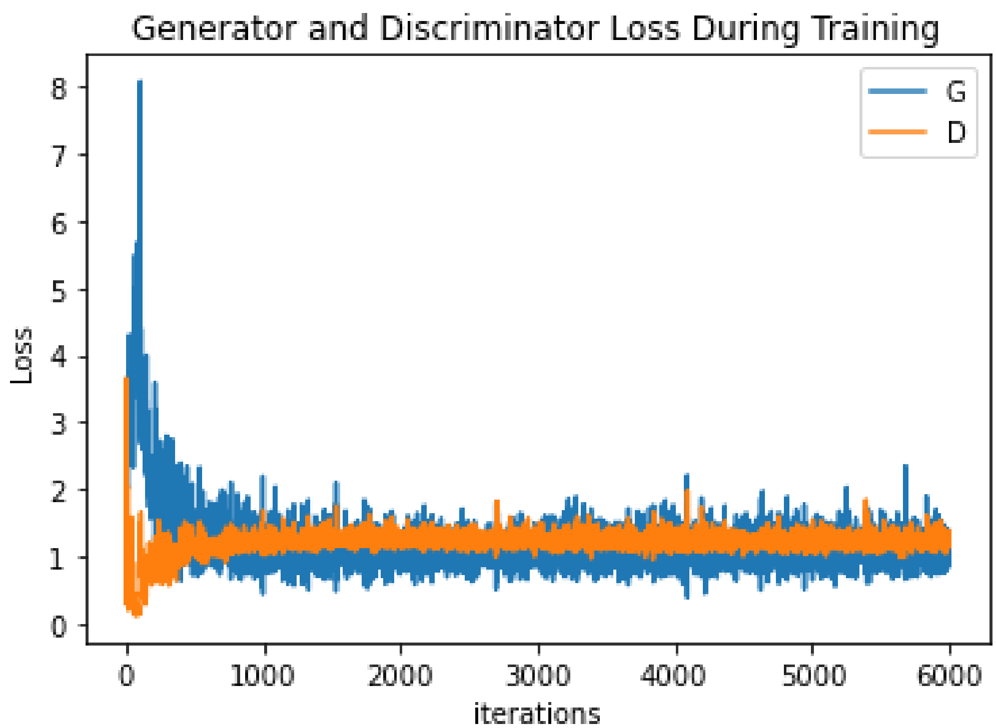
```
seconds = end_time-start_time

b=str(int((seconds%3600)//60))
c=str(int((seconds%3600)%60))
print("Runtime is {} mins {} seconds".format(b, c))
```

```
[0/10][0/600]    Loss_D: 1.5229  Loss_G: 3.2815
[0/10][50/600]   Loss_D: 0.5300  Loss_G: 3.6771
[0/10][100/600]  Loss_D: 0.1685  Loss_G: 4.1434
[0/10][150/600]  Loss_D: 0.5623  Loss_G: 4.0037
[0/10][200/600]  Loss_D: 0.9196  Loss_G: 2.0370
[0/10][250/600]  Loss_D: 0.6872  Loss_G: 2.2990
[0/10][300/600]  Loss_D: 1.0652  Loss_G: 2.7778
[0/10][350/600]  Loss_D: 0.9952  Loss_G: 1.4028
[0/10][400/600]  Loss_D: 1.2540  Loss_G: 0.8044
[0/10][450/600]  Loss_D: 1.2838  Loss_G: 1.5857
[0/10][500/600]  Loss_D: 1.0948  Loss_G: 0.9561
[0/10][550/600]  Loss_D: 1.1395  Loss_G: 1.3395
[1/10][0/600]    Loss_D: 1.1116  Loss_G: 1.1144
[1/10][50/600]   Loss_D: 1.1120  Loss_G: 1.3647
[1/10][100/600]  Loss_D: 1.1087  Loss_G: 0.8401
[1/10][150/600]  Loss_D: 1.2914  Loss_G: 0.9184
[1/10][200/600]  Loss_D: 1.2874  Loss_G: 1.5152
[1/10][250/600]  Loss_D: 1.3083  Loss_G: 0.7232
[1/10][300/600]  Loss_D: 1.3201  Loss_G: 0.8794
[1/10][350/600]  Loss_D: 1.1693  Loss_G: 0.9168
[1/10][400/600]  Loss_D: 1.6322  Loss_G: 1.3784
[1/10][450/600]  Loss_D: 1.2810  Loss_G: 0.9176
[1/10][500/600]  Loss_D: 1.1441  Loss_G: 1.4463
[1/10][550/600]  Loss_D: 1.1874  Loss_G: 1.0383
[2/10][0/600]    Loss_D: 1.2234  Loss_G: 1.0557
[2/10][50/600]   Loss_D: 1.2970  Loss_G: 0.7539
[2/10][100/600]  Loss_D: 1.1244  Loss_G: 1.1105
[2/10][150/600]  Loss_D: 1.2516  Loss_G: 1.0670
[2/10][200/600]  Loss_D: 1.2312  Loss_G: 1.0459
[2/10][250/600]  Loss_D: 1.3691  Loss_G: 1.3394
[2/10][300/600]  Loss_D: 1.2573  Loss_G: 1.0562
[2/10][350/600]  Loss_D: 1.1845  Loss_G: 1.0016
[2/10][400/600]  Loss_D: 1.4072  Loss_G: 0.9005
[2/10][450/600]  Loss_D: 1.2437  Loss_G: 0.9688
[2/10][500/600]  Loss_D: 1.3114  Loss_G: 0.8319
[2/10][550/600]  Loss_D: 1.2203  Loss_G: 0.8284
[3/10][0/600]    Loss_D: 1.3321  Loss_G: 0.9693
[3/10][50/600]   Loss_D: 1.2544  Loss_G: 1.1458
[3/10][100/600]  Loss_D: 1.3554  Loss_G: 1.2632
[3/10][150/600]  Loss_D: 1.3082  Loss_G: 0.9638
[3/10][200/600]  Loss_D: 1.2378  Loss_G: 1.0926
[3/10][250/600]  Loss_D: 1.5545  Loss_G: 0.7634
[3/10][300/600]  Loss_D: 1.2370  Loss_G: 1.0502
[3/10][350/600]  Loss_D: 1.2877  Loss_G: 0.9887
```

```
[3/10][400/600]	Loss_D: 1.2964	Loss_G: 1.0771
[3/10][450/600]	Loss_D: 1.3369	Loss_G: 1.1463
[3/10][500/600]	Loss_D: 1.1540	Loss_G: 1.0540
[3/10][550/600]	Loss_D: 1.3440	Loss_G: 1.2240
[4/10][0/600]	Loss_D: 1.2755	Loss_G: 1.4405
[4/10][50/600]	Loss_D: 1.3875	Loss_G: 0.8160
[4/10][100/600]	Loss_D: 1.2158	Loss_G: 1.1146
[4/10][150/600]	Loss_D: 1.2134	Loss_G: 1.1542
[4/10][200/600]	Loss_D: 1.2474	Loss_G: 1.3020
[4/10][250/600]	Loss_D: 1.1882	Loss_G: 1.4889
[4/10][300/600]	Loss_D: 1.0873	Loss_G: 1.5049
[4/10][350/600]	Loss_D: 1.1858	Loss_G: 0.9847
[4/10][400/600]	Loss_D: 1.1781	Loss_G: 0.9349
[4/10][450/600]	Loss_D: 1.1933	Loss_G: 1.2469
[4/10][500/600]	Loss_D: 1.3202	Loss_G: 1.1627
[4/10][550/600]	Loss_D: 1.2363	Loss_G: 1.0414
[5/10][0/600]	Loss_D: 1.1360	Loss_G: 1.1550
[5/10][50/600]	Loss_D: 1.1244	Loss_G: 1.5812
[5/10][100/600]	Loss_D: 1.3583	Loss_G: 1.1069
[5/10][150/600]	Loss_D: 1.1888	Loss_G: 1.2694
[5/10][200/600]	Loss_D: 1.1808	Loss_G: 0.9655
[5/10][250/600]	Loss_D: 1.2788	Loss_G: 1.3167
[5/10][300/600]	Loss_D: 1.2957	Loss_G: 1.0377
[5/10][350/600]	Loss_D: 1.1719	Loss_G: 0.9251
[5/10][400/600]	Loss_D: 1.2048	Loss_G: 0.9632
[5/10][450/600]	Loss_D: 1.3585	Loss_G: 1.2539
[5/10][500/600]	Loss_D: 1.1566	Loss_G: 1.2561
[5/10][550/600]	Loss_D: 1.1722	Loss_G: 1.2480
[6/10][0/600]	Loss_D: 1.1288	Loss_G: 1.0270
[6/10][50/600]	Loss_D: 1.1448	Loss_G: 0.9280
[6/10][100/600]	Loss_D: 1.2308	Loss_G: 1.0050
[6/10][150/600]	Loss_D: 1.2703	Loss_G: 1.0657
[6/10][200/600]	Loss_D: 1.3208	Loss_G: 0.7394
[6/10][250/600]	Loss_D: 1.1432	Loss_G: 0.9785
[6/10][300/600]	Loss_D: 1.3183	Loss_G: 1.7647
[6/10][350/600]	Loss_D: 1.2069	Loss_G: 1.0248
[6/10][400/600]	Loss_D: 1.3299	Loss_G: 1.0419
[6/10][450/600]	Loss_D: 1.1198	Loss_G: 1.0534
[6/10][500/600]	Loss_D: 1.0552	Loss_G: 1.3815
[6/10][550/600]	Loss_D: 1.0783	Loss_G: 0.8065
[7/10][0/600]	Loss_D: 1.1460	Loss_G: 1.0915
[7/10][50/600]	Loss_D: 1.1967	Loss_G: 0.9134
[7/10][100/600]	Loss_D: 1.1878	Loss_G: 1.0569
[7/10][150/600]	Loss_D: 1.3110	Loss_G: 0.9044
[7/10][200/600]	Loss_D: 1.1649	Loss_G: 0.9923
[7/10][250/600]	Loss_D: 1.1632	Loss_G: 1.0432
[7/10][300/600]	Loss_D: 1.0651	Loss_G: 1.0764
[7/10][350/600]	Loss_D: 1.2485	Loss_G: 1.0397
[7/10][400/600]	Loss_D: 1.2761	Loss_G: 0.7755
[7/10][450/600]	Loss_D: 1.1315	Loss_G: 1.2331
```

```
[7/10][500/600]	Loss_D: 1.2869	Loss_G: 0.7858
[7/10][550/600]	Loss_D: 1.2642	Loss_G: 1.0604
[8/10][0/600]	Loss_D: 1.2187	Loss_G: 0.8180
[8/10][50/600]	Loss_D: 1.2425	Loss_G: 1.2588
[8/10][100/600]	Loss_D: 1.2630	Loss_G: 1.3421
[8/10][150/600]	Loss_D: 1.3347	Loss_G: 1.0684
[8/10][200/600]	Loss_D: 1.1562	Loss_G: 1.3304
[8/10][250/600]	Loss_D: 1.1457	Loss_G: 1.4318
[8/10][300/600]	Loss_D: 1.4600	Loss_G: 0.7215
[8/10][350/600]	Loss_D: 1.2398	Loss_G: 0.8103
[8/10][400/600]	Loss_D: 1.1891	Loss_G: 1.0411
[8/10][450/600]	Loss_D: 1.3787	Loss_G: 0.7326
[8/10][500/600]	Loss_D: 1.0789	Loss_G: 1.2981
[8/10][550/600]	Loss_D: 1.1458	Loss_G: 1.1351
[9/10][0/600]	Loss_D: 1.2257	Loss_G: 1.5096
[9/10][50/600]	Loss_D: 1.2223	Loss_G: 0.8944
[9/10][100/600]	Loss_D: 1.2241	Loss_G: 1.0870
[9/10][150/600]	Loss_D: 1.1713	Loss_G: 1.2950
[9/10][200/600]	Loss_D: 1.2492	Loss_G: 0.9700
[9/10][250/600]	Loss_D: 1.1130	Loss_G: 1.2264
[9/10][300/600]	Loss_D: 1.2179	Loss_G: 0.9800
[9/10][350/600]	Loss_D: 1.3228	Loss_G: 1.2588
[9/10][400/600]	Loss_D: 1.1616	Loss_G: 1.3847
[9/10][450/600]	Loss_D: 1.3226	Loss_G: 0.9659
[9/10][500/600]	Loss_D: 1.2992	Loss_G: 0.9889
[9/10][550/600]	Loss_D: 1.3442	Loss_G: 1.1578
```




```
Runtime is 19 mins 4 seconds
```

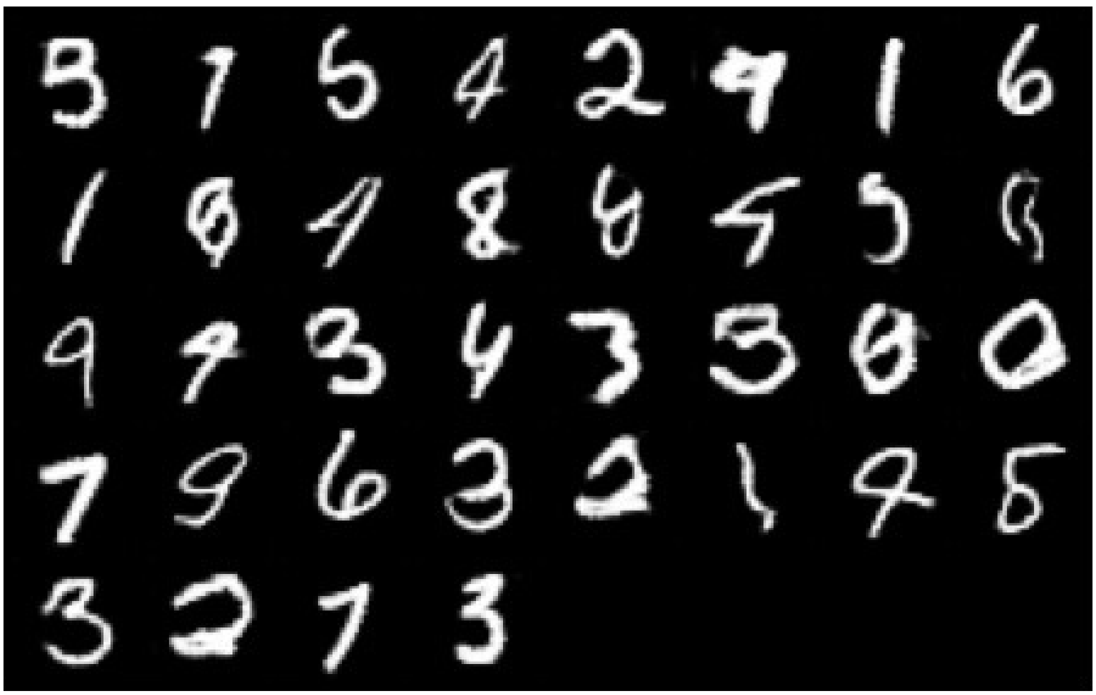
Qualitative Visualisations

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

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

```
<IPython.core.display.HTML object>
```



Citation

I have used the Pytorch doucmentation for DCGAN found here: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html