Generative Adversarial Networks

Dr. Dongchul Kim

thispersondoesnotexist.com

GAN

GAN is an algorithm that creates virtual images using deep learning.

For example, if we create a face, a deep learning algorithm predicts how image pixels should be combined to form the shape of the face.

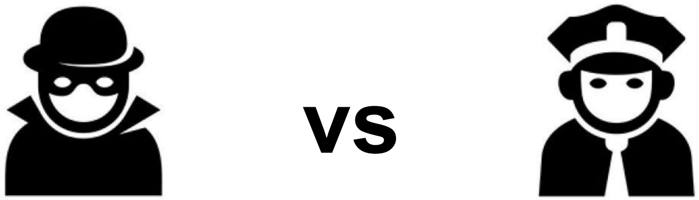
"Adversarial" shows the nature of the GAN algorithm well. This is because hostile/adversarial competition is conducted inside the GAN algorithm to create a 'real' fake.

GAN

Ian Goodfellow first proposed GAN.

To illustrate hostile contention, he gave examples of counterfeiters and police.

The competition between **counterfeit money criminals** who strive to make 'real' counterfeit bills and **police officers** who try to screen them out eventually results in a more sophisticated counterfeit bill.



Generative Adversarial Nets

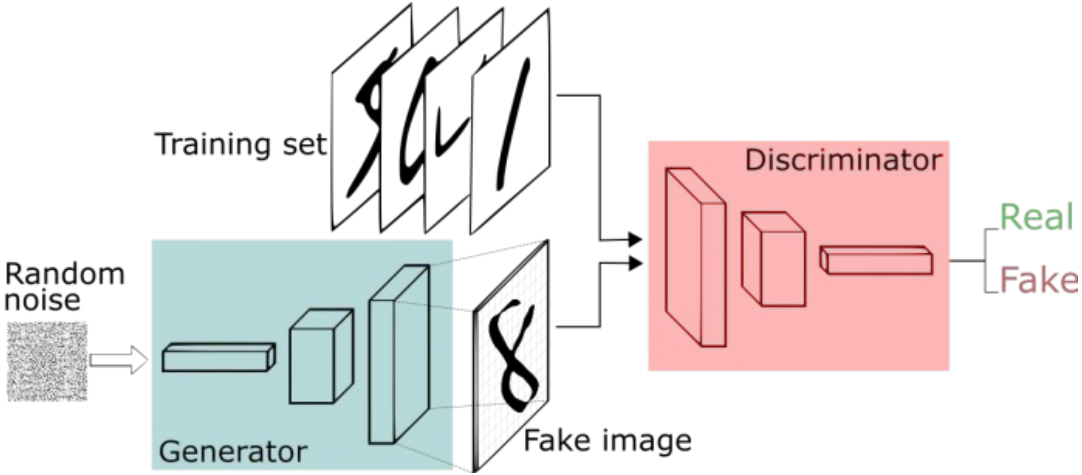
Ian J. Goodfellow, Jean Pouget-Abadie[*], Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair[†], Aaron Courville, Yoshua Bengio[‡]
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

Abstract

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data distribution and D equal to $\frac{1}{2}$ everywhere. In the case where G and D are defined by multilayer perceptrons, the entire system can be trained with backpropagation. There is no need for any Markov chains or unrolled approximate inference networks during either training or generation of samples. Experiments demonstrate the potential of the framework through qualitative and quantitative evaluation of the generated samples.

Generator (criminal) and Discriminator (police)

The place where the fake is created is called the **generator**, and the place where the authenticity is determined is called the **discriminator**.



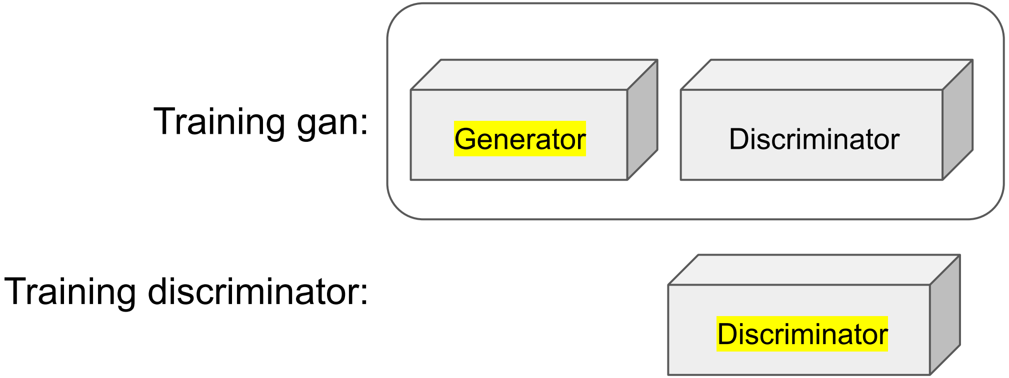
Cost function

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

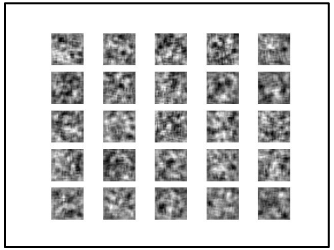
Training

Note that there are two models to train.

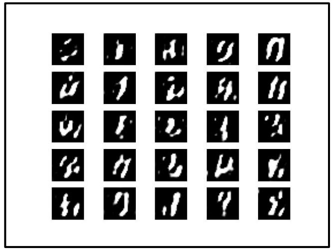
Training the models, **gan** and **discriminator**! Remember that we train only generator when we train gan. Separately, we train discriminator.



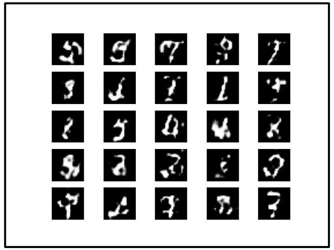
Results (Old code)



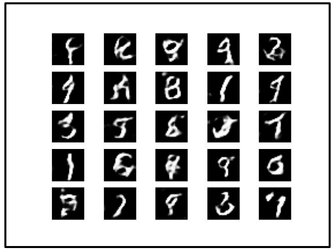
1 epoch



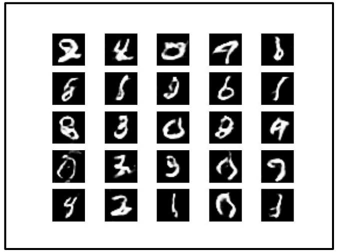
201 epoch



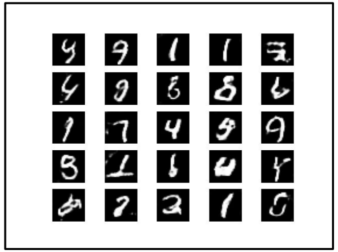
1001 epoch



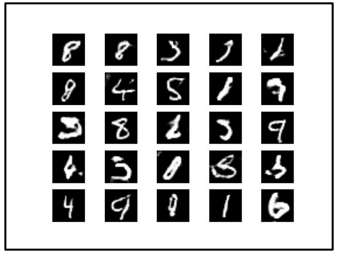
2001 epoch



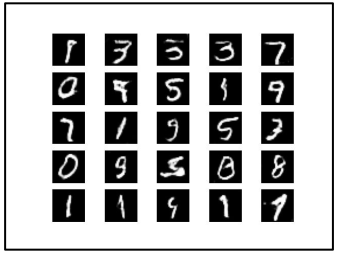
10001 epoch



20001 epoch



30001 epoch



40001 epoch

Evaluation

1. Average Log-likelihood
2. Coverage Metric
3. Inception Score (IS)
4. Modified Inception Score (m-IS)
5. Mode Score
6. AM Score
7. Frechet Inception Distance (FID)
8. Maximum Mean Discrepancy (MMD)
9. The Wasserstein Critic
10. Birthday Paradox Test
11. Classifier Two-sample Tests (C2ST)
12. Classification Performance
13. Boundary Distortion
14. Number of Statistically-Different Bins (NDB)
15. Image Retrieval Performance
16. Generative Adversarial Metric (GAM)
17. Tournament Win Rate and Skill Rating
18. Normalized Relative Discriminative Score (NRDS)
19. Adversarial Accuracy and Adversarial Divergence
20. Geometry Score
21. Reconstruction Error
22. Image Quality Measures (SSIM, PSNR and Sharpness Difference)
23. Low-level Image Statistics
24. Precision, Recall and F1 Score

Pros and Cons of GAN Evaluation Measures

Ali Borji

aliborji@gmail.com

Abstract

Generative models, in particular generative adversarial networks (GANs), have gained significant attention in recent years. A number of GAN variants have been proposed and have been utilized in many applications. Despite large strides in terms of theoretical progress, evaluating and comparing GANs remains a daunting task. While several measures have been introduced, as of yet, there is no consensus as to which measure best captures strengths and limitations of models and should be used for fair model comparison. As in other areas of computer vision and machine learning, it is critical to settle on one or few good measures to steer the progress in this field. In this paper, I review and critically discuss more than 24 quantitative and 5 qualitative measures for evaluating generative models with a particular emphasis on GAN-derived models. I also provide a set of 7 desiderata followed by an evaluation of whether a given measure or a family of measures is compatible with them.

Keywords: Generative Adversarial Nets, Generative Models, Evaluation, Deep Learning, Neural Networks

Implementation

Required Library and Parameters

```
from tqdm import tqdm

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import matplotlib
import matplotlib.pylab as plt

from torchvision.utils import make_grid, save_image
import torchvision.datasets as datasets
import torchvision.transforms as transforms

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 512
epochs = 200
nz = 128 # noise input size
k = 1 # training discriminator per batch iteration
```

Preprocessing

```
transform = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,)),
])
```

`transforms.Compose()`: This function chains together all the listed transformations into a single operation. The transformations are applied in the order they are listed.

`transforms.ToTensor()`: This transformation converts a PIL image or a NumPy array into a PyTorch tensor. It also automatically scales the image's pixel intensity values from [0, 255] to **[0, 1]**.

`transforms.Normalize((0.5,), (0.5,))`: This normalization applies to each channel of the image. The first tuple (0.5,) represents the mean for each channel, and the second tuple (0.5,) represents the standard deviation for each channel. For a grayscale image (as implied by the single values in the tuples), this will subtract 0.5 from each pixel and then divide by 0.5, effectively scaling pixel values to **[-1, 1]**. For colored images, the mean and std would be three-element tuples, corresponding to the RGB channels.

`dataset` and `DataLoader`

```
train_dataset = datasets.MNIST(
    root="./data", train=True, transform=transform, download=True)


train_loader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True, num_workers=4)
```

`num_workers=4`: Specifies the number of subprocesses to use for data loading. More workers can increase the speed of data loading but also consume more CPU memory. The optimal number can vary based on the system's configuration and the specific dataset.

```python
train_dataset = datasets.MNIST(
    root="./data", train=True, transform=transform, download=True)

train_loader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True, num_workers=4)
```

```
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 ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:00<00:00, 186150176.83it/s]Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw


Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 26419998.65it/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:00<00:00, 66675903.33it/s]Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw


Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 4825361.90it/s]Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

Generator

```python
class Generator(nn.Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz                                         # noise vector, size 128
        self.main = nn.Sequential(
            nn.Linear(self.nz, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 784),
            nn.Tanh(),
        )
    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)  # returns an image (28x28), [-1, 1]
```

Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.n_input = 784
        self.main = nn.Sequential(
            nn.Linear(self.n_input, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),           # binary classification
        )
    def forward(self, x):
        x = x.view(-1, 784)         # (28x28) image to 1 x 784
        return self.main(x)
```

Optimizer and Loss function

```
generator = Generator(nz).to(device)
discriminator = Discriminator().to(device)

optim_g = optim.Adam(generator.parameters(), lr=0.0002)
optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)

criterion = nn.BCELoss()      # Binary Cross Entropy
```

`train` function for Discriminator

```
def train_discriminator(optimizer, data_real, data_fake):
    b_size = data_real.size(0)                              # batch size 512
    real_label = torch.ones(b_size, 1).to(device)          # loading on GPU memory
    fake_label = torch.zeros(b_size, 1).to(device)
    optimizer.zero_grad()                                   # Clears old gradients from the last step
    output_real = discriminator(data_real)                  # real
    loss_real = criterion(output_real, real_label)          # real
    output_fake = discriminator(data_fake)                  # fake
    loss_fake = criterion(output_fake, fake_label)          # fake
    loss_real.backward()                                    # backpropagation
    loss_fake.backward()
    optimizer.step()                                        # Updates the discriminator's weights based on the
                                                            # gradients calculated during backpropagation.
    return loss_real + loss_fake
```

train function for Generator

```
def train_generator(optimizer, data_fake):            # optimizer should be for generator's parameters
    b_size = data_fake.size(0)                        # batch size 512
    real_label = torch.ones(b_size,1).to(device)
    optimizer.zero_grad()
    output = discriminator(data_fake)                 # Why we use discriminator() here?
    loss = criterion(output, real_label)              # Note that we compare output of fake input to real label.
    loss.backward()
    optimizer.step()
    return loss
```

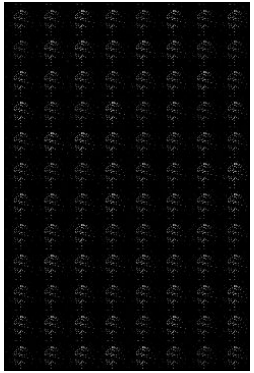
Training Models

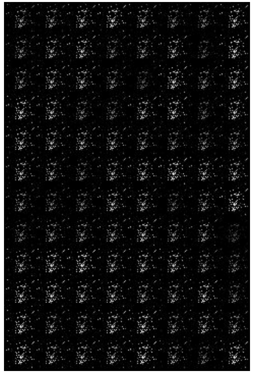
```
import time
since = time.time()

for epoch in range(epochs):
    loss_g = 0.0
    loss_d = 0.0
    for idx, data in tqdm(enumerate(train_loader), total=int(len(train_dataset)/train_loader.batch_size)):
        image, _ = data
        image = image.to(device)
        b_size = len(image)

        for step in range(k):
            data_fake = generator(torch.randn(b_size, nz).to(device)).detach()
            data_real = image
            loss_d += train_discriminator(optim_d, data_real, data_fake)

        data_fake = generator(torch.randn(b_size, nz).to(device))
        loss_g += train_generator(optim_g, data_fake)

    generated_img = generator(torch.randn(b_size, nz).to(device)).cpu().detach()
    generated_img = make_grid(generated_img)
    save_generator_image(generated_img,"./img/gen_img{epoch}.png")
    images.append(generated_img)

    epoch_loss_g = loss_g / idx
    epoch_loss_d = loss_d / idx
    losses_g.append(epoch_loss_g)
    losses_d.append(epoch_loss_d)

    print(f"Epoch {epoch} of {epochs}")
    print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {epoch_loss_d:.8f}")

time_elapsed = time.time() - since
print(time_elapsed // 60, 'min', time_elapsed % 60, 'sec')
```

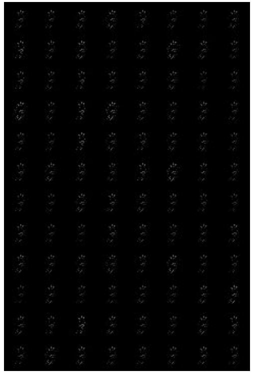
Training Models

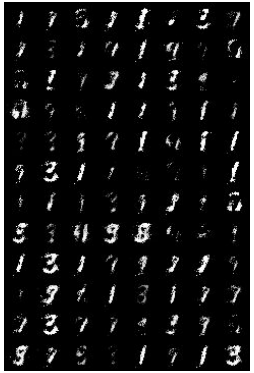
```
for epoch in range(epochs): # epochs
   loss_g = 0.0 # to store loss history
   loss_d = 0.0
   for idx, data in tqdm(enumerate(train_loader), total=int(len(train_dataset)/train_loader.batch_size)):
                                                      # setting for progress bar, num of iteration = bar size
       image, _ = data                  # ignore label
       image = image.to(device)         # MNIST images
       b_size = len(image)

       for step in range(k):            # can train discriminator multiple times but k=1 here to train fast
           data_fake = generator(torch.randn(b_size, nz).to(device)).detach()
           # detach() is to prevent gradients from flowing into the generator during the discriminator's update step

           data_real = image
           loss_d += train_discriminator(optim_d, data_real, data_fake)

       data_fake = generator(torch.randn(b_size, nz).to(device))
       loss_g += train_generator(optim_g, data_fake)
```

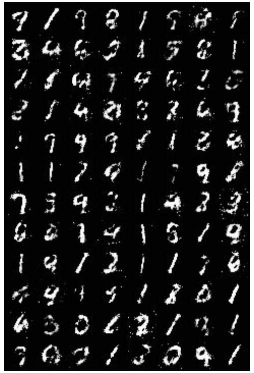
Results

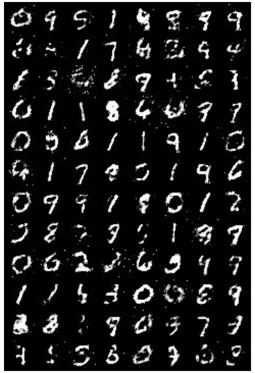

gen_img0.png


gen_img1.png


gen_img10.png

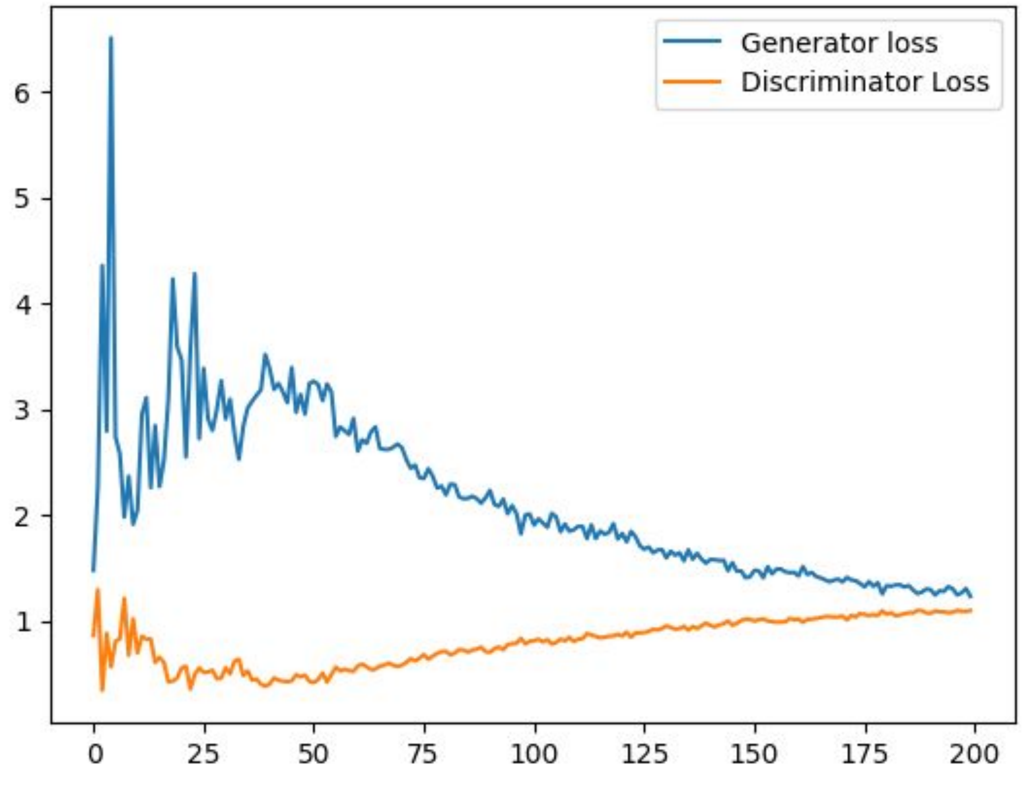

gen_img50.png


gen_img100.png


gen_img150.png

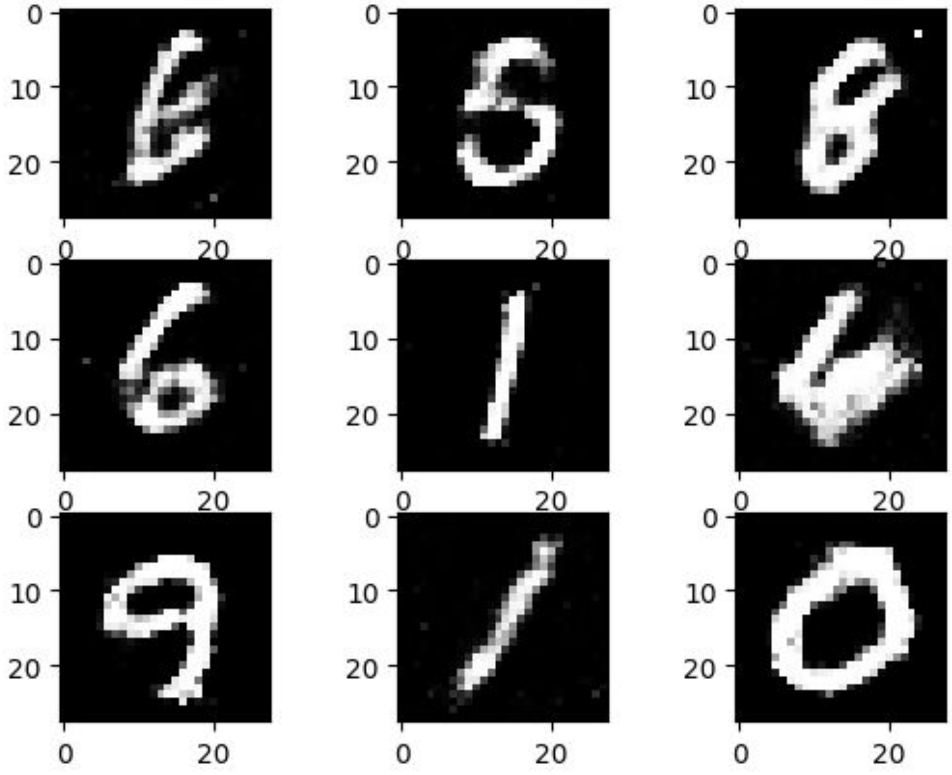

gen_img199.p

Loss



Generator loss
Discriminator Loss
6
5
4
3
2
1
0
25
50
75
100
125
150
175
200

Final Output



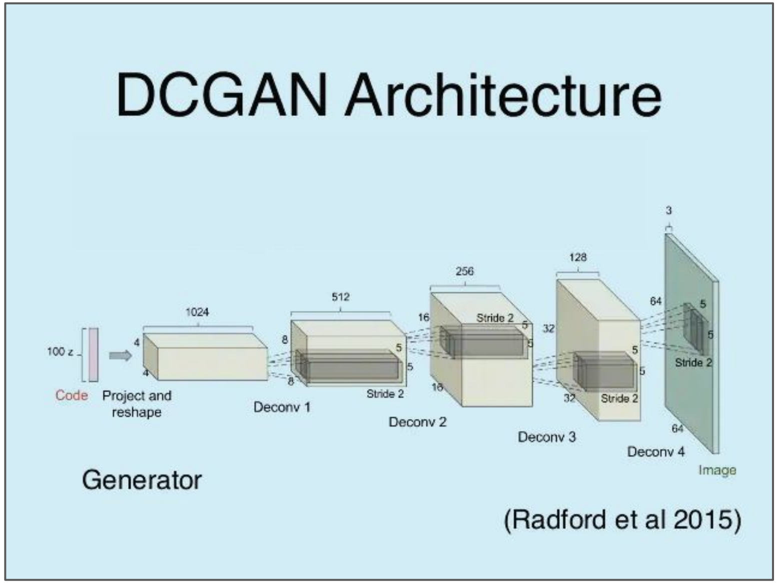
DCGAN

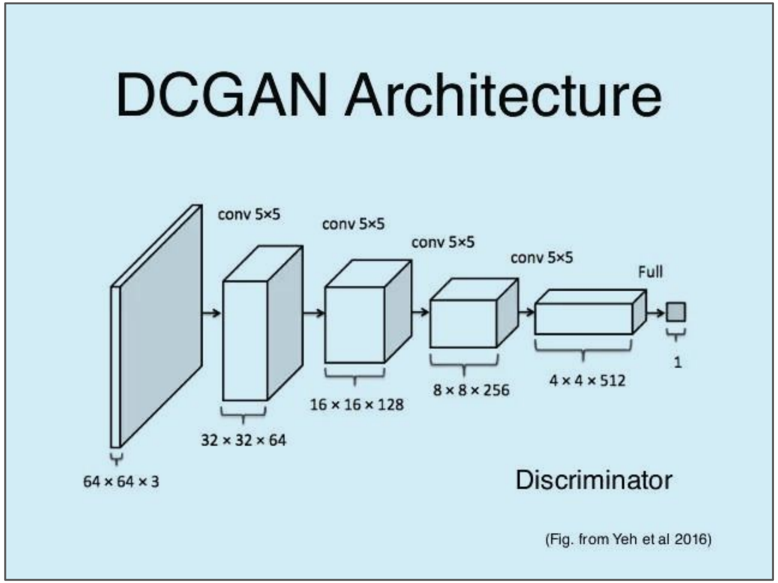
Deep Convolutional GAN

DCGAN

- DCGAN stands for Deep Convolutional Generative Adversarial Network.
- It is a variant of the GAN that specifically incorporates convolutional layers, making it more suited for dealing with image data.
- DCGANs were introduced to improve the stability and performance of traditional GANs by leveraging the architectural traits of CNNs.

DCGAN structure





DCGAN

- Use of Convolutional and Convolutional-Transpose layers in the generator and discriminator, respectively, without any pooling layers. This allows the network to learn its own spatial downsampling and upsampling.
- Batch normalization in both the generator and the discriminator to help stabilize training. Batch normalization normalizes the input to each activation layer so that it has a mean output activation of zero and standard deviation of one.
- Use of ReLU activation in the generator for all layers except for the output, which uses the Tanh function.
- Use of LeakyReLU activation in the discriminator for all layers.

Generator

```
class Generator(nn.Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.fc = nn.Linear(self.nz, 256*7*7)
        self.trans_conv1 = nn.ConvTranspose2d(256, 128, kernel_size = 3, stride = 2, padding = 1, output_padding = 1)
        self.trans_conv2 = nn.ConvTranspose2d(128, 64, kernel_size = 3, stride = 1, padding = 1)
        self.trans_conv3 = nn.ConvTranspose2d(64, 32, kernel_size = 3, stride = 1, padding = 1)
        self.trans_conv4 = nn.ConvTranspose2d(32, 1, kernel_size = 3, stride = 2, padding = 1, output_padding = 1)

    def forward(self, x):
        x = self.fc(x)
        x = x.view(-1, 256, 7, 7)
        x = F.relu(self.trans_conv1(x))
        x = F.relu(self.trans_conv2(x))
        x = F.relu(self.trans_conv3(x))
        x = self.trans_conv4(x)
        x = torch.tanh(x)
        return x
```

nn.ConvTranspose2d( )

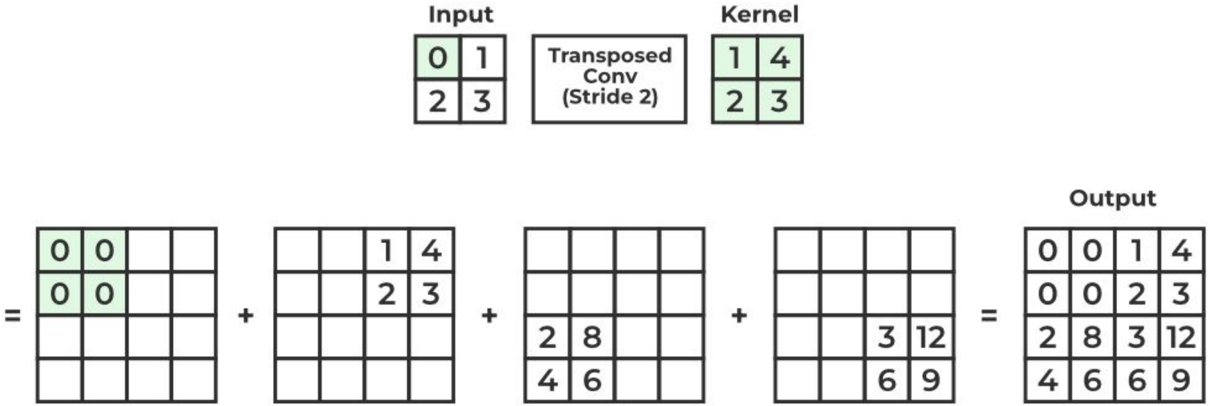
```
import torch.nn as nn

# Example: Upsample from a 128x8x8 feature map to a 64x16x16 feature map
conv_transpose = nn.ConvTranspose2d(in_channels=128, out_channels=64,
                                    kernel_size=4, stride=2, padding=1, output_padding=0)

# Assuming `x` is a batch of feature maps with shape [batch_size, 128, 8, 8]
# The output `y` will have the shape [batch_size, 64, 16, 16]
```

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

  $H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel\_size[0] + output\_padding[0]$

  $W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel\_size[1] + output\_padding[1]$

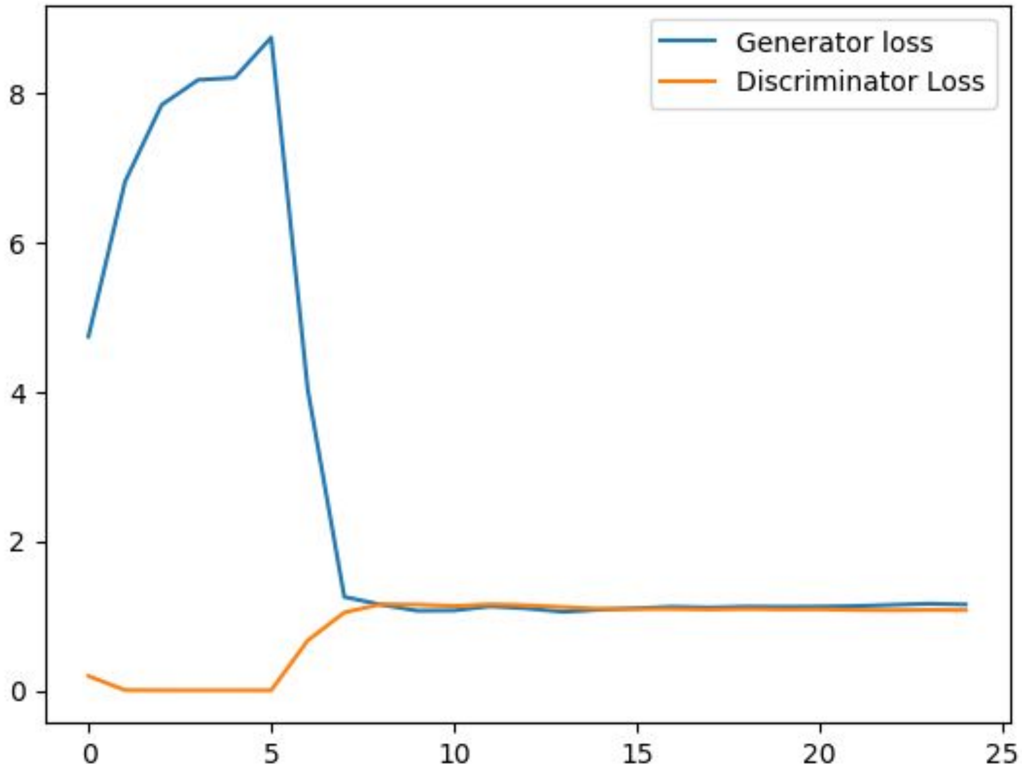
nn.ConvTranspose2d( )

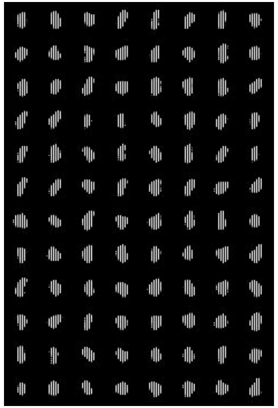


Discriminator

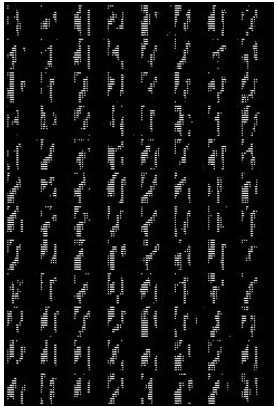
```
class Discriminator (nn.Module):
    def __init__ (self):
        super(). __init__ ()
        self.conv0 = nn.Conv2d( 1, 32, kernel_size = 3, stride = 2, padding = 1)
        self.conv0_bn = nn.BatchNorm2d( 32)
        self.conv1 = nn.Conv2d( 32, 64, kernel_size = 3, stride = 1, padding = 1)
        self.conv1_bn = nn.BatchNorm2d( 64)
        self.conv2 = nn.Conv2d( 64, 128, kernel_size = 3, stride = 1, padding = 1)
        self.conv2_bn = nn.BatchNorm2d( 128)
        self.conv3 = nn.Conv2d( 128, 256, kernel_size = 3, stride = 2, padding = 1)
        self.conv3_bn = nn.BatchNorm2d( 256)
        self.fc = nn.Linear( 12544, 1)
        self.sg = nn.Sigmoid()

    def forward (self, x):
        x = x.view( -1, 1, 28, 28)
        x = F.leaky_relu( self.conv0(x), 0.2)
        #x = self.conv0_bn(x)
        x = F.leaky_relu( self.conv1(x), 0.2)
        #x = self.conv1_bn(x)
        x = F.leaky_relu( self.conv2(x), 0.2)
        #x = self.conv2_bn(x)
        x = F.leaky_relu( self.conv3(x), 0.2)
        #x = self.conv3_bn(x)
        x = x.view( -1, self.num_flat_features(x))
        x = self.fc(x)
        x = self.sg(x)
        return x
```

Loss



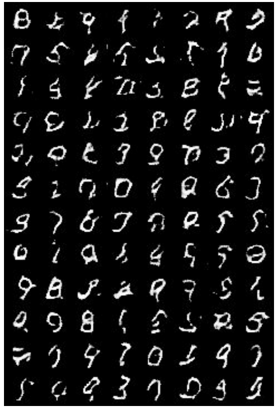


0 epoch



3 epoch

6 epoch



9 epoch

12 epoch

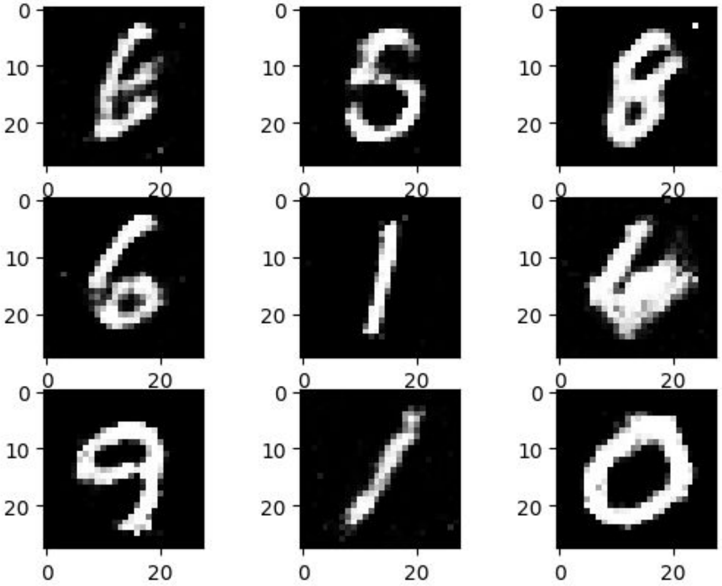
15 epoch

18 epoch

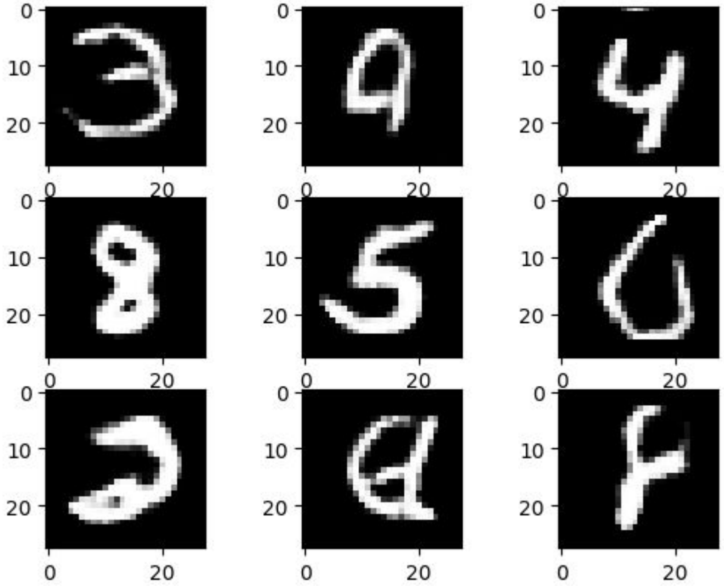
21 epoch

24 epoch

GAN vs DCGAN



GAN (200 epoch)



DCGAN (25 epoch)