Generative Adversarial Networks (GANs):

Today lecture:

0. Recap of loss function
1. Gradient saturation
2. Algorithm explanation
3. Sample code
4. Applications
    a. CycleGAN : for SR and image alignment
        i. IACGAN
    b. Pix-to-PixGAN: for damage fusion

Let's recall our loss function

$$\min_{G} \max_{D} \left\{ E_{x\sim p_{data}(x)} \left[\log D(x)\right] + E_{z\sim p_{z}(z)} \left[\log (1-D(G(z)))\right] \right\}$$

- **Goodfellow et. al., 2017 : Generative adversarial Nets**
    - **Use momentum as an optimization algorithm**

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

---

for number of training iterations **do**

  for k steps **do**

  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.

  • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.

  • Update the discriminator by ascending its stochastic gradient:

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

  end for

  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.

  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

-

- **Training of discriminator**

1. take m noise sample from z with y = 0
2. take m example from x with y = 1
3. **1/m summation operator** is equal to expectation
4. $x^{(i)}$ is coming from $p_{data}(x)$ distribution, whereas $z^{(i)}$ is coming from $p_z(z)$ distribution
5. find loss with y =0 and y =1 and compute gradients with respect to Discriminator loss and update **theta d.**
6. **this training makes the discriminator to smart enough to make distinguish between fake and real**

Training of generator

- Take m noise sample from z with label 1
- Find generator loss using label 1 and compute gradient with respect to its loss and update theta g

(3) Problem in min max function

$$\min_G \max_D \left\{ \underset{x \sim p_{data}(x)}{E} [\log(D(x))] + \underset{z \sim p_z(z)}{E} [\log(1 - D(G(z)))] \right\}$$

$$\max_D \left\{ \underset{x \sim p_{data}(x)}{E} [\log D(x)] + \underset{z \sim p_z(z)}{E} \log(1 - D(G(z))) \right\} \Rightarrow \text{OK}$$

- But for Generator, we have a problem

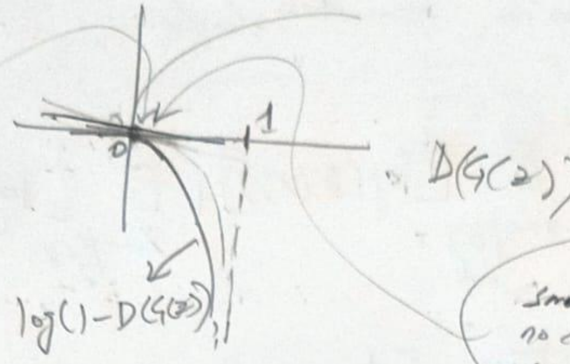
$$\min_G \left\{ \underset{z \sim p_z(z)}{E} \log(1 - D(G(z))) \right\}$$

$\to 0 \Rightarrow \log(1) = 0$

- At the start of optimization process generator is not good enough as it is not learning the distribution of $p_{data}(x)$ and it generates fake images which can easily recognized by discriminator. At start of optimization process

$$\boxed{D(G(z)) \simeq 0}$$

- It means derivatives are also zero



Small change in x, no change in y ⇒ so slope/derivative is zero

At start of optimization problem

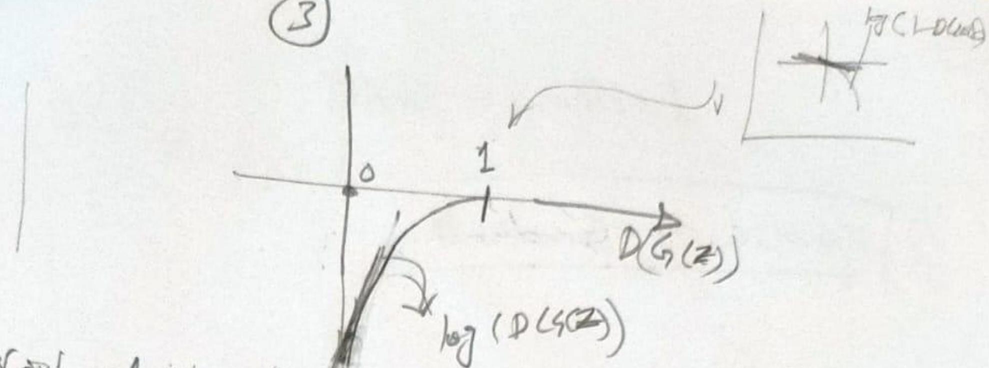
$D(G(z)) \simeq 0 \Rightarrow$ [Tangent line is completely flat]

- Slope at this point ($D(G(z)) = 0$) is very small $\simeq 0$.
- For backward pass when we calculate derivatives it is equal to <u>zero</u> & gradient descent update for generator (w, b) is not happened due to <u>$dw, db = 0$</u>
- Now there is needed to change the function

Expectation of z sample from p_z of z

$$\max_G \left\{ \mathbb{E}_{z \sim p_z(z)} \log(D(G(x)) \right\}$$

is solution to solve the <u>gradient zero problem</u>.



- Now at the start of optimization process $D(G(z)) \simeq 0$
Now the derivative (slope) is very high at this point
which force the G.D to move faster & help
easier for Generator to learn

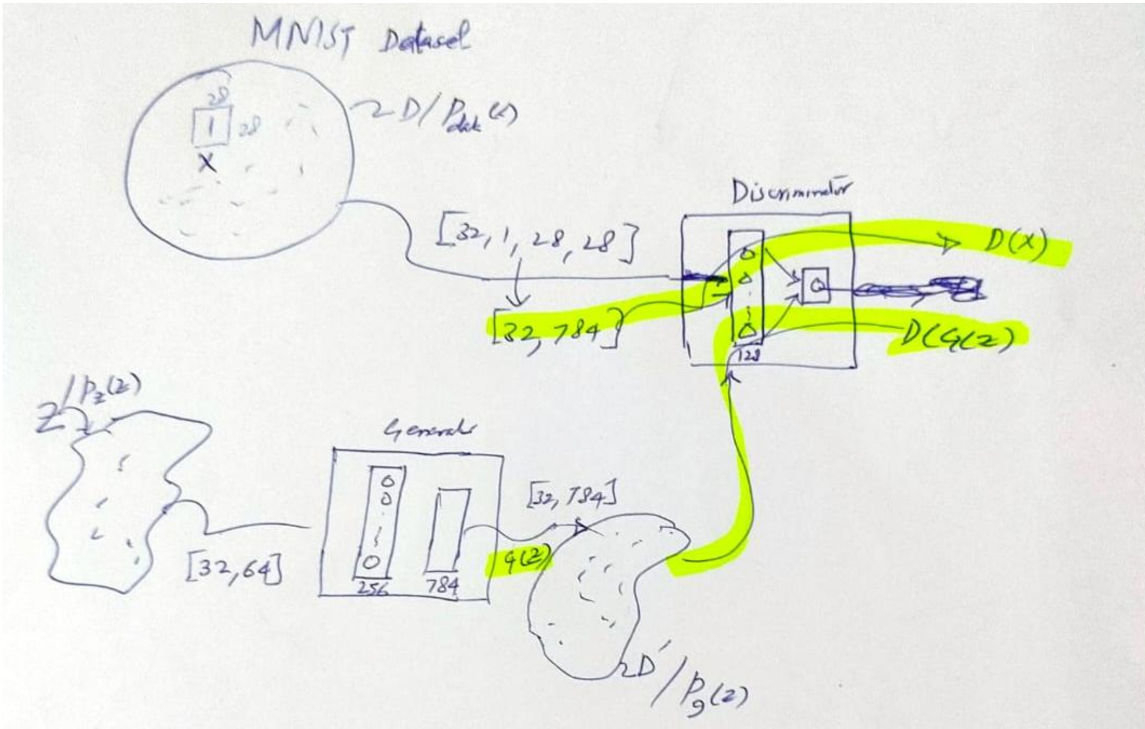
- Now the modified loss function will be 2 diff functions which will maximized D & G separately.

$$\max_D \left\{ \mathop{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathop{E}_{z \sim p_z(z)} \log(1 - D(G(z))) \right\}$$

$$\max_G \left\{ \mathop{E}_{z \sim p_z(z)} \log [D(G(z))] \right\}$$

↳ expectation of z sample from $p_z(z)$

Sample code: https://www.youtube.com/watch?v=OljTVUVzPpM&t=484s




```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.datasets as datasets
from torch.utils.data import DataLoader
import torchvision.transforms as transforms

from torchvision.utils import make_grid, save_image
from IPython.display import Image, display


'''
Home work
# things to try:
1. What happen if you use larger network
2. Better normalization with batchNorm
3. Try different learing rates
4. Change architectue to CNN
'''


class Discriminator(nn.Module):
    def __init__(self, img_dim): # in_features= image_dim = 784 for MNIST
        super().__init__()
        self.disc = nn.Sequential(
            nn.Linear(img_dim, 128),
            nn.LeakyReLU(0.01),
```

```
            nn.Linear(128, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        return self.disc(x)


class Generator(nn.Module):
    def __init__(self, z_dim, img_dim):
        super().__init__()
        self.gen = nn.Sequential(
            nn.Linear(z_dim, 256),
            nn.LeakyReLU(0.01),
            nn.Linear(256, img_dim),
            nn.Tanh(),  # To make outputs is between [-1, 1] as normalize make inputs to [-1,
1] so
        )

    def forward(self, x):
        return self.gen(x)


# Hyperparameters: GAN are very sensitive to hyperparamters so try mutple hyperparamters
device = "cuda" if torch.cuda.is_available() else "cpu"
lr = 3e-4 # andrej karpathy try with 1e-4
z_dim = 64 # may try 128, 256
image_dim = 28 * 28 * 1  # 784
batch_size = 32 # process 32 images at once
num_epochs = 10 # 500

#initilaized discriminator
disc = Discriminator(image_dim).to(device) # weights and bias are intialzed under the hood
gen = Generator(z_dim, image_dim).to(device)

fixed_noise = torch.randn((batch_size, z_dim)).to(device) # use it for testing purpose
#fixed_noise ???
#print ("fixed_noise.shape", fixed_noise.shape) # [32,64]

#transform = transforms.Normalize((0.5,), (0.5,))])
#transform = transforms.Normalize(mean=[0.5, 0.5], std=[0.5, 0.5])
# or researcher use actual MNIST mean =0.1307 and std =0.3081 for mnist dataset

#As value - mean/ std
# For 0: 0-0.5 / 0.5 = -1
# for 1: 1 - 0.5 / 0/5 = 1
# so the range is [-1 to 1]

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

dataset = datasets.FashionMNIST(root="dataset/", transform=transforms, download=True)
loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

opt_disc = optim.Adam(disc.parameters(), lr=lr) # may include beta value
opt_gen = optim.Adam(gen.parameters(), lr=lr)
```

```
'''
#https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-
and-tensorflow-cb03cdcdba0f
In the mathematical model of a GAN I described earlier,
the gradient of this had to be ascended,
but PyTorch and most other Machine Learning frameworks usually minimize functions instead.
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.


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.

'''
criterion = nn.BCELoss()



for epoch in range(num_epochs):
    for batch_idx, (real, _) in enumerate(loader): # use _ for not read label
        # reshape real image to 784 dim
        #print (real.shape) [32,1,28,28]

        real = real.view(-1, 784).to(device) # [32,784]

        #print (real.shape) # [32,784]
        batch_size = real.shape[0]

        ### Train Discriminator: max [log(D(x)) + log(1 - D(G(z)))]
        noise = torch.randn(batch_size, z_dim).to(device)
        fake = gen(noise)
        disc_real = disc(real).view(-1) # flatten() and .view(-1) flattens a tensor in
PyTorch.
        #print ("disc_real.shape",disc_real.shape) = > [32]
        #print ("disc(real).view(-1).shape", disc(real).shape) => [32,1]

        lossD_real = criterion(disc_real, torch.ones_like(disc_real)) # log(D(x))
        disc_fake = disc(fake).view(-1)
        lossD_fake = criterion(disc_fake, torch.zeros_like(disc_fake)) #log(1 - D(G(z))
                                                                       #
torch.zeros_like(disc_fake) creat zeros of same dim as that of disc_fake
                                                                       # 32 is batch size, so
disc produce 32 outputs
        lossD = (lossD_real + lossD_fake) / 2
        disc.zero_grad()
        lossD.backward(retain_graph=True) # retain_graph=True,retain computation graph
otherwise no graph is there
        opt_disc.step()

        ### Train Generator: min log(1 - D(G(z))) <-> max log(D(G(z))
        # where the second option of maximizing doesn't suffer from
        # saturating gradients
        output = disc(fake).view(-1)
```

```
        lossG = criterion(output, torch.ones_like(output)) #log(D(G(z)) here y = 1 but
inreality output is fake
        gen.zero_grad()
        lossG.backward()
        opt_gen.step()

        if batch_idx == 0:
            print(
                f"Epoch [{epoch}/{num_epochs}] Batch {batch_idx}/{len(loader)} \
                      Loss D: {lossD:.4f}, loss G: {lossG:.4f}"
            )

            with torch.no_grad():
                fake = gen(fixed_noise).reshape(-1, 1, 28, 28)# Here -1 for input x specifies
that this dimension should be
                                                                               #dynamically computed based on
the number of input values in x
                save_image(fake, f"gan_images_epoch_{epoch}.png", nrow=5, normalize=True)

                # Display the generated images in Colab
```

DCGAN: https://www.youtube.com/watch?v=IZtv9s_Wx9I

More stable GAN:

Table 1: Generator and discriminator loss functions. The main difference whether the discriminator outputs a probability (MM GAN, NS GAN, DRAGAN) or its output is unbounded (WGAN, WGAN GP, LS GAN, BEGAN), whether the gradient penalty is present (WGAN GP, DRAGAN) and where is it evaluated. We chose those models based on their popularity.

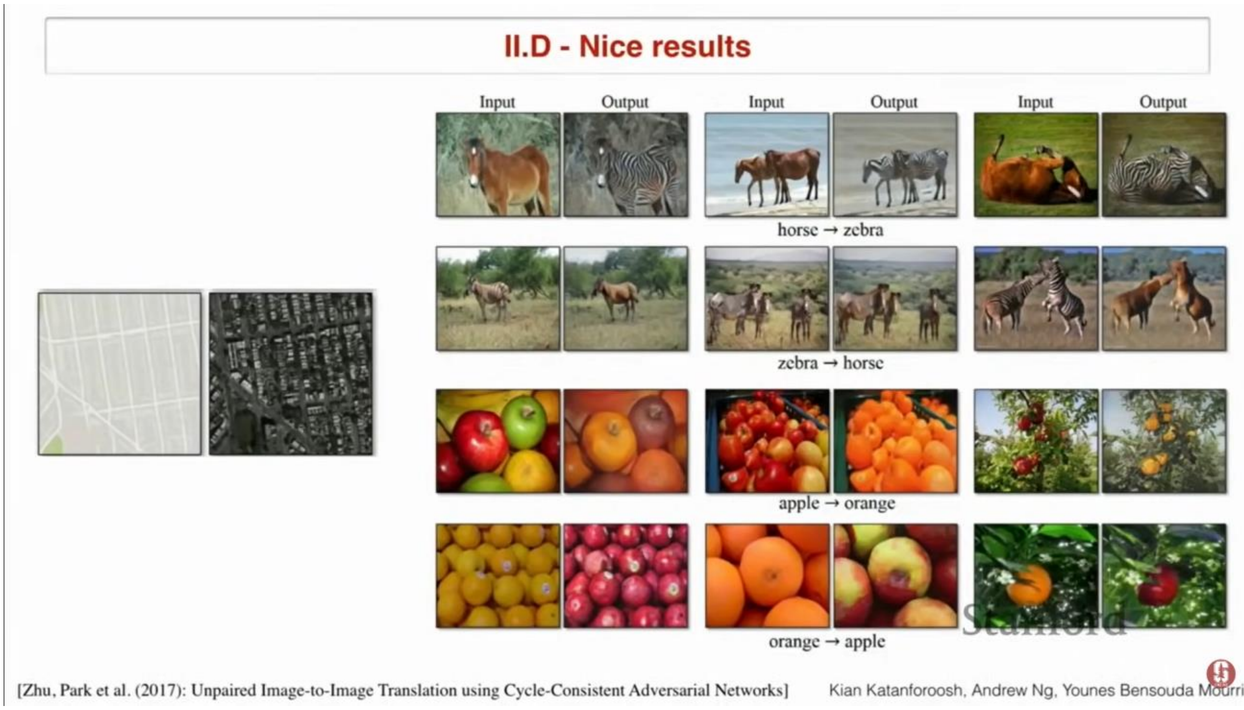
GAN	Discriminator Loss	Generator Loss
MM GAN	$\mathcal{L}_D^{GAN} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$	$\mathcal{L}_G^{GAN} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$
NS GAN	$\mathcal{L}_D^{NSGAN} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$	$\mathcal{L}_G^{NSGAN} = -\mathbb{E}_{\hat{x}\sim p_g}[\log(D(\hat{x}))]$
WGAN	$\mathcal{L}_D^{WGAN} = -\mathbb{E}_{x\sim p_d}[D(x)] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$	$\mathcal{L}_G^{WGAN} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
WGAN GP	$\mathcal{L}_D^{WGANGP} = \mathcal{L}_D^{WGAN} + \lambda\mathbb{E}_{\hat{x}\sim p_g}[(\|\|\nabla D(\alpha x + (1-\alpha\hat{x})\|\|_2 - 1)^2]$	$\mathcal{L}_G^{WGANGP} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
LS GAN	$\mathcal{L}_D^{LSGAN} = -\mathbb{E}_{x\sim p_d}[(D(x)-1)^2] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})^2]$	$\mathcal{L}_G^{LSGAN} = -\mathbb{E}_{\hat{x}\sim p_g}[(D(\hat{x}-1)^2]$
DRAGAN	$\mathcal{L}_D^{DRAGAN} = \mathcal{L}_D^{GAN} + \lambda\mathbb{E}_{\hat{x}\sim p_d+\mathcal{N}(0,c)}[(\|\|\nabla D(\hat{x})\|\|_2 - 1)^2]$	$\mathcal{L}_G^{DRAGAN} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$
BEGAN	$\mathcal{L}_D^{BEGAN} = \mathbb{E}_{x\sim p_d}[\|\|x - \mathrm{AE}(x)\|\|_1] - k_t\mathbb{E}_{\hat{x}\sim p_g}[\|\|\hat{x} - \mathrm{AE}(\hat{x})\|\|_1]$	$\mathcal{L}_G^{BEGAN} = \mathbb{E}_{\hat{x}\sim p_g}[\|\|\hat{x} - \mathrm{AE}(\hat{x})\|\|_1]$

[Lucic, Kurach et al. (2018): Are GANs Created Equal? A Large-Scale Study]

Stanford

Kian Katanforoosh, Andrew Ng, Younes Bensouda Mourri

Applications: CycleGAN



PixtoPixGAN : https://affinelayer.com/pixsrv/