GAN Code Review

Week 3 / SAI-CV-박제건

전반적인 GAN 등장배경

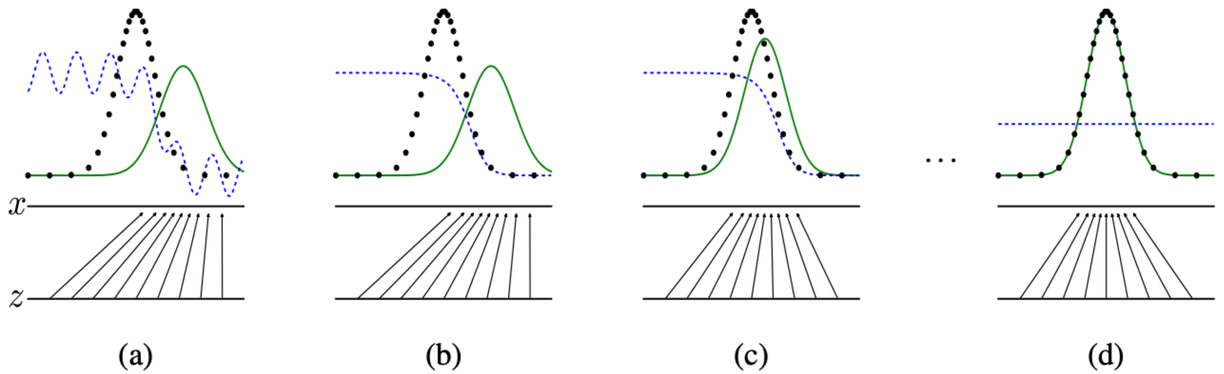
- **딥러닝이 유망한 이유:**
  인공지능이 적용 될 수 있는 여러 데이터 유형의 확률분포를 잘 표현할 수 있는 풍부하고 계층적인 모델을 발견 할 수 있다.
  (역전파, Dropout, Piecewise-linear인 유닛 덕분)

- **이러한 장점은 생성 모델에서는 잘 발휘되지 않음.**
  –> MLE 및 관련 방법(Related Work 참조) 들은 확률 변수를 계산하고 분포를 추정해야 하는데 이는 매우 까다롭고 어려움.

- **따라서 기존의 딥러닝이 잘 동작하던 방법의 장점을 활용 할 수 있는 모델을 만들어보자.**
  –> GAN(Generative Adversarial Nets)

Adversarial Nets

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



💡 생성 모델 (Generator) VS 판별자 모델 (Discriminator)의 MinMax Game

GAN Theoretical Results

✅ 우리가 확인하고 싶은 것 2가지

- Loss Function이 최적일 때 p{data}=p{g} 가 성립하며 전역 최적값(global optimal)을 가지는가?

  ➡️ Convex Function by JSD(Jensen-Shannon Divergence) & 전역 최솟값 : -log(4)

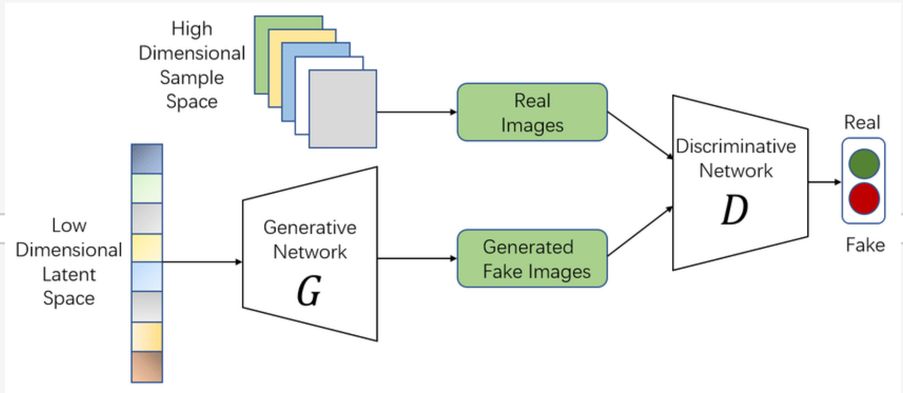
- 제안한 알고리즘이 전역 최적값으로 수렴 가능한가?

  ➡️ Yes!

Libraries & Hyperparameters

1) Libraries & Hyperparameters

```python
import os
import torch
import torch.nn as nn
import torch.utils as utils
import torch.nn.init as init
import torchvision.utils as v_utils
import torchvision.datasets as dset
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
```




```python
# Hyperparameter

epoch = 100
batch_size = 256
learning_rate = 0.0002
z_size = 100
```

⬅️ Low Dimensional Latent Space

Generator Definition

2) Generator

```python
# Generator receives random noise z and create 1*28*28 image

class Generator(nn.Module):
    def __init__(self):
        super(Generator,self).__init__()

        def block(input_dim, output_dim, normalize=True):
            layers = [nn.Linear(input_dim, output_dim)]

            if normalize:
                layers.append(nn.BatchNorm1d(output_dim, momentum=0.8))

            layers.append(nn.LeakyReLU(0.1, inplace=True))
            return layers

        self.model = nn.Sequential(
            *block(z_size, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, 1 * 28 * 28),
            nn.Tanh()
        )

    def forward(self,z):
        out = self.model(z)
        out = out.view(batch_size,1,28,28)
        return out
```

⬅️Generator Input : z_size / Output : 1*28*28 image

• < 반복되는 Block의 구조 >

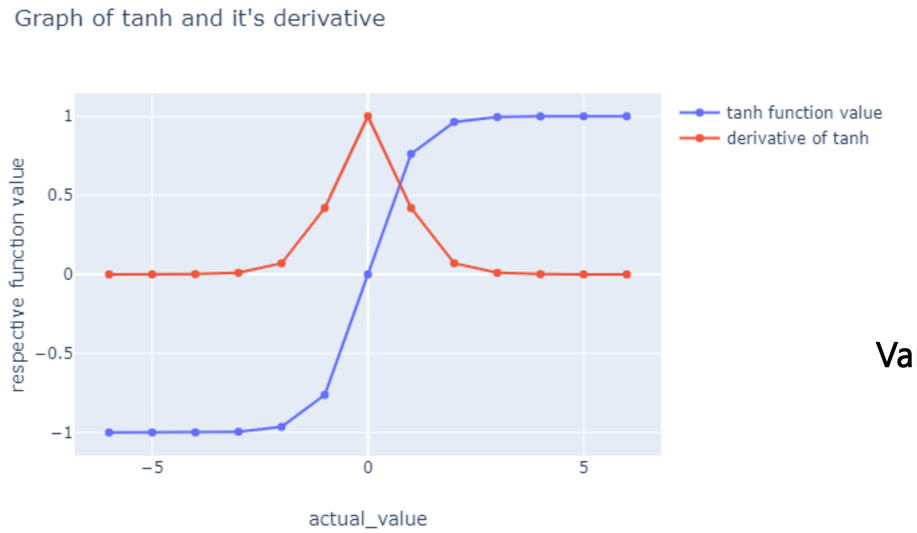
• Linear(in, out)

• Batch Normalization

• Activation(Leaky ReLU)

⬅️ 마지막 Activation Layer로 Tanh()를 사용한 점 주의!

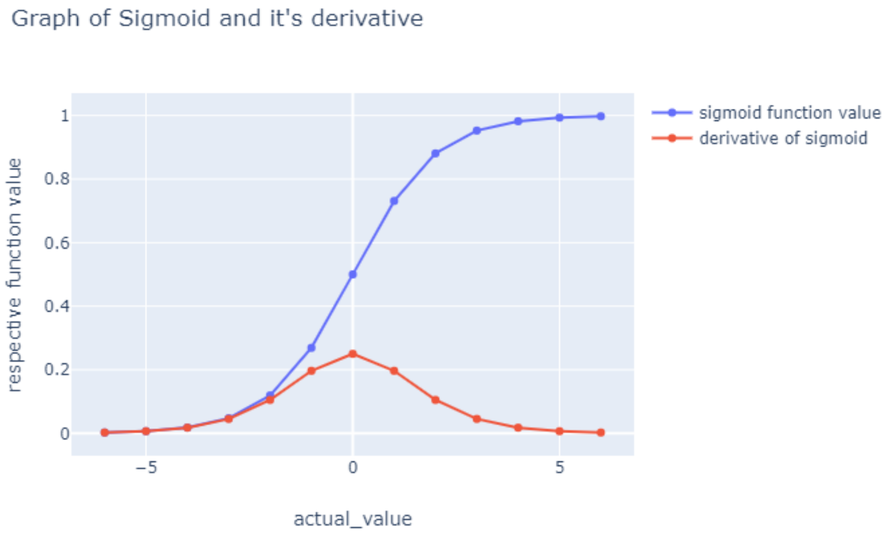
Hyperbolic Tangent를 사용하는 이유



Tanh() VS Sigmoid()

Vanishing Gradient Robustness

Training Speed




```
transforms_train = transforms.Compose([
    transforms.Resize(28),
    transforms.ToTensor(),
    transforms.Normalize([0.5],[0.5])
])
```

⬅️ Normalizing Range [-1, 1] 고려

Discriminator Definition

3) Discriminator

```
# Discriminator receives 1*28*28 image and returns a float number 0~1

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator,self).__init__()

        self.model = nn.Sequential(
            nn.Linear(1 * 28 * 28, 512),
            nn.LeakyReLU(0.1, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.1, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    def forward(self,x):
        out = x.view(batch_size, -1)
        out = self.model(out)
        return out
```

⬅️Discriminator Input : 1*28*28 Image / Output : 0~1 float number

⬅️ **inplace** - can optionally do the operation in-place. (Inplace 연산은 가급적 비추천)

- Non-inplace 연산은 operation 이전에 값을 복사해 새로운 메모리에 할당.
- Inplace 연산을 사용하면 연산 이후 결과값을 그대로 덮어씀.
- 메모리를 절약하는 효과가 있으나 그래디언트 연산이나 참조를 여러번 하는 경우의 side-effect

Data Preprocess & Load

4) Data Load

```
transforms_train = transforms.Compose([
    transforms.Resize(28),
    transforms.ToTensor(),
    transforms.Normalize([0.5],[0.5])
])

mnist_train = dset.MNIST("./",
                         train=True,
                         transform = transforms_train,
                         download=False)

train_loader = torch.utils.data.DataLoader(dataset=mnist_train,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           drop_last=True,
                                           num_workers=8)

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
print(device)
print(torch.cuda.get_device_name(0))

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

⬅사용하는 transform : Resize() / ToTensor() / Normalize()

〈 DataLoader option〉

- Shuffle (every epoch)
- Drop Last (end of batch)
- num_workers (subprocess)

⬅ CUDA 가속을 위한 Device 설정 & Load

```
cuda
NVIDIA GeForce RTX 3090
```

Data Preprocess & Load

5) Loss Function & Optimizer

```
1  loss_func = nn.BCELoss()        ⬅ Loss Function : Binary Cross Entropy
2  loss_func.to(device)
3
4  gen_optim = torch.optim.Adam(generator.parameters(), lr=learning_rate, betas=(0.5, 0.999))      ⬅ Optimizer : Adam
5  dis_optim = torch.optim.Adam(discriminator.parameters(), lr=learning_rate, betas=(0.5,0.999))
6
7
8  ones_label = torch.ones(batch_size,1).to(device)
9  zeros_label = torch.zeros(batch_size,1).to(device)   ⬅ Loss function 을 위한 정답 label 텐서
```

Binary Cross Entropy 보충설명

Cross Entropy: $H_{p,q}(X) = -\sum_{i=1}^{N} p(x_i) \log q(x_i)$

$$
\begin{aligned}
H_{p,q}(Y|X) &= -\sum_{i=1}^{N} \sum_{y \in \{0,1\}} p(y_i \mid x_i) \log q(y_i \mid x_i) \\
&= -\sum_{i=1}^{N} \left[p(y_i = 1 \mid x_i) \log q(y_i = 1 \mid x_i) + p(y_i = 0 \mid x_i) \log q(y_i = 0 \mid x_i) \right] \\
&= -\sum_{i=1}^{N} \left[p(y_i = 1 \mid x_i) \log q(y_i = 1 \mid x_i) + \{1 - p(y_i = 1 \mid x_i)\} \log\{1 - q(y_i = 1 \mid x_i)\} \right] \\
&= -\sum_{i=1}^{N} \left[p(y_i) \log q(y_i) + \{1 - p(y_i)\} \log\{1 - q(y_i)\} \right]
\end{aligned}
$$

Cross Entropy : 두 개의 확률 분포 p와 q에 대해 하나의 사건 X가 갖는 정보량.

Binary Cross Entropy는 class가 2개일 때인 special case.

또는 베르누이 분포(이항분포)의 Log Maximum Likelihood Estimation으로 해석 가능.

$$
arg\min_{\pi} -\sum_{i=1}^{n} \left(y_i \log(\pi) + (1 - y_i) \log(1 - \pi) \right)
$$

$$
\begin{aligned}
D_{KL}(p\|q) &= \sum_{i=1}^{N} p(x_i)(\log \frac{p(x_i)}{q(x_i)}) \\
&= \sum_{i=1}^{N} p(x_i) \log p(x_i) - \sum_{i=1}^{N} p(x_i) \log q(x_i) \\
&= -H_p(X) + H_{p,q}(X)
\end{aligned}
$$

$$
H_{p,q}(X) = D_{KL}(p\|q) + H_p(X)
$$

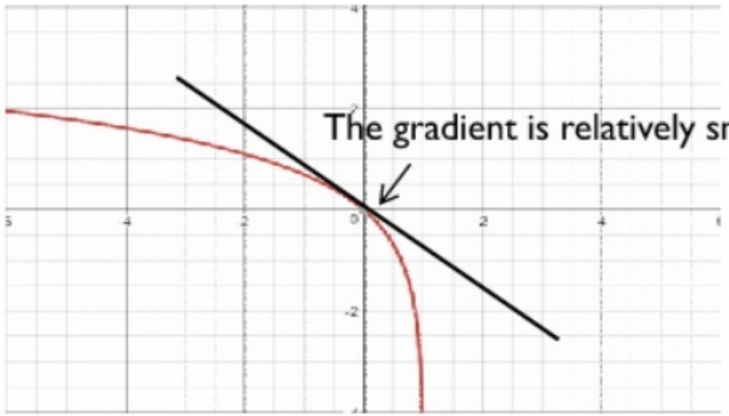
Cross Entropy 최소화 -> KLD 최소화 -> JSD 최소화

(-) 부호의 효과 in BCE

$$\min_G E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

↓ Modification (heuristically motivated)

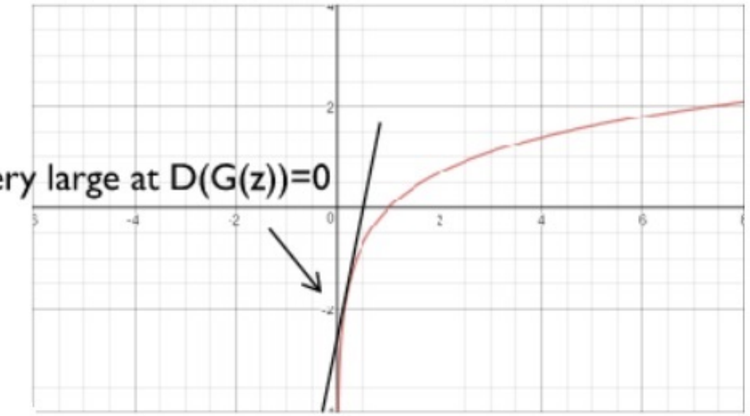
$$\max_G E_{z\sim p_z(z)}[\log D(G(z))]$$



The gradient is relatively small at D(G(z))=0

$y = \log(1 - x)$

The gradient is very large at D(G(z))=0



$y = \log(x)$

Binary Cross Entropy : -y log h(x) - (1-y) log (1-h(x))
y = 0 or 1 (label)

Model Training (1)

6) Train Model

```
try:
    os.mkdir("./result")
except:
    pass

%%time
for i in range(epoch):
    for j,(image, _) in enumerate(train_loader):
        image = image.to(device)

         # Generator Train
        gen_optim.zero_grad()

        # Fake Data
        z = init.kaiming_normal_(torch.Tensor(batch_size, z_size), a=0.2, mode='fan_in', nonlinearity='leaky_relu')
        #z = torch.normal(mean=0, std=1, size=(batch_size,z_size)).to(device)
        gen_fake = generator(z)
        dis_fake = discriminator(gen_fake)

        gen_loss = loss_func(dis_fake, ones_label)
        gen_loss.backward()
        gen_optim.step()

```

⬅️ Latent Vector (Gaussian distribution base)

⬅️ Noise Vector가 생성자를 통과해 이미지 생성 & 판별자로 주입

⬅️ Generator 생성한 이미지를 라벨 1로

Model Training (2)

```
24          # discriminator train
25          dis_optim.zero_grad()
26
27          # Real Data
28          dis_real = discriminator(image)
29
30          # calculate gradient after sum two losses
31          real_loss = loss_func(dis_real,ones_label)
32          fake_loss = loss_func(discriminator(gen_fake.detach()),zeros_label)
33
34          dis_loss = (real_loss + fake_loss)/2
35          dis_loss.backward()
36          dis_optim.step()
37
38      #torch.save([generator,discriminator], './model/vanilla_gan.pkl')
39      v_utils.save_image(gen_fake.cpu().data[0:100],'./result/gen_{}_{}.png'.format(i,j), nrow=10)
40      print("{}th epoch gen_loss: {} dis_loss:{}".format(i,gen_loss.data,dis_loss.data))
```

⬅️ 실제 이미지 DataLoader 통해 판별자에 주입

⬅️ 판별자는 실제 이미지에 대해 라벨 1로

⬅️ 판별자는 생성된 이미지에 대해 라벨 0으로

⬅️ 판별자의 Loss 값 평균

Discriminator(gen_fake.detach())의 이유

TORCH.TENSOR.DETACH

`Tensor.detach()`

Returns a new Tensor, detached from the current graph.

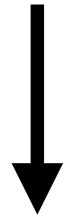
The result will never require gradient.

requires_grad = False
Forward Pass 시 gradient를 저장하지 않음.

grad_fn = False
Backpropagation이 진행되지 않음.

The detach() method constructs a new view on a tensor which is declared not to need gradients,

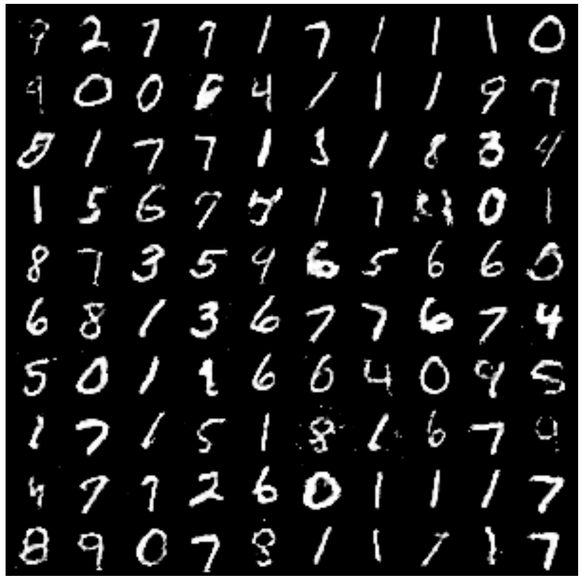
i.e., it is to be excluded from further tracking of operations, and therefore the subgraph involving this view is not recorded.



이미 학습한 생성자의 복사본을 생성하고 학습되지 않도록 생성
G에 영향을 주지 않고 D만 학습하기 위함

Reference : https://pytorch.org/docs/stable/notes/autograd.html

Results & Experiments



300 Epoch 돌렸을 때 결과

Discriminator에 BatchNorm 사용

- Normalize 해 주는 것이 상당히 중요(품질 및 수렴 속도 면에서)
- Activation은 ReLU 및 파생 계열 테스트 결과 LeakyReLU가 무난 결과 품질에 큰 차이는 없고 계산 속도 상 우위
- 학습 초기 D가 강하기 때문에 G와 밸런스 맞추기 위한 노력 (더 큰 모델, gradient 크게)
- Loss 값은 학습 수렴 후 일정 범위 내에서 진동 (Oscillation)하는 경향 (MinMax Game 균형 어려움)
- 때때로 Mode Collapse 현상 발생

감사합니다
Q&A