

# [칼럼] GAN

32기 한재희

## [목차]

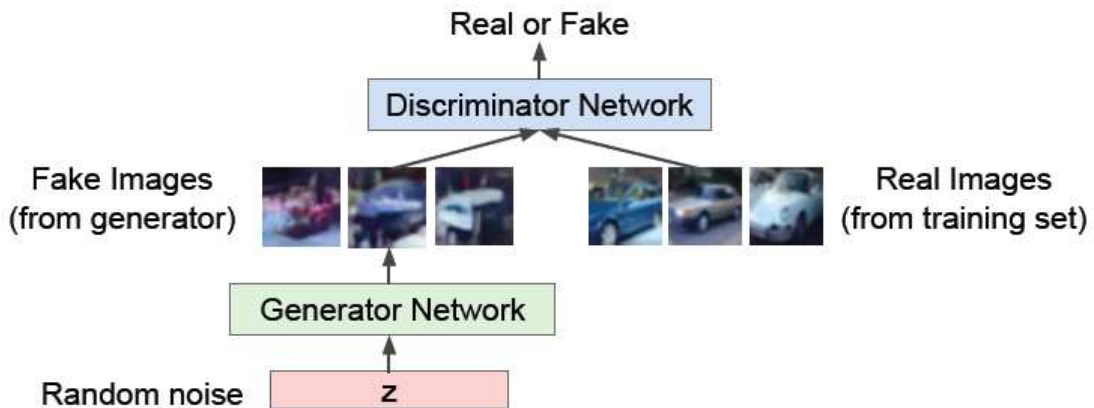
1. GAN이란?
2. GAN 모델 종류와 발전
3. DCGAN 이미지 생성
4. LSGAN 이미지 생성
5. CGAN 이미지 생성
6. 각 모델 FID 작성
7. FID 비교 및 정리

## 1. GAN이란?

생성적 적대 신경망으로 불리는 GAN은 Generative Adversarial Network의 약자로 두 개의 네트워크, 즉 생성자(Generator)와 판별자(Discriminator)로 구성되고 비지도 학습에 사용되는 머신러닝 프레임워크의 한 종류이다.

생성자는 최대한 실제처럼 보이는 데이터를 생성함으로써 판별자를 속이려 하고, 판별자는 실제 데이터와 만들어진 데이터를 구별한다. 생성자와 판별자가 상호 경쟁하며 학습을 진행한다. 이 과정에서 판별자는 실제 데이터와 만들어진 데이터를 잘 구별할 수 있게 되고 생성자는 실제 데이터와 흡사한 데이터를 잘 생성하게 된다.

GAN의 구조 및 학습 과정은 실제 데이터셋을 준비해서 판별자가 학습하는 데 사용되고 생성자는 이 데이터셋과 비슷한 분포를 따라 랜덤한 노이즈 벡터를 입력으로 받아서 가짜 데이터를 생성하게 된다. 이렇게 생성자가 만든 가짜 데이터를 판별자에게 입력하고 입력한 데이터가 실제 데이터인지 가짜 데이터인지 판단하기 위해 판별자의 손실 함수로 실제 데이터는 1, 가짜 데이터는 2로 예측하는 능력을 측정한다.



GAN의 손실 함수는 생성자와 판별자의 경쟁을 수학적으로 표현한다. 기본 손실 함수는 다음과 같이 정의된다.

$$\min_G \max_D V(D, G) = E_{x \sim p_{data}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

여기서  $G$ 는 생성자,  $D$ 는 판별자,  $p_{data}(x)$ 는 실제 데이터 분포,  $p_z(z)$ 는 생성자가 입력으로 받는 노이즈 분포,  $E$ 는 기댓값을 나타낸다. 이 함수는 두 부분으로 나뉜다. 판별자 손실 부분인  $E_{x \sim p_{data}(x)} [\log D(x)]$ 는 실제 데이터  $x$ 에 대해  $D(x)$ 가 1에 가까워지도록 하고 가짜 데이터라고 판단되면 0을 반환한다. 따라서 판별자의 성능이 좋을수록 좌변의 값은 증가하게 될 것이다. 생성자 손실 부분인  $E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$ 는 생성된 데이터  $G(z)$ 에 대해  $D(G(z))$ 가 0에 가까워지도록 한다.

## 1-1. GAN 이미지 생성

GAN을 실제로 구현하기 위해 pytorch로 구현한 코드를 분석해가며 실행하면서 이해해왔다.

## 1-2. 라이브러리 및 하이퍼 파라미터 설정

```
import os
import torch.nn as nn
import torch.utils.data
import torchvision
from torchvision import transforms
from torchvision.utils import save_image
import matplotlib.pyplot as plt
import numpy as np

# Hyper-parameters & Variables setting
num_epoch = 200
batch_size = 100
learning_rate = 0.0002
img_size = 28 * 28
num_channel = 1
dir_name = "GAN_results"

noise_size = 100
hidden_size1 = 256
hidden_size2 = 512
hidden_size3 = 1024
```

이미지 처리, 데이터셋 로드, 모델 정의 및 학습을 위한 라이브러리를 불러온다. 학습할 횟수(num\_epoch), 배치 크기(batch\_size), 학습률(learning\_rate) 등 학습에 필요한 하이퍼 파라미터와 변수들을 설정하고 이미지 크기, 채널 수, 저장 폴더, 생성자에 입력되는 노이즈 벡터 크기, 각 레이어의 뉴런 수 등을 정의해준다.

## 1-3. 장치 설정 및 디렉토리 생성

```
# Device setting
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Now using {} devices".format(device))

# Create a directory for saving samples
if not os.path.exists(dir_name):
    os.makedirs(dir_name)
```

GPU 사용이 가능한지 확인하는 코드로 가능하면 GPU를, 그렇지 않다면 CPU를 사용하도록 설정한다. 생성된 이미지를 저장할 디렉토리를 `os.makedirs()`로 생성한다.

#### 1-4. 데이터셋 로드 및 전처리 과정

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

# MNIST dataset setting
MNIST_dataset = torchvision.datasets.MNIST(root='../data/',
                                             train=True,
                                             transform=transform,
                                             download=True)

# Data loader
data_loader = torch.utils.data.DataLoader(dataset=MNIST_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)
```

데이터 전처리를 위해 **transforms.Compose**를 사용해 이미지 데이터를 텐서로 변환하고 [0, 1] 범위를 가지는 이미지를 [-1, 1]로 정규화한다. 이미지를 텐서로 변환해야 사용한 파이토치 모델이 처리할 수 있기 때문이다. 입력 데이터 범위가 [-1, 1]로 고르게 분포되면 네트워크의 가중치가 균등하게 학습되기 때문에 정규화는 필수적이다. **torchvision.datasets.MNIST**를 사용하여 MNIST 데이터셋을 가져온다. MNIST는 손글씨 숫자(0-9)의 이미지로 구성된 데이터셋이다. 각 이미지는 28x28픽셀 크기이고 머신 러닝과 딥러닝 모델의 성능을 측정하는 데 자주 사용되는 표준 데이터셋이다. **data\_loader**를 사용하여 데이터셋을 배치 단위로 불러오고 데이터를 섞는다. GAN이 MNIST 데이터셋을 효과적으로 학습할 수 있도록 준비하는 과정이다.

#### 1-5. 판별자 정의

```
# Declares discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.linear1 = nn.Linear(img_size, hidden_size3)
        self.linear2 = nn.Linear(hidden_size3, hidden_size2)
        self.linear3 = nn.Linear(hidden_size2, hidden_size1)
        self.linear4 = nn.Linear(hidden_size1, 1)
        self.leaky_relu = nn.LeakyReLU(0.2)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.leaky_relu(self.linear1(x))
        x = self.leaky_relu(self.linear2(x))
        x = self.leaky_relu(self.linear3(x))
        x = self.linear4(x)
        x = self.sigmoid(x)
        return x
```

**nn.Linear** 레이어를 통해 이미지 데이터를 점진적으로 압축해 나가며 활성화 함수를 사용해 비선형성을 추가한다. 마지막 레이어에서 출력값을 [0, 1] 범위로 변환한다. 이때 1은 진짜 이미지, 0은 가짜 이미지를 나타낸다. 여러 층의 신경망을 통해 입력된 이미지를 처리하고 최종적으로 이미지가 진짜일 확률을 출력하게 된다.

## 1-6. 생성자 정의

```
# Declares generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.linear1 = nn.Linear(noise_size, hidden_size1)
        self.linear2 = nn.Linear(hidden_size1, hidden_size2)
        self.linear3 = nn.Linear(hidden_size2, hidden_size3)
        self.linear4 = nn.Linear(hidden_size3, img_size)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.relu(self.linear1(x))
        x = self.relu(self.linear2(x))
        x = self.relu(self.linear3(x))
        x = self.linear4(x)
        x = self.tanh(x)
        return x
```

생성자도 마찬가지로 `nn.Linear` 레이어를 사용하여 판별자와 동일하게 활성화 함수를 사용하고 마지막 레이어에서 출력값을  $[-1, 1]$  범위로 변환한다. 랜덤한 노이즈 벡터를 입력받아서 이를 기반으로 가짜 이미지를 생성하게 된다.

## 1-7. 모델 초기화 및 설정

```
# Initialize generator/Discriminator
discriminator = Discriminator()
generator = Generator()

# Device setting
discriminator = discriminator.to(device)
generator = generator.to(device)

# Loss function & Optimizer setting
criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(discriminator.parameters(), lr=learning_rate)
g_optimizer = torch.optim.Adam(generator.parameters(), lr=learning_rate)
```

먼저 생성자와 판별자 모델을 초기화 해주고 `.to(device)`로 판별자 모델을 GPU 또는 CPU로 전송한다. 생성자 모델도 판별자 모델과 같은 디바이스로 전송하게 되고 `criterion = nn.BCELoss()` 손실 함수로 판별자가 출력하는 확률을 기반으로 손실을 계산한다. 판별자 모델의 파라미터를 계속 업데이트하기 위해 Adam 옵티마이저를 사용하고 학습률로 모델의 파라미터가 업데이트되는 속도를 조절한다. 생성자 모델도 마찬가지로 Adam 옵티마이저를 사용한다.

## 1-8. 모델 학습

```
'''
Training part
'''
for epoch in range(num_epoch):
    for i, (images, label) in enumerate(data_loader):

        # make ground truth (labels) -> 1 for real, 0 for fake
        real_label = torch.full((batch_size, 1), 1, dtype=torch.float32).to(device)
        fake_label = torch.full((batch_size, 1), 0, dtype=torch.float32).to(device)

        # reshape real images from MNIST dataset
        real_images = images.reshape(batch_size, -1).to(device)

        # +-----+
        # |  train Generator  |
        # +-----+

        # Initialize grad
        g_optimizer.zero_grad()
        d_optimizer.zero_grad()

        # make fake images with generator & noise vector 'z'
        z = torch.randn(batch_size, noise_size).to(device)
        fake_images = generator(z)

        # Compare result of discriminator with fake images & real labels
        # If generator deceives discriminator, g_loss will decrease
        g_loss = criterion(discriminator(fake_images), real_label)

        # Train generator with backpropagation
        g_loss.backward()
        g_optimizer.step()

        if (i + 1) % 150 == 0:
            print("Epoch [ {}/{} ] Step [ {}/{} ] d_loss : {:.5f} g_loss : {:.5f}"
                  .format(epoch, num_epoch, i+1, len(data_loader), d_loss.item(), g_loss.item()))

        # print discriminator & generator's performance
        print(" Epoch {}'s discriminator performance : {:.2f} generator performance : {:.2f}"
              .format(epoch, d_performance, g_performance))

        # Save fake images in each epoch
        samples = fake_images.reshape(batch_size, 1, 28, 28)
        save_image(samples, os.path.join(dir_name, 'GAN_fake_samples{}.png'.format(epoch + 1)))

        # Display the generated images
        samples = samples.cpu().detach().numpy()
        samples = np.transpose(samples, (0, 2, 3, 1)) # Convert from (N, 1, 28, 28) to (N, 28, 28, 1)
        fig, ax = plt.subplots(1, 10, figsize=(15, 15))
        for idx in range(10):
            ax[idx].imshow(samples[idx].squeeze(), cmap='gray')
            ax[idx].axis('off')
        plt.show()

        # +-----+
        # |  train Discriminator  |
        # +-----+

        # Initialize grad
        d_optimizer.zero_grad()
        g_optimizer.zero_grad()

        # make fake images with generator & noise vector 'z'
        z = torch.randn(batch_size, noise_size).to(device)
        fake_images = generator(z)

        # Calculate fake & real loss with generated images above & real images
        fake_loss = criterion(discriminator(fake_images), fake_label)
        real_loss = criterion(discriminator(real_images), real_label)
        d_loss = (fake_loss + real_loss) / 2

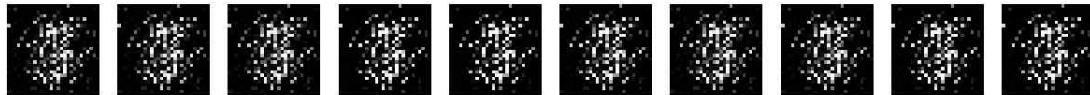
        # Train discriminator with backpropagation
        # In this part, we don't train generator
        d_loss.backward()
        d_optimizer.step()

        d_performance = discriminator(real_images).mean()
        g_performance = discriminator(fake_images).mean()
```

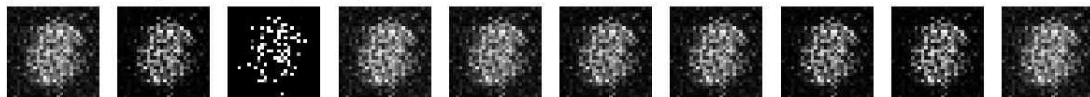
이 구간은 GAN을 학습시킨다. 학습 과정에서 생성자와 판별자가 번갈아 가며 학습하고 판별자는 실제 이미지와 가짜 이미지를 구별하는 능력을 강화하고, 생성자는 점점 더 진짜 같은 이미지를 생성하도록 학습한다.

## 1-9. 학습 결과물

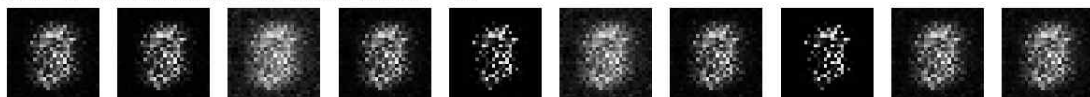
```
Epoch [ 0/200 ] Step [ 150/600 ] d_loss : 0.01995 g_loss : 3.66807
Epoch [ 0/200 ] Step [ 300/600 ] d_loss : 0.21732 g_loss : 8.09666
Epoch [ 0/200 ] Step [ 450/600 ] d_loss : 0.01808 g_loss : 6.00056
Epoch [ 0/200 ] Step [ 600/600 ] d_loss : 0.09284 g_loss : 10.24346
Epoch 0's discriminator performance : 0.96 generator performance : 0.00
```



```
Epoch [ 1/200 ] Step [ 150/600 ] d_loss : 0.00265 g_loss : 8.09449
Epoch [ 1/200 ] Step [ 300/600 ] d_loss : 0.03626 g_loss : 8.78047
Epoch [ 1/200 ] Step [ 450/600 ] d_loss : 0.06027 g_loss : 12.78851
Epoch [ 1/200 ] Step [ 600/600 ] d_loss : 0.15628 g_loss : 17.54221
Epoch 1's discriminator performance : 0.92 generator performance : 0.03
```



```
Epoch [ 2/200 ] Step [ 150/600 ] d_loss : 0.21495 g_loss : 8.79067
Epoch [ 2/200 ] Step [ 300/600 ] d_loss : 0.03669 g_loss : 4.84455
Epoch [ 2/200 ] Step [ 450/600 ] d_loss : 0.29964 g_loss : 4.58767
Epoch [ 2/200 ] Step [ 600/600 ] d_loss : 0.34388 g_loss : 2.75706
Epoch 2's discriminator performance : 0.72 generator performance : 0.05
```



```
Epoch [ 3/200 ] Step [ 150/600 ] d_loss : 0.17015 g_loss : 2.63460
Epoch [ 3/200 ] Step [ 300/600 ] d_loss : 0.40609 g_loss : 2.24951
Epoch [ 3/200 ] Step [ 450/600 ] d_loss : 0.12124 g_loss : 11.95638
Epoch [ 3/200 ] Step [ 600/600 ] d_loss : 0.23747 g_loss : 3.20629
Epoch 3's discriminator performance : 0.90 generator performance : 0.05
```

코랩을 사용해 코드를 실행하면 이렇게 로그 메시지가 뜨면서 이미지가 생성된다. GAN 훈련 중에 특정 에포크와 단계에서 판별자 손실인 d\_loss와 생성자 손실인 g\_loss를 나타낸다. Epoch [ 0/200 ] Step [ 150/600 ] d\_loss : 0.01995 g\_loss : 3.66807 가 제일 처음 결과인데 현재 훈련이 0번째 에포크 진행 중임을 의미한다. 이때 에포크는 전체 데이터셋에 대해 모델이 한번 학습을 완료한 주기를 뜻한다. 그리고 현재 에포크 내에서 150번째 스텝 진행 중이고 이때 스텝은 배치 단위로 데이터를 처리하는 반복 횟수이다. 판별자 손실인 d\_loss가 0.01995임을 나타내는데 이 값은 판별자가 실제 이미지와 가짜 이미지를 얼마나 잘 구분하는지에 대한 손실을 나타내고 있다. 손실 값이 낮을수록 판별자가 더 잘 구분하고 있음을 의미한다. 생성자 손실인 g\_loss는 3.66807로 나타나는데 이 값은 생성자가 판별자를 속여서 가짜 이미지를 진짜 이미지처럼 보이게 만들려는 목표에 대한 손실을 나타낸다. 손실 값이 낮을수록 생성자가 판별자를 더 잘 속이고 있음을 의미한다. 이러한 에포크가 200번째까지 반복하면서 이미지를 생성해나간다.

```
Epoch [ 100/200 ] Step [ 150/600 ] d_loss : 0.46477 g_loss : 1.80949
Epoch [ 100/200 ] Step [ 300/600 ] d_loss : 0.44586 g_loss : 1.82692
Epoch [ 100/200 ] Step [ 450/600 ] d_loss : 0.47110 g_loss : 1.01907
Epoch [ 100/200 ] Step [ 600/600 ] d_loss : 0.47753 g_loss : 1.49410
Epoch 100's discriminator performance : 0.67 generator performance : 0.28
```



절반인 100번째 에포크를 보면 d\_loss는 높아졌고 g\_loss는 낮아지고 생성된 이미지가 처음보다 선명해진 걸 확인할 수 있다.



200번째는 출력이 되지 않고 199번째 에포크까지 출력이 됐는데 로그 메시지를 확인해보면 Epoch 199's discriminator performance : 0.71 generator performance : 0.32 이렇게 출력됐다. 판별자와 생성자의 성능을 최종적으로 나타내는 값인데 판별자의 성능은 0.71, 생성자의 성능은 0.32로 판별자가 71%의 확률로 이미지를 올바르게 분류하고 있고 생성자는 32%의 확률로 판별자를 속이는 데 성공한다는 의미를 가지고 있다. 최종적으로 생성된 이미지를 보면 맨눈으로 봐도 0-9까지의 숫자 형태가 정확히 나오지 않고 중복된 숫자나 외곽선이 흐릿한 이미지가 많은 걸 확인할 수 있다. 현재 훈련된 모델에서는 판별자가 우위를 점하고 있다고 로그 메시지를 통해 알 수 있는데 판별자가 생성자가 만든 가짜 이미지를 잘 구분하고 있으므로 생성자가 더 많은 학습을 통해 진짜 같은 이미지를 생성할 수 있도록 개선될 필요가 있다.



## 2. GAN 모델 종류와 발전

기본 GAN은 가장 단순한 형태로 생성자와 판별자가 서로 경쟁하는 형태로 구성되어 있다. 생성자는 무작위 노이즈 벡터를 입력으로 받아 가짜 데이터를 생성하고, 판별자는 이 데이터가 진짜인지 판별하는 역할을 한다. 이러한 GAN은 이미지 생성과 변환, 텍스트 생성 등 다양한 분야에서 응용되어왔다. 그러나 기본 GAN은 초기 모델로서 학습 불안정성의 문제를 겪는 사례가 종종 발생했다. 이러한 문제를 해결하기 위해 다양한 종류의 GAN 모델들이 제안되었으며 각 모델은 특정 문제를 해결하고 생성 능력을 향상하기 위한 새로운 접근 방식을 도입했다는 특징이 있다.

이번 칼럼에서 다룰 GAN 종류는 DCGAN, LSGAN, CGAN 총 3개다. 먼저 DCGAN은 **Deep Convolutional GAN**으로 딥러닝 모델인 **컨볼루션 신경망(CNN)**을 사용해서 **GAN의 학습 안정성을 개선한 모델**이다. 컨볼루션 신경망(CNN)은 사람의 시각 처리 방식을 모방한 딥러닝 학습 모델이다. DCGAN은 CNN의 컨볼루션 레이어를 사용해서 이미지의 공간적 구조를 보존하면서 해상도를 점진적으로 줄이거나 늘릴 수 있다. 2016년에 발표된 DCGAN에 대한 논문인 Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks에서 테스트 데이터로 침실 사진, 사람 얼굴을 주고 에포크의 차이를 두고 DCGAN을 학습시키는 실험을 했다.



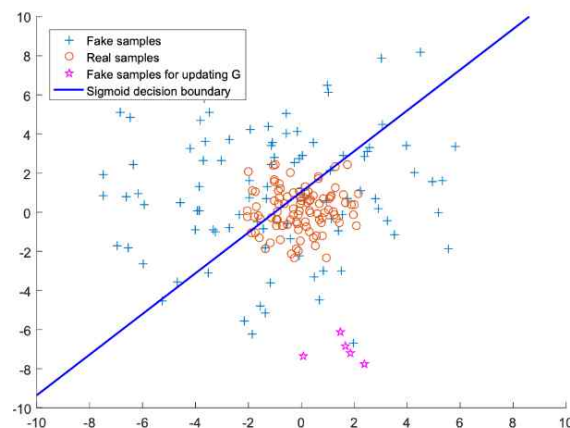


1 에포크를 학습시켰을 때의 결과물이다. 멀리서 보면 그럴싸한 침실 사진이겠지만 자세히 보면 흐릿하고 사진이 깨져있는 걸 확인할 수 있다. 하지만 첫 에포크 만에 이미지를 외워서 그대로 내보내지 않고 이미지를 스스로 만들어냈다는 것을 보여주는 결과다.



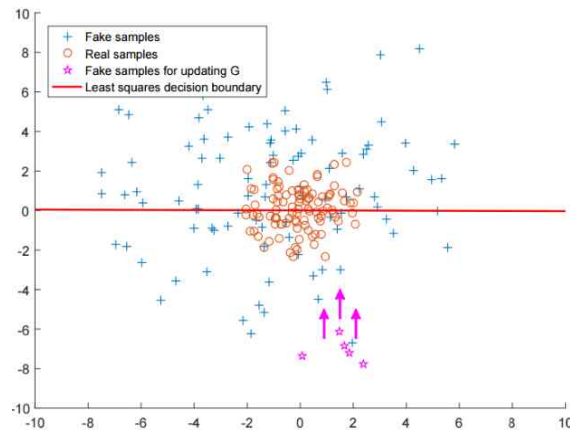
5 에포크를 학습시킨 결과물은 다음과 같다. 언뜻 보기에 확실하게 선명해진 것을 확인할 수 있고 실제로 존재하는 방처럼 이미지를 만들어냈다. 논문에서는 아직 모델이 학습 오류를 줄이지 못하는 상황인 언더피팅(underfitting)이 일어나고 있다고 말하면서 그 증거로 침대의 머리 부분에 약간의 노이즈가 반복되는 것을 확인할 수 있다고 한다. 이를 통해 DCGAN은 생성자와 판별자가 더 복잡한 이미지 패턴을 학습할 수 있는 능력을 키워준다고 볼 수 있다. 특히 얼굴, 동물, 풍경 등의 이미지 생성에서 뛰어난 성능을 발휘한다.

LSGAN은 Least Squares GAN으로 최소 제곱 오차 손실을 사용해서 최소 제곱 GAN이라고 불린다. 주요 특징으로는 먼저 손실 함수의 변경이 있다. LSGAN은 GAN의 손실 함수로 최소 제곱 오차를 사용해서 판별자가 생성된 샘플에 대해 더 부드러운 순간변화율을 제공해서 생성자에게 더 유용한 학습 신호를 제공해 생성된 데이터가 판별자의 결정 경계에서 벗어나는 것을 방지해준다.



위 그림은 손실 함수를 0과 1로 판단하는 기준을 가지고 있는 파란 선으로 나타낸 그림이다. 파란 선을 기준으로 위는 가짜, 아래는 진짜로 판단한다. + 모양은 가짜 데이터, 주황색 동그라미는 진짜 데이터, 별은 가짜 데이터지만 진짜로 분류된 것들을 나타낸다. 별을 보면 생성자가 판별자를 잘 속이고 있다는 걸 볼 수 있지만, 판별자를 속이는 것에서 끝나는 것이 아니라 사람이 봤을 때도 실제 데이터와 비슷하도록 최대한으로 만드는 것이 궁극적인 목표라고 볼 수 있다. 이걸 인지

하고 봤을 때 별은 실제 데이터인 주황색 동그라미와 많이 떨어져 있다. 둘 사이의 거리가 가까울수록 사람까지 속일 수 있다는 뜻인데 이를 이루기 위해 적용한 것이 최소 제곱이다.



최소 제곱 기준선인 빨간 선이 추가된 모습이다. 이 기준선이 생겨서 별들은 주황색 동그라미와 멀리 떨어져 있을수록 페널티를 받아서 점점 주황색 동그라미와 가까워진다. 그래서 최소 제곱을 GAN에 적용하게 되면 가짜 데이터가 판별자를 속일 정도로 정교해지면서 실제 데이터와 확실히 비슷해지는 효과가 있다.



(a) LSGAN







(b) DCGAN

LSGAN 학습 실험 결과를 보면 DCGAN과 비교할 수 있는데 구조는 거의 동일하게 진행하고 손실 함수만 다르게 조정해서 학습시킨 결과물을 보면 LSGAN의 결과물이 훨씬 선명하고 진짜 존재하는 것 같은 방 이미지를 생성한 것을 확인할 수 있다.

**CGAN은 Conditional GAN으로 조건부 GAN이다.** 기본 GAN의 확장으로, 생성자와 판별자가 데이터를 생성하거나 판별할 때 추가적인 조건을 고려하는 모델이다. 지금까지 본 GAN 들은 학습한 이미지와 유사한 사실적인 이미지를 생성할 수 있었지만, 이미지의 유형을 제어할 수 없었는데 이 문제를 해결한 것이 CGAN이다. 일반적인 GAN과의 차이점은 생성자와 판별자 모두 조건 정보를 입력으로 받는다는 것이다. 이 조건 정보는 특정 숫자 클래스, 이미지 속성 등 다양하고 생성자는 이 조건을 바탕으로 특정한 특성을 가진 데이터를 생성하게 된다. CGAN은 주어진 조건에 따라 데이터를 생성할 수 있으므로 멀티모달 데이터의 상관관계를 학습하는 데 효과적이다. 아래의 사진은 CGAN을 멀티모달 데이터에 적용한 예시이다. 텍스트-이미지 변환으로 텍스트 설명을 조

건으로 제공하고 이에 해당하는 이미지를 생성하는 작업이다. 텍스트를 임베딩해서 생성자의 입력으로 사용하고 그 텍스트 임베딩을 기반으로 이미지 특성을 조절한다. 여기서 임베딩이란 사람이 쓰는 자연어를 기계가 이해할 수 있는 숫자 나열인 벡터로 바꾼 결과나 과정 전체를 의미한다.

	User tags + annotations	Generated tags
	montanha, trem, inverno, frio, people, male, plant life, tree, structures, transport, car	taxi, passenger, line, transportation, railway station, passengers, railways, signals, rail, rails
	food, raspberry, delicious, homemade	chicken, fattening, cooked, peanut, cream, cookie, house made, bread, biscuit, bakes
	water, river	creek, lake, along, near, river, rocky, treeline, valley, woods, waters
	people, portrait, female, baby, indoor	love, people, posing, girl, young, strangers, pretty, women, happy, life

### 3. DCGAN 이미지 생성

DCGAN의 주요 특징으로 다양한 Layers를 사용하여 이미지를 처리해서 이미지의 공간적 구조를 유지하면서 더 효과적으로 특징을 추출하고 생성한다. kaggle에서 코드를 참고해 코랩에서 실행해봤다.

#### 3-1. 라이브러리 설정

```
import tensorflow as tf
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time
from IPython import display
```

#### 3-2. 데이터 준비

```
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()

train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype("float32")
train_images = (train_images - 127.5) / 127.5

BUFFER_SIZE = 60000
BATCH_SIZE = 100
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

MNIST 데이터셋을 로드하고 데이터 전처리를 해주기 위해 이미지를 (28, 28, 1) 형태로 변형하고 픽셀값을 [-1, 1] 범위로 정규화한다. `tf.data.Dataset`을 사용하여 데이터셋을 생성하고 무작위로 섞은 후 배치 처리를 한다.

### 3-3. 생성기 모델 정의

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7,7,256)))
    assert model.output_shape == (None,7,7,256)

    model.add(layers.Conv2DTranspose(128,(5,5),strides=(1,1),padding='same',use_bias=False))
    assert model.output_shape == (None,7,7,128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64,(5,5),strides=(2,2),padding='same',use_bias=False))
    assert model.output_shape == (None,14,14,64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1,(5,5),strides=(2,2),padding="same",use_bias=True,activation="tanh"))
    assert model.output_shape == (None,28,28,1)
    return model
```

**Dense Layer**를 사용해서 입력된 노이즈를 통해 7x7x256 크기의 텐서로 변환한다. 이 레이어는 생성 과정에 첫 단계로 낮은 차원의 노이즈를 고차원 텐서로 확장해주는 역할을 한다. **BatchNormalization**과 **LeakyReLU** 함수로 학습 안정성을 높이고 비선형성을 추가해서 더 복잡한 패턴을 학습할 수 있도록 해준다. **Conv2DTranspose** 업샘플링 레이어를 사용해서 텐서를 더 큰 차원인 28 \* 28 크기의 이미지로 생성하게 한다. **Tanh Activation**으로 출력 이미지를 -1에서 1 사이로 변환해서 정규화된 이미지 데이터와 일치하도록 만들어준다.

### 3-4. 판별자 모델 정의

```
def make_discriminator_model():
    model = tf.keras.Sequential()

    model.add(layers.Conv2D(64,(5,5),strides=(2,2),padding='same',input_shape=[28,28,1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128,(5,5),strides=(2,2),padding="same"))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

**Conv2D**로 이미지를 다운샘플링 하여 특징을 추출한다. 생성자 모델처럼 **LeakyReLU** 함수를 사용해 더 복잡한 패턴을 학습할 수 있게 하고 **Dropout** 함수로 모델이 과적합 되지 않도록 방지한다. **Flatten**과 **Dense**로 최종적으로 1개의 값을 출력하여 이미지가 실제인지 판별하게 된다.

### 3-5. 손실 함수와 옵티마이저

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

**BinaryCrossentropy** 이진 분류 손실 함수로, 판별자가 실제 이미지를 1로 예측하고 가짜 이미지를 0으로 정확히 예측하도록 학습한다. **Discriminator Loss**로 판별자의 손실을 계산한다. 실제 이미지에 대해 1의 레이블을 사용하고 가짜 이미지에 대해 0의 레이블을 사용해서 손실을 계산해 준다. **Generator Loss**로 생성자는 가짜 이미지를 실제처럼 보이도록 만드는 손실을 계산한다. 생성자는 판별자가 가짜 이미지를 1로 예측하도록 학습한다.

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

**Adam Optimizer**를 생성자와 판별자 모두 적용해서 파라미터를 업데이트한다.

### 3-6. 체크포인트와 이미지 생성 함수

```
checkpoint_dir = "./training_checkpoints"
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                  discriminator_optimizer=discriminator_optimizer,
                                  generator=generator,
                                  discriminator=discriminator)

EPOCHS = 200
noise_dim = 100
num_examples_to_generate = 16
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

체크포인트를 설정해서 학습 중간에 모델 상태를 저장하여 나중에 복원할 수 있게 한다. GAN 학습 특성상 시간이 길게 소요되는데 긴 학습 과정에서 발생하는 중단을 방지할 수 있다.

```
def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    fig = plt.figure(figsize=(4,4))
    for i in range(predictions.shape[0]):
        plt.subplot(4,4,i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis("off")
    plt.savefig("image_at_epoch_{:04d}.png".format(epoch))
    plt.show()
```



생성된 이미지를 저장하고 시각화하기 위해 `generate_and_save_images` 함수를 만들어준다. 주어진 에포크마다 생성된 이미지를 저장하여 학습 과정을 시각적으로 확인할 수 있다.

### 3-7. 모델 학습

```
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

학습 함수로 한 배치의 이미지를 사용해서 생성자와 판별자의 손실을 계산하고 그라디언트를 업데이트한다. 두 개의 `GradientTape`을 사용하여 각 네트워크의 손실에 대한 그라디언트를 계산한다. 여기서 그라디언트는 수학적 관점으로 볼 때 함수의 기울기를 나타내고 기계 학습에서 볼 땐 경사 하강법 역할을 한다. 손실 함수를 최소화하는 파라미터를 찾기 위해 경사 하강법을 사용해 손실 함수의 기울기를 계산하여 파라미터를 업데이트하게 되는데 이 역할을 그라디언트가 맡게 된다.

```
def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

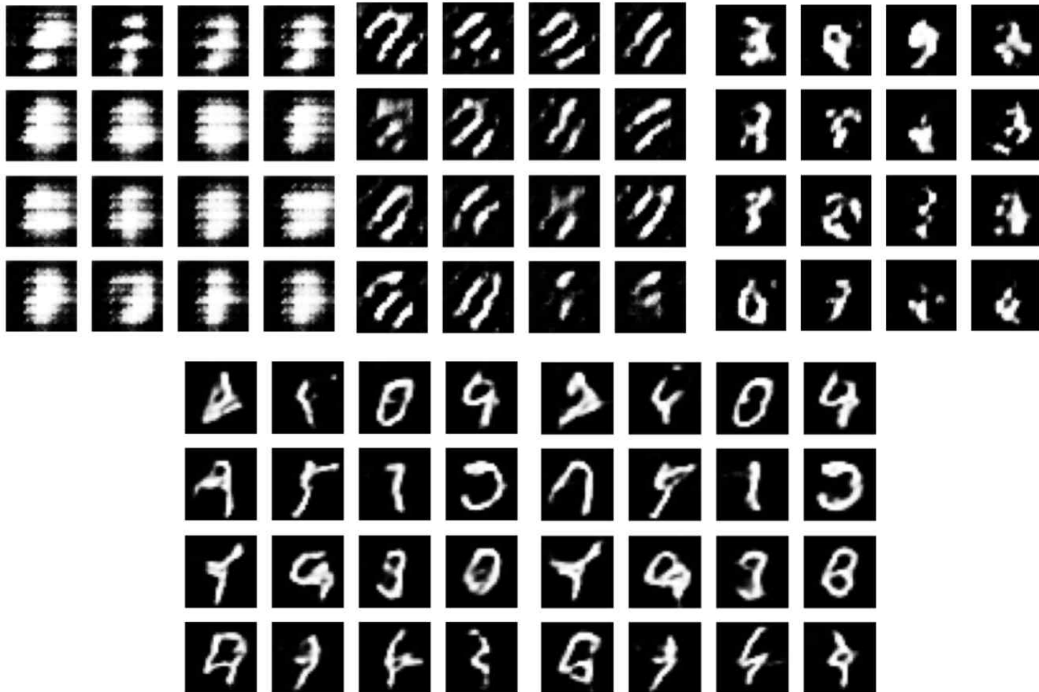
        for image_batch in dataset:
            train_step(image_batch)
            display.clear_output(wait=True)
            generate_and_save_images(generator, epoch+1, seed)
            if (epoch + 1) % 150 == 0:
                checkpoint.save(file_prefix=checkpoint_prefix)
            print('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))
            display.clear_output(wait=True)
            generate_and_save_images(generator, epochs, seed)
```

지정된 에포크 수만큼 모델을 학습시킨다. 기본 GAN 이미지 생성 코드에서 에포크 수를 200으로 설정했기 때문에 200으로 설정했고 에포크마다 `train_step`을 호출하여 모델을 학습하고 학습된 이미지를 저장해 결과를 확인한다. 150 스텝마다 체크포인트를 저장하여 진행 상황을 보존한다.

```
train(train_dataset, EPOCHS)
```

생성되는 이미지가 출력되도록 불러와 준다.

### 3-8. 생성된 이미지 분석



왼쪽부터 시간이 지남에 따라 생성된 이미지 모습이고 오른쪽 아래에 있는 이미지가 최종 출력된 이미지이다. 에포크의 수가 늘어남에 따라 반복 학습이 늘어난다는 뜻인데 이미지를 확인해보니 이미지의 형태가 점점 선명해지는 걸 확인할 수 있고 중복된 이미지의 개수가 기본 GAN보다 적은 걸 확인할 수 있었다. 그리고 3, 7, 9 등 생성된 숫자의 종류가 기본 GAN에 비해 더 다양했다. 기본 GAN 구조에서 CNN(Convolutional Neural Networks)을 적용하여 이미지 생성 실습을 진행했는데 Batch Normalization 배치 정규화를 사용해서 입력 데이터가 치우쳐져 있으면 평균과 분산을 조정해주고 각 레이어에 제대로 전달되도록 학습을 진행하게 해서 중간에 끊기더라도 계속해서 학습이 진행되어 확실히 기본 GAN보다 안정성 부분에서 우수하다고 느꼈다.

## 4. LSGAN 이미지 생성

생성자와 판별자 간의 손실 함수를 최소 제곱 오류로 정의해서 안정적인 학습을 목표로 하는 LSGAN을 이용해서 이미지를 생성하고자 한다.

### 4-1. 라이브러리 설정

```
[1] import numpy as np
import pandas as pd

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```



```
[2] import cv2
import torch
import os
import numpy as np
import pandas as pd
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import torchvision
import shutil
import imageio
from IPython import display
from PIL import Image
from glob import glob
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader
from torchvision.utils import make_grid, save_image
```

필요한 라이브러리를 임포트해준다. 데이터 처리, 이미지 변환, 모델 학습, 시각화 관련 라이브러리들을 포함했다.

#### 4-2. 이미지 전처리

```
transform = transforms.Compose([
    transforms.Resize(32),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5), std=(0.5))
])

data = torchvision.datasets.MNIST(root='./mnist', train=True, download=True, transform=transform)

batch_size = 100
dataloader = DataLoader(data, batch_size=batch_size, shuffle=True)

z_dim = 100
image_dim = (1, 32, 32)
num_classes = 10
```

MNIST 데이터셋을 내려받고 이미지를 32x32로 리사이즈한다. Tensor로 변환하고 정규화해서 데이터 로더를 통해 배치 단위로 불러올 수 있게 설정한다.

#### 4-3. 모델 가중치 초기화

생성자와 판별자의 가중치를 정규 분포로 초기화하는 함수를 만들어줘서 모델 학습의 안정성을 높여준다.

```
[9] def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find("BatchNorm2d") != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)
```

#### 4-4. 생성자 정의

```
[10] class Generator(nn.Module):
    def __init__(self, z_dim):
        super(Generator, self).__init__()

        self.z_dim = z_dim

        self.model = nn.Sequential(
            nn.Linear(z_dim, 128 * (8) ** 2),
            nn.BatchNorm1d(128 * (8) ** 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Unflatten(1, (128, 8, 8)),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 1, kernel_size=3, stride=1, padding=1),
            nn.Tanh()
        )

    def forward(self, noise):
        image = self.model(noise)
        return image
```

Linear 랜덤 노이즈 벡터인 `z_dim`을 입력받아서 고차원으로 변환한다. 이 경우 128x8x8차원의 텐서로 변환한다. `BatchNorm1d`로 배치 정규화를 해서 내부 공변량 이동을 줄여 학습 안정성을 높인다. `LeakyReLU` 비선형 활성화 함수를 사용해주고 `Conv2d` 합성곱 레이어를 통해 이미지의 세부 특징을 학습해준다. 그리고 `Tanh`로 출력 이미지의 픽셀값을 -1과 1 사이로 변환해서 정규화된 입력 이미지에 적합하게 만든다.

#### 4-5. 판별자 정의

```
[12] class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.25),
```

`Conv2d`로 합성곱 레이어를 사용해서 이미지의 특징을 추출한다. 판별자도 마찬가지로 `LeakyReLU` 함수를 사용해 비선형성을 추가해서 모델의 표현력을 증가시킨다. `Dropout2d`를 사용해서 오버피팅을 방지하기 위해 일부 뉴런을 무작위로 비활성화시킨다. 그리고 레이어 출력을 정규화하고 `Flatten`, `Linear`를 이용해 1차원으로 변환한 뒤 판별 결과를 출력한다.

```

        nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1),
        nn.BatchNorm2d(32, 0.8),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Dropout2d(0.25),

        nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
        nn.BatchNorm2d(64, 0.8),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Dropout2d(0.25),

        nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
        nn.BatchNorm2d(128, 0.8),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Dropout2d(0.25),

        nn.Flatten(),
        nn.Linear(128 * 2 ** 2, 1)
    )

    def forward(self, img):
        out = self.model(img)
        return out

```

#### 4-6. 학습 루프

```

import os
import torch
import torch.optim as optim
import torch.nn as nn
from torchvision.utils import save_image
import numpy as np
import matplotlib.pyplot as plt

# Optimizers
generator_optimizer = optim.Adam(generator.parameters(), lr=0.0001, betas=(0.5, 0.995))
discriminator_optimizer = optim.Adam(discriminator.parameters(), lr=0.0001, betas=(0.5, 0.995))

# Loss function
adversarial_loss = nn.MSELoss()

n_epochs = 200

```

전체 구조로는 학습 루프가 여러 에포크 동안 각 배치에 대해 생성자와 판별자를 번갈아 가며 학습하게 된다. 앞서 기본 GAN, DCGAN의 에포크를 200으로 정해놓았으니 LSGAN도 200만큼 반복하도록 n\_epochs를 for 문으로 반복하도록 해줬다. 그리고 Dataloader로 데이터를 배치 단위로 불러와 주고 valid, fake를 사용해 실제 이미지는 1, 가짜 이미지는 0으로 라벨을 생성해준다.

생성자 학습 부분에서 `generator_optimizer.zero_grad()`로 생성자에 대한 기울기를 초기화해 주고 생성자가 사용할 입력인 랜덤 노이즈 벡터 `z`를 생성해준다. `generated_image = generator(z)`로 생성자에게 노이즈를 입력해서 가짜 이미지를 생성한다. `g_loss`로 생성된 이미지가 판별자를 통과한 후의 출력과 `valid` 레이블을 비교해서 손실을 계산한다. 이때 `adversarial_loss`는 생성자가 얼마나 잘했는지를 평가하는 지표이다. `g_loss.backward()` 역전파로 손실을 기반으로 기울기를 계산하여 생성자의 파라미터를 업데이트할 준비를 한다. `generator_optimizer.step()`으로 계산된 기울기를 사용해 생성자의 가중치를 업데이트해준다.

판별자 학습 부분은 마찬가지로 기울기를 초기화해주고 `real_loss`와 `fake_loss`로 실제 이미지, 가짜 이미지 손실을 각각 계산해준다. `fake_loss`에서 생성된 이미지를 판별자에 통과시켜 가짜 이미지 손실을 계산하게 되는데 여기서 `detach()`를 이용해 생성된 이미지의 기울기가 판별자에 영향을 주지 않도록 해준다. `d_loss`로 전체 손실을 계산해주고 `backward`로 손실 기반 기울기를 계산한 후 가중치를 업데이트한다. 150번째 배치마다 현재 에포크, 스텝, 판별자 손실, 생성자 손실을 출력해서 학습 과정을 모니터링하고 에포크마다 생성된 이미지를 저장해서 `reshape`를 통해 이미지를 적절한 형태로 변환하고 `save_image`를 사용해 파일로 저장한 후 각 에포크가 끝나면 출력하도록 해준다.

```
os.makedirs("./lsgan", exist_ok=True)

d_loss_s = []
g_loss_s = []

for epoch in range(n_epochs):
    for batch_idx, (images, _) in enumerate(dataloader):
        real_images = images.to(device)

        valid = torch.ones(images.shape[0], 1, device=device)
        fake = torch.zeros(images.shape[0], 1, device=device)

        # Train generator
        generator_optimizer.zero_grad()
        z = torch.randn(images.shape[0], z_dim, device=device)
        generated_images = generator(z)
        g_loss = adversarial_loss(discriminator(generated_images), valid)
        g_loss.backward()
        generator_optimizer.step()

        # Train discriminator
        discriminator_optimizer.zero_grad()
        real_loss = adversarial_loss(discriminator(real_images), valid)
        fake_loss = adversarial_loss(discriminator(generated_images.detach()), fake)
        d_loss = 0.5 * (real_loss + fake_loss)
        d_loss.backward()
        discriminator_optimizer.step()
```

```
if (batch_idx + 1) % 150 == 0:
    print(f"[Epoch {epoch + 1}/{n_epochs}] [Batch {batch_idx + 1}/{len(dataloader)}] [D loss: {d_loss:.6f}] [G loss: {g_loss:.6f}]")

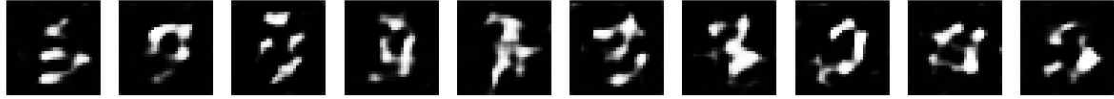
# Save generated images at the end of each epoch
samples = generated_images.data.reshape(-1, 1, 32, 32)
save_image(samples, f"./lsgan/epoch_{epoch + 1}.png", nrow=10, normalize=True)

# Display the generated images
samples = samples.cpu().detach().numpy()
samples = np.transpose(samples, (0, 2, 3, 1)) # Convert from (N, 1, 32, 32) to (N, 32, 32, 1)

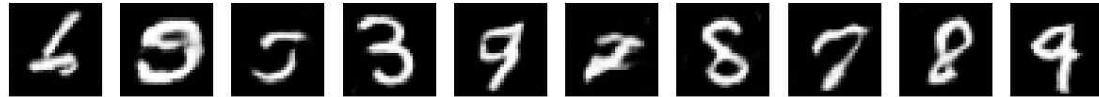
fig, ax = plt.subplots(1, 10, figsize=(15, 15))
for idx in range(10):
    ax[idx].imshow(samples[idx].squeeze(), cmap='gray')
    ax[idx].axis('off')
plt.show()
```

#### 4-7. 생성된 이미지 분석

[Epoch 1/200] [Batch 150/600] [D loss: 0.255564] [G loss: 0.257187]  
[Epoch 1/200] [Batch 300/600] [D loss: 0.264141] [G loss: 0.250276]  
[Epoch 1/200] [Batch 450/600] [D loss: 0.255836] [G loss: 0.248384]  
[Epoch 1/200] [Batch 600/600] [D loss: 0.250614] [G loss: 0.241544]



[Epoch 200/200] [Batch 150/600] [D loss: 0.035833] [G loss: 0.608653]  
[Epoch 200/200] [Batch 300/600] [D loss: 0.110417] [G loss: 0.874984]  
[Epoch 200/200] [Batch 450/600] [D loss: 0.051437] [G loss: 0.521327]  
[Epoch 200/200] [Batch 600/600] [D loss: 0.037905] [G loss: 0.813310]



1 에포크와 200 에포크 상태의 이미지를 비교해보면 1 에포크 땀 테두리가 선명하지 않고 흘러내리는 모습처럼 글씨가 선명하지 않은 걸 확인할 수 있다. 200 에포크 이미지를 확인하면 확실히 1 에포크 때보다 테두리가 선명하고 숫자로 인식할 수 있는 이미지가 생성되었다. 판별자 손실과 생성자 손실을 보면 1 에포크에 비해 200 에포크의 최저 판별자 손실이 더 낮은 걸 확인할 수 있는 반면, 생성자 손실은 비교적 높은 걸 확인할 수 있다.

### 5. CGAN 이미지 생성

Conditional GAN은 기존 GAN을 확장해서 특정 조건에 따라 이미지나 데이터를 생성할 수 있도록 만든 모델이다. GAN의 생성자와 판별자에 원하는 조건을 부여해서 생성된 데이터가 특정 조건을 따르도록 훈련한다. 그래서 코드를 작성할 때 기본 GAN 코드를 응용해 작성했다. 기본적인 형태는 기본 GAN과 같아서 CGAN을 구현한 코드와 기본 GAN 코드의 차이점 위주로 설명하고자 한다.

#### 5-1. 조건 인코딩 추가

기본 GAN에서는 레이블을 고려하지 않고 노이즈만을 입력으로 사용해서 이미지를 생성하는데 CGAN에서는 특정 클래스(숫자 0~9)로 이미지를 생성할 수 있도록 레이블을 조건으로 사용해준다. 레이블은 원-핫 인코딩(one-hot encoding)해서 이미지 데이터와 함께 네트워크에 입력된다.

```
# Hyper-parameters & Variables setting
num_epoch = 35
batch_size = 100
learning_rate = 0.0002
img_size = 28 * 28
num_channel = 1
dir_name = "CGAN_results"

noise_size = 100
hidden_size1 = 256
hidden_size2 = 512
hidden_size3 = 1024

***
FOR CONDITIONAL GAN
***

# The number of MNIST's class label is 10
condition_size = 10
```

```

"""
Training part
"""
for epoch in range(num_epoch):
    for i, (images, label) in enumerate(data_loader):

        # make ground truth (labels) -> 1 for real, 0 for fake
        real_label = torch.full((batch_size, 1), 1, dtype=torch.float32).to(device)
        fake_label = torch.full((batch_size, 1), 0, dtype=torch.float32).to(device)

        # reshape real images from MNIST dataset
        real_images = images.reshape(batch_size, -1).to(device)

        """
        FOR CONDITIONAL GAN
        """

        # Encode MNIST's label's with 'one hot encoding'
        label_encoded = F.one_hot(label, num_classes=10).to(device)
        # concat real images with 'label encoded vector'
        real_images_concat = torch.cat((real_images, label_encoded), 1)

```

## 5-2. 입력 구조 차이

```

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

        self.linear1 = nn.Linear(img_size + condition_size, hidden_size3)
        self.linear2 = nn.Linear(hidden_size3, hidden_size2)
        self.linear3 = nn.Linear(hidden_size2, hidden_size1)
        self.linear4 = nn.Linear(hidden_size1, 1)
        self.leaky_relu = nn.LeakyReLU(0.2)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.leaky_relu(self.linear1(x))
        x = self.leaky_relu(self.linear2(x))
        x = self.leaky_relu(self.linear3(x))
        x = self.linear4(x)
        x = self.sigmoid(x)
        return x

```

```

"""
FOR CONDITIONAL GAN
"""

# concat noise vector z with encoded labels
z_concat = torch.cat((z, label_encoded), 1)
fake_images = generator(z_concat)
fake_images_concat = torch.cat((fake_images, label_encoded), 1)

# Compare result of discriminator with fake images & real labels
# If generator deceives discriminator, g_loss will decrease
g_loss = criterion(discriminator(fake_images_concat), real_label)

# Train generator with backpropagation
g_loss.backward()
g_optimizer.step()

```

기본 GAN 코드에 레이블 정보가 없어서 생성자와 판별자는 단순히 img\_size나 noise\_size 형태의 입력을 받게 되고 CGAN은 생성자와 판별자의 입력을 img\_size + condition\_size나 noise\_size + condition\_size로 입력받게 된다. 생성자와 판별자 정의도 마찬가지로 CGAN에서 각 레이어에 레이블 정보를 포함해서 학습한다.



```

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

        self.linear1 = nn.Linear(noise_size + condition_size, hidden_size1)
        self.linear2 = nn.Linear(hidden_size1, hidden_size2)
        self.linear3 = nn.Linear(hidden_size2, hidden_size3)
        self.linear4 = nn.Linear(hidden_size3, img_size)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.relu(self.linear1(x))
        x = self.relu(self.linear2(x))
        x = self.relu(self.linear3(x))
        x = self.linear4(x)
        x = self.tanh(x)
        return x

'''
FOR CONDITIONAL GAN
'''
# concat noise vector z with encoded labels
z_concat = torch.cat((z, label_encoded), 1)
fake_images = generator(z_concat)
fake_images_concat = torch.cat((fake_images, label_encoded), 1)

# Calculate fake & real loss with generated images above & real images
fake_loss = criterion(discriminator(fake_images_concat), fake_label)
real_loss = criterion(discriminator(real_images_concat), real_label)
d_loss = (fake_loss + real_loss) / 2

```

### 5-3. 조건 체크 함수 유무

레이블이 없는 GAN에는 조건에 따른 이미지 생성 기능이 없고 CGAN에서 학습이 끝난 다음 특정 레이블을 조건으로 설정해서 check\_condition() 함수를 통해 조건에 맞는 이미지를 생성하게 된다. 이 함수로 CGAN이 정상적으로 작동하는지 확인할 수 있다.

```

# For checking CGAN's validity in final step
def check_condition(_generator):
    test_image = torch.empty(0).to(device)

    for i in range(10):
        test_label = torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
        test_label_encoded = F.one_hot(test_label, num_classes=10).to(device)

        # create noise(latent vector) 'z'
        _z = torch.randn(10, noise_size).to(device)
        _z_concat = torch.cat((_z, test_label_encoded), 1)

        test_image = torch.cat((test_image, _generator(_z_concat)), 0)

    _result = test_image.reshape(100, 1, 28, 28)
    save_image(_result, os.path.join(dir_name, 'CGAN_test_result.png'), nrow=10)

```



## 5-4. 생성된 이미지 분석

```
Epoch [ 1/200 ] Step [ 150/600 ] d_loss : 0.04831 g_loss : 2.91128
Epoch [ 1/200 ] Step [ 300/600 ] d_loss : 0.02296 g_loss : 13.38792
Epoch [ 1/200 ] Step [ 450/600 ] d_loss : 0.00307 g_loss : 6.38136
Epoch [ 1/200 ] Step [ 600/600 ] d_loss : 0.00154 g_loss : 6.73907
Epoch 1's discriminator performance : 0.98 generator performance : 0.00
```



```
Epoch [ 200/200 ] Step [ 150/600 ] d_loss : 0.50695 g_loss : 1.93208
Epoch [ 200/200 ] Step [ 300/600 ] d_loss : 0.48571 g_loss : 1.27048
Epoch [ 200/200 ] Step [ 450/600 ] d_loss : 0.46676 g_loss : 0.91437
Epoch [ 200/200 ] Step [ 600/600 ] d_loss : 0.47632 g_loss : 1.47756
Epoch 200's discriminator performance : 0.98 generator performance : 0.00
```



CGAN 이미지 생성 결과이다. 판별자 손실인 d\_loss의 1 에포크 결과를 보면 0.04831에서 100 에포크일 때 d\_loss가 0.47632로 작은 차이로 감소한 것을 볼 수 있다. 이는 판별자가 작동을 잘 하지 못하고 있다는 걸 의미한다. 생성자 손실인 g\_loss를 보면 1 에포크일 때 2.91128이고 100 에포크일 때 1.47756으로 단순히 평면적인 값만 봤을 땐 감소량이 적어보이지만 전체 스텝을 보면 최대가 13.38792이고 100 에포크일 때 최대가 0.91437로 전체적으로 봤을 때 확실히 감소한 모습을 볼 수 있다. 이는 생성자가 더 좋은 이미지를 생성하고 있다는 뜻이다. 그리고 판별자 성능이 0.98로 상대적으로 높는데 이는 생성자가 판별자를 속이는데, 어려움을 겪고 있다는 것을 나타낸다.

## 6. 각 모델 FID 작성

GAN 모델별 성능을 측정하기 위해 FID 지표를 사용할 것이다. FID 는 Fréchet Inception Distance의 약자로 생성 모델의 성능을 평가하기 위해 많이 사용하는 지표다. FID 는 생성된 이미지와 실제 이미지 분포 차이를 측정해서 생성 모델이 얼마나 현실적인 이미지를 생성하는지 평가한다. FID 값이 낮을수록 생성된 이미지가 실제 데이터와 비슷함을 의미한다.

FID 두 개의 가우시안 분포 간의 거리를 계산하는 방식으로 이루어진다. 동작하기 위해 3가지 과정이 필요하다. 먼저 특징 추출로 생성된 이미지와 실제 이미지를 Inception v3와 같은 신경망에 입력해서 특징 벡터를 추출한다. 여기서 이미지의 임베딩 공간에서 특징을 비교하게 되는데 보통 Inception v3 모델의 마지막 레이어 바로 전의 특징 벡터를 사용한다. 두 번째로 평균과 공분산을 계산하게 된다. 실제 이미지와 생성된 이미지의 특징 벡터에 대해 각각 평균 벡터와 공분산 행렬을 계산한다. 이를 통해 각 이미지 집합의 분포를 표현할 수 있게 된다. 마지막으로 Fréchet 거리를 계산한다. 두 분포의 평균 벡터와 공분산 행렬을 사용해서 다음 공식을 통해 Fréchet 거리를 계산한다.

$$FID = \|\mu_{real} - \mu_{fake}\|^2 + Tr(\Sigma_{real} + \Sigma_{fake} - 2(\Sigma_{real} \cdot \Sigma_{fake})^{1/2})$$

여기서  $\mu_{real}$ 과  $\mu_{fake}$ 는 실제 이미지와 생성된 이미지의 평균 벡터를 나타내고  $\Sigma_{real}$ 과  $\Sigma_{fake}$ 는 공분산 행렬이다. 다음으로 FID 지표를 계산하기 위해 작성한 코드를 바탕으로 FID 계산이 어떻게 수행되는지 살펴보자.

## 6-1. GAN FID 코드

`get_inception_features` 함수는 Inception v3 모델을 사용해서 이미지를 입력받아 특징 벡터를 추출한다. 이 함수는 torchvision.models에서 사전 학습된 Inception v3 모델을 가져오고 이를 통해 실제 및 생성된 이미지에서 특징을 추출한다.

```
def get_inception_features(images):
    model = torchvision.models.inception_v3(pretrained=True, transform_input=False).to(device)
    model.fc = nn.Identity()
    model.eval()
    images = images.repeat(1, 3, 1, 1)
    with torch.no_grad():
        features = model(images).cpu().numpy()
    return features
```

`calculate_fid` 함수에서는 추출한 특징 벡터들을 사용해서 실제 이미지와 생성된 이미지의 평균 벡터(mu\_real, mu\_fake)와 공분산 행렬(sigma\_real, sigma\_fake)을 계산한다. 그 후 두 분포 간의 Frechet 거리를 계산해서 FID 점수를 반환하게 된다.

```
def calculate_fid(real_features, fake_features):
    mu_real, sigma_real = real_features.mean(0), np.cov(real_features, rowvar=False)
    mu_fake, sigma_fake = fake_features.mean(0), np.cov(fake_features, rowvar=False)
    diff = mu_real - mu_fake
    covmean = sqrtm(sigma_real @ sigma_fake).real
    return (diff @ diff) + np.trace(sigma_real + sigma_fake - 2 * covmean)
```

`train_model` 함수에서 각 에포크가 끝날 때마다 FID를 계산해서 GAN 모델이 학습하는 동안 성능을 평가한다. `real_images_resized`와 `fake_images_resized`는 Inception 모델의 입력 크기에 맞게 리사이즈 시키고 `get_inception_features`를 사용해 각 이미지를 특징 벡터로 변환해줘서 `calculate_fid` 함수로 두 특징 벡터의 FID 점수를 계산한다. 이렇게 계산된 FID 점수는 `fid_scores` 리스트에 저장되고 첫 번째 에포크와 최종 에포크를 출력해서 결과값을 보여준다.

```
def train_model(generator, discriminator, gan_type="GAN"):
    g_optimizer = optim.Adam(generator.parameters(), lr=learning_rate)
    d_optimizer = optim.Adam(discriminator.parameters(), lr=learning_rate)
    criterion = nn.BCELoss()
    fid_scores = []

    for epoch in range(num_epoch):
        for i, (images, _) in enumerate(data_loader):
            real_images = images.view(batch_size, -1).to(device)
            real_label = torch.full((batch_size, 1), 1, device=device).float()
            fake_label = torch.full((batch_size, 1), 0, device=device).float()
            z = torch.randn(batch_size, noise_size).to(device)
```

```
            real_images_resized = F.interpolate(images, size=(299, 299), mode='bilinear')
            fake_images_resized = F.interpolate(fake_images.view(-1, 1, 28, 28), size=(299, 299), mode='bilinear')
            real_features = get_inception_features(real_images_resized)
            fake_features = get_inception_features(fake_images_resized)
            fid_score = calculate_fid(real_features, fake_features)
            fid_scores.append(fid_score)

            if epoch == 0 or epoch == num_epoch - 1:
                print(f"Epoch [{epoch+1}/{num_epoch}], {gan_type} FID: {fid_score:.2f}")

        print(f"{gan_type} 최종 FID: {fid_scores[-1]:.2f}")
        return fid_scores

generator_gan = GeneratorGAN().to(device)
discriminator_gan = DiscriminatorGAN().to(device)
train_model(generator_gan, discriminator_gan, gan_type="GAN")
```

## 6-2. DCGAN FID 코드

기본 GAN과 달리 DCGAN은 합성곱 층을 사용해서 이미지 생성과 판별을 수행하고 훈련 안정성을 높이기 위해 **BatchNormalization**과 **LeakyReLU**를 사용했다.

```
class GeneratorDCGAN(nn.Module):
    def __init__(self):
        super(GeneratorDCGAN, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(noise_size, 1024, 4, 1, 0, bias=False),
            nn.BatchNorm2d(1024),
            nn.ReLU(True),
            nn.ConvTranspose2d(1024, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, num_channel, 4, 2, 1, bias=False),
            nn.Tanh()
        )
```

```
class DiscriminatorDCGAN(nn.Module):
    def __init__(self):
        super(DiscriminatorDCGAN, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(num_channel, 256, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1024, 4, 2, 1, bias=False),
            nn.BatchNorm2d(1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(1024, 1, 4, 1, 0, bias=False),
        )
```

## 6-3. LSGAN FID 코드

LSGAN에서는 **F.mse\_loss**를 사용해서 손실을 계산해줬다. 생성자는 판별자로부터 1을 기대하고 판별자는 진짜와 가짜를 1과 0으로 구별하게 된다. LSGAN에서도 판별자 아키텍처는 기본 GAN과 비슷하지만 MSE 손실을 계산하기 위해 마지막 출력 레이어에서 sigmoid 함수를 제거해줬다. 여기서 **MSE**란 예측값과 실제값 간의 차이를 제공한 평균값을 계산하는 손실 함수다. LSGAN에서 생성자와 판별자가 훈련할 때 MSE 손실을 사용해서 두 값 간의 차이를 최소화하는 역할로 사용되기도 한다.

```
def train_model(generator, discriminator, gan_type="LSGAN"):
    g_optimizer = optim.Adam(generator.parameters(), lr=learning_rate)
    d_optimizer = optim.Adam(discriminator.parameters(), lr=learning_rate)
    fid_scores = []
```

```

for epoch in range(num_epoch):
    for i, (images, _) in enumerate(data_loader):
        real_images = images.view(batch_size, -1).to(device)
        z = torch.randn(batch_size, noise_size).to(device)

        g_optimizer.zero_grad()
        fake_images = generator(z)
        fake_output = discriminator(fake_images)
        g_loss = F.mse_loss(fake_output, torch.ones(batch_size, 1, device=device))
        g_loss.backward()
        g_optimizer.step()

        d_optimizer.zero_grad()
        real_output = discriminator(real_images)
        d_loss_real = F.mse_loss(real_output, torch.ones(batch_size, 1, device=device))
        fake_output = discriminator(fake_images.detach())
        d_loss_fake = F.mse_loss(fake_output, torch.zeros(batch_size, 1, device=device))
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        d_optimizer.step()

```

#### 6-4. CGAN FID 코드

CGAN은 특정 조건에 대한 이미지를 생성할 수 있도록 설계되어 있어서 원하는 조건에 따라 해당하는 이미지를 생성할 수 있다. 그래서 생성자와 판별자에서 조건 레이블을 입력으로 받아 사용할 수 있도록 수정했고 입력받은 레이블은 원-핫 인코딩을 사용하도록 작성했다. 그리고 생성자와 판별자의 입력으로 노이즈와 레이블을 결합해 사용하고 훈련 중 생성자와 판별자 모두 레이블 정보를 사용할 수 있도록 변경해줬다.

```

class GeneratorCGAN(nn.Module):
    def __init__(self):
        super(GeneratorCGAN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(noise_size + num_classes, hidden_size1),
            nn.ReLU(True),
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU(True),
            nn.Linear(hidden_size2, img_size),
            nn.Tanh()
        )

    def forward(self, noise, labels):
        input = torch.cat((noise, labels), dim=1) #노이즈와 레이블 연결부분
        return self.model(input)

```

```

class DiscriminatorCGAN(nn.Module):
    def __init__(self):
        super(DiscriminatorCGAN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(img_size + num_classes, hidden_size2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_size2, hidden_size1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_size1, 1),
            nn.Sigmoid()
        )

    def forward(self, input, labels):
        input = torch.cat((input, labels), dim=1) #이미지와 레이블 연결부분
        return self.model(input)

```



```

for epoch in range(num_epoch):
    for i, (images, labels) in enumerate(data_loader):
        real_images = images.view(batch_size, -1).to(device)
        labels_onehot = F.one_hot(labels, num_classes).float().to(device) #one-hot 인코딩
        real_label = torch.full((batch_size, 1), 1, device=device).float()
        fake_label = torch.full((batch_size, 1), 0, device=device).float()
        z = torch.randn(batch_size, noise_size).to(device)

```

## 7. FID 비교 및 정리

Epoch [1/200], GAN FID: 540.90	Epoch [100/200], GAN FID: 169.60	Epoch [190/200], GAN FID: 106.10
Epoch [2/200], GAN FID: 417.77	Epoch [101/200], GAN FID: 147.59	Epoch [191/200], GAN FID: 98.16
Epoch [3/200], GAN FID: 436.63	Epoch [102/200], GAN FID: 148.41	Epoch [192/200], GAN FID: 102.54
Epoch [4/200], GAN FID: 426.84	Epoch [103/200], GAN FID: 134.68	Epoch [193/200], GAN FID: 101.12
Epoch [5/200], GAN FID: 479.18	Epoch [104/200], GAN FID: 147.67	Epoch [194/200], GAN FID: 98.14
Epoch [6/200], GAN FID: 358.61	Epoch [105/200], GAN FID: 134.18	Epoch [195/200], GAN FID: 97.31
Epoch [7/200], GAN FID: 339.82	Epoch [106/200], GAN FID: 142.29	Epoch [196/200], GAN FID: 87.02
Epoch [8/200], GAN FID: 348.08	Epoch [107/200], GAN FID: 122.82	Epoch [197/200], GAN FID: 92.22
Epoch [9/200], GAN FID: 324.93	Epoch [108/200], GAN FID: 153.14	Epoch [198/200], GAN FID: 100.66
Epoch [10/200], GAN FID: 299.91	Epoch [109/200], GAN FID: 140.03	Epoch [199/200], GAN FID: 98.92
	Epoch [110/200], GAN FID: 154.11	Epoch [200/200], GAN FID: 103.69
		GAN 최종 FID: 103.69

먼저 GAN의 FID 결과는 540.90 → 103.69로 총 437.21 차이만큼 낮은 값으로 계산되었다.

Epoch [1/200], DCGAN FID: 314.34	Epoch [100/200], DCGAN FID: 52.47	Epoch [190/200], DCGAN FID: 46.34
Epoch [2/200], DCGAN FID: 238.67	Epoch [101/200], DCGAN FID: 53.12	Epoch [191/200], DCGAN FID: 45.89
Epoch [3/200], DCGAN FID: 270.55	Epoch [102/200], DCGAN FID: 51.85	Epoch [192/200], DCGAN FID: 45.57
Epoch [4/200], DCGAN FID: 186.35	Epoch [103/200], DCGAN FID: 52.36	Epoch [193/200], DCGAN FID: 46.01
Epoch [5/200], DCGAN FID: 101.40	Epoch [104/200], DCGAN FID: 52.91	Epoch [194/200], DCGAN FID: 45.92
Epoch [6/200], DCGAN FID: 108.73	Epoch [105/200], DCGAN FID: 53.05	Epoch [195/200], DCGAN FID: 45.74
Epoch [7/200], DCGAN FID: 82.00	Epoch [106/200], DCGAN FID: 54.13	Epoch [196/200], DCGAN FID: 46.19
Epoch [8/200], DCGAN FID: 82.91	Epoch [107/200], DCGAN FID: 53.58	Epoch [197/200], DCGAN FID: 45.42
Epoch [9/200], DCGAN FID: 73.62	Epoch [108/200], DCGAN FID: 54.22	Epoch [198/200], DCGAN FID: 45.66
Epoch [10/200], DCGAN FID: 66.34	Epoch [109/200], DCGAN FID: 53.77	Epoch [199/200], DCGAN FID: 45.81
	Epoch [110/200], DCGAN FID: 52.64	Epoch [200/200], DCGAN FID: 44.99
		DCGAN 최종 FID: 44.99

다음은 DCGAN의 FID 결과이다. 314.34 → 44.99로 총 269.35만큼의 차이가 벌어진 값으로 계산되었다.

Epoch [1/200], LSGAN FID: 532.52	Epoch [100/200], LSGAN FID: 115.76	Epoch [190/200], LSGAN FID: 92.60
Epoch [2/200], LSGAN FID: 449.82	Epoch [101/200], LSGAN FID: 122.74	Epoch [191/200], LSGAN FID: 97.25
Epoch [3/200], LSGAN FID: 419.16	Epoch [102/200], LSGAN FID: 113.96	Epoch [192/200], LSGAN FID: 92.33
Epoch [4/200], LSGAN FID: 376.67	Epoch [103/200], LSGAN FID: 118.61	Epoch [193/200], LSGAN FID: 101.93
Epoch [5/200], LSGAN FID: 413.96	Epoch [104/200], LSGAN FID: 122.17	Epoch [194/200], LSGAN FID: 86.55
Epoch [6/200], LSGAN FID: 396.20	Epoch [105/200], LSGAN FID: 125.53	Epoch [195/200], LSGAN FID: 101.02
Epoch [7/200], LSGAN FID: 288.73	Epoch [106/200], LSGAN FID: 113.57	Epoch [196/200], LSGAN FID: 90.72
Epoch [8/200], LSGAN FID: 284.29	Epoch [107/200], LSGAN FID: 116.94	Epoch [197/200], LSGAN FID: 108.50
Epoch [9/200], LSGAN FID: 259.69	Epoch [108/200], LSGAN FID: 122.30	Epoch [198/200], LSGAN FID: 95.32
Epoch [10/200], LSGAN FID: 267.64	Epoch [109/200], LSGAN FID: 137.93	Epoch [199/200], LSGAN FID: 97.57
	Epoch [110/200], LSGAN FID: 116.75	Epoch [200/200], LSGAN FID: 94.01
		LSGAN 최종 FID: 94.01

LSGAN의 FID 결과이다. 532.52 → 94.01로 총 438.51만큼 차이가 난 값으로 계산되었다.

Epoch [1/200], CGAN FID: 605.47	Epoch [100/200], CGAN FID: 124.62	Epoch [190/200], CGAN FID: 83.51
Epoch [2/200], CGAN FID: 471.86	Epoch [101/200], CGAN FID: 138.78	Epoch [191/200], CGAN FID: 87.89
Epoch [3/200], CGAN FID: 445.55	Epoch [102/200], CGAN FID: 133.18	Epoch [192/200], CGAN FID: 97.53
Epoch [4/200], CGAN FID: 415.27	Epoch [103/200], CGAN FID: 129.69	Epoch [193/200], CGAN FID: 86.98
Epoch [5/200], CGAN FID: 461.10	Epoch [104/200], CGAN FID: 132.31	Epoch [194/200], CGAN FID: 82.97
Epoch [6/200], CGAN FID: 445.83	Epoch [105/200], CGAN FID: 110.93	Epoch [195/200], CGAN FID: 86.67
Epoch [7/200], CGAN FID: 369.66	Epoch [106/200], CGAN FID: 116.88	Epoch [196/200], CGAN FID: 83.30
Epoch [8/200], CGAN FID: 476.05	Epoch [107/200], CGAN FID: 123.59	Epoch [197/200], CGAN FID: 79.97
Epoch [9/200], CGAN FID: 356.20	Epoch [108/200], CGAN FID: 115.66	Epoch [198/200], CGAN FID: 82.08
Epoch [10/200], CGAN FID: 324.40	Epoch [109/200], CGAN FID: 109.21	Epoch [199/200], CGAN FID: 78.90
	Epoch [110/200], CGAN FID: 126.33	Epoch [200/200], CGAN FID: 79.92
		CGAN 최종 FID: 79.92

마지막으로 CGAN의 FID 결과이다. 605.47 → 79.92로 총 525.55만큼의 차이가 나는 결과값이 나왔다.

최종 결과값을 놓고 봤을 땐 DCGAN 모델의 값이 제일 낮게 측정되었고 첫 에포크와 마지막 에포크 간의 차이값을 봤을 땐 CGAN 모델의 값의 차이가 가장 컸다.

이번 칼럼에서는 GAN, DCGAN, LSGAN, CGAN 총 네 가지 모델에 대해 알아보고 MNIST 데이터셋을 기반으로 이미지를 생성해보는 실습을 진행했고 각 모델의 성능을 FID 값으로 측정해보았다. 실습을 통해 GAN의 성능은 데이터셋에 따라 달라진다는 것을 알게 되어 생성하고자 하는 이미지의 특성을 고려해 적합한 데이터셋과 GAN 모델을 선정하는 것이 중요하다는 것을 느꼈다. 그리고 학습률, 배치 사이즈, 노이즈 크기, 에포크 수 등의 파라미터들을 조정해서 각 GAN 모델에 맞는 값을 찾아보는 계기가 되었다. 앞으로도 많은 사람이 원하는 이미지를 생성할 수 있도록 GAN의 발전과 응용에 관한 지속적인 연구가 필요할 것이다. 이를 통해 더 나은 이미지 생성 기술이 개발되고 다양한 분야에서 사용될 수 있기를 바란다.

[참고자료]

<https://github.com/godeastone/GAN-torch> 기본 GAN, CGAN 코드 참고

<https://arxiv.org/abs/1511.06434> DCGAN 관련 논문

<https://arxiv.org/abs/1611.04076> LSGAN 관련 논문

<https://arxiv.org/abs/1411.1784> CGAN 관련 논문

<https://jarikki.tistory.com/26>

<https://kkwong-guin.tistory.com/151>

<https://untitledtblog.tistory.com/158>

<https://www.kaggle.com/code/vimalpillai/deep-convolutional-gans-or-dcgan-with-mnist>

DCGAN 코드 참고

<https://www.kaggle.com/code/alperkaraca1/mnist-least-squares-gan> LSGAN 코드 참고