DCGAN

최석재 *lingua@naver.com*

DCGAN

Deep Convolutional Generative Adversarial Network

GAN 이미지

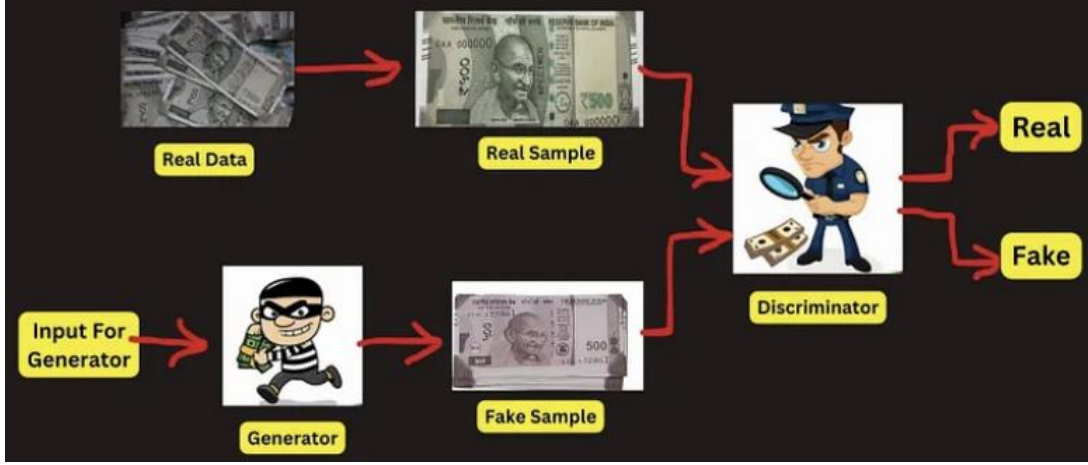
- Ian Goodfellow(2014)가 설계한 가상의 이미지를 생성하는 기법

- GAN을 이용한 데모 이미지
- https://www.thispersondoesnotexist.com/
- https://this-person-does-not-exist.com/en

GAN *Generative Adversarial Networks*

- 생성적 적대 신경망 의미의 GAN은 딥러닝 원리로 가상의 이미지를 생성하는 알고리즘
- 얼굴의 경우, 이미지 픽셀들이 어떻게 조합되어야 우리가 생각하는 '얼굴'의 형상이 되는지를 딥러닝 알고리즘이 예측하여 생성
- 'Adversarial(적대적, 대립적)'이라는 의미는 알고리즘 내에서 적대적인 경합을 벌이기 때문
- GAN의 아이디어를 처음 제안한 Ian Goodfellow는 이를 위조지폐범과 경찰로 비유
- 위조지폐를 만들기 위해 애쓰는 위조지폐범과 이를 가려내기 위해 노력하는 경찰 사이의 경합은 결국 더 정교한 위조지폐를 만들어내게 됨
- 한쪽은 가짜를 만들고, 한쪽은 진짜와 비교하는 경합의 과정을 이용
- 가짜를 만들어 내는 쪽을 생성자(Generator), 진위를 가려내는 쪽을 판별자(Discriminator)라고 부름

위조지폐 시나리오



- GAN 모델을 만드는 주체는 경찰 간부이지만, 사실은 위조지폐 범죄 조직의 두목

1. 최초에 초보 위조지폐 조직원들에게 위조지폐를 만들게 한다
2. 엉터리 위조지폐가 만들어진다
3. 초보 경찰들에게 진짜 지폐를 진짜 지폐라고 알려주고, 위조지폐를 위조지폐라고 알려준다
4. 경찰들이 알려진 내용에 따라 진짜와 위조지폐를 구분하는데, 이때 위조 지폐 중 진짜로 여기거나, 진짜로 여길 확률이 높은 것들을 선별해 그쪽으로 위조지폐 조직원들을 훈련한다
5. 위조지폐 조직원들이 학습에 따라 새로운 위조지폐를 만들어내고, 이것들을 다시 경찰들에게 보여주면서 이것들이 위조지폐인지 여부를 판단하라고 지시한다. 이들이 판단한 결과를 조직원들에게 계속 보여주면서 더 좋은 위조지폐를 만들게 한다
6. 어느 정도 기간이 지나면(새로운 epoch 시기가 되면) 이번에는 경찰들에게 조직원들이 만들어낸 새로운 위조지폐를 보여주면서 이들도 위조지폐임을 새롭게 교육한다. 경찰의 실력이 향상된다. 그리고 경찰의 분류 결과는 다시 조직원에게 보내져 경찰들이 진짜 지폐로 착각할 수 있는 방향으로 조직원들이 다시 교육된다
7. 학습 과정을 통해 경찰들은 진짜와 가짜를 구분할 수 있는 능력을 키우게 되고, 계속 구분 작업을 한다. 구분된 내용 중 진짜 지폐로 볼 확률이 높은 것들이 위조지폐 조직원들에게 계속 알려지므로 조직원들은 더 수준 높은 위조지폐를 만들어낸다
8. 이 과정을 통해 위조지폐 조직원들은 진짜같은 지폐를 점점 잘 만들어낼 수 있게 된다

DCGAN *Deep Convolutional GAN*

- 페이스북의 AI 연구팀에서 발표한 DCGAN은 CNN을 GAN에 적용한 알고리즘으로, 초기의 불안정한 GAN을 크게 보완
- 생성자(Generator)는 처음엔 랜덤한 픽셀 값으로 채워진 가짜 이미지로 시작해서 판별자의 판별 결과에 따라 지속적으로 업데이트하며 점차 원하는 이미지를 만들어 감
- DCGAN은 CNN을 사용하지만, Pooling 과정이 없고, Padding 과정은 포함한다
  - 새로 만들어진 이미지는 비교할 '진짜' 이미지와 크기가 같아야 하기 때문
  - 풀링 과정은 지속적으로 크기를 줄이므로 사용하지 않음
  - 그래도 필터(Mask)를 거치면 다시 크기가 작아지므로 먼저 패딩을 씌워서 필터를 거친 이후에도 원본과 크기가 같아지도록 함

Padding과 Filter 사용

Padding 사용

1	0	0
0	1	1
1	0	1

3 x 3

0	0	0	0	0
0	1	0	0	0
0	0	1	1	0
0	1	0	1	0
0	0	0	0	0

5 x 5

Filter 사용

0	0	0	0	0
0	1	0	0	0
0	0	1	1	0
0	1	0	1	0
0	0	0	0	0

X

1	1	1
0	0	1
0	0	1

=

1	1	0
2	3	1
1	3	2

3 x 3

관련 패키지 로드

- from tensorflow.keras.datasets import mnist
- from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Dropout
- from tensorflow.keras.layers import BatchNormalization, Activation, LeakyReLU, UpSampling2D, Conv2D
- from tensorflow.keras.models import Sequential, Model
- import os
- import numpy as np
- import matplotlib.pyplot as plt

생성된 이미지 저장 경로 생성

- save_path = '/content/gdrive/MyDrive/pytest_img/_generated_images'

_generated_images 폴더 밑에 MNIST 라는 폴더를 만든다

- if not os.path.exists(os.path.join(save_path, 'MNIST/')):
  os.makedirs(os.path.join(save_path, "MNIST/"))

생성자 만들기 (1/3)

생성자는 이미지를 만드는 것이 목적
랜덤한 픽셀 값으로 채워진 가짜 이미지 생성으로부터 시작

- generator = Sequential()

은닉 1층을 128*7*7 개로 만듦
128은 노드의 크기를 위한 차원의 수로서 변경 가능. 데이터가 다양한 정보를 갖게 하기 위하여 충분히 크게 함
7*7은 이미지의 최종 크기를 고려한 수. 뒤에서 이미지를 2배로 키우는 UpSampling을 2번 거치므로 7을 사용하면 7*2*2=28이 되어 original 이미지의 28과 같게 된다
7*7의 수를 변경해도 괜찮지만, Reshape의 결과와 연결되므로 가능한 왜곡이 없게 하려면 은닉층의 수를 이에 맞추는 것이 좋다
input_shape(100,)은 입력층의 shape. 가상의 이미지를 이 크기로 만든다. 일반적으로 사용하는 값이나, 변경 가능

- generator.add(Dense(128*7*7, input_shape=(100,), activation=LeakyReLU(0.2)))

배치 정규화 레이어. 입력 데이터의 평균이 0, 분산이 1이 되도록 값을 일정하게 재배치
배치 정규화는 많은 경우 중요한 단계 중 하나로 인식되는데, DCGAN 개발자들은 이 레이어를 통해 안정적인 학습이 가능하다며 특히 중요하게 생각함

- generator.add(BatchNormalization())

생성자 만들기 (2/3)

Conv2D()에 들어갈 수 있도록 Reshape 레이어를 사용하여 데이터의 shape를 변경(1D → 3D)

- generator.add(Reshape((7, 7, 128)))

UpSampling2D()는 height와 width의 크기를 각각 2배로 늘림
(batch_size, height, width, channel) → (batch_size, height*2, width*2, channel)
(batch_size, 7, 7, 128) → (batch_size, 7*2, 7*2, 128)

- generator.add(UpSampling2D())

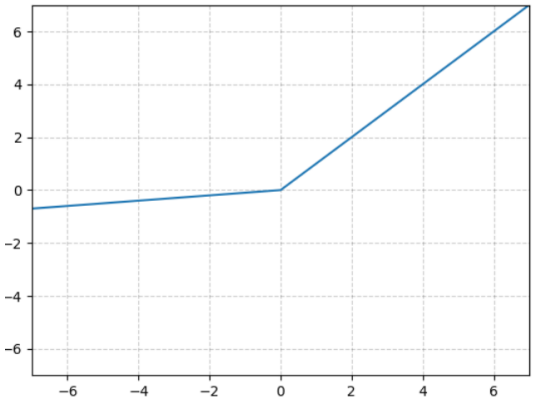
Convolution 레이어. padding='same', 필터 크기 (5, 5)인 64개의 필터를 통과하면서. shape는 (14, 14, 64)

- generator.add(Conv2D(64, kernel_size=5, padding='same'))

배치 정규화 레이어

- generator.add(BatchNormalization())

생성자 만들기 (3/3)



LeakyReLU()는 음수에서 값이 무조건 0이 되는 것을 방지하는 것으로, 0 이하에서도 작은 값을 갖게 함
여기에서는 0보다 작을 경우에는 0.2를 곱하게 한다
GAN에서는 기존에 사용하던 ReLU() 함수를 사용하면 학습이 불안정해지는 경우가 많다

- generator.add(Activation(LeakyReLU(0.2)))

(batch_size, 14, 14, 64) → (batch_size, 14*2, 14*2, 64)

- generator.add(UpSampling2D())

Convolution 레이어. 필터 크기 (5, 5). activation 함수로 tanh를 사용하여 출력값을 -1 ~ 1 사이가 되게 함
훈련 시 사용되는 입력 데이터는 -1 ~ 1 사이의 값으로 정규화하여 입력되므로, 훈련 중엔 계속 이 값이 유지되게 한다
1필터를 갖는 Conv2D 이므로 (28, 28, 1) 이 된다

- generator.add(Conv2D(1, kernel_size=5, padding='same', activation='tanh'))

판별자 만들기 (1/2)

판별자는 이미지의 진위 여부를 가리는 것이 목적

- discriminator = Sequential()

크기 (5, 5)의 필터 64개 사용. strides=2(마스크를 2칸씩 이동)
기본값은 1칸인데, 2칸을 움직이게 하면 height, width 크기가 더 줄어들면서 새로운 특징을 뽑아주는 효과. 즉, 새로운 필터를 적용한 효과가 생김
생성자에서 최종 출력된 크기가 28*28 이기 때문에 input_shape를 28*28로 함
색상이 아닌, 형상만 파악하면 진위 여부를 구별할 수 있으므로 채널은 1채널만 사용

- discriminator.add(Conv2D(64, kernel_size=5, strides=2, input_shape=(28, 28, 1), padding='same'))

- discriminator.add(Activation(LeakyReLU(0.2)))
- discriminator.add(Dropout(0.3))
- discriminator.add(Conv2D(128, kernel_size=5, strides=2, padding='same'))
- discriminator.add(Activation(LeakyReLU(0.2)))
- discriminator.add(Dropout(0.3))

판별자 만들기 (2/2)

판별, 즉 분류를 하기 위하여 1차원 데이터로 변환하는 Flatten() 수행

- discriminator.add(Flatten())

진짜(1), 가짜(0) 중 하나가 되어야 하므로 출력노드를 1로 만들고, Sigmoid 함수를 사용

- discriminator.add(Dense(1, activation='sigmoid'))

discriminator를 컴파일하여 독립적인 모델로 사용할 수 있게 한다.

- discriminator.compile(loss='binary_crossentropy', optimizer='adam')

판별자의 학습 가능 여부를 False로 설정
이렇게 하면 생성자와 판별자를 함께 학습하는 GAN 모델을 컴파일 할 때는 판별자의 가중치는 업데이트되지 않고, 생성자만 학습됨
판별자는 가중치 업데이트는 하지 않지만, 여전히 compile시 설정된 손실함수를 사용하여 손실값은 계산함
계산된 손실값은 GAN 모델 전체를 통해 생성자에게 전달됨. 생성자는 전달된 판별자의 가중치를 통해 자신의 가중치를 업데이트 함
만약 판별자가 같이 업데이트 되면 생성자는 판별자를 속일 수 없게 되므로, 생성자가 판별자를 속일 수 있도록 판별자의 가중치 업데이트를 막는 것

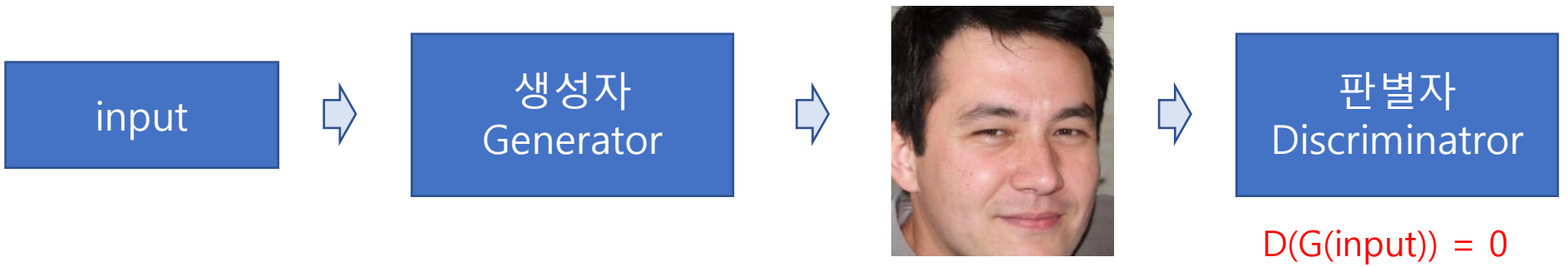
- discriminator.trainable = False

생성자와 판별자 연결

input

x

- 생성자를 G() 라고 할 때, 가상의 입력값 input의 결과는 G(input)
- 이를 판별자 D()에 넣은 결과는 D(G(input))
- 실제(진짜) 데이터 x의 결과는 D(x)
- 즉, G(input)은 가짜 0. x는 진짜 1.
- 처음에는 D(G(input))과 D(x)의 차이가 크지만, 학습이 진행될수록 두 결과는 유사해짐



G()

G(input)

D(G(input)) = 0
D(x) = 1
로 나와야 옳은 판정

생성자와 판별자를 연결하는 GAN 모델

생성자는 100 차원의 크기를 갖는 랜덤 노이즈 벡터를 입력으로 받음
- ginput = Input(shape=(100,))

생성자에서 만들어진 데이터를 판별자에 입력시킴
판별자를 거친 결과(손실함수 계산값)가 dis_output에 저장됨
- dis_output = discriminator(generator(ginput))

inputs에는 랜덤으로 생성된 데이터가 생성자의 결과를 거친 ginput을, outputs에는 판별자를 거친 dis_output을 입력
dis_output은 discriminator(generator(ginput)) 과정을 거친 것이므로,
Functional API의 Model 클래스는 생성자와 판별자를 연결하여 하나의 새로운 모델을 만든다
즉, 생성자로 생성된 이미지를 판별자로 판별하는 모델
- gan = Model(inputs=ginput, outputs=dis_output)

참과 거짓을 구분해야 하므로 손실함수는 binary_crossentropy
discriminator.trainable = False 이후 전체 모델인 gan 모델을 컴파일 하는 것이므로, 이것을 사용할 때는 학습 기능이 꺼진 discriminator가 사용된다
- gan.compile(loss='binary_crossentropy', optimizer='adam')

전체 모델 요약

- gan.summary()

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 100)]             0
 sequential (Sequential)     (None, 28, 28, 1)         865281
 sequential_1 (Sequential)   (None, 1)                 212865
=================================================================
Total params: 1,078,146
Trainable params: 852,609
Non-trainable params: 225,537
_________________________________________________________________
```

inputs

Generator

Discriminator

생성자와 판별자 요약

- generator.summary()

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 6272)	633472
batch_normalization (BatchNormalization)	(None, 6272)	25088
reshape (Reshape)	(None, 7, 7, 128)	0
up_sampling2d (UpSampling2D)	(None, 14, 14, 128)	0
conv2d (Conv2D)	(None, 14, 14, 64)	204864
batch_normalization_1 (BatchNormalization)	(None, 14, 14, 64)	256
activation (Activation)	(None, 14, 14, 64)	0
up_sampling2d_1 (UpSampling2D)	(None, 28, 28, 64)	0
conv2d_1 (Conv2D)	(None, 28, 28, 1)	1601

Total params: 865281 (3.30 MB)
Trainable params: 852609 (3.25 MB)
Non-trainable params: 12672 (49.50 KB)

- discriminator.summary()

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 14, 14, 64)	1664
activation_1 (Activation)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_3 (Conv2D)	(None, 7, 7, 128)	204928
activation_2 (Activation)	(None, 7, 7, 128)	0
dropout_1 (Dropout)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dense_1 (Dense)	(None, 1)	6273

Total params: 212865 (831.50 KB)
Trainable params: 0 (0.00 Byte)
Non-trainable params: 212865 (831.50 KB)

변수 생성

- 판별자와 생성자의 오차를 받을 변수를 생성한다

- d_loss = [] # 판별자의 오차 변수
- g_loss = [] # 생성자의 오차 변수

훈련 함수 (1/5)

- GAN 모델을 훈련시키는 과정

훈련할 epoch, batch_size, 저장 간격

- def gan_train(epoch, batch_size, saving_interval):

MNIST 훈련 데이터 불러오기. 훈련데이터의 독립변수만 사용한다

(X_train, _), (_, _) = mnist.load_data()

(X_train의 행의 크기, 28, 28, channel==1)

X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')

정규화 과정. 중앙값 127.5를 빼준 뒤 127.5로 나눠서 -1~1 사이의 값으로 바꿈
GAN에서는 -1~1의 값을 갖는 tanh를 사용하므로 이 범위를 갖도록 정규화하는 것

X_train = (X_train-127.5) / 127.5

true = np.ones((batch_size, 1)) # 배치 사이즈만큼의 행을 갖는 True(1) 레이블 생성

fake = np.zeros((batch_size, 1)) # 배치 사이즈만큼의 행을 갖는 False(zero) 레이블 생성

①

```
def gan_train(epoch, batch_size, saving_interval):

    # MNIST 훈련 데이터 불러오기
    (X_train, _), (_, _) = mnist.load_data()
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')

    # 정규화
    X_train = (X_train-127.5) / 127.5
    true = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))
```

②

```
    for i in range(epoch):
        # 실제 데이터를 판별자에 입력
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]
        d_loss_real = discriminator.train_on_batch(imgs, true)
```

③

```
        # 가상 이미지를 판별자에 입력
        noise = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(noise)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
```

④

```
        # 판별자와 생성자의 오차 계산
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        g_loss = gan.train_on_batch(noise, true)
```

⑤

```
        if i % saving_interval == 0:
            noise = np.random.normal(0, 1, (25, 100))
            gen_imgs = generator.predict(noise)

            # Rescale images 0 ~ 1
            gen_imgs = 0.5 * gen_imgs + 0.5

            fig, axs = plt.subplots(5, 5)
            count = 0
            for j in range(5):
                for k in range(5):
                    axs[j, k].imshow(gen_imgs[count, :, :, 0], cmap='gray')
                    axs[j, k].axis('off')
                    count += 1
                    fig.savefig("/content/gdrive/MyDrive/pytest_img/gan_imag
```

훈련 함수 (2/5)

```
for i in range(epoch):          # 에포크 단위 학습
    # 실제 이미지를 판별자에 입력하여 학습
    # np.random.randint(a, b, c)는 a부터 b까지의 숫자 중 하나를 랜덤으로 뽑는 과정을 c 번 반복
    # X_train.shape는 (60000, 28, 28)이므로 0부터 60,000개의 훈련데이터 중 하나 가져오기를 batch_size 만큼 반복
    idx = np.random.randint(0, X_train.shape[0], batch_size)

    # 위에서 선택된 숫자 idx를 이용하여 이미지들을 불러 옴
    imgs = X_train[idx]

    # train_on_batch(x, y) 함수는 입력값(x)과 레이블(y)을 받아서 바로 학습을 실시해 모델을 업데이트
    # 즉 batch_size 크기의 imgs 데이터를 받아 바로 학습하고 모델 업데이트. discriminator.compile() 단계의 판별자를 사용한다(훈련가능)
    # return 값은 training loss
    # imgs = X_train[idx] 에서 뽑힌 이미지를 x(진짜)에 넣고,
    # true = np.ones((batch_size, 1)) 에서 만든 배열을 y에 넣어 그것들이 진짜(1)임을 알려줌
    d_loss_real = discriminator.train_on_batch(imgs, true)
```

①
```
def gan_train(epoch, batch_size, saving_interval):

    # MNIST 훈련 데이터 불러오기
    (X_train, _), (_, _) = mnist.load_data()
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')

    # 정규화
    X_train = (X_train-127.5) / 127.5
    true = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))
```

②
```
    for i in range(epoch):
        # 실제 데이터를 판별자에 입력
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]
        d_loss_real = discriminator.train_on_batch(imgs, true)
```

③
```
        # 가상 이미지를 판별자에 입력
        noise = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(noise)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
```

④
```
        # 판별자와 생성자의 오차 계산
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        g_loss = gan.train_on_batch(noise, true)
```

⑤
```
        if i % saving_interval == 0:
            noise = np.random.normal(0, 1, (25, 100))
            gen_imgs = generator.predict(noise)

            # Rescale images 0 ~ 1
            gen_imgs = 0.5 * gen_imgs + 0.5

            fig, axs = plt.subplots(5, 5)
            count = 0
            for j in range(5):
                for k in range(5):
                    axs[j, k].imshow(gen_imgs[count, :, :, 0], cmap='gray')
                    axs[j, k].axis('off')
                    count += 1
                    fig.savefig("/content/gdrive/MyDrive/pytest_img/gan_imag
```

훈련 함수 (3/5)

```
① def gan_train(epoch, batch_size, saving_interval):

    # MNIST 훈련 데이터 불러오기
    (X_train, _), (_, _) = mnist.load_data()
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')

    # 정규화
    X_train = (X_train-127.5) / 127.5
    true = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))

② for i in range(epoch):
        # 실제 데이터를 판별자에 입력
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]
        d_loss_real = discriminator.train_on_batch(imgs, true)

③       # 가상 이미지를 판별자에 입력
        noise = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(noise)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)

④       # 판별자와 생성자의 오차 계산
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        g_loss = gan.train_on_batch(noise, true)

⑤       if i % saving_interval == 0:
            noise = np.random.normal(0, 1, (25, 100))
            gen_imgs = generator.predict(noise)

            # Rescale images 0 ~ 1
            gen_imgs = 0.5 * gen_imgs + 0.5

            fig, axs = plt.subplots(5, 5)
            count = 0
            for j in range(5):
                for k in range(5):
                    axs[j, k].imshow(gen_imgs[count, :, :, 0], cmap='gray')
                    axs[j, k].axis('off')
                    count += 1
                    fig.savefig("/content/gdrive/MyDrive/pytest_img/gan_imag
```

가상 이미지를 판별자에 입력하여 학습. 0부터 1까지의 실수 중 2D (batch_size, 100)개를 랜덤으로 뽑음
(batch_size, 100)은 (batch_size 행, 100 열)의 구조로 랜덤값을 뽑음
100 열인 이유는 생성자에서 100 차원 입력을 요구하였기 때문

```
noise = np.random.normal(0, 1, (batch_size, 100))
```

(batch_size, 100) 구조의 랜덤값(가상 이미지)이 Generator에 들어가고, 결괏값이 gen_imgs로 저장됨
gen_imgs는 랜덤하게 만든 noise 가상 이미지를 generator에 넣어 생성된 이미지(tanh의 결과이므로 -1 ~ 1)
predict() 함수를 사용하였지만, generator의 마지막 출력층이 Conv2D 로서 진짜/가짜 이미지일 확률이 아니라,
generator 모델을 통해 생성된 이미지임
noise의 shape (batch, 100) → gen_imgs의 shape (batch, 28, 28, 1) 로 변환됨

```
gen_imgs = generator.predict(noise)
```

train_on_batch()로 학습 및 모델 업데이트. discriminator.compile()의 판별자를 사용한다(훈련가능)
판별자에게 gen_imgs가 모두 가짜(0) 임을 알려주어 훈련시킴
생성자는 지속적으로 가중치가 업데이트되어 이전 epoch에서 만들어내는 이미지가 점점 진짜와 같아지므로 판별자의 성능도 올라감
판별자는 출력층이 Dense() 층 이진분류로서 판정을 하는 데 사용됨

```
d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
```

※ Model의 가중치 업데이트는 batch_size마다 올라가고,
새로운 가상 이미지는 epoch마다 만들어낸다

훈련 함수 (4/5)

① ② ③ ④ ⑤

```
def gan_train(epoch, batch_size, saving_interval):

    # MNIST 훈련 데이터 불러오기
    (X_train, _), (_, _) = mnist.load_data()
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')

    # 정규화
    X_train = (X_train-127.5) / 127.5
    true = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))

    for i in range(epoch):
        # 실제 데이터를 판별자에 입력
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]
        d_loss_real = discriminator.train_on_batch(imgs, true)

        # 가상 이미지를 판별자에 입력
        noise = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(noise)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)

        # 판별자와 생성자의 오차 계산
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        g_loss = gan.train_on_batch(noise, true)

        if i % saving_interval == 0:
            noise = np.random.normal(0, 1, (25, 100))
            gen_imgs = generator.predict(noise)

            # Rescale images 0 ~ 1
            gen_imgs = 0.5 * gen_imgs + 0.5

            fig, axs = plt.subplots(5, 5)
            count = 0
            for j in range(5):
                for k in range(5):
                    axs[j, k].imshow(gen_imgs[count, :, :, 0], cmap='gray')
                    axs[j, k].axis('off')
                    count += 1
                    fig.savefig("/content/gdrive/MyDrive/pytest_img/gan_imag
```

판별자와 생성자의 오차 계산. d_loss는 판별자의 성능 (손실값)
실제 이미지를 넣은 결과에 대한 오차값이 d_loss_real, 가상 이미지를 넣은 결과에 대한 오차값이 d_loss_fake
둘은 모두 판별자에 대한 성능이므로 이 둘의 평균 오차값을 구하여 판별자의 오차(성능)를 구함
d_loss_real과 d_loss_fake를 더한 뒤 1/2

```
d_loss.append(0.5 * np.add(d_loss_real, d_loss_fake))
```

판별자 가중치를 생성자로 전달하는 과정
train_on_batch()로 학습 및 모델 업데이트. 앞에서 생성한 GAN 모델 사용
g_loss는 생성자의 성능 (손실값)
gan.train_on_batch(noise, true)는 판별자에게 가상 이미지 noise에 대하여 true(1)라고 거짓으로 알려주고 판별자로 넘김
gan 모델 내부에서는 noise → generator에서 28*28*1 의 이미지로 변환 → 판별자로 판별 과정을 거침
판별자는 생성자가 만든 가짜를 진짜(true(1))로 인식하려고 하지만, 초기의 가짜는 진짜와 거리가 멀어 모델의 손실값이 커지게 됨
이 손실을 최소화화기 위해 생성자는 가짜 이미지를 더욱 진짜 이미지처럼 만들게 하는 쪽으로 Conv2D의 필터 가중치가 업데이트 되고,
판별자는 가짜 이미지를 진짜로 판정하는 쪽으로 가중치 업데이트를 해야 하나, trainable = False 하였으므로 판별자는 업데이트 되지 않음
즉, 전체 Model은 판별 결과(확률값)가 여기서 보낸 true(1)과 얼마나 차이나는지를 파악하여 그것이 보정되도록 Model을 업데이트 하는 것

```
g_loss.append(gan.train_on_batch(noise, true))
```

정리: 생성자와 판별자의 성능 향상 방법

- 생성자가 성능을 올리는 방식
  - 현재 epoch에서 batch_size마다 Model의 가중치를 업데이트
  - Model에는 생성자와 판별자 두 모델이 있으며, 여기서 생성자의 가중치만 업데이트
  - train_on_batch()를 이용한 배치 단위 학습이므로 생성자는 해당 epoch 내에서 성능이 계속 향상됨

  - gan.train_on_batch(noise, true) # 가짜를 진짜로 알려줌

- 판별자가 성능을 올리는 방식
  - 이전 epoch 까지의 가중치 업데이트로 성능이 향상된 생성자가 만들어낸 이미지가
  - 가짜(0) 이미지라고 하는 것을 알고 훈련하므로 판별자의 성능이 점점 좋아짐
  - 아래 두 가지는 epoch 단위 학습 (for i in range(epoch):)

  - d_loss_real = discriminator.train_on_batch(imgs, true) # 진짜를 진짜로 학습
  - d_loss_fake = discriminator.train_on_batch(gen_imgs, fake) # 가짜를 가짜로 학습

훈련 함수 (5/5)

```
# 지금까지 업데이트 된 모델로 이미지 생성
# saving_interval로 지정된 때마다 아래의 과정을 진행하여 만든 이미지를 저장
if i % saving_interval == 0:
    noise = np.random.normal(0, 1, (25, 100))    # (25 행, 100 열)의 가상 이미지 생성
    gen_imgs = generator.predict(noise)          # 생성자로 이미지 변환 (32, 100) → (32, 28, 28, 1)

    # -1 ~ 1 사이의 값을 갖는 이미지를 0 ~ 1 사이의 값으로 변환
    # 만들어진 gen_imgs는 (25, 28, 28, 1)
    gen_imgs = 0.5 * gen_imgs + 0.5    # 생성자 함수 마지막층이 -1~1 범위를 갖는 tanh이므로, 0~1로 변환

    fig, axs = plt.subplots(5, 5)                # 생성된 25행의 가상 이미지를 5*5 로 표현할 준비
    count = 0
    for j in range(5):
        for k in range(5):
            # 25개씩 이미지 출력. (j, k) : (0, 0) → (0, 1) → (0, 2) → ... (4, 4)
            # count는 0 ~ 24까지 올라가 총 25장이 출력되며, :, : 에 의해 각 이미지가 가지고 있는 28 픽셀은 모두 출력됨
            axs[j, k].imshow(gen_imgs[count, :, :, 0], cmap='gray')
            axs[j, k].axis('off')
            count += 1
            fig.savefig(os.path.join(save_path, "MNIST/")+"gan_mnist_%d.png" % i)
```

①
```
def gan_train(epoch, batch_size, saving_interval):

    # MNIST 훈련 데이터 불러오기
    (X_train, _), (_, _) = mnist.load_data()
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')

    # 정규화
    X_train = (X_train-127.5) / 127.5
    true = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))
```

②
```
    for i in range(epoch):
        # 실제 데이터를 판별자에 입력
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]
        d_loss_real = discriminator.train_on_batch(imgs, true)
```

③
```
        # 가상 이미지를 판별자에 입력
        noise = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(noise)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
```

④
```
        # 판별자와 생성자의 오차 계산
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        g_loss = gan.train_on_batch(noise, true)
```

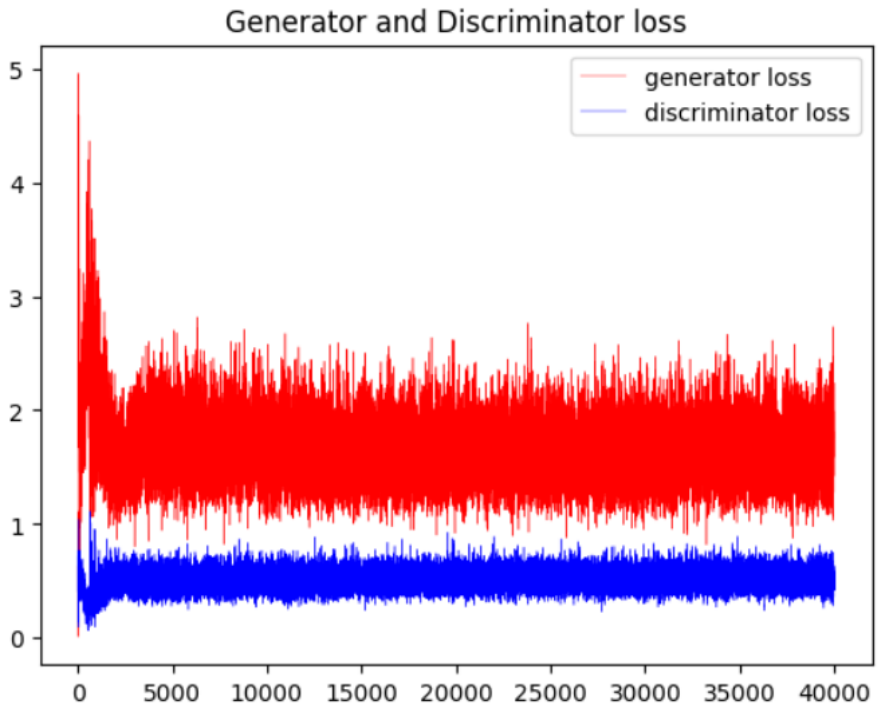
⑤
```
        if i % saving_interval == 0:
            noise = np.random.normal(0, 1, (25, 100))
            gen_imgs = generator.predict(noise)

            # Rescale images 0 ~ 1
            gen_imgs = 0.5 * gen_imgs + 0.5

            fig, axs = plt.subplots(5, 5)
            count = 0
            for j in range(5):
                for k in range(5):
                    axs[j, k].imshow(gen_imgs[count, :, :, 0], cmap='gray')
                    axs[j, k].axis('off')
                    count += 1
                    fig.savefig("/content/gdrive/MyDrive/pytest_img/gan_imag
```

실행

GAN 모델 훈련 실행

- gan_train(40001, 32, 200)

성능 그래프



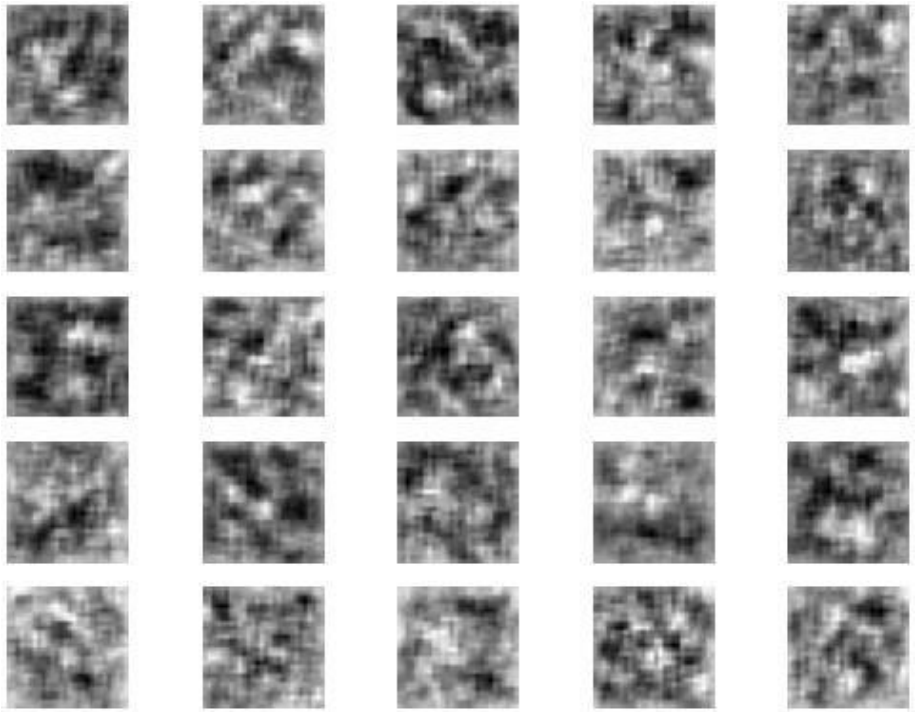
- import matplotlib.pyplot as plt

- epochs = range(0, 40001)

- plt.plot(epochs, g_loss, 'r', label='generator loss', linewidth=0.3)
- plt.plot(epochs, d_loss, 'b', label='discriminator loss', linewidth=0.3)
- plt.title("Generator and Discriminator loss")
- plt.legend()
- plt.show()

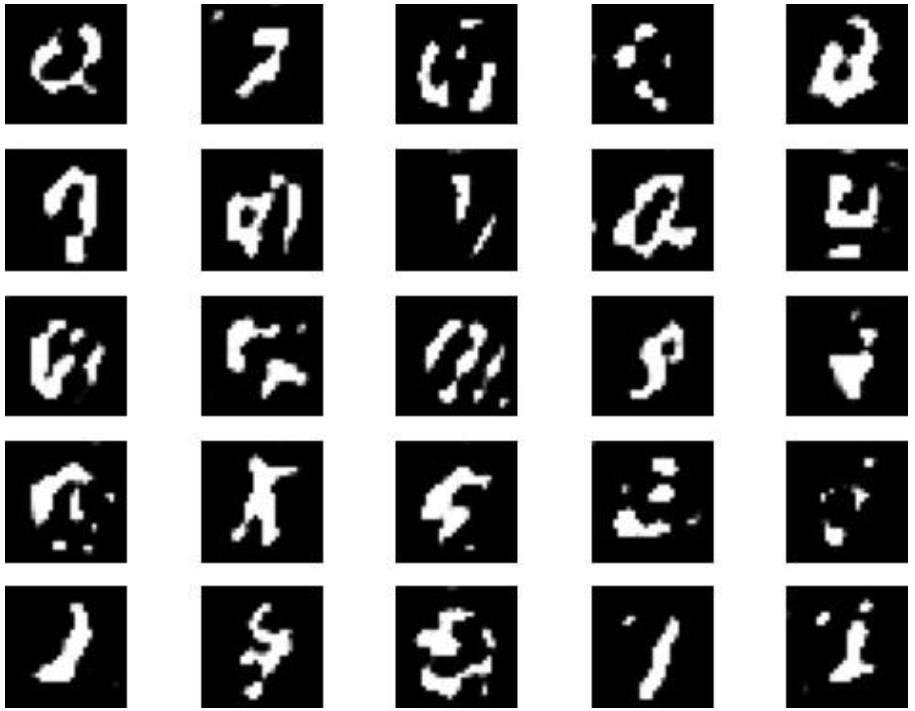
※ 생성자는 초기에 높은 손실값을 보이다가, 3000 회 정도 지나면서 많이 낮아진다

최초 이미지

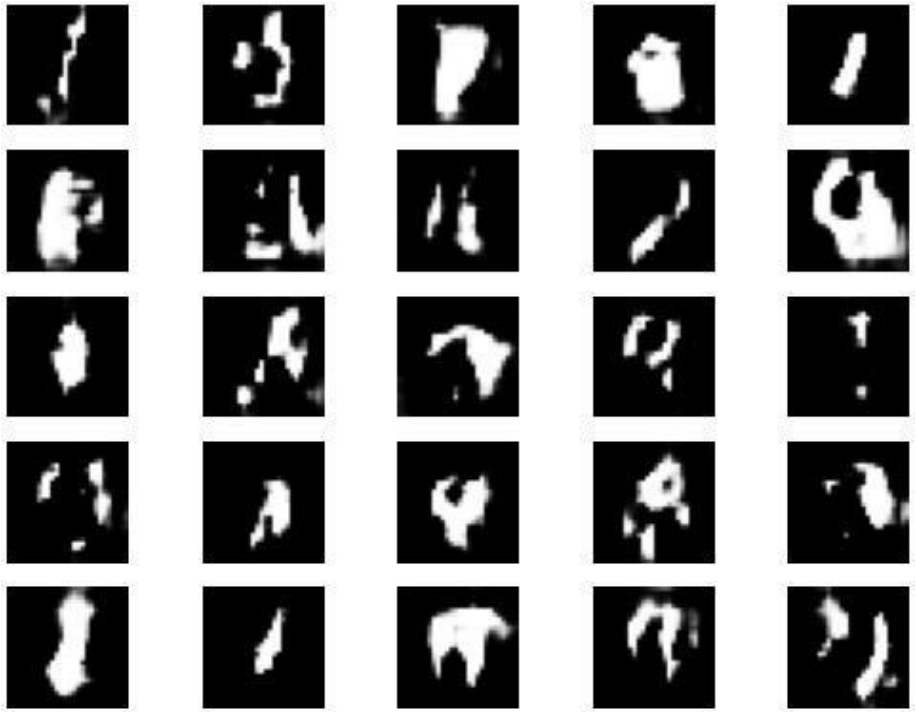


epoch 200 이미지

epoch 400 이미지



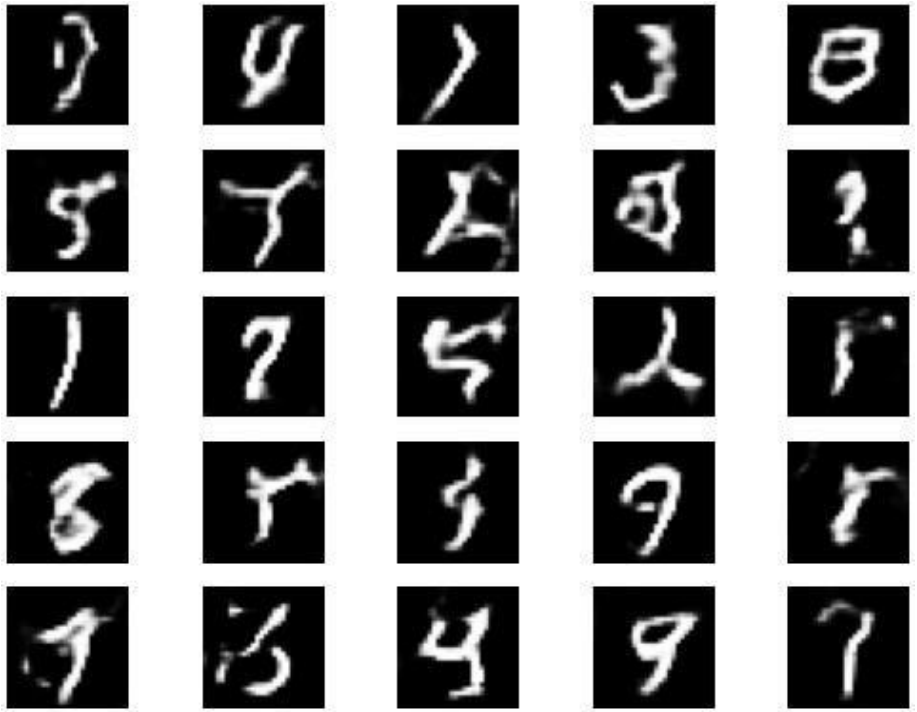
epoch 600 이미지



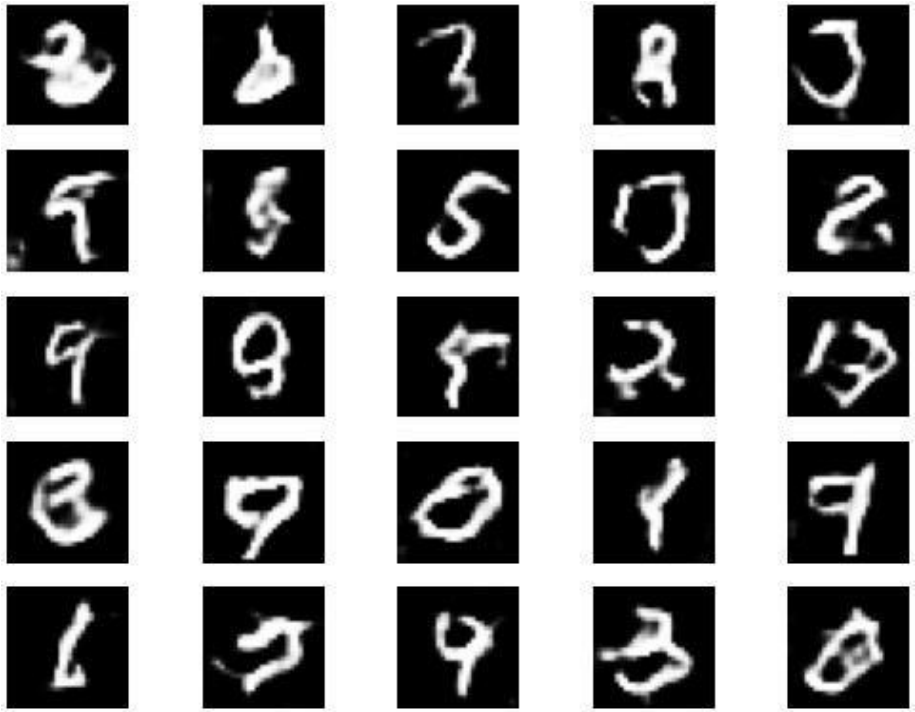
epoch 800 이미지

epoch 1000 이미지

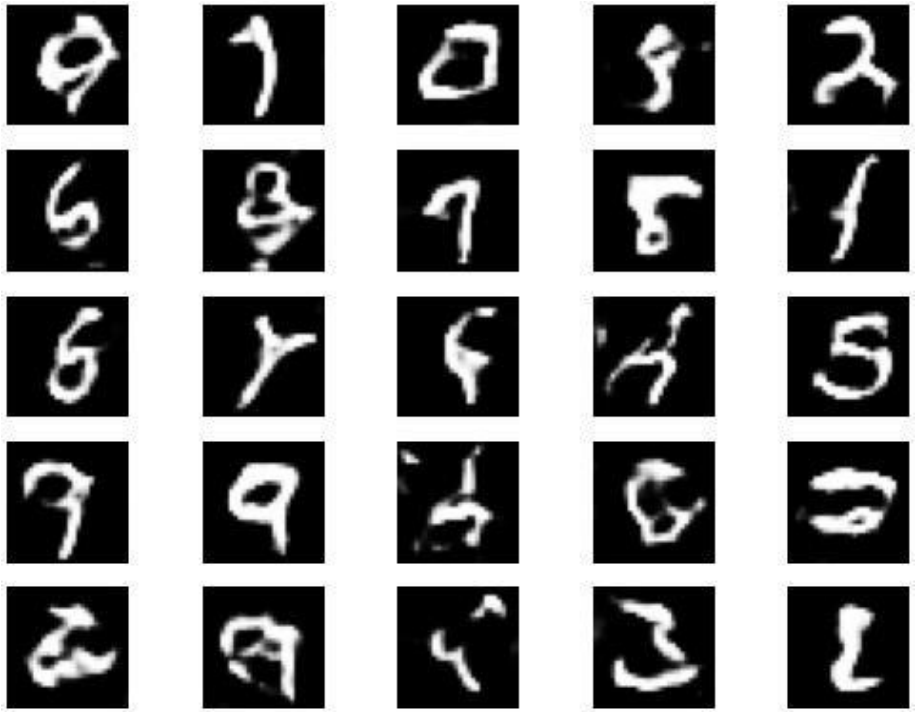
epoch 2000 이미지



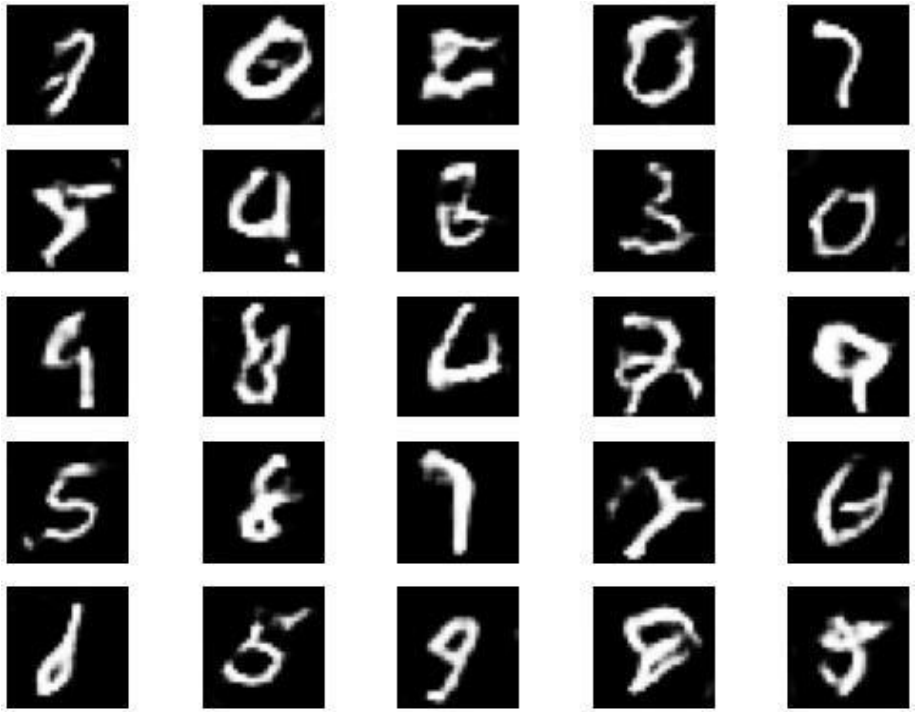
epoch 4000 이미지



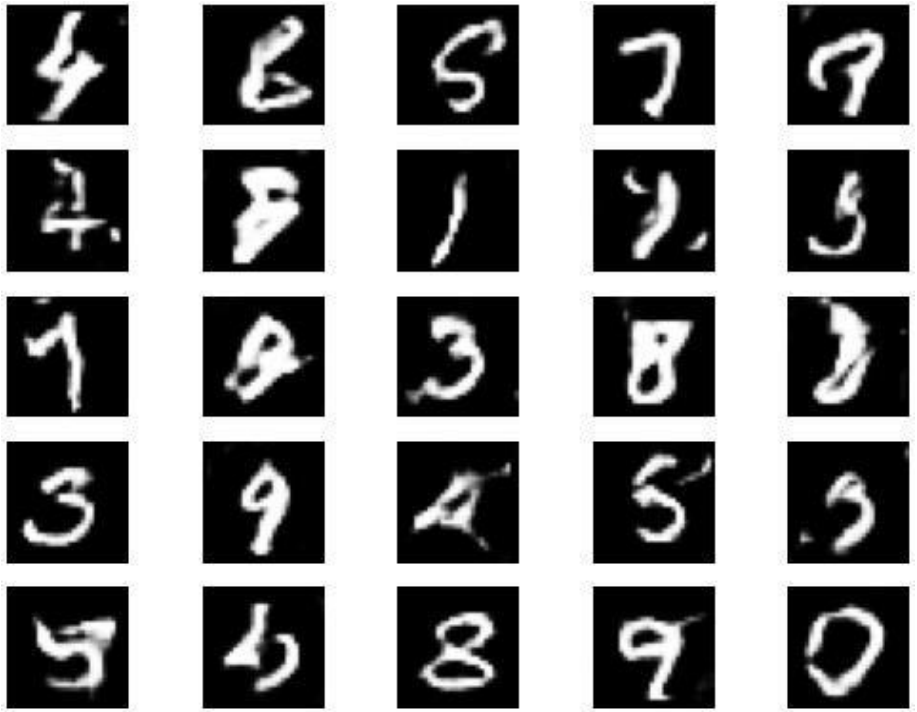
epoch 6000 이미지



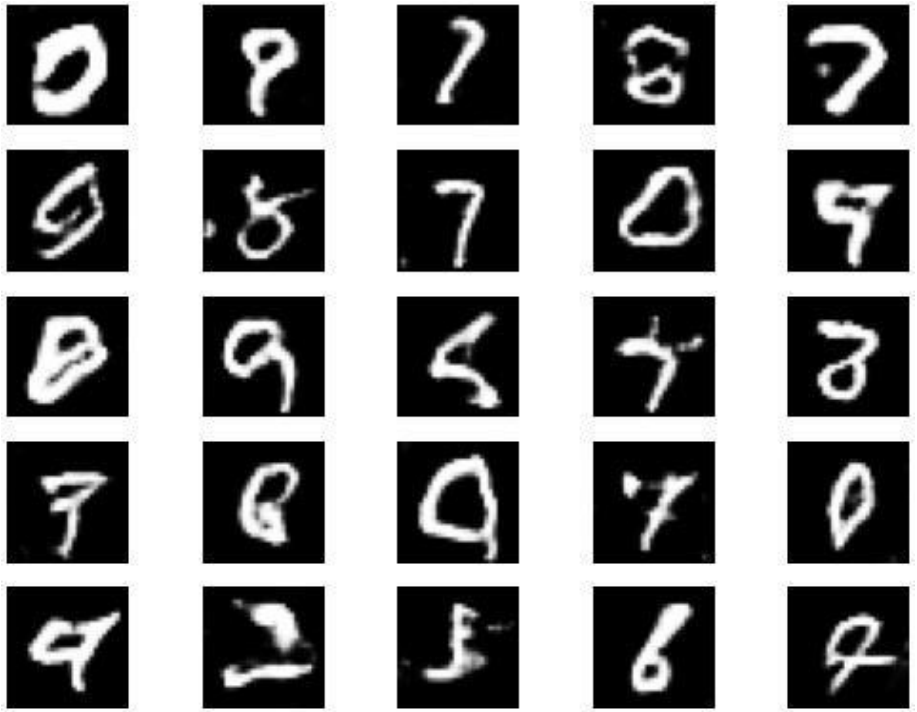
epoch 8000 이미지



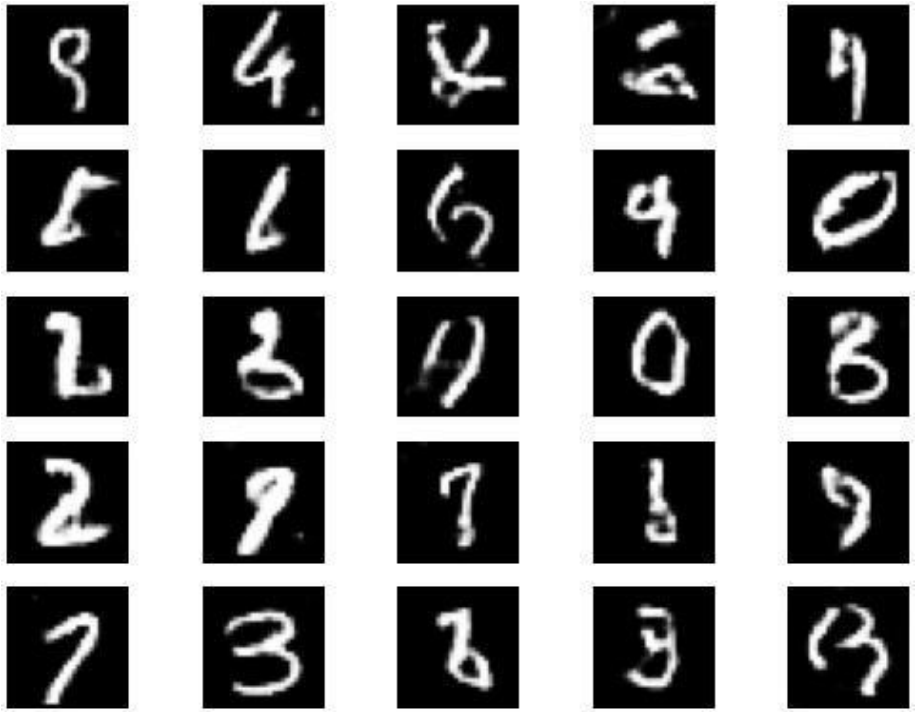
epoch 10000 이미지



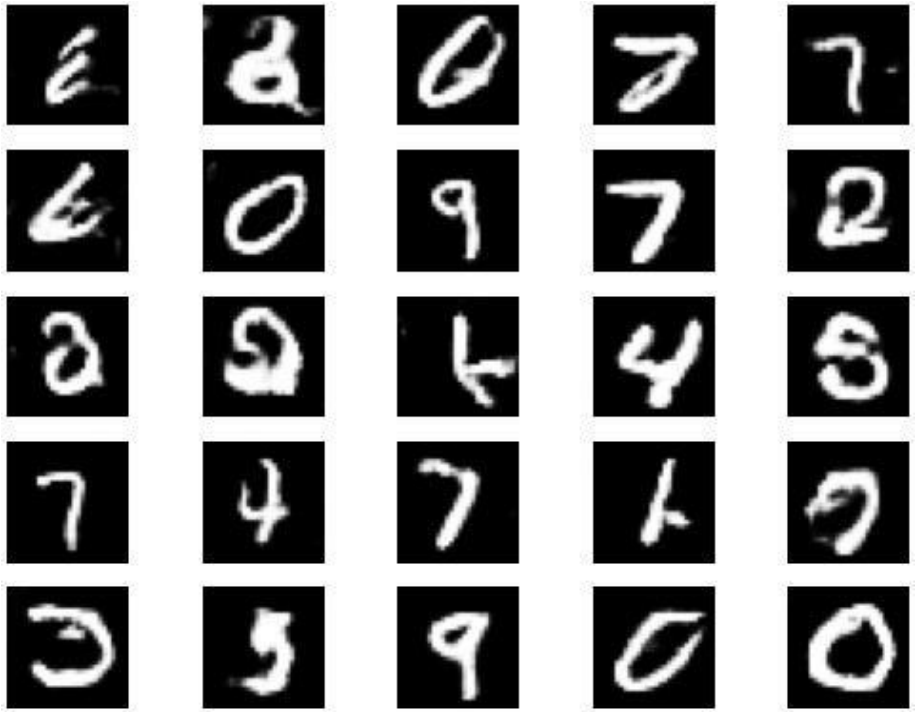
epoch 20000 이미지



epoch 30000 이미지



epoch 40000 이미지



연습문제 1

original

http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

- 사람의 얼굴 사진을 이용하여 가상의 얼굴 이미지를 만드시오
  - pytest_img > img_align_celeba_small 폴더에 있는 100개의 사진을 이용한다
  - Full data는 original_jpg_backup > img_align_celeba 폴더에 약 22만 장
  - 유명인들의 사진 모음으로 다양한 얼굴 특징을 가지고 있음
  - GAN 코드 변경을 최소화하기 위하여 img.resize()를 이용하여 모든 이미지를 28 x 28 로 변경
  - (28, 28, 3) 이미지에서 1개 컬러 채널만 가지고 올 것. 즉, (28, 28, 1)로 입력
  - pytest_img > _generated_images > FACES 폴더를 생성하여 진행
  - epoch는 4,001 회 정도로 설정한다

연습문제 2

original

http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

- 앞과 동일한 과제를 컬러로 출력하시오
  - 연습문제 1의 코드를 Copy 하여 별도의 페이지에서 작성
  - FACES_color 폴더를 생성하여 출력된 이미지를 저장
  - cmap='viridis'