GAN
(generative adversarial network)

2019.04.08

Hanyang univ. AILAB 정지은

What

GAN

- **G = Generative :** ‘그럴듯한 가짜를 생성하는’

그럴듯하다?

= 수학적으로 **실제 데이터의 분포와 비슷한 분포에서 나온 데이터**를 의미

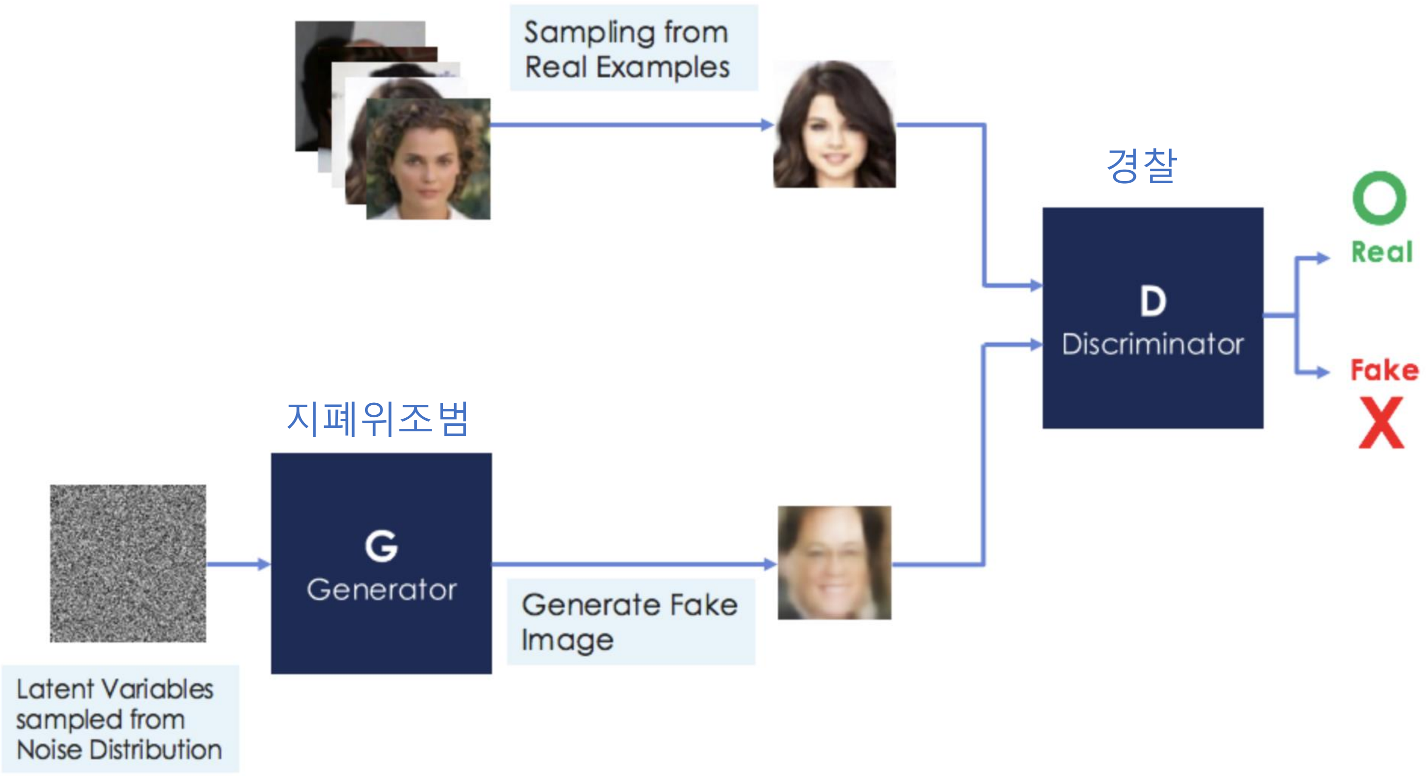
Ex1) 키 172cm, 몸무게 68kg

Ex2) 키 190cm, 몸무게 20kg => 이 조합은 실제 데이터 분포에서는 거의 나오지 않는 조합

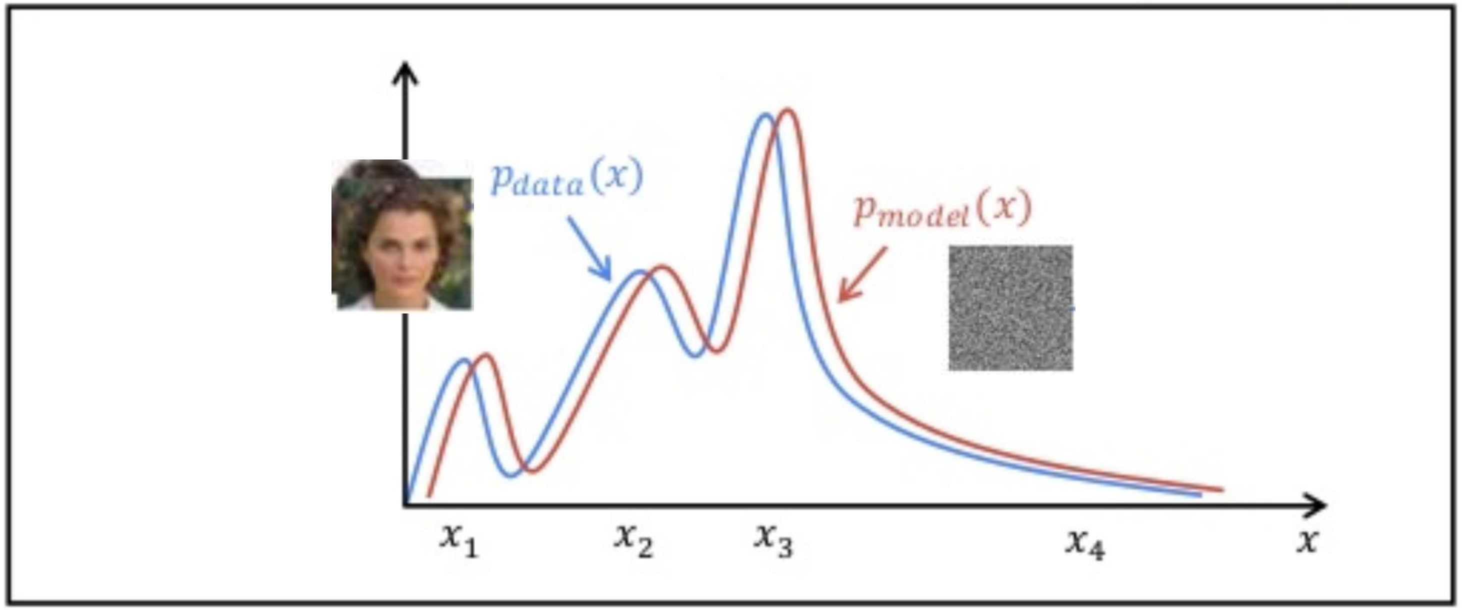
.

GAN

- **A = Adversarial :** 두개의 모델을 **적대적(adversarial)**으로 경쟁시키면서 서로의 성능이 발전
  (Generator vs. Discriminator)

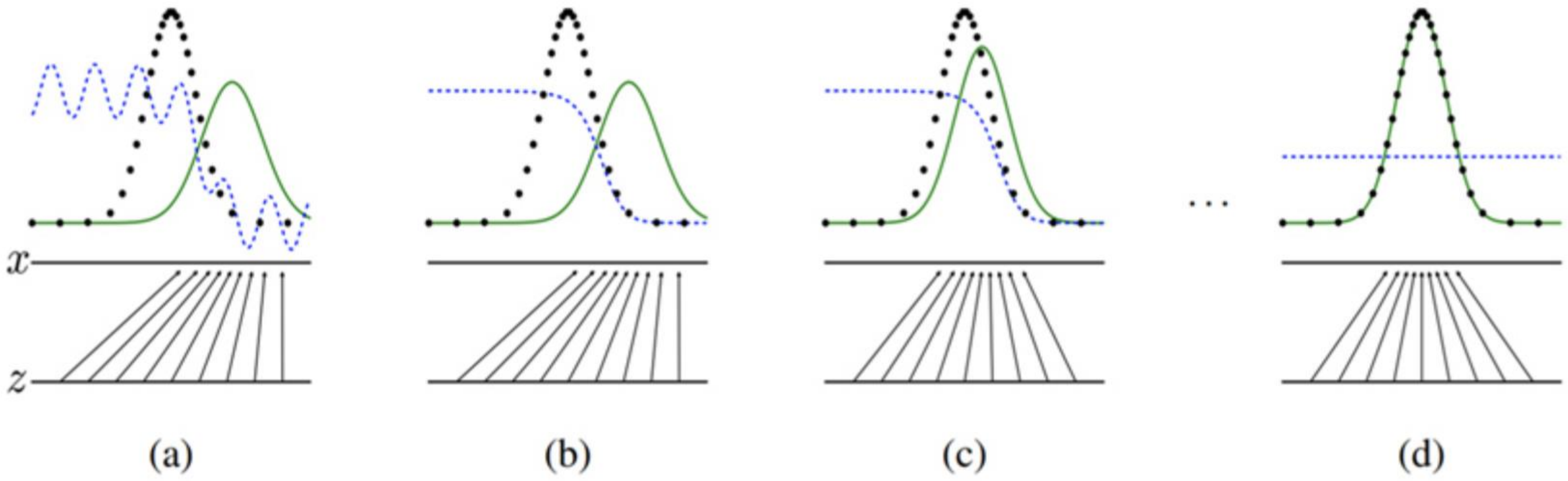


The purpose of the GAN



최적화를 통해 서로 다른 확률분포 간의 차이 줄이기

The purpose of the GAN



※ **검은 점선**: 원 데이터의 확률분포, **녹색 점선**: GAN이 만들어 내는 확률분포, **파란 점선**: 분류자의 확률분포

위로 뻗은 화살표 : x = G(z)의 mapping

<GAN에서 학습을 통해 확률분포를 맞추어 나가는 과정>

GAN

- **N = Network**

- Generator 와 Discriminator가 꼭 neural nets 일 필요는 없음

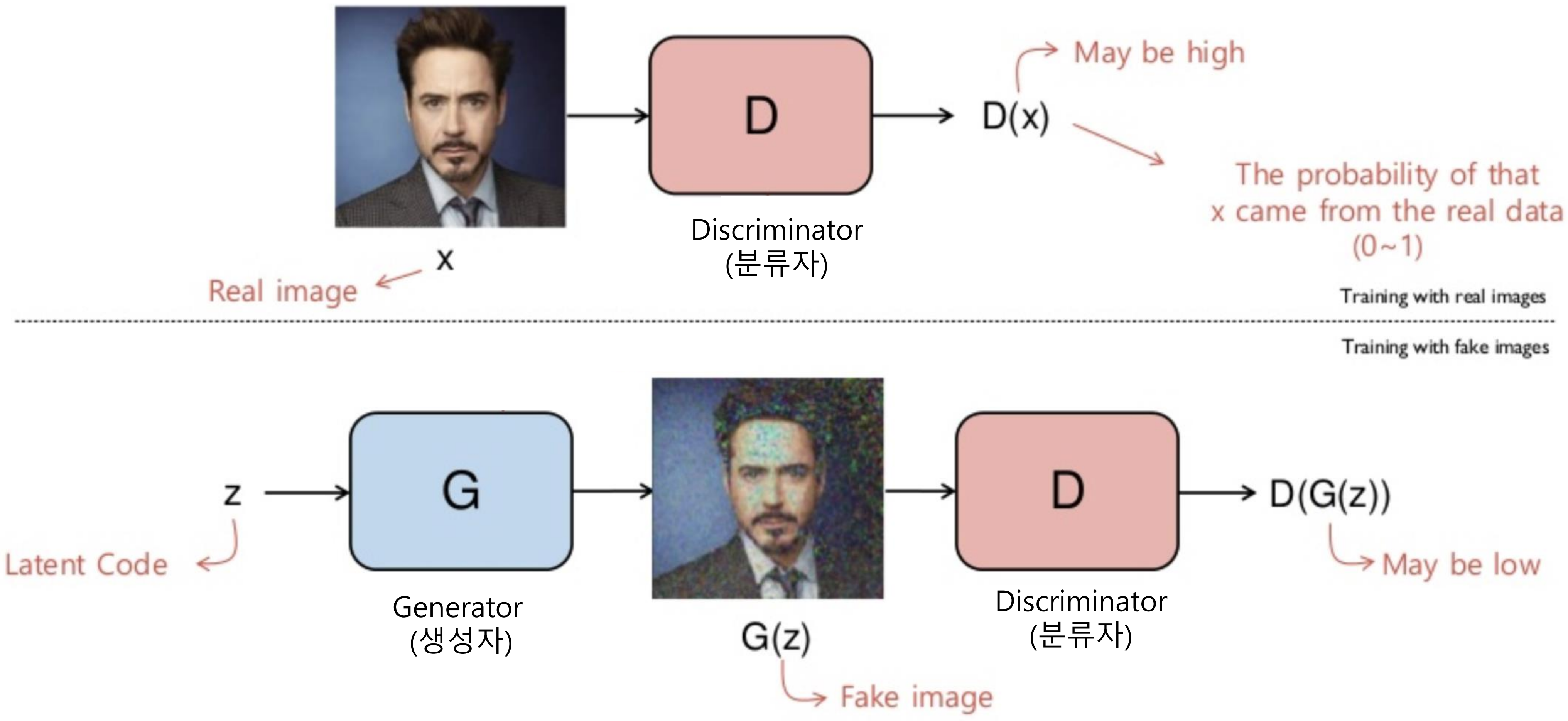
- 뉴럴넷의 장점이 있기 때문에 사용하는 것
  - Non-Linear Activation function
  - Hierarchy structure
  - Backpropagation

- **Generative Adversarial network(GAN)**

  " **생성문제**를 풀기위해 **딥러닝으로 만들어진 모델**을 **적대적학습**이라는 방식으로 학습시키는 알고리즘 "

How

Architecture



GAN loss function

Sample x from real data distribution

Sample z from Gaussian distribution

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

Discriminator loss function

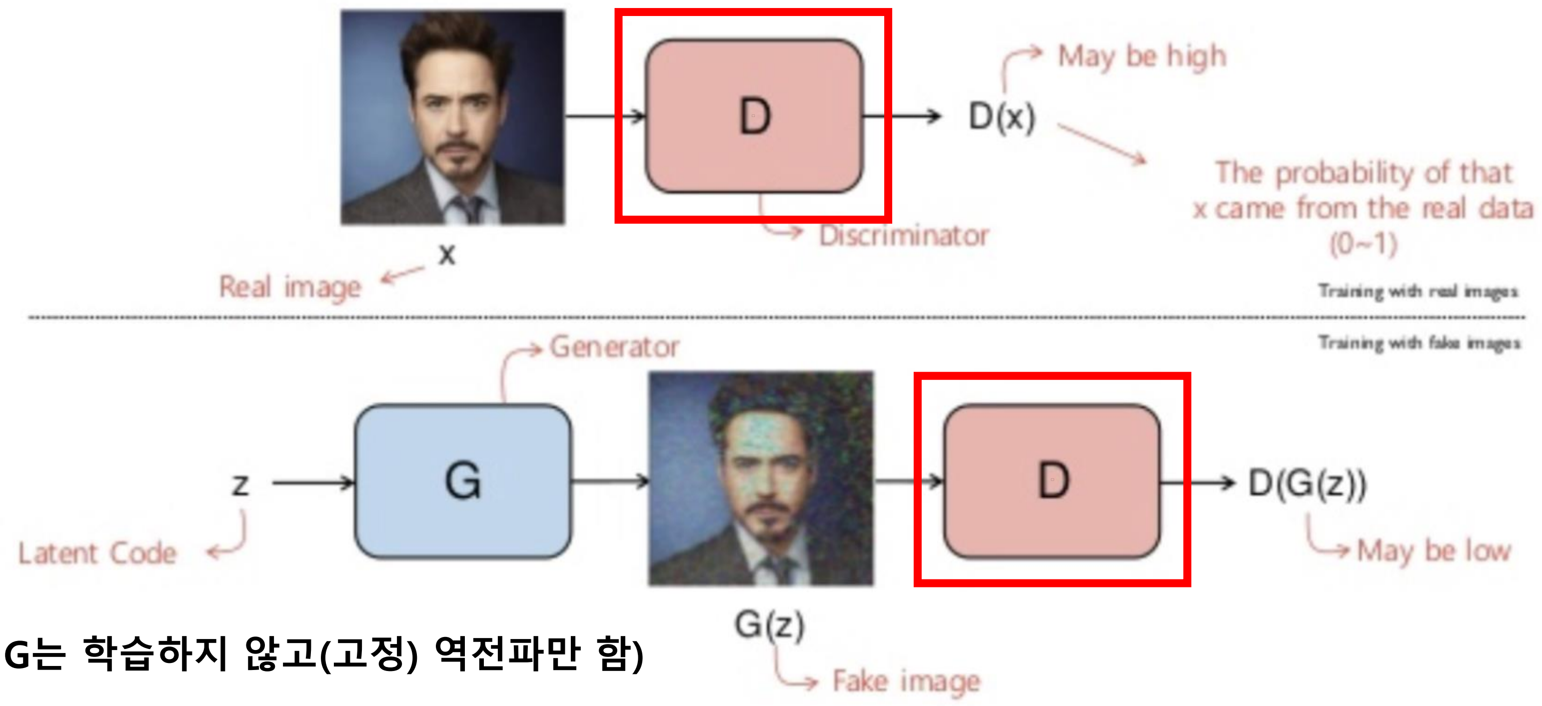
Sample x from real data distribution

Sample z from Gaussian distribution

$$\min_{G} \boxed{\max_{D} V(D,G)} = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_{z}(z)}[\log(1-D(G(z)))]$$

D(x) = 1일때 Maximum

D(G(z)) = 0일때 Maximum



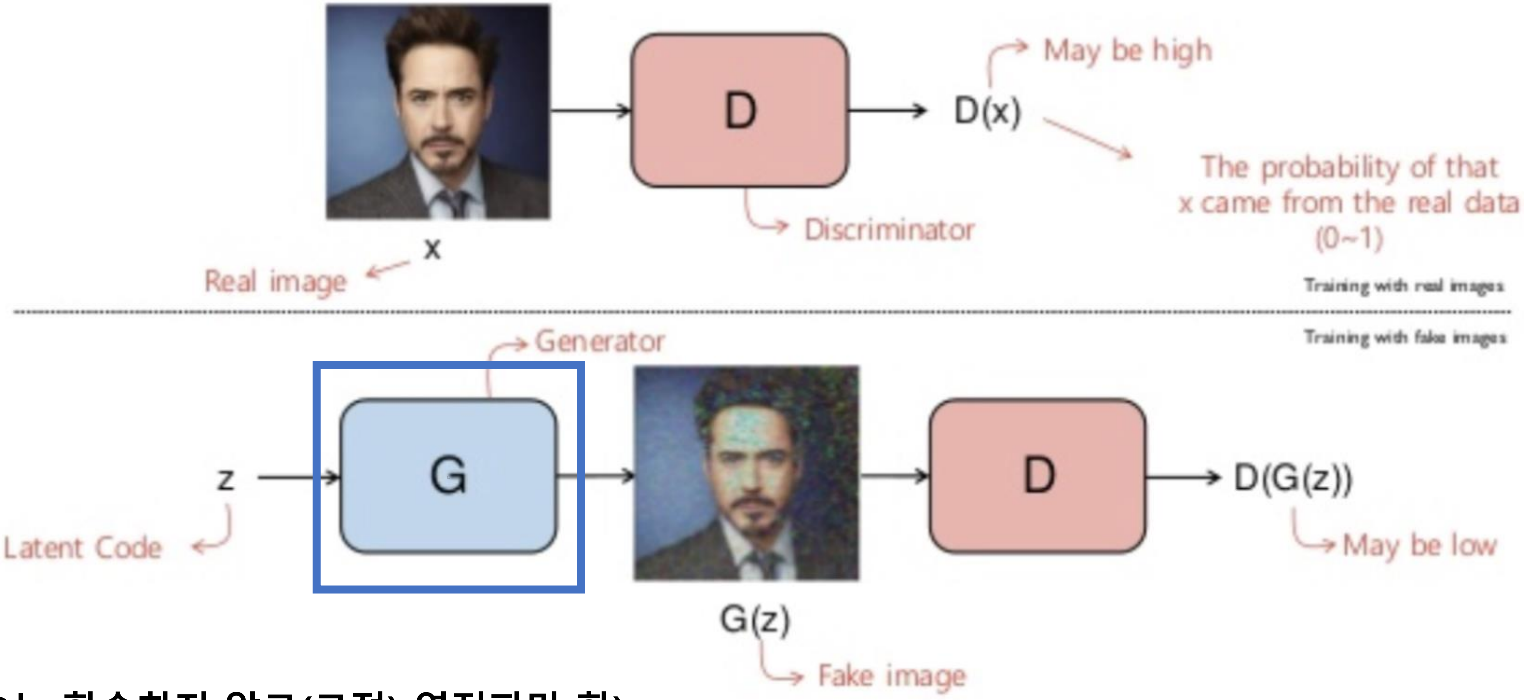
(단, D 학습시에는 G는 학습하지 않고(고정) 역전파만 함)

Generator loss function

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

G is independent

D(G(z)) = 1일때 Minimum



(단, G 학습시에는 D는 학습하지 않고(고정) 역전파만 함)

Generator loss function

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

G is independent

Generator loss function

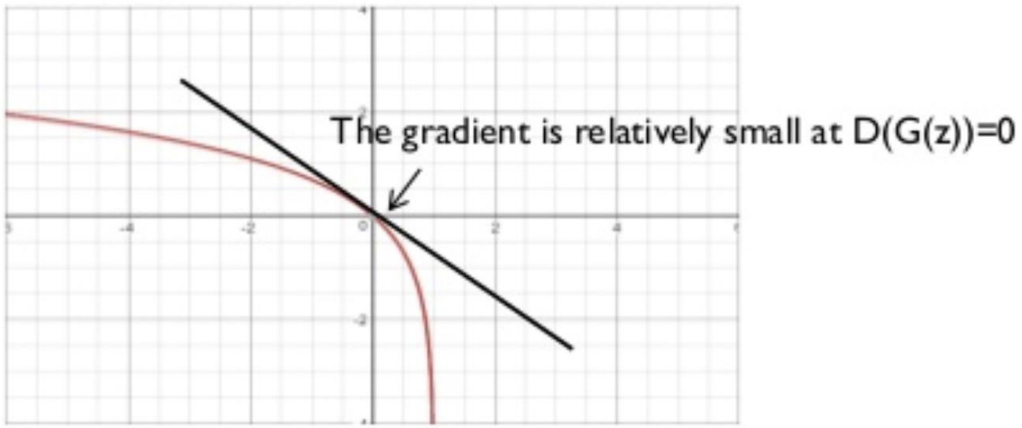
$$\boxed{\min_{G} E_{z\sim p_z(z)}[\log(1 - D(G(z))]}$$

Objective function of G

At the beginning of training, the discriminator can clearly classify the generated image as fake because the quality of the image is very low.

This means that D(G(z)) is almost zero at early stages of training.

Images created by the generator at the beginning of training



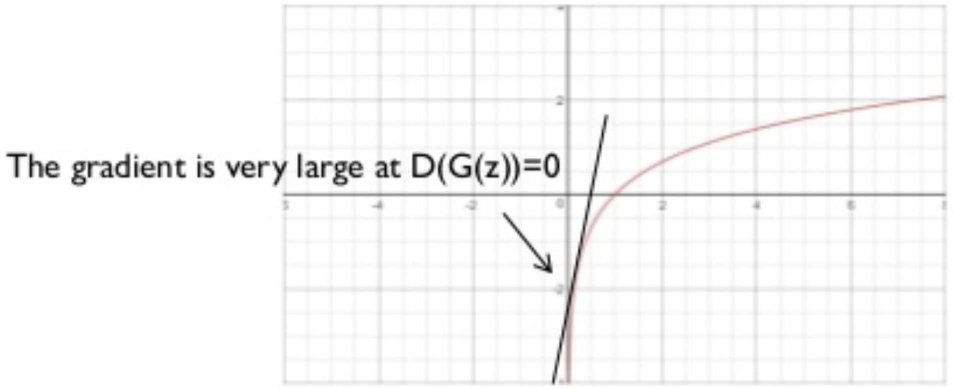
$$y = \log(1 - x)$$

Generator loss function

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

Modification (heuristically motivated)

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



$y = \log(x)$

- 실제로 G의 loss function을 위의 식으로 사용
- 초기 G의 학습을 가속화 가능

Implement
with MNIST

Implement with MNIST

```
# 설정값들을 선언합니다.
num_epoch = 100000
batch_size = 64
num_input = 28 * 28
num_latent_variable = 100
num_hidden = 128
learning_rate = 0.001

# 플레이스 홀더를 선언합니다.
X = tf.placeholder(tf.float32, [None, num_input])              # 인풋 이미지
z = tf.placeholder(tf.float32, [None, num_latent_variable])    # 인풋 Latent Variable

# Generator 변수들 설정
# 100 -> 128 -> 784
with tf.variable_scope('generator'):
    # 히든 레이어 파라미터
    G_W1 = tf.Variable(tf.random_normal(shape=[num_latent_variable, num_hidden], stddev=5e-2))
    G_b1 = tf.Variable(tf.constant(0.1, shape=[num_hidden]))
    # 아웃풋 레이어 파라미터
    G_W2 = tf.Variable(tf.random_normal(shape=[num_hidden, num_input], stddev=5e-2))
    G_b2 = tf.Variable(tf.constant(0.1, shape=[num_input]))

# Discriminator 변수들 설정
# 784 -> 128 -> 1
with tf.variable_scope('discriminator'):
    # 히든 레이어 파라미터
    D_W1 = tf.Variable(tf.random_normal(shape=[num_input, num_hidden], stddev=5e-2))
    D_b1 = tf.Variable(tf.constant(0.1, shape=[num_hidden]))
    # 아웃풋 레이어 파라미터
    D_W2 = tf.Variable(tf.random_normal(shape=[num_hidden, 1], stddev=5e-2))
    D_b2 = tf.Variable(tf.constant(0.1, shape=[1]))
```

Implement with MNIST

```python
def build_generator(X):
    hidden_layer = tf.nn.relu((tf.matmul(X, G_W1) + G_b1))
    output_layer = tf.matmul(hidden_layer, G_W2) + G_b2
    generated_mnist_image = tf.nn.sigmoid(output_layer)

    return generated_mnist_image
```

Generator(생성자) :

INPUT : z from Latent Variable

OUTPUT : 생성된 MNIST 이미지

```python
def build_discriminator(X):
    hidden_layer = tf.nn.relu((tf.matmul(X, D_W1) + D_b1))
    logits = tf.matmul(hidden_layer, D_W2) + D_b2
    predicted_value = tf.nn.sigmoid(logits)

    return predicted_value, logits    # 나중에 손실함수에 넣어주기 위해서 logits 도 return
```

Discriminator(분류자) :

INPUT : 인풋 이미지

OUTPUT :

- predicted_value : 0 or 1
- logits : sigmoid를 씌우기전 출력값

Implement with MNIST

```
# 생성자(Generator)를 선언합니다.
G = build_generator(z)
```

z = 균등분포(Uniform Distribution) or 정규분포(Normal Distribution)에서 무작위로 추출된 값

```
# 구분자(Discriminator)를 선언합니다.
D_real, D_real_logits = build_discriminator(X)  # D(x)
D_fake, D_fake_logits = build_discriminator(G)  # D(G(z))
```

```
# Discriminator의 손실 함수를 정의합니다.
d_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_real_logits, labels=tf.ones_like(D_real_logits)))   # log(D(x))
d_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits, labels=tf.zeros_like(D_fake_logits)))  # log(1-D(G(z)))
d_loss = d_loss_real + d_loss_fake  # log(D(x)) + log(1-D(G(z)))

# Generator의 손실 함수를 정의합니다.
g_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits, labels=tf.ones_like(D_fake_logits)))       # log(D(G(z))
```

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

D(x) = 1일때 Maximum

D(G(z)) = 0일때 Maximum

```
# 전체 파라미터를 Discriminator와 관련된 파라미터와 Generator와 관련된 파라미터로 나눕니다.
tvar = tf.trainable_variables()
dvar = [var for var in tvar if 'discriminator' in var.name]
gvar = [var for var in tvar if 'generator' in var.name]

# Discriminator와 Generator의 Optimizer를 정의합니다.
d_train_step = tf.train.AdamOptimizer(learning_rate).minimize(d_loss, var_list=dvar)
g_train_step = tf.train.AdamOptimizer(learning_rate).minimize(g_loss, var_list=gvar)
```

Implement with MNIST

```
# 생성자(Generator)를 선언합니다.
G = build_generator(z)
```

z = 균등분포(Uniform Distribution) or 정규분포(Normal Distribution)에서 무작위로 추출된 값

```
# 구분자(Discriminator)를 선언합니다.
D_real, D_real_logits = build_discriminator(X)   # D(x)
D_fake, D_fake_logits = build_discriminator(G)   # D(G(z))
```

```
# Discriminator의 손실 함수를 정의합니다.
d_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_real_logits, labels=tf.ones_like(D_real_logits)))    # log(D(x))
d_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits, labels=tf.zeros_like(D_fake_logits)))   # log(1-D(G(z)))
d_loss = d_loss_real + d_loss_fake   # log(D(x)) + log(1-D(G(z)))

# Generator의 손실 함수를 정의합니다.
g_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits, labels=tf.ones_like(D_fake_logits)))         # log(D(G(z))
```

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

D(G(z)) = 1로 Maximize

```
# 전체 파라미터를 Discriminator와 관련된 파라미터와 Generator와 관련된 파라미터로 나눕니다.
tvar = tf.trainable_variables()
dvar = [var for var in tvar if 'discriminator' in var.name]
gvar = [var for var in tvar if 'generator' in var.name]

# Discriminator와 Generator의 Optimizer를 정의합니다.
d_train_step = tf.train.AdamOptimizer(learning_rate).minimize(d_loss, var_list=dvar)
g_train_step = tf.train.AdamOptimizer(learning_rate).minimize(g_loss, var_list=gvar)
```

Implement with MNIST

```python
# 학습 시작
with tf.Session() as sess:
    # 변수들에 초기값을 할당합니다.
    sess.run(tf.global_variables_initializer())

    # num_epoch 횟수만큼 최적화를 수행합니다.
    for i in range(num_epoch):
        # MNIST 이미지를 batch_size만큼 불러옵니다.
        batch_X, _ = mnist.train.next_batch(batch_size)
        # Latent Variable의 인풋으로 사용할 noise를 Uniform Distribution에서 batch_size개 만큼 샘플링합니다.
        batch_noise = np.random.uniform(-1., 1., [batch_size, 100])

        # 500번 반복할때마다 학습된 G를 통해 생성된 이미지를 저장합니다.
        if i % 500 == 0:
            samples = sess.run(G, feed_dict={z: np.random.uniform(-1., 1., [64, 100])})
            fig = plot(samples)
            plt.savefig('generated_output/%s.png' % str(num_img).zfill(3), bbox_inches='tight')
            num_img += 1
            plt.close(fig)

        # Discriminator 최적화를 수행하고 Discriminator의 손실함수를 return합니다.
        _, d_loss_print = sess.run([d_train_step, d_loss], feed_dict={X: batch_X, z: batch_noise})

        # Generator 최적화를 수행하고 Generator 손실함수를 return합니다.
        _, g_loss_print = sess.run([g_train_step, g_loss], feed_dict={z: batch_noise})

        # 100번 반복할때마다 Discriminator의 손실함수와 Generator 손실함수를 출력합니다.
        if i % 100 == 0:
            print('반복(Epoch): %d, Generator 손실함수(g_loss): %f, Discriminator 손실함수(d_loss): %f' % (i, g_loss_print, d_loss_print))
```

GAN_mnist GAN_mnist

```
반복(Epoch): 0, Generator 손실함수(g_loss): 1.262605, Discriminator 손실함수(d_loss): 1.419208
반복(Epoch): 100, Generator 손실함수(g_loss): 2.826195, Discriminator 손실함수(d_loss): 0.604400
반복(Epoch): 200, Generator 손실함수(g_loss): 4.050518, Discriminator 손실함수(d_loss): 0.163514
반복(Epoch): 300, Generator 손실함수(g_loss): 4.773607, Discriminator 손실함수(d_loss): 0.039017
반복(Epoch): 400, Generator 손실함수(g_loss): 5.596404, Discriminator 손실함수(d_loss): 0.010099
반복(Epoch): 500, Generator 손실함수(g_loss): 4.666855, Discriminator 손실함수(d_loss): 0.041880
반복(Epoch): 600, Generator 손실함수(g_loss): 5.627327, Discriminator 손실함수(d_loss): 0.027883
반복(Epoch): 700, Generator 손실함수(g_loss): 6.353477, Discriminator 손실함수(d_loss): 0.022614
반복(Epoch): 800, Generator 손실함수(g_loss): 6.450755, Discriminator 손실함수(d_loss): 0.033502
반복(Epoch): 900, Generator 손실함수(g_loss): 6.995264, Discriminator 손실함수(d_loss): 0.003398
반복(Epoch): 1000, Generator 손실함수(g_loss): 9.158951, Discriminator 손실함수(d_loss): 0.002980
반복(Epoch): 1100, Generator 손실함수(g_loss): 8.082539, Discriminator 손실함수(d_loss): 0.003838
반복(Epoch): 1200, Generator 손실함수(g_loss): 6.271435, Discriminator 손실함수(d_loss): 0.012697
반복(Epoch): 1300, Gener
반복(Epoch): 1400, Gener
반복(Epoch): 1500, Gener
반복(Epoch): 1600, Gener
반복(Epoch): 1700, Gener
반복(Epoch): 1800, Gener
반복(Epoch): 1900, Gener
반복(Epoch): 2000, Gener
반복(Epoch): 2100, Gener
반복(Epoch): 2200, Gener
```

GAN_mnist GAN_mnist

```
반복(Epoch): 9100, Generator 손실함수(g_loss): 3.938745, Discriminator 손실함수(d_loss): 0.374412
반복(Epoch): 9200, Generator 손실함수(g_loss): 3.673210, Discriminator 손실함수(d_loss): 0.772762
반복(Epoch): 9300, Generator 손실함수(g_loss): 3.193578, Discriminator 손실함수(d_loss): 0.636421
반복(Epoch): 9400, Generator 손실함수(g_loss): 3.055054, Discriminator 손실함수(d_loss): 0.491574
반복(Epoch): 9500, Generator 손실함수(g_loss): 3.109817, Discriminator 손실함수(d_loss): 0.642948
반복(Epoch): 9600, Generator 손실함수(g_loss): 3.315445, Discriminator 손실함수(d_loss): 0.561318
반복(Epoch): 9700, Generator 손실함수(g_loss): 2.451287, Discriminator 손실함수(d_loss): 0.843903
반복(Epoch): 9800, Generator 손실함수(g_loss): 3.752067, Discriminator 손실함수(d_loss): 0.785767
반복(Epoch): 9900, Generator 손실함수(g_loss): 3.575946, Discriminator 손실함수(d_loss): 0.388139
반복(Epoch): 10000, Generator 손실함수(g_loss): 3.357232, Discrimin
반복(Epoch): 10100, Generator 손실함수(g_loss): 3.893889, Discrimin
반복(Epoch): 10200, Generator 손실함수(g_loss): 3.122000, Discrimin
반복(Epoch): 10300, Generator 손실함수(g_loss): 3.922439, Discrimin
반복(Epoch): 10400, Generator 손실함수(g_loss): 3.480554, Discrimin
반복(Epoch): 10500, Generator 손실함수(g_loss): 3.166320, Discrimin
반복(Epoch): 10600, Generator 손실함수(g_loss): 3.411620, Discrimin
반복(Epoch): 10700, Generator 손실함수(g_loss): 3.228999, Discrimin
반복(Epoch): 10800, Generator 손실함수(g_loss): 2.573122, Discrimin
반복(Epoch): 10900, Generator 손실함수(g_loss): 2.669272, Discrimin
반복(Epoch): 11000, Generator 손실함수(g_loss): 1.814070, Discrimin
반복(Epoch): 11100, Generator 손실함수(g_loss): 2.398062, Discrimin
반복(Epoch): 11200, Generator 손실함수(g_loss): 2.986684, Discrimin
반복(Epoch): 11300, Generator 손실함수(g_loss): 3.182920, Discrimin
```

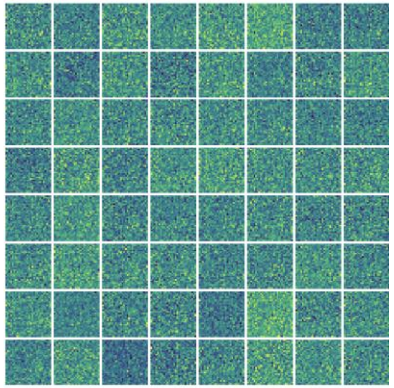
GAN_mnist GAN_mnist

```
반복(Epoch): 98400, Generator 손실함수(g_loss): 2.349915, Discriminator 손실함수(d_loss): 0.476859
반복(Epoch): 98500, Generator 손실함수(g_loss): 2.201246, Discriminator 손실함수(d_loss): 0.521356
반복(Epoch): 98600, Generator 손실함수(g_loss): 2.559083, Discriminator 손실함수(d_loss): 0.478211
반복(Epoch): 98700, Generator 손실함수(g_loss): 2.502994, Discriminator 손실함수(d_loss): 0.566639
반복(Epoch): 98800, Generator 손실함수(g_loss): 2.849582, Discriminator 손실함수(d_loss): 0.461878
반복(Epoch): 98900, Generator 손실함수(g_loss): 2.167712, Discriminator 손실함수(d_loss): 0.702038
반복(Epoch): 99000, Generator 손실함수(g_loss): 2.204830, Discriminator 손실함수(d_loss): 0.484455
반복(Epoch): 99100, Generator 손실함수(g_loss): 2.774933, Discriminator 손실함수(d_loss): 0.512679
반복(Epoch): 99200, Generator 손실함수(g_loss): 2.398154, Discriminator 손실함수(d_loss): 0.747655
반복(Epoch): 99300, Generator 손실함수(g_loss): 2.746816, Discriminator 손실함수(d_loss): 0.276359
반복(Epoch): 99400, Generator 손실함수(g_loss): 2.192427, Discriminator 손실함수(d_loss): 0.687521
반복(Epoch): 99500, Generator 손실함수(g_loss): 2.990449, Discriminator 손실함수(d_loss): 0.386390
반복(Epoch): 99600, Generator 손실함수(g_loss): 2.597083, Discriminator 손실함수(d_loss): 0.386240
반복(Epoch): 99700, Generator 손실함수(g_loss): 2.109751, Discriminator 손실함수(d_loss): 0.466654
반복(Epoch): 99800, Generator 손실함수(g_loss): 2.602525, Discriminator 손실함수(d_loss): 0.620455
반복(Epoch): 99900, Generator 손실함수(g_loss): 1.965860, Discriminator 손실함수(d_loss): 0.551871

Process finished with exit code 0
```

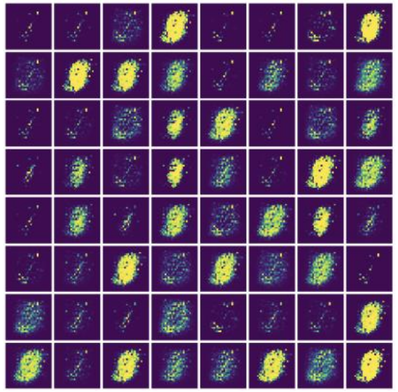
Result

- 생성된 MNIST 이미지 결과

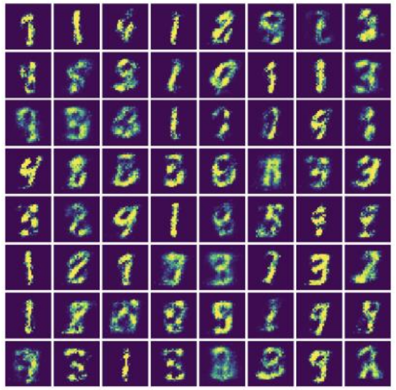


Epoch 0

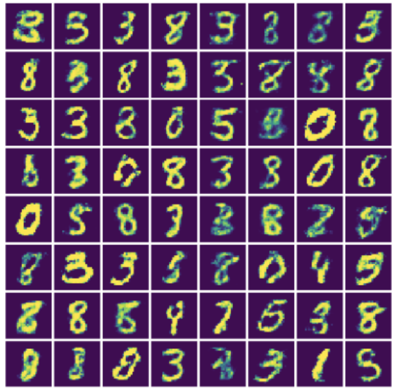
Epoch 500



Epoch 1,000



Epoch 50,000



Epoch 100,000

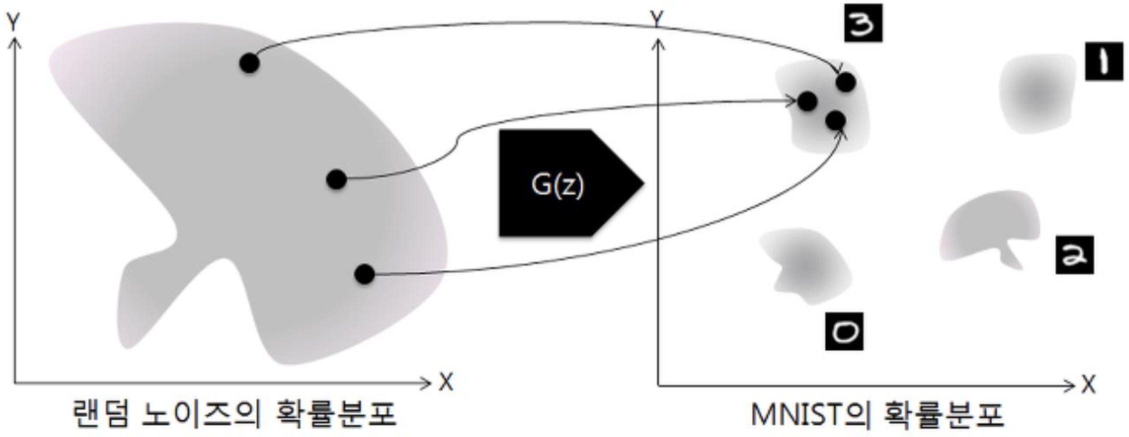
Limitations

1. **Non-convergence** : 모델 파라미터 진동, 불안정 => 수렴하지 않음
2. **Diminished gradient :** discriminator 가 너무 완벽하면, generator의 gradient가 사라짐
3. **Mode Collapse :** 학습시키려는 모형이 실제 데이터의 분포를 모두 커버하지 못하고 다양성을 잃어버리는 현상

$$G^* = \min_G \max_D V(G, D).$$

$$\boxed{G^* = \max_D \min_G V(G, D).}$$

가장 Discriminator 가 헷갈려 할 수 있는 샘플 ‘3’ 만 생성



< Mode Collapse >

Why

Why should we use GAN?

	VAE	GAN
장점	- 모델 평가기준이 명확함 - 학습이 비교적 안정적임	- 훈련이 까다로움
단점	- 이미지가 흐릿함(Blur현상)	- 이미지가 비교적 선명함

Why should we use GAN?

같은 생성모델이라도 더 선명하다.

VAE

GAN

Thank You

Reference

https://dreamgonfly.github.io/2018/03/17/gan-explained.html

https://www.samsungsds.com/global/ko/support/insights/Generative-adversarial-network-AI-2.html

https://www.slideshare.net/NaverEngineering/1-gangenerative-adversarial-network

https://medium.com/@jonathan_hui/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b

https://github.com/TengdaHan/GAN-TensorFlow