Generative adversarial network (GAN)

생성적 적대 신경망

GAN은 머신러닝 분야의 최근 10년 중에서 가장 흥미롭다

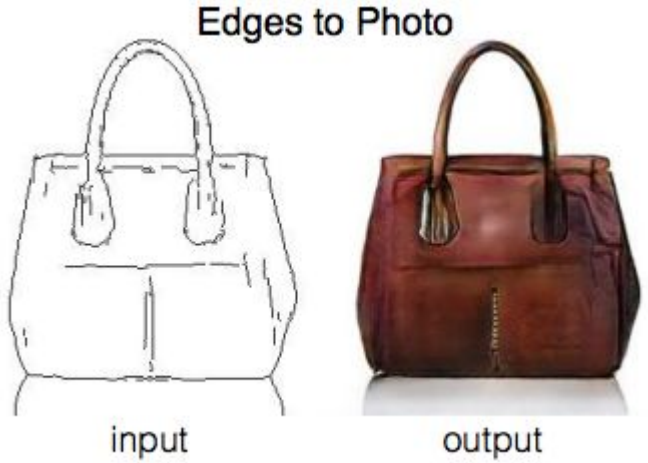
GAN 이란?

- **GAN**(Generative Adversarial Networks)은 머신러닝의 **흥미로운 혁신적 응용**으로써, 기존의 다른 딥러닝 분야와 비교하기가 어렵습니다.
- **GAN**은 **생성 모델**입니다. **GAN**은 **훈련 데이터와 유사한 새로운 데이터를 만듭니다**. 즉, 초상화를 그리거나 교향곡을 작곡하는 것과 같이 작품을 만드는 것입니다. 분류 X
- 예를 들어, **GAN**은 얼굴이 실제 사람의 것이 아니더라도 사람 얼굴 사진처럼 보이는 이미지를 만들 수 있습니다. 다음의 이미지는 GAN에 의해 작성되었습니다.

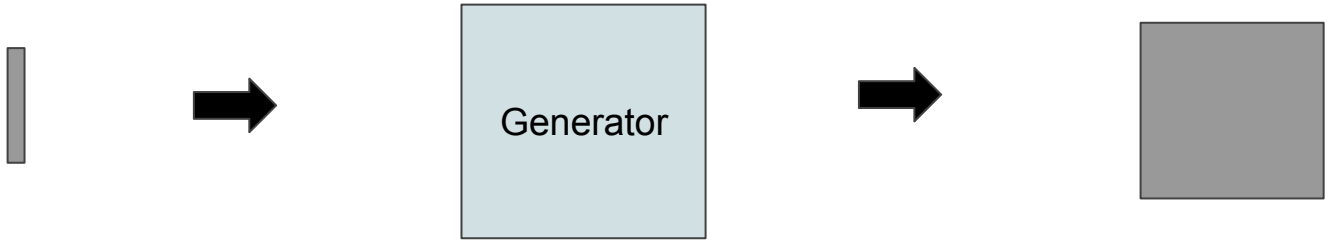
GAN은 무엇을 할 수 있습니까?

GAN 의 주요 목적은 이미지를 생성하는 것이었지만, 음악, 연설, 산문 등의 모든 영역에서의 다양한 생성이 수행되고 있습니다. 그 적용 범위는 매우 넓어, 아래 예에서 처럼 동영상의 일반 말에서 얼룩말을 생성하기도 하고, 스켓치로부터 실제와 같은 사진영상을 생성하기도 합니다.

원본 GAN



Basic Idea of GAN : 생성적 신경망

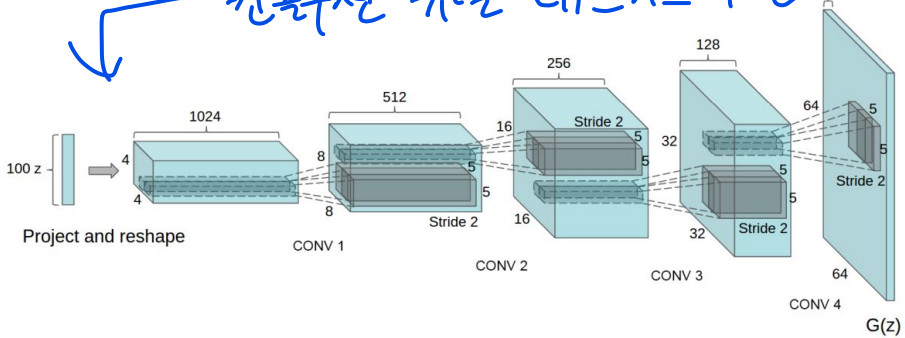


[features]
Input
(vector)

Neural Network
(e.g. Transposed CNN)

컨볼루션 뉴럴 네트워크의 반대

[Images]
output
(matrix)



Basic Idea of GAN : 생성적 신경망

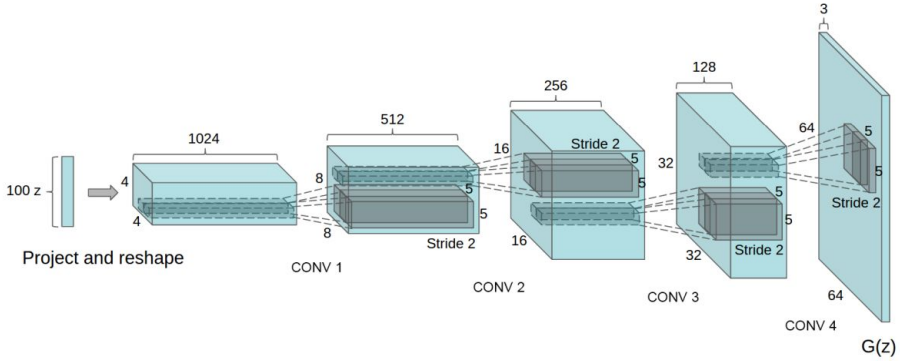
hair length
hair color
smile
...
mouth open
eye color

Generator

[features]
Input
(vector)

Neural Network
(e.g. Transposed CNN)

[Images]
output
(matrix)



Basic Idea of GAN : 생성적 신경망

조절가능

2.1
-0.2
1.7
-0.5
1.0
0.3

Generator

long hair

2.1
3.5
1.7
-0.5
1.0
0.3

Generator

black hair

2.1
3.5
1.7
-0.5
2.5
0.3

Generator

open mouth

2.1
3.5
1.7
-0.5
2.5
3.3

Generator

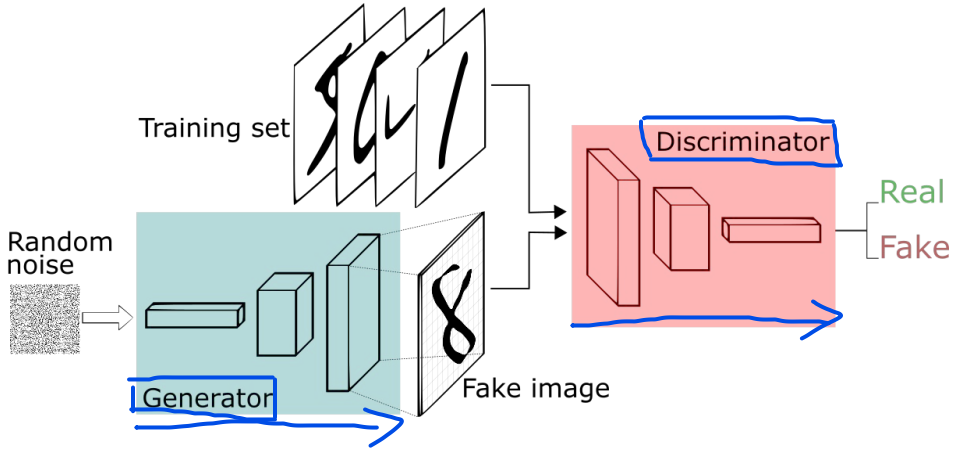
red eye

생성적 적대 신경망(Generative adversarial network)

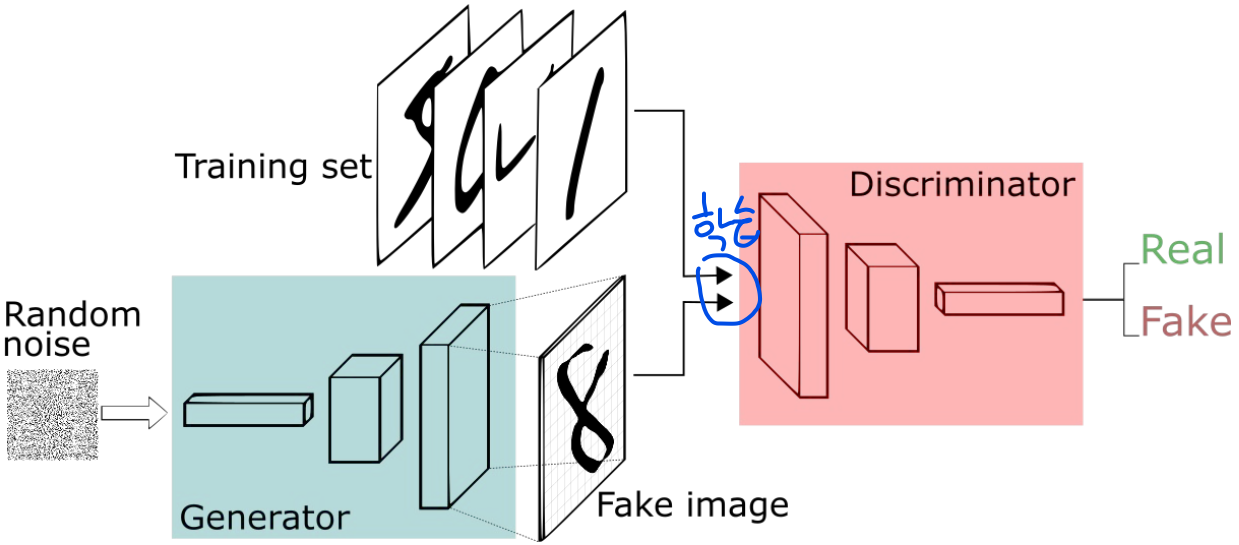
생성적 적대 신경망(GAN)는 **두 네트워크가 서로 경쟁**을 하는 구조이기 때문에 적대적 (adversarial)이라는 의미가 이름에 포함된 심층 신경망의 일종입니다.

- **생성자(Generator)**라는 신경망 **G**는 인공적인 객체(예: 이미지)를 생성.
- **감별자/판별자(Discriminator)**라는 신경망 **D**는 이미지가 진짜인지 아닌지를 파악.

이러한 겨루는 구조를 통해서 사실적인 이미지를 생성 할 수 있는 **생성자 G**를 **훈련**시킵니다.

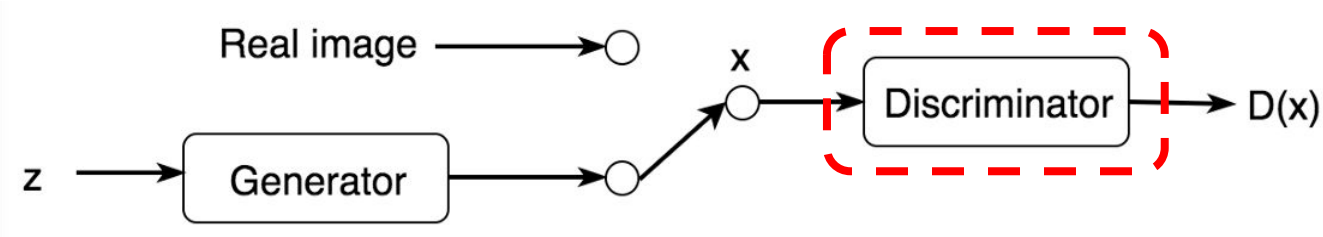


- 처음에 **생성자 G**는 랜덤 노이즈만 발생시킵니다. 개념적으로 GAN의 **감별자 D**가 생성할 이미지에 대한 지침을 **생성자 G**에 제공하게 됩니다.
- GAN은 실제 이미지와 생성된 이미지를 학습함으로써, 어떤 특징들이 이미지가 실제인지를 감별하는데 사용되는지를 **감별자 D**가 학습되도록 합니다.
- 그런 다음 동일한 **감별자 D 가 생성자 G 에 피드백을 제공**하여 실제 이미지처럼 보이는 그림을 만들도록 해줍니다.



How to train D & G ?

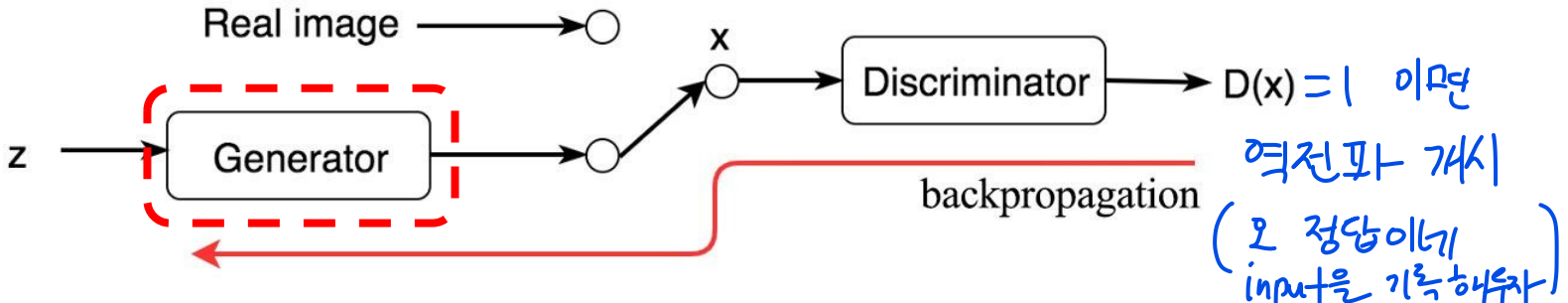
감별자는 실제 이미지(트레이닝 샘플)와 생성된 이미지를 입력으로 사용하여 실제 인지 또는 생성된 것인지를 구별합니다. D(x)의 출력은 입력 x가 실제일 확률, 즉 P(입력 = 실제 이미지)입니다. label을 붙임



우리는 심층신경망 분류기처럼 감별자를 일정 기간동안 훈련시킵니다. 입력이 진짜라면 D(x) = 1 이 되도록, 생성된 것이면 0 이 되도록 훈련을 합니다. 이 프로세스를 통해 감별자 D는 진짜 이미지 식별에 기여하는 특징들을 학습하게 됩니다. 시그모이드를 쓰기도 함

How to train D & G ?

반면에 생성자는 D(x) = 1 (진짜 이미지와 같은)로 이미지를 생성하기를 원합니다. 따라서 우리는 (훈련된 감별자를 고정하고) 이 목표 값을 다시 생성자로 역전파하여 생성자를 훈련시킬 수 있습니다. 즉, 우리는 생성자가 감별자가 진짜로 생각하는 것과 같은 이미지를 생성하도록 훈련시킵니다.



우리는 두 네트워크를 번갈아 가며 훈련시키고, 그들 스스로를 향상시키기 위해 치열한 경쟁에 빠지게 합니다. 결국, 감별자는 진짜와 생성된 것 사이의 작은 차이를 식별하고, 생성자는 감별자가 차이를 말할 수 없는 이미지를 작성합니다. GAN 모델은 결국 수렴하여 자연스러운 모양의 이미지를 생성합니다.

역전파(Backpropagation)

감별자 D는 **x**가 실제 이미지일 가능성을 나타내는 값 **D(x)**를 출력합니다.
우리의 목표는 **실제 이미지를 실제로, 생성된 이미지를 가짜로** 인식 할 수 있는 확률을 최대화 하는 것입니다. 즉, 관찰된 데이터의 최대 가능성을 측정하기 위해 대부분의 딥 러닝에서와 같이 **교차 엔트로피를 사용**합니다.
따라서 **감별자 D**의 **목표**는 다음과 같습니다.

- 진짜 판별: **D(x)** 는 **1**에 가깝게 추정되고,
- 가짜 판별: **D(G(z))** 는 **0**에 가깝게 추정되도록

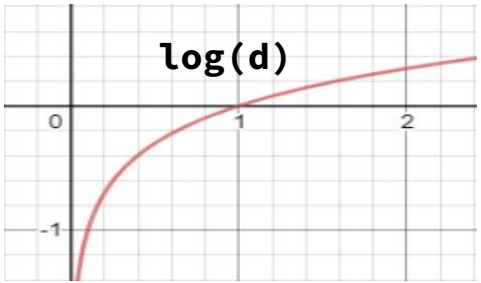
V(D,G)를 최대화시키려고 합니다. V: 비용 함수

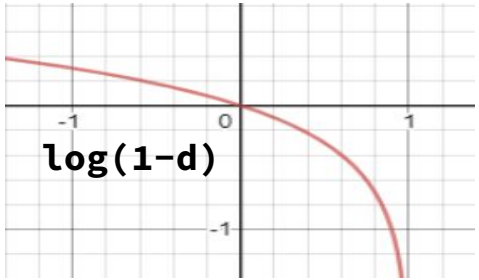
$$\max_D \{V(D,G) = \log D(x) + \log(1 - D(G(z)))\}$$

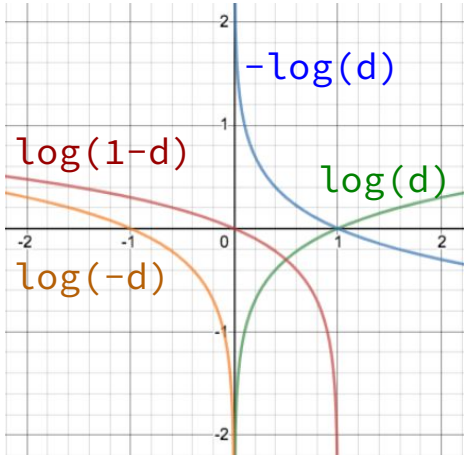
역전파(Backpropagation)

첫번째 항 log D(x)는 0부터 1까지의 값을 갖는 D(x)가 1일 때 가장 커지며,
두번째 항 log (1-D(G(z))) 는 0부터 1까지의 값을 갖는 D(G(z)) 가 0일 때 가장 커짐으로, V(D,G) 가 최대화 되도록, 실제 이미지 x 에 대한 결과는 1, 가짜이미지 G(z) 에 대한 결과는 0 이 되도록 D를 학습시킨다.

$$\max_{D}\{V(D,G) = \log D(x) + \log(1 - D(G(z)))\}$$



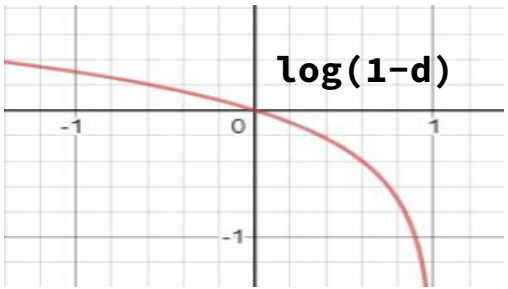




역전파(Backpropagation)

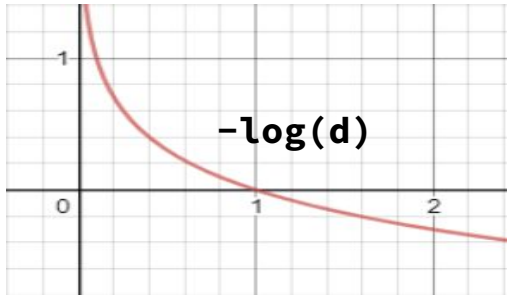
생성자 G 측의 목적 함수는 모델이 **감별자 D**를 속여서, 1 에 가까운 가장 높은 D(G(z)) 값을 주는 **이미지 G(z)**를 생성하기를 원하므로, 목적함수의 두번째 항인 log(1-D(G(z)))가 또는 -log D(G(z)) 가 최소화 되도록, 주어진 D에 대하여, G 가 학습되게 한다.

$$\min_G\{V(G) = \log(1 - D(G(z)))\}$$



또는

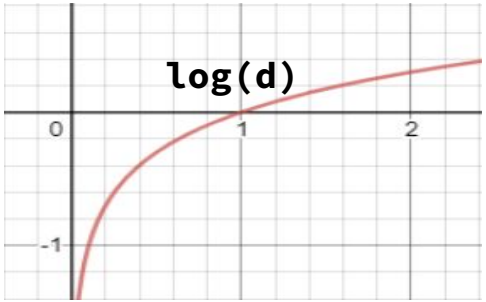
$$\min_G\{V(G) = -\log D(G(z))\}$$

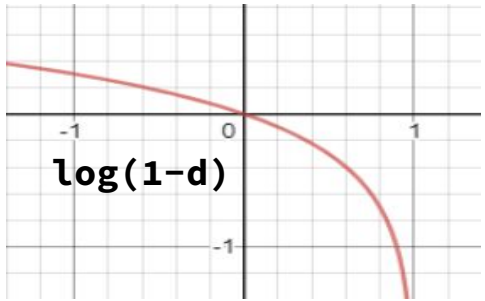


역전파(Backpropagation)

우리는 종종 GAN을 MIN-MAX 게임으로 정의하고, **G는** 전체 목적 함수 **V를 최소화**하고, **D는 V를 최대화**하는 MIN-MAX 최적화 문제를 푼다고 합니다.

$$\min_G \max_D \{V(D,G) = \log D(x) + \log(1 - D(G(z)))\}$$



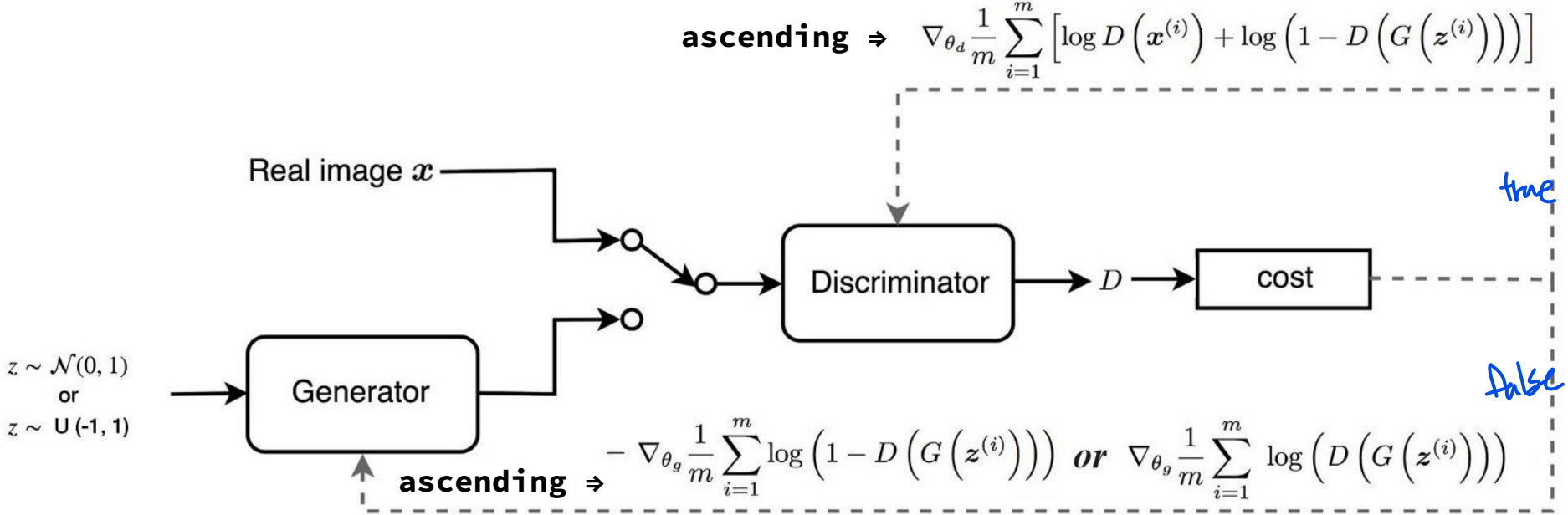


역전파(Backpropagation)

두 가지 목적 함수가 정의되면 **교차 경사 하강**에 의해 함께 학습됩니다.

1. **생성자 G**가 임의의 수를 입력받아 생성한 **이미지를 반환**한다.
2. 이렇게 생성된 이미지는 실제 이미지와 함께 감별자 D에 전달하여, 실제와 가짜를 구분하도록 **감별자 D를 학습시킨다.**
3. 생성자가 새롭게 생성한 **가짜 이미지를 실제로 가정**하고, 일정 수준 학습된 감별자 D의 결과를 비용으로하여 **생성자 G를 학습**시킨다..

우리는 생성자가 좋은 품질의 이미지를 생성할 때까지 **두 네트워크를 교대로 단계적으로 훈련시킵니다**. 다음은 역전파알고리즘에 사용된 데이터 흐름 및 그라디언트를 요약합니다.


ascending ⇒ $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$
Real image $\boldsymbol{x}$
Discriminator
D
cost
true
false
$z \sim \mathcal{N}(0, 1)$
or
$z \sim$ U (-1, 1)
Generator
ascending ⇒
$-\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$ **or** $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$

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

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

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

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

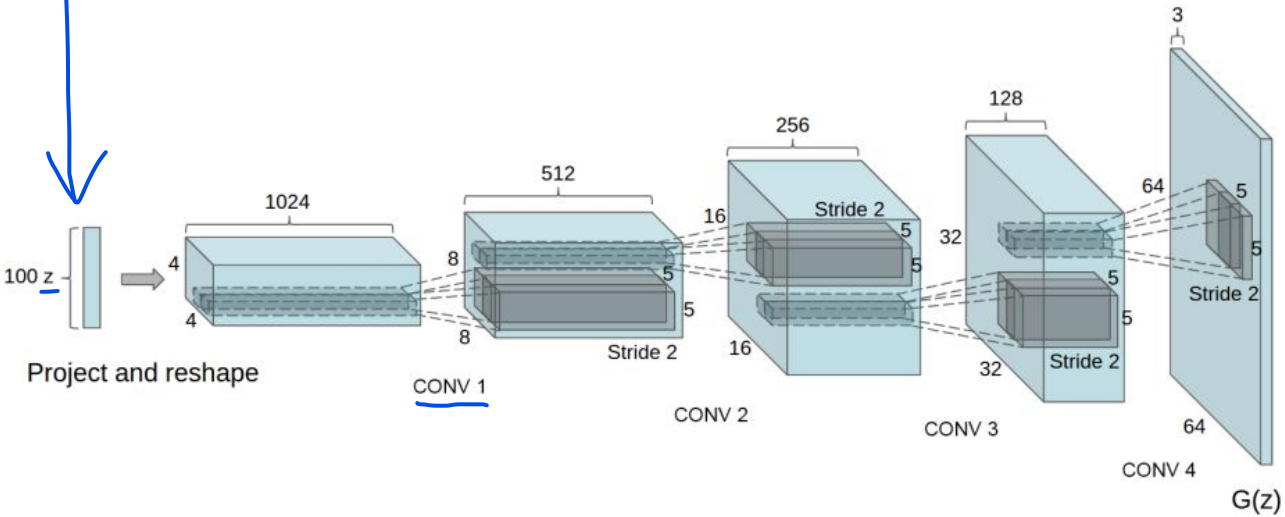
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

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

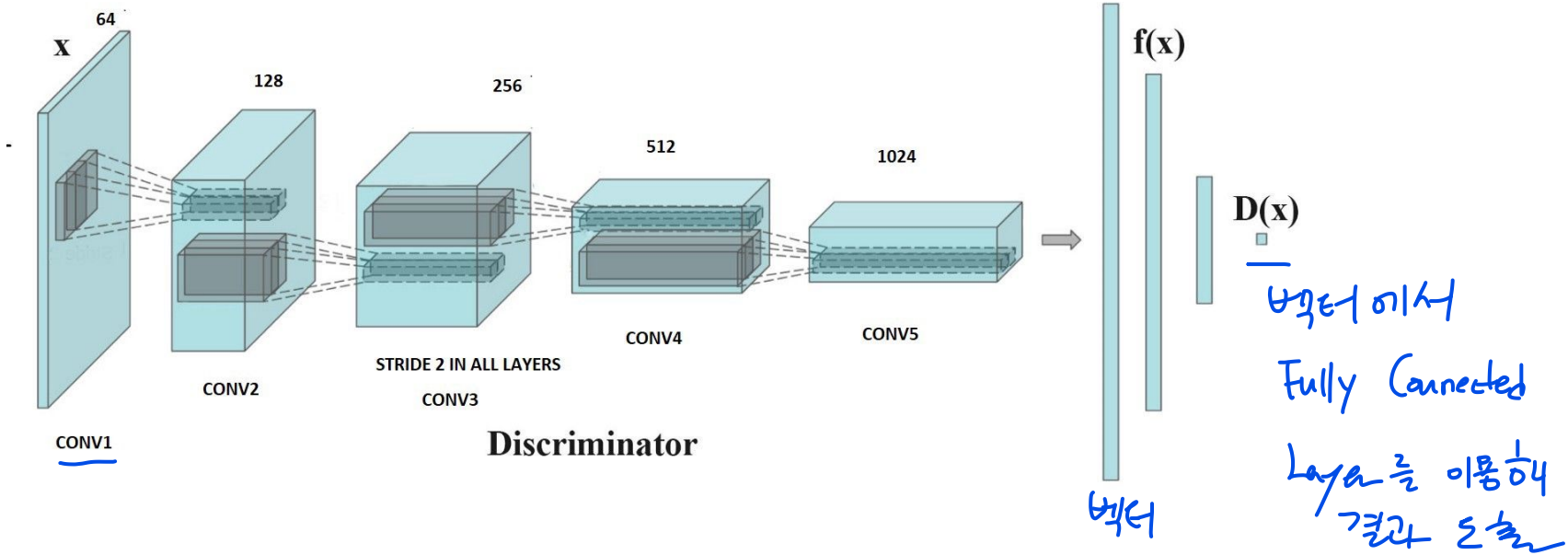
생성자(Generator)

다음은 생성자 네트워크에 가장 많이 사용되는 심층 컨볼루션 망을 활용한 GAN인, **DCGAN** (deep convolutional GAN)입니다. 이는 이미지 **x**를 생성하기 위해 **잠재적 특징(latent features) z** 를 업-샘플링하기 위해 **전치된(transposed) 컨볼루션**을 수행합니다. 이를 거꾸로 된 컨볼루션 딥 러닝 분류기로 볼 수 있습니다



감별자(Discriminator)

감별자는 또한 보폭을 사용하여 안정성을 위해 **다운 샘플링** 및 **배치 정규화**를 수행합니다.



GAN 트레이닝 팁

GAN의 두 개의 네트워크를 트레이닝 하다 보면, 다음과 같은 문제가 생길 수 있다.

- **감별자가 너무 뛰어나면** 0 이나 1 에 매우 가까운 gradient 값을 반환하게 되어, 생성자가 gradient 값을 제대로 반영하기 어렵게 된다.
- **생성자가 너무 뛰어나면** 감별자가 진짜 데이터를 가짜 데이터로 판단할 확률이 높아진다.

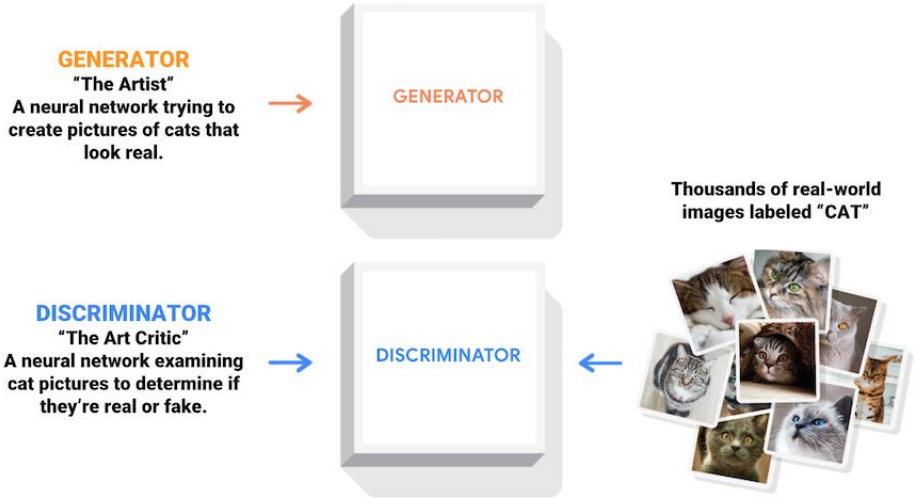
이러한 문제는 두 신경망의 학습률(learning rates)을 각각 설정하여 완화할 수 있다. **두 개의 신경망은 항상 비슷한 "학습 수준"을 유지**해야 한다. 일반적으로 GAN을 트레이닝하는데는 오랜 시간이 걸린다. 하나의 GPU에서는 몇 시간이 걸릴 수도, CPU에서는 하루 이상이 걸릴 수도 있다.

Handwritten digits generation using dcGAN (Example)

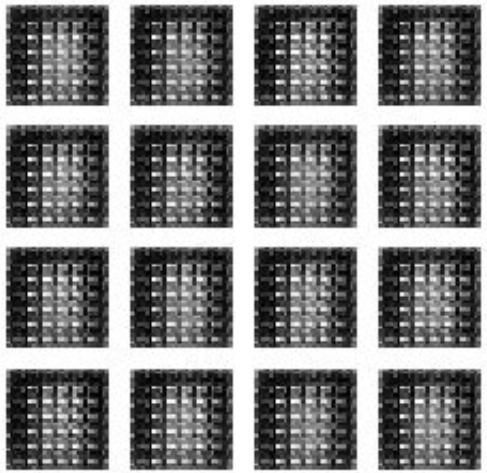
이 튜토리얼은 심층 합성곱 생성적 적대 신경망 (Deep Convolutional Generative Adversarial Networks, DCGAN)을 이용하여, 손으로 쓴 숫자들을 어떻게 생성할 수 있는지 보여줍니다. 이 코드는 케라스 Sequential API와 **tf.GradientTape** 훈련 루프를 사용하여 작성됐습니다.

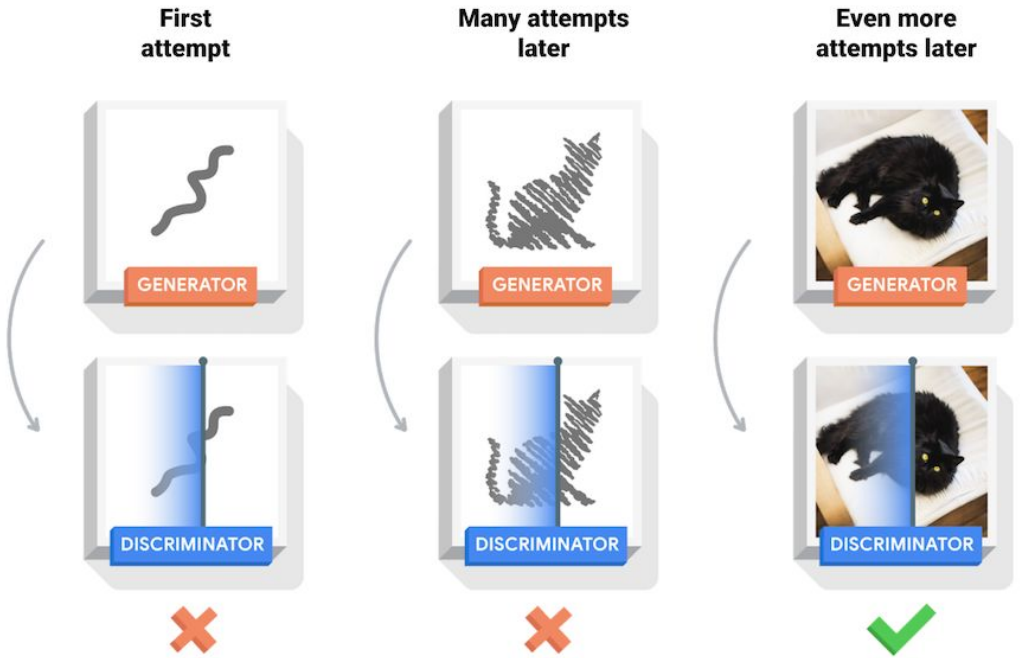
생성적 적대 신경망(GANs)은 무엇인가요?

생성적 적대 신경망 (Generative Adversarial Networks, GANs)은 요즘 컴퓨터 과학에서 가장 흥미로운 아이디어 중 하나입니다. 두개의 모델이 적대적인 과정을 통해 동시에 훈련됩니다. *생성자* ("예술가")는 진짜처럼 보이는 이미지를 생성하도록 배우는 와중에, *감별자* ("예술비평가")는 가짜의 이미지로부터 진짜를 구별하게 되는 것을 배우게 됩니다.



훈련과정 동안 생성자는 점차 실제같은 이미지를 더 잘 생성하게 되고, 감별자는 점차 진짜와 가짜를 더 잘 구별하게됩니다. 이 과정은 감별자가 가짜 이미지에서 진짜 이미지를 더이상 구별하지 못하게 될때 평형상태에 도달하게 됩니다.





이 노트북은 이 과정을 MNIST 데이터를 이용하여 보여줍니다. 왼쪽의 애니메이션은 50 에포크(epoch)동안 훈련한 생성자가 생성해 낸 연속된 이미지들을 보여줍니다. 이미지들은 랜덤한 잡음으로부터 시작되었고, 점차 시간이 지남에 따라 손으로 쓴 숫자들을 닮아가게 됩니다.

텐서플로와 다른 라이브러리 불러오기

```
%tensorflow_version 2.x
import tensorflow as tf
!pip install imageio      # GIF를 만들기위해 설치합니다.
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
```

데이터셋 로딩 및 준비

생성자와 감별자를 훈련하기위해 MNIST 데이터셋을 사용할 것입니다. 생성자는 손글씨 숫자 데이터를 닮은 숫자들을 생성할 것입니다.

```
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (_, _) = 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  # 이미지를 [-1, 1]로 정규화합니다.

BUFFER_SIZE = 60000
BATCH_SIZE = 256
# 데이터 배치를 만들고 섞습니다.
train_dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

모델 만들기

생성자와 감별자는 케라스 Sequential API를 이용해 정의됩니다.

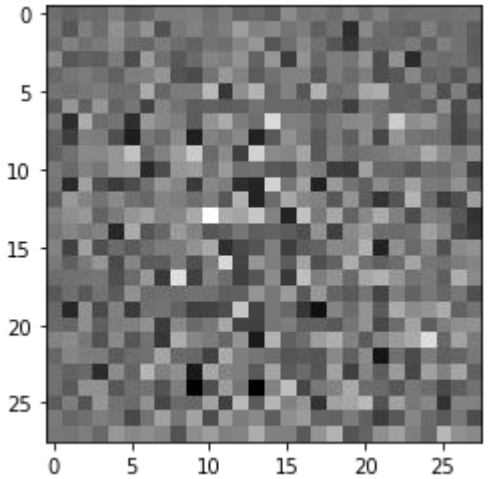
생성자

생성자는 시드값 (seed; 랜덤한 잡음)으로부터 이미지를 생성하기 위해, tf.keras.layers.Conv2DTranspose (업샘플링) 층을 이용합니다. 처음 Dense층은 이 시드값을 입력으로 받습니다. 그 다음 원하는 사이즈 28x28x1의 이미지가 나오도록 업샘플링을 여러번 합니다. tanh를 사용하는 마지막 층을 제외한 나머지 각 층마다 활성함수로 tf.keras.layers.LeakyReLU을 사용하고 있음을 주목합시다.

```python
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) # 주목: 배치사이즈로 None이 주어집니다.

    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=False,
activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

(아직 훈련이 되지않은) 생성자를 이용해 이미지를 생성해봅시다.

```
generator = make_generator_model()

noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```



감별자

감별자는 합성곱 신경망(Convolutional Neural Network, CNN) 기반의 이미지 분류기입니다.

```
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
```

(아직까지 훈련이 되지 않은) 감별자를 사용하여, 생성된 이미지가 진짜인지 가짜인지 판별합니다. 모델은 진짜 이미지에는 양수의 값 (positive values)을, 가짜 이미지에는 음수의 값 (negative values)을 출력하도록 훈련되어집니다.

```
discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)
```

```
tf.Tensor([[-0.00074848]], shape=(1, 1), dtype=float32)
```

손실함수와 옵티마이저 정의

두 모델의 손실함수와 옵티마이저를 정의합니다.

```
# 이 메서드는 크로스 엔트로피 손실함수 (cross entropy loss)를 계산하기 위해 헬퍼 (helper) 함수를 반환합니다.
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

감별자 손실함수

이 메서드는 감별자가 가짜 이미지에서 얼마나 진짜 이미지를 잘 판별하는지 수치화합니다. 진짜 이미지에 대한 감별자의 예측과 1로 이루어진 행렬을 비교하고, 가짜 (생성된) 이미지에 대한 감별자의 예측과 0으로 이루어진 행렬을 비교합니다.

```
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
```

log ...

생성자 손실함수

생성자의 손실함수는 감별자를 얼마나 잘 속였는지에 대해 수치화를 합니다. 직관적으로 생성자가 원활히 수행되고 있다면, 감별자는 가짜 이미지를 진짜 (또는 1)로 분류를 할 것입니다. 여기서 우리는 생성된 이미지에 대한 감별자의 결정을 1로 이루어진 행렬과 비교를 할 것입니다.

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

log...

감별자와 생성자는 따로 훈련되기 때문에, 감별자와 생성자의 옵티마이저는 다릅니다.

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

학습률 조절

체크포인트 저장

이 노트북은 오랫동안 진행되는 훈련이 방해되는 경우에 유용하게 쓰일 수 있는 모델의 저장방법과 복구방법을 보여줍니다.

```
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)
```

훈련 루프 정의하기

```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

# 이 시드를 시간이 지나도 재활용하겠습니다.
# (GIF 애니메이션에서 진전 내용을 시각화하는데 쉽기 때문입니다.)
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

훈련 루프는 생성자가 입력으로 랜덤시드를 받는 것으로부터 시작됩니다. 그 시드값을 사용하여 이미지를 생성합니다. 감별자를 사용하여 (훈련 세트에서 갖고온) 진짜 이미지와 (생성자가 생성해낸) 가짜이미지를 분류합니다. 각 모델의 손실을 계산하고, 그래디언트 (gradients)를 사용해 생성자와 감별자를 업데이트합니다.

```
# `tf.function`이 어떻게 사용되는지 주목해 주세요.
# 이 데코레이터는 함수를 "컴파일"합니다.
@tf.function
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))
```

생성

D에 적용

G 업데이트

D 업데이트

G 업데이트

D 업데이트

역전파

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

    for image batch in dataset:
      train_step(image_batch)

    # GIF를 위한 이미지를 바로 생성합니다.
    display.clear output(wait=True)
    generate_and_save_images(generator, epoch +1, seed)

    # 15 에포크가 지날 때마다 모델을 저장합니다.
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    # print (' 에포크 {} 에서 걸린 시간은 {} 초 입니다'.format(epoch +1, time.time()-start))
    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

  # 마지막 에포크가 끝난 후 생성합니다.
  display.clear output(wait=True)
  generate_and_save_images(generator, epochs, seed)
```

이미지 생성 및 저장

```
def generate and save images(model, epoch, test_input):
  # `training`이 False로 맞춰진 것을 주목하세요.
  # 이렇게 하면 (배치정규화를 포함하여) 모든 층들이 추론 모드로 실행됩니다.
  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()
```

모델 훈련

위에 정의된 train() 메서드를 생성자와 감별자를 동시에 훈련하기 위해 호출합니다. 생성적 적대 신경망을 학습하는 것은 매우 까다로울 수 있습니다. 생성자와 감별자가 서로를 제압하지 않는 것이 중요합니다. (예를 들어 학습률이 비슷하면 한쪽이 우세해집니다.) 훈련 초반부에는 생성된 이미지는 랜덤한 노이즈처럼 보입니다. 훈련이 진행될수록, 생성된 숫자는 점차 진짜처럼 보일 것입니다. 약 50 에포크가 지난 후, MNIST 숫자와 닮은 이미지가 생성됩니다. 코랩에서 기본 설정으로 실행하면, 에포크마다 1분정도 소요될 것입니다.

```
%%time
train(train_dataset, EPOCHS)
```

훈련

```
CPU times: user 2min 14s, sys: 26.4 s, total: 2min 40s
Wall time: 9min 30s
```

마지막 체크포인트를 복구합니다.

```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

GIF 생성

```
# 에포크 숫자를 사용하여 하나의 이미지를 보여줍니다.
def display_image(epoch_no):
  return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))

display_image(EPOCHS)
```

imageio로 훈련 중에 저장된 이미지를 사용해 GIF 애니메이션을 만듭니다.

```
anim_file = 'dcgan.gif'

with imageio.get_writer(anim_file, mode='I') as writer:
  filenames = glob.glob('image*.png')
  filenames = sorted(filenames)
  last = -1
  for i,filename in enumerate(filenames):
    frame = 2*(i**0.5)
    if round(frame) > round(last):
      last = frame
    else:
      continue
    image = imageio.imread(filename)
    writer.append_data(image)
  image = imageio.imread(filename)
  writer.append_data(image)

import IPython
if IPython.version_info > (6,2,0,''):
  display.Image(filename=anim_file)
```

코랩에서 작업하고 있다면, 아래의 코드에서 애니메이션을 다운로드 받을 수 있습니다:

```
try:
  from google.colab import files
except ImportError:
  pass
else:
  files.download(anim_file)
```