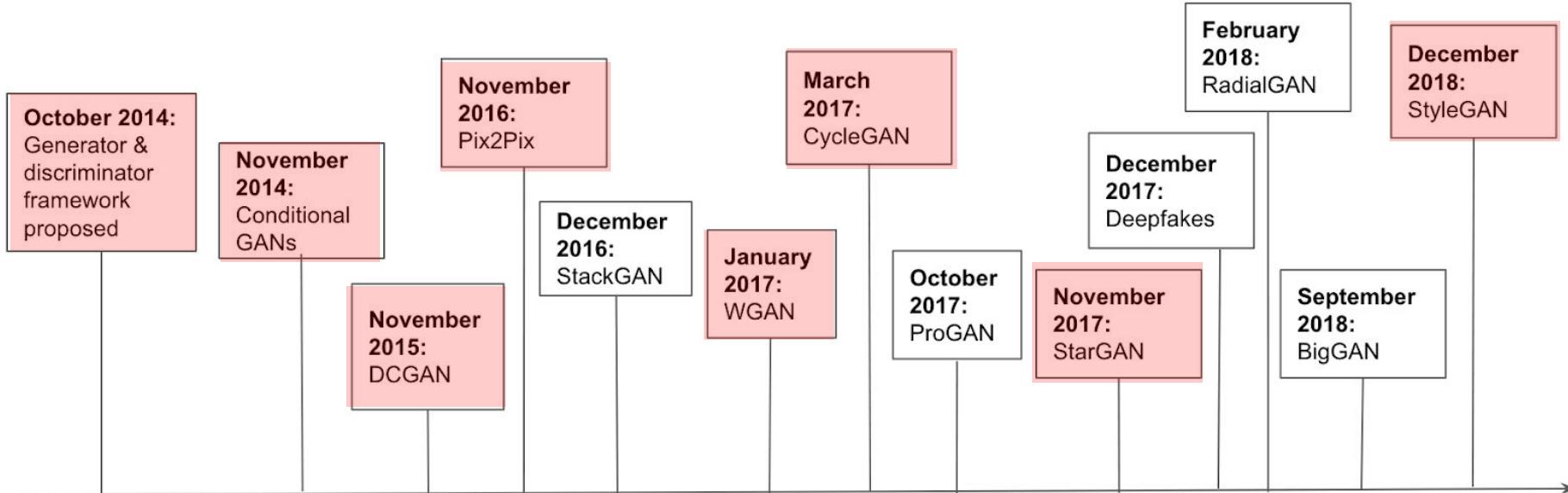


GAN

Generative Adversarial Network

GAN의 발전

Timeline: key developments in GAN research



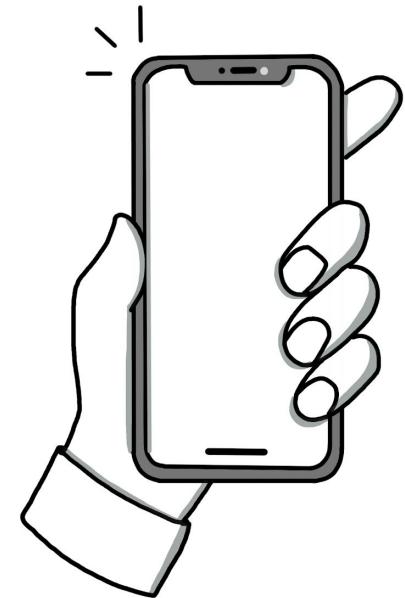
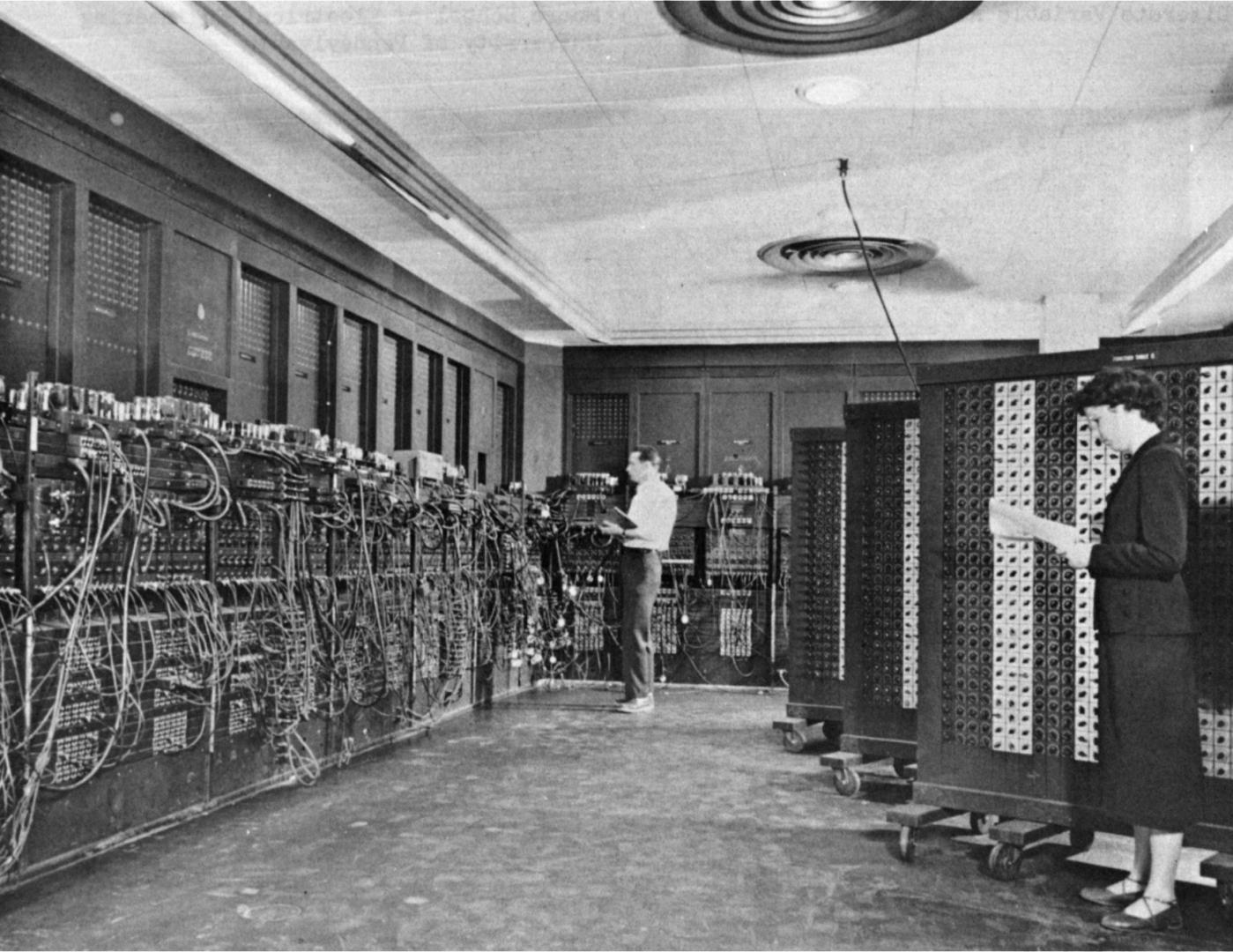


Generative Adversarial Nets (NIPS 2014)

<https://arxiv.org/pdf/1406.2661.pdf>



목표와 전략



```
import tensorflow as tf  
  
# 모델 구조 생성  
X = tf.keras.layers.Input(shape=[1])  
Y = tf.keras.layers.Dense(1)(X)  
model = tf.keras.models.Model(X, Y)  
model.compile(loss='mse')
```

```
# 모델 학습  
model.fit(x_train, y_train, epochs=1000, verbose=0)
```

1. 간단한 코드를 구경하고

2. 코드가 동작하는 것을 경험하고

```
# 모델 구조 생성  
X = tf.keras.layers.Input(shape=[4])  
Y = tf.keras.layers.Dense(3, activation="softmax")(X)  
model = tf.keras.models.Model(X, Y)  
model.compile(  
    loss='sparse_categorical_crossentropy',  
    metrics=['accuracy'])  
  
# 모델 학습  
model.fit(x_train, y_train, epochs=100, verbose=0)  
model.fit(x_train, y_train, epochs=10)
```

3. 해당 코드를 어떻게 이용하면 될지 추측합니다.

```
Epoch 1/10  
1/1 [=====] - 0s 2ms/step - loss: 0.0111  
Epoch 2/10  
1/1 [=====] - 0s 2ms/step - loss: 0.0111  
Epoch 3/10  
1/1 [=====] - 0s 2ms/step - loss: 0.0110  
Epoch 4/10  
1/1 [=====] - 0s 3ms/step - loss: 0.0110  
Epoch 5/10  
1/1 [=====] - 0s 891us/step - loss: 0.0110  
Epoch 6/10  
1/1 [=====] - 0s 2ms/step - loss: 0.0110
```



```
Epoch 1/10  
1/1 [=====] - 0s 3ms/step - loss: 0.7484 - accuracy: 0.6867  
Epoch 2/10  
5/5 [=====] - 0s 2ms/step - loss: 0.7448 - accuracy: 0.6867  
Epoch 3/10  
5/5 [=====] - 0s 3ms/step - loss: 0.7405 - accuracy: 0.6933  
Epoch 4/10  
5/5 [=====] - 0s 3ms/step - loss: 0.7360 - accuracy: 0.7000  
Epoch 5/10  
5/5 [=====] - 0s 3ms/step - loss: 0.7318 - accuracy: 0.6933  
Epoch 6/10  
5/5 [=====] - 0s 3ms/step - loss: 0.7289 - accuracy: 0.6933
```

데이터 준비

```
import tensorflow as tf

(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x_train = (x_train.reshape(-1, 28, 28, 1) - 127.5) / 127.5
x_test = (x_test.reshape(-1, 28, 28, 1) - 127.5) / 127.5

print(x_train.shape, x_train[0].min(), x_train[0].max())
print(x_test.shape, x_test[0].min(), x_test[0].max())

(60000, 28, 28, 1) -1.0 1.0
(10000, 28, 28, 1) -1.0 1.0
```

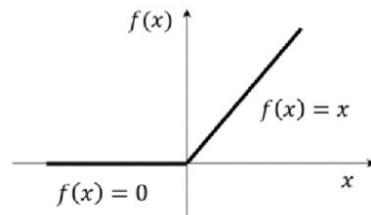
- min-max normalize: 0 ~ 255 범위의 각 픽셀값을 -1.0 ~ 1.0 사이값으로 정규화한다.

Vanilla GAN - Discriminator

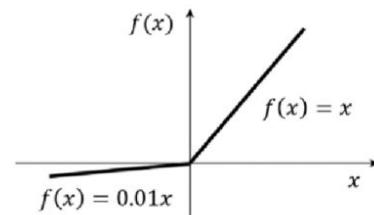
판별기

```
def make_discriminator():
    dx = tf.keras.Input(shape=[28 * 28])
    dh = tf.keras.layers.Dense(128)(dx)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Dense(128)(dh)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dy = tf.keras.layers.Dense(1, activation='sigmoid')(dh)
    return tf.keras.Model(dx, dy, name='discriminator')
```

판별기의 activation 함수는 leakyReLU를 사용한다.



ReLU activation function



LeakyReLU activation function

Vanilla GAN - Generator

```
# 생성기
def make_generator():
    gx = tf.keras.Input(shape=[100])
    gh = tf.keras.layers.Dense(128)(gx)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gh = tf.keras.layers.Dense(128, )(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gy = tf.keras.layers.Dense(28 * 28, activation='tanh')(gh)
    return tf.keras.Model(gx, gy, name='generator')
```

Vanilla GAN 학습

```
discriminator = make_discriminator()
generator = make_generator()
gan = VanillaGAN(generator, discriminator)
gan.fit(x_train.reshape(-1, 28 * 28), epochs=50, batch_size=128)
```

- Class Methods

- `__init__(self, generator, discriminator)`
- `discriminator_loss(self, y_real, y_fake)`
- `generator_loss(self, y_fake)`
- `train_step(self, real)`

Vanilla GAN Model - init

```
class VanillaGAN(tf.keras.Model):
    def __init__(self, generator, discriminator):
        super(VanillaGAN, self).__init__()

        self.generator = generator
        self.discriminator = discriminator

        self.d_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
        self.g_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)

        self.d_loss_metric = tf.keras.metrics.Mean(name="d_loss")
        self.d_acc_metric = tf.keras.metrics.BinaryAccuracy(name="d_acc")
        self.g_loss_metric = tf.keras.metrics.Mean(name="g_loss")
        self.g_acc_metric = tf.keras.metrics.BinaryAccuracy(name="g_acc")

    self.compile()
```

Vanilla GAN Model

- train_step

```
class VanillaGAN(tf.keras.Model):  
  
    def train_step(self, real):  
        noise = tf.random.normal([tf.shape(real)[0], 100])  
  
        with tf.GradientTape() as g_tape, tf.GradientTape() as d_tape:  
            fake = self.generator(noise)  
  
            y_real = self.discriminator(real)  
            y_fake = self.discriminator(fake)  
  
            g_loss = self.generator_loss(y_fake)  
            d_loss = self.discriminator_loss(y_real, y_fake)  
  
            g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)  
            d_grads = d_tape.gradient(d_loss, self.discriminator.trainable_variables)  
  
            self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))  
            self.d_optimizer.apply_gradients(zip(d_grads, self.discriminator.trainable_variables))  
  
            self.update_metrics(y_real, y_fake, g_loss, d_loss)  
        return {  
            "d_acc": self.d_acc_metric.result(),  
            "g_acc": self.g_acc_metric.result(),  
            "d_loss": self.d_loss_metric.result(),  
            "g_loss": self.g_loss_metric.result(),  
        }  

```

Vanilla GAN Model - loss

```
class VanillaGAN(tf.keras.Model):

    def discriminator_loss(self, y_real, y_fake):
        real_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(y_real), y_real)
        fake_loss = tf.keras.losses.binary_crossentropy(tf.zeros_like(y_fake), y_fake)
        d_loss = real_loss + fake_loss
        return d_loss

    def generator_loss(self, y_fake):
        g_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(y_fake), y_fake)
        return g_loss

    def update_metrics(self, y_real, y_fake, g_loss, d_loss):
        self.g_loss_metric.update_state(g_loss)
        self.g_acc_metric.update_state(tf.ones_like(y_fake), y_fake)
        self.d_loss_metric.update_state(d_loss)
        self.d_acc_metric.update_state(tf.ones_like(y_real), y_real)
        self.d_acc_metric.update_state(tf.zeros_like(y_fake), y_fake)
```

Why?

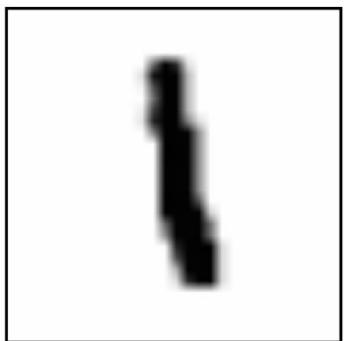
GAN은 왜 필요하지?

그럴듯한 걸 만드는 생성기를 만들고 싶어.

MNIST 이미지 생성 함수?

MNIST 데이터

(28, 28)로 배열되어 있는
784개의 숫자들



?

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	.6	.8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	.7	.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	.7	.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	.5	.1	.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	.1	.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	.1	.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	.1	.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	.1	.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	.9	.1	.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	.3	.1	.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Each image is 28x28 array that can interpret as a big array of numbers. Font: MNIST For ML Beginners

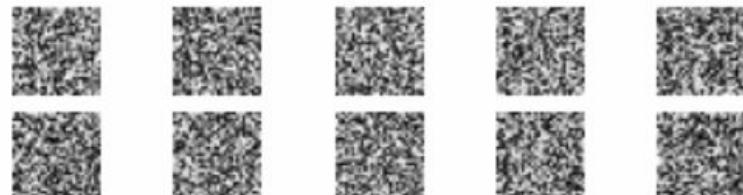
적절한 이미지를 만들어 내는
방법은 없을까?

MNIST 이미지 생성 함수?

```
[17] def generator(number):  
    return np.random.rand(number, 28, 28)  
  
samples = generator(10)
```

적절한 이미지를 만들어 내는
generator 함수를 만드는 것.

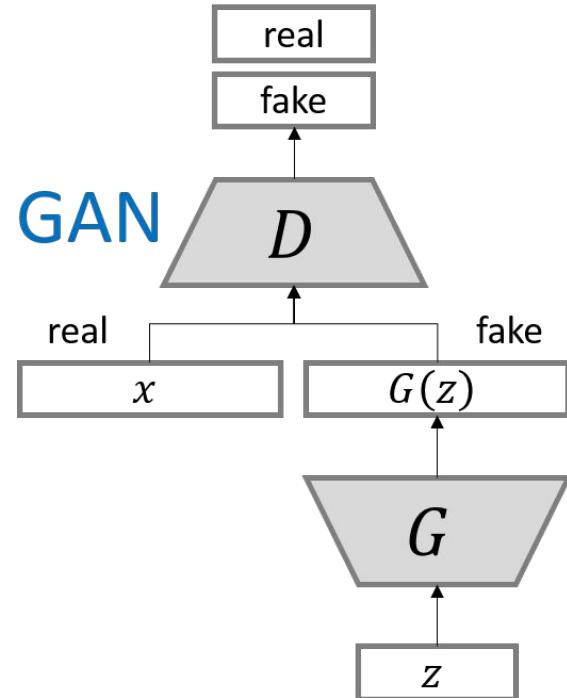
```
[18] for i in range(10):  
    plt.subplot(5, 5, 1 + i)  
    plt.imshow(samples[i], cmap='gray')  
    plt.axis('off')  
  
plt.show()
```



Generative Adversarial Nets (NIPS 2014)

적대적으로 동작하는 G(생성)모델과 D(판별)모델을 동시에 학습시키는 새로운 framework를 제안.

- Generator
 - 실제 이미지 distribution을 학습하여 그럴듯한 이미지를 생성
 - 목적: D가 최대한 실수하게 만드는 것
- Discriminator
 - 진짜 이미지인지 G가 생성한 이미지인지 판별
 - 목적: 최대한 정확하게 진짜/가짜를 판별하는 것이다.



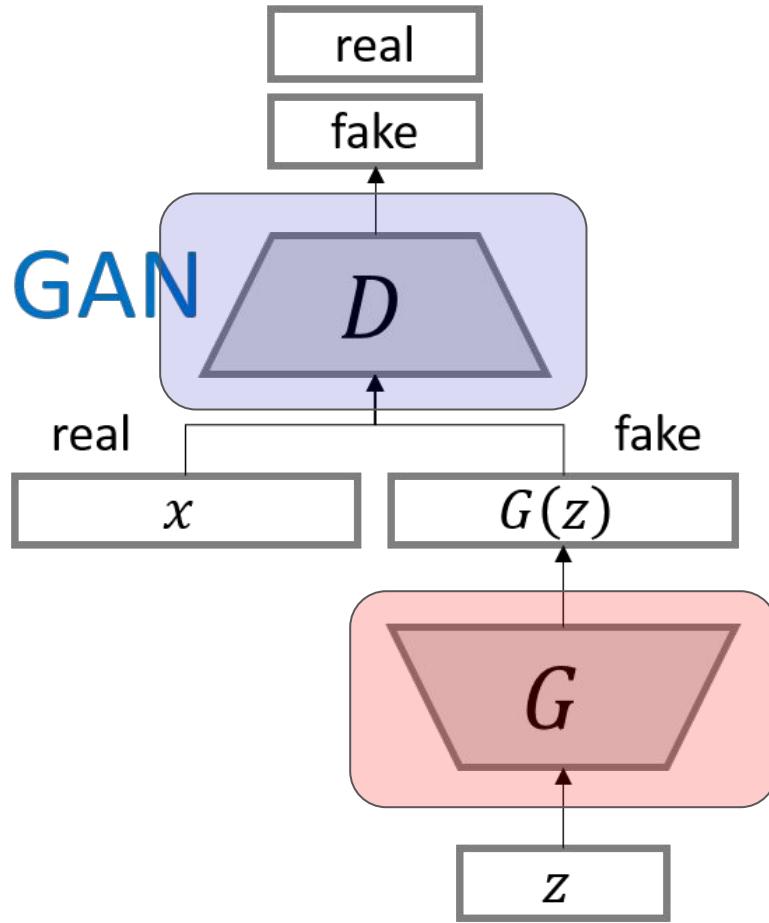
Generator & Discriminator

- Generator: 모작 화가
- Discriminator: 진품 감별사



1. 모작 화가가 **모작**을 만든다.
2. 진품 감별사가 **모작**과 진품으로 학습하여 **감별**을 한다.
3. 모작 화가는 감별사의 능력을 무력화하는 **정밀한 모작**을 만든다.
4. 감별사는 **정밀한 모작**을 보고 **정밀한 감별** 능력을 학습한다.

서로 경쟁하면서 협업이 된다. Adversarial!!

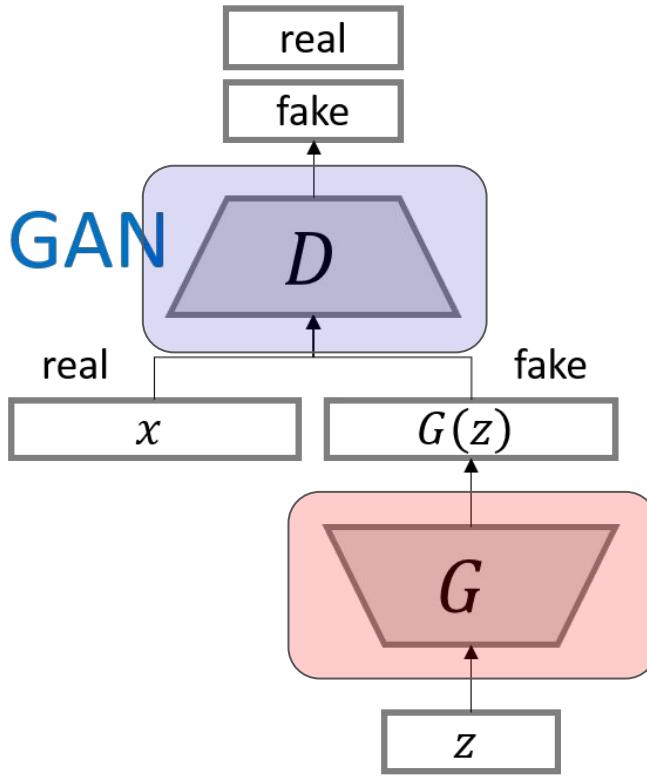


```
# 판별기
def make_discriminator():
    dx = tf.keras.Input(shape=[28 * 28])
    dh = tf.keras.layers.Dense(128)(dx)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Dense(128)(dh)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dy = tf.keras.layers.Dense(1, activation='sigmoid')(dh)
    return tf.keras.Model(dx, dy, name='discriminator')
```

```
# 생성기
def make_generator():
    gx = tf.keras.Input(shape=[100])
    gh = tf.keras.layers.Dense(128)(gx)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gh = tf.keras.layers.Dense(128, )(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gy = tf.keras.layers.Dense(28 * 28, activation='tanh')(gh)
    return tf.keras.Model(gx, gy, name='generator')
```

실제 이미지 판별 결과

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



생성 이미지 판별 결과

```

class VanillaGAN(tf.keras.Model):

    def train_step(self, real):
        noise = tf.random.normal([tf.shape(real)[0], 100])

        with tf.GradientTape() as g_tape, tf.GradientTape() as d_tape:
            fake = self.generator(noise)

            y_real = self.discriminator(real)
            y_fake = self.discriminator(fake)

            g_loss = self.generator_loss(y_fake)
            d_loss = self.discriminator_loss(y_real, y_fake)

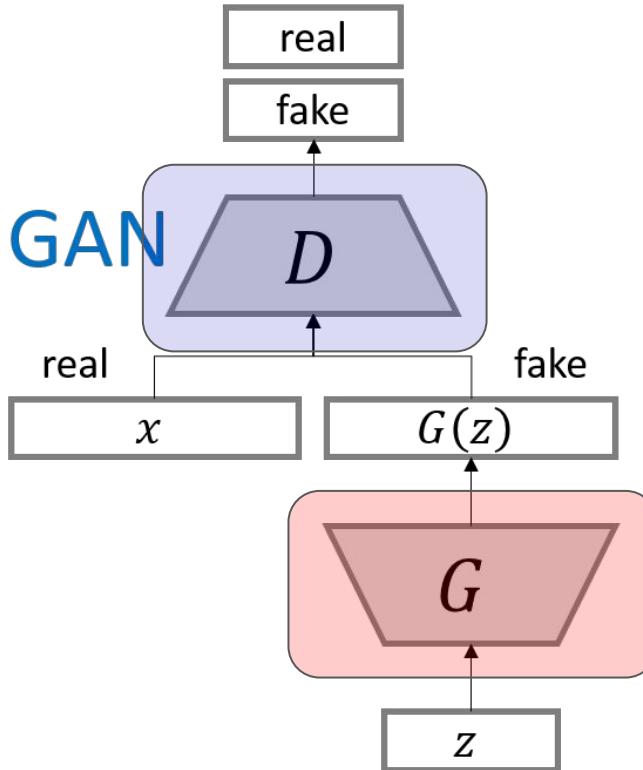
            g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)
            d_grads = d_tape.gradient(d_loss, self.discriminator.trainable_variables)

            self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))
            self.d_optimizer.apply_gradients(zip(d_grads, self.discriminator.trainable_variables))

            self.update_metrics(y_real, y_fake, g_loss, d_loss)
        return {
            "d_acc": self.d_acc_metric.result(),
            "g_acc": self.g_acc_metric.result(),
            "d_loss": self.d_loss_metric.result(),
            "g_loss": self.g_loss_metric.result(),
        }
    
```

실제 이미지 판별 결과

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



생성 이미지 판별 결과

```
class VanillaGAN(tf.keras.Model):

    def discriminator_loss(self, y_real, y_fake):
        real_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(y_real), y_real)
        fake_loss = tf.keras.losses.binary_crossentropy(tf.zeros_like(y_fake), y_fake)
        d_loss = real_loss + fake_loss
        return d_loss

    def generator_loss(self, y_fake):
        g_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(y_fake), y_fake)
        return g_loss

    def update_metrics(self, y_real, y_fake, g_loss, d_loss):
        self.g_loss_metric.update_state(g_loss)
        self.g_acc_metric.update_state(tf.ones_like(y_fake), y_fake)
        self.d_loss_metric.update_state(d_loss)
        self.d_acc_metric.update_state(tf.ones_like(y_real), y_real)
        self.d_acc_metric.update_state(tf.zeros_like(y_fake), y_fake)
```

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**

가짜 샘플 준비

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.

진짜 샘플 준비

- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

판별기 학습

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

end for

가짜 샘플 준비

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

생성기 학습

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

end for

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

학습 및 모니터링

```
import matplotlib.pyplot as plt

class Monitor(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        x_rnd = np.random.randn(5 * 100).reshape(5, 100)
        generated = self.model.generator(x_rnd)
        for i in range(5):
            plt.subplot(1, 5, 1 + i)
            plt.axis('off')
            plt.imshow(generated[i].numpy().reshape(28, 28), cmap='gray_r')
        plt.show()
```

callback을 이용하여 epoch마다 generator로 생성하는 이미지를 출력해준다.

```
discriminator = make_discriminator()
generator = make_generator()
gan = VanillaGAN(generator, discriminator)
gan.fit(x_train.reshape(-1, 28 * 28), epochs=50, batch_size=128, callbacks=[Monitor()])
```

실습. 데이터는 MNIST

만들어보자. VanillaGAN

복사해서 붙여넣기는 금지.

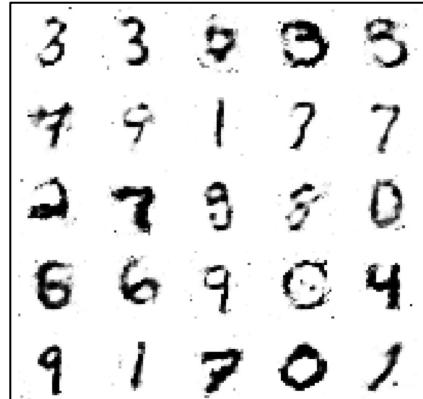
처음부터 끝까지 눈으로 따라가면서 직접 작성해봅시다.

vanillaGAN 결과 확인

```
import matplotlib.pyplot as plt
import numpy as np

x_test = np.random.normal(size=2500).reshape(25, 100)
images = generator.predict(x_test)

for i in range(25):
    plt.subplot(5, 5, 1 + i)
    plt.axis('off')
    plt.imshow(images[i].reshape(28, 28), cmap='gray_r')
```



7	3	9	3	9	9	
1	1	0	6	0	0	
0	1	9	1	2	2	
6	3	2	0	8	8	

a)



b)



c)

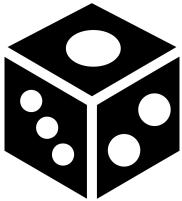


d)

What?

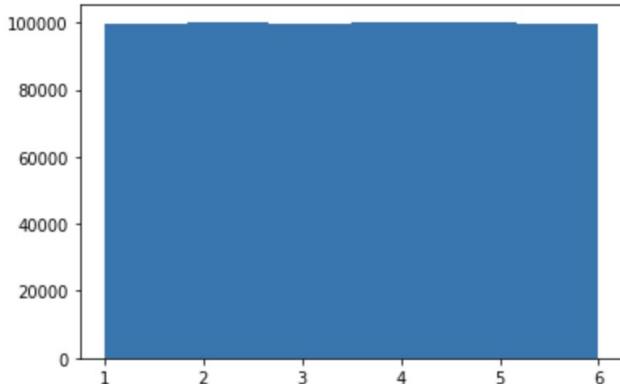
Distribution(분포)을 학습한다는 것은 무슨 뜻이야?
아니, 그 전에 Distribution이란 대체 뭘까?

“실제 이미지 distribution을 학습하여 그럴듯한 이미지를 생성”



Dice

확률변수: 주사위 눈



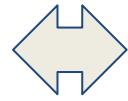
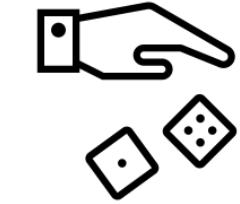
```
import random

def dice():
    return random.randint(1, 6)

number = 600000
result = [dice() for i in range(number)]
```

```
import matplotlib.pyplot as plt
plt.hist(result, bins=6)
plt.show()
```

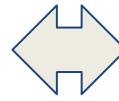
통계



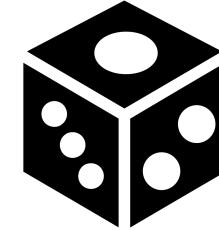
A screenshot of a Microsoft Excel spreadsheet titled "Portfolio_1st_Data.xlsx". The spreadsheet contains data for various stocks, including their symbols, names, purchase price, cost basis, current price, market value, and daily gain/loss percentage. The data is organized into columns A through J.

Stock Name	Symbol	Name	Purchase Price	Cost Basis	Current Price	Market Value	Daily Gain/Loss	Percent Change
Apple	AAPL	Apple Inc.	\$265.00	\$9,000.00	\$344.32	\$11,229.64	\$1,229.64	3.59%
Amazon	AMZN	Amazon.com, Inc.	\$250.00	\$15,000.00	\$250.00	\$15,000.00	\$0.00	0.00%
Safeway	SWY	Safeway, Inc.	\$25.00	\$5,000.00	\$25.00	\$5,000.00	\$0.00	0.00%
Hewlett-Packard Enterprise	HPE	Hewlett-Packard Enterprise Co.	\$50.00	\$10,000.00	\$50.00	\$10,000.00	\$0.00	0.00%
Intel	INTC	Intel Corporation	\$100.00	\$10,000.00	\$100.00	\$10,000.00	\$0.00	0.00%
Goldman Sachs	GS	Goldman Sachs & Co. LLC	\$100.00	\$10,000.00	\$100.00	\$10,000.00	\$0.00	0.00%
Facebook	FB	Facebook, Inc.	\$265.00	\$10,000.00	\$265.00	\$10,000.00	\$0.00	0.00%
Twitter	TWTR	Twitter, Inc.	\$100.00	\$10,000.00	\$100.00	\$10,000.00	\$0.00	0.00%

Data



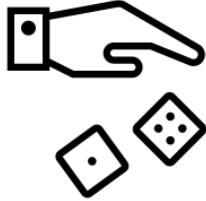
Model



학률



통계

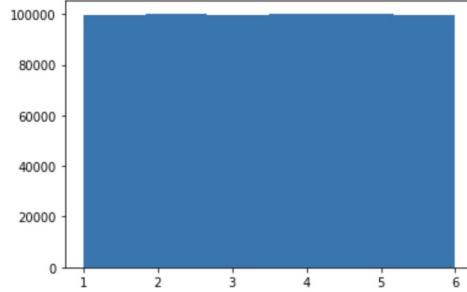


$$1 * \frac{1}{6} + 2 * \frac{1}{6} + 3 * \frac{1}{6} + 4$$

$$* \frac{1}{6} + 5 * \frac{1}{6} + 6 * \frac{1}{6}$$

기댓값: 3.5

확률

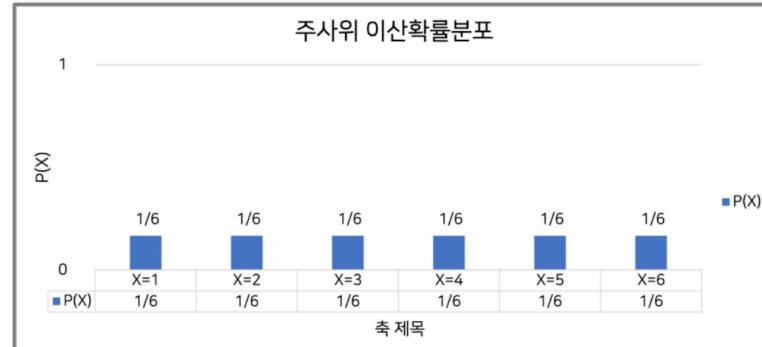


$$(1 + 2 + 3 + 4 + 5 + 6) / 6$$

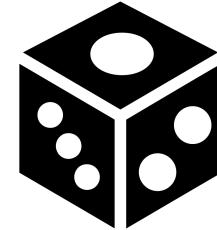
평균: 3.5

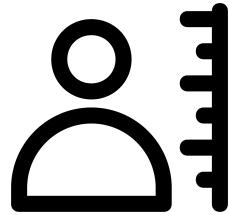
Stock Name	Symbol	Name	Purchase Price	Cost Basis	Current Price	Market Value	Dividend Yield
AAPL	AAPL	Apple Inc.	\$26.96	\$9,046.44	\$244.63	\$21,943.34	5.10%
AMZN	AMZN	Amazon.com, Inc.	\$1,250.00	\$1,250.00	\$1,250.00	\$1,250.00	0.00%
BAC	BAC	Bank of America Corp.	\$25.00	\$3,750.00	\$20.00	\$15,000.00	0.10%
CSCO	CSCO	Cisco Systems, Inc.	\$50.00	\$2,500.00	\$50.00	\$2,500.00	0.00%
FB	FB	Facebook, Inc.	\$200.00	\$40,000.00	\$184.00	\$73,600.00	0.00%
GOOG	GOOG	Alphabet Inc. C	\$1,250.00	\$1,250.00	\$1,250.00	\$1,250.00	0.00%
GOOGL	GOOGL	Alphabet Inc. C	\$1,250.00	\$1,250.00	\$1,250.00	\$1,250.00	0.00%
INTC	INTC	Intel Corporation	\$100.00	\$10,000.00	\$100.00	\$10,000.00	0.00%
MSFT	MSFT	Microsoft Corporation	\$100.00	\$10,000.00	\$100.00	\$10,000.00	0.00%
NKE	NKE	Nike, Inc.	\$60.00	\$3,600.00	\$60.00	\$3,600.00	0.00%
PEP	PEP	PepsiCo, Inc.	\$100.00	\$10,000.00	\$100.00	\$10,000.00	0.00%
TMUS	TMUS	T-Mobile US, Inc.	\$100.00	\$10,000.00	\$100.00	\$10,000.00	0.00%
VZ	VZ	Verizon Communications Inc.	\$100.00	\$10,000.00	\$100.00	\$10,000.00	0.00%
WMT	WMT	Wal-Mart Stores, Inc.	\$100.00	\$10,000.00	\$100.00	\$10,000.00	0.00%
YHOO	YHOO	Yahoo! Inc.	\$100.00	\$10,000.00	\$100.00	\$10,000.00	0.00%

주사위 이산확률분포

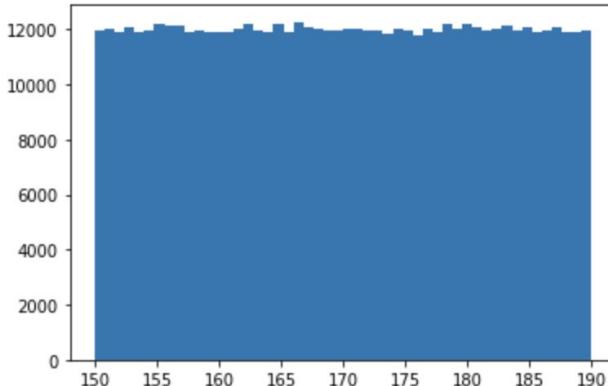


Model





사람
확률변수: 키



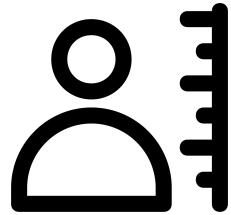
```
import random

def height(min, max):
    return min + (max - min) * random.random()

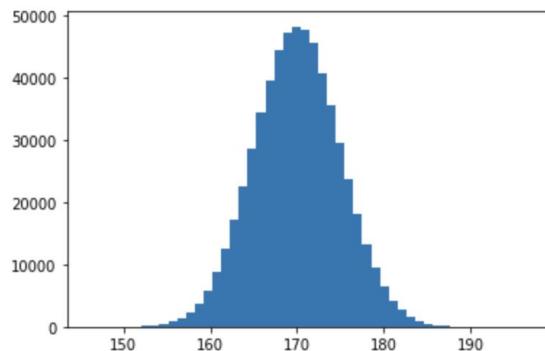
number = 600000
result = [height(150, 190) for i in range(number)]
```

```
import matplotlib.pyplot as plt
plt.hist(result, bins=50)
plt.show()
```

질문 1) 현재의 height() 함수는 키를
생성하기에 괜찮은 함수인가?



사람
확률변수: 키



```
import numpy as np

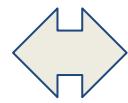
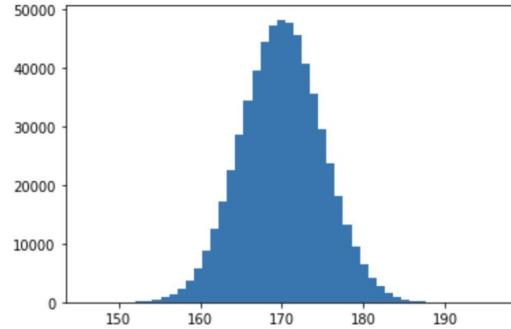
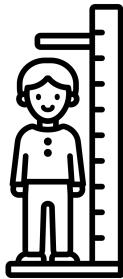
def height(mean, std):
    return mean + std * np.random.randn()

number = 600000
result = [height(170, 5) for i in range(number)]
```

```
import matplotlib.pyplot as plt
plt.hist(result, bins=50)
plt.show()
```

random한 무언가는 정규분포를 이용하면 꽤
그럴듯한 함수가 된다.
질문 2) 그 이유를 설명할 수 있는가?

통계

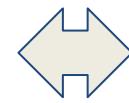


$$\mu = 170 \sigma = 5$$

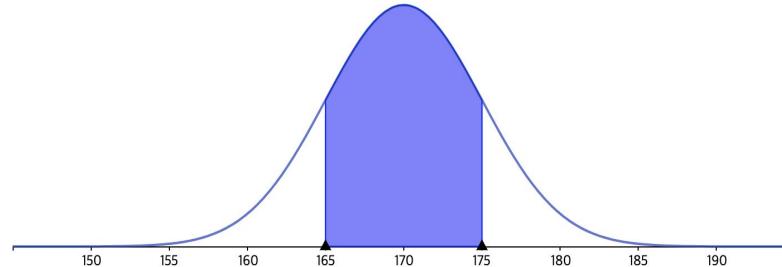
A screenshot of a Microsoft Excel spreadsheet titled "Pandas_iris_Data.xls". The table contains the following data:

Stock Name	Symbol	Name	Purchase Price	Cost Basis	Current Price	Market Value	Total Value	Dividend Yield
Apple	AAPL	Apple Inc.	\$265.00	\$265.00	\$348.13	\$14,313.27	\$14,313.27	3.18%
Amazon	AMZN	Amazon.com, Inc.	\$175.00	\$175.00	\$281.00	\$12,385.30	\$12,385.30	0.69%
Microsoft	MSFT	Microsoft Corporation	\$255.00	\$255.00	\$291.00	\$12,385.30	\$12,385.30	1.14%
Samsung	SSNLF	Samsung Electronics America, Inc.	\$250.00	\$250.00	\$278.00	\$12,385.30	\$12,385.30	0.69%
Facebook	FB	Facebook, Inc.	\$265.00	\$265.00	\$348.13	\$14,313.27	\$14,313.27	3.18%
Goldman Sachs	GS	Goldman Sachs & Co. LLC	\$265.00	\$265.00	\$348.13	\$14,313.27	\$14,313.27	3.18%
Intel	INTC	Intel Corporation	\$265.00	\$265.00	\$348.13	\$14,313.27	\$14,313.27	3.18%
Johnson & Johnson	JNJ	Johnson & Johnson	\$265.00	\$265.00	\$348.13	\$14,313.27	\$14,313.27	3.18%
Qualcomm	QCOM	Qualcomm Incorporated	\$265.00	\$265.00	\$348.13	\$14,313.27	\$14,313.27	3.18%
Verizon	VZ	Verizon Communications Inc.	\$265.00	\$265.00	\$348.13	\$14,313.27	\$14,313.27	3.18%
Twitter	TWTR	Twitter, Inc.	\$265.00	\$265.00	\$348.13	\$14,313.27	\$14,313.27	3.18%

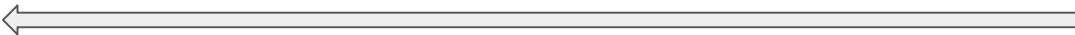
Data



Model

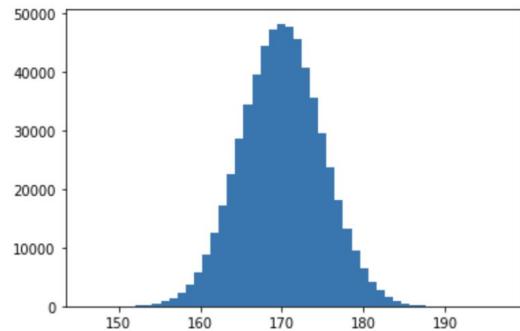
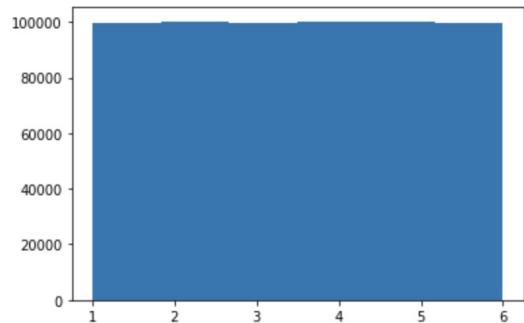


학률

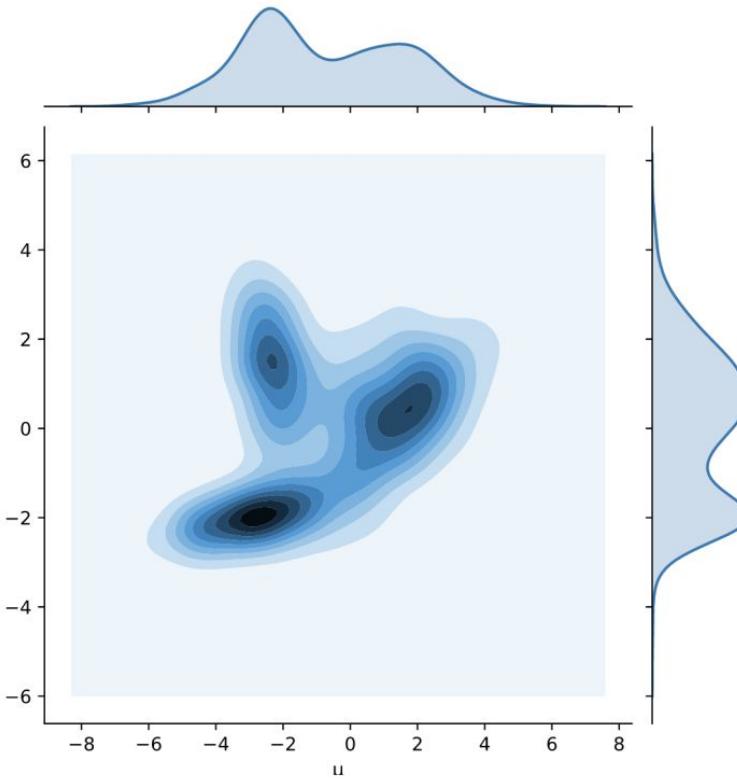


$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

확률분포라는 것은 무엇인가?

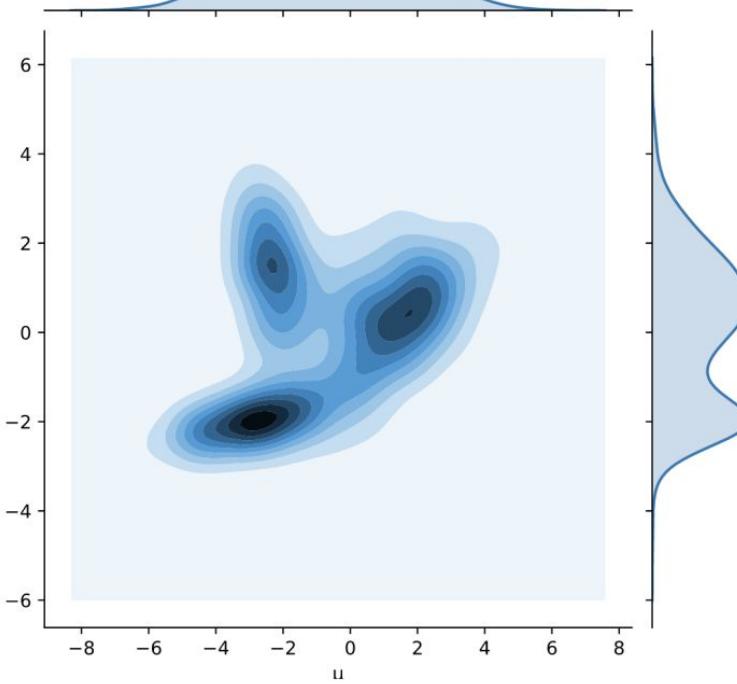
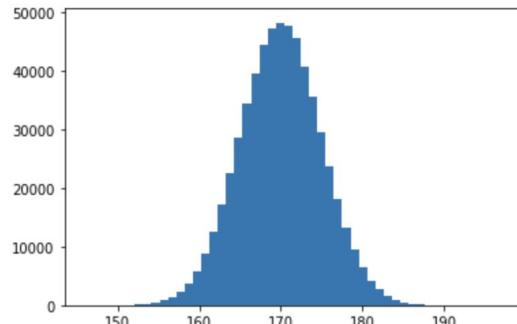
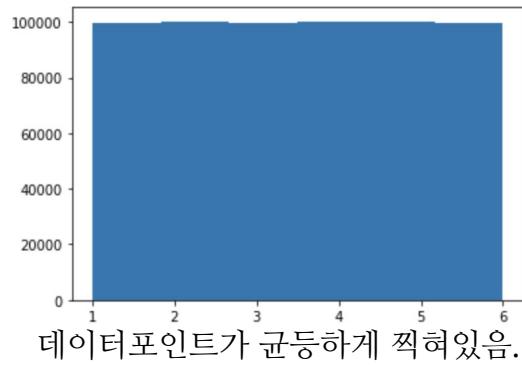


변수가 한 개 – 분포



변수가 두 개 – 분포

확률분포 => 데이터 포인트가 어디서 등장하는지에 대한 정보!!!

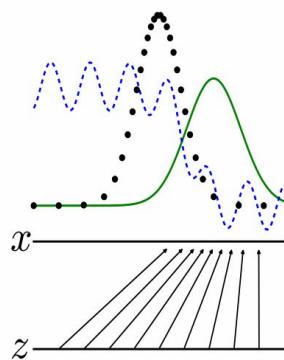


진하게 표시된 곳에 데이터포인트가
집중되어 찍혀있음.

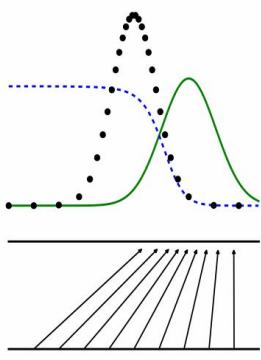
정리 - 확률변수, 확률분포

- 확률 변수
 - 확률적으로 값이 결정되는 변수 (예: 주사위의 눈, 사람의 키)
- 확률 분포
 - 확률변수가 특정한 값을 가질 확률을 알려주는 함수
 - 데이터포인트가 어디에 주로 찍히는지에 대한 정보

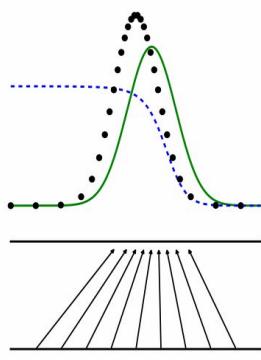
GAN의 동작 원리



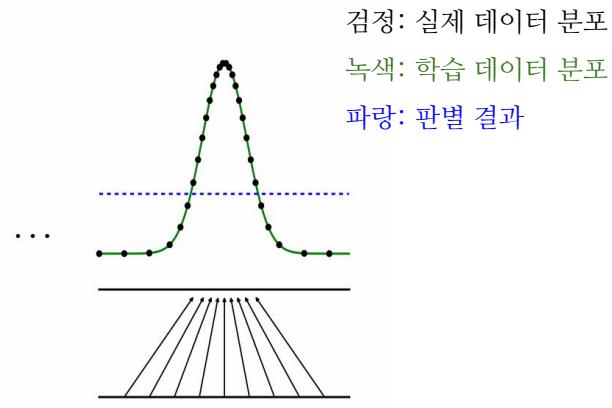
(a)



(b)



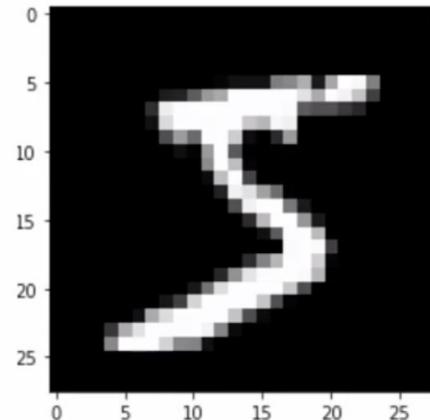
(c)



(d)

- A. 처음에는 생성 **분포**도 실제 분포와 다르고, **판별기**도 엉망이다.
- B. **판별기**가 학습하면서 실제 분포의 데이터와 **생성 분포**를 구분한다.
- C. 생성 **분포**가 실제 분포와 비슷한 형태로 구성되기 시작한다.
- D. 생성 **분포**가 실제 분포와 비슷한 형태를 갖게 되면 **판별기**는 구분을 못하므로 $\frac{1}{2}$ 확률로 찍기를 한다.

MNIST 한 장은 784차원 공간의 한 점



개별 Mnist - (28, 28)

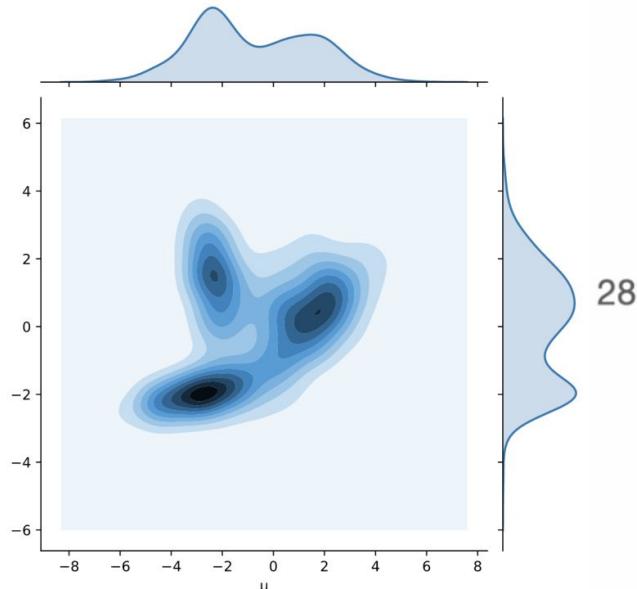
Mnist 셋: (60000, 28, 28)

28

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

784개의 숫자 (= 28 * 28)

MNIST 이미지는 확률변수가 784개

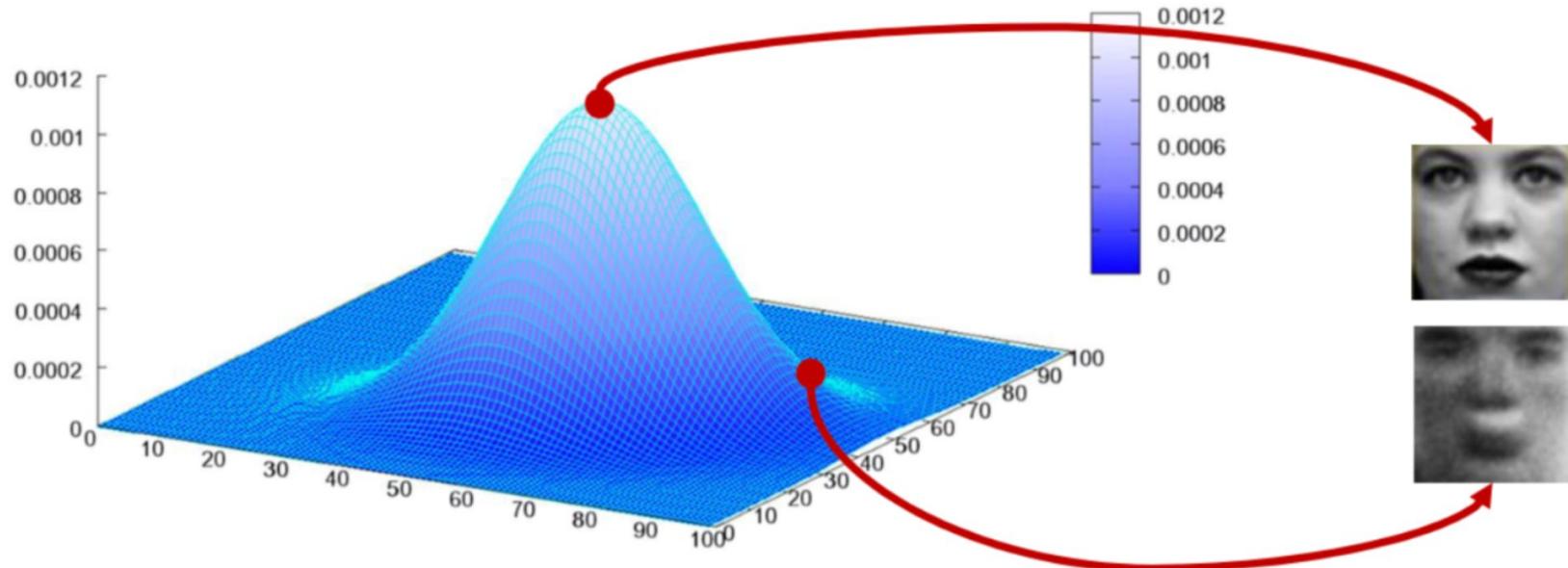


확률변수 2개



이미지의 **분포 함수**를 알고 있다면,

그 함수를 통해 뽑은 데이터는 그럴듯한 이미지가 되겠구나!



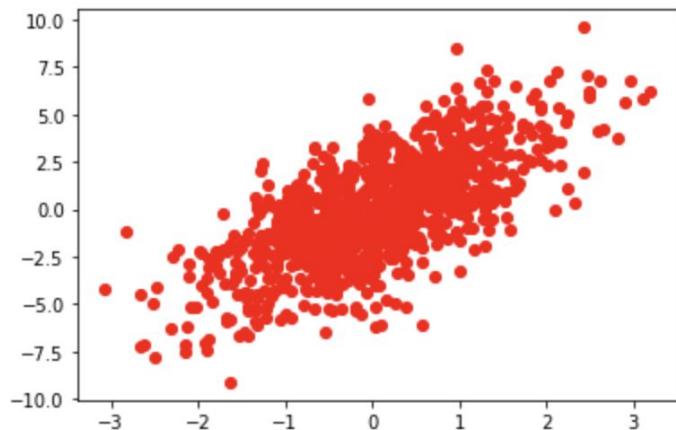
작은 차원의 **분포** 함수를 학습해보자.

```
# 데이터 준비
x_train = np.random.randn(120000).reshape(60000, 2)
print(x_train.shape)

# 분포 변형
x_train[:, 1] = 2 * x_train[:, 0] + 2 * x_train[:, 1]

plt.scatter(x_train[:1000, 0], x_train[:1000, 1], color='red')
plt.show()
```

(60000, 2)



작은 차원의 분포 함수를 학습해보자.

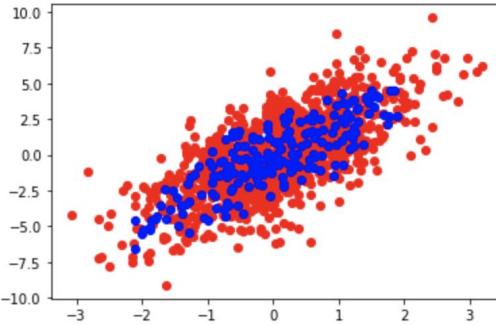
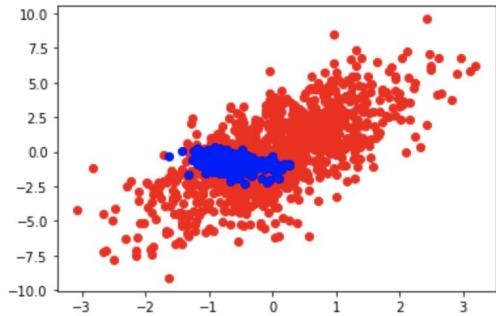
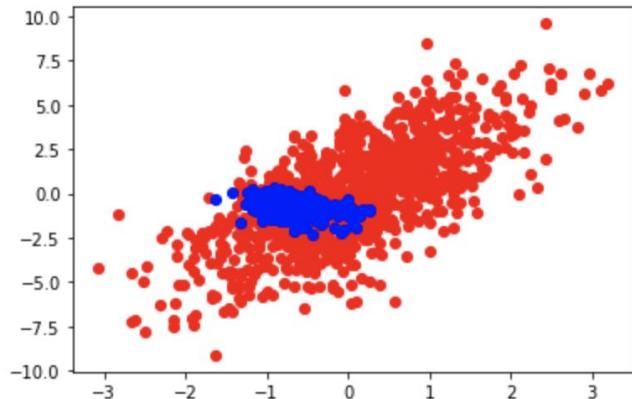
```
class Monitor(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        plt.scatter(x_train[:1000, 0], x_train[:1000, 1], color='red')
        x_gen = generator.predict(np.random.rand(1000).reshape(200, 5))
        plt.scatter(x_gen[:, 0], x_gen[:, 1], color='blue')
        plt.show()
```

학습

```
gan.fit(x_train, epochs=20, batch_size=100, callbacks=[Monitor()])
```

Epoch 1/20

```
600/600 [=====] - ETA: 0s - d_acc: 0.8088 - g_acc:
```



실습. VanillaGAN

작은 차원의 데이터 분포

복사해서 붙여넣기는 금지.

처음부터 끝까지 눈으로 따라가면서 직접 작성해봅시다.

How?

어떻게 학습되는 걸까?

수식의 의미도 살짝 음미해 봅시다.

Loss Function

- MSE (Mean Square Error)

$$loss_i = (y_i - \hat{y}_i)^2$$

- Cross Entropy

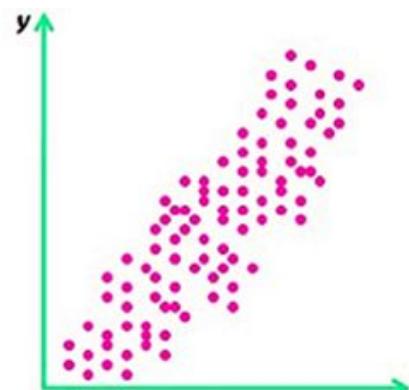
$$error = - \sum_i^n y_i \log \hat{y}_i \quad loss_i = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

- activation function – softmax

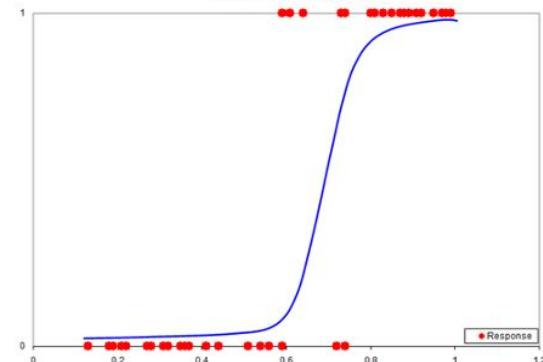
$$softmax(x_i) = \frac{e^{x_i}}{e^{x_1} + e^{x_2} + \dots + e^{x_n}} = \frac{e^{x_i}}{\sum_c^n e^{x_c}}$$

선형회귀와 로지스틱회귀

- 최적의 그래프를 어떻게 찾을까?
 - 그래프와 각 데이터의 **오차**를 구한다.
 - 오차**를 모두 더한다.
 - 오차**의 합이 최소가 되게 한다.

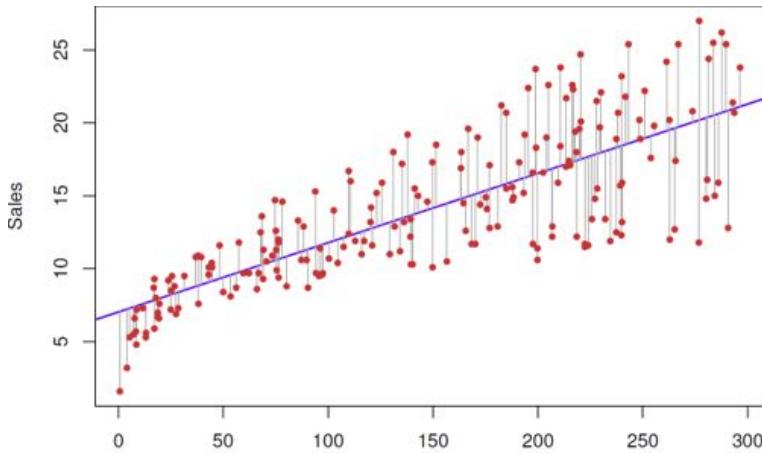


$$y = w_1 x_1 + \hat{w}_0$$



$$y = \text{class}(w_1 x_1 + w_0)$$

선형회귀와 최소제곱추정



분산의 수식과 비교해보자.

$$s^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}$$

$$\hat{y} = f_w(x_i)$$

$$loss_i = (y_i - \hat{y}_i)^2$$

$$sum\ of\ loss = \sum_i^n (y_i - \hat{y}_i)^2$$

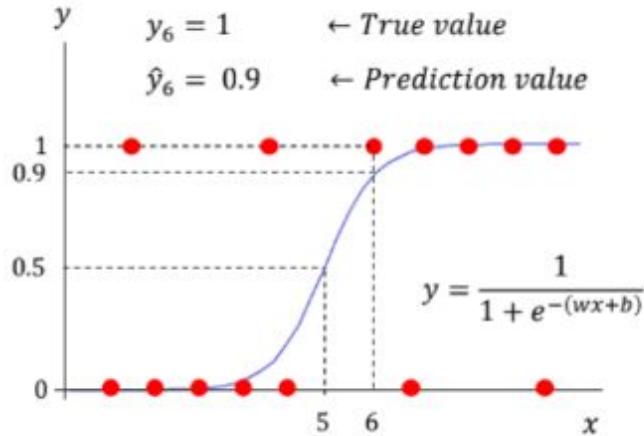
$$\operatorname{argmin}_w \sum_i^n (y_i - f_w(x_i))^2$$

```
1 # 모델 생성 or 선택
```

```
2 model = tf.keras.models.Sequential()
3 model.add(tf.keras.layers.Input(shape=(2, )))
4 model.add(tf.keras.layers.Dense(1))
5 model.compile(
6     optimizer='adam',
7     loss='mse'
8 )
```

로지스틱회귀와 크로스 엔트로피

$$\hat{y} = f_w(x_i)$$

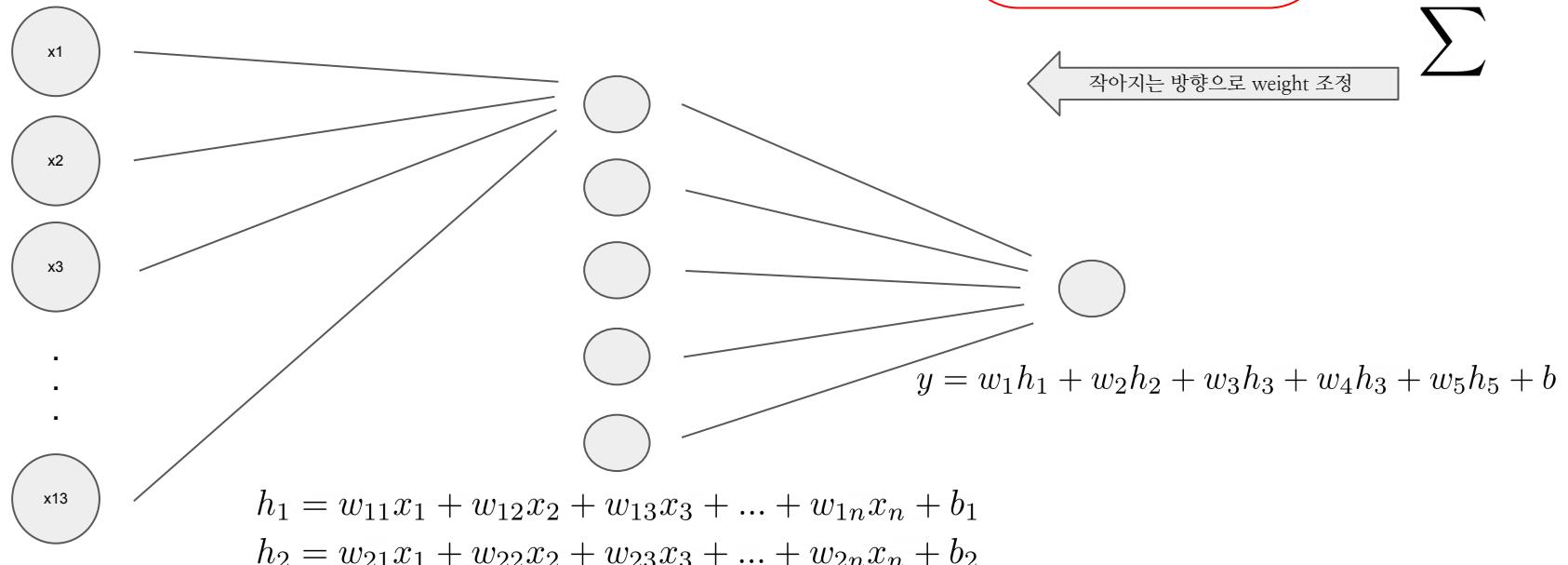
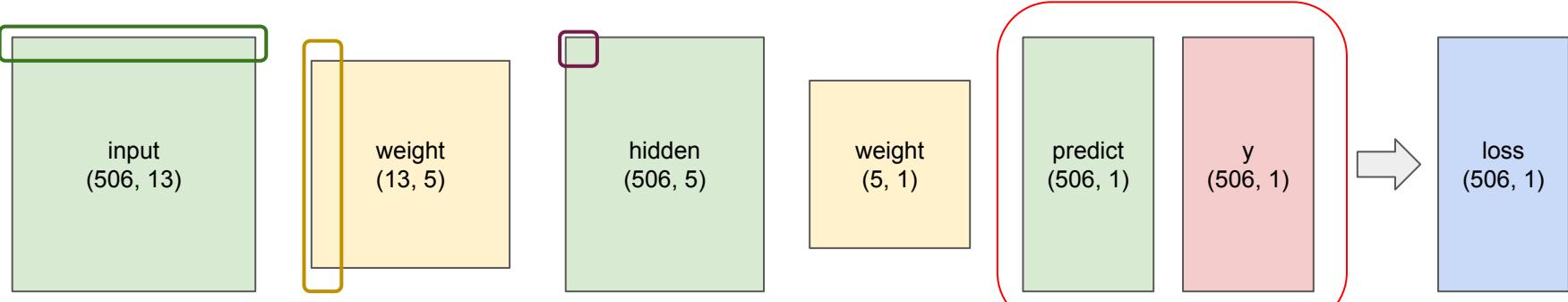


$$E(S) = \sum_{i=1}^c -p_i \log_2 p_i$$

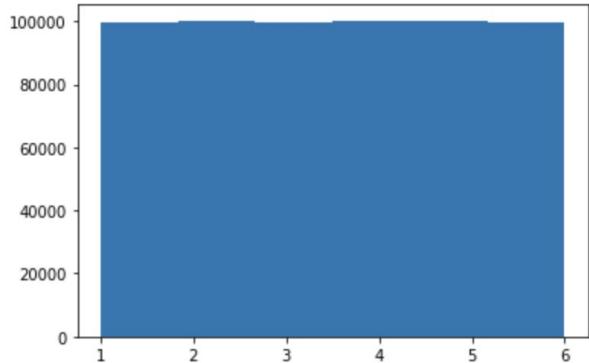
$$\text{Loss} = - \sum y_i \cdot \log \hat{y}_i$$

$$\begin{aligned} loss_i &= -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \\ \operatorname{argmin}_w \sum_i loss_i \end{aligned}$$

```
1 # 모델 생성 or 선택
2 model = tf.keras.models.Sequential()
3 model.add(tf.keras.layers.Input(shape=(2, )))
4 model.add(tf.keras.layers.Dense(1))
5 model.add(tf.keras.layers.Activation("sigmoid"))
6 model.compile(
7     optimizer='adam',
8     loss='binary_crossentropy',
9     metrics=['accuracy'])
10 )
```



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

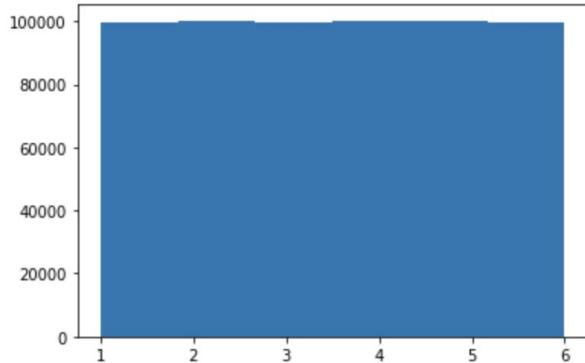


$$\begin{aligned}\text{주사위 기대값} &= 1 * \frac{1}{6} + 2 * \frac{1}{6} + 3 * \frac{1}{6} + \\ &\quad 4 * \frac{1}{6} + 5 * \frac{1}{6} + 6 * \frac{1}{6} \\ &= 3.5\end{aligned}$$

확률변수 X가 연속형 확률변수일 때

$$E[X] = \int_{-\infty}^{\infty} xf(x)dx$$

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



주사위 기대값 = $1 * \frac{1}{6} + 2 * \frac{1}{6} + 3 * \frac{1}{6} +$
 $4 * \frac{1}{6} + 5 * \frac{1}{6} + 6 * \frac{1}{6}$
 $= 3.5$

확률변수 X 가 연속형 확률변수일 때

$$E[X] = \int_{-\infty}^{\infty} xf(x)dx$$

$$\begin{aligned} V(G, D) &= \int_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \log(D(\mathbf{x})) d\mathbf{x} + \int_z p_{\mathbf{z}}(\mathbf{z}) \log(1 - D(g(\mathbf{z}))) dz \\ &= \int_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \log(D(\mathbf{x})) + p_g(\mathbf{x}) \log(1 - D(\mathbf{x})) d\mathbf{x} \end{aligned}$$

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

$$\begin{aligned} V(G, D) &= \int_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \log(D(\mathbf{x})) dx + \int_z p_{\mathbf{z}}(\mathbf{z}) \log(1 - D(g(\mathbf{z}))) dz \\ &= \int_{\mathbf{x}} \boxed{p_{\text{data}}(\mathbf{x}) \log(D(\mathbf{x})) + p_g(\mathbf{x}) \log(1 - D(\mathbf{x}))} dx \end{aligned}$$

- ▶ $y = a \log y + b \log(1 - y)$ 형태
- ▶ $a, b \in R^2$ 와 $y \in [0,1]$ 에 대해 최댓값은 $\frac{a}{a+b}$

$$D_G^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}$$

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

$$D_G^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}$$

$$C(G) = \max_D V(G, D)$$

$$= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D_G^*(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}} [\log(1 - D_G^*(G(\mathbf{z})))]$$

$$= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D_G^*(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim p_g} [\log(1 - D_G^*(\mathbf{x}))]$$

$$= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(\mathbf{x})}{P_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} \right] + \mathbb{E}_{\mathbf{x} \sim p_g} \left[\log \frac{p_g(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} \right]$$

$$C(G) = \max_D V(G, D)$$

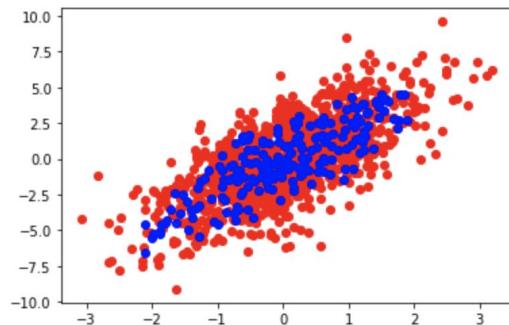
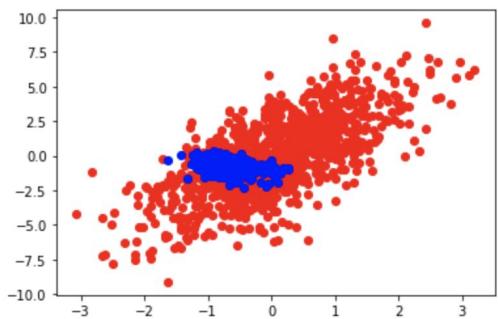
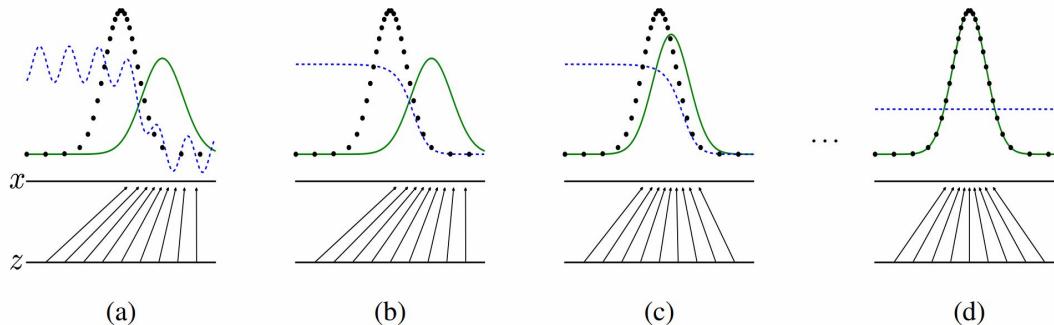
$$= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(\mathbf{x})}{P_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} \right] + \mathbb{E}_{\mathbf{x} \sim p_g} \left[\log \frac{p_g(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} \right]$$

$$= -\log(4) + KL \left(p_{\text{data}} \middle\| \frac{p_{\text{data}} + p_g}{2} \right) + KL \left(p_g \middle\| \frac{p_{\text{data}} + p_g}{2} \right)$$

$$= -\log(4) + 2 \cdot JSD(p_{\text{data}} \| p_g)$$

C(G)를 최대화 하는 것 = JSD를 최대화 하는 것

분포의 차이(거리)가 중요했다.



분포의 차이를 재는 주요한 도구

- KL Divergence
- JS Divergence

분포의 차이를 재는 주요한 도구

- Kullback–Liebler Divergence

- 상대적인 엔트로피 차이
- 비대칭적
- 분포가 같을 때, 0

$$D_{KL} = E[-\log q(x)] - E[-\log p(x)]$$

$$D_{KL}(p\|q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx$$

- Jenson–Shannon Divergence

- $[0, 1]$ 사이의 거리로 한정됨
- 대칭적 (거리 개념)

$$D_{JS}(p\|q) = \frac{1}{2} D_{KL}(p\| \frac{p+q}{2}) + \frac{1}{2} D_{KL}(q\| \frac{p+q}{2})$$



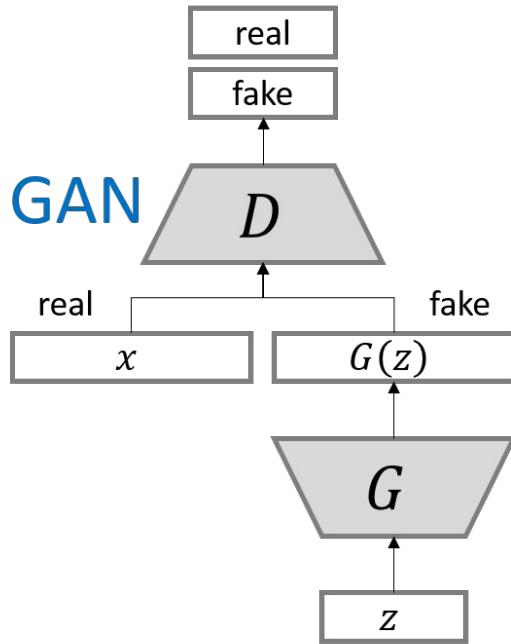
쉬어갑시다.

뇌가 뽑혀 나갈듯한 고통

Vanilla GAN 의미와 한계

- 의미
 - 마르코프 체인이 전혀 필요 없이 backprop만으로 학습이 된다.
 - 특별히 어떤 추론(inference)도 필요 없다.
 - 다양한 함수들이 모델에 접목될 수 있다.
 - 마르코프 체인을 썼을 때에 비해 훨씬 선명한(sharp) 이미지를 결과로 얻을 수 있다.
- 한계
 - 학습이 불안정하고, 고해상도 이미지는 생성하지 못하는 한계
 - imbalance: D, G의 학습이 균형적으로 이루어져야 한다.
 - Mode Collapsing: G가 특정 개체만 생성해 버리는 상태에 돌입할 수 있다.
 - Oscillation: D, G의 학습이 진동하는 경우가 생긴다.

GAN의 세계 입성 완료



```

class VanillaGAN(tf.keras.Model):

    def train_step(self, real):
        noise = tf.random.normal([tf.shape(real)[0], 100])

        with tf.GradientTape() as g_tape, tf.GradientTape() as d_tape:
            fake = self.generator(noise)

            y_real = self.discriminator(real)
            y_fake = self.discriminator(fake)

            g_loss = self.generator_loss(y_fake)
            d_loss = self.discriminator_loss(y_real, y_fake)

            g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)
            d_grads = d_tape.gradient(d_loss, self.discriminator.trainable_variables)

            self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))
            self.d_optimizer.apply_gradients(zip(d_grads, self.discriminator.trainable_variables))

            self.update_metrics(y_real, y_fake, g_loss, d_loss)
        return {
            "d_acc": self.d_acc_metric.result(),
            "g_acc": self.g_acc_metric.result(),
            "d_loss": self.d_loss_metric.result(),
            "g_loss": self.g_loss_metric.result(),
        }
    
```

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



실습. 다음 데이터셋으로 Vanilla GAN을 구성해보자.

Fashion Mnist / Cifar10

복사해서 붙여넣기는 금지.

처음부터 끝까지 눈으로 따라가면서 직접 작성해봅시다.

Deep Convolutional GAN (ICLR 2016)

<https://arxiv.org/pdf/1511.06434.pdf>

DCGAN - Discriminator

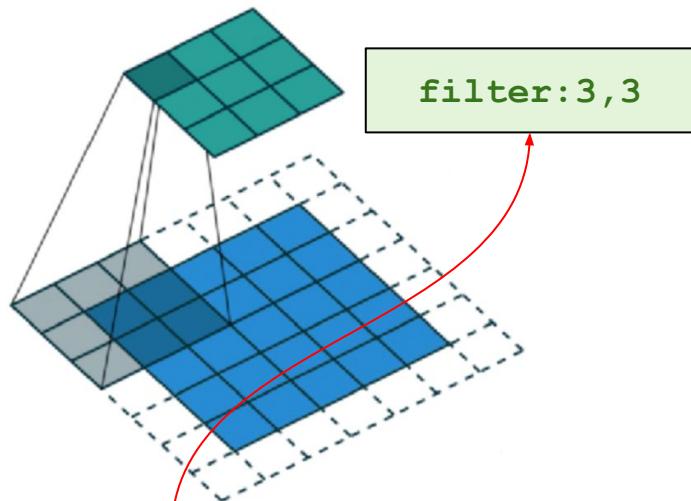
```
# 판별기
def make_discriminator():
    dx = tf.keras.Input([28, 28, 1])
    dh = tf.keras.layers.Conv2D(64, 5, 2, padding='same')(dx)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Conv2D(128, 5, 2, padding='same')(dh)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Flatten()(dh)
    dy = tf.keras.layers.Dense(1, activation='sigmoid')(dh)
    return tf.keras.Model(dx, dy, name='discriminator')
```

DCGAN - Generator

```
# 생성기
def make_generator():
    gx = tf.keras.Input([100])
    gh = tf.keras.layers.Dense(7 * 7 * 128)(gx)
    gh = tf.keras.layers.Reshape([7, 7, 128])(gh)
    gh = tf.keras.layers.Conv2DTranspose(128, 5, 2, padding='same')(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gh = tf.keras.layers.Conv2DTranspose(64, 5, 2, padding='same')(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gy = tf.keras.layers.Conv2D(1, 7, activation='tanh', padding='same')(gh)
    return tf.keras.Model(gx, gy, name='generator')
```

Conv2D vs Conv2DTranspose

판별자(Discriminator)

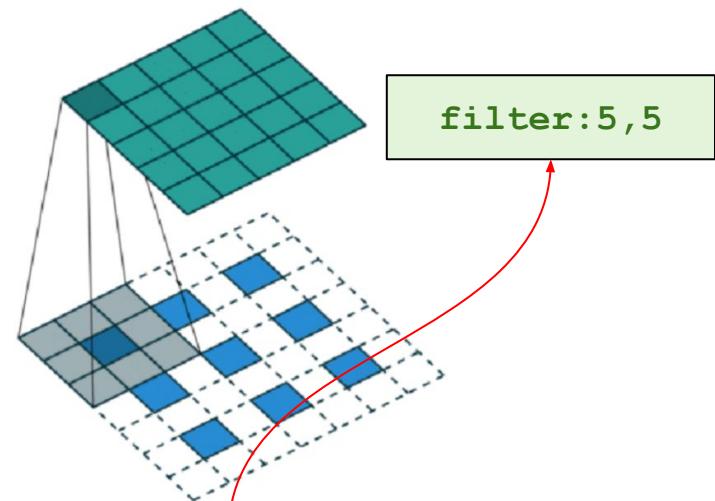


Conv2D (64, 3, 2, padding='same')

출력 특징맵 사이즈가 절반으로 줄어든다.

stride=2

생성자(Generator)



Conv2DTranspose (64, 5, 2, padding='same')

출력 특징맵 사이즈가 두 배로 커진다.

stride=2

DCGAN 학습 및 모니터링

```
class Monitor(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        x_rnd = np.random.randn(5 * 100).reshape(5, 100)
        generated = self.model.generator(x_rnd)
        for i in range(5):
            plt.subplot(1, 5, 1 + i)
            plt.axis('off')
            plt.imshow(generated[i].numpy().reshape(28, 28), cmap='gray_r')
        plt.show()
```

```
tf.keras.backend.clear_session()

discriminator = make_discriminator()
generator = make_generator()
gan = DCGAN(generator, discriminator)
gan.fit(x_train, epochs=50, batch_size=128, callbacks=[Monitor()])
```

callback을 이용하여 epoch마다 generator로 생성하는 이미지를 출력해준다.

DCGAN Model - init

```
class DCGAN(tf.keras.Model):
    def __init__(self, generator, discriminator):
        super(DCGAN, self).__init__()

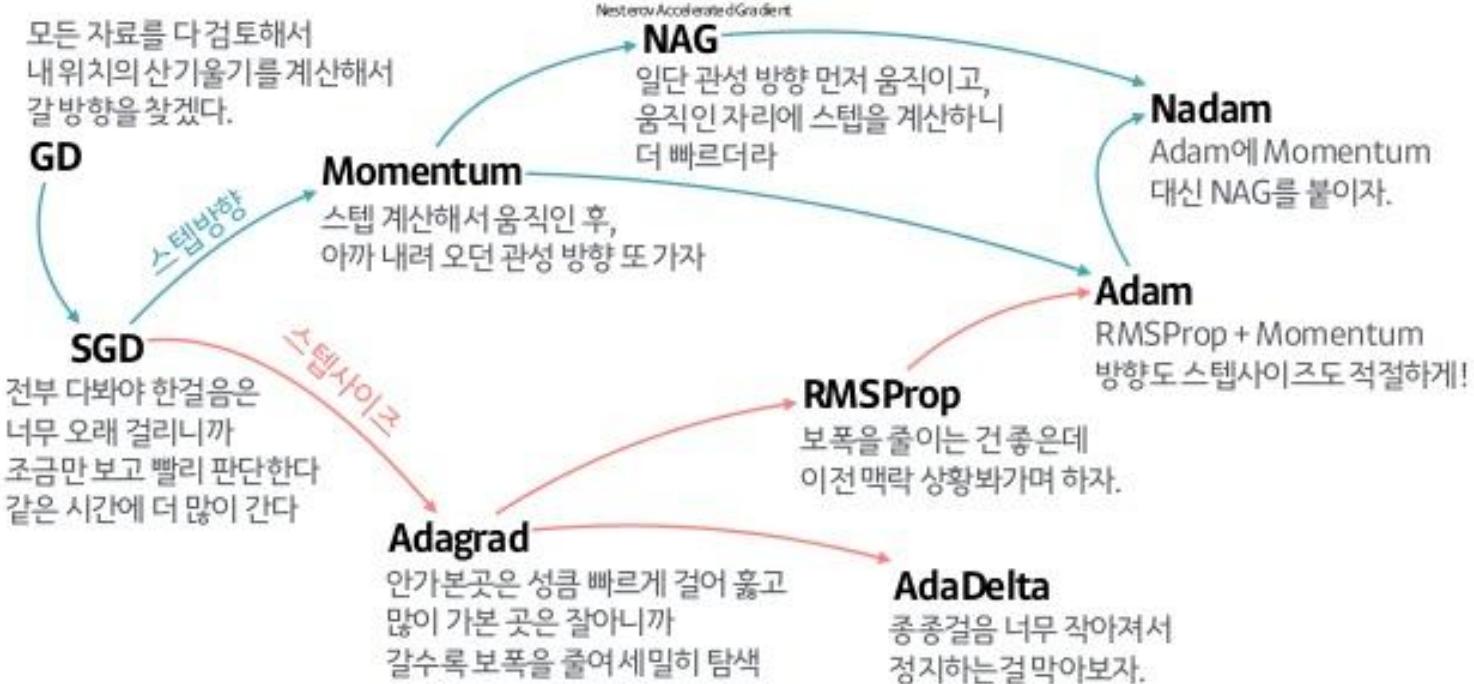
        self.generator = generator
        self.discriminator = discriminator

        self.d_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
        self.g_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)

        self.d_loss_metric = tf.keras.metrics.Mean(name="d_loss")
        self.d_acc_metric = tf.keras.metrics.BinaryAccuracy(name="d_acc")
        self.g_loss_metric = tf.keras.metrics.Mean(name="g_loss")
        self.g_acc_metric = tf.keras.metrics.BinaryAccuracy(name="g_acc")

        self.compile()
```

산내려오는 작은 오솔길 잘찾기(Optimizer)의 발달 계보

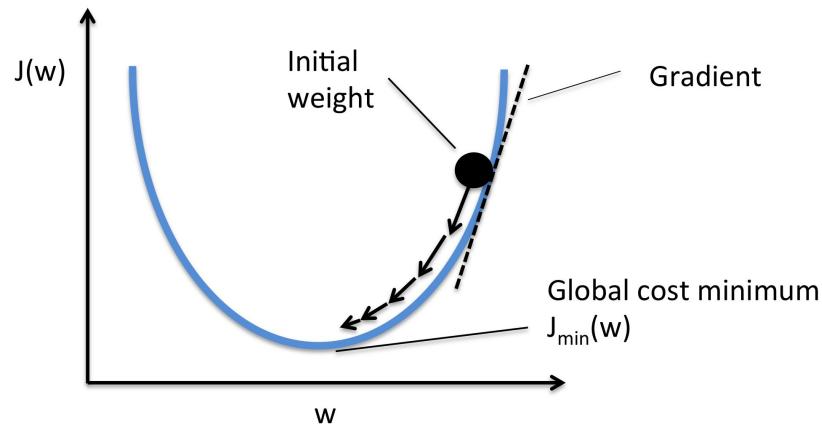


Gradient Descent (경사하강법)

함수의 기울기(경사)를 구하여 함수의 극값에 이를 때까지 기울기가 낮은 쪽으로 반복하여 이동하는 방법.

- 이걸 배치 단위로 하면
=> Stochastic GD (SGD)

$$W(t + 1) = W(t) - \alpha \frac{\partial}{\partial w} Cost(w)$$



(수식의 해석)

다음 가중치 =

현재 가중치에서

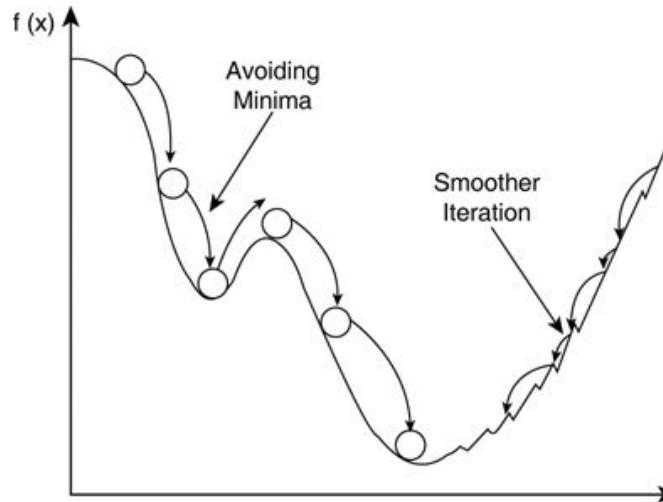
가중치에 따라 Cost(Loss)가 변화되는 양에

학습 비율을 곱한 만큼 뺀 값

Momentum (관성)

이전에 이동했던 방향을 기억해서 다음 이동의 방향에 반영.

$$V(t) = m * V(t - 1) - \alpha \frac{\partial}{\partial w} Cost(w)$$
$$W(t + 1) = W(t) + V(t)$$



(수식의 해석)

다음 가중치 = 현재 가중치랑 이번 변경값을 합한다.

단, 이번 변경값은 GD의 미분 계산량에서 지난번
변경값을 추가로 반영하면 돼

Adagrad (Adaptive Gradient)

많이 이동한 변수(w)는 최적값에 근접했을 것이라는 가정하에, 많이 이동한 변수(w)를 기억해서 다음 이동의 거리를 줄인다.

$$G(t) = G(t - 1) + \left(\frac{\partial}{\partial w(t)} Cost(w(t)) \right)^2$$

$$= \sum_{i=0}^t \left(\frac{\partial}{\partial w(i)} Cost(w(i)) \right)^2$$

$$W(t + 1) = W(t) - \alpha * \frac{1}{\sqrt{G(t) + \epsilon}} * \left(\frac{\partial}{\partial w(i)} Cost(w(i)) \right)$$

- RMSprop – 한동안 이동하지 않았으면 다시 많이 이동하자.

(수식의 해석)

다음 가중치 = 현재 가중치에서 **GD의 미분 계산량**을 뺀 때, **현재까지 해당 가중치를 변경한 양의 합 (제곱합의 제곱근)**으로 나눈 값을 쓰자.

Adam (RMSprop + Momentum)

$$M(t) = \beta_1 M(t-1) + (1 - \beta_1) \frac{\partial}{\partial w(t)} Cost(w(t))$$

$$V(t) = \beta_2 V(t-1) + (1 - \beta_2) \left(\frac{\partial}{\partial w(i)} Cost(w(i)) \right)^2$$

$$\hat{M}(t) = \frac{M(t)}{1 - \beta_1^t} \quad \hat{V}(t) = \frac{V(t)}{1 - \beta_2^t}$$

$$W(t+1) = W(t) - \alpha * \frac{\hat{M}(t)}{\sqrt{\hat{V}(t) + \epsilon}}$$

(수식의 해석)

다음 가중치 =

Momentum과 Adagrad을 모두 반영하여
현재의 가중치에서 뺀다.

- beta_1 - Momentum 적용 비율,
0이면 Momentum 사용하지 않음.
- beta_2 - Adagrad 적용 비율, 0이면
Adagrad 사용하지 않음.

DCGAN Model

- train_step

```
class DCGAN(tf.keras.Model):
    ...
    def train_step(self, real):
        noise = tf.random.normal([tf.shape(real)[0], 100])

        with tf.GradientTape() as g_tape, tf.GradientTape() as d_tape:
            fake = self.generator(noise, training=True)

            y_real = self.discriminator(real, training=True)
            y_fake = self.discriminator(fake, training=True)

            g_loss = self.generator_loss(y_fake)
            d_loss = self.discriminator_loss(y_real, y_fake)

            g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)
            d_grads = d_tape.gradient(d_loss, self.discriminator.trainable_variables)

            self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))
            self.d_optimizer.apply_gradients(zip(d_grads, self.discriminator.trainable_variables))

            self.update_metrics(y_real, y_fake, g_loss, d_loss)
        return {
            "d_acc": self.d_acc_metric.result(),
            "g_acc": self.g_acc_metric.result(),
            "d_loss": self.d_loss_metric.result(),
            "g_loss": self.g_loss_metric.result(),
        }
```

DCGAN Model - loss

```
class DCGAN(tf.keras.Model):

    ...
    def discriminator_loss(self, y_real, y_fake):
        real_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(y_real), y_real)
        fake_loss = tf.keras.losses.binary_crossentropy(tf.zeros_like(y_fake), y_fake)
        d_loss = real_loss + fake_loss
        return d_loss

    def generator_loss(self, y_fake):
        g_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(y_fake), y_fake)
        return g_loss

    def update_metrics(self, y_real, y_fake, g_loss, d_loss):
        self.g_loss_metric.update_state(g_loss)
        self.g_acc_metric.update_state(tf.ones_like(y_fake), y_fake)
        self.d_loss_metric.update_state(d_loss)
        self.d_acc_metric.update_state(tf.ones_like(y_real), y_real)
        self.d_acc_metric.update_state(tf.zeros_like(y_fake), y_fake)
```

학습 및 모니터링

```
import matplotlib.pyplot as plt

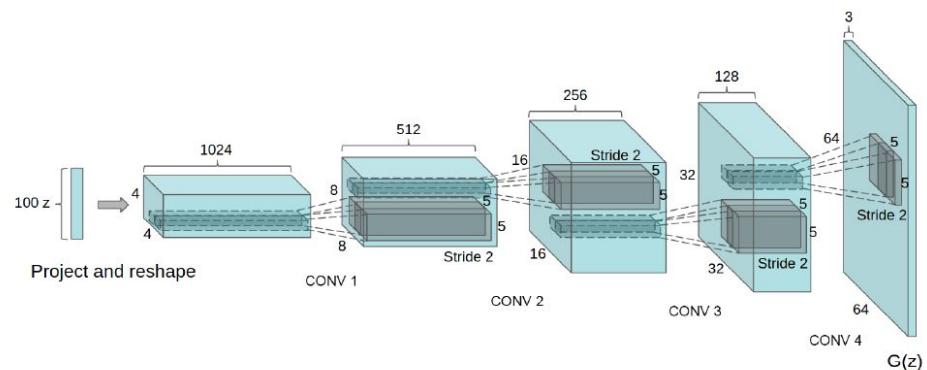
class Monitor(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        x_rnd = np.random.randn(5 * 100).reshape(5, 100)
        generated = self.model.generator(x_rnd)
        for i in range(5):
            plt.subplot(1, 5, 1 + i)
            plt.axis('off')
            plt.imshow(generated[i].numpy().reshape(28, 28), cmap='gray_r')
        plt.show()
```

```
tf.keras.backend.clear_session()

discriminator = make_discriminator()
generator = make_generator()
gan = DCGAN(generator, discriminator)
gan.fit(x_train, epochs=50, batch_size=128, callbacks=[Monitor()])
```

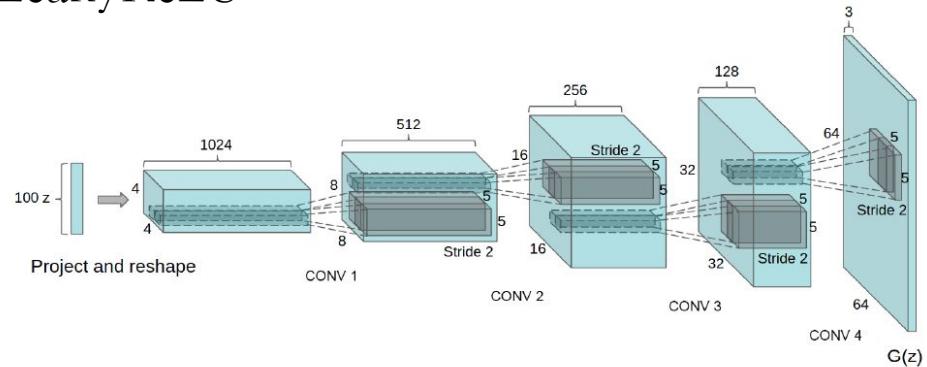
DCGAN (ICLR 2016)

- convolutional GAN 구조로 보다 안정적인 학습이 가능함.
- 필터 시각화를 통해 이미지의 특징을 잘 잡는 필터가 학습됨을 설명함.
- generator의 입력에 사용하는 노이즈 벡터가 이미지의 semantic한 특징에 대해 산술 속성을 보임.



Architecture guidelines

- pooling layer를 사용하지 않음. Convolution stride를 이용하여 대체
- generator/discriminator에 Batch Normalization 사용
- Fully-Connected layer 사용 안함.
- generator 은닉층 활성화 함수 – ReLU
- discriminator 은닉층 활성화 함수 – LeakyReLU



Generator Architecture

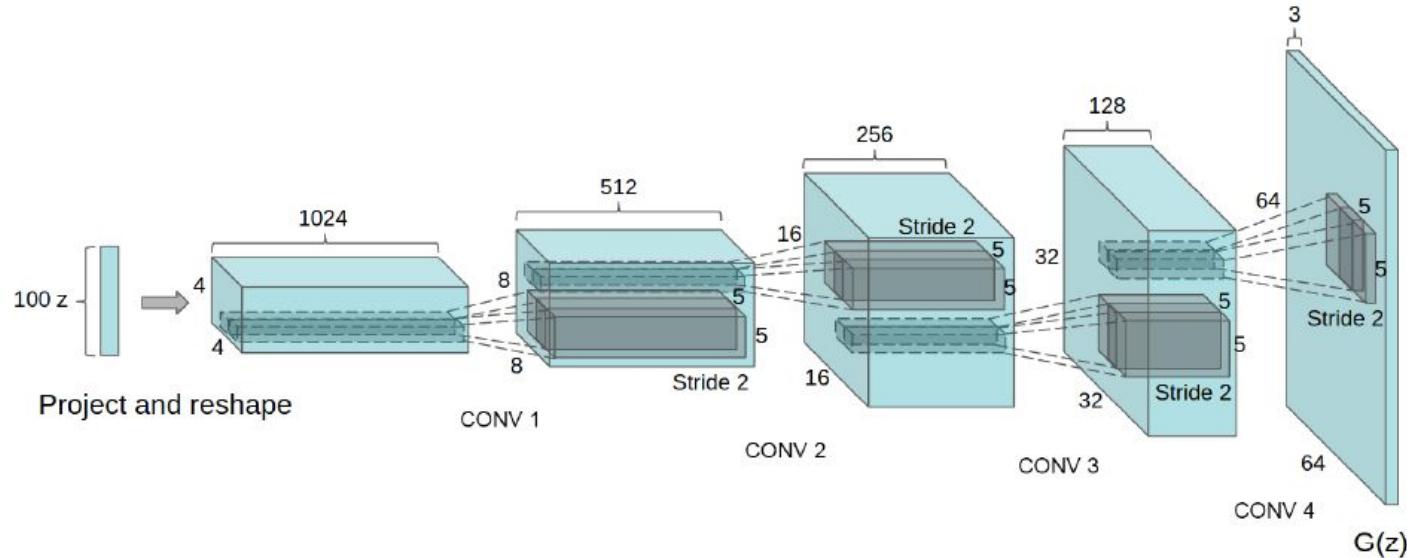


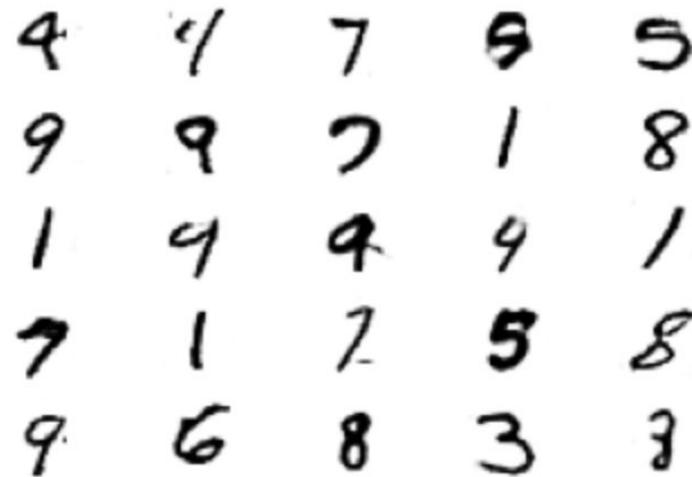
Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

결과 확인

vanilla GAN과 비교하여
훨씬 결과가 좋은 것을
확인 수 있다.

```
import matplotlib.pyplot as plt
import numpy as np

x_test = np.random.normal(size=2500).reshape(25, 100)
images = gan.generator.predict(x_test)
for i in range(25):
    plt.subplot(5, 5, 1 + i)
    plt.axis('off')
    plt.imshow(images[i].reshape(28, 28), cmap='gray_r')
```



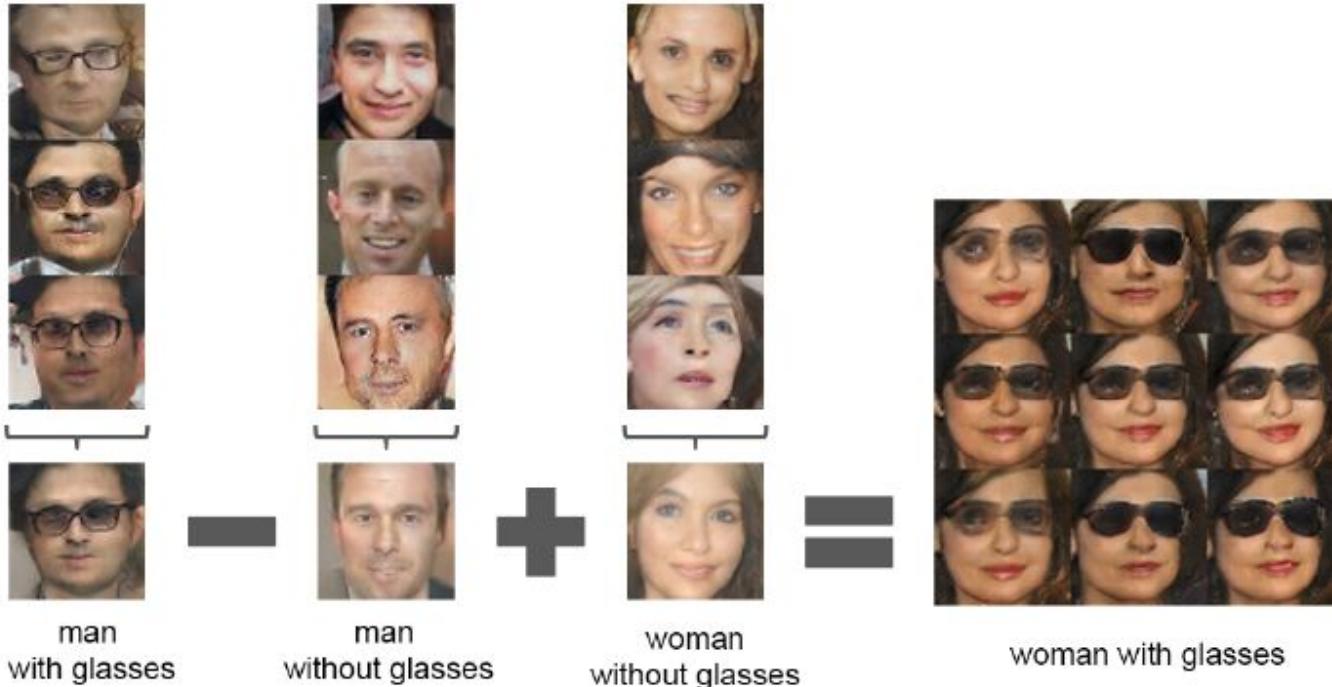
walking in the latent space

학습한 manifold 위를 이동하며 생성되는 이미지 사이에 서서히 창문이 생기고, 가구의 방향/형태가 바뀌는 등의 현상을 관찰할 수 있다. 이는 모델이 흥미로운 representation을 학습한 것을 의미한다.



Figure 4: Top rows: Interpolation between a series of 9 random points in Z show that the space learned has smooth transitions, with every image in the space plausibly looking like a bedroom. In the 6th row, you see a room without a window slowly transforming into a room with a giant window. In the 10th row, you see what appears to be a TV slowly being transformed into a window.

벡터 연산이 가능함을 보여줌



What?

벡터 연산이 가능하다는 것은
무슨 의미가 있지?

Manifold를 학습했다고 할 수 있다.

의미를 보존하는 공간, **Manifold**



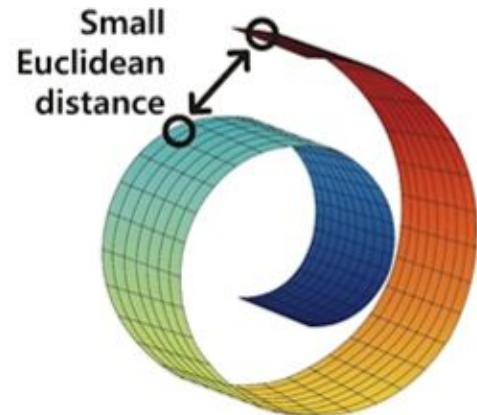
일반적으로 두 이미지를 선형
보간하는 경우의 중간 이미지

잠재 공간(latent space)의 두 벡터를
선형 보간하면 상대적으로 자연스러운
중간 이미지를 보여준다.

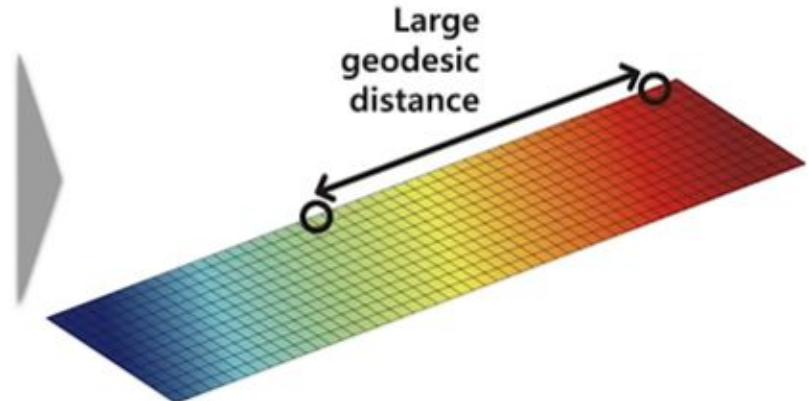
의미를 보존하는 공간, Manifold

고차원 데이터라 할지라도, 의미를 가지는 피쳐를 모아둔 조밀한 저차원 공간이 존재하며 이를 Manifold라고 한다. sparse 한 고차원 데이터를 간추려서 저차원 공간으로 나타낼수 있다는 뜻.

고차원 공간



저차원 공간



의미를 보존하는 공간, Manifold

- 고차원 공간에서 중간값 – 팔이 2개 골프채가 2개로 좌우 이미지의 중간 위치
- manifold 공간에서 중간값 – 공을 치는 중간과정 모습

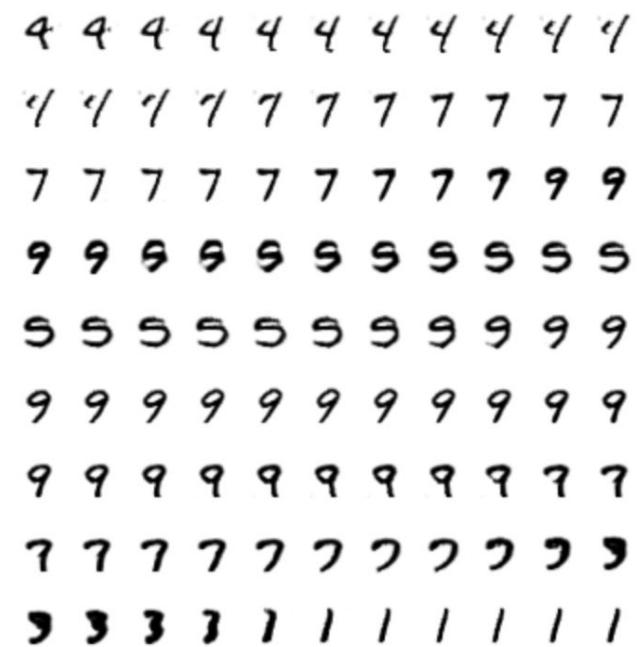


Manifold 상에서 중간 이미지 확인

```
import matplotlib.pyplot as plt
import numpy as np

vectors = []
for j in range(len(x_test) - 1):
    for i in range(11):
        vectors.append((1 - 0.1 * i) * x_test[j] +
                      (0.1 * i) * x_test[j + 1])

images = gan.generator.predict(np.array(vectors))
for j in range(len(x_test) - 1):
    for i in range(11):
        plt.subplot(1, 11, 1 + i)
        plt.axis('off')
        plt.imshow(images[10 * j + i].reshape(28, 28), cmap='gray_r')
plt.show()
```



DCGAN 의미와 한계

- 의미
 - GAN 훈련을 위한 안정적인 구조를 제시
 - 적대적 네트워크를 통해 이미지의 좋은 representation을 학습할 수 있음을 입증.
 - 흥미로운 latent space 학습
- 한계
 - 모델의 불안정성 문제는 여전함
 - 학습을 오래할수록 Mode Collapsing 등의 이상현상 발생가능성 증가함

실습. 다음 데이터셋으로 DCGAN을 구성해보자.

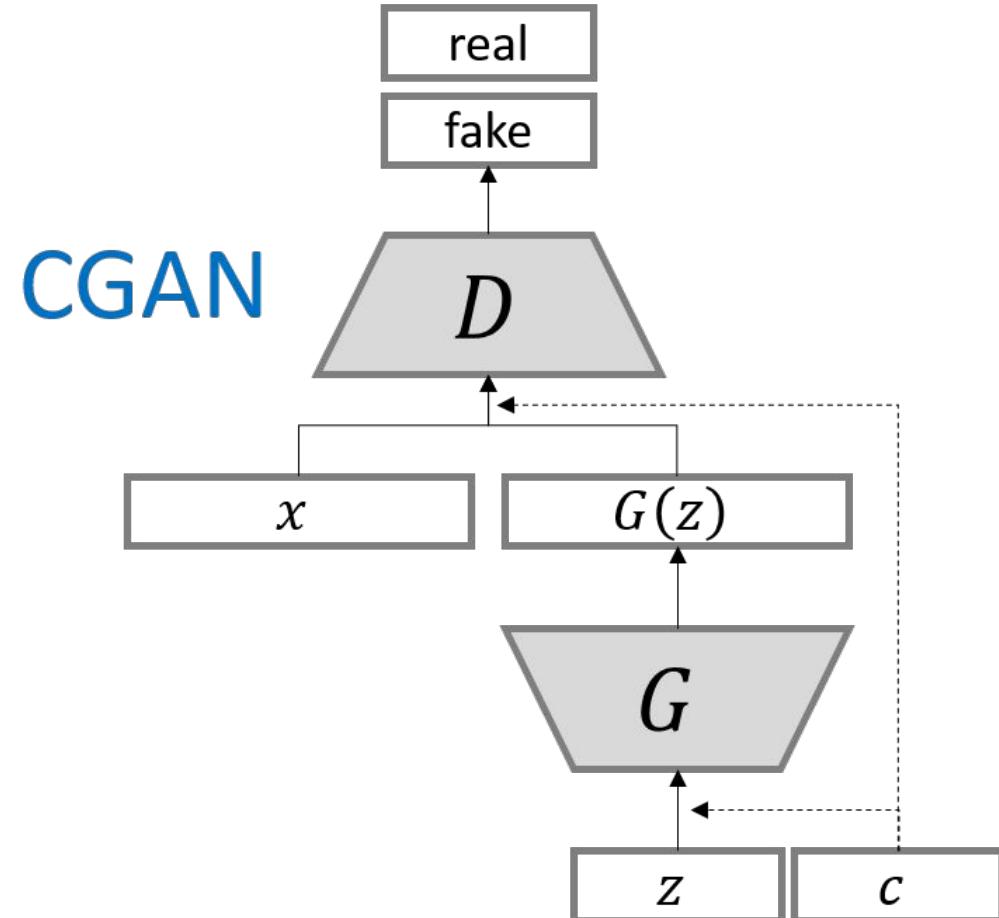
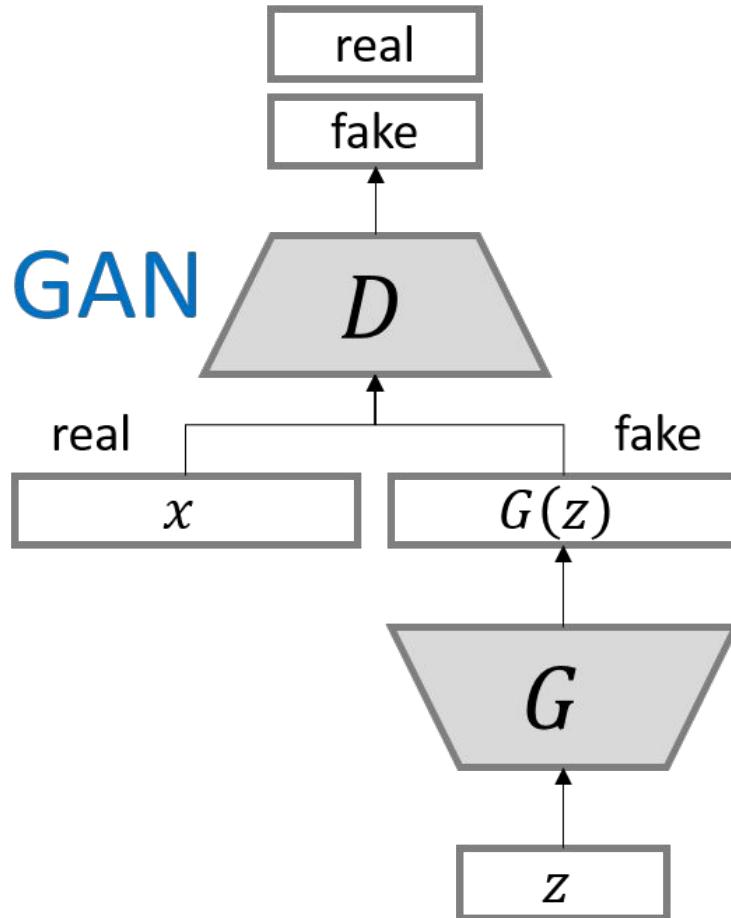
Fashion Mnist / Cifar10

복사해서 붙여넣기는 금지.

처음부터 끝까지 눈으로 따라가면서 직접 작성해봅시다.

Conditional GAN (2014)

<https://arxiv.org/pdf/1411.1784.pdf>



CGAN - Discriminator

```
def make_discriminator():
    dx1 = tf.keras.Input(shape=[28, 28, 1])
    dh1 = tf.keras.layers.Flatten()(dx1)

    dx2 = tf.keras.Input(shape=[1])
    dh2 = tf.keras.layers.Embedding(10, 28 * 28)(dx2)
    dh2 = tf.keras.layers.Flatten()(dh2)

    dh = tf.keras.layers.add([dh1, dh2])

    dh = tf.keras.layers.Reshape([28, 28, 1])(dh)
    dh = tf.keras.layers.Conv2D(64, 5, 2, padding='same')(dh)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Conv2D(128, 5, 2, padding='same')(dh)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Flatten()(dh)
    dy = tf.keras.layers.Dense(1, activation='sigmoid')(dh)
    return tf.keras.Model([dx1, dx2], dy, name='discriminator')
```

CGAN - Generator

```
def make_generator():
    gx1 = tf.keras.Input(shape=[100])

    gx2 = tf.keras.Input(shape=[1])
    gh2 = tf.keras.layers.Embedding(10, 100)(gx2)
    gh2 = tf.keras.layers.Flatten()(gh2)

    gh = tf.keras.layers.add([gx1, gh2])
    gh = tf.keras.layers.Dense(7 * 7 * 128)(gh)
    gh = tf.keras.layers.Reshape([7, 7, 128])(gh)
    gh = tf.keras.layers.Conv2DTranspose(128, 5, 2, padding='same')(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gh = tf.keras.layers.Conv2DTranspose(64, 5, 2, padding='same')(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gy = tf.keras.layers.Conv2D(1, 7, activation='tanh', padding='same')(gh)
    return tf.keras.Model([gx1, gx2], gy, name='generator')
```

CGAN 학습 및 모니터링

```
class Monitor(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        x_rnd = np.random.randn(5 * 100).reshape(5, 100)
        generated = self.model.generator(x_rnd)
        for i in range(5):
            plt.subplot(1, 5, 1 + i)
            plt.axis('off')
            plt.imshow(generated[i].numpy().reshape(28, 28), cmap='gray_r')
        plt.show()
```

```
tf.keras.backend.clear_session()

discriminator = make_discriminator()
generator = make_generator()
gan = CGAN(generator, discriminator)
gan.fit([x_train, y_train.reshape([-1, 1])],
        epochs=50, batch_size=128, callbacks=[Monitor()])
```

CGAN Model - init

```
class CGAN(tf.keras.Model):
    def __init__(self, generator, discriminator):
        super(CGAN, self).__init__()
        self.compile()

        self.generator = generator
        self.discriminator = discriminator

        self.d_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
        self.g_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)

        self.d_loss_metric = tf.keras.metrics.Mean(name="d_loss")
        self.d_acc_metric = tf.keras.metrics.BinaryAccuracy(name="d_acc")
        self.g_loss_metric = tf.keras.metrics.Mean(name="g_loss")
        self.g_acc_metric = tf.keras.metrics.BinaryAccuracy(name="g_acc")
```

CGAN Model

- train_step

```
class CGAN(tf.keras.Model):
    ...
    def train_step(self, inputs):
        real, label = inputs[0]
        noise = tf.random.normal([tf.shape(real)[0], 100])

        with tf.GradientTape() as g_tape, tf.GradientTape() as d_tape:
            fake = self.generator([noise, label], training=True)

            y_real = self.discriminator([real, label], training=True)
            y_fake = self.discriminator([fake, label], training=True)

            g_loss = self.generator_loss(y_fake)
            d_loss = self.discriminator_loss(y_real, y_fake)

            g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)
            d_grads = d_tape.gradient(d_loss, self.discriminator.trainable_variables)

            self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))
            self.d_optimizer.apply_gradients(zip(d_grads, self.discriminator.trainable_variables))

            self.update_metrics(y_real, y_fake, g_loss, d_loss)
        return {
            "d_acc": self.d_acc_metric.result(),
            "g_acc": self.g_acc_metric.result(),
            "d_loss": self.d_loss_metric.result(),
            "g_loss": self.g_loss_metric.result(),
        }
```

CGAN Model - loss

```
class CGAN(tf.keras.Model):

    ...

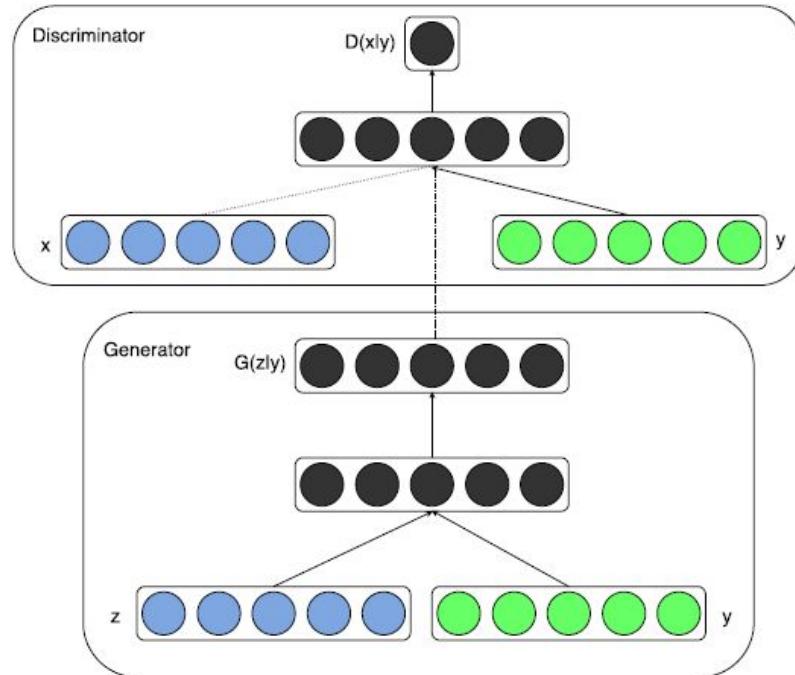
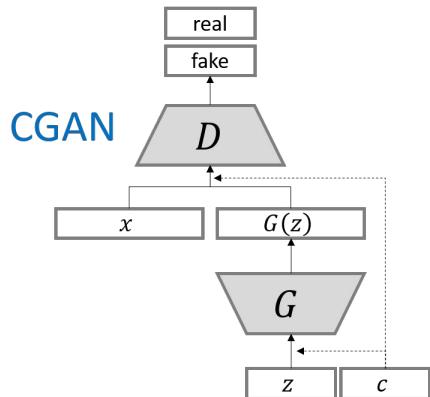
    def discriminator_loss(self, y_real, y_fake):
        real_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(y_real), y_real)
        fake_loss = tf.keras.losses.binary_crossentropy(tf.zeros_like(y_fake), y_fake)
        d_loss = real_loss + fake_loss
        return d_loss

    def generator_loss(self, y_fake):
        g_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(y_fake), y_fake)
        return g_loss

    def update_metrics(self, y_real, y_fake, g_loss, d_loss):
        self.g_loss_metric.update_state(g_loss)
        self.g_acc_metric.update_state(tf.ones_like(y_fake), y_fake)
        self.d_loss_metric.update_state(d_loss)
        self.d_acc_metric.update_state(tf.ones_like(y_real), y_real)
        self.d_acc_metric.update_state(tf.zeros_like(y_fake), y_fake)
```

Conditional GAN (2014)

- 입력 조건에 맞는 이미지를 생성하는 GAN
- discriminator와 generator에 이미지와 보조변수 Y를 함께 입력으로 사용함
- 조건부 이미지 생성은 물론 이미지태깅, multi-modal model의 학습에 이용 가능



CGAN Loss

GAN의 목적함수

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

CGAN의 목적함수

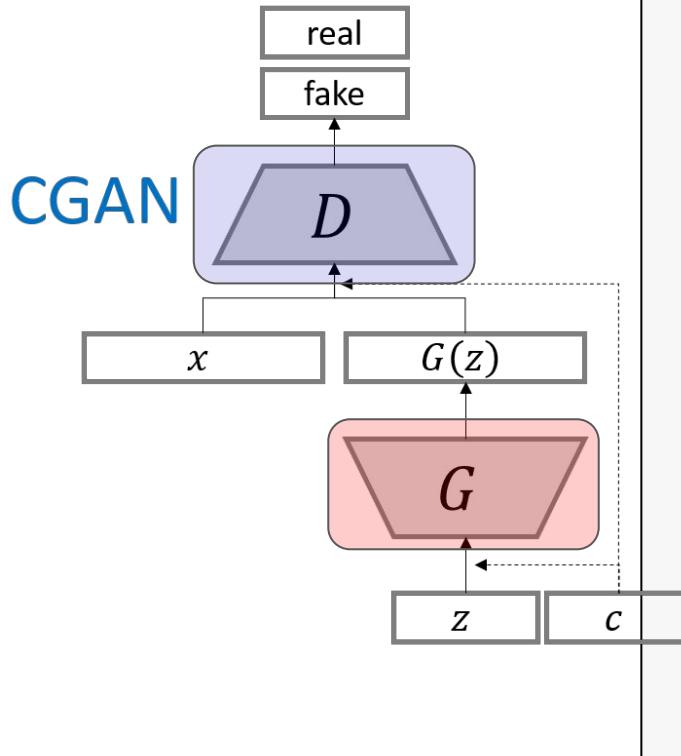
$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x}|y)] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z}|y)))]$$

일반 GAN의 목적함수에 input 변수 x, z만 조건부 변수 $x|y$, $z|y$ 로 바뀐 모습이다.

실제 이미지 판별 결과

생성 이미지 판별 결과

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



```

class CGAN(tf.keras.Model):
    ...
    def train_step(self, inputs):
        real, label = inputs[0]
        noise = tf.random.normal([tf.shape(real)[0], 100])

        with tf.GradientTape() as g_tape, tf.GradientTape() as d_tape:
            fake = self.generator([noise, label], training=True)

            y_real = self.discriminator([real, label], training=True)
            y_fake = self.discriminator([fake, label], training=True)

            g_loss = self.generator_loss(y_fake)
            d_loss = self.discriminator_loss(y_real, y_fake)

            g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)
            d_grads = d_tape.gradient(d_loss, self.discriminator.trainable_variables)

            self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))
            self.d_optimizer.apply_gradients(zip(d_grads, self.discriminator.trainable_variables))

            self.update_metrics(y_real, y_fake, g_loss, d_loss)
            return {
                "d_acc": self.d_acc_metric.result(),
                "g_acc": self.g_acc_metric.result(),
                "d_loss": self.d_loss_metric.result(),
                "g_loss": self.g_loss_metric.result(),
            }
  
```

결과확인

```
import numpy as np

x_test = np.random.normal(size=[10, 100])

for j in range(10):
    images = gan.generator.predict([x_test, np.full(10, j)])
    plt.figure(figsize=(15, 15))
    for i in range(10):
        plt.subplot(1, 10, 1 + i)
        plt.axis('off')
        plt.imshow(images[i, :, :, 0], cmap='gray_r')
    plt.show()
```

결과확인

$G(z, \text{label}: 0)$	→	0 0 0 0 0 0 0 0 0 0
$G(z, \text{label}: 1)$	→	1 1 1 1 1 1 1 1 1 1
$G(z, \text{label}: 2)$	→	2 2 2 2 2 2 2 2 2 2
$G(z, \text{label}: 3)$	→	3 3 3 3 3 3 3 3 3 3
$G(z, \text{label}: 4)$	→	4 4 4 4 4 4 4 4 4 4
$G(z, \text{label}: 5)$	→	5 5 5 5 5 5 5 5 5 5
$G(z, \text{label}: 6)$	→	6 6 6 6 6 6 6 6 6 6
$G(z, \text{label}: 7)$	→	7 7 7 7 7 7 7 7 7 7
$G(z, \text{label}: 8)$	→	8 8 8 8 8 8 8 8 8 8
$G(z, \text{label}: 9)$	→	9 9 9 9 9 9 9 9 9 9

장단점

- 장점
 - 구현이 매우 간단함.
 - 생성되는 데이터를 원하는 대로 통제할 수 있다는 부분에서 의미있는 연구.
 - 다양한 application으로의 확장가능성으로 Conditional GAN의 잠재력을 보임.
- 단점
 - 의미있는 수준의 성능을 보이지는 못함.
 - GAN의 여러 단점을 그대로 가지고 있음.
 - 논문의 결과는 매우 preliminary한 것이라 저자들도 인정하고 있음.

실습. 다음 데이터셋으로 CGAN을 구성해보자.

Fashion Mnist / Cifar10

복사해서 붙여넣기는 금지.

처음부터 끝까지 눈으로 따라가면서 직접 작성해봅시다.

Wasserstein GAN (NIPS 2017)

<https://arxiv.org/pdf/1701.07875.pdf>

WGAN - Original Critic

```
# 판별기
class WeightClip(tf.keras.constraints.Constraint):
    def __call__(self, w):
        return tf.keras.backend.clip(w, -0.01, 0.01)

def make_critic():
    dx = tf.keras.Input([28, 28, 1])
    dh = tf.keras.layers.Conv2D(64, 5, 2, padding='same', kernel_constraint=WeightClip())(dx)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Conv2D(128, 5, 2, padding='same', kernel_constraint=WeightClip())(dh)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Flatten()(dh)
    dy = tf.keras.layers.Dense(1, kernel_constraint=WeightClip())(dh)
    return tf.keras.Model(dx, dy, name='critic')
```

- 판별기의 이름이 Critic(비평가)로 달라졌다.
 - 그냥 판별이 아니라 수준에 따라 별점을 주겠다는 의미
- 판별기의 최종 dense layer에 sigmoid 함수를 사용하지 않는다.
- weight clipping을 (-0.01~0.01)로 해준다.



WGAN - Generator

```
# 생성기
def make_generator():
    gx = tf.keras.Input([100])
    gh = tf.keras.layers.Dense(7 * 7 * 128)(gx)
    gh = tf.keras.layers.Reshape([7, 7, 128])(gh)
    gh = tf.keras.layers.Conv2DTranspose(128, 5, 2, padding='same')(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gh = tf.keras.layers.Conv2DTranspose(64, 5, 2, padding='same')(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gy = tf.keras.layers.Conv2D(1, 7, activation='tanh', padding='same')(gh)
return tf.keras.Model(gx, gy, name='generator')
```

WGAN Model - init

```
class WGAN(tf.keras.Model):
    def __init__(self, generator, critic):
        super(WGAN, self).__init__()
        self.compile()

        self.generator = generator
        self.critic = critic

        self.c_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
        self.g_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)

        self.c_loss_metric = tf.keras.metrics.Mean(name="c_loss")
        self.g_loss_metric = tf.keras.metrics.Mean(name="g_loss")

        self.compile()
```

- 더이상 정확도는 계산하지 않는다.

WGAN Model

- train_step

```
class WGAN(tf.keras.Model):
    ...
    def train_step(self, real):
        noise = tf.random.normal([tf.shape(real)[0], 100])

        with tf.GradientTape() as g_tape, tf.GradientTape() as c_tape:
            fake = self.generator(noise, training=True)

            y_real = self.critic(real, training=True)
            y_fake = self.critic(fake, training=True)

            g_loss = self.generator_loss(y_fake)
            c_loss = self.critic_loss(y_real, y_fake)

            g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)
            c_grads = c_tape.gradient(c_loss, self.critic.trainable_variables)

            self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))
            self.c_optimizer.apply_gradients(zip(c_grads, self.critic.trainable_variables))

            self.update_metrics(g_loss, c_loss)
        return {
            "c_loss": self.c_loss_metric.result(),
            "g_loss": self.g_loss_metric.result(),
        }
```

- 더이상 정확도는 계산하지 않는다.

WGAN Model - loss

```
class WGAN(tf.keras.Model):  
    ...  
    def critic_loss(self, y_real, y_fake):  
        y_true = tf.concat([-tf.ones_like(y_real), tf.ones_like(y_fake)], axis=0)  
        y_pred = tf.concat([y_real, y_fake], axis=0)  
        c_loss = -tf.reduce_mean(y_true * y_pred)  
        return c_loss  
  
    def generator_loss(self, y_fake):  
        g_loss = -tf.reduce_mean(-tf.ones_like(y_fake) * y_fake)  
        return g_loss  
  
    def update_metrics(self, g_loss, c_loss):  
        self.g_loss_metric.update_state(g_loss)  
        self.c_loss_metric.update_state(c_loss)
```

- 더이상 정확도는 계산하지 않는다.
- loss로 Wasserstein Distance를 사용.



경고. 코드의 변화가 간단해 보여서 만만하게 생각하면 안됨. 데미지를 크게 입을 수 있음.

학습 및 모니터링

```
import matplotlib.pyplot as plt

class Monitor(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        x_rnd = np.random.randn(5 * 100).reshape(5, 100)
        generated = self.model.generator(x_rnd)
        for i in range(5):
            plt.subplot(1, 5, 1 + i)
            plt.axis('off')
            plt.imshow(generated[i].numpy().reshape(28, 28), cmap='gray_r')
        plt.show()
```

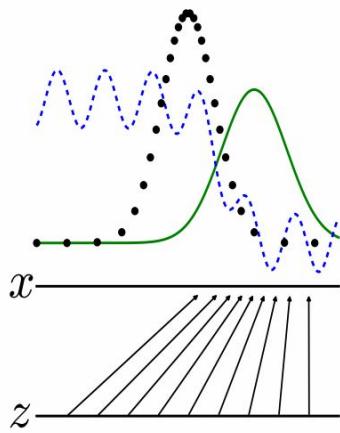
```
tf.keras.backend.clear_session()

critic = make_critic()
generator = make_generator()
gan = WGAN(generator, critic)
gan.fit(x_train, epochs=20, batch_size=128, callbacks=[Monitor()])
```

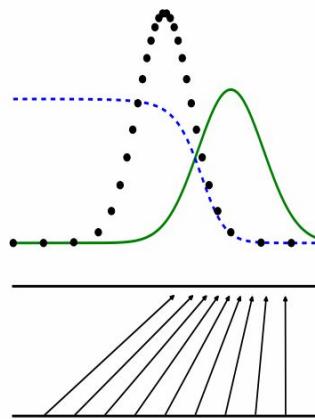
Why?

이름도 어려운 Wasserstein을 배워야
하는 건 왜 때문일까?

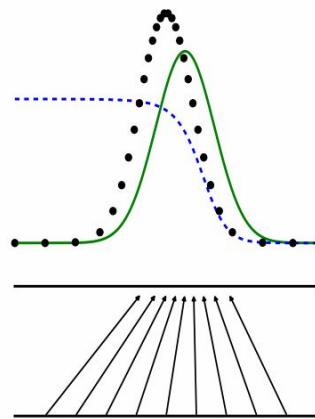
GAN의 분포 학습은 이제 시작
분포를 학습한다는 의미를 조금 깊게 생각해보자



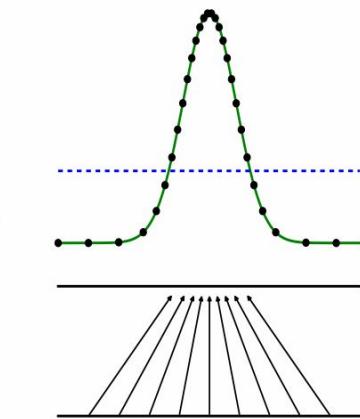
(a)



(b)



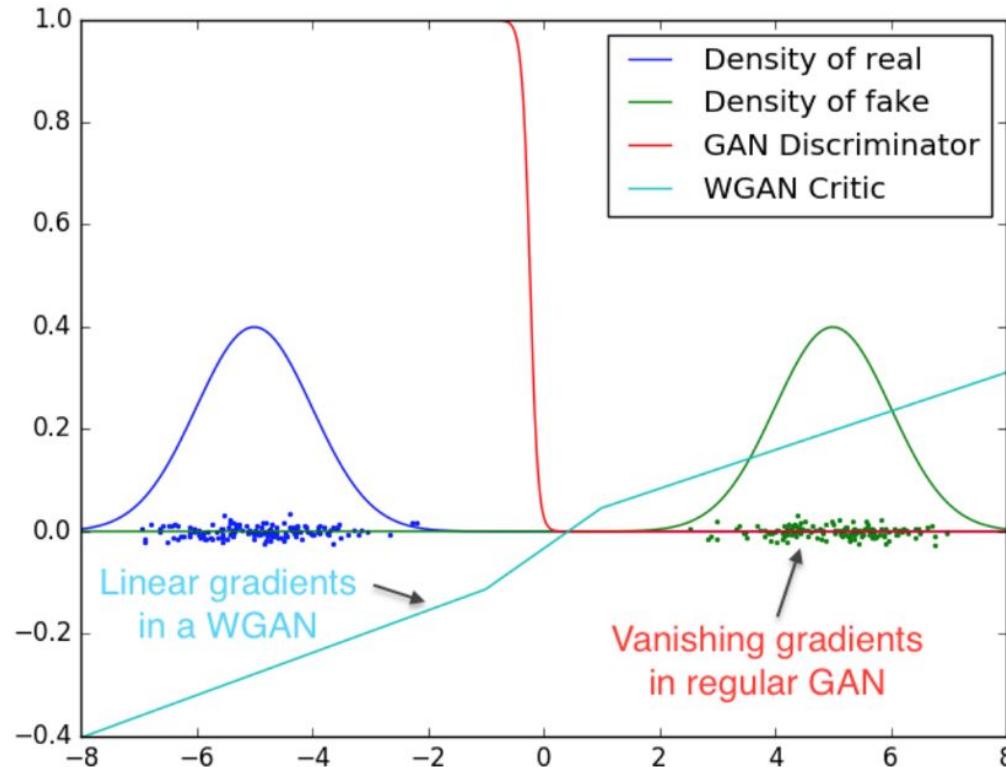
(c)



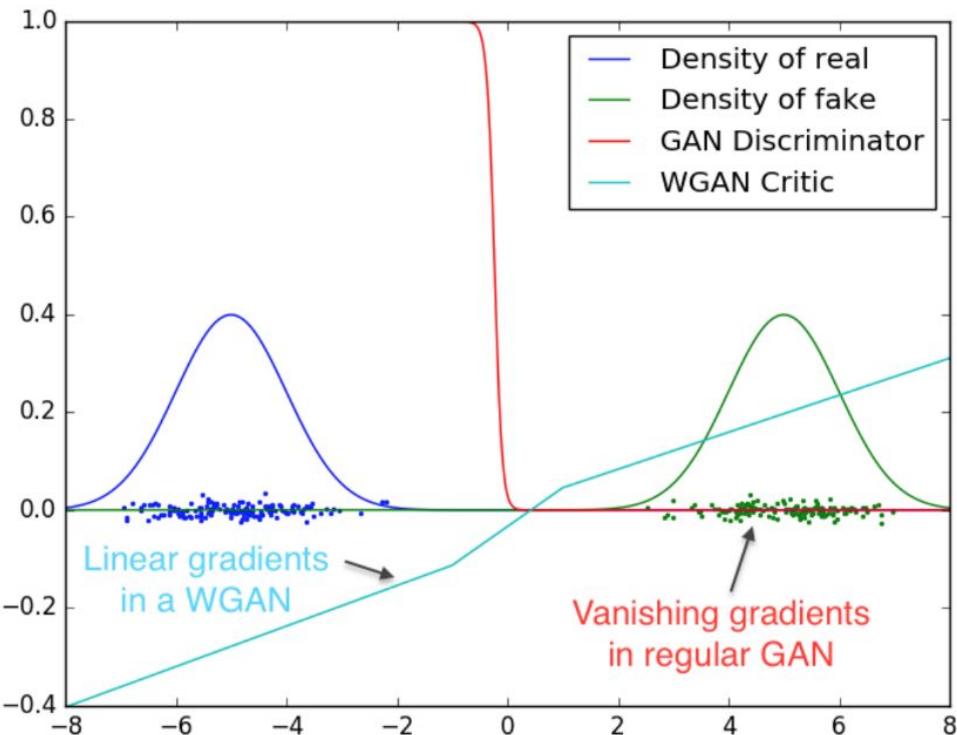
(d)

- G 함수(분포)가 업데이트 되면서 실제 분포를 흉내낸다.
- G 함수의 weights를 조금씩 업데이트 해가는 것.
- 판별기가 G 함수가 생성한 데이터를 1이라고 할 확률이 올라가는 방향으로 업데이트
- 분포의 겹치는 면적이 늘어나면 1이라고 할 확률이 올라간다.

분포가 완전히 분리되면 학습은 어떻게 될까?



분포의 거리(차이)를 계산하는 방법이 없을까?

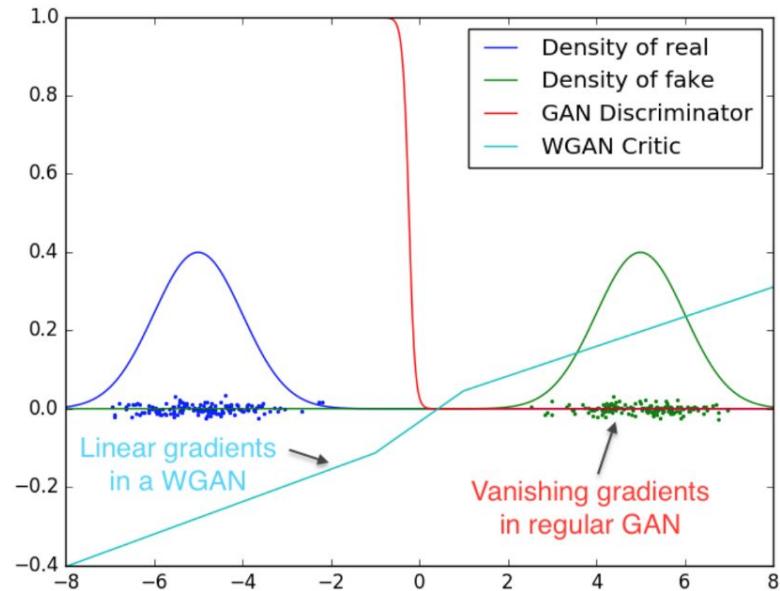


Wasserstein Distance
삽질의 양으로 거리를
측정해보자.

분포의 거리(차이)를 계산하는 방법이 없을까?

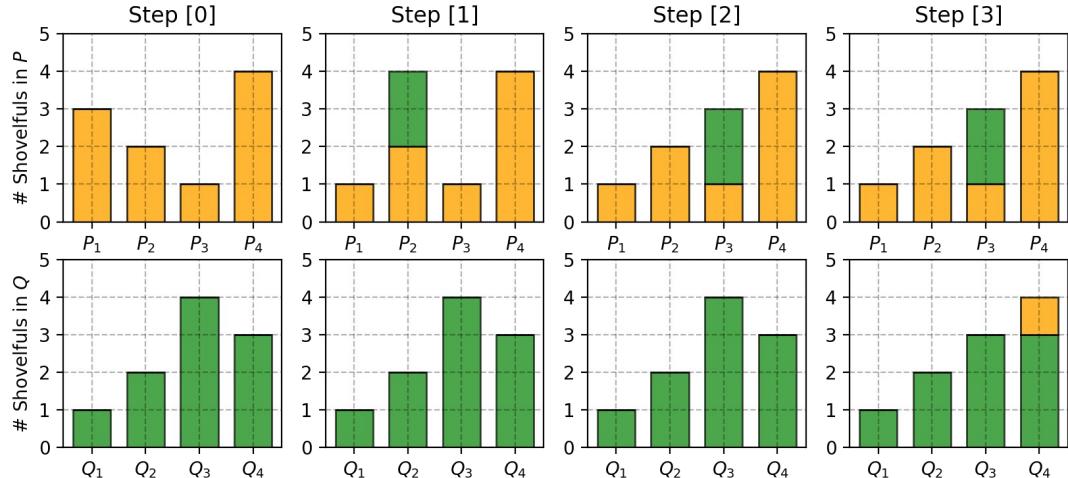
Wasserstein GAN

- Wasserstein 거리 도입. EM(Earth–Mover) Distance 라고도 불림
- 실제 분포에 가까워지는 걸 표현하는 수식
- 겹치는 분포가 아니어도
가까워지는 만큼 값이 작아진다.
- JS Divergence는 분포가 분리되는 경우
loss의 변화가 없어. 학습이 불가하네!



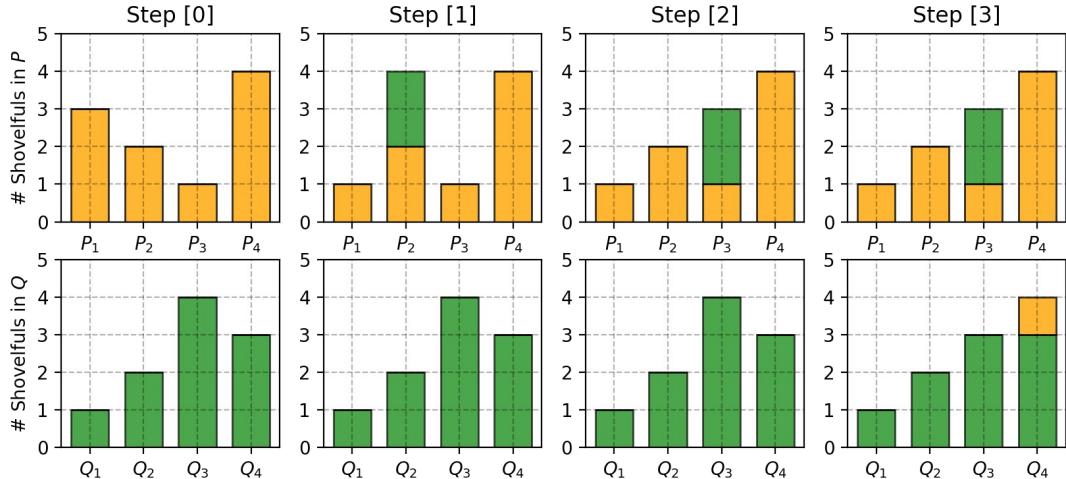
Wasserstein Distance - EM(Earth-Mover) Distance

- 두 확률분포간의 거리를 측정하는 지표
- Earth Mover's distance (EM distance) 라는 다른 이름이 있음.
- 확률 분포의 차이 = 최소한의 흙더미를 옮기는 비용
- 비용은 옮겨진 흙의 양과 이동한 거리를 곱하여 정량화합니다.



P를 Q처럼 바꾸기 위해서는

1. P1에서 2만큼을 P2로 이동시킵니다. $\Rightarrow (P_1, Q_1)$ 이 같아집니다.
2. P2에서 2만큼을 P3로 이동시킵니다. $\Rightarrow (P_2, Q_2)$ 가 같아집니다.
3. Q3에서 1만큼을 Q4로 이동시킵니다. $\Rightarrow (P_3, Q_3)$ 과 (P_4, Q_4) 가 같아집니다.



P를 Q처럼 바꾸기 위해서는

P_i 와 Q_i 가 같아지게 하는데 드는 비용을 δi 라고 표시하면, $\delta i+1 = \delta i + P_i - Q_i$ 로 나타낼수 있습니다. 따라서 위의 과정을 수식으로 표현하면 아래와 같습니다.

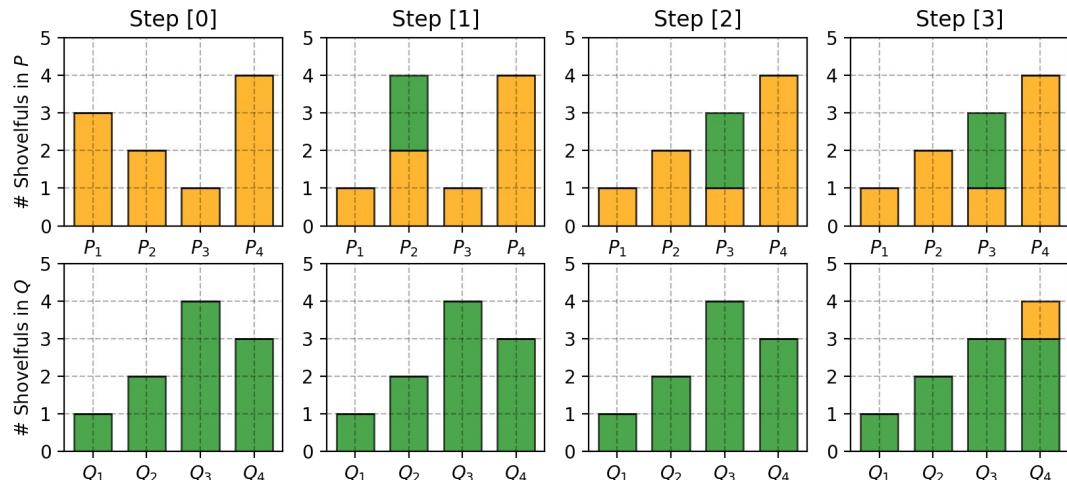
$$\delta_0 = 0$$

$$\delta_1 = 0 + 3 - 1 = 2$$

$$\delta_2 = 2 + 2 - 2 = 2$$

$$\delta_3 = 2 + 1 - 4 = -1$$

$$\delta_4 = -1 + 4 - 3 = 0$$



최종적으로 Earth Mover's distance $W = \sum |\delta i| = 5$ 가 됩니다.

방금 우리가 한 걸 수식으로 표현한 것!

- Wasserstein Distance
- EM(Earth–Mover) Distance

$$W(p_r, p_g) = \inf_{\gamma \sim \Pi(p_r, p_g)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$$

EM 이동 계획들 중 최소값을 찾으라는 의미

그러나!

이 수식을 그대로 이용하기는 어려워서 변환이 필요함.

그리고, 그 유도 과정은 이해가 어려움.

일단 중간과정 스킵하고 결론부터 보자.

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r} [f_w(x)] - \mathbb{E}_{z \sim p_r(z)} [f_w(g_\theta(z))]$$

유도된 수식

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

- 기존의 loss와 비슷한 패턴이 보인다.
- $D(x)$ 함수를 학습하는 대신 $f_w(x)$ 함수를 학습하면 되는 것으로 보임.
- Kantorovich–Rubinstein duality를 이용한 수식 유도.

다음의 내용으로 정리할 수 있다.

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r} [f_w(x)] - \mathbb{E}_{z \sim p_r(z)} [f_w(g_\theta(z))]$$

유도된 수식

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

- 분포의 거리를 측정하는 Wasserstein Distance 거리를 Loss 함수로 사용해보자.
- 거리를 재려면 **상상속의 fw함수**가 필요하다.
- 사용하려는 loss가 이전의 **Gan loss** 함수와 모양이 비슷.
- Gan에서 **D함수**를 학습했듯 **상상속의 fw함수**도 딥러닝이 학습하면 되겠네?
- **상상속의 f함수가 Critic 함수임.**

어려운 수식 유도과정, 조금만 알아보자.

$$W(p_r, p_g) = \inf_{\gamma \sim \Pi(p_r, p_g)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$$

아까 봤던 애. 얘는 못씀. 미분 안되서 안됨.

$$W(p_r, p_g) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim p_r} [f(x)] - \mathbb{E}_{x \sim p_g} [f(x)]$$

미분 가능한 연속함수로 대체하려고 등장한 수식

새로운 수식은 까다로움. 사용을 위해 다음의 조건을 만족해야 함.

1. 유도된 W 함수 \leqq 기존 W 함수
2. 특정 조건을 만족하는 우리에게 필요한 상상속의 f 함수 특징
 - a. 특정 조건(K-lipschitz 연속)을 만족해야만 수식이 성립
 - b. $K=1$: f 함수는 어떤 지점에서도 기울기가 1을 넘지 않음을 의미

상상속의 f함수를 딥러닝이 찾게 하자.

$$W(p_r, p_g) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim p_r}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)]$$

미분 가능한 연속함수로 대체하려고 등장한 수식

$$W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{z \sim p_r(z)}[f_w(g_\theta(z))]$$

딥러닝이 수식을 최대로 만드는 f를 찾으면 된다
f(x) 직접 계산해내는 건 불가능하니,
딥러닝을 통해 $f_w(x)$ 를 찾아보자.

새로운 수식은 까다로움. 사용을 위해 다음의 조건을 만족해야 함.

1. 유도된 W 함수 \leqq 기존 W 함수
2. 특정 조건을 만족하는 우리에게 필요한 상상속의 f 함수 특징
 - a. 특정 조건(K-lipschitz 연속)을 만족해야만 수식이 성립
 - b. K=1: f 함수는 어떤 지점에서도 기울기가 1을 넘지 않음을 의미

WGAN Loss Function

$$W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r} [f_w(x)] - \mathbb{E}_{z \sim p_r(z)} [f_w(g_\theta(z))]$$

- Critic 손실 함수
 - W를 최대화 \Rightarrow -W를 최소화
 $L^{(D)} = -\mathbb{E}_{x \sim p_{data}} D_w(x) + \mathbb{E}_z D_w(G(z))$ log가 없음.
최종 sigmoid activation을 사용하지 않음.
- Generator 손실 함수
 - W를 최소화
 $L^{(G)} = -\mathbb{E}_z D_w(G(z))$ 수식을 간결히 하기 위해 사용하는
판별함수의 정답
 - real: 1.0
 - fake: -1.0

WGAN Model - loss

```
class WGAN(tf.keras.Model):
    ...
    def critic_loss(self, y_real, y_fake):
        y_true = tf.concat([-tf.ones_like(y_real), tf.ones_like(y_fake)], axis=0)
        y_pred = tf.concat([y_real, y_fake], axis=0)
        c_loss = -tf.reduce_mean(y_true * y_pred)
        return c_loss

    def generator_loss(self, y_fake):
        g_loss = -tf.reduce_mean(-tf.ones_like(y_fake) * y_fake)
        return g_loss

    def update_metrics(self, g_loss, c_loss):
        self.g_loss_metric.update_state(g_loss)
        self.c_loss_metric.update_state(c_loss)
```

수식을 간결히 하기 위해 사용하는 판별함수의 정답

- real: -1.0
- fake: 1.0

$$L^{(D)} = -\mathbb{E}_{x \sim p_{data}} D_w(x) + \mathbb{E}_z D_w(G(z))$$

$$L^{(G)} = -\mathbb{E}_z D_w(G(z))$$

코드의 의미. $D(x) \Rightarrow$

- 정답을 맞추면 1 (1, 1) or (-1, -1)
- 정답을 틀리면 -1 (1, -1) or (-1, 1)



고생했어요.
쉬어갑시다.

뇌가 뽑혀 나갈듯한 고통

WGAN 의미와 한계

- 의미

- GAN 분야에서 학습이 잘 안되던 문제를 해결한 핵심 논문
- 판별기가 먼저 학습이 끝나도 generator가 안정적으로 학습할 수 있게 되었다.
 - 실제 구현도 판별기 5회 학습과 생성기 1회 학습의 페어로 제안,

- 한계

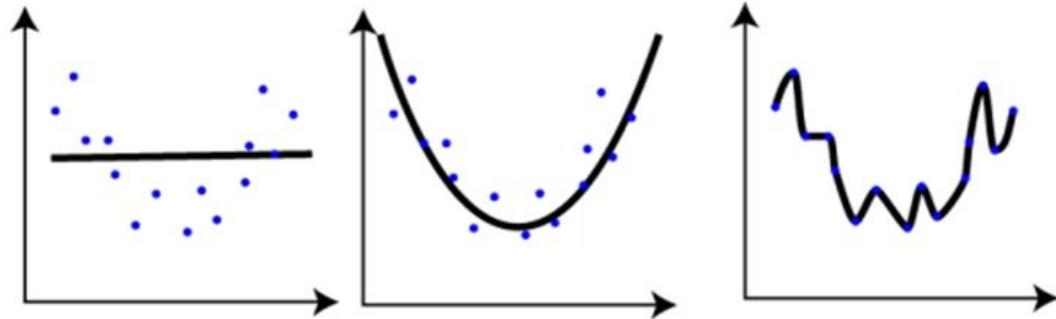
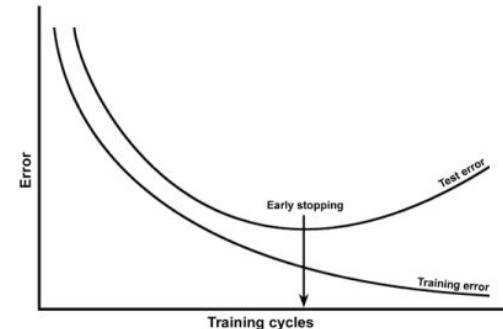
- Momentum-based Optimizer로는 학습이 잘 안됨.
- 1-lipschitz 조건을 만족시키기 위한 방법이 weight clipping이라는 비효율적인 방법.
 - 휴리스틱한 방법의 제안. 학습된 weight가 clipping 포인트에 몰린다.
- Weight clipping is a **clearly terrible way** to enforce a Lipschitz constraint.

Overfitting

Overfitting

훈련 데이터에 특화된 학습을 하게 되어 새로운 데이터에 대한 예측이 오히려 나빠지거나 학습의 효과가 나타나지 않는 상태

- 충분한 데이터 확보
- Dropout / Ensemble
- L1, L2 Regularization
- Data Augmentation



(a)

(b)

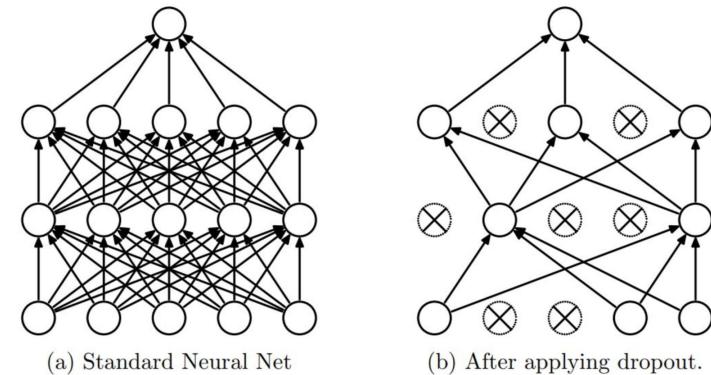
(c)

Dropout

- 지정된 확률의 개수만큼 노드를 랜덤하게 제외하고 학습을 진행.
- 완성된 모델에서는 모든 노드를 사용.
- 효과
 - 오버피팅 방지
 - 성능 향상
 - 앙상블 효과

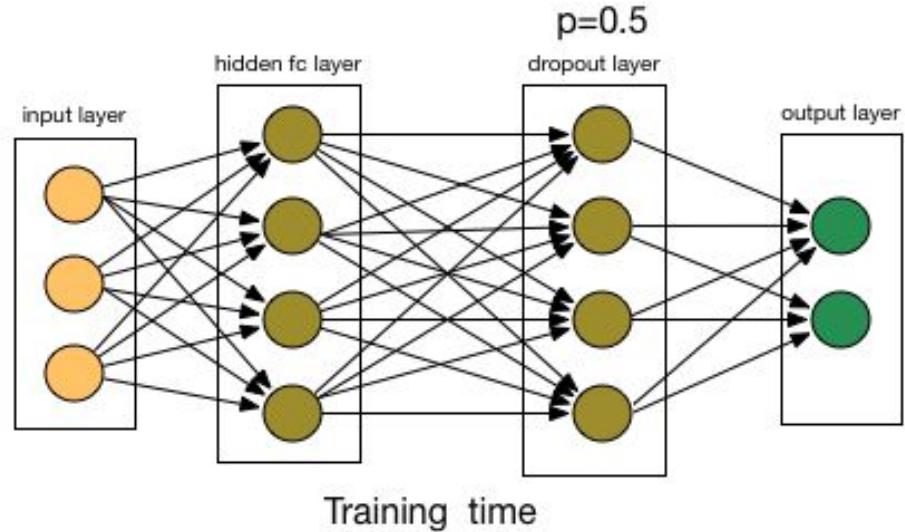
학습이 된다면 최대한 높게 줄수록 좋다.

데이터가 충분하면 $0.4 \sim 0.6$

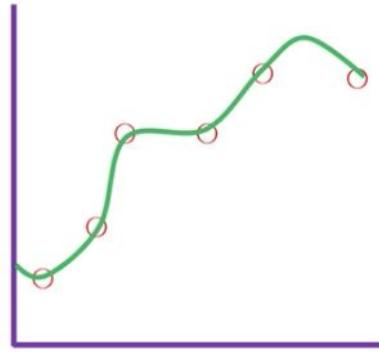


(a) Standard Neural Net

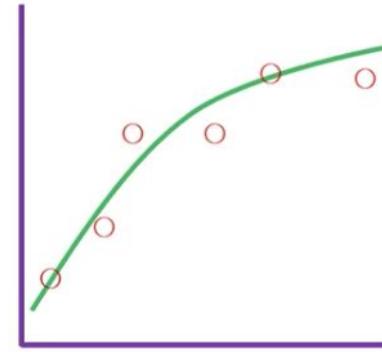
(b) After applying dropout.



L1, L2 Regularization



$$\beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 x^4$$



$$\beta_0 + \beta_1 x + \beta_2 x^2$$

$$\min_{\beta} \sum_{i=1} (y_i - \hat{y}_i)^2 + 5000\beta_3^2 + 5000\beta_4^2$$

L1, L2 Regularization

- L1 (lasso)
 - 중요도가 낮은 가중치부터 0이 됨.
 - 변수 선택의 효과
- L2 (ridge)
 - 아주 큰 가중치에 페널티 부여
 - 가중치들이 전체적으로 평평해짐

L1 Regularization

$$\text{Cost} = \sum_{i=0}^N (y_i - \sum_{j=0}^M x_{ij} W_j)^2 + \lambda \sum_{j=0}^M |W_j|$$

L2 Regularization

$$\text{Cost} = \sum_{i=0}^N (y_i - \sum_{j=0}^M x_{ij} W_j)^2 + \lambda \sum_{j=0}^M W_j^2$$

Loss function

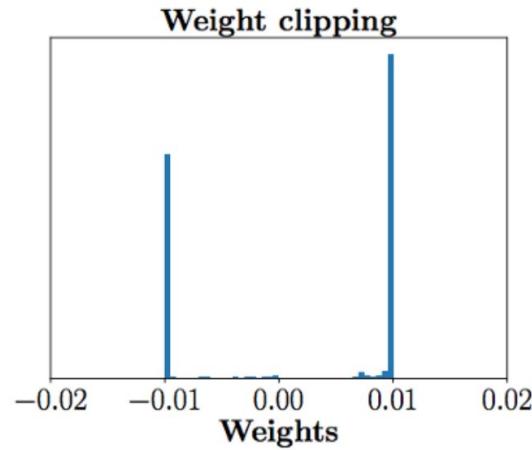
Regularization
Term

WGAN-GP (NIPS 2017)

<https://arxiv.org/pdf/1704.00028.pdf>

WGAN-GP (Gradient Penalty)

WGAN의 한계(Weight Clipping)의 해법을 제시함.



WGAN-GP - Critic

판별기

```
def make_critic():
    dx = tf.keras.Input([28, 28, 1])
    dh = tf.keras.layers.Conv2D(64, 5, 2, padding='same')(dx)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Conv2D(128, 5, 2, padding='same')(dh)
    dh = tf.keras.layers.LeakyReLU(alpha=0.2)(dh)
    dh = tf.keras.layers.Flatten()(dh)
    dy = tf.keras.layers.Dense(1)(dh)
    return tf.keras.Model(dx, dy, name='critic')
```

- 더이상 weight clipping을 해주지 않는다.



WGAN-GP - Generator

```
# 생성기
def make_generator():
    gx = tf.keras.Input([100])
    gh = tf.keras.layers.Dense(7 * 7 * 128)(gx)
    gh = tf.keras.layers.Reshape([7, 7, 128])(gh)
    gh = tf.keras.layers.Conv2DTranspose(128, 5, 2, padding='same')(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gh = tf.keras.layers.Conv2DTranspose(64, 5, 2, padding='same')(gh)
    gh = tf.keras.layers.BatchNormalization()(gh)
    gh = tf.keras.layers.Activation('swish')(gh)
    gy = tf.keras.layers.Conv2D(1, 7, activation='tanh', padding='same')(gh)
return tf.keras.Model(gx, gy, name='generator')
```

WGAN-GP Model - train_step

```
def train_step(self, real):
    noise = tf.random.normal([tf.shape(real)[0], 100])

    with tf.GradientTape() as g_tape, tf.GradientTape() as c_tape:
        fake = self.generator(noise, training=True)

        y_real = self.critic(real, training=True)
        y_fake = self.critic(fake, training=True)

        g_loss = self.generator_loss(y_fake)
        c_loss = self.critic_loss(y_real, y_fake) + self.gradient_penalty(real, fake) * 10

        g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)
        c_grads = c_tape.gradient(c_loss, self.critic.trainable_variables)

        self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))
        self.c_optimizer.apply_gradients(zip(c_grads, self.critic.trainable_variables))

        self.update_metrics(g_loss, c_loss)
    return {
        "c_loss": self.c_loss_metric.result(),
        "g_loss": self.g_loss_metric.result(),
    }
```

WGAN-GP (Gradient Penalty)

1. Loss를 최대화하는 최적의 $f_w(x)$ 함수 f^* 를 찾았다고 가정해보자.

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r} [f_w(x)] - \mathbb{E}_{z \sim p_r(z)} [f_w(g_\theta(z))]$$

2. f^* 는 1-lipshitz 조건을 만족하는 다음 수식으로 완성된다.

$$f^* = \operatorname{argmax}_{\|f\|_L \leq 1} \mathbb{E}_{y \sim \mathbb{P}_r} [f(y)] - \mathbb{E}_{x \sim \mathbb{P}_g} [f(x)]$$

WGAN-GP (Gradient Penalty)

$$f^* = \arg\max_{\|f\|_L \leq 1} \mathbb{E}_{y \sim \mathbb{P}_r}[f(y)] - \mathbb{E}_{x \sim \mathbb{P}_g}[f(x)]$$

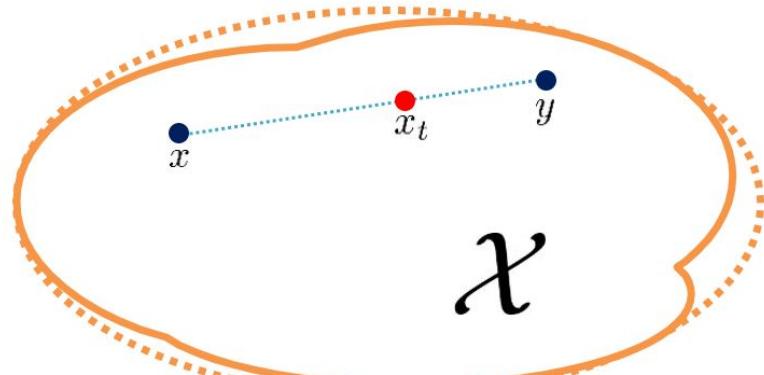
x, y 두 점의 내분점 중 임의의 한 점을 선택

- ▶ $x_t = tx + (1 - t)y$
- ▶ $0 \leq t \leq 1$

그 점은 다음을 만족한다.

f^* 가 1-lipshitz 연속을 만족하기 때문이다.

$$\|\nabla f^*(x_t)\| = 1$$



WGAN-GP (Gradient Penalty)

$$\|\nabla f^*(x_t)\| = 1$$

이 특성을 학습할 수 있게 loss 함수를 조정한다.

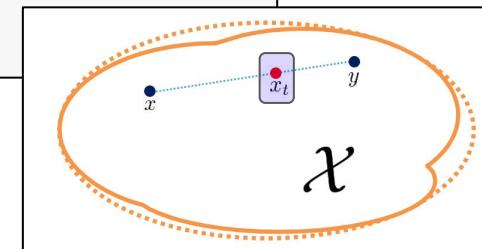
$$L = \underbrace{\mathbb{E}_{\tilde{x} \sim \mathbb{P}_g} [D(\tilde{x})] - \mathbb{E}_{x \sim \mathbb{P}_r} [D(x)]}_{\text{Original critic loss}} + \lambda \underbrace{\mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]}_{\text{Our gradient penalty}}.$$

$$L = \underbrace{\mathbb{E}_{\tilde{x} \sim \mathbb{P}_g} [D(\tilde{x})] - \mathbb{E}_{x \sim \mathbb{P}_r} [D(x)]}_{\text{Original critic loss}} + \underbrace{\lambda \mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]}_{\text{Our gradient penalty}}.$$

```

def gradient_penalty(self, real, fake):
    # real과 fake의 중간 어디쯤의 이미지의 기울기를 이용하여 penalty 계산
    epsilon = tf.random.uniform([tf.shape(real)[0], 1, 1, 1], 0.0, 1.0)
    x_hat = epsilon * real + (1 - epsilon) * fake
    with tf.GradientTape() as tape:
        tape.watch(x_hat)
        y_pred = self.critic(x_hat, training=True)
        grads = tape.gradient(y_pred, x_hat)
    ddx = tf.sqrt(tf.reduce_sum(grads ** 2, axis=[1, 2]))
    gp = tf.reduce_mean((ddx - 1.0) ** 2)
    return gp

```



WGAN-GP (Gradient Penalty)

