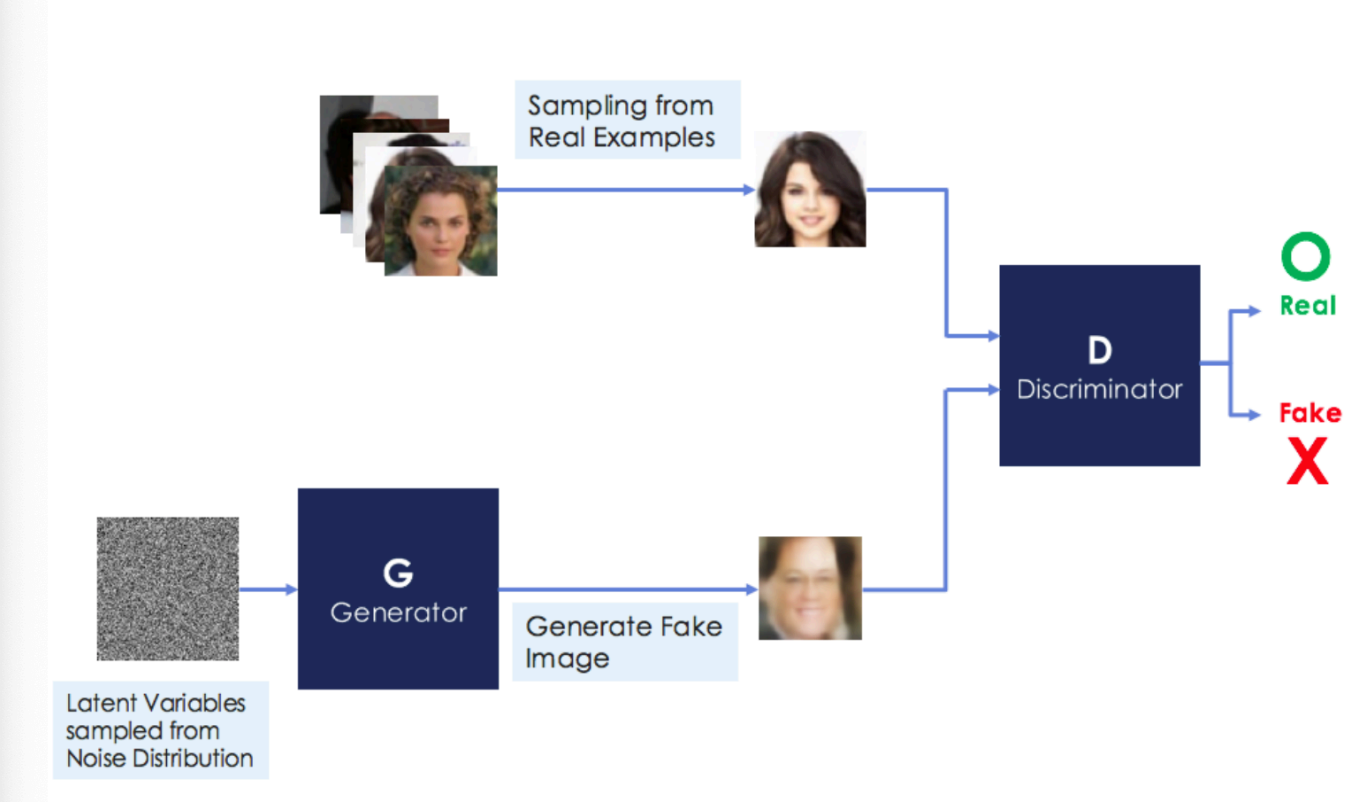
Generative Adversarial Networks (GANs)

Generative Adversarial Networks(GAN)은 적대적 학습(Adversarial Networks) 구조를 이용해서 생성 모델을 학습하는 아키텍쳐입니다. GAN은 Generator(생성자)와 Discriminator(구분자)로 구성되어있습니다. GANs의 구조를 쉽게 이해하기 위해 위조지폐 제작자(생성자)와 위조지폐 단속 경찰(구분자)의 관계로 예를 들겠습니다.



위조지폐 제작자(생성자)는 위조지폐 단속 경찰(구분자)를 잘 속일 수 있도록 실제와 비슷한 위조 지폐를 만들어내도록 학습하고, 위조지폐 단속 경찰(구분자)는 위조지폐 제작자(생성자)가 만들어낸 위조 지폐를 가짜인지 잘 구분하도록 학습합니다. 즉, 위조지폐 제작자(생성자)는 실제 지폐와 비슷한 지폐를 계속해서 생성하는 반면, 위조지폐 단속 경찰(구분자)는 실제 지폐와 위조 지폐의 차이를 확인하려고 노력합니다. 궁극적으로는 실제 지폐와 위조 지폐를 구별할 수 없는 위조 지폐를 만들어낼 수 있는 생성 네트워크를 갖게 됩니다.

이번 프로젝트에서는 무작위 노이즈 벡터에서부터 GANs 학습을 통해 실제와 비슷한 손글씨(그러나 실제 사람이 직접 손으로 쓴 글씨가 아닌, 네트워크가 학습을 통해 만들어낸 "가짜" 손글씨)를 만들어내는 작업을 할 예정입니다.

In [10]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm


from keras.layers import Input
from keras.models import Model, Sequential
from keras.layers.core import Dense, Dropout
from keras.layers.advanced_activations import LeakyReLU
from keras.datasets import mnist
from keras.optimizers import Adam
from keras import initializers
```

plot을 그릴 수 있는 matplotlib와 for 문의 진행 상태를 시각적으로 확인할 수 있는 tqdm을 사용할 것이고 tensorflow를 백엔드로 사용하는 Keras를 해당 프로젝트에서 사용하겠습니다.

In [11]:
```
os.environ["KERAS_BACKEND"] = "tensorflow"

np.random.seed(10)

random_dim = 100
```

os.environ["KERAS_BACKEND"] = "tensorflow"

Keras 가 Tensorflow 를 벡엔드로 사용할 수 있도록 설정합니다.

np.random.seed(10)

랜덤 시드를 설정합니다

random_dim = 100

생성자의 랜덤 노이즈 벡터 차원을 설정합니다

In [12]:
```
def load_minst_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = (x_train.astype(np.float32) - 127.5)/127.5
    x_train = x_train.reshape(60000, 784)

    return (x_train, y_train, x_test, y_test)
```

(x_train, y_train), (x_test, y_test) = mnist.load_data()

MNIST 데이터 셋을 사용하여 0~9 의 단일 자릿수 이미지 세트를 로드합니다

x_train = (x_train.astype(np.float32) - 127.5)/127.5

데이터 값을 -1~1로 normalize합니다

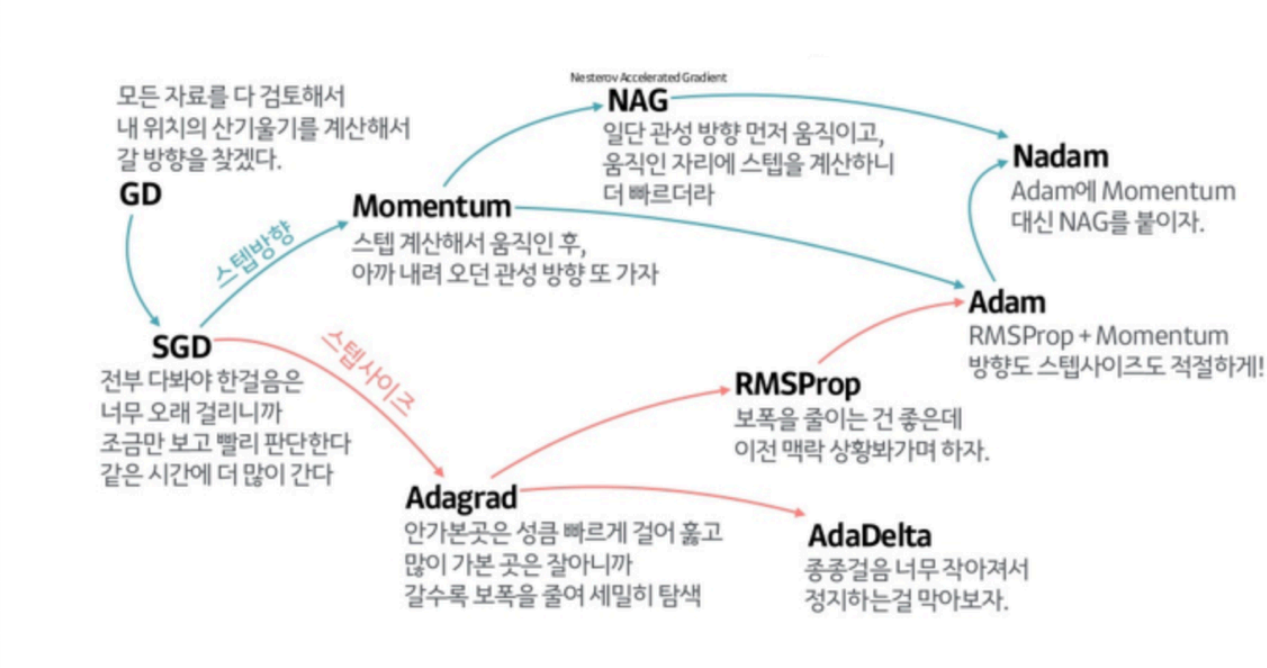
x_train = x_train.reshape(60000, 784)

x_train 의 형태를 한 row당 784 columns를 갖도록 수정합니다

In [13]:
```
def get_optimizer():
    return Adam(lr=0.0002, beta_1=0.5)
```

저희가 수업에서 배웠던 Gradient Descent는 최적값을 찾아 나가기 위해서 한칸 전진할 때마다 모든 데이터 셋을 넣어 주어야 합니다. 이는 학습이 오래걸리는 단점을 유발합니다.

Gradient Descent 대신 더 빠른 Optimizer는 "Stochastic Gradient Descent", "Momentum", "NAG","Adam" 등이 있습니다.



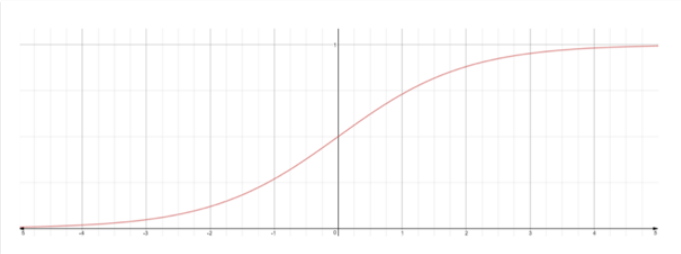
해당 프로젝트에서는 최근에 많이 사용되는 Adam Optimizer를 사용하겠습니다.

이제 생성자와 구분자 네트워크를 만들어 보겠습니다

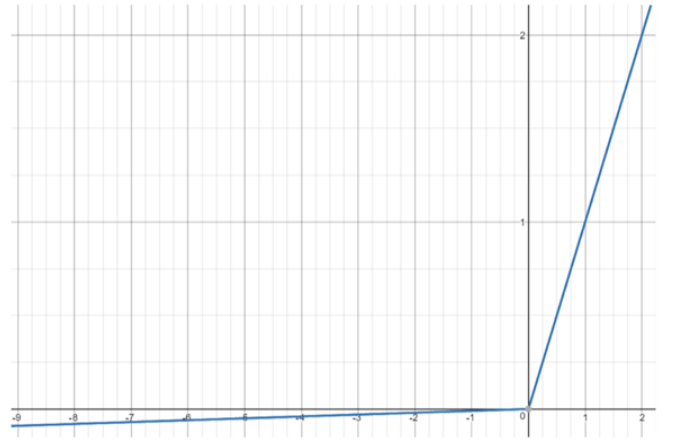
In [14]:
```
def get_generator(optimizer):
    generator = Sequential()
    generator.add(Dense(256, input_dim=random_dim, kernel_initializ
er=initializers.RandomNormal(stddev=0.02)))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(512))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(1024))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(784, activation='tanh'))
    generator.compile(loss='binary_crossentropy', optimizer=optimiz
er)
    return generator

def get_discriminator(optimizer):
    discriminator = Sequential()
    discriminator.add(Dense(1024, input_dim=784, kernel_initializer
=initializers.RandomNormal(stddev=0.02)))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(256))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(1, activation='sigmoid'))
    discriminator.compile(loss='binary_crossentropy', optimizer=opt
imizer)
    return discriminator
```

두 모델 모두 keras.models.Sequential() 를 통해 모델을 구현하고, 만들어낸 모델에 레이어를 추가하기 위해 완전연결 계층인 Dense를 사용합니다. 이때, . Dense의 첫번째 매개변수는 노드의 갯수입니다.

두 모델 모두 활성함수로 "Leakly Relu"를 사용합니다. 활성함수란 뉴럴네트워크의 개별 뉴런에 들어오는 입력신호의 총합을 출력신호로 변환하는 함수입니다. 활성화함수는 대개 비선형함수(non-linear function)를 씁니다. 저희 수업에서는 시그모이드를 사용하였는데, 학습 속도가 다소 느려진다는 단점이 있습니다. 따라서 해당 프로젝트에서는 ReLU(Rectified Linear Unit)의 변형 형태인 Leakly Relu를 사용하겠습니다. 실제로 ReLU 계열은 시그모이드나 하이퍼볼릭탄젠트 함수 대비 학습수렴 속도가 6배나 빠르다고 합니다.

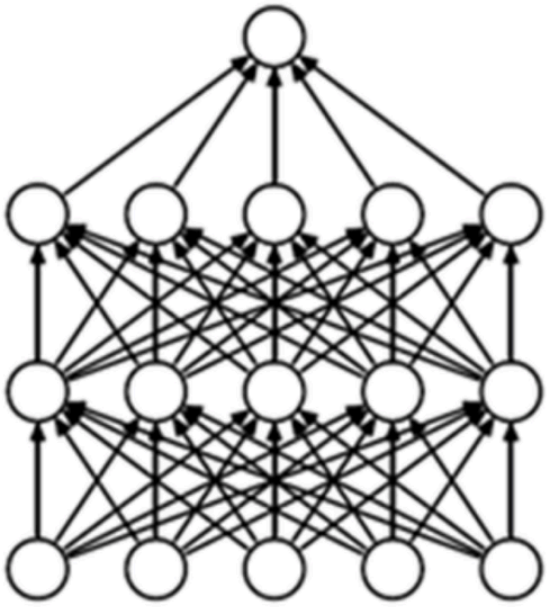


시그모이드

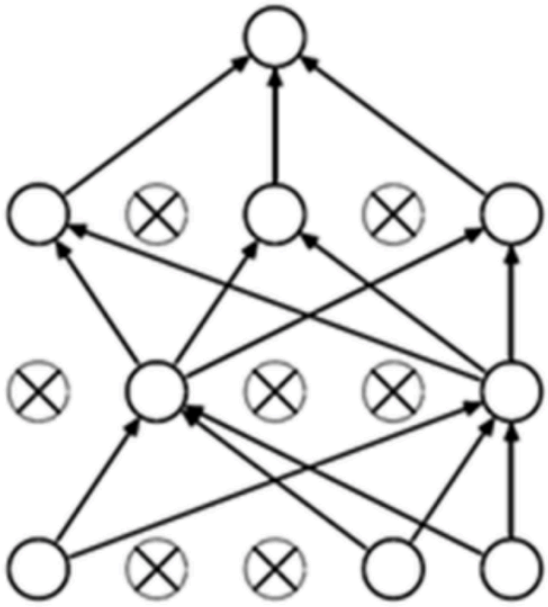


Leakly ReLU

두 모델 모두 세 개의 숨겨진 레이어가 있는 신경 네트워크를 구축합니다. 이때, 구분자의 경우 Dropout 레이어를 추가하는데, 이는 Overfitting을 피하기 위함입니다. Dropout은 망에 있는 모든 layer에 대해 학습을 수행하는 것이 아니라 입력 레이어나 hidden 레이어의 일부 뉴런을 생략하고 줄어든 신경망을 통해 학습을 수행하게 합니다.



(a) Standard Neural Net



(b) After applying dropout.

In [15]:
```
def get_gan_network(discriminator, random_dim, generator, optimizer
):
    discriminator.trainable = False
    gan_input = Input(shape=(random_dim,))
    x = generator(gan_input)
    gan_output = discriminator(x)
    gan = Model(inputs=gan_input, outputs=gan_output)
    gan.compile(loss='binary_crossentropy', optimizer=optimizer)

    return gan
```

discriminator.trainable = False

생성자와 구분자를 동시에 학습시키고 싶을 때 trainable을 False로 설정합니다.

gan_input = Input(shape=(random_dim,))

GAN 입력 노이즈는 100 차원으로 설정했습니다.

x = generator(gan_input)

생성자는 이미지를 output 합니다

gan_output = discriminator(x)

구분자는 이 이미지가 진짜인지 가짜인지 확률로 output 합니다

In [16]:
```
def plot_generated_images(epoch, generator, examples=100, dim=(10,
10), figsize=(10, 10)):
    noise = np.random.normal(0, 1, size=[examples, random_dim])
    generated_images = generator.predict(noise)
    generated_images = generated_images.reshape(examples, 28, 28)

    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest', cm
ap='gray_r')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig('gan_generated_image_epoch_%d.png' % epoch)
```

매 20번째 epoch마다 생성된 이미지를 동일 디렉토리에 저장합니다. 이는 GAN이 트레이닝 되는 과정을 시각적으로 보기 위해서입니다.

In [ ]:
```
def train(epochs=1, batch_size=128):
    x_train, y_train, x_test, y_test = load_minst_data()
    batch_count = x_train.shape[0] // batch_size

    adam = get_optimizer()
    generator = get_generator(adam)
    discriminator = get_discriminator(adam)
    gan = get_gan_network(discriminator, random_dim, generator, ada
m)

    for e in range(1, epochs+1):
        print ('-'*15, 'Epoch %d' % e, '-'*15)
        for _ in tqdm(range(batch_count)):
            noise = np.random.normal(0, 1, size=[batch_size, random
_dim])
            image_batch = x_train[np.random.randint(0, x_train.shap
e[0], size=batch_size)]

            generated_images = generator.predict(noise)
            X = np.concatenate([image_batch, generated_images])

            y_dis = np.zeros(2*batch_size)
            y_dis[:batch_size] = 0.9

            discriminator.trainable = True
            discriminator.train_on_batch(X, y_dis)

            noise = np.random.normal(0, 1, size=[batch_size, random
_dim])
            y_gen = np.ones(batch_size)
            discriminator.trainable = False
            gan.train_on_batch(noise, y_gen)

        if e == 1 or e % 20 == 0:
            plot_generated_images(e, generator)

if __name__ == '__main__':
    train(400, 128)
```

x_train, y_train, x_test, y_test = load_minst_data()

train 데이터와 test 데이터를 가져옵니다.

batch_count = x_train.shape[0] // batch_size

train 데이터를 128 사이즈의 batch 로 나눕니다.

gan = get_gan_network(discriminator, random_dim, generator, adam)

GAN 을 생성합니다

noise = np.random.normal(0, 1, size=[batch_size, random_dim])

입력으로 사용할 random 노이즈와 이미지를 가져옵니다.

discriminator.trainable = True

discriminator.train_on_batch(X, y_dis)

Discriminator를 학습시킵니다.

noise = np.random.normal(0, 1, size=[batch_size, random_dim])

y_gen = np.ones(batch_size)

discriminator.trainable = False

gan.train_on_batch(noise, y_gen)

Generator를 학습시킵니다.

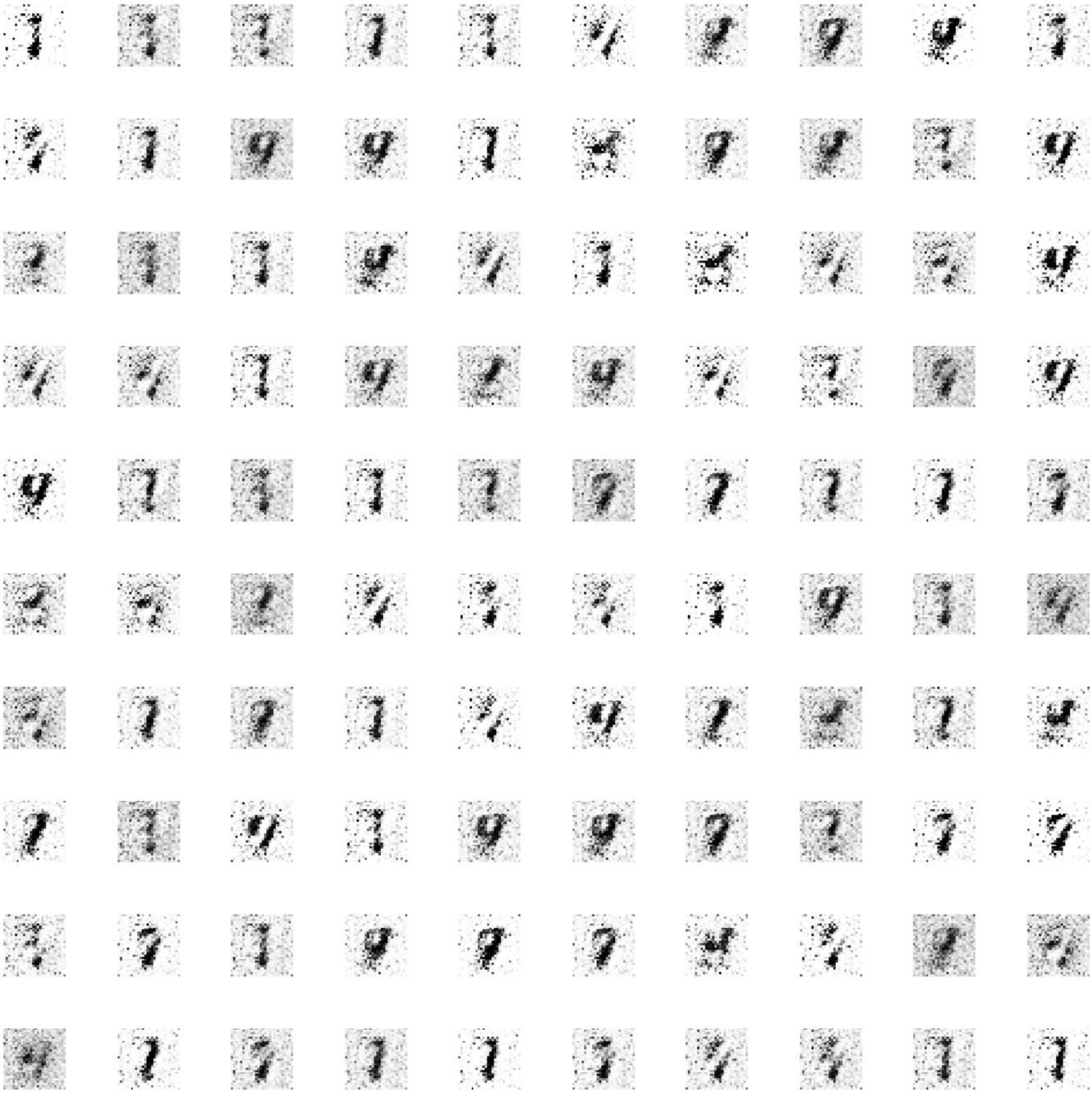
if e == 1 or e % 20 == 0:

plot_generated_images(e, generator)

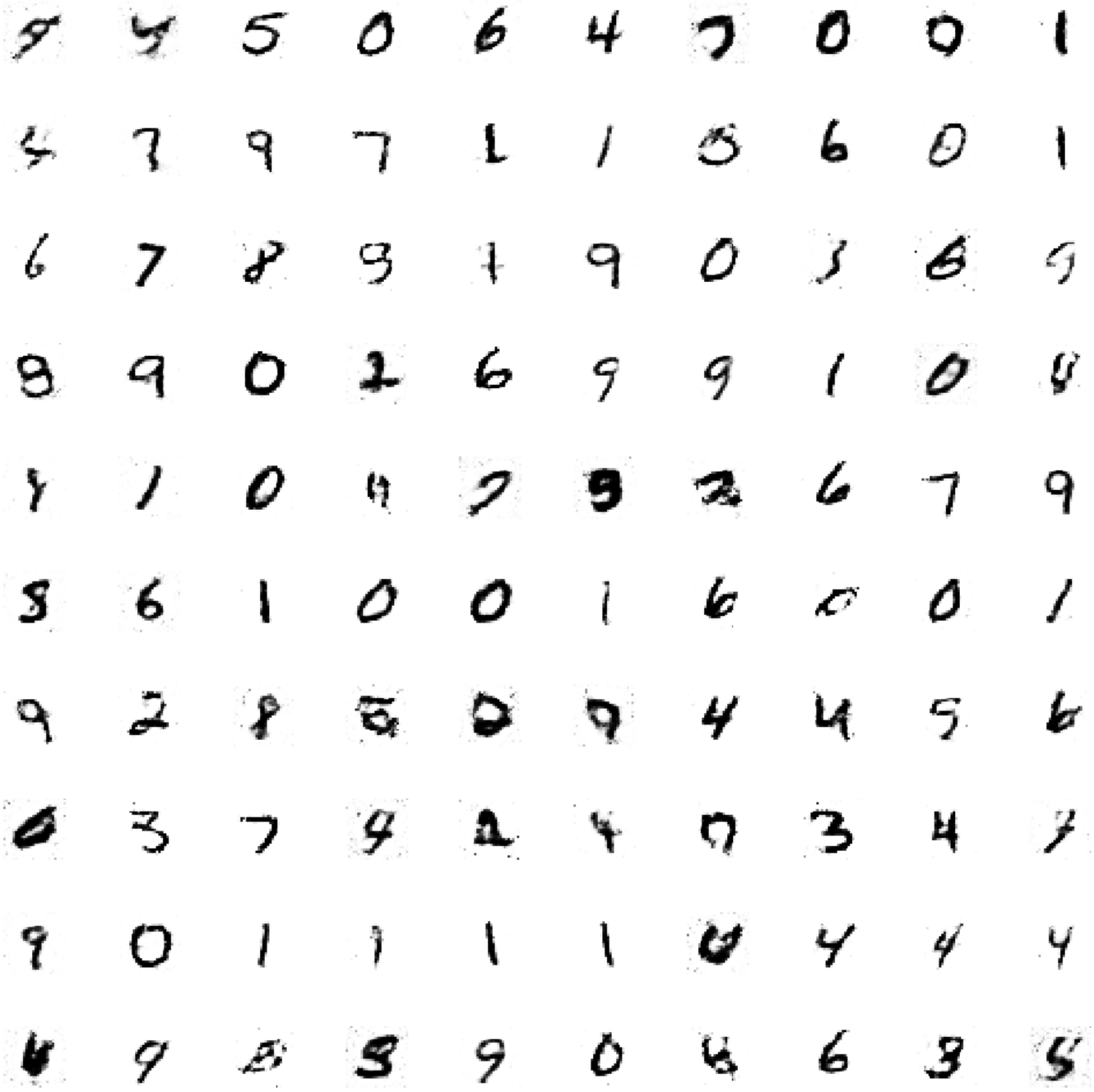
첫번째, 20번째 epoch 마다 이미지를 생성합니다

각 epoch 별로 약 2분 정도 소요되므로 미리 학습한 이미지를 첨부하였습니다.

epoch 1



epoch 100



epoch 200

초기 epoch 의 경우 구조가 없는 무작위 노이즈 값으로 보이지만, 학습을 진행할 수록 숫자가 형성되고, 최종적으로는 뚜렷한 손글씨를 확인할 수 있습니다.

In [ ]: