**Final-Term Project**

**Implementing Basic Generative Adversarial Network (GAN)**

컴퓨터전자시스템공학부

202030014 곽민호

1. **Term Project 목표**

간단한 Generative Adversarial Network (GAN) 구현 및 분석하기

1. **구현 환경**

|  |
| --- |
| Ubuntu 18.01 LST |
| Visual Studio Code |
| Python 3.5 |
| Numpy 1.12.1  Tensorflow 1.4.0  Matplotlib 2.0.1 |

호환성 문제를 피하기 위해 낮은 버전의 Tensorflow를 사용했습니다.

* **Assignment 1-1**

**Analyze a given source code**

**Ubuntu에서 한글 입력이 안되서 개별 첨부한 소스코드는 영어로 작성했습니다.**

'''

version info.

호환성 문제를 피하기 위해 구형 Python 패키지 사용

python==3.5

tensorflow==1.4.0

numpy==1.12.1

matplotlib==2.0.1

'''

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.gridspec as gridspec

import os

from tensorflow.examples.tutorials.mnist import input\_data

################## 함수 정의 ####################

def xavier\_init(size): ## xavier initiation, 각 레이어의 노드 수에 따라 초기화가 진행됩니다.

input\_dim = size[0]

xavier\_variance = 1. / tf.sqrt(input\_dim/2.)

return tf.random\_normal(shape=size, stddev=xavier\_variance)

## xavire varience를 따르는 정규 분포에 해당하는 출력을 return 합니다.

def plot(samples):

## 샘플 이미지를 보여주고 저장합니다. 저장 경로가 존재하지 않으면 새로 만듭니다.

fig = plt.figure(figsize=(4, 4))

gs = gridspec.GridSpec(4, 4)

gs.update(wspace=0.05, hspace=0.05)

for i, sample in enumerate(samples):

ax = plt.subplot(gs[i])

plt.axis('off')

ax.set\_xticklabels([])

ax.set\_yticklabels([])

ax.set\_aspect('equal')

plt.imshow(sample.reshape(28, 28), cmap='Greys\_r')

## 아래 경로 폴더가 없으면 만들고, 있다면 아무것도 하지 않음

if not(os.path.isdir('./sample')):

os.makedirs(os.path.join('./sample'))

## 이미지의 저장 경로와 이미지 품질 설정

plt.savefig('./sample/sample' + str(num) + '.png', dpi=300)

return fig ## 샘플 이미지를 return합니다.

# Random noise setting for Generator,

## Generator에 입력으로 들어갈 tensor의 placeholder를 Z라고 정합니다.

Z = tf.placeholder(tf.float32, shape=[None, 100], name='Z')

#Generator parameter settings

## Generator의 변수 설정

G\_W1 = tf.Variable(xavier\_init([100, 128]), name='G\_W1') # 첫번째 층 weight

G\_b1 = tf.Variable(tf.zeros(shape=[128]), name='G\_b1') # 첫번째 층 bias

G\_W2 = tf.Variable(xavier\_init([128, 784]), name='G\_W2') # 두번째 층 weight

G\_b2 = tf.Variable(tf.zeros(shape=[784]), name='G\_b2') # 두번째 층 bias

## Generator의 변수 집합

theta\_G = [G\_W1, G\_W2, G\_b1, G\_b2]

#Input Image MNIST setting for Discriminator [28x28=784]

## Discriminator에 입력으로 들어갈 tensor의 placeholder를 X라고 정합니다.

X = tf.placeholder(tf.float32, shape=[None, 784], name='X')

#Discriminator parameter settings

## Discriminator 변수 설정

D\_W1 = tf.Variable(xavier\_init([784, 128]), name='D\_W1') # 첫번째 층 weight

D\_b1 = tf.Variable(tf.zeros(shape=[128]), name='D\_b1') # 첫번째 층 bias

D\_W2 = tf.Variable(xavier\_init([128, 1]), name='D\_W2') # 두번째 층 weight

D\_b2 = tf.Variable(tf.zeros(shape=[1]), name='D\_b2') # 두번째 층 bias

## Discriminator의 변수 집합

theta\_D = [D\_W1, D\_W2, D\_b1, D\_b2]

# Generator Network

def generator(z):

G\_h1 = tf.nn.relu(tf.matmul(z, G\_W1) + G\_b1) ## 1번째 층 연산

G\_log\_prob = tf.matmul(G\_h1, G\_W2) + G\_b2 ## 2번째 층 연산

G\_prob = tf.nn.sigmoid(G\_log\_prob) ## sigmoid를 통해 확률 출력

return G\_prob

# Discriminator Network

def discriminator(x):

D\_h1 = tf.nn.relu(tf.matmul(x, D\_W1) + D\_b1) ## 1번째 층 연산

D\_logit = tf.matmul(D\_h1, D\_W2) + D\_b2 ## 2번째 층 연산

D\_prob = tf.nn.sigmoid(D\_logit) ## sigmoid를 통해 확률 출력

return D\_prob, D\_logit

################## 실행 ####################

G\_sample = generator(Z)

D\_real, D\_logit\_real = discriminator(X)

D\_fake, D\_logit\_fake = discriminator(G\_sample)

# Loss functions from the paper

D\_loss = -tf.reduce\_mean(tf.log(D\_real) + tf.log(1. - D\_fake))

G\_loss = -tf.reduce\_mean(tf.log(D\_fake))

# Update D(X)'s parameters

## Adam optimizer를 사용하여 theta\_D를 기준으로 D\_loss를 낮게 학습하기

D\_solver = tf.train.AdamOptimizer().minimize(D\_loss, var\_list=theta\_D)

# Update G(Z)'s parameters

## Adam optimizer를 사용하여 theta\_G를 기준으로 G\_loss를 낮게 학습하기

G\_solver = tf.train.AdamOptimizer().minimize(G\_loss, var\_list=theta\_G)

def sample\_Z(m, n): ## 주어진 크기 m, n의 무작위 균일분포를 따르는 이미지 생성

return np.random.uniform(-1., 1., size=[m, n])

batch\_size = 128

Z\_dim = 100

sess = tf.Session()

sess.run(tf.global\_variables\_initializer())

mnist = input\_data.read\_data\_sets('MNIST/', one\_hot=True)

i = 0 ## 사용되지 않은 변수??

num = 0 ## 샘플 이미지 카운터

################## 반복 학습 및 결과 출력 ####################

for itr in range(1000000): ## max iteration is 1,000,000

if itr % 1000 == 0: ## every 1000 iteration...

samples = sess.run(G\_sample, feed\_dict={Z: sample\_Z(16, Z\_dim)})

## Generator가 Z로부터 생성한 16개의 무작위 이미지를 모아서 샘플을 구성합니다.

num = num + 1000 ## 무작위 이미지 샘플은 1000번 마다 생성합니다

plot(samples) ## sample을 plot하고 해당 이미지를 저장합니다.

X\_mb, \_ = mnist.train.next\_batch(batch\_size)

\_, D\_loss\_curr = sess.run([D\_solver, D\_loss], feed\_dict={X: X\_mb, Z: sample\_Z(batch\_size, Z\_dim)}) ## Discriminator의 현재 Loss를 얻습니다.

\_, G\_loss\_curr = sess.run([G\_solver, G\_loss], feed\_dict={Z: sample\_Z(batch\_size, Z\_dim)}) ## Generator의 현재 Loss를 얻습니다.

## D와 G의 현재 Loss를 터미널에 출력합니다. 해당 출력들은 터미널에서 ‘ > [이름].csv’ 명령어로 해당 출력들을 csv 파일에 저장될 수 있습니다.

print('{:.4},'.format(D\_loss\_curr),'{:.4}'.format(G\_loss\_curr))

* **Assignment 1-2**

**Draw plot of (1) discriminator and (2) generator loss values with respect to iterations**

**<그래프 (1) - Iteration에 따른 Discriminator와 Generator의 Loss 그래프>**

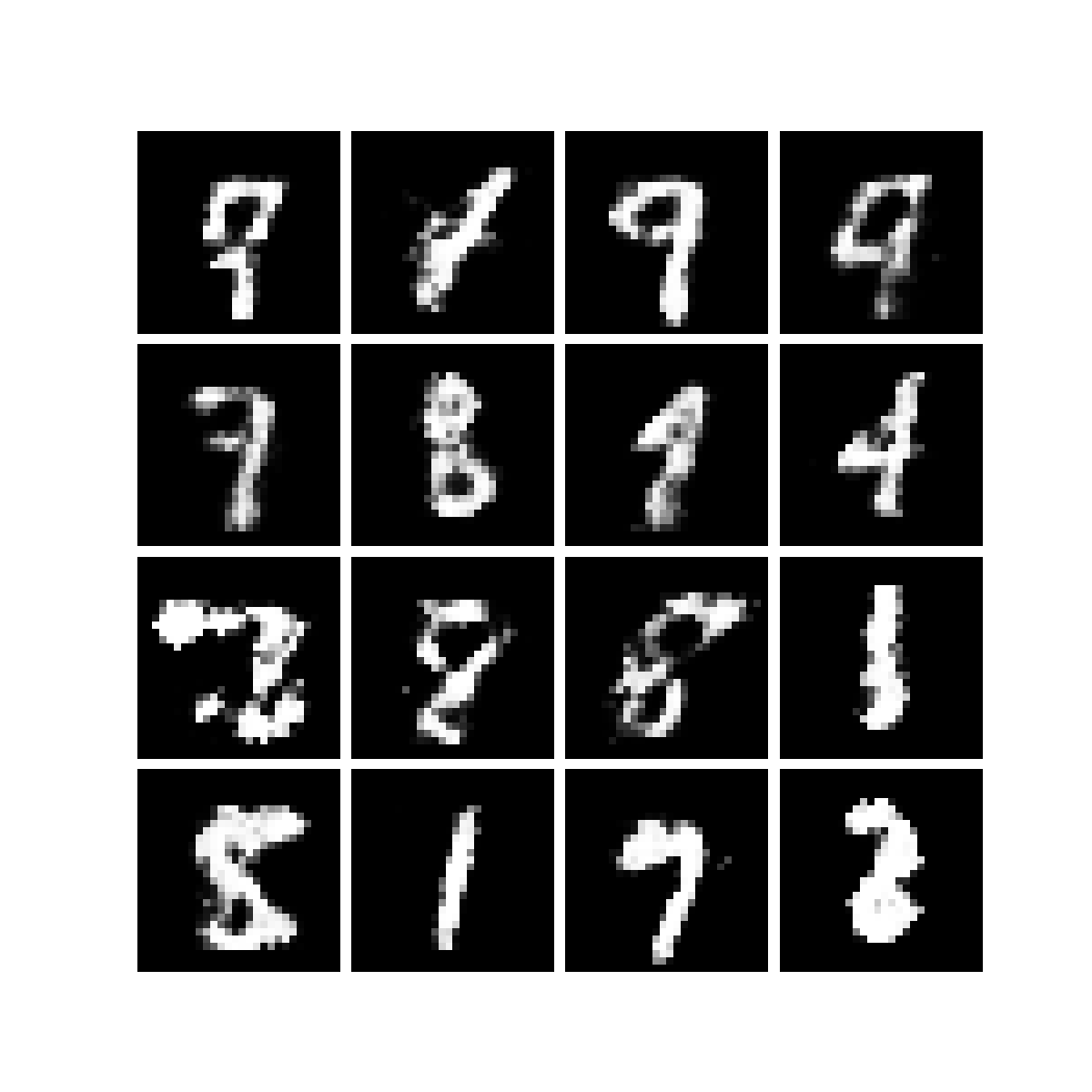
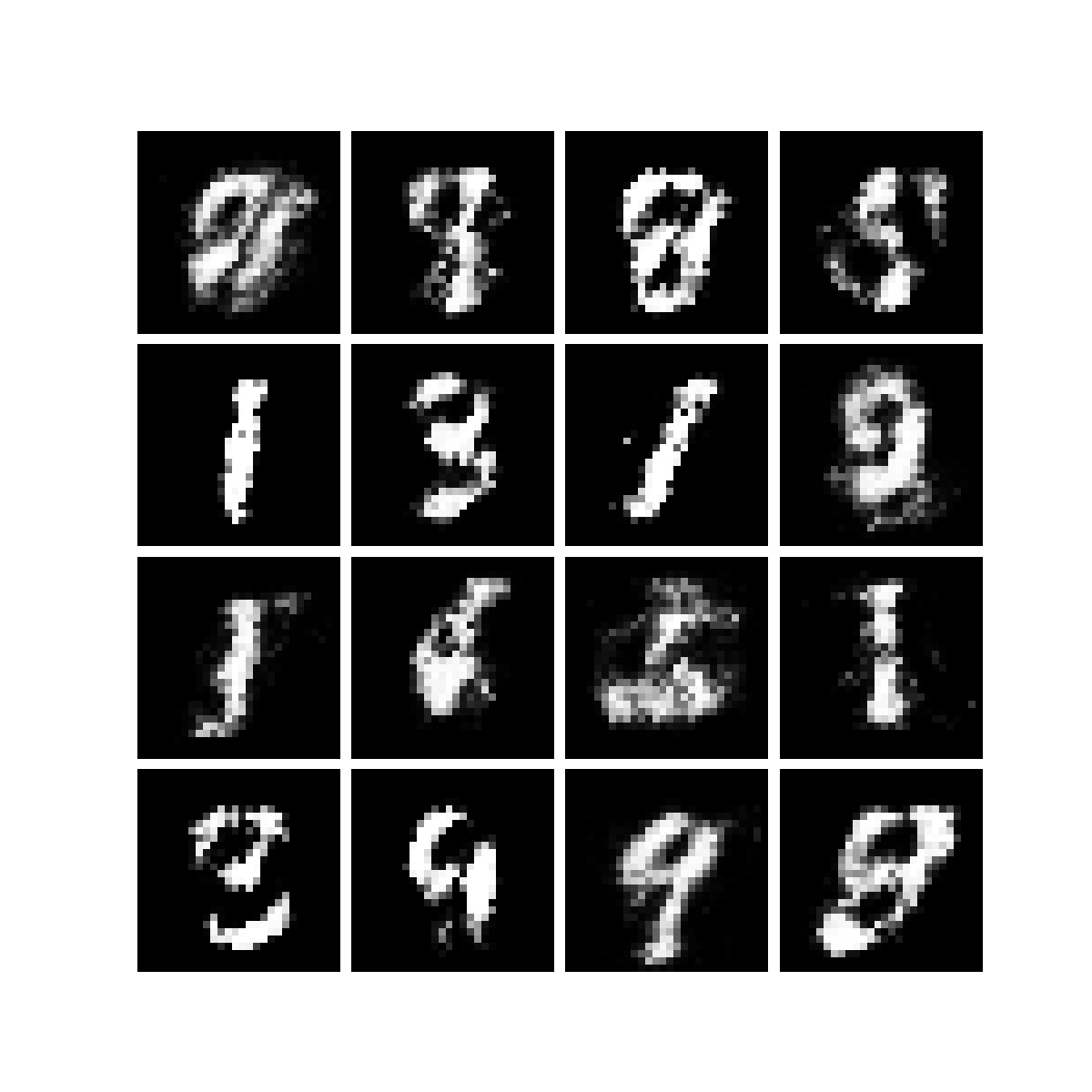
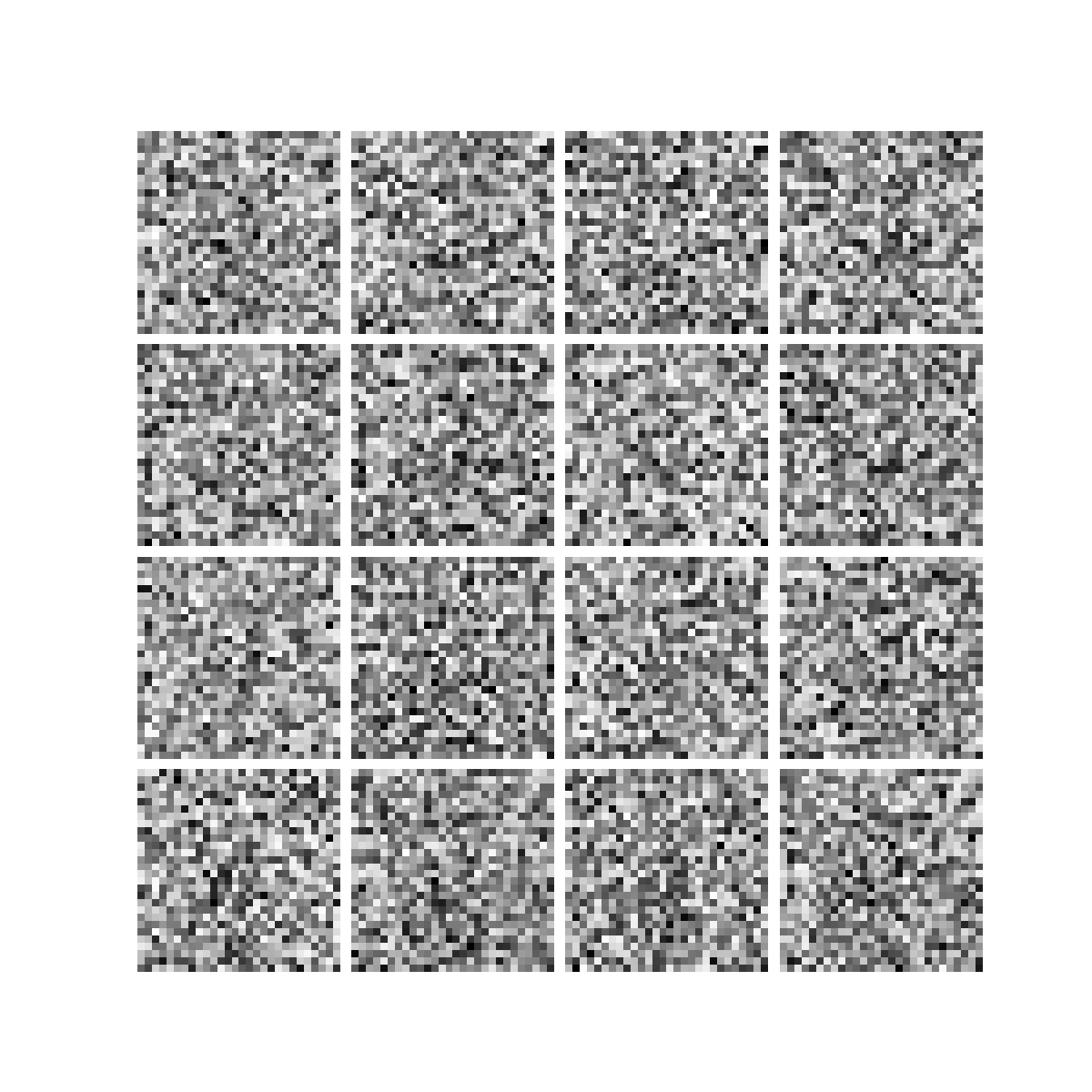
**각각의 Iteration에 해당하는 자세한 Loss 값은 Loss.csv 파일에 있습니다.**

Generator Loss는 초기에 급격히 증가했다가, 학습이 진행될수록 감소하는 경향을 보인다.

Discriminator Loss는 학습이 진행될수록 조금씩 증가하지만, 1 이상의 값으로 올라가지는 않는다.

* **Assignment 1-3**

**Show the generated images at 0, 10000, 60000 iteration**

****

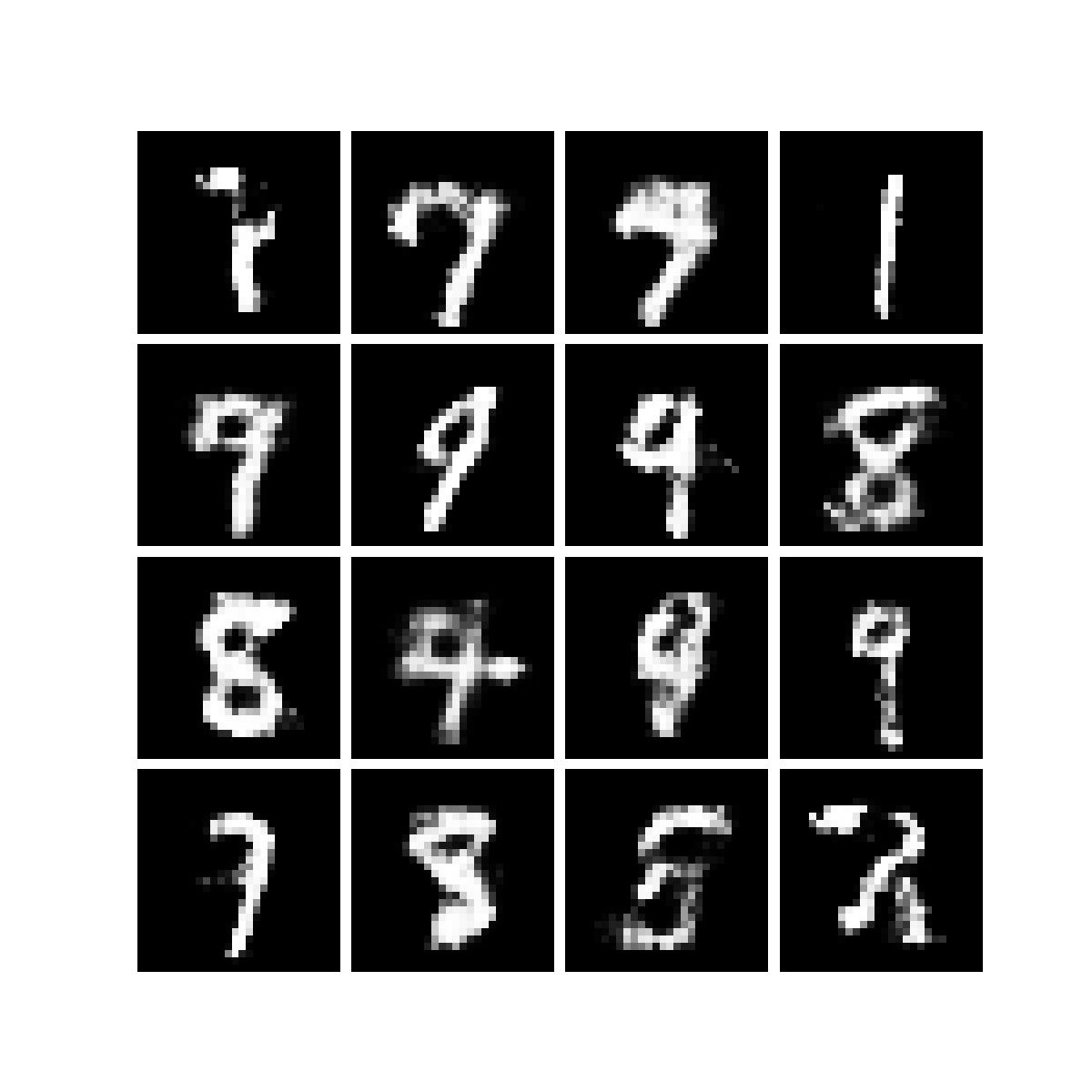
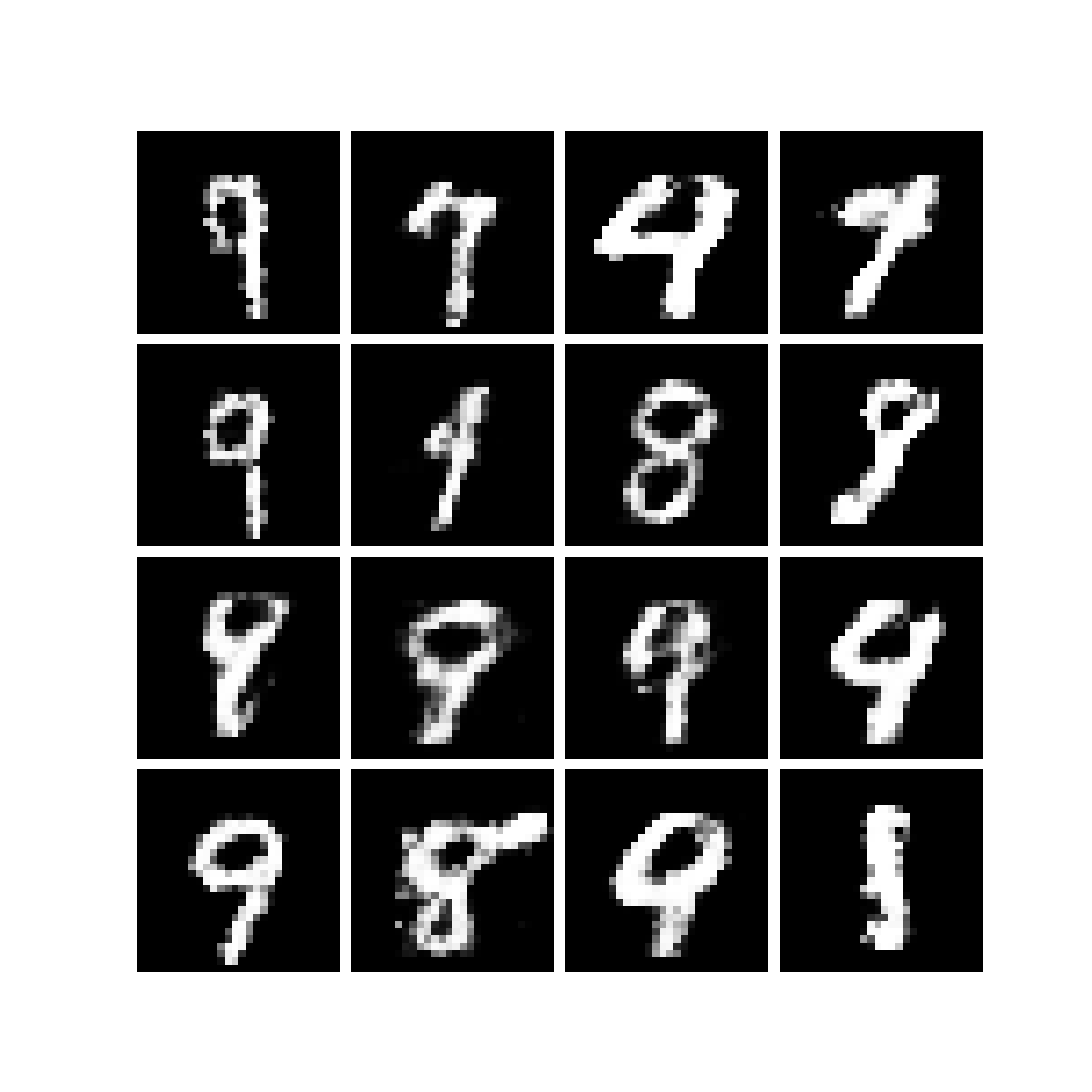
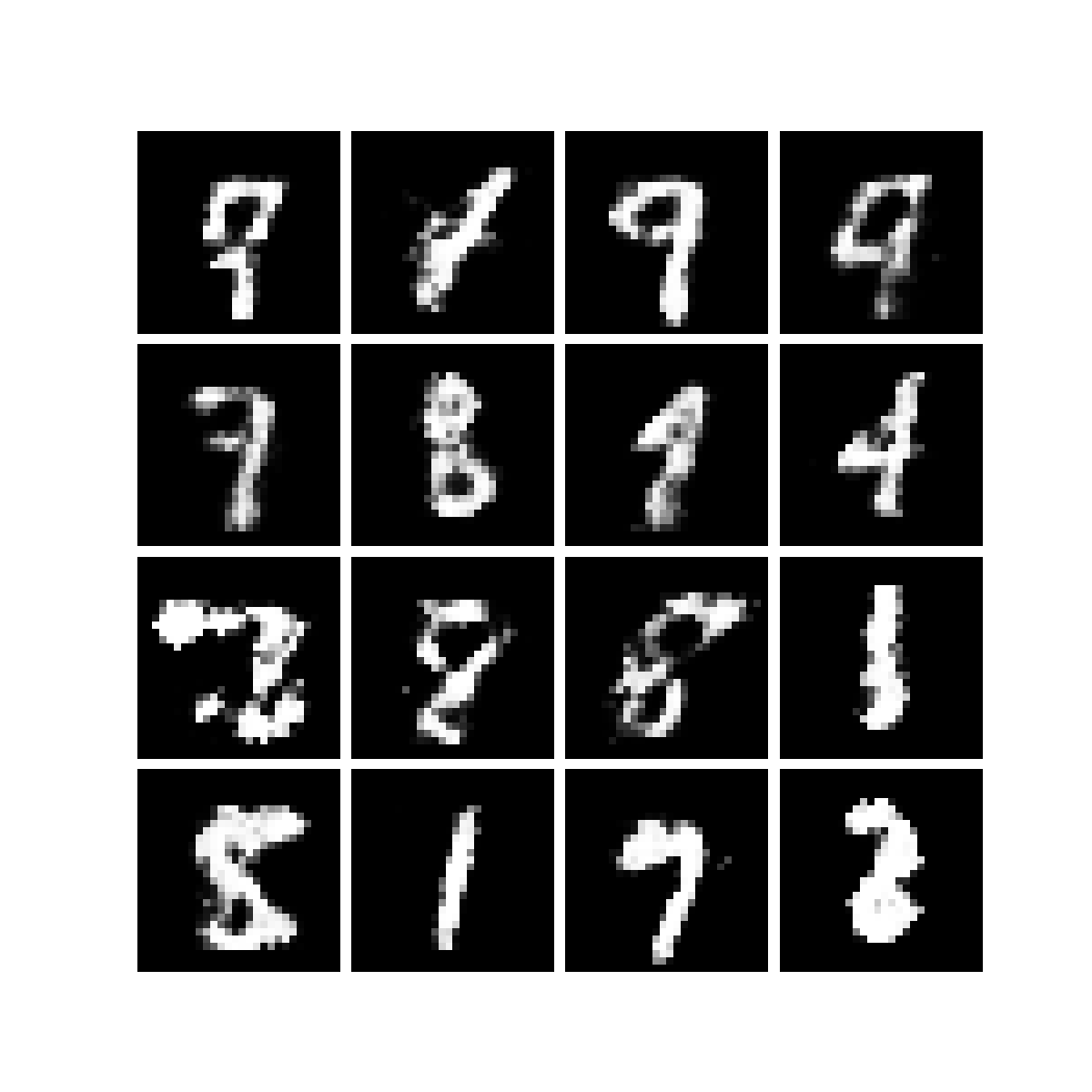
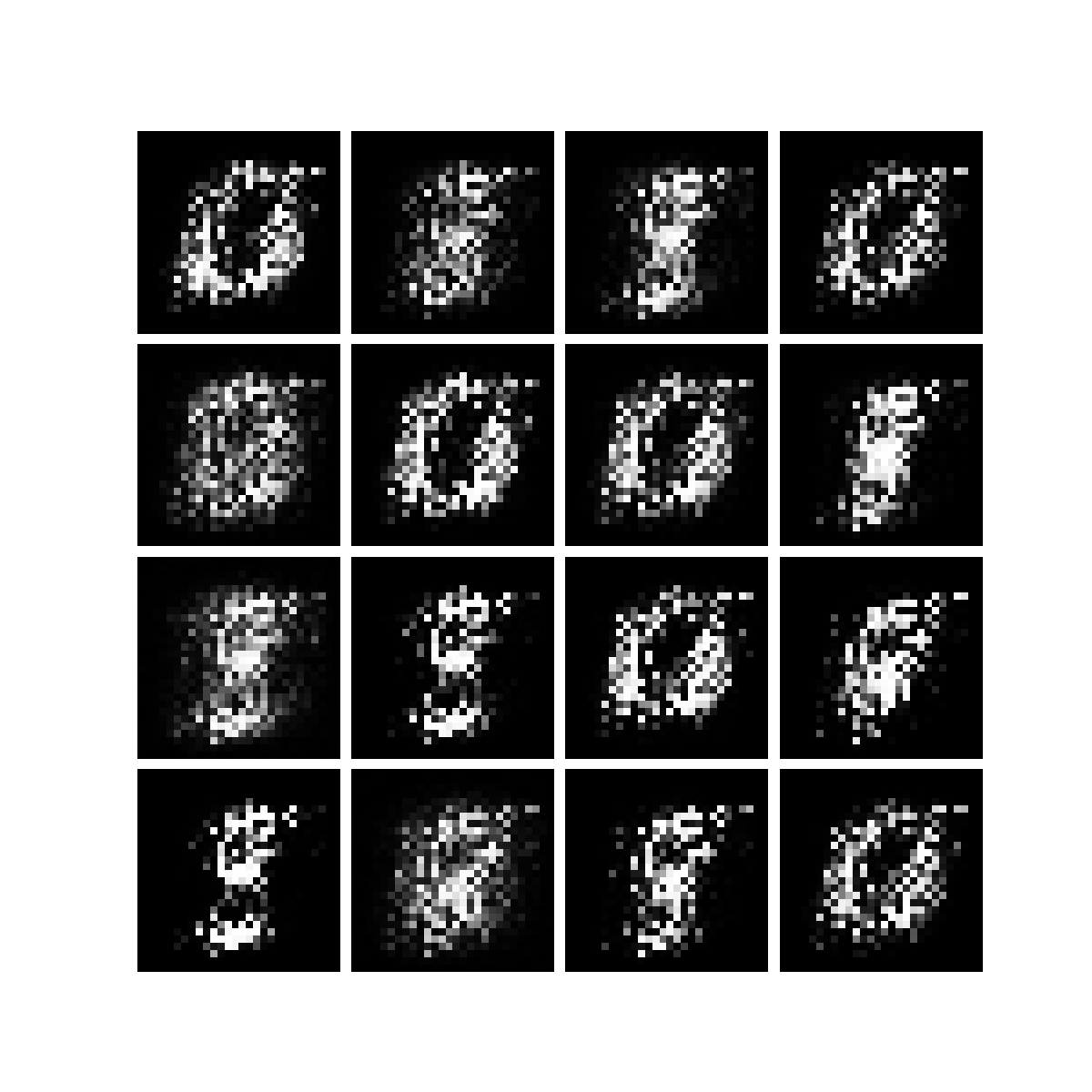
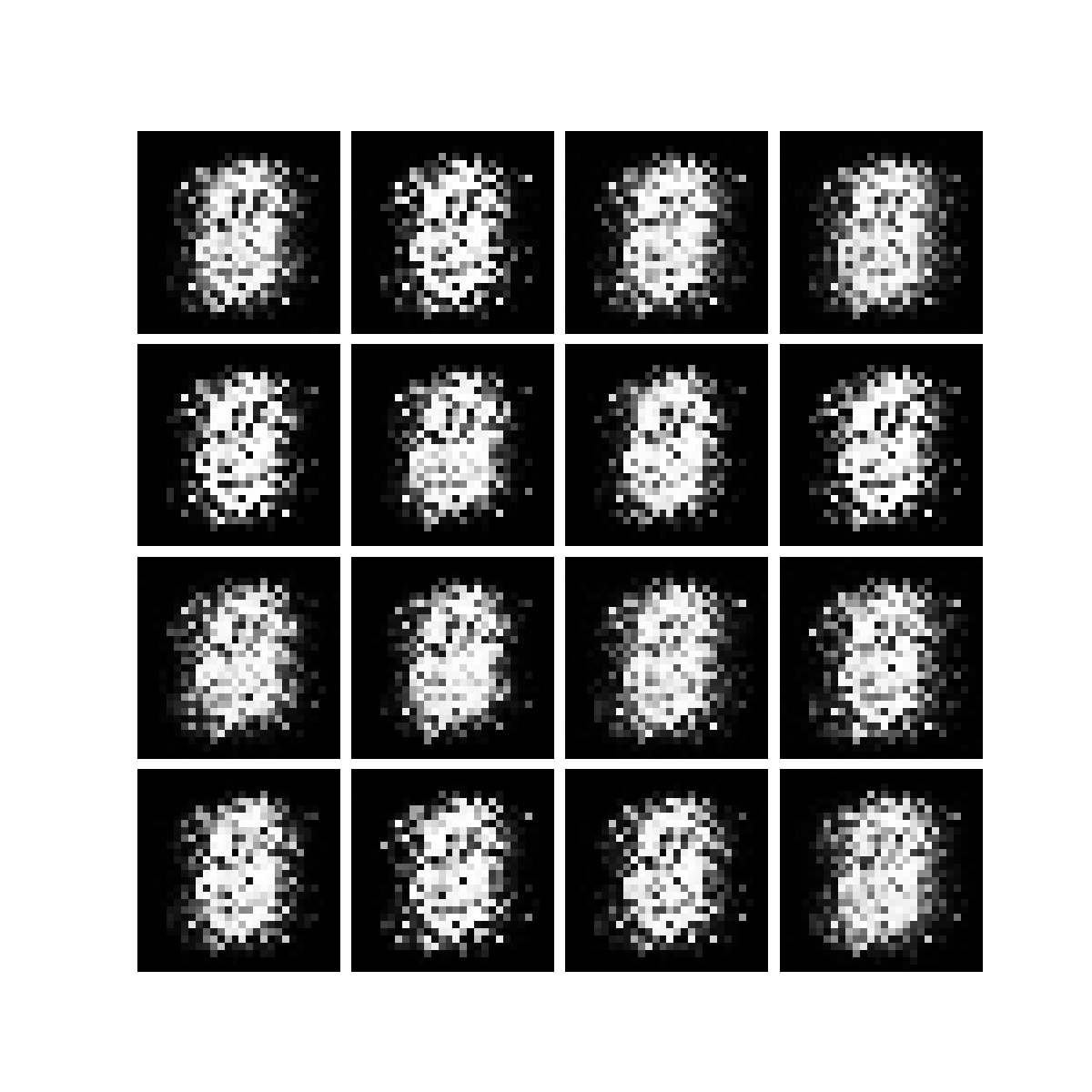
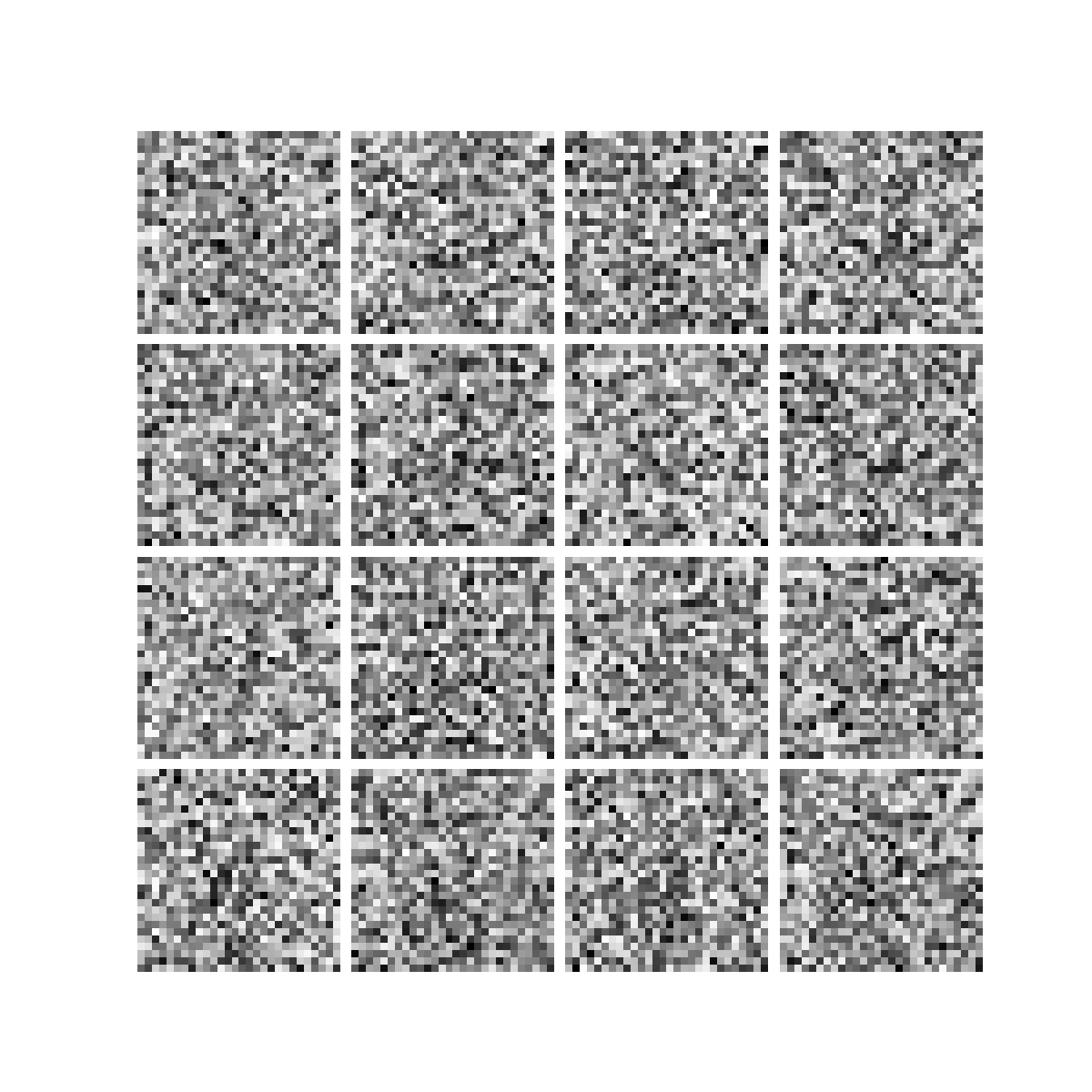
**<그림 (1) - 왼쪽부터 0, 10000, 60000번째 샘플 이미지>**

**각각의 Iteration에 해당하는 샘플 이미지는 sample 폴더에 있습니다.**

학습 초기에 Generator가 생성하는 샘플 이미지는 Z에 의해 만들어진 노이즈에 가깝고, 이후 학습이 진행될수록 노이즈의 경향이 사라지면서 점점 MINIST의 숫자에 가까운 그림을 그리게 되며, 60000번째 Iteration에서는 대부분의 이미지가 숫자로 읽을 수 있는 수준의 그림을 만들어내는 것을 볼 수 있다.

* **Assignment 1-4**

**Discuss experimental result**

****

**<그림 (2) - 0, 1000, 2000, 68000, 69000, 70000 Iteration 샘플 이미지>**

샘플이미지를 통해 Generator network가 input으로 받은 random noise를 MINIST 숫자로 generate 하는 것을 확인 할 수 있다. Generator의 Loss가 줄어들수록 random noise 형상이 사라지며 MINIST 숫자에 가까운 이미지를 만드는 것을 볼 수 있다.

Discriminator는 Generator가 초기에 generate하는 이미지는 잘 구분하여 Loss가 시작부터 극히 적지만 Generator가 학습을 진행하여 Loss가 떨어짐에 따라, Discriminator의 Loss가 조금씩 상승하는 경향을 보이며. 이는 Generator가 생성한 이미지가 Discriminator를 어느정도 속이고 있다는 뜻이다.

결과적으로, 이 실험을 통해 GAN의 핵심인 두 네트워크가 어느 한쪽을 압도하지 않고 서로를 통해 균형을 이루며 학습하고 있다고 이해 할 수 있다.