

**한국외대 기말 과제 보고서**

**과 목 명 인공지능 특강**

**담당교수님 최재영교수님**

**제 출 일 20240617**

**전 공 컴퓨터공학과**

**학 번 202430026**

**이 름 이준용**

* **Analyze a given source code**

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.gridspec as gridspec

from tensorflow.keras.datasets import mnist # 기존 mnist데이터셋 불러오기 패키지가 keras로 바뀌었습니다.

# Eager execution을 비활성화하여 TensorFlow 1.x 기능을 사용

tf.compat.v1.disable\_eager\_execution()

# Xavier 초기화를 위한 함수 정의

def xavier\_init(size):

    input\_dim = size[0]

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

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

# 샘플 이미지를 그리기 위한 함수 정의

def plot(samples):

    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')

    return fig

# 재현성을 위해 랜덤 시드 설정

tf.random.set\_seed(100)

# GPU 사용 가능 여부 확인

print("Num GPUs Available: ", len(tf.config.list\_physical\_devices('GPU')))

# GPU 메모리 사용 설정

gpus = tf.config.list\_physical\_devices('GPU')

if gpus:

    try:

        for gpu in gpus:

            tf.config.experimental.set\_memory\_growth(gpu, True)

        logical\_gpus = tf.config.experimental.list\_logical\_devices('GPU')

        print(len(gpus), "Physical GPUs,", len(logical\_gpus), "Logical GPUs")

    except RuntimeError as e:

        print(e)

# 생성자 네트워크 정의

def generator(z):

    G\_h1 = tf.nn.relu(tf.matmul(z, G\_W1) + G\_b1)

    G\_log\_prob = tf.matmul(G\_h1, G\_W2) + G\_b2

    G\_prob = tf.nn.sigmoid(G\_log\_prob)

    return G\_prob

# 판별자 네트워크 정의

def discriminator(x):

    D\_h1 = tf.nn.relu(tf.matmul(x, D\_W1) + D\_b1)

    D\_logit = tf.matmul(D\_h1, D\_W2) + D\_b2

    D\_prob = tf.nn.sigmoid(D\_logit)

    return D\_prob, D\_logit

# Placeholder 및 변수 설정

Z = tf.compat.v1.placeholder(tf.float32, shape=[None, 100], name='Z')

X = tf.compat.v1.placeholder(tf.float32, shape=[None, 784], name='X')

# 변수 초기화

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

G\_b1 = tf.Variable(tf.zeros([128]), name='G\_b1')

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

G\_b2 = tf.Variable(tf.zeros([784]), name='G\_b2')

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

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

D\_b1 = tf.Variable(tf.zeros([128]), name='D\_b1')

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

D\_b2 = tf.Variable(tf.zeros([1]), name='D\_b2')

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

# 네트워크와 손실 함수 설정

G\_sample = generator(Z)

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

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

# 레이블 스무딩을 사용하여 훈련 안정화

D\_loss\_real = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=D\_logit\_real, labels=tf.ones\_like(D\_logit\_real) \* 0.9))

D\_loss\_fake = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=D\_logit\_fake, labels=tf.zeros\_like(D\_logit\_fake)))

D\_loss = D\_loss\_real + D\_loss\_fake

G\_loss = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=D\_logit\_fake, labels=tf.ones\_like(D\_logit\_fake)))

# Gradient clipping 적용

D\_optimizer = tf.compat.v1.train.AdamOptimizer(learning\_rate=0.0002, beta1=0.5)

G\_optimizer = tf.compat.v1.train.AdamOptimizer(learning\_rate=0.0002, beta1=0.5)

D\_grads\_and\_vars = D\_optimizer.compute\_gradients(D\_loss, var\_list=theta\_D)

D\_grads\_and\_vars = [(tf.clip\_by\_value(grad, -1., 1.), var) for grad, var in D\_grads\_and\_vars if grad is not None]

D\_solver = D\_optimizer.apply\_gradients(D\_grads\_and\_vars)

G\_grads\_and\_vars = G\_optimizer.compute\_gradients(G\_loss, var\_list=theta\_G)

G\_grads\_and\_vars = [(tf.clip\_by\_value(grad, -1., 1.), var) for grad, var in G\_grads\_and\_vars if grad is not None]

G\_solver = G\_optimizer.apply\_gradients(G\_grads\_and\_vars)

# 잠재 변수 샘플링 함수 정의

def sample\_Z(m, n):

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

# 데이터 로드

(train\_images, \_), (\_, \_) = mnist.load\_data()

train\_images = train\_images.astype(np.float32) / 255.

train\_images = train\_images.reshape(-1, 784)

# 세션과 변수 초기화

sess = tf.compat.v1.Session()

sess.run(tf.compat.v1.global\_variables\_initializer())

# 로깅 설정

D\_losses = []

G\_losses = []

sampled\_images = {}

batch\_size = 128

Z\_dim = 100

# 훈련 루프

for itr in range(60001):

    if itr % 1000 == 0:

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

        if itr in [0, 10000, 60000]:

            sampled\_images[itr] = samples

            fig = plot(samples)

            plt.savefig('sample\_{}.png'.format(itr), bbox\_inches='tight')

            plt.close(fig)

            print(f"Saved sample images at iteration {itr}")

    indices = np.random.randint(0, train\_images.shape[0], batch\_size)

    X\_mb = train\_images[indices]

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

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

    if itr % 1000 == 0:

        D\_losses.append(D\_loss\_curr)

        G\_losses.append(G\_loss\_curr)

        print(f'Iter: {itr}, D loss: {D\_loss\_curr:.4f}, G loss: {G\_loss\_curr:.4f}')

# 손실 함수 그래프 그리기

fig, ax = plt.subplots()

ax.plot(D\_losses, label='Discriminator Loss')

ax.plot(G\_losses, label='Generator Loss')

ax.set\_xlabel('Iterations (thousands)')

ax.set\_ylabel('Loss')

ax.legend()

plt.savefig('loss\_plot.png')

plt.show()

# 샘플 이미지 저장

for itr, samples in sampled\_images.items():

    fig = plot(samples)

    plt.savefig('sample\_{}.png'.format(itr), bbox\_inches='tight')

    plt.close(fig)

* + **xavier\_init 함수**

# Xavier 초기화를 위한 함수 정의

def xavier\_init(size):

    input\_dim = size[0]

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

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

* 입력 매개변수:

size: 초기화할 가중치 텐서의 형태를 지정합니다. 예를 들어, size가 [784, 128]이라면 이는 784개의 입력과 128개의 출력을 가지는 가중치 행렬을 초기화하려는 것을 의미합니다.

* 입력 차원 계산:

size 리스트의 첫 번째 요소를 input\_dim 변수에 저장합니다. 이는 입력 뉴런의 수를 의미합니다.

* Xavier 분산 계산:

Xavier 초기화의 표준 편차를 계산합니다. 표준 편차는 1 / sqrt(input\_dim / 2)로 설정됩니다. 이는 입력 뉴런의 수에 따라 가중치의 초기 표준 편차를 조절하여, 모든 레이어에 걸쳐 출력의 분산이 일정하게 유지되도록 합니다.

* 가중치 초기화:

xavier\_variance를 표준 편차로 하는 정규 분포를 사용하여 size 형태의 텐서를 초기화하고 반환합니다. 이는 가중치가 정규 분포를 따르며, 분산이 적절히 조절된 값들로 초기화되도록 합니다.

결론적으로, xavier\_init 함수는 입력 뉴런의 수를 고려하여 가중치 텐서를 정규 분포로 초기화하며, 이 과정에서 Xavier 초기화 방법을 사용하여 신경망 학습의 효율성을 높이고 안정성을 보장합니다.

* **Plot 함수**

# plot 함수

def plot(samples):

    fig = plt.figure(figsize=(4, 4))  # 1. 그림의 크기를 설정하여 새로운 figure 객체 생성

    gs = gridspec.GridSpec(4, 4)  # 2. 4x4 격자(grid) 생성

    gs.update(wspace=0.05, hspace=0.05)  # 3. 격자 간의 공백 설정

    for i, sample in enumerate(samples):  # 4. 샘플 이미지를 순회하면서 각 격자에 배치

        ax = plt.subplot(gs[i])  # 5. 격자 위치에 subplot 생성

        plt.axis('off')  # 6. 축을 끔

        ax.set\_xticklabels([])  # 7. x축 레이블을 비움

        ax.set\_yticklabels([])  # 8. y축 레이블을 비움

        ax.set\_aspect('equal')  # 9. 축의 비율을 동일하게 설정

        plt.imshow(sample.reshape(28, 28), cmap='Greys\_r')  # 10. 이미지를 28x28 크기로 변경하여 흑백으로 표시

    return fig  # 11. 생성된 figure 객체 반환

plot 함수는 샘플 이미지를 그리기 위한 함수로, 주어진 이미지 샘플들을 4x4 격자(grid) 형태로 배열하여 시각화합니다. 이 함수는 특히 MNIST 데이터셋처럼 28x28 픽셀 크기의 흑백 이미지를 그립니다.

* **Generator & discrimintor 함수**

# 생성자 네트워크 정의

def generator(z):

    G\_h1 = tf.nn.relu(tf.matmul(z, G\_W1) + G\_b1)

    G\_log\_prob = tf.matmul(G\_h1, G\_W2) + G\_b2

    G\_prob = tf.nn.sigmoid(G\_log\_prob)

    return G\_prob

generator와 discriminator 함수는 각각 생성자 네트워크와 판별자 네트워크를 정의합니다. 이 네트워크들은 생성적 적대 신경망(GAN)에서 사용되며, 생성자는 새로운 데이터 샘플을 생성하고 판별자는 진짜 데이터와 가짜 데이터를 구별하는 역할을 합니다.

1. 입력:

z: 잠재 공간(latent space)에서 샘플링된 입력 벡터. 일반적으로 정규 분포 또는 균등 분포에서 샘플링된 랜덤 벡터입니다.

1. 첫 번째 은닉층:

입력 벡터 z와 가중치 행렬 G\_W1을 곱하고 편향 G\_b1를 더한 후, ReLU 활성화 함수를 적용합니다. 이는 첫 번째 은닉층의 출력을 계산합니다.

1. 출력층:

첫 번째 은닉층의 출력 G\_h1와 가중치 행렬 G\_W2를 곱하고 편향 G\_b2를 더하여 출력층의 로그 확률을 계산합니다.

출력층의 로그 확률에 시그모이드 활성화 함수를 적용하여 최종 출력을 계산합니다. 이 출력은 0과 1 사이의 값을 가지며, 생성된 가짜 데이터 샘플을 나타냅니다.

1. 출력:

생성된 가짜 데이터 샘플을 반환합니다.

# 판별자 네트워크 정의

def discriminator(x):

    D\_h1 = tf.nn.relu(tf.matmul(x, D\_W1) + D\_b1)

    D\_logit = tf.matmul(D\_h1, D\_W2) + D\_b2

    D\_prob = tf.nn.sigmoid(D\_logit)

    return D\_prob, D\_logit

1. **입력:**

x: 진짜 데이터 샘플 또는 생성자가 생성한 가짜 데이터 샘플.

1. **첫 번째 은닉층:**

입력 벡터 x와 가중치 행렬 D\_W1을 곱하고 편향 D\_b1를 더한 후, ReLU 활성화 함수를 적용합니다. 이는 첫 번째 은닉층의 출력을 계산합니다.

1. **출력층:**

첫 번째 은닉층의 출력 D\_h1와 가중치 행렬 D\_W2를 곱하고 편향 D\_b2를 더하여 출력층의 로그 확률을 계산합니다.

출력층의 로그 확률에 시그모이드 활성화 함수를 적용하여 최종 출력을 계산합니다. 이 출력은 0과 1 사이의 값을 가지며, 입력 데이터가 진짜일 확률을 나타냅니다.

1. **출력:**

D\_prob는 입력 데이터가 진짜일 확률을 나타내고, D\_logit는 그에 대한 로그 확률을 나타냅니다. 두 값을 모두 반환합니다.

* **Placeholder 설정**

# Placeholder 및 변수 설정

Z = tf.compat.v1.placeholder(tf.float32, shape=[None, 100], name='Z')

X = tf.compat.v1.placeholder(tf.float32, shape=[None, 784], name='X')

* 생성자 네트워크의 입력을 위한 placeholder입니다. 이는 잠재 공간(latent space)에서 샘플링된 100차원의 랜덤 벡터를 나타냅니다. None은 배치 크기를 나타내며, 하나의 배치 내에 몇 개의 샘플이 들어갈지 미리 정해지지 않았음을 의미합니다.
* 판별자 네트워크의 입력을 위한 placeholder입니다. 이는 28x28 크기의 이미지가 펼쳐진 784차원의 벡터를 나타냅니다. 마찬가지로 None은 배치 크기를 나타냅니다.
* **변수 초기화**

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

G\_b1 = tf.Variable(tf.zeros([128]), name='G\_b1')

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

G\_b2 = tf.Variable(tf.zeros([784]), name='G\_b2')

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

* 생성자 네트워크의 첫 번째 가중치 행렬입니다. 입력 크기는 100, 출력 크기는 128입니다. Xavier 초기화를 사용하여 초기화됩니다.
* 생성자 네트워크의 첫 번째 편향 벡터입니다. 128차원으로 초기값은 0입니다.
* 생성자 네트워크의 두 번째 가중치 행렬입니다. 입력 크기는 128, 출력 크기는 784입니다. Xavier 초기화를 사용하여 초기화됩니다.
* 생성자 네트워크의 두 번째 편향 벡터입니다. 784차원으로 초기값은 0입니다.
* 생성자 네트워크의 모든 가중치와 편향을 포함하는 리스트입니다. 이는 나중에 최적화할 때 사용됩니다.

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

D\_b1 = tf.Variable(tf.zeros([128]), name='D\_b1')

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

D\_b2 = tf.Variable(tf.zeros([1]), name='D\_b2')

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

* 판별자 네트워크의 첫 번째 가중치 행렬입니다. 입력 크기는 784, 출력 크기는 128입니다. Xavier 초기화를 사용하여 초기화됩니다.
* 판별자 네트워크의 첫 번째 편향 벡터입니다. 128차원으로 초기값은 0입니다.
* 판별자 네트워크의 두 번째 가중치 행렬입니다. 입력 크기는 128, 출력 크기는 1입니다. Xavier 초기화를 사용하여 초기화됩니다.
* 판별자 네트워크의 두 번째 편향 벡터입니다. 1차원으로 초기값은 0입니다.
* 판별자 네트워크의 모든 가중치와 편향을 포함하는 리스트입니다. 이는 나중에 최적화할 때 사용됩니다.
* **네트워크와 손실 함수**

# 네트워크와 손실 함수 설정

G\_sample = generator(Z)

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

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

**생성자 네트워크 출력**:

* 잠재 공간의 랜덤 벡터 Z를 생성자 네트워크에 입력하여 가짜 데이터 샘플 G\_sample을 생성합니다.

**판별자 네트워크 (진짜 데이터)**:

* 실제 데이터 X를 판별자 네트워크에 입력하여, 진짜 데이터에 대한 판별 확률 D\_real과 로그 확률 D\_logit\_real을 얻습니다.

**판별자 네트워크 (가짜 데이터)**:

* 생성된 가짜 데이터 G\_sample을 판별자 네트워크에 입력하여, 가짜 데이터에 대한 판별 확률 D\_fake와 로그 확률 D\_logit\_fake을 얻습니다.
* **레이블 스무딩**

# 레이블 스무딩을 사용하여 훈련 안정화

D\_loss\_real = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=D\_logit\_real, labels=tf.ones\_like(D\_logit\_real) \* 0.9))

D\_loss\_fake = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=D\_logit\_fake, labels=tf.zeros\_like(D\_logit\_fake)))

D\_loss = D\_loss\_real + D\_loss\_fake

G\_loss = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=D\_logit\_fake, labels=tf.ones\_like(D\_logit\_fake)))

**진짜 데이터에 대한 손실 (D\_loss\_real)**:

* 진짜 데이터를 진짜로 판별하도록 학습하는 손실을 계산합니다. labels로 tf.ones\_like(D\_logit\_real) \* 0.9를 사용하여 레이블 스무딩(label smoothing)을 적용합니다. 레이블 스무딩은 네트워크가 1이 아닌 0.9를 목표로 학습하게 하여, 모델이 과적합하는 것을 방지하고 안정적인 학습을 돕습니다.

**가짜 데이터에 대한 손실 (D\_loss\_fake)**:

* 가짜 데이터를 가짜로 판별하도록 학습하는 손실을 계산합니다. labels로 tf.zeros\_like(D\_logit\_fake)를 사용하여 가짜 데이터에 대해 0을 목표로 학습하게 합니다.

**판별자 총 손실 (D\_loss)**:

* 진짜 데이터와 가짜 데이터에 대한 손실을 합하여 판별자의 총 손실을 계산합니다.

**생성자 손실 (G\_loss)**:

* 생성된 가짜 데이터를 진짜로 판별하도록 학습하는 손실을 계산합니다. labels로 tf.ones\_like(D\_logit\_fake)를 사용하여 생성자가 생성한 가짜 데이터가 판별자에 의해 진짜로 인식되도록 학습합니다.
* **Gradient clipping**

# Gradient clipping 적용

D\_optimizer = tf.compat.v1.train.AdamOptimizer(learning\_rate=0.0002, beta1=0.5)

G\_optimizer = tf.compat.v1.train.AdamOptimizer(learning\_rate=0.0002, beta1=0.5)

* 판별자 네트워크의 최적화 알고리즘으로 Adam Optimizer를 사용합니다. 학습률(learning\_rate)은 0.0002로 설정하고, 모멘텀 매개변수(beta1)는 0.5로 설정합니다. 이는 안정적인 학습을 위해 흔히 사용하는 설정입니다.
* 생성자 네트워크의 최적화 알고리즘으로 Adam Optimizer를 사용합니다. 학습률과 모멘텀 매개변수는 동일하게 설정합니다.

D\_grads\_and\_vars = D\_optimizer.compute\_gradients(D\_loss, var\_list=theta\_D)

D\_grads\_and\_vars = [(tf.clip\_by\_value(grad, -1., 1.), var) for grad, var in D\_grads\_and\_vars if grad is not None]

D\_solver = D\_optimizer.apply\_gradients(D\_grads\_and\_vars)

G\_grads\_and\_vars = G\_optimizer.compute\_gradients(G\_loss, var\_list=theta\_G)

G\_grads\_and\_vars = [(tf.clip\_by\_value(grad, -1., 1.), var) for grad, var in G\_grads\_and\_vars if grad is not None]

G\_solver = G\_optimizer.apply\_gradients(G\_grads\_and\_vars)

**그라디언트 계산**:

* 판별자 네트워크의 손실 D\_loss에 대한 그라디언트를 계산합니다. var\_list로 판별자의 모든 가중치와 편향 (theta\_D)을 지정합니다.
* 생성자 네트워크의 손실 G\_loss에 대한 그라디언트를 계산합니다. var\_list로 생성자의 모든 가중치와 편향 (theta\_G)을 지정합니다.

**그라디언트 클리핑**:

* 계산된 그라디언트 값 grad를 -1에서 1 사이로 클리핑합니다. 클리핑된 그라디언트와 해당 변수를 다시 리스트에 저장합니다.
* 계산된 그라디언트 값 grad를 -1에서 1 사이로 클리핑합니다. 클리핑된 그라디언트와 해당 변수를 다시 리스트에 저장합니다.

**그라디언트 적용**:

* 클리핑된 그라디언트를 사용하여 판별자와 생성자 네트워크의 가중치와 편향을 업데이트합니다.

# 세션과 변수 초기화

sess = tf.compat.v1.Session()

sess.run(tf.compat.v1.global\_variables\_initializer())

**세션 생성**:

* 계산 그래프를 실행하는 세션 생성

**변수 초기화**:

* 모델의 모든 변수를 초기화합니다. 이 함수는 그래프에 정의된 변수들을 초기값으로 설정합니다. 여기서 사용된 변수는 생성자와 판별자의 가중치와 편향입니다.

# 로깅 설정

D\_losses = []

G\_losses = []

sampled\_images = {}

batch\_size = 128

Z\_dim = 100

**손실 기록 리스트 초기화**:

* 판별자의 손실 값을 저장할 빈 리스트를 초기화합니다. 학습이 진행되면서 각 학습 단계의 손실 값을 저장합니다.
* 생성자의 손실 값을 저장할 빈 리스트를 초기화합니다. 마찬가지로 각 학습 단계의 손실 값을 저장합니다.

**샘플 이미지 저장 딕셔너리 초기화**:

* 학습 도중 생성된 샘플 이미지를 저장할 딕셔너리를 초기화합니다. 학습 과정에서 특정 반복마다 생성된 이미지를 저장하여 나중에 시각화할 수 있습니다.

**배치 크기 설정**:

* 학습 과정에서 사용할 배치 크기를 128로 설정합니다. 이는 한 번의 학습 단계에서 사용할 데이터 샘플의 수를 의미합니다.

**잠재 공간 차원 설정**:

* 생성자 네트워크에 입력으로 사용할 잠재 공간의 차원을 100으로 설정합니다. 이는 생성자가 학습에 사용할 랜덤 벡터의 차원 수를 의미합니다.
* **훈련 루프**

# 훈련 루프

for itr in range(60001):

    if itr % 1000 == 0:

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

        if itr in [0, 10000, 60000]:

            sampled\_images[itr] = samples

            fig = plot(samples)

            plt.savefig('sample\_{}.png'.format(itr), bbox\_inches='tight')

            plt.close(fig)

            print(f"Saved sample images at iteration {itr}")

    indices = np.random.randint(0, train\_images.shape[0], batch\_size)

    X\_mb = train\_images[indices]

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

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

    if itr % 1000 == 0:

        D\_losses.append(D\_loss\_curr)

        G\_losses.append(G\_loss\_curr)

        print(f'Iter: {itr}, D loss: {D\_loss\_curr:.4f}, G loss: {G\_loss\_curr:.4f}')

**특정 반복마다 샘플 생성**:

1000번의 반복마다 샘플 이미지를 생성합니다.

\_Z 함수를 사용하여 16개의 잠재 변수를 샘플링하고, 이를 생성자 네트워크에 입력하여 샘플 이미지를 생성합니다.

**샘플 이미지 저장**:

반복 번호가 0, 10000, 60000일 때 샘플 이미지를 저장합니다.

생성된 샘플 이미지를 sampled\_images 딕셔너리에 저장합니다.

plot 함수를 사용하여 샘플 이미지를 플롯으로 생성합니다.

생성된 이미지를 파일로 저장합니다. 파일 이름은 sample\_{iteration}.png 형식입니다.

플롯을 닫아 리소스를 해제합니다.

샘플 이미지가 저장되었음을 콘솔에 출력합니다.

**훈련 배치 준비**:

* 데이터에서 무작위로 batch\_size (여기서는 128)개의 인덱스를 샘플링합니다.
* 샘플링된 인덱스를 사용하여 배치 데이터를 준비합니다. X\_mb는 현재 배치의 이미지 데이터를 나타냅니다.

**판별자 손실 및 업데이트**:

* 판별자 네트워크를 업데이트하고 현재 판별자 손실을 계산합니다. D\_solver는 판별자의 파라미터를 업데이트하며, D\_loss는 현재 판별자 손실을 나타냅니다. 피드 딕셔너리(feed\_dict)는 실제 데이터 X\_mb와 생성자 입력 sample\_Z(batch\_size, Z\_dim)을 포함합니다.

**생성자 손실 및 업데이트**:

* 생성자 네트워크를 업데이트하고 현재 생성자 손실을 계산합니다. G\_solver는 생성자의 파라미터를 업데이트하며, G\_loss는 현재 생성자 손실을 나타냅니다. 피드 딕셔너리에는 생성자 입력 sample\_Z(batch\_size, Z\_dim)만 포함됩니다.

**손실 값 로깅**:

* 1000번의 반복마다 손실 값을 로깅합니다.
* 현재 판별자 손실을 D\_losses 리스트에 추가합니다.
* 현재 생성자 손실을 G\_losses 리스트에 추가합니다.
* 현재 반복 번호와 판별자 및 생성자 손실 값을 콘솔에 출력합니다.
* 손실 함수 그래프 출력

# 손실 함수 그래프 그리기

fig, ax = plt.subplots()

ax.plot(D\_losses, label='Discriminator Loss')

ax.plot(G\_losses, label='Generator Loss')

ax.set\_xlabel('Iterations (thousands)')

ax.set\_ylabel('Loss')

ax.legend()

plt.savefig('loss\_plot.png')

plt.show()

# 샘플 이미지 저장

for itr, samples in sampled\_images.items():

    fig = plot(samples)

    plt.savefig('sample\_{}.png'.format(itr), bbox\_inches='tight')

    plt.close(fig)

플롯 및 축 생성을 하여 라벨과 범례를 추가한 그래프를 저장하고 출력하는 코드 입니다.

샘플 이미지 반복하여 이미지 플롯을 생성합니다.

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

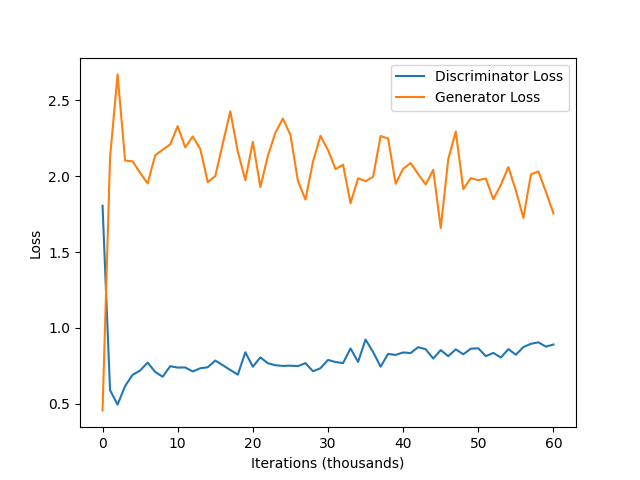
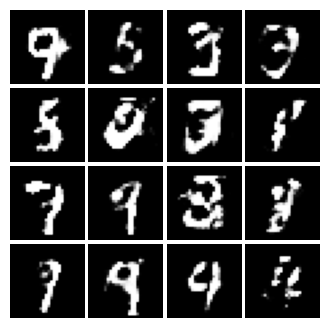
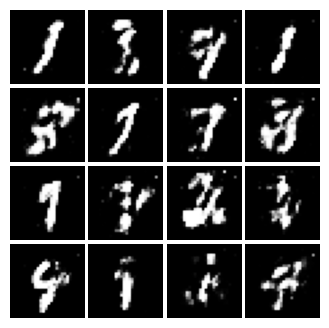
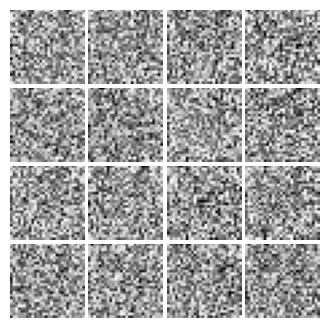


Figure 1 : discriminator and Generator Loss graph

* **Show the generated images at 0, 10,000, 60,000 iterations**

****

<0 iterations> < 10000 iterations> <60000 iterations>

Figure 2 : 0, 10000, 60000 iterations mnist images

* **Discuss experimental results**

**기본 GAN과 이 코드의 차이점**

**1. 레이블 스무딩 (Label Smoothing)**

* **기본 GAN**: 일반적으로 진짜 데이터에 대한 레이블은 1, 가짜 데이터에 대한 레이블은 0으로 설정됩니다.
* **이 코드**: 진짜 데이터에 대한 레이블을 1이 아닌 0.9로 설정하는 레이블 스무딩 기법을 사용합니다. 이는 판별자가 진짜 데이터를 완벽하게 분류하지 못하도록 하여, 모델이 과적합하는 것을 방지하고 더 안정적인 학습을 도모합니다.

D\_loss\_real = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=D\_logit\_real, labels=tf.ones\_like(D\_logit\_real) \* 0.9))

**2. 그라디언트 클리핑 (Gradient Clipping)**

* **기본 GAN**: 일반적으로 그라디언트 클리핑을 사용하지 않습니다.
* **이 코드**: 그라디언트 클리핑을 사용하여 그라디언트 값이 일정 범위를 넘지 않도록 합니다. 이는 그라디언트 폭주 문제를 방지하고, 안정적인 학습을 가능하게 합니다.

D\_grads\_and\_vars = [(tf.clip\_by\_value(grad, -1., 1.), var) for grad, var in D\_grads\_and\_vars if grad is not None]

G\_grads\_and\_vars = [(tf.clip\_by\_value(grad, -1., 1.), var) for grad, var in G\_grads\_and\_vars if grad is not None]

**3. 최적화 함수 설정**

* **기본 GAN**: 다양한 최적화 방법을 사용할 수 있지만, Adam 옵티마이저를 사용하는 것이 일반적입니다.
* **이 코드**: Adam 옵티마이저를 사용하고, 학습률과 모멘텀 파라미터를 설정하여 학습의 안정성을 높입니다.

D\_optimizer = tf.compat.v1.train.AdamOptimizer(learning\_rate=0.0002, beta1=0.5)

G\_optimizer = tf.compat.v1.train.AdamOptimizer(learning\_rate=0.0002, beta1=0.5)

Discuss experimental results

**1. 손실 함수 그래프**

* **그래프 해석**: 그래프를 보면, 판별자 손실(D loss)은 처음에 급격히 감소한 후 비교적 안정적인 값을 유지하고 있습니다. 반면, 생성자 손실(G loss)은 비교적 높은 값을 유지하며 변동이 심합니다. 이는 판별자가 점점 더 가짜 데이터를 잘 구분하게 되면서, 생성자가 더 어렵게 진짜 같은 데이터를 생성해야 하기 때문입니다.

**2. 0, 10000, 60000 반복(iteration) 후 생성된 샘플 이미지**

* **0 반복**: 초기에는 랜덤한 노이즈만 생성되며, 전혀 의미 있는 이미지를 만들어내지 못합니다.
* **10000 반복**: 몇 가지 숫자 모양이 나타나기 시작하지만, 여전히 왜곡이 많고 일부 이미지만 숫자처럼 보입니다.
* **60000 반복**: 대부분의 이미지가 숫자 모양을 갖추고 있으며, MNIST 데이터셋의 손으로 쓴 숫자처럼 보입니다. 그러나 일부 이미지는 여전히 왜곡되어 있습니다.

**결론**

* **레이블 스무딩**: 레이블 스무딩을 사용함으로써 판별자가 너무 강력해지는 것을 방지하고, 생성자가 더 좋은 결과를 얻도록 추가하였습니다.
* **그라디언트 클리핑**: 그라디언트 폭주를 방지하여 학습의 안정성을 높이고, 손실 값의 급격한 변동을 줄이는 것을 추가하였습니다.
* **결과 분석**: 반복 횟수가 증가함에 따라 생성된 샘플 이미지의 품질이 점차 향상됩니다. 초기에는 랜덤한 노이즈만 생성되지만, 학습이 진행될수록 숫자 모양을 갖춘 이미지를 생성하게 됩니다. 이는 GAN이 학습을 통해 점차 진짜 같은 이미지를 생성하는 능력을 키워가는 과정을 보여줍니다.