

Final-Term Exam Project

Implementing Basic Generative Adversarial Network (GAN)

Prof. Jae Young Choi

Pattern Recognition and Machine Intelligence Lab. (PMI)

Hankuk University of Foreign Studies

The building blocks of GAN

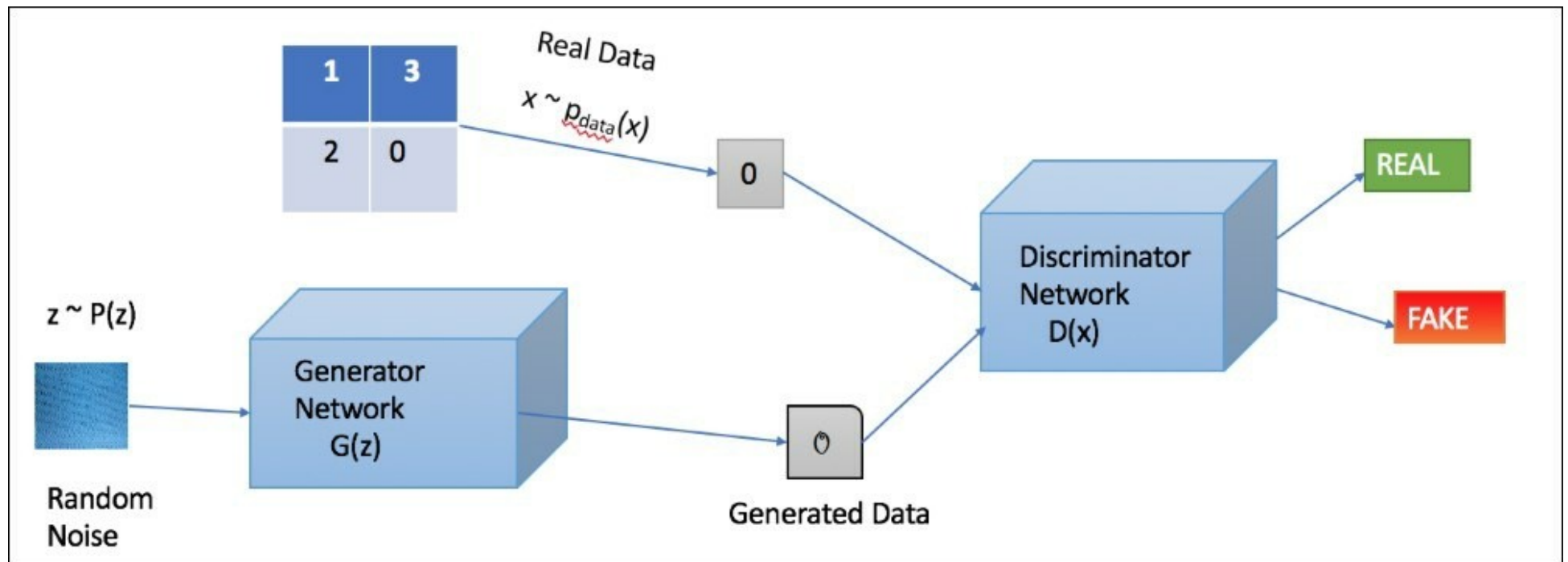
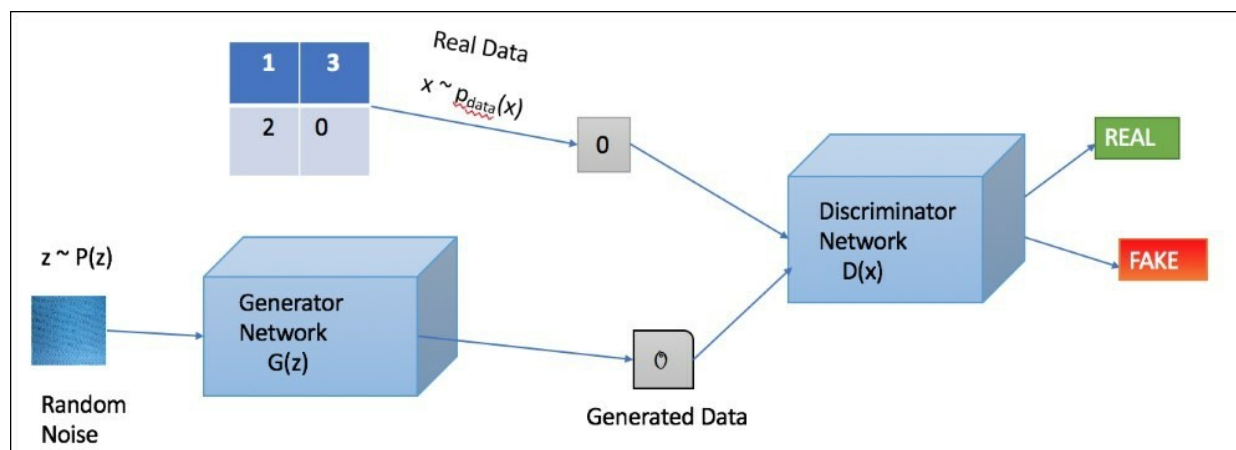


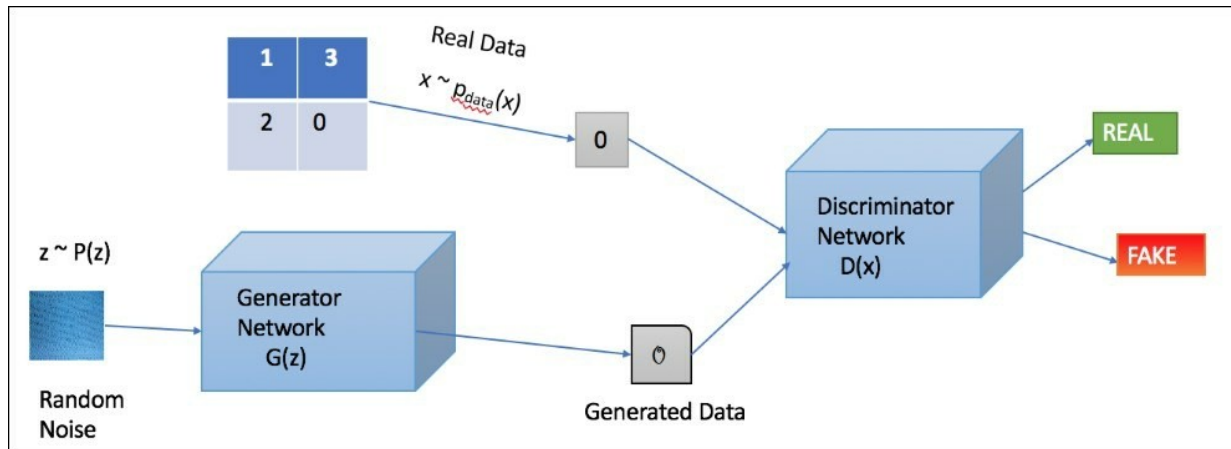
Figure 1b: Generative adversarial network

Generator



The generator network takes as input random noise and tries to generate a sample of data. In the preceding figure, we can see that generator $G(z)$ takes an input z from probability distribution $P(z)$ and generates data that is then fed into a discriminator network $D(x)$.

Discriminator

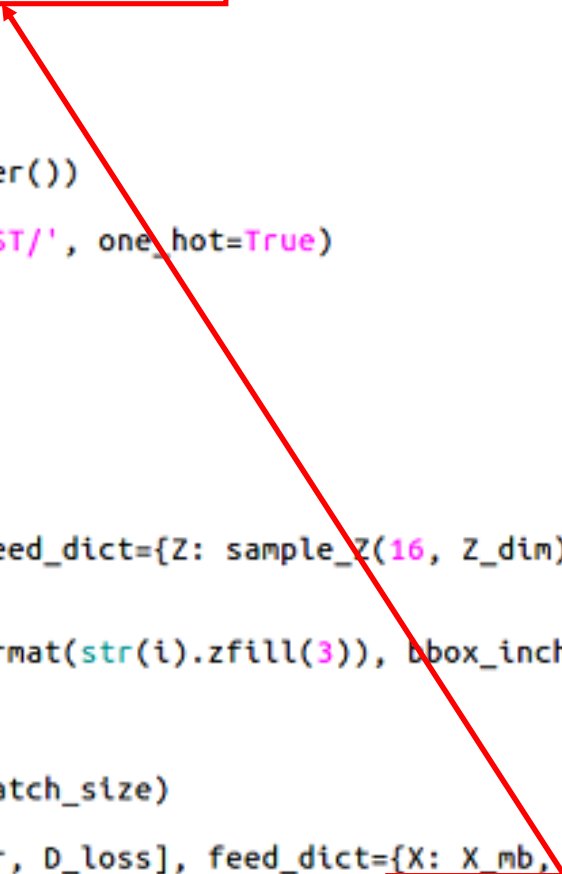


The discriminator network takes input either from the real data or from the generator's generated data and tries to predict whether the input is real or generated. It takes an input x from real data distribution $P_{data}(x)$ and then solves a binary classification problem giving output in the scalar range 0 to 1.

What is input 'z' ?

Sess run

```
98 def sample_Z(m, n):
99     return np.random.uniform(-1., 1., size=[m, n])
100
101 batch_size = 128
102 Z_dim = 100
103
104 sess = tf.Session()
105 sess.run(tf.global_variables_initializer())
106
107 mnist = input_data.read_data_sets('MNIST/', one_hot=True)
108
109 if not os.path.exists('output/'):
110     os.makedirs('output/')
111
112 i = 0
113
114 for itr in range(1000000):
115     if itr % 1000 == 0:
116         samples = sess.run(G_sample, feed_dict={Z: sample_Z(16, Z_dim)})
117
118         fig = plot(samples)
119         plt.savefig('output/{}.png'.format(str(i).zfill(3)), bbox_inches='tight')
120         i += 1
121         plt.close(fig)
122
123     X_mb, _ = mnist.train.next_batch(batch_size)
124
125     _, D_loss_curr = sess.run([D_solver, D_loss], feed_dict={X: X_mb, Z: sample_Z(batch_size, Z_dim)})
126     _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z: sample_Z(batch_size, Z_dim)})
```



numpy.random.uniform

`numpy.random.uniform` (*low=0.0, high=1.0, size=None*)

Draw samples from a `uniform` distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by `uniform`.

Parameters: `low` : *float or array_like of floats, optional*

Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

`high` : *float or array_like of floats*

Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

`size` : *int or tuple of ints, optional*

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

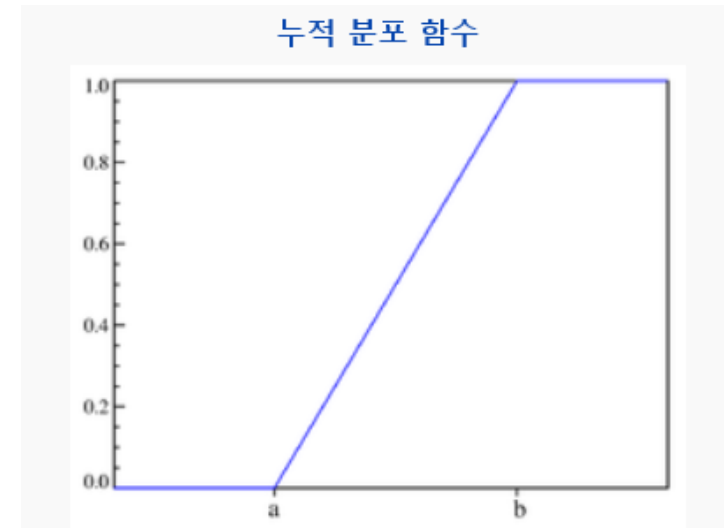
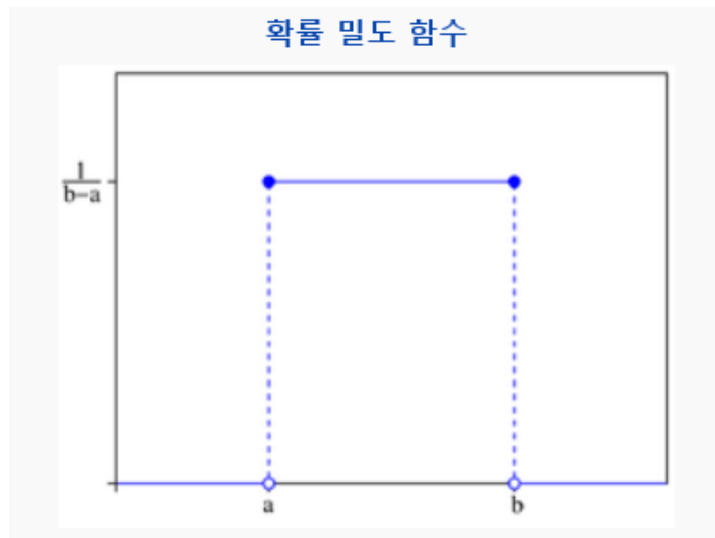
Returns:

`out` : *ndarray or scalar*

Drawn samples from the parameterized uniform distribution.

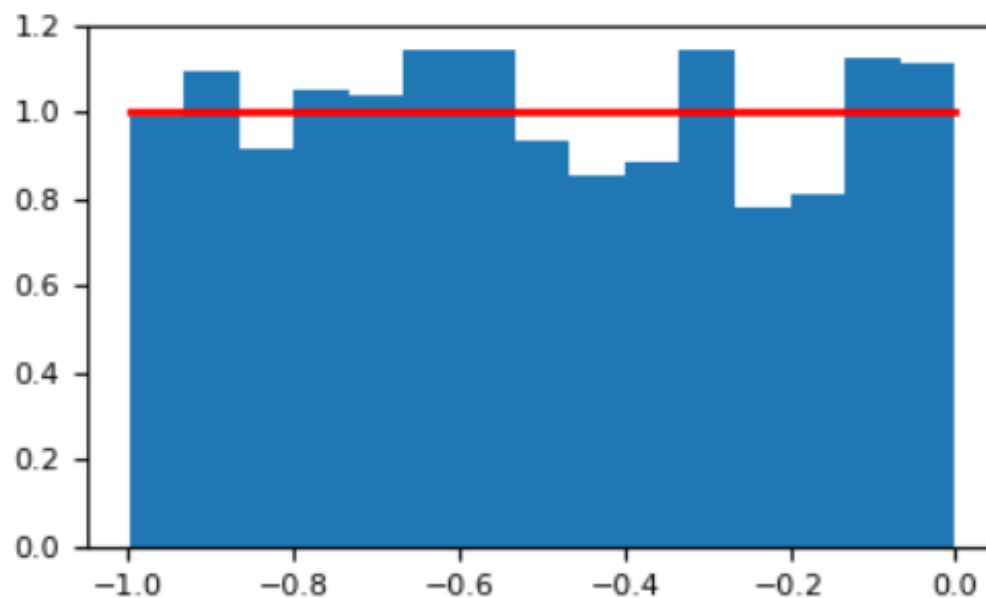
연속균등분포(continuous uniform distribution)

연속균등분포(continuous uniform distribution)은 연속 확률 분포로, 분포가 특정 범위 내에서 균등하게 나타나 있을 경우를 가리킨다. 이 분포는 두 개의 매개변수 a , b 를 받으며, 이때 $[a, b]$ 범위에서 균등한 확률을 가진다.
- 위키백과



numpy.random.uniform Example

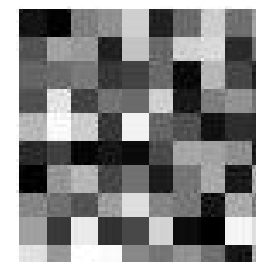
```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```



What is input 'z'?

```
98 def sample_Z(m, n):
99     return np.random.uniform(-1., 1., size=[m, n])
126     _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z: sample_Z(batch_size, Z_dim)})
```

sample_Z.txt	D_logg_save.txt	G_loss
2		
3 1		
4 [[0.19055516 0.67808113 0.77664168 ... 0.9443781 -0.62730144		
5 0.93545081]		
6 [0.365133 1		
7 0.032134 [[-0.04663314 -0.44493603 -0.4339353 ... -0.56040481 0.02028001		
8 [-0.102535 0.28073724]		
9 -0.195336 [-0.96275634 0.59567597 0.38416504 ... -0.98879279 -0.52511382		
10 ... 37 0.95090765]		
11 [-0.581838 [-0.17336111 -0.84157374 -0.70968307 ... 0.09362447 0.01165556		
12 0.927839 0.02457321]		
13 [0.138540 ...		
14 -0.935441 [0.54267131 -0.87528277 0.97255137 ... 0.29205522 -0.49234365		
15 [0.005842 -0.61819233]		
16 0.475843 [0.49600008 0.62631131 -0.41523899 ... 0.50386386 -0.60883917		
44 0.45789924]		
45 [-0.87523505 -0.14243901 -0.96642105 ... -0.22193493 0.94032637		
46 0.70526361]]		



위와 같은 랜덤 노이즈가 z이다.

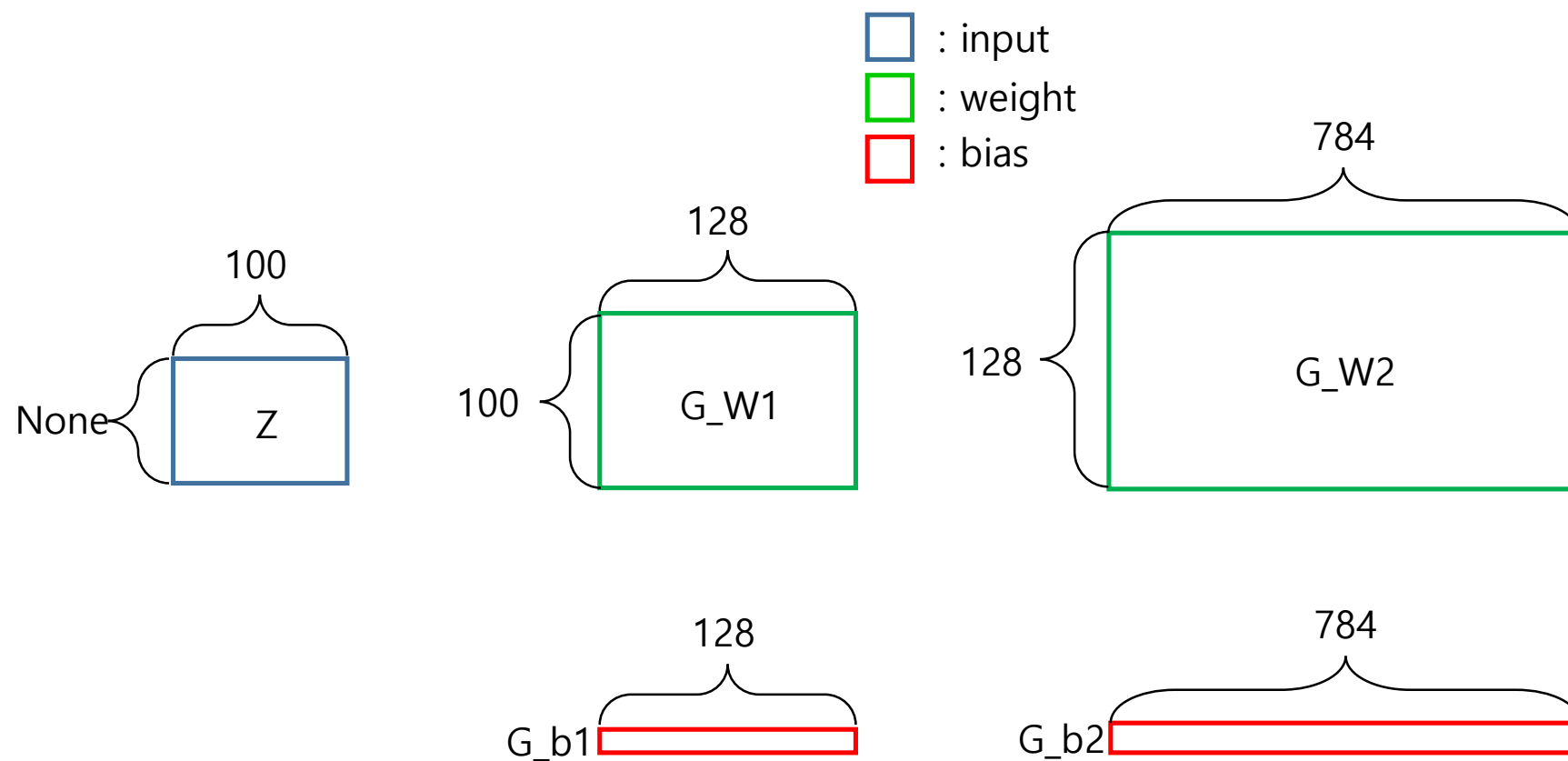
[-1, 1) 사이의 값들이 균일하게 분포하는 [1, 100] 사이즈의 matrix가 input z이다.

Implementation of GAN

Generator variable (code)

```
41 # Random noise setting for Generator
42 Z = tf.placeholder(tf.float32, shape=[None, 100], name='Z')
43
44 #Generator parameter settings
45 G_W1 = tf.Variable(xavier_init([100, 128]), name='G_W1')
46 G_b1 = tf.Variable(tf.zeros(shape=[128]), name='G_b1')
47 G_W2 = tf.Variable(xavier_init([128, 784]), name='G_W2')
48 G_b2 = tf.Variable(tf.zeros(shape=[784]), name='G_b2')
49 theta_G = [G_W1, G_W2, G_b1, G_b2]
```

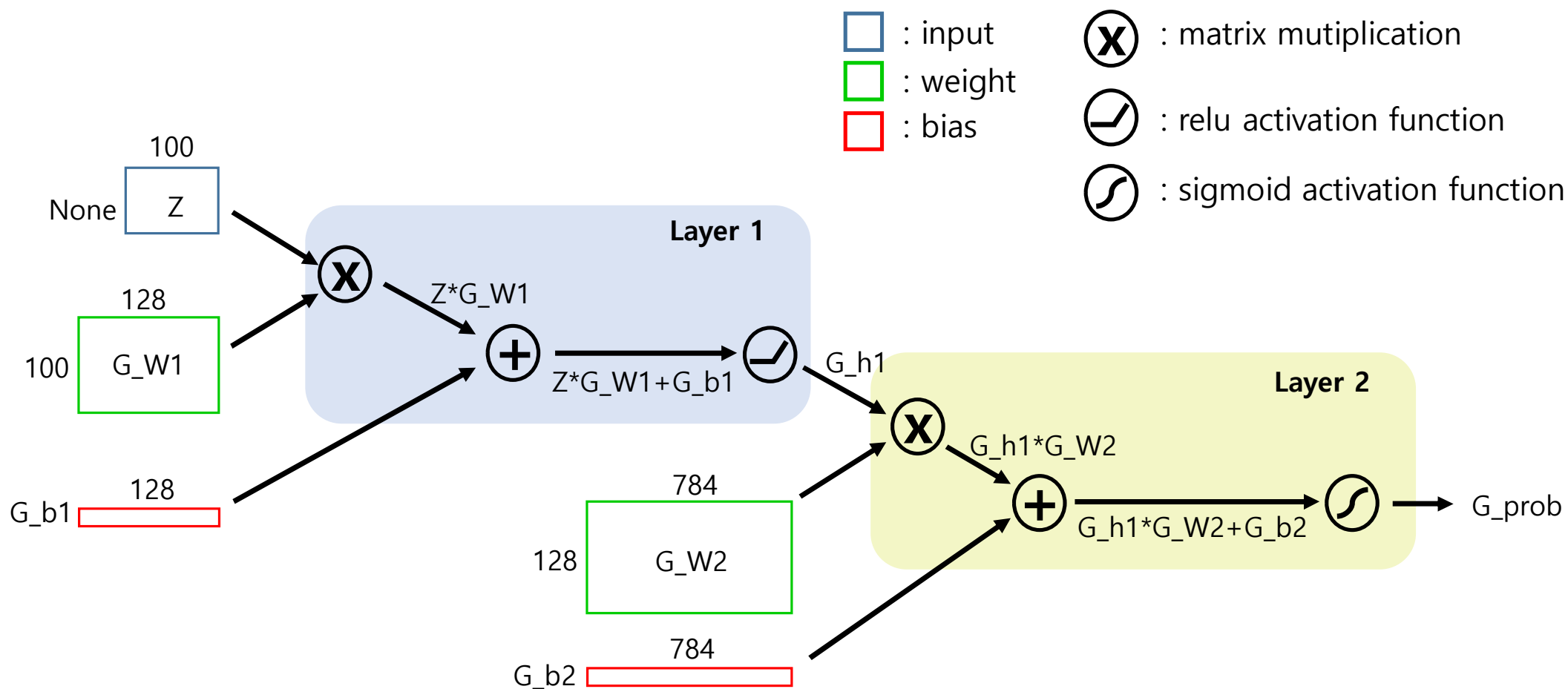
Generator variable



Generator function (code)

```
61 # Generator Network
62 def generator(z):
63     G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)
64     G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
65     G_prob = tf.nn.sigmoid(G_log_prob)
66
67     return G_prob
```

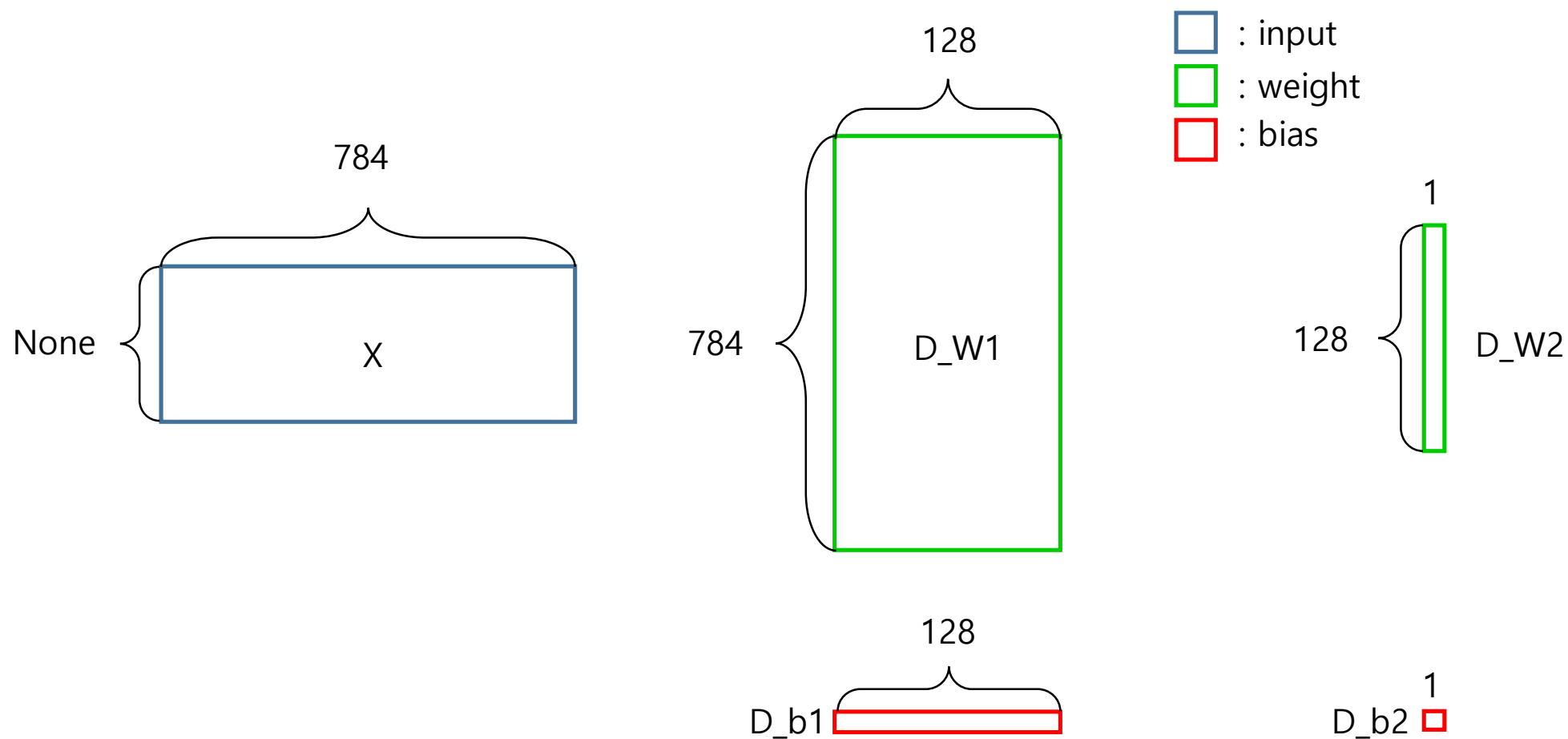
Generator function



Discriminator variable (code)

```
51 #Input Image MNIST setting for Discriminator [28x28=784]
52 X = tf.placeholder(tf.float32, shape=[None, 784], name='X')
53
54 #Discriminator parameter settings
55 D_W1 = tf.Variable(xavier_init([784, 128]), name='D_W1')
56 D_b1 = tf.Variable(tf.zeros(shape=[128]), name='D_b1')
57 D_W2 = tf.Variable(xavier_init([128, 1]), name='D_W2')
58 D_b2 = tf.Variable(tf.zeros(shape=[1]), name='D_b2')
59 theta_D = [D_W1, D_W2, D_b1, D_b2]
```

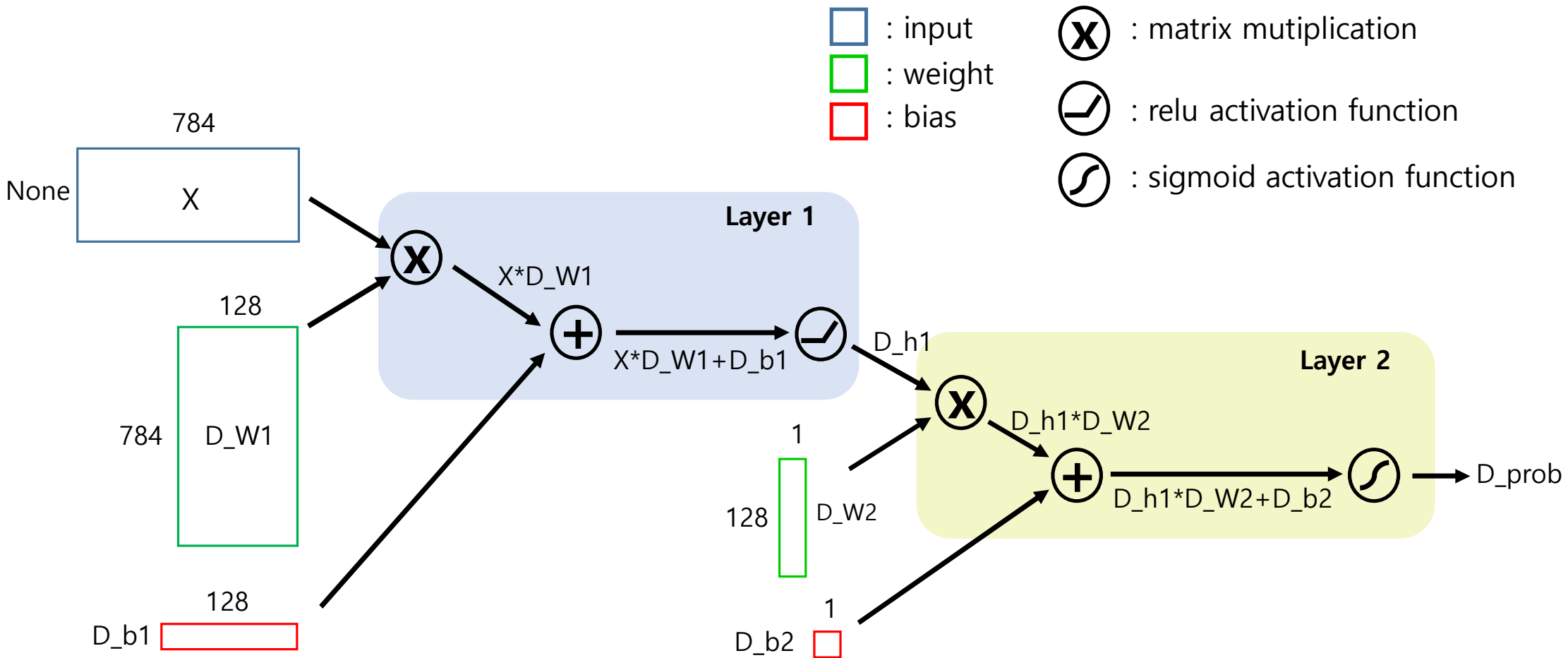

Discriminator variable



Discriminator function (code)

```
69 # Discriminator Network
70 def discriminator(x):
71     D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
72     D_logit = tf.matmul(D_h1, D_W2) + D_b2
73     D_prob = tf.nn.sigmoid(D_logit)
74
75     return D_prob, D_logit
```

Discriminator variable



GAN training algorithm

for number of training iterations **do**

for k steps **do**

m : batch_size

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

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

end for

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

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

end for

Figure 1e: GAN training algorithm pseudo-code

GAN training algorithm

for number of training iterations **do**

for k steps **do**

m : batch_size

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

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

end for

- 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:

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

end for

Figure 1e: GAN training algorithm pseudo-code

Loss function (Discriminator)

```
81 G_sample = generator(Z) →  $G(z)$ 
82
83 D_real, D_logit_real = discriminator(X) →  $D(x)$ 
84 D_fake, D_logit_fake = discriminator(G_sample) →  $D(G(z))$ 
85
86 # Loss functions from the paper
87 D_loss = tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
```

$$\frac{1}{m} \sum_{i=1}^m \left(\log D(x^{(i)}) + \log \left(1 - D(G(z^{(i)})) \right) \right)$$

```
91 # Update D(X)'s parameters
92 D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=theta_D)
```

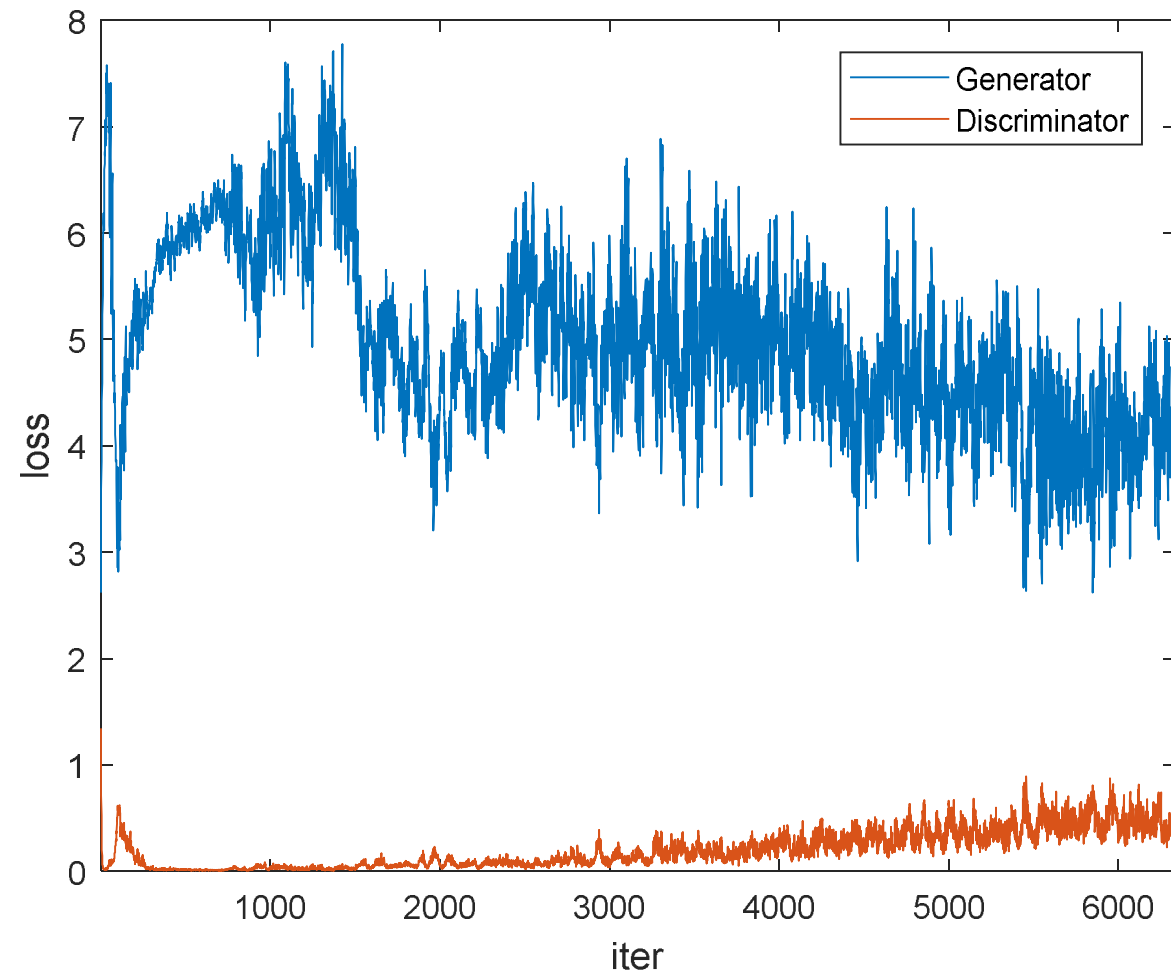
Loss function (Generator)

```
81 G_sample = generator(Z) →  $G(z)$ 
82
83 D_real, D_logit_real = discriminator(X) →  $D(x)$ 
84 D_fake, D_logit_fake = discriminator(G_sample) →  $D(G(z))$ 
85
86 # Loss functions from the paper
87 D_loss = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
88 G_loss = -tf.reduce_mean(tf.log(D_fake))
```

$$\frac{1}{m} \sum_{i=1}^m \log \left(D \left(G(z^{(i)}) \right) \right)$$

```
94 # Update G(Z)'s parameters
95 G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=theta_G)
```

G_loss와 D_loss의 그래프

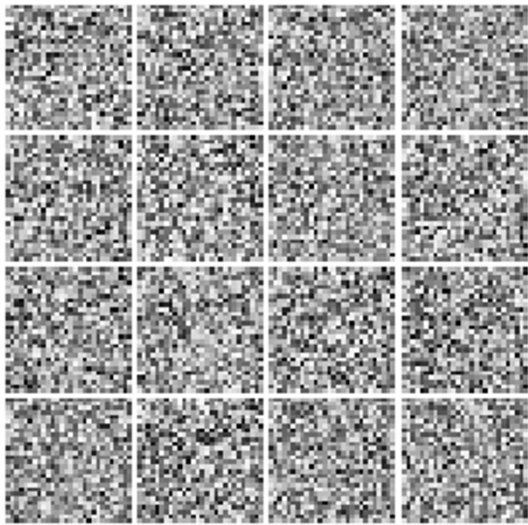


Sess run

```
98 def sample_Z(m, n):
99     return np.random.uniform(-1., 1., size=[m, n])
100
101 batch_size = 128
102 Z_dim = 100
103
104 sess = tf.Session()
105 sess.run(tf.global_variables_initializer())
106
107 mnist = input_data.read_data_sets('MNIST/', one_hot=True)
108
109 if not os.path.exists('output/'):
110     os.makedirs('output/')
111
112 i = 0
113
114 for itr in range(1000000):
115     if itr % 1000 == 0:
116         samples = sess.run(G_sample, feed_dict={Z: sample_Z(16, Z_dim)})
117         fig = plot(samples)
118         plt.savefig('output/{}.png'.format(str(i).zfill(3)), bbox_inches='tight')
119         i += 1
120         plt.close(fig)
121
122 X_mb, _ = mnist.train.next_batch(batch_size)
123
124 _, D_loss_curr = sess.run([D_solver, D_loss], feed_dict={X: X_mb, Z: sample_Z(batch_size, Z_dim)})
125 _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z: sample_Z(batch_size, Z_dim)})
126
127 if itr % 1000 == 0:
128     print('Iter: {}'.format(itr))
129     print('D loss: {:.4}'.format(D_loss_curr))
130     print('G loss: {:.4}'.format(G_loss_curr))
131     print()
```

16개를 generate해서 저장.

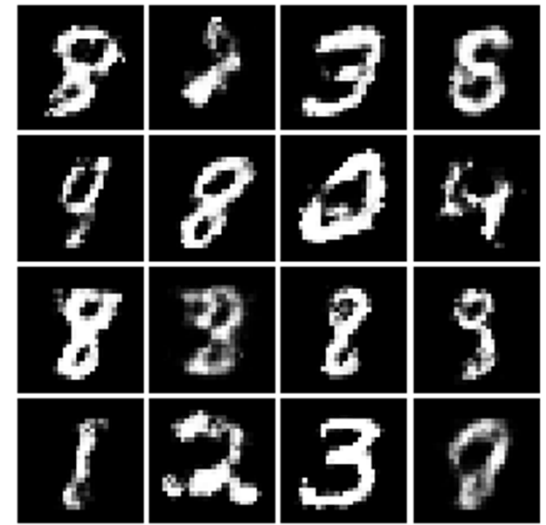
실험 결과



0 iterations



10,000 iterations



60,000 iterations

Assignment

- **Analyze a given source code**
- **Draw plot of (1) discriminator and (2) generator loss values with respect to iterations**
- **Show the generated images at 0, 10,000, 60,000 iterations**
- **Discuss experimental results**

Things You have to Submit

❖ Source code for all assignments

- Using Python for implementing a given codes
- Please conform the following naming formation

“source code for final-term project”

❖ Report material with MS word format

❖ Due date: No later than 20th, December.