GAN; Simple Implementation

기계공학과

2017311435 김승하

Implementation List

1. Vanilla Gan (Basic GAN) [1]
2. Deep Convolutional GAN (DCGAN) [2]
3. Energy-based GAN (EBGAN) [3]
4. Boundary Equilibrium GAN (BEGAN) [3]

수업 때 발표했던 내용이 GAN을 이해하는데 부족하다고 생각되어서,
몇 가지 GAN들의 개념과 코드를 간단하게 정리 했습니다.

Function Description

Tensorflow-Slim

Tensorflow에 포함되어 있는 뉴럴 네트워크를 간단하게 구현할 수 있는 라이브러리

```
import tensorflow.contrib.slim as slim
```

```
slim.fully_connected(
    inputs, num_outputs, activation_fn=nn.relu,
    weights_initializer=initializers.xavier_initializer(), biases_initializer=init_ops.zeros_initializer(),
    reuse=None, scope=None
)
```

inputs: tensor
num_outpus: Integer, fully_connected 레이어의 아웃풋 유닛의 개수
return : tensor

```
slim.conv2d(
    inputs, num_outputs, kernel_size, stride=1, padding='SAME', activation_fn=nn.relu,
    weights_initializer=initializers.xavier_initializer(), biases_initializer=init_ops.zeros_initializer(),
    reuse=None, scope=None
)
```

inputs: tensor
num_outpus : Integer, convolutional 레이어의 아웃풋 필터의 개수
kernel_size : (sequence of) positive Intger(s), 필터 사이즈
return : tensor

1. Vanilla Gan (Basic GAN)

Vanilla GAN ([When did vanilla become a default flavor?])

Ian Goodfellow가 제안한 GAN모델 (2014)

> Discriminant와 Generator, 두 플레이어는
>
> value function V(G, D) 를 놓고 minimax 게임을 한다!

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

Discriminant의 관점 $\cancel{\min_G} \max_D = E_{x\sim p_{data}(x)}[logD(x)] + E_{z\sim p_z(z)}[\log\left(1 - D\left(G(z)\right)\right)]$

Generator의 관점 $\min_G \cancel{\max_D} = \cancel{E_{x\sim p_{data}(x)}[logD(x)]} + E_{z\sim p_z(z)}[\log\left(1 - D\left(G(z)\right)\right)]$

1. Vanilla Gan

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

Hyper Parameter

```
mb_size = 128          # 배치사이즈
X_dim = 28 * 28        # 이미지 디멘션
z_dim = 100            # 인풋 노이즈 디멘션
Hidden_dim = 100       # 히든레이어 디멘션
D_out_dim = 1          # Discriminant out dimension
                       #  T/F 값을 Classification해주므로 1 dim
```

Model

```python
def sample_z(batch_size, n):
    return np.random.uniform(-1., 1., size=[batch_size, n])


def generator(z):
    with tf.variable_scope('G'):
        G_hidden = slim.fully_connected(z, hidden_dim, activation_fn=tf.nn.relu)
        G_prob = slim.fully_connected(G_hidden, X_dim, activation_fn=tf.nn.sigmoid)
    return G_prob


def discriminator(x, reuse=False):
    with tf.variable_scope('D', reuse=reuse):
        D_hidden = slim.fully_connected(x, hidden_dim, activation_fn=tf.nn.relu)
        D_prob = slim.fully_connected(D_hidden, D_out_dim, activation_fn=tf.nn.sigmoid)
    return D_prob


X = tf.placeholder(tf.float32, shape=[None, X_dim])
z = tf.placeholder(tf.float32, shape=[None, z_dim])

G_sample = generator(z)
D_real = discriminator(X)
D_fake = discriminator(G_sample, reuse=True)
```

1. Vanilla Gan

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log \underbrace{D(\boldsymbol{x})}_{\text{D\_real}}] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - \underbrace{D(\overbrace{G(\boldsymbol{z})}^{\text{G\_sample}})}_{\text{D\_fake}})].$$

z : sample_z(mb_size, z_dim)
random uniform distribution으로 생성

Generator
z를 input으로 fully connected layer를 거쳐
Generated image[batch_size, X_dim] 리턴

Discriminator
x를 input으로 fully connected layer를 거쳐
판별값(True/False)[batch_size, 1] 리턴

Data Feeder (Image, z)

G_sample[batch, x_dim] : G로부터 생성된 데이터
D_real[batch, 1] : 실제 이미지에 대한 D의 판별
D_fake[batch, 1] : 생성된 이미지에 대한 D의 판별

Model

```
def sample_z(batch_size, n):
    return np.random.uniform(-1., 1., size=[batch_size, n])


def generator(z):
    with tf.variable_scope('G'):
        G_hidden = slim.fully_connected(z, hidden_dim, activation_fn=tf.nn.relu)
        G_prob = slim.fully_connected(G_hidden, X_dim, activation_fn=tf.nn.sigmoid)
    return G_prob


def discriminator(x, reuse=False):
    with tf.variable_scope('D', reuse=reuse):
        D_hidden = slim.fully_connected(x, hidden_dim, activation_fn=tf.nn.relu)
        D_prob = slim.fully_connected(D_hidden, D_out_dim, activation_fn=tf.nn.sigmoid)
    return D_prob


X = tf.placeholder(tf.float32, shape=[None, X_dim])
z = tf.placeholder(tf.float32, shape=[None, z_dim])

G_sample = generator(z)
D_real = discriminator(X)
D_fake = discriminator(G_sample, reuse=True)
```

mb_size = 128
X_dim = 28 * 28
z_dim = 100
Hidden_dim = 100
D_out_dim = 1

1. Vanilla Gan

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log \underbrace{D(\boldsymbol{x})}_{\text{D\_real}}] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - \underbrace{D(\overbrace{G(\boldsymbol{z})}^{\text{G\_sample}})}_{\text{D\_fake}})].$$

Model

```
G_sample = generator(z)
D_real = discriminator(X)
D_fake = discriminator(G_sample, reuse=True)
```

D의 관점 : $\max_D = \log(\underline{E_{x\sim p_{data}(x)}[D(x)]}) + \log(1 - \underline{E_{z\sim p_z(z)}[D(G(z))]})$

$\min_D(- (\log(\underline{\text{D\_real}}) + \log(1- \underline{\text{D\_fake}})))$

Loss & Optimizer

```
D_loss = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
G_loss = -tf.reduce_mean(tf.log(D_fake))

theta_G = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='G')
theta_D = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='D')

D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=theta_D)
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=theta_G)

```

G의 관점 : $\min_G = \log(1 - \underline{E_{z\sim p_z(z)}[D(G(z))]})$

$\approx \min_G(- \log(\underline{\text{D\_fake}}))$

- log(1 – x)는 x = 0에서 gradient가 작기 때문에 – log(x)로 근사

2. Deep Convolutional GAN (DCGAN)

일반적인 Binary Classification CNN 구조

Convolution연산을 거치며 이미지 사이즈는 줄이고 채널(필터)의 숫자는 늘린다.

Transposed Convolution을 사용한 Generator

Generator는 input noise로부터 feature를 증가시키며 학습해간다.

CNN Classification구조와 반대로 이미지 크기를 증가시켜줘야 되기때문에 Convolution의 역방향 연산, **Transposed Convolution**(fractionally strided convolutions) 을 사용한다.

Model

```
def discriminator(x, reuse=False):
    with tf.variable_scope('D', reuse=reuse):
        x = tf.reshape(x, [-1, 28, 28, 1])
        conv1 = slim.conv2d(
            x,32, kernel_size= 3, stride= 1, activation_fn=lrelu)
        conv2 = slim.conv2d(
            conv1,64, kernel_size= 3, stride= 1, activation_fn=lrelu)
        d_flat = slim.flatten(conv2)
        D_prob = slim.fully_connected(d_flat, 1, activation_fn=tf.nn.sigmoid)
    return D_prob


def generator(z):
    with tf.variable_scope('G'):
        z = slim.fully_connected(z, 28*28*4, activation_fn=None)
        z = tf.reshape(z,[-1,28,28,4])
        g1 = slim.conv2d_transpose(
            z,num_outputs=16, kernel_size= 3, stride= 1, padding='SAME',
            activation_fn=lrelu)
        g_out = slim.conv2d_transpose(
            g1,num_outputs=1, kernel_size= 3, stride= 1, padding='SAME',
            activation_fn=tf.nn.sigmoid)
        g_out  = slim.flatten(g_out)
        return g_out
```

Transposed Convolution에 대한 내용

3. Energy-based GAN (EBGAN)

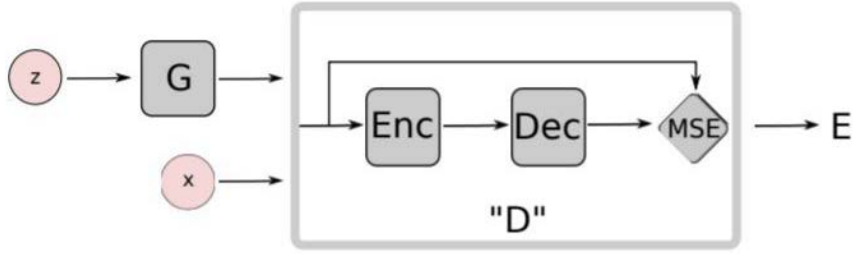


Figure 1: EBGAN architecture with an auto-encoder discriminator.

Discriminator

auto-encoder구조 사용

D_h1 : Encoder

X_recon : Decoder

recon_real

오토인코더(D)가 reconstrunction한 real image

recon_fake

오토인코더(D)가 reconstruction한 generated image

Model

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


def generator(z):
    with tf.variable_scope('G'):
        G_h1 = slim.fully_connected(z, h_dim, activation_fn=tf.nn.relu)
        G_prob = slim.fully_connected(G_h1, X_dim, activation_fn=tf.nn.tanh)
    return G_prob


def discriminator(x, reuse=False):
    with tf.variable_scope('D', reuse=reuse):
        D_h1 = slim.fully_connected(x, h_dim, activation_fn=tf.nn.relu)
        X_recon = slim.fully_connected(D_h1, D_out_dim, activation_fn=tf.nn.tanh)
    return X_recon


X = tf.placeholder(tf.float32, shape=[None, X_dim])
z = tf.placeholder(tf.float32, shape=[None, z_dim])

G_sample = generator(z)

recon_real = discriminator(X)
recon_fake = discriminator(G_sample, reuse=True)
```

```
mb_size = 32
X_dim = 28 * 28
z_dim = 64
h_dim = 32
D_out_dim = X_dim
Margin_loss = 5
```

3. Energy-based GAN (EBGAN)

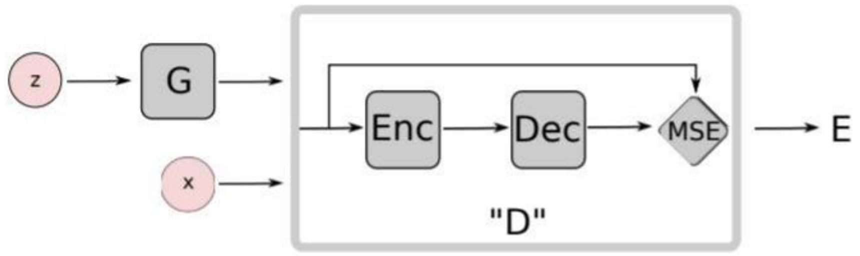


Figure 1: EBGAN architecture with an auto-encoder discriminator.

recon_loss_real

실제 이미지를 reconstruction한 mean square error

recon_loss_fake

생성된 이미지를 reconstruction한 mean square error

$$\mathcal{L}_D(x,z) = D(x) + \max(0, [m - D(G(z))])$$

$$\mathcal{L}_G(z) = D(G(z))$$

Model

```
recon_real = discriminator(X)
recon_fake = discriminator(G_sample, reuse=True)
```

Loss & Optimizer

```
def mse_loss(x, recon_x):
    mse_error = tf.reduce_sum((x - recon_x)**2, 1)
    return tf.reduce_mean(mse_error)


recon_loss_real = mse_loss(X, recon_real)
recon_loss_fake = mse_loss(G_sample, recon_fake)


D_loss = recon_loss_real + tf.maximum(0., margin_loss - recon_loss_fake)
G_loss = recon_loss_fake


theta_G = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='G')
theta_D = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='D')

D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=theta_D)
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=theta_G)
```

4. Boundary Equilibrium GAN (BEGAN)

Discriminant : real data와 fake data의 구분이 목적

BEGAN : 오토인코더 구조를 사용해 real data와 fake data를 구분

=> real data와 fake data의 probability distribution을 최대화

$$W_1(\mu_1,\mu_2) = \inf_{\gamma\in\Gamma(\mu_1,\mu_2)} \mathbb{E}_{(x_1,x_2)\sim\gamma}[|x_1 - x_2|]$$

$$\underbrace{\geqslant \inf |\mathbb{E}[x_1 - x_2]|}_{\text{Jansen inequality}} = |m_1 - m_2|$$

$$\geqslant m_2 - m_1$$

Wasserstein Distance과 Jansen inequality 사용,

probability distribution을 최대화 하기 위해서 real data와 fake data의 Loss distribution을 최대화해야 된다는 것을 보임

Discriminant

real data : auto-encoder loss작게 최적화

fake data : auto-encoder loss 크게 최적화

Loss & Optimizer

```
def AE_loss(x, recon_x):
    # L1_loss
    loss = tf.reduce_mean(tf.reduce_sum(tf.abs(recon_x - x), 1))

    # L2_loss
    # loss = tf.reduce_mean(tf.reduce_sum((x - recon_x)**2, 1))
    return loss


D_real = AE_loss(X, recon_real)
D_fake = AE_loss(G_sample, recon_fake)

k = tf.Variable(0., trainable=False)
D_loss = D_real - k * D_fake
G_loss = D_fake
```

$x_{1,2}$: real/fake data

$\mu_{1,2}$: loss distribution of real/fake data

m_1 : mean of real data sample loss # D_real

m_2 : mean of fake data sample loss # D_fake

$W_1(\mu_1, \mu_2) = m_2 - m_1$

$\Longrightarrow$ max(D_loss) = max(D_real – D_fake)

$\Longrightarrow$ min(D_loss) = min(D_real – D_fake)

4. Boundary Equilibrium GAN (BEGAN)

GAN Object

$$\mathbb{E}\left[\mathcal{L}(x)\right] = \mathbb{E}\left[\mathcal{L}(G(z))\right]$$

D_loss = D_real – D_fake

G_loss = D_fake

문제점

Generator가 Discriminant를 속이는데 급급해

다양한 이미지가 생성이 안됨

BEGAN Object

$\gamma\,\mathbb{E}\left[\mathcal{L}(x)\right] = \mathbb{E}\left[\mathcal{L}(G(z))\right]$, $\gamma \in [0, 1]$

변수 γ를 사용해 Discriminant와 Generator의 균형을 맞춰 줌

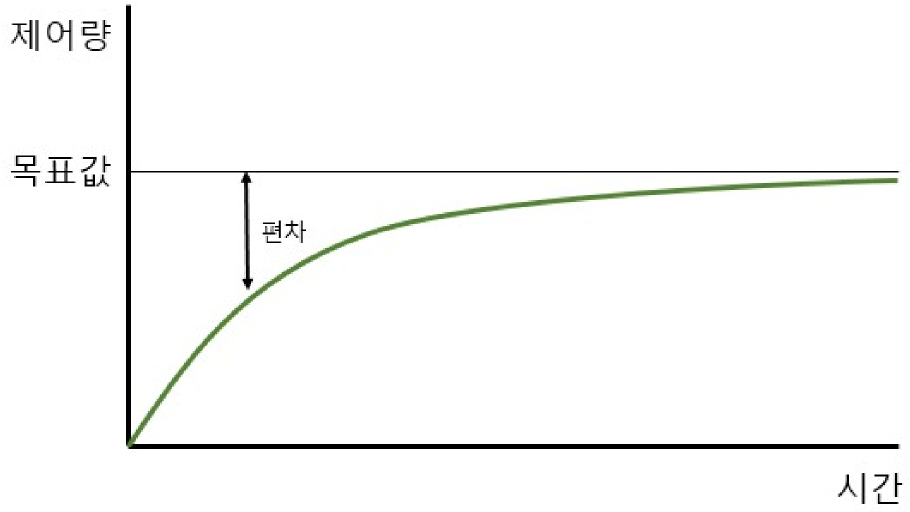
Implementation of BEGAN Object

비례제어(Proportional Control)를 사용해 γ를 반영

$$\mathcal{L}_D = \mathcal{L}(x) - k_t.\mathcal{L}(G(z_D))$$

$$k_{t+1} = k_t + \lambda_k(\gamma\mathcal{L}(x) - \mathcal{L}(G(z_G)))$$

비례제어 : 목표값과 현재 제어값의 오차에 비례하여 제어량을 변화



4. Boundary Equilibrium GAN (BEGAN)

The BEGAN Object is:

$$\begin{cases} \mathcal{L}_D = \mathcal{L}(x) - k_t.\mathcal{L}(G(z_D)) & \text{for } \theta_D \\ \mathcal{L}_G = \mathcal{L}(G(z_G)) & \text{for } \theta_G \\ k_{t+1} = k_t + \lambda_k(\gamma\mathcal{L}(x) - \mathcal{L}(G(z_G))) & \text{for each training step } t \end{cases}$$

$$\mathcal{L}_D = \mathcal{L}(x) - k_t.\mathcal{L}(G(z_D))$$

$$\mathcal{L}_G = \mathcal{L}(G(z_G))$$

$$k_{t+1} = k_t + \lambda_k(\gamma\mathcal{L}(x) - \mathcal{L}(G(z_G)))$$

Coverage Measure

오토인코더가 실제 이미지를 잘 복원하고,

Discriminator와 Generator가 평형이면 Coverage

$$\mathcal{M}_{global} = \underline{\mathcal{L}(x)} + |\underline{\gamma\mathcal{L}(x) - \mathcal{L}(G(z_G))}|$$

Loss & Optimizer

```
def AE_loss(x, recon_x):
    # L1_loss
    loss = tf.reduce_mean(tf.reduce_sum(tf.abs(recon_x - x), 1))

    # L2_loss
    # loss = tf.reduce_mean(tf.reduce_sum((x - recon_x)**2, 1))
    return loss


D_real = AE_loss(X, recon_real)
D_fake = AE_loss(G_sample, recon_fake)

k = tf.Variable(0., trainable=False)
D_loss = D_real - k * D_fake
G_loss = D_fake


theta_G = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='G')
theta_D = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='D')

D_solver = tf.train.AdamOptimizer(learning_rate=lr).minimize(D_loss, var_list=theta_D)
G_solver = tf.train.AdamOptimizer(learning_rate=lr).minimize(G_loss, var_list=theta_G)


balance = gamma * D_real - D_fake

with tf.control_dependencies([D_solver, G_solver]):
    k_update = tf.assign(k, tf.clip_by_value(k + lambda_k * balance, 0, 1))

measure = D_real + tf.abs(balance)
```

REFERENCE

[1] Goodfellow, Ian, et al. "Generative adversarial nets." *Advances in neural information processing systems*. 2014.

[2] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." *arXiv preprint arXiv:1511.06434* (2015).

[3] Zhao, Junbo, Michael Mathieu, and Yann LeCun. "Energy-based generative adversarial network." *arXiv preprint arXiv:1609.03126*(2016).

[4] Berthelot, David, Tom Schumm, and Luke Metz. "Began: Boundary equilibrium generative adversarial networks." *arXiv preprint arXiv:1703.10717* (2017).