UNSUPERVISED AND SEMI-SUPERVISED LEARNING WITH CATEGORICAL GENERATIVE ADVERSARIAL NETWORKS

- Reading the paper in terms of semi-supervised learning -

1. Introduction

CatGAN은 데이터가 진짜인지 가짜인지를 구분할 뿐만 아니라, 진짜 데이터를 분류할 수 있는 discriminator를 가진 GAN 구조라고 볼 수 있습니다.

GAN은 generator와 discriminator의 서로 속이는 게임의 개념으로 알려져 있습니다. GAN이 나오기 이전부터 unsupervised learning에서는 generative와 discriminative의 개념이 존재했었습니다. Unsupervised Learning은 입력 데이터 x에 유의미한 정보 y가 있다는 가정으로부터 시작합니다. 즉 입력 데이터 x에 유의미한 정보 y가 있을 확률 p(yx)를 기반으로 합니다. 그래서 군집화의 경우 generative한 접근은 입력 데이터 x의 정보 y들에 대한 몇 가지 종류의 분포 p(x)를 직접 만드는 것이며, discriminative한 접근은 분포를 예측하는 것이 아니라 처음부터 정보 y를 가지고 데이터 x를 몇 가지 종류로 잘 묶는 것입니다. CatGAN은 unsupervised Learning의 두 접근방식인 generative와 discriminative를 모두 사용해 discriminator를 훈련하는 것에 집중합니다.

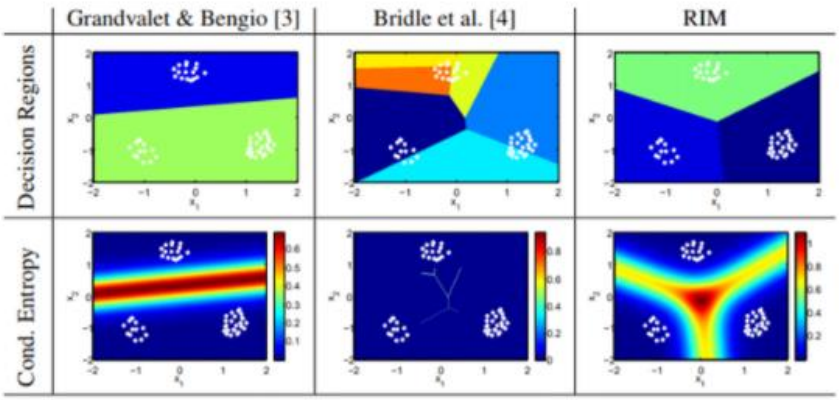
CatGAN은 unsupervised learning 기법이지만, 극소수의 라벨이 존재하는 데이터를 사용하여 semi-supervised learning을 통해 성능을 획기적으로 높일 수 있는 알고리즘입니다. 논문의 저자도 CatGAN 실험 시, semi-supervised learning의 형태로 활용하는 것에 초점을 맞추고 있습니다. 본 과제에서는 semi-supervised learning 관점에서 논문을 해석하고, 어떻게 semi-supervised learning을 수행하는지에 대한 관점으로 진행하였습니다.

2. Background

CatGAN은 discriminative clustering(데이터 군집을 몇 개의 주어진 클래스로 분류하는 것)을 수행하기 위해 RIM(Regularized Information Maximization)이라는 개념을 도입하고 있습니다. RIM은 discriminative clustering 수행 시 class separation(클래스를 잘 나누는 것)과 class balance(적당한 수의 클래스로 분류하는 것)의 trade-off에 주목합니다.

Class separation 관점에서는 클래스를 나누는 경계인 decision boundary에 데이터가 최대한 없게 하여 명확하게 데이터가 구분될 수 있도록 만듭니다. 이 관점에서 unsupervised learning에서는 하단 <그림 1>의 맨 왼쪽처럼 클래스를 충분히 3개로 나눌 수 있음에도 불구하고 1개의 분류 경

계선을 만들어 내는 것을 볼 수 있습니다. 이는 안정적인 decision boundary를 확보하는 것에 편중되어 클래스의 숫자를 줄이는 방향으로 진행되기 때문입니다.

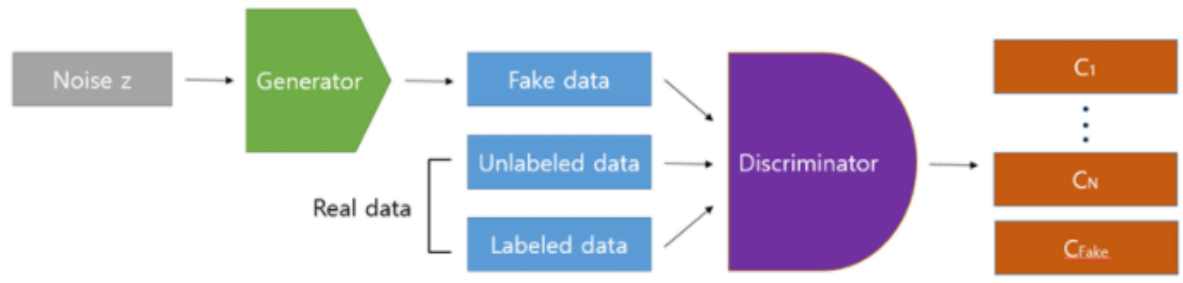


<그림 1> Class Separation, Class balance, RIM 관점에서의 decision boundary

이를 보완하기 위해 class balance 관점을 도입하게 됩니다. 하지만 class balance의 관점에 편중되게 된다면, 적당히 비슷한 속성을 가진 데이터들을 클래스 몇 개로 분류하는 것이 아니라, 개별 데이터 하나하나를 각각의 클래스로 분류하는 것이 최적화 solution이 되는 것이기에 <그림 1>의 중간처럼 과도한 개수로 클래스가 분류될 수 있습니다.

이 두 관점 사이의 trade-off를 적절하게 설정하기 위해 regularization을 도입하게 됩니다. Regularization을 도입하여 맨 오른쪽 그림처럼 클래스가 적당하게 합리적으로 분류된 것을 확인할 수 있습니다.

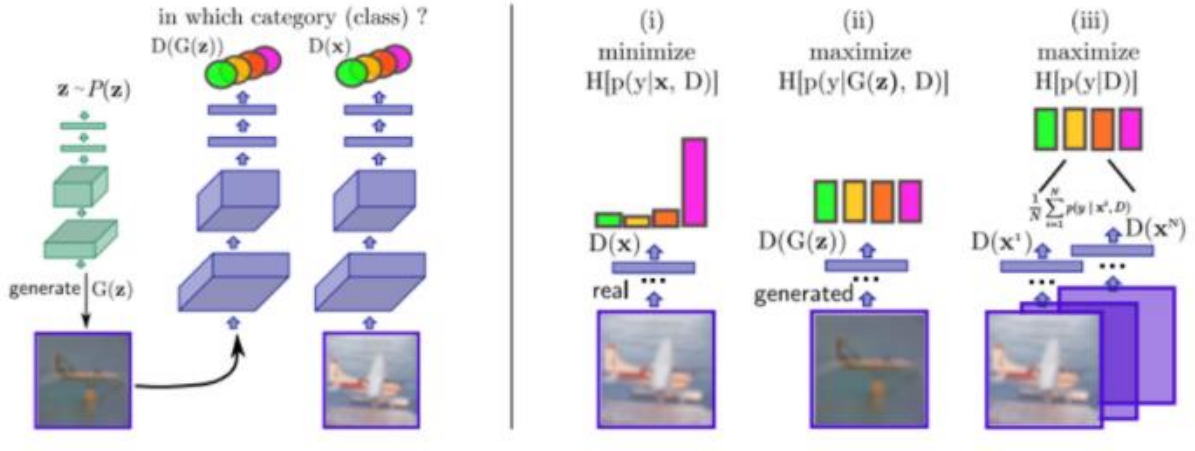
3. Proposed Method



<그림 2> CatGAN의 구조

CatGAN은 이 구조가 original GAN의 일반화 버전이거나, RIM의 확장 버전으로 생각할 수 있습니다. Original GAN의 Discriminator(D)가 진짜와 가짜를 구별하는 것을 목적으로 하였다면, CatGAN

은 개념을 좀 더 확장하여 class까지 구별할 수 있기 때문입니다. CatGAN의 Generator(G)는 semi-supervised Discriminator의 성능을 보강해주는 역할을 하며, D는 흔히 알고 있는 classification network와 거의 동일한 구조를 갖지만 G와의 adversarial training을 통해 성능이 강화되는 구조를 가지고 있습니다. CatGAN이 RIM의 개념을 확장한 것으로 볼 수 있는 이유는 CatGAN의 G가 분류기 역할을 수행하는 D에 대한 regularization 역할을 수행하기 때문입니다.



<그림 3> Discriminator의 관점에서의 목적함수 L_D 도식화

CatGAN에서는 목적함수 설계 시 엔트로피의 개념을 도입하고 있습니다. 즉, semi-supervised 데이터를 각각의 class로 얼마나 잘 분류하는지를 알아볼 수 있는 goodness of fit의 척도로 엔트로피 개념을 사용하였습니다. 따라서 D가 출력하는 확률분포의 엔트로피가 상황에 따라 minimize되거나, maximize되도록 하고 있습니다.

<그림 3>의 왼쪽 그림은 CatGAN의 대략적인 구조를 보여줍니다. 확률 분포 P(z)를 따르는 랜덤 노이즈 z가 초록색 G를 거쳐 가운데 보라색 D를 통과합니다. 한편 진짜 데이터 또한 오른쪽 보라색 D를 지나면서 연두색, 노란색, 빨간색, 분홍색 등등의 클래스 중 어느 클래스에 진짜 데이터가 속하는지를 판별하게 됩니다. G에서 생성된 가짜 데이터는 어느 클래스에 따로 속하지 않습니다.

$$\mathcal{L}_D = \max_D \; H_{\mathcal{X}}\Big[p(y \mid D)\Big] - \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\Big[H\big[p(y \mid \mathbf{x}, D)\big]\Big] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}\Big[H\big[p(y \mid G(\mathbf{z}), D)\big]\Big]$$

<그림 3>의 오른쪽 그림은 목적 함수 L_D의 관점을 나타낸 것입니다. (i)에서 진짜 데이터는 이미지의 해당하는 라벨인 분홍색으로 분류되어야 하기 때문에, x에서 특징 y가 있을 확률인 $p(y|x, D)$는 분홍색에 대해서만 커야하고, 엔트로피 $H[p(y|x, D)]$는 작아져야 합니다. (ii)에서 가짜 데이터 G(z)가 D를 통과한 D(G(z))는 어느 한 클래스에 종속되지 않고 모든 클래스에 들어갈 확률이 비슷해져야 하기 때문에 $p(y|G(z), D)$는 모두 일정한 값이 출력되도록 엔트로피 $H[p(y|G(z), D)]$는 커져야 합니다. (iii)에서 저자들은 진짜 데이터 x가 N개의 클래스에 속할 확률은 서로 같다고 가정하

였으므로 x가 어느 한 클래스에 속할 확률인 $p(y|x^i, D)$는 $\frac{1}{N}\sum_{i=1}^{N} p(y|x^i, D)$과 비슷해져야 합니다. 따라서 x에 대한 주변 확률 분포 $p(y|x^i, D) = p(y|D)$는 똑 같은 확률값을 가지도록 수렴해야하며, 엔트로피 $H[p(y|D)]$는 커져야 합니다.(주변 확률 분포는 확률 변수들에서 관측된 어느 한 확률변수를 의미)

$$\mathcal{L}_G = \min_G -H_G\Big[p(y \mid D)\Big] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}\Big[H\big[p(y \mid G(\mathbf{z}), D)\big]\Big]$$

<그림 3>에는 나와있지 않지만 G의 목적함수 L_G는 adversarial training의 개념에 맞춰 D의 입장과 반대로 생각하면 해석을 쉽게 할 수 있습니다. 첫번째로 G가 만들어낸 가짜 데이터 G(z)는 어느 한 클래스에 속해야만 하므로 $H[p(y|G(z), D)]$는 작아져야 합니다. 두 번째로 D의 (iii)의 경우와 비슷하게, generate된 가짜 데이터 G(z)들이 N개의 클래스에 속할 확률이 서로 같아야 하므로 $p(y|G(z)^i, D) = p(y|D)$가 되며, 엔트로피 $H[p(y|D)]$는 커져야 합니다.

Algorithm	PI-MNIST test error (%) with n labeled examples		
	$n = 100$	$n = 1000$	All
MTC (Rifai et al., 2011)	12.03	100	0.81
PEA (Bachman et al., 2014)	10.79	2.33	1.08
PEA+ (Bachman et al., 2014)	5.21	2.67	-
VAE+SVM (Kingma et al., 2014)	11.82 (± 0.25)	4.24 (± 0.07)	-
SS-VAE (Kingma et al., 2014)	3.33 (± 0.14)	2.4 (± 0.02)	0.96
Ladder Γ-model (Rasmus et al., 2015)	4.34 (± 2.31)	1.71 (± 0.07)	0.79 (± 0.05)
Ladder full (Rasmus et al., 2015)	**1.13 (± 0.04)**	**1.00 (± 0.06)**	-
RIM + NN	16.19 (± 3.45)	10.41 (± 0.89)	
GAN + SVM	28.71 (± 7.41)	13.21 (± 1.28)	
CatGAN (*unsupervised*)	9.7		
CatGAN (semi-supervised)	1.91 (± 0.1)	1.73 (± 0.18)	0.91

<표 1> PI-MNIST 실험 결과

CatGAN은 목적함수에 대한 설명에서 볼 수 있듯, GAN의 원래 목적인 이미지 generating에 목적을 두는 것이 아니라 robust하게 class를 분류할 수 있는 D를 얻고자 함에 있습니다. CatGAN은 사실 unsupervised learning을 할 수 있는 알고리즘입니다. 하지만 논문의 실험결과에서 설명하듯, unsupervised learning을 수행하면 에러율이 9.7%이지만 라벨이 있는 데이터 100개(n=100)를 제공하여 semi-supervised learning을 수행할 경우 에러율이 1.91%로 성능이 크게 개선되는 것을 볼 수 있습니다.

$$CE\big[\mathbf{y}, p(y \mid \mathbf{x}, D)\big] = -\sum_{i=1}^{K} y_i \log p(y = y_i \mid \mathbf{x}, D)$$

$$\mathcal{L}_D^L = \max_D H_{\mathcal{X}}\Big[p(y \mid D)\Big] - \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\Big[H\big[p(y \mid \mathbf{x}, D)\big]\Big] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}\Big[H\big[p(y \mid G(\mathbf{z}), D)\big]\Big]$$
$$+\lambda\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{X}^L}\Big[CE\big[\mathbf{y}, p(y \mid \mathbf{x}, D)\big]\Big],$$

Semi-supervised-learning 시, 라벨이 없는 데이터는 pseudo 라벨을 부착하며, epoch마다 unlabeled data에 라벨을 업데이트합니다. 목적함수는 위와 같이 λ를 포함한 cost weighting term 을 두어 pseudo 라벨의 관측 확률을, 라벨이 있는 데이터의 관측 확률과 일치하도록 만듭니다.

4. Experiments

실험에서는 CatGAN을 통해 Generator가 어떠한 이미지를 생성하는지, Discriminator의 분류 성능이 어느 정도의 성능을 나타내는지 확인하기 위해 구현하였습니다.

```
train_set = torchvision.datasets.MNIST(root=PATH,train=True,
                                       download=True,transform=transform)
train_loader = torch.utils.data.DataLoader(train_set,batch_size=batch_size,
                                           shuffle =True)

test_set = torchvision.datasets.MNIST(root=PATH,train=False,
                                      download=True, transform=transform)

test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
                                           shuffle = False)

train_iter = iter(train_loader)
test_iter = iter(test_loader)

real_batch, labels = next(iter(train_loader))
plt.figure(figsize=(8,8))
plt.axis('off')
plt.title('Training Images')
plt.imshow(np.transpose(vutils.make_grid(real_batch.to(device)[:64],padding=2, normalize=True).cpu(),(1,2,0)))

print(real_batch.size())
```

```
torch.Size([64, 1, 28, 28])
```

CatGAN의 구현에는 torchvision에서 제공하는 MNIST 데이터를 활용하였습니다.

```
def conv_bn_lrelu_layer(in_channels,out_channels,kernel_size,stride=1,padding=0):
    return nn.Sequential(
        nn.Conv2d(in_channels,out_channels,kernel_size,stride=stride,padding=padding),
        nn.BatchNorm2d(out_channels,momentum=0.1,eps=1e-5),
        nn.LeakyReLU(0.2)
    )
def conv_bn_lrelu_drop_layer(in_channels,out_channels,kernel_size,stride=1,padding=0):
    return nn.Sequential(
        nn.Conv2d(in_channels,out_channels,kernel_size,stride=stride,padding=padding),
        nn.BatchNorm2d(out_channels,momentum=0.1,eps=1e-5),
        nn.LeakyReLU(0.2),
        nn.Dropout(0.5)
    )
def tconv_bn_relu_layer(in_channels,out_channels,kernel_size,stride=1,padding=0):
  return nn.Sequential(
      nn.ConvTranspose2d(in_channels,out_channels,kernel_size,stride=stride,padding=padding),
      nn.BatchNorm2d(out_channels,momentum=0.1,eps=1e-5),
      nn.ReLU()
  )

def tconv_bn_lrelu_layer(in_channels,out_channels,kernel_size,stride=1,padding=0):
  return nn.Sequential(
      nn.ConvTranspose2d(in_channels,out_channels,kernel_size,stride=stride,padding=padding),
      nn.BatchNorm2d(out_channels,momentum=0.1,eps=1e-5),
      nn.ReLU()
  )

def tconv_layer(in_channels,out_channels,kernel_size,stride=1,padding=0):
  return nn.ConvTranspose2d(in_channels,out_channels,kernel_size,stride=stride,padding=padding)
```

MNIST는 기본적으로 이미지 데이터이기 때문에 일반적인 neural network 구조보다는 convolution 계층을 활용한 CNN 구조가 더 적합합니다. G에 사용될 transpose convolution 계층을 tconv로, D에 사용될 convolution계층을 conv로 정의하였습니다. D에서는 leaky relu를 사용하게 되는데 이는 가중치 수렴의 안정성을 위해 사용하였습니다.

```
class D(nn.Module):
  def __init__(self):
    super(D,self).__init__()
    self.down_sample_layer1 = conv_lrelu_layer(c_dim,df_dim,4,stride=2,padding=1)          # 14x14
    self.down_sample_layer2 = conv_bn_lrelu_layer(df_dim,df_dim*2,4,stride=2,padding=1)    #7x7
    self.down_sample_layer3 = conv_bn_lrelu_layer(df_dim*2,df_dim*4,3,stride=2,padding=1)
    self.fc_layer4 = fc_layer(df_dim*4*s_h8*s_w8,K)
    self.softmax = nn.Softmax(dim=-1)

  def forward(self, x):
    x = self.down_sample_layer1(x)
    x = self.down_sample_layer2(x)
    x = self.down_sample_layer3(x)
    x = torch.flatten(x,1)
    x = self.fc_layer4(x)
    x = self.softmax(x)

    return x
```

```
class G(nn.Module):
  def __init__(self):
    super(G,self).__init__()
    self.fc_layer1 = fc_layer(latent_size,s_h8*s_w8*gf_dim*4)
    self.bn_layer1 = nn.BatchNorm2d(gf_dim*4)#4x4
    self.up_sample_layer2 = tconv_bn_relu_layer(gf_dim*4,gf_dim*2,3,stride=2,padding=1)   #7x7
    self.up_sample_layer3 = tconv_bn_relu_layer(gf_dim*2,gf_dim,4,stride=2,padding=1)     #14x14
    self.up_sample_layer4 = tconv_layer(gf_dim,c_dim,4,stride=2,padding=1)                #28x28
    self.tanh = nn.Tanh()


  def forward(self, x):
    x = self.fc_layer1(x)
    x = x.view(-1,gf_dim*4,s_h8,s_w8)
    x = self.bn_layer1(x)
    x = self.up_sample_layer2(x)
    x = self.up_sample_layer3(x)
    x = self.up_sample_layer4(x)
    x = self.tanh(x)
    return x
```

D와 G 모두 3개의 convolution 계층으로 구성되어 있습니다.

```
class MarginalHLoss(nn.Module):
  def __init__(self):
    super(MarginalHLoss, self).__init__()
  def forward(self, x):# NxK
    x = x.mean(axis=0)
    x = -torch.sum(x*torch.log(x+1e-6))
    return x

class JointHLoss(nn.Module):
  def __init__(self):
    super(JointHLoss, self).__init__()
  def forward(self, x):
    x = -x*torch.log(x+1e-6)
    x = 1.0/batch_size*x.sum()
    return x
```

Loss 함수는 위와 같이 구현할 수 있습니다. 엔트로피 연산은 torch.log를 이용하여 쉽게 연산할 수 있으며 1e-6은 python 연산상으로 0이 되지 않도록 안전장치를 만든 것입니다.

```
for ep in range(epochs):
    for i, (data, _) in enumerate(train_loader):
        b_size=data.shape[0]

        data = data.to(device=device)

        #Train D
        D_net.zero_grad()

        y_real = D_net(data)
        joint_entropy_real = jointH(y_real)

        marginal_entropy_real = marginalH(y_real)

        z = torch.randn(b_size,latent_size).to(device=device)
        fake_images = G_net(z)
        y_fake = D_net(fake_images.detach())

        joint_entropy_fake = jointH(y_fake)

        loss_D = joint_entropy_real - marginal_entropy_real - joint_entropy_fake
        loss_D.backward(retain_graph=True)
        D_optimizer.step()

        #Train G
        G_net.zero_grad()

        y_fake = D_net(fake_images)

        marginal_entropy_fake = marginalH(y_fake)

        joint_entropy_fake = jointH(y_fake)

        loss_G = joint_entropy_fake-marginal_entropy_fake

        loss_G.backward(retain_graph=True)
        G_optimizer.step()
```

$$\mathcal{L}_D = \max_D H_{\mathcal{X}}\left[p(y \mid D)\right] - \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[H\left[p(y \mid \mathbf{x}, D)\right]\right] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}\left[H\left[p(y \mid G(\mathbf{z}), D)\right]\right]$$

$$\mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[H\left[p(y \mid \mathbf{x}, D)\right]\right] \quad \#minimize$$

$$H_{\mathcal{X}}\left[p(y \mid D)\right] \quad \#maximize$$

$$\#uniform\ distribution\ sampling$$

$$\mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}\left[H\left[p(y \mid G(\mathbf{z}), D)\right]\right] \quad \#maximize$$

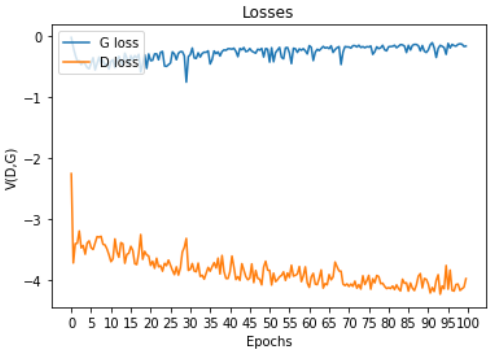
$$\mathcal{L}_G = \min_G -H_G\left[p(y \mid D)\right] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}\left[H\left[p(y \mid G(\mathbf{z}), D)\right]\right]$$

$$H_G\left[p(y \mid D)\right] \quad \#maximize$$

$$\mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}\left[H\left[p(y \mid G(\mathbf{z}), D)\right]\right] \quad \#maximize$$

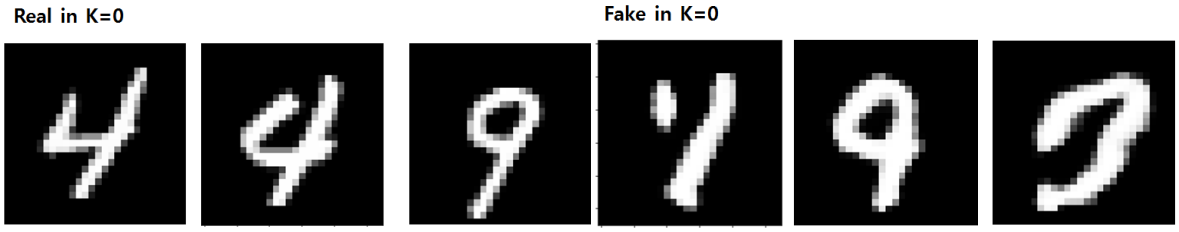
계층을 모두 정의하였으므로, D와 G를 데이터의 batch_size에 따라 훈련을 시킵니다. 여기서 주의할 점은 loss function을 구현할 때 minimize 문제로 풀어야 합니다. 따라서 L_D의 경우 loss 계산 시 코드 상에서는 각 loss term의 부호가 반대로 들어가 있는 것을 확인할 수 있습니다.

```
Epoch [96/100], Step [500/938]
 D_loss: -4.1838, H_x[p(y|D)] : 2.1533, E[H[p(y|x,D)]] : 0.0986, E[H[p(y|G(z),D)]]:2.1291
 G_loss: -0.1631, H_G[p(y|D)] : 2.2922, E[H[p(y|G(z),D)]]: 2.1291
Epoch [97/100], Step [1/938]
 D_loss: -4.0758, H_x[p(y|D)] : 2.0819, E[H[p(y|x,D)]] : 0.1114, E[H[p(y|G(z),D)]]:2.1052
 G_loss: -0.1719, H_G[p(y|D)] : 2.2771, E[H[p(y|G(z),D)]]: 2.1052
Epoch [97/100], Step [500/938]
 D_loss: -4.0697, H_x[p(y|D)] : 2.1490, E[H[p(y|x,D)]] : 0.2340, E[H[p(y|G(z),D)]]:2.1546
 G_loss: -0.1422, H_G[p(y|D)] : 2.2968, E[H[p(y|G(z),D)]]: 2.1546
Epoch [98/100], Step [1/938]
 D_loss: -4.1738, H_x[p(y|D)] : 2.1647, E[H[p(y|x,D)]] : 0.1396, E[H[p(y|G(z),D)]]:2.1487
 G_loss: -0.1329, H_G[p(y|D)] : 2.2817, E[H[p(y|G(z),D)]]: 2.1487
Epoch [98/100], Step [500/938]
 D_loss: -4.1451, H_x[p(y|D)] : 2.1173, E[H[p(y|x,D)]] : 0.1215, E[H[p(y|G(z),D)]]:2.1493
 G_loss: -0.1399, H_G[p(y|D)] : 2.2892, E[H[p(y|G(z),D)]]: 2.1493
Epoch [99/100], Step [1/938]
 D_loss: -4.1210, H_x[p(y|D)] : 2.1767, E[H[p(y|x,D)]] : 0.1713, E[H[p(y|G(z),D)]]:2.1156
 G_loss: -0.1774, H_G[p(y|D)] : 2.2931, E[H[p(y|G(z),D)]]: 2.1156
Epoch [99/100], Step [500/938]
 D_loss: -3.9812, H_x[p(y|D)] : 2.1922, E[H[p(y|x,D)]] : 0.3161, E[H[p(y|G(z),D)]]:2.1052
 G_loss: -0.1703, H_G[p(y|D)] : 2.2755, E[H[p(y|G(z),D)]]: 2.1052
```



100번의 epoch에 대한 G와 D의 loss변화 과정을 그려본 결과 G의 loss는 큰 변화가 없지만 D의 loss는 크게 감소하는 것을 볼 수 있습니다. 이는 G가 생성한 이미지가 점점 실제 이미지와 가까워져 가지만, D의 구별능력이 향상되기 때문에 나오는 결과입니다. 일반적인 GAN에서는 G의 loss도 감소하다가 수렴하는 현상이 관측되는데, CatGAN은 D에 초점을 두고 있기 때문에 G의 loss 변화는 크지 않은 것을 볼 수 있습니다.

학습 완료 후, 실제 이미지와 G가 생성한 이미지를 비교하였습니다. G의 loss가 위의 loss graph처럼 하락하지 않음에도 높은 수준의 이미지를 생성해내는 것을 볼 수 있습니다.



Unsupervised Learning 시에 0 번째 군집(라벨 의미 X)로 분류된 Real데이터와 Fake 데이터의 모습입니다. 4와 9를 굉장히 비슷한 군집으로 묶은 것을 확인할 수 있습니다.

```
In [129]: print(acc)
0.9586454754

In [130]: print(1-acc)
0.0413545246
```

Semi-Supervised Learning 시에 Discriminator만을 이용해서 MNIST 데이터를 분류한 결과 정확도는 95.86%로 논문의 정확도 98.09%보다 낮게 나온 것을 확인할 수 있었습니다.

Unsupervised Learning시에는 10개의 군집과 실제 label의 매칭 과정을 통해 정확도를 측정하였습니다. 정확도는 80.17%로 논문의 정확도 90.3%보다 매우 낮게 측정되었는데, 이는 epoch을 100번만 실행했기 때문에 아직 수렴이 덜된 것으로 생각하였습니다. Label이 있는 데이터를 제공하는 것은 학습의 소요 시간과 수렴 속도에도 영향을 미친다는 것을 알 수 있었습니다.