CS_IOC5008_0856043_HW2 Report

Github link

Brief introduction

1. Development environment

Python version : 3.7.4
Framework : Pytorch
Hardware : NVIDIA GeForce GTX 1080 Ti 11GB

2. How to run the code.

Open the BEGAN-CelebA-CS_IOC5008_0856043_HW2.ipynb
Change the root of the dataset.

The data root should be change!

```
In [61]: data_root = '../data/'
```

Run All

3. After train, the models will store in ./ckpt/

4. Load model and create Samples

Load model and create Samples ¶

In [57]:
```
G_path = sorted(glob.glob(os.path.join('ckpt', '*.pth.tar')))[-2]
print( sorted(glob.glob(os.path.join('ckpt', '*.pth.tar')))[-2])
print(len(sorted(glob.glob(os.path.join('ckpt', '*.pth.tar')))))
state = torch.load(G_path)
G.load_state_dict(state['G'])
```

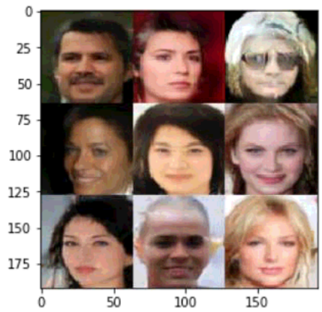
```
ckpt\began093233.pth.tar
5
```

Out[57]:
```
<All keys matched successfully>
```

In [58]:
```
G.eval()
None
```

In [59]:
```
img = get_sample_image(G, n_noise, n_samples=9)
imshow(img)
```

Out[59]:
```
<matplotlib.image.AxesImage at 0x23ee7661808>
```

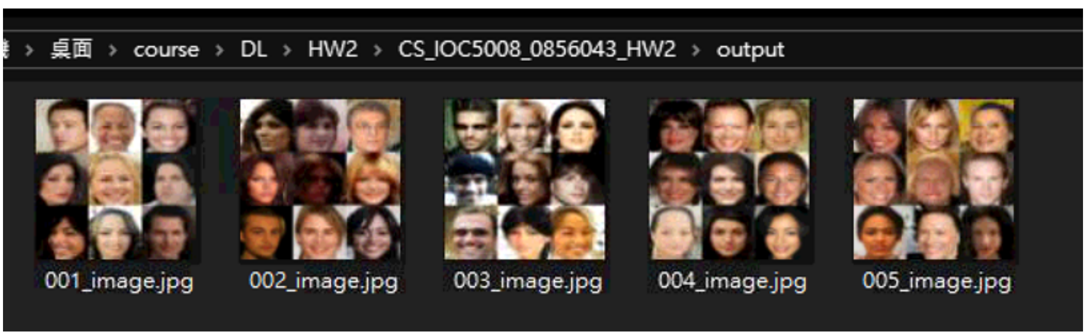


In [29]:
```
def tensor2img(tensor):
    img = (np.transpose(tensor.detach().cpu().numpy(), [1,2,0])+1)/2.
    return img
```

In [30]:
```
def get_sample_image(G, n_noise=100, n_samples=64):
    """
        save sample 100 images
    """
    n_rows = int(np.sqrt(n_samples))
    z = (torch.rand(size=[n_samples, n_noise])*2-1).to(DEVICE) # U[-1, 1]
    x_fake = G(z)
    x_fake = torch.cat([torch.cat([x_fake[n_rows*j+i] for i in range(n_rows)], dim=1) for j in range(n_rows)], dim=2)
    result = tensor2img(x_fake)
    return result
```

In [60]:
```
for i in range(5):
    img = get_sample_image(G, n_noise, n_samples=9)
    img = skimage.transform.resize(img,(450,450))
    imsave('output/{:03d}_image.jpg'.format(i+1), img)
```

The output image will create in ./output/

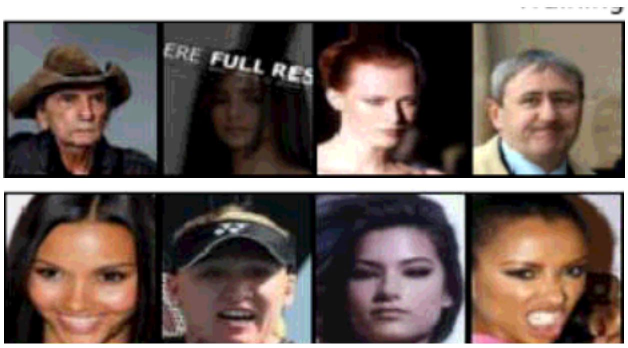


Methodology

1.Data pre-precess

```
transform = transforms.Compose([
                               transforms.Resize((IMAGE_DIM[0]+30,IMAGE_DIM[1]+30)),
                               transforms.CenterCrop(64),
                               transforms.ToTensor(),
                               transforms.Normalize(mean=(0.5, 0.5, 0.5),
                               std=(0.5, 0.5, 0.5))
                              ]
)
```

First, resize the image to (64+30, 64+30) and then Center Crop to (64, 64).
That will make the image focus on the face and decrease the background.



The first image is just resize to (64, 64).
The second image is resize to (64+30, 64+30) and then center crop (64, 64).

2. Model architecture

I using the Boundary Equilibrium GAN (BEGAN) that released on May 2017.

(1)The model is base on the Energy based GAN (EBGAN).

Instead of designing a discriminator similar to a classifier, the discriminator uses an **autoencoder** which extracts latent features of the input image by an encoder and reconstruct it again with the decoder.

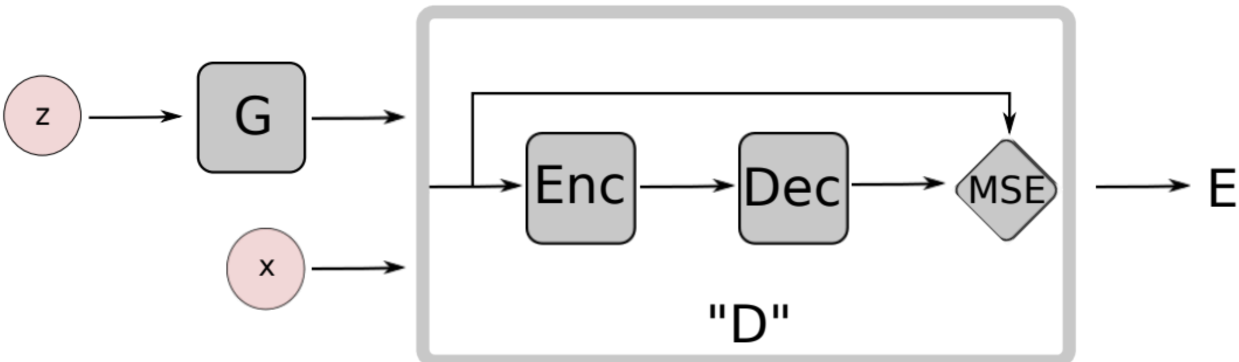
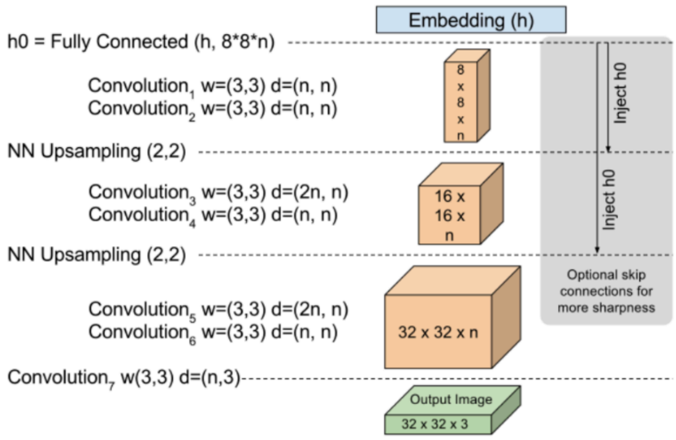


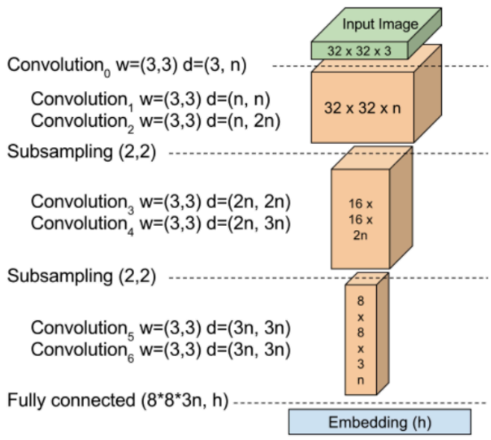
Figure 1: EBGAN architecture with an auto-encoder discriminator.

(2)The model also inspired by the Wasserstein GANs that introduced a loss that also acts as a measure of convergence

The BEGAN Model



(a) Generator/Decoder



(b) Encoder

Implement

```
class Discriminator(nn.Module):
    """
        Convolutional Discriminator
    """
    def __init__(self, in_channel=1, n_filters=128, hidden_dim=64):
        super(Discriminator, self).__init__()
        self.encoder = Encoder(in_channel=in_channel, n_filters=n_filters, hidden_dim=hidden_dim)
        self.decoder = Decoder(out_channel=in_channel, n_filters=n_filters, n_noise=hidden_dim)

    def forward(self, x):
        h = self.encoder(x)
        x_ = self.decoder(h)
        return x_
```

```
class Generator(nn.Module):
    """
        Convolutional Generator
    """
    def __init__(self, out_channel=1, n_filters=128, n_noise=64):
        super(Generator, self).__init__()
        self.decoder = Decoder(out_channel=out_channel, n_filters=n_filters, n_noise=n_noise)

    def forward(self, h):
        x_ = self.decoder(h)
        return x_
```

3.Loss function, Convergence measure

The method aims to match auto-encoder loss distributions using a loss derived from the Wasserstein distance.

Wasserstein distance lower bound for auto-encoders

$$\mathcal{L}(v) = |v - D(v)|^\eta \text{ where } \begin{cases} D : \mathbb{R}^{N_x} \mapsto \mathbb{R}^{N_x} & \text{is the autoencoder function.} \\ \eta \in \{1, 2\} & \text{is the target norm.} \\ v \in \mathbb{R}^{N_x} & \text{is a sample of dimension } N_x. \end{cases}$$

The BEGAN objective is:

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

Convergence measure

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

Implement

```
# Training Discriminator
x = images.to(DEVICE)
x_outputs = D(x)
D_x_loss = criterion(x_outputs, x)

z = (torch.rand(size=[batch_size, n_noise])*2-1).to(DEVICE)
x_fake = G(z)
z_outputs = D(x_fake.detach())
D_z_loss = criterion(z_outputs, x_fake)

D_loss = D_x_loss - k_t*D_z_loss

D.zero_grad()
D_loss.backward()
D_opt.step()


# Training Generator
z = (torch.rand(size=[batch_size, n_noise])*2-1).to(DEVICE)
x_fake = G(z)
z_outputs = D(x_fake)
G_loss = criterion(x_fake, z_outputs)

G.zero_grad()
G_loss.backward()
G_opt.step()

bal = (gamma*D_x_loss - G_loss).detach()
k_t = min(max(k_t + lr_k*bal, 0), 1)
M_global = D_x_loss.detach() + torch.abs(bal)
```

4.Optimizer and learning rate scheduler

```
D_opt = torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
G_opt = torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

```
D_scheduler = torch.optim.lr_scheduler.MultiStepLR(D_opt, milestones=[3, 10, 17], gamma=0.6)
G_scheduler = torch.optim.lr_scheduler.MultiStepLR(G_opt, milestones=[3, 10, 17], gamma=0.6)
```

```
criterion = nn.L1Loss()
```

5.Hyperparameter
IMAGE_DIM = (64, 64, 3)
batch_size = 32
n_noise = 64
max_epoch = 20
lr_k = 0.001
gamma = 0.7

Findings or Summery

1.GAN has a lot of uncertainty, but there are so many variations of GANs try to solve the problem. Such as WGAN try to let the converge better.

2. Even if there are so many strong model can train very well, it can't always output a good result. Sometimes the image has some observable problems.

3.The GAN needs to train for a very long time. In my case, since using the Convolutional Encoder, the parameters are very large in both Generator and Discriminator. But the output is well, so I think it is worthy.

```
a = (sum(p.numel() for p in D.parameters() if p.requires_grad))
b = (sum(p.numel() for p in G.parameters() if p.requires_grad))
print(a)
print(b)
print(a/b)
```

```
13889603
3019651
4.599737850499942
```

Reference

1.The EBGAN paper https://arxiv.org/abs/1609.03126

2.The BEGAN paper https://arxiv.org/abs/1703.10717

3. Introduction to the BEGAN & EBGAN link

4. Introduction to the BEGAN (Chinese) link