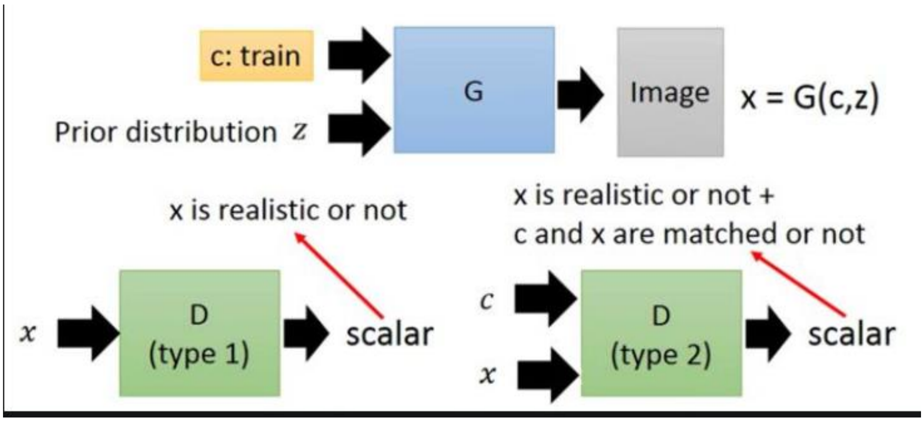
# LAB 07 cGAN and cNF

## Introduction

cGAN :

The ideal of cGAN is let two model improve by each other by training , one is generator which purpose is generating images , another is discriminator which purpose is judging the images created by generator .




NF :

The ideal of NF is to find an invertible transform ( f ) between the data distribution ( Px ) and latent ( Py ) .

$$p_x(\,x\,) = p_y(\,f\,(\,x\,)\,) * |\det J\,f\,(\,x\,)\,|$$
$$p_y(\,y) = p_x(\,f^{-1}(\,y\,)\,) * |\det J\,f^{-1}(\,y\,)\,|$$

## Implement

Task1

1. cGAN
    1. Architecture
       My architecture is DCGANs .
    2. Generator and Discriminator
       To implement conditional GAN , I use the fully connective layer to convert the length of the condition from 24 to 100 at the generator , and the length of latent input is 200 , so the total is 300 , and output size is 64x64 .

```python
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.nz = 300
        self.ngf = 64
        self.fc = nn.Linear(24,100)
        self.act = nn.ReLU()
        self.network = nn.Sequential(
                nn.ConvTranspose2d(self.nz, self.ngf * 8, 4, 1, 0), #
                nn.BatchNorm2d(self.ngf * 8),
                nn.ReLU(True),

                nn.ConvTranspose2d(self.ngf * 8, self.ngf * 4, 4, 2, 1),
                nn.BatchNorm2d(self.ngf * 4),
                nn.ReLU(True),

                nn.ConvTranspose2d(self.ngf * 4, self.ngf * 2, 4, 2, 1),
                nn.BatchNorm2d(self.ngf * 2),
                nn.ReLU(True),

                # state size. (ngf*2) x 16 x 16
                nn.ConvTranspose2d(self.ngf * 2, self.ngf, 4, 2, 1),
                nn.BatchNorm2d(self.ngf),
                nn.ReLU(True),

                # state size. 3 x 64 x 64
                nn.ConvTranspose2d(self.ngf, 3, 4, 2, 1),
                nn.Tanh()
            )

    def forward(self, x, cond):
        cond = cond.permute(0,3,2,1)
        cond = self.act(self.fc(cond.float()).permute(0,3,1,2))
        x = torch.cat([x, cond], 1)
        out = self.network(x.float())

        return out
```

At the discriminator , I use the fully connective layer to convert the size of the condition from 24 to 1x64x64 ( one channel ) and connect it to the input image (three channels) , so the size of input is batch size x 4 x 64 x 64 , and the output is a scalar .

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.ndf = 64
        self.fc = nn.Linear(24,self.ndf*self.ndf)
        self.act = nn.LeakyReLU()
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(4, self.ndf, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),

            # state size. (ndf) x 32 x 32
            nn.Conv2d(self.ndf, self.ndf * 2, 4, 2, 1),
            nn.BatchNorm2d(self.ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(self.ndf * 2, self.ndf * 4, 4, 2, 1),
            nn.BatchNorm2d(self.ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),

            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(self.ndf * 4, self.ndf * 8, 4, 2, 1),
            nn.BatchNorm2d(self.ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),

            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(self.ndf * 8, 1, 4, 1, 0),
            nn.Sigmoid()
        )
    def forward(self, x, cond=None):
        cond = cond.permute(0,2,3,1)
        cond = self.act(self.fc(cond).view(cond.shape[0],1,self.ndf,self.ndf))
        x = torch.cat([x, cond], 1)
        output = self.main(x)

        return output.view(-1, 1).squeeze(0)
```

3. Loss Function

First , The loss function of the discriminator is designed according to the algorithm of the cGANs and I add the weight to each sub loss to improve the performance of the model .

- In each training iteration:
  - Sample m positive examples $\{(c^1,x^1),(c^2,x^2),\dots,(c^m,x^m)\}$ from database
  - Sample m noise samples $\{z^1,z^2,\dots,z^m\}$ from a distribution
  - Obtaining generated data $\{\tilde{x}^1,\tilde{x}^2,\dots,\tilde{x}^m\}$, $\tilde{x}^i = G(c^i,z^i)$
  - Sample m objects $\{\hat{x}^1,\hat{x}^2,\dots,\hat{x}^m\}$ from database
  - Update discriminator parameters $\theta_d$ to maximize
    - $$\boxed{\tilde{V} = \frac{1}{m}\sum_{i=1}^{m} \log D(c^i,x^i) + \frac{1}{m}\sum_{i=1}^{m} \log\left(1 - D(c^i,\tilde{x}^i)\right) + \frac{1}{m}\sum_{i=1}^{m} \log\left(1 - D(c^i,\hat{x}^i)\right)}$$
    - $\theta_d \leftarrow \theta_d + \eta\nabla\tilde{V}(\theta_d)$

There has three sub loss .

```
fake_img = netG(z, conds) # G(c, z) : x~

d_real = netD(inputs, conds.float()) # D(c, x)
d_fake = netD(fake_img.detach(), conds.float()) # D(c, x~)
d_real_head = netD(inputs_head, conds.float()) # D(c, x^)

# Discriminator loss
d_real_loss = criterion(d_real, label_real)
d_fake_loss = criterion(d_fake, label_fake)
d_real_head_loss = criterion(d_real_head, label_fake)

D_loss = 0.1*d_real_loss + 0.1*d_fake_loss + 0.8*d_real_head_loss
```

```
criterion = torch.nn.BCELoss().cuda()
```

```
label_real = torch.ones((inputs.shape[0],1)).cuda() # Real label
label_fake = torch.zeros((inputs.shape[0],1)).cuda() # Fake label
```

Second , The loss function of the generator is also designed according to the algorithm of the cGANs .

- Sample m noise samples$\{z^1,z^2,\dots,z^m\}$ from a distribution
- Sample m conditions $\{c^1,c^2,\dots,c^m\}$ from a database
- Update generator parameters $\theta_g$ to maximize
  - $\tilde{V} = \frac{1}{m}\sum_{i=1}^{m} \log\left(D\left(G\left(c^i,z^i\right)\right)\right)$, $\theta_g \leftarrow \theta_g - \eta\nabla\tilde{V}(\theta_g)$

```
def gen_z(num):
    tmp = torch.zeros((num, 200, 1, 1))
    z = torch.randn_like(tmp).cuda()

    return z
```

```
z = gen_z(inputs.shape[0]) # latent
fake_img2 = netG(z, conds) # G(c, z)
d_fake2 = netD(fake_img2, conds.float()) # D(G(c, z))

# Generator loss
G_loss = criterion(d_fake2, label_real)
G_loss.backward()
optimizer_G.step()
```

4. Implement

Training Function :

My learning rate is 2e-4 , batch size is 64, epochs 200 and for each iteration , generator will be trained 5 times more than the discriminator to make generator powerful enough . This is what one epoch do .

```
optimizer_G = torch.optim.Adam(netG.parameters(), lr=args.lr, betas=(0.5, 0.99))
optimizer_D = torch.optim.Adam(netD.parameters(), lr=args.lr, betas=(0.5, 0.99))
```

```
for i_batch, sampled_batched in enumerate(train_dataloader):
    netG.train()
    netD.train()
    # -------------------------------------- Setup ----------------------------------- #
    inputs = sampled_batched['Image'].cuda() # shape batch 3 64 64 ( h w ) , x
    conds = sampled_batched['cond'].cuda() # shape batch 1 24

    train_dataloader_switch = DataLoader(train_datasets_switch, batch_size = inputs.shape[0], shuffle = True)
    inputs_head = next(iter(train_dataloader_switch))['Image'].cuda() # shape batch 3 64 64 ( h w ) , x^

    z = gen_z(inputs.shape[0]) # latent : shape batch 100 1 1 ( h w ) , z
    label_real = torch.ones((inputs.shape[0],1)).cuda() # Real label
    label_fake = torch.zeros((inputs.shape[0],1)).cuda() # Fake label

    # -------------------------------------Training Discriminator ------------------------------------ #
    optimizer_D.zero_grad()
    fake_img = netG(z, conds) # G(c, z) : x~

    d_real = netD(inputs, conds.float()) # D(c, x)
    d_fake = netD(fake_img.detach(), conds.float()) # D(c, x~)
    d_real_head = netD(inputs_head, conds.float()) # D(c, x^)

    # Discriminator loss
    d_real_loss = criterion(d_real, label_real)
    d_fake_loss = criterion(d_fake, label_fake)
    d_real_head_loss = criterion(d_real_head, label_fake)

    D_loss = 0.1*d_real_loss + 0.1*d_fake_loss + 0.8*d_real_head_loss


    loss_d.append(D_loss.item())
    D_loss.backward()
    optimizer_D.step()
```

```
# -------------------------------------- Training Generator ------------------------------------ #
label_real = 0.9*label_real
for _ in range(args.cycle):
    optimizer_G.zero_grad()

    z = gen_z(inputs.shape[0]) # latent
    fake_img2 = netG(z, conds) # G(c, z)
    d_fake2 = netD(fake_img2, conds.float()) # D(G(c, z))

    # Generator loss
    G_loss = criterion(d_fake2, label_real)
    G_loss.backward()
    optimizer_G.step()
```

Testing Function :

```
z_test = gen_z(len(test_datasets))
```

```
with torch.no_grad():
    conds_test = next(iter(test_dataloader))['cond'].cuda()
    gen_img = netG(z_test, conds_test)
    conds_test = torch.squeeze(conds_test, -1)
    conds_test = torch.squeeze(conds_test, -1)
    score = Eval.eval(gen_img, conds_test)

    conds_new = next(iter(new_dataloader))['cond'].cuda()
    gen_img_new = netG(z_test, conds_new)
    conds_new = torch.squeeze(conds_new, -1)
    conds_new = torch.squeeze(conds_new, -1)
    score_new = Eval.eval(gen_img_new, conds_new)
```

The size of the z_test is 32x200x1x1 , and test the test.json and new.json .

Dataloader :

The dataloader has two mode , one is training : it will return both images and conditions , another is testing : it will only return conditions and you can select test.json or new.json .

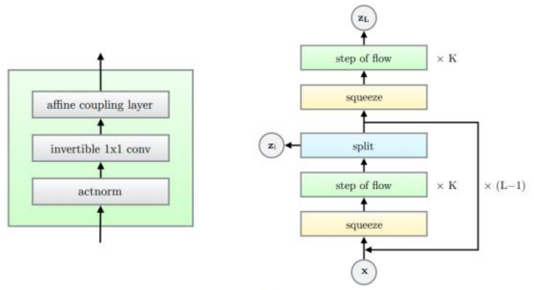
```python
def get_iCLEVR_data(root_folder,mode):
    if mode == 'train':
        data = json.load(open(os.path.join(root_folder,'train.json')))
        obj = json.load(open(os.path.join(root_folder,'objects.json')))
        img = list(data.keys())
        label = list(data.values())
        for i in range(len(label)):
            for j in range(len(label[i])):
                label[i][j] = obj[label[i][j]]
            tmp = np.zeros(len(obj))
            tmp[label[i]] = 1
            label[i] = tmp
        return np.squeeze(img), np.squeeze(label)

    else:
        if mode == 'test':
            print('get test data')
            data = json.load(open(os.path.join(root_folder,'test.json')))
        else:
            print('get new data')
            data = json.load(open(os.path.join(root_folder,'new.json')))
        obj = json.load(open(os.path.join(root_folder,'objects.json')))
        label = data
        for i in range(len(label)):
            for j in range(len(label[i])):
                label[i][j] = obj[label[i][j]]
            tmp = np.zeros(len(obj))
            tmp[label[i]] = 1
            label[i] = tmp

        return None, label
```

```python
class ICLEVRLoader(data.Dataset):
    def __init__(self, root_folder, trans=None, cond=False, mode='train'):
        self.root_folder = root_folder
        self.mode = mode
        self.img_list, self.label_list = get_iCLEVR_data(root_folder,mode)
        if self.mode == 'train':
            print("> Found %d images..." % (len(self.img_list)))

        # self.cond = cond
        # self.num_classes = 24
        self.transform = transforms.Compose([transforms.ToPILImage(),
                                             transforms.Resize((64,64)),
                                             transforms.ToTensor(),
                                             transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])

    def __len__(self):
        """'return the size of dataset"""
        return len(self.label_list)

    def __getitem__(self, idx):
        if self.mode == 'train':
            image = cv2.imread(self.root_folder+'/images/'+self.img_list[idx])[:,:,[2,1,0]]
            self.shape = image.shape
            Cond = self.label_list[idx]
            image_tensor = self.transform(image).float()
            Cond = np.expand_dims(Cond, 1)
            Cond = np.expand_dims(Cond, 2)

            sample = {"Image": image_tensor, "cond": Cond}
        else:
            Cond = self.label_list[idx]
            Cond = np.expand_dims(Cond, 1)
            Cond = np.expand_dims(Cond, 2)

            sample = {"cond": Cond}

        return sample
```

2. cNF

1. Architecture

I choose the Glow . : https://github.com/5yearsKim/Conditional-Normalizing-Flow




(a) One step of our flow.

(b) Multi-scale architecture (Dinh et al., 2016).

| Description | Function | Reverse Function | Log-determinant |
|---|---|---|---|
| Actnorm.<br>See Section 3.1. | $\forall i,j: \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$ | $\forall i,j: \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$ | $h \cdot w \cdot \mathtt{sum}(\log\|\mathbf{s}\|)$ |
| Invertible $1 \times 1$ convolution.<br>$\mathbf{W}: [c \times c]$.<br>See Section 3.2. | $\forall i,j: \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$ | $\forall i,j: \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$ | $h \cdot w \cdot \log\|\det(\mathbf{W})\|$<br>or<br>$h \cdot w \cdot \mathtt{sum}(\log\|\mathbf{s}\|)$<br>(see eq. (10)) |
| Affine coupling layer.<br>See Section 3.3 and<br>(Dinh et al., 2014) | $\mathbf{x}_a, \mathbf{x}_b = \mathtt{split}(\mathbf{x})$<br>$(\log\mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{x}_b)$<br>$\mathbf{s} = \exp(\log\mathbf{s})$<br>$\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$<br>$\mathbf{y}_b = \mathbf{x}_b$<br>$\mathbf{y} = \mathtt{concat}(\mathbf{y}_a, \mathbf{y}_b)$ | $\mathbf{y}_a, \mathbf{y}_b = \mathtt{split}(\mathbf{y})$<br>$(\log\mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{y}_b)$<br>$\mathbf{s} = \exp(\log\mathbf{s})$<br>$\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$<br>$\mathbf{x}_b = \mathbf{y}_b$<br>$\mathbf{x} = \mathtt{concat}(\mathbf{x}_a, \mathbf{x}_b)$ | $\mathtt{sum}(\log(\|\mathbf{s}\|))$ |

Paper : https://arxiv.org/pdf/1807.03039.pdf

2. Model

The source code is a conditional NF , but its origin condition size looks is same as the images ( channels x64x64 ) , so I use the fully connective layer to change my conditions from 24 to 3x64x64 ( the num of channel doesn't have to be 3 , it can be any number ) .

```python
class Glow(nn.Module):

    def __init__(self, num_channels, num_levels, num_steps, mode='sketch'):
        super(Glow, self).__init__()

        # Use bounds to rescale images before converting to logits, not learned
        self.register_buffer('bounds', torch.tensor([0.95], dtype=torch.float32))
        self.flows = _Glow(in_channels=4 * 3,  # RGB image after squeeze
                           cond_channels=3,
                           mid_channels=num_channels,
                           num_levels=num_levels,
                           num_steps=num_steps)
        self.mode = mode
        self.fc = nn.Linear(24,3*64*64)
        self.act = nn.ReLU()

    def forward(self, x, x_cond, reverse=False):
        if reverse:
            sldj = torch.zeros(x.size(0), device=x.device)
        else:
            # Expect inputs in [0, 1]
            if x.min() < 0 or x.max() > 1:
                raise ValueError('Expected x in [0, 1], got min/max {}/{}'
                                 .format(x.min(), x.max()))

            # De-quantize and convert to logits
            x, sldj = self._pre_process(x)
        if self.mode == 'gray':
            x_cond, _ = self._pre_process(x_cond)

        x_cond = x_cond.permute(0,3,2,1)
        x_cond = self.act(self.fc(x_cond.float()).permute(0,3,1,2).view(x_cond.shape[0],3, 64,64))
        x = squeeze(x)
        x_cond = squeeze(x_cond)
        x, sldj = self.flows(x, x_cond, sldj, reverse)
        x = squeeze(x, reverse=True)
```

The red boxes are my implementation .

Algorithm of loss function :

$$\log\left(p_X(x)\right) = \log\left(p_Z(f(x))\right) + \log\left(\left|\det\left(\frac{\partial f(x)}{\partial x^T}\right)\right|\right)$$

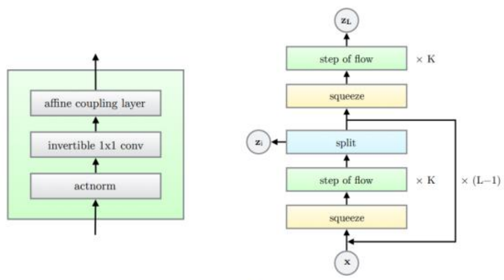
```
criterion = util.NLLLoss().cuda()
```

```
class NLLLoss(nn.Module):
    """Negative log-likelihood loss assuming isotropic gaussian with unit norm.
    Args:
        k (int or float): Number of discrete values in each input dimension.
            E.g., `k` is 256 for natural images.
    See Also:
        Equation (3) in the RealNVP paper: https://arxiv.org/abs/1605.08803
    """
    def __init__(self, k=512):
        super(NLLLoss, self).__init__()
        self.k = k

    def forward(self, z, sldj):
        prior_ll = -0.5 * (z ** 2 + np.log(2 * np.pi))
        prior_ll = prior_ll.flatten(1).sum(-1) \
            - np.log(self.k) * np.prod(z.size()[1:])
        ll = prior_ll + sldj
        nll = -ll.mean()

        return nll
```

3. Implement

Training Function :

My learning rate is 2e-4 , batch size is 16, epochs 150 , num_channel is 512 ( the channel in coupling , invertible conv, actnorm ) , num_level is 4 ( L ), num_steps is 6 ( K ) .( The source code using 1e-3, 4, 128, 3, 8 )




(a) One step of our flow. (b) Multi-scale architecture (Dinh et al., 2016).

This is what one epoch do . ( reverse : False : data to latent , True : latent to data )

```
optimizer = optim.Adam(net.parameters(), lr=args.lr)
```

```
for i_batch, sampled_batched in enumerate(train_dataloader):
    print(str(i_batch) + '/'+str(int(len(train_datasets)/args.batch_size)+1),end='\r')

    inputs = sampled_batched['Image'].cuda()
    conds = sampled_batched['cond'].cuda()

    optimizer.zero_grad()

    z, sldj = net(inputs.float(), conds.float(), reverse=False)

    loss = criterion(z, sldj)
    loss_meter.update(loss.item(), z.size(0))
    loss.backward()

    optimizer.step()
```

Testing Function :

The test_dataloader can be test.json or new.json .

```
z_test = torch.randn((32, 3, 64, 64), dtype=torch.float32).cuda()
conds_test = next(iter(test_dataloader))['cond'].cuda()
```

```
with torch.no_grad():
    gen_img, _ = net(z_test.float(), conds_test.float(), reverse=True)
    gen_img = torch.tanh(gen_img)
    score = Eval.eval(gen_img, conds_test)
```

Dataloader :

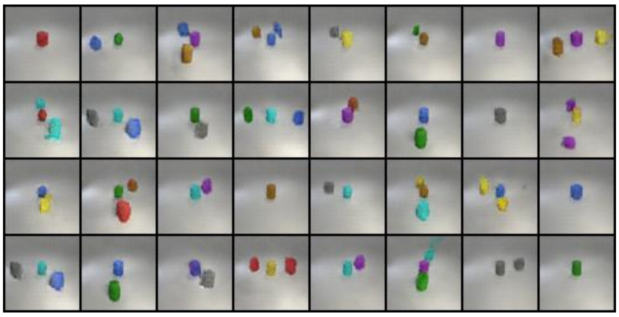
The dataloader is same as cGANs .

Result :

cGAN :

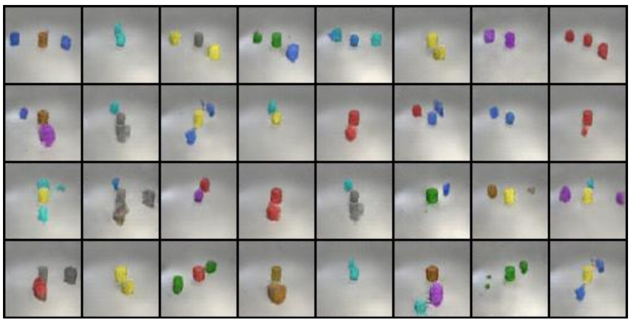
1. Test

```
Score :  0.7083333333333334
```



2. New

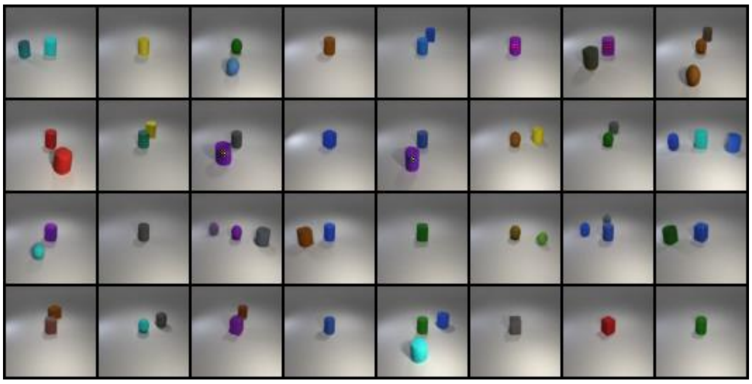
```
Score :  0.7085714285714286
```



cNF :

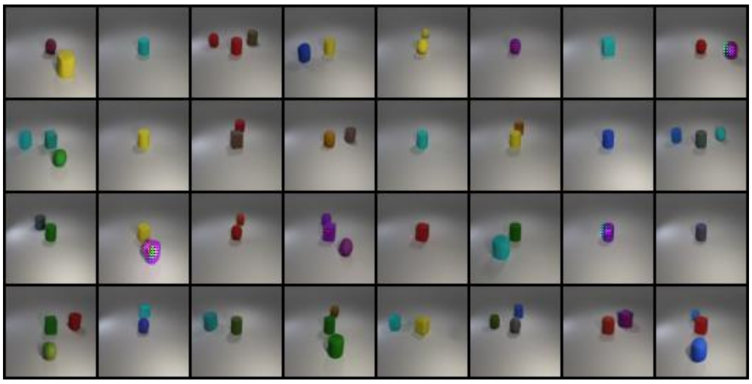
1. Test

Score : 0.6033333333333334



2. New

Score : 0.6071428571428571



Discussion :

In the case of cGAN , It is hard to balance the abilities of generator and discriminator . If one of them is too powerful than another one , the training may fail . During this lab , I tried different learning rate , weight of sub loss in discriminator and the num of training times for generator in one iteration . I found that adjusting the weight of loss is the most efficient method to find good enough result . If using too big learning rate (like 0.001) , the loss can't smoothly reduce , if too small , it takes too much time for training ( It took me 1.5 days to get my final result . ) . If using more training times for generator also takes too much time . After set the appropriate parameters , I only modified the weight loss and finally I use 0.1, 0.1, 0.8 for three sub loss :

```
d_real = netD(inputs, conds.float()) # D(c, x)
d_fake = netD(fake_img.detach(), conds.float()) # D(c, x~)
d_real_head = netD(inputs_head, conds.float()) # D(c, x^)

# Discriminator loss
d_real_loss = criterion(d_real, label_real)
d_fake_loss = criterion(d_fake, label_fake)
d_real_head_loss = criterion(d_real_head, label_fake)

D_loss = 0.1*d_real_loss + 0.1*d_fake_loss + 0.8*d_real_head_loss
```

For adding conditions , I just make its size close to image (1x64x64) and use the activation function .

In the case of NF , because I used Glow , it has some parameters for model structure , like num_channel , num_level and num_steps . Although using bigger parameters , the model will be better , but it also takes more vram and time .(I use 512/4/6 , batch size 16 , epoch 150 , it needs 6G vram and 1.5 days to train .) I found that make conditions have more channels will have more chance to get better result , so I use fully connective layer to change size from 24 to 3x64x64 . For the learning rate , I just using 2e-4 , same as cGAN case . I also found that appropriate small batch size will get better result , as my testing , 16 is the best .

The different between the two model , is the target of learning . NF is to find an invertible transform for data distribution and latent , but GAN is like teaching it to generate a data like image . In theory , NF is easier to training than GAN .

Task2

The NF model I used in this part is totally same as NF model in task1 , the only different is that origin conditions size is 40 and epochs is 50 .

For three tasks , the latent I used is first using two latent to interpolate and select the middle one as input latent , that will make higher quality result image .

```
def interpolate(z1=None, z2=None):
    split = False
    input_z = torch.zeros((1, 3, 64, 64), dtype=torch.float32).cuda()
    if (z1 == None ) and (z2 == None):
        split = True
        z1 = torch.randn((1, 3, 64, 64), dtype=torch.float32).cuda()
        z2 = torch.randn((1, 3, 64, 64), dtype=torch.float32).cuda()

    input_z = torch.cat([input_z, z1], 0)
    tmp = None

    for i in range(6):
        tmp = torch.lerp(z1, z2, 0.125*(i+1)).cuda()
        input_z = torch.cat([input_z, tmp], 0)

    input_z = torch.cat([input_z, z2], 0)
    input_z = input_z[1:]

    if split:
        return input_z[4].unsqueeze(0)
    else:
        return input_z
```

## Conditional face generation :

I choose 4 data in training datasets and change the condition label .

```
4
5_o_Clock_Shadow Arched_Eyebrows Attractive Bags_Under_Eyes Bald Bangs Big_Lips Big_Nose Black_Hair Blond_Hair Blurry Brown_Hair Bushy_Ey
0.jpg  1 1 1 1 -1 -1 1 -1 1 1 -1 1 1 -1 -1 -1 -1 -1 1 1 -1 1 -1 -1 1 -1 -1 1 -1 -1 -1 1 -1 1 -1 -1 1 -1 -1 1
1.jpg  -1 -1 1 -1 -1 1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 1 -1 1 -1 -1 -1 -1 1 -1 1 -1 -1 1 -1 -1 1
2.jpg  -1 -1 1 1 -1 -1 1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 -1 1 1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 1 -1 1 1 -1 -1 -1 -1 1
3.jpg  -1 -1 -1 -1 -1 -1 -1 1 1 -1 1 -1 1 -1 -1 -1 -1 -1 1 1 -1 1 -1 -1 1 -1 1 -1 -1 -1 -1 1 -1 1 -1 -1 1 -1 -1 1
```

```
test_datasets = CelebALoader('/content/data/task_2/', cond=True)
test_dataloader = DataLoader(test_datasets, batch_size = 4, shuffle = False)
```

```
sampled_batched = next(iter(test_dataloader))
conds = sampled_batched['cond'].cuda()
```

Randomly initial 4 latent and add the conditions .

```
Z = None
for i in range(conds.shape[0]):
  z = interpolate()
  if i == 0:
    Z = z.clone()
  else:
    Z = torch.cat([Z, z], 0)

img, _ = net(Z.float(), conds.float(), reverse=True)
img = torch.tanh(img)
util.save_image(img, os.path.join('/content/z1.png'), nrow=8, normalize=True)
```

Image :

## Linear interpolation :

I randomly generate two latent and interpolate 6 images and randomly choose the conditions in datasets .

```
test_datasets = CelebALoader('/content/data/task_2/', cond=True)
test_dataloader = DataLoader(test_datasets, batch_size = 1, shuffle = False)
```

```
sampled_batched = next(iter(test_dataloader))
conds = sampled_batched['cond'].cuda()
```

```
for i in range(conds.shape[0]):
  cond = conds[i].unsqueeze(0)
  z_1 = interpolate()
  z_2 = interpolate()
  z = interpolate(z_1, z_2)
  img, _ = net(z.float(), cond.float(), reverse=True)
  img = torch.tanh(img)
  util.save_image(img, os.path.join('/content/z_'+str(i)+'.png'), nrow=8, normalize=True)
  print('Generate one ...')
```

Images :

## Attribute manipulation :

I select two attributes : smiling and gold color hair . I choose five to ten images that contain the attribute from datasets , get the average of the sum of those latent generated by model and use different scalar when add it to the target image latent . Randomly choose the conditions in datasets .

In dataloader : select attribute images . ( 505 is target image )

```
IDX = ['154.jpg', '222.jpg', '32.jpg', '297.jpg', '333.jpg', '505.jpg']
IDX = ['1184.jpg', '1208.jpg', '1061.jpg', '297.jpg', '1046.jpg', '1141.jpg', '1378.jpg', '1192.jpg', '520.jpg', '368.jpg', '505.jpg']
Img_ = []
label_ = []
for name in IDX:
  ind = img_list.index(name)
  Img_.append(img_list[ind])
  label_.append(label_list[ind])

return Img_, label_
```

```
test_datasets = CelebALoader('/content/data/task_2/', cond=True)
num = len(train_test)
test_dataloader = DataLoader(train_datasets, batch_size = num, shuffle = False)
```

```
sampled_batched = next(iter(test_dataloader))
conds = sampled_batched['cond'].cuda()
inputs = sampled_batched['Image'].cuda()
```

```
# get attribute latent
z_pos = 0
for i in range(num-1):
  cond = conds[i].unsqueeze(0)
  input_ = inputs[i].unsqueeze(0)
  z_, _ = net(input_.float(), cond.float(), reverse=False)
  z_pos += z_

z_pos = z_pos/(num-1)
```

```
# addd to target image (no smile)
cond = conds[-1].unsqueeze(0)
input_ = inputs[-1].unsqueeze(0)
z_in, _ = net(input_.float(), cond.float(), reverse=False)

Input = torch.zeros((1, 3, 64, 64), dtype=torch.float32).cuda()
Input = torch.cat([Input, z_in], 0)
for i in range(4):
  tmp = z_in + 0.25*(i+1)*z_pos
  Input = torch.cat([Input, tmp], 0)
Input = Input[1:]
img, _ = net(Input, cond.float(), reverse=True)
util.save_image(img, os.path.join('/content/z3.png'), nrow=8, normalize=True)
```

Images :

Smiling :

Gold color hair :