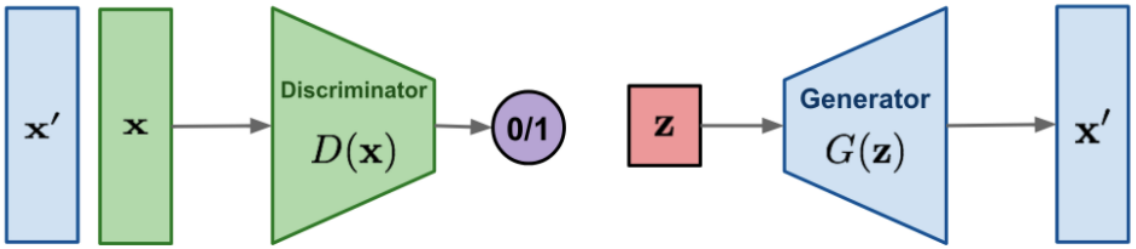
Lab 7 Report
309551064 張凱翔

● Introduction

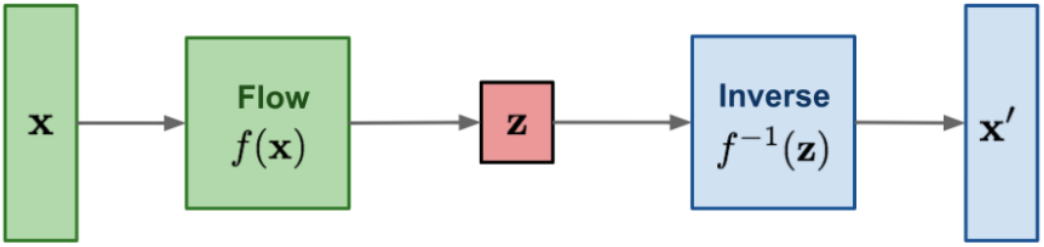
There are two tasks in this lab:

1. Use conditional GAN and conditional NF to generate image of objects according to condition
2. Use conditional NF to generate human face according to condition, including conditional face generation, linear interpolation and attribute manipulation

GAN contains two models, generator and discriminator. The discriminator model learns to distinguish the real data from the fake samples that are produced by the generator model. The generator model learns to fool the discriminator model into believing the fake samples it generated as real. Two models are trained as they are playing a minimax game.



NF transforms a simple distribution into a complex one by applying a sequence of invertible transformation functions. Flowing through a chain of transformations, repeatedly substitute the variable for the new one according to the change of variables theorem and eventually obtain a probability distribution of the final target variable.



● Implementation details

○ Describe how you implement your model

■ Task 1

I use conditional Self-Attention GAN and conditional GLOW for this task and the detailed explanation for two types of models is shown below:

Conditional Self-Attention GAN

I reorganize the provided function to get the data wanted in my way. Then, I call the function ‘get_iCLEVR_data’ with the mode specified to get training data and testing data.

```
train_data = get_iCLEVR_data('train')
test_data = get_iCLEVR_data('test')
```

This function contains two parts, which are training mode and testing mode. Training mode reads the ‘train.json’ to get image names and their corresponding labels. Then, use the ‘objects.json’ to transform literal labels into a vector of 24 dimensions where the index of the object that appears in the image is set to 1. Finally, return an array of image names and an array of their labels. Testing mode only goes through the same procedure but only

returns an array of labels since we need to generate the corresponding images with noise.

```
def get_iCLEVR_data(mode):
    if mode == 'train':
        data = json.load(open('train.json'))
        obj = json.load(open('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:
        data = json.load(open('test.json'))
        obj = json.load(open('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
```

After getting the data, I call the class ‘ICLEVRDataset’ to prepare the dataset for training and testing.

```
train_dataset = ICLEVRDataset(train_data, 'train')
test_dataset = ICLEVRDataset(test_data, 'test')
```

This class of dataset initially splits data into images and labels, then declares a transformer that resizes, permutes and normalizes the image. Function ‘__getitem__’ contains two parts, which are training mode and testing mode. In training mode, get the image name in the array and read the image.Then, apply transformations to the image. Finally, return the image data and its label of that index. In testing mode, only return the label of that index.

```
class ICLEVRDataset(Dataset):
    def __init__(self, data, mode):
        self.img_list, self.label_list = data
        self.mode = mode
        self.transform = transforms.Compose([transforms.Resize((IMG_SIZE, IMG_SIZE)),
                                             transforms.ToTensor(),
                                             transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5, 0.5))])
    def __len__(self):
        return len(self.label_list)

    def __getitem__(self, index):
        if self.mode == 'train':
            image_path = './images/' + self.img_list[index]
            img = Image.open(image_path).convert('RGB')
            img = self.transform(img).numpy()

            return img, self.label_list[index]

        if self.mode == 'test':
            return self.label_list[index]
```

I call DataLoader to automatically get a batch of data with training data shuffled.

```
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers = 8)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

I use ‘BCEloss’ as loss function, ‘Adam’ as optimizer, ‘model_G’ as generator and ‘model_D’ as discriminator. ‘model_E’ is the evaluation model provided by TA.

```
adversarial_loss = torch.nn.BCELoss()

model_G = Generator()
model_D = Discriminator()
model_E = evaluation_model()

optimizer_G = torch.optim.Adam(model_G.parameters(), lr=LR_G, betas=(B1, B2))
optimizer_D = torch.optim.Adam(model_D.parameters(), lr=LR_D, betas=(B1, B2))

model_G.to(device)
model_D.to(device)
adversarial_loss.to(device)
```

Now, let's go through the training process. Before getting into the training process, I initialize two variables to store the best score and a fixed noise for testing in order to reproduce the result. First, train the discriminator where the inputs of the discriminator are the real images and their labels. The discriminator should predict them to be real, so the loss is calculated with predictions and a vector of 1. Then, train the discriminator where the inputs of the discriminator are the images generated by the generator and the labels in training data. The discriminator should predict them to be fake, so the loss is calculated with predictions and a vector of 0. At last, I train the generator two times to strengthen the performance of it. Generate the images with noise and use them and their labels as the inputs of the discriminator. The purpose of the generator is to fool the discriminator into believing the fake samples it generated as real. Therefore, the loss is calculated with predictions and a vector of 1. I print the loss of the discriminator to monitor whether the process is going into failure mode.

```
best = 0
fix_z = torch.randn(32, LATENT_DIM).to(device)
for epoch in tqdm(range(EPOCHS)):
    model_G.train()
    model_D.train()
    for imgs, labels in train_loader:
        imgs = imgs.to(device).float()
        labels =  labels.to(device).float()

        real = torch.full((imgs.size(0), 1), 1., requires_grad=False, device=device)
        fake = torch.full((imgs.size(0), 1), 0., requires_grad=False, device=device)

        # Train Discriminator with real image
        optimizer_D.zero_grad()
        pred = model_D(imgs, labels)
        real_loss = adversarial_loss(pred, real)
        real_loss.backward()
        optimizer_D.step()

        # Train Discriminator with fake image
        optimizer_D.zero_grad()
        z = torch.randn(imgs.size(0), LATENT_DIM).to(device)
        gen_imgs = model_G(z, labels)
        pred = model_D(gen_imgs[0].detach(), labels)
        fake_loss = adversarial_loss(pred, fake)
        fake_loss.backward()
        optimizer_D.step()

        # Train Generator with fake image
        for _ in range(2):
            optimizer_G.zero_grad()
            z = torch.randn(imgs.size(0), LATENT_DIM).to(device)
            gen_imgs = model_G(z, labels)
            predicts = model_D(gen_imgs[0], labels)
            loss_g = adversarial_loss(predicts, real)
            loss_g.backward()
            optimizer_G.step()

    print(real_loss.item() + fake_loss.item())
```

At the end of every epoch, I use the evaluation model and fix noise mentioned above to evaluate the generator and save the images generated as well as the best generator.

```
model_G.eval()
model_D.eval()
with torch.no_grad():
    for labels in test_loader:
        labels = labels.to(device).float()
        gen_imgs = model_G(fix_z, labels)
        score = model_E.eval(gen_imgs[0], labels)
        if best < score:
            best = score
            torch.save(model_G.state_dict(), 'generator.pt')
        save_image(gen_imgs[0].data, './output/%d.png' % epoch, nrow=6, normalize=True)
        print('EPOCHS {} : {}'.format(epoch, score))
        print(best)
```

Finally, let's take a look at the generator and discriminator. My generator and discriminator is Self-Attention GAN and the conditional architecture is cGAN. The generator takes noises and labels as inputs. The overall architecture of the generator is five transposed convolutional layers and two self-attention layers. I use a linear layer to project condition into latent dimension, concatenate with noise and add two dimensions to it. Then, go through three transposed convolutional layers with spectral normalization which is used to stabilize training procedure and batch normalization. Next, go through a self-attention layer followed by a transposed convolutional layer to downsize the output to demand dimension. Finally, go through a self-attention layer also followed by a transposed convolutional layer to get the image generated.

```
class Generator(nn.Module):
    def __init__(self, conv_dim=64):
        super(Generator, self).__init__()
        self.conditionExpand = nn.Sequential(nn.Linear(24, LATENT_DIM), nn.LeakyReLU())

        repeat_num = int(np.log2(self.imsize)) - 3
        mult = 2 ** repeat_num
        layer1 = []
        layer1.append(SpectralNorm(nn.ConvTranspose2d(LATENT_DIM + LATENT_DIM, conv_dim * mult, 4)))
        layer1.append(nn.BatchNorm2d(conv_dim * mult))
        layer1.append(nn.ReLU())

        curr_dim = conv_dim * mult
        layer2 = []
        layer2.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4, 2, 1)))
        layer2.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer2.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)
        layer3 = []
        layer3.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4, 2, 1)))
        layer3.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer3.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)
        layer4 = []
        layer4.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4, 2, 1)))
        layer4.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer4.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)
        last = []
        last.append(nn.ConvTranspose2d(curr_dim, 3, 4, 2, 1))
        last.append(nn.Tanh())

        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)
        self.l4 = nn.Sequential(*layer4)
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn(128, 'relu')
        self.attn2 = Self_Attn(64,  'relu')
```

```
def forward(self, z, c):
    c = self.conditionExpand(c).view(-1, LATENT_DIM)
    z = torch.cat((z, c), dim=1)
    z = z.view(z.size(0), z.size(1), 1, 1)
    out = self.l1(z)
    out = self.l2(out)
    out = self.l3(out)
    out = self.attn1(out)
    out = self.l4(out)
    out = self.attn2(out)
    out = self.last(out)
    return out
```

The discriminator takes images and labels as inputs. The overall architecture of the discriminator is five convolutional layers and two self-attention layers. I use a linear layer to project condition into image size * image size dimension and resize it to (1, image size, image size), concatenate with noise in order to add a channel to it. Then, go through three convolutional layers with spectral normalization which is used to stabilize training procedure. Next, go through a self-attention layer followed by a convolutional layer to upsize the output to demand dimension. Finally, go through a self-attention layer also followed by a convolutional layer and apply a sigmoid function to get the prediction between 0 and 1.

```
class Discriminator(nn.Module):
    def __init__(self, conv_dim=64):
        super(Discriminator, self).__init__()
        self.conditionExpand = nn.Sequential(nn.Linear(24, IMG_SIZE * IMG_SIZE), nn.LeakyReLU())

        layer1 = []
        layer1.append(SpectralNorm(nn.Conv2d(3 + 1, conv_dim, 4, 2, 1)))
        layer1.append(nn.LeakyReLU(0.1))

        curr_dim = conv_dim
        layer2 = []
        layer2.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer2.append(nn.LeakyReLU(0.1))

        curr_dim = curr_dim * 2
        layer3 = []
        layer3.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer3.append(nn.LeakyReLU(0.1))

        curr_dim = curr_dim * 2
        layer4 = []
        layer4.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer4.append(nn.LeakyReLU(0.1))
        self.l4 = nn.Sequential(*layer4)

        curr_dim = curr_dim * 2
        last = []
        last.append(nn.Conv2d(curr_dim, 1, 4))

        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn(256, 'relu')
        self.attn2 = Self_Attn(512, 'relu')
        self.sigmoid = nn.Sigmoid()
```

```
def forward(self, x, c):
    c = self.conditionExpand(c).view(-1, 1, IMG_SIZE, IMG_SIZE)
    x = torch.cat((x, c), dim=1)
    out = self.l1(x)
    out = self.l2(out)
    out = self.l3(out)
    out = self.attn1(out)
    out = self.l4(out)
    out = self.attn2(out)
    out = self.last(out)
    return self.sigmoid(out.squeeze(3).squeeze(2))
```

Conditional GLOW

This part was the last part I completed, so I just simply combined the function from other parts. Preliminary work for data preparation is the same as mentioned in self-attention GAN. The training process is similar to the self-attention GAN and the only difference is the way of getting generated images and loss function. The model architecture is also the same as the GLOW model mentioned in task 2 with some hyperparameter changed.

```
for epoch in tqdm(range(EPOCHS)):
    model.train()
    for imgs, labels in train_loader:
        optimizer.zero_grad()

        imgs = imgs.to(device).float()
        labels = labels.to(device).float()
        num_label = torch.sum(labels, 1).unsqueeze(1) - 1
        z, sldj, pred = model(imgs, labels, reverse=False)
        loss = loss_fn(z, sldj)
        loss.backward()
        optimizer.step()

    model.eval()
    for labels in test_loader:
        labels = labels.to(device).float()
        z, sldj = model(fix_noise, labels, reverse=True)

        score = model_E.eval(z, labels)
        if best < score:
            best = score
            torch.save(model.state_dict(), 'nf.pt')
        print('EPOCHS {} : {}'.format(epoch, score))
        save_image(z.data, './output_NF/%d.png' % epoch, nrow=8, normalize=True)
        print('BEST: {}'.format(best))
        print()
```

■ Task 2

I use conditional GLOW for first application and GLOW for second application and third application and the detailed explanation for two types of models is shown below:

Conditional GLOW

Because I finish the work of GLOW first , I write that part of the report first. Therefore, I only mentioned the difference between conditional GLOW and GLOW in this part. Preliminary work and the training process is the same, the only difference is the input of the model and the architecture of the GLOW model.

Conditional GLOW takes both images and labels as input.

```
z, sldj = model(imgs, labels, reverse=False)
```

In the model, it linearly projects the condition vectors into the shape of the input image except the channel which is 1. Then, squeeze it as the images do and pass it to the level module.

```
class Glow(nn.Module):
    def __init__(self, num_channels, num_levels, num_steps):
        super(Glow, self).__init__()
        self.c2c = nn.Linear(40, IMG_SIZE * IMG_SIZE)
        self.register_buffer('bounds', torch.tensor([0.95], dtype=torch.float32))
        self.flows = _Glow(in_channels=4 * 3,
                           cond_channels=4,
                           mid_channels=num_channels,
                           num_levels=num_levels,
                           num_steps=num_steps)
    def forward(self, x, x_cond, reverse=False):
        x_cond = self.c2c(x_cond).view(x.size(0), 1, IMG_SIZE, IMG_SIZE)

        if reverse:
            sldj = torch.zeros(x.size(0), device=x.device)
        else:
            x, sldj = self._pre_process(x)

        x = squeeze(x)
        x_cond = squeeze(x_cond)
        x, sldj = self.flows(x, x_cond, sldj, reverse)
        x = squeeze(x, reverse=True)

        return x, sldj
```

In the level module, the condition vector is passed to the steps or squeezed and passed to the next level.

```
class _Glow(nn.Module):
    def __init__(self, in_channels, cond_channels, mid_channels, num_levels, num_steps):
        super(_Glow, self).__init__()
        self.steps = nn.ModuleList([_FlowStep(in_channels=in_channels,
                                              cond_channels=cond_channels,
                                              mid_channels=mid_channels)
                                    for _ in range(num_steps)])

        if num_levels > 1:
            self.next = _Glow(in_channels=2 * in_channels,
                              cond_channels=4 * cond_channels,
                              mid_channels=mid_channels,
                              num_levels=num_levels - 1,
                              num_steps=num_steps)
        else:
            self.next = None

    def forward(self, x, x_cond, sldj, reverse=False):
        if not reverse:
            for step in self.steps:
                x, sldj = step(x, x_cond, sldj, reverse)

        if self.next is not None:
            x = squeeze(x)
            x_cond = squeeze(x_cond)
            x, x_split = x.chunk(2, dim=1)
            x, sldj = self.next(x, x_cond, sldj, reverse)
            x = torch.cat((x, x_split), dim=1)
            x = squeeze(x, reverse=True)
            x_cond = squeeze(x_cond, reverse=True)

        if reverse:
            for step in reversed(self.steps):
                x, sldj = step(x, x_cond, sldj, reverse)

        return x, sldj
```

There are three components in a step and the only one that uses the condition vector is the Affine Coupling Layer.

```
class _FlowStep(nn.Module):
    def __init__(self, in_channels, cond_channels, mid_channels):
        super(_FlowStep, self).__init__()

        self.norm = ActNorm(in_channels, return_ldj=True)
        self.conv = InvConv(in_channels)
        self.coup = Coupling(in_channels // 2, cond_channels, mid_channels)

    def forward(self, x, x_cond, sldj=None, reverse=False):
        if reverse:
            x, sldj = self.coup(x, x_cond, sldj, reverse)
            x, sldj = self.conv(x, sldj, reverse)
            x, sldj = self.norm(x, sldj, reverse)
        else:
            x, sldj = self.norm(x, sldj, reverse)
            x, sldj = self.conv(x, sldj, reverse)
            x, sldj = self.coup(x, x_cond, sldj, reverse)

        return x, sldj
```

The original NN function consists of three convolution layers followed by normalization. In the conditional version, I do the same thing to condition vectors in order to have the same dimension and add images and condition vectors up. The procedure that happens at each convolution layer can be referred to as residual.

```
def forward(self, x, x_cond):
    x = self.in_norm(x)
    x = self.in_conv(x) + self.in_condconv(x_cond)
    x = F.relu(x)

    x = self.mid_conv1(x) + self.mid_condconv1(x_cond)
    x = self.mid_norm(x)
    x = F.relu(x)

    x = self.mid_conv2(x) + self.mid_condconv2(x_cond)
    x = self.out_norm(x)
    x = F.relu(x)

    x = self.out_conv(x)

    return x
```

GLOW

I reorganize the provided function to get the data wanted in my way. Then, I call the function 'get_CelebA_data' to get training data.

```
train_data = get_CelebA_data()
```

This function reads the image folder to get all the images name and reads 'CelebA-HQ-attribute-anno.txt' to get the corresponding attribute labels for each image. At last, return a list of image names and a list of their labels.

```
def get_CelebA_data():
    img_list = os.listdir('./CelebA-HQ-img')
    label_list = []
    f = open('CelebA-HQ-attribute-anno.txt', 'r')
    num_imgs = int(f.readline()[:-1])
    attrs = f.readline()[:-1].split(' ')
    for idx in range(num_imgs):
        line = f.readline()[:-1].split(' ')
        label = line[2:]
        label = list(map(int, label))
        label_list.append(label)
    f.close()
    return img_list, label_list
```

After getting the data, I call the class 'CelebADataset' to prepare the dataset for training.

```
train_dataset = CelebADataset(train_data)
```

This class of dataset initially splits data into images and labels, then declares a transformer that resizes, permutes and normalizes the image. Function '__getitem__' gets the image name in the list and reads the image.Then, apply transformations to the image. Finally, return the image data and its label of that index.

```
class CelebADataset(Dataset):
    def __init__(self, data):
        self.img_list, self.label_list = data
        self.transform = transforms.Compose([transforms.Resize((IMG_SIZE, IMG_SIZE)),
                                             transforms.ToTensor()])
    def __len__(self):
        return len(self.label_list)

    def __getitem__(self, index):
        image_path = './CelebA-HQ-img/' + self.img_list[index]
        img = Image.open(image_path).convert('RGB')
        img = self.transform(img).numpy()

        return img, self.label_list[index]
```

I use customized 'NLLloss' as loss function, 'Adam' as optimizer.

```
model = Glow(num_channels=NUM_CHANNELS, num_levels=NUM_LEVELS, num_steps=NUM_STEPS)
model.to(device)
loss_fn = NLLLoss()

loss_fn.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=LR)
```

Now, let's go through the training process. Before getting into the training process, I initialize one variable to store the best loss in order to pick a great model. In every epoch, there is a variable of class 'AverageMeter' to calculate the average loss in one epoch which is also referred to as an indicator of model performance. In each batch, there is no need for labels, so the labels become dummies. Input images into GLOW model with option 'reverse' set to false to get latent and log-determinant which afterwards are the inputs to calculate loss. After calculating the loss, update the variable of class 'AverageMeter' to measure average loss. If the loss is smaller than the best loss so far, update the best loss and save the model.

```
best_loss = 1e9

for epoch in tqdm(range(EPOCHS)):
    model.train()
    loss_meter = AverageMeter()
    for x, _ in train_loader:
        x = x.to(device)
        optimizer.zero_grad()
        z, sldj = model(x, reverse=False)
        loss = loss_fn(z, sldj)
        loss_meter.update(loss.item(), x.size(0))
        loss.backward()
        optimizer.step()
    if loss_meter.avg < best_loss:
        torch.save(model.state_dict(), 'test.pt')
        best_loss = loss_meter.avg
```

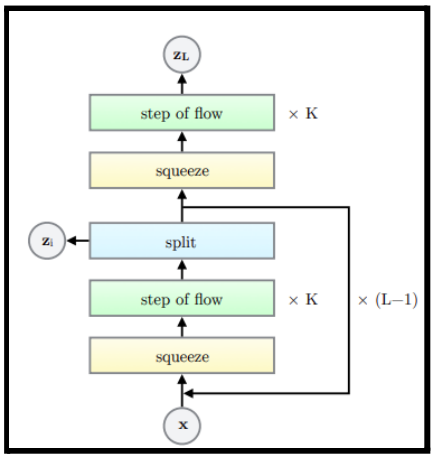
```
class AverageMeter(object):
    def __init__(self):
        self.val = 0.
        self.avg = 0.
        self.sum = 0.
        self.count = 0.

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count
```

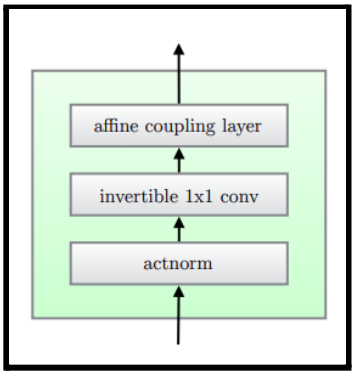
The 'NLLLoss' is to calculate the negative log-likelihood loss which assumes to be isotropic gaussian. This loss function is proposed in the paper **DENSITY ESTIMATION USING REAL NVP**.

```
class NLLLoss(nn.Module):
    def __init__(self, k=256):
        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
```

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

The GLOW model architecture is shown below, which consists of multi-level modules and in each module contains a squeeze layer, multiple steps of flow. In each step, there are three components, which are ctnorm, Invertible 1 × 1 convolution and Affine Coupling Layers.





In the initialization of the GLOW module, use bounds to rescale images before converting to logits and declare the number of channels, levels and steps which is required to construct a GLOW module. If ‘reverse’ is set to true, initialize log-determinant to be all 0. If not, then dequantize the input images and convert them to logits and save log-determinant of Jacobian of initial transform. Then, go through a squeeze layer, multiple levels of flow and finally a squeeze layer with reverse set to true. The squeeze layer aims to downsize the input dimension by adding the number of channels, and the reverse one upsizes the input dimension by reducing the number of channels.

```python
class Glow(nn.Module):
    def __init__(self, num_channels, num_levels, num_steps):
        super(Glow, self).__init__()

        self.register_buffer('bounds', torch.tensor([0.9], dtype=torch.float32))
        self.flows = _Glow(in_channels=4 * 3,
                           mid_channels=num_channels,
                           num_levels=num_levels,
                           num_steps=num_steps)

    def forward(self, x, reverse=False):
        if reverse:
            sldj = torch.zeros(x.size(0), device=x.device)
        else:
            x, sldj = self._pre_process(x)

        x = squeeze(x)
        x, sldj = self.flows(x, sldj, reverse)
        x = squeeze(x, reverse=True)

        return x, sldj

    def _pre_process(self, x):
        y = (x * 255. + torch.rand_like(x)) / 256.
        y = (2 * y - 1) * self.bounds
        y = (y + 1) / 2
        y = y.log() - (1. - y).log()

        ldj = F.softplus(y) + F.softplus(-y) \
            - F.softplus((1. - self.bounds).log() - self.bounds.log())
        sldj = ldj.flatten(1).sum(-1)

        return y, sldj
```

```python
def squeeze(x, reverse=False):
    b, c, h, w = x.size()
    if reverse:
        x = x.view(b, c // 4, 2, 2, h, w)
        x = x.permute(0, 1, 4, 2, 5, 3).contiguous()
        x = x.view(b, c // 4, h * 2, w * 2)
    else:
        x = x.view(b, c, h // 2, 2, w // 2, 2)
        x = x.permute(0, 1, 3, 5, 2, 4).contiguous()
        x = x.view(b, c * 2 * 2, h // 2, w // 2)

    return x
```

In the initialization of the multi-level module, specify the content in one level such as number of the steps of flow and if the level is larger than 1, call the module itself with the number of levels minus 1 to generate a new level. If ‘reverse’ is set to false, then go through the steps like the architecture shown above, otherwise go through the reverse version. If there is a next level, squeeze and split the initial input and use a portion of it as input to the next level. Finally, concatenate the output and a portion of initial input and squeeze it as the final output.

```python
class _Glow(nn.Module):
    def __init__(self, in_channels, mid_channels, num_levels, num_steps):
        super(_Glow, self).__init__()
        self.steps = nn.ModuleList([_FlowStep(in_channels=in_channels,
                                              mid_channels=mid_channels)
                                    for _ in range(num_steps)])

        if num_levels > 1:
            self.next = _Glow(in_channels=2 * in_channels,
                              mid_channels=mid_channels,
                              num_levels=num_levels - 1,
                              num_steps=num_steps)
        else:
            self.next = None

    def forward(self, x, sldj, reverse=False):
        if not reverse:
            for step in self.steps:
                x, sldj = step(x, sldj, reverse)

        if self.next is not None:
            x = squeeze(x)
            x, x_split = x.chunk(2, dim=1)
            x, sldj = self.next(x, sldj, reverse)
            x = torch.cat((x, x_split), dim=1)
            x = squeeze(x, reverse=True)

        if reverse:
            for step in reversed(self.steps):
                x, sldj = step(x, sldj, reverse)

        return x, sldj
```

In each step, there are three components, which are Actnorm, Invertible 1 × 1 convolution and Affine Coupling Layers. As I mentioned above, if ‘reverse’ is set to false, then go through the normal one, otherwise go through the reverse version.

```
class _FlowStep(nn.Module):
    def __init__(self, in_channels, mid_channels):
        super(_FlowStep, self).__init__()

        self.norm = ActNorm(in_channels, return_ldj=True)
        self.conv = InvConv(in_channels)
        self.coup = Coupling(in_channels // 2, mid_channels)

    def forward(self, x, sldj=None, reverse=False):
        if reverse:
            x, sldj = self.coup(x, sldj, reverse)
            x, sldj = self.conv(x, sldj, reverse)
            x, sldj = self.norm(x, sldj, reverse)
        else:
            x, sldj = self.norm(x, sldj, reverse)
            x, sldj = self.conv(x, sldj, reverse)
            x, sldj = self.coup(x, sldj, reverse)

        return x, sldj
```

The ActNorm layer aims to normalize the batch with two trainable weights, mean and sigma. For initialization, use the mean and variance of the first batch as value. if ‘reverse’ is set to false, then normalize the batch with learnable mean and sigma, otherwise denormalize it. It also can use non-trainable weights to normalize and denormalize the batch.

```
class ActNorm(nn.Module):
    def __init__(self, num_features, scale=1., return_ldj=False):
        super(ActNorm, self).__init__()
        self.register_buffer('is_initialized', torch.zeros(1))
        self.bias = nn.Parameter(torch.zeros(1, num_features, 1, 1))
        self.logs = nn.Parameter(torch.zeros(1, num_features, 1, 1))
        self.num_features = num_features
        self.scale = float(scale)
        self.eps = 1e-6
        self.return_ldj = return_ldj
    def initialize_parameters(self, x):
        if not self.training:
            return
        with torch.no_grad():
            bias = -mean_dim(x.clone(), dim=[0, 2, 3], keepdims=True)
            v = mean_dim((x.clone() + bias) ** 2, dim=[0, 2, 3], keepdims=True)
            logs = (self.scale / (v.sqrt() + self.eps)).log()
            self.bias.data.copy_(bias.data)
            self.logs.data.copy_(logs.data)
            self.is_initialized += 1.
    def _center(self, x, reverse=False):
        if reverse:
            return x - self.bias
        else:
            return x + self.bias
    def _scale(self, x, sldj, reverse=False):
        logs = self.logs
        if reverse:
            x = x * logs.mul(-1).exp()
        else:
            x = x * logs.exp()
        if sldj is not None:
            ldj = logs.sum() * x.size(2) * x.size(3)
            if reverse:
                sldj = sldj - ldj
            else:
                sldj = sldj + ldj
        return x, sldj
    def forward(self, x, ldj=None, reverse=False):
        if not self.is_initialized:
            self.initialize_parameters(x)
        if reverse:
            x, ldj = self._scale(x, ldj, reverse)
            x = self._center(x, reverse)
        else:
            x = self._center(x, reverse)
            x, ldj = self._scale(x, ldj, reverse)
        if self.return_ldj:
            return x, ldj
        return x
```

The Invertible 1 × 1 convolution layer aims to generalize the permutation. It first random initializes a matrix and computes the qr factorization of it to get the orthonormal matrix and use it as the kernel of the 1 × 1 convolution layer. Because of the procedure of deriving the orthonormal matrix, the matrix is invertible. If 'reverse' is set to false, then use the orthonormal matrix as the kernel, otherwise the inverse one.

```python
class InvConv(nn.Module):
    def __init__(self, num_channels):
        super(InvConv, self).__init__()
        self.num_channels = num_channels

        w_init = np.random.randn(num_channels, num_channels)
        w_init = np.linalg.qr(w_init)[0].astype(np.float32)
        self.weight = nn.Parameter(torch.from_numpy(w_init))

    def forward(self, x, sldj, reverse=False):
        ldj = torch.slogdet(self.weight)[1] * x.size(2) * x.size(3)

        if reverse:
            weight = torch.inverse(self.weight.double()).float()
            sldj = sldj - ldj
        else:
            weight = self.weight
            sldj = sldj + ldj

        weight = weight.view(self.num_channels, self.num_channels, 1, 1)
        z = F.conv2d(x, weight)

        return z, sldj
```

The Affine Coupling Layer follows the implementation of the paper '[Nice: non-linear independent components estimation](#)'. Its formula is shown below with normal one and reverse one. The difference between the two formulas is at the fourth line. Therefore, specify the function in two different ways for the fourth line. The NN function consists of three convolution layers followed by normalization. Before going through the last convolution layer, the weight and bias should be set to 0 to help train the network.

```python
class Coupling(nn.Module):
    def __init__(self, in_channels, mid_channels):
        super(Coupling, self).__init__()
        self.nn = NN(in_channels, mid_channels, 2 * in_channels)
        self.scale = nn.Parameter(torch.ones(in_channels, 1, 1))
    def forward(self, x, ldj, reverse=False):
        x_change, x_id = x.chunk(2, dim=1)
        st = self.nn(x_id)
        s, t = st[:, 0::2, ...], st[:, 1::2, ...]
        s = self.scale * torch.tanh(s)
        if reverse:
            x_change = x_change * s.mul(-1).exp() - t
            ldj = ldj - s.flatten(1).sum(-1)
        else:
            x_change = (x_change + t) * s.exp()
            ldj = ldj + s.flatten(1).sum(-1)
        x = torch.cat((x_change, x_id), dim=1)
        return x, ldj
class NN(nn.Module):
    def __init__(self, in_channels, mid_channels, out_channels,
                 use_act_norm=False):
        super(NN, self).__init__()
        norm_fn = ActNorm if use_act_norm else nn.BatchNorm2d

        self.in_norm = norm_fn(in_channels)
        self.in_conv = nn.Conv2d(in_channels, mid_channels,
                                 kernel_size=3, padding=1, bias=False)
        nn.init.normal_(self.in_conv.weight, 0., 0.05)

        self.mid_norm = norm_fn(mid_channels)
        self.mid_conv = nn.Conv2d(mid_channels, mid_channels,
                                  kernel_size=1, padding=0, bias=False)
        nn.init.normal_(self.mid_conv.weight, 0., 0.05)

        self.out_norm = norm_fn(mid_channels)
        self.out_conv = nn.Conv2d(mid_channels, out_channels,
                                  kernel_size=3, padding=1, bias=True)
        nn.init.zeros_(self.out_conv.weight)
        nn.init.zeros_(self.out_conv.bias)
    def forward(self, x):
        x = self.in_norm(x)
        x = F.relu(x)
        x = self.in_conv(x)
        x = self.mid_norm(x)
        x = F.relu(x)
        x = self.mid_conv(x)
        x = self.out_norm(x)
        x = F.relu(x)
        x = self.out_conv(x)
        return x
```

$\mathbf{x}_a, \mathbf{x}_b = \text{split}(\mathbf{x})$	$\mathbf{y}_a, \mathbf{y}_b = \text{split}(\mathbf{y})$
$(\log \mathbf{s}, \mathbf{t}) = \text{NN}(\mathbf{x}_b)$	$(\log \mathbf{s}, \mathbf{t}) = \text{NN}(\mathbf{y}_b)$
$\mathbf{s} = \exp(\log \mathbf{s})$	$\mathbf{s} = \exp(\log \mathbf{s})$
$\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$	$\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$
$\mathbf{y}_b = \mathbf{x}_b$	$\mathbf{x}_b = \mathbf{y}_b$
$\mathbf{y} = \text{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{x} = \text{concat}(\mathbf{x}_a, \mathbf{x}_b)$

- **Specify the hyperparameters**
  - Task 1

    Conditional Self-Attention GAN

    1. **epochs = 1000**
    2. **batch size = 64**
    3. **learning rate for generator = 1e-4**
    4. **learning rate for discriminator = 4e-4**
    5. **beta 1 for both adam optimizer = 0**
    6. **beta 2 for both adam optimizer = 0.9**
    7. **latent dimension = 100**
    8. **image size = 64 * 64**
    9. **image channel = 3**

    Conditional GLOW

    1. **epochs = 1000**
    2. **batch size = 64**
    3. **learning rate = 1e-3**
    4. **number of hidden channels = 256**
    5. **number of levels = 2**
    6. **number of steps = 3**
  - Task 2

    Conditional GLOW

    1. **epochs = 500**
    2. **batch size = 64**
    3. **learning rate = 1e-3**
    4. **number of hidden channels = 128**
    5. **number of levels = 3**
    6. **number of steps = 8**

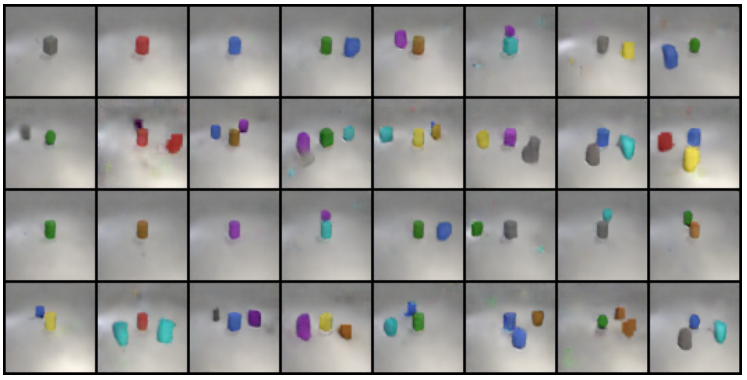
    GLOW

    1. **epochs = 200**
    2. **batch size = 32**
    3. **learning rate = 1e-3**
    4. **number of hidden channels = 128**
    5. **number of levels = 3**
    6. **number of steps = 8**
    7. **image size = 64 * 64**

- **Results and discussion**
  - **Task 1**
    - Show your results based on the testing data

Conditional Self-Attention GAN

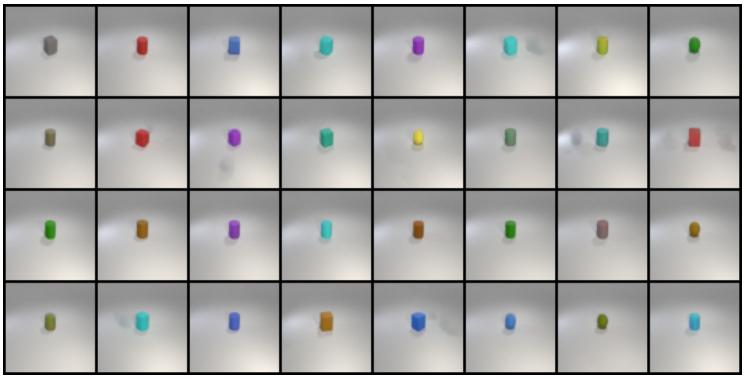
test.json



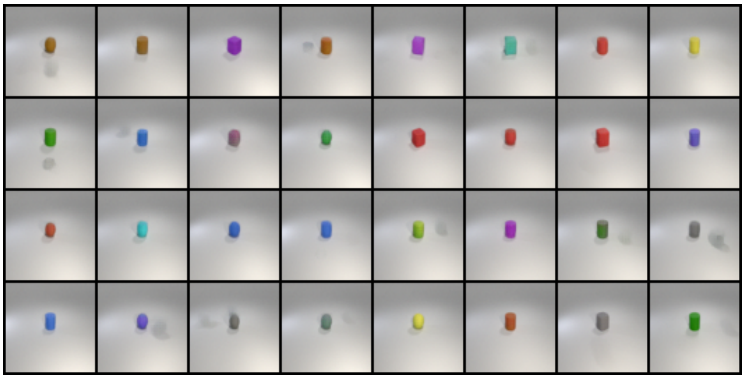
new_test.json

Conditional GLOW

test.json



new_test.json



■ Classification accuracy on test.json and new test.json using cGAN and cNF

Conditional Self-Attention GAN

test.json

Accuracy: 0.7083333333333334

```
(DLP_final) kai@ED716-ESC4000-G4:~/hdd0/Lab7/task_1/best$ CUDA_VISIBLE_DEVICES=1 python3 Lab7_task1_GAN_inference.py
0.7083333333333334
```

new_test.json

Accuracy: 0.7023809523809523

```
(DLP_final) kai@ED716-ESC4000-G4:~/hdd0/Lab7/task_1/best$ CUDA_VISIBLE_DEVICES=1 python3 Lab7_task1_GAN_inference.py
0.7023809523809523
```

Conditional GLOW

test.json

Accuracy :0.4305555555555556

```
(DLP_final) kai@ED716-ESC4000-G4:~/hdd0/Lab7/task_1$ CUDA_VISIBLE_DEVICES=1 python3 nf_inference.py
0.4305555555555556
```

new_test.json

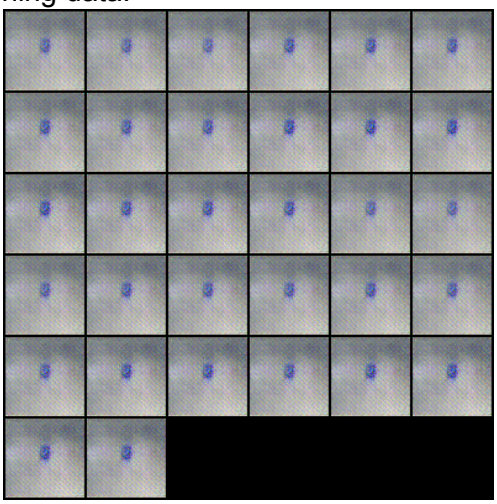
Accuracy: 0.42857142857142855

```
(DLP_final) kai@ED716-ESC4000-G4:~/hdd0/Lab7/task_1$ CUDA_VISIBLE_DEVICES=1 python3 nf_inference.py
0.42857142857142855
```

■ Discuss the results of different models architectures

When training conditional GAN, I tried Auxiliary Classifier GAN first, which is also known as AC GAN. This architecture not only asks the discriminator to classify the real image and fake image, but also the class of the input images. However, the performance is always around 0.3 no matter what I change the procedure. I think the reason is that the multilabel problem is a little bit hard for my model, so the performance is not so good. Therefore, I switched my model architecture to cGAN.

I also try to use generated labels when generating the images. It turns out that the generated images look alike during the training process. Therefore, I use the labels in training data.



During the beginning of the training process, I found out that the loss of discriminator drops quickly, so the generator could not learn well. Hence, I train the generator twice in one batch.

When training conditional NF, I found out that the generated images are quite good but the images only contained one object. Therefore, I try to add one loss function to let the model predict the number of objects in the images. It turns out that the results are getting worse. At the end, the accuracy can only achieve 0.4.

○ Task 2

■ Conditional face generation

I load the pretrained model of conditional GLOW. Then, generate gaussian noises and attribute conditions as the input to conditional GLOW with reverse set to true to generate human face.

```
model = Glow(num_channels=NUM_CHANNELS, num_levels=NUM_LEVELS, num_steps=NUM_STEPS)
model.load_state_dict(torch.load("./task2_1.pt"))
model.to(device)
model.eval()

z = torch.randn((64, 3, 64, 64), dtype=torch.float32, device=device)
c = torch.full((64, 40), -1, device=device).float()
for i in range(64):
    c[i][31]= 1.
    c[i][9]= 1.
    c[i][39]= 1.
    c[i][20]= 1.

x, _ = model(z, c, reverse=True)
images = torch.sigmoid(x)
images_concat = torchvision.utils.make_grid(images, nrow=int(64 ** 0.5), padding=2, pad_value=255)
torchvision.utils.save_image(images_concat, './output/conditional.png')
```

Blond, Smiling, Young

Blond, Smiling, Young

Man, Black Hair, Young

Man, Blond Hair, Young, Smiling

■ Linear interpolation

I load the pretrained model of GLOW. Then, read two images and apply transformation to them. After preprocessing, I use the GLOW model to generate latent vectors for both images and calculate the difference between them. Finally, I generate 6 images between two images by adding portions of difference to the original image and use the reverse version of GLOW to generate them.

```
model = Glow(num_channels=NUM_CHANNELS, num_levels=NUM_LEVELS, num_steps=NUM_STEPS)
model.load_state_dict(torch.load("./task2_2.pt"))
model.to(device)
model.eval()

transform = transforms.Compose([transforms.Resize((IMG_SIZE, IMG_SIZE)),
                                transforms.ToTensor()])

a_img_name = '60.jpg'
b_img_name = '18.jpg'

image_path = './CelebA-HQ-img/' + a_img_name
a_img = Image.open(image_path).convert('RGB')
a_img = transform(a_img)
numpy_a_img = a_img.numpy()

image_path = './CelebA-HQ-img/' + b_img_name
b_img = Image.open(image_path).convert('RGB')
b_img = transform(b_img)
numpy_b_img = b_img.numpy()

a_img = a_img.to(device).unsqueeze(0)
a_z, _ = model(a_img, reverse=False)

b_img = b_img.to(device).unsqueeze(0)
b_z, _ = model(b_img, reverse=False)

difference = (b_z.detach().cpu().numpy() - a_z.detach().cpu().numpy()).squeeze(0)

all_img = np.full((8, 3, 64, 64), a_z.cpu().detach().numpy())
for i in range(8):
    all_img[i] += difference / 8 * i
all_img = torch.from_numpy(all_img).to(device)
x, _ = model(all_img, reverse=True)
images = torch.sigmoid(x)
images_concat = torchvision.utils.make_grid(images, nrow=8, padding=2, pad_value=255)
torchvision.utils.save_image(images_concat, './output/interpolation.png')
```

■ Attribute manipulation

I construct a new dataset called ‘AttributeDataset’ to get the images that the specified attribute is positive or negative and call DataLoader to automatically get a batch of data.

```
attribute_pos_dataset = AttributeDataset(train_data, -9, 1)
attribute_pos_loader = DataLoader(attribute_pos_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=12)

attribute_neg_dataset = AttributeDataset(train_data, -9, -1)
attribute_neg_loader = DataLoader(attribute_neg_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=12)
```

When calling this dataset, should specify the attribute and whether it is positive or negative. Then, find out all the images satisfied with the requirement and apply transformation to them.

```
class AttributeDataset(Dataset):
    def __init__(self, data, cond, pos_neg):
        img_list, label_list = data
        self.transform = transforms.Compose([transforms.Resize((IMG_SIZE, IMG_SIZE)),
                                             transforms.ToTensor()])
        self.img_list = []
        for i in range(len(label_list)):
            if label_list[i][cond] == pos_neg:
                self.img_list.append(img_list[i])

    def __len__(self):
        return len(self.img_list)

    def __getitem__(self, index):
        image_path = './CelebA-HQ-img/' + self.img_list[index]
        img = Image.open(image_path).convert('RGB')
        img = self.transform(img).numpy()

        return img
```

After getting positive and negative images, I load the pretrained model of GLOW.

```
model = Glow(num_channels=NUM_CHANNELS, num_levels=NUM_LEVELS, num_steps=NUM_STEPS)
model.load_state_dict(torch.load("./task2_2.pt"))
model.to(device)
model.eval()
```

I use the GLOW model to generate latent vectors for both positive images and negative images and calculate mean vectors for both positive and negative to get the difference between them.

```
pos = np.zeros((3, 64, 64))

for img in attribute_pos_loader:
    img = img.to(device)
    z, _ = model(img, reverse=False)
    pos += torch.sum(z, 0).cpu().detach().numpy()

neg = np.zeros((3, 64, 64))

for img in attribute_neg_loader:
    img = img.to(device)
    z, _ = model(img, reverse=False)
    neg += torch.sum(z, 0).cpu().detach().numpy()

pos = pos / len(attribute_pos_dataset)
neg = neg / len(attribute_neg_dataset)

dif = (pos - neg)
```

I read one image and apply transformation to it. After preprocessing, I use the GLOW model to generate a latent vector of it. Finally, I generate 6 images between positive and negative images by adding portions of difference to the original image and use the reverse version of GLOW to generate them.

```
img_name = '900.jpg'

transform = transforms.Compose([transforms.Resize((IMG_SIZE, IMG_SIZE)),
                                transforms.ToTensor()])
image_path = './CelebA-HQ-img/' + img_name
img = Image.open(image_path).convert('RGB')
img = transform(img)
numpy_img = img.numpy()

img = img.to(device).unsqueeze(0)
z, _ = model(img, reverse=False)

all_img = np.full((8, 3, 64, 64), z.cpu().detach().numpy())

for i in range(8):
    all_img[i] += dif / 8 * i

all_img = torch.from_numpy(all_img).to(device)
x, _ = model(all_img, reverse=True)
images = torch.sigmoid(x)
images_concat = torchvision.utils.make_grid(images, nrow=8, padding=2, pad_value=255)
torchvision.utils.save_image(images_concat, './output/attribute.png')
```

image: 900.jpg

Smile:

Bags Under Eyes:

image: 10959.jpg

Smile:

Mouth Slightly Open: