DLP Lab7

電控碩 0860077 王國倫

1. Introduction

In this Lab, we need to implement generative adversarial network (GAN) and normalizing flow (NF) network for two tasks.

For the first task, we need use GAN and NF with its condition to generate synthetic image, the condition is multi-label. For example, "red cube" and "blue cylinder". The network must generate these objects in synthetic image, then use TA pre-trained classifier for evaluation, the overall procedure shown the Fig.1.

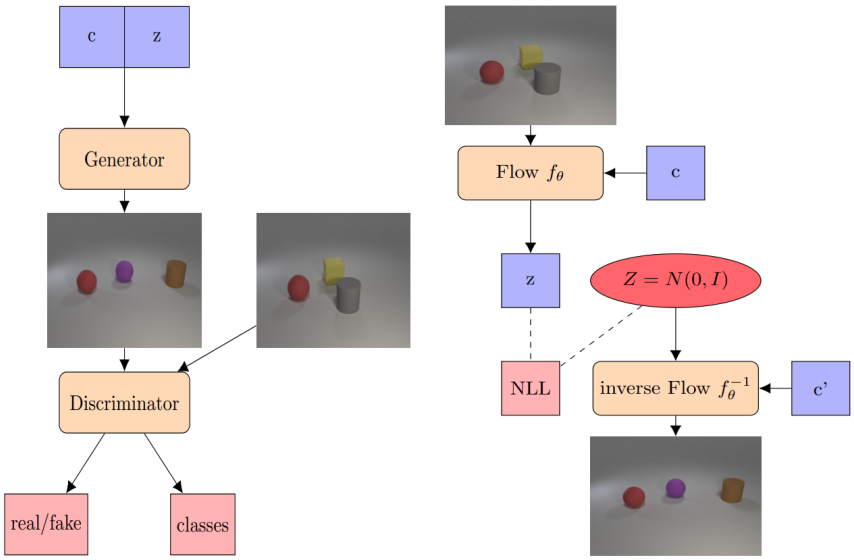

Fig1. The overall procedure for using cGAN and cNF

The second task, we need to generate human face, and only use NF to achieve this task, we can decide by ourselves whether to use condition. The task required three applications, including face generation, linear interpolation and attribute manipulation.

- The face generation given some condition and latent vector to generate human face, but the latent vector not use mapped from input image, and the condition must design by ourselves.
- The linear interpolation pick two images, then interpolate at least five images according to their latent vector, the example shown Fig.2.

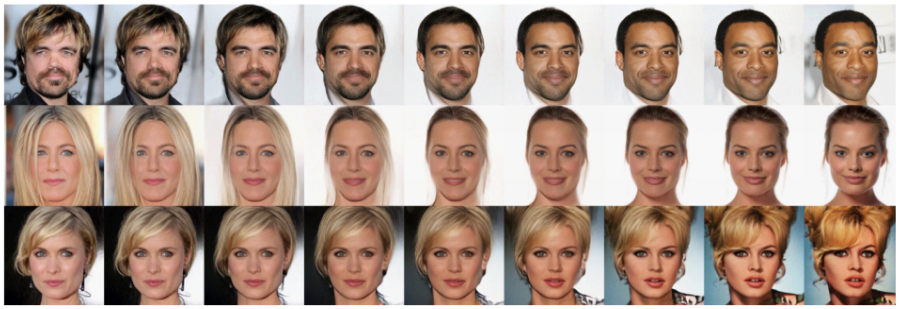

Fig.2 The linear interpolation [1]

- The attribute manipulation select an attribute, and calculate mean vector of images with this attribute and without. Then choose one image and use difference between two mean vector with different scaling to change the image attribute, see Fig.3.

Fig.3 The attribute manipulation for smiling [1]

2. Implementation details

A. cGAN

In this Lab, my GAN architecture based on DCGAN and add auxiliary classifier (Auxiliary GAN [2]) to achieve more higher score for evaluation, the difference between cGAN and ACGAN shown Fig.4.

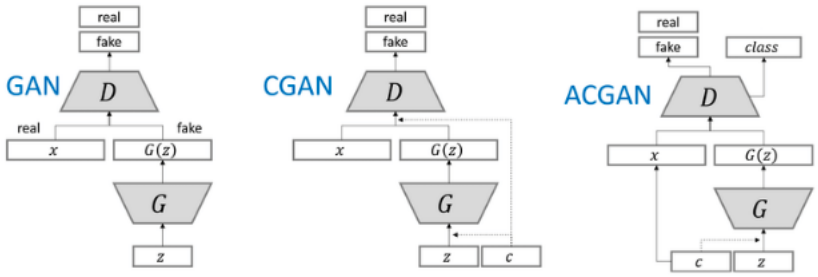

Fig.4 GAN, CGAN and ACGAN architectures, where x denotes the real image, c the class label, z the noise vector, G the Generator, and D the Discriminator.

In my architecture, I combined CGAN and ACGAN, let CGAN add auxiliary classifier to improve performance, the generator and discriminator code show below.

For the generator, I use 200 as class latent size, and use fully connection layer and ReLU to transform, then concatenate latent vector to generate synthetic image, and use ConvTranspose2d to deconvolution to origin resolution (3*64*64), also use weight initial to help GAN training well.

For the discriminator, I transform condition to 1*64*64 and concatenate origin image 3*64*64, then fed into discriminator to distinguish real or fake and its classes, and also use weight initial to help GAN training well.

```
#!/usr/bin/env python3

import torch.nn as nn
import torch
import torch.nn.functional as F
import numpy as np


class Generator(nn.Module):
    def __init__(self, hidden_size, n_class):
        super(Generator, self).__init__()

        self.embedding =  nn.Sequential(
            nn.Linear(n_class, 200),
            nn.ReLU(),
        )

        self.net = nn.Sequential(
            nn.ConvTranspose2d(200 + hidden_size, 512, kernel_size=4, stride=2, \
            padding=0),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            *self.make_block(512, 256),
            *self.make_block(256, 128),
            *self.make_block(128, 64),
            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1),
            nn.Tanh()
        )

    def forward(self, z, c):
        # z -> batch_size * hidden_size
        # c -> batch_size * n_class
        _, hidden_size = z.shape
        _, n_class = c.shape

        # transfer z -> batch_size * hidden_size * 1 * 1
        # transfer c -> batch_size * n_class * 1 * 1
        z = z.view(-1, hidden_size, 1, 1)
        c = self.embedding(c).view(-1, 200, 1, 1)

        # concatenation z and c
        x = torch.cat((z, c), dim=1)

        # output -> batch_size * 3 * 64 * 64
        output = self.net(x)
        return output

    def make_block(self, input, output):
        block = [nn.ConvTranspose2d(input, output, kernel_size=4, stride=2, \
        padding=1)]
        block.append(nn.BatchNorm2d(output))
        block.append(nn.ReLU())
        return block

    def weight_init(self, mean, std):
        for m in self._modules:
            if isinstance(self._modules[m], nn.ConvTranspose2d) or \
                isinstance(self._modules[m], nn.Conv2d):
                self._modules[m].weight.data.normal_(mean, std)
                self._modules[m].bias.data.zero_()


class Discriminator(nn.Module):
    def __init__(self, n_class, img_size):
        super(Discriminator, self).__init__()

        self.embedding = nn.Sequential(
            nn.Linear(n_class, int(np.prod(img_size) / 3)),
            nn.LeakyReLU()
        )

        self.net = nn.Sequential(
            *self.make_block(4, 64),
            *self.make_block(64,128),
```

```
            *self.make_block(128,256),
            *self.make_block(256,512)
        )

        self.dis = nn.Sequential(
            nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0),
            nn.Sigmoid()
        )

        self.aux = nn.Sequential(
            nn.Linear(512*4*4, n_class),
            nn.Sigmoid()
        )

    def forward(self, x, c):
        # x -> batch_size * 3 * 64 * 64
        # c -> batch_size * n_class
        _, ch, w, h = x.shape

        # transfer c -> batch_size * 1 * 64 * 64
        c = self.embedding(c).view(-1, 1, w, h)

        # concatenation x and c
        x = torch.cat((x, c), dim=1)

        # output -> batch_size * 512 * 1 * 1
        output = self.net(x)

        classes = self.aux(output.view(output.shape[0], -1))
        realfake = self.dis(output).view(-1)

        return classes, realfake

    def make_block(self, input, output):
        block = [nn.Conv2d(input, output, kernel_size=4, stride=2, padding=1)]
        block.append(nn.BatchNorm2d(output))
        block.append(nn.LeakyReLU())
        return block

    def weight_init(self, mean, std):
        for m in self._modules:
            if isinstance(self._modules[m], nn.ConvTranspose2d) or \
                isinstance(self._modules[m], nn.Conv2d):
                self._modules[m].weight.data.normal_(mean, std)
                self._modules[m].bias.data.zero_()
```

The cGAN hyper-parameters

In the training, I train ratio is 1:5 means train discriminator ones and train generator five times for each mini-batch, and the discriminator loss weight is 1:3 means real and fake loss is 1, and the classes loss is 3.

- Learning rate : 0.0002
- Hidden size : 200
- Epochs : 200
- Batch size : 32
- Train ratio (D:G) : 1:5
- Discriminator loss weight (R/F:classes) : 1:3

B. cNF

In this Lab, I select Glow [1] as my cNF to handle task1 and task2, the Glow refer this Github, this Github implement a cNF with Glow architecture and use CelebA dataset to train and inference.

Regrading Glow code, I almost use its Github provided code and modify condition process, so I will describe I modify part.

In the glow.py, I add self.embedding and transform condition to 1*64*64 in forward, then fed into glow to generate latent vector and log-determinant, calculate NLLloss with these, and update network.

Because of the classes is different for task1 and task2, so I add augment n_class let me can easy to use it. Fixed the mode is sketch, because task1 and task2 dataset are RGB images.

```
class Glow(nn.Module):

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

        self.embedding = nn.Sequential(
            nn.Linear(n_class, int(np.prod(img_size) / 3)),
            nn.LeakyReLU()
        )

        # 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=4,
                           mid_channels=num_channels,
                           num_levels=num_levels,
                           num_steps=num_steps)
        self.mode = mode

    def forward(self, x, x_cond, reverse=False):
        # x -> batch_size * 3 * 64 * 64
        # c -> batch_size * n_class
        _, _, w, h = x.shape

        # transfer c -> batch_size * 1 * 64 * 64
        x_cond = self.embedding(x_cond).view(-1, 1, w, h)

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

        return x, sldj
```

The cNF hyper-parameters for task 1

- Learning rate : 0.0002
- Epochs : 250
- Channel : 512
- Level : 4
- Step : 6
- Batch size : 16

The cNF hyper-parameters for task2

- Learning rate : 0.0002

- Epochs : 50
- Channel : 512
- Level : 4
- Step : 6
- Batch size : 16

3. Results and discussion

A. Task1

Generated result

- cGAN

  The generated result for test.json and new_test.json will show Fig.5 and Fig.6, respectively.

Fig.5 The generated result for test.json

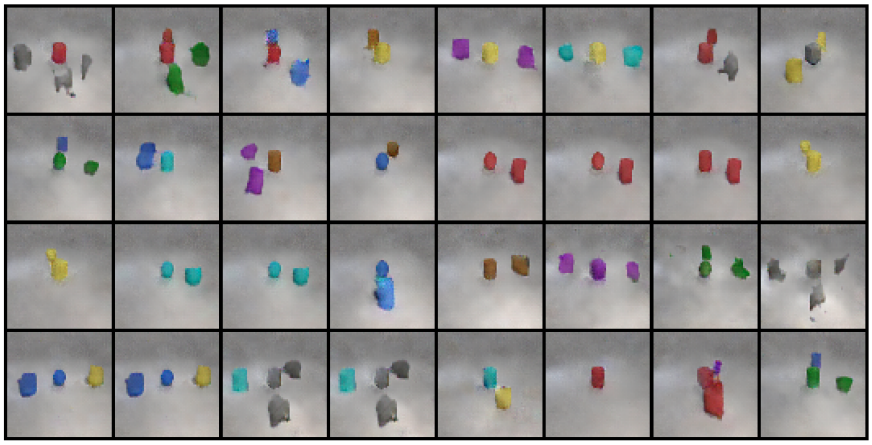

Fig.6 The generated result for new_test.json

- cNF

The generated result for test.json and new_test.json will show Fig.7 and Fig.8, respectively.

Fig.7 The generated result for test.json

Fig.8 The generated result for new_test.json

Score for test.json and new_test.json

Score

Network	dataset	Column
cGAN	test.json	0.70
cGAN	new_test.json	0.71
cNF	test.json	0.45
cNF	new_test.json	0.47

The screen shot for result, see Fig.9 ~ Fig.12

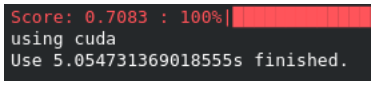

Fig.9 cGAN score for test.json

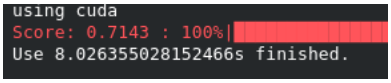

Fig.10 cGAN score for new_test.json

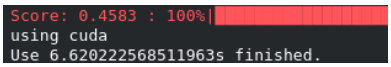

Fig.11 cNF score for test.json

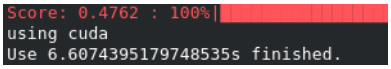

Fig.12 cNF score for new_test.json

Discuss the results of different models architectures

The cGAN training loss and score show Fig.13, and the cNF training loss and score show Fig.14. In this Lab, I use the basic cDCGAN and combined DCGAN and ACGAN to train and try to find the highest score. The Fig.13 point out the combined DCGAN and ACGAN can let score get 0.6~0.7, on the contrary, the cNF score is slowly increase, and its score roughly 0.4. According to the score result, I think if the epochs more high can get more high score, but limit to hardware and time, the 250 epochs need training 31 hours, and I use 2060 6G GPU to training, this situation I think I already do best job for this Lab.

We can see the generated image, the cNF quality of generated images better than cGAN, but the score is opposite, in my opinion, cNF need more time to learn, hardware maybe can improve time problem or try to different hyper-parameter, because of time factor, I can't try to other parameter to compare, so I only show the best result on this.

Fig.13 cGAN result

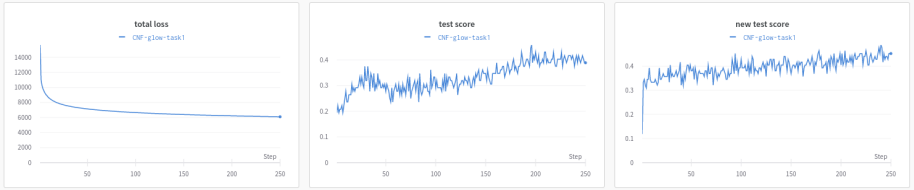

Fig.14 cNF result

B. Task2

Conditional face generation

In the face generation, I create condition with randomly generate 1 or -1, and only 20% is 1, the other is -1, because I see the condition ratio roughly 0.2~0.3. After decided condition,

the latent vector z also random from noise, then fed into glow network to generated synthetic human face, the result show Fig.15.

```
def face_generation(self):

  self.glow.eval()

  z = torch.randn(5, 3, 64, 64).cuda() if self.args.cuda \
      else torch.randn(5, 3, 64, 64)

  # calculate condition
  for i in range(5):
      cond = torch.tensor([1 if random.random() < 0.2 else -1 for _ in range(40)]).view(1, -1)
      label = cond if i == 0 else torch.cat((label, cond), dim=0)

  label = Variable(label)

  # using cuda
  if self.args.cuda:
      label = label.cuda()

  with torch.no_grad():

      generated_img, _ = self.glow(z, label.float(), reverse=True)

      generated_img = torch.tanh(generated_img)

  grid = make_grid(generated_img, nrow=8, normalize=True)
  save_image(grid, format="png", fp=os.path.join(self.img_path, self.file_name + "_face_generation_re
  wandb.log({"generated picture for face": wandb.Image(os.path.join(self.img_path, self.file_name + "
```

Fig.15 Face generation result

Linear interpolation

In the linear interpolation, I randomly sample data, and use four images to generate latent vector, then calculate the difference, add origin latent vector with different scaling, fed into glow to generated image, the result see Fig.16.

```
def interpolation(self):

    # sample from training dataset
    sample = next(iter(self.trainloader))
    img, con = sample

    img, con = Variable(img), Variable(con)

    # using cuda
    if self.args.cuda:
        img = img.cuda()
        con = con.cuda()

    with torch.no_grad():

        z, _ = self.glow(img[0:4], con[0:4].float(), reverse=False)

        z_a, z_b = z[0:3], z[1:4]
        z_diff = z_b - z_a

        for i in range(8):
            alpha = (1.0 / 7) * i
            z_c = z_a + alpha * z_diff
```

```
            if i == 0:
                generated_img, _ = self.glow(z_c, con[1:4].float(), reverse=True)
            else:
                g, _ = self.glow(z_c, con[1:4].float(), reverse=True)
                generated_img = torch.cat((generated_img, g), dim=0)

        t = generated_img.view(8, 3, 3, 64, 64)
        generated_img = t.permute(1, 0, 2, 3, 4)
        generated_img = torch.tanh(generated_img.reshape(24, 3, 64, 64))

    grid = make_grid(generated_img, nrow=8, normalize=True)
    save_image(grid, format="png", fp=os.path.join(self.img_path, self.file_name + "_interpolation_res
    wandb.log({"generated picture for interpolation": wandb.Image(os.path.join(self.img_path, self.fi
```

Fig.16 Linear interpolation result

Attribute manipulation

In the attribute manipulation, I also randomly sample data, and divided into two parts, with this attribute and without attribute, then calculate its mean and difference, pick the first image to adjust its attribute, if its with attribute, then minus difference with scaling, let generated images without attribute and vice versa. I choose smiling and chubby as my control attribute, the result show Fig.17 and Fig.18, respectively.

```
def manipulation(self):

    # sample from training dataset
    sample = next(iter(self.trainloader))
    img, con = sample

    img, con = Variable(img), Variable(con)

    # using cuda
    if self.args.cuda:
        img = img.cuda()
        con = con.cuda()

    # select attribute (smiling)
    # att = random.randint(0,40)
    att = 31

    with torch.no_grad():

        z, _ = self.glow(img, con.float(), reverse=False)

        a_with = con[1:,att] == 1
        a_without = con[1:,att] == -1
        index_w, index_wo = torch.nonzero(a_with).view(-1), torch.nonzero(a_without).view(-1)

        z_w, z_wo = z[index_w], z[index_wo]
        z_diff = z_w.mean(0) - z_wo.mean(0)

        for i in range(5):

            alpha = (1.0 / 4) * i
            if con[0, att] == 1:
                z_c = z[0] - alpha * z_diff
```

```
            else:
                z_c = z[0] + alpha * z_diff
            z_c = z_c.view(1, 3, 64, 64)
            if i == 0:
                generated_img, _ = self.glow(z_c, con[0].float(), reverse=True)
            else:
                g, _ = self.glow(z_c, con[0].float(), reverse=True)
                generated_img = torch.cat((generated_img, g), dim=0)

        generated_img = torch.tanh(generated_img)

    grid = make_grid(generated_img, nrow=8, normalize=True)
    save_image(grid, format="png", fp=os.path.join(self.img_path, self.file_name + "_mainpulation_resu
    wandb.log({"generated picture for mainpulation": wandb.Image(os.path.join(self.img_path, self.file
```

Fig.17 The smiling result (no smile → smile)

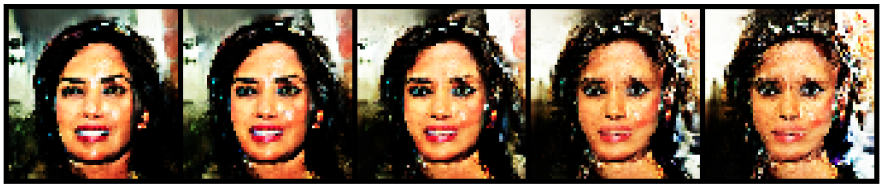

Fig.18 The chubby result (chubby → no chubby)

4. Reference

1. Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1×1 convolutions, 2018.
2. Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans, 2016.