DLP HW5

310551053 Chieh-Ming Jiang

August 19th 2021

1 Introduction

In this assignment, I am going to implement conditional GAN to generate images. GAN is composed of discriminator and generator. Generator is the most important part in GAN, which is used to generate images, while discriminator acts as assistant to help the training of generator.
The training data is composed of 24 objects with different colors and shapes.

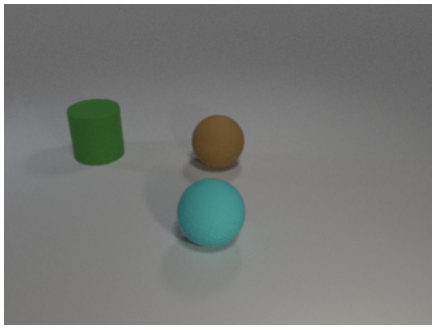

Figure 1: CLEVR Data

2 Implementation details

1. Data preprocessing
   I loaded the images and do some transformations on them, and save these images as torch files.

```python
with open('objects.json', 'r') as f:
    objects = json.load(f)
with open('train.json', 'r') as f:
    labels = json.load(f)
trans = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.5, 0.5, 0.5],
        std=[0.5, 0.5, 0.5]
    )
])
label_list = []
img_list = []
num_images = 6003
for idx in tqdm(range(num_images)):
    for sub_idx in range(3):
        key = f'CLEVR_train_{idx:06}_{sub_idx}.png'
        label_list.append(torch.LongTensor([objects[labels[key][-1]]]))
        img_list.append(trans(PIL.Image.open(f'images/{key}').convert("RGB")))
label_list = torch.stack(label_list)
img_list = torch.stack(img_list)
torch.save(label_list, 'labels.pth')
torch.save(img_list, 'images.pth')
```

Figure 2: Data preprocessing

2. dataloader
   Load the data from the preprocessed torch file.

```python
class CLEVRDataset(Dataset):
    def __init__(self,img_path,json_path):
        self.labels = torch.load('labels.pth')
        self.images = torch.load('images.pth')

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

    def __getitem__(self, index):
        return self.images[index], self.labels[index]
```

Figure 3: Dataloader

3. cDCGAN

I chose cDCGAN as my GAN architecture. cDCGAN combines the advantage of DCGAN and cGAN and perform better.

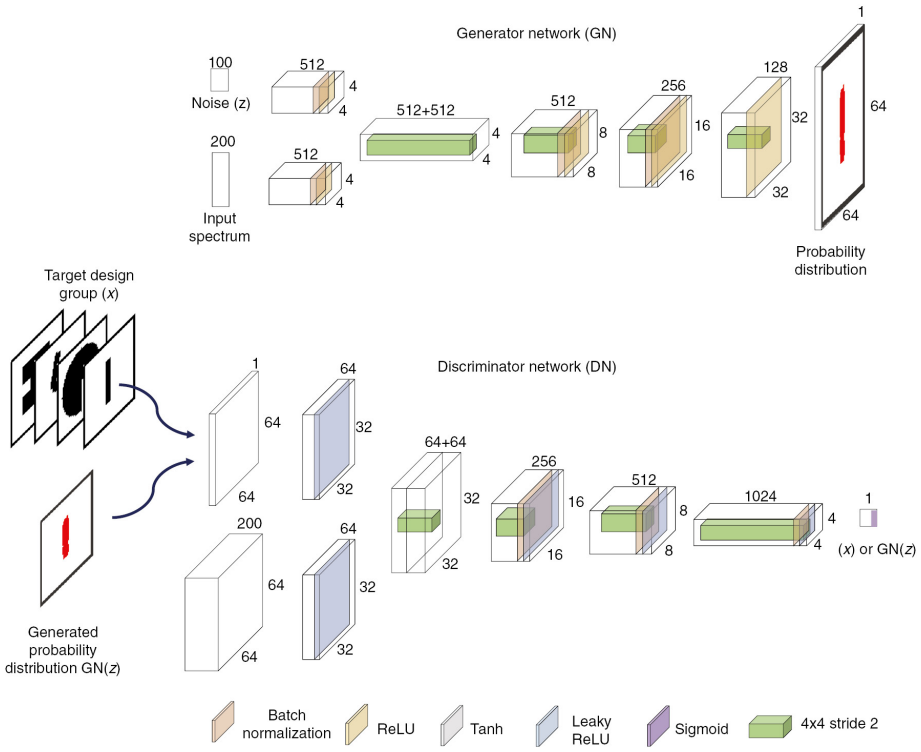

Figure 4: cDCGAN

4. Discriminator
   In discriminator, I first expand the dimension of condition and concatenate it with the images. Then the features will proceed with a series of Conv2d, normalization and leaky relu . Finally, output the value with sigmoid. The discriminator acts like a tiny ResNet.

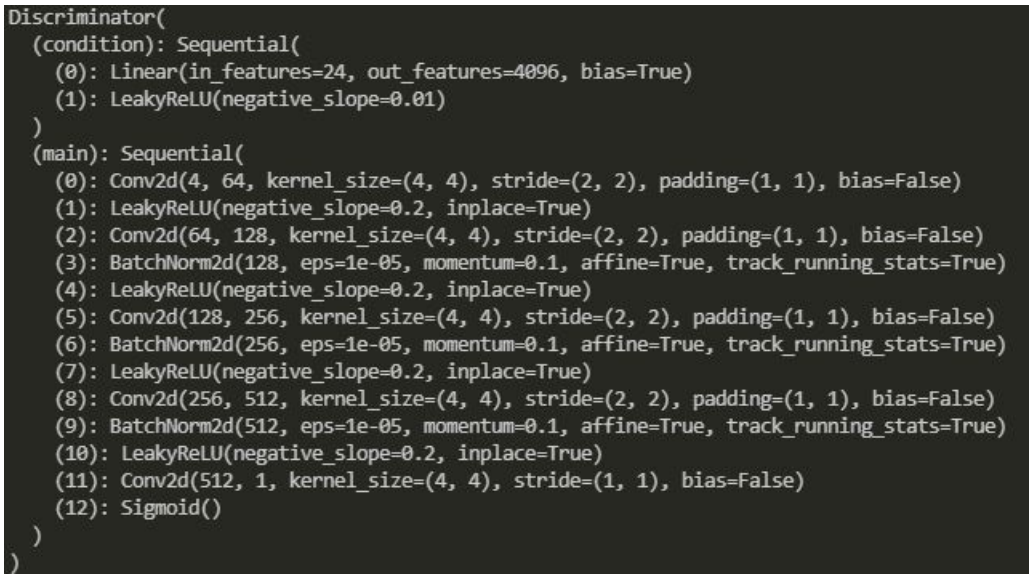

Figure 5: Discriminator

```python
class Discriminator(nn.Module):
    def __init__(self, img_shape, condition_size, ndf = 64, nc = 4):
        super(Discriminator, self).__init__()
        self.H,self.W,self.C = img_shape
        self.condition = nn.Sequential(
            nn.Linear(24,self.H*self.W*1),
            nn.LeakyReLU()
        )
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x , c):
        c = self.condition(c).view(-1,1,self.H,self.W)
        out = torch.cat((x,c),dim = 1)
        out = self.main(out)
        out = out.view(-1)
        return out
```

Figure 6: Discriminator

5. Generator
   In generator, I expand the condition as what I done in discriminator. Then sample a batch of random latent and concatenate with the conditions. In contrast to discriminator, the features proceed with ConvTranspose2d, normalization and relu. The discriminator uses Conv2d to get the features, while the generator uses ConvTranspose2d to generate images from the

features.

```
Generator(
  (condition): Sequential(
    (0): Linear(in_features=24, out_features=200, bias=True)
    (1): ReLU()
  )
  (main): Sequential(
    (0): ConvTranspose2d(300, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Figure 7: Generator

```python
class Generator(nn.Module):
    def __init__(self, latent_size, condition_size , ngf = 64, nc = 3):
        super(Generator,self).__init__()
        self.latent_size = latent_size
        self.condition_size = condition_size
        self.condition = nn.Sequential(
            nn.Linear(24, condition_size),
            nn.ReLU()
        )
        self.main = nn.Sequential(
            nn.ConvTranspose2d( latent_size + condition_size, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )
    def forward(self,z,c):
        z = z.view(-1, self.latent_size, 1, 1)
        c = self.condition(c).view(-1, self.condition_size, 1 , 1)
        out = torch.cat((z,c),dim = 1)
        out = self.main(out)
        return out
```

Figure 8: Generator

6. Loss function
   Because the discriminator is used to distinct fake and real images. So I choose nn.BCELoss() as my loss function, which calculates the binary cross entropy loss between output and target.

7. hyperparameters

   - latent size : 100
   - condition size : 200
   - epochs : 100
   - learning rate : 0.0002
   - batch size : 64

3 Results and discussion

3.1 Results

1. Generated images

Figure 9: Generated images

2. Accuracy

```
Testing Score:  0.7083333333333334
```

Figure 10: Accuracy

3.2 Discussion

1. Balance between discriminator and generator
   In each iteration, I trained the genertor four times, while trained the discriminator once. If the discriminator can easily judge the real images and fake images, the gradient may be too small for generator to update the model parameters. As a consequence, I trained the generator more times in an iteration to make a balance between discriminator and generator.

2. The initial weight of model is important
   For the convolution layer, I initialize the weight as N(0, 0.02). As for the batchnorm layer, I initialize the weight as N(1, 0.02). The initialization stabilizes the training process.