Lab7 Let's Play GANs

Deep Learning and Practice

資科工碩1 游騰德 310551054

Introduction

This lab is to implement a conditional GAN for images generating. The model is to generate corresponding synthetic images according to the given multi-label conditions . The networks to be implemented will be trained on i-CLEVR dataset. This dataset contains 18,009 training data. There are totally 24 objects in i-CLEVR dataset with 3 shapes and 8 colors. The condition contains no more than 3 objects. The generator plays an important role in GAN to generate images, and the discriminator aids the generator to perform better by distinguishing the differences between real and fake images.

Implementation details

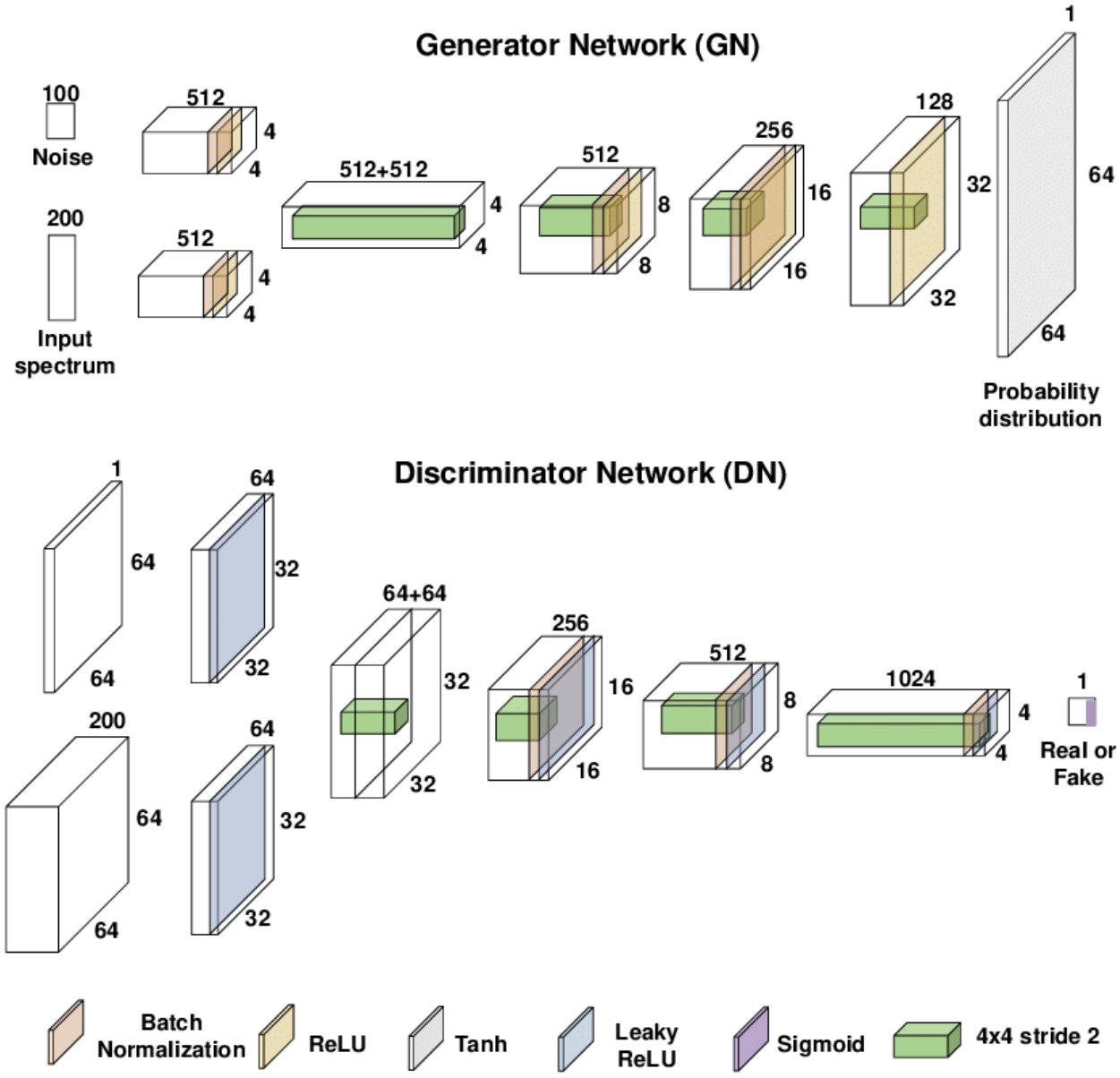
Dataloader

To increase the efficiency, I load the json file and generate one hot labels for all images in the beginning once. For memory issue, I left the image to be loaded by the pillow **Image** module every time the image data is accessed. The image will then be transformed by resizing and normalizing.

```
class CLEVRDataset(Dataset):
    def __init__(self, root='../data'):
        self.root = root
        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Resize((64, 64)),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        ])
        with open(f'{root}/objects.json') as f:
            self.objects = json.load(f)
        with open(f'{root}/train.json') as f:
            labels = json.load(f)
            self.keys = [key for key in labels]
            self.labels = [self.get_one_hot_label(labels[key]) for key in labels]

    def get_one_hot_label(self, label):
        one_hot_label = torch.zeros(len(self.objects))
        for obj in label:
            one_hot_label[self.objects[obj]] = 1
        return one_hot_label

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

    def __getitem__(self, index):
        key = self.keys[index]
        img = Image.open(f'{self.root}/iclevr/{key}').convert('RGB')
        img = self.transform(img)
        return img, self.labels[index]
```

Models

I chose cDCGAN (Conditional Deep Convolution Adversarial Network) to implement. According to my research, cDCGAN combines the benefits of both DCGAN and cGAN. Additionally, cDCGAN has higher computational speed as well as better performance comparing to general GANs.



cDCGAN structure

Generator

When forwarding through the generator, the one hot labels representing the conditions will first go through the **in_c** module, and then combine with the noise. The combined vector will then be passed through 4 hidden modules and the **out** module to generate images. **Batch normalization** and **ReLU** are used in hidden layers, and the final activation of the generator is **Tanh**.

```
Generator(
  (in_c): Sequential(
    (c_linear): Linear(in_features=24, out_features=1024, bias=True)
    (c_actv): ReLU()
  )
  (hidden1): Sequential(
    (h_conv_1): ConvTranspose2d(1124, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (h_bn_1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (h_actv_1): ReLU()
  )
  (hidden2): Sequential(
    (h_conv_2): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (h_bn_2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (h_actv_2): ReLU()
  )
  (hidden3): Sequential(
    (h_conv_3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (h_bn_3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (h_actv_3): ReLU()
  )
  (hidden4): Sequential(
    (h_conv_4): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (h_bn_4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (h_actv_4): ReLU()
  )
  (out): Sequential(
    (o_conv): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (o_actv): Tanh()
  )
)
```

Generator structure

```
    def forward(self, z, c):
        z = z.view(-1, self.z_size, 1, 1)
        h_c = self.in_c(c).view(-1, self.c_size, 1, 1)
        x = torch.cat([z, h_c], dim=1)
        x = self.hidden1(x)
        x = self.hidden2(x)
        x = self.hidden3(x)
        x = self.hidden4(x)
        out = self.out(x)
        return out
```

Generator forwarding design

Discriminator

When forwarding through the generator, the one hot labels representing the conditions will also go through the **in_c** module. The difference between the generator **in_c** module and the **in_c** module here is the output size. Due to concatenating to the input images to the discriminator, the output size of the **in_c** module here is increased to 4096, which is the size of an input image. After concatenating the images and the conditional features, the vector is then passed through all hidden modules and the **out** module. Finally, the output of size 1 will represent the quality of such input image. **Batch normalization** and **LeakyReLU** are used in hidden layers, and the final activation of the discriminator is **Sigmoid**.

```
Discriminator(
  (in_c): Sequential(
    (c_linear): Linear(in_features=24, out_features=4096, bias=True)
    (c_actv): LeakyReLU(negative_slope=0.2)
  )
  (hidden1): Sequential(
    (h_conv_1): Conv2d(4, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (h_actv_1): LeakyReLU(negative_slope=0.2)
  )
  (hidden2): Sequential(
    (h_conv_2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (h_bn_2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (h_actv_2): LeakyReLU(negative_slope=0.2)
  )
  (hidden3): Sequential(
    (h_conv_3): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (h_bn_3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (h_actv_3): LeakyReLU(negative_slope=0.2)
  )
  (hidden4): Sequential(
    (h_conv_4): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (h_bn_4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (h_actv_4): LeakyReLU(negative_slope=0.2)
  )
  (out): Sequential(
    (o_conv): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (o_actv): Sigmoid()
  )
)
```

Discriminator structure

```python
    def forward(self, img, c):
        h_c = self.in_c(c).view(-1, 1, self.img_shape[0], self.img_shape[1])
        x = torch.cat([img, h_c], dim=1)
        x = self.hidden1(x)
        x = self.hidden2(x)
        x = self.hidden3(x)
        x = self.hidden4(x)
        out = self.out(x)
        return out.view(-1)
```

Discriminator forwarding design

Loss function

Since the goal of the discriminator is to distinguish the real and fake images, I calculate binary cross-entropy loss between the outputs and targets. I utilized **BCELoss()** from **torch.nn**, which is a commonly used criterion in GAN that can prevent the generator and the discriminator from going easy for each other.

Hyperparameters

- Latent size = 100
- Condition size = 1024
- Batch size = 128
- Learning rate = 0.0002
- Epochs = 200
- Optimizer: Adam(betas=(0.5, 0.999))

Results

test.json

Generated images

Evaluator F1-score accuracy

- 0.7222222222222222

```
test.json Test Acc: 0.7222222222222222
```

new_test.json

Generated images

Evaluator F1-score accuracy

- 0.75

```
new_test.json Test Acc: 0.75
```

Discussion

Feature extractors

At the beginning of model design, I referenced this GitHub repository to aid my model structure design. The design consist of **in_c** and **in_z** modules in the generator. The **in_c** module is just the same as the aforementioned one, and the **in_z** module is a module that extract the feature from the input noise **z**. When forwarding through the generator, the condition labels and the noise will be separately passed through **in_c** and **in_z**, respectively. The both extracted features will the be concatenated and then fed into the hidden layers. In the discriminator, there was as **in_img** module that act like **in_z** in the generator to extract the feature of the input images. These designs, however, can can only reach F1-score accuracy of 0.15, which was extremely unsatisfactory.

I abandoned feature extractors **in_z** and **in_img**, and fintuned the input and output size of the hidden layers to become my final design. This new modified design results in achieving satisfactory F1-score accuracy above 0.7 on both test.json and new_test.json.

Generator and discriminator balancing

During training, I realized that the generator loss to be very stable. My guess was that the gradient for the generator was too small due to the discriminator's ability. Consequently, I modified the training process to train the generator four times more than that of the discriminator, hoping that the generated images can be more difficult for the discriminator to distinguish. This modification for the training process did benefit to increase the testing F1-score accuracy from 0.6 to 0.7.

Weights initialization

I realized that the initialize weights is such an important cause to the failure of cGAN training. As a result, I normalize the weights in every convolution layer to follow N(0, 0.2) and set the bias as false. Also, in every batch normalization layer, I normalize the weights to follow N(1, 0.2) and set the bias to 0. I believe that the initialization aids the training of cGAN to become stable.