Deep Learning and Practice Lab 5

309554005 黃睿宇

1 Introduction

To achieve higher generation capacity, especially in computer vision, generative adversarial network (GAN) is proposed and has been widely applied on style transfer and image synthesis. In this lab, we implement a conditional GAN to generate synthetic images according to multi-label conditions. Given a specific condition, our model generates the corresponding synthetic images (see in Fig. 1). For example, given “red cube” and “blue cylinder”, our model then generates the synthetic images with red cube and blue cylinder and meanwhile, input our generated images to a pre-trained classifier for evaluation.

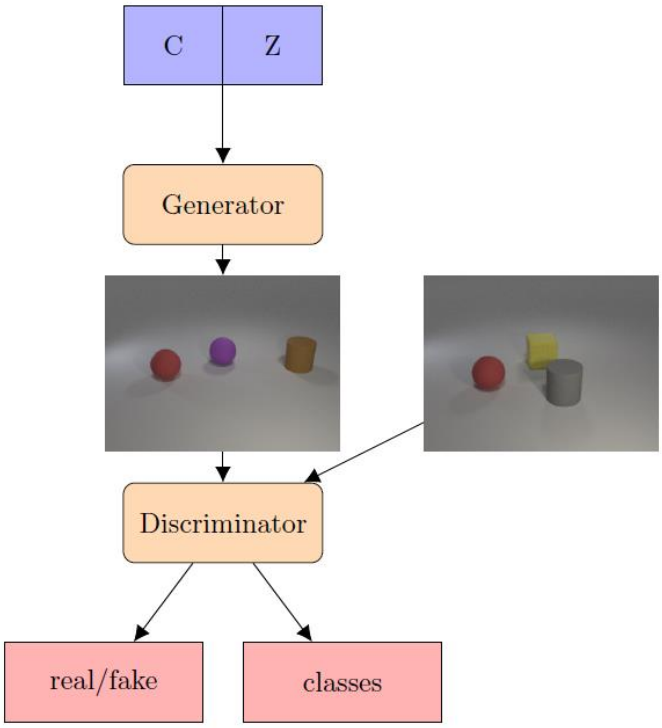


Figure 1: The illustration of cGAN.

There are totally 24 objects in i-CLEVR datasets with 3 shapes and 8 colors in our dataset, and thus the condition will be a 24-dim one-hot vector, e.g., [0,1,0,0, … ,0,0]. In training dataset, there are 18012 training data. The training file is a dictionary where keys are filenames and values are objects. In testing dataset, there are 32 testing data. The testing file is a list where each element includes multiple objects.

2 Implementation Details

2.1 Generator

We use cDCGAN as the model architecture. In generator, condition vector passes through a fully connected layer to augment the dimension from 24 to 200. Then we concatenate condition vector with 100-dim noise z and passes the vector through transpose convolution layer. Finally, it'll become fake images.

```
class Generator(nn.Module):
    def __init__(self, z_dim, c_dim):
        super(Generator, self).__init__()
        self.z_dim = z_dim
        self.c_dim = c_dim
        self.conditionExpand = nn.Sequential(
            nn.Linear(24, c_dim),
            nn.ReLU()
        )
        kernel_size = (4, 4)
        channels = [z_dim + c_dim, 512, 256, 128, 64]
        paddings = [(0, 0), (1, 1), (1, 1), (1, 1)]
        for i in range(1, len(channels)):
            setattr(self, 'convT' + str(i), nn.Sequential(
                nn.ConvTranspose2d(channels[i-1], channels[i],
                    kernel_size, stride=(2, 2), padding=paddings[i-1]),
                nn.BatchNorm2d(channels[i]),
                nn.ReLU()
            ))
        self.convT5 = nn.ConvTranspose2d(
            64, 3, kernel_size, stride=(2, 2), padding=(1, 1))
        self.tanh = nn.Tanh()

    def forward(self, z, c):
        """
        :param z: (batch_size, 100) tensor
        :param c: (batch_size, 24) tensor
        :return: (batch_size, 3, 64, 64) tensor
        """
```

```
        z = z.view(-1, self.z_dim, 1, 1)
        c = self.conditionExpand(c).view(-1, self.c_dim, 1, 1)
        out = torch.cat((z, c), dim=1)  # become(N, z_dim+c_dim, 1, 1)
        out = self.convT1(out)  # become(N, 512, 4, 4)
        out = self.convT2(out)  # become(N, 256, 8, 8)
        out = self.convT3(out)  # become(N, 128, 16, 16)
        out = self.convT4(out)  # become(N, 64, 32, 32)
        out = self.convT5(out)  # become(N, 3, 64, 64)
        out = self.Tanh(out)    # output value between [-1, +1]
        return out
```

2.2 Discriminator

In discriminator, 24-dim condition vector passes through a fully connected layer and reshapes into a 1*64*64 image. Then we concatenate condition vector with image in generator, then it becomes a 3+1*64*64 image. After passing through five convolution layer, we'll get a scalar.

```
class Discriminator(nn.Module):
    def __init__(self, img_shape, c_dim):
        super(Discriminator, self).__init__()
        self.H, self.W, self.C = img_shape
        self.conditionExpand = nn.Sequential(
            nn.Linear(24, self.H * self.W * 1),
            nn.LeakyReLU()
        )
        kernel_size = (4, 4)
        channels = [4, 64, 128, 256, 512]
        for i in range(1, len(channels)):
            setattr(self, 'convT' + str(i), nn.Sequential(
                nn.Conv2d(channels[i-1], channels[i],
                    kernel_size, stride=(2, 2), padding=(1, 1)),
                nn.BatchNorm2d(channels[i]),
                nn.LeakyReLU()
            ))
        self.conv5 = nn.Conv2d(
            512, 1, kernel_size, stride=(1, 1))
        self.sigmoid = nn.Sigmoid()
```

```
    def forward(self, X, c):
        """
        :param X: (batch_size, 3, 64, 64) tensor
        :param c: (batch_size, 24) tensor
        :return: (batch_size) tensor
        """
        c = self.conditionExpand(c).view(-1, 1, self.H, self.W)
        out = torch.cat((X, c), dim=1)  # become(N, 4, 64, 4)
        out = self.conv1(out)  # become(N, 64, 32, 32)
        out = self.conv2(out)  # become(N, 128, 16, 16)
        out = self.conv3(out)  # become(N, 256, 8, 8)
        out = self.conv4(out)  # become(N, 512, 4, 4)
        out = self.conv5(out)  # become(N, 1, 1, 1)
        out = self.Sigmoid(out)    # output value between [0, 1]
        out = out.view(-1)
        return out
```

2.3 Loss function

Since the output is a scalar that represents whether the image is true or not, it can be seen as a two-class problem. Thus, we use binary cross entropy as our loss function.

```
def train(dataloader, g_model, d_model, z_dim, epochs, lr):
    Criterion = nn.BCELoss()
    optimizer_g = torch.optim.Adam(
        g_model.parameters(), lr, betas=(0.5, 0.99))
    optimizer_d = torch.optim.Adam(
        d_model.parameters(), lr, betas=(0.5, 0.99))
    evaluation_model = EvaluationModel()
```

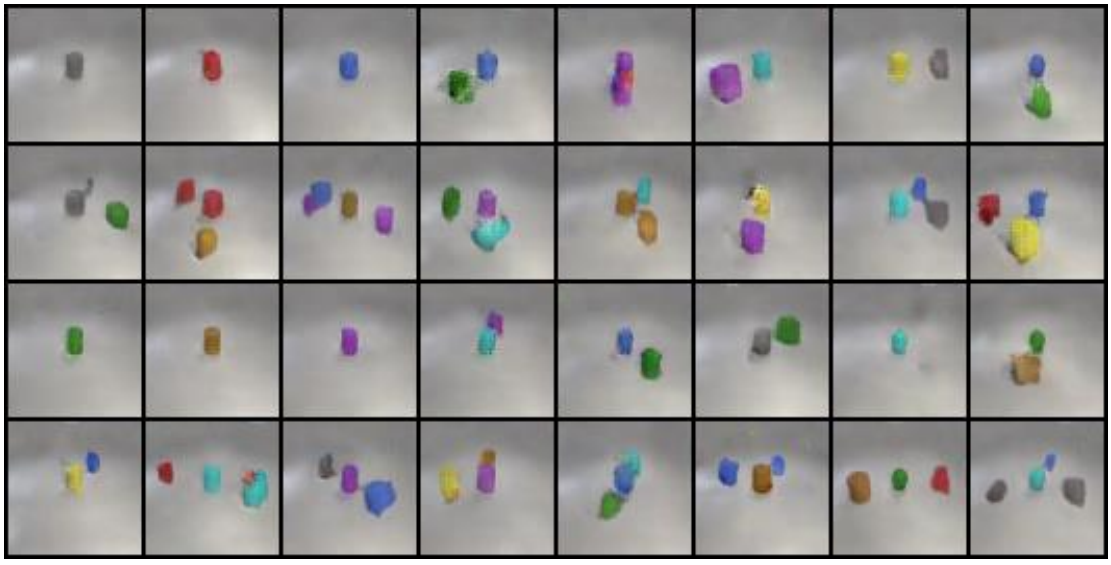
2.4 Hyperparameters

- Optimizer: Adam
- Loss function: binary cross entropy
- Epochs = 200
- Batch size = 64
- Learning rate = 2e-4
- Latent size = 100

- Condition size = 200

3 Results and Discussion

3.1 Results



```
/home/ubuntu/anaconda3/envs/pytorch/lib/python3.8/site-packages/torch/nn/functi
onal.py:718: UserWarning: Named tensors and all their associated APIs are an ex
perimental feature and subject to change. Please do not use them for anything i
mportant until they are released as stable. (Triggered internally at  /opt/cond
a/conda-bld/pytorch_1623448234945/work/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_m
ode)
Test score: 0.708333
```

3.2 Discussion

- In training phase, generator loss should be as close as possible to discriminator loss, so that we adjusted the ratio of training times of generator to discriminator to 4:1, which made their loss be close to each other in the early stage of training.

- As for the accuracy, we have some discoveries:
  1) Training generator four times or five times have similar results.
  2) Doing batch normalization can raise the accuracy.
  3) The condition vector used in generating fake images is chose in existing training data instead of generating randomly, since generating condition vector randomly might get a worse result.