DL HW6

1. Introduction

In this assignment, we should generate images contains multiple colorful objects by conditional GAN. GAN is generative adversarial network. It can view as the competition between the generator and discriminator. Generator try to fool the discriminator, and discriminator try to identify the true images. This technique can generate many different images by different input noise. However, input noise is chosen randomly. We don't know that really means. So we can give it some hint as condition and we can control the output. In this task, what kind of object should be synthesized depends on the condition we input. There are three kind of shapes: cube, sphere, cylinder and eight kinds of colors: gray, red, blue, green, brown, purple, cyan, yellow. I use one hot vector to represent the labels of objects. I use DCGAN as main architecture of my generator and discriminator. In discriminator part, I also combine the concept of ACGAN.

2. Implementation details

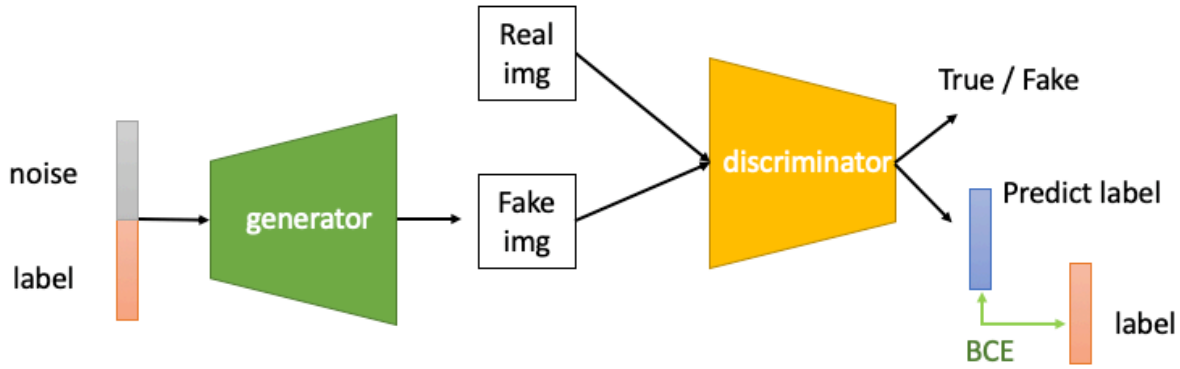
A. Describe how you implement your model

- Dataloader

The image resize to 64 x 64 and normalize by transforms.Normalize(mean = (0.5, 0.5, 0.5), std = (0.5, 0.5, 0.5))

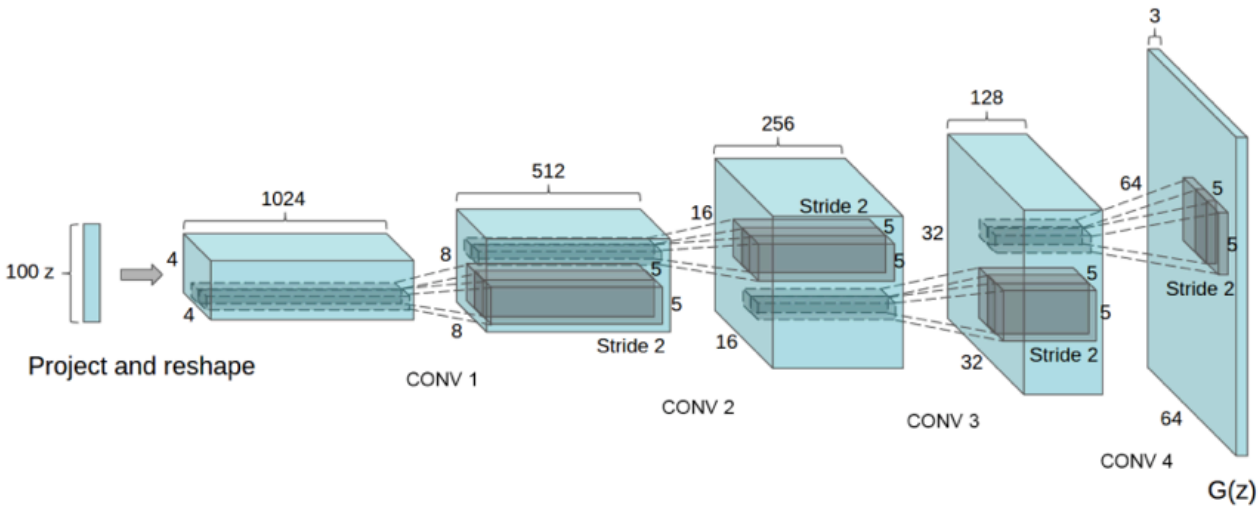
```
class ImageLoader(data.Dataset):
    def __init__(self, root, mode, size):
        self.root = root
        self.mode = mode
        if mode == 'train':
            self.img_name, self.label, self.object_dict = getData(mode)
        else:
            self.label, self.object_dict = getData(mode)
        self.transform = transforms.Compose(
                                [transforms.Resize(size, interpolation=Image.BILINEAR),
                                transforms.ToTensor(),
                                transforms.Normalize(mean = (0.5, 0.5, 0.5), std = (0.5, 0.5, 0.5))
                                ])

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

    def __getitem__(self, index):
        if self.mode == 'train':
            path = self.root + self.img_name[index]
            label = self.label[index]
            img = Image.open(path).convert('RGB')
            img = self.transform(img)

            return img, torch.from_numpy(label.astype(np.float32))
        else:
            label = self.label[index]
            return torch.from_numpy(label.astype(np.float32))
```

- Main architectures



The main architecture illustrates in the above image. I concat the gaussian noise and label , and input to the generator. Both shape are 24 x 1 x 1. The network of generator and discriminator is DCGAN. There is a little difference in discriminator. My discriminator has two output. Like the ACGAN, the discriminator output the probability and the multi-label.

- Generator



The structure is the same as the above image. Only the input is different. I use 24 x 1 x 1 gaussian noise and concat 24 x 1 x 1 label. The structure uses *nn.ConvTranspose2d* to recover the noise to the image. At the last layer, adds the Tanh() to let output values between -1 and 1.

```python
# Generator
class Generator(nn.Module):
    def __init__(self, inputSize, hiddenSize, outputSize):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(inputSize, hiddenSize*8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(hiddenSize*8),
            nn.ReLU(True),

            nn.ConvTranspose2d(hiddenSize*8, hiddenSize*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(hiddenSize*4),
            nn.ReLU(True),

            nn.ConvTranspose2d(hiddenSize*4, hiddenSize*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(hiddenSize*2),
            nn.ReLU(True),

            nn.ConvTranspose2d(hiddenSize*2, hiddenSize, 4, 2, 1, bias=False),
            nn.BatchNorm2d(hiddenSize),
            nn.ReLU(True),

            nn.ConvTranspose2d(hiddenSize, outputSize, 4, 2, 1, bias=False),
            nn.Tanh())

    def forward(self, input):
        return self.main(input)
```

- Discriminator

The structure is like inverse of the generator. Using *nn.Conv2d* to compute the probability of the true or fake image. When output label, I use flatten layer at the second last in stead of *nn.Conv2d.* Then use *nn.Linear* to output 24 channel.

```python
# Discriminator
class Discriminator(nn.Module):
    def __init__(self, inputSize, hiddenSize):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(inputSize, hiddenSize, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(hiddenSize, hiddenSize*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(hiddenSize*2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(hiddenSize*2, hiddenSize*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(hiddenSize*4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(hiddenSize*4, hiddenSize*8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(hiddenSize*8),
            nn.LeakyReLU(0.2, inplace=True),


            )
        self.pro = nn.Sequential(
            nn.Conv2d(hiddenSize*8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()

            )

        self.label = nn.Sequential(
            nn.Flatten(start_dim=1, end_dim=3),
            nn.Linear(in_features=8192, out_features=24, bias=True),
            nn.Sigmoid()
            )

    def forward(self, input):
        main = self.main(input)
        pro = self.pro(main)
        label = self.label(main)
        return pro, label
```

- Loss function

There are two kinds of loss. One is adversarial loss L_{adv}, and the other is classification loss L_{class}. Both loss compute by binary cross entropy.

For generator, we hope the synthesized image can fool the discriminator. So the probability of fake image should be close to 1. Also, the classification should be same as the input label.

$$Loss_G = L_{adv}(D_{pro}(G(x)),1) + L_{class}(D_{class}(G(x)), label)$$

```
# Update G
optimizerG.zero_grad()

pro_true, label_true = D(img.cuda(GPUID[0]))
D_true_class_loss = criterion(label_true, label.cuda(GPUID[0]))
gen_img = G(noise.cuda(GPUID[0]))
pro_fake, label_fake = D(gen_img.cuda(GPUID[0]))
G_adv_loss = criterion(pro_fake.view(batch_size, 1), torch.ones(batch_size, 1).cuda(GPUID[0]))*0.1
G_class_loss = criterion(label_fake, label.cuda(GPUID[0])) * 20
G_loss = G_adv_loss + G_class_loss
G_loss.backward()
optimizerG.step()

G_avg += G_loss.data.cpu()
G_adv_avg += G_adv_loss.data.cpu()
G_class_avg += G_class_loss.data.cpu()
```

For discriminator, we hope the identify the true image c. So the probability of fake image should be close to 0 and true one should be close to 1. Also, the true classification should be same as the input label.

$$Loss_D = L_{adv}(D_{pro}(real\_img),1) + L_{adv}(D_{pro}(G(x)),0) + L_{class}(D_{class}(real\_img), label)$$

```
# Update D
optimizerD.zero_grad()
pro_true, label_true = D(img.cuda(GPUID[0]))

pro_fake, label_fake = D(gen_img.cuda(GPUID[0]))
D_true_adv_loss = criterion(pro_true.view(batch_size, 1), torch.ones(batch_size, 1).cuda(GPUID[0]))
D_true_class_loss = criterion(label_true, label.cuda(GPUID[0])) * 10
D_fake_adv_loss = criterion(pro_fake.view(batch_size, 1), torch.zeros(batch_size, 1).cuda(GPUID[0]))

D_loss = D_true_adv_loss + D_true_class_loss + D_fake_adv_loss
D_loss.backward()
optimizerD.step()

D_avg += D_loss.data.cpu()
D_true_adv_avg += D_true_adv_loss.data.cpu()
D_true_class_avg += D_true_class_loss.data.cpu()
D_fake_adv_avg += D_fake_adv_loss.data.cpu()

pro_fake_avg += pro_fake[0].mean().data.cpu()
pro_true_avg += pro_true[0].mean().data.cpu()
```

B. specify the hyperparameters

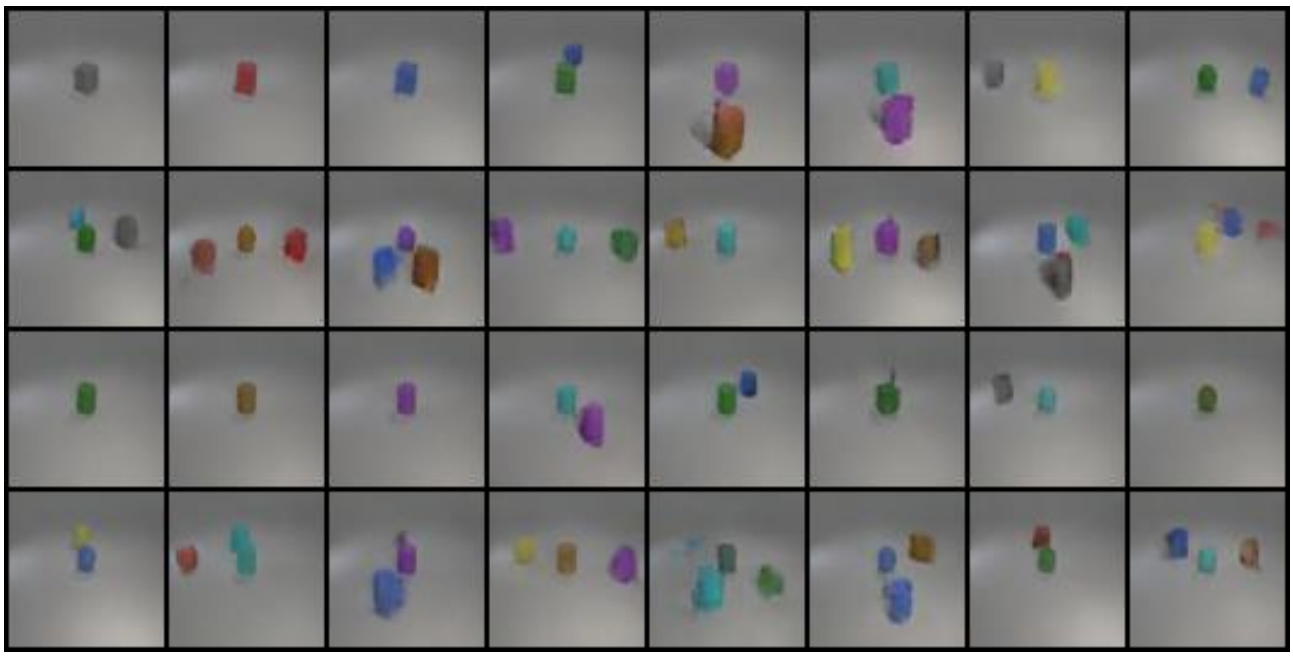
image size = (64, 64)
batch size = 16
learning rate = 0.0002
epoch num = 704
decay = 5e-4
inputSize = 48 (24 noise , 24 label)
hiddenSize = 64
outputSize = 3

beta1 = 0.5
Optimizer Adam betas=(beta1, 0.999)
D_true_adv_loss * 1
D_true_class_loss * 10
D_fake_adv_loss * 1
G_adv_loss * 0.1
G_class_loss * 20

2. Results and discussion

A. Show your results based on the testing data

accuracy = 0.80555



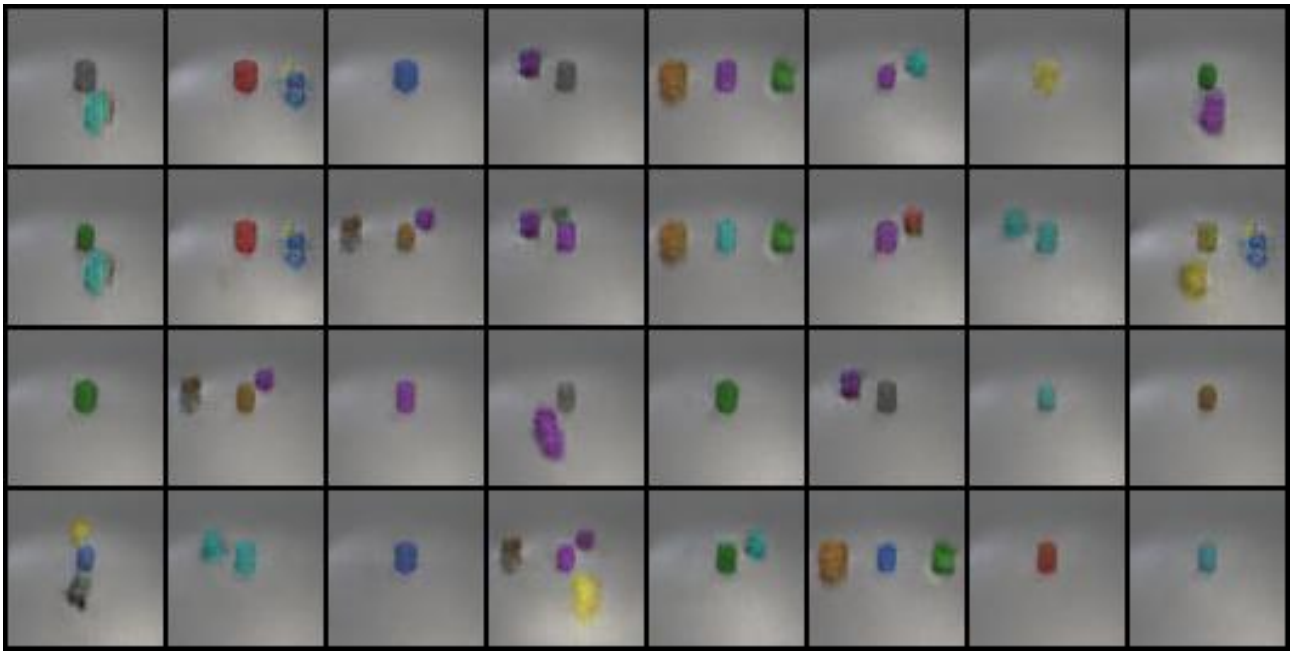
B. Discuss the results of different models architectures.

In the beginning, I also add the fake classification loss in the discriminator.

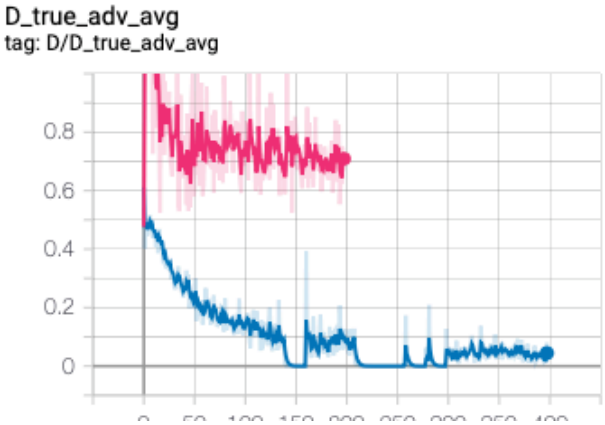
$$Loss_D = L_{adv}(D_{pro}(real\_img),1) + L_{adv}(D_{pro}(G(x)),0) + L_{class}(D_{class}(real\_img), label) + L_{class}(D_{class}(G(x)), label)$$

The below image is the result. This cause the model can't learn the correct class well. Because this loss exists in both generator and discriminator. It causes the whole model tends to make the fake label loss lower. Generator can only produce the image those label can fool the discriminator but not the correct label. Therefore, I remove this term in the

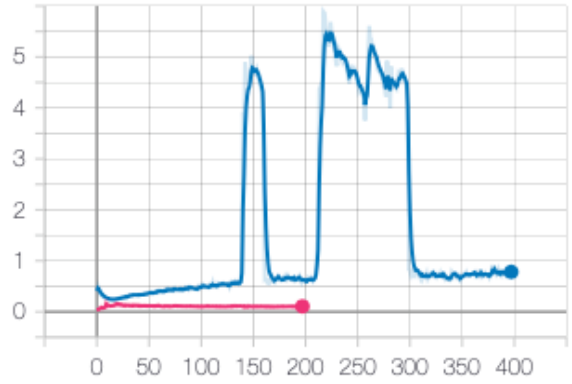
discriminator and it more make sense. Discriminator is optimized by true label and have the ability to classify the correct label. Generator also try to generate the correct label to fool the discriminator.



I also observe the training loss in the process. I find the loss of discriminator is very low and generator is unstable. I think it means the power of discriminator is too strong. It identify the image well but generator haven't learn anything. Therefore, I adjust the update frequency of the discriminator. I update it each 100 steps. I also try update each 10 steps and 200 steps find 100 is best. The update times of generator is 100 times than discriminator.

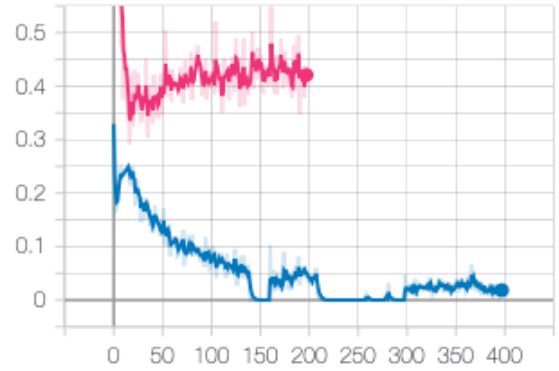


G_adv_avg
tag: G/G_adv_avg



Above image shows the trend of loss. The blue line represents origin trend, the pink one is new version of update discriminator each 100 steps. We can look more clear at below images. It's the probability of fake and true images discriminator identify. We can find the both probability in pink one are around 0.5. It means discriminator hardly identify true and fake one.

pro_fake_avg
tag: Pro/pro_fake_avg



pro_true_avg
tag: Pro/pro_true_avg