Report

• Introduction (5%)

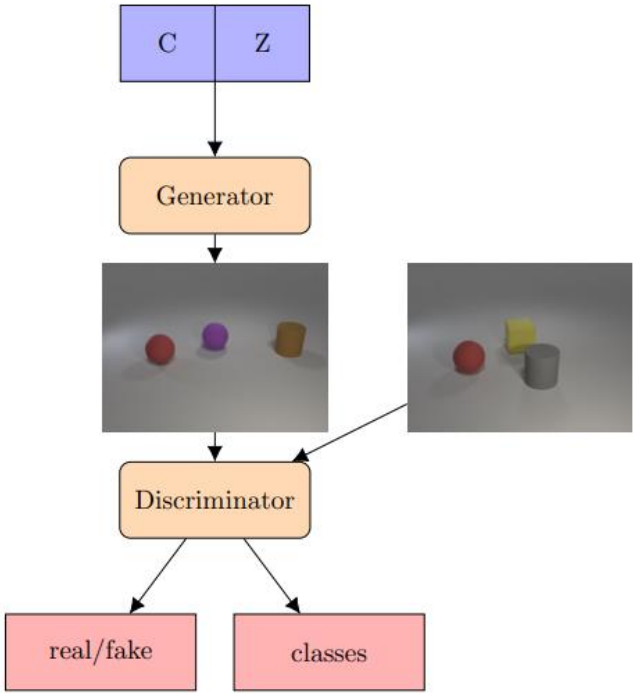


Figure 1: The illustration of cGAN.

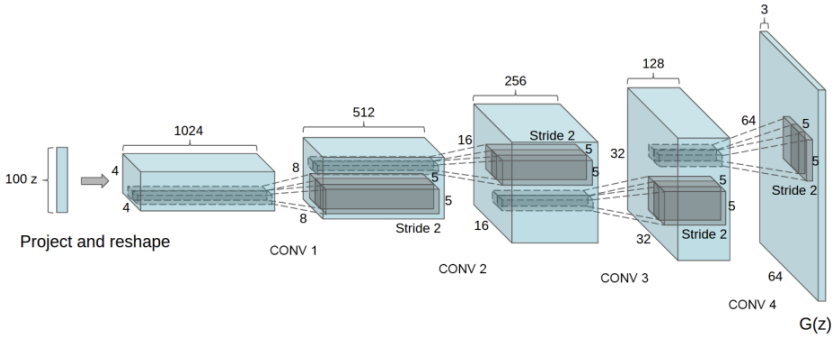
According the slide, we need to implement and train for cGAN(Conditional GAN) model in this lab. cGAN is like GAN but we also need to input the condition data to generator and discriminator and they need to generate the image with the condition like “gray cube” and discriminate that and output whose is real or fake.

• Implementation details (15%)

– Describe how you implement your model, including your choice of cGAN, model architectures, and loss functions. (10%)

My model of cGAN is based on DCGAN paper https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html :

And the “C” in the DCGAN is stand for CNN. This model is composed with convolution layers, batch norm layers, and LeakyReLU activations.



The main different is that we only need to input noise in the paper's generator. Then we also need to input the condition in the model. However, there are only 24 type of object in objects.json. So, we need to expand the condition vector to similar size to noise / image (for generator / discriminator).

```
class Generator(nn.Module):
    def __init__(self, nz, nc, ngf):
        super(Generator,self).__init__()
        self.nz = nz
        self.nc = nc
        self.ngf = ngf

        self.expand = nn.Sequential(
            nn.Linear(24, nc),
            nn.BatchNorm1d(nc),
            nn.ReLU()
        )

        self.main = nn.Sequential(
            # input is Z, going into a convolutio
            nn.ConvTranspose2d( nc + nz, self.ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(self.ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(self.ngf * 8, self.ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( self.ngf * 4, self.ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( self.ngf * 2, self.ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( self.ngf, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, z, c):
        z = z.view(-1,self.nz,1,1)
        c = self.expand(c).view(-1, self.nc, 1, 1)
        x = torch.cat((z, c), dim=1)
        return self.main(x)
```

```python
class Discriminator(nn.Module):
    def __init__(self, img_shape, ndf):
        super(Discriminator, self).__init__()
        self.img_shape = img_shape
        self.ndf = 64

        self.expand = nn.Sequential(
            nn.Linear(24, self.img_shape[0] * self.img_shape[1]),
            nn.BatchNorm1d(self.img_shape[0] * self.img_shape[1]),
            nn.LeakyReLU()
        )

        self.act = nn.LeakyReLU();
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(4, self.ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(self.ndf, self.ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(self.ndf * 2, self.ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(self.ndf * 4, self.ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(self.ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, X, c):
        c = self.expand(c).view(-1, 1, self.img_shape[0], self.img_shape[1])
        x = torch.cat((X, c), dim=1)
        return self.main(x).view(-1)
```

Loss function

```python
criterion  = nn.BCELoss()
```

Using the BSEloss in pytorch, which is:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right]$$

LossD: maximum (log(D(x)) + log(1 - D(G(z)))log(D(x))+log(1−D(G(z)))).
LossG: minmun log(D(G(z)))

– Specify the hyperparameters (learning rate, epochs, etc.) (5%)
According the upper web site, it gives some advice about hyperparameters.

- learning rate: 0.002 (base on the paper)
- Size of z (nz): 100 (base on the paper)
- Size of c (nc): 100 (different to the site, the parameter is used for expand the size of)
- ndf: 64 (the parameter for the size of discriminator)
- ngf: 64 (the parameter for the size of generator)
- batch_size: 128
- beta1 = 0.5 (for Adam optimizers)

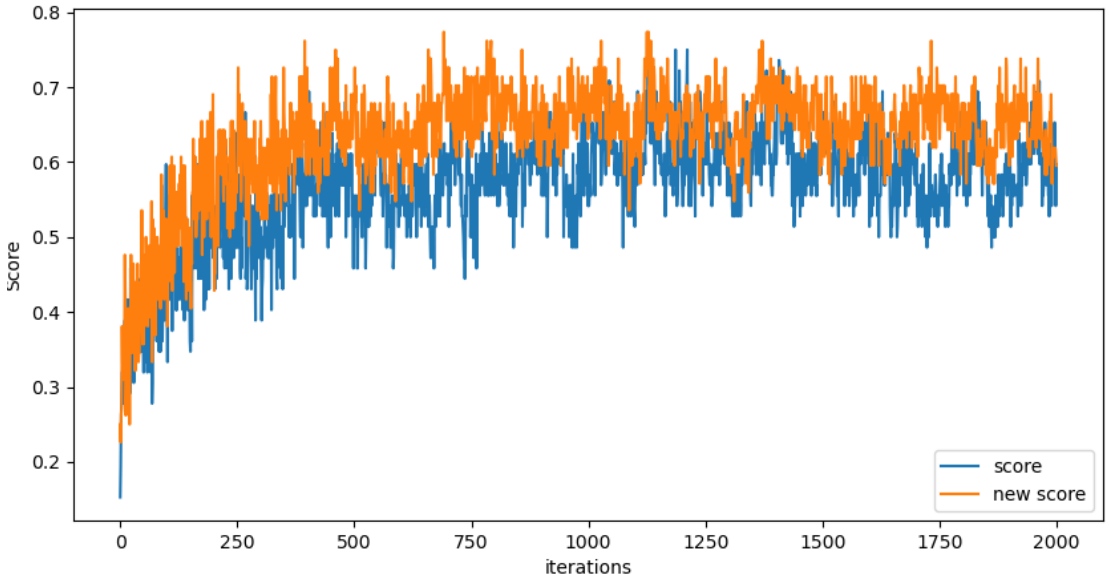
- epoch = 300 ~ 2000 (according the batch_size, nz, nc,)

• Results and discussion (30%)

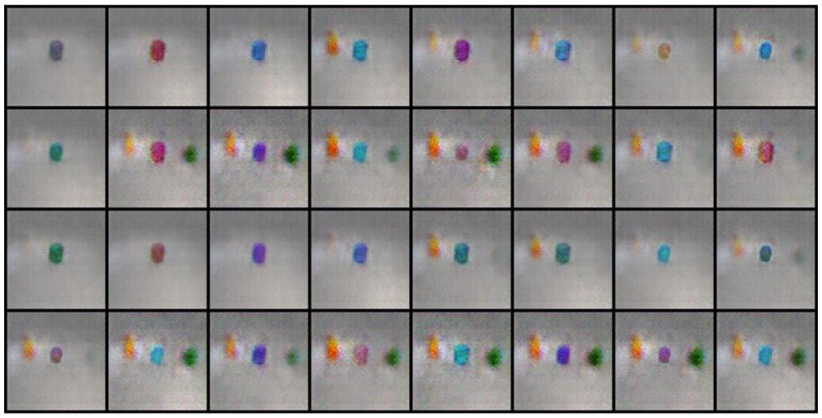
– Show your results based on the testing data. (5%) (including images)

I test with the evaluator in every epoch. And the following figure is the result:

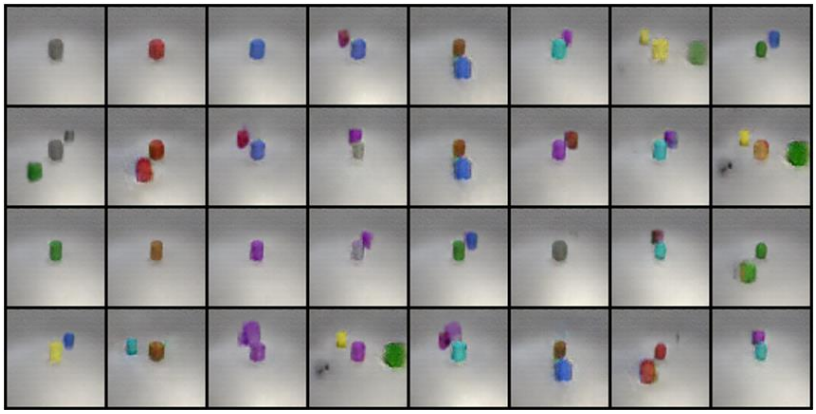
The “new” score is the score for the new_test.json



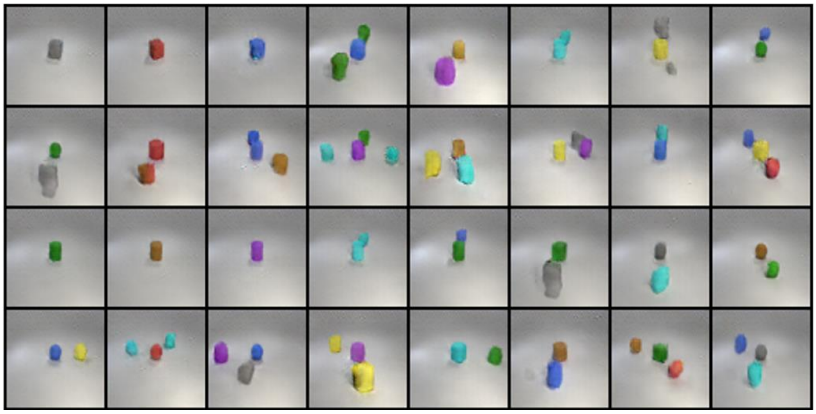
We the score is 0.125



When the score is 0.389

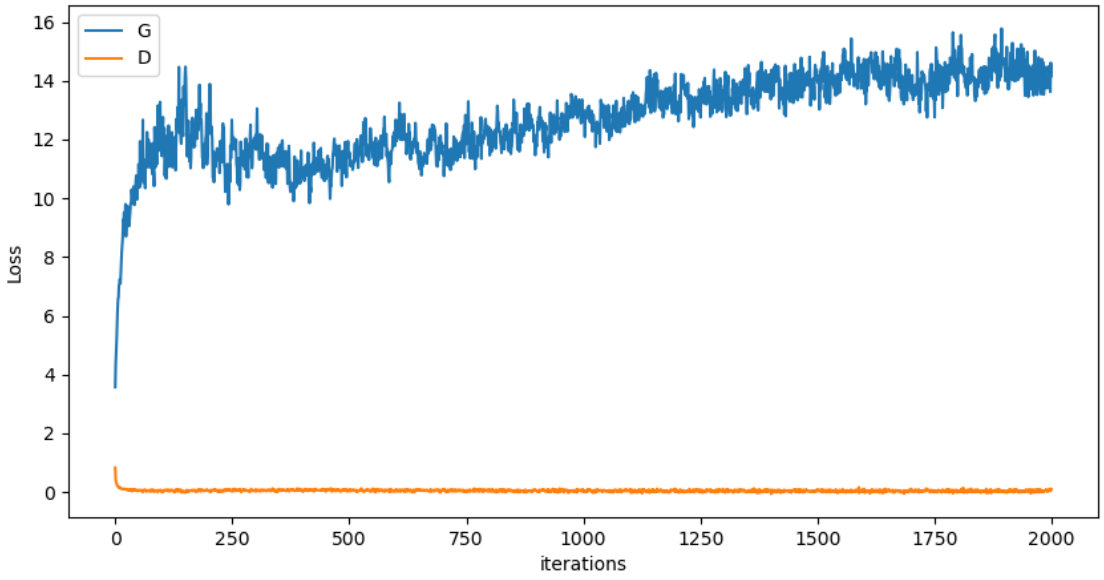


When the score is 0.611



– Discuss the results of different models architectures. (25%)
For example, what is the effect with or without some specific loss terms, or what kinds of condition design is more effective to help cGAN.

One of the most important problem is that discriminator is obviously stronger then generator. The following is the figure of loss, we can find that the loss of D is very small to the loss of G.



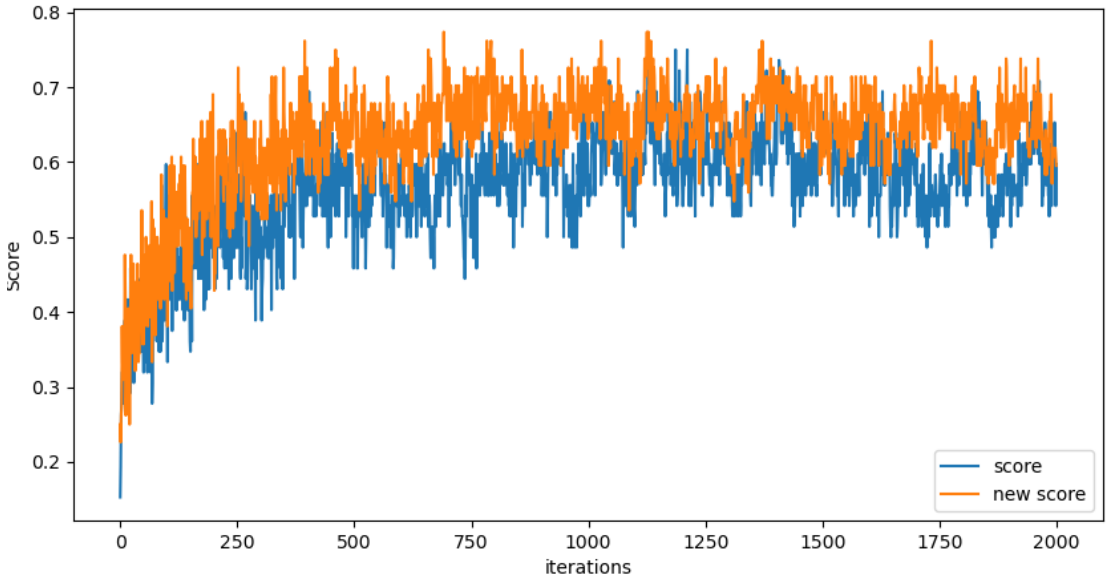
So, I tried some different design. One is increasing the size and layer in generator. However, it just converges in less epoch, not other obvious effect.

In another try, I modified the value of label. Because sometime I train data and not converge to a good score. So I give some random value to make the model to not stay in local optimal.

```
real_label = torch.full((b_size,), 1, dtype=torch.float, device = device) - torch.randn(b_size, device = device) * 0.05
fake_label = torch.full((b_size,), 0, dtype=torch.float, device = device) + torch.randn(b_size, device = device) * 0.05
```

With the implement, I get the 0.7 score for the two testcase. But I am not sure it really works or I have a good luck, training time is too long.

• Experimental results (50%) (based on results shown in your report)



(This figure again)

At the epoch 1128, I get the highest score in those 2000 epoch.