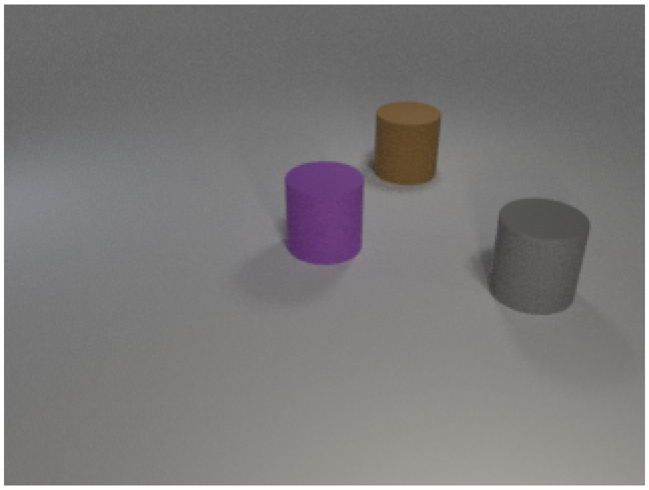
Lab5 Let's Play GANs

311554046 林愉修

1. Introduction

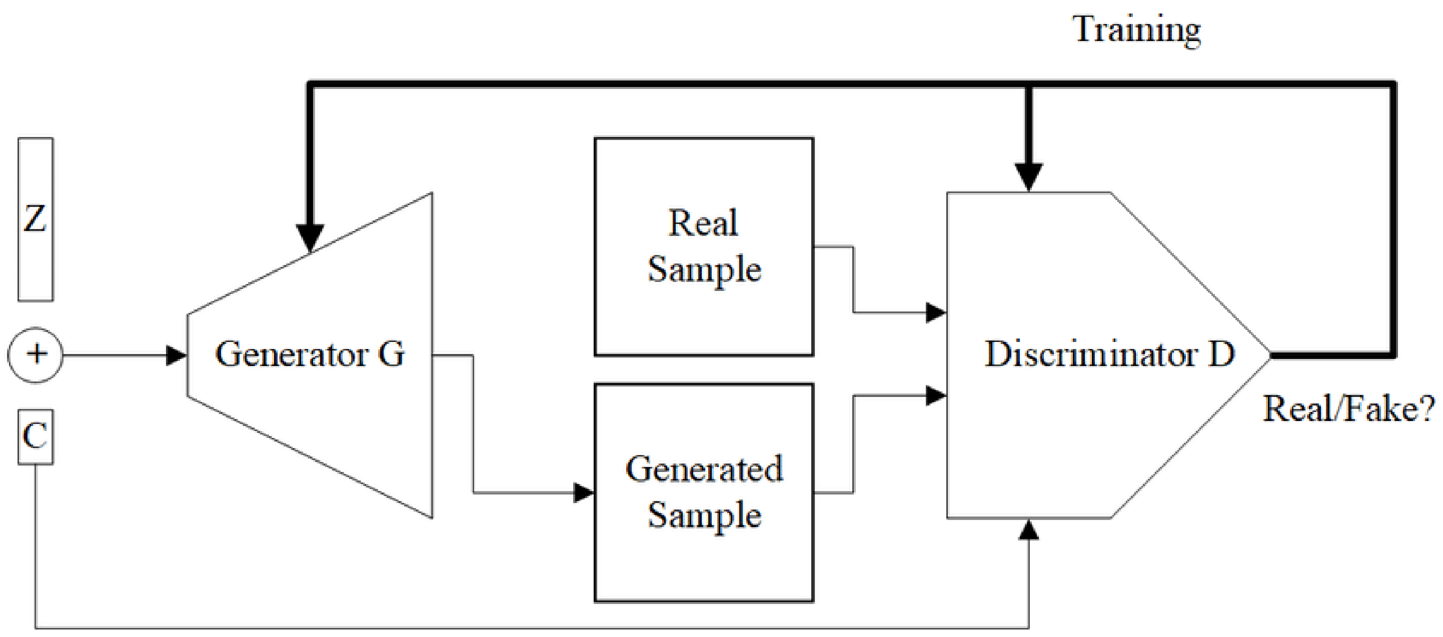
In this lab, we are going to implement a conditional GAN (cGAN) to generate synthetic images according to multi-label conditions, which is provided with the i-CLEVR dataset. The model is expected to take a condition as input and generate an image with the objects specified in the condition. There are totally 24 objects in the dataset with 3 shapes and 8 colors, so the condition is processed as an **24-dim one-hot vector**.



2. Implementation Details

2.1 cGAN

A common way to implement cGAN is to feed the condition vector into **both generator and discriminator**, as the figure shown below.



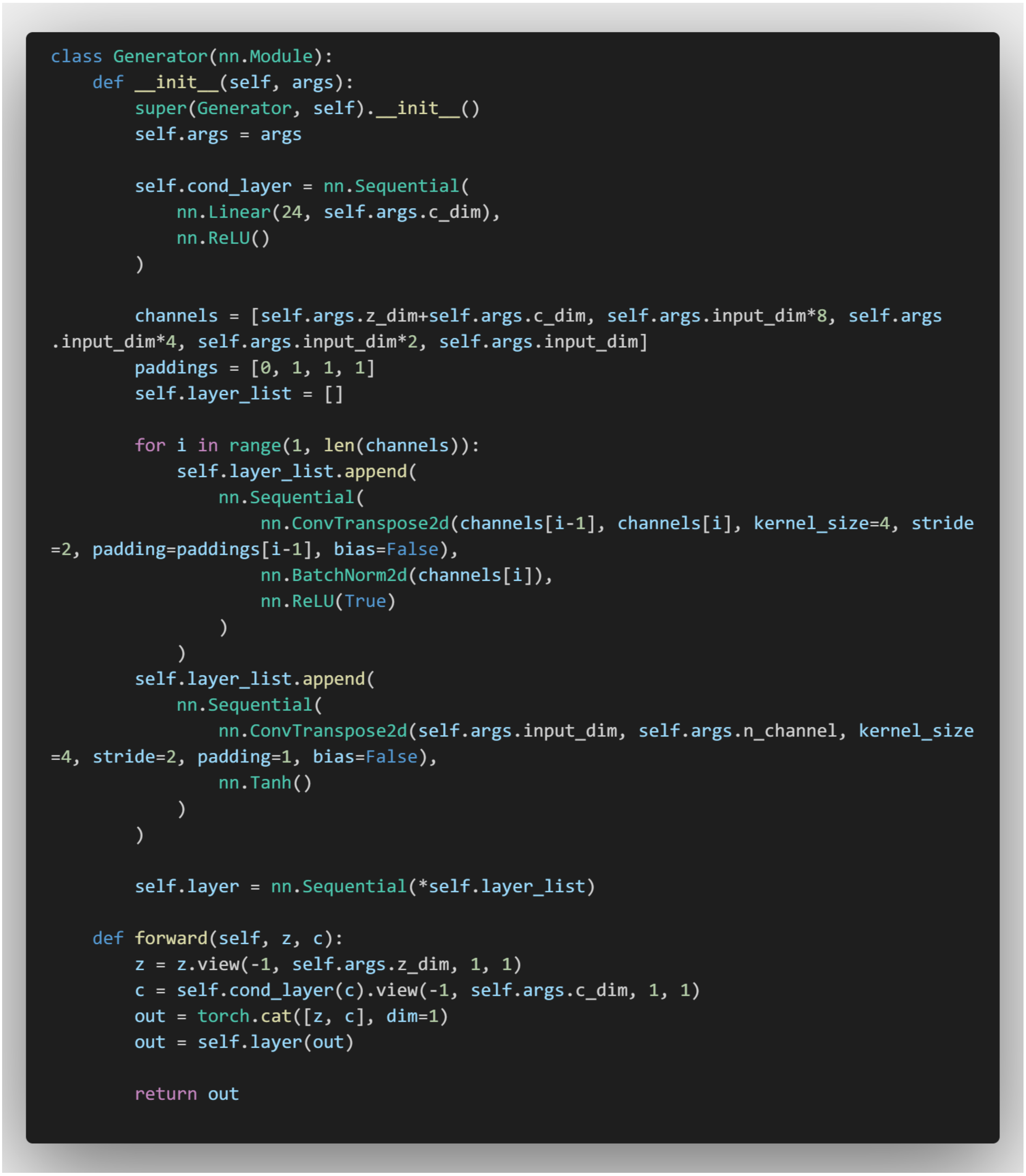
The generator is thus encouraged to generate image considering specific codition, and the discriminator not only classifies real and fake images, but also learns to determine whether a given image match the condition.

2.2 Model Architectures

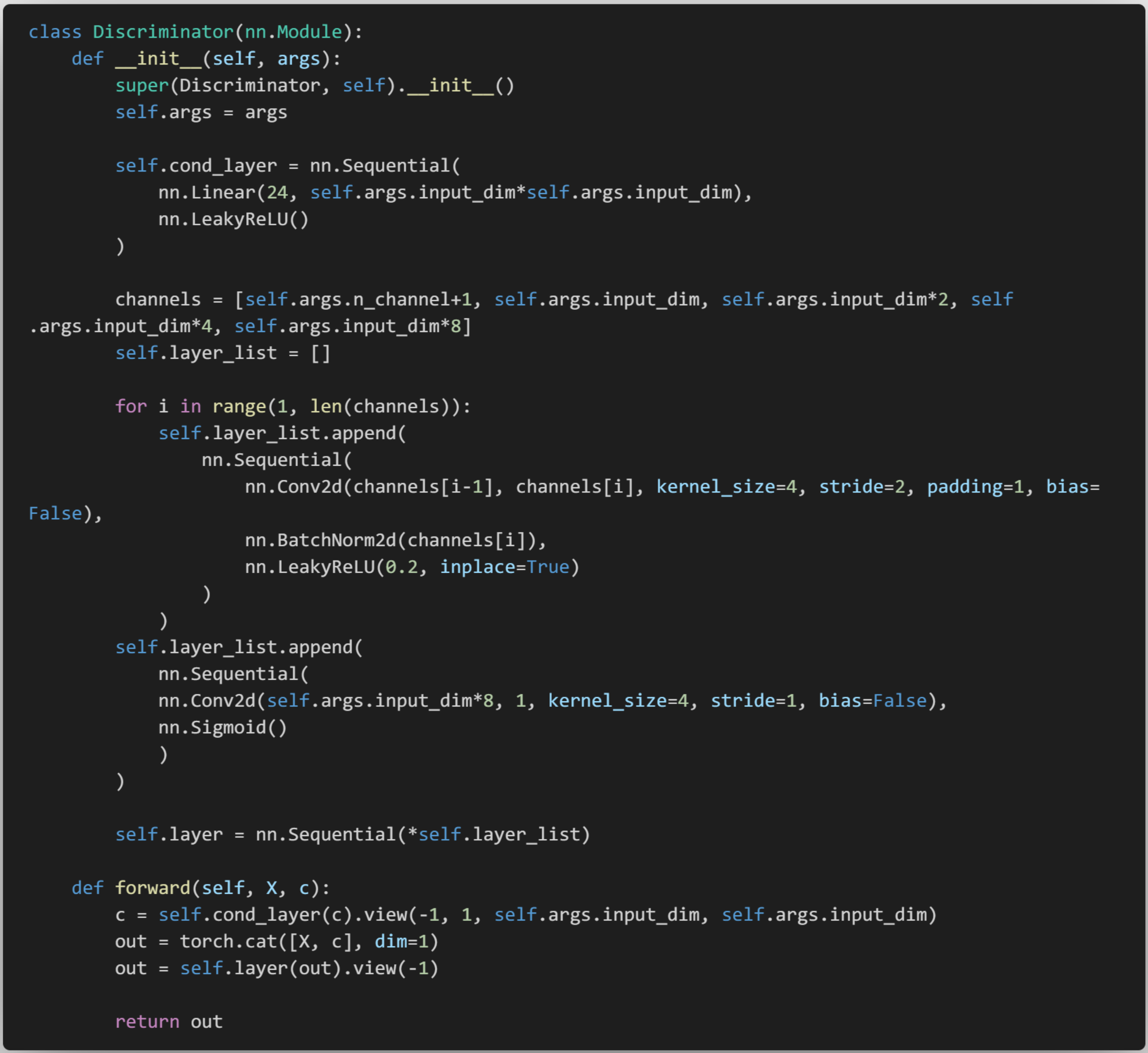
I use **Deep Convolution GAN** (DCGAN) as the model architecture.

The condition vector will first pass through a fully-connected layer, projecting to higher dimension (24->200), and this feature vector is then concatenated with a Gaussian noise (100-dim) and fed into the generator.

The discriminator, as mentioned above, also takes the condition vector as a part of its inputs. By feeding the condition vector into a fully-connected layer and reshape, the shape of the condition (1, 64, 64) becomes the same as of the input image, and is then concatenated with the real image or generated image which shape is (3, 64, 64). This (4, 64, 64) tensor is the input for the discriminator.

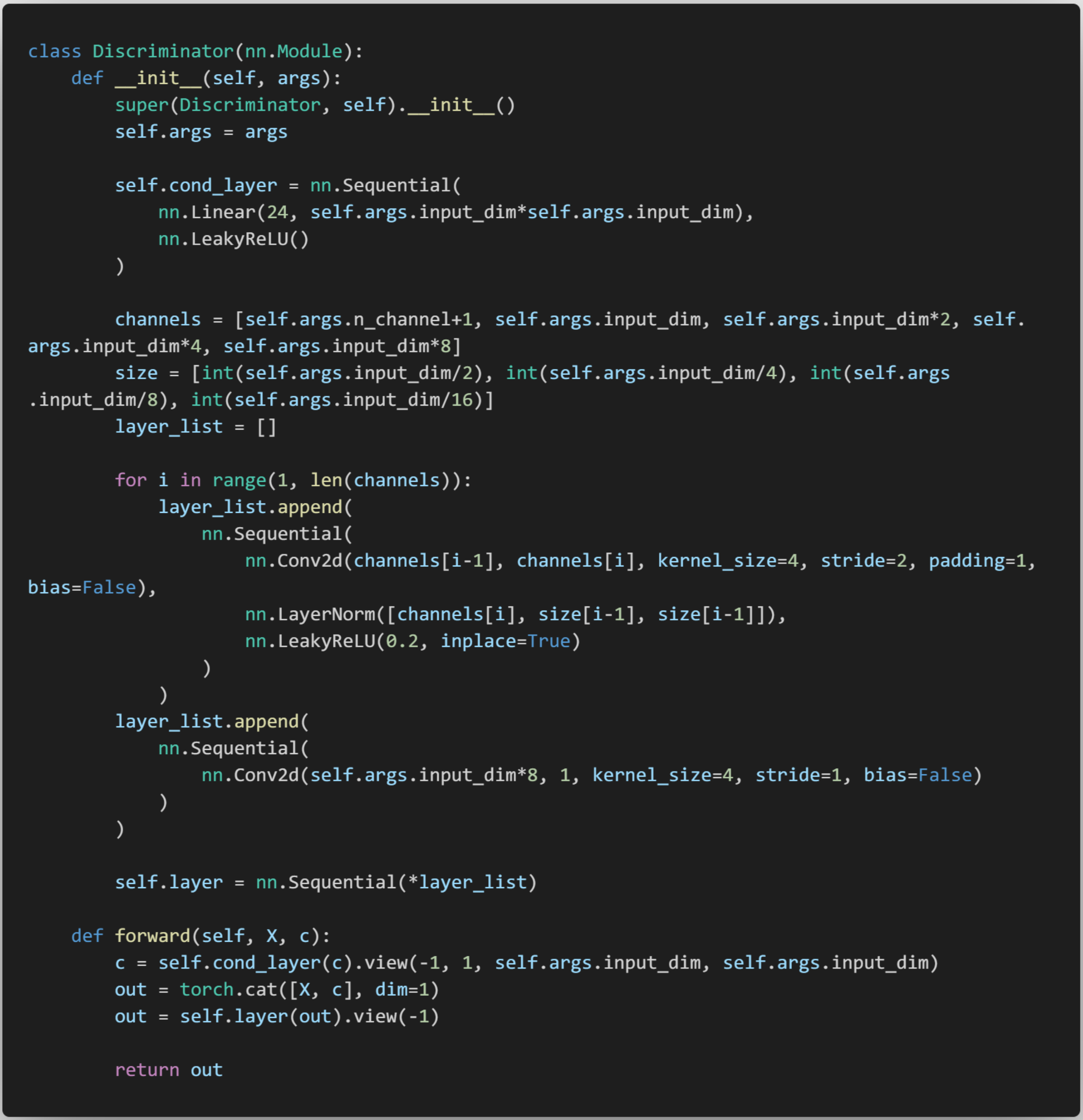


```python
class Generator(nn.Module):
    def __init__(self, args):
        super(Generator, self).__init__()
        self.args = args

        self.cond_layer = nn.Sequential(
            nn.Linear(24, self.args.c_dim),
            nn.ReLU()
        )

        channels = [self.args.z_dim+self.args.c_dim, self.args.input_dim*8, self.args
.input_dim*4, self.args.input_dim*2, self.args.input_dim]
        paddings = [0, 1, 1, 1]
        self.layer_list = []

        for i in range(1, len(channels)):
            self.layer_list.append(
                nn.Sequential(
                    nn.ConvTranspose2d(channels[i-1], channels[i], kernel_size=4, stride
=2, padding=paddings[i-1], bias=False),
                    nn.BatchNorm2d(channels[i]),
                    nn.ReLU(True)
                )
            )
        self.layer_list.append(
            nn.Sequential(
                nn.ConvTranspose2d(self.args.input_dim, self.args.n_channel, kernel_size
=4, stride=2, padding=1, bias=False),
                nn.Tanh()
            )
        )

        self.layer = nn.Sequential(*self.layer_list)

    def forward(self, z, c):
        z = z.view(-1, self.args.z_dim, 1, 1)
        c = self.cond_layer(c).view(-1, self.args.c_dim, 1, 1)
        out = torch.cat([z, c], dim=1)
        out = self.layer(out)

        return out
```



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

        self.cond_layer = nn.Sequential(
            nn.Linear(24, self.args.input_dim*self.args.input_dim),
            nn.LeakyReLU()
        )

        channels = [self.args.n_channel+1, self.args.input_dim, self.args.input_dim*2, self
.args.input_dim*4, self.args.input_dim*8]
        self.layer_list = []

        for i in range(1, len(channels)):
            self.layer_list.append(
                nn.Sequential(
                    nn.Conv2d(channels[i-1], channels[i], kernel_size=4, stride=2, padding=1, bias=
False),
                    nn.BatchNorm2d(channels[i]),
                    nn.LeakyReLU(0.2, inplace=True)
                )
            )
        self.layer_list.append(
            nn.Sequential(
            nn.Conv2d(self.args.input_dim*8, 1, kernel_size=4, stride=1, bias=False),
            nn.Sigmoid()
            )
        )

        self.layer = nn.Sequential(*self.layer_list)

    def forward(self, X, c):
        c = self.cond_layer(c).view(-1, 1, self.args.input_dim, self.args.input_dim)
        out = torch.cat([X, c], dim=1)
        out = self.layer(out).view(-1)

        return out
```

Note that I also train the model with **WGAN-GP** loss function, since the dicriminator (critic) in WGAN will try to approximately fit the EM distance, which is a regression task, so we have to **remove the sigmoid function** in the last layer. Because the gradient penalty is performed on each sample seperately, **batch normaliztion** in the discriminator **should be removed**, and using such as layer, weight or instance normalization.



```
class Discriminator(nn.Module):
    def __init__(self, args):
        super(Discriminator, self).__init__()
        self.args = args

        self.cond_layer = nn.Sequential(
            nn.Linear(24, self.args.input_dim*self.args.input_dim),
            nn.LeakyReLU()
        )

        channels = [self.args.n_channel+1, self.args.input_dim, self.args.input_dim*2, self.
args.input_dim*4, self.args.input_dim*8]
        size = [int(self.args.input_dim/2), int(self.args.input_dim/4), int(self.args
.input_dim/8), int(self.args.input_dim/16)]
        layer_list = []

        for i in range(1, len(channels)):
            layer_list.append(
                nn.Sequential(
                    nn.Conv2d(channels[i-1], channels[i], kernel_size=4, stride=2, padding=1,
bias=False),
                    nn.LayerNorm([channels[i], size[i-1], size[i-1]]),
                    nn.LeakyReLU(0.2, inplace=True)
                )
            )
        layer_list.append(
            nn.Sequential(
                nn.Conv2d(self.args.input_dim*8, 1, kernel_size=4, stride=1, bias=False)
            )
        )

        self.layer = nn.Sequential(*layer_list)

    def forward(self, X, c):
        c = self.cond_layer(c).view(-1, 1, self.args.input_dim, self.args.input_dim)
        out = torch.cat([X, c], dim=1)
        out = self.layer(out).view(-1)

        return out
```

2.3 Loss Functions

In this lab, I tried two types of loss functions. The introduction of the loss functions is in the following section.

2.3.1 GAN

The ordinary value function for training **GAN** is a minimax game.

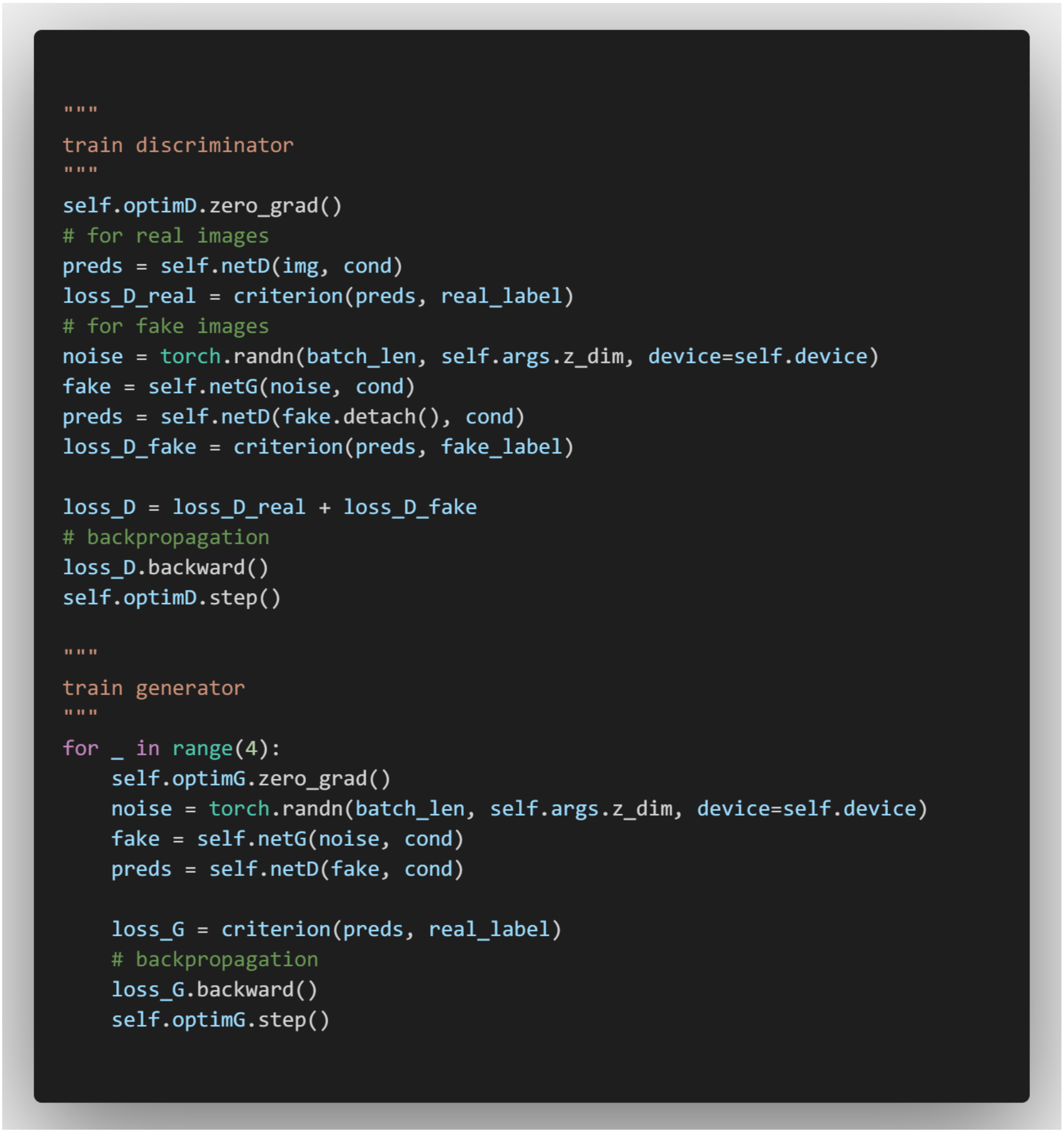
$$V(\theta^{(D)}, \theta^{(G)}) = \mathbb{E}_{x \sim p_{data}}[logD(x)] + \mathbb{E}_{z \sim p_z}[log(1 - D(G(z)))]$$

And the obejective is to find a generator,

$$\theta^{(G)*} = arg\underset{\theta^{(G)}}{min}\underset{\theta^{(D)}}{max}V(\theta^{(D)}, \theta^{(G)})$$

where D represents the discriminator with parameters $\theta^{(D)}$ and G represents the generator with parameters $\theta^{(G)}$. The goal of the discriminator is to maximize the function while the generator try to minimize it, that is, the generator is trying to generate images that are able to trick the discriminator so that the discriminator believes that the images is from real data.

My implementation code is as the figure shown below. I applied a trick to train GAN, which basically update the generator **four times** in each training iteration. The main purpose of it is **weakening the discriminator** so that the generator could be trained.



```
"""
train discriminator
"""
self.optimD.zero_grad()
# for real images
preds = self.netD(img, cond)
loss_D_real = criterion(preds, real_label)
# for fake images
noise = torch.randn(batch_len, self.args.z_dim, device=self.device)
fake = self.netG(noise, cond)
preds = self.netD(fake.detach(), cond)
loss_D_fake = criterion(preds, fake_label)

loss_D = loss_D_real + loss_D_fake
# backpropagation
loss_D.backward()
self.optimD.step()

"""
train generator
"""
for _ in range(4):
    self.optimG.zero_grad()
    noise = torch.randn(batch_len, self.args.z_dim, device=self.device)
    fake = self.netG(noise, cond)
    preds = self.netD(fake, cond)

    loss_G = criterion(preds, real_label)
    # backpropagation
    loss_G.backward()
    self.optimG.step()
```

2.3.2 WGAN-GP

The **WGAN** was introduced to improve the stability of the training process, and it points out the problem in the ordinary GAN, which would lead to gradient vanishing. WGAN adopts **earth mover distance** (or **Wassertein distance**) as the optimization goal and solve the problem. The objective function is formulated as follow:

$$W(P_r, P_\theta) = \underset{\{f_w\}_{w \in W}}{max} \mathbb{E}_{x \sim P_r} f_w(x) - \mathbb{E}_{z \sim p(z)} f_w(g_\theta(z))$$

Assuming $\{f_w\}_{w \in W}$ is a family of 1-Lipschitz functions that represents the critic (discriminator), the final objective becomes

$$\theta^* = arg\underset{\theta}{min}\underset{w \in W}{max} \mathbb{E}_{x \sim P_r} f_w(x) - \mathbb{E}_{z \sim p(z)} f_w(g_\theta(z))$$

The critic (discirminator) aims to maximize the distance while the generator try to minimize it. In WGAN algorithm, the critic will be updated **five times** in each training iteration.

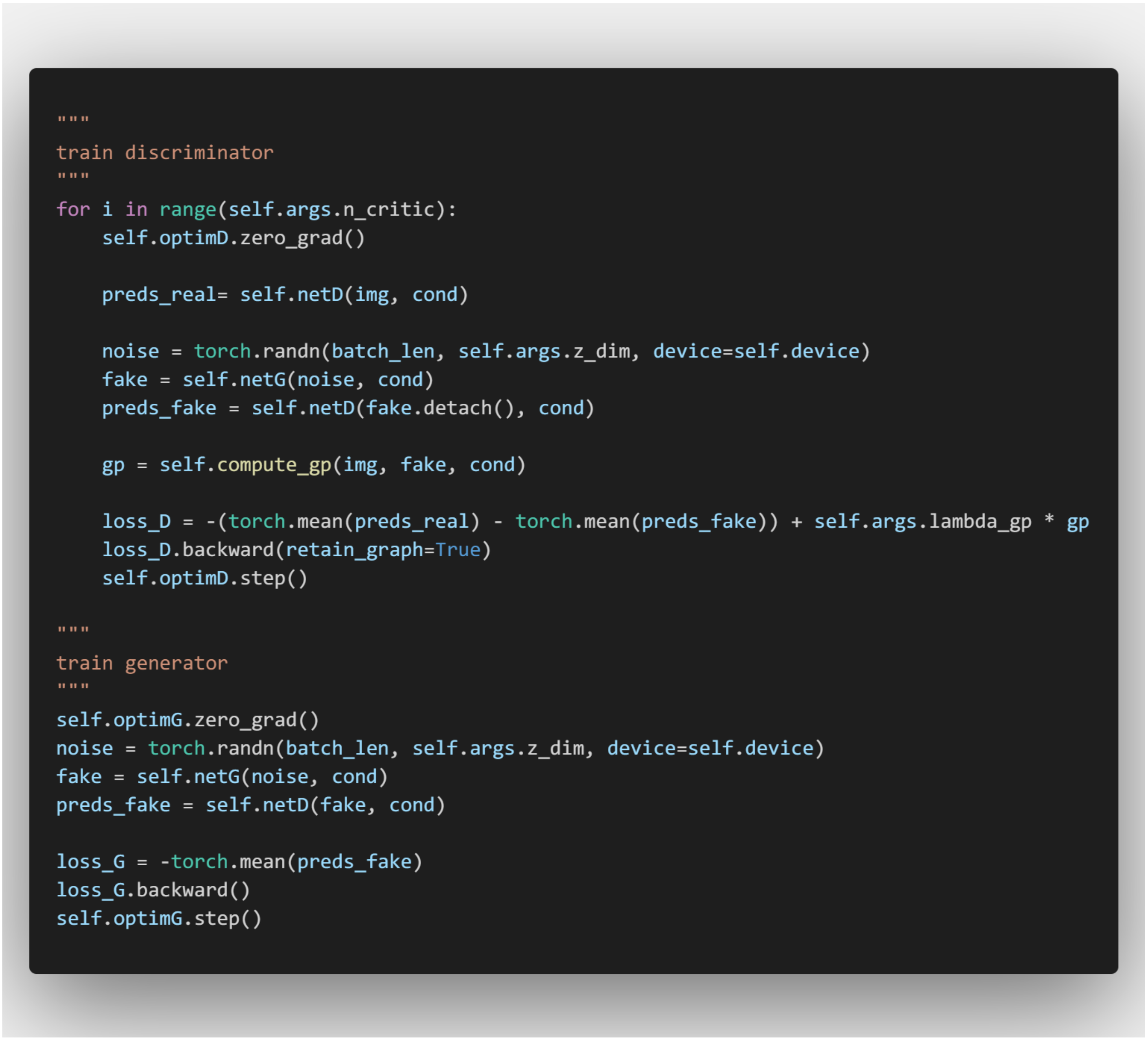
However, there are still a problem in WGAN, and that is the restriction of Lipschitz on critic is implemented by **weight clipping**, which may, for one thing, result in **gradient vanishing** or **gradient exploding**, for another thing, the value of network weights are all near the **extremum** in the weight clipping interval. With this problem, the critic's capability may be limited.

WGAN-GP is proposed to solve the problem, which replace the weight clipping with gradient penalty, the obejective function could be formulated as follow,

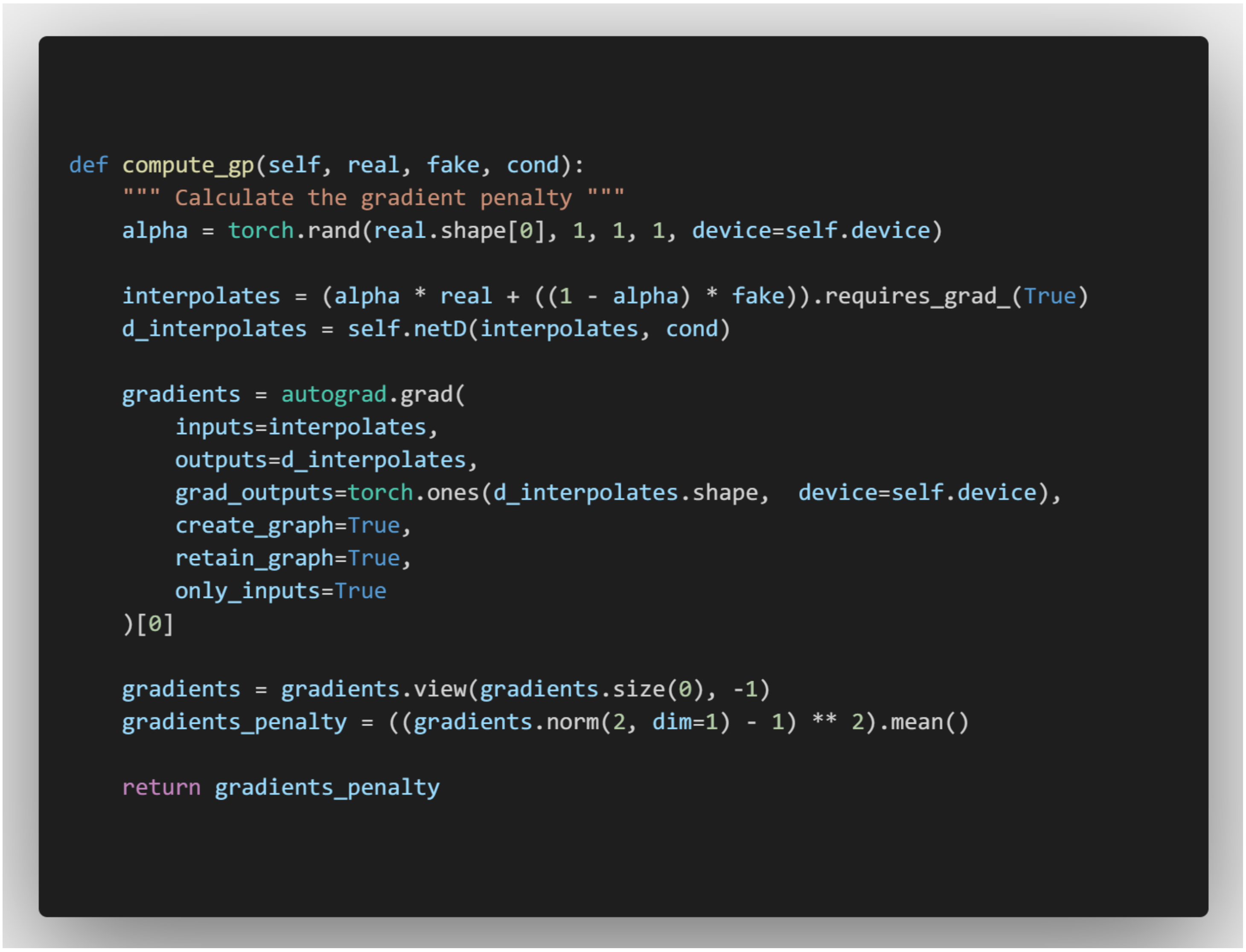
$$W(P_r, P_\theta) = \mathbb{E}_{x \sim P_r} f_w(x) - \mathbb{E}_{z \sim p(z)} + \lambda \mathbb{E}_{x \sim P_{\hat{x}}} [||\nabla_x f_w(x)||_2 - 1]^2$$

where
$\epsilon \sim Uniform[0, 1]$, $x_r \sim P_r$, $x_\theta \sim P_\theta$,

$$\hat{x} = \epsilon x_r + (1 - \epsilon) x_\theta$$



```
"""
train discriminator
"""
for i in range(self.args.n_critic):
    self.optimD.zero_grad()

    preds_real= self.netD(img, cond)

    noise = torch.randn(batch_len, self.args.z_dim, device=self.device)
    fake = self.netG(noise, cond)
    preds_fake = self.netD(fake.detach(), cond)

    gp = self.compute_gp(img, fake, cond)

    loss_D = -(torch.mean(preds_real) - torch.mean(preds_fake)) + self.args.lambda_gp * gp
    loss_D.backward(retain_graph=True)
    self.optimD.step()

"""
train generator
"""
self.optimG.zero_grad()
noise = torch.randn(batch_len, self.args.z_dim, device=self.device)
fake = self.netG(noise, cond)
preds_fake = self.netD(fake, cond)

loss_G = -torch.mean(preds_fake)
loss_G.backward()
self.optimG.step()
```



```python
def compute_gp(self, real, fake, cond):
    """ Calculate the gradient penalty """
    alpha = torch.rand(real.shape[0], 1, 1, 1, device=self.device)

    interpolates = (alpha * real + ((1 - alpha) * fake)).requires_grad_(True)
    d_interpolates = self.netD(interpolates, cond)

    gradients = autograd.grad(
        inputs=interpolates,
        outputs=d_interpolates,
        grad_outputs=torch.ones(d_interpolates.shape,  device=self.device),
        create_graph=True,
        retain_graph=True,
        only_inputs=True
    )[0]

    gradients = gradients.view(gradients.size(0), -1)
    gradients_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()

    return gradients_penalty
```

2.4 Hyperparameters

Loss Function	Epoch	Learning Rate	Batch Size	Optimizer	λ
GAN	300	0.0002	128	Adam, betas=(0.5, 0.99)	-
WGAN-GP	300	0.0002	128	Adam, betas=(0.5, 0.99)	10

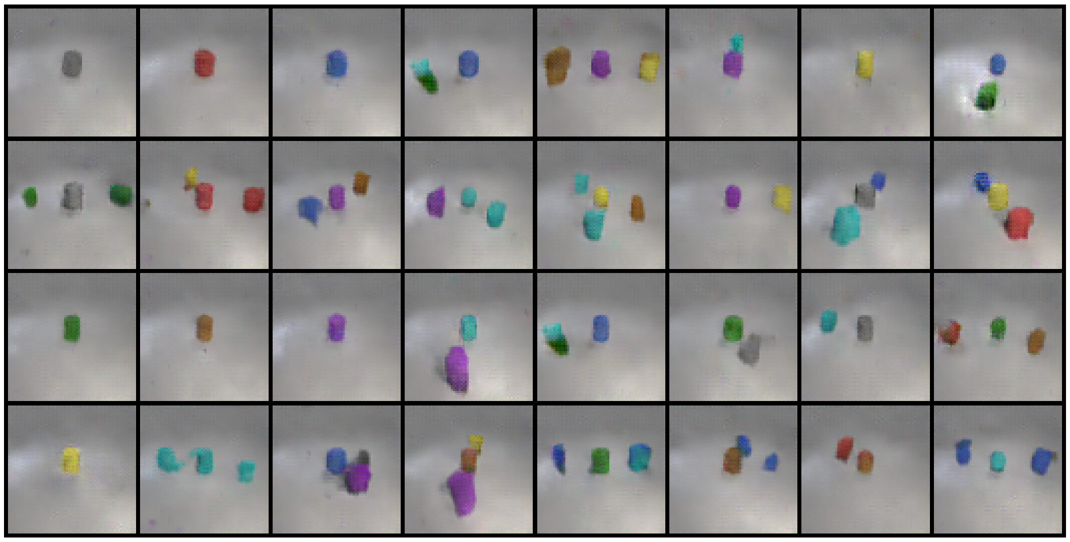
3. Results and Discussion

3.1 Generated Images

3.1.1 test.json

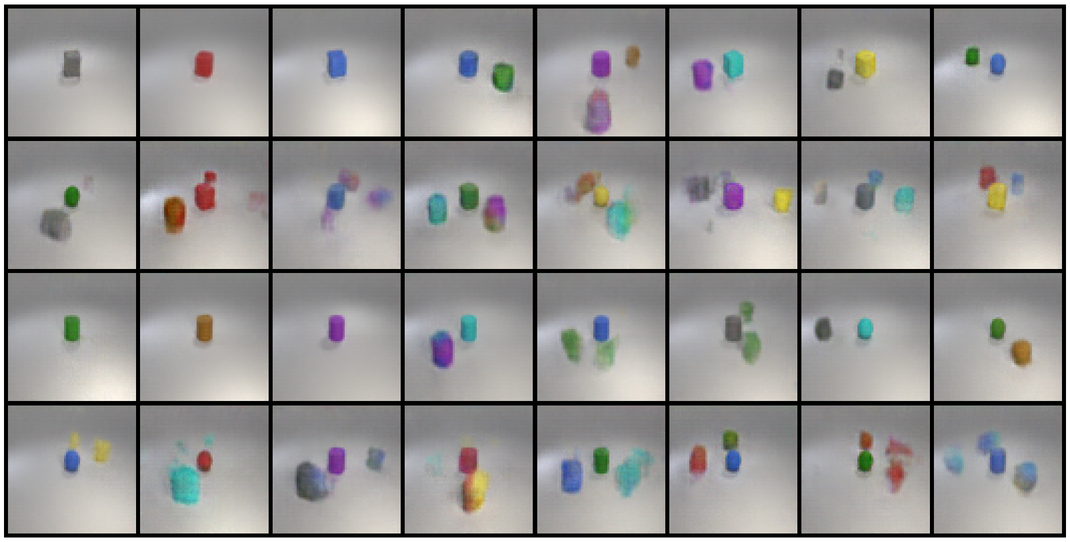
- GAN

  The accuracy is **69.44%**.



- WGAN-GP

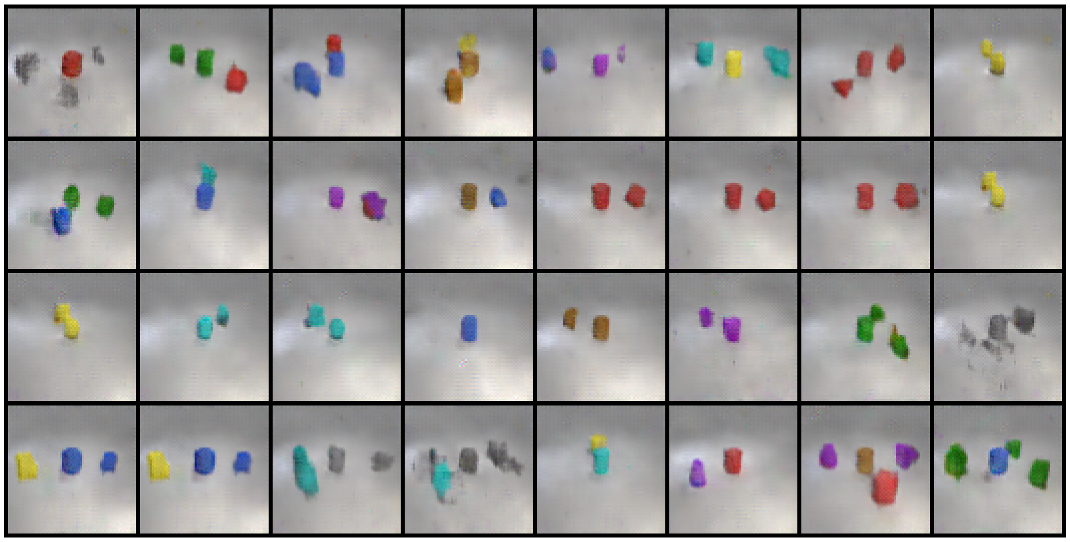
  The accuracy is **70.83%**.



3.1.2 new_test.json

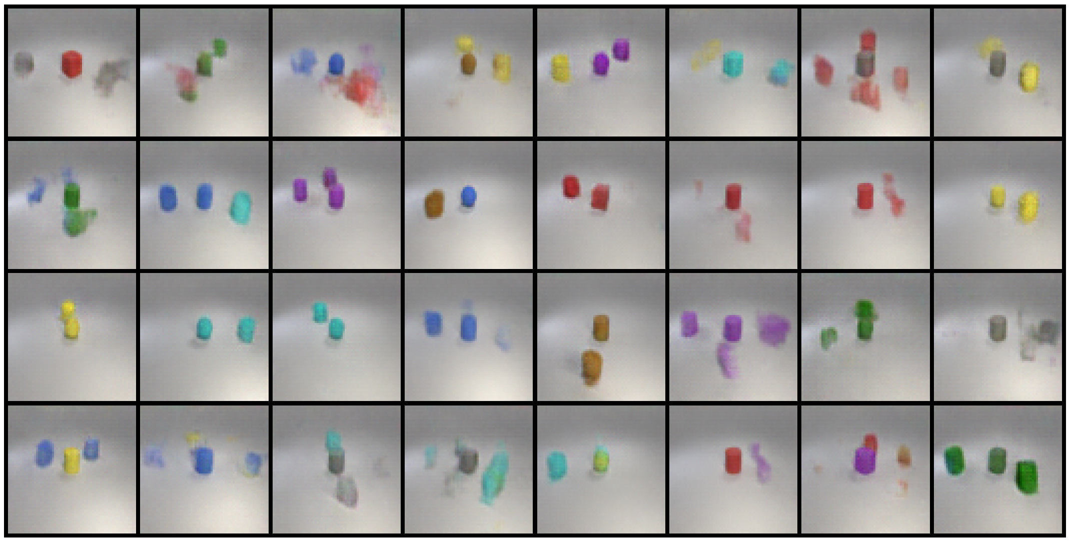
- GAN

  The accuracy is **67.86%**.



- WGAN-GP

  The accuracy is **75%**.



3.2 Disccusion

During the testing, I found that **WGAN-GP is very sensitive to the random noise**, that is, different random noise can significantly affect the accuracy of the generated images while GAN is relatively stable. Although the images generated by WGAN-GP can obtain higher accuracy, but it looks like there are many **ghosting** in the images.

I also found that if I **didn't feed the condion into the discriminator**, the accuracy would become extremely terrible. The reason of it may due to the **generator couldn't learn how to recover the conditions** since the discriminator don't tell whether the generated images match the specific conditions