Summer 2022 Deep Learning

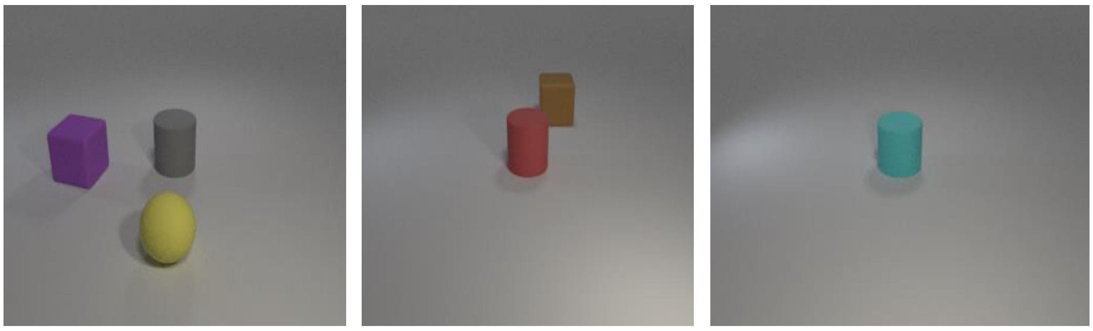
Report of Lab #5

Student name: 劉子齊 Jonathan
Student ID: 311605004

Part 1: Introduction

In Lab5, our goal is to implement a conditional GAN to generate synthetic images through the given multi-label conditions. Also, we would have to design our own generator and discriminator, and choose the loss function and the optimizer that fit our model the best.

For the dataset in this lab, we got over 18, 000 images with corresponding labels for the training data, which was recorded in the "train.json". Each training image got 1 to 3 objects in it. The objects in the image could vary in 3 different shapes and 8 different colors. The following figures are how the input data looked like.

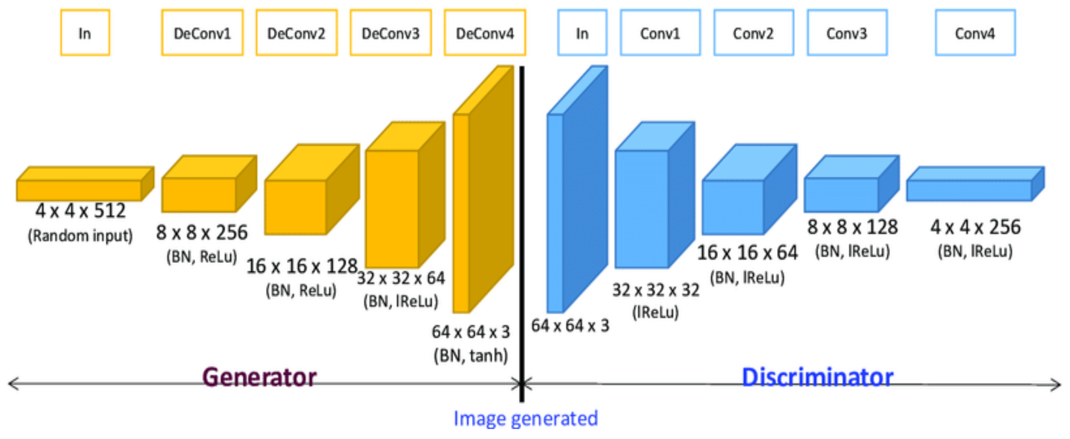


With the images and labels fed into the discriminator we designed and trained, the discriminator should be capable to classify the objects and the colors of the objects in the given image. Besides, the generator we constructed should also be capable to generate the image corresponding to the object condition we gave through the initialized noise we provided.

Part 2: Implementation Details

Part 2-A: Architecture of my GAN

The GAN architecture I chose for this lab is the conditional GAN. Besides, I designed the generator and the discriminator as the architecture as DCGAN, which was shown in the following figure.

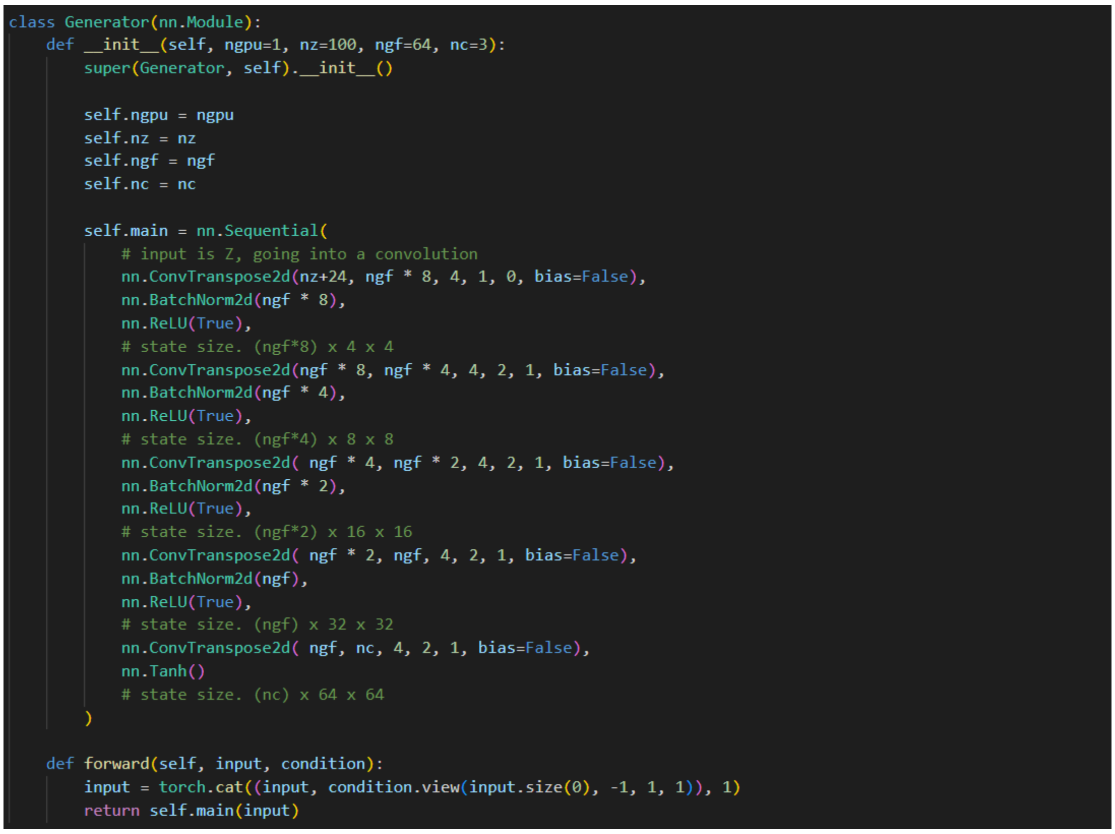


For conditional GANs, we would concatenate the condition data with the images and the latent variables in the form of one-hot vectors. In the “Conditional DCGAN”, the generator was constructed by Deconvolution layers, Batch Normalize layers, and ReLU layers, just as the architecture of the DCGAN. Besides, the conditional data would directly be concatenated into the latent variable in the generator.

Then we would come to the discriminator, which was constructed by Convolution layers, Batch Normalize layers, and Leaky ReLU layers. The conditional data would be passed into a linear layer here in the output size of 64 x 64, which made the condition data turned into an input image sized array. Afterwards, we would concatenate the condition array after the image.

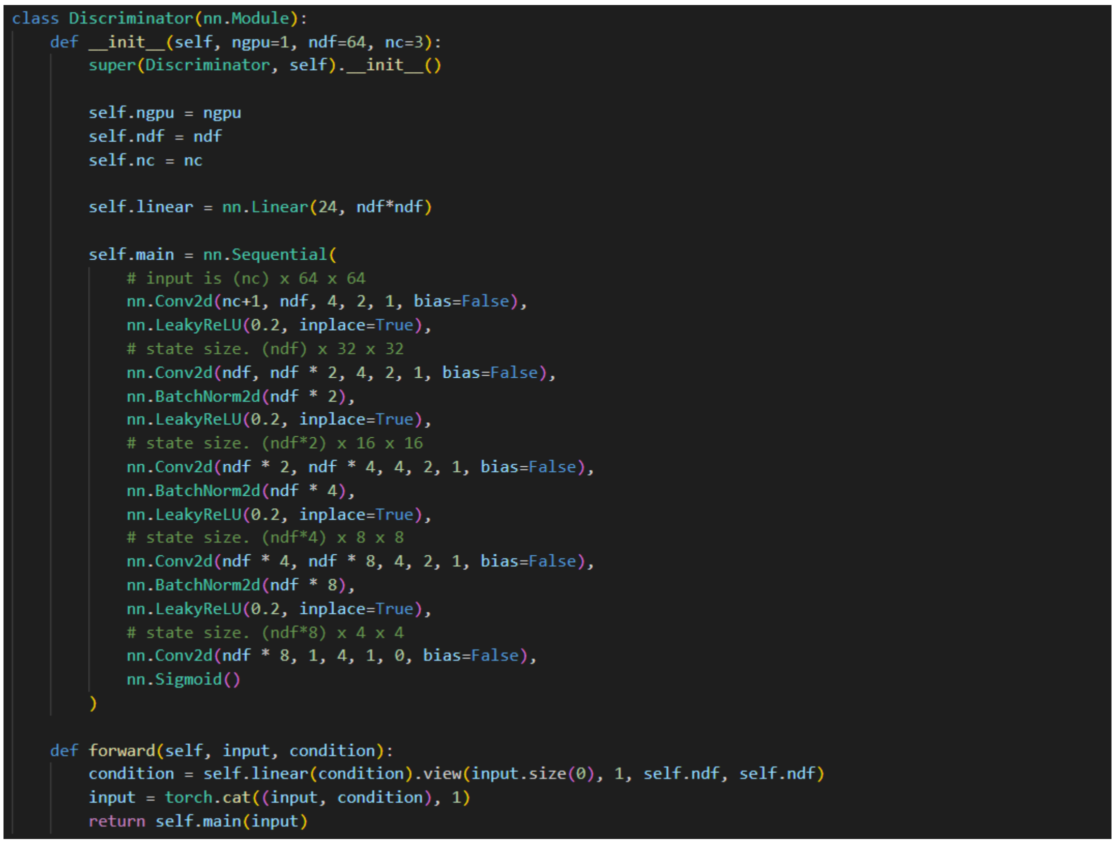
Hence, this made the architecture of our “Conditional DCGAN” different from the “DCGAN” architecture in the figure above, we would have 4 channels for the image. The following figures were how I implemented the generator and the discriminator of the “Conditional DCGAN”:

➢ Generator (Conditional DCGAN)



```python
class Generator(nn.Module):
    def __init__(self, ngpu=1, nz=100, ngf=64, nc=3):
        super(Generator, self).__init__()

        self.ngpu = ngpu
        self.nz = nz
        self.ngf = ngf
        self.nc = nc

        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(nz+24, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input, condition):
        input = torch.cat((input, condition.view(input.size(0), -1, 1, 1)), 1)
        return self.main(input)
```

➢ Discriminator (Conditional DCGAN)



```python
class Discriminator(nn.Module):
    def __init__(self, ngpu=1, ndf=64, nc=3):
        super(Discriminator, self).__init__()

        self.ngpu = ngpu
        self.ndf = ndf
        self.nc = nc

        self.linear = nn.Linear(24, ndf*ndf)

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

    def forward(self, input, condition):
        condition = self.linear(condition).view(input.size(0), 1, self.ndf, self.ndf)
        input = torch.cat((input, condition), 1)
        return self.main(input)
```

Part 2-B: Training Loss Function

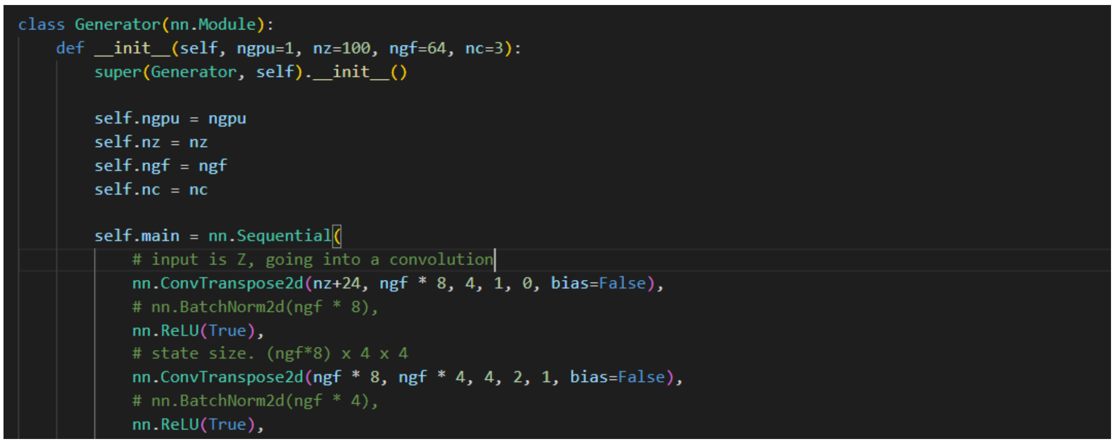
Initially, I set the loss function to BCE Loss. Then I found the performance of the model was just not even good, and this confused me. After some research, I found Wasserstein GAN might be able to solve my problem.

Since in the training process of the conditional DCGAN, we could easily find that there was an imbalance between the generator and the discriminator. When the discriminator was getting too weak or too strong, it wouldn't give the generator useful feedback to make some improvements. In this case, further training wouldn't necessarily make the conditional DCGAN model better.

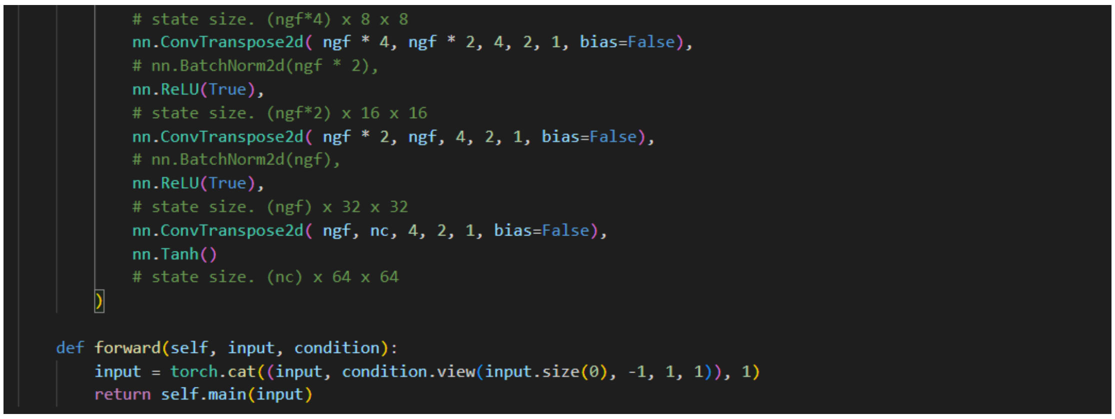
However, this problem could be solved through the Wasserstein loss of the WGAN. When Wasserstein loss is adopted, the balance of generator and discriminator was no longer that important. Thanks to the characteristic of WGAN, which containing linear gradients that were almost 100% continuous and differentiable, solved the gradient vanishing problem in the training process of traditional GANs.

Besides, I applied the Gradient Penalty of WGAN to enforce the Lipschitz constraint for the discriminator. There was actually another method called weight clipping. However, even the author of the paper of WGAN said that large clipping parameter could lead to slow training and prevent the discriminator from reaching optimality. Also, a clipping too small could easily lead to vanishing gradients, the exact problem WGAN was proposed to solve. Hence, I chose to apply gradient penalty in the WGAN. Below were the figures of how I implemented WGAN and gradient penalty.

- Generator (Conditional DCGAN + WGAN + Gradient Penalty)

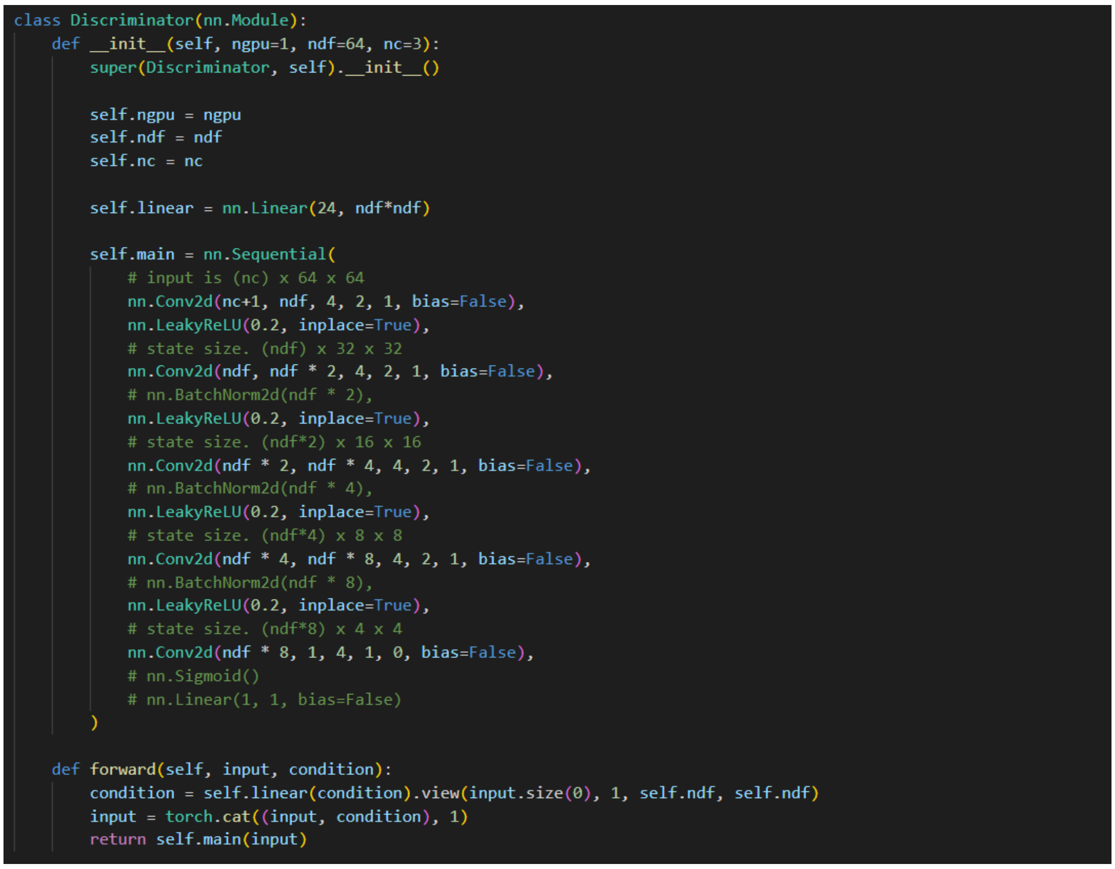


```
class Generator(nn.Module):
    def __init__(self, ngpu=1, nz=100, ngf=64, nc=3):
        super(Generator, self).__init__()

        self.ngpu = ngpu
        self.nz = nz
        self.ngf = ngf
        self.nc = nc

        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(nz+24, ngf * 8, 4, 1, 0, bias=False),
            # nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
```



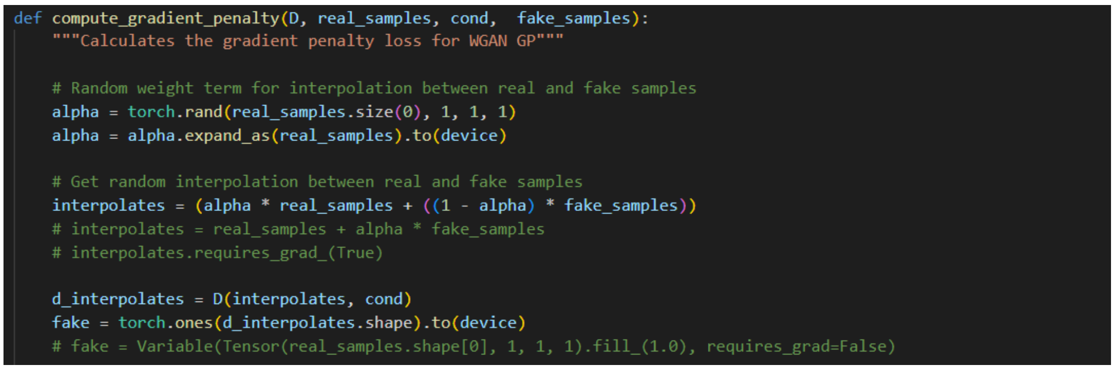
```python
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input, condition):
        input = torch.cat((input, condition.view(input.size(0), -1, 1, 1)), 1)
        return self.main(input)
```

➢ Discriminator (Conditional DCGAN + WGAN + Gradient Penalty)

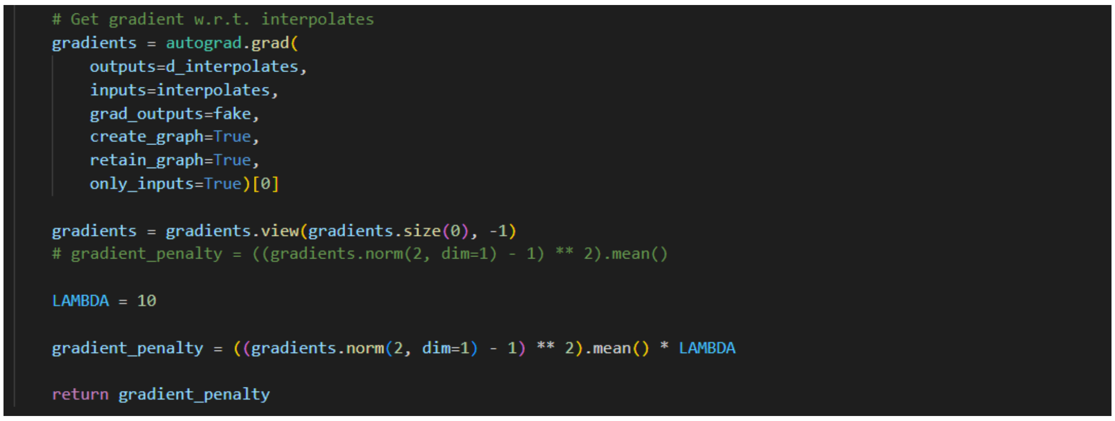


```python
class Discriminator(nn.Module):
    def __init__(self, ngpu=1, ndf=64, nc=3):
        super(Discriminator, self).__init__()

        self.ngpu = ngpu
        self.ndf = ndf
        self.nc = nc

        self.linear = nn.Linear(24, ndf*ndf)

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

    def forward(self, input, condition):
        condition = self.linear(condition).view(input.size(0), 1, self.ndf, self.ndf)
        input = torch.cat((input, condition), 1)
        return self.main(input)
```

➢ Gradient Penalty (Conditional DCGAN + WGAN + Gradient Penalty)

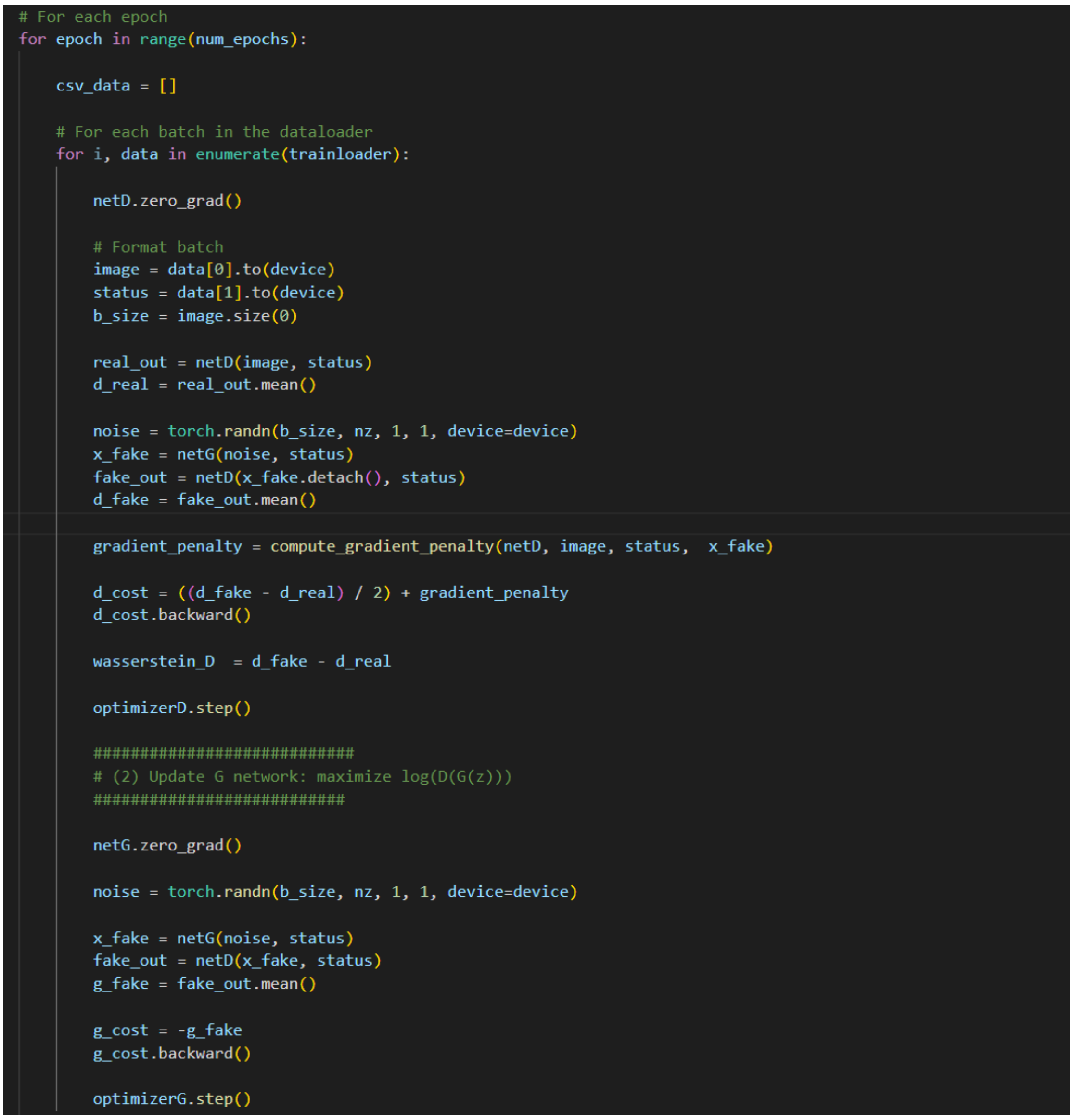


```python
def compute_gradient_penalty(D, real_samples, cond,  fake_samples):
    """Calculates the gradient penalty loss for WGAN GP"""

    # Random weight term for interpolation between real and fake samples
    alpha = torch.rand(real_samples.size(0), 1, 1, 1)
    alpha = alpha.expand_as(real_samples).to(device)

    # Get random interpolation between real and fake samples
    interpolates = (alpha * real_samples + ((1 - alpha) * fake_samples))
    # interpolates = real_samples + alpha * fake_samples
    # interpolates.requires_grad_(True)

    d_interpolates = D(interpolates, cond)
    fake = torch.ones(d_interpolates.shape).to(device)
    # fake = Variable(Tensor(real_samples.shape[0], 1, 1, 1).fill_(1.0), requires_grad=False)
```



```python
    # Get gradient w.r.t. interpolates
    gradients = autograd.grad(
        outputs=d_interpolates,
        inputs=interpolates,
        grad_outputs=fake,
        create_graph=True,
        retain_graph=True,
        only_inputs=True)[0]

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

    LAMBDA = 10

    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * LAMBDA

    return gradient_penalty
```

➢ Training Function (Conditional DCGAN + WGAN + Gradient Penalty)

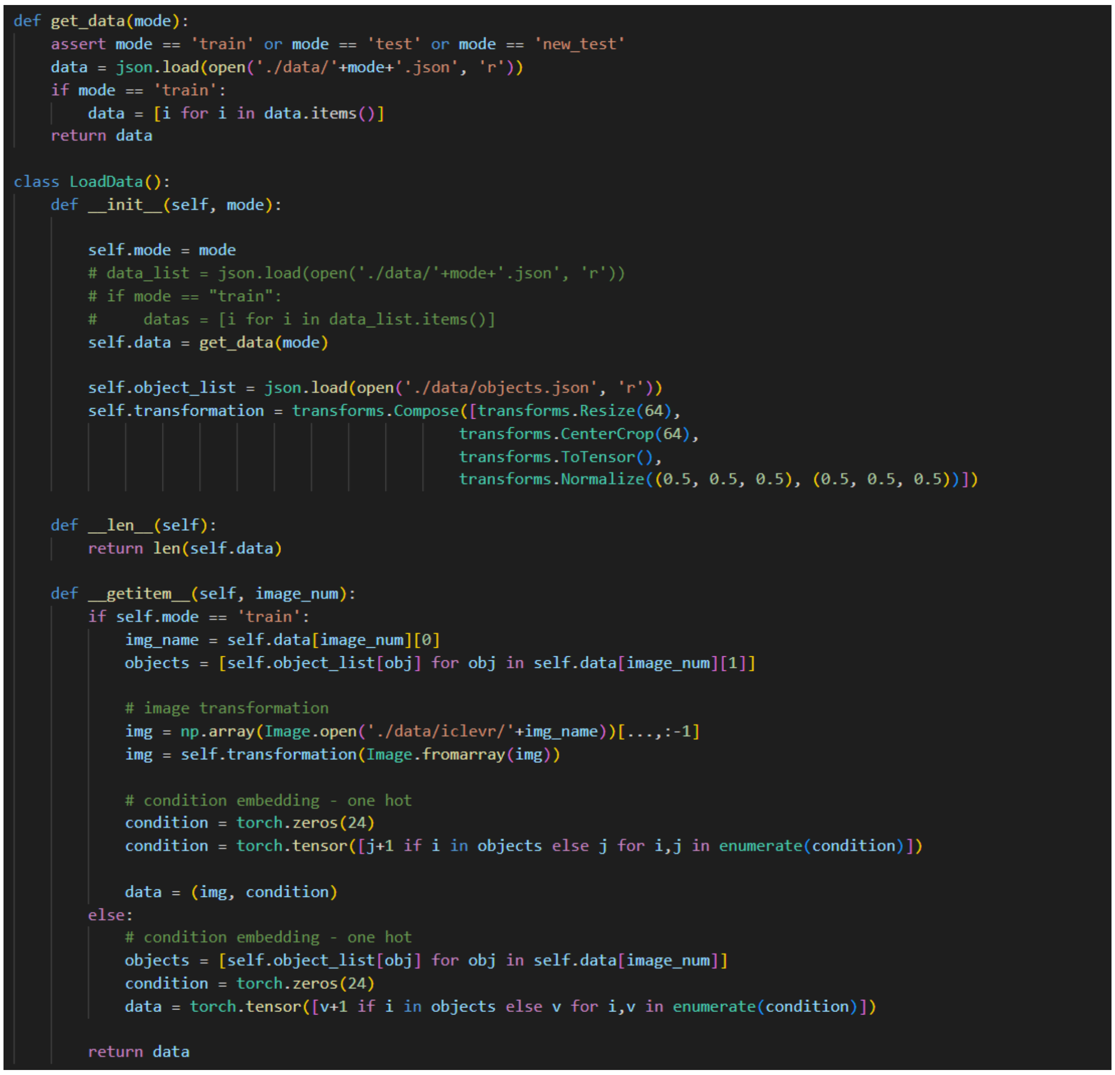


```python
# For each epoch
for epoch in range(num_epochs):

    csv_data = []

    # For each batch in the dataloader
    for i, data in enumerate(trainloader):

        netD.zero_grad()

        # Format batch
        image = data[0].to(device)
        status = data[1].to(device)
        b_size = image.size(0)

        real_out = netD(image, status)
        d_real = real_out.mean()

        noise = torch.randn(b_size, nz, 1, 1, device=device)
        x_fake = netG(noise, status)
        fake_out = netD(x_fake.detach(), status)
        d_fake = fake_out.mean()

        gradient_penalty = compute_gradient_penalty(netD, image, status,  x_fake)

        d_cost = ((d_fake - d_real) / 2) + gradient_penalty
        d_cost.backward()

        wasserstein_D  = d_fake - d_real

        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################

        netG.zero_grad()

        noise = torch.randn(b_size, nz, 1, 1, device=device)

        x_fake = netG(noise, status)
        fake_out = netD(x_fake, status)
        g_fake = fake_out.mean()

        g_cost = -g_fake
        g_cost.backward()

        optimizerG.step()
```

Part 2-C: Training Loss Function

Below is the implementation of the data loading function. For the transformation of the input images, I first resize and crop the image to the size of 64 x 64, then I did normalization on the image with the mean of 0.5 and the std of 0.5 on all values.

Besides, different from DCGAN, we were using conditional DCGAN, so we would also need to get the condition data here. To turn the original condition into a one-hot vector, I first declared a torch vector with 24 elements, since there’s 24 kind of objects in total. Then I input the corresponding condition to the condition tensor through enumerating all the conditions.



```python
def get_data(mode):
    assert mode == 'train' or mode == 'test' or mode == 'new_test'
    data = json.load(open('./data/'+mode+'.json', 'r'))
    if mode == 'train':
        data = [i for i in data.items()]
    return data

class LoadData():
    def __init__(self, mode):

        self.mode = mode
        # data_list = json.load(open('./data/'+mode+'.json', 'r'))
        # if mode == "train":
        #     datas = [i for i in data_list.items()]
        self.data = get_data(mode)

        self.object_list = json.load(open('./data/objects.json', 'r'))
        self.transformation = transforms.Compose([transforms.Resize(64),
                                                   transforms.CenterCrop(64),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

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

    def __getitem__(self, image_num):
        if self.mode == 'train':
            img_name = self.data[image_num][0]
            objects = [self.object_list[obj] for obj in self.data[image_num][1]]

            # image transformation
            img = np.array(Image.open('./data/iclevr/'+img_name))[...,:-1]
            img = self.transformation(Image.fromarray(img))

            # condition embedding - one hot
            condition = torch.zeros(24)
            condition = torch.tensor([j+1 if i in objects else j for i,j in enumerate(condition)])

            data = (img, condition)
        else:
            # condition embedding - one hot
            objects = [self.object_list[obj] for obj in self.data[image_num]]
            condition = torch.zeros(24)
            data = torch.tensor([v+1 if i in objects else v for i,v in enumerate(condition)])

        return data
```

Part 2-D: Hyperparameters

Below are the hyperparameters I adopted to the experience that I got the best result:

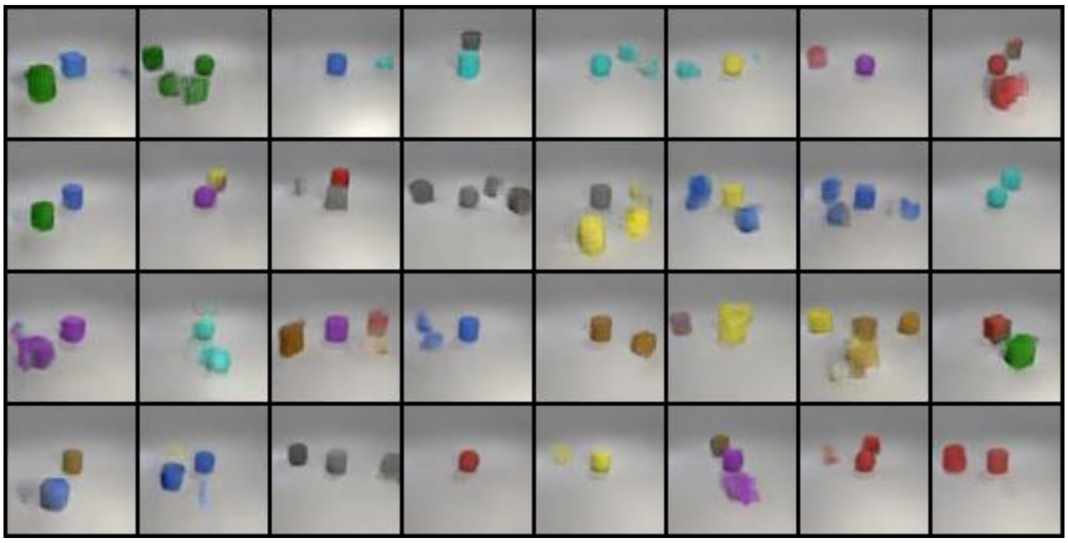
- Num Epochs = 500
- Generator Learning Rate = 0.0002
- Discriminator Learning Rate = 0.0002
- Batch Size = 128
- Nc = 3 (Number of Channels)
- Nz = 100 (Size of z latent vector)
- Ngf = 64 (Size of feature maps in generator)
- Ndf = 64 (Size of feature maps in discriminator)
- Optimizer = torch.optim. RMSprop ()
- Ngpu = 1 (Number of GPU)

Part 3: Experimental Results & Discussions

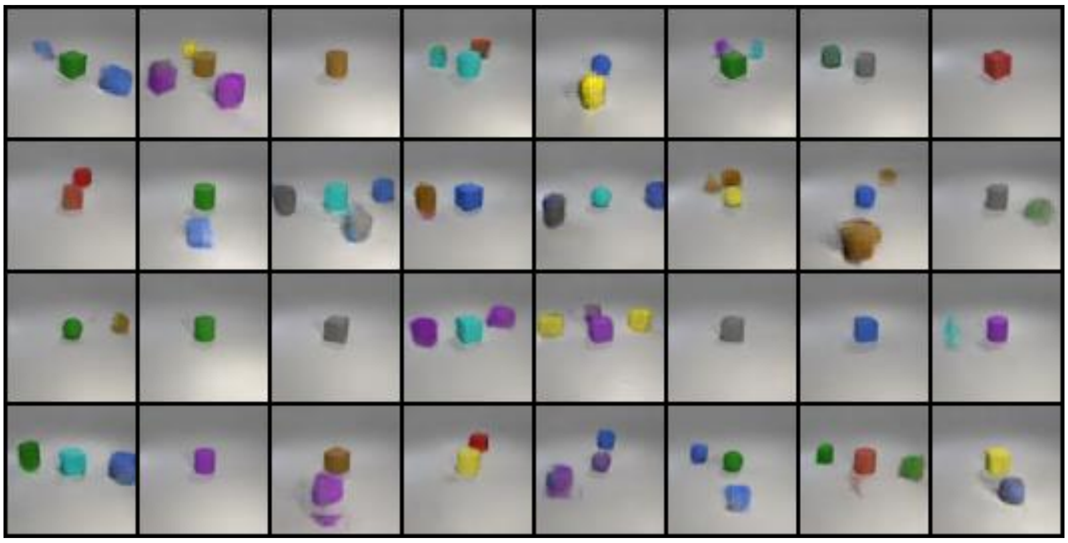
Part 3-A: Best Results

Below is the best result of all the experiments I had conducted, this experiment followed the hyperparameter settings mentioned above. Besides, this experiment was also done in WGAN with gradient penalty which was based on the architecture of conditional DCGAN.

- The synthetic images of "new_test.json" with the accuracy of 74.2381%



➢ The synthetic images of "test.json" with the accuracy of 70.0556%



Part 3-B: Further Discussions

As I mentioned before, I first tried on the purely conditional DCGAN, which without WGAN and gradient penalty, that took BCE loss as its loss function. This turned out a huge gap between the loss of generator and discriminator, which the loss of the discriminator was far lower.

So, then I lower the learning rate of the learning rate of the discriminator from 0.0002 to 0.0001, and kept the learning rate of the generator at 0.0002, trying to solve this problem through different learning rate. Although the model did perform better, the highest accuracy obtained was still only 55.9524%, which was actually kind of poor.

After applying WGAN and gradient penalty on conditional DCGAN, the cost of generator and the discriminator were a lot more stable. Also, by changing the optimizer from Adam to RMSprop we not only get a better and earlier convergence, but also get a better result with the accuracy of 74.2381%.

As a conclusion, we could tell that the imbalance of generator and the discriminator effected the model a lot. Through applying WGAN to the original conditional DCGAN structure, we solved the imbalance problem to some extent, and made a great improvement on the performance of the model from the accuracy of 55.9524% to 74.2381%.