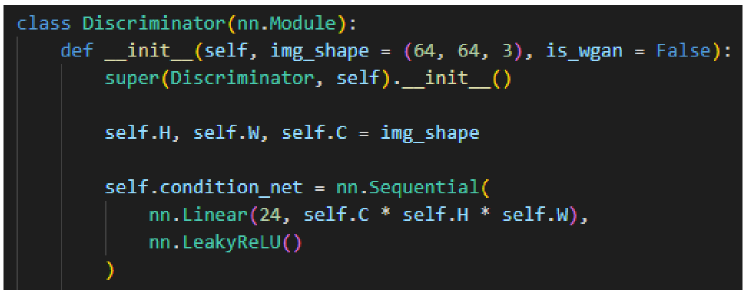
● Introduction

In this lab, I implement the DCGAN by using pytorch to generate different kinds of image according to multi-label conditions. For the loss function, I tried the GAN and WGAN approaches.

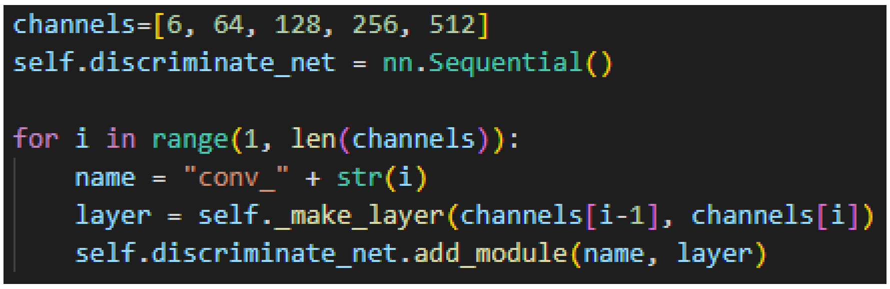
● Implementation Details

Discriminator:

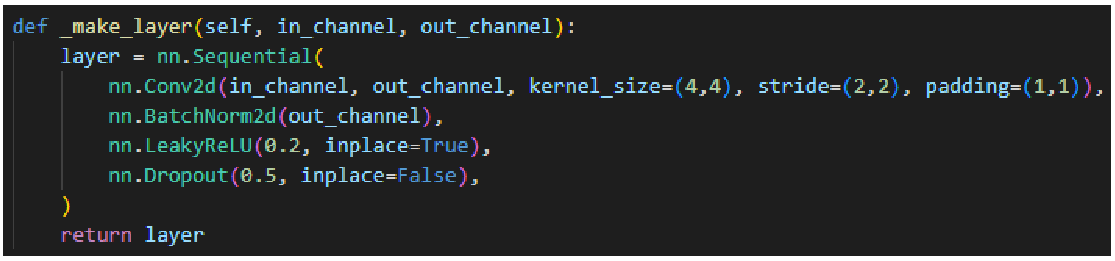


```
class Discriminator(nn.Module):
    def __init__(self, img_shape = (64, 64, 3), is_wgan = False):
        super(Discriminator, self).__init__()

        self.H, self.W, self.C = img_shape

        self.condition_net = nn.Sequential(
            nn.Linear(24, self.C * self.H * self.W),
            nn.LeakyReLU()
        )
```

First, I would save the image size to discriminate. And for the condition, I would use a fully-connected layer to make its size equal to the input image to cancat it to the input image.

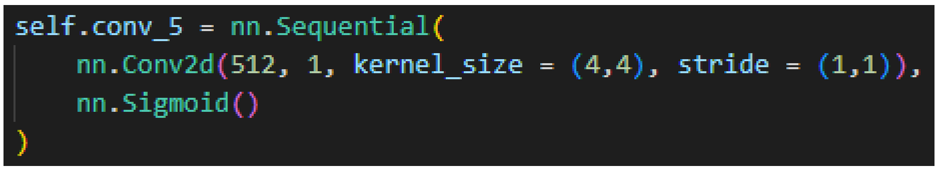


```
channels=[6, 64, 128, 256, 512]
self.discriminate_net = nn.Sequential()

for i in range(1, len(channels)):
    name = "conv_" + str(i)
    layer = self._make_layer(channels[i-1], channels[i])
    self.discriminate_net.add_module(name, layer)
```



```
def _make_layer(self, in_channel, out_channel):
    layer = nn.Sequential(
        nn.Conv2d(in_channel, out_channel, kernel_size=(4,4), stride=(2,2), padding=(1,1)),
        nn.BatchNorm2d(out_channel),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Dropout(0.5, inplace=False),
    )
    return layer
```

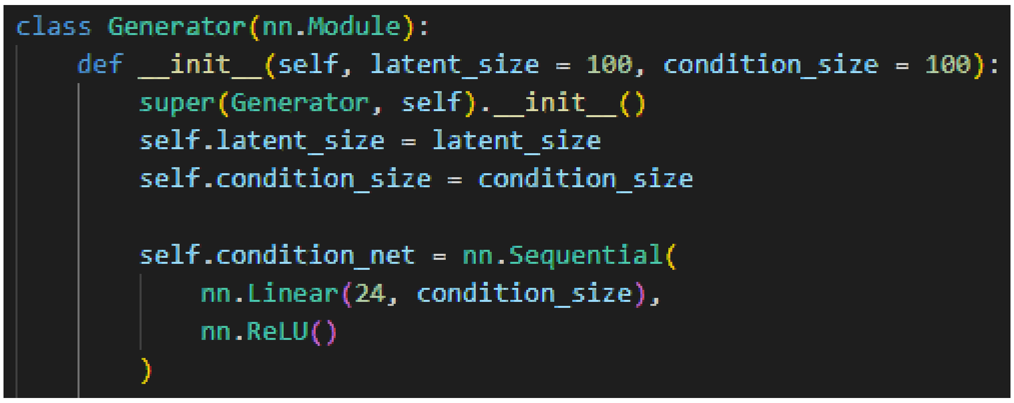
Then, I would use a deep convolution network to extract features. The image size would change from (64,64,6) to (16,16,64)…to (1,1,512).



```
self.conv_5 = nn.Sequential(
    nn.Conv2d(512, 1, kernel_size = (4,4), stride = (1,1)),
    nn.Sigmoid()
)
```

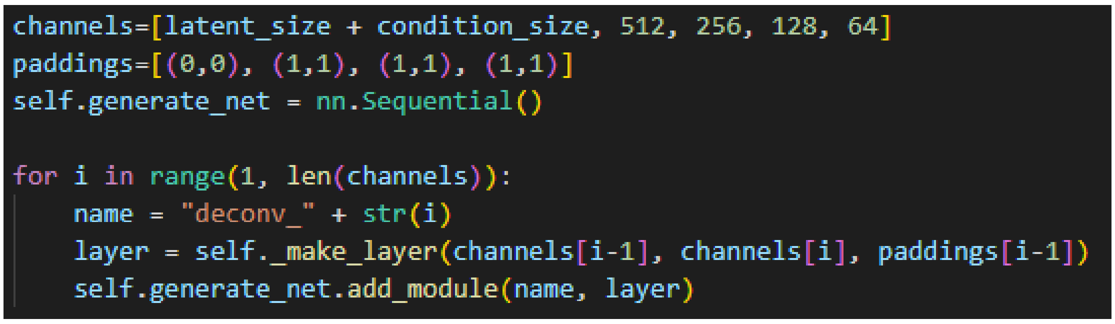
Finally, use a sigmoid activation function to output the probity of real image.

Generator:

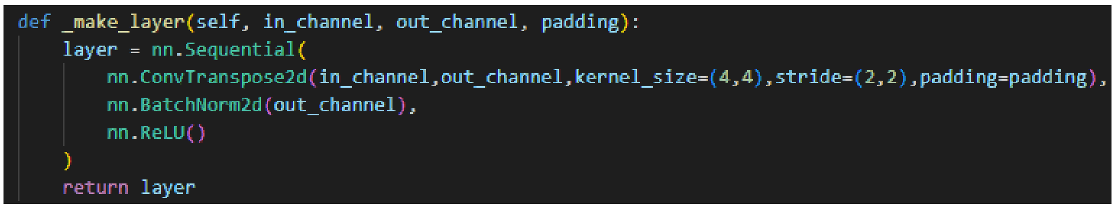


```python
class Generator(nn.Module):
    def __init__(self, latent_size = 100, condition_size = 100):
        super(Generator, self).__init__()
        self.latent_size = latent_size
        self.condition_size = condition_size

        self.condition_net = nn.Sequential(
            nn.Linear(24, condition_size),
            nn.ReLU()
        )
```

First, save the latent size and condition size. Then, use a fully-connected layer to make it bigger.

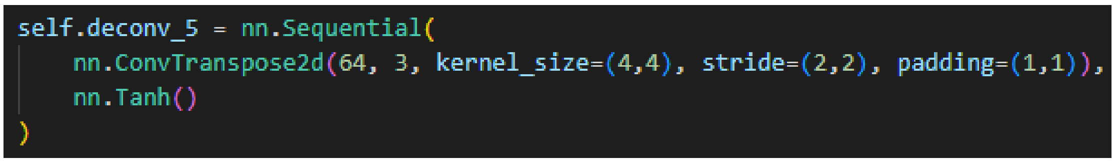


```python
channels=[latent_size + condition_size, 512, 256, 128, 64]
paddings=[(0,0), (1,1), (1,1), (1,1)]
self.generate_net = nn.Sequential()

for i in range(1, len(channels)):
    name = "deconv_" + str(i)
    layer = self._make_layer(channels[i-1], channels[i], paddings[i-1])
    self.generate_net.add_module(name, layer)
```



```python
def _make_layer(self, in_channel, out_channel, padding):
    layer = nn.Sequential(
        nn.ConvTranspose2d(in_channel,out_channel,kernel_size=(4,4),stride=(2,2),padding=padding),
        nn.BatchNorm2d(out_channel),
        nn.ReLU()
    )
    return layer
```

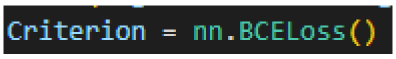
Then, use the convolution transpose layers to upsample the images. Its size would change from (1,1,(latent_size+condition_size)) to (16,16,64)



```python
self.deconv_5 = nn.Sequential(
    nn.ConvTranspose2d(64, 3, kernel_size=(4,4), stride=(2,2), padding=(1,1)),
    nn.Tanh()
)
```

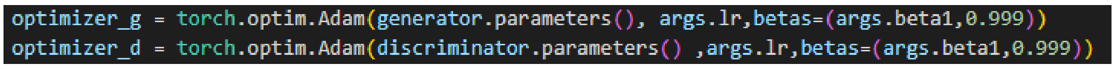
Finally. Use the convolution transpose again to make the image to (64,64,3)

Training:

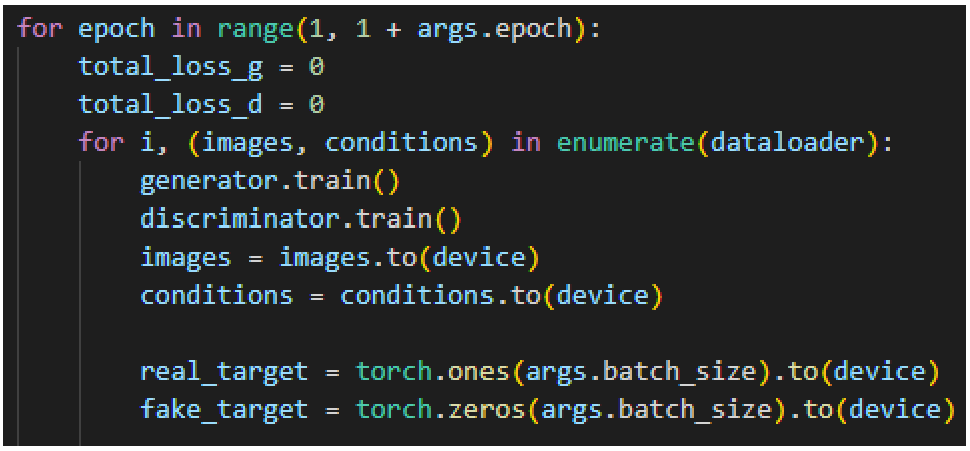


```python
Criterion = nn.BCELoss()
```

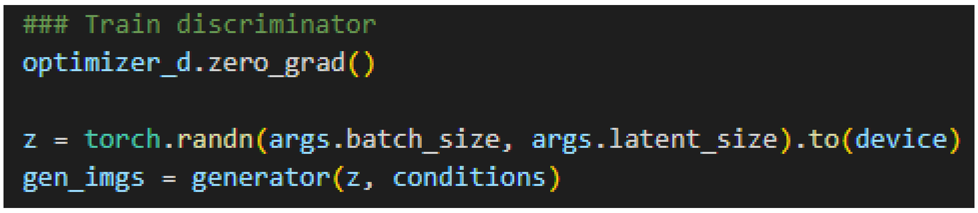
Use binary cross entropy as its loss function, since it a binary classification problem.


```
optimizer_g = torch.optim.Adam(generator.parameters(), args.lr,betas=(args.beta1,0.999))
optimizer_d = torch.optim.Adam(discriminator.parameters() ,args.lr,betas=(args.beta1,0.999))
```

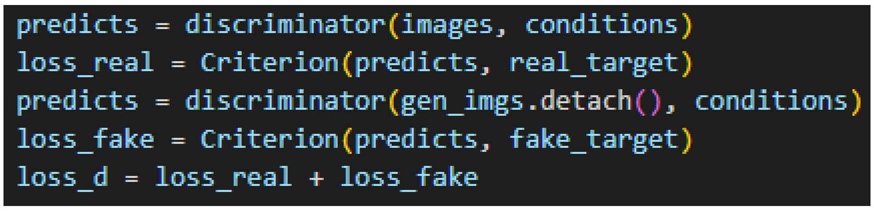
Use adam optimizer.


```
for epoch in range(1, 1 + args.epoch):
    total_loss_g = 0
    total_loss_d = 0
    for i, (images, conditions) in enumerate(dataloader):
        generator.train()
        discriminator.train()
        images = images.to(device)
        conditions = conditions.to(device)

        real_target = torch.ones(args.batch_size).to(device)
        fake_target = torch.zeros(args.batch_size).to(device)
```

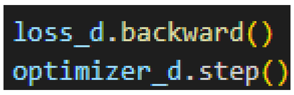
In each epoch, I would get batch data from the data loader. And send them to GPU memory. And set the training target for the discriminator for real and fake targets.


```
### Train discriminator
optimizer_d.zero_grad()

z = torch.randn(args.batch_size, args.latent_size).to(device)
gen_imgs = generator(z, conditions)
```

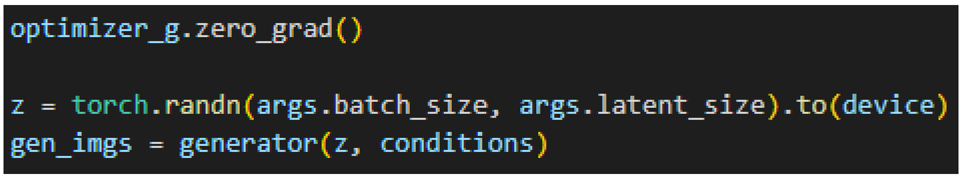
I would train the discriminator first. So I would use randn to generate the latent vectors for generator and make the generator to generate fake images.


```
predicts = discriminator(images, conditions)
loss_real = Criterion(predicts, real_target)
predicts = discriminator(gen_imgs.detach(), conditions)
loss_fake = Criterion(predicts, fake_target)
loss_d = loss_real + loss_fake
```

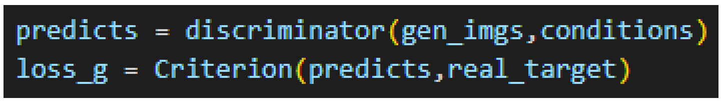
Then, send the real images and fake images and condition to the discriminator to predict and use BCE loss to evaluate.


```
loss_d.backward()
optimizer_d.step()
```

After that, do the back propagation and update discriminator’s parameters

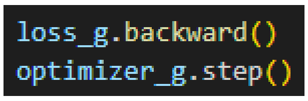

```
optimizer_g.zero_grad()

z = torch.randn(args.batch_size, args.latent_size).to(device)
gen_imgs = generator(z, conditions)
```

And the discriminator is trained, Its time to train the generator. First, I would use randn to generate the latent vectors for generator and make the generator to generate fake images.



```
predicts = discriminator(gen_imgs,conditions)
loss_g = Criterion(predicts,real_target)
```

And send the generated image to the discriminator to predict. The loss function is trying to make the discriminator to predict these images are real image.

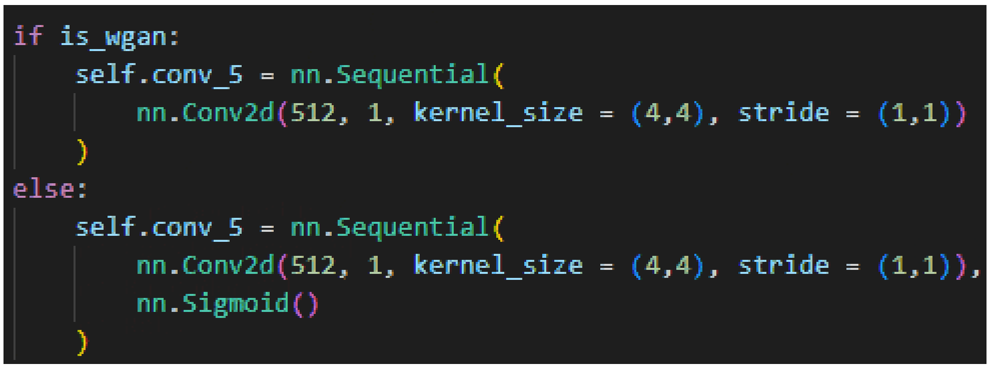


```
loss_g.backward()
optimizer_g.step()
```

After that, do the back propagation and update generator’s parameters

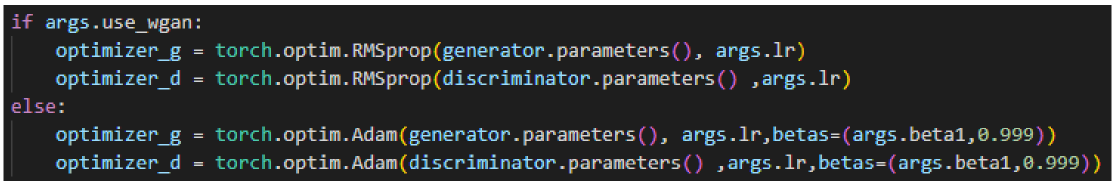
WGAN:

According to its paper, WGAN only need to change some code.



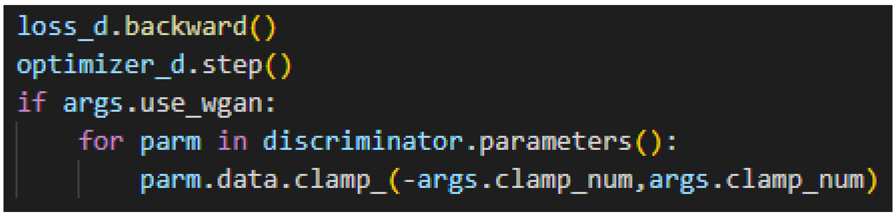
```
if is_wgan:
    self.conv_5 = nn.Sequential(
        nn.Conv2d(512, 1, kernel_size = (4,4), stride = (1,1))
    )
else:
    self.conv_5 = nn.Sequential(
        nn.Conv2d(512, 1, kernel_size = (4,4), stride = (1,1)),
        nn.Sigmoid()
    )
```

1, remove the sigmoid function in discriminator.



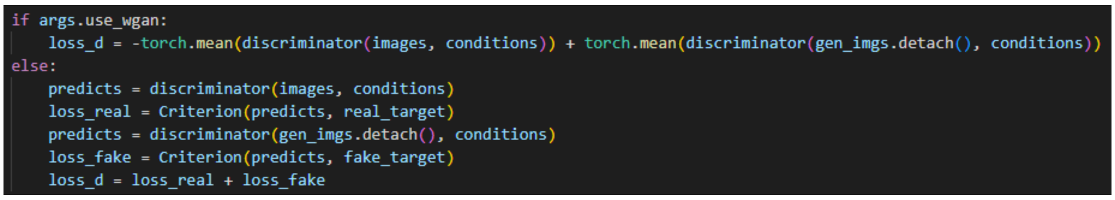
```
if args.use_wgan:
    optimizer_g = torch.optim.RMSprop(generator.parameters(), args.lr)
    optimizer_d = torch.optim.RMSprop(discriminator.parameters() ,args.lr)
else:
    optimizer_g = torch.optim.Adam(generator.parameters(), args.lr,betas=(args.beta1,0.999))
    optimizer_d = torch.optim.Adam(discriminator.parameters() ,args.lr,betas=(args.beta1,0.999))
```

2, Don’t use momentum related optimizers.



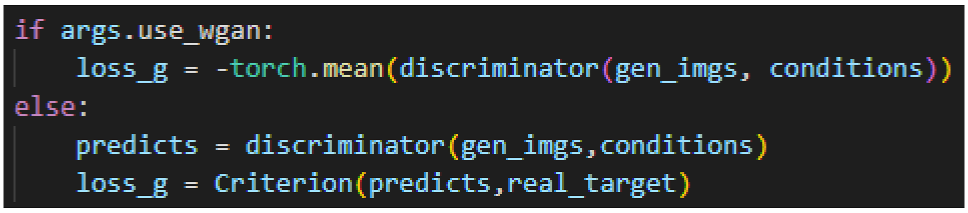
```
loss_d.backward()
optimizer_d.step()
if args.use_wgan:
    for parm in discriminator.parameters():
        parm.data.clamp_(-args.clamp_num,args.clamp_num)
```

3, Clamp the parameters after updating the them.



```
if args.use_wgan:
    loss_d = -torch.mean(discriminator(images, conditions)) + torch.mean(discriminator(gen_imgs.detach(), conditions))
else:
    predicts = discriminator(images, conditions)
    loss_real = Criterion(predicts, real_target)
    predicts = discriminator(gen_imgs.detach(), conditions)
    loss_fake = Criterion(predicts, fake_target)
    loss_d = loss_real + loss_fake
```

$$g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$$

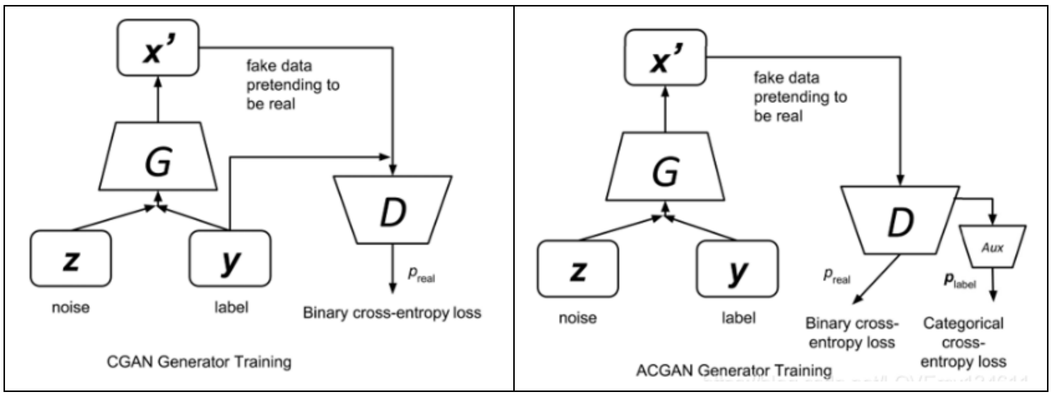


```python
if args.use_wgan:
    loss_g = -torch.mean(discriminator(gen_imgs, conditions))
else:
    predicts = discriminator(gen_imgs,conditions)
    loss_g = Criterion(predicts,real_target)
```

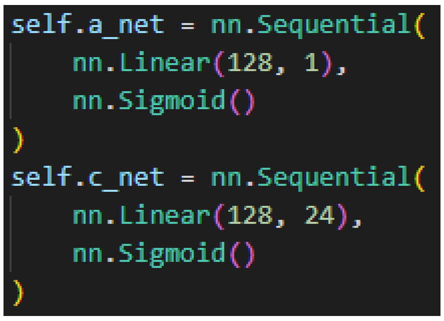
$$g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$$

4, Don’t use the log likelihood loss functions.

ACGAN:

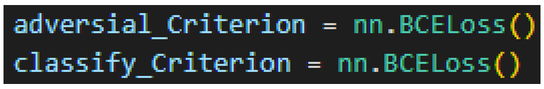

x’
fake data pretending to be real
G
D
z
y
noise
label
p_{real}
Binary cross-entropy loss
CGAN Generator Training
x’
fake data pretending to be real
G
D
Aux
z
y
noise
label
p_{real}
p_{label}
Binary cross-entropy loss
Categorical cross-entropy loss
ACGAN Generator Training

I also implement the ACGAN. Compare with other cGAN using condition as the input of discriminator, the ACGAN’s discriminator only takes image as its input and it would output real/fake and its classification result.



```python
self.a_net = nn.Sequential(
    nn.Linear(128, 1),
    nn.Sigmoid()
)
self.c_net = nn.Sequential(
    nn.Linear(128, 24),
    nn.Sigmoid()
)
```

For its generator, I modify I its last output to make it output the real/fake and classification result.



```python
adversial_Criterion = nn.BCELoss()
classify_Criterion = nn.BCELoss()
```

For its loss function, I add a binary cross entropy loss as its classification loss since it’s a multi-label problem.

```
adversial, classify = discriminator(images)
loss_real = adversial_Criterion(adversial, real_target) + classify_Criterion(classify, conditions)
```

```
z = torch.randn(args.batch_size, args.latent_size).to(device)
gen_imgs = generator(z, conditions)
adversial, classify = discriminator(gen_imgs.detach())
loss_fake = adversial_Criterion(adversial, fake_target)
```

```
loss_d = loss_real + loss_fake
loss_d.backward()
optimizer_d.step()
```

During the training of discriminator, I would send the real image and generated image into the discriminator respectively and get its real/fake result and classification result. After that, put them into different loss function to calculate their loss respectively. Then, do the backpropagation and update their parameters,

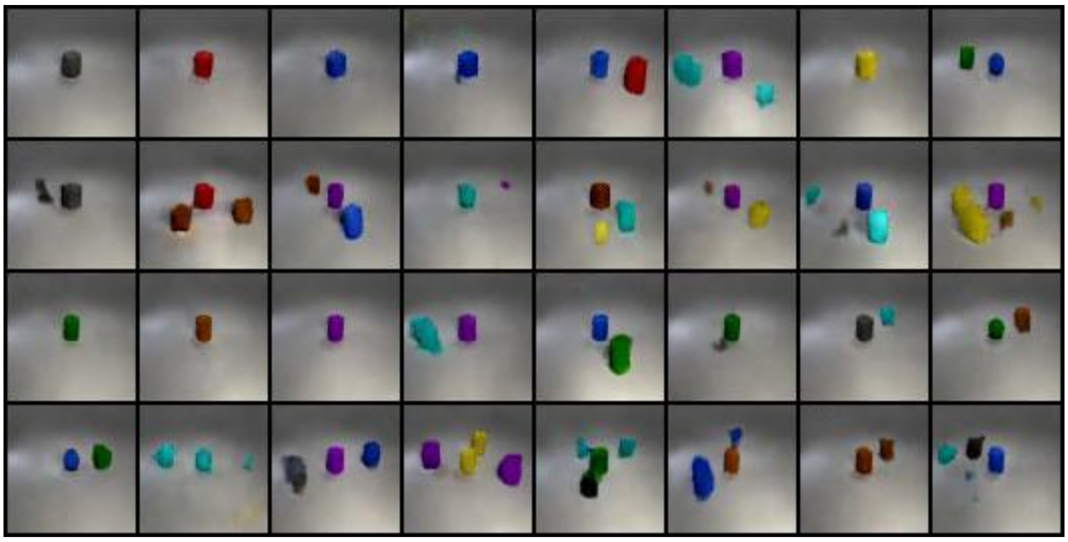
```
z = torch.randn(args.batch_size, args.latent_size).to(device)
gen_imgs = generator(z, conditions)
adversial, classify = discriminator(gen_imgs)
loss_g = adversial_Criterion(adversial, real_target) + classify_Criterion(classify, conditions)

loss_g.backward()
optimizer_g.step()
```

During the training of generator, I would send the generated image into the discriminator respectively and get its real/fake result and classification result. After that, put them into different loss function to calculate their loss respectively. Then, do the backpropagation and update their parameters,

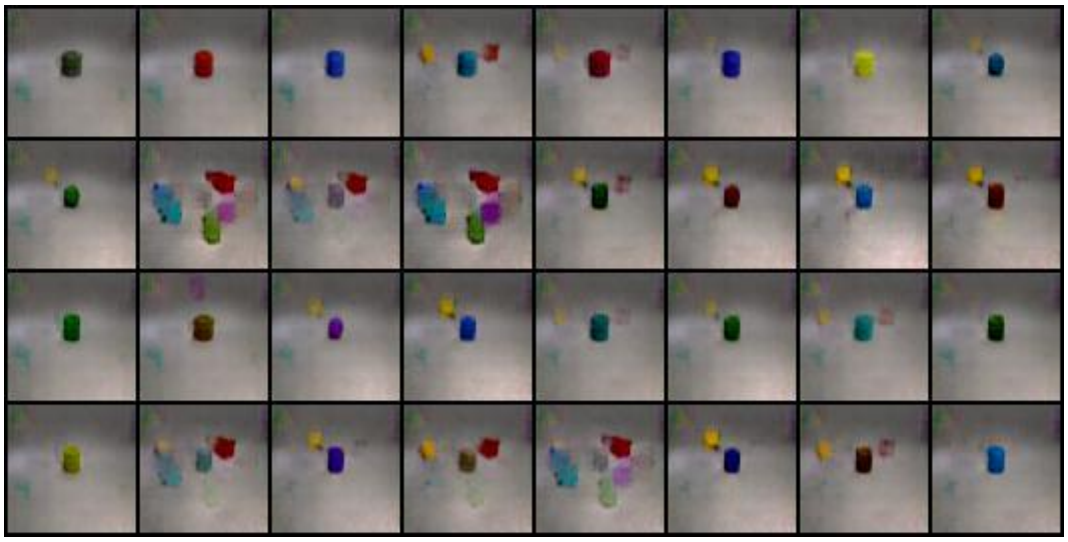
● Results and Discussion

■ Show your results based on the testing data.

DCGAN:



DCGAN with WGAN loss:



Discuss the results of different model's architectures.

DCGAN V.S. DCGAN with WGAN loss:

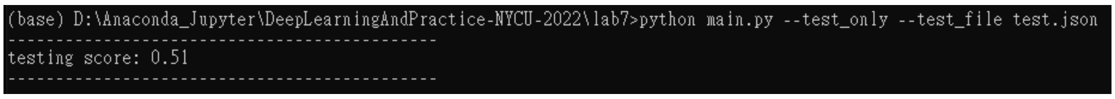
In my DCGAN implementation, it can only achieve about 40~45% accuracy in test set after training for a whole night. However, after using WGAN, it can reach about 60% accuracy in test set. And I noticed that the converge speed of WGAN is slower than the DCGAN. In 50 epochs of training, the accuracy of WGAN is lower than the DCGAN.

ACGAN:

In my implementation of ACGAN, serious mode collapse happened. My ACGAN works good in the first 40 epochs. But after that, my ACGAN start to produce all the same images. And its quality is becoming worse. At the end, it can only produce images that full of strange noise. Maybe I implement it somewhere wrongly.

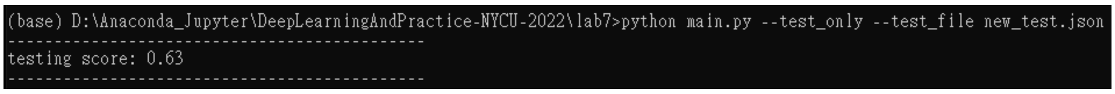
● Experimental Results

test.json:



```
(base) D:\Anaconda_Jupyter\DeepLearningAndPractice-NYCU-2022\lab7>python main.py --test_only --test_file test.json
--------------------------------------------
testing score: 0.51
--------------------------------------------
```

new_test.json



```
(base) D:\Anaconda_Jupyter\DeepLearningAndPractice-NYCU-2022\lab7>python main.py --test_only --test_file new_test.json
--------------------------------------------
testing score: 0.63
--------------------------------------------
```