Deep Learning and Practice Report
Lab7: Let's Play GANs

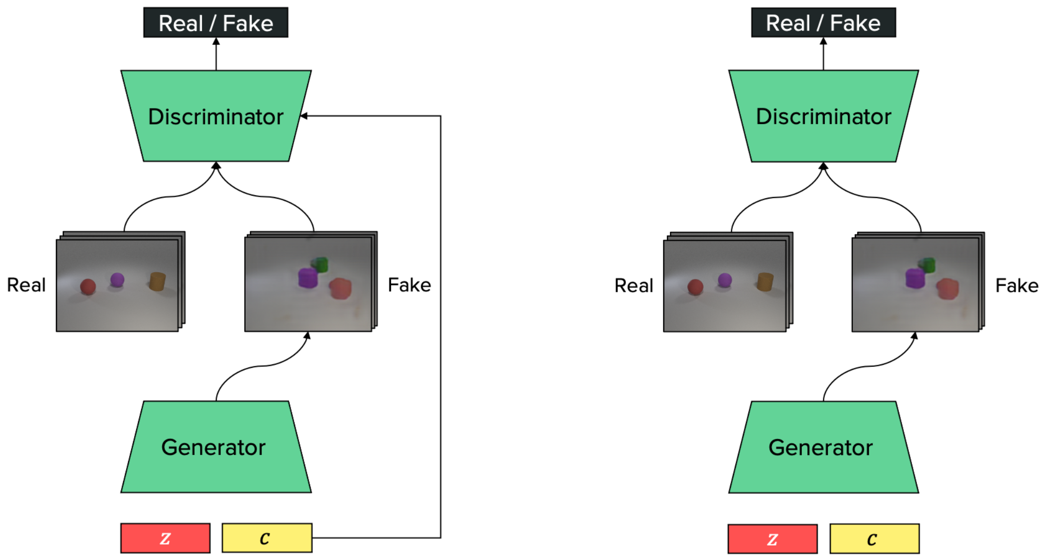
1 Introduction

Generative **a**dversarial **n**etwork (**GAN**) is a classical framework for training generative neural networks. This approach has been widely applied to several computer vision tasks such as image style transfer and image synthesis. In this lab, we are asked to implement a **c**onditional **GAN** (**cGAN**) to generate synthetic images according to multi-label conditions. The i-ClEVR dataset is provided. Specifically, given a specific condition, the model should be able to generate an image with corresponding objects such as red cube or blue cylinder. We need to evaluate our generator with a pre-trained object classifier based on ResNet18 architecture.

2 Implementation Details

2.1 Conditional GAN

A common way to implement cGAN is to feed the condition information into both generator and discriminator. The generator is encouraged to generate realistic images corresponding to a specific condition. Apart from classifying real and fake images, the discriminator also learns to determine whether a given image matches its condition. To further observe the model's behavior, I perform 2 types of setting as shown in figure 1. Type 1 illustrates the framework described above, and the only difference in type 2 is that I didn't send the condition into the discriminator.

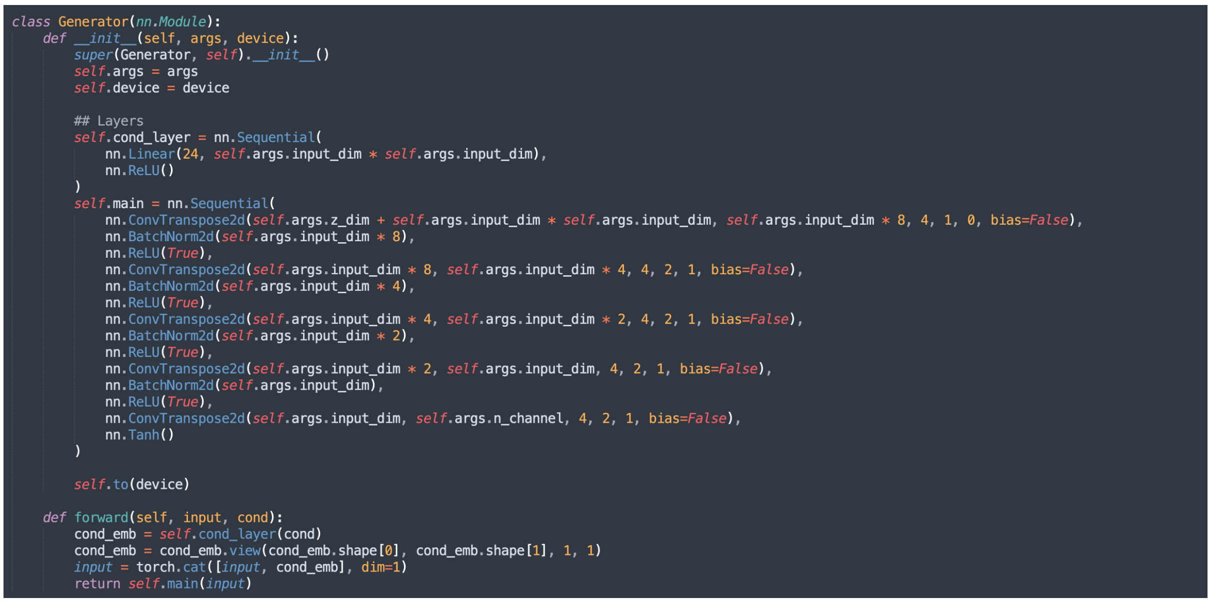


(a) Type 1 (b) Type 2

Figure 1: My conditional GAN (cGAN) framework.

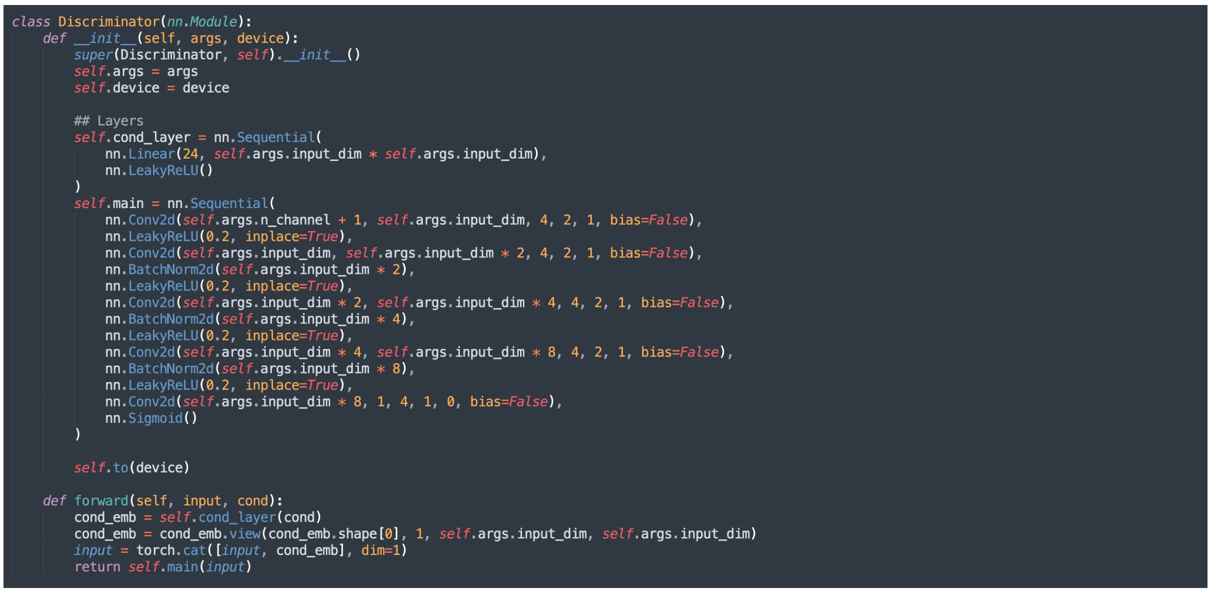
2.2 Model Architectures

I use **d**eep **c**onvolutional **GAN** (**DCGAN**) as my basic model architecture. To incorporate the condition as a part of inputs, I first transform the condition into one-hot-like vectors with length 24 (the number of object categories in this dataset). For example, an image may have the condition [”red cube”, ”blue cylinder”]. The element whose indices correspond to ”red cube” and ”blue cylinder” would be 1, and 0 for the rest. Afterwards, the one-hot-like condition vector is projected to higher dimension via a linear layer, and this feature vector is then concatenated with a random noise and fed into the generator. Similarly, the discriminator also takes the feature vector as a part of its inputs. By reshaping the higher dimensional vector into the same shape of the input image, the condition finally serves as the 4-th channel of the discriminator’s input. Figure 2 and 3 shows the code of my generator and discriminator.



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

        ## Layers
        self.cond_layer = nn.Sequential(
            nn.Linear(24, self.args.input_dim * self.args.input_dim),
            nn.ReLU()
        )
        self.main = nn.Sequential(
            nn.ConvTranspose2d(self.args.z_dim + self.args.input_dim * self.args.input_dim, self.args.input_dim * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(self.args.input_dim * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(self.args.input_dim * 8, self.args.input_dim * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.args.input_dim * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d(self.args.input_dim * 4, self.args.input_dim * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.args.input_dim * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d(self.args.input_dim * 2, self.args.input_dim, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.args.input_dim),
            nn.ReLU(True),
            nn.ConvTranspose2d(self.args.input_dim, self.args.n_channel, 4, 2, 1, bias=False),
            nn.Tanh()
        )

        self.to(device)

    def forward(self, input, cond):
        cond_emb = self.cond_layer(cond)
        cond_emb = cond_emb.view(cond_emb.shape[0], cond_emb.shape[1], 1, 1)
        input = torch.cat([input, cond_emb], dim=1)
        return self.main(input)
```

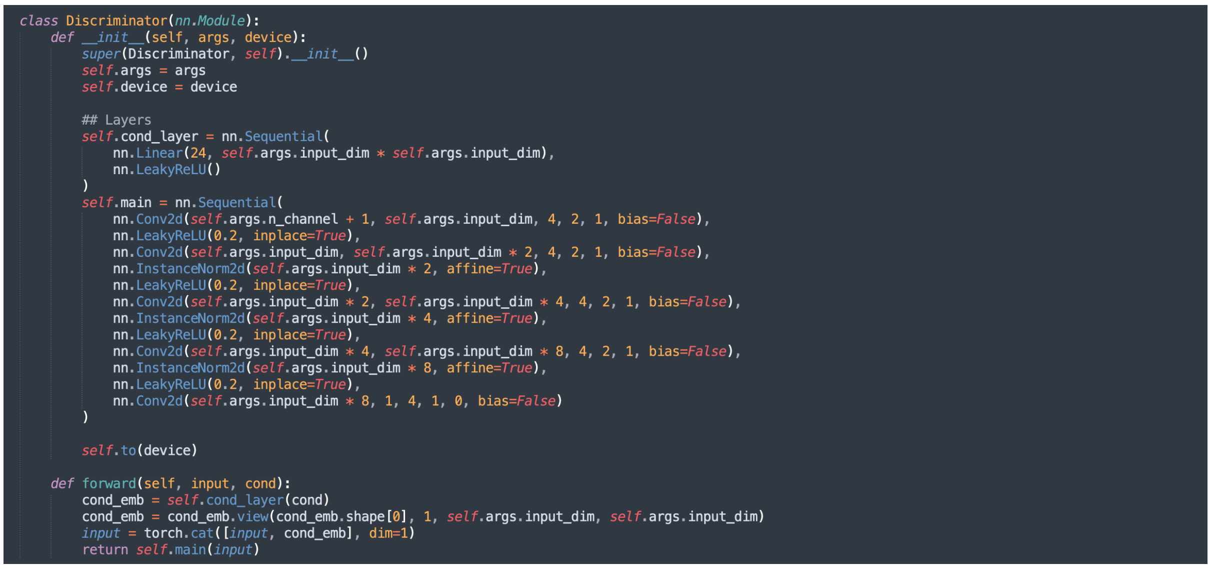
Figure 2: Conditional Generator.



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

        ## Layers
        self.cond_layer = nn.Sequential(
            nn.Linear(24, self.args.input_dim * self.args.input_dim),
            nn.LeakyReLU()
        )
        self.main = nn.Sequential(
            nn.Conv2d(self.args.n_channel + 1, self.args.input_dim, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(self.args.input_dim, self.args.input_dim * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.args.input_dim * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(self.args.input_dim * 2, self.args.input_dim * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.args.input_dim * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(self.args.input_dim * 4, self.args.input_dim * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.args.input_dim * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(self.args.input_dim * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

        self.to(device)

    def forward(self, input, cond):
        cond_emb = self.cond_layer(cond)
        cond_emb = cond_emb.view(cond_emb.shape[0], 1, self.args.input_dim, self.args.input_dim)
        input = torch.cat([input, cond_emb], dim=1)
        return self.main(input)
```

Figure 3: Conditional Discriminator.

I also train my models with WGAN-GP objective. Since the gradient penalty is performed on each sample separately, batch normalization in the discriminator should be removed. Another noticeable modification is that the sigmoid activation at the last layer of the original discriminator is also removed. The code of my WGAN-GP discriminator is shown in figure 4.



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

        ## Layers
        self.cond_layer = nn.Sequential(
            nn.Linear(24, self.args.input_dim * self.args.input_dim),
            nn.LeakyReLU()
        )
        self.main = nn.Sequential(
            nn.Conv2d(self.args.n_channel + 1, self.args.input_dim, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(self.args.input_dim, self.args.input_dim * 2, 4, 2, 1, bias=False),
            nn.InstanceNorm2d(self.args.input_dim * 2, affine=True),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(self.args.input_dim * 2, self.args.input_dim * 4, 4, 2, 1, bias=False),
            nn.InstanceNorm2d(self.args.input_dim * 4, affine=True),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(self.args.input_dim * 4, self.args.input_dim * 8, 4, 2, 1, bias=False),
            nn.InstanceNorm2d(self.args.input_dim * 8, affine=True),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(self.args.input_dim * 8, 1, 4, 1, 0, bias=False)
        )

        self.to(device)

    def forward(self, input, cond):
        cond_emb = self.cond_layer(cond)
        cond_emb = cond_emb.view(cond_emb.shape[0], 1, self.args.input_dim, self.args.input_dim)
        input = torch.cat([input, cond_emb], dim=1)
        return self.main(input)
```

Figure 4: Conditional discriminator for WGAN-GP objective.

2.3 Loss Functions

Training GAN is known to be extremely difficult. Either the discriminator or generator may become too strong, and this usually leads to imbalanced ability between 2 networks. With different loss functions applied, the training process of GAN can be significantly affected. In this lab, I've tried 2 types of objective functions, which I discussed in the following subsections. Let's first introduce a new variable c that denotes the condition corresponding to the real data x.

2.3.1 Vanilla GAN

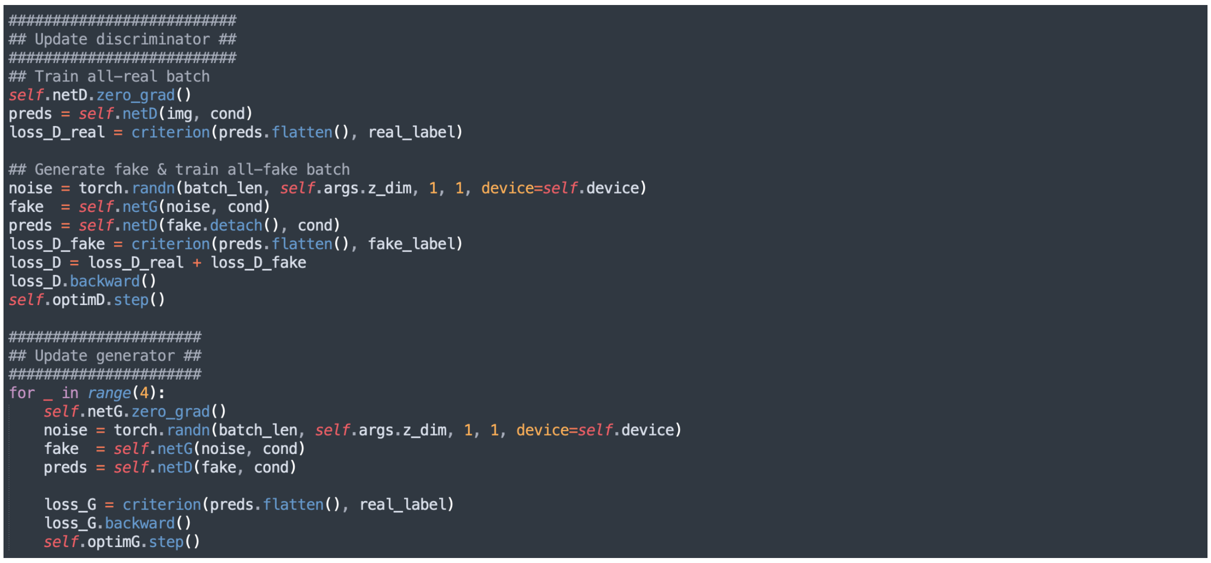
The ordinary objective functions for training GAN can be formulated as a minimax game,

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

where D represents the discriminator with parameters $\theta^{(D)}$ and G represents the generator with parameters θ^{G}. The generator takes a random noise z as input. The goal of generator is to minimize the function while the discriminator attempts to maximize it simultaneously. Figure 5 shows my code of the procedure for training with vanilla GAN. To balance the ability of generator and discriminator, I update the generator 4 times each iteration because I found that the discriminator often becomes too strong in later training stage.

2.3.2 WGAN-GP

To alleviate a bunch of issues occurred in ordinary GAN, several approaches have been proposed. WGAN is a classical one that adopt **e**arth **m**over distance (**EM** / Wassertein distance) as the conver-



```
##########################
## Update discriminator ##
##########################
## Train all-real batch
self.netD.zero_grad()
preds = self.netD(img, cond)
loss_D_real = criterion(preds.flatten(), real_label)

## Generate fake & train all-fake batch
noise = torch.randn(batch_len, self.args.z_dim, 1, 1, device=self.device)
fake  = self.netG(noise, cond)
preds = self.netD(fake.detach(), cond)
loss_D_fake = criterion(preds.flatten(), fake_label)
loss_D = loss_D_real + loss_D_fake
loss_D.backward()
self.optimD.step()

######################
## Update generator ##
######################
for _ in range(4):
    self.netG.zero_grad()
    noise = torch.randn(batch_len, self.args.z_dim, 1, 1, device=self.device)
    fake  = self.netG(noise, cond)
    preds = self.netD(fake, cond)

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

Figure 5: Training procedure for vanilla GAN.

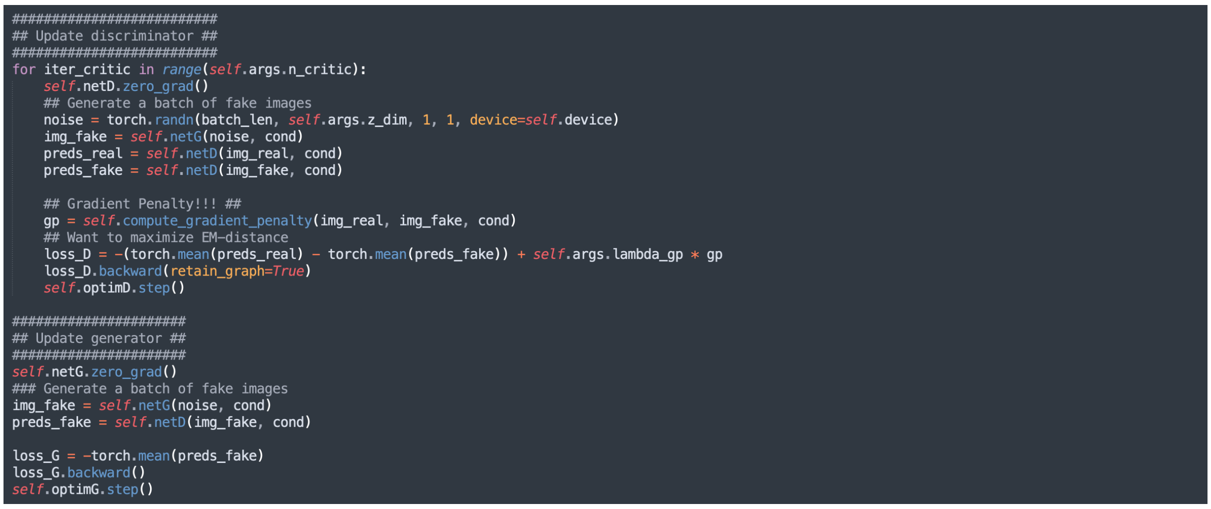
gence criterion. The modified objective function can be written as follows,

$$W(P_r, P_\theta) = \mathbb{E}_{x\sim P_r}[f_w(x,c)] - \mathbb{E}_{z\sim p_z}[g_\theta(z,c)] \tag{2}$$

which is the EM distance between the real data distribution P_r and the fake one P_θ, and $\{f_w\}_{w\in\mathcal{W}}$ is a family of 1-Lipschitz functions that represents the discriminator (critic). The discriminator (critic) aims to maximize this distance while the generator is trained to minimize it. Also, for the original WGAN algorithm, the discriminator will be updated 5 times in each training iteration. Hence, gradient clipping will be performed on discriminator to somehow limit the discriminator's ability. Additionally, gradient penalty is proposed to solve the same problem and it is claimed to be better. A newly defined objective can be expressed as follows,

$$W(P_r, P_\theta) = \mathbb{E}_{x\sim P_r}[f_w(x,c)] - \mathbb{E}_{z\sim p_z}[g_\theta(z,c)] + \lambda\mathbb{E}_{\hat{x}\sim P_{\hat{x}}}[(\|\nabla_{\hat{x}} f_w(\hat{x},c)\|_2 - 1)^2] \tag{3}$$

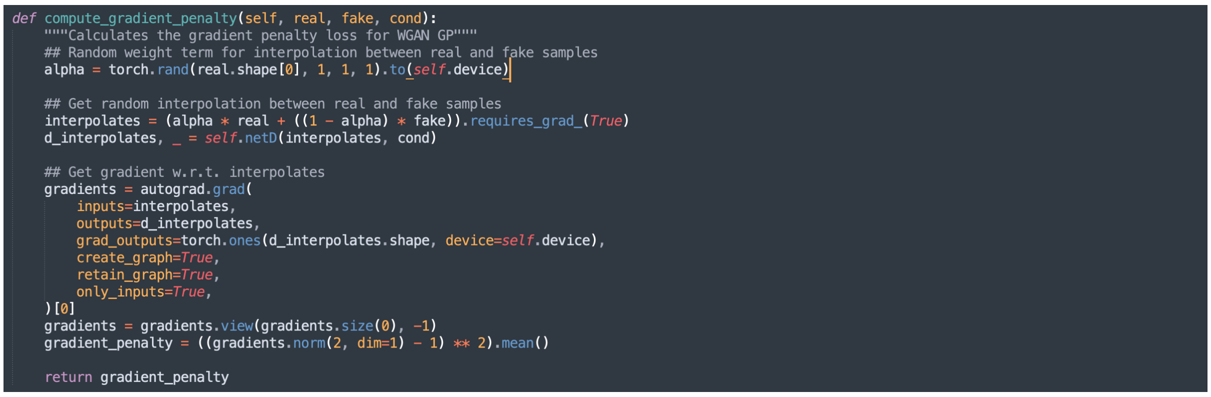
where the last term is the gradient penalty term.



```
##########################
## Update discriminator ##
##########################
for iter_critic in range(self.args.n_critic):
    self.netD.zero_grad()
    ## Generate a batch of fake images
    noise = torch.randn(batch_len, self.args.z_dim, 1, 1, device=self.device)
    img_fake = self.netG(noise, cond)
    preds_real = self.netD(img_real, cond)
    preds_fake = self.netD(img_fake, cond)

    ## Gradient Penalty!!! ##
    gp = self.compute_gradient_penalty(img_real, img_fake, cond)
    ## Want to maximize EM-distance
    loss_D = -(torch.mean(preds_real) - torch.mean(preds_fake)) + self.args.lambda_gp * gp
    loss_D.backward(retain_graph=True)
    self.optimD.step()

######################
## Update generator ##
######################
self.netG.zero_grad()
### Generate a batch of fake images
img_fake = self.netG(noise, cond)
preds_fake = self.netD(img_fake, cond)

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

Figure 6: Training procedure for WGAN-GP.

Figure 6 shows the code for training wgan, and figure 7 shows the code of gradient penalty.



```python
def compute_gradient_penalty(self, real, fake, cond):
    """Calculates the gradient penalty loss for WGAN GP"""
    ## Random weight term for interpolation between real and fake samples
    alpha = torch.rand(real.shape[0], 1, 1, 1).to(self.device)

    ## Get random interpolation between real and fake samples
    interpolates = (alpha * real + ((1 - alpha) * fake)).requires_grad_(True)
    d_interpolates, _ = self.netD(interpolates, cond)

    ## Get gradient w.r.t. interpolates
    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)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()

    return gradient_penalty
```

Figure 7: Gradient penalty for WGAN-GP.

2.4 Hyperparameters

The hyperparameters are list in table 1. I mostly follow the settings specified in the original papers since different hyperparameters usually influence the training process of GAN significantly.

Loss Function	**Learning Rate for D / G**	**Batch Size**	**Optimizer**	β_1	β_2	λ_{GP}
Vanilla GAN	0.0002 / 0.0002	128	Adam	0.5	0.999	–
WGAN-GP	0.0001 / 0.0001	128	Adam	0.5	0.9	10

Table 1: Hyperparameters of vanilla GAN and WGAN-gp.

3 Results and Discussions

3.1 Generated Images

3.1.1 test.json

Figure 8 and 9 shows the results of ”test.json” generated from models trained with vanilla GAN and WGAN-GP respectively. The accuracy of figure 8 is 58.33% and the accuracy of figure 9 is 63.98%.

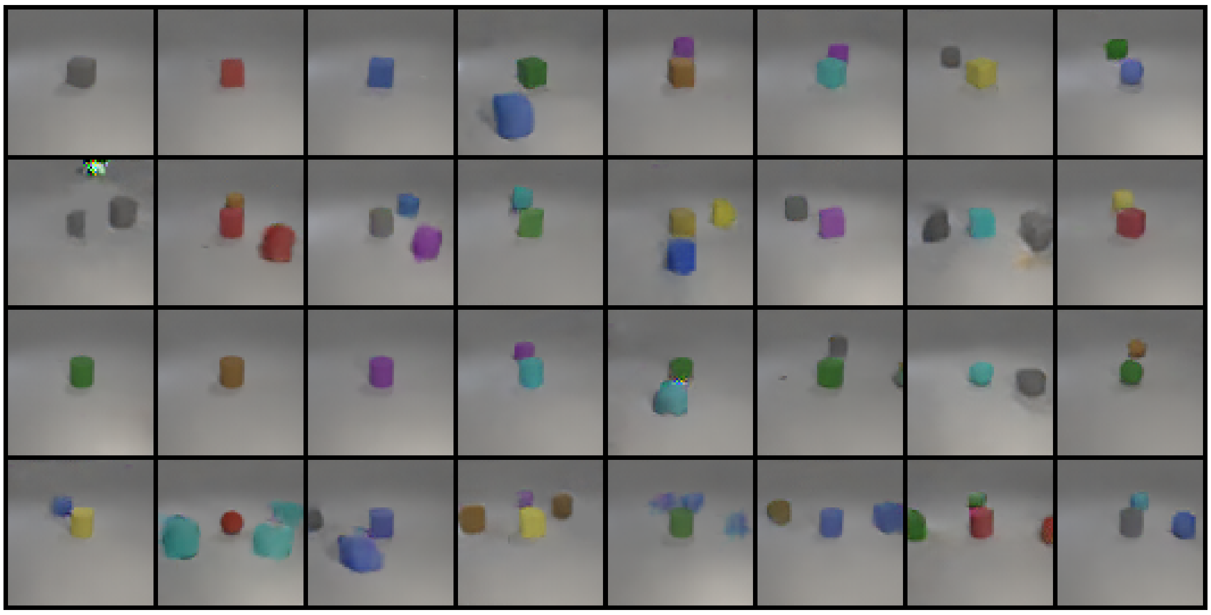


Figure 8: Fake images generated from model trained with vanilla GAN.

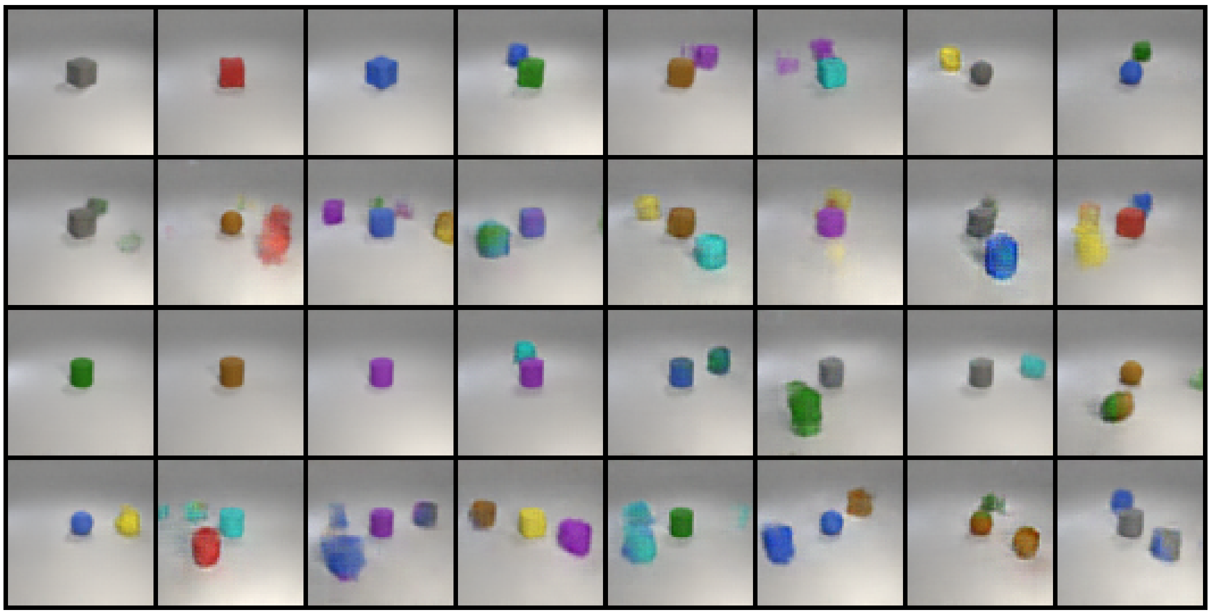


Figure 9: Fake images generated from model trained with WGAN-GP.

3.1.2 new_test.json

Figure 10 and 11 shows the results of ”test.json” generated from models trained with vanilla GAN and WGAN-GP respectively. The accuracy of figure 10 is 57.14% and the accuracy of figure 11 is 64.29%. From the results of both ”test.json” and ”new_test.json”, we can see that although models trained with WGAN-GP produces higher classification accuracy, they produce noisier images. On the other hand, models trained with vanilla GAN have cleaner output images. Also, I find that different random noise vector can significantly influence the accuracy of WGAN-GP models while the same phenomenon is not observed on vanilla GAN models. A possible reason is that WGAN-GP models are not stable as its output being much noisier, thus easily making corruption with respect to different random noise.

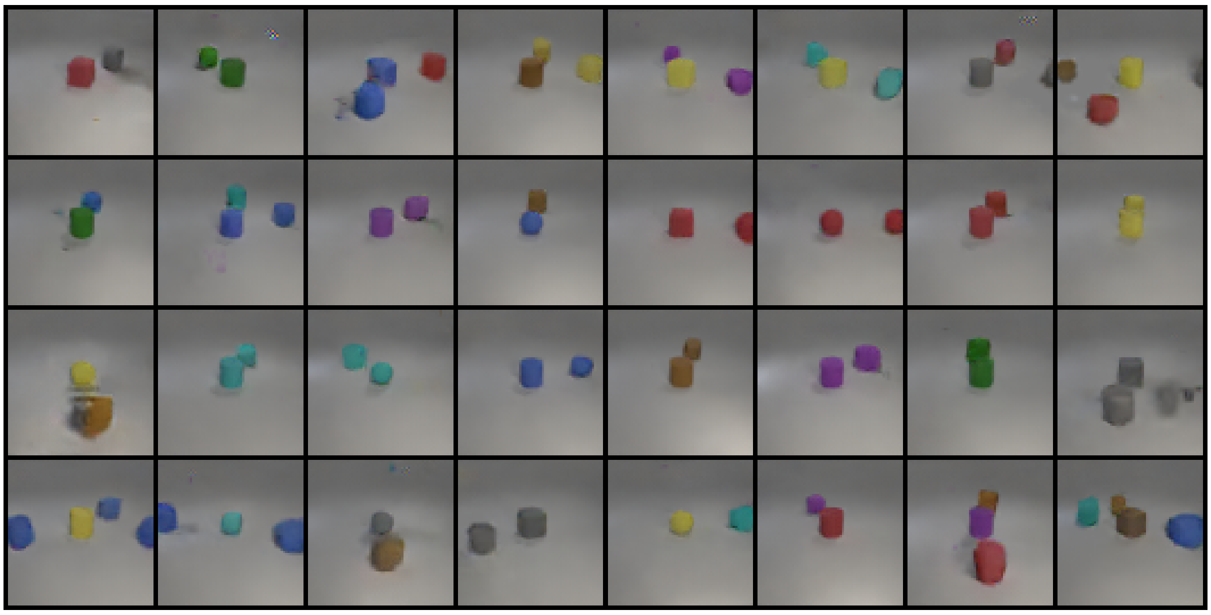


Figure 10: Fake images generated from model trained with vanilla GAN.

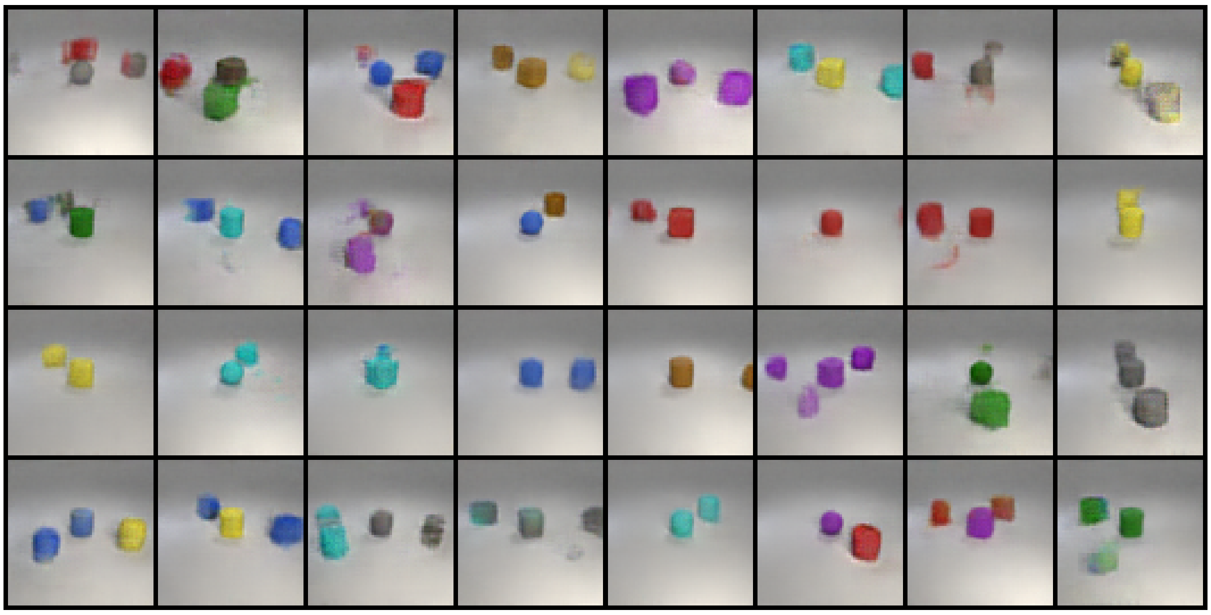


Figure 11: Fake images generated from model trained with WGAN-GP.

Architecture	cGAN	Loss Function	test.json Accuracy	new_test.json Accuracy
DCGAN	Type 2	Vanilla GAN	$< 20.00\%$	–
DCGAN	Type 1	Vanilla GAN	58.33%	57.14%
DCGAN	Type 1	WGAN-GP	63.98%	64.29%

Table 2: Evaluation results using pre-trained classifier.

3.2 Classification Accuracy

Table 2 displays the models' performance with different training settings. WGAN-GP with type 1 conditional GAN framework has highest accuracy among all models.

3.3 Discussion

As mentioned in section 2.1, I have performed experiments on 2 types of conditional GAN framework. The classification accuracy of "test.json" is shown in table 2. Obviously, we can see that using type 2 framework which doesn't provide the discriminator any condition information produces an awful accuracy. To further provide an insight about the results, the generated images of type 2 generator is displayed in figure 12. Interestingly, I find that the generator in fact learns to generate real images. However, it fails to recover the details of the condition. Moreover, the generator even also learns to generate only one object in each image.

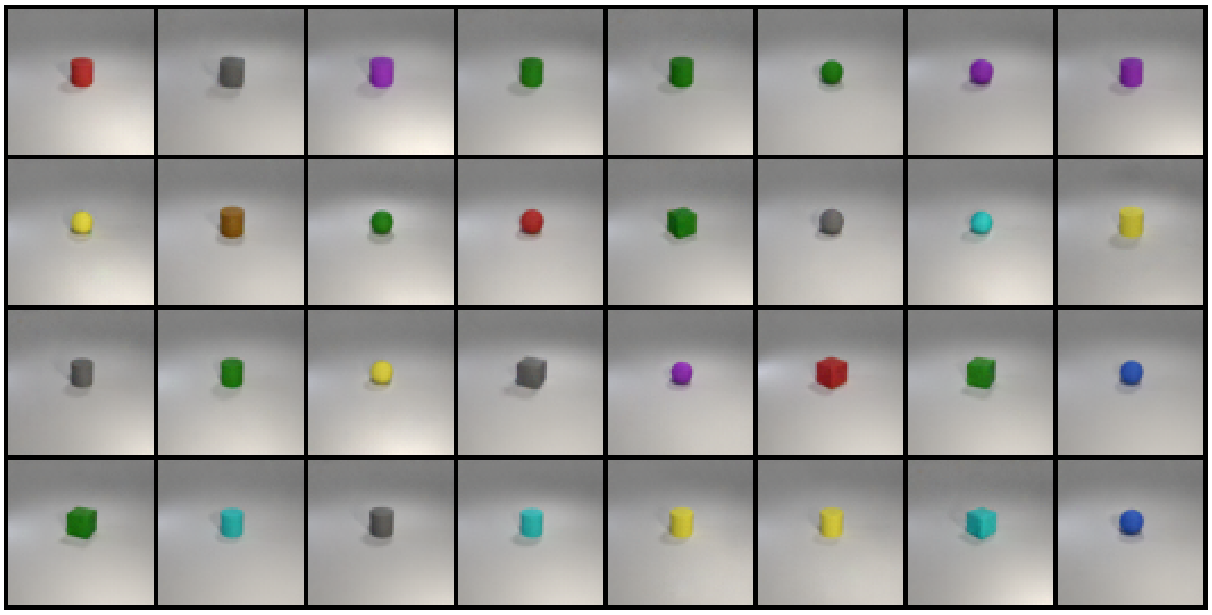
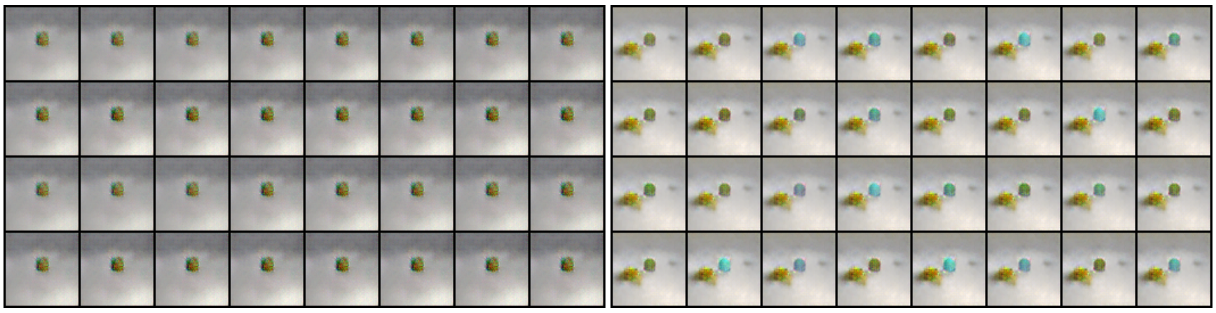


Figure 12: Fake images on "test.json" generated from model with type 2 cGAN framework.

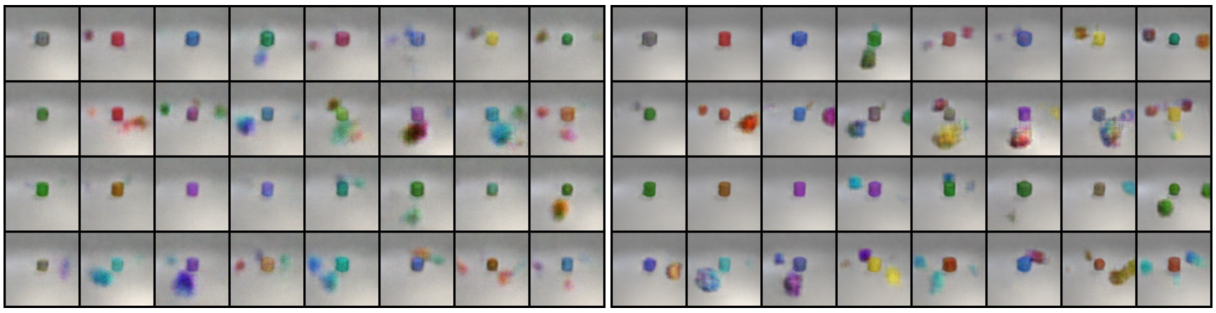
Another observation is about the performance between vanilla GAN and WGAN-GP in the early training stage. Results of vanilla GAN and WGAN-GP in the early training epochs are shown in figure 13 and 14 respectively. Firstly, I find that the generator trained with WGAN-GP can produce images with much higher quality compared to the one trained with vanilla GAN. Also, model trained with vanilla GAN suffers from mode collapse while the model trained with WGAN-GP brings out rich diversity.



(a) Epoch 5

(b) Epoch 20

Figure 13: Results of vanilla GAN on ”test.json” in early training stage.



(a) Epoch 5

(b) Epoch 20

Figure 14: Results of WGAN-GP on ”test.json” in early training stage.