DLP Lab5: Let's Play GANs

Zhi-Yi Chin
joycenerd.cs09@nycu.edu.tw

August 23, 2021

1 Introduction

In this lab, we implement conditional GAN to generate synthetic images in multi-label conditions. Given a latent vector z, which is randomly generated, and a dependent vector, we trained a generator to generate objects with specific colors and shapes; conditional GAN should do this.

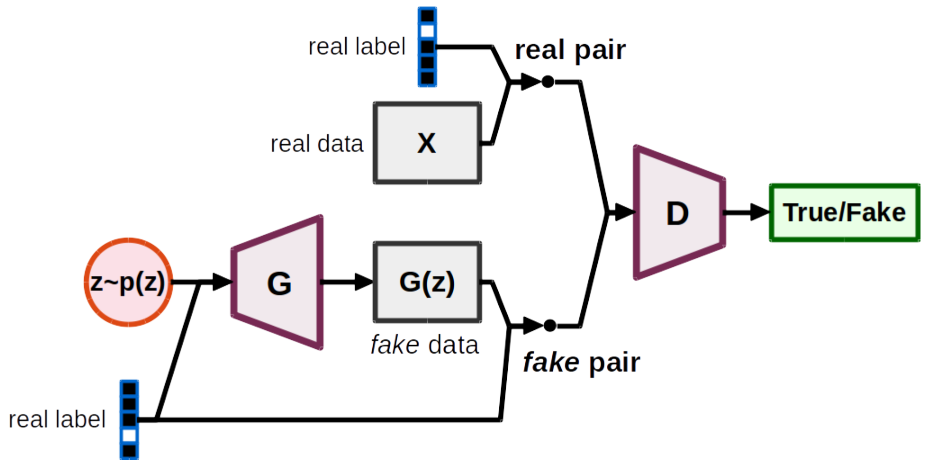

Figure 1: Conditional GAN.

2 Implementation Detail

2.1 Data pre-processing

We are using CLEVR [2] as our dataset. For dealing with the labels in the JSON file, we use the code provided by the TA. For images, we apply transformations that resize the image to 64× 64, transform the value into a tensor and normalize the pixel values.

```python
data_transform=transforms.Compose([
        transforms.Resize((args.im_size,args.im_size)),
        transforms.ToTensor(),
        transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
    ])
```

Listing 1: Image transformation.

```python
class ICLEVRLoader(data.Dataset):
    def __init__(self, root_folder, trans=None, cond=False, mode='train'):
        self.root_folder = root_folder
        self.mode = mode
        self.img_list, self.label_list = get_iCLEVR_data(root_folder,mode)
        if self.mode == 'train':
            print("> Found %d images..." % (len(self.img_list)))

        self.cond = cond
        self.num_classes = 24
        self.trans=trans

    def __len__(self):
        """'return the size of dataset"""
        return len(self.label_list)

    def __getitem__(self, index):
        if self.mode=='train':
            img=Image.open(os.path.join(self.root_folder,
                               'images',self.img_list[index])).convert('RGB')
            if self.trans:
                img=self.trans(img)
            if self.cond:
                cond=torch.Tensor(self.label_list[index])
                return img,cond
            else:
                return img
        else:
            cond=torch.Tensor(self.label_list[index])
            return cond
```

Listing 2: Custom dataset.

2.2 Model

We have tried three different kinds of GAN networks: self-attention GAN, conditional DCGAN, and Wasserstein GAN with ResNet backbone. For every GAN model, we concatenate sample z latent vector from the standard normal distribution. We perform embedding on conditions to the size of 100 by a linear layer and relu as an activation function. At last, we concatenate the input, an image, or a latent vector with the embedded condition.

2.2.1 Self-attention GAN

A Self-attention GAN (SAGAN) [5] is a DCGAN that utilizes self-attention layers. For convolution, they convolve nearby pixels and extract features out of the local block; they work "locally" in each layer. In contrast, self-attention layers learn from distant blocks. [5] states that DCGAN could fail in capturing geometric or structural patterns that occur consistently with multi-class datasets.

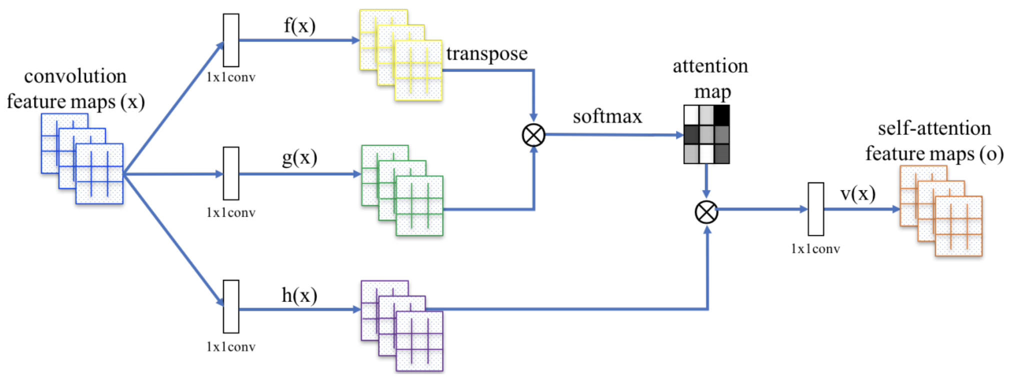

Figure 2: Self-attention mechanism for SAGAN.

First, we construct the self-attention module, which will be used in both generator and discriminator later.

```python
class Self_Attn(nn.Module):
    """ Self attention Layer"""
    def __init__(self,in_dim,activation):
        super(Self_Attn,self).__init__()
        self.chanel_in = in_dim
        self.activation = activation

        self.query_conv = nn.Conv2d(in_channels = in_dim ,
                                    out_channels = in_dim//8 , kernel_size= 1)
        self.key_conv = nn.Conv2d(in_channels = in_dim ,
                                  out_channels = in_dim//8 , kernel_size= 1)
        self.value_conv = nn.Conv2d(in_channels = in_dim ,
                                    out_channels = in_dim , kernel_size= 1)
        self.gamma = nn.Parameter(torch.zeros(1))

        self.softmax  = nn.Softmax(dim=-1) #
    def forward(self,x):
        m_batchsize,C,width ,height = x.size()
        proj_query=self.query_conv(x)
        proj_query=proj_query.view(m_batchsize,-1,width*height).permute(0,2,1)
        proj_key =  self.key_conv(x).view(m_batchsize,-1,width*height)
        energy =  torch.bmm(proj_query,proj_key) # transpose check
        attention = self.softmax(energy) # BX (N) X (N)
        proj_value = self.value_conv(x).view(m_batchsize,-1,width*height)

        out = torch.bmm(proj_value,attention.permute(0,2,1) )
        out = out.view(m_batchsize,C,width,height)

        out = self.gamma*out + x
        return out,attention
```

Listing 3: Self-attention layer.

For the normalization layer in the generator and the discriminator, we use spectral normalization. It is a novel weight normalization technique proposed in [3] for a stabilized training process.

```python
class SpectralNorm(nn.Module):
    def __init__(self, module, name='weight', power_iterations=1):
        super(SpectralNorm, self).__init__()
        self.module = module
        self.name = name
        self.power_iterations = power_iterations
        if not self._made_params():
            self._make_params()

    def _update_u_v(self):
        u = getattr(self.module, self.name + "_u")
        v = getattr(self.module, self.name + "_v")
        w = getattr(self.module, self.name + "_bar")

        height = w.data.shape[0]
        for _ in range(self.power_iterations):
            v.data = l2normalize(torch.mv(torch.t(w.view(height,-1).data),
                                          u.data))
            u.data = l2normalize(torch.mv(w.view(height,-1).data, v.data))

        # sigma = torch.dot(u.data, torch.mv(w.view(height,-1).data, v.data))
        sigma = u.dot(w.view(height, -1).mv(v))
        setattr(self.module, self.name, w / sigma.expand_as(w))

    def _made_params(self):
        try:
            u = getattr(self.module, self.name + "_u")
            v = getattr(self.module, self.name + "_v")
            w = getattr(self.module, self.name + "_bar")
            return True
        except AttributeError:
            return False

    def _make_params(self):
        w = getattr(self.module, self.name)

        height = w.data.shape[0]
        width = w.view(height, -1).data.shape[1]

        u = Parameter(w.data.new(height).normal_(0, 1), requires_grad=False)
        v = Parameter(w.data.new(width).normal_(0, 1), requires_grad=False)
        u.data = l2normalize(u.data)
        v.data = l2normalize(v.data)
        w_bar = Parameter(w.data)

        del self.module._parameters[self.name]

        self.module.register_parameter(self.name + "_u", u)
        self.module.register_parameter(self.name + "_v", v)
        self.module.register_parameter(self.name + "_bar", w_bar)

    def forward(self, *args):
        self._update_u_v()
        return self.module.forward(*args)
```

Listing 4: Spectral normalization.

Then we define the generator and the discriminator as follow:

```python
class Generator(nn.Module):
    """Generator."""

    def __init__(self, batch_size, image_size=64, z_dim=100, conv_dim=64,
                 num_cond=24,c_size=100):
        super(Generator, self).__init__()
        self.imsize = image_size
        self.c_size=c_size
        self.embed_c=nn.Sequential(
            nn.Linear(num_cond,c_size),
            nn.ReLU(inplace=True)
        )

        layer1 = []
        layer2 = []
        layer3 = []
        last = []

        repeat_num = int(np.log2(self.imsize)) - 3
        mult = 2 ** repeat_num # 8
        layer1.append(SpectralNorm(nn.ConvTranspose2d(z_dim+c_size,
                                                      conv_dim * mult, 4)))
        layer1.append(nn.BatchNorm2d(conv_dim * mult))
        layer1.append(nn.ReLU())

        curr_dim = conv_dim * mult

        layer2.append(SpectralNorm(nn.ConvTranspose2d(curr_dim,
                                                      int(curr_dim / 2),
                                                      4, 2, 1)))
        layer2.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer2.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)

        layer3.append(SpectralNorm(nn.ConvTranspose2d(curr_dim,
                                                      int(curr_dim / 2),
                                                      4, 2, 1)))
        layer3.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer3.append(nn.ReLU())

        if self.imsize == 64:
            layer4 = []
            curr_dim = int(curr_dim / 2)
            layer4.append(SpectralNorm(nn.ConvTranspose2d(curr_dim,
                                                          int(curr_dim / 2),
                                                          4, 2, 1)))
            layer4.append(nn.BatchNorm2d(int(curr_dim / 2)))
            layer4.append(nn.ReLU())
            self.l4 = nn.Sequential(*layer4)
            curr_dim = int(curr_dim / 2)

        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)

        last.append(nn.ConvTranspose2d(curr_dim, 3, 4, 2, 1))
        last.append(nn.Tanh())
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn( conv_dim*2, 'relu')
        self.attn2 = Self_Attn( conv_dim,  'relu')

    def forward(self, z,c):
        z = z.view(z.size(0), z.size(1), 1, 1)
        c_embd=self.embed_c(c).reshape(-1,self.c_size,1,1)
        z=torch.cat((z,c_embd),dim=1)
        out=self.l1(z)
        out=self.l2(out)
        out=self.l3(out)
        out,p1 = self.attn1(out)
        out=self.l4(out)
        out,p2 = self.attn2(out)
        out=self.last(out)

        return out, p1, p2
```

Listing 5: The generator of SAGAN.

```
class Discriminator(nn.Module):
    """Discriminator, Auxiliary Classifier."""

    def __init__(self, batch_size=64, image_size=64, conv_dim=64,num_cond=24):
        super(Discriminator, self).__init__()
        self.imsize = image_size
        self.embed_c=nn.Sequential(
            nn.Linear(num_cond,self.imsize*self.imsize),
            nn.ReLU(inplace=True)
        )

        layer1 = []
        layer2 = []
        layer3 = []
        last = []

        layer1.append(SpectralNorm(nn.Conv2d(4, conv_dim, 4, 2, 1)))
        layer1.append(nn.LeakyReLU(0.1))
        # layer1.append(nn.ReLU())

        curr_dim = conv_dim

        layer2.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer2.append(nn.LeakyReLU(0.1))
        curr_dim = curr_dim * 2

        layer3.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer3.append(nn.LeakyReLU(0.1))
        # layer3.append(nn.Softplus())
        curr_dim = curr_dim * 2

        if self.imsize == 64:
            layer4 = []
            layer4.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2,
                                                 1)))
            layer4.append(nn.LeakyReLU(0.1))
            # layer4.append(nn.Tanh())
            self.l4 = nn.Sequential(*layer4)
            curr_dim = curr_dim*2
        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)

        last.append(nn.Conv2d(curr_dim, 1, 4))
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn(conv_dim*4, 'relu')
        self.attn2 = Self_Attn(conv_dim*8, 'relu')
        self.sigmoid=nn.Sigmoid()

    def forward(self, x,c):
        c_embd=self.embed_c(c).reshape(-1,1,self.imsize,self.imsize)
        x=torch.cat((x,c_embd),dim=1)

        out = self.l1(x)
        out = self.l2(out)
        out = self.l3(out)
        out,p1 = self.attn1(out)
        out=self.l4(out)
        out,p2 = self.attn2(out)
        out=self.last(out)
        # out=self.sigmoid(out)

        return out.squeeze(), p1, p2
```

Listing 6: The discriminator of SAGAN.

2.2.2 Conditional DCGAN

Deep convolutional GAN (DCGAN) [4] uses deep convolutional layers in its architecture instead of fully connected layers. Our model architecture contains five convolutional/transpose convolutional layers with Batch Normalization and Leaky ReLU activation and a sigmoid activation for the discriminator output layer and tanh activation for the generator output layer.

```python
class Generator(nn.Module):
    def __init__(self, n_z,n_c,num_conditions,n_ch_g):
        super(Generator, self).__init__()
        self.n_z = n_z
        self.n_c = n_c
        n_ch = [n_ch_g*8, n_ch_g*4, n_ch_g*2, n_ch_g]

        self.embed_c= nn.Sequential(
            nn.Linear(num_conditions, n_c),
            nn.ReLU(inplace=True))

        model = [
            nn.ConvTranspose2d(
                n_z+n_c, n_ch[0], kernel_size=4, stride=2, padding=0,
                bias=False),
            nn.BatchNorm2d(n_ch[0]),
            nn.ReLU(inplace=True)
        ]
        for i in range(1, len(n_ch)):
            model += [
                nn.ConvTranspose2d(
                    n_ch[i-1], n_ch[i], kernel_size=4, stride=2, padding=1,
                    bias=False),
                nn.BatchNorm2d(n_ch[i]),
                # nn.ReLU(inplace=True)
            ]
        model += [
            nn.ConvTranspose2d(
                n_ch[-1], 3, kernel_size=4, stride=2, padding=1,
                bias=False),
            nn.Tanh()
        ]
        self.model = nn.Sequential(*model)

    def forward(self, z, c):
        z = z.reshape(-1, self.n_z, 1, 1)
        c_embd = self.embed_c(c).reshape(-1, self.n_c, 1, 1)
        x = torch.cat((z, c_embd), dim=1)
        return self.model(x)
```

Listing 7: The generator of cDCGAN.

```
class Discriminator(nn.Module):
    def __init__(self, img_sz,n_ch_d,num_conditions):
        super(Discriminator, self).__init__()
        self.img_sz = img_sz
        n_ch = [n_ch_d, n_ch_d*2, n_ch_d*4, n_ch_d*8]

        self.embed_c= nn.Sequential(
            nn.Linear(num_conditions, img_sz*img_sz),
            nn.ReLU(inplace=True))

        model = [
            nn.Conv2d(
                4, n_ch[0], kernel_size=4, stride=2, padding=1,
                bias=False),
            nn.BatchNorm2d(n_ch[0]),
            nn.LeakyReLU(0.2, inplace=True)
        ]

        act=[nn.ReLU(),nn.LeakyReLU(0.2),nn.Softplus(),nn.Tanh()]
        for i in range(1, len(n_ch)):
            model += [
                nn.Conv2d(
                    n_ch[i-1], n_ch[i], kernel_size=4, stride=2, padding=1,
                    bias=False),
                nn.BatchNorm2d(n_ch[i]),
                nn.LeakyReLU(0.2, inplace=True)
            ]
        model += [
            nn.Conv2d(
                n_ch[-1], 1, kernel_size=4, stride=1, padding=0,
                bias=False),
            # nn.Sigmoid()
        ]
        self.model = nn.Sequential(*model)

    def forward(self, image, c):
        c_embd = self.embed_c(c).reshape(-1, 1, self.img_sz, self.img_sz)
        x = torch.cat((image, c_embd), dim=1)
        return self.model(x).reshape(-1)
```

Listing 8: The discriminator of cDCGAN.

2.2.3 Wasserstein GAN

Wasserstein GAN (WGAN) [1] is a type of generative adversarial network that minimizes an approximation of the earth mover's distance (EM) rather than the Jensen-Shannon divergence as in the original GAN formulation. It leads to more stable training than original GANs with less evidence of mode collapse and meaningful curves that can be used for debugging and searching hyperparameters. Our implementation of WGAN utilizes residual blocks. Below is the generator and the discriminator of WGAN.

```python
class Generator(nn.Module):
    def __init__(self,num_cond,c_size,z_size):
        super(Generator, self).__init__()

        self.z_size=z_size
        self.c_size=c_size

        self.embed_c=nn.Sequential(
            nn.Linear(num_cond,c_size),
            nn.ReLU(inplace=True)
        )

        self.model = nn.Sequential(                          # 128 x 1 x 1
            nn.ConvTranspose2d(128+c_size, 128, 8, 1, 0),       # 128 x 8 x 8
            ResidualBlock(128, 128, 3, resample='up'),  # 128 x 16 x 16
            ResidualBlock(128, 128, 3, resample='up'),  # 128 x 32 x 32
            ResidualBlock(128, 128, 3, resample='up'),  # 128 x 32 x 32
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 3, 3, padding=(3-1)//2),     # 3 x 64 x 64
            nn.Tanh()
        )

    def forward(self, z,c):
        z = z.reshape(-1, self.z_size, 1, 1)
        c_embd=self.embed_c(c).reshape(-1, self.c_size, 1, 1)
        x = torch.cat((z, c_embd), dim=1)
        img = self.model(x)
        return img
```

Listing 9: The generator of WGAN.

```python
class Discriminator(nn.Module):
    def __init__(self,num_cond,im_size):
        super(Discriminator, self).__init__()
        n_output = 128
        self.im_size=im_size

        self.embed_c=nn.Sequential(
            nn.Linear(num_cond,self.im_size*self.im_size),
            nn.ReLU(inplace=True)
        )

        self.DiscBlock1 = DiscBlock1(n_output)

        self.model = nn.Sequential(
            ResidualBlock(n_output, n_output, 3, resample='down', bn=False,
                          spatial_dim=32),
            ResidualBlock(n_output, n_output, 3, resample=None, bn=False,
                          spatial_dim=16),
            ResidualBlock(n_output, n_output, 3, resample=None, bn=False,
                          spatial_dim=16),
            nn.ReLU(inplace=True),
        )
        self.l1 = nn.Sequential(nn.Linear(128, 1))                  # 128 x 1

    def forward(self, x,c):
        c_embd=self.embed_c(c).reshape(-1,1,self.im_size,self.im_size)
        x = torch.cat((x, c_embd), dim=1)
        # x = x.view(-1, 3, 32, 32)
        y = self.DiscBlock1(x)
        y = self.model(y)
        y = y.view(x.size(0), 128, -1)
        y = y.mean(dim=2)
        out = self.l1(y).unsqueeze_(1).unsqueeze_(2)
        return out
```

Listing 10: The discriminator of WGAN.

2.3 Loss function

We have experimented with three different kinds of loss functions: binary cross-entropy loss (BCE), Wasserstein distance with gradient penalty (wgan-gp), and hinge loss.

2.3.1 Binary cross entropy loss

Binary cross-entropy loss compares each predicted probability to actual class output, which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. That means how close or far from the actual value. No matter we are training the generator, discriminator, fake image, or real image, binary cross-entropy loss can be made by just calling `nn.BCELoss()`. Also, we will always add a sigmoid layer at the end of the discriminator if using BCE loss.

2.3.2 Wasserstein distance with gradient penalty

WGAN-GP is a Wasserstein loss formulation plus a gradient norm penalty to achieve Lipschitz continuity. A Gradient Penalty is a soft version of the Lipschitz constraint, which follows functions that are 1-Lipschitz if and only if the gradients are of the norm at most one everywhere. The squared difference from norm one is used as the gradient penalty. The gradient penalty is only added when training the discriminator.

```
# discriminator real loss
d_loss_real=-torch.mean(d_out_real)

# discriminator fake loss
d_loss_fake = d_out_fake.mean()

# gradient penalty
d_loss_gp=grad_penalty(inputs,fake_images,conds,device,D,net)
d_loss_gp = lambda_gp * d_loss_gp

# generator loss
g_loss=-torch.mean(g_out_fake)
```

Listing 11: WGAN-GP loss

```python
def grad_penalty(real_images,fake_images,labels,device,D,net):
    alpha = torch.rand(real_images.size(0), 1, 1,
                       1).to(device).expand_as(real_images)
    interpolated = Variable(alpha * real_images.data + (1 - alpha) *
                            fake_images.data, requires_grad=True)
    if net=='sagan':
        out,_,_ = D(interpolated,labels)
    else:
        out=D(interpolated,labels)

    grad = torch.autograd.grad(outputs=out,
                               inputs=interpolated,
                               grad_outputs=torch.ones(out.size()).to(device),
                               retain_graph=True,
                               create_graph=True,
                               only_inputs=True)[0]

    grad = grad.view(grad.size(0), -1)
    grad_l2norm = torch.sqrt(torch.sum(grad ** 2, dim=1))
    d_loss_gp = torch.mean((grad_l2norm - 1) ** 2)
    return d_loss_gp
```

Listing 12: Gradient penalty.

2.3.3 Hinge loss

Hinge loss is mainly known in the application of SVM but has recently been used in GAN.

```python
# discriminator real loss
d_loss_real = torch.nn.ReLU()(1.0 - d_out_real).mean()

# discriminator fake loss
d_loss_fake = torch.nn.ReLU()(1.0 + d_out_fake).mean()

# generator loss
g_loss=-torch.mean(g_out_fake)
```

Listing 13: Hinge loss.

3 Experimental Results

network	g lr	d lr	adv loss	num epochs	β_1	β_2	g conv dim	d conv dim	acc
sagan	0.0001	0.0004	wgan-gp	500	0.0	0.9	64	64	71.67%
sagan	0.0001	0.0004	hinge	200	0.0	0.9	64	64	46.67%
sagan	0.0001	0.0004	bce	200	0.0	0.9	64	64	56.39%
cdcgan	0.0001	0.0002	bce	300	0.5	0.999	300	100	68.89%
cdcgan	0.0002	0.0002	wgan-gp	300	0.5	0.999	300	100	63.33%
wgan	0.0001	0.0001	wgan-gp	200	0.5	0.9	128	128	60.56%
sagan	0.0001	0.0004	wgan-gp	300	0.0	0.9	300	100	**77.22%**

Table 1: All the training results.

3.1 Result of SAGAN

Here we only show the best result of SAGAN. The hyperparamenters are shown as follow:

- Generator learning rate: 0.0001.
- Discriminator learning rate: 0.0004.
- Adversarial loss: wgan-gp.
- Number of epochs: 300.
- Generator activation functions: [relu, relu, relu, relu, tanh].
- Discriminator activation functions: [leaky relu, leaky relu, leaky relu, leaky relu].
- Adam optimizer β_1: 0.0.
- Adam optimizer β_2: 0.9.
- Generator convolution dimension: 300.
- Discriminator convolution dimension: 100.
- Batch size: 128.
- Image size: 64.
- Latent size: 128.
- Condition size: 100.
- Gradient penalty λ: 10.

The testing accuracy of SAGAN is 77.22%.

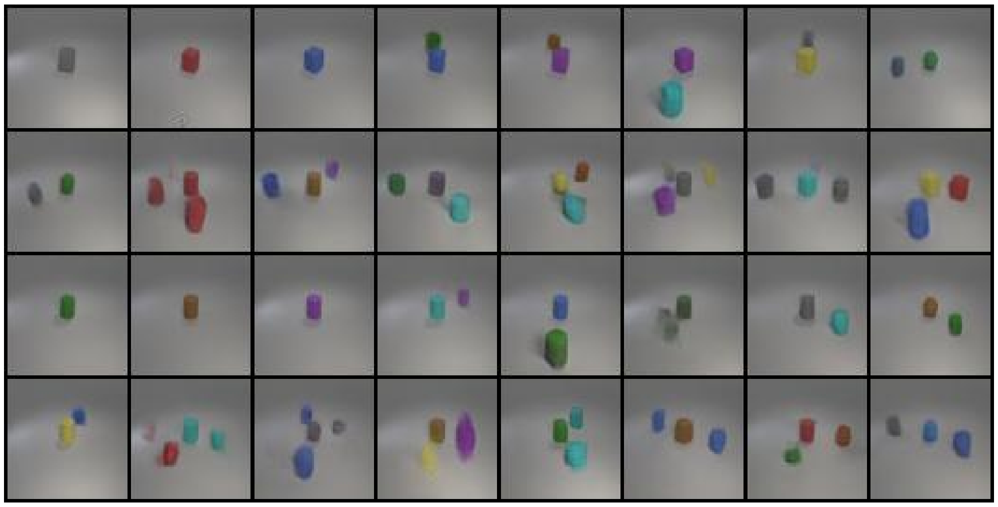

Figure 3: Best resulting generated images of SAGAN.

3.2 Result of cDCGAN

Here we only show the best result of cDCGAN. The hyperparameters are shown as follow:

- Generator learning rate: 0.0001.
- Discriminator learning rate: 0.0002.
- Adversarial loss: bce.
- Number of epochs: 300.
- Generator activation functions: [relu, relu, relu, relu, tanh].
- Discriminator activation functions: [leaky relu, leaky relu, leaky relu, leaky relu, sigmoid].
- Adam optimizer β_1: 0.5.
- Adam optimizer β_2: 0.999.
- Generator convolution dimension: 300.
- Discriminator convolution dimension: 100.
- Batch size: 128.
- Image size: 64.
- Latent size: 128.
- Condition size: 100.

The testing accuracy of cDCGAN is 68.89%.

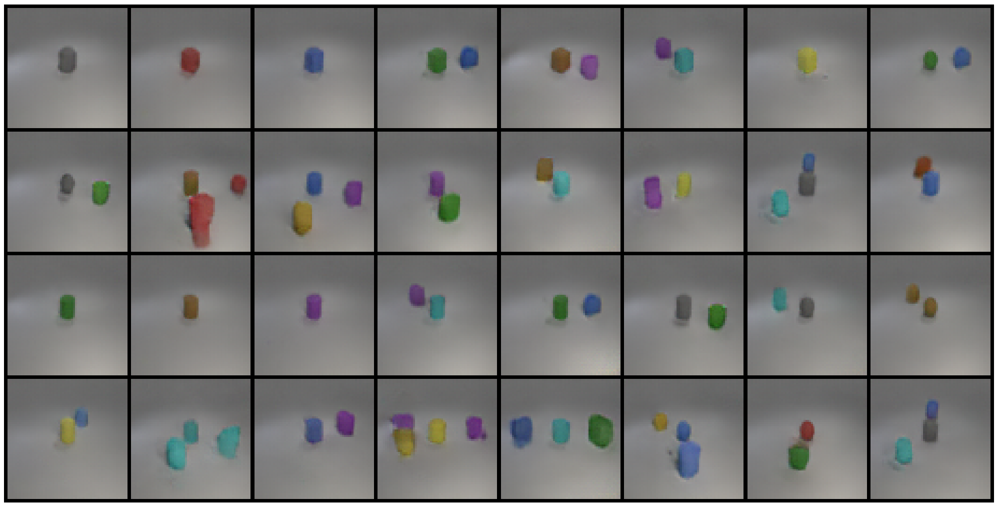

Figure 4: Best resulting generated images of cDCGAN.

3.3 Result of WGAN

- Generator learning rate: 0.0001.
- Discriminator learning rate: 0.0001.
- Adversarial loss: wgan-gp.
- Number of epochs: 300.
- Generator activation functions: [relu, relu, relu, relu, tanh].
- Discriminator activation functions: [relu, relu, relu, relu].
- Adam optimizer β_1: 0.5.
- Adam optimizer β_2: 0.9.
- Generator convolution dimension: 128.
- Discriminator convolution dimension: 128.
- Batch size: 128.
- Image size: 64.
- Latent size: 128.
- Condition size: 100.

The testing accuracy of WGAN is 60.56%

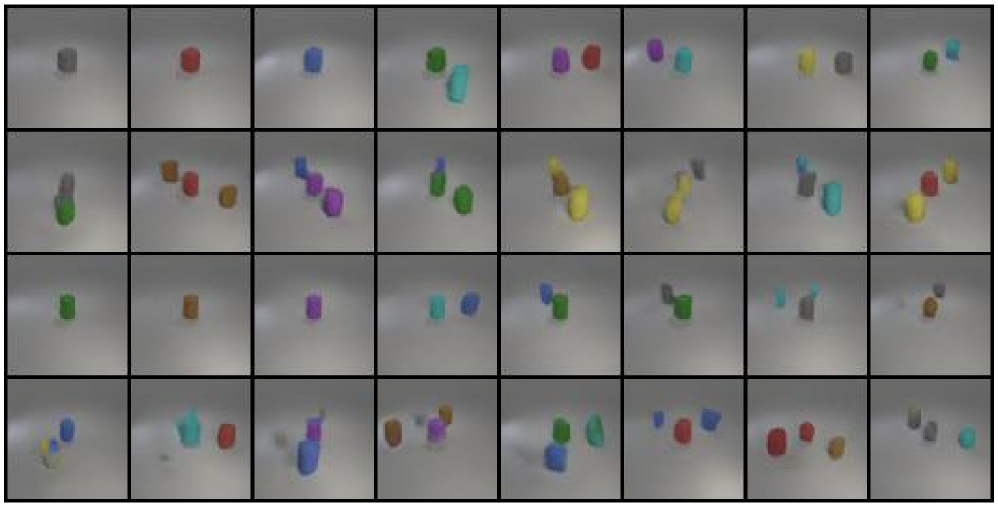

Figure 5: Best resulting generated images of WGAN.

4 Discussion

Based on all the models we have tried, SAGAN performs the best. We think this is because SAGAN has a self-attention mechanism, spectral normalization, and can utilize wgan-gp as the loss function. In cDCGAN, we found out that wgan-gp does not work well with the model. Since wgan-gp works well with SAGAN, we think if we combine WGAN and wgan-gp, we can get a better result, but it comes out that WGAN performs the worst. We do not know the reason why but maybe we have done something wrong in the WGAN model.

We think it is impossible to compare the loss function because different models have suitable loss functions. In our experiment, SAGAN performs well when using wgan-gp but not so good when using hinge loss and bce loss. However, in cDCGAN, using BCE loss performs the best, wgan-gp performs slightly less good.

We find a little trick that can boost the testing accuracy is to use a larger convolution dimension. From Table 1, we can tell that by increasing the generator and discriminator convolution dimension, the accuracy increased by 6%.

References

[1] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. Improved training of wasserstein gans. *arXiv preprint arXiv:1704.00028*, 2017.

[2] J. Johnson, B. Hariharan, L. Van Der Maaten, L. Fei-Fei, C. Lawrence Zitnick, and R. Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 2901–2910, 2017.

[3] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957*, 2018.

[4] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[5] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena. Self-attention generative adversarial networks. In *International conference on machine learning (ICML)*, pages 7354–7363. PMLR, 2019.