# Deep Learning and Practice Lab Report

Lab7 : Let’s Play GANs

## 1. Introduction

The target of this assignment is using conditional GAN to generate a sequence of pseudo images based on given object labels.

GAN basically contains 2 models: discriminator and generator. Discriminator classifies whether given images are real or returns the degree of authenticity and generator try to produce those pseudo image which are real enough to cheat discriminator. Two models train alternately and finally we can get a generator which has the ability to produce quality pseudo images.

- Dataset

  There are totally 24 objects in i-CLEVR datasets with 3 shapes and 8 colors and the object ID is from 0 to 23. Each sample image contains 1 to 3 objects. The account of training dataset is 18009 and the test dataset has 32 groups of object labels.

## 2. Implementation details

- Hyper Parameters Setting

| Variable name | | Value or Type |
|---|---|---|
| Iteration | epoch | 30000 |
| | Iteration for D | 4 |
| | Iteration for G | 1 |
| Training batch size | | 64 |
| Learning rate | Learning rate for D | 0.0001 |
| | Learning rate for G | 0.0003 |
| Optimizer | type | Adam |
| | Beta1 | 0.5 |
| | Beta2 | 0.999 |
| Loss function type | | hinge |
| Trick | Add noise | True |

- Data Loader

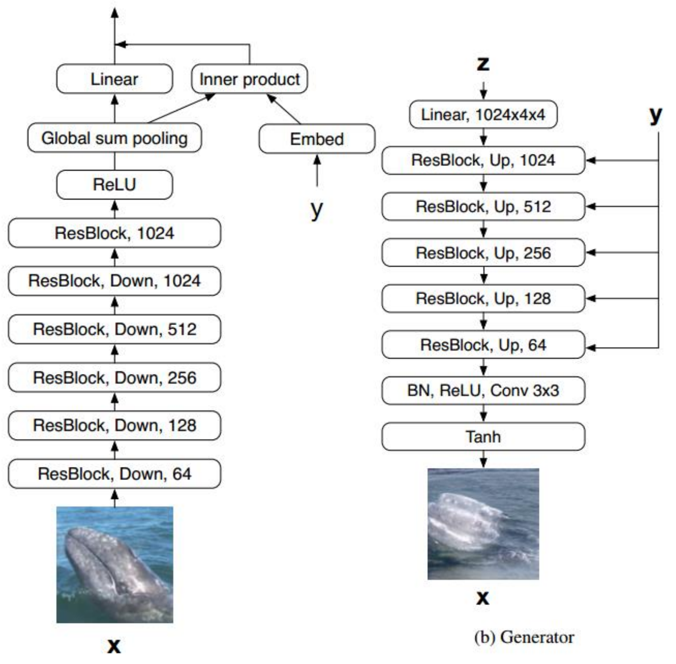
  For training, data loader will return image and label id list with length 3. Image is transformed from PIL image to tensor and label id list is expanded to length 3 by additional class {24} which means “null”.

  For testing, data loader returns label id list after same process and the respective one-hot code for evaluation.

- Model structure

I applied the technique of “SNGAN and cGAN with Projection” as my conditional GAN structure. This structure projection the conditional information (label) into both discriminator and generator.

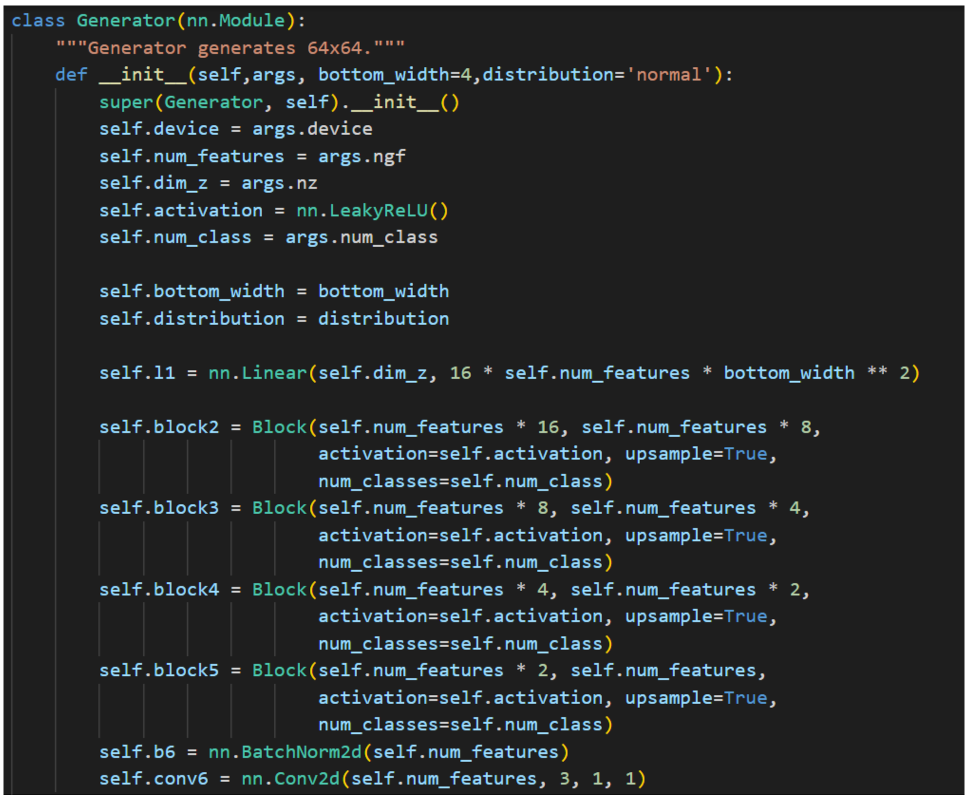
Both discriminator and generator base on Resblocks.

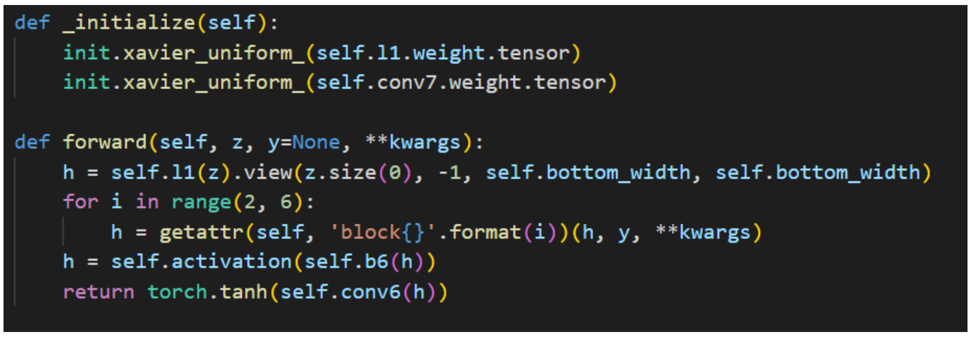


(b) Generator

(a) Discriminator

i. Generator

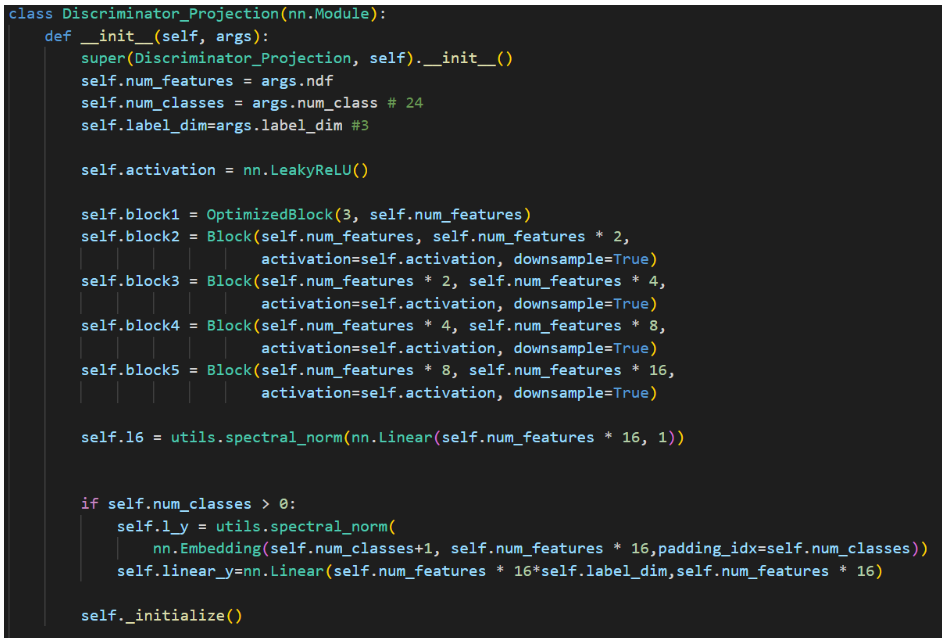

```python
class Generator(nn.Module):
    """Generator generates 64x64."""
    def __init__(self,args, bottom_width=4,distribution='normal'):
        super(Generator, self).__init__()
        self.device = args.device
        self.num_features = args.ngf
        self.dim_z = args.nz
        self.activation = nn.LeakyReLU()
        self.num_class = args.num_class

        self.bottom_width = bottom_width
        self.distribution = distribution

        self.l1 = nn.Linear(self.dim_z, 16 * self.num_features * bottom_width ** 2)

        self.block2 = Block(self.num_features * 16, self.num_features * 8,
                            activation=self.activation, upsample=True,
                            num_classes=self.num_class)
        self.block3 = Block(self.num_features * 8, self.num_features * 4,
                            activation=self.activation, upsample=True,
                            num_classes=self.num_class)
        self.block4 = Block(self.num_features * 4, self.num_features * 2,
                            activation=self.activation, upsample=True,
                            num_classes=self.num_class)
        self.block5 = Block(self.num_features * 2, self.num_features,
                            activation=self.activation, upsample=True,
                            num_classes=self.num_class)
        self.b6 = nn.BatchNorm2d(self.num_features)
        self.conv6 = nn.Conv2d(self.num_features, 3, 1, 1)
```



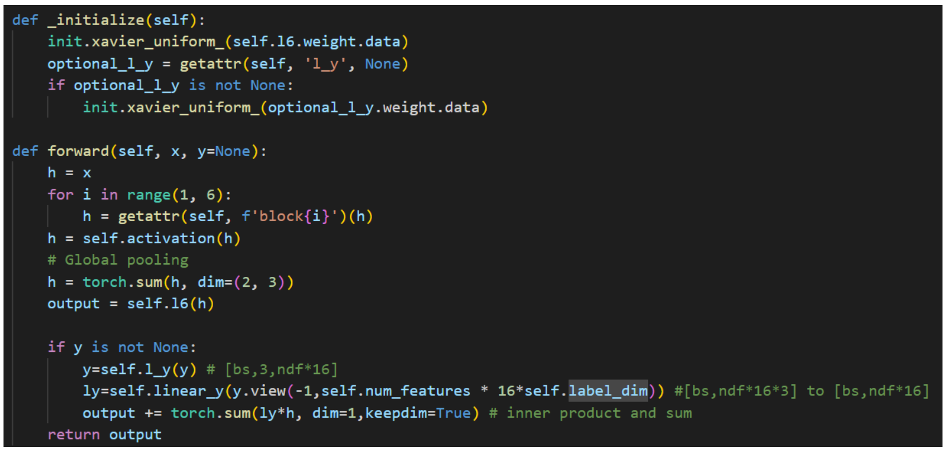
```python
def _initialize(self):
    init.xavier_uniform_(self.l1.weight.tensor)
    init.xavier_uniform_(self.conv7.weight.tensor)

def forward(self, z, y=None, **kwargs):
    h = self.l1(z).view(z.size(0), -1, self.bottom_width, self.bottom_width)
    for i in range(2, 6):
        h = getattr(self, 'block{}'.format(i))(h, y, **kwargs)
    h = self.activation(self.b6(h))
    return torch.tanh(self.conv6(h))
```

ii. Discriminator

It returns a score of given image and labels. Higher score infers to more likely being real, and lower score means that the image is more fake.



```python
class Discriminator_Projection(nn.Module):
    def __init__(self, args):
        super(Discriminator_Projection, self).__init__()
        self.num_features = args.ndf
        self.num_classes = args.num_class # 24
        self.label_dim=args.label_dim #3

        self.activation = nn.LeakyReLU()

        self.block1 = OptimizedBlock(3, self.num_features)
        self.block2 = Block(self.num_features, self.num_features * 2,
                            activation=self.activation, downsample=True)
        self.block3 = Block(self.num_features * 2, self.num_features * 4,
                            activation=self.activation, downsample=True)
        self.block4 = Block(self.num_features * 4, self.num_features * 8,
                            activation=self.activation, downsample=True)
        self.block5 = Block(self.num_features * 8, self.num_features * 16,
                            activation=self.activation, downsample=True)

        self.l6 = utils.spectral_norm(nn.Linear(self.num_features * 16, 1))


        if self.num_classes > 0:
            self.l_y = utils.spectral_norm(
                nn.Embedding(self.num_classes+1, self.num_features * 16,padding_idx=self.num_classes))
            self.linear_y=nn.Linear(self.num_features * 16*self.label_dim,self.num_features * 16)

        self._initialize()
```

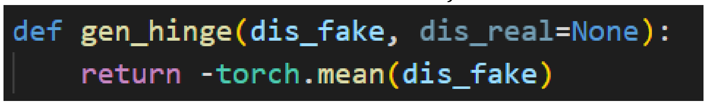


```python
def _initialize(self):
    init.xavier_uniform_(self.l6.weight.data)
    optional_l_y = getattr(self, 'l_y', None)
    if optional_l_y is not None:
        init.xavier_uniform_(optional_l_y.weight.data)

def forward(self, x, y=None):
    h = x
    for i in range(1, 6):
        h = getattr(self, f'block{i}')(h)
    h = self.activation(h)
    # Global pooling
    h = torch.sum(h, dim=(2, 3))
    output = self.l6(h)

    if y is not None:
        y=self.l_y(y) # [bs,3,ndf*16]
        ly=self.linear_y(y.view(-1,self.num_features * 16*self.label_dim)) #[bs,ndf*16*3] to [bs,ndf*16]
        output += torch.sum(ly*h, dim=1,keepdim=True) # inner product and sum
    return output
```

- Loss Function

  For generator:

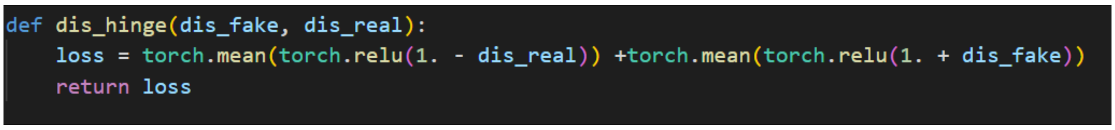
$$\mathrm{Loss_G} = -mean(D_{fake})$$



```python
def gen_hinge(dis_fake, dis_real=None):
    return -torch.mean(dis_fake)
```

  For discriminator:

$$\mathrm{Loss_D} = mean(1 - \mathrm{D_{real}}) + mean(1 + \mathrm{D_{fake}})$$



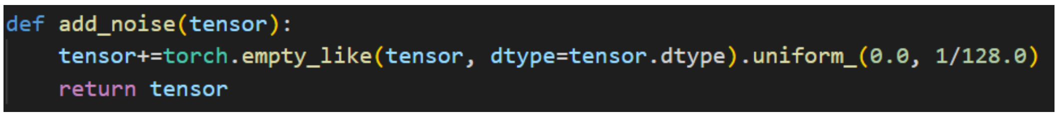
```python
def dis_hinge(dis_fake, dis_real):
    loss = torch.mean(torch.relu(1. - dis_real)) +torch.mean(torch.relu(1. + dis_fake))
    return loss
```

- Trick in training

  I test this trick in training, but only take “adding noise” in my best result in the end.

  i. *Adding noise*

     Adding noise into model input can help the model training. Sometime the GAN broken because the generative distributions are weird. Adding noise into training can help the model close to common distribution.



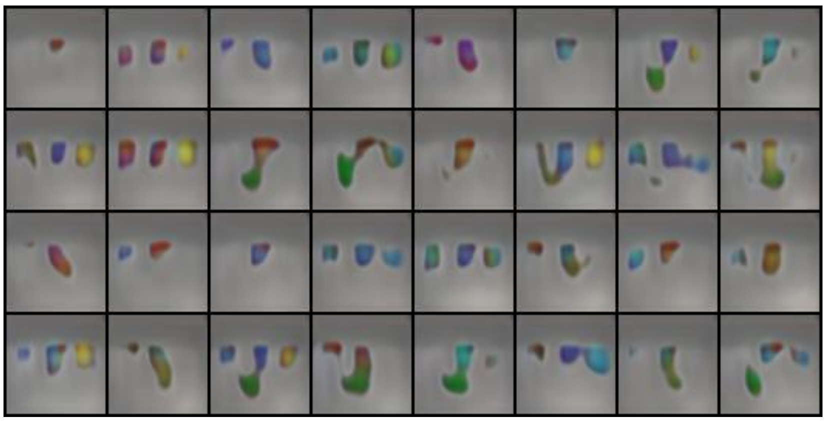
```python
def add_noise(tensor):
    tensor+=torch.empty_like(tensor, dtype=tensor.dtype).uniform_(0.0, 1/128.0)
    return tensor
```

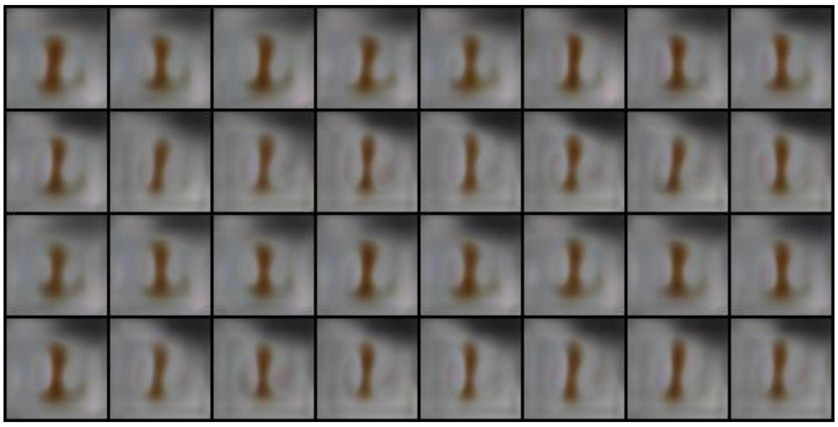
  ii. *Flip label*

     This trick is to swap the binary label from 0 to 1 or from 1 to 0. This skill can confuse the discriminator and avoid the failure of too strong discriminator.

     In experiments, flipping label will cause the generator produce same object to fool discriminator.

     Result of original test after iteration 4000 and 8000



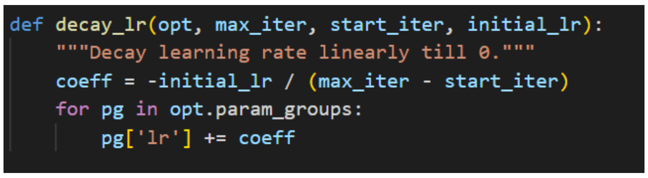


iii. *Relativistic loss*

This idea comes from making the score of generated images closer to real data score. Therefore, it calculates the “distance” between scores of real and fake images. Unfortunately, it brings out more terrible results after training.

iv. *Decay learning rate*

To avoid swinging, I modified learning rate within training and start decay after 4000 iterations.

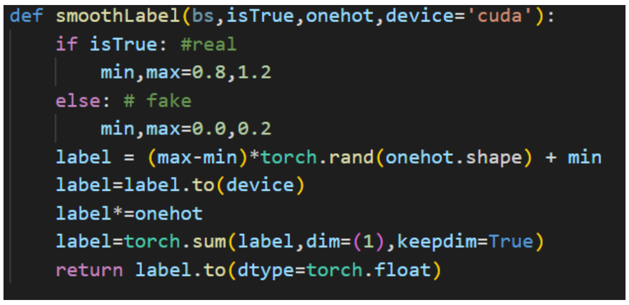


```python
def decay_lr(opt, max_iter, start_iter, initial_lr):
    """Decay learning rate linearly till 0."""
    coeff = -initial_lr / (max_iter - start_iter)
    for pg in opt.param_groups:
        pg['lr'] += coeff
```

v. *Smooth label*

If using binary label such like True and False. It is more suitable for smooth label instead of discrete label. For example, if the sample is true, make label change to 0.8~1.2 from 1; otherwise, change to 0~0.2 from 0 if it is a fake sample.

I only use this trick in DCGAN structure. In final structure, I didn’t use binary label in computation.

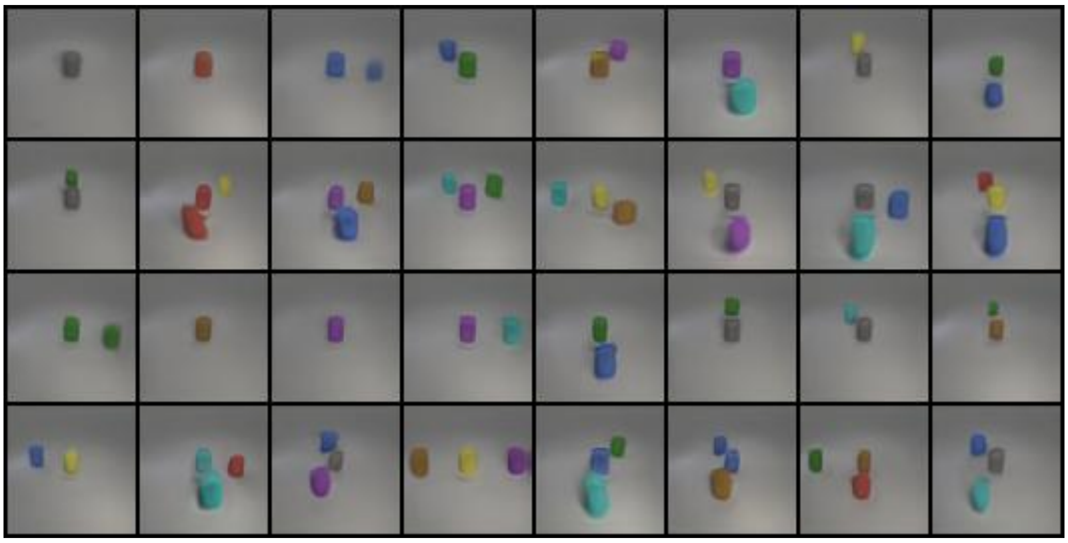


```python
def smoothLabel(bs,isTrue,onehot,device='cuda'):
    if isTrue: #real
        min,max=0.8,1.2
    else: # fake
        min,max=0.0,0.2
    label = (max-min)*torch.rand(onehot.shape) + min
    label=label.to(device)
    label*=onehot
    label=torch.sum(label,dim=(1),keepdim=True)
    return label.to(dtype=torch.float)
```

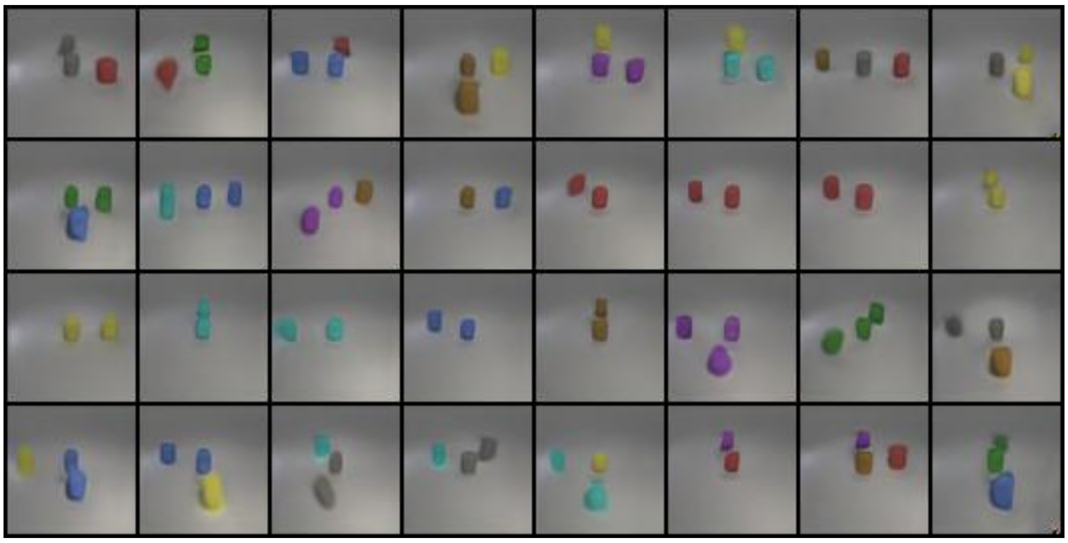
## 3. Results

- Best result

  Original test score=0.791667 (iteration 37000)

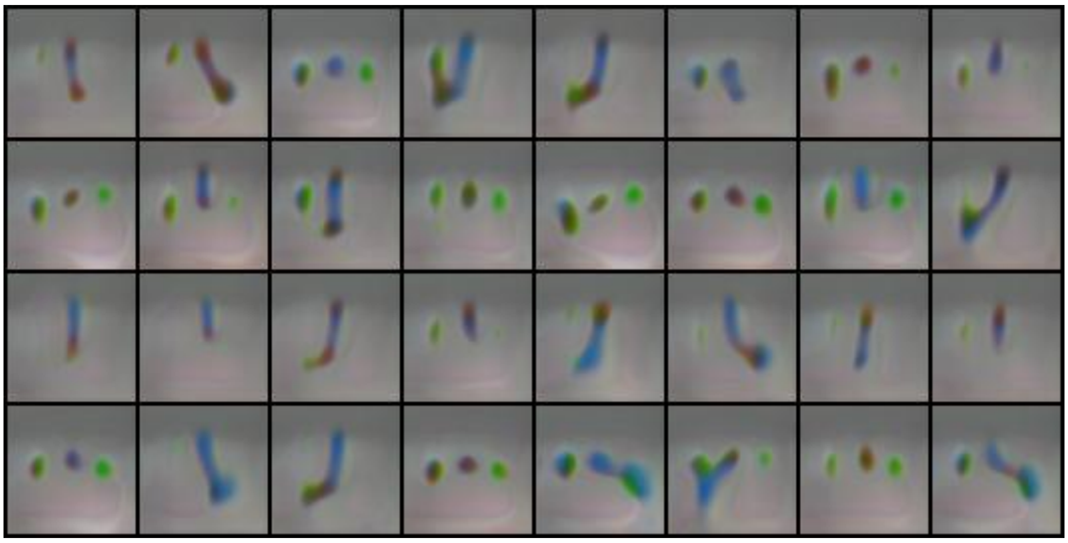


New test score= 0.821429 (iteration 37000)

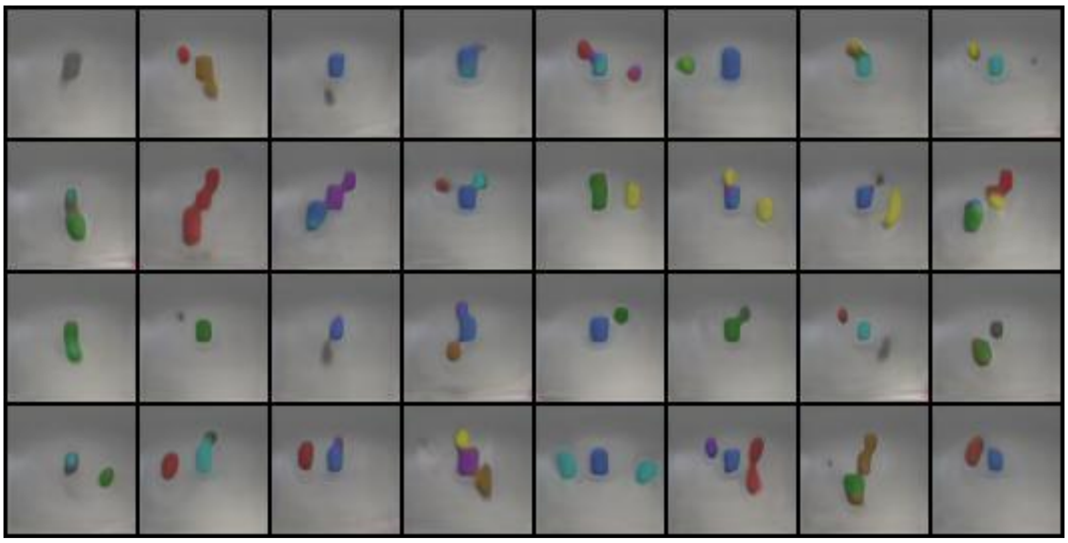


- Observation from Experiment Process

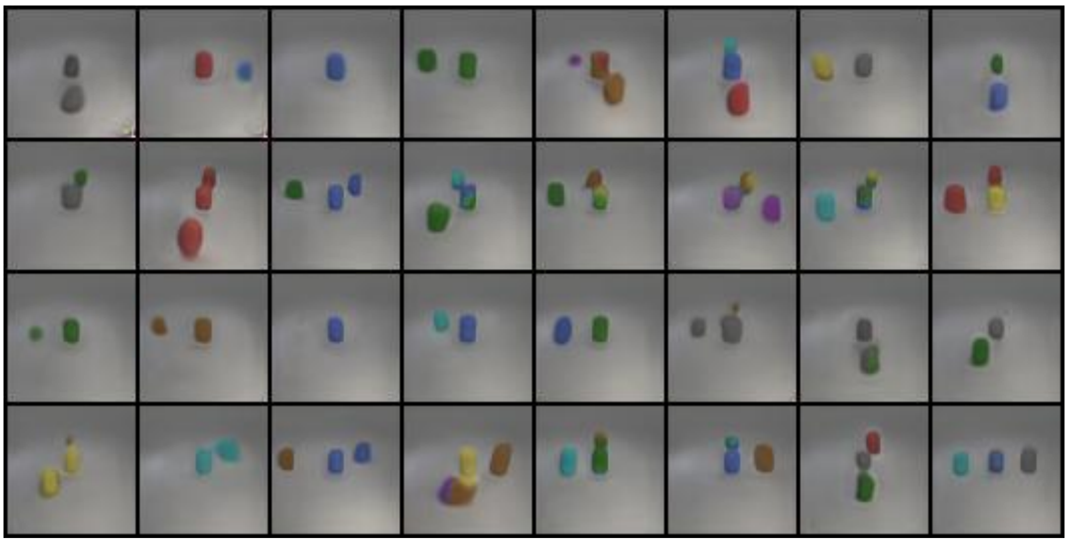
Followed is using original test data to generate pseudo images within training. The generated image can show blurred objects since iteration 500, and score starts to reach 0.22.



The generated object become clearer after iteration 5000.



The quality of generated image keeps increasing. Bellowed images are the result after iteration 15000.



## 4. Discussion

- cDCGAN vs. cGAN with projection

    The result of cDCGAN is poorer than using projection structure. The most different element is the application of label. Using projection embed label information and project them in the output of discriminator output and combine the latent code from label in each layer of generator. In other hand, cDCGAN just “concat” image and label before throw them into network.

- Sum label or Using Linear layer

    Because our samples have multiple labels in one image, we need to embed them and do reshape to match layer input size. I try “sum” them and transfer by linear layer. The results do not have significant difference.

- Proportion between the Iteration of Discriminator and Generator

    In training, we should guarantee the ability of discriminator to lead generator training. However, if discriminator the generator will give up learning which means model collapse. Overall, I train discriminator more times than generator with lower learning rate to control the balance.

    There are some model try to deal with the balance problem, but I did not try those method in this assignment.

## 5. Reference

SNGAN and cGANs with projection discriminator

How to Train a GAN? Tips and tricks to make GANs work