■ Introduction (5%)

In Lab 6, we implemented a conditional GAN and generated synthetic images based on multi-labels conditions. Our training dataset has 18009 images which include at least one object in each image. All objects in an image would be described in the corresponding label. Figure 1 shows two examples of training dataset.

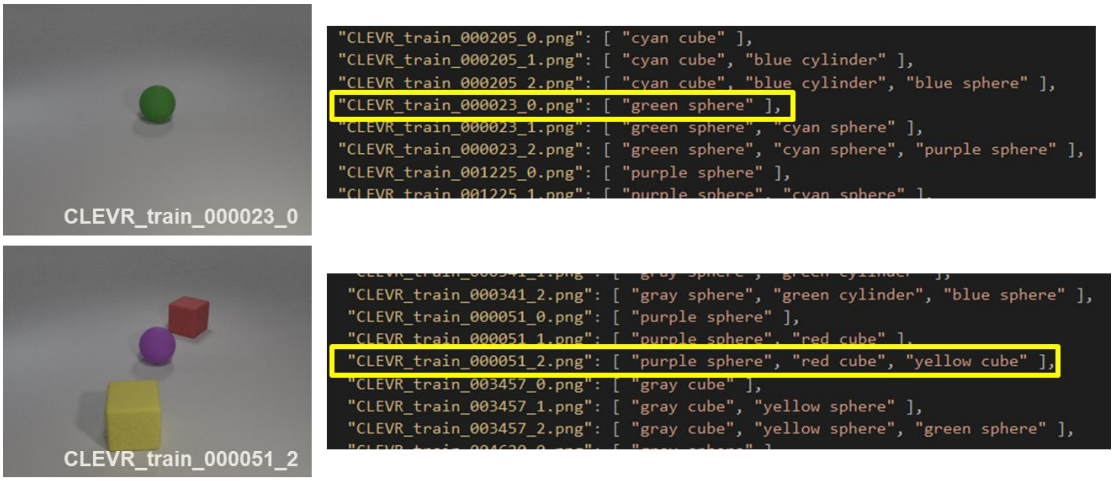


Fig. 1 Two examples of training dataset

The testing dataset has 32 different conditions shown in Fig. 2. Our objective is generating synthetic images which matched the 32 conditions by our conditional GAN.

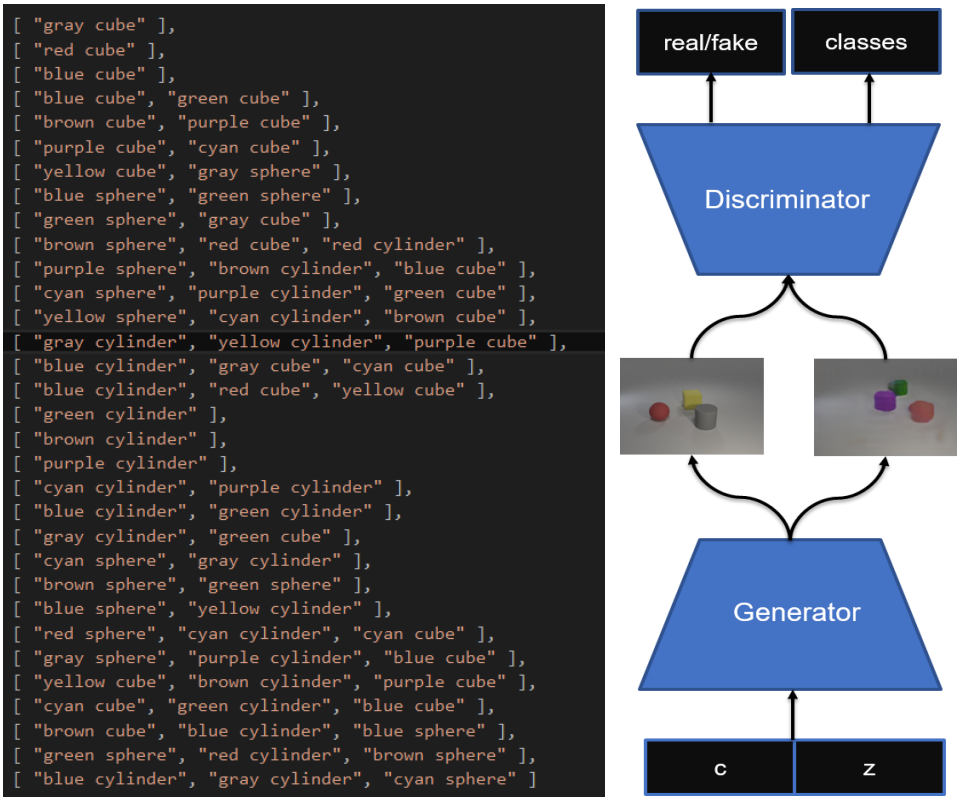


Fig. 2 testing dataset (left) and conditional GAN (right)

- Implementation details (15%)
  - Describe how you implement your model, including your choice of cGAN, model architectures, and loss functions. (10%)

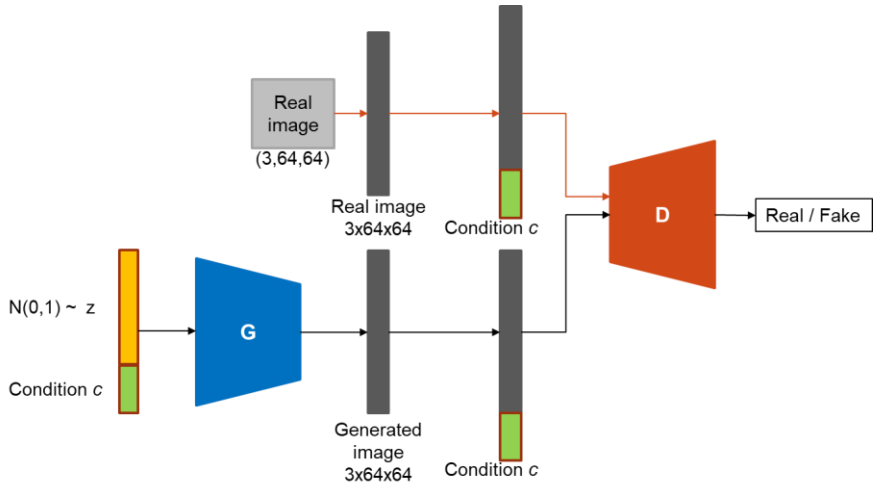
    I implemented two cGANs and the details were described below.

    1. One-hot-code for multi-labels condition

       Objects include 3 shapes (i.e., cube, sphere, and cylinder) and 8 colors (i.e., gray, red, blue, green, brown, purple, cyan, and yellow), therefore, there are 24 kinds of objects. We used an array with length 24 to represent one image. If the *i*-th object existed in the image, array [*i*] was set to 1. Otherwise, array [*i*] was set to 0.

    2. Conditional WGAN

       Architecture (cGAN):



       Architecture (Generator):

```
class Generator(nn.Module):
    def __init__(self, img_shape, latent_dim, n_classes):
        super(Generator, self).__init__()
        self.img_shape = img_shape
        self.n_classes = n_classes
        self.latent_dim = latent_dim

        self.label_emb = nn.Embedding(n_classes, n_classes)

        def block(in_feat, out_feat, normalize=True):
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat, 0.8))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *block(latent_dim + n_classes, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )
```

```
    def forward(self, z, labels):
        # Concatenate label embedding and image to produce input
        gen_input = torch.cat((labels, z), -1)
        img = self.model(gen_input)
        img = img.view(img.shape[0], * self.img_shape)
        return img
```

Architecture (Discriminator):

```
class Discriminator(nn.Module):
    def __init__(self, img_shape, n_classes):
        super(Discriminator, self).__init__()

        self.img_shape = img_shape
        self.n_classes = n_classes
        self.label_embedding = nn.Embedding(n_classes, n_classes)

        self.model = nn.Sequential(
            nn.Linear(n_classes + int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 512),
            nn.Dropout(0.4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1),
        )

    def forward(self, img, labels):
        # Concatenate label embedding and image to produce input
        d_in = torch.cat((img.view(img.size(0), -1), labels), -1)
        validity = self.model(d_in)
        return validity
```

Loss function of Discriminator (WGAN-GP):

$$L = \underbrace{\mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g}[D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r}[D(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}}\sim\mathbb{P}_{\hat{\boldsymbol{x}}}}\left[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2\right]}_{\text{Our gradient penalty}}.$$

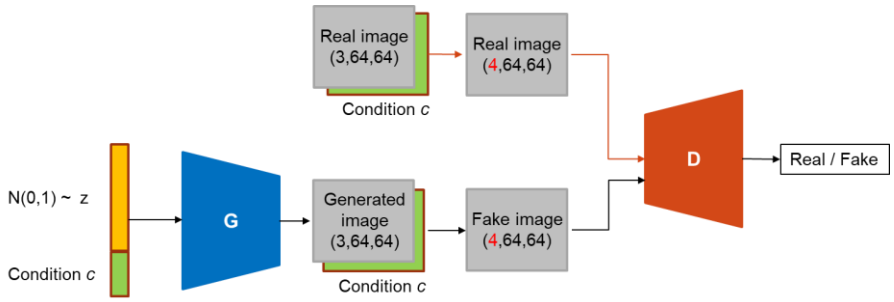
```
# Real images
real_validity = discriminator(real_imgs, labels)
# Fake images
fake_validity = discriminator(fake_imgs, labels)
# Gradient penalty
gradient_penalty = compute_gradient_penalty(
    discriminator, real_imgs.data, fake_imgs.data,
    labels.data)
# Adversarial loss
d_loss = -torch.mean(real_validity) + torch.mean(fake_validity) + lambda_gp * gradient_penalty
d_loss_value = d_loss.item()
```

Loss function of Generator:

```
fake_validity = discriminator(fake_imgs, labels)
g_loss = -torch.mean(fake_validity)
g_loss_value = g_loss.item()
```

3. Conditional DCGAN

Architecture (cGAN):



Architecture (Generator):

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.main = nn.Sequential(
            nn.ConvTranspose2d(latent_dim + n_classes, 512, 4, 1, 0, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, channels, 4, 2, 1, bias=False),
            nn.Tanh(),
        )
```

```
    def forward(self, z, attr):
        attr = attr.view(-1, n_classes, 1, 1)
        x = torch.cat([z, attr], 1)
        return self.main(x)
```

Architecture (Discriminator):

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.feature_input = nn.Linear(n_classes, 64 * 64)
        self.main = nn.Sequential(
            nn.Conv2d(channels + 1, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1, 4, 1, 0, bias=False),
        )

    def forward(self, img, attr):
        attr = self.feature_input(attr).view(-1, 1, 64, 64)
        x = torch.cat([img, attr], 1)
        return self.main(x).view(-1, 1)
```

Loss function of Discriminator:

```python
self.loss = nn.MSELoss()
```

```python
d_real = self.discriminator(real, attr)

fake = self.generator(noise, attr)
d_fake = self.discriminator(fake.detach(), attr)  # not update generato

d_loss = self.loss(d_real, label_real) + self.loss(d_fake, label_fake)
d_loss_value = d_loss.item()
```

Loss function of Generator:

```python
noise.data.resize_(batch_size, latent_dim, 1, 1).normal_(0, 1)
fake = self.generator(noise, attr)
d_fake = self.discriminator(fake, attr)
g_loss = self.loss(d_fake, label_real)  # trick the fake into being real
g_loss_value = g_loss.item()
```

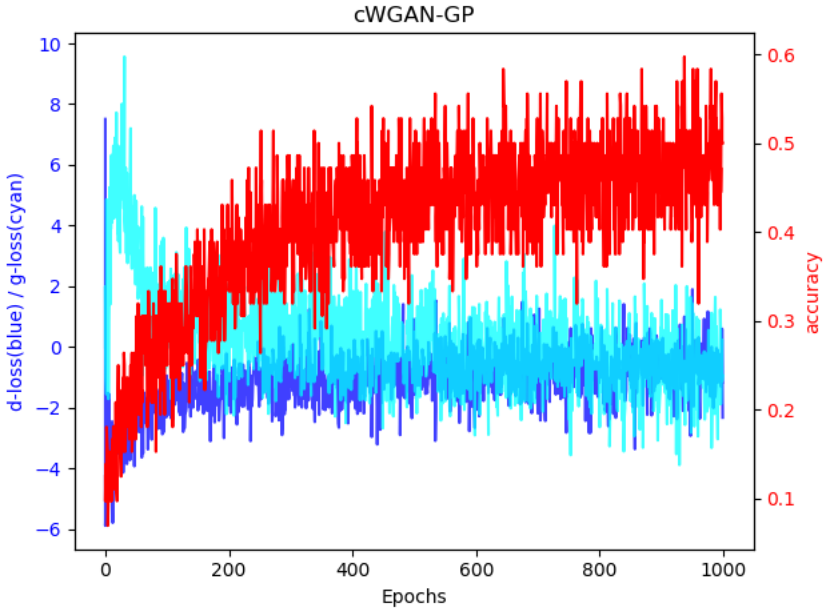
- Specify the hyperparameters (learning rate, epochs, etc.) (5%)
  1. Conditional WGAN

```
channels = 3
image_size = 64
img_shape = (channels, image_size, image_size)
lr = 0.0002
b1 = 0.5
b2 = 0.999
latent_dim = 576
n_epochs = 1000
n_critic = 5
sample_interval = 400
batch_size = 32

cuda = True if torch.cuda.is_available() else False
n_classes = 24

# Loss weight for gradient penalty
lambda_gp = 10
```

2. Conditional DCGAN

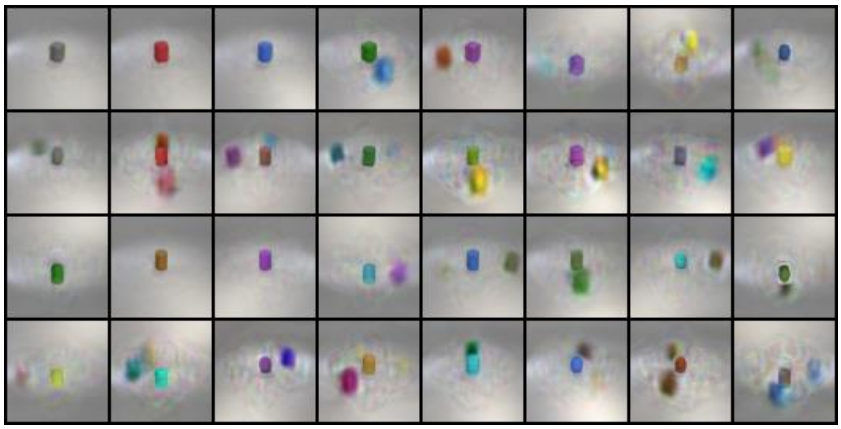
```
channels = 3
image_size = 64
img_shape = (channels, image_size, image_size)
lr = 0.0002
b1 = 0.5
b2 = 0.999
latent_dim = 256
n_epochs = 1000
n_critic = 1
sample_interval = 400
batch_size = 32
# Size of feature maps in generator
ngf = 64
# Size of feature maps in discriminator
ndf = 64

cuda = True if torch.cuda.is_available() else False
n_classes = 24
```

- Results and discussion (20%)
  - Show your results based on the testing data. (5%)
    1. Conditional WGAN

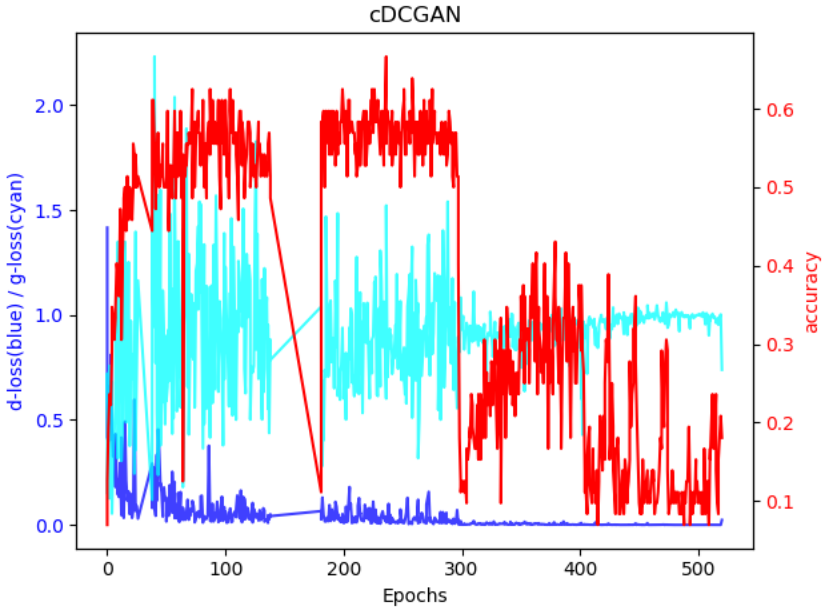


The highest accuracy in training stage is 0.597 at 937 epoch.

The result is shown below.



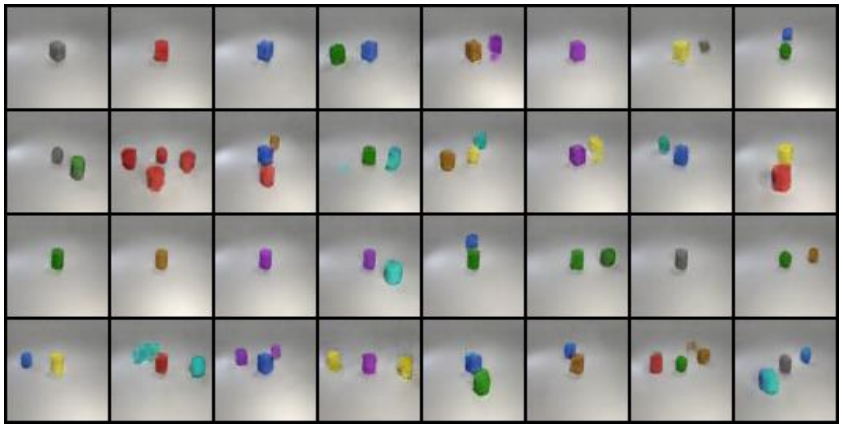
2. Conditional DCGAN



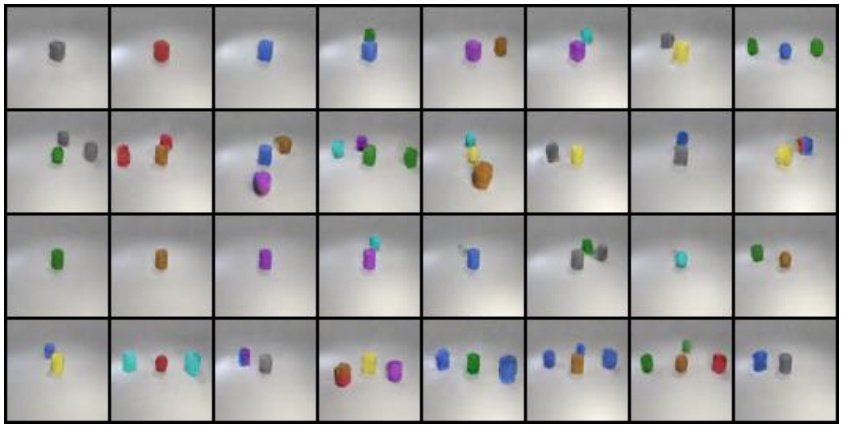
The highest accuracy in training stage is 0.667.

The result is shown below.

Test Accuracy = 0.667

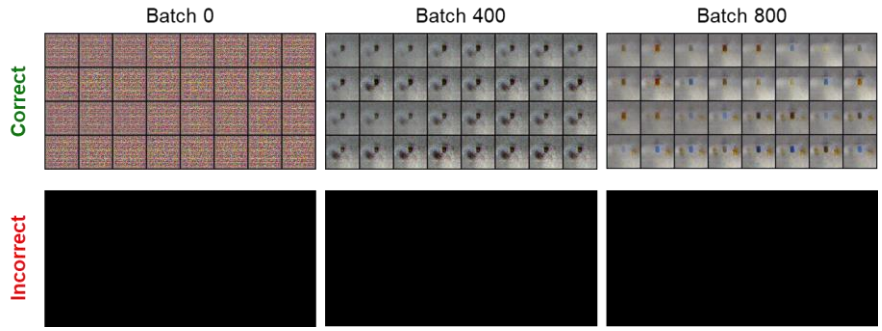


Test Accuracy = 0.722



■ Discuss the results of different models architectures. (15%)

1. According to the paper of WGAN, the results of WGAN would be better than DCGAN. However, in our experimental results, DCGAN is better than WGAN. I think the reason is that I didn't use convolution term in Generator and Discriminator. But, WGAN is more stable than DCGAN in training stage.
2. DCGAN is very unstable. The loss of generator or discriminator may stop increasing or decreasing while set different hyper parameters, e.g., latent-dimension, or update Generator at different interval. For example, if I set latent-dimension 128 and update Generator each 2 batches, the result is correct. However, if I set latent-dimension 128 and update Generator each 5 batches, or set latent-dimension 256 and update Generator each 2 batches, the result would be incorrect.



   Also, the accuracy in training stage is very unstable, it may become very low value, e.g., 0.1 after a higher accuracy, e.g., 0.5.
3. For improving the probability of success, Generator would be update more than one times, that is update Discriminator N times first and then update Generator M times, where $N \geqq 1$ and $M \geqq 1$.