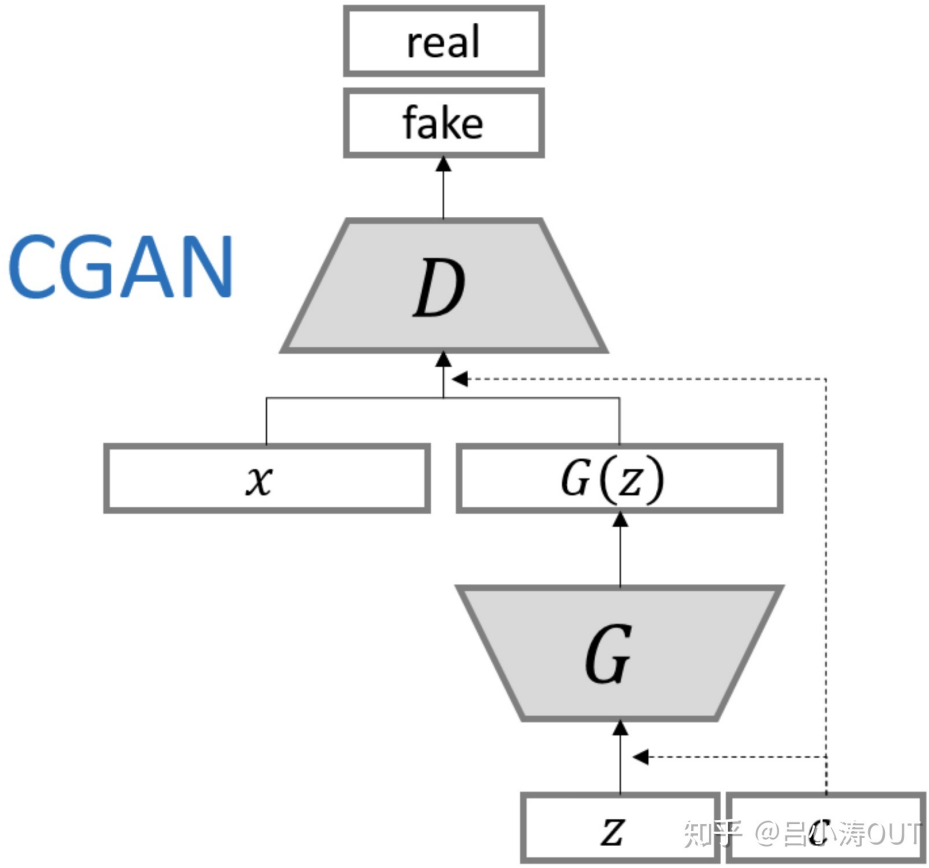
1. Introduction

In this lab, I would like to implement a ***conditional GAN (Generative Adversarial Network)*** to generate sythetic images according to ***multi-label conditions***. GAN is a highly prominent **generative** model, widely used for *style transfer* and *image synthesis* tasks in computer vision. After the synthetic images are generated, we will also use a ***pre-trained classifier*** based on ***ResNet18*** for evaluation purpose.



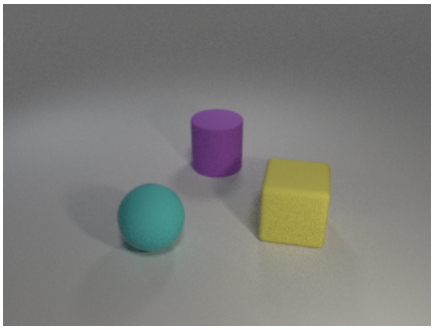
(figure source: **https://www.codeprj.com/zh/blog/aea12c1.html** (https://www.codeprj.com/zh/blog/aea12c1.html))

The above is the basic structure of a ***conditional GAN***. In my implementation, we will use **random noise tensors** z as well as **condition tensors** c as the input of the ***generator*** module G. The outputs of generator are ***synthesized images*** $G(z)$. These fake images along with the true images x would become the inputs of the

discriminator module D. This module acts as a **classifier**, and would try to classfy these figures into either **real images** or **fake images** (synthesized ones).

1.1 Dataset and Conditions

The dataset is composed of a series of images containing objects with different shapes and colors.
The following figure is an example figure:



For each image, there are at most **3** objects and at least **1** object. All the objects within the same image are *different*, either in shape or in color. For each object, there are **3** possible shapes:

1. sphere
2. cube
3. cylinder

and **8** possible colors:

1. gray
2. red
3. blue
4. green
5. purple
6. cyan
7. brown
8. yellow

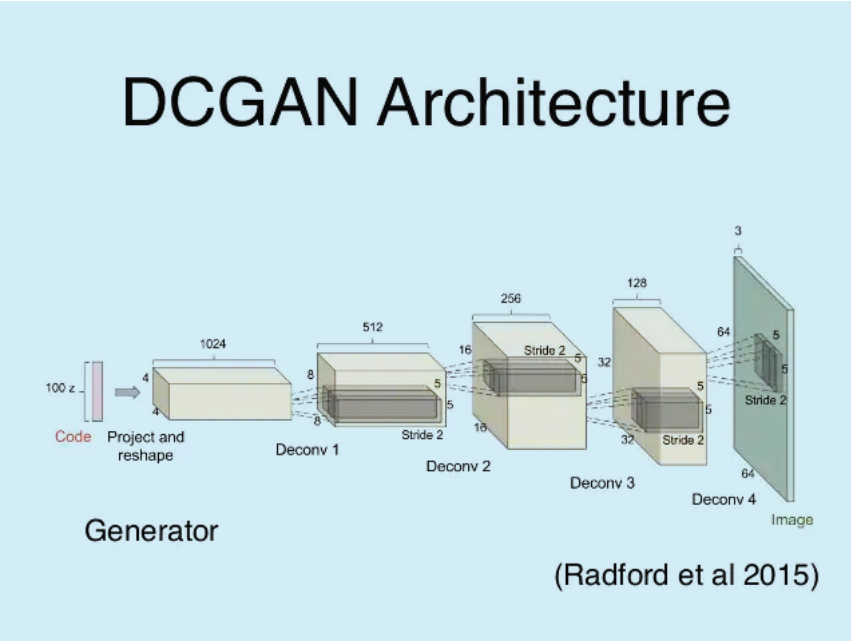
totally $8 \times 3 = 24$ combinations.
There are **18009** training data and 32 for both validation data and testing data. For the training data, aside from objects descriptions `train.json`, which act as ***conditions*** for GAN model here, there are also **ground truth images** as well. As for both validation data `test.json` and testing

data `new_test.json`, there are only object descriptions (*conditions*) for each images. The ground truth images are not available.

2. Implementation Details

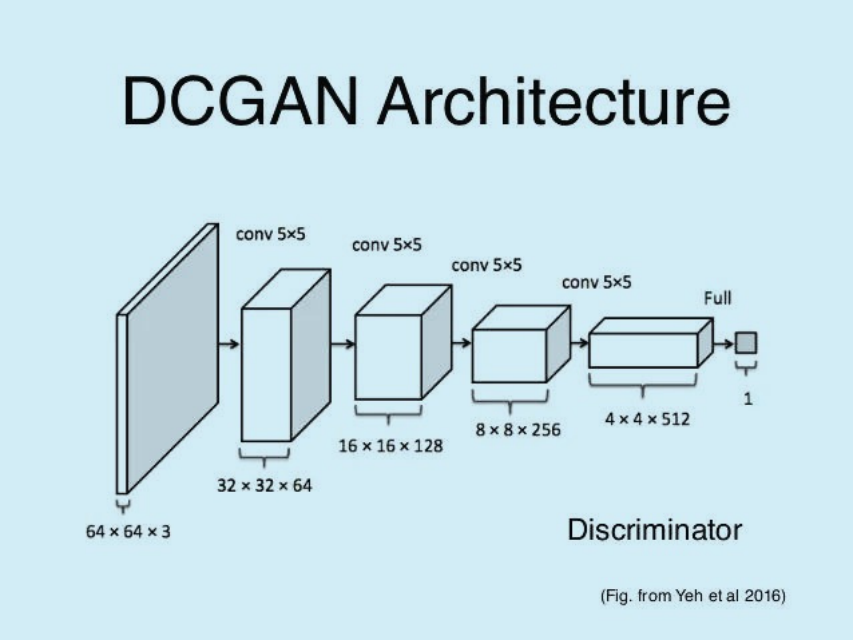
2.1 DCGAN

The model structure of cGAN I chose is ***cDCGAN***, standing for **conditional Deep Convolutional Generative Adversarial Networks**. In DCGAN, the ***generator*** is composed of several **deconvolution layers**, while the ***discriminator*** consists of several **convolution layers**. The reason why I chose this model architecture, is that the ground truth answers are ***images***. Also, DCGAN is easier to implement and more straightforward to understand than other GAN architectures.
This is the architecture of **generator** in the original DCGAN paper (namely, ***without conditions***).



(image source: **https://analyticsarora.com/how-to-implement-dcgan-to-generate-synthetic-samples/** (https://analyticsarora.com/how-to-implement-dcgan-to-generate-synthetic-samples/))
where the input z is a **100-dimensional latent vector**. After a series of projections, reshape and deconvolutions, the output will be a 64×64, 3-channel image.

As for the architecture of discriminator in the original DCGAN (again, ***without conditions***),



(image source: **https://analyticsarora.com/how-to-implement-dcgan-to-generate-synthetic-samples/** (https://analyticsarora.com/how-to-implement-dcgan-to-generate-synthetic-samples/))

A 64×64, 3-channel image will be passed to a series of convolution layers, and finally a single scalar value will be output. The output value is between 0 and 1, indicating whether the discriminator thinks this image is a ***real*** image, or a ***fake*** image generated by the generator.
Other major features are:

1. **There are no pooling layers or upsampling layers**. For increasing/decreasing the size of feature maps, the ***fractional-strided convolutions*** are used in the ***generator*** while the ***strided convolutions*** are used in the ***discriminator***.
2. ***Batch Normalization*** is used between several layers in both **generator** and **discriminator**.
3. As for the activation functions, ***Leaky ReLU*** is used for most layers in the **discriminator** except the last one, while ***ReLU*** is used for most layers in the **generator** except the last one.
4. For the last layer before output in the **generator**, ***hyperbolic tangent function*** is used, because the image read by `PIL.Image.open()` has pixel values fall into **[0, 1]** interval, but after the **normalization**

transform with $mean = 0.5$ and $standard\ deviation = 0.5$ for each RGB channels, all pixel values for all 3 channels fall into the range **[-1, 1]**.

```
self.transformations = transforms.Compose([transforms.Resize((64, 64)),
transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

So, `nn.Tanh()` is used as the activation function for the last layer.

5. For the last layer before output in the **discriminator**, ***sigmoid function*** is used, because the output value should fall into **[0, 1]** interval, where the value close to **0** denotes **fake images**. Otherwise, the value should be close to **1**.

So, `nn.Sigmoid()` is used as the activation function for the last layer.

(P.S. Regarding the reason for resizing images into size $(64, 64)$, it is because the resolution of input for pretrained classifier is 64x64.)

Following, I will elaborate how I implemented **conditions** into the structure of my cDCGAN model, as well as the training details including the loss function.

2.2 Generator

Before we get into the code, I need to clarify some settings. First, the dimension for latent vector z is **100**, which is just the same value used in the original DCGAN paper.

```
z_dim = 100
```

As for the condition c for each image, it is a **24**-d tensor originally, because it is actually an ***one-hot vector*** of all **24** possible combinations of 8 colors and 3 shapes. Again, since there are at most **3** and at least **1** object(s) in each figure, there are **1~3** ones in each c, with other dimensions being zeros.

However, I do not directly input the 24-d tensors into the model. I will first ***expand*** the dimension up to **300** via a ***linear layer*** with **ReLU** as the activation function before

further processes. The purpose is to **expand the space to save more encoded information** about the condition given.

```
c_dim = 300 # in main.py
# in model.py
class Generator(nn.Module):
    def __init__(self, z_dim, c_dim):
        ......
        self.expand = nn.Sequential(
            nn.Linear(24, c_dim),
            nn.ReLU()
        )
        ......
```

My generator has similar architecture as the DCGAN in the original paper. The only difference is the **channels of feature maps** in each layer, since I need to consider the **conditions tensors**.

```
class Generator(nn.Module):
    def __init__(self, z_dim, c_dim):
        super().__init__()
        self.z_dim = z_dim
        self.c_dim = c_dim
        self.expand = nn.Sequential(
            nn.Linear(24, c_dim),
            nn.ReLU()
        )
        channels = [z_dim + c_dim, 512, 256, 128, 64]
        paddings = [0, 1, 1, 1]
        strides = [2, 2, 2, 2]
        for i in range(1, len(channels)):
            setattr(self, "deconv" + str(i), nn.Sequential(
                nn.ConvTranspose2d(channels[i-1], channels[i], 4, strides[i-1],
                                   paddings[i-1]),
                nn.BatchNorm2d(channels[i]),
                nn.ReLU()
            ))
        self.deconv5 = nn.Sequential(
            nn.ConvTranspose2d(64, 3, 4, 2, 1),
            nn.Tanh()
        )

    def forward(self, z, c):
        # reshape to (B, z_dim, 1, 1), and then deconvs to get back to a fake imag
        z = z.view(-1, self.z_dim, 1, 1)
        c = self.expand(c).view(-1, self.c_dim, 1, 1)
        output = torch.cat((z, c), 1) # (B, z_dim + c_dim, 1, 1)
        output = self.deconv1(output) # (B, 512, 4, 4)
        output = self.deconv2(output) # (B, 256, 8, 8)
        output = self.deconv3(output) # (B, 128, 16, 16)
        output = self.deconv4(output) # (B, 64, 32, 32)
        output = self.deconv5(output) # (B, 3, 64, 64), in [-1, 1]
        return output

    def init_weights(self, mean = 0, std = 0.02):
        for m in self._modules:
            if isinstance(self._modules[m], nn.ConvTranspose2d):
                self._modules[m].weight.data.normal_(mean, std)
                self._modules[m].bias.data.zero_()
```

To start using the **generator**, one must initialize the weights and biases of all **deconvolution layers**, like so in `main.py` .

```
generator = Generator(z_dim, c_dim).to(device)
generator.init_weights()
```

Here, the default settings are to initialize biases as **0** and initialize weights as values sampled from the specific Gaussian $N(\mu = 0, \sigma = 0.02)$, which is suggested by the original DCGAN paper.
The default batch size B is **64** in my implementation, so the shape of input latent tensors z is $(B = 64, z\_dim = 100)$, as specified before. To make it into a **feature-map-like** tensor, we will first reshape it into the shape $(64, 100, 1, 1)$. Now, z can be deemed as 1×1 feature maps with **100** channels.
As for the conditions c, just like we mentioned before, we will first **expand** it from $(B = 64, 24)$ into $(64, c\_dim = 300)$ via a linear layer, to *expand the space for the purpose of storing more encoded information*. Also, with the intention of concatenating it with z later on, we will also reshape c into a **feature-map-like** tensor. Thus, the shape becomes $(64, 300, 1, 1)$. Now, c can be deemed as 1×1 feature maps with **300** channels.
Concatenated at dimension **1**, we get **64** 1×1 feature maps with **400** channels each. The current tensor shape is $(64, 400, 1, 1)$.
Now, this tensor will go through several **deconvolution layers**. The height and width of feature maps will basically grow **twice** as large after each layer.
The **kernel sizes** of all deconvolution layers are **4**, and the **strides** are all **2** pixels. All deconvolution layers except the first one have **1-pixel padding**. Also, as aforementioned in section ***2.1 DCGAN*** above, there are **batch normalization** layers in between every 2 deconvolution layers. Finally, all deconvolution layers except the last one use **ReLU** function as the activation functions. The last deconvolution layer uses **Hyperbolic Tangent** function as the activation function.

1. Input size $(64, 400, 1, 1) \rightarrow$ **deconvolution layer 1** (no padding) $\rightarrow (64, 512, 4, 4)$

2. $(64, 512, 4, 4) \rightarrow$ **deconvolution layer 2** $\rightarrow$ $(64, 256, 8, 8)$
3. $(64, 256, 8, 8) \rightarrow$ **deconvolution layer 3** $\rightarrow$ $(64, 128, 16, 16)$
4. $(64, 128, 16, 16) \rightarrow$ **deconvolution layer 4** $\rightarrow$ $(64, 64, 32, 32)$
5. $(64, 64, 32, 32) \rightarrow$ **deconvolution layer 5** $\rightarrow$ Output shape $(64, 3, 64, 64)$

Each output tensoor can be regarded as a generated image having **3** channels, corresponding to **RGB** channels, and having the size of (64×64).

2.3 Discriminator

Again, the dimension of latent vectors z is **100**, while that of conditions c is originally **24**, but will be expand to **300** for more space to store encoded information. The expansion is done by a **linear** layer with **Leaky ReLU** function as the activation function, see the `self.expand` part below.

```
class Discriminator(nn.Module):
    def __init__(self, img_shape, c_dim):
        super().__init__()
        self.H, self.W, self.C = img_shape
        self.expand = nn.Sequential(
            nn.Linear(24, self.H * self.W),
            nn.LeakyReLU()
        )
        channels = [4, 64, 128, 256, 512]
        for i in range(1, len(channels)):
            setattr(self, "conv" + str(i), nn.Sequential(
                nn.Conv2d(channels[i-1], channels[i], 4, 2, 1),
                nn.BatchNorm2d(channels[i]),
                nn.LeakyReLU()
            ))
        self.conv5 = nn.Sequential(
            nn.Conv2d(512, 1, 4),
            nn.Sigmoid()
        )

    def forward(self, imgs, c):
        c = self.expand(c).view(-1, 1, self.H, self.W) # c becomes (B, 1, H=64, W=
        output = torch.cat((imgs, c), 1) # (B, 4, 64, 64), 4 because 3+1
        output = self.conv1(output) # (B, 64, 32, 32)
        output = self.conv2(output) # (B, 128, 16, 16)
        output = self.conv3(output) # (B, 256, 8, 8)
        output = self.conv4(output) # (B, 512, 4, 4)
        output = self.conv5(output) # (B, 1, 1, 1) output = binary in [0, 1]
        output = torch.squeeze(output) # (B,)
        return output

    def init_weights(self, mean = 0, std = 0.02):
        for m in self._modules:
            if isinstance(self._modules[m], nn.Conv2d):
                self._modules[m].weight.data.normal_(mean, std)
                self._modules[m].bias.data.zero_()
```

To start using the **discriminator**, one must initialize the weights and biases of all **convolution layers**, like so in `main.py` .

```
discriminator = Discriminator((64, 64, 3), c_dim).to(device)
discriminator.init_weights()
```

Here, the default settings are to initialize biases as **0** and initialize weights as values sampled from the specific Gaussian $N(\mu = 0, \sigma = 0.02)$, which is suggested by the original DCGAN paper.
The default batch size B is **64** in my implementation. The images have size 64×64 and **3** channels for RGB colors. Again, this size is the resolution of input for the **ResNet18 evaluator**.
Therefore, the shape of images input in the discriminator is $(64, 3, 64, 64)$.
As for the conditions *c*, we will first **expand** it from $(B = 64, 24)$ into $(64, H \times W = 64 \times 64 = 4096)$ via a linear layer, to *expand the space for the purpose of storing more encoded information*. Also, with the intention of

concatenating it with the input images later on, we will also reshape c into a **feature-map-like** tensor. Thus, the shape becomes $(64, 1, 64, 64)$. Now, c can be deemed as 64×64 feature maps with only **1** channel.
Concatenated at dimension **1**, we get **64** 64×64 resolution feature maps with $3 + 1 = 4$ channels each (***RGB*** and ***condition*** channels). The current tensor shape is $(64, 4, 64, 64)$.
Now, this tensor will go through several **convolution layers**. The height and width of feature maps will basically shrink **twice** as large after each layer.
The **kernel sizes** of all convolution layers are **4**, and the **strides** are all **2** pixels. All convolution layers have **1-pixel padding**. Also, as aforementioned in section ***2.1 DCGAN*** above, there are **batch normalization** layers in between every 2 convolution layers. Finally, all convolution layers except the last one use **Leaky ReLU** function as the activation functions. The last convolution layer uses **Sigmoid** function as the activation function.

1. Input size $(64, 4, 64, 64) \rightarrow$ **convolution layer 1** $\rightarrow$ $(64, 64, 32, 32)$
2. $(64, 64, 32, 32) \rightarrow$ **convolution layer 2** $\rightarrow$ $(64, 128, 16, 16)$
3. $(64, 128, 16, 16) \rightarrow$ **convolution layer 3** $\rightarrow$ $(64, 256, 8, 8)$
4. $(64, 256, 8, 8) \rightarrow$ **convolution layer 4** $\rightarrow (64, 512, 4, 4)$
5. $(64, 512, 4, 4) \rightarrow$ **convolution layer 5** $\rightarrow$ Output shape $(64, 1, 1, 1)$

After an extra `torch.squeeze()` layer that discards unnecessary **1-dimensionalities**, the final output shape is $(64,)$.
Each dimension correspond to an image in the input mini-batch. All values are within **[0, 1]** interval, indicating whether the corresponding image is real or not, judged by the **discriminator**.

2.4 Loss function

Here, I used the ***binary cross-entropy*** as the loss function, since the task is in fact a **binary classification** task for classfying whether the given image is **real** or **synthesized**.
Let n be the input size (typically batch size 64, but smaller with the last batch), y_i be the prediction and $\hat{y}_i$ be the corresponding ground truth. Both $y_i, \hat{y}_i \in [0, 1],\ \forall i = 1, \dots, n$.

$$BCE = -\frac{1}{n}(\sum_{i=1}^{n} \hat{y}_i \log y_i + (1 - \hat{y}_i) \log(1 - y_i))$$

```
import torch.nn as nn
loss_fn = nn.BCELoss()
```

I did not apply other auxiliary losses, sine the result is already at an acceptable level.

```
# in main.py
for e in range(epochs):
    generator_loss, discriminator_loss = 0, 0
    ......
    for i, (imgs, objs) in enumerate(train_dataloader):
        ......
        '''train discriminator first'''
        # imgs are ground truth images, objs are conditions.
        predictions = discriminator(imgs, objs)
        loss_reals = loss_fn(predictions, reals) # reals is all ones tensor.

        z = torch.randn(batch_size, z_dim).to(device) # latent tensors z
        fake_imgs = generator(z, objs) # synthesized images
        predictions = discriminator(fake_imgs.detach(), objs)
        loss_fakes = loss_fn(predictions, fakes) # fakes is all zeros tensor.

        '''loss for discriminator in this batch'''
        discriminator_loss_batched = loss_reals + loss_fakes
        ......

        '''train generator later'''
        for _ in range(4):
            ......
            z = torch.randn(batch_size, z_dim).to(device) # latent tensors z
            fake_imgs = generator(z, objs) # synthesized images
            predictions = discriminator(fake_imgs, objs) # judged by discriminator
            '''loss for generator in this batch'''
            generator_loss_batched = loss_fn(predictions, reals) # all 1s in reals
            ......
        '''total generator and discriminator losses in current epoch
        (not averaged yet, averaged later in output writing part)'''
        generator_loss += generator_loss_batched.item()
        discriminator_loss += discriminator_loss_batched.item()
        ......
```

So, In every epoch, the total losses for generator and discriminator are **averaged** from the losses of **all mini-batches**.

During the training of discriminator, the generator is fixed. The loss for the **discriminator** comes from 2 parts: **the loss yielded while failing to recognize the real images**, denoted `loss_reals` in my code implementation, and **the loss yielded while failing to see through the fake images**, denoted `loss_fakes` in my code.
During the training of generator, the discriminator is fixed. The loss for **generator** occurs while **synthesized images are exposed and rejected by the discriminator**.
Here we only focus on the details of ***losses***. For those who are interested in training details, please refer to the later section.

2.5 Hyper-parameters Specification

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
z_dim = 100
c_dim = 300
epochs = 500
lr_generator = 1e-4
lr_discriminator = 4e-4
batch_size = 64
loss_fn = nn.BCELoss()
```

1. The dimension for latent vectors z is **100**, which is suggested in the original DCGAN paper.
2. The orignal dimension for conditions tensors c is **24** for all 24 combinations of shapes and colors. Since we need to expand the space for storing more encoded information, we will expand the dimensionality to **300** when input into the **generator**.
   For the conditions input c in the **discriminator**, we will expand it to be a **64*64 = 4096-d** tensor for the same purpose. Here the conditions would act as a single channel.
3. Total training epochs are **500**, ensuring sufficient time for convergence.
4. The learning rate for the **generator** is 10^{-4} while that for the **discriminator** is 4×10^{-4}. The difference comes from the **proportion of training iterations for generator to discriminator**. Since limited training time and that we practically value the **generator** more, we

trained the generator *more times* in an iteration.
To balance the effect, the learning rates are set to ***inversly proportional*** to this proportion.

> In my code inplementation, the ratio of training steps for ***generator*** to those for ***discriminator*** in an iteration is $4:1$.

5. The default batch size is **64**.
6. The loss function used is ***binary cross entropy***, for the task being a *binary classification problem*.

2.6 Training Processes

```
# in main.py
'''experimental settings'''
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
z_dim = 100
c_dim = 300
epochs = 500
lr_generator = 1e-4
lr_discriminator = 4e-4
batch_size = 64
loss_fn = nn.BCELoss()

# training data (containing conditions and ground truth images)
train_dataset = CLEVRDataset("D://DL_lab5_data/iclevr")
train_dataloader = DataLoader(train_dataset, batch_size = batch_size,
shuffle = True, num_workers = 4)

# create models and initialize
generator = Generator(z_dim, c_dim).to(device)
discriminator = Discriminator((64, 64, 3), c_dim).to(device)
evaluator = evaluation_model()
generator.init_weights() # (0, 0.02)
discriminator.init_weights() # (0, 0.02)

# optimizer
optimizer_generator = optim.Adam(generator.parameters(), lr = lr_generator)
optimizer_discriminator = optim.Adam(discriminator.parameters(),
lr = lr_discriminator)

'''training'''
# ground truth labels, reals is all 1s tensor and fakes is all 0s tensor.
reals = torch.ones(batch_size).to(device)
fakes = torch.zeros(batch_size).to(device)
# the last batch might be smaller
reals_last_batch = torch.ones(len(train_dataloader.dataset) % batch_size).to(devi
fakes_last_batch = torch.zeros(len(train_dataloader.dataset) % batch_size).to(dev

# get the conditions for evaluation data
test_objs = test_obj_getter().to(device) # test.json
# latent vectors used in evaluation
z_for_evaluation = torch.randn(len(test_objs), z_dim).to(device)

for e in range(epochs):
        generator_loss, discriminator_loss = 0, 0
        generator.train()
        discriminator.train()

        for i, (imgs, objs) in enumerate(train_dataloader):
            imgs = imgs.to(device)
            objs = objs.to(device)

            '''train discriminator first'''
            optimizer_discriminator.zero_grad()

            # for real images
            predictions = discriminator(imgs, objs)
            if predictions.size(0) == batch_size:
                loss_reals = loss_fn(predictions, reals)
            else: #last batch
                loss_reals = loss_fn(predictions, reals_last_batch)
            # for fake images
            if predictions.size(0) == batch_size:
                z = torch.randn(batch_size, z_dim).to(device)
            else:
                z = torch.randn(len(train_dataloader.dataset) % batch_size, z_dim
                .to(device)
            fake_imgs = generator(z, objs)
            predictions = discriminator(fake_imgs.detach(), objs)
            if predictions.size(0) == batch_size:
                loss_fakes = loss_fn(predictions, fakes)
            else:
                loss_fakes = loss_fn(predictions, fakes_last_batch)

            discriminator_loss_batched = loss_reals + loss_fakes
            discriminator_loss_batched.backward()
            optimizer_discriminator.step()

            '''train generator later'''
            for _ in range(4):
                optimizer_generator.zero_grad()
```

```
                # generate fake images and compute loss w.r.t. real images
                if predictions.size(0) == batch_size:
                    z = torch.randn(batch_size, z_dim).to(device)
                else:
                    z = torch.randn(len(train_dataloader.dataset) % batch_size,
                    z_dim).to(device)
                fake_imgs = generator(z, objs)
                predictions = discriminator(fake_imgs, objs)

                if predictions.size(0) == batch_size:
                    generator_loss_batched = loss_fn(predictions, reals)
                else:
                    generator_loss_batched = loss_fn(predictions, reals_last_batc
                generator_loss_batched.backward()
                optimizer_generator.step()

            generator_loss += generator_loss_batched.item()
            discriminator_loss += discriminator_loss_batched.item()

        '''evaluation'''
        generator.eval()
        discriminator.eval()

        with torch.no_grad():
            fake_imgs = generator(z_for_evaluation, test_objs)
        score = evaluator.eval(fake_imgs, test_objs)

        if score > best_score:
            best_score = score
            best_models_weights = deepcopy(generator.state_dict())
            torch.save(best_models_weights, os.path.join(model_dir,
            "model_epoch_{:03d}_score_{:.2f}.pt".format(e+1, score)))
            save_image(fake_imgs, os.path.join(result_img_dir,
            "epoch_{:03d}.png".format(e+1)), nrow = 8, normalize = True)
```

Training dataset is composed of **(image, condition)** pairs, totally **18009** pairs. The default batch size is **64**, hence there are 282 mini-batches, and the last one is smaller (size = 25). Due to different sizes of mini-batches, the processes of **generating latent vectors** z and **calculating losses** will be considered in 2 cases: mini-batch size = **64** or **25**.

Training Phase

In the training phase, we train the **discriminator first**, and fix the generator.
As mentioned before, there are 2 different sources contribute to the loss of **discriminator**: **the loss from failure to recognize the real images**, denoted `loss_reals` in my code implementation, and **the loss from failure to see through the fake images**, denoted `loss_fakes` in my code.
First, feed the real images to the discriminator. The predictions (judgements) are then compared with the

label `reals`, which is just a **all 1s' tensor** (since the inputs are ground truth images). The binary cross entropy is then computed as the loss `loss_reals`.
And then, we will randomly generate a **latent tensor** z which has the size of $(B, z\_dim = 100)$. B is either 64 or 25. Every values in z are sample from the standard Normal distribution $N(0, 1)$ via `torch.randn()`.
The **latent tensors** z together with the **conditions** c are then passed to the fixed generator. The generator will generate synthesized images. These *fake* images are then fed to the discriminator. The predictions (judgements) are then compared with the labels `fakes`, which is just a **all 0s' tensor** (since the inputs are all synthesized images). The binary cross entropy is then computed as the loss `loss_fakes`.
Finally, both the losses from the real images `loss_reals` and from the fake images `loss_fakes` will then be added up as the total loss for the **discriminator**. We will use this summed up loss to conduct backpropagation.
After the discriminator is trained, we **then** train the **generator**, with the discriminator fixed. Just as I brought up at the previous section, in each iteration (training a mini-batch), ***the discriminator is trained*** 1 ***time while the generator is trained*** 4 ***times***. During each of these 4 times, **latent tensors** z sampled from standard Gaussian distribution along with the **condition tensors** c are given to the generator as the inputs. The generator output **fake images**, which will then *judge* by the previously trained discriminator. The predictions (judgements) made by the discriminator are then used to calculate the binary cross entropy, which happens to be the total loss for the **generator**. We will use this loss to execute backpropagation.
Note that while calculating the loss for the generator, the **all 1s' tensor** labels `reals` is used here. This is because the generator should try to synthesize images ***just like the real ones***. So, from the perspective of generator, it should expect the discriminator being deceived and classfying the fake images as *real ones*.

Validation Phase

In validation phase, similarly a random latent vector z having size $(32, 100)$ will be sampled from $N(0, 1)$ first. **100** is the dimensionality of latent vector and **32** is the size of validation data.
And then, z together with the conditions c will be used to synthesized 32 fakes images by the trained **generator**. These generated images as well as the corresponding conditions will then be sent to the **evaluator** module, which is just a **pre-trained ResNet-18 classifier** from the `evaluator.py` in the sample code. The evaluator will output a validation **score**.
If the score is the current best one, we will renew the highest score and update the best model weights by saving them. Also, the best models will be used to draw the synthesize images with the **32** conditions given in the validation data. These images will be shown later in the ***Result and Discussion*** section.

3. Results and Discussion

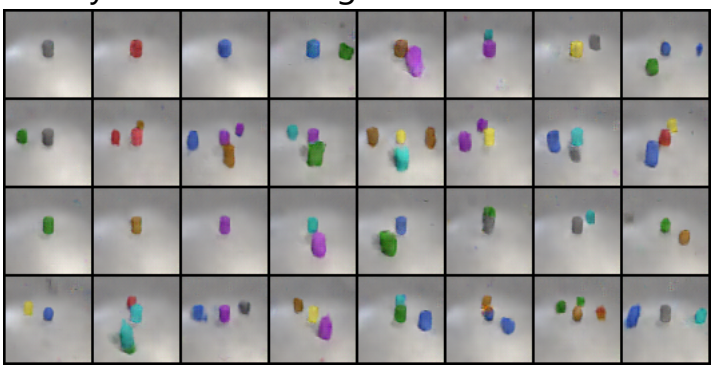
3.1 Scores and Synthesized Images

The following is the scores tested on the validation data ***test.json*** and the testing data ***new_test.json***. We conducted **3** inferences for each of them, and the scores are averaged over these 3 inferences. The reason is that the latent vectors z are randomly sampled every time.

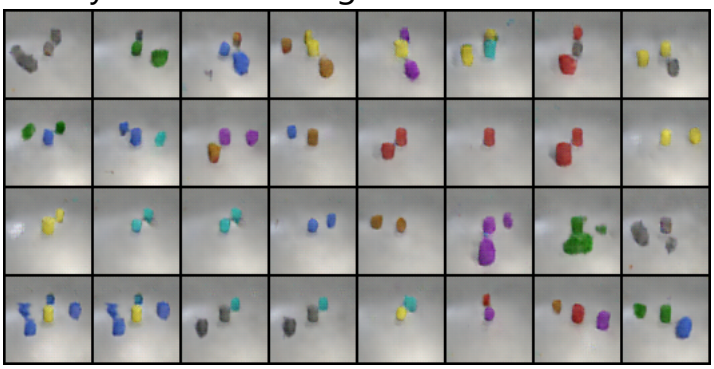
```
(base) C:\Users\Witty\Downloads\Lab5>C:/Users/Witty/anaconda3/python.exe c:/Users/Witty/Downloads/Lab5/test.py --path test.json
average score = 0.72

(base) C:\Users\Witty\Downloads\Lab5>C:/Users/Witty/anaconda3/python.exe c:/Users/Witty/Downloads/Lab5/test.py --path new_test.json
average score = 0.73
```

For the **validation** data, the average score is **0.72**. For the **testing** data, the average score is **0.73**.

The synthesized images for conditions in **test.json**

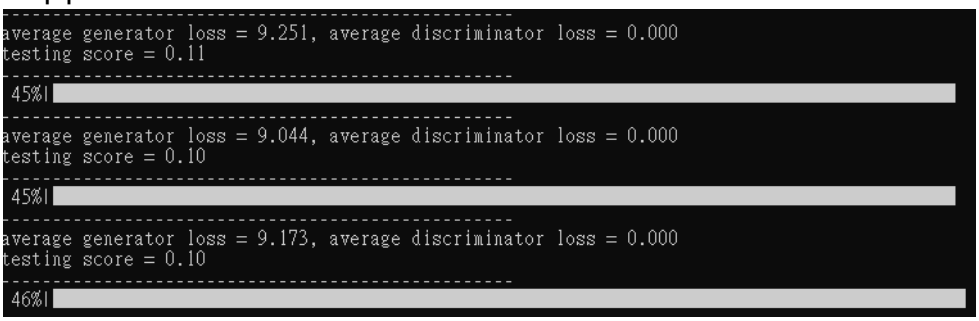


The synthesized images for conditions in **new_test.json**



3.2 Discussion of Different Models Architectures

Before the current hyper-parameters settings, I actually have tried different values. There is once a weird failure happened.



Before the training epochs shown in the above figure, the validation score was at around **0.4~0.5** and was slowly but steadily growing. Nevertheless, starting from some epoch, the loss of the **generator** surges, while the loss of the **discriminator** plunges to **0**. In the meantime, the scores also skydives from **0.4~0.5** to around only **0.1**. The training processes completely fail from that epoch, and do not recover later.

After some research on the web, I conjecture this might be the so-called **convergence failure** problem. In this case, the **discriminator** dominates, overpowering the

generator. Discriminator classfies most of the images correctly. In turn, the generator cannot produce any image that fool the discriminator and thus fail to learn.
I suspect this might be due to the fast learning of the **discriminator**. Originally, I set the learning rates of both the **Adam optimizers** for the generator and the discriminator to be 2×10^{-4}. However, the problem solved when I change the learning rates into my current settings,

1. learning rate of the optimizer for the **generator** = 1×10^{-4}
2. learning rate of the optimizer for the **discriminator** = 4×10^{-4}

This shows that **the GAN is very sensitive to the change of hyper-parameters values, even it is as slight as a hair**.

Some differences between DCGAN and ordinary GAN

1. In **DCGAN**, the **batch normalization layers** are used between several layers in both ***generator*** and ***discriminator***.
   This can solve the *bad initialized values* problem, and prevent the **mode collapse** problem at the same time.
2. The generator and the discriminator are connected directly with the **convolution** and **deconvolution** layers, without the usage of fully-connected layers. This is why the modules are called ***DCGAN***.
3. As for the activation functions, ***Leaky ReLU*** is used for most layers in the **discriminator** except the last one, while ***ReLU*** is used for most layers in the **generator** except the last one.
4. For the last layer before output in the **generator**, ***hyperbolic tangent function*** is used, because the image read by `PIL.Image.open()` has pixel values fall into **[0, 1]** interval, but after the **normalization transform** with $mean = 0.5$ and $standard\ deviation = 0.5$ for each RGB channels, all pixel values for all 3 channels fall into the range **[-1, 1]**.

So, `nn.Tanh()` is used as the activation function for the last layer.

5. For the last layer before output in the **discriminator**, ***sigmoid function*** is used, because the output value should fall into **[0, 1]** interval, where the value close to **0** denotes **fake images**. Otherwise, the value should be close to **1**.
   So, `nn.Sigmoid()` is used as the activation function for the last layer.
6. **There are no pooling layers or upsampling layers**.
   For increasing/decreasing the size of feature maps, the ***fractional-strided convolutions*** are used in the ***generator*** while the ***strided convolutions*** are used in the ***discriminator***.

The design of conditions

Recall that in the **generator**, the dimensionality of the conditions c is expanded from the original **24** to **300**. As I explained earlier, the purpose is to create more space available for storing encoded information of the conditions. This also allows the generator in cDCGAN generate images with much more *variety*.

About the **discriminator**, the conditions c are expanded from the original **24** to **4096**, which is just 64×64. In my design of conditions, I treat these conditions as an **extra channel**, which tells the discriminator *how the ground truth should look like*, i.e. *what objects should be contained*. The discriminator would then based on this channel, to see if the other 3 channels above (**RGB**) form a ***real*** image.