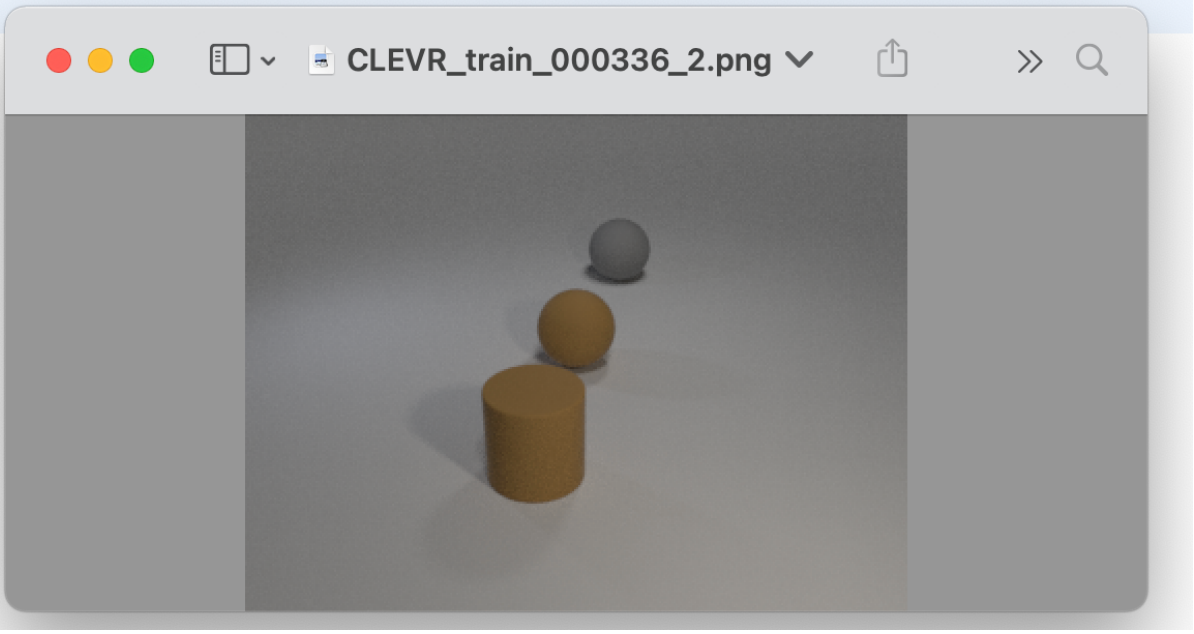
Introduction:

本次作業是要實作一個 conditional GAN 產生出所需要的圖片，condition labels 的種類共有 24 個，一種物體對應著一種 label，且有時可能會同時出現多個 condition 在同一個圖片當中，也就是說 model 要有能夠處理 multi-labels 的能力。

在訓練集中，種共有 18009 個 images，每張 image 都有對應的 label，測試集集是給予 32 個 label 用來當作 condition 作為給予 model 的輸入如圖一，其 image 與 labels 的對應關係如圖二。

```
2  "CLEVR_train_000336_2.png": ["brown sphere", "gray sphere", "brown cylinder"],
3
```



圖一，圖片與 labels 的對應。

```
[["gray cube"], ["red cube"], ["blue cube"], ["blue cube", "green cube"], ["brown
    cube", "purple cube"], ["purple cube", "cyan cube"], ["yellow cube", "gray
    sphere"], ["blue sphere", "green sphere"], ["green sphere", "gray cube"],
    ["brown sphere", "red cube", "red cylinder"], ["purple sphere", "brown
    cylinder", "blue cube"], ["cyan sphere", "purple cylinder", "green cube"],
    ["yellow sphere", "cyan cylinder", "brown cube"], ["gray cylinder", "yellow
    cylinder", "purple cube"], ["blue cylinder", "gray cube", "cyan cube"],
    ["blue cylinder", "red cube", "yellow cube"], ["green cylinder"], ["brown
    cylinder"], ["purple cylinder"], ["cyan cylinder", "purple cylinder"], ["blue
    cylinder", "green cylinder"], ["gray cylinder", "green cube"], ["cyan
    sphere", "gray cylinder"], ["brown sphere", "green sphere"], ["blue sphere",
    "yellow cylinder"], ["red sphere", "cyan cylinder", "cyan cube"], ["gray
    sphere", "purple cylinder", "blue cube"], ["yellow cube", "brown cylinder",
    "purple cube"], ["cyan cube", "green cylinder", "blue cube"], ["brown cube",
    "blue cylinder", "blue sphere"], ["green sphere", "red cylinder", "brown
    sphere"], ["blue cylinder", "gray cylinder", "cyan sphere"]]
```

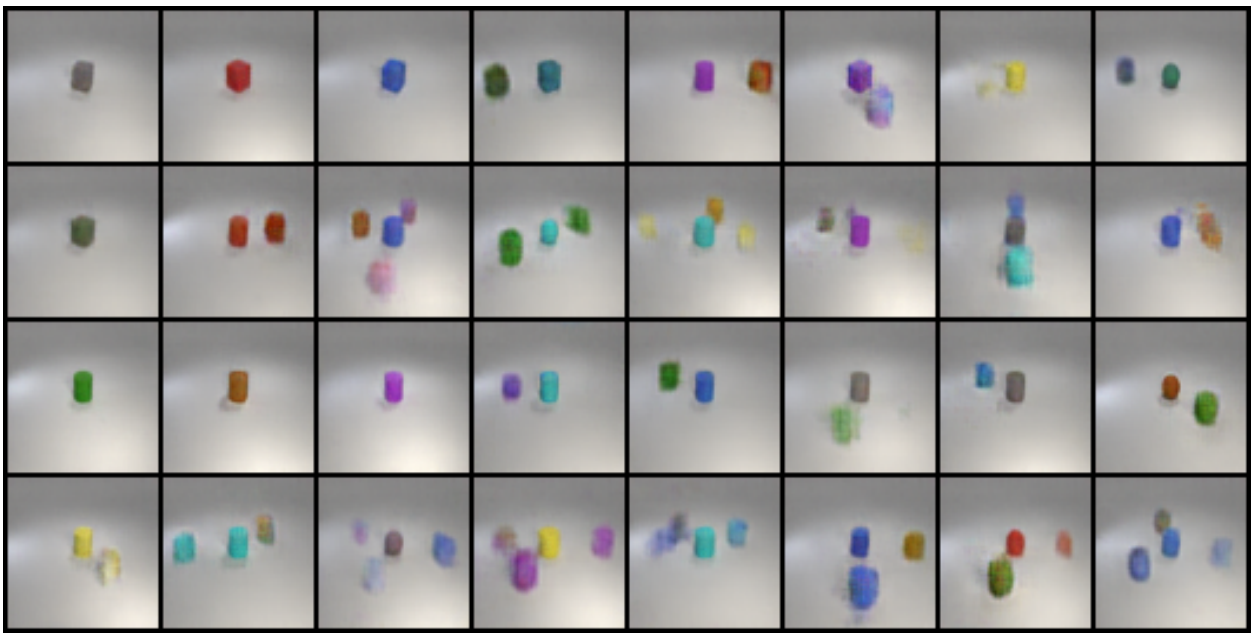
圖二，測試集所提供的 labels。

為了方便使 lables 送入 model 中，我們還需要將 condition 轉換成長度為 24 的 one hot vector，圖三則為用於轉換之 table。

```
{"gray cube": 0, "red cube": 1, "blue cube": 2, "green cube": 3,
    "brown cube": 4, "purple cube": 5, "cyan cube": 6, "yellow
    cube": 7, "gray sphere": 8, "red sphere": 9, "blue sphere":
    10, "green sphere": 11, "brown sphere": 12, "purple sphere":
    13, "cyan sphere": 14, "yellow sphere": 15, "gray cylinder":
    16, "red cylinder": 17, "blue cylinder": 18, "green
    cylinder": 19, "brown cylinder": 20, "purple cylinder": 21,
    "cyan cylinder": 22, "yellow cylinder": 23}
```

圖三，用於將 condition labels 轉換至 one hot vector 的 table

再經由多次訓練之後，model 便可以依照送進來的 condition，做相應的圖片生成，如圖四為模型經由 58 次迭代的結果。



圖四，實驗生成圖片之一，於 epoch 58

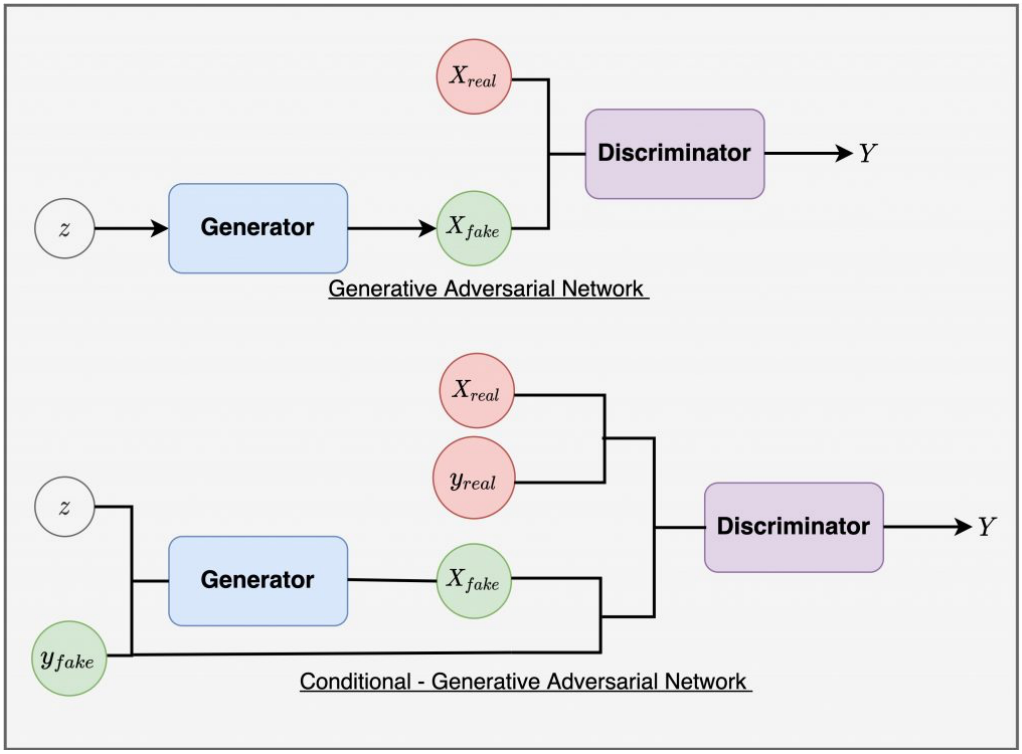
Implementation details:

這次我所與用的 GAN 模型架構為 WGAN-GP，WGAN 大致上的模型架構繼承了 DCGAN 的卷積網路用於生成及鑑別，但有別 DCGAN 是一個 well-trained 的模型，亦即是貢獻了一個很好訓練的模型架構及參數，WGAN 則是從源頭上解決了 origin GAN 難以訓練的問題。由原生 Origin 測量模型本身的分佈與訓練資料的分佈的 JS divergence，改為 EM distance，前者無法提供分佈之間的相應距離，也就是說沒有辦法判斷，模型提供的分佈跟訓練資料的差別。後者則可以給予，給予了模型能夠測量的依據。WGAN-GP 則是在 WGAN 的基礎上進行了改良，加上了 gradient penalty 來取代 WGAN 的 weight clipping，並在生成的圖片中間上了 gaussian noise。

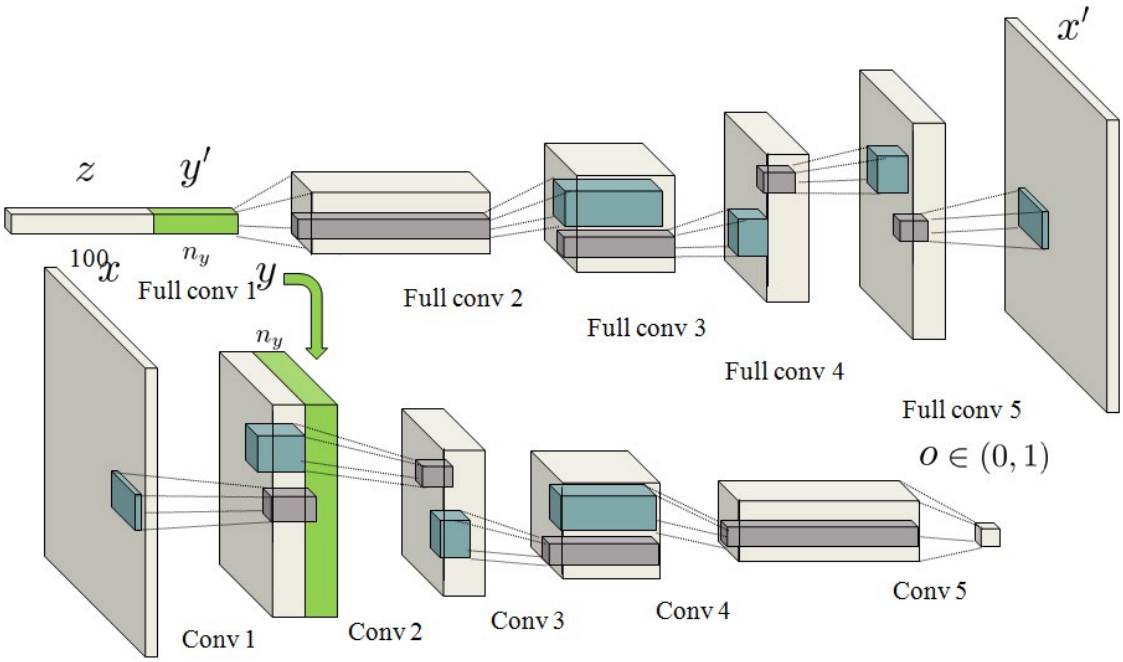
而如果要在上述所提到的 GAN 中加上 condition 的功能常見得實作方式有以下兩種：

一，可以將所有可能出現的 condition 送入 embedding 中，也就是說先計算所有可能出現的種類，例如有 24 個分類所有的可能就是 2^24，但是所提供的訓練集中有提到，一次最多只會出現不超過四個的 condition，整體的數量會更少，將每一個可能出現的 condition 做編號，送至 image * image 的 embedding 中，在與原本 image 的圖片做相連，也就是說把原本 3*64*64 的 image 加上 1*64*64 的 embedding condition 變成 4*64*64 再送入 model 中，便達成 image + label 的效果。

二，可以將原本 condition labels 的 one hot vector 直接送入一個 24 to image*image 的 linear fc 中再將這一維向量轉換成 image * image 的二維張量，一樣如方法一加入成圖片的第四個維度。整體架構如圖五、圖六。



圖五，GAN 與 cGAN



圖六，conditional DCGAN

而基本上我的 Generator 與 discriminator 基本上跟 DCGAN 的差距不大，如圖六、圖七。

```python
class Generator(nn.Module):
    def __init__(self,
                 num_classes=24,
                 latent_dim=100,
                 num_G_feature=64,
                 num_channels=3,
                 img_size = 64,
                 ):
        super(Generator, self).__init__()
        self.img_size = img_size
        self.num_classes = num_classes
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            # input: N * latent_dim * 1 * 1
        self._block(latent_dim + num_classes, num_G_feature*8, 4, 1, 0),
        self._block(num_G_feature*8, num_G_feature*4, 4, 2, 1),
        self._block(num_G_feature*4, num_G_feature*2, 4, 2, 1),
        self._block(num_G_feature*2, num_G_feature, 4, 2, 1),
        nn.ConvTranspose2d(num_G_feature, num_channels, 4, 2, 1, bias=False),
        nn.Tanh()
        )
        # output: N * num_channels * 64 * 64
    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
      return nn.Sequential(
          nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(True),

      )
    def forward(self, z, labels):
      #latent vector z: N * noise_dim * 1 * 1
      labels = labels.view(-1, self.num_classes, 1, 1)
      #print(z.size(),labels.size())
      x = torch.cat([z, labels], dim=1) # N * C * H * W
      #print(x.size())
      return self.main(x)
```

圖六，Generator 架構。

```python
class Discriminator(nn.Module):
    def __init__(self,
                 num_channels=3,
                 num_D_feature=64,
                 num_classes=24,
                 img_size=64
                 ):
        super(Discriminator, self).__init__()
        self.img_size = img_size

        self.labels_input = nn.Linear(24, self.img_size*self.img_size)
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            # in_channels, out_channels, kernel_size:, stride, padding
            nn.Conv2d(num_channels+1, num_D_feature, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2),
            # state size. (ndf) x 32 x 32
            self._block(num_D_feature, num_D_feature * 2, 4, 2, 1),
            # state size. (ndf*2) x 16 x 16
            self._block(num_D_feature * 2, num_D_feature * 4, 4, 2, 1),
            # state size. (ndf*4) x 8 x 8
            self._block(num_D_feature * 4, num_D_feature * 8, 4, 2, 1),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(num_D_feature * 8, 1, 4, 2, 0, bias=False),
            #nn.Sigmoid()
        )
    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
      return nn.Sequential(
          nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, bias=False),
          nn.InstanceNorm2d(out_channels, affine=True),
          nn.LeakyReLU(0.2),
      )
    def forward(self, x, labels):
        labels = self.labels_input(labels).view(-1,1,self.img_size,self.img_size)
        #print(x.size(),labels.size())
        x = torch.cat([x, labels],dim=1) # N * C * H * W

        #print(x.size())
        return self.main(x)
```

圖七，discriminator 架構採用 InstanceNorm 並移除 DCGAN 中的 sigmoid。

在訓練之前需初始化 model 的 weight 此參數由原始 paper 提供。如圖八。

```python
def initialize_weights(model):
    # Initializes weights according to the DCGAN paper
    for m in model.modules():
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.BatchNorm2d)):
            nn.init.normal_(m.weight.data, 0.0, 0.02)
```

圖八，初始化 model 的參數

```
disc_interation = 5
batch_size = 128
img_size = 64
latent_size = 100
embed_size = 24
learning_rate = 2e-4
Lambda_GP = 10
optim_gen = optim.Adam(gen.parameters(), lr=learning_rate, betas=(0.0,0.9) )
optim_disc = optim.Adam(disc.parameters(), lr=learning_rate, betas=(0.0,0.9) )
```

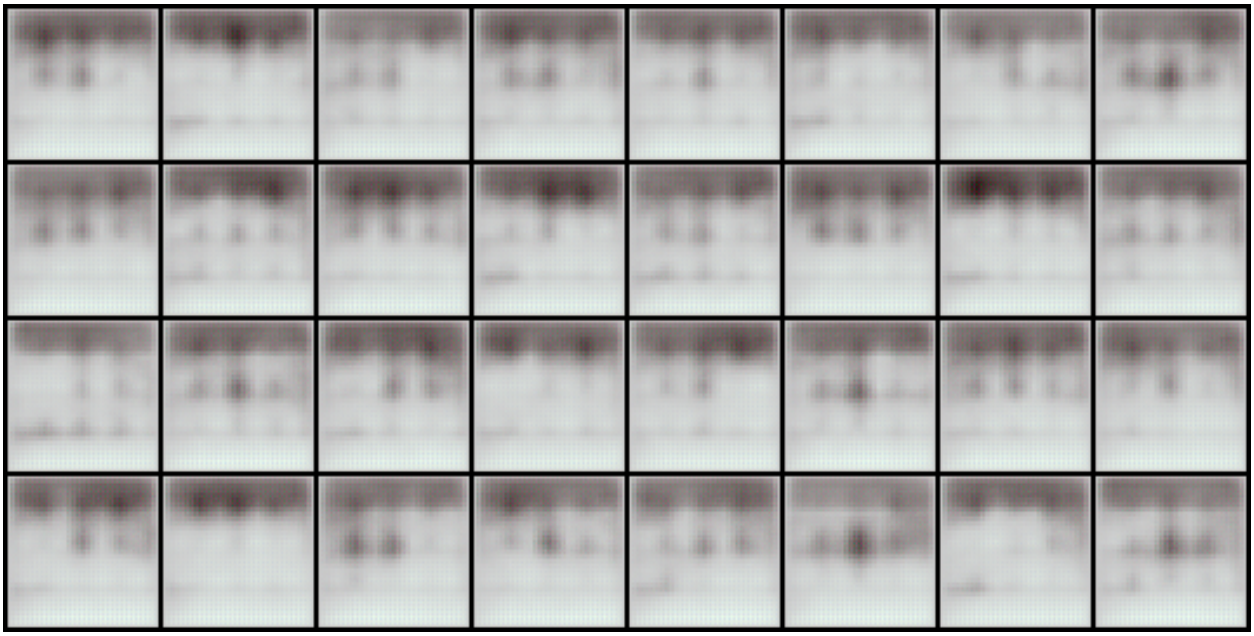
圖九，model 所使用之 hyperparameters，為論文提供。

```
def gradient_penalty(critic, labels, real, fake, device="cpu"):
    BATCH_SIZE, C, H, W = real.shape
    alpha = torch.rand((BATCH_SIZE, 1, 1, 1)).repeat(1, C, H, W).to(device)
    interpolated_images = real * alpha + fake * (1 - alpha)

    # Calculate critic scores
    mixed_scores = critic(interpolated_images,labels)

    # Take the gradient of the scores with respect to the images
    gradient = torch.autograd.grad(
        inputs=interpolated_images,
        outputs=mixed_scores,
        grad_outputs=torch.ones_like(mixed_scores),
        create_graph=True,
        retain_graph=True,
    )[0]
    gradient = gradient.view(gradient.shape[0], -1)
    gradient_norm = gradient.norm(2, dim=1)
    gradient_penalty = torch.mean((gradient_norm - 1) ** 2)
    return gradient_penalty
```

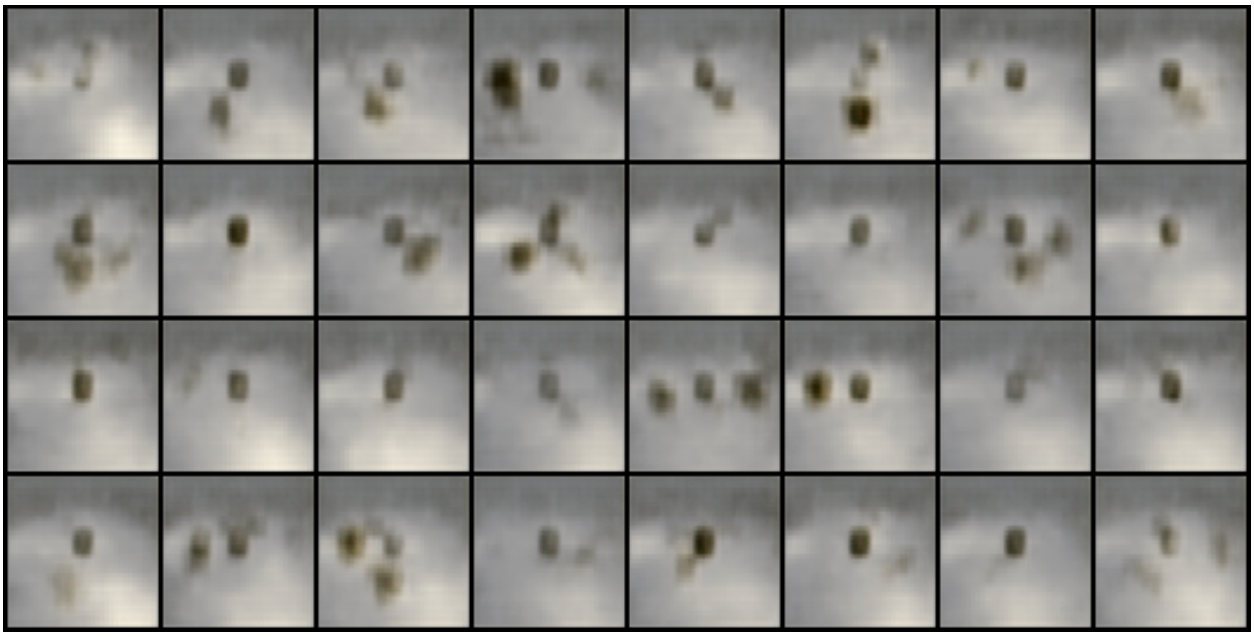
圖十，WGAN-GP 中的 gradient penalty。

```
for _ in range(disc_interation):
  noise = torch.randn(cur_batch_size, latent_size, 1, 1).to(device)

  fake_img = gen(noise, labels)
  disc_real = disc(real_img, labels).reshape(-1)
  disc_fake = disc(fake_img, labels).reshape(-1)

  gp = gradient_penalty(disc, labels,real_img, fake_img, device)
  loss_disc = ( -( torch.mean(disc_real) - torch.mean(disc_fake))
            + Lambda_GP * gp )

  disc.zero_grad()
  loss_disc.backward(retain_graph=True)
  optim_disc.step()

gen_fake = disc(fake_img, labels).view(-1)
loss_gen = -torch.mean(gen_fake)
gen.zero_grad()
loss_gen.backward()
optim_gen.step()
```

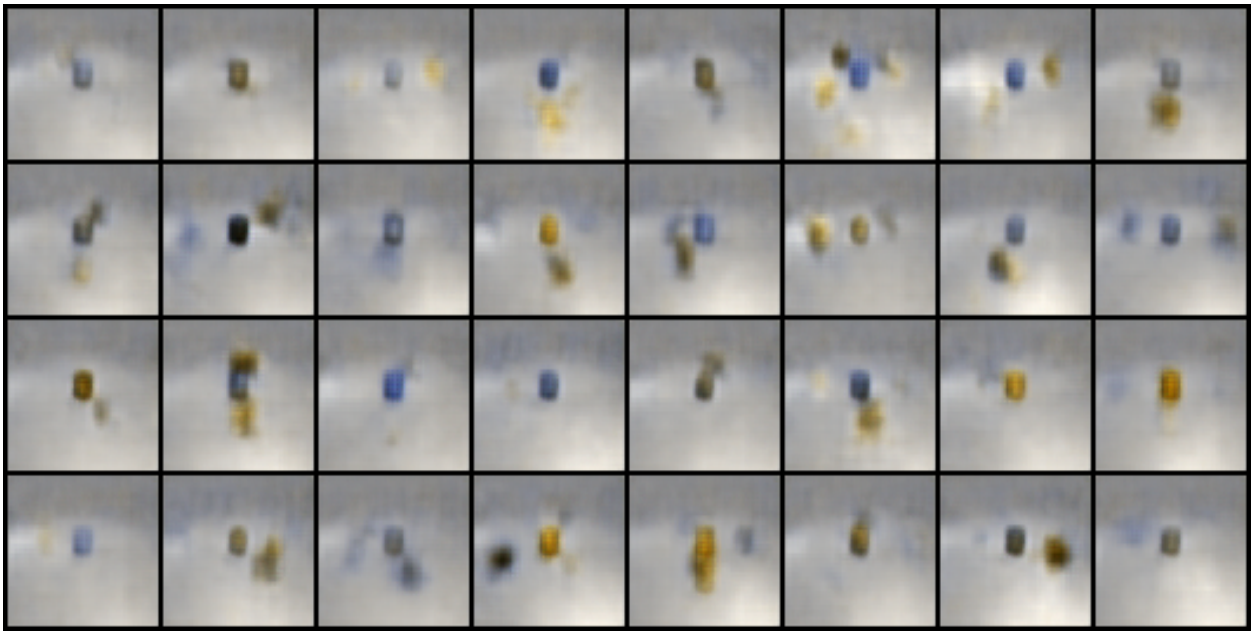
圖十一，因為要在同一個 epoch 中把 discriminator 多訓練幾次，因此將 loss=G_loss+D_loss 拆開，並在 D_loss 中加入 GP 作為訓練的一部分。



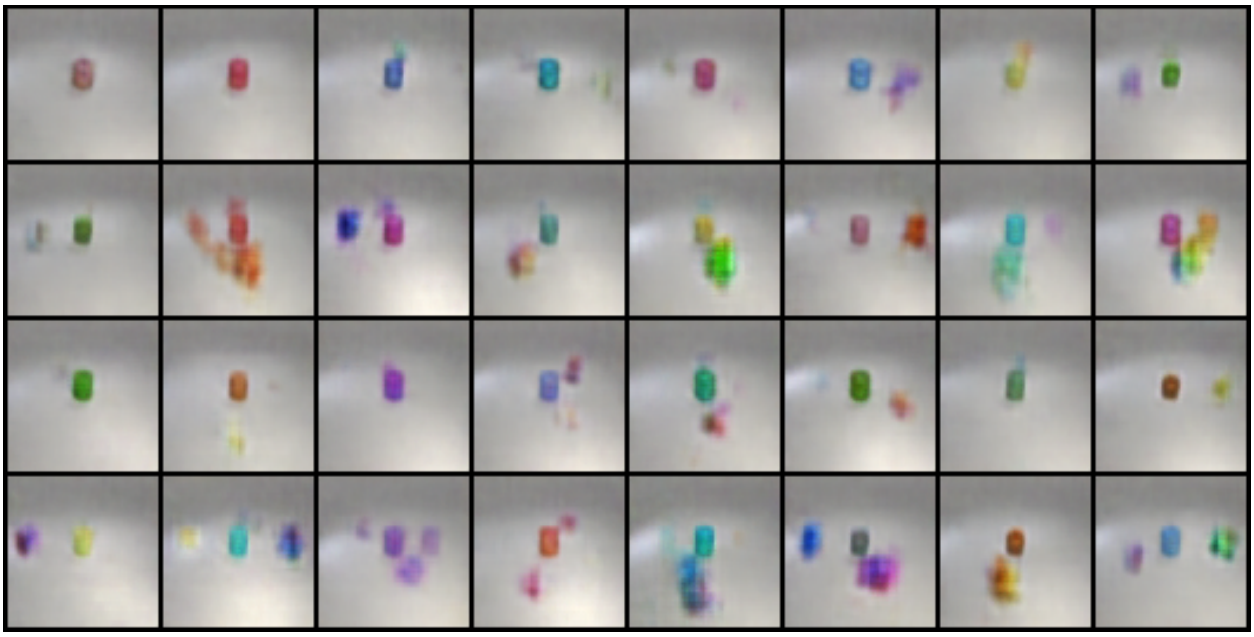
圖十二，epoch 0。



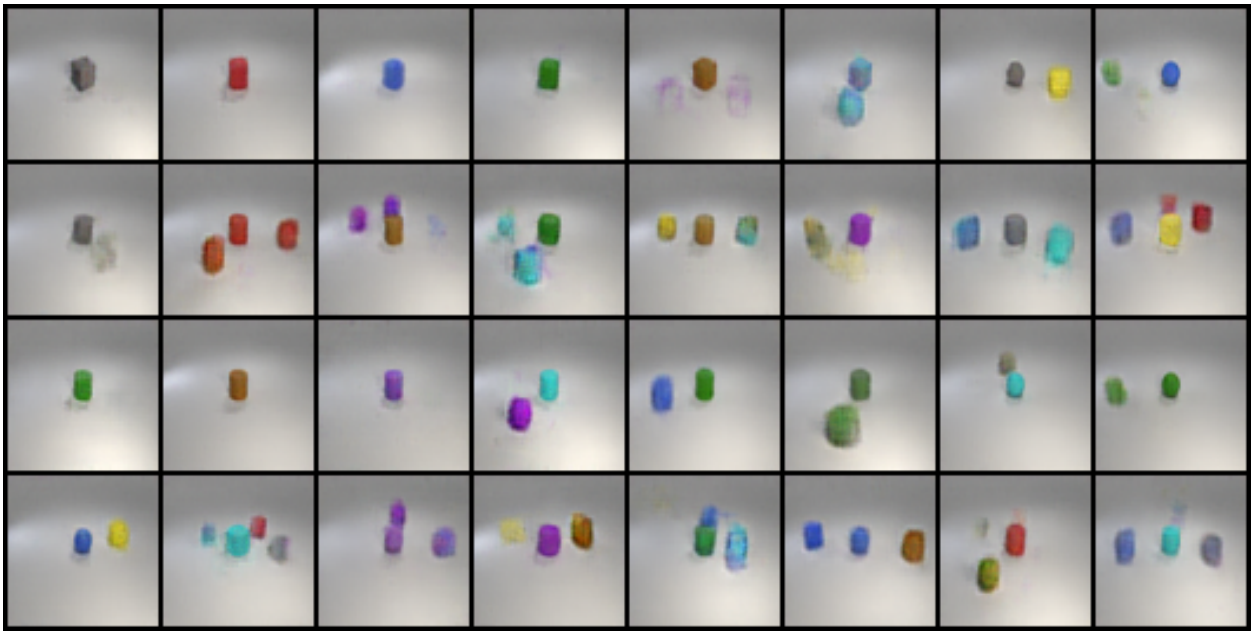
圖十三，epoch 4。



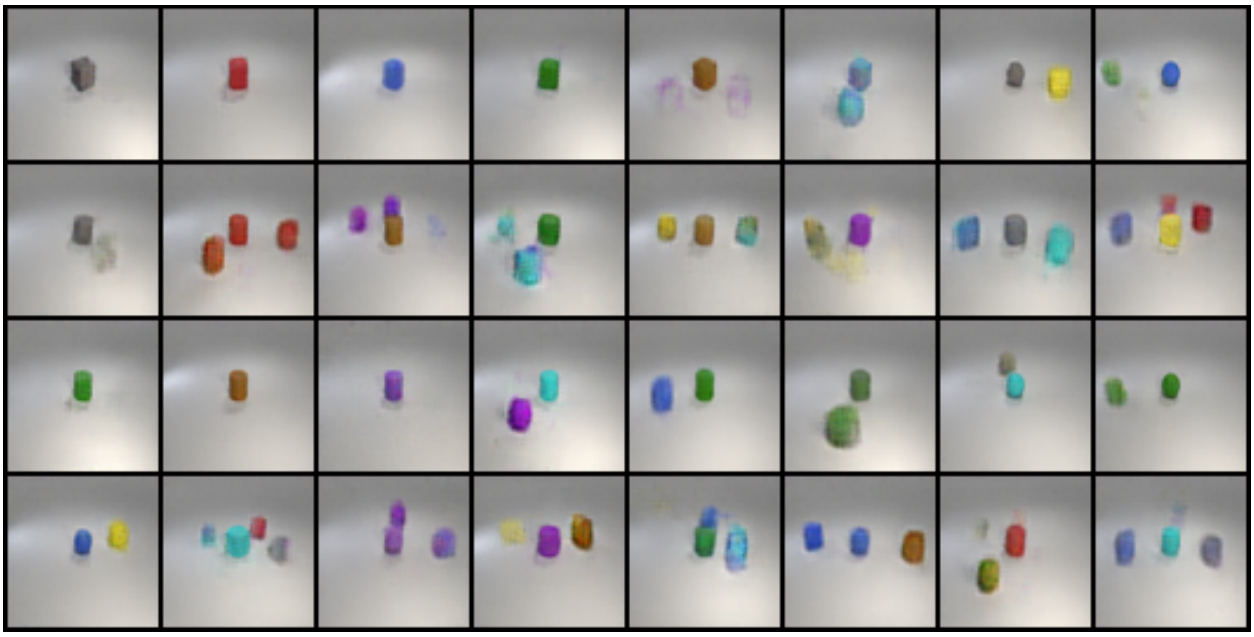
圖十三，epoch 8。



圖十四，epoch 24。



圖十五，epoch 40



圖十二，epoch 86、test_acc=0.625

Discussion:

- 通常論文所提供的 hyperparameter 通常都是最好訓練的，修改參數絕大多數都不會比較好。
- 將實驗中，將 discriminator 的 InstanceNorm 改成常見的 BatchNorm，前者在前中期的訓練效果中會緩慢的提高 acc，且 G_loss、D_loss 穩定，但後者往往在中後期會發生梯度爆炸的情形，G_loss、D_loss 會飆高。
- Generator 與 Discriminator 的架構最好是完全對稱的，不同的卷積結構會導致 latent space 中 gaussian noise 的解讀不盡相同，導致結果不優。
- 實驗中，如果 Generator 和 Discriminator 的訓練次數如果是 1:1 的話會導致模型在前期一直產生不出具有意義的圖片，生成出的圖片就如 noise 一樣，相較訓練次數 1:5 會需要迭代很多次，其原因可能為，如果 discriminator 連最基本的判斷能力都沒有的會，generator 的練訓會變得毫無方向。
- DCGAN 非常吃原始 paper 所提供的參數，任何一點參數的改動或是是 G 和 D 的架構變化，對 DCGAN 來說都是致命的，輕則模型表現不好、重則模型整個無法訓練，DCGAN 感覺更像是提供了一個經過爆搜所找到一組參數及架構，但也沒有一個依據作為訓練的標準。
- WGAN-GP 相較 WGAN、DCGAN 訓練上較穩定不大會出現 G_loss、D_loss 爆炸的情形，且訓練出的 acc 看來都有穩定且緩慢的上升，至少都會跟先前一樣好。
- WGAN-GP 因為有 EM distance 和 gradient penalty 的緣故，訓練上感覺會有一個穩定的方向可供參考。
- 在實驗過程中發現，通常單一 labels 的情況會先訓練好，再來依次為兩個 label，以此類推上去。換句話說，模型感覺會先學會一個物體的圖片，再來為兩個物品，及更多。
- Colab pro 真爛，老是不讓我連 GPU。