一、 Introduction

注意：因為 data 太大，壓縮之後也有 1.99gb, 我就把裡面的 image 刪掉了，所以如果要 train 的話可能會出錯，因為 image folder 裡面沒有 image

1. dataset.py: rewrite the file that TA attached
2. evaluator.py: the pretrained model for computing the acore
3. main.py:test test.json and show the score and save the image
4. best_eval_test.png:best image

5. acgan+dcgan.py(ipynb):acgan+dcgan, but I do a little change in the dcgan
6. cgan+dcgan.py(ipynb):cgan+dcgan
7. cgan+dcgan+wgan.py:try different loss function, but performance is bad
8. cgan+srgan.py:failed, I don't know where is the mistake..it generate some weird images
9. new_acgan+dcgan.py: change the loss function based on architecture:acgan+dcgan
   ----------folder----------
10. cgan_srgan: 一些失敗的結果，不知道是參數調得不好還是寫錯
11. cgan_dcgan_wgan:表現不好，應該是參數調不好
12. cgan_dcgan:表現也不好，最高只有到將近 60%
13. acgan_dcgan:表現還不錯，換了新的 loss function 後可以到 80%左右

懶人包：基本上如果只是要看最佳的表現的檔案的 code，看 new_acgan+dcgan.py(ipynb)就夠了，說穿了其他都是嘗試。
最佳的結果放在 acgan_dcgan folder 的'new'的部分。
Main.(py)ipynb:只是測試我最佳的 model 的表現，並將十次的表現算出來顯示，然後把圖片存起來而已。

二、 implementation details

1. Describe how you implement your model, including your choice of cGAN, model architectures, and loss functions.
   Here, I choose the acgan+dcgan as example，因為其他的表現好像都不太好，不知道是寫錯還是參數沒調好，此處我針對 acgan+dcgan 做介紹就好
   Model architecture:

Generator:

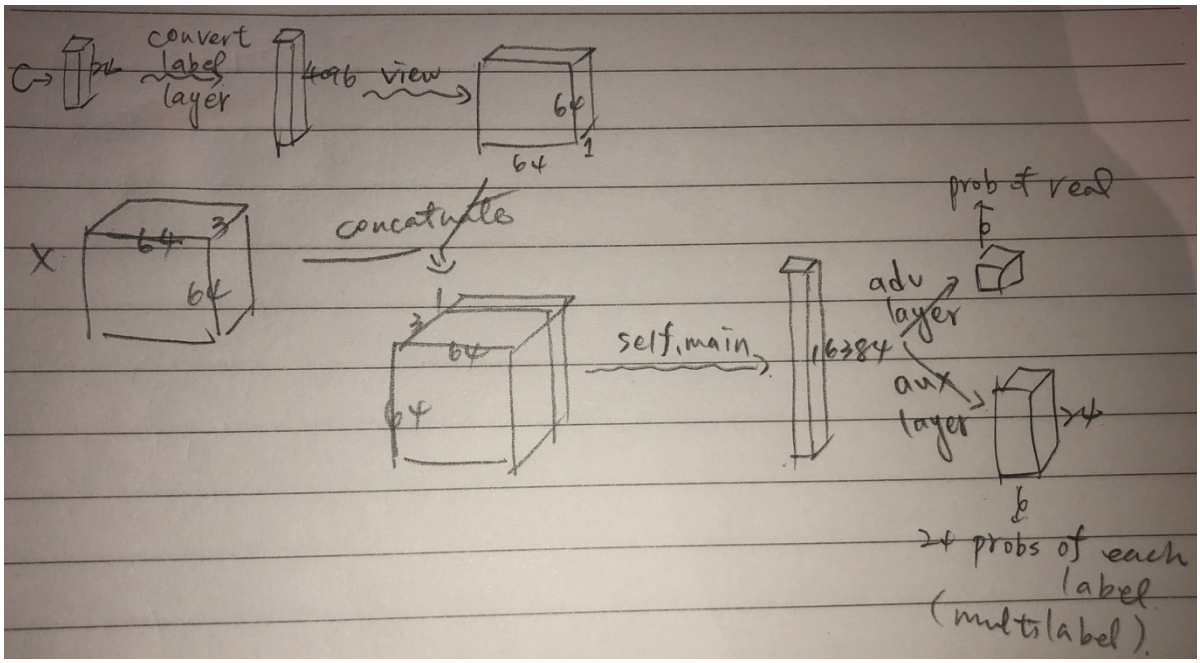
```
Generator(
  (conditionExpand): Sequential(
    (0): Linear(in_features=24, out_features=256, bias=True)
    (1): ReLU()
  )
  (l1): Sequential(
    (0): Linear(in_features=356, out_features=32768, bias=True)
  )
  (conv_blocks): Sequential(
    (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (1): Upsample(scale_factor=2.0, mode=nearest)
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Upsample(scale_factor=2.0, mode=nearest)
    (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): Tanh()
  )
)
```

首先我們用 conditionexpand 擴張 condition(label)的維度成 256
然後把他跟 input(bs, z_dim=100)串接在一起，於是就有了一個(batch size, 356)的 tensor, 將它放入 l1 layer 轉換成(batch size, 128, 16, 16)的 tensor
經過一連串的運算之後，得到一個(bs, 3, 64, 64)的向量
此處，我更改了 dcgan 的 generator,我把 transposeconv 換成了 upsampling(nearest mode)然後去做，表現好像有比 transposeconv 好了一點點

Discriminator:

```
Discriminator(
  (convert_label_layer): Sequential(
    (0): Linear(in_features=24, out_features=4096, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
  )
  (main): Sequential(
    (0): Conv2d(4, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(512, 1024, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Flatten()
  )
  (adv_layer): Sequential(
    (0): Linear(in_features=16384, out_features=1, bias=True)
    (1): Sigmoid()
  )
  (aux_layer): Sequential(
    (0): Linear(in_features=16384, out_features=24, bias=True)
    (1): Sigmoid()
  )
)
```

此處，給個示意圖比較方便



最後，這個 discriminator 會有兩個 return:
一個是基本款的 real 的 probability
另一個是輸出一個 24 維度的 tensor,每一個都代表了屬於那個 label 的機率(multilabel)
因為他是多標籤，也就是說我們不能用最基本款的 cross entropy 去算他的 loss
於是我把它改成針對每個 label,去求取 bce loss, i.e.有那個 label 或是沒有那個 label
然後再取平均，作為 auxiliary 的 loss
另外一個 loss 則是最基本款的 adversarial loss, 此處不贅述
總結:
以這兩個 loss 為基礎

```
def adversarial_criterion():
    return nn.BCELoss().to(device)

def auxiliary_criterion():
    return nn.BCELoss().to(device)
#why auxiliary loss be set as bceloss instead of ce loss?
#because a img contain many labels
#rather than only one label
```

Generator loss	(Adversarial_loss(假資料丟進 D 的機率, 滿滿都是 1 的 tensor))/2
Discriminator loss	假設 D 放入假資料出來的 return 一個是 adv_out_fake, 一個是 aux_out_fake 真的則是 adv_out_real, aux_out_real Loss_real = (adversarial_loss(adv_out_real, 滿滿都是 1 的 tensor)+auxiliary_loss(aux_out_real, label))/2

	Loss_fake = (adversarial_loss(adv_out_fake, 滿滿都是 0 的 tensor)+auxiliary_loss(aux_out_fake, label))/2 Total_loss = (loss_real+loss_fake)/2

2. Specify the hyperparameters

   以 acgan+dcgan 為例：

```
z_dim = 100
c_dim = 256
n_classes = 24
img_size = 64
batch_size = 64
upsample_block_num = 6#放大幾次
epochs = 200
lr_d = 0.0004
lr_g = 0.0001
```

   Upsample_block_num 意思是你的 image 的 size 要放大幾次

   比如說 initial size=(bs, 3, 4, 4)那放大一次就是變成(bs, 3, 8, 8)

三、 Results and discussion

1. Show your results based on the testing data

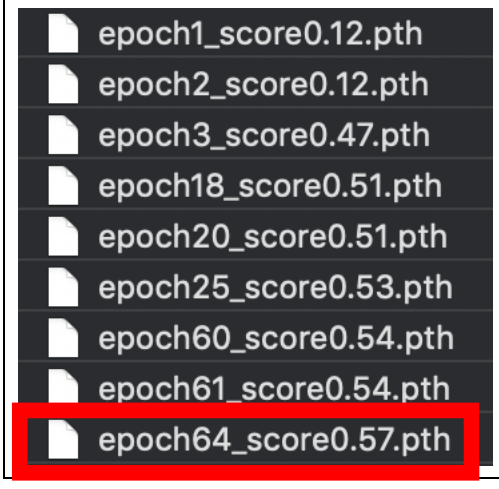
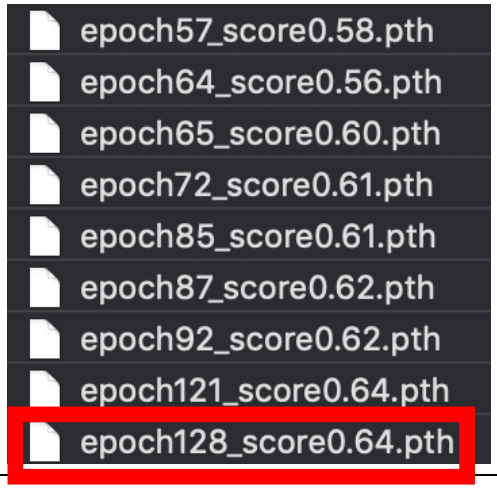
其實我覺得效果還不錯，雖然放大之後會變得很奇怪，但是生成的圖片是還不錯的，除了某些地方雜訊比較多之外。

這東西的 acc 還有來到平均 80 幾%：

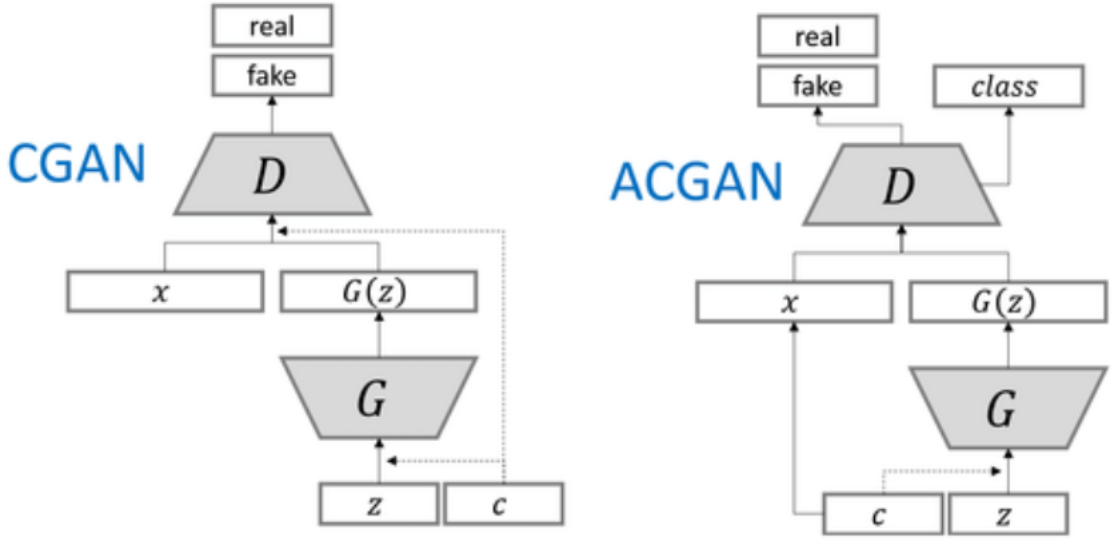
```
score: 0.85
score: 0.86
score: 0.85
score: 0.88
score: 0.83
score: 0.86
score: 0.83
score: 0.83
score: 0.82
score: 0.86

avg score: 0.85
```

這是連續做十次的結果，可以看到還是不錯的。

2. Discuss the results of different model architectures.
   此處，我先以 cgan 和 acgan 做比較：
   先說結論：acgan 表現明顯比 cgan 好些，在我 generator 跟 discriminator 還有 lossfunction 設計不變的情況下，大概差了快 10%左右：

Cgan+dcgan	Acgan+dcgan
	

   我們先看一下這兩個傢伙的架構：



應該是因為 acgan 的 loss 比原始的 cgan 多了一個的關係，也就是說，最後我的 discriminator 不只要 fit adversarial loss, 還要 fit auxiliary loss, 這件事情保證了 acgan 生出來的東西在 given label 的情況下對於圖片是在哪個 condition 狀況下生出來的這件事情有更多的顧慮和 optimize, 也因此，自然在 pretrained model 那個地方的表現就會比較好！

我還發現如果在 acgan+dcgan 的架構下，我把 perceptral loss 初始的 vgg 置換成 resnet,效果會更好！從原本的 0.64 來到了 0.85:

```python
def adversarial_criterion():
    return nn.BCELoss().to(device)

def auxiliary_criterion():
    return nn.BCELoss().to(device)

class GeneratorLoss(nn.Module):
    def __init__(self):
        #modify the path to your own path
        super(GeneratorLoss, self).__init__()
        checkpoint = torch.load('data/classifier_weight.pth')
        self.resnet = models.resnet18(pretrained=False)
        self.resnet.fc = nn.Sequential(
            nn.Linear(512, 24),
            nn.Sigmoid()
        )
        self.resnet.load_state_dict(checkpoint['model'])
        self.loss_network = nn.Sequential(*list(self.resnet.children())[:-1])
        for param in self.loss_network.parameters():
            param.requires_grad = False
        self.loss_network.eval()
        self.loss_network = self.loss_network.cuda()

        self.classnum = 24
        self.mse_loss = nn.MSELoss()
#         self.Llloss = nn.LlLoss()
        self.tv_loss = TVLoss()

    def forward(self, out_labels, out_images, target_images):
        # Adversarial Loss
        adversarial_loss = nn.BCELoss()(out_labels, smooth_label('real', out_labels.shape))
        # Perception Loss
        perception_loss = self.mse_loss(self.loss_network(out_images), self.loss_network(target_images))
        # Image Loss
        image_loss = self.mse_loss(out_images, target_images)
        # TV Loss
        return image_loss + 0.001 * adversarial_loss + 0.1 * perception_loss + 2e-8 * tv_loss
#         return adversarial_loss

# introduction of tv loss: https://www.daimajiaoliu.com/daima/479773d4d1003fe
class TVLoss(nn.Module):
    def __init__(self, tv_loss_weight=1):
        super(TVLoss, self).__init__()
        self.tv_loss_weight = tv_loss_weight

    def forward(self, x):
        batch_size = x.size()[0]
        h_x = x.size()[2]
        w_x = x.size()[3]
        count_h = self.tensor_size(x[:, :, 1:, :])
        count_w = self.tensor_size(x[:, :, :, 1:])
        h_tv = torch.pow((x[:, :, 1:, :] - x[:, :, :h_x - 1, :]), 2).sum()
        w_tv = torch.pow((x[:, :, :, 1:] - x[:, :, :, :w_x - 1]), 2).sum()
        return self.tv_loss_weight * 2 * (h_tv / count_h + w_tv / count_w) / batch_size

    @staticmethod
    def tensor_size(t):
        return t.size()[1] * t.size()[2] * t.size()[3]
```

Image loss:針對 pixel wise 的 loss

Adversarial loss: standard adv loss

Perception loss:此處特別的是，我將擷取高維度特徵計算 loss,但是我把在 srgan 的論文中的 vgg 架構換成了 resnet, 並將 pretrian model 作為擷取特徵的網路,賦予權重 0.1

Tv loss: total variation loss,用於 denoise

Auxiliary loss:就是在 discriminator 計算 label 的 loss

最後把這幾個 loss 組合，分別拿去給 generator 使用

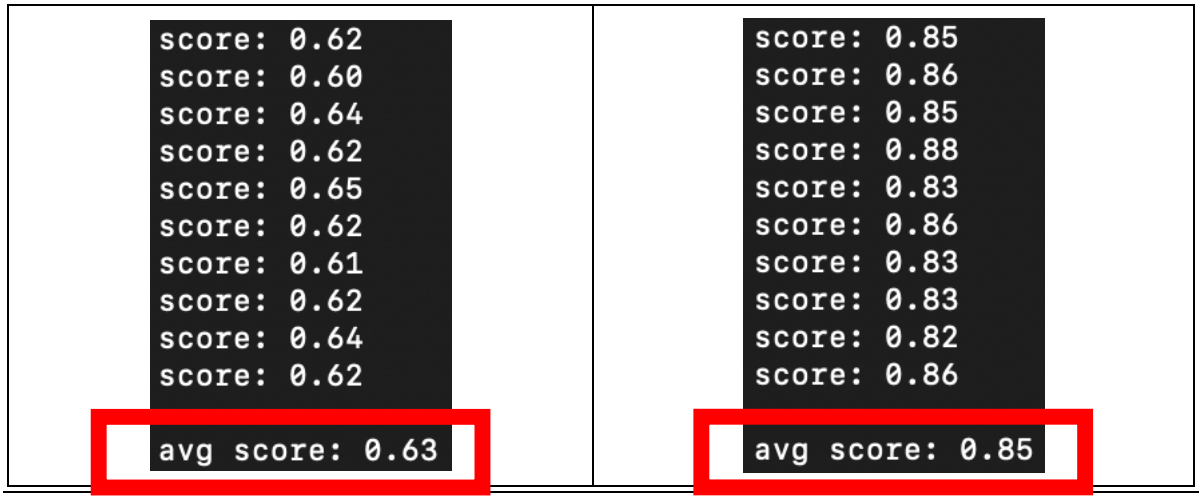
至於 discriminator 的 loss

就是單純的 adv loss 跟 auxiliary loss 做線性組合，以下是 code：

```
############################
#   Update D network: maximize D(x)-1-D(G(z))
############################
  for _ in range(4):

fake = G(random_z(label.size(0), z_dim), label)

optimizer_d.zero_grad()

real_pred, real_aux = D(real, label)

loss_real = (adversarial_loss(real_pred, smooth_label('real', real_pred.shape)
             + auxiliary_loss(real_aux, label)) / 2


fake_pred, fake_aux = D(fake, label)
loss_fake = (adversarial_loss(fake_pred, torch.zeros(fake_pred.shape, device=de ice))
             + auxiliary_loss(fake_aux, label)) / 2

loss_d = (loss_real + loss_fake) / 2
      if flag!=1:

loss_d.backward()
optimizer_d.step()

total_loss_d += loss_d.item()

############################
#   Update G network: minimize 1-D(G(z)) + Perception Loss + Image Loss + TV Loss
############################
#generate fake img
for _ in range(4):
    optimizer_g.zero_grad()

    fake_img = G(random_z(label.size(0), z_dim), label)

    validity, pred_label = D(fake_img, label)

    loss_au = 0.2 * auxiliary_loss(pred_label, label)

    loss_ge = 0.8 * generator_criterion(validity, fake_img, real)

    loss_g = loss_au + loss_ge

    loss_g.backward()
```

換了不同的 loss，更換不同的 pretrain model 並運用到 loss 竟然可以讓表現暴增了 20 幾%，我真的很意外。

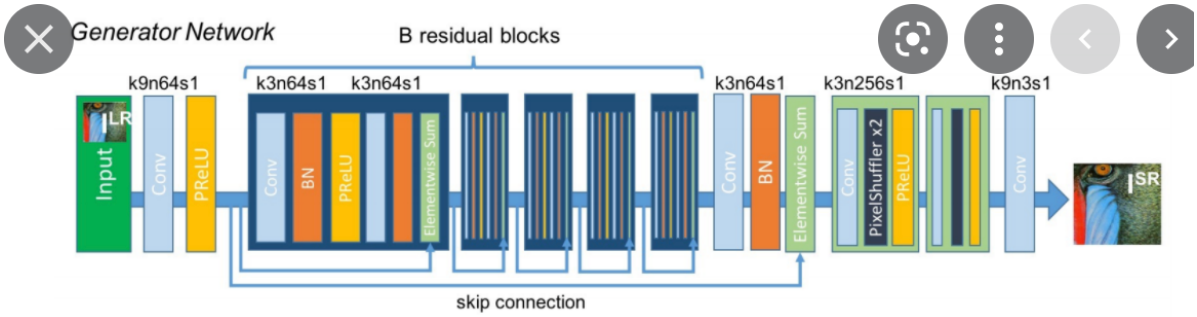
Acgan+dcgan.ipynb(py) Loss : adversarial loss + auxiliary loss	New_acgan+dcgan.ipynb(py) Loss : adversarial loss + auxiliary loss + loss from srgan



後來我也嘗試過利用不同的 generator 和 discriminator 甚至是不同的 loss function:
比如說，我使用 srgan 作為我的 generator 和 discriminator:
Generator 的架構非常龐大：

```
generator(
  (conditionExpand): Sequential(
    (0): Linear(in_features=24, out_features=256, bias=True)
    (1): ReLU()
  )
  (block1): Sequential(
    (0): Conv2d(356, 64, kernel_size=(9, 9), stride=(1, 1), padding=(4, 4))
    (1): PReLU(num_parameters=1)
  )
  (block2): ResidualBlock(
    (block1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (block2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (act): PReLU(num_parameters=1)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (block3): ResidualBlock(
    (block1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (block2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (act): PReLU(num_parameters=1)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (block4): ResidualBlock(
    (block1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (block2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (act): PReLU(num_parameters=1)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (block5): ResidualBlock(
    (block1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (block2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (act): PReLU(num_parameters=1)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (block6): ResidualBlock(
    (block1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (block2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (act): PReLU(num_parameters=1)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (block7): Sequential(
    (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (block8): Sequential(
    (0): UpsampleBlock(
      (conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (pixel_shuffle): PixelShuffle(upscale_factor=2)
      (act): PReLU(num_parameters=1)
    )
    (1): UpsampleBlock(
      (conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (pixel_shuffle): PixelShuffle(upscale_factor=2)
      (act): PReLU(num_parameters=1)
    )
    (2): UpsampleBlock(
      (conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (pixel_shuffle): PixelShuffle(upscale_factor=2)
      (act): PReLU(num_parameters=1)
    )
    (3): UpsampleBlock(
      (conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (pixel_shuffle): PixelShuffle(upscale_factor=2)
      (act): PReLU(num_parameters=1)
    )
    (4): UpsampleBlock(
      (conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (pixel_shuffle): PixelShuffle(upscale_factor=2)
      (act): PReLU(num_parameters=1)
    )
    (5): UpsampleBlock(
      (conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (pixel_shuffle): PixelShuffle(upscale_factor=2)
      (act): PReLU(num_parameters=1)
    )
    (6): Conv2d(64, 3, kernel_size=(9, 9), stride=(1, 1), padding=(4, 4))
  )
)
```

簡化一下：

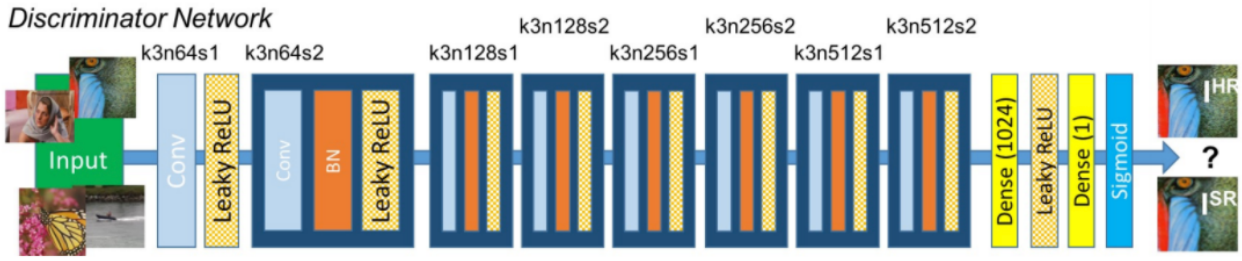


這架構其實就是一堆 residual block 相加，implement 並不難

然後這是 discriminator:

```
discriminator(
  (convert_label_layer): Sequential(
    (0): Linear(in_features=24, out_features=4096, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
  )
  (net): Sequential(
    (0): Conv2d(4, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2)
    (8): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (9): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2)
    (11): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): LeakyReLU(negative_slope=0.2)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): LeakyReLU(negative_slope=0.2)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (19): LeakyReLU(negative_slope=0.2)
    (20): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (21): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (22): LeakyReLU(negative_slope=0.2)
    (23): AdaptiveAvgPool2d(output_size=1)
    (24): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
    (25): LeakyReLU(negative_slope=0.2)
    (26): Conv2d(1024, 1, kernel_size=(1, 1), stride=(1, 1))
  )
)
```

一樣簡化：



後來，我把他的 loss 設置成 srgan 裡面建議的:

```
class GeneratorLoss(nn.Module):
    def __init__(self):
        super(GeneratorLoss, self).__init__()
        vgg = models.vgg16(pretrained=True)
        loss_network = nn.Sequential(*list(vgg.features)[:31]).eval()
        for param in loss_network.parameters():
            param.requires_grad = False
        self.loss_network = loss_network
        self.mse_loss = nn.MSELoss()
#         self.L1loss = nn.L1Loss()
        self.tv_loss = TVLoss()

    def forward(self, out_labels, out_images, target_images):
        # Adversarial Loss
        adversarial_loss = nn.BCELoss()(out_labels, smooth_label('real', out_labels.shape))
        # Perception Loss
        perception_loss = self.mse_loss(self.loss_network(out_images), self.loss_network(target_images))
        # Image Loss
        image_loss = self.mse_loss(out_images, target_images)
        # TV Loss
        tv_loss = self.tv_loss(out_images)
        return image_loss + 0.001 * adversarial_loss + 0.006 * perception_loss + 2e-8 * tv_loss
#         return adversarial_loss

# introduction of tv loss: https://www.daimajiaoliu.com/daima/479773d4d1003fe
class TVLoss(nn.Module):
    def __init__(self, tv_loss_weight=1):
        super(TVLoss, self).__init__()
        self.tv_loss_weight = tv_loss_weight

    def forward(self, x):
        batch_size = x.size()[0]
        h_x = x.size()[2]
        w_x = x.size()[3]
        count_h = self.tensor_size(x[:, :, 1:, :])
        count_w = self.tensor_size(x[:, :, :, 1:])
        h_tv = torch.pow((x[:, :, 1:, :] - x[:, :, :h_x - 1, :]), 2).sum()
        w_tv = torch.pow((x[:, :, :, 1:] - x[:, :, :, :w_x - 1]), 2).sum()
        return self.tv_loss_weight * 2 * (h_tv / count_h + w_tv / count_w) / batch_size

    @staticmethod
    def tensor_size(t):
        return t.size()[1] * t.size()[2] * t.size()[3]
```

本來以為表現會更好，結果效果超差：以下是做了 145 次的結果

完全不知道到底在生成什麼東西，我就覺得我應該是哪裡打錯了，但是又一直找不出錯誤，我在想有幾個原因：

1. Data 太少：只有 18005 筆 data 或許不夠生成 multilabel 的 data?
2. Hyperparameter 選取不良：我有嘗試過以下這幾組 lr (0.0004, 0.0001), (0.0001, 0.0004), (0.00004, 0.00001), (0.0002, 0.0002)但是生成的結果一樣糟糕

3. code 不知道哪裡打錯：這應該是最有可能的。

還有一個需要紀錄的有趣事情：

對於不同的 epoch 來說，會有相對不同的 gloss and dloss:比如以下例子：

```
100%|          | 282/282 [03:26<00:00,  1.37it/s]
Epoch[137/200]
score: 0.3472222222222222
generator_loss:0.04199897
discriminator_loss:0.00164753

100%|          | 282/282 [03:26<00:00,  1.37it/s]
Epoch[138/200]
score: 0.3888888888888889
generator_loss:0.04254139
discriminator_loss:0.00174412
```

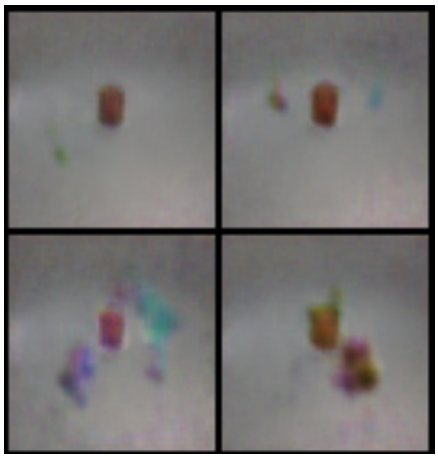
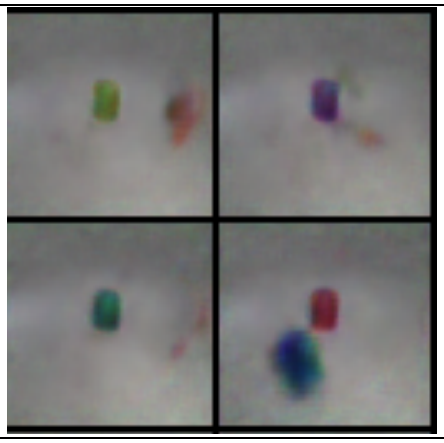
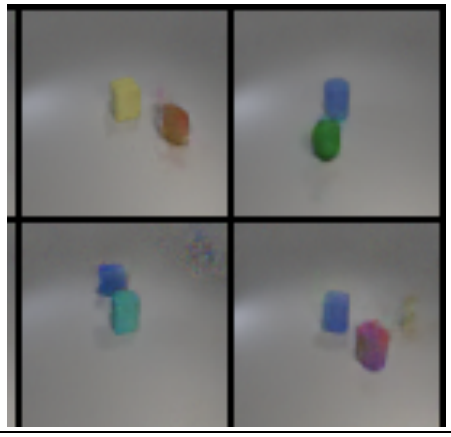
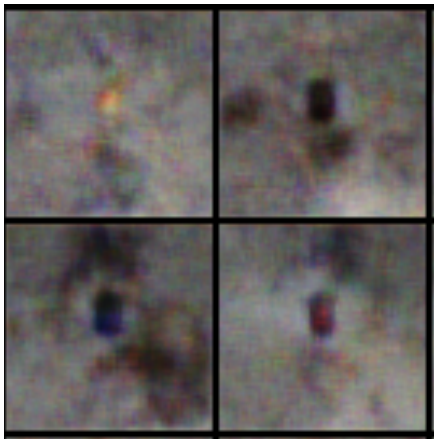
你可以看到

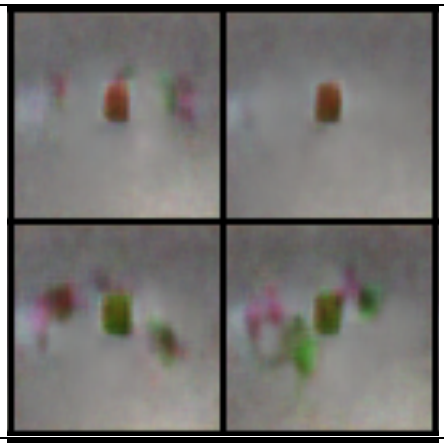
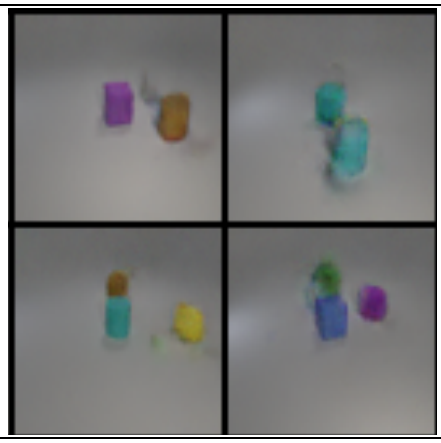
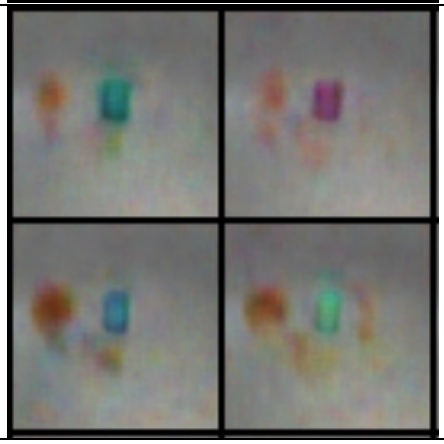
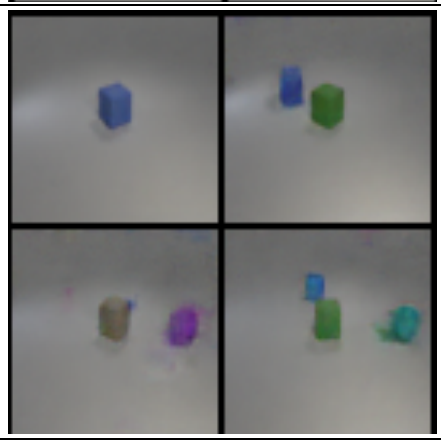
1. 不能以 gan 的 loss 決定輸出圖片品質的好壞：因為像上圖，下面這個 epoch score 比較好但是 loss 卻比較高
2. 超參數不好 finetune:根據上點，在以往的深度訓練當中，我們習慣以 loss 或是 acc 來當作判斷一個 model 訓練的好不好的準則，但是在 gan 當中這件事情是行不通的(因為一個

generator 的 loss 是看當前的 discriminator 所影響的，有點像是比較出來的概念)！因此 hyperparameters 的調整只能看經驗，還有 paper 提出的一些 trick.

3. 所以基本上我調整 lr 都是憑感覺，然後再搜尋一些前人的經驗，但是在這邊，我突然有一種選對 loss 架構會比選 lr 還關鍵的感覺
4. Gan 真的好難訓練

丟上一些一開始的訓練照片：

model	loss	Best acc	Epoch1 image	best epoch image
Cgan dcgan	Adv loss	0.57 (ep64)		
Cgan dcgan	Use loss in srgan	0.57 (ep135)		
Cgan dcgan wgan	loss of wgan	0.42 (ep143)		

acgan dcgan	Adv loss	0.64 (ep128)		
new acgan dcgan	Loss in srgan but replace vgg to resnet	0.85 (ep55)		

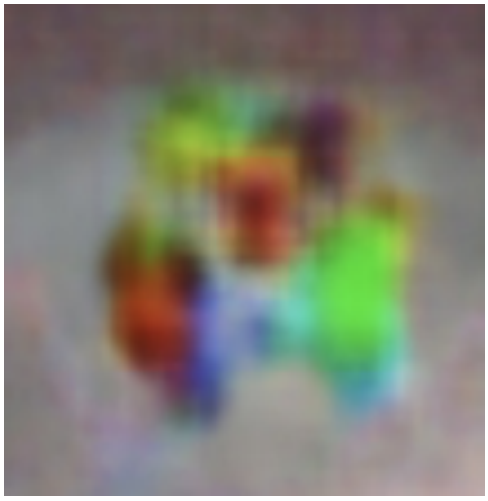
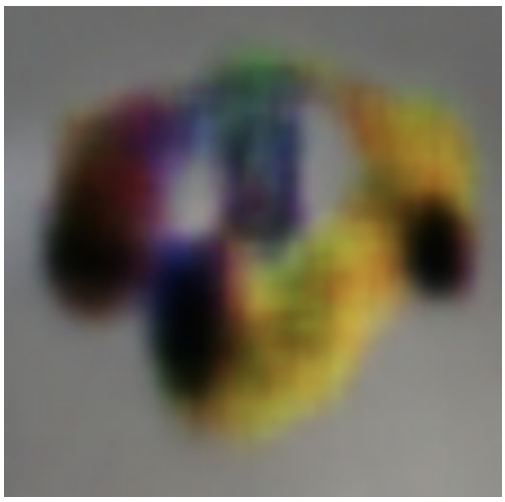
由上圖你可以看到不同組合對應到的 acc 跟生成的圖：

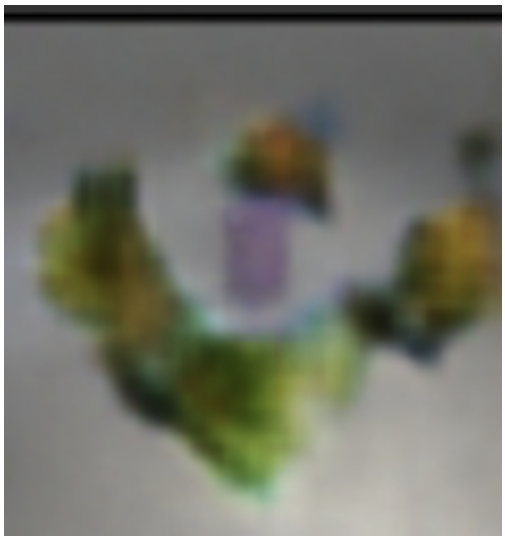
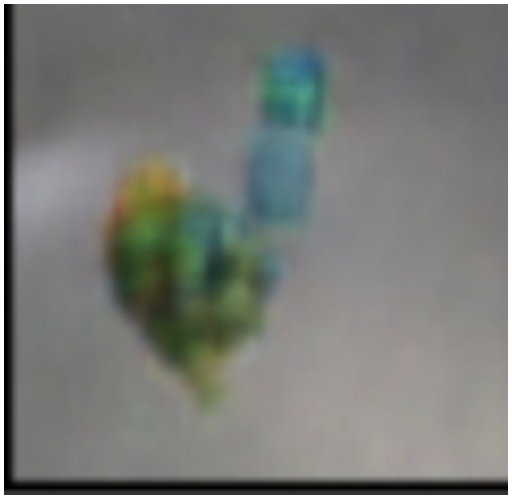
1. Wgan 的表現是最差的，這其實出乎我的意料之外，我一直以為他會是最好的，想來或許是我參數調整的不好的原因，有趣的是，我覺得在做 gan 時，看他一開始的生成圖片真的很有趣，有一些還會出現一些奇怪的形狀，像是 wgan 的第一個生圖片看起來像是被惡靈附身一樣，整個烏煙瘴氣的，特別有趣，其他的第一個生成圖片其實就都大同小異了，基本上就是模糊不清到後來漸漸清晰。
2. 不同的 loss 架構會影響生成器的好壞。此點可以看 acgan+dcgan 那邊，我覺得也是一個很 trivial 的事實
3. 因為 data 的本質我看起來他的分配是比較明顯的，這個很抽象，因為 data 的非白色點基本上都聚集在中間，要去模擬他的 dist 其實相對來說會比其他的 task 容易，比如說複雜的人臉或是動物等等。所以應該會比其他 task 好 train.
4. Performance of cgan loss(standard loss):0.57
   Performance of acgan loss:0.64
   Performance of acgan loss + srgan loss:0.85

   0.85.>>0.64>0.57

   總結：選對 loss function 或是更改 loss function 很重要！

最後附上幾個神奇的結果，我覺得很有趣：

	看起來像是什麼奇怪的細胞集合體 或是一團惡夢 找不到圖..
	No comment

	很像三頭蛇或是三朵香菇之類的 
	這個最有趣，看起來像是詛咒別人..