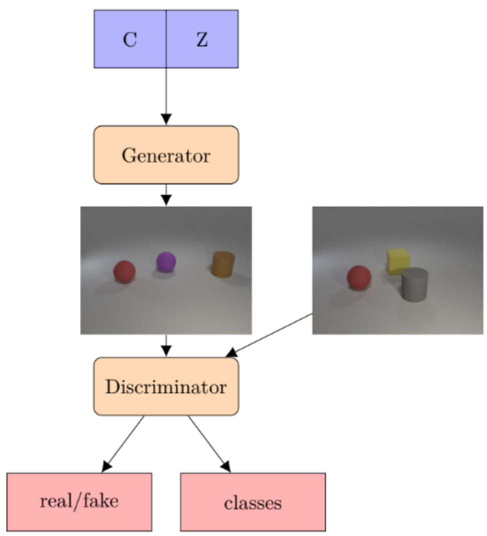
Lab5 - Let's Play GANs

- Student Info
  1. Student ID: 310555024
  2. Student Name: 林廷翰

- Report (50%)
  1. Introduction (5%)



此次lab的目的是訓練一個condition GAN。Generator的input是一個C及Z，C代表的是condition vector，而Z代表的是一個latent random variable，output則為一張image。而Discriminator則在Generator產生的image及real dataset之間做真實性判斷。

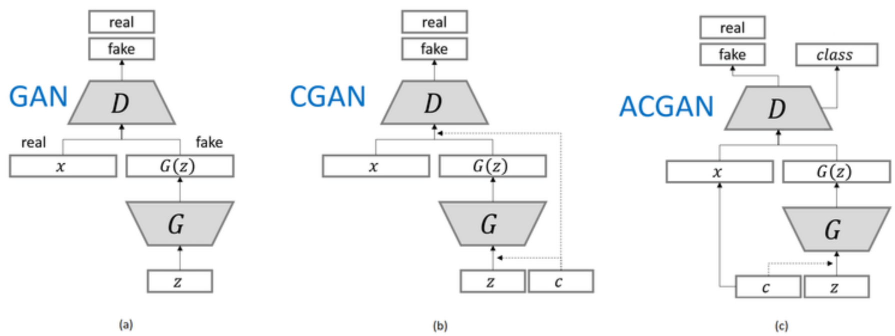
從dataset的角度來看，dataset為ICLEVR，資料類型為幾何圖片，一共有24種不同的幾何物體，因此我們的condition會是一個dimension為24 的one hot vector，

且一次最多不會超過三種物體，也就是說condition的24-dim中只會有1~3 個 1 出現。

2. Implementation details (15%)

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

  1. Architecture



此次training選擇的架構為ACGAN，如上圖C，比較大的差異在於在一般的conditional GAN，discriminator的output為一個0~1的數字，用於判斷image的真實性，而ACGAN的output除了真實性分數外，還多了一個class，用於計算classification loss。

  2. Generator

先將condition vector做embedding，再將結果與noise結合，作為Generator的輸入，implementation部分，則參考DCGAN，其中包含五層的convolution layer。

```
# condition embedding
self.label_emb = nn.Sequential(
    nn.Linear(self.n_classes, self.nc),
    nn.LeakyReLU(0.2, True)
)
```

```python
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(self.nz + self.nc, self.ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(self.ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(self.ngf * 8, self.ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(self.ngf * 4, self.ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(self.ngf * 2, self.ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(self.ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(self.ngf, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (rgb channel = 3) x 64 x 64
        )
```

```python
    def forward(self, noise, labels):
        label_emb = self.label_emb(labels).view(-1, self.nc, 1, 1)
        gen_input = torch.cat((label_emb, noise), 1)
        out = self.main(gen_input)
        return out
```

3. Discriminator

   Discriminator部分和DCGAN也有些相似，不過差異在於最後會有兩個output，分別為真實性分數及multi-label classifier。

```python
        self.main = nn.Sequential(
            # input is (rgb channel = 3) x 64 x 64
            nn.Conv2d(3, self.ndf, 3, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.5, inplace=False),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(self.ndf, self.ndf * 2, 3, 1, 0, bias=False),
            nn.BatchNorm2d(self.ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.5, inplace=False),
            # state size. (ndf*2) x 30 x 30
            nn.Conv2d(self.ndf * 2, self.ndf * 4, 3, 2, 1, bias=False),
            nn.BatchNorm2d(self.ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.5, inplace=False),
            # state size. (ndf*4) x 16 x 16
            nn.Conv2d(self.ndf * 4, self.ndf * 8, 3, 1, 0, bias=False),
            nn.BatchNorm2d(self.ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.5, inplace=False),
            # state size. (ndf*8) x 14 x 14
            nn.Conv2d(self.ndf * 8, self.ndf * 16, 3, 2, 1, bias=False),
            nn.BatchNorm2d(self.ndf * 16),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.5, inplace=False),
            # state size (ndf*16) x 8 x 8
            nn.Conv2d(self.ndf * 16, self.ndf * 32, 3, 1, 0, bias=False),
            nn.BatchNorm2d(self.ndf * 32),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.5, inplace=False),
```

```python
        # discriminator fc
        self.fc_dis = nn.Sequential(
            nn.Linear(5 * 5 * self.ndf * 32, 1),
            nn.Sigmoid()
        )
```

```
# aux-classifier fc
self.fc_aux = nn.Sequential(
    nn.Linear(5 * 5 * self.ndf * 32, self.n_classes),
    nn.Sigmoid()
)
```

4. Loss Function

   由於使用ACGAN作為GAN架構，因此會有兩種loss，分別為`dis_errD`及`aux_errD`分別代表真實性分數及multi-label classifier的loss。並使用`aux_weight`來放大multi-label classifier的loss，以增加accuracy。

```
dis_output, aux_output = discriminator(real_image)
dis_errD_real = dis_criterion(dis_output, real_label)
aux_errD_real = aux_criterion(aux_output, aux_label)
errD_real = dis_errD_real + args.aux_weight * aux_errD_real
errD_real.backward()
D_x = dis_output.mean().item()
# compute the current classification accuracy
accuracy = compute_acc(aux_output, aux_label)
```

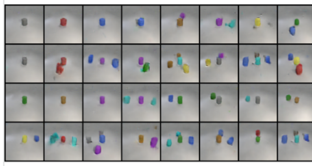
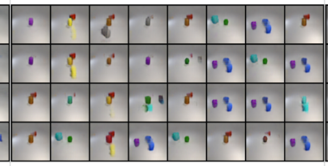
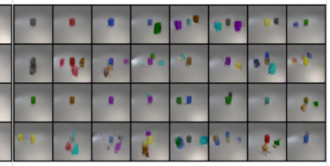
```
dis_output, aux_output = discriminator(fake_image.detach())
dis_errD_fake = dis_criterion(dis_output, fake_label)
aux_errD_fake = aux_criterion(aux_output, aux_label)
errD_fake = dis_errD_fake + args.aux_weight * aux_errD_fake
errD_fake.backward()
D_G_z1 = dis_output.mean().item()
```

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

  1. DCGAN
     - Epoch → 300
     - Learning rate → 0.0002
     - Loss Function → BCELOSS
  2. ACGAN
     - Epoch → 400
     - Learning rate → 0.0002
     - Aux weight → 128
     - Loss Function → BCELOSS

3. Results and discussion (30%)

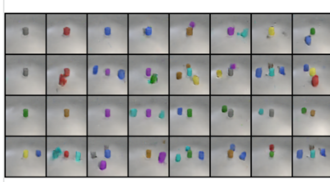
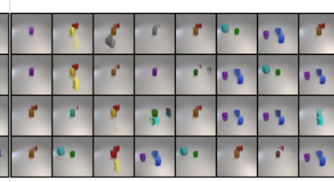
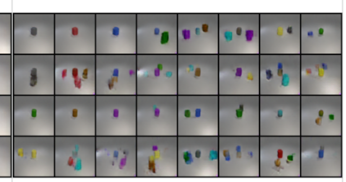
- Show your results based on the testing data (including images). (5%)

	DCGAN	ACGAN_aux_weight=1	ACGAN_aux_weight=128
Inference Image			
Accuracy	Sample 1: 68.06% Sample 2: 70.83% Sample 3: 70.83% Sample 4: 69.44% Sample 5: 70.83% Sample 6: 70.83% Sample 7: 69.44% Sample 8: 69.44% Sample 9: 69.44% Sample 10: 70.83% Average acc: 70.00% Process finished with exit code 0	Sample 1: 25.00% Sample 2: 16.67% Sample 3: 18.06% Sample 4: 20.83% Sample 5: 18.06% Sample 6: 18.06% Sample 7: 19.44% Sample 8: 13.89% Sample 9: 19.44% Sample 10: 19.44% Average acc: 18.89% Process finished with exit code 0	Sample 1: 63.89% Sample 2: 70.83% Sample 3: 72.22% Sample 4: 68.06% Sample 5: 69.44% Sample 6: 63.89% Sample 7: 70.83% Sample 8: 69.44% Sample 9: 66.67% Sample 10: 65.28% Average acc: 68.06% Process finished with exit code 0
Avg Accuracy	70.00%	18.89%	68.06%

上圖的結果接基於test.json，表現最好的是DCGAN，accuracy達到了70%。

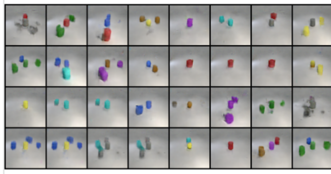
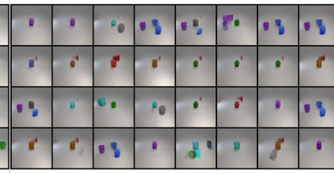
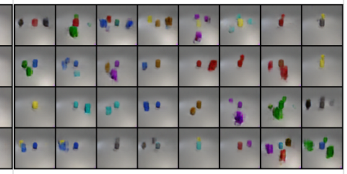
- Discuss the results of different model architectures. (25%)
  1. 在test.json下，DCGAN除了結果比較清楚以外，效果也比較好，但是儘管ACGAN_aux_weight=1的表現非常差，但當aux_weight上升時，會有相對好的表現。
  2. DCGAN的輸出圖片相對起來比較清楚，而ACGAN則相反，我認為原因是當aux_weight變大時，ACGAN更專注在分類問題上。
  3. ACGAN在aux_weight為128時，儘管train了400個epoch，效果仍是差於DCGAN，可能是我的aux_weight調太大導致。

- Experimental Results (50%)
  1. test.json

	DCGAN	ACGAN_aux_weight=1	ACGAN_aux_weight=128
Inference Image			
Accuracy	Sample 1: 68.06% Sample 2: 70.83% Sample 3: 70.83% Sample 4: 69.44% Sample 5: 70.83% Sample 6: 70.83% Sample 7: 69.44% Sample 8: 69.44% Sample 9: 69.44% Sample 10: 70.83% Average acc: 70.00% Process finished with exit code 0	Sample 1: 25.00% Sample 2: 16.67% Sample 3: 18.06% Sample 4: 20.83% Sample 5: 18.06% Sample 6: 18.06% Sample 7: 19.44% Sample 8: 13.89% Sample 9: 19.44% Sample 10: 19.44% Average acc: 18.89% Process finished with exit code 0	Sample 1: 63.89% Sample 2: 70.83% Sample 3: 72.22% Sample 4: 68.06% Sample 5: 69.44% Sample 6: 63.89% Sample 7: 70.83% Sample 8: 69.44% Sample 9: 66.67% Sample 10: 65.28% Average acc: 68.06% Process finished with exit code 0
Avg Accuracy	70.00%	18.89%	68.06%

表現最好的是DCGAN，accuracy達到了70.83%，avg accuracy達到70.00%。

2. new_test.json

	DCGAN	**ACGAN_aux_weight=1**	**ACGAN_aux_weight=128**
Inference Image			
Accuracy	Sample 1: 60.71% Sample 2: 60.71% Sample 3: 59.52% Sample 4: 63.10% Sample 5: 60.71% Sample 6: 59.52% Sample 7: 59.52% Sample 8: 58.33% Sample 9: 59.52% Sample 10: 59.52% Average acc: 60.12% Process finished with exit code 0	Sample 1: 9.52% Sample 2: 11.90% Sample 3: 8.33% Sample 4: 10.71% Sample 5: 9.52% Sample 6: 8.33% Sample 7: 8.33% Sample 8: 9.52% Sample 9: 9.52% Sample 10: 4.76% Average acc: 9.05% Process finished with exit code 0	Sample 1: 64.29% Sample 2: 70.24% Sample 3: 67.86% Sample 4: 67.86% Sample 5: 71.43% Sample 6: 70.24% Sample 7: 66.67% Sample 8: 67.86% Sample 9: 69.05% Sample 10: 65.48% Average acc: 68.10% Process finished with exit code 0
Avg Accuracy	60.12%	9.05%	68.10%

表現最好的是ACGAN_aux_weight=128，accuracy達到了71.43%，avg accuracy達到68.10%。