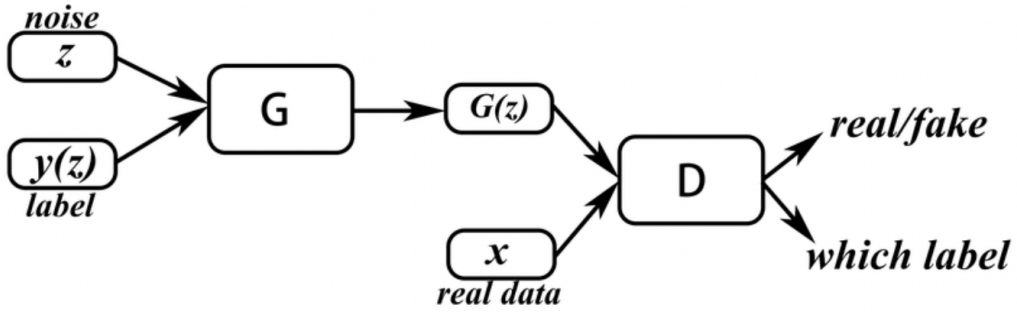
Deep Learning and Practice Lab5 – Let's Play GANs

310554009 杜葳葳

1. Introduction

GAN 的 Component 有 Generator 以及 Discriminator。 Generator 的任務就是產生圖片來騙過 Discriminator ， Discriminator 的任務就是努力判斷 Generator 所產生圖片的品質。



2. Implementation details

A. Describe how you implement your model, including your choice of cGAN, model architectures, and loss functions.

a. Dataset

分別讀入訓練資料與測試資料，並如同 lab3 和 lab4 的方式，先創建 Dataset 再迭代轉為 Dataloader。

b. Architecture:

使用 BCE 作為 Loss function，依序呼叫 Generator、Discriminator、Evaluation model，並使用 Adam 作為 optimizer。

```
adversarial_loss = torch.nn.BCELoss()

model_G = Generator()
model_D = Discriminator()
model_E = evaluation_model()

optimizer_G = torch.optim.Adam(model_G.parameters(), lr = learning_rate_G, betas = (B1, B2))
optimizer_D = torch.optim.Adam(model_D.parameters(), lr = learning_rate_D, betas = (B1, B2))

model_G.to(device)
model_D.to(device)
adversarial_loss.to(device)

best = 0
fix_z = torch.randn(32, LATENT_DIM).to(device)
```

c. Generator

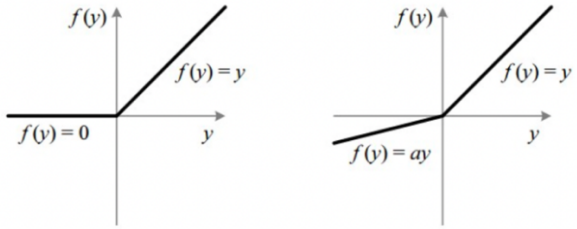
cGAN (條件生成對抗網路) 的 generator 為 noise 和 label 的聯合輸入，主要架構是以五層 transposed convolutional layers 和兩層 self-attention layers。

用一個 linear layer 將 condition 轉到 latent dimension，並將它跟 noise 接在一起，並加上兩個維度。接著，進行反卷積操作，採用 Spectral Normalization、以防止訓練不穩定，然後是 Batch normalization 層，這部分連續三次。再通過 self-attention layer + transposed convolutional layer 來降低 output 的維度。最後，再通過一個 self-attention layer + transposed convolutional layer，得到產生的 image。

```
class Generator(nn.Module):
    def __init__(self, conv_dim=64):
        super(Generator, self).__init__()
        self.conditionExpand = nn.Sequential(nn.Linear(24, LATENT_DIM), nn.LeakyReLU())

        repeat_num = int(np.log2(IMG_SIZE)) - 3
        mult = 2 ** repeat_num
        layer1 = []
        layer1.append(SpectralNorm(nn.ConvTranspose2d(LATENT_DIM + LATENT_DIM, conv_dim * mult, 4)))
        layer1.append(nn.BatchNorm2d(conv_dim * mult))
        layer1.append(nn.ReLU())

        curr_dim = conv_dim * mult
        layer2 = []
        layer2.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4, 2, 1)))
        layer2.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer2.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)
        layer3 = []
        layer3.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4, 2, 1)))
        layer3.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer3.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)
        layer4 = []
        layer4.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4, 2, 1)))
        layer4.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer4.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)
        last = []
        last.append(nn.ConvTranspose2d(curr_dim, 3, 4, 2, 1))
        last.append(nn.Tanh())

        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)
        self.l4 = nn.Sequential(*layer4)
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn(128, 'relu')
        self.attn2 = Self_Attn(64,  'relu')

    def forward(self, z, c):
        c = self.conditionExpand(c).view(-1, LATENT_DIM)
        z = torch.cat((z, c), dim=1)
        z = z.view(z.size(0), z.size(1), 1, 1)
        out = self.l1(z)
        out = self.l2(out)
        out = self.l3(out)
        out = self.attn1(out)
        out = self.l4(out)
        out = self.attn2(out)
        out = self.last(out)
        return out
```

d. Discriminator

與 Generator 比較不同的是，Discriminator 使用 Leaky ReLU 作為 activation function。



首先，使用 Linear layer 將 condition 投射到 img size x img size dimension，將它和 noise 相接，通過三層 Spectral Normalization + Batch normalization 的 layer，接著通過 self-attention layer + convolution layer 將 output 升維，最後，通過 self-attention + convolutional layer + sigmoid function，得到介於 0 到 1 之間的預測值。

```python
class Discriminator(nn.Module):
    def __init__(self, conv_dim=64):
        super(Discriminator, self).__init__()
        self.conditionExpand = nn.Sequential(nn.Linear(24, IMG_SIZE * IMG_SIZE), nn.LeakyReLU())

        layer1 = []
        layer1.append(SpectralNorm(nn.Conv2d(3 + 1, conv_dim, 4, 2, 1)))
        layer1.append(nn.LeakyReLU(0.1))

        curr_dim = conv_dim
        layer2 = []
        layer2.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer2.append(nn.LeakyReLU(0.1))

        curr_dim = curr_dim * 2
        layer3 = []
        layer3.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer3.append(nn.LeakyReLU(0.1))

        curr_dim = curr_dim * 2
        layer4 = []
        layer4.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
        layer4.append(nn.LeakyReLU(0.1))
        self.l4 = nn.Sequential(*layer4)

        curr_dim = curr_dim * 2
        last = []
        last.append(nn.Conv2d(curr_dim, 1, 4))
        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn(256, 'relu')
        self.attn2 = Self_Attn(512, 'relu')
        self.sigmoid = nn.Sigmoid()

    def forward(self, x, c):
        c = self.conditionExpand(c).view(-1, 1, IMG_SIZE, IMG_SIZE)
        x = torch.cat((x, c), dim=1)
        out = self.l1(x)
        out = self.l2(out)
        out = self.l3(out)
        out = self.attn1(out)
        out = self.l4(out)
        out = self.attn2(out)
        out = self.last(out)
        return self.sigmoid(out.squeeze(3).squeeze(2))
```

e. Training process

先用正確的照片訓練 Discriminator，Discriminator 應該預測為 Real，接著用 Generator 產生的照片來訓練 Discriminator，這時 Discriminator 應該要預測為 Fake，最後，我將 Generator 訓練兩次來增強模型表現。

```python
for epoch in tqdm(range(EPOCHS)):
    model_G.train()
    model_D.train()
    for imgs, labels in train_loader:
        imgs = imgs.to(device).float()
        labels =  labels.to(device).float()

        real = torch.full((imgs.size(0), 1), 1., requires_grad=False, device=device)
        fake = torch.full((imgs.size(0), 1), 0., requires_grad=False, device=device)

        # Train Discriminator with real image
        optimizer_D.zero_grad()
        pred = model_D(imgs, labels)
        real_loss = adversarial_loss(pred, real)
        real_loss.backward()
        optimizer_D.step()

        # Train Discriminator with fake image
        optimizer_D.zero_grad()
        z = torch.randn(imgs.size(0), LATENT_DIM).to(device)
        gen_imgs = model_G(z, labels)
        pred = model_D(gen_imgs.detach(), labels)
        fake_loss = adversarial_loss(pred, fake)
        fake_loss.backward()
        optimizer_D.step()
```

```python
        # Train Generator with fake image
        for _ in range(2):
            optimizer_G.zero_grad()
            z = torch.randn(imgs.size(0), LATENT_DIM).to(device)
            gen_imgs = model_G(z, labels)
            predicts = model_D(gen_imgs, labels)
            loss_g = adversarial_loss(predicts, real)
            loss_g.backward()
            optimizer_G.step()
```

f. Loss function: BCE Loss (Binary Cross Entropy Loss)

利用 BCE 來計算 target 和 output 之間的二元交叉熵，是 Cross entropy 的一個特例，只能用於二元分類、不能用於多分類，但因為包含正類和負類的 Loss，比 Cross entropy 有更多的資訊量。

$$\text{BCELoss}(O, T) = -\frac{1}{n}\sum_{i}\left((T[i] * \log(O[i])) + (1 - T[i]) * \log(1 - O[i])\right)$$

g. Testing process

在每個 epoch 的最後，使用 evaluation model 和固定 noise 來衡量 generator 的好壞，並將最佳的 model 存起來。

```
model_G.eval()
model_D.eval()
with torch.no_grad():
    for labels in test_loader:
        labels = labels.to(device).float()
        gen_imgs = model_G(fix_z, labels)
        score = model_E.eval(gen_imgs, labels)
        if best < score:
            best = score
            torch.save(model_G.state_dict(), 'generator.pt')
        save_image(gen_imgs[0].data, './output/%d.png' % epoch, nrow=8, normalize=True)
        print('EPOCHS {} : {}'.format(epoch, score))
```

B. Specify the hyperparameters

epochs: 2000

batch size: 64

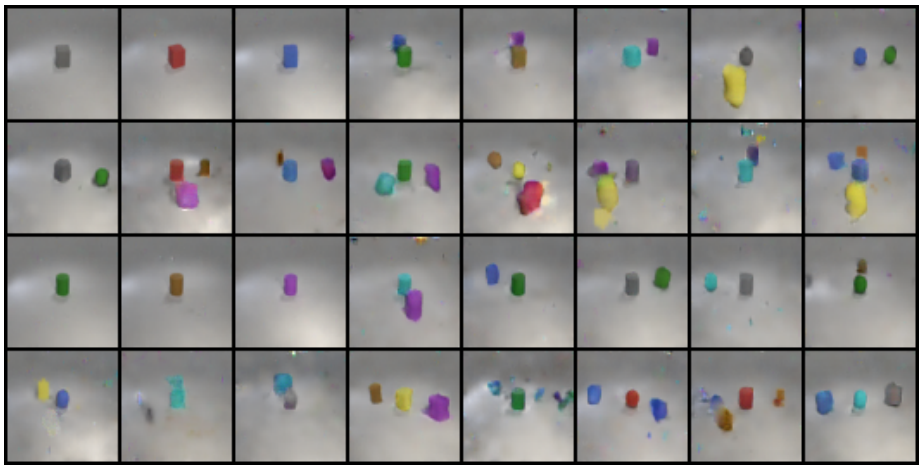
learning rate (generator): 1e-4

learning rate (discriminator): 4e-4

3. Results and discussion

A. Show your results based on the testing data.

```
/home/wwdu/anaconda3/envs/dl/lib/python3.6/site-packages/torch/nn/functi
onal.py:718: UserWarning: Named tensors and all their associated APIs ar
e an experimental feature and subject to change. Please do not use them
for anything important until they are released as stable. (Triggered int
ernally at  /pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation,
 ceil_mode)
0.8055555555555556
```

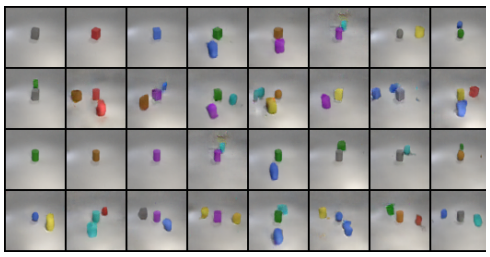


B. Discuss the results of different models architectures.

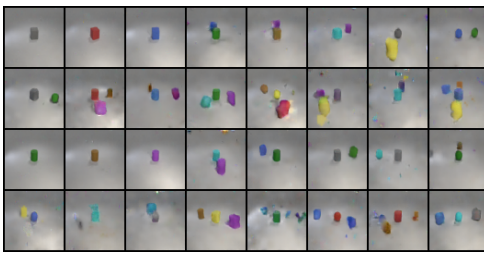
分別嘗試不同 epoch 數 1000, 2000, 3000，發現在單個 cube 的案例中，模型的表現都不錯，但在多個 cube 的處理上，隨 epoch 數上升，不管是 training loss 還是 testing score 都有逐漸變好的趨勢。

# epochs	1000	2000	3000
Testing score	65.27 %	**80.55 %**	76.38 %

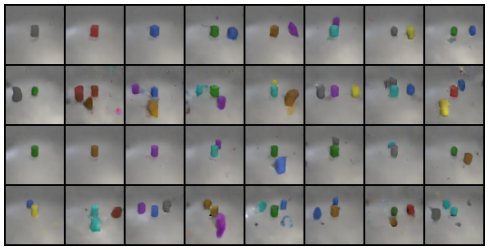
1000 個 epochs



3000 個 epochs

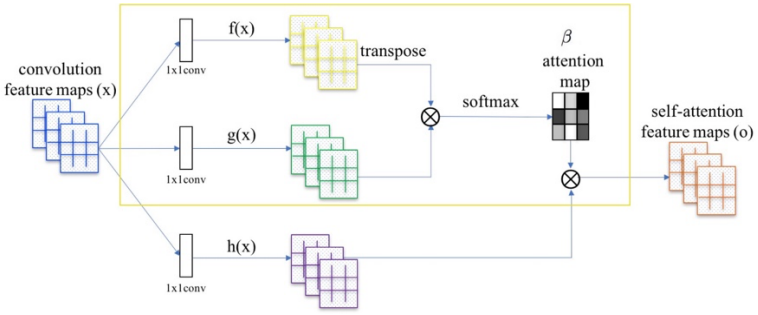


2000 個 epochs



一開始使用 ACGAN， Discriminator 的 input 只有圖片、沒有 label，故 Discriminator 要預測 real/fake 和 cube 的類別，嘗試的結果效果不好，在想可能是因為 cube 的類別有 24 類、有點多，Discriminator 無法判斷，所以後來採用 cGAN 的架構。

Generator 和 Discriminator 的設計採用 [Self-attention GAN]



比起 CNN 對周圍區域的理解，SAGAN 可以學到更長距離的資訊，主要在 Generator 和 Discriminator 都加入 Spectral normalization、並使用不同的 learning rate。