Lab7 Report

Introduction

用 cGAN 來生成立體幾何圖形的圖片。圖形包含 3 種形狀和 8 種顏色，總共 24 種幾何圖形。另外，一張圖片可以包含 1 至 3 個圖形。一張圖片可以有多個 target 代表無法使用 pytorch 中的 CrossEntropyLoss，因其不支援 multi-target 操作。這裡要改用 Binary Cross Entropy (BCELoss) 來實作。

```
criterion = nn.BCELoss()
```

Implementation details

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

我的架構使用的是 DCGAN 接上 condition。DCGAN 的架構、參數參考自 pytorch 官方的 DCGAN TUTORIAL (https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)。而 condition 採用直接 concatenate 的方式，串接的位置在 Generator 的 input layer 和 Discriminator 的 first hidden layer。下圖為 Generator 的 model 設計:

```python
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(100 + 100, 64 * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.ReLU(True),
            # state size. (64*8) x 4 x 4
            nn.ConvTranspose2d(64 * 8, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.ReLU(True),
            # state size. (64*4) x 8 x 8
            nn.ConvTranspose2d(64 * 4, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.ReLU(True),
            # state size. (64*2) x 16 x 16
            nn.ConvTranspose2d(64 * 2, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            # state size. (64) x 32 x 32
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. 3 x 64 x 64
        )

        self.condition_fc = nn.Linear(24, 100)

    def forward(self, input, cond):
        cond = self.condition_fc(cond).view(-1, 100, 1, 1)
        input = torch.cat((input, cond), dim=1)
        return self.main(input)
```

condition 的 size 為 batch_size * 24。先經過一層 Linear 擴充成 batch_size * 100，再與 input 串接(torch.cat)。串接前要用 view 調整 size 讓 input 和 condition 大小一致。因此，第一層反卷積 (nn.ConvTranspose2d) 的 input size 是 latent code z 的 100 加上 condition 的 100。接著是 Discriminator 的 model 設計:

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            # input is 3 x 64 x 64
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.main2 = nn.Sequential(
            # state size. (64) x 32 x 32
            nn.Conv2d(64 + 4, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64*2) x 16 x 16
            nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64*4) x 8 x 8
            nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (64*8) x 4 x 4
            nn.Conv2d(64 * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

        self.condition_fc = nn.Linear(24, 64*64)

    def forward(self, input, cond):
        #cond = self.condition_fc(cond).view(-1, 1, 64, 64)
        #input = torch.cat((input, cond), dim=1)

        output = self.main(input)  # 64*64*32*32
        cond = self.condition_fc(cond).view(-1, 4, 32, 32)
        output = torch.cat((output, cond), dim=1)
        output = self.main2(output)

        return output.view(-1)
```

Discriminator 分為 main 和 main2 兩層是因為經過第一層後要與 condition 串接，再進入第二層。這樣就能實現將 condition 串接在 first hidden layer。一樣先將 condition 過一層 Linear，這裡選擇 size 64*64 是為了方便調整大小以便串接。input 過了第一層 hidden layer 後的 size 是 64*64*32*32，因此將 condition 調整成 64*4*32*32，就可以成功串接起來。所以，第二層 hidden layer (nn.Conv2d) 的 input size 就是原先 input 的 64 加上 condition 的 4。

接下來是計算 loss 的部分。對於 Discriminator 而言，其目標是辨別真偽，也就是辨別圖片是否由 Generator 產生。用 objective function 來表示如下 (取自講義):

– **Discriminator cost** (cross-entropy cost)

$$J^{(D)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) = -E_{\boldsymbol{x}\sim p_{\text{data}}} \log D(\boldsymbol{x}) - E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}} \log(1 - D(G(\boldsymbol{z})))$$

where $D(\boldsymbol{x})$ denotes the probability of $\boldsymbol{x}$ being real

訓練方式為先從 training data 取出一個 batch 的真實資料傳入 Discriminator，並告訴 Discriminator 這些是真的:

```
############################
# (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
###########################
## Train with all-real batch
netD.zero_grad()
# Format batch
b_size = len(images)
label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
# Forward pass real batch through D
output = netD(images, conditions)
# Calculate loss on all-real batch
errD_real = criterion(output, label)
# Calculate gradients for D in backward pass
errD_real.backward()
D_x = output.mean().item()
```

接著固定 Generator 並從 Generator 生成同樣數量的假資料，並告訴 Discriminator 這些是假的:

```
## Train with all-fake batch
# Generate batch of latent vectors
noise = torch.randn(b_size, 100, 1, 1, device=device)
# Generate fake image batch with G
fake = netG(noise, conditions)
label.fill_(fake_label)
# Classify all fake batch with D
output = netD(fake.detach(), conditions)
# Calculate D's loss on the all-fake batch
errD_fake = criterion(output, label)
# Calculate the gradients for this batch, accumulated (summed) with previous gradients
errD_fake.backward()
# D(G(z)) before D update
D_G_z1 = output.mean().item()
```

最後把兩種 loss 的 gradient 加總起來更新。loss function 就用 Introduction 提到的 BCELoss:

```
# Compute error of D as sum over the fake and the real batches
errD = errD_real + errD_fake
# Update D
optimizerD.step()
d_losses += errD.item()
```

經過這樣的訓練，Discriminator 辨別的功力就會越來越好。

然後是 Generator。Generator 的目標就是讓 Discriminator 辨別不出來。寫成 objective function 就是 (取自講義):

– **Generator cost**

$$J^{(G)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) = -J^{(D)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)})$$

訓練方式就是固定 Discriminator，並從 Generator 生成一個 batch 的資料，並希望 Discriminator 認為它是真的。loss function 一樣使用 BCELoss:

```
############################
# (2) Update G network: maximize log(D(G(z)))
###########################
netG.zero_grad()
noise = torch.randn(b_size, 100, 1, 1, device=device)
fake = netG(noise, conditions)
label.fill_(real_label) # fake labels are real for generator cost
# Since we just updated D, perform another forward pass of all-fake batch through D
output = netD(fake, conditions)
# Calculate G's loss based on this output
errG = criterion(output, label)
# Calculate gradients for G
errG.backward()
# D(G(z)) after D update
D_G_z2 = output.mean().item()
# Update G
optimizerG.step()
```

經過這樣的訓練，Generator 生成的圖片就會越來越像真的。

綜合以上，整個訓練過程就是固定 G、訓練 D，固定 D、訓練 G

這樣反覆交替運行、互相對抗，並希望兩者一起越變越好。但有時不會這麼順利，若 Generator、Discriminator 其中一方太強，就會導致對抗的平衡被打破。而這個 model 在這個 lab 就有 Discriminator 太強，導致 Generator 幾乎沒有梯度可以更新的問題。這個問題在下方 discussion 部分進行探討。

Specify the hyperparameters
learning rate = 0.0002
batch size = 64
number of epochs = 250
D_iter = 1, G_iter = 5 (Dx1, Gx5)
(在每一個 epoch 中，Discriminator train 1 次、Generator train 5 次)

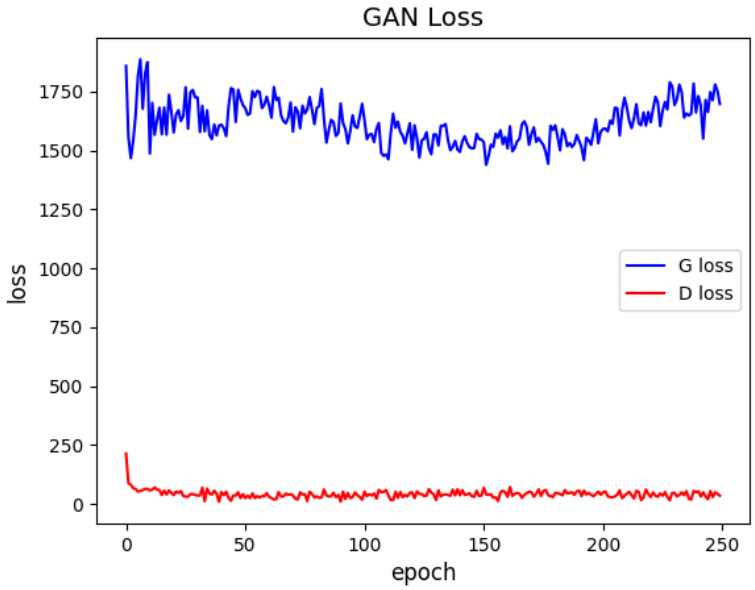
Results and discussion

Show your results based on the testing data

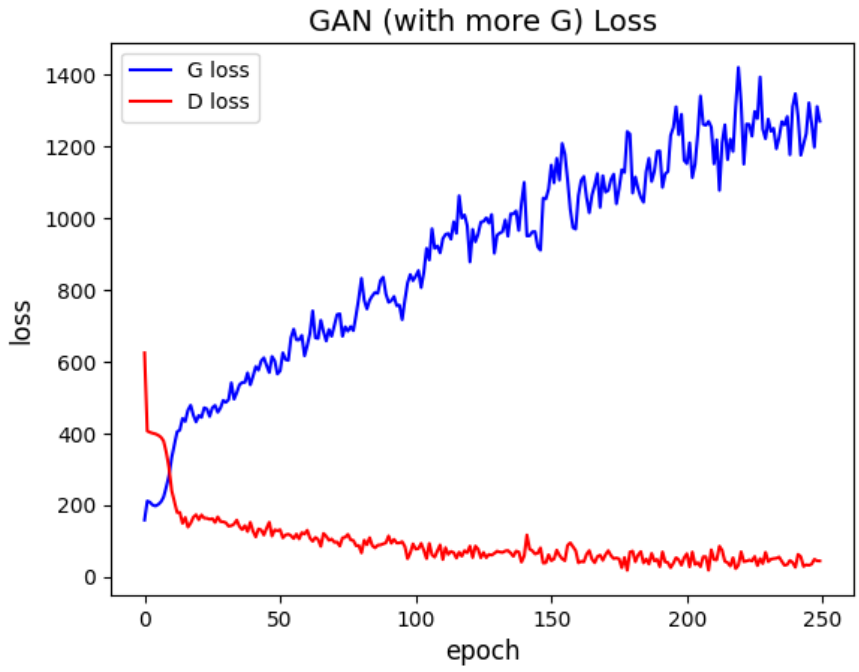
皆呈現於下方 Experimental Result 部分。

Discuss the results of different models architectures

在 Implementation 後段有提到，原本的 model 在此 lab 會產生 Discriminator dominated 的問題。將 training loss 繪製成折線圖更可以觀察到此情況:



我們可以發現，Discriminator 的 loss 一直處於很低的數值。代表 Discriminator 從頭到尾都是比 Generator 強很多的。若稍微修改 train 的過程，原本是一次 D、一次 G、一次 D、一次 G...，改成一次 D、五次 G、一次 D、五次 G...。這樣 G train 的次數就會是 D 的 5 倍。我期待這種方式能夠使兩者的 loss 更加接近。而結果如下:



從圖中可以發現，這個策略在前期是有效的，且 G 和 D 整體的 loss 也比原本的還要低。這也反映在 test 的正確率上:

(using test.json; setting manual seed=10)

Dx1 Gx1 (origin)	Dx1 Gx5 (modified)
score: 0.6805555555555556 score: 0.6805555555555556 score: 0.6944444444444444 score: 0.6805555555555556 score: 0.6944444444444444 score: 0.6805555555555556 score: 0.6805555555555556 score: 0.6805555555555556 score: 0.7083333333333334 score: 0.6944444444444444 max score: 0.7083333333333334 avg score: 0.6874999999999999	score: 0.7222222222222222 score: 0.7083333333333334 score: 0.7083333333333334 score: 0.7222222222222222 score: 0.7222222222222222 score: 0.7083333333333334 score: 0.6944444444444444 score: 0.6944444444444444 score: 0.7083333333333334 score: 0.7222222222222222 max score: 0.7222222222222222 avg score: 0.7111111111111111

然而，這並不能從根本解決此問題。我們會發現這個策略只在前期會有效果，雖然從頭到尾 Generator 都 train 得比較多次，但過了前期之後，Discriminator 還是完全輾壓了 Generator。因此可能要從 model 架構上或 loss 計算上下手。例如我嘗試過使用 WGAN，但反而導致 Generator 壓制了 Discriminator，出來的效果甚至沒有比原本好。我認為應該是我有地方做錯或參數設定不當，這裡就不多做討論。

另一方面，condition 串接的位置也會對整個 model 造成影響。一開始我將 condition 都串接在 input 位置，後來試了一下，發現對 Discriminator 來說，串接在第一層 hidden layer 似乎表現得更好。以下是兩者比較:

(using test.json; setting manual seed=10; using model Dx1, Gx5)

G, D cat in input	G cat in input, D cat in first hidden
score: 0.7222222222222222 score: 0.7083333333333334 score: 0.7083333333333334 score: 0.7222222222222222 score: 0.7222222222222222 score: 0.7083333333333334 score: 0.6944444444444444 score: 0.6944444444444444 score: 0.7083333333333334 score: 0.7222222222222222 max score: 0.7222222222222222 avg score: 0.7111111111111111	score: 0.7361111111111112 score: 0.7222222222222222 score: 0.75 score: 0.7361111111111112 score: 0.7361111111111112 score: 0.7222222222222222 score: 0.7222222222222222 score: 0.75 score: 0.7361111111111112 score: 0.7361111111111112 max score: 0.75 avg score: 0.7347222222222222

Experimental Result

Model: cGAN trained with Dx1, Gx5.
Condition concatenated in G's input and D's first hidden layer.

隨機 sample z 10 次，並取得 10 次 test 的最高分及平均分。

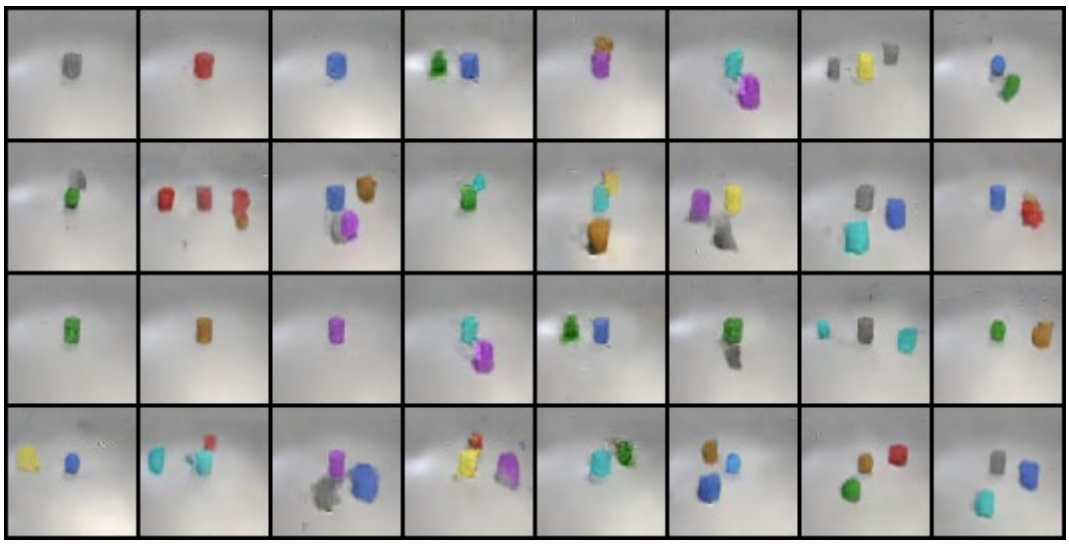
test.json

```
score: 0.7361111111111112
score: 0.7222222222222222
score: 0.75
score: 0.7361111111111112
score: 0.7361111111111112
score: 0.7222222222222222
score: 0.7222222222222222
score: 0.75
score: 0.7361111111111112
score: 0.7361111111111112

max score: 0.75
avg score: 0.7347222222222222
```

best score = 0.75

best generated image:



new_test.json

```
score: 0.6190476190476191
score: 0.5952380952380952
score: 0.5833333333333334
score: 0.6071428571428571
score: 0.6309523809523809
score: 0.5714285714285714
score: 0.6071428571428571
score: 0.5952380952380952
score: 0.6190476190476191
score: 0.6071428571428571

max score: 0.6309523809523809
avg score: 0.6035714285714285
```

best score = 0.63
best generated image: