

# CSC413: Programming Assignment 4

Tianyu Du (1003801647)

2020/03/29 at 15:39:34

## 1 Part 1: Deep Convolutional GAN (DCGAN) [4pt]

### 1.1 Generator

**Implementation** Note that I reshaped the input in `forward` method as well.

```
1 class DCGenerator(nn.Module):
2     def __init__(self, noise_size, conv_dim, spectral_norm=False):
3         super(DCGenerator, self).__init__()
4
5         self.conv_dim = conv_dim # 32
6
7         #####
8         ## FILL THIS IN: CREATE ARCHITECTURE ##
9         #####
10        self.linear_bn = nn.Sequential(
11            nn.Linear(noise_size, self.conv_dim*4*4*4),
12            nn.BatchNorm1d(self.conv_dim*4*4*4)
13        ) # (100, 2048)
14        self.upconv1 = upconv(self.conv_dim*4, self.conv_dim*2, 5, stride=2, padding=2,
15        batch_norm=True, spectral_norm=spectral_norm)
16        self.upconv2 = upconv(self.conv_dim*2, self.conv_dim, 5, stride=2, padding=2,
17        batch_norm=True, spectral_norm=spectral_norm)
18        self.upconv3 = upconv(self.conv_dim, 3, 5, stride=2, padding=2, batch_norm=False,
19        spectral_norm=spectral_norm)
20
21    def forward(self, z):
22        """
23        ...
24        """
25        batch_size = z.size(0)
26        #####
27        ##             RESHAPED HERE AS WELL           ##
28        #####
29        z = z.view(batch_size, -1)
30        out = F.relu(self.linear_bn(z)).view(-1, self.conv_dim*4, 4, 4)      # BS x 128 x 4 x
31
32        out = F.relu(self.upconv1(out)) # BS x 64 x 8 x 8
33        out = F.relu(self.upconv2(out)) # BS x 32 x 16 x 16
34        out = F.tanh(self.upconv3(out)) # BS x 3 x 32 x 32
35
36        out_size = out.size()
37        if out_size != torch.Size([batch_size, 3, 32, 32]):
38            raise ValueError("expect {} x 3 x 32 x 32, but get {}".format(batch_size,
39                             out_size))
40        return out
```

## 1.2 Training Loop

### Implementation

```

1 def gan_training_loop(dataloader, test_dataloader, opts):
2     ...
3
4     for d_i in range(opts.d_train_iters):
5         d_optimizer.zero_grad()
6
7         # FILL THIS IN
8         # 1. Compute the discriminator loss on real images
9         D_real_loss = torch.mean((D(real_images) - 1) ** 2) / 2
10
11        # 2. Sample noise
12        noise = sample_noise(real_images.shape[0], opts.noise_size)
13
14        # 3. Generate fake images from the noise
15        fake_images = G(noise)
16
17        # 4. Compute the discriminator loss on the fake images
18        D_fake_loss = torch.mean(D(fake_images) ** 2) / 2
19
20        # ---- Gradient Penalty ----
21        if opts.gradient_penalty:
22            ...
23        else:
24            gp = 0.0
25
26        # -----
27        # 5. Compute the total discriminator loss
28        D_total_loss = D_real_loss + D_fake_loss + gp
29
30        D_total_loss.backward()
31        d_optimizer.step()
32
33 ######
34 #####          TRAIN THE GENERATOR          #####
35 ##########
36
37 g_optimizer.zero_grad()
38
39 # FILL THIS IN
40 # 1. Sample noise
41 noise = sample_noise(real_images.shape[0], opts.noise_size)
42
43 # 2. Generate fake images from the noise
44 fake_images = G(noise)
45
46 # 3. Compute the generator loss
47 G_loss = torch.mean((D(fake_images) - 1)**2)
48
49 G_loss.backward()
50 g_optimizer.step()
51
52

```

### 1.3 Experiments

**Experiment 1** These two figures below shows the generated results at 1,000 iterations and 20,000 iterations without gradient penalty. The generated results at 20,000 are much more sharper (i.e., with clearer boundaries) than 1,000 iteration results, but colours of generated samples after 20,000 iterations are darker.

Figure 1.1: 1,000 iters (left) and 20,000 iters (right)



**Experiment 2** Two figures below are generated samples from a GAN with gradient penalty after being trained for 1,000 iterations and 20,000 iterations. Compared samples in figure 1.1, these samples are now with even clearer boundaries, and colours are more vivid. The gradient penalty term added prevent gradient exploding while training the discriminator, and helps stabilize the training. So that adding gradient penalty improve generalizability of the model.

Figure 1.2: 1,000 iters (left) and 20,000 iters (right)



## 2 Part 2: CycleGAN [3pt]

### 2.1 CycleGAN Experiments

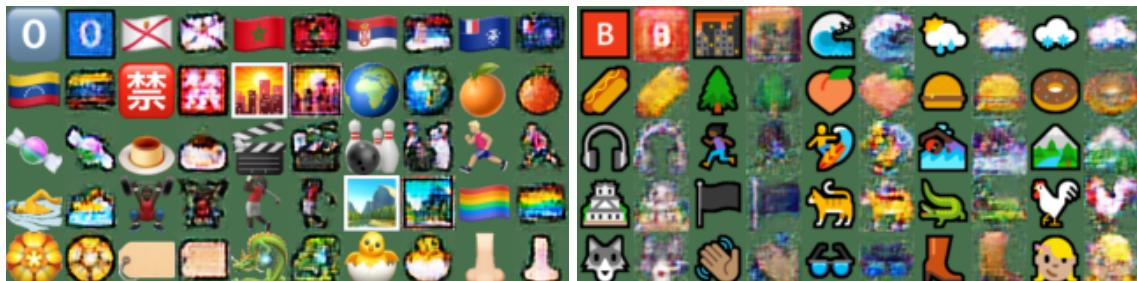
#### 2.1.1 Question 1

**Answer** Default configurations at 200 iterations:

Figure 2.1: Iteration 200, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



Figure 2.2: Iteration 5000, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



#### 2.1.2 Question 2

**Answer** Changed the random seed to `SEED = 42` (originally 4). With the new random seed, some images in the final  $Y \rightarrow X$  translations are more clear and with less background noise. Such as boot (row 5, item 4), mountain (row 3, item 5), and donut (row 2, item 5).

[Explain differences](#)

Figure 2.3: Iteration 200, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



Figure 2.4: Iteration 5000, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



### 2.1.3 Question 3

[Add comment](#)

**Comments** I tried three different level of lambda values for the cycle consistency, translation results are attached below:

**Configuration 1** `lambda_cycle = 0`, all other hyper-parameters are default, including the random seed.

Figure 2.5: `lambda_cycle = 0`, iteration 200, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$

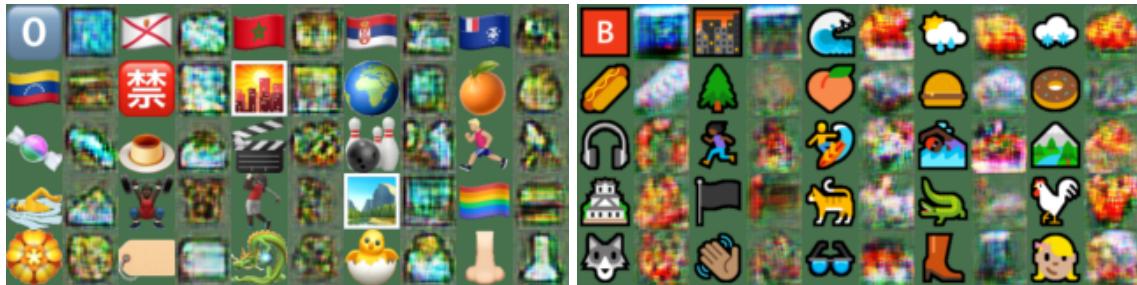


Figure 2.6: `lambda_cycle = 0`, iteration 5000, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



**Configuration 2** `lambda_cycle = 0.03`, all other hyper-parameters are default, including the random seed.

Figure 2.7:  $\text{lambda\_cycle} = 0.03$ , iteration 200, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



Figure 2.8:  $\text{lambda\_cycle} = 0.03$ , iteration 5000, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



**Configuration 3**  $\text{lambda\_cycle} = 0.3$ , all other hyper-parameters are default, including the random seed.

Figure 2.9:  $\text{lambda\_cycle} = 0.3$ , iteration 200, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



Figure 2.10:  $\text{lambda\_cycle} = 0.3$ , iteration 5000, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



**Configuration 4**  $\text{lambda\_cycle} = 1.0$ , all other hyper-parameters are default, including the random seed.

Figure 2.11: `lambda_cycle = 1.0`, iteration 200, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



$t$

Figure 2.12: `lambda_cycle = 1.0`, iteration 5000, left:  $X \rightarrow Y$ , right:  $Y \rightarrow X$



### 3 Part 3: BigGAN [2pt]

#### 3.1 BigGAN Experiments

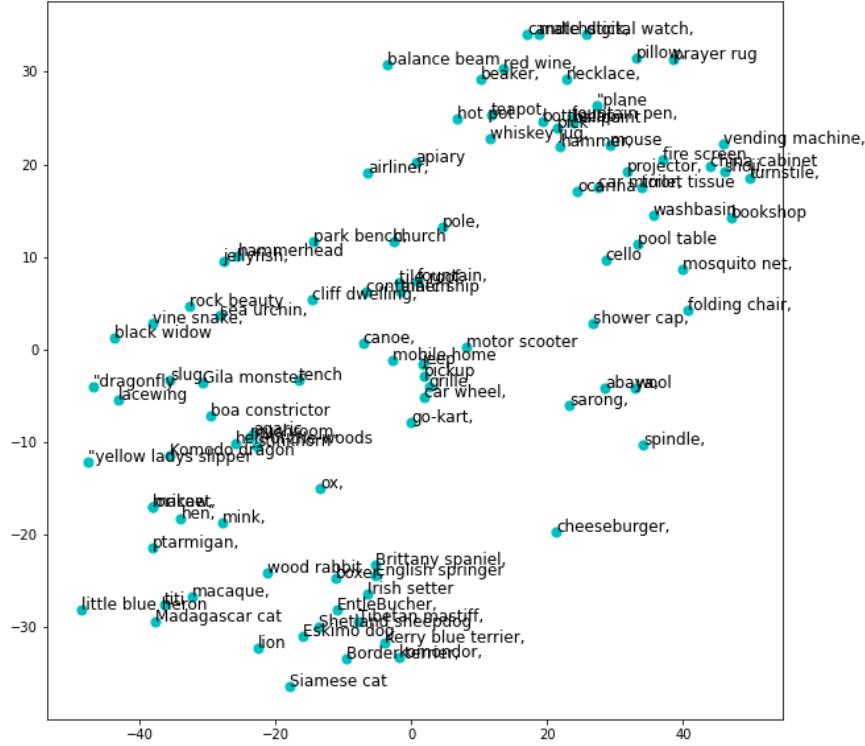
##### 3.1.1 Question 1

**Method** Given two classes with embeddings  $\mathbf{r}_{class1}$  and  $\mathbf{r}_{class2}$  in T-SNE space, to decide whether they are good candidate for linear interpolation, I looked at convex combinations of two classes' embeddings in figure 3.1. Let  $\Phi$  denote the set of convex combinations:

$$\Phi = \{\alpha \mathbf{r}_{class1} + (1 - \alpha) \mathbf{r}_{class2} : \alpha \in [0, 1]\} \quad (3.1)$$

Linear interpolations between these two classes are basically taking points in set  $\Phi$  and visualizing them. If there are many other classes' embeddings on or near the set  $\Phi$ , points in  $\Phi$  are likely to be associated with meaningful visualizations. Therefore, linear interpolations between these two classes are likely to generate meaningful results. Otherwise, these two classes are not good candidates for linear interpolation.

Figure 3.1: T-SNE Plot



**Good Candidate Match 1** There are many classes near the segment between *folding chair* and *vending machine* (right-up corner), so linear interpolation should be meaningful.

**Good Candidate Match 2** Similarly, there are many classes on the line between bookshop and hot pot (right-up corner), so they are ideal for linear interpolation.

**Bad Candidate Match 1** There are no other classes' embeddings are in between embeddings of Go-Kart and Chessburger in T-SNE plot, so they are not good match.

**Bad Candidate Pair 2** Similarly, Ox and Chessburger are not good match.

### 3.1.2 Question 2

## Implementation

```

1 #Linear interpolation between two class embeddings
2 def generate_linear_interpolate_sample(G, batch_size, class_label1, class_label2, alpha):
3     G.eval()
4     G.to(DEVICE)
5     with torch.no_grad():
6         z = torch.randn(batch_size, G.dim_z).to(DEVICE)
7         class1_emb = G.shared(torch.tensor(class_label1).to(DEVICE)*torch.ones((batch_size,))
8 ).to(DEVICE).long())
9         class2_emb = G.shared(torch.tensor(class_label2).to(DEVICE)*torch.ones((batch_size,))
10 ).to(DEVICE).long())
11
12 #####
```

```

11     ##      FILL THIS IN: CREATE NEW EMBEDDING  ##
12 #####
13 new_emb = alpha * class1_emb + (1 - alpha) * class2_emb
14
15 images = G(z, new_emb)
16 return images
17

```

## Linear Interpolation Results

Figure 3.2: Good Match 1: Folding Chair and Vending Machine



Figure 3.3: Good Match 2 Hot Pot and Bookshop



Figure 3.4: Bad Match 1 Go-Kart and Chessburger



Figure 3.5: Bad Match 2 Ox and Chessburger

