CSC413: Programming Assignment 4

Tianyu Du (1003801647)

2020/03/31 at 22:37:02

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

1.1 Generator

Implementation

```
class DCGenerator(nn.Module):
    def __init__(self, noise_size, conv_dim, spectral_norm=False):
        super(DCGenerator, self).__init__()

        self.conv_dim = conv_dim # 32

        self.linear_bn = upconv(100, 128, kernel_size=4, stride=3, padding=2, batch_norm=True)
        self.upconv1 = upconv(self.conv_dim*4, self.conv_dim*2, 5, stride=2, padding=2, batch_norm=True, spectral_norm=spectral_norm)
        self.upconv2 = upconv(self.conv_dim*2, self.conv_dim, 5, stride=2, padding=2, batch_norm=True, spectral_norm=spectral_norm)
        self.upconv3 = upconv(self.conv_dim, 3, 5, stride=2, padding=2, batch_norm=False, spectral_norm=spectral_norm)

```

1.2 Training Loop

Implementation

```
def gan_training_loop(dataloader, test_dataloader, opts):
    ...

            for d_i in range(opts.d_train_iters):
                d_optimizer.zero_grad()

                # FILL THIS IN
                # 1. Compute the discriminator loss on real images
                D_real_loss = torch.mean((D(real_images) - 1) ** 2) / 2

                # 2. Sample noise
                noise = sample_noise(real_images.shape[0], opts.noise_size)

                # 3. Generate fake images from the noise
                fake_images = G(noise)

                # 4. Compute the discriminator loss on the fake images
                D_fake_loss = torch.mean(D(fake_images) ** 2) / 2

                # ---- Gradient Penalty ----
```

```
            if opts.gradient_penalty:
                ...
            else:
                gp = 0.0

            # -------------------------
            # 5. Compute the total discriminator loss
            D_total_loss = D_real_loss + D_fake_loss + gp

            D_total_loss.backward()
            d_optimizer.step()

        ########################################
        ###          TRAIN THE GENERATOR       ###
        ########################################

        g_optimizer.zero_grad()

        # FILL THIS IN
        # 1. Sample noise
        noise = sample_noise(real_images.shape[0], opts.noise_size)

        # 2. Generate fake images from the noise
        fake_images = G(noise)

        # 3. Compute the generator loss
        G_loss = torch.mean((D(fake_images) - 1)**2)

        G_loss.backward()
        g_optimizer.step()
        ...

```

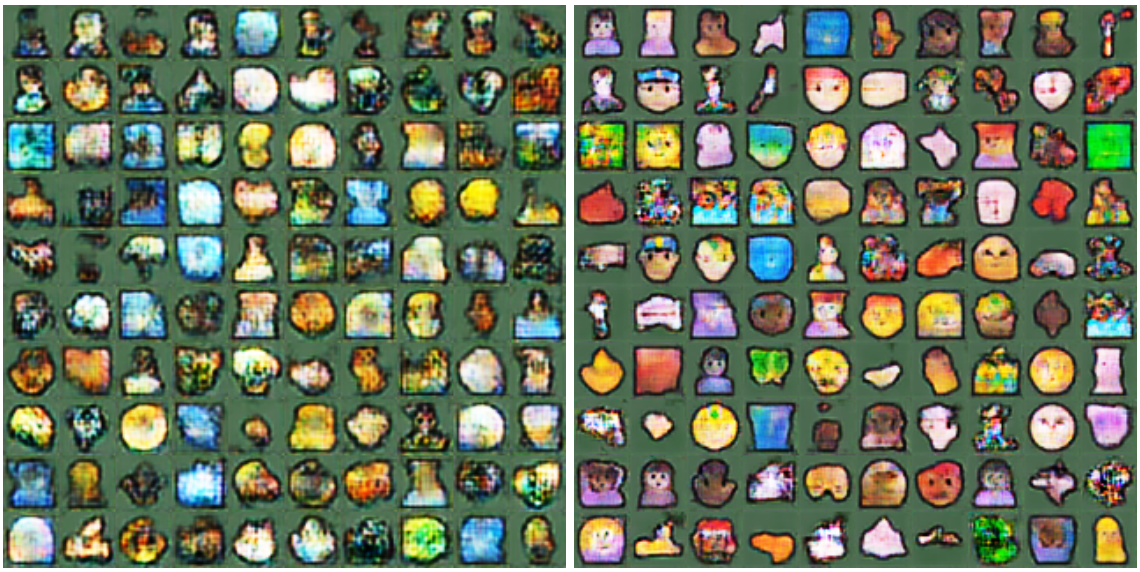
1.3 Experiments

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 shaper (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 Note: this result is generated using the original starter code. 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: $X \to Y$, right: $Y \to X$

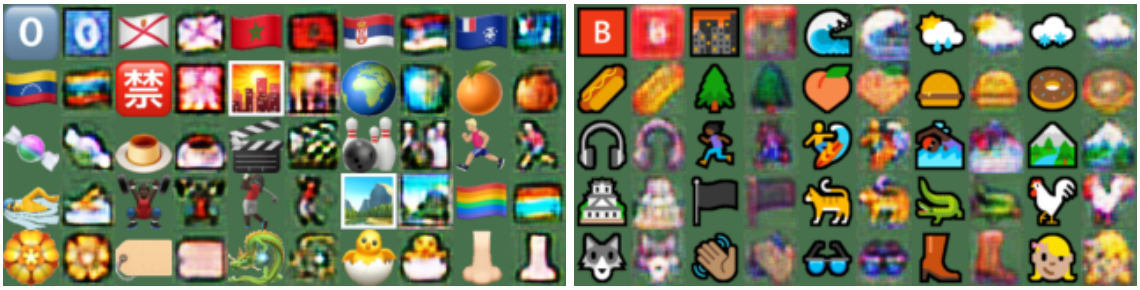


Figure 2.2: Iteration 5000, left: $X \to Y$, right: $Y \to X$

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 $Y \to X$ 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). Since the weights of generator and discriminator are randomly initialized, therefore, the results are different using another random seed.

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

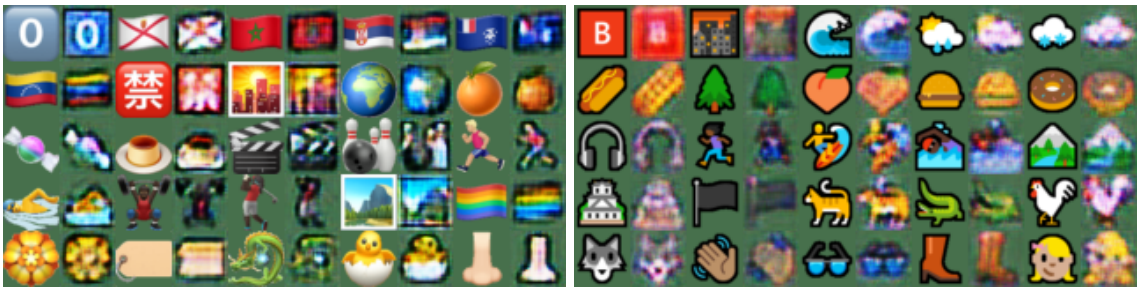


Figure 2.4: Iteration 5000, left: $X \to Y$, right: $Y \to X$

2.1.3 Question 3

Comments I tried three different level of lambda values for the cycle consistency, translation results are attached below. It turns out that using a larger value of `lambda_cycle` makes the reconstructed figure more similar to the original figure. Moreover, with a larger value, the colour reconstruction is more accurate and the background noises are reduced.

Without enforcing cycle consistency, given a set of input images, the network can generate any random permutation of input images in the target domain (i.e., another style of emoji). Any of these permutation generates the same loss. Therefore, only using the loss without enforcing cycle consistency cannot ensure that the learned function can map an individual input to the corresponding desired output, and the output images tend to be blurred and have low qualities. Adding cycle consistency guarantees efficient training and high quality of output.

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

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

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

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

Figure 2.7: `lambda_cycle = 0.03`, iteration 200, left: $X \to Y$, right: $Y \to X$

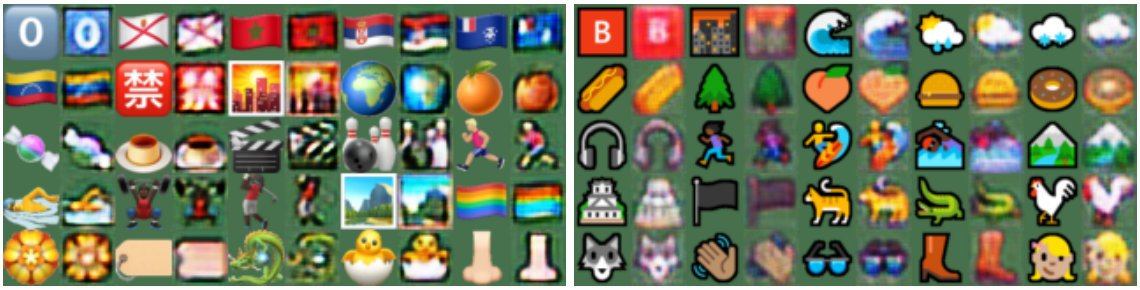
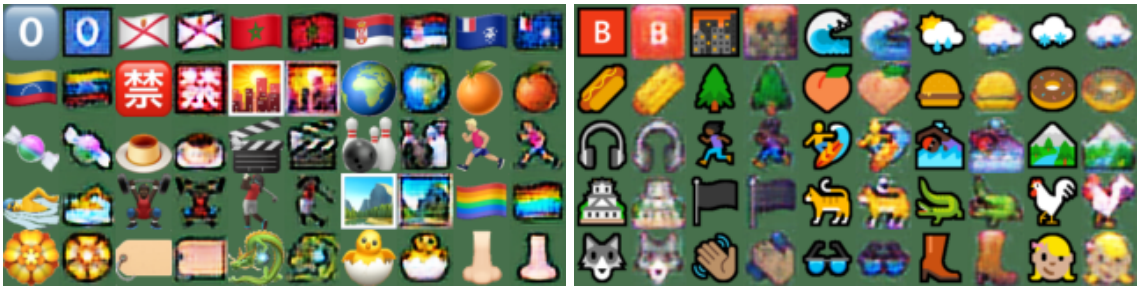


Figure 2.8: `lambda_cycle = 0.03`, iteration 5000, left: $X \to Y$, right: $Y \to X$



Configuration 3 `lambda_cycle = 0.3`, all other hyper-parameters are default, including the random seed.

Figure 2.9: `lambda_cycle = 0.3`, iteration 200, left: $X \to Y$, right: $Y \to X$

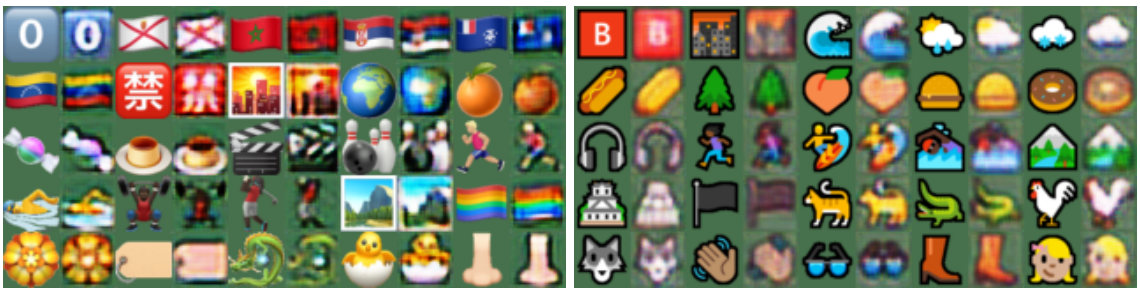
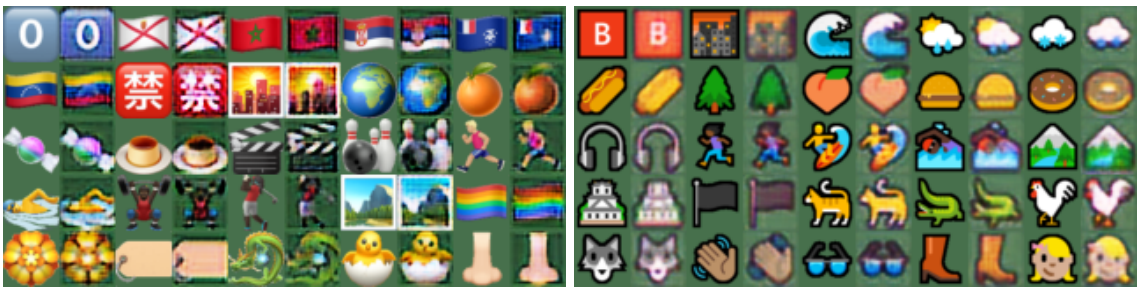
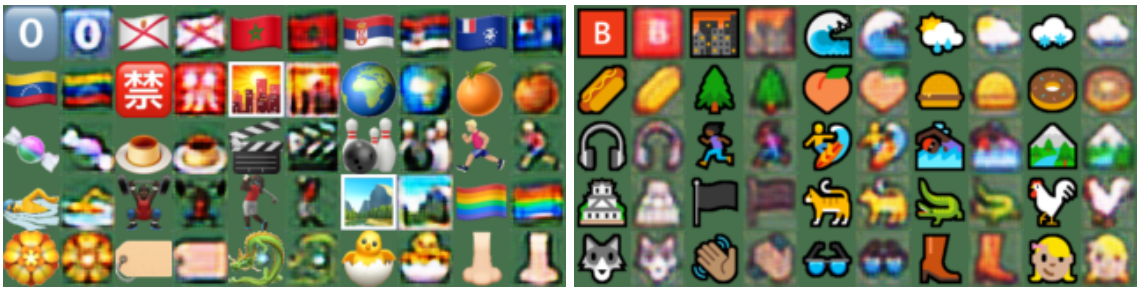


Figure 2.10: `lambda_cycle = 0.3`, iteration 5000, left: $X \to Y$, right: $Y \to X$



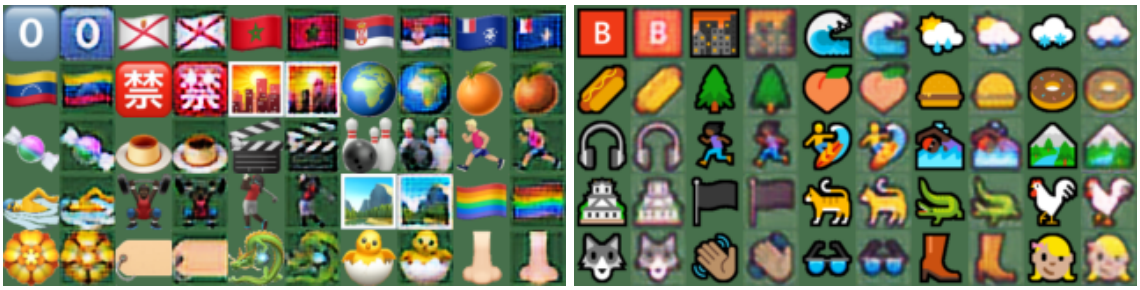
Configuration 4 `lambda_cycle = 1.0`, all other hyper-parameters are default, including the random seed.

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



t

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



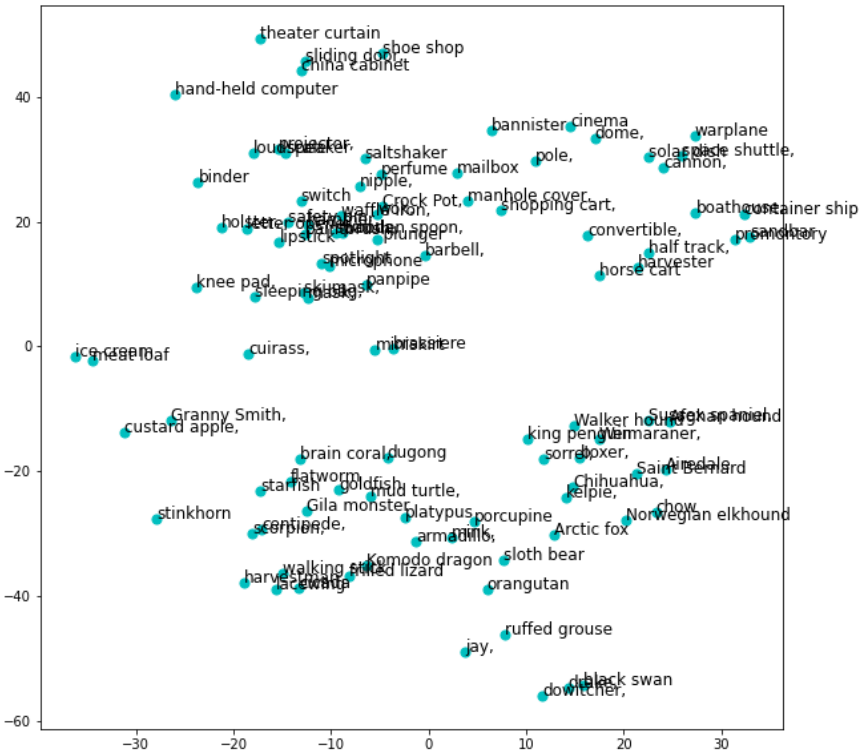
3 Part 3: BigGAN [2pt]

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 plot, to decide whether they are good candidate for linear interpolation by checking i) whether they are close in the T-SNE plot (so these two classes are similar) and ii) whether they are in the same cluster in T-SNE plot.

Figure 3.1: T-SNE Plot



Good Matches `Chihuahua` and `king penguin` are both animals so they are closed to each other in the T-SNE plot. Moreover, they are in the same cluster (right bottom). Similarly, `jay` and `sloth bear` are good matches as well (left bottom).

Bad Matches `horse cart` and `panpipe` are not similar and they are far away from each other in the T-SNE plot, so they are not good matches. Similarly, `hand-held computer` and `bannister` are not good matches neither.

3.1.2 Question 2

Implementation

```
#Linear interpolation between two class embedings
def generate_linear_interpolate_sample(G, batch_size, class_label1, class_label2, alpha):
    G.eval()
    G.to(DEVICE)
    with torch.no_grad():
        z = torch.randn(batch_size, G.dim_z).to(DEVICE)
        class1_emb = G.shared(torch.tensor(class_label1).to(DEVICE)*torch.ones((batch_size,)
    ).to(DEVICE).long())
        class2_emb = G.shared(torch.tensor(class_label2).to(DEVICE)*torch.ones((batch_size,)
    ).to(DEVICE).long())

        ##########################################
        ##   FILL THIS IN: CREATE NEW EMBEDDING  ##
        ##########################################
        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 Matches: `Chihuahua` and `king penguin`

Figure 3.3: Good Matches: `jay` and `sloth bear`

Figure 3.4: Bad Matches: `horse cart` and `panpipe`

Figure 3.5: Bad Matches: `hand-held computer` and `bannister`