CSC413PA4

Che Liu, 1002246839

April 1, 2020

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

1. Implementation: Implement this architecture by filling in the __init__ method of the DCGenerator class, shown below. Note that the forward pass of DCGenerator is already provided for you. [1pt]

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

        self.conv_dim = conv_dim
        print(conv_dim)
        ########################################
        ##   FILL THIS IN: CREATE ARCHITECTURE   ##
        ########################################
        # 1D conv for fc
        self.linear_bn = upconv(in_channels=noise_size, out_channels=128*4*4, kernel_size=1,\
                                padding=0, stride=0)
        self.upconv1 = upconv(in_channels=128, out_channels=64, kernel_size=5)
        self.upconv2 = upconv(in_channels=64, out_channels=32, kernel_size=5)
        self.upconv3 = upconv(in_channels=32, out_channels=3, kernel_size=5, batch_norm=False)

```

Figure 1: Screenshot of __init__ of DCGenerator

2. Implementation: Fill in the gan_training_loop function in the GAN section of the notebook. [1pt]

```
# FILL THIS IN
# 1. Compute the discriminator loss on real images
m = real_images.shape[0]
D_real_loss = torch.sum((D(real_images)-1)**2)/(2*m)

# 2. Sample noise
noise = sample_noise(m, 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.sum(D(fake_images)**2)/(2*m)

# ---- Gradient Penalty ----
if opts.gradient_penalty:
    alpha = torch.rand(real_images.shape[0], 1, 1, 1)
    alpha = alpha.expand_as(real_images).cuda()
    interp_images = Variable(alpha * real_images.data + alpha * fake_images.data, requires_grad=True).cuda()
    D_interp_output = D(interp_images)

    gradients = torch.autograd.grad(outputs=D_interp_output, inputs=interp_images,
                                    grad_outputs=torch.ones(D_interp_output.size()).cuda(),
                                    create_graph=True, retain_graph=True)[0]
    gradients = gradients.view(real_images.shape[0], -1)
    gradients_norm = torch.sqrt(torch.sum(gradients ** 2, dim=1) + 1e-12)

    gp = gp_weight * gradients_norm.mean()
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()
```

Figure 2: Screenshot of DCGAN discriminator loss implementation

```
# FILL THIS IN
# 1. Sample noise
noise = sample_noise(m, opts.noise_size)

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

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

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

Figure 3: Screenshot of DCGAN generator loss implementation

3. How does the generator performance evolve over time? Include in your write-up some representative samples (e.g. one early in the training, one with satisfactory image quality, and one towards the end of training, and give the iteration number for those samples. Briefly comment on the quality of the samples. [1pt]

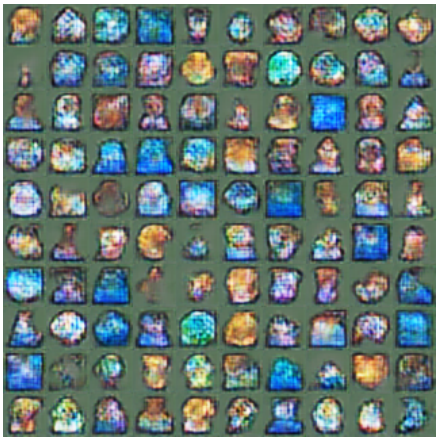

Figure 4: Samples after 1000 iterations

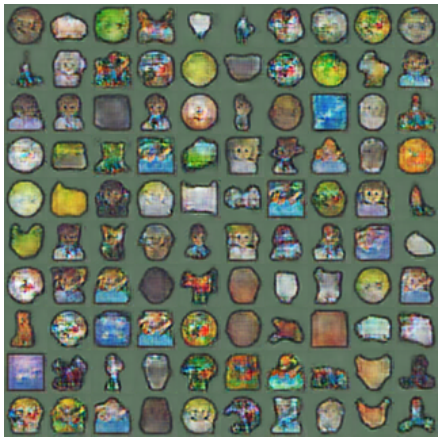

Figure 5: Samples after 10000 iterations

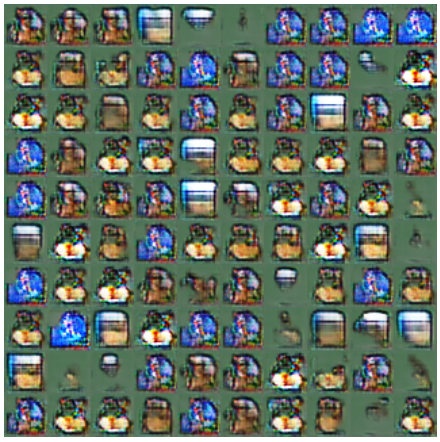

Figure 6: Samples after 20000 iterations

The samples after training for 1000, 10000, and 20000 iteration are shown in the figures above. It seems that the quality of the samples improved and then deteriorated over time. After 1000 iterations, the samples are of very limited resolution and it is hard to tell what the emojis are. After 10000 iterations, the quality improved significantly and one can tell that there are male and female figures in the samples. However, there are a lot of duplicates in the generated samples after training for 20000 iterations. The quality of the samples deteriorated between the 10000th and 20000th iteration.

4. Try turn on the gradient_penalty flag in the args_dict and train the model again. Are you able to stabilize the training? Briefly explain why the gradient penalty can help. You are welcome to check out the related literature above for gradient penalty. [1pt]

Figure 7: Samples after 1000 iterations

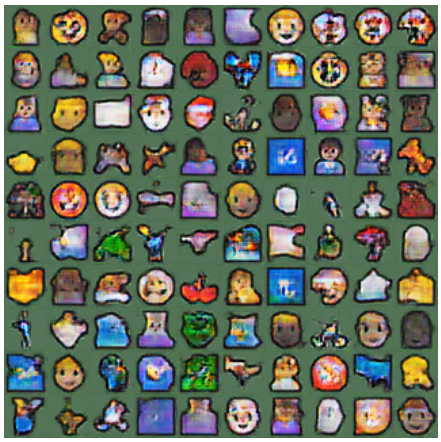

Figure 8: Samples after 10000 iterations

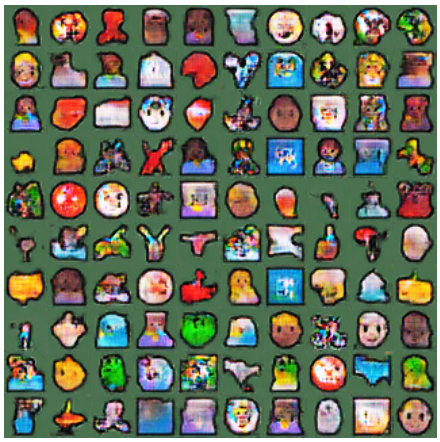

Figure 9: Samples after 20000 iterations

Training was able to be stabilized after turning on gradient penalty. As can be seen in the figures above, the generated samples improved consistently throughout 20000 iterations of training instead of resulting in many duplicates. In high dimensional space, minor changes in the generator can result in vastly different outputs. Therefore, the discriminator needs to be trained more to catch up. The gradient penalty penalizes the discriminator gradient so the Jacobian norm is bigger resulting in a larger update for each iteration.

Part 2: CycleGAN [3pt]

1. Train the CycleGAN to translate Apple emojis to Windows emojis in the Training - CycleGAN section of the notebook. Include in your writeup the samples from both generators at either iteration 200 and samples from a later iteration. [1pt]

The samples from both generators after training for 5000 iterations are shown below:

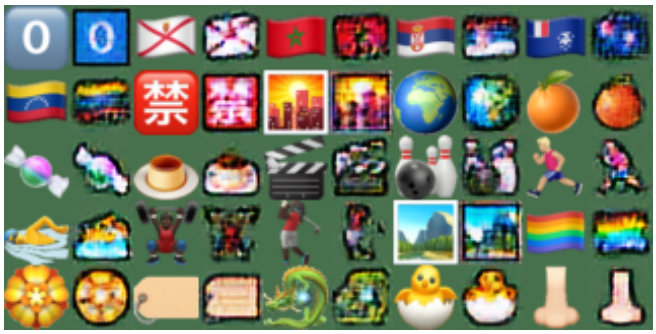

Figure 10: Samples after 5000 iterations for the X to Y generator

Figure 11: Samples after 5000 iterations for the Y to X generator

2. Change the random seed and train the CycleGAN again. What are the most noticeable difference between the *similar* quality samples from the different random seeds? Explain why there is such a difference?

With a different random seed, the samples from both generators after training for 5000 iterations are shown below:

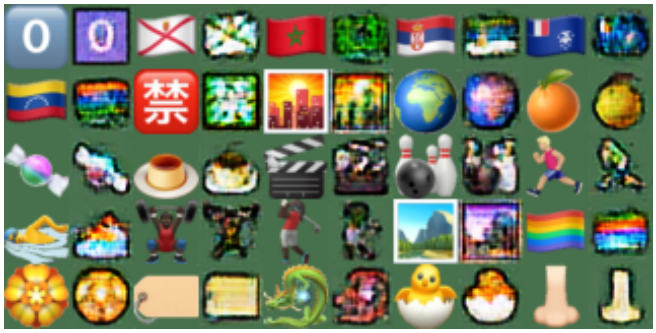

Figure 12: Samples after 5000 iterations for the X to Y generator

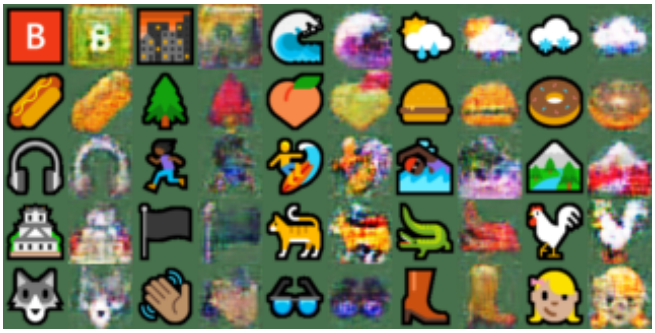

Figure 13: Samples after 5000 iterations for the Y to X generator

The most noticeable difference is that the color of the emojis are different from before. This is because in the X to Y to X cycle, the color mapping in the reconstruction from X domain in Y domain is different with different random seeds.

3. Changing the default lambda_cycle hyperparameters and train the CycleGAN again. Try a couple of different values including without the cycle-consistency loss. (i.e. lambda_cycle = 0) For different values of lambda_cycle, include in your writeup some samples from both generators at either iteration 200 and samples from a later iteration. Do you notice a difference between the results with and without the cycle consistency loss? Write down your observations (positive or negative) in your writeup. Can you explain these results, i.e., why there is or isn't a difference among the experiments?

With a lambda_cycle value set to 0, 0.05, and 0.10, the samples after training for 5000 iterations are as shown below:

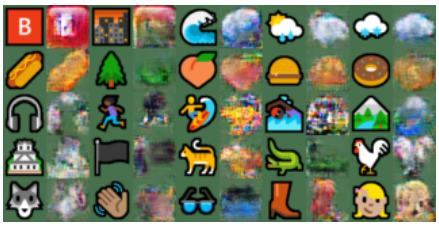

Figure 14: Samples with lambda_cycle = 0 after 1000 iterations

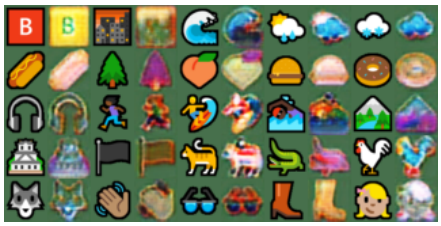

Figure 15: Samples with lambda_cycle = 0.05 after 1000 iterations

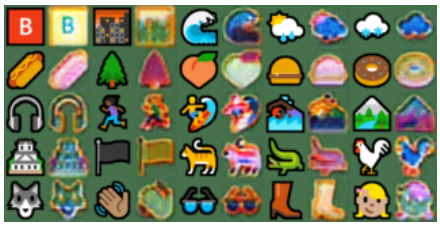

Figure 16: Samples with lambda_cycle = 0.10 after 1000 iterations

It is observed that the reconstructed image does not really look like the original image with a lambda_cycle value of 0. With a greater value, the reconstruction and the original looks more alike. The cycle-consistency loss is defined as follows:

$$\lambda_{cycle}\mathcal{J}_{cycle}^{(X\to Y\to X)} = \lambda_{cycle}\frac{1}{m}\sum_{i=1}^{m}\left\|y^{(i)} - G_{X\to Y}\left(G_{Y\to X}\left(y^{(i)}\right)\right)\right\|_1$$

The loss penalizes the L1 difference between the original image and the reconstructed image. For a small lambda_cycle value, the cycle consistency loss is small so the model does not learn to keep the reconstruction similar to the original image that it started with.

Part 3: BigGAN [2pt]

1. Based on T-SNE visualization of class embeddings, which two classes are good candidates for interpolation? Which two classes might not be a good match for interpolation? In each case, give 2 examples. Briefly explain your choice.

Good candidates for interpolation:
(tiger cat, Persian cat)
(Bedlington terrier, Irish terrier)

Bad candidates for interpolation:
(sunglasses, goldfish)
(stopwatch, great white shark)

Two classes whose class embeddings are close should be good candidates for interpolation. Two classes whose class embeddings are far away should be bad candidates for interpolation. Intuitively, cats of different breed can give birth to a new breed somewhere in between.

2. Complete generate_linear_interpolate_sample function. Verify the examples you gave in the previous question by generating samples. Include the four sets of generated images in your report.

```
##########################################
##   FILL THIS IN: CREATE NEW EMBEDDING  ##
##########################################
new_emb = alpha*class1_emb+(1-alpha)*class2_emb
```

Figure 17: Screenshot of interpolation implementation

Figure 18: Interpolation of tiger cat and Persian cat

Figure 19: Interpolation of Bedlington terrier and Irish terrier

Figure 20: Interpolation of sunglasses and goldfish

Figure 21: Interpolation of stopwatch and great white shark

Interpolation of tiger cat and Persian cat produced some sort of cats in between. Interpolation of Bedlington terrier and Irish terriers produced some sort of dogs in between. Interpolation of sunglasses and goldfish, and interpolation of stopwatch and shark produced nonsensical results.