CSC321: Assignment 3

Due on April 4, 2018

Wenxin Chen - 1002157676

April 5, 2018

1 Deep Convolutional GAN

1.1 Implement the Discriminator of the DCGAN

1.1.1 Padding

The output size, MxM, of a convolutional unit given an input of size NxN is expressed by

$$M = \frac{N - K}{S} + 1$$

Where K is the kernel size and S is the stride size. Given K = 4 and S = 2, we have $M = \frac{N}{2} - 1$. Thus, if we add a padding of 1 on all sides of the image the final size will be MxM where $M = \frac{N}{2}$ as desired.

1.1.2 Implementation

```python
class DCDiscriminator(nn.Module):
    """Defines the architecture of the discriminator network.
       Note: Both discriminators D_X and D_Y have the same architecture in this assignment.
    """
    def __init__(self, conv_dim=64):
        super(DCDiscriminator, self).__init__()

        self.conv1 = conv(3, 32, 4, stride=2, padding=1, batch_norm=True, init_zero_weights=False)
        self.conv2 = conv(32, 64, 4, stride=2, padding=1, batch_norm=True, init_zero_weights=False)
        self.conv3 = conv(64, 128, 4, stride=2, padding=1, batch_norm=True, init_zero_weights=False)
        self.conv4 = conv(128, 1, 4, stride=2, padding=0, batch_norm=False, init_zero_weights=False)

```

Listing 1 : Discriminator Implementation

1.2 Generator

1.2.1 Implementation

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

        self.deconv1 = deconv(100, 128, 4, stride=2, padding=0, batch_norm=True)
        self.deconv2 = deconv(128, 64, 4, stride=2, padding=1, batch_norm=True)
        self.deconv3 = deconv(64, 32, 4, stride=2, padding=1, batch_norm=True)
        self.deconv4 = deconv(32, 3, 4, stride=2, padding=1, batch_norm=False)

```

Listing 2 : Generator Implementation

1.3 Training Loop

1.3.1 Implementation

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

# 2. Sample noise
noise = sample_noise(opts.noise_size)

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

# 4. Compute the discriminator loss on the fake images
D_fake_loss = float(1)/2/batch_size*torch.sum((D.forward(fake_images)) ** 2)

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

D_total_loss.backward()
d_optimizer.step()

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

g_optimizer.zero_grad()

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

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

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

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

Listing 3 : Training Implementation

1.4 Experiment

The results of Vanilla GAN can be seen on the next page. Initially the samples do not resemble emojis at all and appear as edge-less blobs of colour. As the training progresses distinct shapes and edges are seen, but the end result still does not completely resemble an emoji. Furthermore we see that the improvements after 2000 or so iterations are not that significant, indicating that the accuracy of the model output may be plateauing.

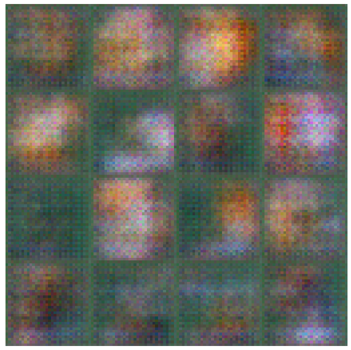

Figure 1: Vanilla GAN output after 200 Iterations

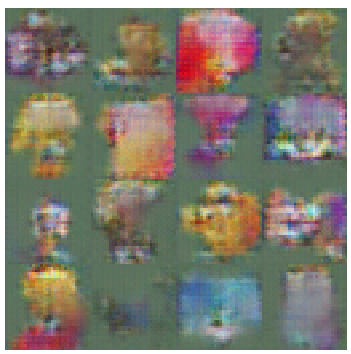

Figure 2: Vanilla GAN output after 2000 Iterations

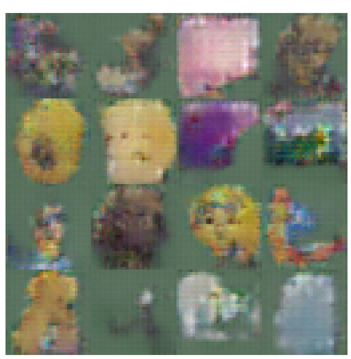

Figure 3: Vanilla GAN output after 5600 Iterations

2 CycleGAN

2.1 Generator

2.1.1 Implementation

```

class CycleGenerator(nn.Module):
"""Defines the architecture of the generator network.
   Note: Both generators G_XtoY and G_YtoX have the same architecture in this
    assignment.
"""
def __init__(self, conv_dim=64, init_zero_weights=False):
    super(CycleGenerator, self).__init__()

    # 1. Define the encoder part of the generator (that extracts features from the
    input image)
    self.conv1 = conv(3, 32, 4, stride=2, padding=1, batch_norm=True,
    init_zero_weights=False)
    self.conv2 = conv(32, 64, 4, stride=2, padding=1, batch_norm=True,
    init_zero_weights=False)

    # 2. Define the transformation part of the generator
    self.resnet_block = ResnetBlock(64)

    # 3. Define the decoder part of the generator (that builds up the output image
    from features)
    self.deconv1 = deconv(64, 32, 4, stride=2, padding=1, batch_norm=True)
    self.deconv2 = deconv(32, 3, 4, stride=2, padding=1, batch_norm=False)

```

Listing 4 : CycleGAN Generator Implementation

2.2 Experiments

2.2.1 CycleGAN

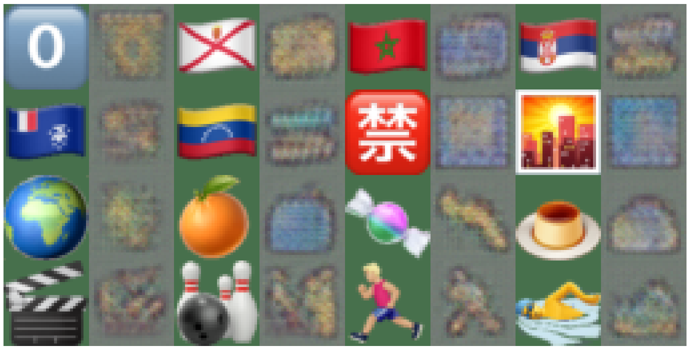

Figure 4: Cycle GAN output after 600 Iterations (X->Y)

Figure 5: Cycle GAN output after 600 Iterations (Y->X)

2.2.2 CycleGAN with Cycle Consistency

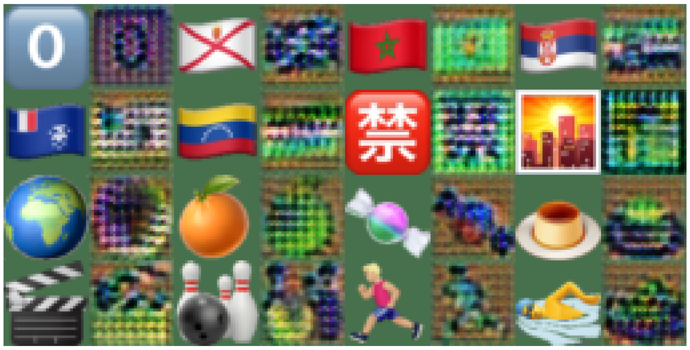

Figure 6: Cycle GAN with Cycle Consistency output after 600 Iterations (X->Y)

Figure 7: Cycle GAN with Cycle Consistency output after 600 Iterations (Y->X)

2.2.3 CycleGAN (Pretrained)

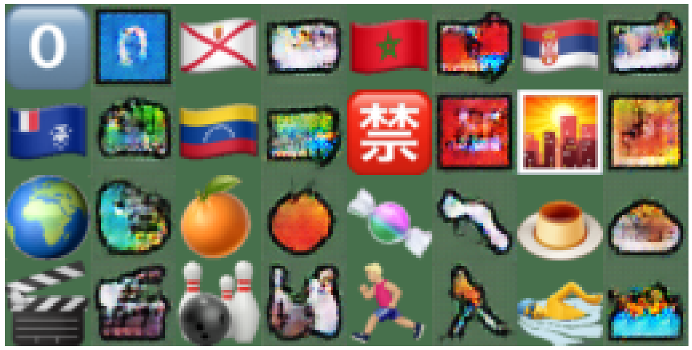

Figure 8: Pretrained Cycle GAN with Cycle Consistency output after 100 Iterations (X->Y)

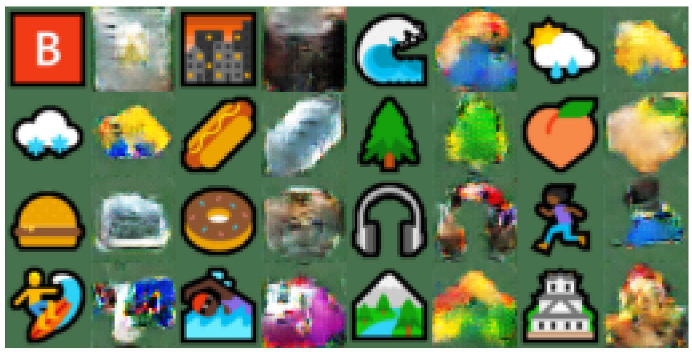

Figure 9: Pretrained Cycle GAN with Cycle Consistency output after 100 Iterations (Y->X)

2.2.4 CycleGAN with Cycle Consistency (Pretrained)

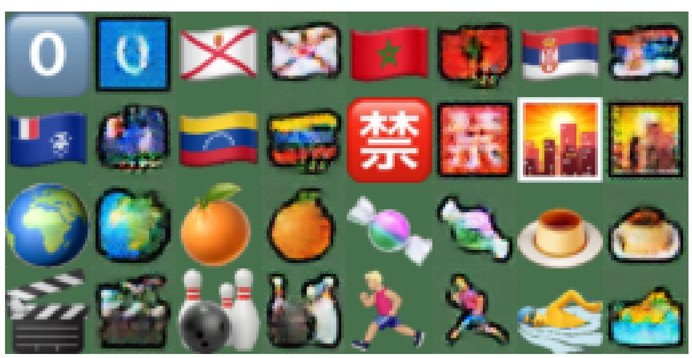

Figure 10: Pretrained Cycle GAN with Cycle Consistency output after 100 Iterations (X->Y)

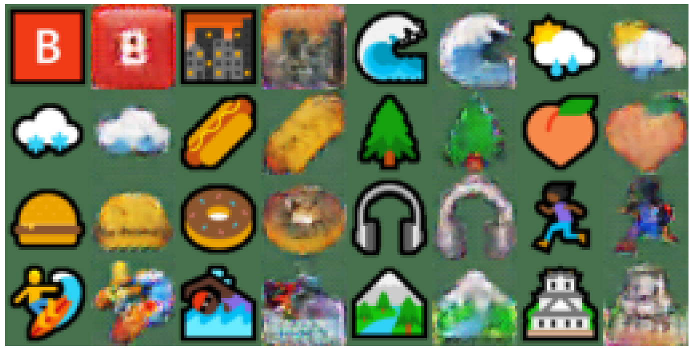

Figure 11: Pretrained Cycle GAN with Cycle Consistency output after 100 Iterations (Y->X)

2.2.5 Observations

The outputs of the models with cycle consistency appear to be much more accurate compared to the ones without. The effects of cycle consistency can be observed especially in the earlier stages of training, as seen in figures 6 and 7 compared to figures 4 and 5. Observe figure 4: the model has learned to add black edges around the input emoji , a feature of emoji set Y. However it fails to consistently output an emoji that resembles the input emoji in shape. Now observe figure 7, in which the black edges are not as visible but the output image is a better match to the input image.

With cycle consistency, the generators are forced to output images that are close enough to the original image to be converted back to the input through the other generator, while still being similar enough to the other image class that the discriminator is fooled. This balance is what creates the better results observed in the models with cycle consistency.