Part 1 DCGAN

Implementation

1. DCGAN Generator

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

        self.conv_dim = conv_dim
        ##########################################
        ##   FILL THIS IN: CREATE ARCHITECTURE   ##
        ##########################################

        self.linear_bn = upconv(in_channels=100, out_channels=self.conv_dim*4, kernel_size=3, stride=2, padding=2, spectral_norm=spectral_norm)
        self.upconv1 = upconv(in_channels=self.conv_dim*4, out_channels=self.conv_dim*2, kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.upconv2 = upconv(in_channels=self.conv_dim*2, out_channels=self.conv_dim, kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.upconv3 = upconv(in_channels=self.conv_dim, out_channels=3, kernel_size=5, stride=2, spectral_norm=spectral_norm)
```

2. DCGAN training loop

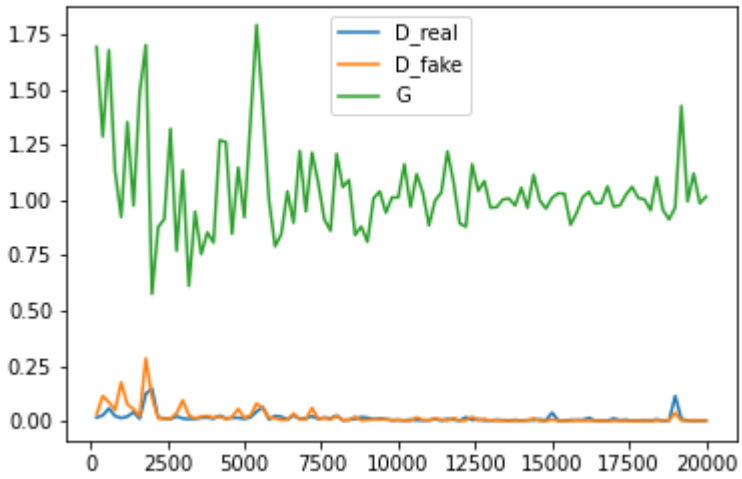
```
try:
    for iteration in range(1, opts.train_iters + 1):

        # Reset data_iter for each epoch
        if iteration % iter_per_epoch == 0:
            train_iter = iter(dataloader)

        real_images, real_labels = train_iter.next()
        real_images, real_labels = to_var(real_images), to_var(real_labels).long().squeeze()

        # ones = Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(1.0), requires_grad=False)

        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. 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)
```

```
    # ---------------------------
    # 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(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)
```

Experiment

1. The generator performance improves over time generally.

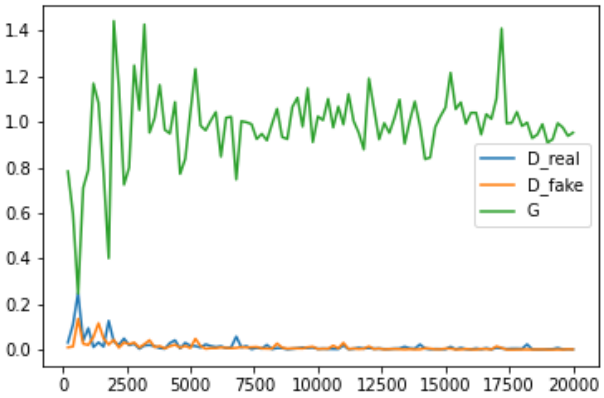
The above samples are from 200 (top left), 5000 (top middle), 10000 (top right), 12000 (bottom left), 15000 (bottom middle), 20000 (bottom right).



The initial results were terrible as expected since the network hasn't learned anything yet. Then once it reaches 10000 iterations, the emojis are starting to appear with distinct colors. However, at 15000 iterations, although the loss is slightly lower, the emojis are starting to look alike, which can be confirmed at 20000 iterations. I believe this is because the network is trying to create an emoji that would resembles to most of the emojis to obtain a lower overall loss. In addition, the G loss is very unstable along the iterations. To conclude, the result at 10000 iterations is arguably better than the result at 20000 iterations.

2. After turning on gradient penalty, here are the results I got:

The above samples are from 200 (top left), 5000 (top middle), 10000 (top right), 12000 (bottom left), 15000 (bottom middle), 20000 (bottom right).



Judging from just the results, I can't really see a huge difference from not using gradient penalty. However, if we look at the loss graph, the G loss is only slightly smoother than before and we can see some sort of curve reaching a convergence, so the training can be said to be slightly more stabilized. This is due to the gradient penalty since it penalizes large or exploding gradient values in order to keep the overall gradient less likely to change to much in a small number of iterations. However, I would say that the effect of gradient penalty is not that impactful overall in terms of stabilizing training.

Note: the above answer was using the wrong code with alpha * fake_images.data

Part 2 CycleGAN Experiments

1. The results of X -> Y Generator (left), Y -> X Generator (right) at iteration 200

2. The results of X -> Y Generator (left), Y -> X Generator (right) at iteration 200 using seed = 69

The most noticeable difference between the similar quality samples is that the part of emoji that starts to look alike are different. For example, the zero emoji (top left of X -> Y generator), the colors are different between the 2 seeds where one started from a lighter shade of blue and the other started from a darker shade of blue. The different seed would generate different weight initialization, which explains why our results at iteation 200 are different since the initialised weights were different.

3. Using the same seed = 69:
   Results with lambda_cycle = 0:

   Results with lambda_cycle = 0.01:

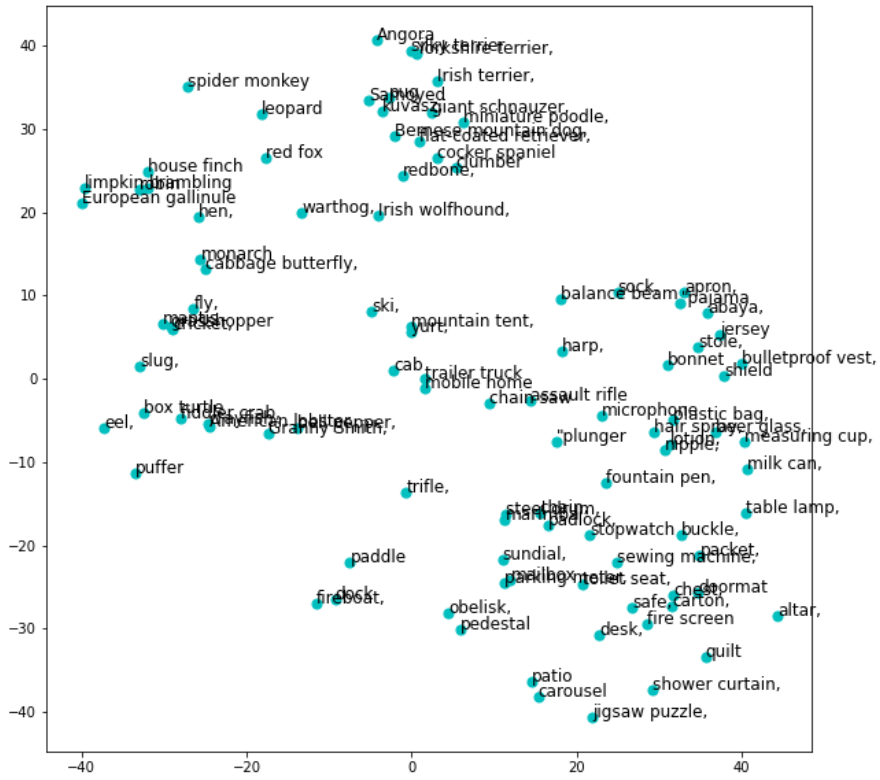
Results with lambda_cycle = 0.03:

Results with lambda_cycle = 0.06:

From the results without the cycle-consistency loss, we can see that the results were much worse than the ones using the default values (0.015). The results using 0.01 were very similar to the default ones as expected since their values are very close to each other. Moving to the results using 0.03, we can see that there are slightly more details to the emojis. For example, the black flag appears to be black already. However, from the results using 0.06, they look worse than the results using 0.03. For example, the black flag appears to be green!

A certain amunt of cycle-consitstency loss definitely helps with the training from the cases using lambda_cycle = 0.01, 0.06 since the concept is that the output of X -> Y generator using output of Y -> X should match with the original image, thus forming a cycle. Cycle-consistency loss acts as an additional regularizer by measuring the difference between the generated output of the X -> Y generator and the original input which will guide the learning much better. However, similar to other regularizer, too much regularization is not helpful to the training process, which happened in the last case using lambda_cycle = 0.06 where the results are over penalized.

Part 3 BigGAN Experiments



1. From the T-SNE visualization, I would say that trailer truck (867) and mobile home (660), pajama (697) and apron (411) would be good pairs of classes for interpolation.
   On the other hand, I would say that alter (406) and spider monkey (381), Irish terrier (184) and pedestal (708) would be bad pairs for interpolation.
   This is because the good pairs are very close to each other on the T-SNE mapping, whereas the bad pairs would be very far apart from each other. Therefore, the good pairs would have a smoother transition from one class to another, but the bad pairs would have to 'travel' a 'larger distance' and result in an unnatural transition from one class to another.
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

    images = G(z, new_emb)
  return images
```

Trailer truck (867) mobile home (660)

Pajama (697) apron (411)

Alter (406) and spider monkey (381)

Irish terrier (184) and pedestal (708)

As we can see, the good pairs have a much better transition. For example, the trailer truck and mobile home would have some similar structures on a road setting along the transition. On the other hand, the bad pairs either have sudden transition or transition that does not make sense. For example, the transition of Irish terrier and pedestal have photos that are very far from either of the classes, like the shoe photo at the top middle.