dlnd_face_generation

March 31, 2020

1 Face Generation

In this project, you'll define and train a DCGAN on a dataset of faces. Your goal is to get a generator network to generate *new* images of faces that look as realistic as possible!

The project will be broken down into a series of tasks from **loading in data to defining and training adversarial networks**. At the end of the notebook, you'll be able to visualize the results of your trained Generator to see how it performs; your generated samples should look like fairly realistic faces with small amounts of noise.

1.0.1 Get the Data

You'll be using the CelebFaces Attributes Dataset (CelebA) to train your adversarial networks.

This dataset is more complex than the number datasets (like MNIST or SVHN) you've been working with, and so, you should prepare to define deeper networks and train them for a longer time to get good results. It is suggested that you utilize a GPU for training.

1.0.2 Pre-processed Data

Since the project's main focus is on building the GANs, we've done *some* of the pre-processing for you. Each of the CelebA images has been cropped to remove parts of the image that don't include a face, then resized down to 64x64x3 NumPy images. Some sample data is show below.

If you are working locally, you can download this data by clicking here

This is a zip file that you'll need to extract in the home directory of this notebook for further loading and processing. After extracting the data, you should be left with a directory of data `processed_celeba_small/`

```
In [1]: # can comment out after executing
        !unzip processed_celeba_small.zip
```

```
Archive:  processed_celeba_small.zip
replace processed_celeba_small/.DS_Store? [y]es, [n]o, [A]ll, [N]one, [r]ename: ^C
```

```
In [40]: data_dir = 'processed_celeba_small/'

         """
```

```
DON'T MODIFY ANYTHING IN THIS CELL
"""
import pickle as pkl
import matplotlib.pyplot as plt
import numpy as np
import problem_unittests as tests
#import helper

%matplotlib inline
```

1.1 Visualize the CelebA Data

The CelebA dataset contains over 200,000 celebrity images with annotations. Since you're going to be generating faces, you won't need the annotations, you'll only need the images. Note that these are color images with 3 color channels (RGB) each.

1.1.1 Pre-process and Load the Data

Since the project's main focus is on building the GANs, we've done *some* of the pre-processing for you. Each of the CelebA images has been cropped to remove parts of the image that don't include a face, then resized down to 64x64x3 NumPy images. This *pre-processed* dataset is a smaller subset of the very large CelebA data.

> There are a few other steps that you'll need to **transform** this data and create a **DataLoader**.

Exercise: Complete the following `get_dataloader` **function, such that it satisfies these requirements:**

- Your images should be square, Tensor images of size `image_size x image_size` in the x and y dimension.
- Your function should return a DataLoader that shuffles and batches these Tensor images.

ImageFolder To create a dataset given a directory of images, it's recommended that you use PyTorch's ImageFolder wrapper, with a root directory `processed_celeba_small/` and data transformation passed in.

```
In [41]: # necessary imports
         import torch
         from torchvision import datasets
         from torchvision import transforms
```

```
In [42]: def get_dataloader(batch_size, image_size, data_dir='processed_celeba_small/'):
             """
             Batch the neural network data using DataLoader
             :param batch_size: The size of each batch; the number of images in a batch
             :param img_size: The square size of the image data (x, y)
             :param data_dir: Directory where image data is located
             :return: DataLoader with batched data
```

```
    """

    # TODO: Implement function and return a dataloader
    image_aug = transforms.Compose([transforms.Resize(image_size), transforms.CenterCro
    imagenet_data = datasets.ImageFolder(data_dir, transform=image_aug)
    data_loader = torch.utils.data.DataLoader(imagenet_data,
                                              batch_size=batch_size + 1,
                                              shuffle=True)

    return data_loader
```

1.2 Create a DataLoader

Exercise: Create a DataLoader `celeba_train_loader` with appropriate hyperparameters. Call the above function and create a dataloader to view images. * You can decide on any reasonable `batch_size` parameter * Your `image_size` **must be** `32`. Resizing the data to a smaller size will make for faster training, while still creating convincing images of faces!

```
In [43]: # Define function hyperparameters
         batch_size = 32
         img_size = 32

         """
         DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
         """
         # Call your function and get a dataloader
         celeba_train_loader = get_dataloader(batch_size, img_size)
```

Next, you can view some images! You should seen square images of somewhat-centered faces.

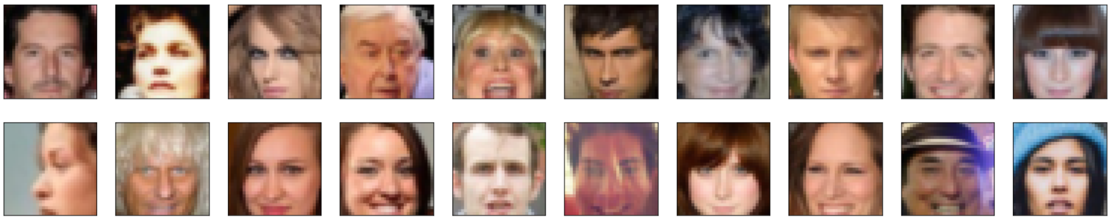
Note: You'll need to convert the Tensor images into a NumPy type and transpose the dimensions to correctly display an image, suggested `imshow` code is below, but it may not be perfect.

```
In [5]: # helper display function
        def imshow(img):
            npimg = img.numpy()
            plt.imshow(np.transpose(npimg, (1, 2, 0)))

        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        # obtain one batch of training images
        dataiter = iter(celeba_train_loader)
        images, _ = dataiter.next() # _ for no labels

        # plot the images in the batch, along with the corresponding labels
        fig = plt.figure(figsize=(20, 4))
        plot_size=20
        for idx in np.arange(plot_size):
```

```
    ax = fig.add_subplot(2, plot_size/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
```



Exercise: Pre-process your image data and scale it to a pixel range of -1 to 1 You need to do a bit of pre-processing; you know that the output of a tanh activated generator will contain pixel values in a range from -1 to 1, and so, we need to rescale our training images to a range of -1 to 1. (Right now, they are in a range from 0-1.)

In [6]:
```
# TODO: Complete the scale function
def scale(x, feature_range=(-1, 1)):
    ''' Scale takes in an image x and returns that image, scaled
       with a feature_range of pixel values from -1 to 1.
       This function assumes that the input x is already scaled from 0-1.'''
    # assume x is scaled to (0, 1)
    # scale to feature_range and return scaled x
    min, max = feature_range
    x = x * (max - min) + min
    return x
```

In [7]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
# check scaled range
# should be close to -1 to 1
img = images[0]
scaled_img = scale(img)

print('Min: ', scaled_img.min())
print('Max: ', scaled_img.max())
```

```
Min:  tensor(-1.)
Max:  tensor(0.7176)
```

---

2 Define the Model

A GAN is comprised of two adversarial networks, a discriminator and a generator.

2.1 Discriminator

Your first task will be to define the discriminator. This is a convolutional classifier like you've built before, only without any maxpooling layers. To deal with this complex data, it's suggested you use a deep network with **normalization**. You are also allowed to create any helper functions that may be useful.

Exercise: Complete the Discriminator class

- The inputs to the discriminator are 32x32x3 tensor images
- The output should be a single value that will indicate whether a given image is real or fake

```
In [8]: import torch.nn as nn
        import torch.nn.functional as F
```

```
In [9]: # helper conv function
        def conv(in_channels, out_channels, kernel_size, stride=2, padding=1, batch_norm=True):
            """Creates a convolutional layer, with optional batch normalization.
            """
            layers = []
            conv_layer = nn.Conv2d(in_channels, out_channels,
                                   kernel_size, stride, padding, bias=False)

            # append conv layer
            layers.append(conv_layer)

            if batch_norm:
                # append batchnorm layer
                layers.append(nn.BatchNorm2d(out_channels))

            # using Sequential container
            return nn.Sequential(*layers)
```

```
In [10]: class Discriminator(nn.Module):
             def __init__(self, conv_dim):
                 """
                 Initialize the Discriminator Module
                 :param conv_dim: The depth of the first convolutional layer
                 """
                 super(Discriminator, self).__init__()
                 self.conv_dim = conv_dim
                 # complete init function
                 self.conv1 = conv(3, conv_dim, 4, batch_norm=False) # x, y = 64 depth = 3
                 self.conv2 = conv(conv_dim, conv_dim * 2, 4) # x, y = 32 depth = 64
                 self.conv3 = conv(conv_dim * 2, conv_dim * 4, 4) # x, y = 16 depth = 128

                 self.fc = nn.Linear(conv_dim*4*4*4, 1)
                 self.out = nn.Sigmoid()
                 self.dropout = nn.Dropout(0.5)
```

```
    def forward(self, x):
        """
        Forward propagation of the neural network
        :param x: The input to the neural network
        :return: Discriminator logits; the output of the neural network
        """
        # define feedforward behavior
        x = F.leaky_relu(self.conv1(x), 0.2)
        # x = self.dropout(x)
        x = F.leaky_relu(self.conv2(x), 0.2)
        # x = self.dropout(x)
        x = F.leaky_relu(self.conv3(x), 0.2)
        # x = self.dropout(x)

        x = x.view(-1, self.conv_dim*4*4*4)

        x = self.fc(x)
        x = self.dropout(x)

        return x


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_discriminator(Discriminator)
```

```
Tests Passed
```

2.2 Generator

The generator should upsample an input and generate a *new* image of the same size as our training data `32x32x3`. This should be mostly transpose convolutional layers with normalization applied to the outputs.

Exercise: Complete the Generator class

- The inputs to the generator are vectors of some length `z_size`
- The output should be a image of shape `32x32x3`

```
In [11]: # helper deconv function
         def deconv(in_channels, out_channels, kernel_size, stride=2, padding=1, batch_norm=True
             """Creates a transpose convolutional layer, with optional batch normalization.
             """
             layers = []
             # append transpose conv layer
```

```
        layers.append(nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, pa
        # optional batch norm layer
        if batch_norm:
            layers.append(nn.BatchNorm2d(out_channels))
        return nn.Sequential(*layers)
```

In [12]:
```
class Generator(nn.Module):

    def __init__(self, z_size, conv_dim):
        """
        Initialize the Generator Module
        :param z_size: The length of the input latent vector, z
        :param conv_dim: The depth of the inputs to the *last* transpose convolutional
        """
        super(Generator, self).__init__()

        # complete init function
        self.conv_dim = conv_dim

        self.fc = nn.Linear(z_size, conv_dim*4*4*4)

        self.t_conv1 = deconv(conv_dim*4, conv_dim*2, 4 )
        self.t_conv2 = deconv(conv_dim*2, conv_dim, 4)
        self.t_conv3 = deconv(conv_dim, 3, 4, batch_norm=False)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        """
        Forward propagation of the neural network
        :param x: The input to the neural network
        :return: A 32x32x3 Tensor image as output
        """
        # define feedforward behavior
        x = self.fc(x)
        x = self.dropout(x)

        x = x.view(-1, self.conv_dim*4, 4, 4)

        x = F.relu(self.t_conv1(x))
        # x = self.dropout(x)
        x = F.relu(self.t_conv2(x))
        # x = self.dropout(x)
        x = F.tanh(self.t_conv3(x))
        return x

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
```

```
tests.test_generator(Generator)
```

```
Tests Passed
```

2.3 Initialize the weights of your networks

To help your models converge, you should initialize the weights of the convolutional and linear layers in your model. From reading the original DCGAN paper, they say: > All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02.

So, your next task will be to define a weight initialization function that does just this!

You can refer back to the lesson on weight initialization or even consult existing model code, such as that from the `networks.py` file in CycleGAN Github repository to help you complete this function.

Exercise: Complete the weight initialization function

- This should initialize only **convolutional** and **linear** layers
- Initialize the weights to a normal distribution, centered around 0, with a standard deviation of 0.02.
- The bias terms, if they exist, may be left alone or set to 0.

```
In [13]: from torch.nn import init
         def weights_init_normal(m):
             """
             Applies initial weights to certain layers in a model .
             The weights are taken from a normal distribution
             with mean = 0, std dev = 0.02.
             :param m: A module or layer in a network
             """
             init_gain=0.02
             # classname will be something like:
             # `Conv`, `BatchNorm2d`, `Linear`, etc.
             classname = m.__class__.__name__

             # TODO: Apply initial weights to convolutional and linear layers

             # classname will be something like:
             # `Conv`, `BatchNorm2d`, `Linear`, etc.
             classname = m.__class__.__name__
             # print(classname)
             if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear
                 init.normal_(m.weight.data, 0.0, init_gain)
                 if hasattr(m, 'bias') and m.bias is not None:
                     init.constant_(m.bias.data, 0.0)
             elif classname.find('BatchNorm2d') != -1:  # BatchNorm Layer's weight is not a matr
                 init.normal_(m.weight.data, 1.0, init_gain)
                 init.constant_(m.bias.data, 0.0)
```

2.4 Build complete network

Define your models' hyperparameters and instantiate the discriminator and generator from the classes defined above. Make sure you've passed in the correct input arguments.

In [44]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
def build_network(d_conv_dim, g_conv_dim, z_size):
    # define discriminator and generator
    D = Discriminator(d_conv_dim)
    G = Generator(z_size=z_size, conv_dim=g_conv_dim)

    # initialize model weights
    D.apply(weights_init_normal)
    G.apply(weights_init_normal)

    print(D)
    print()
    print(G)

    return D, G
```

Exercise: Define model hyperparameters

In [37]:
```
# Define model hyperparams
d_conv_dim = 64
g_conv_dim = 64
z_size = 100

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
D, G = build_network(d_conv_dim, g_conv_dim, z_size)
```

```
Discriminator(
  (conv1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
  (conv2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (conv3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (fc): Linear(in_features=4096, out_features=1, bias=True)
  (out): Sigmoid()
```

```
  (dropout): Dropout(p=0.5)
)

Generator(
  (fc): Linear(in_features=100, out_features=4096, bias=True)
  (t_conv1): Sequential(
    (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (t_conv2): Sequential(
    (0): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (t_conv3): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
  (dropout): Dropout(p=0.5)
)
```

2.4.1 Training on GPU

Check if you can train on GPU. Here, we'll set this as a boolean variable `train_on_gpu`. Later, you'll be responsible for making sure that >* Models, * Model inputs, and * Loss function arguments

Are moved to GPU, where appropriate.

```
In [16]: """
         DON'T MODIFY ANYTHING IN THIS CELL
         """
         import torch

         # Check for a GPU
         train_on_gpu = torch.cuda.is_available()
         if not train_on_gpu:
             print('No GPU found. Please use a GPU to train your neural network.')
         else:
             print('Training on GPU!')
```

```
Training on GPU!
```

2.5 Discriminator and Generator Losses

Now we need to calculate the losses for both types of adversarial networks.

2.5.1 Discriminator Losses

- For the discriminator, the total loss is the sum of the losses for real and fake images, `d_loss = d_real_loss + d_fake_loss`.
- Remember that we want the discriminator to output 1 for real images and 0 for fake images, so we need to set up the losses to reflect that.

2.5.2 Generator Loss

The generator loss will look similar only with flipped labels. The generator's goal is to get the discriminator to *think* its generated images are *real*.

Exercise: Complete real and fake loss functions **You may choose to use either cross entropy or a least squares error loss to complete the following `real_loss` and `fake_loss` functions.**

In [38]:
```
def real_loss(D_out):
    '''Calculates how close discriminator outputs are to being real.
       param, D_out: discriminator logits
       return: real loss'''

    batch_size = D_out.size(0)
    # label smoothing
    labels = torch.ones(batch_size) * 0.9
    if train_on_gpu:
        labels = labels.cuda()
    criterion = nn.BCEWithLogitsLoss()
    loss = criterion(D_out.squeeze(), labels)
    return loss

def fake_loss(D_out):
    '''Calculates how close discriminator outputs are to being fake.
       param, D_out: discriminator logits
       return: fake loss'''
    batch_size = D_out.size(0)
    labels = torch.zeros(batch_size) # fake labels = 0
    if train_on_gpu:
        labels = labels.cuda()
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)
    return loss
```

2.6 Optimizers

Exercise: Define optimizers for your Discriminator (D) and Generator (G) Define optimizers for your models with appropriate hyperparameters.

In [48]:
```
import torch.optim as optim
lr = 0.0002
```

```
beta1= 0.5
beta2= 0.99

# Create optimizers for the discriminator D and generator G
d_optimizer = optim.Adam(D.parameters(), lr, [beta1, beta2])
g_optimizer = optim.Adam(G.parameters(), lr, [beta1, beta2])
```

2.7 Training

Training will involve alternating between training the discriminator and the generator. You'll use your functions `real_loss` and `fake_loss` to help you calculate the discriminator losses.

- You should train the discriminator by alternating on real and fake images
- Then the generator, which tries to trick the discriminator and should have an opposing loss function

Saving Samples You've been given some code to print out some loss statistics and save some generated "fake" samples.

Exercise: Complete the training function Keep in mind that, if you've moved your models to GPU, you'll also have to move any model inputs to GPU.

```
In [46]: def train(D, G, n_epochs, print_every=50):
             '''Trains adversarial networks for some number of epochs
                param, D: the discriminator network
                param, G: the generator network
                param, n_epochs: number of epochs to train for
                param, print_every: when to print and record the models' losses
                return: D and G losses'''

             # move models to GPU
             if train_on_gpu:
                 D.cuda()
                 G.cuda()

             # D.train()
             # keep track of loss and generated, "fake" samples
             samples = []
             losses = []

             # Get some fixed data for sampling. These are images that are held
             # constant throughout training, and allow us to inspect the model's performance
             sample_size=16
             fixed_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
             fixed_z = torch.from_numpy(fixed_z).float()
             # move z to GPU if available
```

```python
if train_on_gpu:
    fixed_z = fixed_z.cuda()

# epoch training loop
for epoch in range(n_epochs):

    # batch training loop
    for batch_i, (real_images, _) in enumerate(celeba_train_loader):

        batch_size = real_images.size(0)
        real_images = scale(real_images)

        # ===============================================
        #         YOUR CODE HERE: TRAIN THE NETWORKS
        # ===============================================

        # 1. Train the discriminator on real and fake images

        d_optimizer.zero_grad()

        if train_on_gpu:
            real_images = real_images.cuda()

        d_real = D(real_images)
        d_real_loss = real_loss(d_real)

        # Generate fake images
        z = np.random.uniform(-1, 1, size=(batch_size, z_size))
        z = torch.from_numpy(z).float()

        if train_on_gpu:
            z = z.cuda()

        fake_images = G(z)

        d_fake = D(fake_images)
        d_fake_loss = fake_loss(d_fake)

        # add up loss and perform backprop
        d_loss = d_real_loss + d_fake_loss

        d_loss.backward()
        d_optimizer.step()

        # 2. Train the generator with an adversarial loss
        g_optimizer.zero_grad()

        # Generate fake images
```

```
            z = np.random.uniform(-1, 1, size=(batch_size, z_size))
            z = torch.from_numpy(z).float()

            if train_on_gpu:
                z = z.cuda()

            fake_images = G(z)
            # D.eval()
            # Compute the discriminator losses on fake images
            # using flipped labels!
            g_fake = D(fake_images)
            g_loss = real_loss(g_fake) # use real loss to flip labels
            # g_loss = torch.mean(g_fake_out**2)
            # D.train()
            # perform backprop
            g_loss.backward()
            g_optimizer.step()

            # ===============================================
            #             END OF YOUR CODE
            # ===============================================

            # Print some loss stats
            if batch_i % print_every == 0:
                # append discriminator loss and generator loss
                losses.append((d_loss.item(), g_loss.item()))
                # print discriminator and generator loss
                print('Epoch [{:5d}/{:5d}] | d_loss: {:6.4f} | g_loss: {:6.4f}'.format(
                        epoch+1, n_epochs, d_loss.item(), g_loss.item()))


        ## AFTER EACH EPOCH##
        # this code assumes your generator is named G, feel free to change the name
        # generate and save sample, fake images
        G.eval() # for generating samples
        samples_z = G(fixed_z)
        samples.append(samples_z)
        G.train() # back to training mode
        print('---------------Epoch [{:5d}/{:5d}]---------------'.format(epoch+1, n_epo
    # Save training generator samples
    with open('train_samples.pkl', 'wb') as f:
        pkl.dump(samples, f)

    # finally return losses
    return losses
```

Set your number of training epochs and train your GAN!

```
In [49]: # set number of epochs
```

```python
n_epochs = 5


"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# call training function
losses = train(D, G, n_epochs=n_epochs)
```

```
Epoch [    1/    5] | d_loss: 1.3303 | g_loss: 1.8101
Epoch [    1/    5] | d_loss: 0.9189 | g_loss: 2.4704
Epoch [    1/    5] | d_loss: 0.9958 | g_loss: 2.9792
Epoch [    1/    5] | d_loss: 0.9461 | g_loss: 2.6381
Epoch [    1/    5] | d_loss: 1.0764 | g_loss: 2.0608
Epoch [    1/    5] | d_loss: 1.0608 | g_loss: 1.3316
Epoch [    1/    5] | d_loss: 0.9782 | g_loss: 1.9480
Epoch [    1/    5] | d_loss: 1.1256 | g_loss: 2.1165
Epoch [    1/    5] | d_loss: 1.1463 | g_loss: 0.9794
Epoch [    1/    5] | d_loss: 0.9937 | g_loss: 2.7013
Epoch [    1/    5] | d_loss: 1.1576 | g_loss: 1.8628
Epoch [    1/    5] | d_loss: 0.9833 | g_loss: 1.4450
Epoch [    1/    5] | d_loss: 1.1488 | g_loss: 1.0338
Epoch [    1/    5] | d_loss: 1.0447 | g_loss: 2.1604
Epoch [    1/    5] | d_loss: 1.2242 | g_loss: 1.6244
Epoch [    1/    5] | d_loss: 1.1264 | g_loss: 1.6144
Epoch [    1/    5] | d_loss: 1.0533 | g_loss: 1.9974
Epoch [    1/    5] | d_loss: 1.1206 | g_loss: 1.4049
Epoch [    1/    5] | d_loss: 1.2711 | g_loss: 1.6288
Epoch [    1/    5] | d_loss: 1.0261 | g_loss: 1.5810
Epoch [    1/    5] | d_loss: 1.2386 | g_loss: 1.3667
Epoch [    1/    5] | d_loss: 1.4252 | g_loss: 2.0231
Epoch [    1/    5] | d_loss: 1.3349 | g_loss: 1.4028
Epoch [    1/    5] | d_loss: 1.1925 | g_loss: 1.4533
Epoch [    1/    5] | d_loss: 1.0897 | g_loss: 1.7333
Epoch [    1/    5] | d_loss: 1.0480 | g_loss: 1.5573
Epoch [    1/    5] | d_loss: 1.1793 | g_loss: 1.8161
Epoch [    1/    5] | d_loss: 1.0713 | g_loss: 1.2107
Epoch [    1/    5] | d_loss: 1.1129 | g_loss: 1.9358
Epoch [    1/    5] | d_loss: 1.0904 | g_loss: 1.1871
Epoch [    1/    5] | d_loss: 1.1940 | g_loss: 1.4800
Epoch [    1/    5] | d_loss: 1.2391 | g_loss: 1.3657
Epoch [    1/    5] | d_loss: 1.1433 | g_loss: 1.5695
Epoch [    1/    5] | d_loss: 1.1558 | g_loss: 1.3800
Epoch [    1/    5] | d_loss: 1.3519 | g_loss: 1.4693
Epoch [    1/    5] | d_loss: 1.0961 | g_loss: 0.8930
Epoch [    1/    5] | d_loss: 1.1069 | g_loss: 1.0004
Epoch [    1/    5] | d_loss: 1.0716 | g_loss: 0.9899
Epoch [    1/    5] | d_loss: 1.1847 | g_loss: 0.9073
```

```
Epoch [    1/    5] | d_loss: 1.2216 | g_loss: 0.9398
Epoch [    1/    5] | d_loss: 1.1167 | g_loss: 1.1747
Epoch [    1/    5] | d_loss: 1.4117 | g_loss: 0.9760
Epoch [    1/    5] | d_loss: 1.3926 | g_loss: 1.3685
Epoch [    1/    5] | d_loss: 1.3819 | g_loss: 0.8303
Epoch [    1/    5] | d_loss: 1.2440 | g_loss: 1.0139
Epoch [    1/    5] | d_loss: 1.2711 | g_loss: 1.3819
Epoch [    1/    5] | d_loss: 1.3498 | g_loss: 1.3272
Epoch [    1/    5] | d_loss: 1.0687 | g_loss: 1.8989
Epoch [    1/    5] | d_loss: 1.2891 | g_loss: 1.5795
Epoch [    1/    5] | d_loss: 1.2074 | g_loss: 1.7629
Epoch [    1/    5] | d_loss: 1.1507 | g_loss: 0.9320
Epoch [    1/    5] | d_loss: 1.1847 | g_loss: 1.2096
Epoch [    1/    5] | d_loss: 1.0529 | g_loss: 1.5578
Epoch [    1/    5] | d_loss: 1.1819 | g_loss: 1.3015
Epoch [    1/    5] | d_loss: 1.1715 | g_loss: 1.4931
---------------Epoch [    1/    5]---------------
Epoch [    2/    5] | d_loss: 1.1627 | g_loss: 1.6272
Epoch [    2/    5] | d_loss: 1.1574 | g_loss: 1.1744
Epoch [    2/    5] | d_loss: 0.9652 | g_loss: 1.9074
Epoch [    2/    5] | d_loss: 1.1130 | g_loss: 0.9996
Epoch [    2/    5] | d_loss: 1.2565 | g_loss: 2.2107
Epoch [    2/    5] | d_loss: 1.0739 | g_loss: 1.2958
Epoch [    2/    5] | d_loss: 1.1988 | g_loss: 1.3033
Epoch [    2/    5] | d_loss: 1.1181 | g_loss: 1.1162
Epoch [    2/    5] | d_loss: 1.0106 | g_loss: 1.2677
Epoch [    2/    5] | d_loss: 1.1012 | g_loss: 1.4630
Epoch [    2/    5] | d_loss: 1.3881 | g_loss: 1.1198
Epoch [    2/    5] | d_loss: 1.3192 | g_loss: 1.0885
Epoch [    2/    5] | d_loss: 1.0824 | g_loss: 1.4317
Epoch [    2/    5] | d_loss: 1.0844 | g_loss: 1.1226
Epoch [    2/    5] | d_loss: 1.0748 | g_loss: 1.1634
Epoch [    2/    5] | d_loss: 1.0967 | g_loss: 1.6271
Epoch [    2/    5] | d_loss: 1.0849 | g_loss: 1.6522
Epoch [    2/    5] | d_loss: 1.0246 | g_loss: 1.7299
Epoch [    2/    5] | d_loss: 1.0046 | g_loss: 1.4066
Epoch [    2/    5] | d_loss: 1.0270 | g_loss: 1.1948
Epoch [    2/    5] | d_loss: 1.0424 | g_loss: 1.7010
Epoch [    2/    5] | d_loss: 1.0945 | g_loss: 1.0000
Epoch [    2/    5] | d_loss: 1.1017 | g_loss: 1.6701
Epoch [    2/    5] | d_loss: 1.1835 | g_loss: 1.5545
Epoch [    2/    5] | d_loss: 1.0190 | g_loss: 1.7677
Epoch [    2/    5] | d_loss: 1.1803 | g_loss: 1.1579
Epoch [    2/    5] | d_loss: 1.1818 | g_loss: 1.1371
Epoch [    2/    5] | d_loss: 1.1402 | g_loss: 1.4203
Epoch [    2/    5] | d_loss: 1.3288 | g_loss: 0.7296
Epoch [    2/    5] | d_loss: 1.1618 | g_loss: 1.0568
Epoch [    2/    5] | d_loss: 1.1307 | g_loss: 0.9331
```

```
Epoch [    2/    5] | d_loss: 1.2627 | g_loss: 1.4149
Epoch [    2/    5] | d_loss: 1.0511 | g_loss: 1.0326
Epoch [    2/    5] | d_loss: 1.0870 | g_loss: 0.7921
Epoch [    2/    5] | d_loss: 1.1752 | g_loss: 1.0170
Epoch [    2/    5] | d_loss: 0.9872 | g_loss: 2.2428
Epoch [    2/    5] | d_loss: 0.9704 | g_loss: 1.5596
Epoch [    2/    5] | d_loss: 1.1806 | g_loss: 1.2286
Epoch [    2/    5] | d_loss: 1.1301 | g_loss: 1.1052
Epoch [    2/    5] | d_loss: 1.2500 | g_loss: 0.9430
Epoch [    2/    5] | d_loss: 1.1189 | g_loss: 1.3304
Epoch [    2/    5] | d_loss: 1.0920 | g_loss: 1.4507
Epoch [    2/    5] | d_loss: 1.2245 | g_loss: 1.3676
Epoch [    2/    5] | d_loss: 1.2325 | g_loss: 1.1162
Epoch [    2/    5] | d_loss: 1.0051 | g_loss: 1.3649
Epoch [    2/    5] | d_loss: 1.0410 | g_loss: 1.1180
Epoch [    2/    5] | d_loss: 1.0804 | g_loss: 1.0517
Epoch [    2/    5] | d_loss: 1.1395 | g_loss: 0.8095
Epoch [    2/    5] | d_loss: 1.0239 | g_loss: 1.5515
Epoch [    2/    5] | d_loss: 1.0218 | g_loss: 1.3618
Epoch [    2/    5] | d_loss: 1.0925 | g_loss: 1.3726
Epoch [    2/    5] | d_loss: 1.0264 | g_loss: 1.5255
Epoch [    2/    5] | d_loss: 1.1533 | g_loss: 1.3677
Epoch [    2/    5] | d_loss: 1.1058 | g_loss: 1.1827
Epoch [    2/    5] | d_loss: 1.0797 | g_loss: 1.5106
---------------Epoch [    2/    5]---------------
Epoch [    3/    5] | d_loss: 1.1000 | g_loss: 1.3319
Epoch [    3/    5] | d_loss: 1.0873 | g_loss: 1.1500
Epoch [    3/    5] | d_loss: 1.1452 | g_loss: 1.7198
Epoch [    3/    5] | d_loss: 1.0845 | g_loss: 1.0732
Epoch [    3/    5] | d_loss: 1.0860 | g_loss: 1.0945
Epoch [    3/    5] | d_loss: 1.0938 | g_loss: 1.6446
Epoch [    3/    5] | d_loss: 1.2805 | g_loss: 1.0559
Epoch [    3/    5] | d_loss: 1.1542 | g_loss: 2.0289
Epoch [    3/    5] | d_loss: 1.1909 | g_loss: 0.9805
Epoch [    3/    5] | d_loss: 1.1098 | g_loss: 0.9959
Epoch [    3/    5] | d_loss: 1.0269 | g_loss: 0.9256
Epoch [    3/    5] | d_loss: 1.0814 | g_loss: 1.2544
Epoch [    3/    5] | d_loss: 0.9906 | g_loss: 1.0117
Epoch [    3/    5] | d_loss: 1.0729 | g_loss: 1.8364
Epoch [    3/    5] | d_loss: 1.2751 | g_loss: 1.2982
Epoch [    3/    5] | d_loss: 0.9977 | g_loss: 1.3714
Epoch [    3/    5] | d_loss: 1.0661 | g_loss: 1.5213
Epoch [    3/    5] | d_loss: 1.0772 | g_loss: 1.0108
Epoch [    3/    5] | d_loss: 1.0085 | g_loss: 1.5079
Epoch [    3/    5] | d_loss: 1.1231 | g_loss: 1.4835
Epoch [    3/    5] | d_loss: 1.4694 | g_loss: 1.7580
Epoch [    3/    5] | d_loss: 1.1405 | g_loss: 1.0256
Epoch [    3/    5] | d_loss: 1.2566 | g_loss: 1.6881
```

```
Epoch [    3/    5] | d_loss: 1.1109 | g_loss: 1.4400
Epoch [    3/    5] | d_loss: 1.0676 | g_loss: 1.0189
Epoch [    3/    5] | d_loss: 0.9563 | g_loss: 1.7508
Epoch [    3/    5] | d_loss: 1.3666 | g_loss: 1.5719
Epoch [    3/    5] | d_loss: 1.1240 | g_loss: 0.9188
Epoch [    3/    5] | d_loss: 1.2709 | g_loss: 2.5072
Epoch [    3/    5] | d_loss: 1.1778 | g_loss: 1.3333
Epoch [    3/    5] | d_loss: 0.9441 | g_loss: 1.3916
Epoch [    3/    5] | d_loss: 1.1445 | g_loss: 1.2752
Epoch [    3/    5] | d_loss: 1.4651 | g_loss: 1.0675
Epoch [    3/    5] | d_loss: 1.2772 | g_loss: 0.9323
Epoch [    3/    5] | d_loss: 1.1410 | g_loss: 1.1289
Epoch [    3/    5] | d_loss: 1.1266 | g_loss: 1.1640
Epoch [    3/    5] | d_loss: 0.9976 | g_loss: 0.8858
Epoch [    3/    5] | d_loss: 1.0037 | g_loss: 2.0342
Epoch [    3/    5] | d_loss: 0.8988 | g_loss: 2.1061
Epoch [    3/    5] | d_loss: 1.5124 | g_loss: 1.7126
Epoch [    3/    5] | d_loss: 1.0760 | g_loss: 1.4591
Epoch [    3/    5] | d_loss: 1.1751 | g_loss: 1.0302
Epoch [    3/    5] | d_loss: 1.1298 | g_loss: 0.9334
Epoch [    3/    5] | d_loss: 1.0711 | g_loss: 1.5438
Epoch [    3/    5] | d_loss: 0.9823 | g_loss: 1.4549
Epoch [    3/    5] | d_loss: 1.2090 | g_loss: 1.4320
Epoch [    3/    5] | d_loss: 1.0282 | g_loss: 1.3394
Epoch [    3/    5] | d_loss: 1.2050 | g_loss: 1.4569
Epoch [    3/    5] | d_loss: 1.2131 | g_loss: 2.0083
Epoch [    3/    5] | d_loss: 0.9533 | g_loss: 1.1056
Epoch [    3/    5] | d_loss: 1.0064 | g_loss: 1.4138
Epoch [    3/    5] | d_loss: 1.0988 | g_loss: 1.5264
Epoch [    3/    5] | d_loss: 1.0082 | g_loss: 1.5590
Epoch [    3/    5] | d_loss: 1.1167 | g_loss: 0.9503
Epoch [    3/    5] | d_loss: 1.3031 | g_loss: 0.8754
---------------Epoch [    3/    5]---------------
Epoch [    4/    5] | d_loss: 1.1631 | g_loss: 1.0342
Epoch [    4/    5] | d_loss: 1.1896 | g_loss: 1.5345
Epoch [    4/    5] | d_loss: 1.1322 | g_loss: 1.1642
Epoch [    4/    5] | d_loss: 1.1228 | g_loss: 1.2765
Epoch [    4/    5] | d_loss: 1.0777 | g_loss: 1.0005
Epoch [    4/    5] | d_loss: 1.0334 | g_loss: 1.3469
Epoch [    4/    5] | d_loss: 0.9633 | g_loss: 1.3444
Epoch [    4/    5] | d_loss: 1.3342 | g_loss: 3.0526
Epoch [    4/    5] | d_loss: 1.1330 | g_loss: 1.2455
Epoch [    4/    5] | d_loss: 1.0691 | g_loss: 1.4331
Epoch [    4/    5] | d_loss: 1.1526 | g_loss: 1.0826
Epoch [    4/    5] | d_loss: 1.1560 | g_loss: 1.2532
Epoch [    4/    5] | d_loss: 1.0685 | g_loss: 1.2925
Epoch [    4/    5] | d_loss: 1.3204 | g_loss: 0.9528
Epoch [    4/    5] | d_loss: 1.1147 | g_loss: 1.3582
```

```
Epoch [    4/    5] | d_loss: 1.1402 | g_loss: 1.3257
Epoch [    4/    5] | d_loss: 1.2576 | g_loss: 1.1800
Epoch [    4/    5] | d_loss: 1.2081 | g_loss: 1.1648
Epoch [    4/    5] | d_loss: 1.1446 | g_loss: 1.1924
Epoch [    4/    5] | d_loss: 1.0679 | g_loss: 1.1193
Epoch [    4/    5] | d_loss: 1.6515 | g_loss: 0.6358
Epoch [    4/    5] | d_loss: 1.2448 | g_loss: 1.4196
Epoch [    4/    5] | d_loss: 1.0860 | g_loss: 1.1965
Epoch [    4/    5] | d_loss: 1.1412 | g_loss: 1.4594
Epoch [    4/    5] | d_loss: 1.1428 | g_loss: 1.8324
Epoch [    4/    5] | d_loss: 1.5442 | g_loss: 1.1552
Epoch [    4/    5] | d_loss: 1.0780 | g_loss: 1.3112
Epoch [    4/    5] | d_loss: 1.0376 | g_loss: 1.4759
Epoch [    4/    5] | d_loss: 1.1290 | g_loss: 1.2284
Epoch [    4/    5] | d_loss: 0.9387 | g_loss: 1.2377
Epoch [    4/    5] | d_loss: 1.1796 | g_loss: 1.0022
Epoch [    4/    5] | d_loss: 1.0793 | g_loss: 1.1137
Epoch [    4/    5] | d_loss: 1.1039 | g_loss: 0.9580
Epoch [    4/    5] | d_loss: 1.1311 | g_loss: 1.2953
Epoch [    4/    5] | d_loss: 1.1277 | g_loss: 1.2415
Epoch [    4/    5] | d_loss: 1.1065 | g_loss: 1.3449
Epoch [    4/    5] | d_loss: 1.0006 | g_loss: 1.3867
Epoch [    4/    5] | d_loss: 1.1558 | g_loss: 1.2903
Epoch [    4/    5] | d_loss: 1.2643 | g_loss: 1.0820
Epoch [    4/    5] | d_loss: 1.3165 | g_loss: 1.1980
Epoch [    4/    5] | d_loss: 1.0459 | g_loss: 0.9204
Epoch [    4/    5] | d_loss: 1.0697 | g_loss: 0.9093
Epoch [    4/    5] | d_loss: 1.0334 | g_loss: 1.0981
Epoch [    4/    5] | d_loss: 1.0242 | g_loss: 1.3021
Epoch [    4/    5] | d_loss: 1.0894 | g_loss: 1.8243
Epoch [    4/    5] | d_loss: 1.0142 | g_loss: 1.2327
Epoch [    4/    5] | d_loss: 0.9746 | g_loss: 1.9779
Epoch [    4/    5] | d_loss: 1.0978 | g_loss: 1.3948
Epoch [    4/    5] | d_loss: 1.1170 | g_loss: 1.5695
Epoch [    4/    5] | d_loss: 1.0449 | g_loss: 1.4499
Epoch [    4/    5] | d_loss: 1.0132 | g_loss: 1.6129
Epoch [    4/    5] | d_loss: 1.0961 | g_loss: 1.3747
Epoch [    4/    5] | d_loss: 1.1106 | g_loss: 1.2598
Epoch [    4/    5] | d_loss: 1.0414 | g_loss: 1.1297
Epoch [    4/    5] | d_loss: 1.0223 | g_loss: 0.9315
---------------Epoch [    4/    5]---------------
Epoch [    5/    5] | d_loss: 1.5800 | g_loss: 1.0662
Epoch [    5/    5] | d_loss: 1.0986 | g_loss: 1.5570
Epoch [    5/    5] | d_loss: 1.3748 | g_loss: 1.4584
Epoch [    5/    5] | d_loss: 1.0725 | g_loss: 1.3480
Epoch [    5/    5] | d_loss: 1.3001 | g_loss: 1.0941
Epoch [    5/    5] | d_loss: 1.6287 | g_loss: 0.9990
Epoch [    5/    5] | d_loss: 1.0766 | g_loss: 1.0211
```

```
Epoch [    5/    5] | d_loss: 1.0724 | g_loss: 1.4202
Epoch [    5/    5] | d_loss: 1.2767 | g_loss: 1.2965
Epoch [    5/    5] | d_loss: 1.0913 | g_loss: 1.7250
Epoch [    5/    5] | d_loss: 1.3201 | g_loss: 1.3872
Epoch [    5/    5] | d_loss: 1.1233 | g_loss: 1.1420
Epoch [    5/    5] | d_loss: 1.0417 | g_loss: 1.4755
Epoch [    5/    5] | d_loss: 1.1008 | g_loss: 1.3268
Epoch [    5/    5] | d_loss: 1.0311 | g_loss: 1.7044
Epoch [    5/    5] | d_loss: 1.1262 | g_loss: 0.9980
Epoch [    5/    5] | d_loss: 1.0637 | g_loss: 1.2031
Epoch [    5/    5] | d_loss: 1.2018 | g_loss: 1.2482
Epoch [    5/    5] | d_loss: 1.0289 | g_loss: 1.4648
Epoch [    5/    5] | d_loss: 1.4631 | g_loss: 1.8113
Epoch [    5/    5] | d_loss: 1.2087 | g_loss: 1.2641
Epoch [    5/    5] | d_loss: 1.0632 | g_loss: 1.2208
Epoch [    5/    5] | d_loss: 1.0690 | g_loss: 1.1960
Epoch [    5/    5] | d_loss: 1.2447 | g_loss: 1.6455
Epoch [    5/    5] | d_loss: 1.0945 | g_loss: 0.9510
Epoch [    5/    5] | d_loss: 1.0486 | g_loss: 1.2811
Epoch [    5/    5] | d_loss: 1.1892 | g_loss: 1.4895
Epoch [    5/    5] | d_loss: 1.1955 | g_loss: 1.6984
Epoch [    5/    5] | d_loss: 1.0672 | g_loss: 1.5833
Epoch [    5/    5] | d_loss: 1.0685 | g_loss: 1.6787
Epoch [    5/    5] | d_loss: 1.0731 | g_loss: 1.4402
Epoch [    5/    5] | d_loss: 1.1193 | g_loss: 1.3584
Epoch [    5/    5] | d_loss: 1.1736 | g_loss: 0.8729
Epoch [    5/    5] | d_loss: 1.1918 | g_loss: 1.8786
Epoch [    5/    5] | d_loss: 1.0325 | g_loss: 1.3899
Epoch [    5/    5] | d_loss: 1.1024 | g_loss: 0.9886
Epoch [    5/    5] | d_loss: 0.9862 | g_loss: 1.5016
Epoch [    5/    5] | d_loss: 1.2605 | g_loss: 1.6372
Epoch [    5/    5] | d_loss: 1.0174 | g_loss: 2.9381
Epoch [    5/    5] | d_loss: 1.1860 | g_loss: 1.4207
Epoch [    5/    5] | d_loss: 1.2058 | g_loss: 1.0968
Epoch [    5/    5] | d_loss: 0.9666 | g_loss: 1.4738
Epoch [    5/    5] | d_loss: 1.0771 | g_loss: 1.6436
Epoch [    5/    5] | d_loss: 1.0786 | g_loss: 1.5552
Epoch [    5/    5] | d_loss: 0.8524 | g_loss: 1.4676
Epoch [    5/    5] | d_loss: 1.0555 | g_loss: 1.4262
Epoch [    5/    5] | d_loss: 0.9890 | g_loss: 1.1580
Epoch [    5/    5] | d_loss: 1.0852 | g_loss: 1.1590
Epoch [    5/    5] | d_loss: 1.0275 | g_loss: 1.5473
Epoch [    5/    5] | d_loss: 1.0663 | g_loss: 1.3242
Epoch [    5/    5] | d_loss: 0.9070 | g_loss: 1.4726
Epoch [    5/    5] | d_loss: 1.3565 | g_loss: 0.8893
Epoch [    5/    5] | d_loss: 1.1367 | g_loss: 1.4613
Epoch [    5/    5] | d_loss: 1.2253 | g_loss: 2.0900
Epoch [    5/    5] | d_loss: 1.0446 | g_loss: 0.8971
```

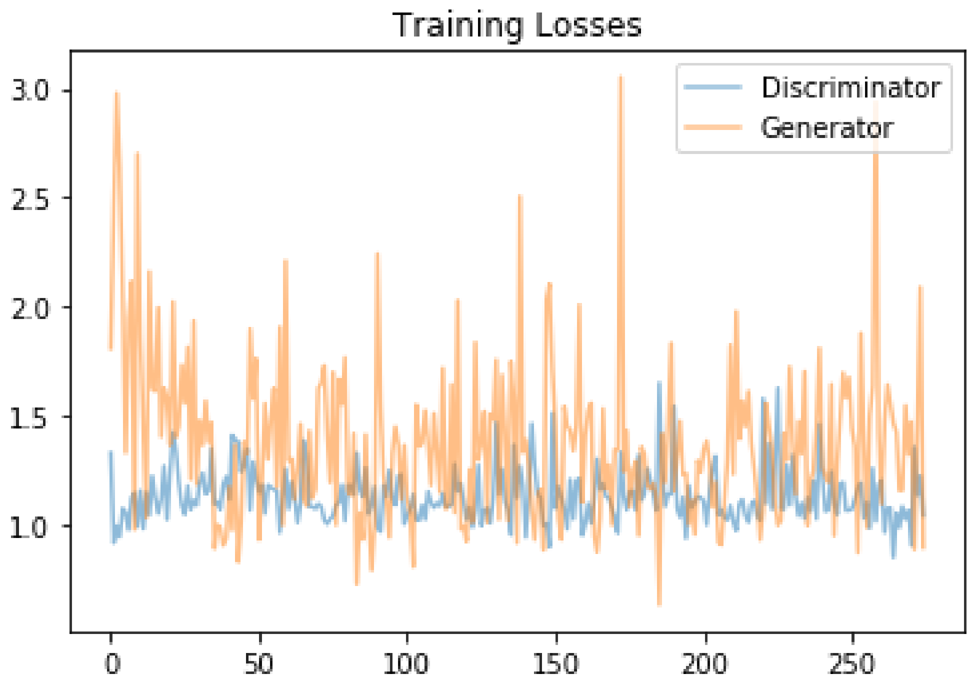
```
---------------Epoch [    5/    5]---------------
```

2.8 Training loss

Plot the training losses for the generator and discriminator, recorded after each epoch.

```
In [50]: fig, ax = plt.subplots()
         losses = np.array(losses)
         plt.plot(losses.T[0], label='Discriminator', alpha=0.5)
         plt.plot(losses.T[1], label='Generator', alpha=0.5)
         plt.title("Training Losses")
         plt.legend()
```

```
Out[50]: <matplotlib.legend.Legend at 0x7f01f018ad30>
```



2.9 Generator samples from training

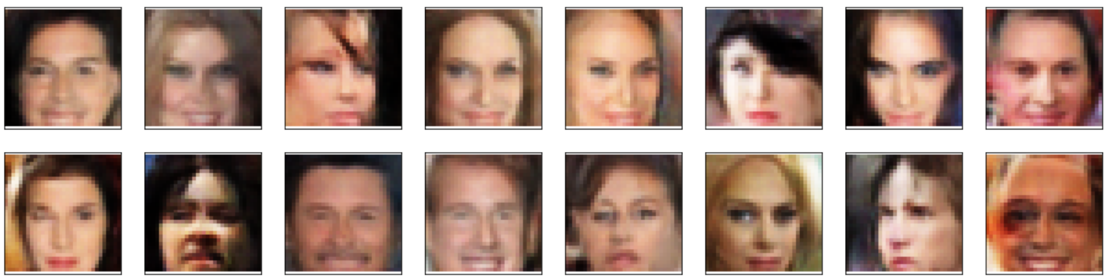
View samples of images from the generator, and answer a question about the strengths and weaknesses of your trained models.

```
In [51]: # helper function for viewing a list of passed in sample images
         def view_samples(epoch, samples):
             fig, axes = plt.subplots(figsize=(16,4), nrows=2, ncols=8, sharey=True, sharex=True
             for ax, img in zip(axes.flatten(), samples[epoch]):
```

```
        img = img.detach().cpu().numpy()
        img = np.transpose(img, (1, 2, 0))
        img = ((img + 1)*255 / (2)).astype(np.uint8)
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
        im = ax.imshow(img.reshape((32,32,3)))
```

In [52]:
```
# Load samples from generator, taken while training
with open('train_samples.pkl', 'rb') as f:
    samples = pkl.load(f)
```

In [53]:
```
_ = view_samples(-1, samples)
```



2.9.1 Question: What do you notice about your generated samples and how might you improve this model?

When you answer this question, consider the following factors: * The dataset is biased; it is made of "celebrity" faces that are mostly white * Model size; larger models have the opportunity to learn more features in a data feature space * Optimization strategy; optimizers and number of epochs affect your final result

Answer: - I tried various hyper parameters like - lr=0.0005, beta1=0.2 - lr=0.0002, beta1=0.5 The 2nd option worked but gave more g_loss. So I analyzed some blogs and papers and got the suggestion that I should change batchsize, so I changed it from 16 to 32. Then the generating loss reduced gradually. - As I analyzed the generated samples, I wanted my images to be more clear with discriminator and generator loss as close as possible. Also wanted some diversity in ages and skin color. - I think of making the following improvemnts : - based on suggestions, provided by mentor Knowledge center. I think of increasing the image size provided by the generator to 128x 128, then resizing it to 32x32 and providing it to discriminator. - Also using a different dataset of cartoons or faces of all diversity to train images would help like one by IBM - Maybe using some other method for smoothing the value for loss calculation.

2.9.2 Submitting This Project

When submitting this project, make sure to run all the cells before saving the notebook. Save the notebook file as "dlnd_face_generation.ipynb" and save it as a HTML file under "File" -> "Download as". Include the "problem_unittests.py" files in your submission.