dlnd_face_generation

March 27, 2019

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 [42]:
```
# can comment out after executing
#!unzip processed_celeba_small.zip
```

In [43]:
```
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 [44]: # necessary imports
         import torch
         from torchvision import datasets
         from torchvision import transforms
```

```
In [45]: from torch.utils.data import DataLoader
```

```
In [46]: import os
```

```
In [47]: 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
```

```
    """

    # define the transformations
    transform = transforms.Compose([transforms.Resize(image_size),
                                   transforms.ToTensor()])

    # create dataset and dataloader
    num_workers = 0
    train_data = datasets.ImageFolder(root=data_dir, transform=transform)
    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True, num_wo


    return train_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 [48]:
```
# Define function hyperparameters
batch_size = 64
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.

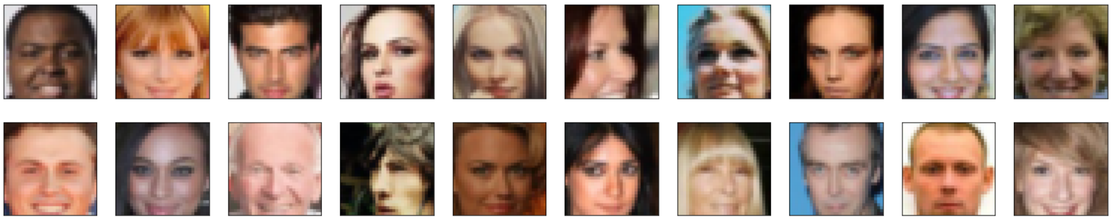
Note: You'll need to convert the Tensor images into a NumPy type and transpose the dimensions to correctly display an image, suggested `imshow` code is below, but it may not be perfect.

In [49]:
```
# 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 [50]:
```
# 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

    x_min, x_max = feature_range[0], feature_range[1]
    x = x*(x_max-x_min)-1

    return x
```

In [51]:
```
"""
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(-0.8824)
Max:  tensor(0.8118)
```

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 [52]:
```
import torch.nn as nn
import torch.nn.functional as F
```

In [53]:
```
# since we will be using batch normalization layers, a helper function will be useful
def conv(in_channels, out_channels, kernel_size=4, stride=2, padding=1, batch_norm=Tru
    """Helper function to create conv layer with opt. batch normalisation. These are
    stride convolutional layers that half input size. Don't need maxpooling layers.
    """

    layers = []

    # need to set the bias to zero when using batch norm layers
    conv_layer = nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_
                           stride=stride, padding=padding, bias=False)
    layers.append(conv_layer)

    # optional batch norm layer
    if batch_norm:
        batch_norm = nn.BatchNorm2d(num_features=out_channels)
        layers.append(batch_norm)

    # use the Sequential wrapper
    conv_layers = nn.Sequential(*layers)

    return conv_layers
```

In [54]:
```
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__()

        # save class variables
        self.conv_dim = conv_dim

        # create all layers. Essentially, I just reproduce the architecture from the 
        self.conv_1 = conv(3, conv_dim, batch_norm=False) # fist conv layer should ha
        # output: 16*16*conv_dim
        self.conv_2 = conv(conv_dim, conv_dim*2, batch_norm=True) # output: 8*8*2*con
        self.conv_3 = conv(conv_dim*2, conv_dim*4, batch_norm=True) # output: 4*4*4*c
        self.fc_1 = nn.Linear(4*4*4*conv_dim, 1, )

    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
        """

        # convolutional part of the classifier
        x = F.leaky_relu(self.conv_1(x), negative_slope=0.2)
        x = F.leaky_relu(self.conv_2(x), negative_slope=0.2)
        x = F.leaky_relu(self.conv_3(x), negative_slope=0.2)

        # reshape and feed through fully connected layer
        # batch size first
        x = x.view(-1, 4*4*4*self.conv_dim)
        x = self.fc_1(x) # no activation funcito, will use BCE with sigmoid later

        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 [55]:
```
# again, useful to create a helper function here
def deconv(in_channels, out_channels, kernel_size=4, stride=2, padding=1, batch_norm=T
    """This created a transposed convolutional layer with optional batch norm applied
    This will increase input dimensions in every direction by factor 2
    """

    layers = []
    deconv = nn.ConvTranspose2d(in_channels=in_channels, out_channels=out_channels, ke
                                stride=stride, padding=padding, bias=False)
    layers.append(deconv)

    # optional batch norm layer
    if batch_norm:
        layers.append(nn.BatchNorm2d(num_features=out_channels))

    return nn.Sequential(*layers)
```

In [56]:
```
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 convolutiona
        """
        super(Generator, self).__init__()
        self.conv_dim = conv_dim

        # We start with a fully connected layer to bring z to the right dimensions
        self.fc_1 = nn.Linear(z_size, 4*4*conv_dim*4)

        # couple of deconv layers
        self.deconv_1 = deconv(conv_dim*4, conv_dim*2)
        self.deconv_2 = deconv(conv_dim*2, conv_dim)
        self.deconv_3 = deconv(conv_dim, 3, batch_norm=False)


    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
        """
        # feed through fc layers to get to the right size
        x = F.relu(self.fc_1(x))
        x = x.view(-1, self.conv_dim*4, 4, 4)
```

```
            # feed through deconv layers to upsample
            x = F.relu(self.deconv_1(x))
            x = F.relu(self.deconv_2(x))
            x = F.tanh(self.deconv_3(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 [57]: from torch.nn import init
```

```
In [58]: 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
             """
             # classname will be something like:
             # `Conv`, `BatchNorm2d`, `Linear`, etc.
             classname = m.__class__.__name__

             # TODO: Apply initial weights to convolutional and linear layers
             if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Line
                 init.normal_(m.weight.data, 0.0, 0.02)
```

```python
            if hasattr(m, 'bias') and m.bias is not None:
                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 [59]:
```python
"""
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 [60]:
```python
# Define model hyperparams
d_conv_dim = 32
g_conv_dim = 32
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(
  (conv_1): Sequential(
    (0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
  (conv_2): Sequential(
    (0): Conv2d(32, 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)
  )
  (conv_3): 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)
  )
  (fc_1): Linear(in_features=2048, out_features=1, bias=True)
)

Generator(
  (fc_1): Linear(in_features=100, out_features=2048, bias=True)
  (deconv_1): 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)
  )
  (deconv_2): Sequential(
    (0): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (deconv_3): Sequential(
    (0): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
)
```

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 [61]:
```python
"""
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 [62]:
```
train_on_gpu
```

Out[62]:
```
True
```

In [63]:
```
def real_loss(D_out, smooth=False):
    '''Calculates how close discriminator outputs are to being real.
       param, D_out: discriminator logits
       return: real loss'''

    batch_size = D_out.size(0)
    if smooth:
        labels = torch.ones(batch_size)*0.9
    else:
        labels = torch.ones(batch_size)

    # move to GPU if available
    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'''

    # no label smoothing here
    batch_size = D_out.size(0)
    labels = torch.zeros(batch_size)
    if train_on_gpu:
        labels = labels.cuda()
```

```
        criterion = nn.BCEWithLogitsLoss()
        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 [64]: import torch.optim as optim
```

```
In [65]: # Create optimizers for the discriminator D and generator G

         lr = 0.002
         beta1 = 0.5
         beta2 = 0.999

         d_optimizer = optim.Adam(D.parameters(), lr=lr, betas=[beta1, beta2])
         g_optimizer = optim.Adam(G.parameters(), lr=lr, betas=[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 [66]: from workspace_utils import active_session
```

```
In [67]: 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'''
```

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

# 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
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
        # zero out accumulated gradients
        d_optimizer.zero_grad()

        # move images to the GPU if available
        if train_on_gpu:
            real_images = real_images.cuda()

        # loss on real images
        out_real = D.forward(real_images)
        d_loss_real = real_loss(out_real, smooth=True)

        # loss on 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.forward(z)
            out_fake = D.forward(fake_images)
            d_loss_fake = fake_loss(out_fake)

            # sum them up and run a step of optimization
            d_loss = d_loss_fake + d_loss_real
            d_loss.backward()
            d_optimizer.step()

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

            # create 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.forward(z)
            out_fake = D.forward(fake_images)
            g_loss = real_loss(out_fake, smooth=True)
            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

    # Save training generator samples
    with open('train_samples.pkl', 'wb') as f:
        pkl.dump(samples, f)
```

```
        # save the losses
        with open('losses.pkl', 'wb') as f:
            pkl.dump(losses, f)

        # finally return losses
        return losses
```

Set your number of training epochs and train your GAN!

```
In [68]: # define helper funcitons for saving and loading models

         def save_models(filename, D, G):
             filename_G = os.path.splitext(os.path.basename(filename))[0] +  '_G_.pt'
             filename_D = os.path.splitext(os.path.basename(filename))[0] + '_D_.pt'

             torch.save(D, os.path.join('saved_models', filename_G))
             torch.save(G, os.path.join('saved_models', filename_D))

In [69]: def load_models(filename):
             filename_G = os.path.splitext(os.path.basename(filename))[0] +  '_G_.pt'
             filename_D = os.path.splitext(os.path.basename(filename))[0] + '_D_.pt'

             D = torch.load(os.path.join('saved_models', filename_D))
             G = torch.load(os.path.join('saved_models', filename_G))

             return D, G

In [70]: # set number of epochs
         n_epochs = 20


         """
         DON'T MODIFY ANYTHING IN THIS CELL
         """
         # call training function
         with active_session():
             losses = train(D, G, n_epochs=n_epochs, print_every=200)
             # save models
             save_models('test_run', D, G)
```

```
Epoch [    1/   20] | d_loss: 1.4085 | g_loss: 1.6797
Epoch [    1/   20] | d_loss: 1.9654 | g_loss: 2.8596
Epoch [    1/   20] | d_loss: 0.9133 | g_loss: 3.2011
Epoch [    1/   20] | d_loss: 1.2665 | g_loss: 2.3290
Epoch [    1/   20] | d_loss: 0.8755 | g_loss: 2.9497
Epoch [    1/   20] | d_loss: 0.9073 | g_loss: 2.1558
Epoch [    1/   20] | d_loss: 1.0728 | g_loss: 1.8908
Epoch [    1/   20] | d_loss: 1.1615 | g_loss: 1.7977
```

```
Epoch [    2/   20] | d_loss: 1.0007 | g_loss: 2.0010
Epoch [    2/   20] | d_loss: 1.2092 | g_loss: 1.0342
Epoch [    2/   20] | d_loss: 1.1393 | g_loss: 2.9866
Epoch [    2/   20] | d_loss: 1.2985 | g_loss: 2.7369
Epoch [    2/   20] | d_loss: 0.9051 | g_loss: 1.6610
Epoch [    2/   20] | d_loss: 1.0236 | g_loss: 1.9128
Epoch [    2/   20] | d_loss: 1.1514 | g_loss: 1.4379
Epoch [    2/   20] | d_loss: 1.0792 | g_loss: 1.1933
Epoch [    3/   20] | d_loss: 1.1338 | g_loss: 1.0980
Epoch [    3/   20] | d_loss: 1.1549 | g_loss: 2.1727
Epoch [    3/   20] | d_loss: 1.1001 | g_loss: 1.4267
Epoch [    3/   20] | d_loss: 1.0624 | g_loss: 1.8107
Epoch [    3/   20] | d_loss: 1.2498 | g_loss: 0.9424
Epoch [    3/   20] | d_loss: 1.1725 | g_loss: 1.0707
Epoch [    3/   20] | d_loss: 1.1299 | g_loss: 2.1788
Epoch [    3/   20] | d_loss: 1.2431 | g_loss: 0.9933
Epoch [    4/   20] | d_loss: 1.2462 | g_loss: 1.8119
Epoch [    4/   20] | d_loss: 1.1436 | g_loss: 1.4189
Epoch [    4/   20] | d_loss: 1.1219 | g_loss: 1.7847
Epoch [    4/   20] | d_loss: 1.2202 | g_loss: 0.9966
Epoch [    4/   20] | d_loss: 1.1492 | g_loss: 0.9284
Epoch [    4/   20] | d_loss: 1.1970 | g_loss: 1.4127
Epoch [    4/   20] | d_loss: 1.2645 | g_loss: 1.0563
Epoch [    4/   20] | d_loss: 1.0906 | g_loss: 1.4178
Epoch [    5/   20] | d_loss: 1.0363 | g_loss: 1.1238
Epoch [    5/   20] | d_loss: 1.1172 | g_loss: 1.2252
Epoch [    5/   20] | d_loss: 1.0411 | g_loss: 1.4810
Epoch [    5/   20] | d_loss: 1.1754 | g_loss: 1.4299
Epoch [    5/   20] | d_loss: 1.0971 | g_loss: 1.5812
Epoch [    5/   20] | d_loss: 1.2562 | g_loss: 1.3027
Epoch [    5/   20] | d_loss: 1.1893 | g_loss: 1.5567
Epoch [    5/   20] | d_loss: 1.0495 | g_loss: 0.9759
Epoch [    6/   20] | d_loss: 1.0843 | g_loss: 1.4443
Epoch [    6/   20] | d_loss: 0.9862 | g_loss: 1.2099
Epoch [    6/   20] | d_loss: 1.3647 | g_loss: 1.9470
Epoch [    6/   20] | d_loss: 1.3128 | g_loss: 2.0859
Epoch [    6/   20] | d_loss: 0.9008 | g_loss: 1.7177
Epoch [    6/   20] | d_loss: 0.9097 | g_loss: 1.8823
Epoch [    6/   20] | d_loss: 1.0772 | g_loss: 1.8933
Epoch [    6/   20] | d_loss: 0.9980 | g_loss: 1.2596
Epoch [    7/   20] | d_loss: 1.1620 | g_loss: 1.6711
Epoch [    7/   20] | d_loss: 0.9617 | g_loss: 1.8993
Epoch [    7/   20] | d_loss: 0.9540 | g_loss: 1.3879
Epoch [    7/   20] | d_loss: 0.8085 | g_loss: 2.0945
Epoch [    7/   20] | d_loss: 1.0186 | g_loss: 1.5824
Epoch [    7/   20] | d_loss: 1.0008 | g_loss: 2.2853
Epoch [    7/   20] | d_loss: 1.0539 | g_loss: 2.1235
Epoch [    7/   20] | d_loss: 0.9257 | g_loss: 1.3333
```

```
Epoch [    8/   20] | d_loss: 1.3719 | g_loss: 2.5110
Epoch [    8/   20] | d_loss: 1.0630 | g_loss: 2.1658
Epoch [    8/   20] | d_loss: 0.8910 | g_loss: 1.3444
Epoch [    8/   20] | d_loss: 1.1450 | g_loss: 1.5528
Epoch [    8/   20] | d_loss: 0.9285 | g_loss: 1.3445
Epoch [    8/   20] | d_loss: 0.8835 | g_loss: 1.8925
Epoch [    8/   20] | d_loss: 1.0031 | g_loss: 1.2902
Epoch [    8/   20] | d_loss: 1.1903 | g_loss: 1.5033
Epoch [    9/   20] | d_loss: 0.9921 | g_loss: 1.2919
Epoch [    9/   20] | d_loss: 1.7581 | g_loss: 2.3276
Epoch [    9/   20] | d_loss: 1.1013 | g_loss: 1.2751
Epoch [    9/   20] | d_loss: 0.7197 | g_loss: 2.6704
Epoch [    9/   20] | d_loss: 1.0329 | g_loss: 2.3747
Epoch [    9/   20] | d_loss: 0.9465 | g_loss: 2.0504
Epoch [    9/   20] | d_loss: 0.9179 | g_loss: 1.7869
Epoch [    9/   20] | d_loss: 0.8754 | g_loss: 2.5746
Epoch [   10/   20] | d_loss: 1.0108 | g_loss: 2.0277
Epoch [   10/   20] | d_loss: 1.0832 | g_loss: 2.0219
Epoch [   10/   20] | d_loss: 1.0131 | g_loss: 2.1754
Epoch [   10/   20] | d_loss: 0.9172 | g_loss: 2.0244
Epoch [   10/   20] | d_loss: 1.0627 | g_loss: 1.9043
Epoch [   10/   20] | d_loss: 0.9397 | g_loss: 2.2459
Epoch [   10/   20] | d_loss: 0.8826 | g_loss: 2.1297
Epoch [   10/   20] | d_loss: 0.7076 | g_loss: 2.3231
Epoch [   11/   20] | d_loss: 1.4315 | g_loss: 2.8529
Epoch [   11/   20] | d_loss: 0.8496 | g_loss: 2.0763
Epoch [   11/   20] | d_loss: 0.7054 | g_loss: 2.8200
Epoch [   11/   20] | d_loss: 0.9234 | g_loss: 1.5576
Epoch [   11/   20] | d_loss: 0.8166 | g_loss: 1.5277
Epoch [   11/   20] | d_loss: 0.7818 | g_loss: 1.7172
Epoch [   11/   20] | d_loss: 1.1832 | g_loss: 2.0288
Epoch [   11/   20] | d_loss: 0.9384 | g_loss: 1.4757
Epoch [   12/   20] | d_loss: 1.0270 | g_loss: 1.3097
Epoch [   12/   20] | d_loss: 0.8983 | g_loss: 2.0235
Epoch [   12/   20] | d_loss: 0.9679 | g_loss: 2.6909
Epoch [   12/   20] | d_loss: 0.9389 | g_loss: 1.1859
Epoch [   12/   20] | d_loss: 0.9258 | g_loss: 1.4827
Epoch [   12/   20] | d_loss: 0.9246 | g_loss: 0.9269
Epoch [   12/   20] | d_loss: 0.8519 | g_loss: 1.7244
Epoch [   12/   20] | d_loss: 0.7193 | g_loss: 1.4648
Epoch [   13/   20] | d_loss: 1.6197 | g_loss: 1.1044
Epoch [   13/   20] | d_loss: 0.9497 | g_loss: 1.8480
Epoch [   13/   20] | d_loss: 0.5861 | g_loss: 2.3280
Epoch [   13/   20] | d_loss: 0.9832 | g_loss: 2.0721
Epoch [   13/   20] | d_loss: 0.7572 | g_loss: 1.6757
Epoch [   13/   20] | d_loss: 1.0164 | g_loss: 1.8160
Epoch [   13/   20] | d_loss: 0.8732 | g_loss: 1.0368
Epoch [   13/   20] | d_loss: 0.9519 | g_loss: 2.1948
```

```
Epoch [   14/   20] | d_loss: 1.0249 | g_loss: 2.0040
Epoch [   14/   20] | d_loss: 0.7439 | g_loss: 1.7291
Epoch [   14/   20] | d_loss: 0.5619 | g_loss: 2.3486
Epoch [   14/   20] | d_loss: 0.9228 | g_loss: 1.3780
Epoch [   14/   20] | d_loss: 1.3818 | g_loss: 0.4998
Epoch [   14/   20] | d_loss: 0.9649 | g_loss: 1.9225
Epoch [   14/   20] | d_loss: 0.9694 | g_loss: 1.9332
Epoch [   14/   20] | d_loss: 0.9179 | g_loss: 2.4936
Epoch [   15/   20] | d_loss: 1.4050 | g_loss: 4.0071
Epoch [   15/   20] | d_loss: 0.7551 | g_loss: 1.8494
Epoch [   15/   20] | d_loss: 0.8381 | g_loss: 1.3455
Epoch [   15/   20] | d_loss: 0.6899 | g_loss: 2.2511
Epoch [   15/   20] | d_loss: 0.7439 | g_loss: 2.0075
Epoch [   15/   20] | d_loss: 1.0011 | g_loss: 1.8389
Epoch [   15/   20] | d_loss: 0.6235 | g_loss: 1.8882
Epoch [   15/   20] | d_loss: 0.9006 | g_loss: 1.3794
Epoch [   16/   20] | d_loss: 0.6907 | g_loss: 2.2416
Epoch [   16/   20] | d_loss: 1.3841 | g_loss: 1.1147
Epoch [   16/   20] | d_loss: 0.7801 | g_loss: 2.2977
Epoch [   16/   20] | d_loss: 0.6376 | g_loss: 2.6631
Epoch [   16/   20] | d_loss: 0.6358 | g_loss: 2.7277
Epoch [   16/   20] | d_loss: 0.9399 | g_loss: 1.7749
Epoch [   16/   20] | d_loss: 0.7202 | g_loss: 1.7689
Epoch [   16/   20] | d_loss: 1.0842 | g_loss: 2.9711
Epoch [   17/   20] | d_loss: 0.9619 | g_loss: 2.4094
Epoch [   17/   20] | d_loss: 0.6688 | g_loss: 2.6273
Epoch [   17/   20] | d_loss: 1.4767 | g_loss: 3.8651
Epoch [   17/   20] | d_loss: 0.8388 | g_loss: 2.5390
Epoch [   17/   20] | d_loss: 1.5722 | g_loss: 1.1288
Epoch [   17/   20] | d_loss: 0.7022 | g_loss: 2.7000
Epoch [   17/   20] | d_loss: 2.5118 | g_loss: 5.6599
Epoch [   17/   20] | d_loss: 0.6498 | g_loss: 2.6023
Epoch [   18/   20] | d_loss: 1.1505 | g_loss: 1.6230
Epoch [   18/   20] | d_loss: 0.6662 | g_loss: 1.4589
Epoch [   18/   20] | d_loss: 0.6843 | g_loss: 1.8510
Epoch [   18/   20] | d_loss: 0.7024 | g_loss: 2.4766
Epoch [   18/   20] | d_loss: 0.7368 | g_loss: 2.4390
Epoch [   18/   20] | d_loss: 0.6951 | g_loss: 2.1719
Epoch [   18/   20] | d_loss: 0.8790 | g_loss: 2.0607
Epoch [   18/   20] | d_loss: 0.7513 | g_loss: 2.8165
Epoch [   19/   20] | d_loss: 0.8602 | g_loss: 1.9574
Epoch [   19/   20] | d_loss: 0.6102 | g_loss: 2.9343
Epoch [   19/   20] | d_loss: 0.8086 | g_loss: 1.6789
Epoch [   19/   20] | d_loss: 0.5976 | g_loss: 2.0870
Epoch [   19/   20] | d_loss: 1.4322 | g_loss: 2.0248
Epoch [   19/   20] | d_loss: 0.7777 | g_loss: 2.0528
Epoch [   19/   20] | d_loss: 0.6091 | g_loss: 3.1853
Epoch [   19/   20] | d_loss: 0.7229 | g_loss: 1.8909
```

```
Epoch [   20/   20] | d_loss: 1.3079 | g_loss: 3.7941
Epoch [   20/   20] | d_loss: 0.6446 | g_loss: 2.0223
Epoch [   20/   20] | d_loss: 0.7095 | g_loss: 2.0462
Epoch [   20/   20] | d_loss: 0.8128 | g_loss: 2.8174
Epoch [   20/   20] | d_loss: 0.9917 | g_loss: 2.4630
Epoch [   20/   20] | d_loss: 0.9123 | g_loss: 1.7819
Epoch [   20/   20] | d_loss: 0.7884 | g_loss: 3.0656
Epoch [   20/   20] | d_loss: 1.3286 | g_loss: 0.8428
```

```
/opt/conda/lib/python3.6/site-packages/torch/serialization.py:193: UserWarning: Couldn't retrie
  "type " + obj.__name__ + ". It won't be checked "
/opt/conda/lib/python3.6/site-packages/torch/serialization.py:193: UserWarning: Couldn't retrie
  "type " + obj.__name__ + ". It won't be checked "
```

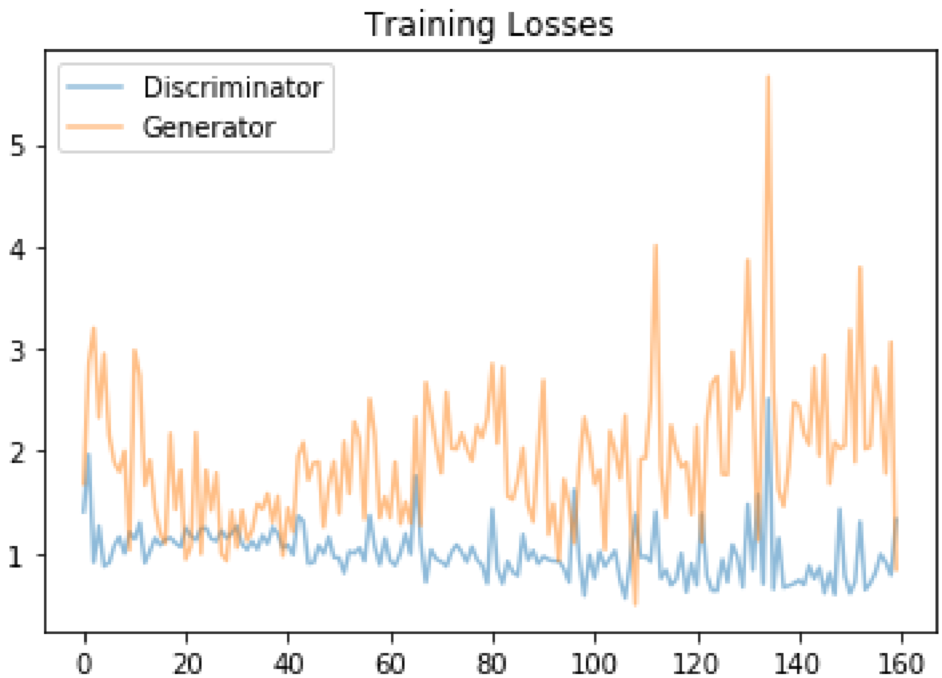
2.8 Training loss

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

In [71]:
```
# load in the models
# D, G = load_models('test_save')
```

In [72]:
```
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[72]:
```
<matplotlib.legend.Legend at 0x7f135ad78f60>
```




```
In [73]: #losses_no_smoothing = losses
```

```
In [74]: #losses_smoothing = losses
```

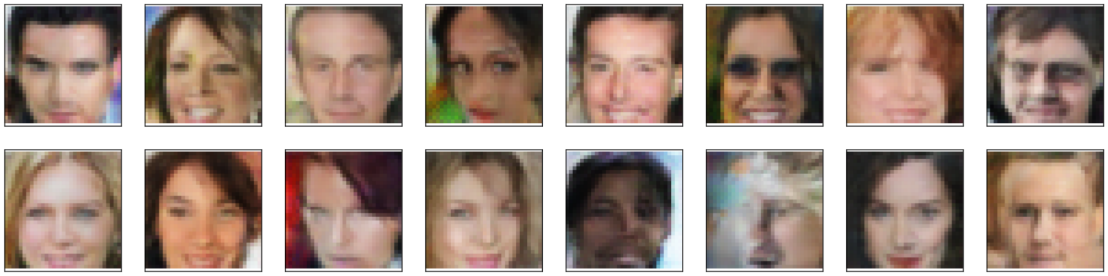
2.9 Generator samples from training

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

```
In [75]: # 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=Tru
             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 [76]: # Load samples from generator, taken while training
         with open('train_samples.pkl', 'rb') as f:
             samples = pkl.load(f)
```

```
In [77]: _ = 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:

What do I notice about the generated samples? * first of all, they do look like faces (mostly) which is good * a bit blurry * somethimes looks like two faces merged into one

How might I improve this model? * Larger, more balanced dataset would allow one to create more realistic and more diverse faces * Increasing the model size: more convolutional layers! Potentially using a pre-trained classifier. Would however make it difficult for the generator to keep up in the learning process * Try different loss functions, mean squared loss for example, as we saw in the lectures * Run for more epochs, potentially try SGD or other optimizers, play around with the model parameters

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.