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.

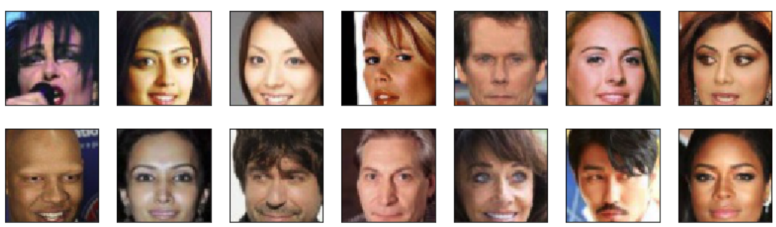
Get the Data

You'll be using the CelebFaces Attributes Dataset (CelebA) (http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html) 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.

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 (https://s3.amazonaws.com/video.udacity-data.com/topher/2018/November/5be7eb6f_processed-celeba-small/processed-celeba-small.zip)

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
```

In [2]:
```
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
```

Visualize the CelebA Data

The CelebA (http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html) 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) (https://en.wikipedia.org/wiki/Channel_(digital_image)#RGB_Images) each.

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 (https://pytorch.org/docs/stable/torchvision/datasets.html#imagefolder) wrapper, with a root directory `processed_celeba_small/` and data transformation passed in.

In [3]:
```
# necessary imports
import torch
from torchvision import datasets
from torchvision import transforms
```

In [4]:
```
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
    transform_train = transforms.Compose([
        transforms.Resize(size=(image_size, image_size)),
        transforms.ToTensor()
    ])
    # datasets
    dataset_train = datasets.ImageFolder(data_dir ,transform=transform_train)
    num_workers = 0
    # build DataLoaders
    train_loader = torch.utils.data.DataLoader(dataset=dataset_train,
                                               batch_size=batch_size,
                                               shuffle=True,
                                               num_workers=num_workers)

    return train_loader
```

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 [5]:
```
# Define function hyperparameters
batch_size = 128 # tuned
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)
```

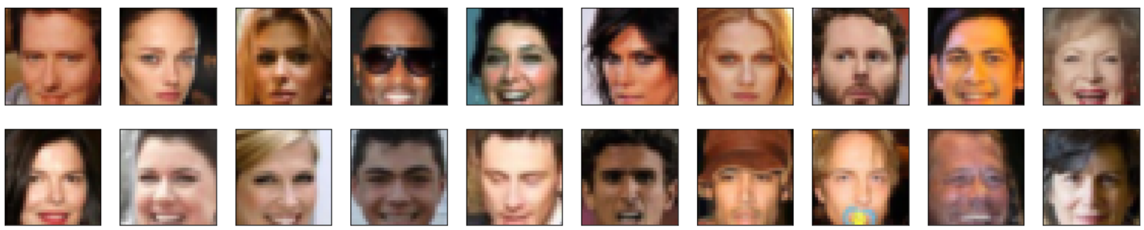
Next, you can view some images! You should seen square images of somewhat-centered faces.

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 [6]:
```
# 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 [7]:
```
# 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 = (max - min)* x + min

    return x
```

In [8]:
```
"""
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(-0.9608)
Max:  tensor(0.8275)
```

Define the Model

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

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 [9]:
```
import torch.nn as nn
import torch.nn.functional as F
```

In [10]:
```
# add a helper conv function with optional batch_norm
# use kernel size =4; stride = 2
def conv(in_channels, out_channels, kernel_size=4, 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)

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__()

        # complete init function

        self.conv_dim = conv_dim

        self.conv1 = conv(3, conv_dim, 4, batch_norm = False)   # out depth= 16, d
ownsize 16x16
        self.conv2 = conv(conv_dim, conv_dim*2, 4)              # out depth= 32, d
ownsize 8x8
        self.conv3 = conv(conv_dim*2, conv_dim*4, 4)            # out depth= 64, d
ownsize 4x4

        self.fc = nn.Linear(conv_dim*4 * 4*4, 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
        """
        # define feedforward behavior
        out = F.leaky_relu(self.conv1(x), 0.2)
        out = F.leaky_relu(self.conv2(out), 0.2)
        out = F.leaky_relu(self.conv3(out), 0.2)

        # flatten
        out = out.view(-1, self.conv_dim*4*4*4)

        # final output layer; -- logits without softmax
        out = self.fc(out)
        return out
```

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

```
Tests Passed
```

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=4, stride=2, padding=1, batch_no
rm=True):
    """Creates a transposed-convolutional layer, with optional batch normalizatio
n.
    """
    # create a sequence of transpose + optional batch norm layers
    layers = []
    transpose_conv_layer = nn.ConvTranspose2d(in_channels, out_channels,
                                              kernel_size, stride, padding, bias=F
alse)
    # append transpose convolutional layer
    layers.append(transpose_conv_layer)

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

    return nn.Sequential(*layers)


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 convoluti
onal layer
        """
        super(Generator, self).__init__()

        # complete init function
        self.conv_dim = conv_dim

        # first, fully-connected layer
        self.fc = nn.Linear(z_size, conv_dim*4*4*4)

        # transpose conv layers
        self.t_conv1 = deconv(conv_dim*4, conv_dim*2)
        self.t_conv2 = deconv(conv_dim*2, conv_dim)
        self.t_conv3 = 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
        """
        # define feedforward behavior

        # fully-connected + reshape
        out = self.fc(x)
        out = out.view(-1, self.conv_dim*4, 4, 4) # (batch_size, depth, 4, 4)

        # hidden transpose conv layers + relu
        out = F.relu(self.t_conv1(out))
        out = F.relu(self.t_conv2(out))

        # last layer + tanh activation
```

```
        out = self.t_conv3(out)
        out = F.tanh(out)

        return out

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

```
Tests Passed
```

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 (https://arxiv.org/pdf/1511.06434.pdf), 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 (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/blob/master/models/networks.py) 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 [12]:
```
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
    """
    # 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('L
inear') != -1):
        init.normal_(m.weight.data, 0.0, 0.02)
        if hasattr(m, 'bias') and m.bias is not None:
            init.constant_(m.bias.data, 0.0)
```

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 [13]:
```
"""
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 [14]:
```
# Define model hyperparams
d_conv_dim = 32
g_conv_dim = 32
z_size = 120

"""
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, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=F
alse)
  )
  (conv2): 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_sta
ts=True)
  )
  (conv3): 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_st
ats=True)
  )
  (fc): Linear(in_features=2048, out_features=1, bias=True)
)

Generator(
  (fc): Linear(in_features=120, out_features=2048, bias=True)
  (t_conv1): 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_sta
ts=True)
  )
  (t_conv2): 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_sta
ts=True)
  )
  (t_conv3): Sequential(
    (0): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
  )
)
```

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 [15]:
```
"""
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!
```

Discriminator and Generator Losses

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

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.

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 [16]:
```
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
    smooth = 1
    if smooth:
        # smooth, real labels = 0.9
        labels = torch.ones(batch_size)*0.9
    else:
        labels = torch.ones(batch_size) # real labels = 1
    # move labels to GPU if available
    if train_on_gpu:
        labels = labels.cuda()
    # binary cross entropy with logits loss
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    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
```

Optimizers

Exercise: Define optimizers for your Discriminator (D) and Generator (G)

Define optimizers for your models with appropriate hyperparameters.

In [17]:
```
import torch.optim as optim

# params
lr = 0.00015
beta1=0.3
beta2=0.999 # default value

# 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])
```

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 [18]:
```
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()

    # 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 performanc
e
    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

            d_optimizer.zero_grad()
            # 1-1. Train with real images

            # Compute the discriminator losses on real images
            if train_on_gpu:
                real_images = real_images.cuda()

            D_real = D(real_images)
            d_real_loss = real_loss(D_real)

            # 1-2. Train with fake images

            # Generate fake images
            z = np.random.uniform(-1, 1, size=(batch_size, z_size))
            z = torch.from_numpy(z).float()
            # move x to GPU, if available
            if train_on_gpu:
                z = z.cuda()
            fake_images = G(z)
```

```
            # Compute the discriminator losses on fake images
            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()

            # 2-1. Train with fake images and flipped labels

            # 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)

            # Compute the discriminator losses on fake images
            # using flipped labels!
            D_fake = D(fake_images)
            g_loss = real_loss(D_fake) # use real loss to flip labels

            # 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}'.fo
rmat(
                        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 nam
e
        # 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)

    # finally return losses
```

```
    return losses
```

Set your number of training epochs and train your GAN!

In [19]:
```
# set number of epochs
n_epochs = 50


"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# call training function
losses = train(D, G, n_epochs=n_epochs, print_every=500)
```

```
Epoch [    1/   50] | d_loss: 1.4753 | g_loss: 0.6830
Epoch [    1/   50] | d_loss: 0.9698 | g_loss: 1.9364
Epoch [    2/   50] | d_loss: 1.2657 | g_loss: 1.2205
Epoch [    2/   50] | d_loss: 1.2391 | g_loss: 0.8156
Epoch [    3/   50] | d_loss: 1.1130 | g_loss: 1.0573
Epoch [    3/   50] | d_loss: 1.1219 | g_loss: 0.9208
Epoch [    4/   50] | d_loss: 1.4415 | g_loss: 0.8623
Epoch [    4/   50] | d_loss: 1.1735 | g_loss: 0.9667
Epoch [    5/   50] | d_loss: 1.2080 | g_loss: 1.0042
Epoch [    5/   50] | d_loss: 1.1205 | g_loss: 1.1401
Epoch [    6/   50] | d_loss: 1.2084 | g_loss: 1.4391
Epoch [    6/   50] | d_loss: 1.1061 | g_loss: 0.7848
Epoch [    7/   50] | d_loss: 1.1301 | g_loss: 0.9895
Epoch [    7/   50] | d_loss: 1.1080 | g_loss: 1.5180
Epoch [    8/   50] | d_loss: 1.0329 | g_loss: 1.0068
Epoch [    8/   50] | d_loss: 1.0517 | g_loss: 0.9835
Epoch [    9/   50] | d_loss: 1.0870 | g_loss: 1.0530
Epoch [    9/   50] | d_loss: 0.8209 | g_loss: 1.4603
Epoch [   10/   50] | d_loss: 1.0364 | g_loss: 1.0176
Epoch [   10/   50] | d_loss: 1.0624 | g_loss: 0.8453
Epoch [   11/   50] | d_loss: 1.0410 | g_loss: 1.3708
Epoch [   11/   50] | d_loss: 1.1001 | g_loss: 1.0331
Epoch [   12/   50] | d_loss: 0.9823 | g_loss: 1.1091
Epoch [   12/   50] | d_loss: 1.0776 | g_loss: 0.9599
Epoch [   13/   50] | d_loss: 1.0128 | g_loss: 1.0887
Epoch [   13/   50] | d_loss: 0.9622 | g_loss: 1.6304
Epoch [   14/   50] | d_loss: 1.0120 | g_loss: 1.2035
Epoch [   14/   50] | d_loss: 0.8871 | g_loss: 1.2114
Epoch [   15/   50] | d_loss: 0.9620 | g_loss: 1.1517
Epoch [   15/   50] | d_loss: 0.9645 | g_loss: 1.2397
Epoch [   16/   50] | d_loss: 1.0248 | g_loss: 1.5851
Epoch [   16/   50] | d_loss: 0.9409 | g_loss: 1.5218
Epoch [   17/   50] | d_loss: 1.1107 | g_loss: 1.0155
Epoch [   17/   50] | d_loss: 0.9264 | g_loss: 1.5466
Epoch [   18/   50] | d_loss: 0.9364 | g_loss: 0.8970
Epoch [   18/   50] | d_loss: 0.9048 | g_loss: 1.2333
Epoch [   19/   50] | d_loss: 0.9608 | g_loss: 1.1387
Epoch [   19/   50] | d_loss: 0.8745 | g_loss: 1.5552
Epoch [   20/   50] | d_loss: 0.8941 | g_loss: 1.7462
Epoch [   20/   50] | d_loss: 0.7439 | g_loss: 1.7269
Epoch [   21/   50] | d_loss: 0.7744 | g_loss: 1.6529
Epoch [   21/   50] | d_loss: 0.8305 | g_loss: 1.4188
Epoch [   22/   50] | d_loss: 0.8076 | g_loss: 1.5956
Epoch [   22/   50] | d_loss: 2.2760 | g_loss: 2.9168
Epoch [   23/   50] | d_loss: 0.6940 | g_loss: 1.6370
Epoch [   23/   50] | d_loss: 0.7684 | g_loss: 1.8654
Epoch [   24/   50] | d_loss: 0.8397 | g_loss: 1.1569
Epoch [   24/   50] | d_loss: 0.7342 | g_loss: 1.7755
Epoch [   25/   50] | d_loss: 0.7748 | g_loss: 1.6542
Epoch [   25/   50] | d_loss: 0.7413 | g_loss: 2.1124
Epoch [   26/   50] | d_loss: 0.7631 | g_loss: 1.6215
Epoch [   26/   50] | d_loss: 0.7560 | g_loss: 1.4174
Epoch [   27/   50] | d_loss: 0.6732 | g_loss: 1.7030
Epoch [   27/   50] | d_loss: 0.9934 | g_loss: 0.9892
Epoch [   28/   50] | d_loss: 0.6655 | g_loss: 1.9294
Epoch [   28/   50] | d_loss: 0.8678 | g_loss: 1.5337
Epoch [   29/   50] | d_loss: 0.9499 | g_loss: 2.1283
Epoch [   29/   50] | d_loss: 0.7631 | g_loss: 1.5536
Epoch [   30/   50] | d_loss: 0.6240 | g_loss: 1.8341
Epoch [   30/   50] | d_loss: 0.6578 | g_loss: 1.7167
Epoch [   31/   50] | d_loss: 0.6784 | g_loss: 1.2038
```

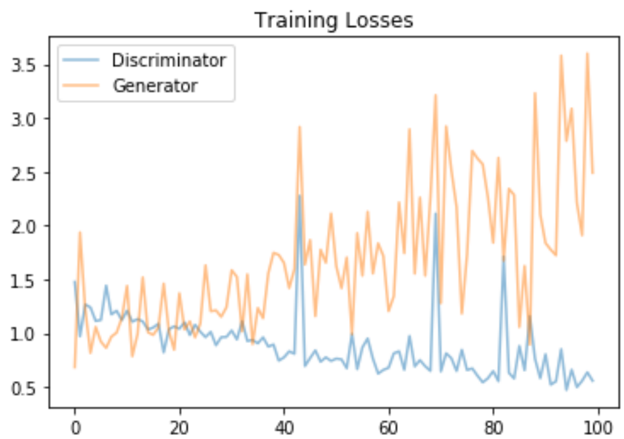
```
Epoch [   31/   50] | d_loss: 0.8120 | g_loss: 1.3430
Epoch [   32/   50] | d_loss: 0.8311 | g_loss: 2.2152
Epoch [   32/   50] | d_loss: 0.6590 | g_loss: 1.7422
Epoch [   33/   50] | d_loss: 0.9734 | g_loss: 2.8948
Epoch [   33/   50] | d_loss: 0.6905 | g_loss: 1.5526
Epoch [   34/   50] | d_loss: 0.7491 | g_loss: 2.2602
Epoch [   34/   50] | d_loss: 0.6919 | g_loss: 1.5334
Epoch [   35/   50] | d_loss: 0.6488 | g_loss: 2.2613
Epoch [   35/   50] | d_loss: 2.1106 | g_loss: 3.2115
Epoch [   36/   50] | d_loss: 0.6439 | g_loss: 1.2781
Epoch [   36/   50] | d_loss: 0.8127 | g_loss: 2.9217
Epoch [   37/   50] | d_loss: 0.7667 | g_loss: 2.5148
Epoch [   37/   50] | d_loss: 0.6480 | g_loss: 2.1704
Epoch [   38/   50] | d_loss: 0.8448 | g_loss: 1.1812
Epoch [   38/   50] | d_loss: 0.6568 | g_loss: 1.7281
Epoch [   39/   50] | d_loss: 0.6720 | g_loss: 2.6937
Epoch [   39/   50] | d_loss: 0.6035 | g_loss: 2.6228
Epoch [   40/   50] | d_loss: 0.5423 | g_loss: 2.5694
Epoch [   40/   50] | d_loss: 0.5811 | g_loss: 2.2582
Epoch [   41/   50] | d_loss: 0.6466 | g_loss: 1.8398
Epoch [   41/   50] | d_loss: 0.5533 | g_loss: 2.6295
Epoch [   42/   50] | d_loss: 1.7122 | g_loss: 1.6727
Epoch [   42/   50] | d_loss: 0.6318 | g_loss: 2.3411
Epoch [   43/   50] | d_loss: 0.5780 | g_loss: 2.2840
Epoch [   43/   50] | d_loss: 0.8808 | g_loss: 1.0563
Epoch [   44/   50] | d_loss: 0.6550 | g_loss: 1.6244
Epoch [   44/   50] | d_loss: 1.1575 | g_loss: 0.8923
Epoch [   45/   50] | d_loss: 0.7601 | g_loss: 3.2301
Epoch [   45/   50] | d_loss: 0.5807 | g_loss: 2.1021
Epoch [   46/   50] | d_loss: 0.8037 | g_loss: 1.8370
Epoch [   46/   50] | d_loss: 0.5209 | g_loss: 1.7758
Epoch [   47/   50] | d_loss: 0.5519 | g_loss: 1.7228
Epoch [   47/   50] | d_loss: 0.8509 | g_loss: 3.5799
Epoch [   48/   50] | d_loss: 0.4717 | g_loss: 2.7856
Epoch [   48/   50] | d_loss: 0.6597 | g_loss: 3.0861
Epoch [   49/   50] | d_loss: 0.4977 | g_loss: 2.2116
Epoch [   49/   50] | d_loss: 0.5570 | g_loss: 1.9090
Epoch [   50/   50] | d_loss: 0.6357 | g_loss: 3.5995
Epoch [   50/   50] | d_loss: 0.5572 | g_loss: 2.4899
```

Training loss

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

In [20]:
```
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[20]: <matplotlib.legend.Legend at 0x7ff7be2bffd0>



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 [21]:
```
# 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 [22]:
```
# Load samples from generator, taken while training
with open('train_samples.pkl', 'rb') as f:
    samples = pkl.load(f)
```

In [23]:
```
_ = view_samples(-1, samples)
```

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: (Write your answer in this cell)

The generated samples created realistic-looking human faces, however, some of the generated faces don't have very high quality. There are a few ways we can further improve the model:

1. since the dataset is biased with mostly white faces, the generated faces are mostly white. To overcome this, we need to introduce more training samples of other races.
2. To improve model performance, we may consider using larger/deeper models to learn more features.
3. The parameters for optimizers given by the paper is lr = 0.0002 and beta1=0.5, but based on experiments, the learning rate and beta1 need to reduce further to prevent instability. Also, the number of epoches makes a big difference on the model. From the Training loss VS epochs plot, we can see that the if early-stopping is chosen at epoch 44, the model could avoid overfitting and perform better.

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.