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.

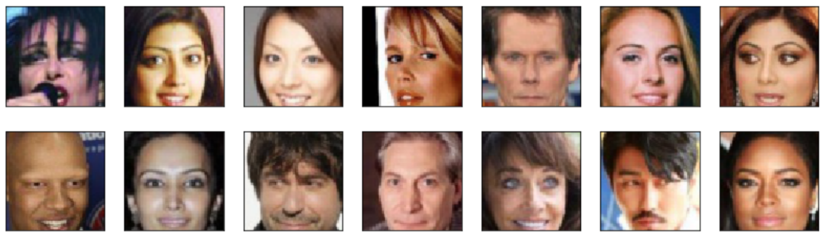
Get the Data

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]:
```
data_dir = '/floyd/input/celeba'

"""
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 [2]:
```
# necessary imports
import torch
from torchvision import datasets
from torchvision import transforms
```

In [3]:
```
def get_dataloader(batch_size, image_size, data_dir=data_dir):
    """
    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
    """
    transform = transforms.Compose([
        transforms.Resize(image_size),
        transforms.ToTensor()
    ])

    images = datasets.ImageFolder(data_dir, transform=transform)

    return torch.utils.data.DataLoader(images, batch_size=batch_size, sh
uffle=True)
```

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 [4]:
```
# Define function hyperparameters
batch_size = 128
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)
```

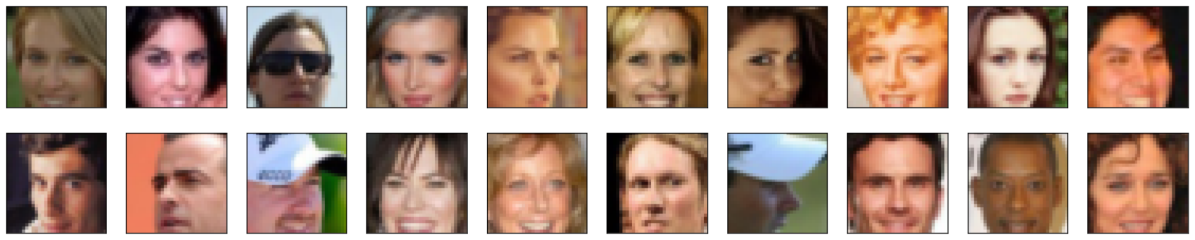
Next, you can view some images! You should seen square images of somewhat-centered faces.

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

    return np.interp(x, (0, 1), feature_range) # https://codereview.stac
kexchange.com/a/185794
```

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:  -0.9294117614626884
Max:  0.32549023628234863
```

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

Disclaimer

The following implementation was almost entirely inspired by the DCGAN implementation found in the DLND github repo at https://github.com/udacity/deep-learning-v2-pytorch/tree/master/dcgan-svhn (https://github.com/udacity/deep-learning-v2-pytorch/tree/master/dcgan-svhn) .

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

# helper conv function
def conv(in_channels, out_channels, kernel_size, stride=2, padding=1, ba
tch_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 [9]:
```
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

        # 32x32 input
        self.conv1 = conv(3, conv_dim, 4, batch_norm=False) # first laye
r, no batch_norm
        # 16x16 out
        self.conv2 = conv(conv_dim, conv_dim*2, 4)
        # 8x8 out
        self.conv3 = conv(conv_dim*2, conv_dim*4, 4)
        # 4x4 out

        # final, fully-connected layer
        self.fc = nn.Linear(conv_dim*4*4*4, 1)

    def forward(self, x):
        # all hidden layers + leaky relu activation
        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
        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 [10]:
```
# helper deconv function
def deconv(in_channels, out_channels, kernel_size, stride=2, padding=1, batch_norm=True):
    """Creates a transposed-convolutional layer, with optional batch normalization.
    """
    # create a sequence of transpose + optional batch norm layers
    layers = []
    transpose_conv_layer = nn.ConvTranspose2d(in_channels, out_channels,
                                              kernel_size, stride, padding, bias=False)
    # 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)
```

```
In [11]: class Generator(nn.Module):

             def __init__(self, z_size, conv_dim=32):
                 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, 4)
                 self.t_conv2 = deconv(conv_dim*2, conv_dim, 4)
                 self.t_conv3 = deconv(conv_dim, 3, 4, batch_norm=False)


             def forward(self, x):
                 if type(x) == 'numpy.ndarray':
                     x = torch.from_numpy(x)

                 # 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 = torch.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]:
```
# inspired by https://stackoverflow.com/a/55546528
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 classname == 'Linear' or 'Conv' in classname:
        # m.weight.data shoud be taken from a normal distribution
        m.weight.data.normal_(0.0,0.02)
```

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]:
```python
# Define model hyperparams
d_conv_dim = 10
g_conv_dim = 10
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, 10, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
  )
  (conv2): Sequential(
    (0): Conv2d(10, 20, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
  )
  (conv3): Sequential(
    (0): Conv2d(20, 40, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
  )
  (fc): Linear(in_features=640, out_features=1, bias=True)
)

Generator(
  (fc): Linear(in_features=100, out_features=640, bias=True)
  (t_conv1): Sequential(
    (0): ConvTranspose2d(40, 20, kernel_size=(4, 4), stride=(2, 2), pad
ding=(1, 1), bias=False)
    (1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
  )
  (t_conv2): Sequential(
    (0): ConvTranspose2d(20, 10, kernel_size=(4, 4), stride=(2, 2), pad
ding=(1, 1), bias=False)
    (1): BatchNorm2d(10, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
  )
  (t_conv3): Sequential(
    (0): ConvTranspose2d(10, 3, kernel_size=(4, 4), stride=(2, 2), padd
ing=(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, smooth=False):
    batch_size = D_out.size(0)
    # label smoothing
    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):
    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 [54]:
```
import torch.optim as optim

# params
lr = 0.0002
beta1=0.5
beta2=0.999 # default value

# Create optimizers for the discriminator and generator
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 [55]: 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
          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 = torch.from_numpy(scale(real_images))
                     real_images = real_images.type(torch.FloatTensor)

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

                     # Train Discriminator
                     d_optimizer.zero_grad()

                     # 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)

                     # 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()


            # Train Generator
            g_optimizer.zero_grad()

            # 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}'.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 chan
ge 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)

    # finally return losses
    return losses
```

Set your number of training epochs and train your GAN!

In [56]:
```
# set number of epochs
n_epochs = 6

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

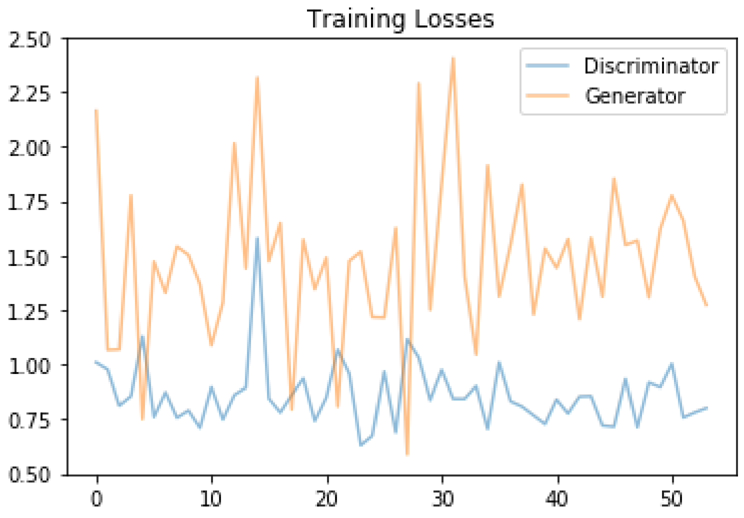
```
Epoch [    1/    6] | d_loss: 1.0095 | g_loss: 2.1635
Epoch [    1/    6] | d_loss: 0.9769 | g_loss: 1.0663
Epoch [    1/    6] | d_loss: 0.8129 | g_loss: 1.0685
Epoch [    1/    6] | d_loss: 0.8557 | g_loss: 1.7768
Epoch [    1/    6] | d_loss: 1.1283 | g_loss: 0.7511
Epoch [    1/    6] | d_loss: 0.7606 | g_loss: 1.4724
Epoch [    1/    6] | d_loss: 0.8729 | g_loss: 1.3290
Epoch [    1/    6] | d_loss: 0.7571 | g_loss: 1.5416
Epoch [    1/    6] | d_loss: 0.7902 | g_loss: 1.5026
Epoch [    2/    6] | d_loss: 0.7105 | g_loss: 1.3681
Epoch [    2/    6] | d_loss: 0.8965 | g_loss: 1.0875
Epoch [    2/    6] | d_loss: 0.7489 | g_loss: 1.2838
Epoch [    2/    6] | d_loss: 0.8607 | g_loss: 2.0159
Epoch [    2/    6] | d_loss: 0.8939 | g_loss: 1.4407
Epoch [    2/    6] | d_loss: 1.5801 | g_loss: 2.3160
Epoch [    2/    6] | d_loss: 0.8448 | g_loss: 1.4741
Epoch [    2/    6] | d_loss: 0.7810 | g_loss: 1.6496
Epoch [    2/    6] | d_loss: 0.8615 | g_loss: 0.7951
Epoch [    3/    6] | d_loss: 0.9357 | g_loss: 1.5732
Epoch [    3/    6] | d_loss: 0.7425 | g_loss: 1.3454
Epoch [    3/    6] | d_loss: 0.8492 | g_loss: 1.4902
Epoch [    3/    6] | d_loss: 1.0675 | g_loss: 0.8089
Epoch [    3/    6] | d_loss: 0.9575 | g_loss: 1.4745
Epoch [    3/    6] | d_loss: 0.6302 | g_loss: 1.5183
Epoch [    3/    6] | d_loss: 0.6742 | g_loss: 1.2178
Epoch [    3/    6] | d_loss: 0.9669 | g_loss: 1.2151
Epoch [    3/    6] | d_loss: 0.6897 | g_loss: 1.6268
Epoch [    4/    6] | d_loss: 1.1156 | g_loss: 0.5888
Epoch [    4/    6] | d_loss: 1.0313 | g_loss: 2.2893
Epoch [    4/    6] | d_loss: 0.8374 | g_loss: 1.2501
Epoch [    4/    6] | d_loss: 0.9747 | g_loss: 1.8330
Epoch [    4/    6] | d_loss: 0.8437 | g_loss: 2.4058
Epoch [    4/    6] | d_loss: 0.8434 | g_loss: 1.4031
Epoch [    4/    6] | d_loss: 0.9013 | g_loss: 1.0453
Epoch [    4/    6] | d_loss: 0.7054 | g_loss: 1.9142
Epoch [    4/    6] | d_loss: 1.0093 | g_loss: 1.3118
Epoch [    5/    6] | d_loss: 0.8323 | g_loss: 1.5512
Epoch [    5/    6] | d_loss: 0.8084 | g_loss: 1.8262
Epoch [    5/    6] | d_loss: 0.7681 | g_loss: 1.2290
Epoch [    5/    6] | d_loss: 0.7282 | g_loss: 1.5321
Epoch [    5/    6] | d_loss: 0.8405 | g_loss: 1.4434
Epoch [    5/    6] | d_loss: 0.7765 | g_loss: 1.5757
Epoch [    5/    6] | d_loss: 0.8542 | g_loss: 1.2072
Epoch [    5/    6] | d_loss: 0.8560 | g_loss: 1.5819
Epoch [    5/    6] | d_loss: 0.7216 | g_loss: 1.3113
Epoch [    6/    6] | d_loss: 0.7161 | g_loss: 1.8537
Epoch [    6/    6] | d_loss: 0.9336 | g_loss: 1.5483
Epoch [    6/    6] | d_loss: 0.7132 | g_loss: 1.5690
Epoch [    6/    6] | d_loss: 0.9171 | g_loss: 1.3081
Epoch [    6/    6] | d_loss: 0.8956 | g_loss: 1.6225
Epoch [    6/    6] | d_loss: 1.0031 | g_loss: 1.7759
Epoch [    6/    6] | d_loss: 0.7582 | g_loss: 1.6610
Epoch [    6/    6] | d_loss: 0.7815 | g_loss: 1.4036
Epoch [    6/    6] | d_loss: 0.8012 | g_loss: 1.2729
```

Training loss

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

In [57]:
```
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[57]:
```
<matplotlib.legend.Legend at 0x7f81d077ffd0>
```



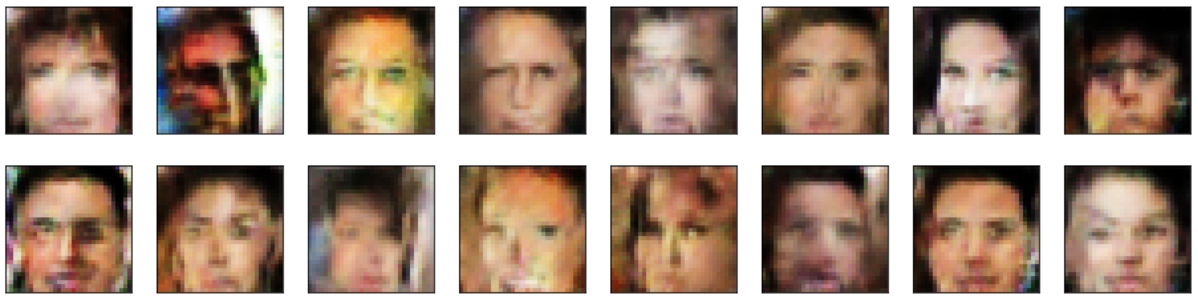
Generator samples from training

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

In [58]:
```
# 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=Tr
ue, 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 [59]:
```
# Load samples from generator, taken while training
with open('train_samples.pkl', 'rb') as f:
    samples = pkl.load(f)
```

In [60]:
```
_ = 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)

It took several attempts trying different hyperparameter values before obtaining interesting examples. Nevertheless, the generated images are more abstract compared to the original images used for training. The generated images look like abstract paintings.

- Improving the quality of these images -

We could potentially improve the quality of these images with the following strategies:

```
1. Increasing the number of epochs
2. Trying different learning rates, beta 1, and beta 2
3. Increasing the model size
4. Trying different model architecture than DCGAN, such as [BigGAN](https://
towardsdatascience.com/must-read-papers-on-gans-b665bbae3317)
5. Testing different activation functions for the neural network layers.
6. Using high-definition images
```

- Making the generated images more representative of the general population -

To make the generated images more representative of a bigger spectrum of the general population, we could have augmented the dataset with pictures of "regular" people (as opposed to celebrities), people of different ethnicities, people of different age groups, people with face handicaps/deformations/skin conditions, people using different facial expressions, faces taken at different angles, etc.

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