dlnd_face_generation

May 14, 2020

1 Face Generation

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

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

1.0.1 Get the Data

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

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

1.0.2 Pre-processed Data

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

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

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

```
In [73]: # can comment out after executing
         #!unzip processed_celeba_small.zip
```

```
In [74]: data_dir = 'processed_celeba_small/'

         """
         DON'T MODIFY ANYTHING IN THIS CELL
         """
         import pickle as pkl
         from PIL import Image, ImageFile
```

```
import matplotlib.pyplot as plt
import numpy as np
import problem_unittests as tests
#import helper
from workspace_utils import active_session

ImageFile.LOAD_TRUNCATED_IMAGES = True
%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 [75]:
```
# necessary imports
import torch
from torchvision import datasets
from torchvision import transforms
```

In [76]:
```
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
    img_transform = transforms.Compose([transforms.Resize(image_size),
                                        transforms.CenterCrop(image_size),
                                        transforms.ToTensor()])

    img_data = datasets.ImageFolder(data_dir, transform=img_transform)
    data_loader = torch.utils.data.DataLoader(img_data,
                                              batch_size=batch_size + 1,
                                              shuffle=True)

    return data_loader
```

1.2 Create a DataLoader

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

In [77]:
```
# Define function hyperparameters
batch_size = 100
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.

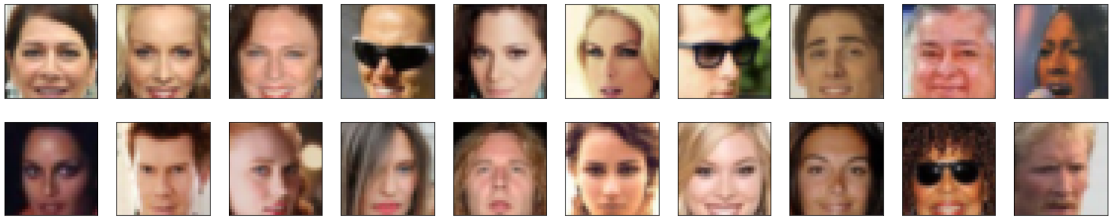
Note: You'll need to convert the Tensor images into a NumPy type and transpose the dimensions to correctly display an image, suggested `imshow` code is below, but it may not be perfect.

In [78]:
```
# 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 [79]: # 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
             return x * (max - min) + min
```

```
In [80]: """
         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(-0.8667)
Max:  tensor(0.9922)
```

---

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 [81]:
```
import torch.nn as nn
import torch.nn.functional as F
```

In [82]:
```
# Convolution Helper function
def conv(in_channels, out_channels, kernel_size, padding=1, stride=2,  batch_norm=True)
    """
    Creates a convolutional layer (batch normalization: optional)
    """
    layers = []
    conv_layer = nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                           kernel_size=kernel_size, stride=stride, padding=padding, bia

    layers.append(conv_layer)

    if batch_norm:
        layers.append(nn.BatchNorm2d(out_channels))

    return nn.Sequential(*layers)
```

In [83]:
```
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

        # Define all convolutional layers
        # Input: RGB Input  and output a single value
        self.conv1 = conv(3, conv_dim, 4, batch_norm= False)
        self.conv2 = conv(conv_dim    , conv_dim * 2, 4)
        self.conv3 = conv(conv_dim * 2, conv_dim * 4, 4)
        self.conv4 = conv(conv_dim * 4, conv_dim * 8, 4)
```

```
        self.fc = nn.Linear(conv_dim*8*2*2, 1)
        self.dropout = nn.Dropout(0.4)



    def forward(self, x):
        """
        Forward propagation of the neural network
        :param x: The input to the neural network
        :return: Discriminator logits; the output of the neural network
        """
        # define feedforward behavior

        x = F.leaky_relu(self.conv1(x), 0.15)
        x = F.leaky_relu(self.conv2(x), 0.15)
        x = F.leaky_relu(self.conv3(x), 0.15)
        x = F.leaky_relu(self.conv4(x), 0.15)

        x = x.view(-1, self.conv_dim*8*2*2)
        x = self.fc(x)
        x = self.dropout(x)

        return x


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

```
Tests Passed
```

2.2 Generator

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

Exercise: Complete the Generator class

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

In [84]:
```
# Deconvolution helper function
def deconv(in_channels, out_channels, kernel_size, stride=2, padding=1, batch_norm=True
```

```
    """
    Creates a transpose convolutional layer (batch normalization: optional)
    """
    layers = []

    layers.append(nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, pa

    if batch_norm:
        layers.append(nn.BatchNorm2d(out_channels))

    return nn.Sequential(*layers)
```

In [85]:
```
class Generator(nn.Module):

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

        # complete init function
        self.conv_dim = conv_dim

        self.fc = nn.Linear(z_size, conv_dim*8*2*2)

        self.t_conv1 = deconv(conv_dim*8, conv_dim*4, 4)
        self.t_conv2 = deconv(conv_dim*4, conv_dim*2, 4)
        self.t_conv3 = deconv(conv_dim*2, conv_dim, 4)
        self.t_conv4 = deconv(conv_dim, 3, 4, batch_norm=False)
        self.dropout = nn.Dropout(0.4)


    def forward(self, x):
        """
        Forward propagation of the neural network
        :param x: The input to the neural network
        :return: A 32x32x3 Tensor image as output
        """
        # define feedforward behavior
        x = self.fc(x)
        x = self.dropout(x)
        x = x.view(-1, self.conv_dim*8, 2, 2)

        x = F.relu(self.t_conv1(x))
        x = F.relu(self.t_conv2(x))
        x = F.relu(self.t_conv3(x))
```

```
        x = F.tanh(self.t_conv4(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 [86]: 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

             isLinear = classname.find('Linear') != -1
             isConv = classname.find('Conv') != -1

             if (isLinear or isConv):
                 init.normal_(m.weight.data, 0.0, 0.02)
```

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 [87]:
```
"""
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 [88]:
```
# Define model hyperparams
d_conv_dim = 64
g_conv_dim = 64
z_size = 100

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

```
Discriminator(
  (conv1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
  (conv2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (conv3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (conv4): Sequential(
    (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
```

```
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (fc): Linear(in_features=2048, out_features=1, bias=True)
  (dropout): Dropout(p=0.4)
)

Generator(
  (fc): Linear(in_features=100, out_features=2048, bias=True)
  (t_conv1): Sequential(
    (0): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (t_conv2): Sequential(
    (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (t_conv3): Sequential(
    (0): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (t_conv4): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
  (dropout): Dropout(p=0.4)
)
```

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 [89]:
```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 [90]:
```
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)
    labels = torch.ones(batch_size)


    if train_on_gpu:
        labels = labels.cuda()

    criteria = nn.BCEWithLogitsLoss()

    loss = criteria(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)

    if train_on_gpu:
        labels = labels.cuda()
```

```
        criteria = nn.BCEWithLogitsLoss()


        loss = criteria(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 [91]: import torch.optim as optim

         lr = 0.0001

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

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 [92]: def train(D, G, n_epochs, print_every=150):
             '''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 = scale(real_images)

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

        ###### 1. Train the discriminator on real and fake images   ######

        if train_on_gpu:
            real_images = real_images.cuda()

        d_optimizer.zero_grad()
        d_out = D(real_images)
        d_real_loss = real_loss(d_out)


        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)     # Generating fake images

        d_fake_out = D(fake_images)
        d_fake_loss = fake_loss(d_fake_out)
```

```python
            d_loss = d_real_loss + d_fake_loss   # Discriminator loss

            d_loss.backward()   #Back-propagating
            d_optimizer.step()

            ###### 2. Train the generator with an adversarial loss     ######

            g_optimizer.zero_grad()
            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)    # Generating fake images

            g_fake_out = D(fake_images)
            g_loss = real_loss(g_fake_out) # Generator Loss

            g_loss.backward()    # Back-propagating
            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)

    # finally return losses
```

```
        return losses
```

Set your number of training epochs and train your GAN!

```
In [93]: # set number of epochs
         n_epochs = 10


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

```
Epoch [    1/   10] | d_loss: 1.5996 | g_loss: 0.9336
Epoch [    1/   10] | d_loss: 0.6044 | g_loss: 4.7014
Epoch [    1/   10] | d_loss: 0.4809 | g_loss: 5.1123
Epoch [    1/   10] | d_loss: 0.5723 | g_loss: 4.9083
Epoch [    1/   10] | d_loss: 0.5798 | g_loss: 4.9697
Epoch [    1/   10] | d_loss: 0.5188 | g_loss: 5.1604
Epoch [    2/   10] | d_loss: 0.5692 | g_loss: 5.4608
Epoch [    2/   10] | d_loss: 0.5201 | g_loss: 5.7189
Epoch [    2/   10] | d_loss: 0.6013 | g_loss: 5.9625
Epoch [    2/   10] | d_loss: 0.5886 | g_loss: 3.8452
Epoch [    2/   10] | d_loss: 0.6922 | g_loss: 4.7416
Epoch [    2/   10] | d_loss: 0.6388 | g_loss: 4.6381
Epoch [    3/   10] | d_loss: 0.5516 | g_loss: 3.6414
Epoch [    3/   10] | d_loss: 0.6161 | g_loss: 4.3235
Epoch [    3/   10] | d_loss: 0.5950 | g_loss: 5.7169
Epoch [    3/   10] | d_loss: 0.5938 | g_loss: 4.6465
Epoch [    3/   10] | d_loss: 0.7016 | g_loss: 4.6146
Epoch [    3/   10] | d_loss: 1.2648 | g_loss: 4.7249
Epoch [    4/   10] | d_loss: 0.5750 | g_loss: 5.4363
Epoch [    4/   10] | d_loss: 0.7077 | g_loss: 3.8382
Epoch [    4/   10] | d_loss: 0.5308 | g_loss: 5.4808
Epoch [    4/   10] | d_loss: 0.6384 | g_loss: 7.1999
Epoch [    4/   10] | d_loss: 0.6925 | g_loss: 2.9453
Epoch [    4/   10] | d_loss: 0.6600 | g_loss: 3.4061
Epoch [    5/   10] | d_loss: 0.5535 | g_loss: 3.3218
Epoch [    5/   10] | d_loss: 0.5022 | g_loss: 5.8987
Epoch [    5/   10] | d_loss: 0.5233 | g_loss: 6.4770
Epoch [    5/   10] | d_loss: 0.5286 | g_loss: 4.8444
Epoch [    5/   10] | d_loss: 0.6138 | g_loss: 4.0087
Epoch [    5/   10] | d_loss: 0.6961 | g_loss: 3.9259
Epoch [    6/   10] | d_loss: 0.5359 | g_loss: 3.0906
Epoch [    6/   10] | d_loss: 0.6369 | g_loss: 9.9766
Epoch [    6/   10] | d_loss: 0.5399 | g_loss: 3.9559
```

```
Epoch [    6/   10] | d_loss: 0.5235 | g_loss: 5.4955
Epoch [    6/   10] | d_loss: 0.6078 | g_loss: 3.6102
Epoch [    6/   10] | d_loss: 0.4359 | g_loss: 7.0673
Epoch [    7/   10] | d_loss: 0.7824 | g_loss: 4.1064
Epoch [    7/   10] | d_loss: 0.5682 | g_loss: 4.4292
Epoch [    7/   10] | d_loss: 0.6627 | g_loss: 5.0312
Epoch [    7/   10] | d_loss: 0.5867 | g_loss: 6.8107
Epoch [    7/   10] | d_loss: 0.6366 | g_loss: 3.1957
Epoch [    7/   10] | d_loss: 0.5254 | g_loss: 5.6437
Epoch [    8/   10] | d_loss: 0.5803 | g_loss: 2.9363
Epoch [    8/   10] | d_loss: 0.5730 | g_loss: 3.7927
Epoch [    8/   10] | d_loss: 0.6189 | g_loss: 6.6726
Epoch [    8/   10] | d_loss: 0.5907 | g_loss: 4.8084
Epoch [    8/   10] | d_loss: 0.7560 | g_loss: 3.1475
Epoch [    8/   10] | d_loss: 0.5811 | g_loss: 8.6553
Epoch [    9/   10] | d_loss: 0.5558 | g_loss: 6.3670
Epoch [    9/   10] | d_loss: 0.6252 | g_loss: 7.0374
Epoch [    9/   10] | d_loss: 0.6037 | g_loss: 3.5393
Epoch [    9/   10] | d_loss: 0.5150 | g_loss: 10.2220
Epoch [    9/   10] | d_loss: 0.7053 | g_loss: 4.9219
Epoch [    9/   10] | d_loss: 0.5620 | g_loss: 3.8978
Epoch [   10/   10] | d_loss: 0.6818 | g_loss: 6.2311
Epoch [   10/   10] | d_loss: 0.5937 | g_loss: 3.3589
Epoch [   10/   10] | d_loss: 0.5432 | g_loss: 6.3465
Epoch [   10/   10] | d_loss: 0.6184 | g_loss: 6.5758
Epoch [   10/   10] | d_loss: 0.6314 | g_loss: 4.3630
Epoch [   10/   10] | d_loss: 0.7130 | g_loss: 5.1584
```

2.8 Training loss

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

```
In [94]: 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[94]: <matplotlib.legend.Legend at 0x7fa608e056d8>
```

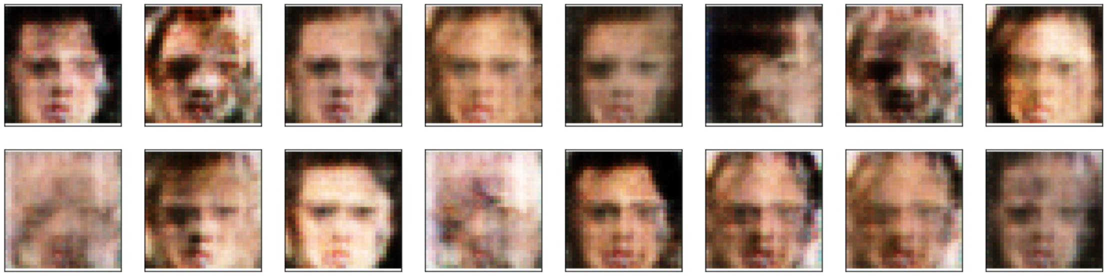
2.9 Generator samples from training

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

```
In [95]: # 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 [96]: # Load samples from generator, taken while training
         with open('train_samples.pkl', 'rb') as f:
             samples = pkl.load(f)
```

```
In [97]: _ = view_samples(-1, samples)
```



In [ ]:

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:

- The generated images mostly depict white individuals without a sense of generality. This is mostly due to the skewness in the dataset which contain celebrity faces which are mostly white.
- I have tried models of different sizes with d_conv_dim = (32/64/128). The larger models took longer to converge to a stable loss (in proportion to the increase in size) with the d_conv_dim=128 model taking approximately 22 epochs to reach the minimum loss, after which it started increasing. In order to maintain a trade-off between generating accurate images and doing so in a reasonable time period, I chose to use d_conv_dim=64.
- I discovered that running the training for larger number of epochs (>20) did not improve performance. Moreover, optimizers such as SGD proved to be inefficient. Hence, I chose the Adam optimizer with a learning rate of 0.0001.

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.