

# dlnd\_face\_generation

January 7, 2021

## 1 Face Generation

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

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

### 1.0.1 Get the Data

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

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

### 1.0.2 Pre-processed Data

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

If you are working locally, you can download this data [by clicking here](#)

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

```
In [1]: # Connect to Google Drive on Google Colab
        from google.colab import drive
        drive.mount('/content/gdrive')
```

Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/con

```
In [2]: import os
        print(os.getcwd())
        NEW_PATH = "/content/gdrive/MyDrive/DLND/Face Generation"
```

```

os.chdir(NEW_PATH)
print(os.getcwd())
import sys
sys.path.append("/content/gdrive/MyDrive/DLND/Face Generation")
%cd /content/gdrive/MyDrive/DLND/Face Generation
!ls

/content
/content/gdrive/MyDrive/DLND/Face Generation
/content
dlnd_face_generation.ipynb  processed_celeba_small.zip
problem_unittests.py      __pycache__

```

```

In [3]: # can comment out after executing
!unzip processed_celeba_small.zip

```

Streaming output truncated to the last 5000 lines.

```

inflating: processed_celeba_small/celeba/New Folder With Items/052332.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052333.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052334.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052335.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052336.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052337.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052338.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052339.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052340.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052341.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052342.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052343.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052344.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052345.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052346.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052347.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052348.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052349.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052350.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052351.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052352.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052353.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052354.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052355.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052356.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052357.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052358.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052359.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052360.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/052361.jpg

```

[illegible]

[illegible]







[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]



[illegible]





[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]







[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]





[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]



[illegible]

[illegible]





[illegible]





```
inflating: processed_celeba_small/celeba/New Folder With Items/057306.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057307.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057308.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057309.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057310.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057311.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057312.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057313.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057314.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057315.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057316.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057317.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057318.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057319.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057320.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057321.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057322.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057323.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057324.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057325.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057326.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057327.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057328.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057329.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057330.jpg
inflating: processed_celeba_small/celeba/New Folder With Items/057331.jpg
```

```
In [4]: data_dir = 'processed_celeba_small/'
```

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""

import pickle as pkl
import matplotlib.pyplot as plt
import numpy as np
import problem_unittests as tests
#import helper

%matplotlib inline
```

## 1.1 Visualize the CelebA Data

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

### 1.1.1 Pre-process and Load the Data

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

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

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

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

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

```
In [5]: # necessary imports
import torch
from torchvision import datasets
from torchvision import transforms

In [6]: 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 = transforms.Compose([
        transforms.Resize(image_size),
        transforms.CenterCrop(image_size),
        transforms.ToTensor()
    ])

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

    # create and return DataLoaders
    data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                              batch_size=batch_size,
                                              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 [7]: # Define function hyperparameters
        batch_size = 64
        img_size = 32

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

        # Call your function and get a dataloader
        celeba_train_loader = get_dataloader(batch_size, img_size)
```

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

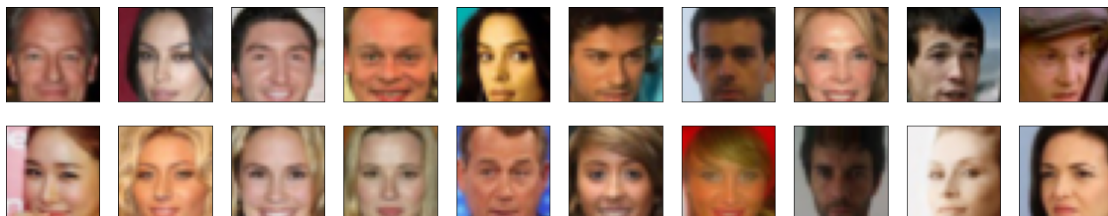
Note: You'll need to convert the Tensor images into a NumPy type and transpose the dimensions to correctly display an image, suggested `imshow` code is below, but it may not be perfect.

```
In [8]: # 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 [9]: # TODO: Complete the scale function
def scale(x, feature_range=(-1, 1)):
    ''' Scale takes in an image x and returns that image, scaled
        with a feature_range of pixel values from -1 to 1.
        This function assumes that the input x is already scaled from 0-1. '''
    # assume x is scaled to (0, 1)
    # scale to feature_range and return scaled x

    min, max = feature_range
    x = x * (max - min) + min
    return x
```

```
In [10]: """
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""

# check scaled range
# should be close to -1 to 1
img = images[0]
scaled_img = scale(img)

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

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

---

## 2 Define the Model

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

### 2.1 Discriminator

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

**Exercise: Complete the Discriminator class**

- The inputs to the discriminator are 32x32x3 tensor images

- The output should be a single value that will indicate whether a given image is real or fake

```
In [11]: import torch.nn as nn
         import torch.nn.functional as F
```

```
In [12]: 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 = nn.Conv2d(3, conv_dim, kernel_size=4, stride=2, padding=1, bias=False)
        self.batch_norm1 = nn.BatchNorm2d(conv_dim)
        self.conv2 = nn.Conv2d(conv_dim, conv_dim*2, kernel_size=4, stride=2, padding=1, bias=False)
        self.batch_norm2 = nn.BatchNorm2d(conv_dim*2)
        self.conv3 = nn.Conv2d(conv_dim*2, conv_dim*4, kernel_size=4, stride=2, padding=1, bias=False)
        self.batch_norm3 = nn.BatchNorm2d(conv_dim*4)
        self.conv4 = nn.Conv2d(conv_dim*4, conv_dim*8, kernel_size=4, stride=2, padding=1, bias=False)
        self.batch_norm4 = nn.BatchNorm2d(conv_dim*8)
        self.conv5 = nn.Conv2d(conv_dim*8, conv_dim*16, kernel_size=4, stride=2, padding=1, bias=False)
        self.fc = nn.Linear(conv_dim*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
        x = F.leaky_relu(self.batch_norm1(self.conv1(x)), 0.2)
        x = F.leaky_relu(self.batch_norm2(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.batch_norm3(self.conv3(x)), 0.2)
        x = F.leaky_relu(self.batch_norm4(self.conv4(x)), 0.2)
        x = self.conv5(x)
        # flatten
        x = x.view(-1, self.conv_dim*4*4)
        # final output layer
        x = F.sigmoid(self.fc(x))

        return x
```

```

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

```

Tests Passed

```

/usr/local/lib/python3.6/dist-packages/torch/nn/functional.py:1639: UserWarning: nn.functional.sigmoid is deprecated. Use torch.sigmoid instead.
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid instead.")

```

## 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 [13]: # 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)

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

# 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)

# dropout layer
self.dropout = nn.Dropout(0.5)

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

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

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

    return x

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

tests.test_generator(Generator)

```

Tests Passed

```

/usr/local/lib/python3.6/dist-packages/torch/nn/functional.py:1628: UserWarning: nn.functional.tanh
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")

```



## 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 [14]: 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.find('Conv') != -1 or classname.find('Linear') != -1:
             nn.init.normal_(m.weight.data, 0, 0.02)
         if hasattr(m, 'bias') and m.bias is not None:
             m.bias.data.fill_(0)
```

## 2.4 Build complete network

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

```
In [15]: """
         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 [16]: *# 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): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv4): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv5): Conv2d(512, 1024, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (fc): Linear(in_features=1024, out_features=1, bias=True)

```

)

Generator(

```

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

```

)

### 2.4.1 Training on GPU

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

Are moved to GPU, where appropriate.

```
In [17]: """
         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 [18]: def real_loss(D_out, smooth=False):
         '''Calculates how close discriminator outputs are to being real.
         param, D_out: discriminator logits
         return: real loss'''
         batch_size = D_out.size(0)
         if smooth:
```

```

        labels = torch.ones(batch_size)*0.9
    else:
        labels = torch.ones(batch_size)
    if train_on_gpu:
        labels = labels.cuda()
    criterion = nn.BCELoss()
    # 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)
    if train_on_gpu:
        labels = labels.cuda()
    criterion = nn.BCELoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)
    return loss

```

## 2.6 Optimizers

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

In [19]: `import torch.optim as optim`

```

# Create optimizers for the discriminator D and generator G
lr=0.0003
g_optimizer = optim.Adam(G.parameters(), lr=lr, betas=(0.3, 0.999))
d_optimizer = optim.Adam(D.parameters(), lr=lr, betas=(0.3, 0.999))

```

---

## 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 [20]: 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 = scale(real_images)

            # =====
            #          YOUR CODE HERE: TRAIN THE NETWORKS
            # =====
            d_optimizer.zero_grad()

            # 1. Train the discriminator on real and fake images
            if train_on_gpu:
                real_images = real_images.cuda()
            D_real = D(real_images)
            d_real_loss = real_loss(D_real)
```

```

# generate fake images
z = np.random.uniform(-1, 1, size=(batch_size, z_size))
z = torch.from_numpy(z).float()
if train_on_gpu:
    z = z.cuda()
fake_images = G(z)
D_fake = D(fake_images)
d_fake_loss = fake_loss(D_fake)

# add real loss and fake loss
d_loss = d_real_loss + d_fake_loss
d_loss.backward()
d_optimizer.step()

# 2. Train the generator with an adversarial loss
g_optimizer.zero_grad()
# generate fake images
z = np.random.uniform(-1, 1, size=(batch_size, z_size))
z = torch.from_numpy(z).float()
if train_on_gpu:
    z = z.cuda()
fake_images = G(z)
# compute the discriminator losses on fake images
D_fake = D(fake_images)
# use real loss to flip labels
g_loss = real_loss(D_fake)
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: {:.4f} | g_loss: {:.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:
        pickle.dump(samples, f)

    # finally return losses
    return losses

```

Set your number of training epochs and train your GAN!

```

In [21]: # set number of epochs
         n_epochs = 9

```

```

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

```

```

/usr/local/lib/python3.6/dist-packages/torch/nn/functional.py:1639: UserWarning: nn.functional.sigmoid
warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid instead.")

```

```

/usr/local/lib/python3.6/dist-packages/torch/nn/functional.py:1628: UserWarning: nn.functional.tanh
warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")

```

```

Epoch [ 1/ 9] | d_loss: 1.3869 | g_loss: 1.5864
Epoch [ 1/ 9] | d_loss: 1.7090 | g_loss: 13.1677
Epoch [ 1/ 9] | d_loss: 1.5223 | g_loss: 0.0443
Epoch [ 1/ 9] | d_loss: 0.4208 | g_loss: 1.7222
Epoch [ 1/ 9] | d_loss: 2.4648 | g_loss: 0.3329
Epoch [ 1/ 9] | d_loss: 0.3307 | g_loss: 3.1481
Epoch [ 1/ 9] | d_loss: 0.4144 | g_loss: 2.3285
Epoch [ 1/ 9] | d_loss: 0.6115 | g_loss: 1.2974
Epoch [ 1/ 9] | d_loss: 1.3620 | g_loss: 0.2456
Epoch [ 1/ 9] | d_loss: 1.1359 | g_loss: 4.1940
Epoch [ 1/ 9] | d_loss: 0.5322 | g_loss: 2.2054
Epoch [ 1/ 9] | d_loss: 0.6055 | g_loss: 1.6726
Epoch [ 1/ 9] | d_loss: 0.5197 | g_loss: 1.9397
Epoch [ 1/ 9] | d_loss: 1.7333 | g_loss: 0.5600
Epoch [ 1/ 9] | d_loss: 0.6408 | g_loss: 1.6781
Epoch [ 1/ 9] | d_loss: 1.0931 | g_loss: 1.6446
Epoch [ 1/ 9] | d_loss: 1.3046 | g_loss: 1.5344
Epoch [ 1/ 9] | d_loss: 0.5757 | g_loss: 2.4309
Epoch [ 1/ 9] | d_loss: 0.8850 | g_loss: 1.2451
Epoch [ 1/ 9] | d_loss: 0.9796 | g_loss: 0.4989
Epoch [ 1/ 9] | d_loss: 0.4742 | g_loss: 1.6576
Epoch [ 1/ 9] | d_loss: 1.2075 | g_loss: 2.7544
Epoch [ 1/ 9] | d_loss: 0.3531 | g_loss: 2.4391

```

Epoch [	1/	9]	d_loss: 0.6319	g_loss: 1.4661
Epoch [	1/	9]	d_loss: 1.1582	g_loss: 4.5075
Epoch [	1/	9]	d_loss: 3.2163	g_loss: 0.9765
Epoch [	1/	9]	d_loss: 1.4750	g_loss: 5.2260
Epoch [	1/	9]	d_loss: 0.9633	g_loss: 0.5486
Epoch [	1/	9]	d_loss: 0.8149	g_loss: 1.3845
Epoch [	2/	9]	d_loss: 1.5421	g_loss: 6.2653
Epoch [	2/	9]	d_loss: 0.8382	g_loss: 3.7912
Epoch [	2/	9]	d_loss: 0.7830	g_loss: 3.4746
Epoch [	2/	9]	d_loss: 0.4573	g_loss: 1.9690
Epoch [	2/	9]	d_loss: 0.9355	g_loss: 3.8907
Epoch [	2/	9]	d_loss: 1.2764	g_loss: 1.1358
Epoch [	2/	9]	d_loss: 0.4644	g_loss: 2.0846
Epoch [	2/	9]	d_loss: 1.9459	g_loss: 1.3851
Epoch [	2/	9]	d_loss: 0.6253	g_loss: 3.2101
Epoch [	2/	9]	d_loss: 1.2487	g_loss: 4.7702
Epoch [	2/	9]	d_loss: 0.8403	g_loss: 0.7865
Epoch [	2/	9]	d_loss: 0.5598	g_loss: 1.4025
Epoch [	2/	9]	d_loss: 0.7380	g_loss: 1.0696
Epoch [	2/	9]	d_loss: 0.9652	g_loss: 3.6179
Epoch [	2/	9]	d_loss: 0.8436	g_loss: 0.4537
Epoch [	2/	9]	d_loss: 0.5177	g_loss: 3.2016
Epoch [	2/	9]	d_loss: 1.0142	g_loss: 4.9281
Epoch [	2/	9]	d_loss: 0.4894	g_loss: 2.5317
Epoch [	2/	9]	d_loss: 0.5148	g_loss: 1.7413
Epoch [	2/	9]	d_loss: 0.6074	g_loss: 1.5565
Epoch [	2/	9]	d_loss: 0.5213	g_loss: 1.1189
Epoch [	2/	9]	d_loss: 0.4522	g_loss: 4.1900
Epoch [	2/	9]	d_loss: 0.4758	g_loss: 2.2547
Epoch [	2/	9]	d_loss: 0.3835	g_loss: 2.2996
Epoch [	2/	9]	d_loss: 0.7449	g_loss: 1.3510
Epoch [	2/	9]	d_loss: 0.3822	g_loss: 1.4731
Epoch [	2/	9]	d_loss: 0.4326	g_loss: 3.4253
Epoch [	2/	9]	d_loss: 0.5302	g_loss: 1.0396
Epoch [	2/	9]	d_loss: 0.4496	g_loss: 3.2937
Epoch [	3/	9]	d_loss: 1.9662	g_loss: 3.5406
Epoch [	3/	9]	d_loss: 2.2276	g_loss: 1.9887
Epoch [	3/	9]	d_loss: 1.3362	g_loss: 2.6330
Epoch [	3/	9]	d_loss: 1.0121	g_loss: 1.6018
Epoch [	3/	9]	d_loss: 1.1324	g_loss: 0.5190
Epoch [	3/	9]	d_loss: 0.6267	g_loss: 2.7386
Epoch [	3/	9]	d_loss: 1.4555	g_loss: 1.7512
Epoch [	3/	9]	d_loss: 0.9079	g_loss: 1.7002
Epoch [	3/	9]	d_loss: 0.6959	g_loss: 4.3628
Epoch [	3/	9]	d_loss: 0.1622	g_loss: 3.5600
Epoch [	3/	9]	d_loss: 0.5090	g_loss: 5.4614
Epoch [	3/	9]	d_loss: 0.7704	g_loss: 3.6651
Epoch [	3/	9]	d_loss: 0.5715	g_loss: 3.0847



Epoch [	3/	9]	d_loss: 0.7250	g_loss: 4.5205
Epoch [	3/	9]	d_loss: 0.3658	g_loss: 3.1634
Epoch [	3/	9]	d_loss: 0.8650	g_loss: 3.7981
Epoch [	3/	9]	d_loss: 0.5879	g_loss: 4.0397
Epoch [	3/	9]	d_loss: 0.5368	g_loss: 1.6812
Epoch [	3/	9]	d_loss: 0.2487	g_loss: 3.9159
Epoch [	3/	9]	d_loss: 0.5003	g_loss: 4.7329
Epoch [	3/	9]	d_loss: 0.4685	g_loss: 2.3815
Epoch [	3/	9]	d_loss: 0.3042	g_loss: 4.4166
Epoch [	3/	9]	d_loss: 0.7867	g_loss: 0.9831
Epoch [	3/	9]	d_loss: 0.4097	g_loss: 3.8393
Epoch [	3/	9]	d_loss: 0.3597	g_loss: 2.1723
Epoch [	3/	9]	d_loss: 0.5933	g_loss: 2.2387
Epoch [	3/	9]	d_loss: 1.2030	g_loss: 3.6510
Epoch [	3/	9]	d_loss: 0.3823	g_loss: 2.3427
Epoch [	3/	9]	d_loss: 0.2344	g_loss: 4.2793
Epoch [	4/	9]	d_loss: 2.8975	g_loss: 9.9686
Epoch [	4/	9]	d_loss: 0.1684	g_loss: 4.1520
Epoch [	4/	9]	d_loss: 0.3362	g_loss: 2.9914
Epoch [	4/	9]	d_loss: 0.3790	g_loss: 3.2681
Epoch [	4/	9]	d_loss: 1.1895	g_loss: 5.6891
Epoch [	4/	9]	d_loss: 0.7493	g_loss: 3.7473
Epoch [	4/	9]	d_loss: 0.2084	g_loss: 2.7223
Epoch [	4/	9]	d_loss: 0.2462	g_loss: 3.8728
Epoch [	4/	9]	d_loss: 0.4253	g_loss: 3.4075
Epoch [	4/	9]	d_loss: 0.8379	g_loss: 4.6684
Epoch [	4/	9]	d_loss: 0.3183	g_loss: 3.0902
Epoch [	4/	9]	d_loss: 0.2104	g_loss: 3.1040
Epoch [	4/	9]	d_loss: 0.2643	g_loss: 3.3038
Epoch [	4/	9]	d_loss: 0.3856	g_loss: 5.0902
Epoch [	4/	9]	d_loss: 0.2523	g_loss: 1.7738
Epoch [	4/	9]	d_loss: 0.2600	g_loss: 2.7192
Epoch [	4/	9]	d_loss: 0.1583	g_loss: 3.3139
Epoch [	4/	9]	d_loss: 0.9489	g_loss: 0.9729
Epoch [	4/	9]	d_loss: 0.1289	g_loss: 2.7293
Epoch [	4/	9]	d_loss: 0.3529	g_loss: 2.0577
Epoch [	4/	9]	d_loss: 0.8545	g_loss: 5.6128
Epoch [	4/	9]	d_loss: 1.1636	g_loss: 2.4880
Epoch [	4/	9]	d_loss: 0.6842	g_loss: 4.8692
Epoch [	4/	9]	d_loss: 0.2609	g_loss: 2.9460
Epoch [	4/	9]	d_loss: 0.7726	g_loss: 1.6858
Epoch [	4/	9]	d_loss: 0.2328	g_loss: 2.4569
Epoch [	4/	9]	d_loss: 0.3833	g_loss: 3.7560
Epoch [	4/	9]	d_loss: 0.1950	g_loss: 2.6023
Epoch [	4/	9]	d_loss: 0.4990	g_loss: 3.7707
Epoch [	5/	9]	d_loss: 1.1381	g_loss: 7.0016
Epoch [	5/	9]	d_loss: 0.1092	g_loss: 3.3716
Epoch [	5/	9]	d_loss: 0.2719	g_loss: 4.9380

Epoch [	5/	9]	d_loss: 0.3127	g_loss: 5.9361
Epoch [	5/	9]	d_loss: 0.2776	g_loss: 4.5558
Epoch [	5/	9]	d_loss: 1.1340	g_loss: 2.5395
Epoch [	5/	9]	d_loss: 0.3112	g_loss: 4.6515
Epoch [	5/	9]	d_loss: 0.5563	g_loss: 5.2716
Epoch [	5/	9]	d_loss: 0.6980	g_loss: 1.5427
Epoch [	5/	9]	d_loss: 0.1312	g_loss: 2.9846
Epoch [	5/	9]	d_loss: 0.3808	g_loss: 3.1686
Epoch [	5/	9]	d_loss: 0.5308	g_loss: 3.7022
Epoch [	5/	9]	d_loss: 0.8764	g_loss: 0.8674
Epoch [	5/	9]	d_loss: 0.4440	g_loss: 2.6200
Epoch [	5/	9]	d_loss: 2.5026	g_loss: 8.1525
Epoch [	5/	9]	d_loss: 0.6546	g_loss: 3.4424
Epoch [	5/	9]	d_loss: 0.1135	g_loss: 2.5878
Epoch [	5/	9]	d_loss: 2.1709	g_loss: 8.4184
Epoch [	5/	9]	d_loss: 0.1994	g_loss: 3.6993
Epoch [	5/	9]	d_loss: 0.2920	g_loss: 3.7912
Epoch [	5/	9]	d_loss: 0.9659	g_loss: 2.2139
Epoch [	5/	9]	d_loss: 0.1817	g_loss: 4.0522
Epoch [	5/	9]	d_loss: 0.2365	g_loss: 4.2864
Epoch [	5/	9]	d_loss: 0.1275	g_loss: 4.1879
Epoch [	5/	9]	d_loss: 0.3796	g_loss: 2.5324
Epoch [	5/	9]	d_loss: 0.2009	g_loss: 3.8019
Epoch [	5/	9]	d_loss: 0.2313	g_loss: 3.9172
Epoch [	5/	9]	d_loss: 0.2131	g_loss: 3.4396
Epoch [	5/	9]	d_loss: 0.1629	g_loss: 4.0840
Epoch [	6/	9]	d_loss: 6.6257	g_loss: 6.7071
Epoch [	6/	9]	d_loss: 1.6857	g_loss: 0.5226
Epoch [	6/	9]	d_loss: 0.2566	g_loss: 3.8986
Epoch [	6/	9]	d_loss: 0.5121	g_loss: 5.5549
Epoch [	6/	9]	d_loss: 0.2238	g_loss: 4.4837
Epoch [	6/	9]	d_loss: 0.0468	g_loss: 5.5158
Epoch [	6/	9]	d_loss: 0.4667	g_loss: 4.5560
Epoch [	6/	9]	d_loss: 0.2560	g_loss: 3.3788
Epoch [	6/	9]	d_loss: 0.2030	g_loss: 2.2985
Epoch [	6/	9]	d_loss: 0.6159	g_loss: 3.6454
Epoch [	6/	9]	d_loss: 0.4156	g_loss: 3.8950
Epoch [	6/	9]	d_loss: 0.1345	g_loss: 4.0536
Epoch [	6/	9]	d_loss: 0.1691	g_loss: 5.2322
Epoch [	6/	9]	d_loss: 0.4651	g_loss: 3.4177
Epoch [	6/	9]	d_loss: 0.1506	g_loss: 4.4597
Epoch [	6/	9]	d_loss: 0.1349	g_loss: 3.7783
Epoch [	6/	9]	d_loss: 0.1857	g_loss: 2.6000
Epoch [	6/	9]	d_loss: 0.2483	g_loss: 3.1528
Epoch [	6/	9]	d_loss: 0.6742	g_loss: 4.9956
Epoch [	6/	9]	d_loss: 0.2290	g_loss: 4.1220
Epoch [	6/	9]	d_loss: 0.0545	g_loss: 2.7702
Epoch [	6/	9]	d_loss: 0.9274	g_loss: 1.3784

Epoch [	6/	9]	d_loss: 0.3644	g_loss: 4.1320
Epoch [	6/	9]	d_loss: 0.1511	g_loss: 4.3851
Epoch [	6/	9]	d_loss: 0.2152	g_loss: 2.7105
Epoch [	6/	9]	d_loss: 0.1157	g_loss: 5.2074
Epoch [	6/	9]	d_loss: 0.3018	g_loss: 5.9969
Epoch [	6/	9]	d_loss: 0.4179	g_loss: 2.8117
Epoch [	6/	9]	d_loss: 0.2874	g_loss: 2.9750
Epoch [	7/	9]	d_loss: 0.2669	g_loss: 5.6936
Epoch [	7/	9]	d_loss: 0.1077	g_loss: 5.5075
Epoch [	7/	9]	d_loss: 0.2798	g_loss: 3.8365
Epoch [	7/	9]	d_loss: 0.2182	g_loss: 6.2572
Epoch [	7/	9]	d_loss: 0.6458	g_loss: 2.2634
Epoch [	7/	9]	d_loss: 0.0480	g_loss: 5.5294
Epoch [	7/	9]	d_loss: 0.4238	g_loss: 2.1442
Epoch [	7/	9]	d_loss: 1.1726	g_loss: 6.5464
Epoch [	7/	9]	d_loss: 0.1512	g_loss: 5.2646
Epoch [	7/	9]	d_loss: 0.1839	g_loss: 4.9597
Epoch [	7/	9]	d_loss: 0.1827	g_loss: 8.2553
Epoch [	7/	9]	d_loss: 0.2405	g_loss: 4.5048
Epoch [	7/	9]	d_loss: 0.3020	g_loss: 3.9870
Epoch [	7/	9]	d_loss: 0.1053	g_loss: 3.6965
Epoch [	7/	9]	d_loss: 0.2506	g_loss: 5.0487
Epoch [	7/	9]	d_loss: 0.2209	g_loss: 3.0014
Epoch [	7/	9]	d_loss: 0.2470	g_loss: 3.9530
Epoch [	7/	9]	d_loss: 0.5405	g_loss: 1.4933
Epoch [	7/	9]	d_loss: 0.6760	g_loss: 5.8885
Epoch [	7/	9]	d_loss: 0.2432	g_loss: 5.1857
Epoch [	7/	9]	d_loss: 0.1637	g_loss: 6.2713
Epoch [	7/	9]	d_loss: 0.0904	g_loss: 3.7847
Epoch [	7/	9]	d_loss: 0.1525	g_loss: 3.1000
Epoch [	7/	9]	d_loss: 0.1997	g_loss: 5.4802
Epoch [	7/	9]	d_loss: 0.1832	g_loss: 2.9446
Epoch [	7/	9]	d_loss: 0.3351	g_loss: 2.6623
Epoch [	7/	9]	d_loss: 0.1148	g_loss: 4.0276
Epoch [	7/	9]	d_loss: 0.0731	g_loss: 5.1703
Epoch [	7/	9]	d_loss: 0.1982	g_loss: 4.6701
Epoch [	8/	9]	d_loss: 3.3506	g_loss: 9.9586
Epoch [	8/	9]	d_loss: 0.1270	g_loss: 3.7622
Epoch [	8/	9]	d_loss: 0.0963	g_loss: 5.9563
Epoch [	8/	9]	d_loss: 0.9642	g_loss: 2.7648
Epoch [	8/	9]	d_loss: 0.7916	g_loss: 5.6521
Epoch [	8/	9]	d_loss: 0.2152	g_loss: 5.9664
Epoch [	8/	9]	d_loss: 0.1170	g_loss: 6.2532
Epoch [	8/	9]	d_loss: 0.0924	g_loss: 3.0928
Epoch [	8/	9]	d_loss: 0.1877	g_loss: 4.9353
Epoch [	8/	9]	d_loss: 0.5013	g_loss: 6.9880
Epoch [	8/	9]	d_loss: 0.0971	g_loss: 2.3097
Epoch [	8/	9]	d_loss: 0.1248	g_loss: 5.9435

Epoch [	8/	9]	d_loss: 0.1914	g_loss: 6.1226
Epoch [	8/	9]	d_loss: 0.2869	g_loss: 7.3339
Epoch [	8/	9]	d_loss: 0.2757	g_loss: 5.8185
Epoch [	8/	9]	d_loss: 0.4263	g_loss: 5.0918
Epoch [	8/	9]	d_loss: 0.4503	g_loss: 6.7644
Epoch [	8/	9]	d_loss: 0.3682	g_loss: 5.0452
Epoch [	8/	9]	d_loss: 0.1532	g_loss: 5.5405
Epoch [	8/	9]	d_loss: 0.0506	g_loss: 5.7186
Epoch [	8/	9]	d_loss: 1.0205	g_loss: 4.4961
Epoch [	8/	9]	d_loss: 0.3041	g_loss: 6.4121
Epoch [	8/	9]	d_loss: 0.1754	g_loss: 3.6321
Epoch [	8/	9]	d_loss: 0.0696	g_loss: 6.4764
Epoch [	8/	9]	d_loss: 0.4517	g_loss: 2.4894
Epoch [	8/	9]	d_loss: 0.8184	g_loss: 6.4752
Epoch [	8/	9]	d_loss: 0.0778	g_loss: 5.2447
Epoch [	8/	9]	d_loss: 0.2299	g_loss: 4.7714
Epoch [	8/	9]	d_loss: 0.1626	g_loss: 3.9193
Epoch [	9/	9]	d_loss: 0.1653	g_loss: 6.0146
Epoch [	9/	9]	d_loss: 0.0215	g_loss: 5.8712
Epoch [	9/	9]	d_loss: 0.2781	g_loss: 7.7830
Epoch [	9/	9]	d_loss: 0.0378	g_loss: 6.3741
Epoch [	9/	9]	d_loss: 0.3599	g_loss: 9.5994
Epoch [	9/	9]	d_loss: 0.0438	g_loss: 5.2294
Epoch [	9/	9]	d_loss: 0.2487	g_loss: 7.1007
Epoch [	9/	9]	d_loss: 0.2758	g_loss: 5.3811
Epoch [	9/	9]	d_loss: 0.5436	g_loss: 8.9572
Epoch [	9/	9]	d_loss: 0.2793	g_loss: 3.3984
Epoch [	9/	9]	d_loss: 0.1872	g_loss: 3.4469
Epoch [	9/	9]	d_loss: 0.2039	g_loss: 3.9865
Epoch [	9/	9]	d_loss: 0.8759	g_loss: 10.2778
Epoch [	9/	9]	d_loss: 0.1746	g_loss: 6.9233
Epoch [	9/	9]	d_loss: 0.0703	g_loss: 5.1062
Epoch [	9/	9]	d_loss: 0.1614	g_loss: 5.0824
Epoch [	9/	9]	d_loss: 0.0374	g_loss: 6.5800
Epoch [	9/	9]	d_loss: 3.0839	g_loss: 11.4430
Epoch [	9/	9]	d_loss: 2.1368	g_loss: 4.3691
Epoch [	9/	9]	d_loss: 0.0608	g_loss: 6.2579
Epoch [	9/	9]	d_loss: 0.1479	g_loss: 9.0934
Epoch [	9/	9]	d_loss: 0.2333	g_loss: 2.9387
Epoch [	9/	9]	d_loss: 0.8008	g_loss: 7.6003
Epoch [	9/	9]	d_loss: 0.1008	g_loss: 4.6810
Epoch [	9/	9]	d_loss: 0.3219	g_loss: 8.4006
Epoch [	9/	9]	d_loss: 0.1539	g_loss: 6.0522
Epoch [	9/	9]	d_loss: 0.1056	g_loss: 5.8204
Epoch [	9/	9]	d_loss: 1.7646	g_loss: 15.1023
Epoch [	9/	9]	d_loss: 0.1121	g_loss: 5.1324

## 2.8 Training loss

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

```
In [22]: 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[22]: <matplotlib.legend.Legend at 0x7f081cf2e668>
```



## 2.9 Generator samples from training

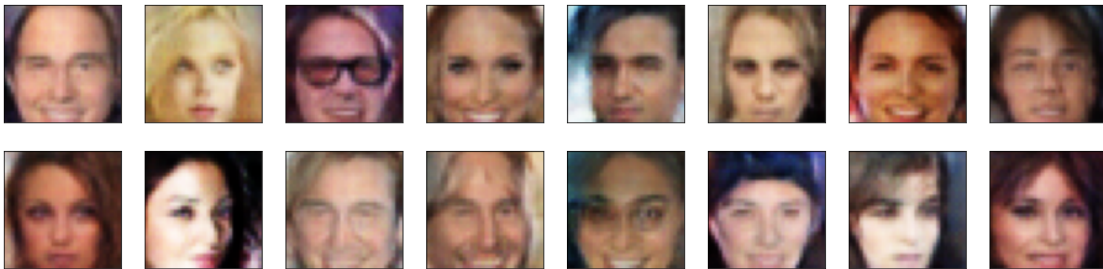
View samples of images from the generator, and answer a question about the strengths and weaknesses of your trained models.

```
In [23]: # 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.axis.set_visible(False)
```

```
ax.yaxis.set_visible(False)
im = ax.imshow(img.reshape((32,32,3)))
```

```
In [24]: # Load samples from generator, taken while training
with open('train_samples.pkl', 'rb') as f:
    samples = pickle.load(f)
```

```
In [25]: _ = view_samples(-1, samples)
```



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

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

**Answer:** (Write your answer in this cell) - I used 5 convolutional layers but some images are not clear due to the complex human features. To tackle this problem, we can add more convolutional layers to improve the output performance. - Conv\_dim = 64. The model could be improved if we increase it, then the effective statistic was enhanced. - Epochs = 7. Because there is no improvement occurs, so we limit the epochs which is depended on the spread between generator and discriminator batch size.

### 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 "dln\_d\_face\_generation.ipynb" and save it as a HTML file under "File" -> "Download as". Include the "problem\_unittests.py" files in your submission.

```
In [25]:
```