dlnd_face_generation

May 31, 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 [1]: # can comment out after executing
        #!unzip processed_celeba_small.zip
```

```
In [2]: data_dir = 'processed_celeba_small/'

        """
        DON'T MODIFY ANYTHING IN THIS CELL
        """
        import pickle as pkl
        import matplotlib.pyplot as plt
```

```
import numpy as np
import problem_unittests as tests
#import helper

%matplotlib inline
```

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

```
In [4]: def get_dataloader(batch_size, image_size, data_dir='processed_celeba_small/'):
            """
            Batch the neural network data using DataLoader
            :param batch_size: The size of each batch; the number of images in a batch
            :param img_size: The square size of the image data (x, y)
            :param data_dir: Directory where image data is located
            :return: DataLoader with batched data
            """
            # Define the transformation for size, square nd tensor transformation
            transformation = transforms.Compose([transforms.Resize(image_size),
                                                 transforms.CenterCrop(image_size),
```

```
                                     transforms.ToTensor()])
    # ImageFolder wrapper for the folder with the content and the defined transformation
    image_data = datasets.ImageFolder(data_dir, transform=transformation)
    # Data Loader with pre-defined batch size and shuffling the batches
    data_loader = torch.utils.data.DataLoader(image_data, batch_size=batch_size, shuffle
    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 [5]: # 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)
```

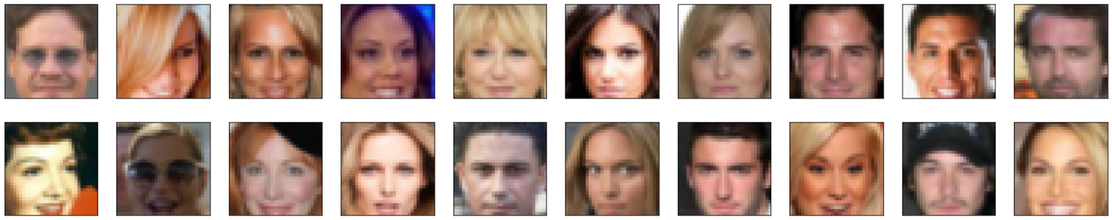
Next, you can view some images! You should see square images of somewhat-centered faces.

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 [6]: # helper display function
        def imshow(img):
            npimg = img.numpy()
            plt.imshow(np.transpose(npimg, (1, 2, 0)))

        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        # obtain one batch of training images
        dataiter = iter(celeba_train_loader)
        images, _ = dataiter.next() # _ for no labels

        # plot the images in the batch, along with the corresponding labels
        fig = plt.figure(figsize=(20, 4))
        plot_size=20
        for idx in np.arange(plot_size):
            ax = fig.add_subplot(2, plot_size/2, idx+1, xticks=[], yticks=[])
            imshow(images[idx])
```



Exercise: Pre-process your image data and scale it to a pixel range of -1 to 1 You need to do a bit of pre-processing; you know that the output of a `tanh` activated generator will contain pixel values in a range from -1 to 1, and so, we need to rescale our training images to a range of -1 to 1. (Right now, they are in a range from 0-1.)

```
In [7]: # TODO: Complete the scale function
        def scale(x, feature_range=(-1, 1)):
            ''' Scale takes in an image x and returns that image, scaled
               with a feature_range of pixel values from -1 to 1.
               This function assumes that the input x is already scaled from 0-1.'''
            # assume x is scaled to (0, 1)
            # scale to feature_range and return scaled x
            min, max = feature_range
            x = x * (max - min) + min
            return x
```

```
In [8]: """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        # check scaled range
        # should be close to -1 to 1
        img = images[0]
        scaled_img = scale(img)

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

```
Min:  tensor(-0.7098)
Max:  tensor(0.9294)
```

---

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

In [10]:
```
# Creating a conv helper function
def conv(in_channels, out_channels, kernel_size=4, stride=2, padding=1, batch_norm=True
    layers = []
    conv_layer = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, bia

    # 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 [11]:
```
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, batch_norm = False)
        # 16x16 input
        self.conv2 = conv(conv_dim, conv_dim*2)
        # 8x8 input
        self.conv3 = conv(conv_dim*2, conv_dim*4)
        # 4x4 out
```

```
        # fully connected layer Image: 4x4 * conv_dim*4
        self.fc = nn.Linear(conv_dim*4*4*4, 1)

    def forward(self, x):
        """
        Forward propagation of the neural network
        :param x: The input to the neural network
        :return: Discriminator logits; the output of the neural network
        """
        # Applying Relus on the 3 conv layers
        x = F.leaky_relu(self.conv1(x), 0.2)
        x = F.leaky_relu(self.conv2(x), 0.2)
        x = F.leaky_relu(self.conv3(x), 0.2)

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

        # FC Layer
        x = self.fc(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 [12]: # deconv helper function
         def deconv(in_channels, out_channels, kernel_size = 4, stride = 2, padding = 1, batch_n
             # create a sequence of transpose + optional batch norm layers
             layers = []
             transpose_conv_layer = nn.ConvTranspose2d(in_channels, out_channels, kernel_size, s
             # 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 [13]:
```
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__()

        self.conv_dim = conv_dim

        # Fully connected layer from z values:
        self.fc = nn.Linear(z_size, conv_dim*4*4*4)

        # Transposed convolutional layers with helper function
        self.t_conv1 = deconv(conv_dim*4, conv_dim*2)
        self.t_conv2 = deconv(conv_dim*2, conv_dim)
        self.t_conv3 = deconv(conv_dim, 3, batch_norm=False)

    def forward(self, x):
        """
        Forward propagation of the neural network
        :param x: The input to the neural network
        :return: A 32x32x3 Tensor image as output
        """
        # FC Layer
        x = self.fc(x)
        # Resepahing to Tensor
        x = x.view(-1, self.conv_dim*4, 4, 4)

        # Transpose conv layers and relu activation
        x = F.relu(self.t_conv1(x))
        x = F.relu(self.t_conv2(x))

        # Last Layer and tanh activation
        x = self.t_conv3(x)
        x = F.tanh(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 [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__

             if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear
                 m.weight.data.normal_(0, 0.02)

                 # Bias term to 0
                 if hasattr(m, 'bias') and m.bias is not None:
                     m.bias.data.zero_()
```

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): 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)
  )
  (fc): Linear(in_features=4096, 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)
  )
)
```

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]:
```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 [18]: 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)

             # move labels to GPU if available
             if train_on_gpu:
                 labels = labels.cuda()

             # binary cross entropy with logits loss
             criterion = nn.BCEWithLogitsLoss()
             # calculate loss
             loss = criterion(D_out.squeeze(), labels)
             return loss

         def fake_loss(D_out):
             '''Calculates how close discriminator outputs are to being fake.
                param, D_out: discriminator logits
                return: fake loss'''
             batch_size = D_out.size(0)
             labels = torch.zeros(batch_size)

             if train_on_gpu:
                 labels = labels.cuda()

             criterion = nn.BCEWithLogitsLoss()
             # 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

         # params
```

```
lr = 0.0002
beta1 = 0.5
beta2 = 0.999

# Create optimizers for the discriminator D and generator G
d_optimizer = optim.Adam(D.parameters(), lr, [beta1, beta2])
g_optimizer = optim.Adam(G.parameters(), lr, [beta1, beta2])
```

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
        # ===============================================

        # 1. Train the discriminator on real and fake images

        # Set grad to zero
        d_optimizer.zero_grad()

        # Train on GPU
        if train_on_gpu:
            real_images = real_images.cuda()

        # Apply Discriminator on the real images
        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()

        # Move to GPU
        if train_on_gpu:
            z = z.cuda()

        # Apply Generator on z to generate fake images
        fake_images = G(z)

        # Running the fake images on the Discriminator
        D_fake = D(fake_images)
        d_fake_loss = fake_loss(D_fake)

        d_loss = d_real_loss + d_fake_loss
        d_loss.backward()
        d_optimizer.step()
```

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

            # Set the Gradient to zero
            g_optimizer.zero_grad()

            # generate z
            z = np.random.uniform(-1, 1, size=(batch_size, z_size))
            z = torch.from_numpy(z).float()

            # Move to GPU
            if train_on_gpu:
                z = z.cuda()

            # create fake immages with the Generator
            fake_images = G(z)

            # Apply Discriminator and calculate loss
            D_fake = D(fake_images)
            g_loss = real_loss(D_fake)

            # back-prop and optimizer
            g_loss.backward()
            g_optimizer.step()

            # ================================================
            #              END OF YOUR CODE
            # ================================================

            # Print some loss stats
            if batch_i % print_every == 0:
                # append discriminator loss and generator loss
                losses.append((d_loss.item(), g_loss.item()))
                # print discriminator and generator loss
                print('Epoch [{:5d}/{:5d}] | d_loss: {:6.4f} | g_loss: {:6.4f}'.format(
                        epoch+1, n_epochs, d_loss.item(), g_loss.item()))


        ## AFTER EACH EPOCH##
        # this code assumes your generator is named G, feel free to change the name
        # generate and save sample, fake images
        G.eval() # for generating samples
        samples_z = G(fixed_z)
        samples.append(samples_z)
        G.train() # back to training mode

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

```
            pkl.dump(samples, f)

        # finally return losses
        return losses
```

Set your number of training epochs and train your GAN!

```
In [21]: # set number of epochs
         n_epochs = 10


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

```
Epoch [    1/   10] | d_loss: 1.3848 | g_loss: 1.0317
Epoch [    1/   10] | d_loss: 0.0865 | g_loss: 4.0679
Epoch [    1/   10] | d_loss: 0.2049 | g_loss: 3.2902
Epoch [    1/   10] | d_loss: 0.3237 | g_loss: 3.4050
Epoch [    1/   10] | d_loss: 0.5329 | g_loss: 3.7378
Epoch [    1/   10] | d_loss: 2.0889 | g_loss: 4.2348
Epoch [    1/   10] | d_loss: 0.4855 | g_loss: 1.8386
Epoch [    1/   10] | d_loss: 0.7826 | g_loss: 1.5993
Epoch [    1/   10] | d_loss: 0.7872 | g_loss: 1.5260
Epoch [    1/   10] | d_loss: 0.6386 | g_loss: 2.4055
Epoch [    1/   10] | d_loss: 0.7807 | g_loss: 1.4434
Epoch [    1/   10] | d_loss: 0.8968 | g_loss: 0.7620
Epoch [    1/   10] | d_loss: 0.9718 | g_loss: 1.6253
Epoch [    1/   10] | d_loss: 0.9436 | g_loss: 1.3210
Epoch [    1/   10] | d_loss: 1.0262 | g_loss: 0.9289
Epoch [    2/   10] | d_loss: 0.9623 | g_loss: 1.2973
Epoch [    2/   10] | d_loss: 0.9433 | g_loss: 0.9757
Epoch [    2/   10] | d_loss: 0.9030 | g_loss: 1.1766
Epoch [    2/   10] | d_loss: 1.1302 | g_loss: 1.5276
Epoch [    2/   10] | d_loss: 1.1149 | g_loss: 1.1764
Epoch [    2/   10] | d_loss: 1.1557 | g_loss: 1.4719
Epoch [    2/   10] | d_loss: 0.8660 | g_loss: 1.3539
Epoch [    2/   10] | d_loss: 1.0941 | g_loss: 1.4021
Epoch [    2/   10] | d_loss: 1.0820 | g_loss: 1.1755
Epoch [    2/   10] | d_loss: 1.1008 | g_loss: 0.9431
Epoch [    2/   10] | d_loss: 1.0564 | g_loss: 1.4423
Epoch [    2/   10] | d_loss: 1.1263 | g_loss: 1.4151
Epoch [    2/   10] | d_loss: 1.0669 | g_loss: 1.6489
Epoch [    2/   10] | d_loss: 1.1321 | g_loss: 1.0313
Epoch [    2/   10] | d_loss: 1.0512 | g_loss: 1.5004
Epoch [    3/   10] | d_loss: 0.9489 | g_loss: 1.5805
```

```
Epoch [    3/   10] | d_loss: 0.9582 | g_loss: 1.3813
Epoch [    3/   10] | d_loss: 1.2932 | g_loss: 1.6055
Epoch [    3/   10] | d_loss: 1.1473 | g_loss: 0.9118
Epoch [    3/   10] | d_loss: 1.1042 | g_loss: 0.9264
Epoch [    3/   10] | d_loss: 1.1654 | g_loss: 0.6028
Epoch [    3/   10] | d_loss: 1.0934 | g_loss: 0.8811
Epoch [    3/   10] | d_loss: 0.9749 | g_loss: 1.2898
Epoch [    3/   10] | d_loss: 1.1713 | g_loss: 0.8371
Epoch [    3/   10] | d_loss: 0.9626 | g_loss: 1.0160
Epoch [    3/   10] | d_loss: 0.8344 | g_loss: 1.5120
Epoch [    3/   10] | d_loss: 1.1922 | g_loss: 1.6833
Epoch [    3/   10] | d_loss: 1.3811 | g_loss: 0.5569
Epoch [    3/   10] | d_loss: 1.1223 | g_loss: 1.1463
Epoch [    3/   10] | d_loss: 1.0987 | g_loss: 1.4966
Epoch [    4/   10] | d_loss: 1.0906 | g_loss: 0.8383
Epoch [    4/   10] | d_loss: 1.0489 | g_loss: 1.6813
Epoch [    4/   10] | d_loss: 0.9405 | g_loss: 1.2938
Epoch [    4/   10] | d_loss: 0.8860 | g_loss: 1.0859
Epoch [    4/   10] | d_loss: 1.2250 | g_loss: 0.7602
Epoch [    4/   10] | d_loss: 1.0216 | g_loss: 1.0612
Epoch [    4/   10] | d_loss: 0.9219 | g_loss: 1.5956
Epoch [    4/   10] | d_loss: 1.0511 | g_loss: 0.7719
Epoch [    4/   10] | d_loss: 0.9834 | g_loss: 0.7999
Epoch [    4/   10] | d_loss: 1.4256 | g_loss: 0.4232
Epoch [    4/   10] | d_loss: 1.1274 | g_loss: 1.7267
Epoch [    4/   10] | d_loss: 0.9195 | g_loss: 2.0237
Epoch [    4/   10] | d_loss: 1.1150 | g_loss: 0.7380
Epoch [    4/   10] | d_loss: 1.1390 | g_loss: 0.5704
Epoch [    4/   10] | d_loss: 0.9500 | g_loss: 0.8966
Epoch [    5/   10] | d_loss: 0.9530 | g_loss: 1.5228
Epoch [    5/   10] | d_loss: 1.0534 | g_loss: 1.5873
Epoch [    5/   10] | d_loss: 1.3852 | g_loss: 0.4986
Epoch [    5/   10] | d_loss: 0.9195 | g_loss: 1.4738
Epoch [    5/   10] | d_loss: 1.0895 | g_loss: 0.9799
Epoch [    5/   10] | d_loss: 0.7773 | g_loss: 1.4730
Epoch [    5/   10] | d_loss: 0.9868 | g_loss: 1.5019
Epoch [    5/   10] | d_loss: 0.8869 | g_loss: 1.0780
Epoch [    5/   10] | d_loss: 0.9954 | g_loss: 1.1470
Epoch [    5/   10] | d_loss: 0.9349 | g_loss: 1.1140
Epoch [    5/   10] | d_loss: 1.0955 | g_loss: 1.1504
Epoch [    5/   10] | d_loss: 0.9404 | g_loss: 1.0557
Epoch [    5/   10] | d_loss: 1.0428 | g_loss: 1.3134
Epoch [    5/   10] | d_loss: 0.8455 | g_loss: 1.2000
Epoch [    5/   10] | d_loss: 1.1956 | g_loss: 0.7354
Epoch [    6/   10] | d_loss: 0.8416 | g_loss: 1.1681
Epoch [    6/   10] | d_loss: 1.0153 | g_loss: 0.8833
Epoch [    6/   10] | d_loss: 1.5240 | g_loss: 2.8041
Epoch [    6/   10] | d_loss: 0.7251 | g_loss: 1.7810
```

```
Epoch [    6/   10] | d_loss: 1.0628 | g_loss: 0.9683
Epoch [    6/   10] | d_loss: 0.9565 | g_loss: 1.1104
Epoch [    6/   10] | d_loss: 1.1749 | g_loss: 1.7561
Epoch [    6/   10] | d_loss: 1.0282 | g_loss: 1.7698
Epoch [    6/   10] | d_loss: 1.0964 | g_loss: 1.2815
Epoch [    6/   10] | d_loss: 1.1126 | g_loss: 0.9641
Epoch [    6/   10] | d_loss: 1.4566 | g_loss: 1.9784
Epoch [    6/   10] | d_loss: 1.0194 | g_loss: 0.8733
Epoch [    6/   10] | d_loss: 0.9961 | g_loss: 1.0358
Epoch [    6/   10] | d_loss: 1.6942 | g_loss: 0.6681
Epoch [    6/   10] | d_loss: 0.7524 | g_loss: 2.1149
Epoch [    7/   10] | d_loss: 0.9397 | g_loss: 1.5187
Epoch [    7/   10] | d_loss: 1.0464 | g_loss: 1.3926
Epoch [    7/   10] | d_loss: 0.8807 | g_loss: 1.3250
Epoch [    7/   10] | d_loss: 0.8966 | g_loss: 1.6015
Epoch [    7/   10] | d_loss: 0.9429 | g_loss: 1.1847
Epoch [    7/   10] | d_loss: 0.8048 | g_loss: 1.6319
Epoch [    7/   10] | d_loss: 0.9552 | g_loss: 1.3203
Epoch [    7/   10] | d_loss: 1.0162 | g_loss: 0.9360
Epoch [    7/   10] | d_loss: 0.9814 | g_loss: 1.3738
Epoch [    7/   10] | d_loss: 0.9114 | g_loss: 1.9042
Epoch [    7/   10] | d_loss: 1.0784 | g_loss: 0.8970
Epoch [    7/   10] | d_loss: 0.7878 | g_loss: 1.2650
Epoch [    7/   10] | d_loss: 1.0364 | g_loss: 0.8628
Epoch [    7/   10] | d_loss: 0.9953 | g_loss: 1.3935
Epoch [    7/   10] | d_loss: 1.2202 | g_loss: 0.7879
Epoch [    8/   10] | d_loss: 1.1796 | g_loss: 1.2244
Epoch [    8/   10] | d_loss: 1.1412 | g_loss: 0.9225
Epoch [    8/   10] | d_loss: 0.9066 | g_loss: 1.3896
Epoch [    8/   10] | d_loss: 1.0572 | g_loss: 0.7858
Epoch [    8/   10] | d_loss: 1.0933 | g_loss: 0.9334
Epoch [    8/   10] | d_loss: 1.3175 | g_loss: 0.9971
Epoch [    8/   10] | d_loss: 1.0659 | g_loss: 1.0750
Epoch [    8/   10] | d_loss: 0.9292 | g_loss: 1.6820
Epoch [    8/   10] | d_loss: 1.3219 | g_loss: 1.0629
Epoch [    8/   10] | d_loss: 0.9993 | g_loss: 1.3302
Epoch [    8/   10] | d_loss: 0.9065 | g_loss: 1.3843
Epoch [    8/   10] | d_loss: 0.9728 | g_loss: 1.1199
Epoch [    8/   10] | d_loss: 0.9195 | g_loss: 1.7747
Epoch [    8/   10] | d_loss: 0.7410 | g_loss: 1.6200
Epoch [    8/   10] | d_loss: 0.9098 | g_loss: 0.7428
Epoch [    9/   10] | d_loss: 1.7157 | g_loss: 3.1344
Epoch [    9/   10] | d_loss: 0.9292 | g_loss: 1.3809
Epoch [    9/   10] | d_loss: 0.9104 | g_loss: 0.9060
Epoch [    9/   10] | d_loss: 0.8020 | g_loss: 1.3174
Epoch [    9/   10] | d_loss: 0.9076 | g_loss: 1.0880
Epoch [    9/   10] | d_loss: 0.9653 | g_loss: 0.7784
Epoch [    9/   10] | d_loss: 0.7974 | g_loss: 1.1021
```

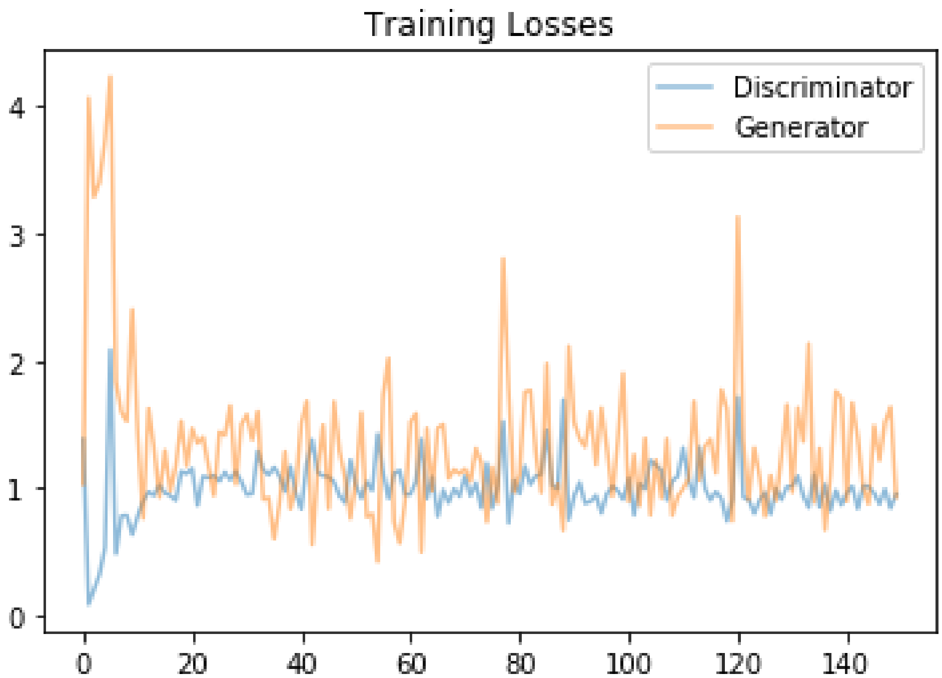
```
Epoch [    9/   10] | d_loss: 0.9970 | g_loss: 0.9002
Epoch [    9/   10] | d_loss: 0.9096 | g_loss: 1.2509
Epoch [    9/   10] | d_loss: 1.0104 | g_loss: 1.6557
Epoch [    9/   10] | d_loss: 1.0152 | g_loss: 0.9667
Epoch [    9/   10] | d_loss: 1.0969 | g_loss: 1.6327
Epoch [    9/   10] | d_loss: 0.9415 | g_loss: 1.3703
Epoch [    9/   10] | d_loss: 0.8479 | g_loss: 2.1367
Epoch [    9/   10] | d_loss: 1.1096 | g_loss: 0.8890
Epoch [   10/   10] | d_loss: 0.8526 | g_loss: 1.3131
Epoch [   10/   10] | d_loss: 1.0300 | g_loss: 0.6714
Epoch [   10/   10] | d_loss: 0.8223 | g_loss: 1.0999
Epoch [   10/   10] | d_loss: 0.9820 | g_loss: 1.7601
Epoch [   10/   10] | d_loss: 0.8686 | g_loss: 1.7095
Epoch [   10/   10] | d_loss: 0.9627 | g_loss: 0.8961
Epoch [   10/   10] | d_loss: 1.0138 | g_loss: 1.6701
Epoch [   10/   10] | d_loss: 0.8379 | g_loss: 1.3998
Epoch [   10/   10] | d_loss: 1.0188 | g_loss: 1.0016
Epoch [   10/   10] | d_loss: 1.0127 | g_loss: 0.8766
Epoch [   10/   10] | d_loss: 0.9614 | g_loss: 1.4913
Epoch [   10/   10] | d_loss: 0.8748 | g_loss: 1.2173
Epoch [   10/   10] | d_loss: 0.9916 | g_loss: 1.5193
Epoch [   10/   10] | d_loss: 0.8429 | g_loss: 1.6373
Epoch [   10/   10] | d_loss: 0.9534 | g_loss: 0.9294
```

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 0x7f531c3e6978>
```



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.xaxis.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 = pkl.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:

I am quite surpised by the output of the network. Most of the images look like true faces. I do see some slight glithces, even though those could still be real people.

All my faces in that sample seem to be good in detecting the features of a normal face, hence my convolutional network is doing a fine job in feature detection.

It seems to only clash when it is about face vs. background and face vs. hair/hat. That seems to be the areas for error.

Obvisouyly, we could deeopen the network by adding another layer or adjusting the depth of the layers. Since the feature detection is good, I would be inclined to rather work on changing the training to more epochs, increase the `batch_size` to 256, I also could try out a smaller learning rate as well as smoothing.

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.