dlnd_face_generation

September 17, 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
            """

            # TODO: Implement function and return a dataloader
            transform = transforms.Compose([transforms.Resize(image_size),
                                          transforms.ToTensor()])
        image_dataset = datasets.ImageFolder(data_dir, transform)
        data_loader = torch.utils.data.DataLoader(image_dataset, batch_size=batch_size, shuf

        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 = 64 # batch size reduced to 64 according to reviewer comments https://reviev
        img_size = 32

        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        # Call your function and get a dataloader
        celeba_train_loader = get_dataloader(batch_size, img_size)
```

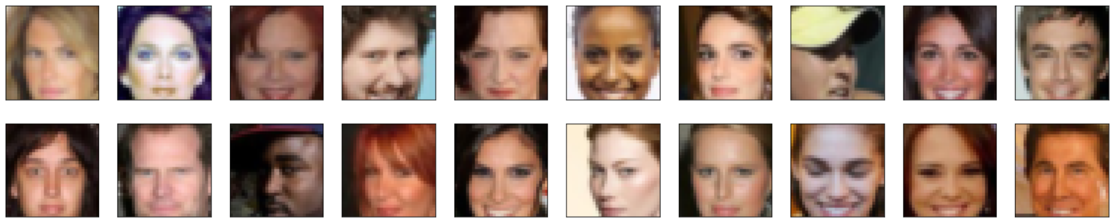
Next, you can view some images! You should seen square images of somewhat-centered faces.

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

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

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

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



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

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

            return out
```

```
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(-0.8275)
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]: def conv(in_channels, out_channels, kernel_size, stride=2, padding=1, batch_norm=True):
             """
             Helper function to quickly create a convolutional layer, optionally with batch norn
             """
             layers = []
             conv_layer = nn.Conv2d(in_channels=in_channels,
                                    out_channels=out_channels,
                                    kernel_size=kernel_size,
                                    stride=stride,
                                    padding=padding,
                                    bias=False)
             layers.append(conv_layer)

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

             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__()
                 self.conv_dim = conv_dim
                 self.conv1 = conv(3, conv_dim, 4, batch_norm=False) # dimensions (16, 16, conv_
                 self.conv2 = conv(conv_dim, conv_dim*2, 4) # dimensions (8, 8, conv_dim*2)
                 self.conv3 = conv(conv_dim*2, conv_dim*4, 4) # dimensions (4, 4, conv_dim*4)
                 self.conv4 = conv(conv_dim*4, conv_dim*8, 4) # dimensions (2, 2, conv_dim*8)

                 self.classifier = nn.Linear(conv_dim*8*2*2, 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
                # Use leaky ReLU instead of regular ReLU
                out = F.leaky_relu(self.conv1(x), 0.2)
                out = F.leaky_relu(self.conv2(out), 0.2)
                out = F.leaky_relu(self.conv3(out), 0.2)
                out = F.leaky_relu(self.conv4(out), 0.2)

                out = out.view(-1, self.conv_dim*8*2*2)
                out = self.classifier(out)
                return out


        """
        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]: def deconv(in_channels, out_channels, kernel_size, stride=2, padding=1, batch_norm=True
             """
             Helper function to quickly create a transpose convolutional layer, optionally with
             """
             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 [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__()

                 # complete init function
                 self.conv_dim = conv_dim

                 self.fc = nn.Linear(z_size, conv_dim*4*4*4)
                 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)
                 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 = F.relu(self.t_conv2(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
```

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
             # First check if submodule has weights
             if hasattr(m, 'weight') and classname.find('Conv') or classname.find('Linear') != -
                 m.weight.data.normal_(0.0, 0.02)
                 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): 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)
  )
  (classifier): Linear(in_features=2048, 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)
)
```

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]: ## Implemented one-sided label smoothing as suggested by reviewer.
         # Consider one-sided label smoothing. It's done to prevent discriminator
         # from being too strong as well as to help it generalise better by reducing labels from

         def real_loss(D_out, smooth=True):
             '''Calculates how close discriminator outputs are to being real.
                param, D_out: discriminator logits
                return: real loss'''
             batch_size = D_out.shape[0]

             if smooth:
                 # smooth labels as suggested by reviewer
                 # prevent discriminator from being too strong
                 # and to help it generalise better
                 # NOTE: smooth is ON by default
                 labels = torch.ones(batch_size) * 0.9
             else:
                 labels = torch.ones(batch_size)

             if train_on_gpu:
                 labels = labels.cuda()

             criterion = nn.BCEWithLogitsLoss()
             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.shape[0]
             labels = torch.zeros(batch_size)

             if train_on_gpu:
                 labels = labels.cuda()

             criterion = nn.BCEWithLogitsLoss()
             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
         # values for lr and beta1 taken from reviewer suggestions:
         # Beta1: 0.5 seems to be the best choice here
         #  as explained in this paper: https://arxiv.org/pdf/1511.06434.pdf
         lr = 0.0005
         beta1 = 0.5
         beta2 = 0.99

         # 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=100):
             '''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
        # ===============================================

        if train_on_gpu:
            real_images = real_images.cuda()

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

        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)

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

        # 2. Train the generator with an adversarial loss
        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)
                g_loss = real_loss(D_fake)

                g_optimizer.zero_grad()
                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]: # Copy in functions from workspace_utils.py file as we had in other projects
         # this lets us persist the Jupyter notebook session during long training runs
```

```python
import signal
from contextlib import contextmanager
import requests


DELAY = INTERVAL = 4 * 60  # interval time in seconds
MIN_DELAY = MIN_INTERVAL = 2 * 60
KEEPALIVE_URL = "https://nebula.udacity.com/api/v1/remote/keep-alive"
TOKEN_URL = "http://metadata.google.internal/computeMetadata/v1/instance/attributes/kee
TOKEN_HEADERS = {"Metadata-Flavor":"Google"}


def _request_handler(headers):
    def _handler(signum, frame):
        requests.request("POST", KEEPALIVE_URL, headers=headers)
    return _handler


@contextmanager
def active_session(delay=DELAY, interval=INTERVAL):
    """
    Example:

    from workspace_utils import active session

    with active_session():
        # do long-running work here
    """
    token = requests.request("GET", TOKEN_URL, headers=TOKEN_HEADERS).text
    headers = {'Authorization': "STAR " + token}
    delay = max(delay, MIN_DELAY)
    interval = max(interval, MIN_INTERVAL)
    original_handler = signal.getsignal(signal.SIGALRM)
    try:
        signal.signal(signal.SIGALRM, _request_handler(headers))
        signal.setitimer(signal.ITIMER_REAL, delay, interval)
        yield
    finally:
        signal.signal(signal.SIGALRM, original_handler)
        signal.setitimer(signal.ITIMER_REAL, 0)


def keep_awake(iterable, delay=DELAY, interval=INTERVAL):
    """
    Example:

    from workspace_utils import keep_awake
```

```
            for i in keep_awake(range(5)):
                # do iteration with lots of work here
            """
            with active_session(delay, interval): yield from iterable
```

```
In [22]: # set number of epochs
         n_epochs = 5


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

```
Epoch [    1/    5] | d_loss: 1.3873 | g_loss: 0.7138
Epoch [    1/    5] | d_loss: 0.8371 | g_loss: 2.0252
Epoch [    1/    5] | d_loss: 1.2354 | g_loss: 2.7778
Epoch [    1/    5] | d_loss: 0.8863 | g_loss: 2.3913
Epoch [    1/    5] | d_loss: 0.8573 | g_loss: 2.3710
Epoch [    1/    5] | d_loss: 1.0336 | g_loss: 3.0748
Epoch [    1/    5] | d_loss: 0.9199 | g_loss: 1.5909
Epoch [    1/    5] | d_loss: 0.8290 | g_loss: 3.6985
Epoch [    1/    5] | d_loss: 1.0093 | g_loss: 3.4507
Epoch [    1/    5] | d_loss: 1.6099 | g_loss: 0.5932
Epoch [    1/    5] | d_loss: 0.7949 | g_loss: 3.5117
Epoch [    1/    5] | d_loss: 0.6062 | g_loss: 4.8264
Epoch [    1/    5] | d_loss: 1.0448 | g_loss: 2.4904
Epoch [    1/    5] | d_loss: 0.5643 | g_loss: 3.5526
Epoch [    1/    5] | d_loss: 1.4800 | g_loss: 1.7626
Epoch [    2/    5] | d_loss: 0.8921 | g_loss: 3.0043
Epoch [    2/    5] | d_loss: 0.7473 | g_loss: 2.9584
Epoch [    2/    5] | d_loss: 0.8502 | g_loss: 2.6375
Epoch [    2/    5] | d_loss: 0.7779 | g_loss: 2.0753
Epoch [    2/    5] | d_loss: 0.7469 | g_loss: 4.4926
Epoch [    2/    5] | d_loss: 0.6515 | g_loss: 3.9602
Epoch [    2/    5] | d_loss: 1.0134 | g_loss: 0.5472
Epoch [    2/    5] | d_loss: 0.6075 | g_loss: 2.9176
Epoch [    2/    5] | d_loss: 0.7026 | g_loss: 2.7524
Epoch [    2/    5] | d_loss: 0.6416 | g_loss: 3.8345
Epoch [    2/    5] | d_loss: 0.6978 | g_loss: 1.9649
Epoch [    2/    5] | d_loss: 0.8137 | g_loss: 2.4185
Epoch [    2/    5] | d_loss: 1.6049 | g_loss: 3.6193
Epoch [    2/    5] | d_loss: 0.8531 | g_loss: 4.2770
Epoch [    2/    5] | d_loss: 1.0194 | g_loss: 2.0933
Epoch [    3/    5] | d_loss: 1.1446 | g_loss: 1.3539
Epoch [    3/    5] | d_loss: 1.6267 | g_loss: 3.6505
Epoch [    3/    5] | d_loss: 0.7808 | g_loss: 2.2602
Epoch [    3/    5] | d_loss: 0.6955 | g_loss: 1.8102
Epoch [    3/    5] | d_loss: 0.6500 | g_loss: 3.7339
Epoch [    3/    5] | d_loss: 1.1105 | g_loss: 1.0559
Epoch [    3/    5] | d_loss: 0.9226 | g_loss: 0.9869
Epoch [    3/    5] | d_loss: 0.6044 | g_loss: 4.6443
Epoch [    3/    5] | d_loss: 0.8624 | g_loss: 4.8968
Epoch [    3/    5] | d_loss: 1.3766 | g_loss: 1.5978
Epoch [    3/    5] | d_loss: 0.8239 | g_loss: 1.8439
Epoch [    3/    5] | d_loss: 0.4964 | g_loss: 4.1755
Epoch [    3/    5] | d_loss: 0.6857 | g_loss: 2.2430
Epoch [    3/    5] | d_loss: 0.7830 | g_loss: 3.7617
Epoch [    3/    5] | d_loss: 0.6886 | g_loss: 1.0220
Epoch [    4/    5] | d_loss: 0.5778 | g_loss: 3.4476
Epoch [    4/    5] | d_loss: 1.0474 | g_loss: 4.0051
Epoch [    4/    5] | d_loss: 0.4837 | g_loss: 3.1511
Epoch [    4/    5] | d_loss: 0.6176 | g_loss: 1.9128
Epoch [    4/    5] | d_loss: 1.2545 | g_loss: 0.9177
Epoch [    4/    5] | d_loss: 0.8811 | g_loss: 3.8571
Epoch [    4/    5] | d_loss: 0.5481 | g_loss: 3.0050
Epoch [    4/    5] | d_loss: 0.7154 | g_loss: 1.4381
Epoch [    4/    5] | d_loss: 0.6954 | g_loss: 1.9759
Epoch [    4/    5] | d_loss: 0.6424 | g_loss: 2.2608
Epoch [    4/    5] | d_loss: 0.7098 | g_loss: 4.1022
Epoch [    4/    5] | d_loss: 0.4823 | g_loss: 3.4520
Epoch [    4/    5] | d_loss: 1.1535 | g_loss: 3.0427
Epoch [    4/    5] | d_loss: 0.6017 | g_loss: 3.3492
Epoch [    4/    5] | d_loss: 0.4840 | g_loss: 3.6042
Epoch [    5/    5] | d_loss: 0.6269 | g_loss: 2.2338
Epoch [    5/    5] | d_loss: 1.6976 | g_loss: 0.6449
Epoch [    5/    5] | d_loss: 1.4855 | g_loss: 6.0430
Epoch [    5/    5] | d_loss: 0.4048 | g_loss: 2.8007
Epoch [    5/    5] | d_loss: 0.5378 | g_loss: 4.2105
Epoch [    5/    5] | d_loss: 0.9573 | g_loss: 3.7084
Epoch [    5/    5] | d_loss: 0.5059 | g_loss: 2.8615
Epoch [    5/    5] | d_loss: 0.5595 | g_loss: 2.0545
Epoch [    5/    5] | d_loss: 0.4535 | g_loss: 1.9877
Epoch [    5/    5] | d_loss: 1.0800 | g_loss: 4.8556
Epoch [    5/    5] | d_loss: 0.4744 | g_loss: 2.8875
Epoch [    5/    5] | d_loss: 0.5316 | g_loss: 2.6183
Epoch [    5/    5] | d_loss: 0.6578 | g_loss: 5.0103
Epoch [    5/    5] | d_loss: 0.4511 | g_loss: 4.3253
Epoch [    5/    5] | d_loss: 0.6428 | g_loss: 4.0961
```

```
In [23]: %%javascript
         IPython.notebook.save_notebook()
```

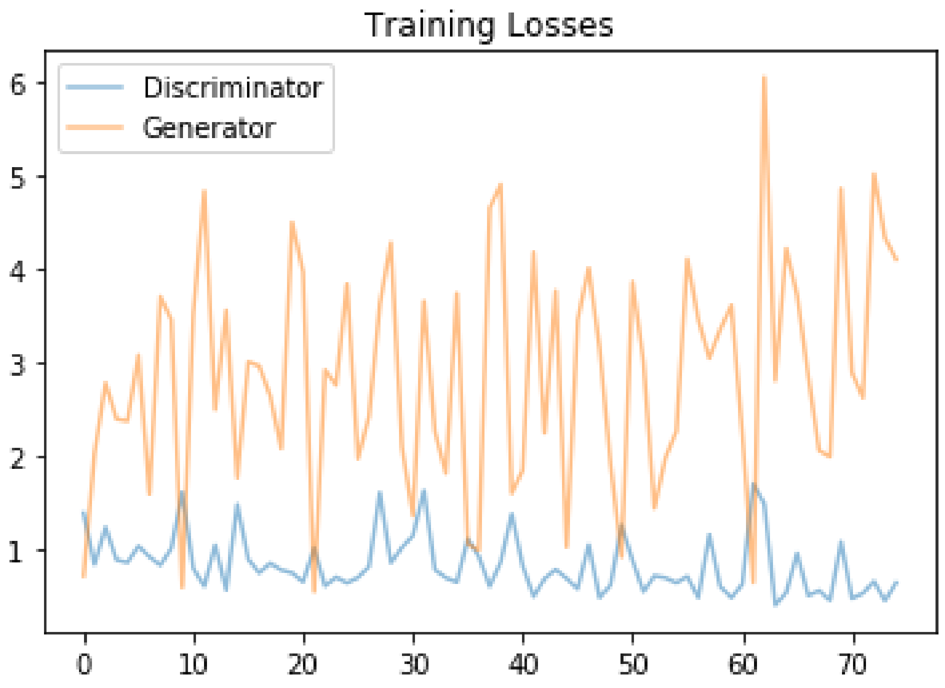
```
<IPython.core.display.Javascript object>
```

2.8 Training loss

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

```
In [24]: 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[24]: <matplotlib.legend.Legend at 0x7fed720a4b70>
```



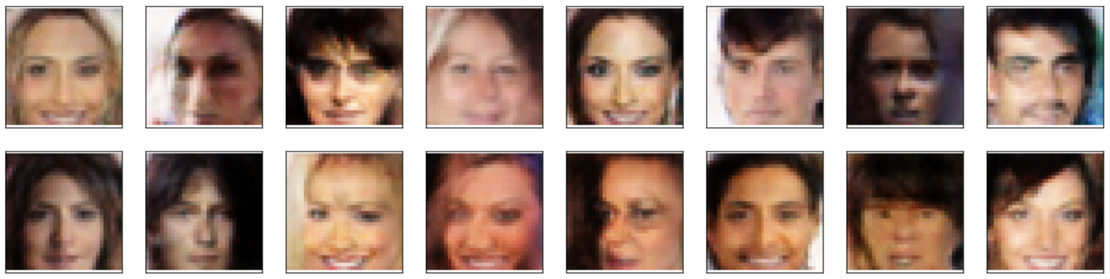
2.9 Generator samples from training

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

```
In [25]: # 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 [26]: # Load samples from generator, taken while training
         with open('train_samples.pkl', 'rb') as f:
             samples = pkl.load(f)
```

```
In [27]: _ = view_samples(-1, samples)
```



```
In [28]: %%javascript
         IPython.notebook.save_notebook()

<IPython.core.display.Javascript object>
```

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)

- The trained GAN will have a latent space representation of human faces that reflect the bias of the dataset (a.k.a. mostly white faces). This is a dataset bias problem which leads to neural network bias (in this case the GAN), and can be seen in the samples. Adding other celebrity faces which are not only white faces can alleviate this issue.
- On a side note, the resolution of the photos are very low, which means that our GAN does not have much high-quality information to train on. Thus, our generated samples also won't look very good or high-quality.
- Larger models will be able to learn the faces at higher fidelity. However, this comes at a cost of much higher computational resource requirement and also we run the risk of overfitting or memorizing faces. I have not tried this yet due to the long time the model needs to train.

- Like stated above, adding more model complexity and training for more epochs may increase model performance, but this carries the high cost of computation time. For me on a laptop GPU this would unfortunately be not possible. Only through the Udacity GPU instance can I do any sort of testing like this (however, that still also takes a long time). Different optimizer hyperparameters may also help to improve the model performance. Due to the time constraint, I have not tried many different combinations.

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.