dlnd_face_generation

July 30, 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
```

```python
import numpy as np
import problem_unittests as tests
#import helper
from torch.nn import init

%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]:
```python
# necessary imports
import torch
from torchvision import datasets
from torchvision import transforms
```

In [4]:
```python
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
    """
    transform=transforms.Compose([transforms.Resize(image_size),
                                  transforms.ToTensor()])
```

```
    image_path='./'+data_dir
    # TODO: Implement function and return a dataloader
    dataset=datasets.ImageFolder(image_path,transform)
    data_loader=torch.utils.data.DataLoader(dataset=dataset,batch_size=batch_size,shuffl
    return data_loader
```

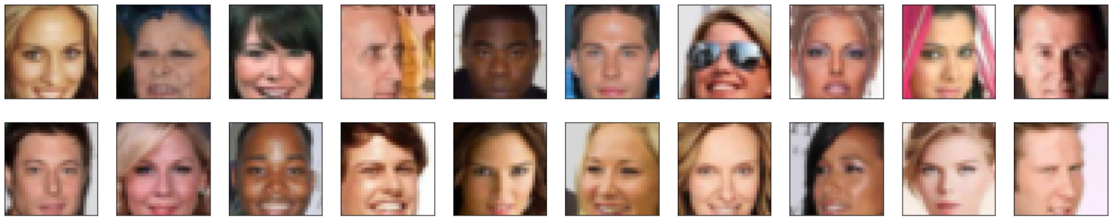
1.2 Create a DataLoader

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 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
    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(-0.9294)
Max:  tensor(0.8980)
```

---

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

def conv(in_channels,out_channels,kernel_size,stride=2,padding=1,batch_norm=True):
    layers=[]
    conv_layer=nn.Conv2d(in_channels,out_channels,kernel_size,stride,padding)
    layers.append(conv_layer)
    if batch_norm:
        layers.append(nn.BatchNorm2d(out_channels))
    return nn.Sequential(*layers)
```

In [10]:
```
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__()
        #(32,32,3)
        self.conv_dim=conv_dim
        self.conv1=conv(3,conv_dim,4,batch_norm=False)#(16,16,conv_dim)
        self.conv2=conv(conv_dim,conv_dim*2,4)#(8,8,conv_dim*2)
        self.conv3=conv(conv_dim*2,conv_dim*4,4)#(4,4,conv_dim*4)
        self.conv4=conv(conv_dim*4,conv_dim*8,4)#(2,2,conv_dim*8)
        #self.conv5=conv(conv_dim*8,1,4,stride=1,batch_norm=False)#(1,1,1)
        # complete init function
        self.fc=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
        x=F.relu(self.conv1(x))
```

```
            x=F.relu(self.conv2(x))
            x=F.relu(self.conv3(x))
            x=F.relu(self.conv4(x))
            x=x.view(-1,self.conv_dim*8*2*2)
            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 [11]: # 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__()
        self.conv_dim=conv_dim
        self.fc=nn.Linear(z_size,conv_dim*8*2*2)
        self.t_conv1=deconv(conv_dim*8,conv_dim*4,4)
        self.t_conv2=deconv(conv_dim*4,conv_dim*2,4)
        self.t_conv3=deconv(conv_dim*2,conv_dim,4)
        self.t_conv4=deconv(conv_dim,3,4,batch_norm=False)
        # complete init function


    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 = x.view(-1, self.conv_dim*8, 2, 2)
        x=F.relu(self.t_conv1(x))
        x=F.relu(self.t_conv2(x))
        x=F.relu(self.t_conv3(x))
        x=F.tanh(self.t_conv4(x))

        return x

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

```
Tests Passed
```

2.3 Initialize the weights of your networks

To help your models converge, you should initialize the weights of the convolutional and linear layers in your model. From reading the original DCGAN paper, they say: > All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02.

So, your next task will be to define a weight initialization function that does just this!

You can refer back to the lesson on weight initialization or even consult existing model code, such as that from the `networks.py` file in CycleGAN Github repository to help you complete this function.

Exercise: Complete the weight initialization function

- This should initialize only **convolutional** and **linear** layers
- Initialize the weights to a normal distribution, centered around 0, with a standard deviation of 0.02.
- The bias terms, if they exist, may be left alone or set to 0.

```
In [12]: 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 hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear
                         init.normal_(m.weight.data, 0.0, 0.02)
```

2.4 Build complete network

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

```
In [13]: """
         DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
         """
         def build_network(d_conv_dim, g_conv_dim, z_size):
             # define discriminator and generator
             D = Discriminator(d_conv_dim)
             G = Generator(z_size=z_size, conv_dim=g_conv_dim)

             # initialize model weights
             D.apply(weights_init_normal)
             G.apply(weights_init_normal)

             print(D)
             print()
             print(G)

             return D, G
```

Exercise: Define model hyperparameters

```
In [14]: # 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))
  )
  (conv2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (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))
    (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))
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (fc): Linear(in_features=2048, out_features=1, bias=True)
)

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

)

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 [15]: """
         DON'T MODIFY ANYTHING IN THIS CELL
         """
         import torch

         # Check for a GPU
         train_on_gpu = torch.cuda.is_available()
         if not train_on_gpu:
             print('No GPU found. Please use a GPU to train your neural network.')
         else:
             print('Training on GPU!')
```

```
Training on GPU!
```

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 [16]: def real_loss(D_out):
             '''Calculates how close discriminator outputs are to being real.
                param, D_out: discriminator logits
```

```
        return: real loss'''
    batch_size=D_out.size(0)
    labels=torch.ones(batch_size)
    if train_on_gpu:
        labels = labels.cuda()
    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.size(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 [17]: import torch.optim as optim
         lr =0.0001
         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 [18]: 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
                     # ===============================================
                     #model,loss inputs,model parameters
                     # 1. Train the discriminator on real and fake images
                     if train_on_gpu:
                         real_images = real_images.cuda()
                     d_optimizer.zero_grad()
                     d_real=D(real_images)
                     d_real_loss = real_loss(d_real)
```

```python
        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_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()
        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_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 [ ]: # set number of epochs
        n_epochs = 30


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

```
Epoch [    1/   30] | d_loss: 1.4302 | g_loss: 1.1209
Epoch [    1/   30] | d_loss: 0.0229 | g_loss: 5.4185
Epoch [    1/   30] | d_loss: 0.0136 | g_loss: 5.9364
Epoch [    1/   30] | d_loss: 0.0893 | g_loss: 4.0562
Epoch [    1/   30] | d_loss: 0.4106 | g_loss: 2.1254
Epoch [    1/   30] | d_loss: 0.2894 | g_loss: 3.9335
Epoch [    1/   30] | d_loss: 0.2366 | g_loss: 4.8584
Epoch [    1/   30] | d_loss: 0.2668 | g_loss: 4.5111
Epoch [    1/   30] | d_loss: 0.3121 | g_loss: 3.8714
Epoch [    1/   30] | d_loss: 0.6781 | g_loss: 4.2079
Epoch [    1/   30] | d_loss: 0.7427 | g_loss: 3.5638
Epoch [    1/   30] | d_loss: 0.5774 | g_loss: 2.5724
Epoch [    1/   30] | d_loss: 0.6520 | g_loss: 3.6680
Epoch [    1/   30] | d_loss: 0.6874 | g_loss: 2.4949
Epoch [    1/   30] | d_loss: 0.4734 | g_loss: 3.6043
Epoch [    2/   30] | d_loss: 0.6737 | g_loss: 4.2524
Epoch [    2/   30] | d_loss: 0.4832 | g_loss: 3.1472
Epoch [    2/   30] | d_loss: 0.5448 | g_loss: 2.9717
Epoch [    2/   30] | d_loss: 0.5693 | g_loss: 2.7880
Epoch [    2/   30] | d_loss: 0.7660 | g_loss: 2.0470
Epoch [    2/   30] | d_loss: 0.8745 | g_loss: 2.1320
Epoch [    2/   30] | d_loss: 0.5297 | g_loss: 2.6997
Epoch [    2/   30] | d_loss: 0.6612 | g_loss: 3.1633
Epoch [    2/   30] | d_loss: 0.5794 | g_loss: 2.8597
Epoch [    2/   30] | d_loss: 0.5211 | g_loss: 2.7478
Epoch [    2/   30] | d_loss: 0.8273 | g_loss: 2.4318
Epoch [    2/   30] | d_loss: 0.6400 | g_loss: 2.0155
Epoch [    2/   30] | d_loss: 0.7229 | g_loss: 1.5570
Epoch [    2/   30] | d_loss: 0.5936 | g_loss: 2.2120
```

```
Epoch [    2/   30] | d_loss: 0.6387 | g_loss: 2.1705
Epoch [    3/   30] | d_loss: 0.6189 | g_loss: 1.6740
Epoch [    3/   30] | d_loss: 0.5783 | g_loss: 2.5163
Epoch [    3/   30] | d_loss: 0.5740 | g_loss: 2.0923
Epoch [    3/   30] | d_loss: 0.8128 | g_loss: 1.6608
Epoch [    3/   30] | d_loss: 0.8103 | g_loss: 1.8471
Epoch [    3/   30] | d_loss: 0.5876 | g_loss: 2.6625
Epoch [    3/   30] | d_loss: 0.7718 | g_loss: 2.4507
Epoch [    3/   30] | d_loss: 0.6522 | g_loss: 2.0079
Epoch [    3/   30] | d_loss: 1.8103 | g_loss: 3.8837
Epoch [    3/   30] | d_loss: 0.5504 | g_loss: 2.1797
Epoch [    3/   30] | d_loss: 0.8102 | g_loss: 1.3474
Epoch [    3/   30] | d_loss: 0.6861 | g_loss: 1.9335
Epoch [    3/   30] | d_loss: 0.6052 | g_loss: 1.6894
Epoch [    3/   30] | d_loss: 0.6605 | g_loss: 2.2912
Epoch [    3/   30] | d_loss: 0.7661 | g_loss: 1.5209
Epoch [    4/   30] | d_loss: 0.6749 | g_loss: 2.1011
Epoch [    4/   30] | d_loss: 0.7348 | g_loss: 1.9324
Epoch [    4/   30] | d_loss: 0.7422 | g_loss: 2.6675
Epoch [    4/   30] | d_loss: 0.8928 | g_loss: 3.4010
Epoch [    4/   30] | d_loss: 1.3045 | g_loss: 0.5952
Epoch [    4/   30] | d_loss: 0.7828 | g_loss: 1.6450
Epoch [    4/   30] | d_loss: 1.5737 | g_loss: 3.8742
Epoch [    4/   30] | d_loss: 0.7430 | g_loss: 2.5177
Epoch [    4/   30] | d_loss: 1.2476 | g_loss: 3.7343
Epoch [    4/   30] | d_loss: 0.5444 | g_loss: 2.1865
Epoch [    4/   30] | d_loss: 0.7389 | g_loss: 1.6979
Epoch [    4/   30] | d_loss: 0.6133 | g_loss: 1.3526
Epoch [    4/   30] | d_loss: 0.5617 | g_loss: 1.5918
Epoch [    4/   30] | d_loss: 0.6698 | g_loss: 1.9589
Epoch [    4/   30] | d_loss: 0.9088 | g_loss: 1.1061
Epoch [    5/   30] | d_loss: 0.5118 | g_loss: 1.5496
Epoch [    5/   30] | d_loss: 0.4809 | g_loss: 2.5281
Epoch [    5/   30] | d_loss: 0.6689 | g_loss: 1.8224
Epoch [    5/   30] | d_loss: 0.6500 | g_loss: 1.6294
Epoch [    5/   30] | d_loss: 0.6680 | g_loss: 2.0143
Epoch [    5/   30] | d_loss: 0.3661 | g_loss: 2.2704
Epoch [    5/   30] | d_loss: 1.1068 | g_loss: 1.2407
Epoch [    5/   30] | d_loss: 0.4139 | g_loss: 1.6422
Epoch [    5/   30] | d_loss: 0.7802 | g_loss: 2.6824
Epoch [    5/   30] | d_loss: 0.5898 | g_loss: 2.8917
Epoch [    5/   30] | d_loss: 0.5947 | g_loss: 1.8015
Epoch [    5/   30] | d_loss: 0.7414 | g_loss: 2.3615
Epoch [    5/   30] | d_loss: 0.9932 | g_loss: 1.4245
Epoch [    5/   30] | d_loss: 0.6349 | g_loss: 1.4195
Epoch [    5/   30] | d_loss: 1.3092 | g_loss: 0.9058
Epoch [    6/   30] | d_loss: 0.5579 | g_loss: 2.1486
Epoch [    6/   30] | d_loss: 0.6967 | g_loss: 1.9528
```

```
Epoch [    6/   30] | d_loss: 0.5970 | g_loss: 2.5189
Epoch [    6/   30] | d_loss: 1.1296 | g_loss: 3.0465
Epoch [    6/   30] | d_loss: 0.9997 | g_loss: 0.8241
Epoch [    6/   30] | d_loss: 1.2012 | g_loss: 4.6179
Epoch [    6/   30] | d_loss: 0.6552 | g_loss: 1.6493
Epoch [    6/   30] | d_loss: 0.5130 | g_loss: 2.8935
Epoch [    6/   30] | d_loss: 0.5110 | g_loss: 2.2839
Epoch [    6/   30] | d_loss: 0.6162 | g_loss: 1.3939
Epoch [    6/   30] | d_loss: 0.5092 | g_loss: 1.2407
Epoch [    6/   30] | d_loss: 0.5197 | g_loss: 2.0113
Epoch [    6/   30] | d_loss: 0.5462 | g_loss: 1.5772
Epoch [    6/   30] | d_loss: 0.5561 | g_loss: 1.9710
Epoch [    6/   30] | d_loss: 1.2481 | g_loss: 1.6505
Epoch [    7/   30] | d_loss: 1.1312 | g_loss: 0.3651
Epoch [    7/   30] | d_loss: 0.4997 | g_loss: 1.2344
Epoch [    7/   30] | d_loss: 0.6572 | g_loss: 1.8346
Epoch [    7/   30] | d_loss: 0.6242 | g_loss: 2.2032
Epoch [    7/   30] | d_loss: 0.8257 | g_loss: 2.8191
Epoch [    7/   30] | d_loss: 1.3818 | g_loss: 4.6477
Epoch [    7/   30] | d_loss: 0.7420 | g_loss: 3.0796
Epoch [    7/   30] | d_loss: 0.5719 | g_loss: 2.8768
Epoch [    7/   30] | d_loss: 0.5081 | g_loss: 2.4079
Epoch [    7/   30] | d_loss: 1.4932 | g_loss: 4.0698
Epoch [    7/   30] | d_loss: 1.2009 | g_loss: 1.3540
Epoch [    7/   30] | d_loss: 0.4842 | g_loss: 3.7238
Epoch [    7/   30] | d_loss: 0.3884 | g_loss: 2.2590
Epoch [    7/   30] | d_loss: 0.5001 | g_loss: 3.1914
Epoch [    7/   30] | d_loss: 0.5101 | g_loss: 3.0997
Epoch [    8/   30] | d_loss: 1.1526 | g_loss: 1.1025
Epoch [    8/   30] | d_loss: 0.3759 | g_loss: 2.8005
Epoch [    8/   30] | d_loss: 0.6524 | g_loss: 2.9467
Epoch [    8/   30] | d_loss: 0.5422 | g_loss: 2.2259
Epoch [    8/   30] | d_loss: 0.5558 | g_loss: 2.4345
Epoch [    8/   30] | d_loss: 0.5075 | g_loss: 1.9224
Epoch [    8/   30] | d_loss: 1.1718 | g_loss: 4.0443
Epoch [    8/   30] | d_loss: 0.6819 | g_loss: 0.7574
Epoch [    8/   30] | d_loss: 0.6515 | g_loss: 1.2267
Epoch [    8/   30] | d_loss: 0.7656 | g_loss: 1.5941
Epoch [    8/   30] | d_loss: 0.8120 | g_loss: 3.0907
Epoch [    8/   30] | d_loss: 0.7099 | g_loss: 1.2468
Epoch [    8/   30] | d_loss: 0.6745 | g_loss: 2.4188
Epoch [    8/   30] | d_loss: 0.4390 | g_loss: 1.7009
Epoch [    8/   30] | d_loss: 0.6676 | g_loss: 2.3634
Epoch [    9/   30] | d_loss: 0.7428 | g_loss: 2.1073
Epoch [    9/   30] | d_loss: 0.4213 | g_loss: 0.6805
Epoch [    9/   30] | d_loss: 0.6915 | g_loss: 3.7512
Epoch [    9/   30] | d_loss: 0.3527 | g_loss: 2.1598
Epoch [    9/   30] | d_loss: 0.4520 | g_loss: 1.1388
```

```
Epoch [    9/   30] | d_loss: 0.5319 | g_loss: 1.7918
Epoch [    9/   30] | d_loss: 0.6344 | g_loss: 3.1254
Epoch [    9/   30] | d_loss: 0.6502 | g_loss: 1.2882
Epoch [    9/   30] | d_loss: 0.5620 | g_loss: 3.1610
Epoch [    9/   30] | d_loss: 0.4087 | g_loss: 2.1027
Epoch [    9/   30] | d_loss: 0.4449 | g_loss: 3.4761
Epoch [    9/   30] | d_loss: 0.6671 | g_loss: 1.0270
Epoch [    9/   30] | d_loss: 0.4573 | g_loss: 1.8874
Epoch [    9/   30] | d_loss: 1.1848 | g_loss: 2.8689
Epoch [    9/   30] | d_loss: 0.2500 | g_loss: 3.0485
Epoch [   10/   30] | d_loss: 0.8341 | g_loss: 2.2006
Epoch [   10/   30] | d_loss: 0.5932 | g_loss: 3.5261
Epoch [   10/   30] | d_loss: 0.4848 | g_loss: 3.4772
Epoch [   10/   30] | d_loss: 0.3726 | g_loss: 2.3402
Epoch [   10/   30] | d_loss: 0.7312 | g_loss: 2.2311
Epoch [   10/   30] | d_loss: 0.3854 | g_loss: 2.2759
Epoch [   10/   30] | d_loss: 0.4751 | g_loss: 2.7093
Epoch [   10/   30] | d_loss: 0.3361 | g_loss: 3.1835
Epoch [   10/   30] | d_loss: 0.5041 | g_loss: 1.6778
Epoch [   10/   30] | d_loss: 0.4144 | g_loss: 2.7549
Epoch [   10/   30] | d_loss: 0.3276 | g_loss: 3.6350
Epoch [   10/   30] | d_loss: 0.6110 | g_loss: 3.3204
Epoch [   10/   30] | d_loss: 1.1571 | g_loss: 3.3143
Epoch [   10/   30] | d_loss: 1.0082 | g_loss: 2.3235
Epoch [   10/   30] | d_loss: 0.3602 | g_loss: 3.2859
Epoch [   11/   30] | d_loss: 0.4816 | g_loss: 4.1054
Epoch [   11/   30] | d_loss: 0.4404 | g_loss: 2.1623
Epoch [   11/   30] | d_loss: 0.3151 | g_loss: 3.6197
Epoch [   11/   30] | d_loss: 0.5300 | g_loss: 2.7388
Epoch [   11/   30] | d_loss: 0.7534 | g_loss: 3.6443
Epoch [   11/   30] | d_loss: 0.5926 | g_loss: 3.7755
Epoch [   11/   30] | d_loss: 1.2991 | g_loss: 3.6795
Epoch [   11/   30] | d_loss: 0.6038 | g_loss: 3.0726
Epoch [   11/   30] | d_loss: 0.4840 | g_loss: 2.3923
Epoch [   11/   30] | d_loss: 0.9674 | g_loss: 0.3775
Epoch [   11/   30] | d_loss: 0.6245 | g_loss: 2.5706
Epoch [   11/   30] | d_loss: 0.2668 | g_loss: 2.8192
Epoch [   11/   30] | d_loss: 0.2212 | g_loss: 2.6156
Epoch [   11/   30] | d_loss: 0.3354 | g_loss: 3.3477
Epoch [   11/   30] | d_loss: 0.3806 | g_loss: 2.0161
Epoch [   12/   30] | d_loss: 0.9631 | g_loss: 1.1113
Epoch [   12/   30] | d_loss: 0.6678 | g_loss: 2.6118
Epoch [   12/   30] | d_loss: 0.4482 | g_loss: 2.8481
Epoch [   12/   30] | d_loss: 0.2242 | g_loss: 2.9106
Epoch [   12/   30] | d_loss: 0.9893 | g_loss: 4.3604
Epoch [   12/   30] | d_loss: 0.5692 | g_loss: 3.8446
Epoch [   12/   30] | d_loss: 0.4083 | g_loss: 3.9637
Epoch [   12/   30] | d_loss: 0.5697 | g_loss: 1.4819
```

```
Epoch [   12/   30] | d_loss: 0.3234 | g_loss: 3.7925
Epoch [   12/   30] | d_loss: 0.7629 | g_loss: 1.6370
Epoch [   12/   30] | d_loss: 0.3394 | g_loss: 2.9260
Epoch [   12/   30] | d_loss: 0.4702 | g_loss: 3.9673
Epoch [   12/   30] | d_loss: 0.5405 | g_loss: 2.4049
Epoch [   12/   30] | d_loss: 0.7601 | g_loss: 3.0436
Epoch [   12/   30] | d_loss: 1.4048 | g_loss: 0.9690
Epoch [   13/   30] | d_loss: 0.7319 | g_loss: 2.0302
Epoch [   13/   30] | d_loss: 0.2851 | g_loss: 3.4823
Epoch [   13/   30] | d_loss: 0.3025 | g_loss: 2.0639
Epoch [   13/   30] | d_loss: 0.6758 | g_loss: 3.6509
Epoch [   13/   30] | d_loss: 0.4795 | g_loss: 0.9764
Epoch [   13/   30] | d_loss: 0.2336 | g_loss: 2.7176
Epoch [   13/   30] | d_loss: 0.6971 | g_loss: 3.7150
Epoch [   13/   30] | d_loss: 0.2606 | g_loss: 3.5191
Epoch [   13/   30] | d_loss: 0.6042 | g_loss: 1.7930
Epoch [   13/   30] | d_loss: 0.1868 | g_loss: 3.1445
Epoch [   13/   30] | d_loss: 1.0204 | g_loss: 2.1769
Epoch [   13/   30] | d_loss: 2.0285 | g_loss: 0.9614
Epoch [   13/   30] | d_loss: 1.2719 | g_loss: 0.6003
Epoch [   13/   30] | d_loss: 0.6898 | g_loss: 0.8581
Epoch [   13/   30] | d_loss: 0.4155 | g_loss: 2.2876
Epoch [   14/   30] | d_loss: 0.3286 | g_loss: 2.2132
Epoch [   14/   30] | d_loss: 0.5784 | g_loss: 3.1743
Epoch [   14/   30] | d_loss: 0.3122 | g_loss: 3.3770
Epoch [   14/   30] | d_loss: 0.2665 | g_loss: 5.1789
Epoch [   14/   30] | d_loss: 0.5451 | g_loss: 2.7998
Epoch [   14/   30] | d_loss: 0.8403 | g_loss: 4.9059
Epoch [   14/   30] | d_loss: 0.0764 | g_loss: 1.5873
Epoch [   14/   30] | d_loss: 0.6635 | g_loss: 3.6573
Epoch [   14/   30] | d_loss: 0.3114 | g_loss: 4.0825
Epoch [   14/   30] | d_loss: 0.2709 | g_loss: 4.9224
Epoch [   14/   30] | d_loss: 0.1660 | g_loss: 3.7465
Epoch [   14/   30] | d_loss: 0.2287 | g_loss: 1.6542
Epoch [   14/   30] | d_loss: 0.1926 | g_loss: 4.3092
Epoch [   14/   30] | d_loss: 0.3515 | g_loss: 4.4306
Epoch [   14/   30] | d_loss: 0.5976 | g_loss: 3.5197
Epoch [   15/   30] | d_loss: 0.3410 | g_loss: 2.5390
Epoch [   15/   30] | d_loss: 0.1284 | g_loss: 3.5202
Epoch [   15/   30] | d_loss: 0.2983 | g_loss: 4.4421
Epoch [   15/   30] | d_loss: 0.1778 | g_loss: 3.6580
Epoch [   15/   30] | d_loss: 0.1500 | g_loss: 4.8011
Epoch [   15/   30] | d_loss: 0.3606 | g_loss: 1.1824
Epoch [   15/   30] | d_loss: 0.3242 | g_loss: 2.2871
Epoch [   15/   30] | d_loss: 1.0599 | g_loss: 1.8298
Epoch [   15/   30] | d_loss: 0.2078 | g_loss: 2.3256
Epoch [   15/   30] | d_loss: 0.3071 | g_loss: 1.2375
Epoch [   15/   30] | d_loss: 0.1110 | g_loss: 3.3778
```

```
Epoch [   15/   30] | d_loss: 0.3261 | g_loss: 3.0524
Epoch [   15/   30] | d_loss: 0.4414 | g_loss: 2.3285
Epoch [   15/   30] | d_loss: 0.4073 | g_loss: 1.6603
Epoch [   15/   30] | d_loss: 0.4905 | g_loss: 3.6640
Epoch [   16/   30] | d_loss: 0.1464 | g_loss: 1.5062
Epoch [   16/   30] | d_loss: 0.1766 | g_loss: 3.7671
Epoch [   16/   30] | d_loss: 1.0566 | g_loss: 0.2282
Epoch [   16/   30] | d_loss: 0.2983 | g_loss: 2.7231
Epoch [   16/   30] | d_loss: 0.7642 | g_loss: 5.5232
Epoch [   16/   30] | d_loss: 0.5174 | g_loss: 3.2670
Epoch [   16/   30] | d_loss: 0.2652 | g_loss: 3.7781
Epoch [   16/   30] | d_loss: 0.2056 | g_loss: 3.0359
Epoch [   16/   30] | d_loss: 0.3799 | g_loss: 3.6539
Epoch [   16/   30] | d_loss: 0.1669 | g_loss: 3.3139
Epoch [   16/   30] | d_loss: 0.2632 | g_loss: 2.3479
Epoch [   16/   30] | d_loss: 0.1198 | g_loss: 4.5247
Epoch [   16/   30] | d_loss: 0.1412 | g_loss: 2.1575
Epoch [   16/   30] | d_loss: 0.5180 | g_loss: 1.3914
Epoch [   16/   30] | d_loss: 0.3906 | g_loss: 1.7644
Epoch [   17/   30] | d_loss: 0.6403 | g_loss: 5.5299
Epoch [   17/   30] | d_loss: 0.1853 | g_loss: 2.8204
Epoch [   17/   30] | d_loss: 0.1191 | g_loss: 3.6029
Epoch [   17/   30] | d_loss: 0.3638 | g_loss: 2.7401
Epoch [   17/   30] | d_loss: 0.8504 | g_loss: 3.8306
Epoch [   17/   30] | d_loss: 0.3498 | g_loss: 2.3895
Epoch [   17/   30] | d_loss: 0.3605 | g_loss: 2.8570
Epoch [   17/   30] | d_loss: 0.7550 | g_loss: 1.4558
Epoch [   17/   30] | d_loss: 0.4618 | g_loss: 2.5363
Epoch [   17/   30] | d_loss: 2.2614 | g_loss: 0.9537
Epoch [   17/   30] | d_loss: 0.2223 | g_loss: 2.9174
Epoch [   17/   30] | d_loss: 0.1796 | g_loss: 3.6544
Epoch [   17/   30] | d_loss: 2.2601 | g_loss: 3.3906
Epoch [   17/   30] | d_loss: 0.4217 | g_loss: 2.9196
Epoch [   17/   30] | d_loss: 0.8008 | g_loss: 1.3239
Epoch [   18/   30] | d_loss: 0.1498 | g_loss: 4.2421
Epoch [   18/   30] | d_loss: 0.3548 | g_loss: 1.6452
Epoch [   18/   30] | d_loss: 0.2060 | g_loss: 2.1091
Epoch [   18/   30] | d_loss: 0.2219 | g_loss: 2.5935
Epoch [   18/   30] | d_loss: 0.7807 | g_loss: 3.8886
Epoch [   18/   30] | d_loss: 0.3766 | g_loss: 3.2146
Epoch [   18/   30] | d_loss: 0.1993 | g_loss: 3.9395
Epoch [   18/   30] | d_loss: 0.3982 | g_loss: 4.4152
Epoch [   18/   30] | d_loss: 0.5718 | g_loss: 2.6378
Epoch [   18/   30] | d_loss: 0.7564 | g_loss: 2.9956
Epoch [   18/   30] | d_loss: 0.5649 | g_loss: 3.1878
Epoch [   18/   30] | d_loss: 0.3969 | g_loss: 2.8150
Epoch [   18/   30] | d_loss: 0.2832 | g_loss: 3.7810
Epoch [   18/   30] | d_loss: 0.3619 | g_loss: 2.9027
```

```
Epoch [   18/   30] | d_loss: 0.2203 | g_loss: 1.7723
Epoch [   19/   30] | d_loss: 0.0934 | g_loss: 2.6548
Epoch [   19/   30] | d_loss: 0.3240 | g_loss: 2.7317
Epoch [   19/   30] | d_loss: 0.1974 | g_loss: 2.5571
Epoch [   19/   30] | d_loss: 0.8844 | g_loss: 0.6871
Epoch [   19/   30] | d_loss: 0.6236 | g_loss: 4.2751
Epoch [   19/   30] | d_loss: 0.4718 | g_loss: 1.3532
Epoch [   19/   30] | d_loss: 0.4206 | g_loss: 2.5370
Epoch [   19/   30] | d_loss: 0.1893 | g_loss: 2.6743
Epoch [   19/   30] | d_loss: 1.6137 | g_loss: 2.5135
Epoch [   19/   30] | d_loss: 0.0899 | g_loss: 3.8439
Epoch [   19/   30] | d_loss: 0.1761 | g_loss: 5.3878
Epoch [   19/   30] | d_loss: 0.5693 | g_loss: 3.2470
Epoch [   19/   30] | d_loss: 0.0487 | g_loss: 3.9015
Epoch [   19/   30] | d_loss: 0.2343 | g_loss: 1.3799
Epoch [   19/   30] | d_loss: 0.1376 | g_loss: 2.7522
Epoch [   20/   30] | d_loss: 0.4556 | g_loss: 3.3132
Epoch [   20/   30] | d_loss: 0.6810 | g_loss: 3.4933
Epoch [   20/   30] | d_loss: 0.1144 | g_loss: 4.7776
Epoch [   20/   30] | d_loss: 0.2687 | g_loss: 1.1358
Epoch [   20/   30] | d_loss: 0.2194 | g_loss: 3.4811
Epoch [   20/   30] | d_loss: 0.0949 | g_loss: 3.0775
Epoch [   20/   30] | d_loss: 0.7591 | g_loss: 4.8218
Epoch [   20/   30] | d_loss: 0.1572 | g_loss: 3.6559
Epoch [   20/   30] | d_loss: 0.1184 | g_loss: 3.5778
Epoch [   20/   30] | d_loss: 0.1537 | g_loss: 1.8080
Epoch [   20/   30] | d_loss: 0.2719 | g_loss: 1.3949
Epoch [   20/   30] | d_loss: 0.4601 | g_loss: 1.6249
Epoch [   20/   30] | d_loss: 0.2935 | g_loss: 3.7512
Epoch [   20/   30] | d_loss: 0.3542 | g_loss: 3.4336
Epoch [   20/   30] | d_loss: 0.2238 | g_loss: 2.6161
Epoch [   21/   30] | d_loss: 1.2488 | g_loss: 3.6288
Epoch [   21/   30] | d_loss: 0.2510 | g_loss: 3.3689
Epoch [   21/   30] | d_loss: 0.2614 | g_loss: 3.6168
Epoch [   21/   30] | d_loss: 0.2364 | g_loss: 2.0196
Epoch [   21/   30] | d_loss: 0.3664 | g_loss: 2.0372
Epoch [   21/   30] | d_loss: 1.0295 | g_loss: 2.1804
Epoch [   21/   30] | d_loss: 1.1003 | g_loss: 5.1491
Epoch [   21/   30] | d_loss: 0.2642 | g_loss: 4.1956
Epoch [   21/   30] | d_loss: 0.2374 | g_loss: 3.1665
Epoch [   21/   30] | d_loss: 0.4214 | g_loss: 8.0177
Epoch [   21/   30] | d_loss: 0.0913 | g_loss: 5.6753
Epoch [   21/   30] | d_loss: 0.5846 | g_loss: 3.8030
Epoch [   21/   30] | d_loss: 0.6788 | g_loss: 3.3307
Epoch [   21/   30] | d_loss: 0.6920 | g_loss: 1.0704
Epoch [   21/   30] | d_loss: 0.5270 | g_loss: 3.0409
Epoch [   22/   30] | d_loss: 0.2815 | g_loss: 2.9725
Epoch [   22/   30] | d_loss: 0.2086 | g_loss: 2.2494
```

```
Epoch [   22/   30] | d_loss: 0.5422 | g_loss: 3.8758
Epoch [   22/   30] | d_loss: 0.3430 | g_loss: 1.9094
Epoch [   22/   30] | d_loss: 0.5860 | g_loss: 3.1969
Epoch [   22/   30] | d_loss: 0.1991 | g_loss: 2.7265
Epoch [   22/   30] | d_loss: 0.3771 | g_loss: 3.8528
Epoch [   22/   30] | d_loss: 0.8869 | g_loss: 4.2992
Epoch [   22/   30] | d_loss: 0.7597 | g_loss: 4.3551
Epoch [   22/   30] | d_loss: 0.3331 | g_loss: 6.2854
Epoch [   22/   30] | d_loss: 0.4357 | g_loss: 2.2016
Epoch [   22/   30] | d_loss: 0.0841 | g_loss: 3.7764
Epoch [   22/   30] | d_loss: 0.2675 | g_loss: 2.1605
Epoch [   22/   30] | d_loss: 0.1669 | g_loss: 5.2854
Epoch [   22/   30] | d_loss: 0.3626 | g_loss: 2.0660
Epoch [   23/   30] | d_loss: 0.1582 | g_loss: 3.3396
Epoch [   23/   30] | d_loss: 1.0684 | g_loss: 2.0790
Epoch [   23/   30] | d_loss: 0.3752 | g_loss: 3.2454
Epoch [   23/   30] | d_loss: 0.1423 | g_loss: 1.8223
Epoch [   23/   30] | d_loss: 0.0672 | g_loss: 3.0820
Epoch [   23/   30] | d_loss: 0.3616 | g_loss: 3.6734
Epoch [   23/   30] | d_loss: 0.1037 | g_loss: 3.9220
Epoch [   23/   30] | d_loss: 0.2593 | g_loss: 2.9039
Epoch [   23/   30] | d_loss: 0.0946 | g_loss: 4.3956
Epoch [   23/   30] | d_loss: 0.7316 | g_loss: 3.3325
Epoch [   23/   30] | d_loss: 0.3928 | g_loss: 2.8027
Epoch [   23/   30] | d_loss: 0.3625 | g_loss: 3.8922
Epoch [   23/   30] | d_loss: 0.1192 | g_loss: 3.7193
Epoch [   23/   30] | d_loss: 0.1345 | g_loss: 5.1719
Epoch [   23/   30] | d_loss: 0.5496 | g_loss: 0.6445
Epoch [   24/   30] | d_loss: 0.4840 | g_loss: 1.5710
Epoch [   24/   30] | d_loss: 0.1259 | g_loss: 2.6169
Epoch [   24/   30] | d_loss: 0.8298 | g_loss: 0.1950
Epoch [   24/   30] | d_loss: 0.6171 | g_loss: 3.1595
Epoch [   24/   30] | d_loss: 0.2318 | g_loss: 1.3587
Epoch [   24/   30] | d_loss: 0.2200 | g_loss: 3.7393
Epoch [   24/   30] | d_loss: 0.1826 | g_loss: 2.6831
Epoch [   24/   30] | d_loss: 0.0696 | g_loss: 3.3546
Epoch [   24/   30] | d_loss: 0.1736 | g_loss: 2.7255
Epoch [   24/   30] | d_loss: 0.2071 | g_loss: 3.5527
Epoch [   24/   30] | d_loss: 0.1221 | g_loss: 2.0819
Epoch [   24/   30] | d_loss: 0.9505 | g_loss: 2.7224
Epoch [   24/   30] | d_loss: 0.3637 | g_loss: 4.0501
Epoch [   24/   30] | d_loss: 1.0567 | g_loss: 3.5445
Epoch [   24/   30] | d_loss: 0.2221 | g_loss: 1.9618
Epoch [   25/   30] | d_loss: 0.1853 | g_loss: 2.5866
Epoch [   25/   30] | d_loss: 0.2578 | g_loss: 4.5291
Epoch [   25/   30] | d_loss: 0.2414 | g_loss: 2.6330
Epoch [   25/   30] | d_loss: 0.5022 | g_loss: 4.8547
Epoch [   25/   30] | d_loss: 1.7526 | g_loss: 9.5183
```

```
Epoch [   25/   30] | d_loss: 0.1054 | g_loss: 2.6458
Epoch [   25/   30] | d_loss: 0.1202 | g_loss: 2.8165
Epoch [   25/   30] | d_loss: 0.6723 | g_loss: 0.7778
Epoch [   25/   30] | d_loss: 0.0443 | g_loss: 2.2287
Epoch [   25/   30] | d_loss: 0.3030 | g_loss: 3.0369
Epoch [   25/   30] | d_loss: 0.5743 | g_loss: 4.7321
Epoch [   25/   30] | d_loss: 0.3884 | g_loss: 2.3586
Epoch [   25/   30] | d_loss: 0.2070 | g_loss: 3.8259
Epoch [   25/   30] | d_loss: 0.0672 | g_loss: 3.4788
Epoch [   25/   30] | d_loss: 0.8542 | g_loss: 1.2083
Epoch [   26/   30] | d_loss: 0.4782 | g_loss: 6.7860
Epoch [   26/   30] | d_loss: 0.2418 | g_loss: 2.7937
Epoch [   26/   30] | d_loss: 0.1376 | g_loss: 5.6021
Epoch [   26/   30] | d_loss: 0.9826 | g_loss: 2.2684
Epoch [   26/   30] | d_loss: 0.1864 | g_loss: 3.4253
Epoch [   26/   30] | d_loss: 0.1321 | g_loss: 4.3113
Epoch [   26/   30] | d_loss: 0.1192 | g_loss: 3.6731
Epoch [   26/   30] | d_loss: 0.0573 | g_loss: 5.0270
Epoch [   26/   30] | d_loss: 1.1512 | g_loss: 4.8239
Epoch [   26/   30] | d_loss: 0.1793 | g_loss: 1.7486
Epoch [   26/   30] | d_loss: 0.4027 | g_loss: 3.9979
Epoch [   26/   30] | d_loss: 0.0956 | g_loss: 3.9019
Epoch [   26/   30] | d_loss: 0.0829 | g_loss: 4.1846
Epoch [   26/   30] | d_loss: 0.2995 | g_loss: 3.7579
Epoch [   26/   30] | d_loss: 0.3780 | g_loss: 1.8280
Epoch [   27/   30] | d_loss: 0.1666 | g_loss: 3.0400
Epoch [   27/   30] | d_loss: 1.0144 | g_loss: 1.0732
Epoch [   27/   30] | d_loss: 0.7207 | g_loss: 0.7878
Epoch [   27/   30] | d_loss: 0.3787 | g_loss: 5.6569
Epoch [   27/   30] | d_loss: 0.0462 | g_loss: 3.6510
Epoch [   27/   30] | d_loss: 0.8693 | g_loss: 0.9678
Epoch [   27/   30] | d_loss: 0.0665 | g_loss: 2.5926
Epoch [   27/   30] | d_loss: 0.1674 | g_loss: 4.4425
Epoch [   27/   30] | d_loss: 1.0650 | g_loss: 5.6006
Epoch [   27/   30] | d_loss: 0.5260 | g_loss: 4.5912
Epoch [   27/   30] | d_loss: 0.0567 | g_loss: 4.7967
Epoch [   27/   30] | d_loss: 0.9607 | g_loss: 5.1893
Epoch [   27/   30] | d_loss: 0.1488 | g_loss: 2.0330
Epoch [   27/   30] | d_loss: 0.0441 | g_loss: 2.0064
Epoch [   27/   30] | d_loss: 0.0933 | g_loss: 2.5669
Epoch [   28/   30] | d_loss: 0.0941 | g_loss: 5.5531
Epoch [   28/   30] | d_loss: 0.2049 | g_loss: 4.4549
Epoch [   28/   30] | d_loss: 0.2026 | g_loss: 1.4864
Epoch [   28/   30] | d_loss: 0.0933 | g_loss: 4.5626
Epoch [   28/   30] | d_loss: 0.9308 | g_loss: 0.9364
Epoch [   28/   30] | d_loss: 0.9051 | g_loss: 0.5088
Epoch [   28/   30] | d_loss: 1.3834 | g_loss: 3.3734
Epoch [   28/   30] | d_loss: 0.2212 | g_loss: 2.3925
```

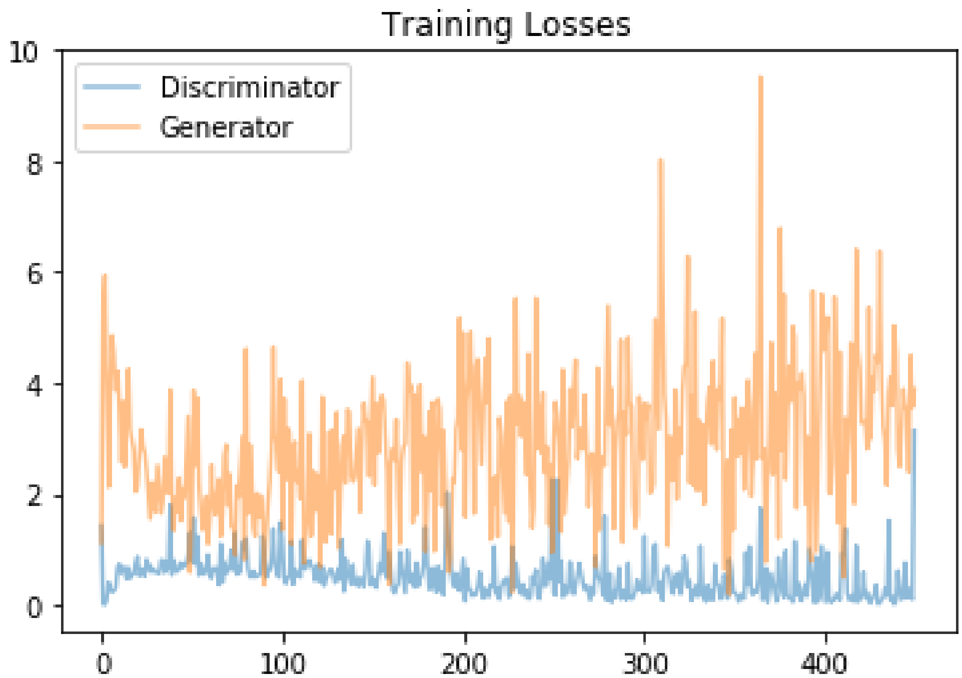
```
Epoch [   28/   30] | d_loss: 0.0991 | g_loss: 3.0543
Epoch [   28/   30] | d_loss: 0.1001 | g_loss: 4.7223
Epoch [   28/   30] | d_loss: 0.1782 | g_loss: 4.0038
Epoch [   28/   30] | d_loss: 0.0650 | g_loss: 1.8287
Epoch [   28/   30] | d_loss: 1.0780 | g_loss: 6.4117
Epoch [   28/   30] | d_loss: 0.0616 | g_loss: 4.5927
Epoch [   28/   30] | d_loss: 0.1104 | g_loss: 4.2685
Epoch [   29/   30] | d_loss: 0.1669 | g_loss: 3.2812
Epoch [   29/   30] | d_loss: 0.6002 | g_loss: 3.2996
Epoch [   29/   30] | d_loss: 0.1472 | g_loss: 3.3195
Epoch [   29/   30] | d_loss: 0.2332 | g_loss: 2.8141
Epoch [   29/   30] | d_loss: 0.0800 | g_loss: 5.3622
Epoch [   29/   30] | d_loss: 0.0365 | g_loss: 2.9950
Epoch [   29/   30] | d_loss: 0.1353 | g_loss: 4.1043
Epoch [   29/   30] | d_loss: 0.1953 | g_loss: 3.8510
Epoch [   29/   30] | d_loss: 0.3688 | g_loss: 4.5013
Epoch [   29/   30] | d_loss: 0.0257 | g_loss: 4.3902
Epoch [   29/   30] | d_loss: 0.0334 | g_loss: 6.3756
Epoch [   29/   30] | d_loss: 0.1021 | g_loss: 4.0608
Epoch [   29/   30] | d_loss: 0.1332 | g_loss: 3.2135
Epoch [   29/   30] | d_loss: 0.2958 | g_loss: 2.9477
Epoch [   29/   30] | d_loss: 0.1717 | g_loss: 2.1578
Epoch [   30/   30] | d_loss: 1.5267 | g_loss: 3.8336
Epoch [   30/   30] | d_loss: 0.0922 | g_loss: 4.1112
Epoch [   30/   30] | d_loss: 0.1525 | g_loss: 3.5951
Epoch [   30/   30] | d_loss: 0.0193 | g_loss: 5.0361
Epoch [   30/   30] | d_loss: 0.0660 | g_loss: 4.5513
Epoch [   30/   30] | d_loss: 0.6192 | g_loss: 2.8055
Epoch [   30/   30] | d_loss: 0.2549 | g_loss: 2.4954
Epoch [   30/   30] | d_loss: 0.1056 | g_loss: 3.2448
Epoch [   30/   30] | d_loss: 0.1866 | g_loss: 3.8927
```

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 0x7f60e2da5780>
```



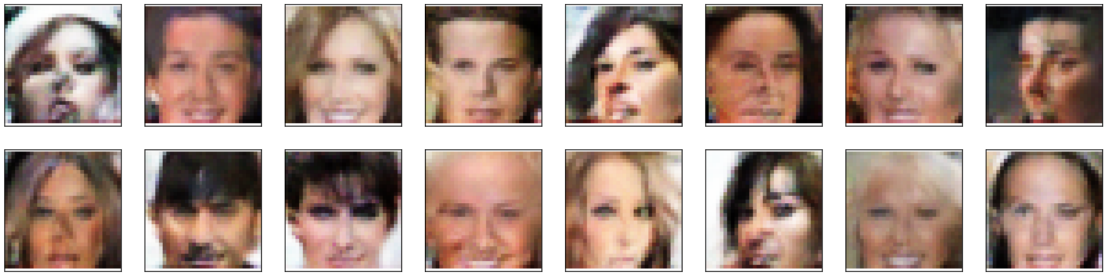
2.9 Generator samples from training

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

```
In [28]: # 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 [30]: # 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)
```



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) 1)We can try label smoothing in this model. 2)We can also try residual blocks in this model to overcome vanishing or exploding gradient problem . 3)We can also input high resolution image to avoid blurness in generated output image but training time will increase. 4)We can increase depth of convolutional layer 5)I train model for 30 epochs that led to learn descriptor to learn more complex features from real images and fake images.so initially it is giving very high error for generated image but later on it starts to decrease. 6)Most of images are white so model is not generalizing better.

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.