dlnd_face_generation

January 8, 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
```

```
Archive:  processed_celeba_small.zip
replace processed_celeba_small/.DS_Store? [y]es, [n]o, [A]ll, [N]one, [r]ename: ^C
```

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

```
In [21]: 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(),
                                   ])

    data = datasets.ImageFolder(data_dir,transform=transform)
    loader = torch.utils.data.DataLoader(data,batch_size=batch_size,shuffle=True)
    return 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 [22]:
```
# Define function hyperparameters
batch_size = 64
img_size = 32

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

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

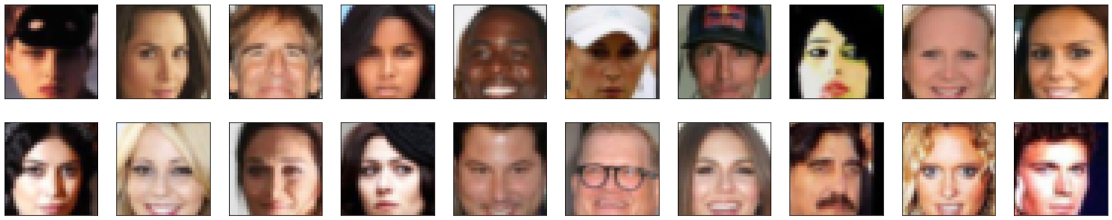
Note: You'll need to convert the Tensor images into a NumPy type and transpose the dimensions to correctly display an image, suggested `imshow` code is below, but it may not be perfect.

In [23]:
```
# 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 [24]: # 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
             x = x*2 - 1
             return x
```

```
In [25]: """
         DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
         """
         # check scaled range
         # should be close to -1 to 1
         img = images[0]
         scaled_img = scale(img)

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

```
Min:  tensor(-1.)
Max:  tensor(0.9922)
```

---

2 Define the Model

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

2.1 Discriminator

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

Exercise: Complete the Discriminator class

- The inputs to the discriminator are 32x32x3 tensor images
- The output should be a single value that will indicate whether a given image is real or fake

In [26]:
```
import torch.nn as nn
import torch.nn.functional as F
```

In [27]:
```
def conv(input_dim,output_dim,k_size,stride=2,padding=1,batch_norm=True):
    layers = []
    conv_layer = nn.Conv2d(input_dim,output_dim,kernel_size=k_size,stride=stride,paddin
    layers.append(conv_layer)
    if batch_norm:
        layers.append(nn.BatchNorm2d(output_dim))

    return nn.Sequential(*layers)
```

In [28]:
```
class Discriminator(nn.Module):

    def __init__(self, conv_dim):
        """
        Initialize the Discriminator Module
        :param conv_dim: The depth of the first convolutional layer
        """
        super(Discriminator, self).__init__()

        # complete init function
        self.conv_dim = conv_dim

        self.conv1 = conv(3,conv_dim,4,batch_norm=False)
        self.conv2 = conv(conv_dim,conv_dim*2,4)
        self.conv3 = conv(conv_dim*2,conv_dim*4,4)
        self.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
        """
        # define feedforward behavior
        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 = out.view(-1,self.conv_dim*4*4*4)
        out = self.fc(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 [29]: def deconv(input_dim,output_dim,k_size,stride=2,padding=1,batch_norm=True):
             layers = []
             deconv_layer = nn.ConvTranspose2d(input_dim,output_dim,kernel_size=k_size,stride=st
             layers.append(deconv_layer)
             if batch_norm:
                 layers.append(nn.BatchNorm2d(output_dim))

             return nn.Sequential(*layers)
```

```
In [30]: 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
```

```python
        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)



    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
        out = self.fc(x)
        out = out.view(-1,self.conv_dim*4,4,4)
        out = F.relu(self.t_conv1(out))
        out = F.relu(self.t_conv2(out))
        out = (self.t_conv3(out))

        return torch.tanh(out)

"""
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 [31]: 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
                 m.weight.data.normal_(0.0, 0.02)

                 # The bias terms, if they exist, set 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 [32]: """
         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 [33]: # 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), 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))
  )
)
```

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 [34]:
```
"""
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 [42]: def real_loss(D_out,smooth=False):
             '''Calculates how close discriminator outputs are to being real.
                param, D_out: discriminator logits
                return: real loss'''
             batch_size = D_out.size(0)
             if smooth:
                 labels = torch.ones(batch_size)*0.9
             else:
                 labels = torch.ones(batch_size)
             if train_on_gpu:
                 labels = labels.cuda()
             criterion = nn.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 [43]: import torch.optim as optim

         # Create optimizers for the discriminator D and generator G
         d_optimizer = optim.Adam(D.parameters(),0.0002,[0.5,0.999])
         g_optimizer = optim.Adam(G.parameters(),0.0002,[0.5,0.999])
```

2.7 Training

Training will involve alternating between training the discriminator and the generator. You'll use your functions `real_loss` and `fake_loss` to help you calculate the discriminator losses.

- You should train the discriminator by alternating on real and fake images
- Then the generator, which tries to trick the discriminator and should have an opposing loss function

Saving Samples You've been given some code to print out some loss statistics and save some generated "fake" samples.

Exercise: Complete the training function Keep in mind that, if you've moved your models to GPU, you'll also have to move any model inputs to GPU.

```
In [44]: 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)
        if train_on_gpu:
            real_images = real_images.cuda()

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

        # 1. Train the discriminator on real and fake images
        d_optimizer.zero_grad()
        d_real = D(real_images)
        r_loss = real_loss(d_real)

        z = np.random.uniform(-1,1,size=(batch_size,z_size))
        z = torch.from_numpy(z).float().cuda()
        fake_images = G(z)
        d_fake = D(fake_images)
        f_loss = fake_loss(d_fake)
        d_loss = r_loss + f_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().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 [45]:
```
# set number of epochs
n_epochs = 20


"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# call training function
from workspace_utils import active_session

with active_session():
    losses = train(D, G, n_epochs=n_epochs)
```

```
Epoch [    1/   20] | d_loss: 1.3480 | g_loss: 1.0309
Epoch [    1/   20] | d_loss: 0.0596 | g_loss: 4.3338
Epoch [    1/   20] | d_loss: 0.2951 | g_loss: 5.9889
Epoch [    1/   20] | d_loss: 0.2663 | g_loss: 3.2091
Epoch [    1/   20] | d_loss: 0.5952 | g_loss: 4.5564
Epoch [    1/   20] | d_loss: 0.6664 | g_loss: 1.0254
Epoch [    1/   20] | d_loss: 0.7505 | g_loss: 1.8546
Epoch [    1/   20] | d_loss: 0.7117 | g_loss: 1.8672
Epoch [    1/   20] | d_loss: 0.9756 | g_loss: 1.0255
Epoch [    1/   20] | d_loss: 1.0054 | g_loss: 1.0732
Epoch [    1/   20] | d_loss: 1.1325 | g_loss: 2.1904
Epoch [    1/   20] | d_loss: 1.1070 | g_loss: 0.9205
Epoch [    1/   20] | d_loss: 0.9329 | g_loss: 1.8831
Epoch [    1/   20] | d_loss: 0.9681 | g_loss: 1.3660
Epoch [    1/   20] | d_loss: 0.9398 | g_loss: 1.3640
Epoch [    1/   20] | d_loss: 1.0577 | g_loss: 0.9102
Epoch [    1/   20] | d_loss: 0.8654 | g_loss: 1.6449
Epoch [    1/   20] | d_loss: 0.9095 | g_loss: 1.1789
Epoch [    1/   20] | d_loss: 0.9595 | g_loss: 1.9290
Epoch [    1/   20] | d_loss: 0.9943 | g_loss: 1.5262
Epoch [    1/   20] | d_loss: 1.3388 | g_loss: 2.0135
Epoch [    1/   20] | d_loss: 0.8842 | g_loss: 2.1508
Epoch [    1/   20] | d_loss: 1.1139 | g_loss: 1.1286
Epoch [    1/   20] | d_loss: 1.1043 | g_loss: 0.8610
Epoch [    1/   20] | d_loss: 1.3661 | g_loss: 1.1650
Epoch [    1/   20] | d_loss: 0.9415 | g_loss: 1.2119
Epoch [    1/   20] | d_loss: 1.4146 | g_loss: 0.8346
Epoch [    1/   20] | d_loss: 1.7142 | g_loss: 1.6055
Epoch [    1/   20] | d_loss: 1.5643 | g_loss: 0.9038
Epoch [    2/   20] | d_loss: 1.1144 | g_loss: 1.4772
Epoch [    2/   20] | d_loss: 1.2103 | g_loss: 1.2153
Epoch [    2/   20] | d_loss: 1.1775 | g_loss: 0.8522
Epoch [    2/   20] | d_loss: 1.0735 | g_loss: 1.2664
Epoch [    2/   20] | d_loss: 1.1063 | g_loss: 1.2801
Epoch [    2/   20] | d_loss: 1.0524 | g_loss: 1.3756
Epoch [    2/   20] | d_loss: 1.0184 | g_loss: 1.1537
Epoch [    2/   20] | d_loss: 1.1663 | g_loss: 1.1026
Epoch [    2/   20] | d_loss: 1.0629 | g_loss: 1.2177
Epoch [    2/   20] | d_loss: 1.1609 | g_loss: 1.6679
Epoch [    2/   20] | d_loss: 1.1031 | g_loss: 1.2018
Epoch [    2/   20] | d_loss: 1.1981 | g_loss: 1.0121
Epoch [    2/   20] | d_loss: 0.9227 | g_loss: 0.8958
Epoch [    2/   20] | d_loss: 1.3372 | g_loss: 1.5782
Epoch [    2/   20] | d_loss: 1.1242 | g_loss: 0.9433
Epoch [    2/   20] | d_loss: 1.3368 | g_loss: 0.8309
Epoch [    2/   20] | d_loss: 1.1729 | g_loss: 1.1142
Epoch [    2/   20] | d_loss: 0.9121 | g_loss: 1.7167
Epoch [    2/   20] | d_loss: 0.8934 | g_loss: 1.2172
```

```
Epoch [    2/   20] | d_loss: 1.0530 | g_loss: 0.8876
Epoch [    2/   20] | d_loss: 1.0269 | g_loss: 1.9825
Epoch [    2/   20] | d_loss: 0.9038 | g_loss: 1.5737
Epoch [    2/   20] | d_loss: 0.8423 | g_loss: 1.7604
Epoch [    2/   20] | d_loss: 1.0883 | g_loss: 0.9781
Epoch [    2/   20] | d_loss: 1.1540 | g_loss: 0.4701
Epoch [    2/   20] | d_loss: 1.0207 | g_loss: 1.4692
Epoch [    2/   20] | d_loss: 0.9603 | g_loss: 1.0775
Epoch [    2/   20] | d_loss: 0.9622 | g_loss: 1.0784
Epoch [    2/   20] | d_loss: 0.9656 | g_loss: 1.8820
Epoch [    3/   20] | d_loss: 1.0876 | g_loss: 1.1980
Epoch [    3/   20] | d_loss: 0.9690 | g_loss: 1.2989
Epoch [    3/   20] | d_loss: 1.3129 | g_loss: 1.0464
Epoch [    3/   20] | d_loss: 1.1195 | g_loss: 0.6705
Epoch [    3/   20] | d_loss: 0.8086 | g_loss: 2.1296
Epoch [    3/   20] | d_loss: 0.7795 | g_loss: 1.1356
Epoch [    3/   20] | d_loss: 0.9686 | g_loss: 1.8830
Epoch [    3/   20] | d_loss: 0.9047 | g_loss: 1.5416
Epoch [    3/   20] | d_loss: 0.9869 | g_loss: 1.8623
Epoch [    3/   20] | d_loss: 0.8015 | g_loss: 1.5389
Epoch [    3/   20] | d_loss: 0.9245 | g_loss: 1.2045
Epoch [    3/   20] | d_loss: 1.2536 | g_loss: 0.5724
Epoch [    3/   20] | d_loss: 0.6108 | g_loss: 2.0351
Epoch [    3/   20] | d_loss: 0.9772 | g_loss: 1.4336
Epoch [    3/   20] | d_loss: 0.9045 | g_loss: 1.2597
Epoch [    3/   20] | d_loss: 0.8418 | g_loss: 1.8670
Epoch [    3/   20] | d_loss: 0.8330 | g_loss: 1.8754
Epoch [    3/   20] | d_loss: 0.9553 | g_loss: 1.2259
Epoch [    3/   20] | d_loss: 0.9299 | g_loss: 1.2782
Epoch [    3/   20] | d_loss: 1.0352 | g_loss: 1.1541
Epoch [    3/   20] | d_loss: 0.9125 | g_loss: 1.5867
Epoch [    3/   20] | d_loss: 0.9446 | g_loss: 1.7109
Epoch [    3/   20] | d_loss: 0.8096 | g_loss: 1.1635
Epoch [    3/   20] | d_loss: 0.7837 | g_loss: 1.4888
Epoch [    3/   20] | d_loss: 1.1648 | g_loss: 2.1578
Epoch [    3/   20] | d_loss: 0.9111 | g_loss: 1.5655
Epoch [    3/   20] | d_loss: 0.9971 | g_loss: 1.8613
Epoch [    3/   20] | d_loss: 0.9144 | g_loss: 1.1386
Epoch [    3/   20] | d_loss: 0.9744 | g_loss: 1.4872
Epoch [    4/   20] | d_loss: 0.8155 | g_loss: 1.4649
Epoch [    4/   20] | d_loss: 1.1306 | g_loss: 1.3194
Epoch [    4/   20] | d_loss: 0.8065 | g_loss: 1.6668
Epoch [    4/   20] | d_loss: 1.1667 | g_loss: 1.8306
Epoch [    4/   20] | d_loss: 0.8537 | g_loss: 1.6753
Epoch [    4/   20] | d_loss: 0.8529 | g_loss: 1.2762
Epoch [    4/   20] | d_loss: 0.9347 | g_loss: 2.3318
Epoch [    4/   20] | d_loss: 0.6950 | g_loss: 1.4198
Epoch [    4/   20] | d_loss: 0.7433 | g_loss: 1.5769
```

```
Epoch [    4/   20] | d_loss: 0.9455 | g_loss: 1.1095
Epoch [    4/   20] | d_loss: 1.1073 | g_loss: 0.8586
Epoch [    4/   20] | d_loss: 0.7423 | g_loss: 1.8527
Epoch [    4/   20] | d_loss: 0.9244 | g_loss: 1.6760
Epoch [    4/   20] | d_loss: 0.8306 | g_loss: 1.5297
Epoch [    4/   20] | d_loss: 0.8734 | g_loss: 1.5820
Epoch [    4/   20] | d_loss: 1.1240 | g_loss: 2.2160
Epoch [    4/   20] | d_loss: 1.4715 | g_loss: 2.0550
Epoch [    4/   20] | d_loss: 0.4792 | g_loss: 2.3048
Epoch [    4/   20] | d_loss: 1.2321 | g_loss: 2.3196
Epoch [    4/   20] | d_loss: 0.6105 | g_loss: 1.4032
Epoch [    4/   20] | d_loss: 0.9913 | g_loss: 0.9680
Epoch [    4/   20] | d_loss: 0.8574 | g_loss: 1.2164
Epoch [    4/   20] | d_loss: 0.9523 | g_loss: 1.9299
Epoch [    4/   20] | d_loss: 1.0297 | g_loss: 1.1602
Epoch [    4/   20] | d_loss: 0.9409 | g_loss: 1.6869
Epoch [    4/   20] | d_loss: 0.8161 | g_loss: 1.6687
Epoch [    4/   20] | d_loss: 0.8224 | g_loss: 0.8802
Epoch [    4/   20] | d_loss: 0.9964 | g_loss: 1.4736
Epoch [    4/   20] | d_loss: 1.0023 | g_loss: 1.1981
Epoch [    5/   20] | d_loss: 0.8697 | g_loss: 1.7282
Epoch [    5/   20] | d_loss: 1.2984 | g_loss: 1.9294
Epoch [    5/   20] | d_loss: 0.7691 | g_loss: 1.7419
Epoch [    5/   20] | d_loss: 0.8408 | g_loss: 1.5533
Epoch [    5/   20] | d_loss: 1.0396 | g_loss: 1.8517
Epoch [    5/   20] | d_loss: 0.8920 | g_loss: 1.0679
Epoch [    5/   20] | d_loss: 1.3779 | g_loss: 0.3827
Epoch [    5/   20] | d_loss: 1.0284 | g_loss: 2.8893
Epoch [    5/   20] | d_loss: 0.6531 | g_loss: 1.3391
Epoch [    5/   20] | d_loss: 0.8354 | g_loss: 1.3589
Epoch [    5/   20] | d_loss: 1.3174 | g_loss: 0.9882
Epoch [    5/   20] | d_loss: 0.7037 | g_loss: 2.0504
Epoch [    5/   20] | d_loss: 0.7975 | g_loss: 2.0182
Epoch [    5/   20] | d_loss: 0.7075 | g_loss: 1.7912
Epoch [    5/   20] | d_loss: 0.8673 | g_loss: 2.2867
Epoch [    5/   20] | d_loss: 0.9124 | g_loss: 2.0342
Epoch [    5/   20] | d_loss: 0.9788 | g_loss: 1.2505
Epoch [    5/   20] | d_loss: 1.1504 | g_loss: 1.2947
Epoch [    5/   20] | d_loss: 0.9509 | g_loss: 0.9902
Epoch [    5/   20] | d_loss: 0.7627 | g_loss: 1.2030
Epoch [    5/   20] | d_loss: 0.9246 | g_loss: 1.4171
Epoch [    5/   20] | d_loss: 0.7517 | g_loss: 1.1022
Epoch [    5/   20] | d_loss: 0.9384 | g_loss: 1.5925
Epoch [    5/   20] | d_loss: 0.8550 | g_loss: 1.0341
Epoch [    5/   20] | d_loss: 1.6071 | g_loss: 3.2804
Epoch [    5/   20] | d_loss: 0.8363 | g_loss: 1.1404
Epoch [    5/   20] | d_loss: 0.9611 | g_loss: 2.0531
Epoch [    5/   20] | d_loss: 0.8224 | g_loss: 1.2046
```

```
Epoch [    5/   20] | d_loss: 0.8375 | g_loss: 1.0888
Epoch [    6/   20] | d_loss: 0.9323 | g_loss: 3.0724
Epoch [    6/   20] | d_loss: 0.8902 | g_loss: 0.9290
Epoch [    6/   20] | d_loss: 1.4922 | g_loss: 0.6528
Epoch [    6/   20] | d_loss: 0.9077 | g_loss: 2.1094
Epoch [    6/   20] | d_loss: 1.1886 | g_loss: 1.9663
Epoch [    6/   20] | d_loss: 0.8555 | g_loss: 1.3682
Epoch [    6/   20] | d_loss: 0.5775 | g_loss: 2.0095
Epoch [    6/   20] | d_loss: 0.7699 | g_loss: 1.5647
Epoch [    6/   20] | d_loss: 1.0923 | g_loss: 2.1974
Epoch [    6/   20] | d_loss: 0.9143 | g_loss: 1.4893
Epoch [    6/   20] | d_loss: 0.8769 | g_loss: 1.7451
Epoch [    6/   20] | d_loss: 0.7786 | g_loss: 1.2413
Epoch [    6/   20] | d_loss: 0.6655 | g_loss: 1.7827
Epoch [    6/   20] | d_loss: 0.7018 | g_loss: 2.0941
Epoch [    6/   20] | d_loss: 1.0003 | g_loss: 1.5205
Epoch [    6/   20] | d_loss: 0.7821 | g_loss: 1.3475
Epoch [    6/   20] | d_loss: 0.7658 | g_loss: 1.1743
Epoch [    6/   20] | d_loss: 0.9455 | g_loss: 1.1286
Epoch [    6/   20] | d_loss: 0.8852 | g_loss: 1.9057
Epoch [    6/   20] | d_loss: 0.8232 | g_loss: 1.2556
Epoch [    6/   20] | d_loss: 0.6605 | g_loss: 1.6245
Epoch [    6/   20] | d_loss: 0.8829 | g_loss: 1.1516
Epoch [    6/   20] | d_loss: 0.7617 | g_loss: 1.5807
Epoch [    6/   20] | d_loss: 0.8579 | g_loss: 0.8480
Epoch [    6/   20] | d_loss: 0.6296 | g_loss: 1.5259
Epoch [    6/   20] | d_loss: 0.9245 | g_loss: 1.3684
Epoch [    6/   20] | d_loss: 0.8919 | g_loss: 1.3935
Epoch [    6/   20] | d_loss: 0.8945 | g_loss: 1.1368
Epoch [    6/   20] | d_loss: 0.7901 | g_loss: 2.2499
Epoch [    7/   20] | d_loss: 0.9460 | g_loss: 0.6792
Epoch [    7/   20] | d_loss: 0.6538 | g_loss: 1.8892
Epoch [    7/   20] | d_loss: 0.8893 | g_loss: 1.6088
Epoch [    7/   20] | d_loss: 0.7657 | g_loss: 0.8082
Epoch [    7/   20] | d_loss: 0.7156 | g_loss: 1.3590
Epoch [    7/   20] | d_loss: 0.7231 | g_loss: 2.2983
Epoch [    7/   20] | d_loss: 0.7446 | g_loss: 1.5770
Epoch [    7/   20] | d_loss: 1.1315 | g_loss: 3.1057
Epoch [    7/   20] | d_loss: 0.5905 | g_loss: 1.9401
Epoch [    7/   20] | d_loss: 0.5512 | g_loss: 2.0361
Epoch [    7/   20] | d_loss: 0.8878 | g_loss: 0.7851
Epoch [    7/   20] | d_loss: 0.8497 | g_loss: 2.2330
Epoch [    7/   20] | d_loss: 0.6834 | g_loss: 2.0735
Epoch [    7/   20] | d_loss: 0.8293 | g_loss: 1.3475
Epoch [    7/   20] | d_loss: 0.6284 | g_loss: 1.0280
Epoch [    7/   20] | d_loss: 0.7632 | g_loss: 1.6294
Epoch [    7/   20] | d_loss: 0.8030 | g_loss: 1.4751
Epoch [    7/   20] | d_loss: 0.8694 | g_loss: 1.1433
```

```
Epoch [    7/   20] | d_loss: 0.7831 | g_loss: 1.4572
Epoch [    7/   20] | d_loss: 0.9155 | g_loss: 1.5320
Epoch [    7/   20] | d_loss: 0.9726 | g_loss: 0.8235
Epoch [    7/   20] | d_loss: 0.9135 | g_loss: 1.8179
Epoch [    7/   20] | d_loss: 1.1021 | g_loss: 2.2879
Epoch [    7/   20] | d_loss: 0.8870 | g_loss: 1.0174
Epoch [    7/   20] | d_loss: 0.6394 | g_loss: 1.3835
Epoch [    7/   20] | d_loss: 0.9474 | g_loss: 1.9871
Epoch [    7/   20] | d_loss: 0.6868 | g_loss: 1.1515
Epoch [    7/   20] | d_loss: 0.5377 | g_loss: 1.8686
Epoch [    7/   20] | d_loss: 0.6723 | g_loss: 1.8396
Epoch [    8/   20] | d_loss: 0.7585 | g_loss: 1.2342
Epoch [    8/   20] | d_loss: 0.5407 | g_loss: 1.6516
Epoch [    8/   20] | d_loss: 0.6134 | g_loss: 1.7502
Epoch [    8/   20] | d_loss: 1.2484 | g_loss: 2.8929
Epoch [    8/   20] | d_loss: 0.9239 | g_loss: 2.0723
Epoch [    8/   20] | d_loss: 0.9701 | g_loss: 3.3788
Epoch [    8/   20] | d_loss: 0.5365 | g_loss: 2.4593
Epoch [    8/   20] | d_loss: 0.8480 | g_loss: 1.8181
Epoch [    8/   20] | d_loss: 0.5895 | g_loss: 2.1064
Epoch [    8/   20] | d_loss: 0.5973 | g_loss: 2.0311
Epoch [    8/   20] | d_loss: 0.7222 | g_loss: 2.4175
Epoch [    8/   20] | d_loss: 0.8225 | g_loss: 0.9162
Epoch [    8/   20] | d_loss: 1.2016 | g_loss: 2.1682
Epoch [    8/   20] | d_loss: 0.6031 | g_loss: 2.1053
Epoch [    8/   20] | d_loss: 0.5304 | g_loss: 1.4526
Epoch [    8/   20] | d_loss: 0.4926 | g_loss: 2.1304
Epoch [    8/   20] | d_loss: 0.7350 | g_loss: 1.3003
Epoch [    8/   20] | d_loss: 0.7014 | g_loss: 1.2474
Epoch [    8/   20] | d_loss: 0.4936 | g_loss: 2.1601
Epoch [    8/   20] | d_loss: 0.8131 | g_loss: 2.6285
Epoch [    8/   20] | d_loss: 0.7929 | g_loss: 1.3584
Epoch [    8/   20] | d_loss: 0.5930 | g_loss: 1.5065
Epoch [    8/   20] | d_loss: 0.6721 | g_loss: 1.7144
Epoch [    8/   20] | d_loss: 0.5710 | g_loss: 2.6464
Epoch [    8/   20] | d_loss: 0.6369 | g_loss: 2.3301
Epoch [    8/   20] | d_loss: 0.8092 | g_loss: 1.3424
Epoch [    8/   20] | d_loss: 0.6319 | g_loss: 1.2415
Epoch [    8/   20] | d_loss: 0.6222 | g_loss: 1.5127
Epoch [    8/   20] | d_loss: 1.0987 | g_loss: 0.8754
Epoch [    9/   20] | d_loss: 0.6490 | g_loss: 1.5663
Epoch [    9/   20] | d_loss: 0.8978 | g_loss: 1.7756
Epoch [    9/   20] | d_loss: 0.7483 | g_loss: 2.6735
Epoch [    9/   20] | d_loss: 0.5208 | g_loss: 1.3173
Epoch [    9/   20] | d_loss: 0.7025 | g_loss: 3.6321
Epoch [    9/   20] | d_loss: 0.4032 | g_loss: 1.6227
Epoch [    9/   20] | d_loss: 0.7522 | g_loss: 2.9397
Epoch [    9/   20] | d_loss: 0.9689 | g_loss: 0.5807
```

```
Epoch [    9/   20] | d_loss: 0.7572 | g_loss: 1.7260
Epoch [    9/   20] | d_loss: 0.9072 | g_loss: 3.2143
Epoch [    9/   20] | d_loss: 0.5038 | g_loss: 1.5165
Epoch [    9/   20] | d_loss: 0.5614 | g_loss: 2.0965
Epoch [    9/   20] | d_loss: 0.5795 | g_loss: 2.1387
Epoch [    9/   20] | d_loss: 0.7278 | g_loss: 1.5213
Epoch [    9/   20] | d_loss: 0.6982 | g_loss: 1.5812
Epoch [    9/   20] | d_loss: 0.6671 | g_loss: 1.6160
Epoch [    9/   20] | d_loss: 0.5696 | g_loss: 1.7116
Epoch [    9/   20] | d_loss: 0.5481 | g_loss: 1.6191
Epoch [    9/   20] | d_loss: 1.7708 | g_loss: 4.2585
Epoch [    9/   20] | d_loss: 0.4754 | g_loss: 2.2254
Epoch [    9/   20] | d_loss: 0.3969 | g_loss: 1.6656
Epoch [    9/   20] | d_loss: 0.7112 | g_loss: 1.5928
Epoch [    9/   20] | d_loss: 0.5263 | g_loss: 2.1847
Epoch [    9/   20] | d_loss: 0.4883 | g_loss: 2.9838
Epoch [    9/   20] | d_loss: 0.5445 | g_loss: 2.5823
Epoch [    9/   20] | d_loss: 0.6611 | g_loss: 2.1176
Epoch [    9/   20] | d_loss: 0.8740 | g_loss: 1.4289
Epoch [    9/   20] | d_loss: 0.4920 | g_loss: 2.2778
Epoch [    9/   20] | d_loss: 0.6035 | g_loss: 1.7914
Epoch [   10/   20] | d_loss: 1.5531 | g_loss: 0.4381
Epoch [   10/   20] | d_loss: 0.4897 | g_loss: 1.6111
Epoch [   10/   20] | d_loss: 0.4713 | g_loss: 1.0022
Epoch [   10/   20] | d_loss: 0.9938 | g_loss: 1.0417
Epoch [   10/   20] | d_loss: 0.4779 | g_loss: 2.8764
Epoch [   10/   20] | d_loss: 0.4581 | g_loss: 2.7829
Epoch [   10/   20] | d_loss: 0.5797 | g_loss: 2.4364
Epoch [   10/   20] | d_loss: 1.3470 | g_loss: 3.1910
Epoch [   10/   20] | d_loss: 0.5794 | g_loss: 2.4901
Epoch [   10/   20] | d_loss: 0.7815 | g_loss: 1.0885
Epoch [   10/   20] | d_loss: 0.9534 | g_loss: 2.7046
Epoch [   10/   20] | d_loss: 0.6333 | g_loss: 2.7053
Epoch [   10/   20] | d_loss: 0.5150 | g_loss: 1.7195
Epoch [   10/   20] | d_loss: 0.3831 | g_loss: 1.7835
Epoch [   10/   20] | d_loss: 0.2970 | g_loss: 2.4504
Epoch [   10/   20] | d_loss: 0.6714 | g_loss: 0.9723
Epoch [   10/   20] | d_loss: 0.4220 | g_loss: 1.9268
Epoch [   10/   20] | d_loss: 0.4392 | g_loss: 2.8963
Epoch [   10/   20] | d_loss: 0.7707 | g_loss: 2.2726
Epoch [   10/   20] | d_loss: 0.4168 | g_loss: 2.1738
Epoch [   10/   20] | d_loss: 0.3917 | g_loss: 1.3395
Epoch [   10/   20] | d_loss: 0.5242 | g_loss: 2.1110
Epoch [   10/   20] | d_loss: 0.4841 | g_loss: 1.5901
Epoch [   10/   20] | d_loss: 0.4416 | g_loss: 3.4458
Epoch [   10/   20] | d_loss: 0.6160 | g_loss: 2.2429
Epoch [   10/   20] | d_loss: 0.3094 | g_loss: 2.9267
Epoch [   10/   20] | d_loss: 0.6706 | g_loss: 3.1354
```

```
Epoch [   10/   20] | d_loss: 0.3832 | g_loss: 1.9370
Epoch [   10/   20] | d_loss: 0.4023 | g_loss: 2.5792
Epoch [   11/   20] | d_loss: 0.5019 | g_loss: 1.9395
Epoch [   11/   20] | d_loss: 0.4092 | g_loss: 1.4747
Epoch [   11/   20] | d_loss: 0.8089 | g_loss: 1.5328
Epoch [   11/   20] | d_loss: 0.3883 | g_loss: 2.5288
Epoch [   11/   20] | d_loss: 0.5047 | g_loss: 2.3399
Epoch [   11/   20] | d_loss: 0.4124 | g_loss: 2.2264
Epoch [   11/   20] | d_loss: 0.5256 | g_loss: 1.2202
Epoch [   11/   20] | d_loss: 0.4802 | g_loss: 2.5401
Epoch [   11/   20] | d_loss: 0.4431 | g_loss: 2.5790
Epoch [   11/   20] | d_loss: 0.2686 | g_loss: 2.5913
Epoch [   11/   20] | d_loss: 0.4431 | g_loss: 2.9050
Epoch [   11/   20] | d_loss: 0.3728 | g_loss: 2.2642
Epoch [   11/   20] | d_loss: 0.4024 | g_loss: 1.8549
Epoch [   11/   20] | d_loss: 0.4417 | g_loss: 2.8300
Epoch [   11/   20] | d_loss: 1.2892 | g_loss: 0.0239
Epoch [   11/   20] | d_loss: 0.5768 | g_loss: 2.0080
Epoch [   11/   20] | d_loss: 0.3982 | g_loss: 3.1556
Epoch [   11/   20] | d_loss: 0.3990 | g_loss: 3.0966
Epoch [   11/   20] | d_loss: 0.5204 | g_loss: 1.9083
Epoch [   11/   20] | d_loss: 0.2240 | g_loss: 2.9378
Epoch [   11/   20] | d_loss: 0.9087 | g_loss: 2.5823
Epoch [   11/   20] | d_loss: 0.4511 | g_loss: 3.5350
Epoch [   11/   20] | d_loss: 0.4056 | g_loss: 2.1673
Epoch [   11/   20] | d_loss: 0.5026 | g_loss: 2.0502
Epoch [   11/   20] | d_loss: 0.5167 | g_loss: 2.0204
Epoch [   11/   20] | d_loss: 0.4105 | g_loss: 2.2231
Epoch [   11/   20] | d_loss: 0.4270 | g_loss: 1.4367
Epoch [   11/   20] | d_loss: 0.4805 | g_loss: 2.1491
Epoch [   11/   20] | d_loss: 0.3844 | g_loss: 2.7341
Epoch [   12/   20] | d_loss: 0.4448 | g_loss: 2.9351
Epoch [   12/   20] | d_loss: 0.5632 | g_loss: 0.8443
Epoch [   12/   20] | d_loss: 0.4846 | g_loss: 1.2955
Epoch [   12/   20] | d_loss: 0.7443 | g_loss: 4.0613
Epoch [   12/   20] | d_loss: 0.2633 | g_loss: 3.5914
Epoch [   12/   20] | d_loss: 0.8285 | g_loss: 4.4752
Epoch [   12/   20] | d_loss: 0.5898 | g_loss: 2.7254
Epoch [   12/   20] | d_loss: 0.4429 | g_loss: 1.5896
Epoch [   12/   20] | d_loss: 0.4198 | g_loss: 1.8928
Epoch [   12/   20] | d_loss: 0.3385 | g_loss: 2.4608
Epoch [   12/   20] | d_loss: 0.1920 | g_loss: 3.2042
Epoch [   12/   20] | d_loss: 0.3049 | g_loss: 1.9113
Epoch [   12/   20] | d_loss: 0.6011 | g_loss: 1.8903
Epoch [   12/   20] | d_loss: 0.5231 | g_loss: 1.3358
Epoch [   12/   20] | d_loss: 0.2896 | g_loss: 3.2379
Epoch [   12/   20] | d_loss: 0.3297 | g_loss: 2.9145
Epoch [   12/   20] | d_loss: 0.3719 | g_loss: 1.8471
```

```
Epoch [   12/   20] | d_loss: 0.3506 | g_loss: 1.3461
Epoch [   12/   20] | d_loss: 0.7013 | g_loss: 2.0018
Epoch [   12/   20] | d_loss: 0.3946 | g_loss: 2.0897
Epoch [   12/   20] | d_loss: 0.6259 | g_loss: 3.8418
Epoch [   12/   20] | d_loss: 0.4916 | g_loss: 1.6904
Epoch [   12/   20] | d_loss: 0.5364 | g_loss: 3.4752
Epoch [   12/   20] | d_loss: 0.3219 | g_loss: 2.9395
Epoch [   12/   20] | d_loss: 0.5006 | g_loss: 2.8999
Epoch [   12/   20] | d_loss: 0.5474 | g_loss: 4.5475
Epoch [   12/   20] | d_loss: 0.3979 | g_loss: 3.3147
Epoch [   12/   20] | d_loss: 0.3451 | g_loss: 2.3128
Epoch [   12/   20] | d_loss: 0.4153 | g_loss: 3.2742
Epoch [   13/   20] | d_loss: 0.4838 | g_loss: 2.0589
Epoch [   13/   20] | d_loss: 0.3266 | g_loss: 3.1137
Epoch [   13/   20] | d_loss: 0.1986 | g_loss: 2.6069
Epoch [   13/   20] | d_loss: 0.3191 | g_loss: 2.7462
Epoch [   13/   20] | d_loss: 0.8921 | g_loss: 0.9293
Epoch [   13/   20] | d_loss: 0.4551 | g_loss: 2.0658
Epoch [   13/   20] | d_loss: 0.6648 | g_loss: 3.1816
Epoch [   13/   20] | d_loss: 0.4831 | g_loss: 1.5925
Epoch [   13/   20] | d_loss: 0.4823 | g_loss: 3.9909
Epoch [   13/   20] | d_loss: 0.4238 | g_loss: 3.7747
Epoch [   13/   20] | d_loss: 0.2955 | g_loss: 3.3660
Epoch [   13/   20] | d_loss: 0.4076 | g_loss: 3.5597
Epoch [   13/   20] | d_loss: 0.1224 | g_loss: 2.8658
Epoch [   13/   20] | d_loss: 0.2471 | g_loss: 2.5849
Epoch [   13/   20] | d_loss: 0.3094 | g_loss: 2.9054
Epoch [   13/   20] | d_loss: 0.2317 | g_loss: 3.0938
Epoch [   13/   20] | d_loss: 0.2612 | g_loss: 2.9553
Epoch [   13/   20] | d_loss: 0.3421 | g_loss: 2.4624
Epoch [   13/   20] | d_loss: 0.5360 | g_loss: 1.4526
Epoch [   13/   20] | d_loss: 0.3667 | g_loss: 2.5831
Epoch [   13/   20] | d_loss: 0.5191 | g_loss: 3.6991
Epoch [   13/   20] | d_loss: 0.3748 | g_loss: 2.8161
Epoch [   13/   20] | d_loss: 0.4140 | g_loss: 2.2833
Epoch [   13/   20] | d_loss: 0.2174 | g_loss: 2.2538
Epoch [   13/   20] | d_loss: 0.2328 | g_loss: 2.5787
Epoch [   13/   20] | d_loss: 0.3488 | g_loss: 2.4705
Epoch [   13/   20] | d_loss: 0.4585 | g_loss: 3.3711
Epoch [   13/   20] | d_loss: 0.3738 | g_loss: 2.9176
Epoch [   13/   20] | d_loss: 0.4299 | g_loss: 2.8313
Epoch [   14/   20] | d_loss: 0.3795 | g_loss: 3.4297
Epoch [   14/   20] | d_loss: 0.2478 | g_loss: 2.8307
Epoch [   14/   20] | d_loss: 0.2623 | g_loss: 2.9163
Epoch [   14/   20] | d_loss: 0.4701 | g_loss: 2.0113
Epoch [   14/   20] | d_loss: 0.3652 | g_loss: 3.0321
Epoch [   14/   20] | d_loss: 0.2134 | g_loss: 2.8760
Epoch [   14/   20] | d_loss: 1.3833 | g_loss: 3.8274
```

```
Epoch [   14/   20] | d_loss: 0.3172 | g_loss: 1.8524
Epoch [   14/   20] | d_loss: 0.2862 | g_loss: 2.8116
Epoch [   14/   20] | d_loss: 0.3342 | g_loss: 3.5019
Epoch [   14/   20] | d_loss: 0.3571 | g_loss: 1.8220
Epoch [   14/   20] | d_loss: 1.2969 | g_loss: 4.0198
Epoch [   14/   20] | d_loss: 0.2511 | g_loss: 2.1747
Epoch [   14/   20] | d_loss: 0.1485 | g_loss: 2.3847
Epoch [   14/   20] | d_loss: 0.1837 | g_loss: 2.4316
Epoch [   14/   20] | d_loss: 0.2979 | g_loss: 2.3241
Epoch [   14/   20] | d_loss: 0.3749 | g_loss: 1.2660
Epoch [   14/   20] | d_loss: 0.1420 | g_loss: 3.8811
Epoch [   14/   20] | d_loss: 0.1835 | g_loss: 2.7794
Epoch [   14/   20] | d_loss: 0.3181 | g_loss: 2.0106
Epoch [   14/   20] | d_loss: 0.2184 | g_loss: 2.7804
Epoch [   14/   20] | d_loss: 0.3242 | g_loss: 2.3973
Epoch [   14/   20] | d_loss: 0.2162 | g_loss: 4.2026
Epoch [   14/   20] | d_loss: 0.2425 | g_loss: 4.4625
Epoch [   14/   20] | d_loss: 0.8000 | g_loss: 1.5241
Epoch [   14/   20] | d_loss: 0.4124 | g_loss: 3.1854
Epoch [   14/   20] | d_loss: 0.2742 | g_loss: 3.5855
Epoch [   14/   20] | d_loss: 0.1489 | g_loss: 4.0348
Epoch [   14/   20] | d_loss: 0.4496 | g_loss: 2.1677
Epoch [   15/   20] | d_loss: 0.4902 | g_loss: 2.0417
Epoch [   15/   20] | d_loss: 0.5324 | g_loss: 2.3511
Epoch [   15/   20] | d_loss: 0.2343 | g_loss: 3.1214
Epoch [   15/   20] | d_loss: 0.2088 | g_loss: 3.2666
Epoch [   15/   20] | d_loss: 0.1560 | g_loss: 2.8702
Epoch [   15/   20] | d_loss: 0.4049 | g_loss: 2.9691
Epoch [   15/   20] | d_loss: 0.2197 | g_loss: 2.4885
Epoch [   15/   20] | d_loss: 0.6347 | g_loss: 1.8909
Epoch [   15/   20] | d_loss: 0.2577 | g_loss: 2.7738
Epoch [   15/   20] | d_loss: 0.2634 | g_loss: 2.1864
Epoch [   15/   20] | d_loss: 0.3372 | g_loss: 2.2697
Epoch [   15/   20] | d_loss: 0.1407 | g_loss: 2.7733
Epoch [   15/   20] | d_loss: 0.3075 | g_loss: 2.4783
Epoch [   15/   20] | d_loss: 0.4536 | g_loss: 1.5547
Epoch [   15/   20] | d_loss: 0.2782 | g_loss: 4.4161
Epoch [   15/   20] | d_loss: 0.2440 | g_loss: 3.3497
Epoch [   15/   20] | d_loss: 0.2897 | g_loss: 2.7490
Epoch [   15/   20] | d_loss: 0.1517 | g_loss: 2.9387
Epoch [   15/   20] | d_loss: 0.3626 | g_loss: 1.9157
Epoch [   15/   20] | d_loss: 0.2748 | g_loss: 3.5278
Epoch [   15/   20] | d_loss: 0.5707 | g_loss: 3.8089
Epoch [   15/   20] | d_loss: 0.6167 | g_loss: 1.2760
Epoch [   15/   20] | d_loss: 0.2403 | g_loss: 2.2230
Epoch [   15/   20] | d_loss: 0.1918 | g_loss: 2.9063
Epoch [   15/   20] | d_loss: 0.3081 | g_loss: 2.0469
Epoch [   15/   20] | d_loss: 0.3766 | g_loss: 3.9646
```

```
Epoch [   15/   20] | d_loss: 0.1856 | g_loss: 3.1556
Epoch [   15/   20] | d_loss: 0.4668 | g_loss: 3.5233
Epoch [   15/   20] | d_loss: 0.3086 | g_loss: 3.7003
Epoch [   16/   20] | d_loss: 1.0249 | g_loss: 1.7013
Epoch [   16/   20] | d_loss: 0.3142 | g_loss: 3.5634
Epoch [   16/   20] | d_loss: 0.3960 | g_loss: 2.5616
Epoch [   16/   20] | d_loss: 0.2410 | g_loss: 3.2529
Epoch [   16/   20] | d_loss: 0.1414 | g_loss: 3.2429
Epoch [   16/   20] | d_loss: 2.4153 | g_loss: 3.6666
Epoch [   16/   20] | d_loss: 0.2714 | g_loss: 3.1697
Epoch [   16/   20] | d_loss: 0.2520 | g_loss: 3.1266
Epoch [   16/   20] | d_loss: 0.3917 | g_loss: 2.3212
Epoch [   16/   20] | d_loss: 0.2502 | g_loss: 3.3922
Epoch [   16/   20] | d_loss: 0.3765 | g_loss: 3.6932
Epoch [   16/   20] | d_loss: 0.1800 | g_loss: 2.5037
Epoch [   16/   20] | d_loss: 2.4355 | g_loss: 8.9504
Epoch [   16/   20] | d_loss: 0.8850 | g_loss: 5.0540
Epoch [   16/   20] | d_loss: 0.2435 | g_loss: 2.9752
Epoch [   16/   20] | d_loss: 0.1629 | g_loss: 2.9210
Epoch [   16/   20] | d_loss: 0.2297 | g_loss: 2.6818
Epoch [   16/   20] | d_loss: 0.5623 | g_loss: 2.2055
Epoch [   16/   20] | d_loss: 0.1068 | g_loss: 4.1581
Epoch [   16/   20] | d_loss: 6.9496 | g_loss: 0.4812
Epoch [   16/   20] | d_loss: 0.2357 | g_loss: 2.8847
Epoch [   16/   20] | d_loss: 0.2054 | g_loss: 2.6302
Epoch [   16/   20] | d_loss: 0.3579 | g_loss: 3.3548
Epoch [   16/   20] | d_loss: 0.1948 | g_loss: 3.4114
Epoch [   16/   20] | d_loss: 0.2392 | g_loss: 3.0850
Epoch [   16/   20] | d_loss: 1.3598 | g_loss: 8.5041
Epoch [   16/   20] | d_loss: 0.1563 | g_loss: 3.4211
Epoch [   16/   20] | d_loss: 0.2474 | g_loss: 2.1108
Epoch [   16/   20] | d_loss: 1.4993 | g_loss: 0.3369
Epoch [   17/   20] | d_loss: 0.9490 | g_loss: 1.1543
Epoch [   17/   20] | d_loss: 0.1783 | g_loss: 3.0513
Epoch [   17/   20] | d_loss: 0.3056 | g_loss: 2.1673
Epoch [   17/   20] | d_loss: 0.3739 | g_loss: 1.5515
Epoch [   17/   20] | d_loss: 0.3309 | g_loss: 2.2020
Epoch [   17/   20] | d_loss: 0.2808 | g_loss: 2.7444
Epoch [   17/   20] | d_loss: 0.3331 | g_loss: 3.3483
Epoch [   17/   20] | d_loss: 0.2758 | g_loss: 2.7138
Epoch [   17/   20] | d_loss: 0.2029 | g_loss: 3.1554
Epoch [   17/   20] | d_loss: 0.1565 | g_loss: 3.0584
Epoch [   17/   20] | d_loss: 0.1861 | g_loss: 3.4521
Epoch [   17/   20] | d_loss: 0.2428 | g_loss: 4.0636
Epoch [   17/   20] | d_loss: 0.1830 | g_loss: 3.5414
Epoch [   17/   20] | d_loss: 0.1768 | g_loss: 2.5700
Epoch [   17/   20] | d_loss: 0.8087 | g_loss: 4.8428
Epoch [   17/   20] | d_loss: 0.1202 | g_loss: 2.8321
```

```
Epoch [   17/   20] | d_loss: 0.1923 | g_loss: 3.8523
Epoch [   17/   20] | d_loss: 0.2783 | g_loss: 4.0959
Epoch [   17/   20] | d_loss: 5.4934 | g_loss: 5.7584
Epoch [   17/   20] | d_loss: 0.2103 | g_loss: 3.1039
Epoch [   17/   20] | d_loss: 0.2679 | g_loss: 2.5930
Epoch [   17/   20] | d_loss: 0.1770 | g_loss: 3.0331
Epoch [   17/   20] | d_loss: 0.1319 | g_loss: 3.4606
Epoch [   17/   20] | d_loss: 0.8147 | g_loss: 4.8024
Epoch [   17/   20] | d_loss: 0.2145 | g_loss: 2.8052
Epoch [   17/   20] | d_loss: 0.1882 | g_loss: 3.7268
Epoch [   17/   20] | d_loss: 0.2629 | g_loss: 3.2929
Epoch [   17/   20] | d_loss: 0.3128 | g_loss: 4.5024
Epoch [   17/   20] | d_loss: 2.9165 | g_loss: 0.2659
Epoch [   18/   20] | d_loss: 0.6092 | g_loss: 4.8856
Epoch [   18/   20] | d_loss: 0.2071 | g_loss: 4.6217
Epoch [   18/   20] | d_loss: 0.1896 | g_loss: 3.2672
Epoch [   18/   20] | d_loss: 0.2932 | g_loss: 3.0835
Epoch [   18/   20] | d_loss: 0.1485 | g_loss: 3.4651
Epoch [   18/   20] | d_loss: 0.1148 | g_loss: 4.1680
Epoch [   18/   20] | d_loss: 0.1282 | g_loss: 3.8724
Epoch [   18/   20] | d_loss: 0.2809 | g_loss: 3.2912
Epoch [   18/   20] | d_loss: 0.4087 | g_loss: 3.4620
Epoch [   18/   20] | d_loss: 0.0889 | g_loss: 4.7212
Epoch [   18/   20] | d_loss: 0.0586 | g_loss: 4.5828
Epoch [   18/   20] | d_loss: 1.4665 | g_loss: 3.9411
Epoch [   18/   20] | d_loss: 0.1559 | g_loss: 4.1619
Epoch [   18/   20] | d_loss: 0.7814 | g_loss: 1.5535
Epoch [   18/   20] | d_loss: 0.2097 | g_loss: 3.1184
Epoch [   18/   20] | d_loss: 0.2117 | g_loss: 3.8850
Epoch [   18/   20] | d_loss: 0.1944 | g_loss: 3.0881
Epoch [   18/   20] | d_loss: 0.1735 | g_loss: 3.3185
Epoch [   18/   20] | d_loss: 0.2115 | g_loss: 2.4028
Epoch [   18/   20] | d_loss: 0.3132 | g_loss: 4.1629
Epoch [   18/   20] | d_loss: 0.6517 | g_loss: 2.5501
Epoch [   18/   20] | d_loss: 0.6377 | g_loss: 1.8514
Epoch [   18/   20] | d_loss: 0.2490 | g_loss: 1.7926
Epoch [   18/   20] | d_loss: 0.6949 | g_loss: 5.3362
Epoch [   18/   20] | d_loss: 0.2189 | g_loss: 4.1644
Epoch [   18/   20] | d_loss: 0.1340 | g_loss: 2.5085
Epoch [   18/   20] | d_loss: 0.1714 | g_loss: 4.2562
Epoch [   18/   20] | d_loss: 0.2821 | g_loss: 3.0614
Epoch [   18/   20] | d_loss: 1.6466 | g_loss: 1.4904
Epoch [   19/   20] | d_loss: 0.4878 | g_loss: 5.4886
Epoch [   19/   20] | d_loss: 0.1946 | g_loss: 4.5530
Epoch [   19/   20] | d_loss: 0.1646 | g_loss: 3.6879
Epoch [   19/   20] | d_loss: 0.1626 | g_loss: 2.3109
Epoch [   19/   20] | d_loss: 0.1732 | g_loss: 2.8991
Epoch [   19/   20] | d_loss: 0.8801 | g_loss: 1.3846
```

```
Epoch [   19/   20] | d_loss: 0.0933 | g_loss: 4.2965
Epoch [   19/   20] | d_loss: 0.1892 | g_loss: 3.1199
Epoch [   19/   20] | d_loss: 0.2212 | g_loss: 2.4643
Epoch [   19/   20] | d_loss: 0.5162 | g_loss: 2.0531
Epoch [   19/   20] | d_loss: 0.1806 | g_loss: 3.3383
Epoch [   19/   20] | d_loss: 0.2315 | g_loss: 2.6902
Epoch [   19/   20] | d_loss: 0.1161 | g_loss: 3.3685
Epoch [   19/   20] | d_loss: 0.1493 | g_loss: 2.8221
Epoch [   19/   20] | d_loss: 0.2927 | g_loss: 3.6095
Epoch [   19/   20] | d_loss: 0.7096 | g_loss: 5.4428
Epoch [   19/   20] | d_loss: 0.2799 | g_loss: 5.2975
Epoch [   19/   20] | d_loss: 0.4555 | g_loss: 3.7494
Epoch [   19/   20] | d_loss: 0.3033 | g_loss: 2.6532
Epoch [   19/   20] | d_loss: 0.4190 | g_loss: 3.5895
Epoch [   19/   20] | d_loss: 0.2795 | g_loss: 3.1797
Epoch [   19/   20] | d_loss: 0.1075 | g_loss: 4.2995
Epoch [   19/   20] | d_loss: 0.2913 | g_loss: 3.6233
Epoch [   19/   20] | d_loss: 0.1962 | g_loss: 3.0745
Epoch [   19/   20] | d_loss: 0.2043 | g_loss: 2.6045
Epoch [   19/   20] | d_loss: 0.1522 | g_loss: 2.9880
Epoch [   19/   20] | d_loss: 0.2300 | g_loss: 3.6926
Epoch [   19/   20] | d_loss: 0.0665 | g_loss: 3.3475
Epoch [   19/   20] | d_loss: 0.0913 | g_loss: 3.4912
Epoch [   20/   20] | d_loss: 0.1940 | g_loss: 3.5184
Epoch [   20/   20] | d_loss: 2.4026 | g_loss: 0.0855
Epoch [   20/   20] | d_loss: 0.1840 | g_loss: 3.8665
Epoch [   20/   20] | d_loss: 0.1295 | g_loss: 3.5306
Epoch [   20/   20] | d_loss: 0.1736 | g_loss: 4.2908
Epoch [   20/   20] | d_loss: 0.1588 | g_loss: 4.3619
Epoch [   20/   20] | d_loss: 0.2539 | g_loss: 2.3975
Epoch [   20/   20] | d_loss: 0.2040 | g_loss: 4.2976
Epoch [   20/   20] | d_loss: 0.1566 | g_loss: 3.5105
Epoch [   20/   20] | d_loss: 0.1003 | g_loss: 3.2687
Epoch [   20/   20] | d_loss: 0.4532 | g_loss: 2.6159
Epoch [   20/   20] | d_loss: 0.3370 | g_loss: 2.3743
Epoch [   20/   20] | d_loss: 0.1840 | g_loss: 3.4065
Epoch [   20/   20] | d_loss: 0.9408 | g_loss: 3.0633
Epoch [   20/   20] | d_loss: 0.1814 | g_loss: 3.6085
Epoch [   20/   20] | d_loss: 0.1990 | g_loss: 2.8849
Epoch [   20/   20] | d_loss: 0.1184 | g_loss: 3.8153
Epoch [   20/   20] | d_loss: 0.1796 | g_loss: 4.5169
Epoch [   20/   20] | d_loss: 0.1464 | g_loss: 3.9888
Epoch [   20/   20] | d_loss: 0.1732 | g_loss: 4.0055
Epoch [   20/   20] | d_loss: 0.0974 | g_loss: 3.1928
Epoch [   20/   20] | d_loss: 0.1324 | g_loss: 4.2492
Epoch [   20/   20] | d_loss: 0.2548 | g_loss: 3.1733
Epoch [   20/   20] | d_loss: 0.1781 | g_loss: 3.0097
Epoch [   20/   20] | d_loss: 0.1619 | g_loss: 2.9121
```

```
Epoch [   20/   20] | d_loss: 0.2249 | g_loss: 4.0663
Epoch [   20/   20] | d_loss: 0.1518 | g_loss: 1.1877
Epoch [   20/   20] | d_loss: 0.1212 | g_loss: 3.6681
Epoch [   20/   20] | d_loss: 0.1209 | g_loss: 4.2359
```

2.8 Training loss

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

In [46]:
```
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[46]: <matplotlib.legend.Legend at 0x7f9935d7f7b8>

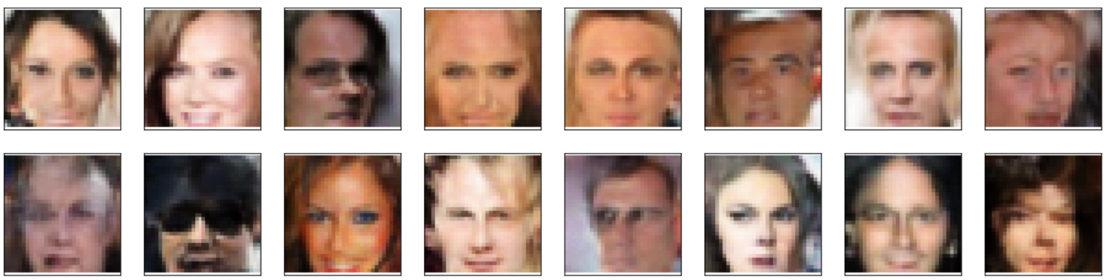
2.9 Generator samples from training

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

```
In [47]: # 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 [48]: # Load samples from generator, taken while training
         with open('train_samples.pkl', 'rb') as f:
             samples = pkl.load(f)
```

```
In [49]: _ = 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: The generated samples seems to be realistic but most the images are blurry. With the specified hyper parameters, this model could this at its best. But, I feel tuning the hyperparameters could make the model learn much better. And, with increase in number of epochs would be 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.