dlnd_face_generation

May 10, 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 [ ]: # 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:
```

```
In [1]: 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 [2]:
```
# necessary imports
import torch
from torchvision import datasets
from torchvision import transforms
from torch.utils.data import DataLoader
```

In [3]:
```
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()
    ])

    dataset = datasets.ImageFolder(data_dir, transform)
    loader = DataLoader(dataset, 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 [4]: # Define function hyperparameters
        batch_size = 32
        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)
```

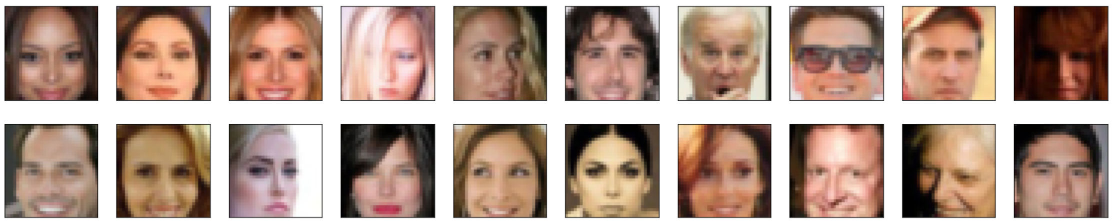
Next, you can view some images! You should seen square images of somewhat-centered faces.
Note: You'll need to convert the Tensor images into a NumPy type and transpose the dimensions to correctly display an image, suggested `imshow` code is below, but it may not be perfect.

```
In [5]: # 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 [6]:
```
# 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
    range_min, range_max = feature_range
    return (range_max - range_min) * x + range_min

    return x
```

In [7]:
```
"""
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(-0.9686)
Max:  tensor(0.7725)
```

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 [8]:
```
import torch.nn as nn
import torch.nn.functional as F
```

In [9]:
```
def conv(inputs, outputs, kernal=4, stride=2, padding=1, batch_norm=True):
    layers=[]
    conv_layer = nn.Conv2d(inputs, outputs, kernal, stride, padding, bias=False)
    layers.append(conv_layer)

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

    return nn.Sequential(*layers)

def de_conv(inputs, outputs, kernal=4, stride=2, padding=1, batch_norm=True):
    layers=[]
    deconv_layer = nn.ConvTranspose2d(inputs, outputs, kernal, stride, padding, bias=Fal
    layers.append(deconv_layer)

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

    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__()

        # complete init function
```

```
        self.conv_dim = conv_dim
        self.leaky_relu = 0.2

        self.conv1 = conv(3, conv_dim, batch_norm=False)
        self.conv2 = conv(conv_dim, conv_dim*2)
        self.conv3 = conv(conv_dim*2, conv_dim*4)
        self.conv4 = conv(conv_dim*4, conv_dim*8)

        self.fc = nn.Linear(conv_dim*32, 1)

        self.convolution = [
            self.conv1, self.conv2, self.conv3, self.conv4
        ]


    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

        for i,layer in enumerate(self.convolution):
            x = F.leaky_relu(layer(x), self.leaky_relu)

        x = x.view(-1, self.conv_dim*32)
        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]:
```
class Generator(nn.Module):

    def __init__(self, z_size, conv_dim):
        """
        Initialize the Generator Module
        :param z_size: The length of the input latent vector, z
        :param conv_dim: The depth of the inputs to the *last* transpose convolutional
        """
        super(Generator, self).__init__()

        # complete init function

        self.conv_dim = conv_dim

        self.de_conv1 = de_conv(conv_dim*8, conv_dim*4)
        self.de_conv2 = de_conv(conv_dim*4, conv_dim*2)
        self.de_conv3 = de_conv(conv_dim*2, conv_dim)
        self.de_conv4 = de_conv(conv_dim, 3, batch_norm=False)

        self.fc = nn.Linear(z_size, conv_dim * 32)

        self.de_convolution = [
            self.de_conv1, self.de_conv2, self.de_conv3, self.de_conv4
        ]


    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)
        batch_size = x.shape[0]
        x = x.view(batch_size, self.conv_dim*8, 2, 2)

        for i, layer in enumerate(self.de_convolution):
            if (i < len(self.de_convolution)-1):
                x = F.relu(layer(x))
            else:
                x = layer(x)

        x = torch.tanh(x)
```

```
        return x

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

```
Tests Passed
```

2.3 Initialize the weights of your networks

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

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

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

Exercise: Complete the weight initialization function

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

In [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 'Conv' in classname:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif 'Linear' in classname:
        torch.nn.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 = 32
g_conv_dim = 32
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, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
  (conv2): Sequential(
    (0): Conv2d(32, 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)
  )
  (conv3): 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)
  )
  (conv4): 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=1024, out_features=1, bias=True)
)
```

```
Generator(
  (de_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)
  )
  (de_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)
  )
  (de_conv3): Sequential(
    (0): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (de_conv4): Sequential(
    (0): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
  (fc): Linear(in_features=100, out_features=1024, bias=True)
)
```

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, smooth=False):
             '''Calculates how close discriminator outputs are to being real.
                param, D_out: discriminator logits
                return: real loss'''
             batch_size = D_out.shape[0]
             if smooth:
                 labels = torch.ones(batch_size)*0.9
             else:
                 labels = torch.ones(batch_size)

             labels = labels.to(device)
             criterion = nn.BCEWithLogitsLoss()
             loss = criterion(D_out.squeeze(), labels)
             return loss

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

         # Create optimizers for the discriminator D and generator G
```

```
beta1 = 0.5
beta2 = 0.99
lr = 0.0002

d_optimizer = optim.Adam(D.parameters(), lr, betas=(beta1, beta2))
g_optimizer = optim.Adam(G.parameters(), lr, betas=(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:
```

```python
    fixed_z = fixed_z.cuda()

# epoch training loop
for epoch in range(n_epochs):

    # batch training loop
    for batch_i, (real_images, _) in enumerate(celeba_train_loader):

        batch_size = real_images.size(0)
        real_images = scale(real_images)

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

        # 1. Train the discriminator on real and fake images
        #real
        d_optimizer.zero_grad()
        real_images = real_images.to(device)
        D_real = D(real_images)
        d_real_loss = real_loss(D_real)

        #fake
        z = np.random.uniform(-1, 1, size=(batch_size, z_size))
        z = torch.from_numpy(z).float()
        # move x to GPU, if available
        z = z.to(device)
        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()

        #fake
        z = np.random.uniform(-1, 1, size=(batch_size, z_size))
        z = torch.from_numpy(z).float()
        z = z.to(device)
        fake_images = G(z)

        D_fake = D(fake_images)
        g_loss = real_loss(D_fake, smooth=True)

        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 [20]:
```
# set number of epochs
n_epochs = 10
device = 'cuda' if torch.cuda.is_available() else 'cpu'
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# call training function
losses = train(D, G, n_epochs=n_epochs)
```

```
Epoch [    1/   10] | d_loss: 1.7115 | g_loss: 2.2706
Epoch [    1/   10] | d_loss: 0.7443 | g_loss: 2.3955
Epoch [    1/   10] | d_loss: 0.4676 | g_loss: 2.4537
Epoch [    1/   10] | d_loss: 1.6854 | g_loss: 4.4466
Epoch [    1/   10] | d_loss: 0.8940 | g_loss: 3.6208
Epoch [    1/   10] | d_loss: 0.7626 | g_loss: 3.5884
Epoch [    1/   10] | d_loss: 0.8584 | g_loss: 2.2259
```

```
Epoch [    1/   10] | d_loss: 1.1342 | g_loss: 3.0811
Epoch [    1/   10] | d_loss: 0.9293 | g_loss: 3.3216
Epoch [    1/   10] | d_loss: 0.4989 | g_loss: 2.6375
Epoch [    1/   10] | d_loss: 1.6300 | g_loss: 3.9620
Epoch [    1/   10] | d_loss: 0.7727 | g_loss: 2.1940
Epoch [    1/   10] | d_loss: 0.9882 | g_loss: 3.4171
Epoch [    1/   10] | d_loss: 0.8035 | g_loss: 2.2798
Epoch [    1/   10] | d_loss: 1.3252 | g_loss: 1.4092
Epoch [    1/   10] | d_loss: 0.5580 | g_loss: 2.4405
Epoch [    1/   10] | d_loss: 0.5866 | g_loss: 2.4308
Epoch [    1/   10] | d_loss: 0.9186 | g_loss: 1.5644
Epoch [    1/   10] | d_loss: 0.5873 | g_loss: 2.9115
Epoch [    1/   10] | d_loss: 0.9033 | g_loss: 0.7105
Epoch [    1/   10] | d_loss: 0.7640 | g_loss: 3.5747
Epoch [    1/   10] | d_loss: 0.5752 | g_loss: 2.3450
Epoch [    1/   10] | d_loss: 0.6357 | g_loss: 1.5810
Epoch [    1/   10] | d_loss: 0.3571 | g_loss: 2.3716
Epoch [    1/   10] | d_loss: 1.0642 | g_loss: 0.8850
Epoch [    1/   10] | d_loss: 0.7865 | g_loss: 1.8510
Epoch [    1/   10] | d_loss: 0.5513 | g_loss: 2.8403
Epoch [    1/   10] | d_loss: 0.4082 | g_loss: 1.7853
Epoch [    1/   10] | d_loss: 0.6556 | g_loss: 2.8476
Epoch [    1/   10] | d_loss: 0.9868 | g_loss: 1.4377
Epoch [    1/   10] | d_loss: 0.5688 | g_loss: 1.6313
Epoch [    1/   10] | d_loss: 0.7844 | g_loss: 2.9035
Epoch [    1/   10] | d_loss: 0.5837 | g_loss: 2.2894
Epoch [    1/   10] | d_loss: 0.8679 | g_loss: 1.0470
Epoch [    1/   10] | d_loss: 0.7063 | g_loss: 1.7445
Epoch [    1/   10] | d_loss: 0.6011 | g_loss: 2.7372
Epoch [    1/   10] | d_loss: 0.9527 | g_loss: 2.8324
Epoch [    1/   10] | d_loss: 0.8678 | g_loss: 1.1892
Epoch [    1/   10] | d_loss: 0.9354 | g_loss: 0.8276
Epoch [    1/   10] | d_loss: 0.6608 | g_loss: 2.0721
Epoch [    1/   10] | d_loss: 0.8975 | g_loss: 0.7926
Epoch [    1/   10] | d_loss: 1.3824 | g_loss: 3.5285
Epoch [    1/   10] | d_loss: 0.5324 | g_loss: 2.5461
Epoch [    1/   10] | d_loss: 0.4394 | g_loss: 2.0983
Epoch [    1/   10] | d_loss: 0.6907 | g_loss: 3.9815
Epoch [    1/   10] | d_loss: 0.4154 | g_loss: 2.8305
Epoch [    1/   10] | d_loss: 0.9881 | g_loss: 1.5822
Epoch [    1/   10] | d_loss: 0.6511 | g_loss: 2.6505
Epoch [    1/   10] | d_loss: 0.6348 | g_loss: 2.6582
Epoch [    1/   10] | d_loss: 0.8157 | g_loss: 3.1759
Epoch [    1/   10] | d_loss: 0.8979 | g_loss: 1.0187
Epoch [    1/   10] | d_loss: 0.8439 | g_loss: 1.4173
Epoch [    1/   10] | d_loss: 0.8668 | g_loss: 2.3040
Epoch [    1/   10] | d_loss: 0.4373 | g_loss: 1.9389
Epoch [    1/   10] | d_loss: 1.0480 | g_loss: 2.2979
```

```
Epoch [    1/   10] | d_loss: 0.3981 | g_loss: 2.6470
Epoch [    1/   10] | d_loss: 0.7924 | g_loss: 1.0274
Epoch [    2/   10] | d_loss: 0.7896 | g_loss: 2.0232
Epoch [    2/   10] | d_loss: 0.3375 | g_loss: 2.9182
Epoch [    2/   10] | d_loss: 0.5871 | g_loss: 1.7161
Epoch [    2/   10] | d_loss: 0.5284 | g_loss: 2.4899
Epoch [    2/   10] | d_loss: 0.6491 | g_loss: 2.3323
Epoch [    2/   10] | d_loss: 0.8323 | g_loss: 3.8602
Epoch [    2/   10] | d_loss: 0.6761 | g_loss: 1.4122
Epoch [    2/   10] | d_loss: 0.6525 | g_loss: 2.1567
Epoch [    2/   10] | d_loss: 0.8215 | g_loss: 2.0384
Epoch [    2/   10] | d_loss: 1.0254 | g_loss: 2.9662
Epoch [    2/   10] | d_loss: 1.0013 | g_loss: 3.0924
Epoch [    2/   10] | d_loss: 0.5975 | g_loss: 1.3381
Epoch [    2/   10] | d_loss: 0.5315 | g_loss: 3.4277
Epoch [    2/   10] | d_loss: 0.3152 | g_loss: 2.7944
Epoch [    2/   10] | d_loss: 0.6808 | g_loss: 1.9125
Epoch [    2/   10] | d_loss: 1.0215 | g_loss: 1.1895
Epoch [    2/   10] | d_loss: 0.6781 | g_loss: 2.2308
Epoch [    2/   10] | d_loss: 0.7566 | g_loss: 0.7707
Epoch [    2/   10] | d_loss: 0.6821 | g_loss: 1.0421
Epoch [    2/   10] | d_loss: 0.5882 | g_loss: 3.3965
Epoch [    2/   10] | d_loss: 0.7273 | g_loss: 1.4877
Epoch [    2/   10] | d_loss: 0.9883 | g_loss: 2.9593
Epoch [    2/   10] | d_loss: 1.2546 | g_loss: 3.6495
Epoch [    2/   10] | d_loss: 0.7285 | g_loss: 2.1267
Epoch [    2/   10] | d_loss: 0.3772 | g_loss: 1.4307
Epoch [    2/   10] | d_loss: 0.4989 | g_loss: 3.0772
Epoch [    2/   10] | d_loss: 0.8510 | g_loss: 1.6622
Epoch [    2/   10] | d_loss: 0.5411 | g_loss: 3.1863
Epoch [    2/   10] | d_loss: 0.5133 | g_loss: 2.3323
Epoch [    2/   10] | d_loss: 0.3716 | g_loss: 1.6744
Epoch [    2/   10] | d_loss: 0.8080 | g_loss: 1.2851
Epoch [    2/   10] | d_loss: 0.5318 | g_loss: 2.6421
Epoch [    2/   10] | d_loss: 1.0951 | g_loss: 1.2054
Epoch [    2/   10] | d_loss: 1.2779 | g_loss: 4.5683
Epoch [    2/   10] | d_loss: 0.6260 | g_loss: 1.4580
Epoch [    2/   10] | d_loss: 0.4077 | g_loss: 2.3518
Epoch [    2/   10] | d_loss: 1.0708 | g_loss: 2.2022
Epoch [    2/   10] | d_loss: 0.4783 | g_loss: 1.5167
Epoch [    2/   10] | d_loss: 0.5930 | g_loss: 1.9109
Epoch [    2/   10] | d_loss: 0.4911 | g_loss: 2.6907
Epoch [    2/   10] | d_loss: 0.7575 | g_loss: 1.4142
Epoch [    2/   10] | d_loss: 1.0184 | g_loss: 1.2809
Epoch [    2/   10] | d_loss: 0.5499 | g_loss: 1.4392
Epoch [    2/   10] | d_loss: 0.4195 | g_loss: 1.6923
Epoch [    2/   10] | d_loss: 0.4203 | g_loss: 1.8970
Epoch [    2/   10] | d_loss: 0.4612 | g_loss: 2.4945
```

```
Epoch [    2/   10] | d_loss: 0.6782 | g_loss: 2.0660
Epoch [    2/   10] | d_loss: 0.5341 | g_loss: 2.6423
Epoch [    2/   10] | d_loss: 0.7139 | g_loss: 1.7435
Epoch [    2/   10] | d_loss: 0.8540 | g_loss: 2.4712
Epoch [    2/   10] | d_loss: 0.5418 | g_loss: 1.9811
Epoch [    2/   10] | d_loss: 0.5338 | g_loss: 2.5009
Epoch [    2/   10] | d_loss: 0.3712 | g_loss: 1.9275
Epoch [    2/   10] | d_loss: 0.5476 | g_loss: 2.1624
Epoch [    2/   10] | d_loss: 0.1926 | g_loss: 3.5388
Epoch [    2/   10] | d_loss: 0.4271 | g_loss: 1.1422
Epoch [    2/   10] | d_loss: 0.2904 | g_loss: 1.6636
Epoch [    3/   10] | d_loss: 0.8633 | g_loss: 1.0045
Epoch [    3/   10] | d_loss: 0.5527 | g_loss: 1.7300
Epoch [    3/   10] | d_loss: 0.4856 | g_loss: 1.8126
Epoch [    3/   10] | d_loss: 0.7660 | g_loss: 1.8152
Epoch [    3/   10] | d_loss: 0.4571 | g_loss: 1.1599
Epoch [    3/   10] | d_loss: 0.7637 | g_loss: 0.6860
Epoch [    3/   10] | d_loss: 0.2336 | g_loss: 3.0185
Epoch [    3/   10] | d_loss: 0.7245 | g_loss: 1.5691
Epoch [    3/   10] | d_loss: 0.6492 | g_loss: 0.9146
Epoch [    3/   10] | d_loss: 0.5537 | g_loss: 2.0714
Epoch [    3/   10] | d_loss: 1.0603 | g_loss: 3.4992
Epoch [    3/   10] | d_loss: 0.5980 | g_loss: 2.8993
Epoch [    3/   10] | d_loss: 0.6849 | g_loss: 2.5239
Epoch [    3/   10] | d_loss: 0.6115 | g_loss: 2.3783
Epoch [    3/   10] | d_loss: 1.1736 | g_loss: 3.1302
Epoch [    3/   10] | d_loss: 0.4761 | g_loss: 1.9497
Epoch [    3/   10] | d_loss: 0.6053 | g_loss: 2.6631
Epoch [    3/   10] | d_loss: 0.5631 | g_loss: 2.7001
Epoch [    3/   10] | d_loss: 0.6399 | g_loss: 2.7419
Epoch [    3/   10] | d_loss: 0.8201 | g_loss: 2.2091
Epoch [    3/   10] | d_loss: 0.7554 | g_loss: 0.9881
Epoch [    3/   10] | d_loss: 0.5794 | g_loss: 2.6246
Epoch [    3/   10] | d_loss: 0.9323 | g_loss: 2.4523
Epoch [    3/   10] | d_loss: 0.4391 | g_loss: 1.5727
Epoch [    3/   10] | d_loss: 1.0861 | g_loss: 2.2472
Epoch [    3/   10] | d_loss: 0.7096 | g_loss: 1.8702
Epoch [    3/   10] | d_loss: 0.6458 | g_loss: 2.2493
Epoch [    3/   10] | d_loss: 0.6543 | g_loss: 0.8500
Epoch [    3/   10] | d_loss: 0.9532 | g_loss: 4.1090
Epoch [    3/   10] | d_loss: 0.7020 | g_loss: 1.7076
Epoch [    3/   10] | d_loss: 0.9680 | g_loss: 1.4221
Epoch [    3/   10] | d_loss: 0.5519 | g_loss: 2.0259
Epoch [    3/   10] | d_loss: 0.7625 | g_loss: 1.4897
Epoch [    3/   10] | d_loss: 0.3650 | g_loss: 2.4957
Epoch [    3/   10] | d_loss: 0.5794 | g_loss: 1.5785
Epoch [    3/   10] | d_loss: 0.4972 | g_loss: 1.6989
Epoch [    3/   10] | d_loss: 0.8968 | g_loss: 2.1459
```

```
Epoch [    3/   10] | d_loss: 0.6044 | g_loss: 2.7910
Epoch [    3/   10] | d_loss: 0.5015 | g_loss: 2.4364
Epoch [    3/   10] | d_loss: 0.7287 | g_loss: 2.5512
Epoch [    3/   10] | d_loss: 0.7995 | g_loss: 1.4408
Epoch [    3/   10] | d_loss: 0.4359 | g_loss: 2.1850
Epoch [    3/   10] | d_loss: 0.6319 | g_loss: 3.5223
Epoch [    3/   10] | d_loss: 0.5977 | g_loss: 2.5163
Epoch [    3/   10] | d_loss: 1.4267 | g_loss: 3.3067
Epoch [    3/   10] | d_loss: 0.4107 | g_loss: 1.9202
Epoch [    3/   10] | d_loss: 0.6788 | g_loss: 1.5124
Epoch [    3/   10] | d_loss: 0.6364 | g_loss: 2.0006
Epoch [    3/   10] | d_loss: 0.7304 | g_loss: 1.4181
Epoch [    3/   10] | d_loss: 0.7998 | g_loss: 1.0347
Epoch [    3/   10] | d_loss: 0.8433 | g_loss: 1.1642
Epoch [    3/   10] | d_loss: 0.5176 | g_loss: 1.8488
Epoch [    3/   10] | d_loss: 0.5891 | g_loss: 2.1747
Epoch [    3/   10] | d_loss: 1.0920 | g_loss: 3.5724
Epoch [    3/   10] | d_loss: 0.5468 | g_loss: 2.7583
Epoch [    3/   10] | d_loss: 0.7107 | g_loss: 1.6771
Epoch [    3/   10] | d_loss: 0.2932 | g_loss: 2.9998
Epoch [    4/   10] | d_loss: 0.9015 | g_loss: 2.1378
Epoch [    4/   10] | d_loss: 0.7213 | g_loss: 2.2498
Epoch [    4/   10] | d_loss: 0.5972 | g_loss: 1.1661
Epoch [    4/   10] | d_loss: 0.3927 | g_loss: 1.4921
Epoch [    4/   10] | d_loss: 0.9312 | g_loss: 2.5162
Epoch [    4/   10] | d_loss: 0.5138 | g_loss: 1.6360
Epoch [    4/   10] | d_loss: 1.4394 | g_loss: 3.2950
Epoch [    4/   10] | d_loss: 0.5988 | g_loss: 1.9539
Epoch [    4/   10] | d_loss: 0.7612 | g_loss: 2.1098
Epoch [    4/   10] | d_loss: 0.7648 | g_loss: 1.3968
Epoch [    4/   10] | d_loss: 0.3521 | g_loss: 1.8148
Epoch [    4/   10] | d_loss: 0.5689 | g_loss: 1.8820
Epoch [    4/   10] | d_loss: 0.7619 | g_loss: 0.3939
Epoch [    4/   10] | d_loss: 0.6390 | g_loss: 3.2382
Epoch [    4/   10] | d_loss: 0.4282 | g_loss: 1.7232
Epoch [    4/   10] | d_loss: 0.8581 | g_loss: 2.1954
Epoch [    4/   10] | d_loss: 0.6243 | g_loss: 1.7216
Epoch [    4/   10] | d_loss: 0.4344 | g_loss: 1.9251
Epoch [    4/   10] | d_loss: 1.4966 | g_loss: 1.3406
Epoch [    4/   10] | d_loss: 0.5978 | g_loss: 2.0464
Epoch [    4/   10] | d_loss: 0.8089 | g_loss: 2.9311
Epoch [    4/   10] | d_loss: 0.4578 | g_loss: 2.0656
Epoch [    4/   10] | d_loss: 0.6151 | g_loss: 1.2443
Epoch [    4/   10] | d_loss: 0.5182 | g_loss: 1.8804
Epoch [    4/   10] | d_loss: 0.3674 | g_loss: 1.6941
Epoch [    4/   10] | d_loss: 1.3272 | g_loss: 2.7521
Epoch [    4/   10] | d_loss: 0.3595 | g_loss: 2.8968
Epoch [    4/   10] | d_loss: 0.6282 | g_loss: 1.4007
```

```
Epoch [    4/   10] | d_loss: 0.7516 | g_loss: 1.9041
Epoch [    4/   10] | d_loss: 1.1822 | g_loss: 1.0112
Epoch [    4/   10] | d_loss: 1.1062 | g_loss: 3.3300
Epoch [    4/   10] | d_loss: 1.1017 | g_loss: 1.2293
Epoch [    4/   10] | d_loss: 0.7470 | g_loss: 2.3250
Epoch [    4/   10] | d_loss: 1.8155 | g_loss: 2.9126
Epoch [    4/   10] | d_loss: 0.3193 | g_loss: 1.3285
Epoch [    4/   10] | d_loss: 1.2392 | g_loss: 3.9527
Epoch [    4/   10] | d_loss: 0.8219 | g_loss: 1.8752
Epoch [    4/   10] | d_loss: 0.5929 | g_loss: 2.6627
Epoch [    4/   10] | d_loss: 0.7172 | g_loss: 2.4230
Epoch [    4/   10] | d_loss: 1.0219 | g_loss: 0.9887
Epoch [    4/   10] | d_loss: 0.6096 | g_loss: 1.6153
Epoch [    4/   10] | d_loss: 0.4698 | g_loss: 1.7044
Epoch [    4/   10] | d_loss: 0.6843 | g_loss: 2.2335
Epoch [    4/   10] | d_loss: 0.5795 | g_loss: 1.4392
Epoch [    4/   10] | d_loss: 0.4860 | g_loss: 1.2465
Epoch [    4/   10] | d_loss: 1.0171 | g_loss: 3.8642
Epoch [    4/   10] | d_loss: 0.7183 | g_loss: 1.7711
Epoch [    4/   10] | d_loss: 0.5891 | g_loss: 1.0894
Epoch [    4/   10] | d_loss: 0.4070 | g_loss: 1.9708
Epoch [    4/   10] | d_loss: 1.0244 | g_loss: 3.4204
Epoch [    4/   10] | d_loss: 0.9490 | g_loss: 1.9569
Epoch [    4/   10] | d_loss: 0.5189 | g_loss: 1.8108
Epoch [    4/   10] | d_loss: 0.3671 | g_loss: 1.5830
Epoch [    4/   10] | d_loss: 0.9073 | g_loss: 1.9855
Epoch [    4/   10] | d_loss: 0.5468 | g_loss: 1.5389
Epoch [    4/   10] | d_loss: 0.8532 | g_loss: 1.6275
Epoch [    4/   10] | d_loss: 0.6218 | g_loss: 2.6154
Epoch [    5/   10] | d_loss: 0.5621 | g_loss: 3.7957
Epoch [    5/   10] | d_loss: 0.5437 | g_loss: 1.6474
Epoch [    5/   10] | d_loss: 0.4997 | g_loss: 2.3293
Epoch [    5/   10] | d_loss: 0.6015 | g_loss: 2.8911
Epoch [    5/   10] | d_loss: 0.6863 | g_loss: 2.4482
Epoch [    5/   10] | d_loss: 0.6993 | g_loss: 2.0711
Epoch [    5/   10] | d_loss: 0.2809 | g_loss: 2.9249
Epoch [    5/   10] | d_loss: 0.7556 | g_loss: 2.2413
Epoch [    5/   10] | d_loss: 0.4557 | g_loss: 2.7869
Epoch [    5/   10] | d_loss: 0.3189 | g_loss: 1.9697
Epoch [    5/   10] | d_loss: 0.6691 | g_loss: 2.7934
Epoch [    5/   10] | d_loss: 0.6804 | g_loss: 4.4354
Epoch [    5/   10] | d_loss: 0.8726 | g_loss: 1.1672
Epoch [    5/   10] | d_loss: 0.5179 | g_loss: 2.6936
Epoch [    5/   10] | d_loss: 0.6322 | g_loss: 1.6396
Epoch [    5/   10] | d_loss: 0.4494 | g_loss: 2.8005
Epoch [    5/   10] | d_loss: 0.5443 | g_loss: 2.1698
Epoch [    5/   10] | d_loss: 0.4348 | g_loss: 1.9405
Epoch [    5/   10] | d_loss: 0.4524 | g_loss: 1.0625
```

```
Epoch [    5/   10] | d_loss: 1.2304 | g_loss: 0.9984
Epoch [    5/   10] | d_loss: 0.6291 | g_loss: 1.9005
Epoch [    5/   10] | d_loss: 0.9000 | g_loss: 2.6962
Epoch [    5/   10] | d_loss: 0.3193 | g_loss: 1.6385
Epoch [    5/   10] | d_loss: 0.4907 | g_loss: 2.0436
Epoch [    5/   10] | d_loss: 0.6386 | g_loss: 1.6151
Epoch [    5/   10] | d_loss: 0.2956 | g_loss: 2.7369
Epoch [    5/   10] | d_loss: 0.5257 | g_loss: 2.8186
Epoch [    5/   10] | d_loss: 0.4820 | g_loss: 2.0985
Epoch [    5/   10] | d_loss: 0.4024 | g_loss: 1.4484
Epoch [    5/   10] | d_loss: 1.8589 | g_loss: 3.7317
Epoch [    5/   10] | d_loss: 0.6443 | g_loss: 1.7527
Epoch [    5/   10] | d_loss: 0.7990 | g_loss: 2.4372
Epoch [    5/   10] | d_loss: 0.4462 | g_loss: 3.0554
Epoch [    5/   10] | d_loss: 0.6932 | g_loss: 2.6485
Epoch [    5/   10] | d_loss: 0.6084 | g_loss: 1.0946
Epoch [    5/   10] | d_loss: 0.4487 | g_loss: 1.7274
Epoch [    5/   10] | d_loss: 0.7716 | g_loss: 2.0088
Epoch [    5/   10] | d_loss: 0.5378 | g_loss: 2.2775
Epoch [    5/   10] | d_loss: 0.3841 | g_loss: 2.2454
Epoch [    5/   10] | d_loss: 0.5907 | g_loss: 2.7446
Epoch [    5/   10] | d_loss: 0.8011 | g_loss: 4.0820
Epoch [    5/   10] | d_loss: 0.5521 | g_loss: 1.1059
Epoch [    5/   10] | d_loss: 0.6623 | g_loss: 1.9052
Epoch [    5/   10] | d_loss: 0.2745 | g_loss: 1.8911
Epoch [    5/   10] | d_loss: 0.8752 | g_loss: 3.0781
Epoch [    5/   10] | d_loss: 1.4385 | g_loss: 1.0329
Epoch [    5/   10] | d_loss: 0.6415 | g_loss: 3.3822
Epoch [    5/   10] | d_loss: 0.4821 | g_loss: 1.8818
Epoch [    5/   10] | d_loss: 0.4351 | g_loss: 2.8028
Epoch [    5/   10] | d_loss: 0.8254 | g_loss: 2.1699
Epoch [    5/   10] | d_loss: 0.7568 | g_loss: 0.9907
Epoch [    5/   10] | d_loss: 1.1123 | g_loss: 0.9164
Epoch [    5/   10] | d_loss: 0.6718 | g_loss: 3.0858
Epoch [    5/   10] | d_loss: 0.3089 | g_loss: 3.1048
Epoch [    5/   10] | d_loss: 0.3427 | g_loss: 2.7552
Epoch [    5/   10] | d_loss: 0.9910 | g_loss: 0.5319
Epoch [    5/   10] | d_loss: 0.5985 | g_loss: 3.1299
Epoch [    6/   10] | d_loss: 0.4396 | g_loss: 2.5269
Epoch [    6/   10] | d_loss: 0.6009 | g_loss: 2.4385
Epoch [    6/   10] | d_loss: 0.3583 | g_loss: 2.9915
Epoch [    6/   10] | d_loss: 0.2391 | g_loss: 1.4178
Epoch [    6/   10] | d_loss: 0.2577 | g_loss: 2.2010
Epoch [    6/   10] | d_loss: 0.6858 | g_loss: 1.6638
Epoch [    6/   10] | d_loss: 0.2814 | g_loss: 3.0442
Epoch [    6/   10] | d_loss: 0.7307 | g_loss: 1.7278
Epoch [    6/   10] | d_loss: 0.6612 | g_loss: 1.5871
Epoch [    6/   10] | d_loss: 0.7043 | g_loss: 1.5152
```

```
Epoch [    6/   10] | d_loss: 0.3434 | g_loss: 2.0879
Epoch [    6/   10] | d_loss: 0.8519 | g_loss: 2.7044
Epoch [    6/   10] | d_loss: 0.6685 | g_loss: 1.4388
Epoch [    6/   10] | d_loss: 0.3319 | g_loss: 2.5412
Epoch [    6/   10] | d_loss: 0.4434 | g_loss: 2.3971
Epoch [    6/   10] | d_loss: 0.4779 | g_loss: 3.6005
Epoch [    6/   10] | d_loss: 0.3130 | g_loss: 2.6579
Epoch [    6/   10] | d_loss: 0.6440 | g_loss: 1.9269
Epoch [    6/   10] | d_loss: 0.6009 | g_loss: 2.8376
Epoch [    6/   10] | d_loss: 1.2174 | g_loss: 2.2058
Epoch [    6/   10] | d_loss: 0.4723 | g_loss: 0.7935
Epoch [    6/   10] | d_loss: 0.5363 | g_loss: 2.2827
Epoch [    6/   10] | d_loss: 0.3252 | g_loss: 2.0760
Epoch [    6/   10] | d_loss: 0.9384 | g_loss: 0.7853
Epoch [    6/   10] | d_loss: 0.1345 | g_loss: 3.1798
Epoch [    6/   10] | d_loss: 0.9301 | g_loss: 2.7628
Epoch [    6/   10] | d_loss: 0.5829 | g_loss: 2.6293
Epoch [    6/   10] | d_loss: 0.5576 | g_loss: 2.7962
Epoch [    6/   10] | d_loss: 0.5857 | g_loss: 1.3247
Epoch [    6/   10] | d_loss: 0.6053 | g_loss: 2.3104
Epoch [    6/   10] | d_loss: 0.3257 | g_loss: 2.4557
Epoch [    6/   10] | d_loss: 0.3842 | g_loss: 2.3979
Epoch [    6/   10] | d_loss: 0.4198 | g_loss: 1.6626
Epoch [    6/   10] | d_loss: 0.3384 | g_loss: 1.6562
Epoch [    6/   10] | d_loss: 0.2620 | g_loss: 2.7809
Epoch [    6/   10] | d_loss: 0.6066 | g_loss: 4.0367
Epoch [    6/   10] | d_loss: 0.4981 | g_loss: 2.5724
Epoch [    6/   10] | d_loss: 0.1764 | g_loss: 2.9463
Epoch [    6/   10] | d_loss: 0.1821 | g_loss: 2.5285
Epoch [    6/   10] | d_loss: 0.3926 | g_loss: 2.7258
Epoch [    6/   10] | d_loss: 0.6973 | g_loss: 2.1105
Epoch [    6/   10] | d_loss: 1.1481 | g_loss: 2.3339
Epoch [    6/   10] | d_loss: 0.7756 | g_loss: 1.2690
Epoch [    6/   10] | d_loss: 0.8047 | g_loss: 2.5589
Epoch [    6/   10] | d_loss: 0.5113 | g_loss: 4.2425
Epoch [    6/   10] | d_loss: 0.7882 | g_loss: 1.4130
Epoch [    6/   10] | d_loss: 0.4896 | g_loss: 0.7886
Epoch [    6/   10] | d_loss: 0.4630 | g_loss: 0.9908
Epoch [    6/   10] | d_loss: 0.4288 | g_loss: 1.4029
Epoch [    6/   10] | d_loss: 0.2458 | g_loss: 2.7263
Epoch [    6/   10] | d_loss: 0.3363 | g_loss: 1.9026
Epoch [    6/   10] | d_loss: 0.1996 | g_loss: 3.3890
Epoch [    6/   10] | d_loss: 0.6478 | g_loss: 1.5309
Epoch [    6/   10] | d_loss: 0.6421 | g_loss: 0.7192
Epoch [    6/   10] | d_loss: 1.1349 | g_loss: 0.7827
Epoch [    6/   10] | d_loss: 1.6382 | g_loss: 2.9540
Epoch [    6/   10] | d_loss: 0.5401 | g_loss: 3.5973
Epoch [    7/   10] | d_loss: 0.4056 | g_loss: 2.6570
```

```
Epoch [    7/   10] | d_loss: 0.4164 | g_loss: 1.8527
Epoch [    7/   10] | d_loss: 0.4159 | g_loss: 2.1574
Epoch [    7/   10] | d_loss: 0.3464 | g_loss: 3.2395
Epoch [    7/   10] | d_loss: 0.4042 | g_loss: 2.1799
Epoch [    7/   10] | d_loss: 0.2735 | g_loss: 2.3588
Epoch [    7/   10] | d_loss: 0.4399 | g_loss: 1.9402
Epoch [    7/   10] | d_loss: 1.0772 | g_loss: 1.1102
Epoch [    7/   10] | d_loss: 0.6479 | g_loss: 2.8637
Epoch [    7/   10] | d_loss: 0.7941 | g_loss: 2.5372
Epoch [    7/   10] | d_loss: 2.1828 | g_loss: 2.4804
Epoch [    7/   10] | d_loss: 0.2446 | g_loss: 3.8056
Epoch [    7/   10] | d_loss: 0.6369 | g_loss: 2.4858
Epoch [    7/   10] | d_loss: 0.3171 | g_loss: 2.5963
Epoch [    7/   10] | d_loss: 0.5061 | g_loss: 1.4902
Epoch [    7/   10] | d_loss: 0.8557 | g_loss: 2.5239
Epoch [    7/   10] | d_loss: 0.1664 | g_loss: 3.1196
Epoch [    7/   10] | d_loss: 0.9782 | g_loss: 4.1785
Epoch [    7/   10] | d_loss: 0.4588 | g_loss: 1.9684
Epoch [    7/   10] | d_loss: 0.2384 | g_loss: 2.0042
Epoch [    7/   10] | d_loss: 0.4823 | g_loss: 0.5093
Epoch [    7/   10] | d_loss: 0.6928 | g_loss: 2.0342
Epoch [    7/   10] | d_loss: 0.6878 | g_loss: 3.7084
Epoch [    7/   10] | d_loss: 0.6175 | g_loss: 2.7601
Epoch [    7/   10] | d_loss: 0.2997 | g_loss: 2.4690
Epoch [    7/   10] | d_loss: 0.8348 | g_loss: 3.4716
Epoch [    7/   10] | d_loss: 0.2267 | g_loss: 2.8243
Epoch [    7/   10] | d_loss: 0.4091 | g_loss: 2.5992
Epoch [    7/   10] | d_loss: 2.0748 | g_loss: 0.9145
Epoch [    7/   10] | d_loss: 0.2330 | g_loss: 2.8362
Epoch [    7/   10] | d_loss: 0.3300 | g_loss: 2.5117
Epoch [    7/   10] | d_loss: 0.7173 | g_loss: 1.9761
Epoch [    7/   10] | d_loss: 0.6068 | g_loss: 2.8502
Epoch [    7/   10] | d_loss: 1.3662 | g_loss: 6.1075
Epoch [    7/   10] | d_loss: 0.3372 | g_loss: 4.0720
Epoch [    7/   10] | d_loss: 0.3592 | g_loss: 3.1426
Epoch [    7/   10] | d_loss: 0.6130 | g_loss: 2.9244
Epoch [    7/   10] | d_loss: 0.2925 | g_loss: 3.1215
Epoch [    7/   10] | d_loss: 0.3754 | g_loss: 2.5370
Epoch [    7/   10] | d_loss: 0.2141 | g_loss: 1.4360
Epoch [    7/   10] | d_loss: 0.2886 | g_loss: 1.9425
Epoch [    7/   10] | d_loss: 0.1611 | g_loss: 3.7099
Epoch [    7/   10] | d_loss: 0.2230 | g_loss: 2.7082
Epoch [    7/   10] | d_loss: 0.5372 | g_loss: 2.5580
Epoch [    7/   10] | d_loss: 0.1387 | g_loss: 1.8393
Epoch [    7/   10] | d_loss: 0.3526 | g_loss: 2.9564
Epoch [    7/   10] | d_loss: 0.5143 | g_loss: 2.1555
Epoch [    7/   10] | d_loss: 0.2883 | g_loss: 2.7579
Epoch [    7/   10] | d_loss: 0.2083 | g_loss: 2.1915
```

```
Epoch [    7/   10] | d_loss: 0.6219 | g_loss: 2.4000
Epoch [    7/   10] | d_loss: 0.4138 | g_loss: 2.6270
Epoch [    7/   10] | d_loss: 0.3771 | g_loss: 2.6508
Epoch [    7/   10] | d_loss: 0.2087 | g_loss: 2.6441
Epoch [    7/   10] | d_loss: 0.5107 | g_loss: 2.0393
Epoch [    7/   10] | d_loss: 0.7435 | g_loss: 1.8431
Epoch [    7/   10] | d_loss: 0.3400 | g_loss: 1.9841
Epoch [    7/   10] | d_loss: 0.3173 | g_loss: 1.8681
Epoch [    8/   10] | d_loss: 0.8716 | g_loss: 1.4906
Epoch [    8/   10] | d_loss: 0.2062 | g_loss: 4.4571
Epoch [    8/   10] | d_loss: 0.2329 | g_loss: 3.1893
Epoch [    8/   10] | d_loss: 0.4233 | g_loss: 2.2640
Epoch [    8/   10] | d_loss: 0.5714 | g_loss: 2.5576
Epoch [    8/   10] | d_loss: 0.4598 | g_loss: 3.1077
Epoch [    8/   10] | d_loss: 0.5772 | g_loss: 1.9069
Epoch [    8/   10] | d_loss: 0.3977 | g_loss: 2.4767
Epoch [    8/   10] | d_loss: 0.2312 | g_loss: 1.3345
Epoch [    8/   10] | d_loss: 0.3522 | g_loss: 1.2442
Epoch [    8/   10] | d_loss: 0.6428 | g_loss: 1.9277
Epoch [    8/   10] | d_loss: 0.1505 | g_loss: 2.5109
Epoch [    8/   10] | d_loss: 0.1605 | g_loss: 3.1802
Epoch [    8/   10] | d_loss: 0.2468 | g_loss: 1.5525
Epoch [    8/   10] | d_loss: 0.3789 | g_loss: 1.4773
Epoch [    8/   10] | d_loss: 1.0131 | g_loss: 2.1597
Epoch [    8/   10] | d_loss: 0.4723 | g_loss: 2.3384
Epoch [    8/   10] | d_loss: 1.0298 | g_loss: 2.3233
Epoch [    8/   10] | d_loss: 0.2910 | g_loss: 1.9266
Epoch [    8/   10] | d_loss: 0.2730 | g_loss: 1.8028
Epoch [    8/   10] | d_loss: 0.2232 | g_loss: 1.9444
Epoch [    8/   10] | d_loss: 0.3846 | g_loss: 1.6435
Epoch [    8/   10] | d_loss: 0.3458 | g_loss: 3.0746
Epoch [    8/   10] | d_loss: 0.8700 | g_loss: 3.2496
Epoch [    8/   10] | d_loss: 0.6327 | g_loss: 3.8832
Epoch [    8/   10] | d_loss: 0.1916 | g_loss: 2.6305
Epoch [    8/   10] | d_loss: 0.1386 | g_loss: 2.2468
Epoch [    8/   10] | d_loss: 0.5334 | g_loss: 2.5571
Epoch [    8/   10] | d_loss: 0.3693 | g_loss: 2.6392
Epoch [    8/   10] | d_loss: 0.4264 | g_loss: 3.4539
Epoch [    8/   10] | d_loss: 0.3284 | g_loss: 3.3063
Epoch [    8/   10] | d_loss: 0.1079 | g_loss: 2.8869
Epoch [    8/   10] | d_loss: 0.4524 | g_loss: 3.1025
Epoch [    8/   10] | d_loss: 0.8720 | g_loss: 2.6006
Epoch [    8/   10] | d_loss: 0.3302 | g_loss: 2.5945
Epoch [    8/   10] | d_loss: 0.3328 | g_loss: 1.7125
Epoch [    8/   10] | d_loss: 1.0585 | g_loss: 2.3189
Epoch [    8/   10] | d_loss: 0.5231 | g_loss: 3.1792
Epoch [    8/   10] | d_loss: 0.4307 | g_loss: 2.1111
Epoch [    8/   10] | d_loss: 0.5102 | g_loss: 1.5004
```

```
Epoch [    8/   10] | d_loss: 0.4774 | g_loss: 1.1809
Epoch [    8/   10] | d_loss: 0.5178 | g_loss: 2.6657
Epoch [    8/   10] | d_loss: 0.5985 | g_loss: 2.9035
Epoch [    8/   10] | d_loss: 0.3139 | g_loss: 4.2923
Epoch [    8/   10] | d_loss: 0.3470 | g_loss: 1.6844
Epoch [    8/   10] | d_loss: 1.3810 | g_loss: 3.0824
Epoch [    8/   10] | d_loss: 0.3328 | g_loss: 1.5249
Epoch [    8/   10] | d_loss: 0.1864 | g_loss: 3.4671
Epoch [    8/   10] | d_loss: 0.3522 | g_loss: 1.8367
Epoch [    8/   10] | d_loss: 0.3712 | g_loss: 3.7199
Epoch [    8/   10] | d_loss: 0.2176 | g_loss: 1.9118
Epoch [    8/   10] | d_loss: 0.8641 | g_loss: 4.6956
Epoch [    8/   10] | d_loss: 0.6543 | g_loss: 3.5769
Epoch [    8/   10] | d_loss: 0.5120 | g_loss: 3.4316
Epoch [    8/   10] | d_loss: 0.4614 | g_loss: 2.6978
Epoch [    8/   10] | d_loss: 0.7263 | g_loss: 0.7381
Epoch [    8/   10] | d_loss: 0.6908 | g_loss: 1.3072
Epoch [    9/   10] | d_loss: 1.0941 | g_loss: 2.5530
Epoch [    9/   10] | d_loss: 0.5088 | g_loss: 2.9595
Epoch [    9/   10] | d_loss: 0.3338 | g_loss: 1.9588
Epoch [    9/   10] | d_loss: 0.3544 | g_loss: 2.3362
Epoch [    9/   10] | d_loss: 1.0926 | g_loss: 1.3908
Epoch [    9/   10] | d_loss: 0.5831 | g_loss: 3.9936
Epoch [    9/   10] | d_loss: 0.8035 | g_loss: 3.5976
Epoch [    9/   10] | d_loss: 0.4309 | g_loss: 4.6451
Epoch [    9/   10] | d_loss: 0.3683 | g_loss: 3.2717
Epoch [    9/   10] | d_loss: 0.7782 | g_loss: 2.4426
Epoch [    9/   10] | d_loss: 0.4564 | g_loss: 3.7228
Epoch [    9/   10] | d_loss: 0.2457 | g_loss: 2.6098
Epoch [    9/   10] | d_loss: 0.5104 | g_loss: 2.8243
Epoch [    9/   10] | d_loss: 0.2724 | g_loss: 4.1845
Epoch [    9/   10] | d_loss: 0.6471 | g_loss: 3.8773
Epoch [    9/   10] | d_loss: 0.6824 | g_loss: 1.4156
Epoch [    9/   10] | d_loss: 2.9624 | g_loss: 3.6636
Epoch [    9/   10] | d_loss: 0.5742 | g_loss: 3.2745
Epoch [    9/   10] | d_loss: 0.3307 | g_loss: 3.2897
Epoch [    9/   10] | d_loss: 0.2040 | g_loss: 2.2757
Epoch [    9/   10] | d_loss: 0.4122 | g_loss: 1.2393
Epoch [    9/   10] | d_loss: 0.3014 | g_loss: 3.0093
Epoch [    9/   10] | d_loss: 1.4728 | g_loss: 3.7480
Epoch [    9/   10] | d_loss: 0.3856 | g_loss: 3.1889
Epoch [    9/   10] | d_loss: 0.6488 | g_loss: 1.3911
Epoch [    9/   10] | d_loss: 0.2044 | g_loss: 2.3627
Epoch [    9/   10] | d_loss: 0.3569 | g_loss: 3.0112
Epoch [    9/   10] | d_loss: 0.1679 | g_loss: 3.4435
Epoch [    9/   10] | d_loss: 0.3497 | g_loss: 0.6440
Epoch [    9/   10] | d_loss: 0.3606 | g_loss: 3.1178
Epoch [    9/   10] | d_loss: 0.2790 | g_loss: 3.6106
```

```
Epoch [    9/   10] | d_loss: 0.3935 | g_loss: 2.8637
Epoch [    9/   10] | d_loss: 0.2496 | g_loss: 1.2956
Epoch [    9/   10] | d_loss: 0.8122 | g_loss: 2.5003
Epoch [    9/   10] | d_loss: 0.0737 | g_loss: 3.0447
Epoch [    9/   10] | d_loss: 0.0865 | g_loss: 4.6079
Epoch [    9/   10] | d_loss: 0.5057 | g_loss: 2.7170
Epoch [    9/   10] | d_loss: 0.3878 | g_loss: 3.0471
Epoch [    9/   10] | d_loss: 0.3277 | g_loss: 2.4433
Epoch [    9/   10] | d_loss: 0.2659 | g_loss: 4.0890
Epoch [    9/   10] | d_loss: 0.3502 | g_loss: 3.1943
Epoch [    9/   10] | d_loss: 0.3435 | g_loss: 3.2248
Epoch [    9/   10] | d_loss: 0.5119 | g_loss: 3.2228
Epoch [    9/   10] | d_loss: 0.1809 | g_loss: 2.1550
Epoch [    9/   10] | d_loss: 0.7454 | g_loss: 3.1918
Epoch [    9/   10] | d_loss: 0.1820 | g_loss: 3.1474
Epoch [    9/   10] | d_loss: 0.4248 | g_loss: 2.2851
Epoch [    9/   10] | d_loss: 0.3551 | g_loss: 1.9690
Epoch [    9/   10] | d_loss: 0.5243 | g_loss: 1.7773
Epoch [    9/   10] | d_loss: 0.2357 | g_loss: 1.5417
Epoch [    9/   10] | d_loss: 0.0984 | g_loss: 3.6464
Epoch [    9/   10] | d_loss: 0.3889 | g_loss: 2.5588
Epoch [    9/   10] | d_loss: 0.3839 | g_loss: 2.6611
Epoch [    9/   10] | d_loss: 0.1540 | g_loss: 2.6244
Epoch [    9/   10] | d_loss: 0.1652 | g_loss: 3.3196
Epoch [    9/   10] | d_loss: 0.8016 | g_loss: 4.7441
Epoch [    9/   10] | d_loss: 0.3005 | g_loss: 2.7775
Epoch [   10/   10] | d_loss: 0.2840 | g_loss: 3.1337
Epoch [   10/   10] | d_loss: 0.6627 | g_loss: 1.6363
Epoch [   10/   10] | d_loss: 0.6339 | g_loss: 2.6743
Epoch [   10/   10] | d_loss: 0.2068 | g_loss: 3.1470
Epoch [   10/   10] | d_loss: 0.2536 | g_loss: 3.4481
Epoch [   10/   10] | d_loss: 1.2718 | g_loss: 1.9911
Epoch [   10/   10] | d_loss: 0.3174 | g_loss: 2.0109
Epoch [   10/   10] | d_loss: 0.1546 | g_loss: 4.6120
Epoch [   10/   10] | d_loss: 0.1782 | g_loss: 3.7523
Epoch [   10/   10] | d_loss: 0.0870 | g_loss: 3.2499
Epoch [   10/   10] | d_loss: 0.3235 | g_loss: 3.9206
Epoch [   10/   10] | d_loss: 0.3407 | g_loss: 2.7918
Epoch [   10/   10] | d_loss: 0.5278 | g_loss: 3.1032
Epoch [   10/   10] | d_loss: 0.6646 | g_loss: 3.4761
Epoch [   10/   10] | d_loss: 0.2268 | g_loss: 3.7802
Epoch [   10/   10] | d_loss: 0.6567 | g_loss: 1.8805
Epoch [   10/   10] | d_loss: 0.6600 | g_loss: 3.0973
Epoch [   10/   10] | d_loss: 0.5449 | g_loss: 3.3718
Epoch [   10/   10] | d_loss: 0.6258 | g_loss: 1.3602
Epoch [   10/   10] | d_loss: 0.3912 | g_loss: 2.3014
Epoch [   10/   10] | d_loss: 0.2591 | g_loss: 3.0892
Epoch [   10/   10] | d_loss: 0.5204 | g_loss: 1.0963
```

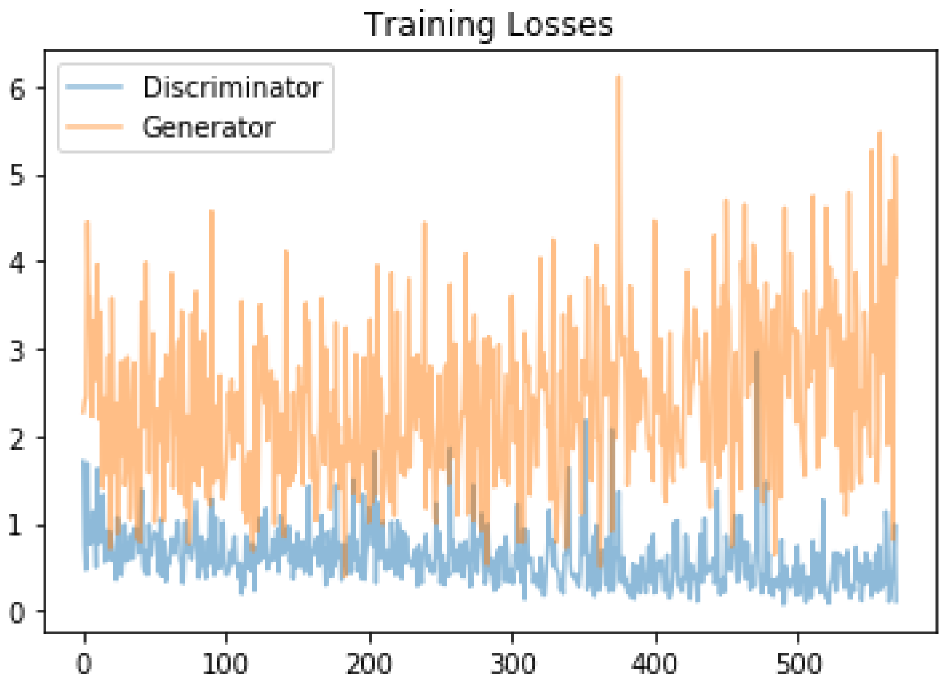
```
Epoch [   10/   10] | d_loss: 0.6874 | g_loss: 1.9489
Epoch [   10/   10] | d_loss: 0.2555 | g_loss: 4.7863
Epoch [   10/   10] | d_loss: 0.1505 | g_loss: 2.3743
Epoch [   10/   10] | d_loss: 0.2431 | g_loss: 1.3807
Epoch [   10/   10] | d_loss: 0.3044 | g_loss: 3.1562
Epoch [   10/   10] | d_loss: 0.2310 | g_loss: 2.3087
Epoch [   10/   10] | d_loss: 0.7297 | g_loss: 3.8725
Epoch [   10/   10] | d_loss: 0.3846 | g_loss: 3.3462
Epoch [   10/   10] | d_loss: 0.3550 | g_loss: 1.5848
Epoch [   10/   10] | d_loss: 0.5399 | g_loss: 1.4596
Epoch [   10/   10] | d_loss: 0.1174 | g_loss: 2.5229
Epoch [   10/   10] | d_loss: 0.3565 | g_loss: 2.1134
Epoch [   10/   10] | d_loss: 0.3059 | g_loss: 3.4175
Epoch [   10/   10] | d_loss: 0.6621 | g_loss: 3.0540
Epoch [   10/   10] | d_loss: 0.3262 | g_loss: 2.2350
Epoch [   10/   10] | d_loss: 0.4244 | g_loss: 2.1956
Epoch [   10/   10] | d_loss: 0.5571 | g_loss: 1.7697
Epoch [   10/   10] | d_loss: 0.1610 | g_loss: 5.2644
Epoch [   10/   10] | d_loss: 0.4329 | g_loss: 3.0819
Epoch [   10/   10] | d_loss: 0.3155 | g_loss: 2.9560
Epoch [   10/   10] | d_loss: 0.6533 | g_loss: 3.4914
Epoch [   10/   10] | d_loss: 0.2191 | g_loss: 1.4764
Epoch [   10/   10] | d_loss: 0.7485 | g_loss: 2.0754
Epoch [   10/   10] | d_loss: 0.2829 | g_loss: 5.4697
Epoch [   10/   10] | d_loss: 0.2469 | g_loss: 2.7154
Epoch [   10/   10] | d_loss: 0.4406 | g_loss: 3.2599
Epoch [   10/   10] | d_loss: 0.3091 | g_loss: 3.9355
Epoch [   10/   10] | d_loss: 1.1334 | g_loss: 3.5250
Epoch [   10/   10] | d_loss: 0.8706 | g_loss: 1.8975
Epoch [   10/   10] | d_loss: 0.1063 | g_loss: 4.1422
Epoch [   10/   10] | d_loss: 0.1594 | g_loss: 4.6899
Epoch [   10/   10] | d_loss: 0.3844 | g_loss: 4.5382
Epoch [   10/   10] | d_loss: 0.5619 | g_loss: 0.8165
Epoch [   10/   10] | d_loss: 0.9877 | g_loss: 5.1969
Epoch [   10/   10] | d_loss: 0.1086 | g_loss: 3.8340
```

2.8 Training loss

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

In [21]:
```
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[21]:
```
<matplotlib.legend.Legend at 0x7fe1d47ab1d0>
```



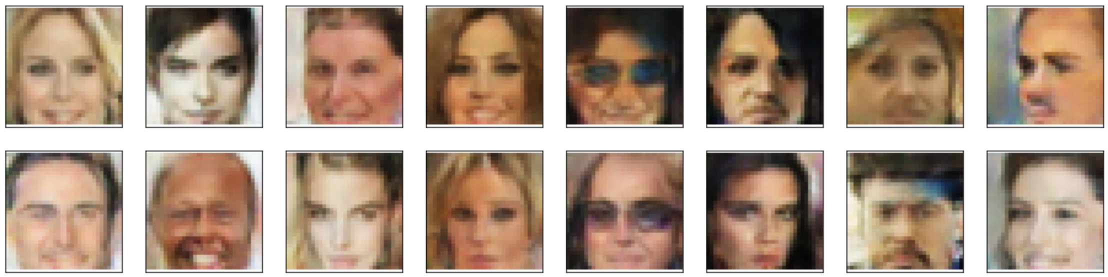
2.9 Generator samples from training

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

```
In [22]: # 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 [23]: # Load samples from generator, taken while training
         with open('train_samples.pkl', 'rb') as f:
             samples = pkl.load(f)
```

```
In [24]: _ = 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: 1. To minimize the biasing the dataset should be more diverse soo that there can be non-white, asians, etc faces needs to be added to the pre-existing dataset 2. As we can see the images generated are not very clear(pixelated, eyes, face boundaries are not clear) and therefore to overcome this problem the model size should be more deep to help it learn more features and the no.of epochs must be increased as the loss is not constant. 3. In this project I have used Adam optimizer as it is works better than SGD in most cases. Talking about the epochs - the no.of epochs must be increased as the loss is not constant yet for 10 epochs.

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.