dlnd_face_generation

March 9, 2019

1 Face Generation

In this project, you'll define and train a DCGAN on a dataset of faces. Your goal is to get a generator network to generate *new* images of faces that look as realistic as possible!

The project will be broken down into a series of tasks from **loading in data to defining and training adversarial networks**. At the end of the notebook, you'll be able to visualize the results of your trained Generator to see how it performs; your generated samples should look like fairly realistic faces with small amounts of noise.

1.0.1 Get the Data

You'll be using the CelebFaces Attributes Dataset (CelebA) to train your adversarial networks.

This dataset is more complex than the number datasets (like MNIST or SVHN) you've been working with, and so, you should prepare to define deeper networks and train them for a longer time to get good results. It is suggested that you utilize a GPU for training.

1.0.2 Pre-processed Data

Since the project's main focus is on building the GANs, we've done *some* of the pre-processing for you. Each of the CelebA images has been cropped to remove parts of the image that don't include a face, then resized down to 64x64x3 NumPy images. Some sample data is show below.

If you are working locally, you can download this data by clicking here

This is a zip file that you'll need to extract in the home directory of this notebook for further loading and processing. After extracting the data, you should be left with a directory of data `processed_celeba_small/`

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

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

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

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

%matplotlib inline
```

1.1 Visualize the CelebA Data

The CelebA dataset contains over 200,000 celebrity images with annotations. Since you're going to be generating faces, you won't need the annotations, you'll only need the images. Note that these are color images with 3 color channels (RGB) each.

1.1.1 Pre-process and Load the Data

Since the project's main focus is on building the GANs, we've done *some* of the pre-processing for you. Each of the CelebA images has been cropped to remove parts of the image that don't include a face, then resized down to 64x64x3 NumPy images. This *pre-processed* dataset is a smaller subset of the very large CelebA data.

> There are a few other steps that you'll need to **transform** this data and create a **DataLoader**.

Exercise: Complete the following `get_dataloader` function, such that it satisfies these requirements:

- Your images should be square, Tensor images of size `image_size x image_size` in the x and y dimension.
- Your function should return a DataLoader that shuffles and batches these Tensor images.

ImageFolder To create a dataset given a directory of images, it's recommended that you use PyTorch's ImageFolder wrapper, with a root directory `processed_celeba_small/` and data transformation passed in.

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

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

            # TODO: Implement function and return a dataloader
```

```
    transformer = transforms.Compose([transforms.Resize(img_size),
                                      transforms.ToTensor()])

    train_dataset = datasets.ImageFolder(root=data_dir, transform = transformer)
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, n

    return train_loader
```

1.2 Create a DataLoader

Exercise: Create a DataLoader `celeba_train_loader` **with appropriate hyperparameters.** Call the above function and create a dataloader to view images. * You can decide on any reasonable `batch_size` parameter * Your `image_size` **must be** `32`. Resizing the data to a smaller size will make for faster training, while still creating convincing images of faces!

In [5]:
```
# Define function hyperparameters
batch_size = 128
img_size = 32

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

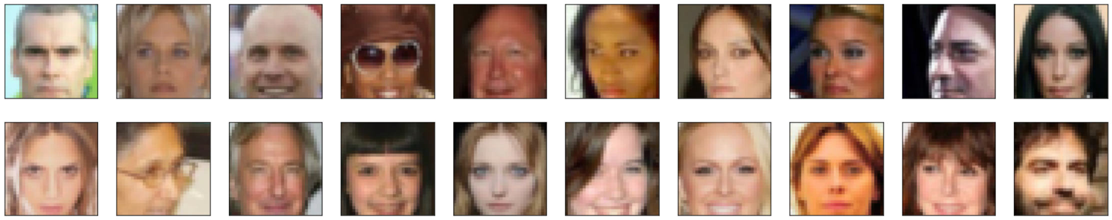
Next, you can view some images! You should seen square images of somewhat-centered faces.

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

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

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

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



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

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

            min, max = feature_range
            x = x * (max - min) + min

            return x
```

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

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

```
Min:  tensor(-0.7569)
Max:  tensor(1.)
```

---

2 Define the Model

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

2.1 Discriminator

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

Exercise: Complete the Discriminator class

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

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

```
In [10]: def conv(in_channels, out_channels, kernel_size = 4, stride = 2, padding = 1, batch_nor
             layers = []
             conv_layer = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, bia
             layers.append(conv_layer)

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

             return nn.Sequential(*layers)
```

```
In [11]: class Discriminator(nn.Module):

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

                 # complete init function
                 self.conv_dim = conv_dim

                 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.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
            x = F.leaky_relu(self.conv1(x), 0.2)
            x = F.leaky_relu(self.conv2(x), 0.2)
            x = F.leaky_relu(self.conv3(x), 0.2)

            x = x.view(x.size(0), -1)
            x = self.fc(x)

            return x


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

```
Tests Passed
```

2.2 Generator

The generator should upsample an input and generate a *new* image of the same size as our training data 32x32x3. This should be mostly transpose convolutional layers with normalization applied to the outputs.

Exercise: Complete the Generator class

- The inputs to the generator are vectors of some length `z_size`
- The output should be a image of shape `32x32x3`

```
In [12]: def deconv(in_channels, out_channels, kernel_size=4, stride=2, padding=1, batch_norm=Tr
             layers = []
             transpose_conv_layer = nn.ConvTranspose2d(in_channels, out_channels,
                                                       kernel_size, stride, padding, bias=False)
             layers.append(transpose_conv_layer)
             if batch_norm:
                 layers.append(nn.BatchNorm2d(out_channels))

             return nn.Sequential(*layers)
```

```
In [13]: class Generator(nn.Module):

             def __init__(self, z_size, conv_dim):
                 """
                 Initialize the Generator Module
                 :param z_size: The length of the input latent vector, z
```

```
        :param conv_dim: The depth of the inputs to the *last* transpose convolutional
        """
        super(Generator, self).__init__()

        # complete init function
        self.conv_dim = conv_dim
        self.fc = nn.Linear(z_size, conv_dim * 4 * 4 * 4)

        self.deconv1 = deconv(conv_dim * 4, conv_dim * 2)
        self.deconv2 = deconv(conv_dim * 2, conv_dim)
        self.deconv3 = deconv(conv_dim, 3, batch_norm = False)

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

        x = F.relu(self.deconv1(x))
        x = F.relu(self.deconv2(x))

        x = self.deconv3(x)
        x = F.tanh(x)

        return x

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

```
Tests Passed
```

2.3 Initialize the weights of your networks

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

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

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

Exercise: Complete the weight initialization function

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

```
In [14]: from torch.nn import init
         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
             classname = m.__class__.__name__
             if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear
                 init.normal_(m.weight.data, 0.0, 0.02)
                 if hasattr(m, 'bias') and m.bias is not None:
                     init.constant_(m.bias.data, 0.0)
             elif classname.find('BatchNorm2d') != -1:  # BatchNorm Layer's weight is not a matr
                 init.normal_(m.weight.data, 1.0, 0.02)
                 init.constant_(m.bias.data, 0.0)
```

2.4 Build complete network

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

```
In [15]: """
         DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
         """
         def build_network(d_conv_dim, g_conv_dim, z_size):
             # define discriminator and generator
             D = Discriminator(d_conv_dim)
             G = Generator(z_size=z_size, conv_dim=g_conv_dim)

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

             print(D)
             print()
```

```
        print(G)

        return D, G
```

Exercise: Define model hyperparameters

```
In [16]: # Define model hyperparams
         d_conv_dim = 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)
  )
  (fc): Linear(in_features=2048, out_features=1, bias=True)
)

Generator(
  (fc): Linear(in_features=100, out_features=2048, bias=True)
  (deconv1): 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)
  )
  (deconv2): 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)
  )
  (deconv3): Sequential(
    (0): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
)
```

2.4.1 Training on GPU

Check if you can train on GPU. Here, we'll set this as a boolean variable `train_on_gpu`. Later, you'll be responsible for making sure that >* Models, * Model inputs, and * Loss function arguments

Are moved to GPU, where appropriate.

```
In [17]: """
         DON'T MODIFY ANYTHING IN THIS CELL
         """
         import torch

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

```
Training on GPU!
```

2.5 Discriminator and Generator Losses

Now we need to calculate the losses for both types of adversarial networks.

2.5.1 Discriminator Losses

- For the discriminator, the total loss is the sum of the losses for real and fake images, `d_loss = d_real_loss + d_fake_loss`.
- Remember that we want the discriminator to output 1 for real images and 0 for fake images, so we need to set up the losses to reflect that.

2.5.2 Generator Loss

The generator loss will look similar only with flipped labels. The generator's goal is to get the discriminator to *think* its generated images are *real*.

Exercise: Complete real and fake loss functions You may choose to use either cross entropy or a least squares error loss to complete the following `real_loss` and `fake_loss` functions.

```
In [18]: def real_loss(D_out):
             '''Calculates how close discriminator outputs are to being real.
                param, D_out: discriminator logits
                return: real loss'''
             batch_size = D_out.size(0)
             labels = torch.ones(batch_size)
```

```
    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 [19]: import torch.optim as optim

         # Create optimizers for the discriminator D and generator G
         lr = 0.0002
         beta1 = 0.5
         beta2 = 0.999

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

2.7 Training

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

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

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

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

```
In [20]: if train_on_gpu:
             D.cuda()
             G.cuda()
```

```
In [21]: def train_discriminator(D, G, real_images):
             D.train()
             G.train()

             d_optimizer.zero_grad()

             if train_on_gpu:
                 real_images = real_images.cuda()

             d_real = D(real_images)
             d_real_loss = real_loss(d_real)

             z = np.random.uniform(-1, 1, size=(batch_size, z_size))
             z = torch.from_numpy(z).float()
             if train_on_gpu:
                 z = z.cuda()

             fake_images = G(z)
             d_fake = D(fake_images)
             d_fake_loss = fake_loss(d_fake)

             total_loss = d_real_loss + d_fake_loss
             total_loss.backward()

             d_optimizer.step()

             return total_loss
```

```
In [22]: def train_generator(D, G):
             D.train()
             G.train()

             g_optimizer.zero_grad()

             z = np.random.uniform(-1, 1, size=(batch_size, z_size))
             z = torch.from_numpy(z).float()
             if train_on_gpu:
                 z = z.cuda()

             fake_images = G(z)
```

```
        g_fake = D(fake_images)
        g_fake_loss = real_loss(g_fake)

        g_fake_loss.backward()
        g_optimizer.step()

        return g_fake_loss
```

In [23]:
```
def train(D, G, n_epochs, print_every=50):
    '''Trains adversarial networks for some number of epochs
       param, D: the discriminator network
       param, G: the generator network
       param, n_epochs: number of epochs to train for
       param, print_every: when to print and record the models' losses
       return: D and G losses'''

    # move models to GPU
    if train_on_gpu:
        D.cuda()
        G.cuda()

    # keep track of loss and generated, "fake" samples
    samples = []
    losses = []

    # Get some fixed data for sampling. These are images that are held
    # constant throughout training, and allow us to inspect the model's performance
    sample_size=16
    fixed_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
    fixed_z = torch.from_numpy(fixed_z).float()
    # move z to GPU if available
    if train_on_gpu:
        fixed_z = fixed_z.cuda()

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

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

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

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

            # 1. Train the discriminator on real and fake images
```

```
                d_loss = train_discriminator(D, G, real_images)

                # 2. Train the generator with an adversarial loss
                g_loss = train_generator(D, G)


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

            if train_on_gpu:
                fixed_z = fixed_z.cuda()

            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 [31]:
```
# set number of epochs
n_epochs = 30


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

```
Epoch [    1/   30] | d_loss: 0.9539 | g_loss: 2.1015
Epoch [    1/   30] | d_loss: 0.8253 | g_loss: 0.9344
Epoch [    1/   30] | d_loss: 0.7897 | g_loss: 1.3852
Epoch [    1/   30] | d_loss: 0.9879 | g_loss: 1.3554
Epoch [    1/   30] | d_loss: 0.8262 | g_loss: 1.2832
Epoch [    1/   30] | d_loss: 0.8207 | g_loss: 1.2280
Epoch [    1/   30] | d_loss: 0.8901 | g_loss: 1.6333
Epoch [    1/   30] | d_loss: 0.8400 | g_loss: 1.3665
Epoch [    1/   30] | d_loss: 1.1498 | g_loss: 0.8591
Epoch [    1/   30] | d_loss: 1.0581 | g_loss: 1.8592
Epoch [    1/   30] | d_loss: 0.8082 | g_loss: 1.1269
Epoch [    1/   30] | d_loss: 0.9794 | g_loss: 1.8357
Epoch [    1/   30] | d_loss: 0.9958 | g_loss: 1.8131
Epoch [    1/   30] | d_loss: 0.9363 | g_loss: 0.8025
Epoch [    1/   30] | d_loss: 1.1680 | g_loss: 1.8990
Epoch [    2/   30] | d_loss: 0.9531 | g_loss: 1.3568
Epoch [    2/   30] | d_loss: 0.8339 | g_loss: 1.1657
Epoch [    2/   30] | d_loss: 1.0140 | g_loss: 1.7741
Epoch [    2/   30] | d_loss: 0.8587 | g_loss: 1.1952
Epoch [    2/   30] | d_loss: 0.7999 | g_loss: 1.2611
Epoch [    2/   30] | d_loss: 0.8146 | g_loss: 1.8760
Epoch [    2/   30] | d_loss: 1.0744 | g_loss: 1.5509
Epoch [    2/   30] | d_loss: 0.7889 | g_loss: 1.4836
Epoch [    2/   30] | d_loss: 1.8496 | g_loss: 1.8012
Epoch [    2/   30] | d_loss: 0.8836 | g_loss: 1.5402
Epoch [    2/   30] | d_loss: 0.8135 | g_loss: 1.3501
Epoch [    2/   30] | d_loss: 0.9675 | g_loss: 1.0970
Epoch [    2/   30] | d_loss: 0.9598 | g_loss: 1.6291
Epoch [    2/   30] | d_loss: 0.9847 | g_loss: 1.2570
Epoch [    2/   30] | d_loss: 0.8693 | g_loss: 1.2131
Epoch [    3/   30] | d_loss: 1.1283 | g_loss: 1.8999
Epoch [    3/   30] | d_loss: 1.0908 | g_loss: 1.8193
Epoch [    3/   30] | d_loss: 1.0144 | g_loss: 1.5633
Epoch [    3/   30] | d_loss: 0.9124 | g_loss: 1.5977
Epoch [    3/   30] | d_loss: 1.0216 | g_loss: 1.1807
Epoch [    3/   30] | d_loss: 1.0376 | g_loss: 0.9406
Epoch [    3/   30] | d_loss: 0.7098 | g_loss: 1.9933
Epoch [    3/   30] | d_loss: 0.9767 | g_loss: 1.5476
Epoch [    3/   30] | d_loss: 1.1211 | g_loss: 0.5954
Epoch [    3/   30] | d_loss: 0.9024 | g_loss: 0.8967
Epoch [    3/   30] | d_loss: 1.1525 | g_loss: 0.4566
Epoch [    3/   30] | d_loss: 0.9252 | g_loss: 0.9467
Epoch [    3/   30] | d_loss: 1.0604 | g_loss: 1.0910
Epoch [    3/   30] | d_loss: 0.7527 | g_loss: 1.7535
Epoch [    3/   30] | d_loss: 0.9464 | g_loss: 1.2578
Epoch [    4/   30] | d_loss: 0.9106 | g_loss: 1.4285
Epoch [    4/   30] | d_loss: 0.9904 | g_loss: 1.3165
Epoch [    4/   30] | d_loss: 0.9158 | g_loss: 1.3130
```

```
Epoch [    4/   30] | d_loss: 2.0946 | g_loss: 0.6695
Epoch [    4/   30] | d_loss: 0.9401 | g_loss: 1.4471
Epoch [    4/   30] | d_loss: 1.0050 | g_loss: 0.8786
Epoch [    4/   30] | d_loss: 0.8965 | g_loss: 1.5851
Epoch [    4/   30] | d_loss: 0.9488 | g_loss: 1.1810
Epoch [    4/   30] | d_loss: 0.9333 | g_loss: 1.6978
Epoch [    4/   30] | d_loss: 0.9003 | g_loss: 1.9538
Epoch [    4/   30] | d_loss: 0.9367 | g_loss: 2.2112
Epoch [    4/   30] | d_loss: 1.0538 | g_loss: 0.9820
Epoch [    4/   30] | d_loss: 0.8131 | g_loss: 1.4575
Epoch [    4/   30] | d_loss: 0.8351 | g_loss: 1.2591
Epoch [    4/   30] | d_loss: 0.7059 | g_loss: 1.4082
Epoch [    5/   30] | d_loss: 0.9192 | g_loss: 0.7394
Epoch [    5/   30] | d_loss: 1.0467 | g_loss: 2.1138
Epoch [    5/   30] | d_loss: 0.8789 | g_loss: 1.8610
Epoch [    5/   30] | d_loss: 0.8729 | g_loss: 1.1797
Epoch [    5/   30] | d_loss: 0.9525 | g_loss: 1.3018
Epoch [    5/   30] | d_loss: 1.1110 | g_loss: 0.8449
Epoch [    5/   30] | d_loss: 1.0399 | g_loss: 1.3719
Epoch [    5/   30] | d_loss: 0.8413 | g_loss: 1.6438
Epoch [    5/   30] | d_loss: 0.8886 | g_loss: 1.2085
Epoch [    5/   30] | d_loss: 1.0295 | g_loss: 1.1605
Epoch [    5/   30] | d_loss: 0.8814 | g_loss: 1.6205
Epoch [    5/   30] | d_loss: 0.8476 | g_loss: 1.7110
Epoch [    5/   30] | d_loss: 1.2138 | g_loss: 0.4135
Epoch [    5/   30] | d_loss: 0.8829 | g_loss: 1.7049
Epoch [    5/   30] | d_loss: 1.0604 | g_loss: 0.7725
Epoch [    6/   30] | d_loss: 0.9769 | g_loss: 0.8527
Epoch [    6/   30] | d_loss: 0.9872 | g_loss: 1.2077
Epoch [    6/   30] | d_loss: 0.9509 | g_loss: 1.5206
Epoch [    6/   30] | d_loss: 1.0222 | g_loss: 1.2526
Epoch [    6/   30] | d_loss: 0.9490 | g_loss: 1.4485
Epoch [    6/   30] | d_loss: 1.0279 | g_loss: 1.4841
Epoch [    6/   30] | d_loss: 1.0443 | g_loss: 1.2029
Epoch [    6/   30] | d_loss: 1.0338 | g_loss: 1.7041
Epoch [    6/   30] | d_loss: 1.1140 | g_loss: 1.6457
Epoch [    6/   30] | d_loss: 1.7293 | g_loss: 3.9016
Epoch [    6/   30] | d_loss: 0.8402 | g_loss: 1.7750
Epoch [    6/   30] | d_loss: 0.9464 | g_loss: 1.3189
Epoch [    6/   30] | d_loss: 0.8153 | g_loss: 1.8534
Epoch [    6/   30] | d_loss: 0.9754 | g_loss: 1.0809
Epoch [    6/   30] | d_loss: 0.9428 | g_loss: 1.2551
Epoch [    7/   30] | d_loss: 0.8246 | g_loss: 1.2614
Epoch [    7/   30] | d_loss: 0.9529 | g_loss: 1.3795
Epoch [    7/   30] | d_loss: 1.0598 | g_loss: 0.8804
Epoch [    7/   30] | d_loss: 1.0005 | g_loss: 1.0491
Epoch [    7/   30] | d_loss: 0.8500 | g_loss: 1.1261
Epoch [    7/   30] | d_loss: 0.8960 | g_loss: 1.2869
```

```
Epoch [    7/   30] | d_loss: 0.9275 | g_loss: 1.4830
Epoch [    7/   30] | d_loss: 0.9719 | g_loss: 1.9509
Epoch [    7/   30] | d_loss: 0.8642 | g_loss: 2.0965
Epoch [    7/   30] | d_loss: 0.9568 | g_loss: 1.1227
Epoch [    7/   30] | d_loss: 0.9245 | g_loss: 0.9960
Epoch [    7/   30] | d_loss: 0.9446 | g_loss: 1.5597
Epoch [    7/   30] | d_loss: 0.8744 | g_loss: 0.9816
Epoch [    7/   30] | d_loss: 1.0140 | g_loss: 1.1982
Epoch [    7/   30] | d_loss: 1.1125 | g_loss: 1.5825
Epoch [    8/   30] | d_loss: 1.0430 | g_loss: 1.4253
Epoch [    8/   30] | d_loss: 0.9002 | g_loss: 1.8377
Epoch [    8/   30] | d_loss: 0.7494 | g_loss: 1.5490
Epoch [    8/   30] | d_loss: 0.8212 | g_loss: 1.4583
Epoch [    8/   30] | d_loss: 1.0053 | g_loss: 1.3060
Epoch [    8/   30] | d_loss: 0.9986 | g_loss: 0.9882
Epoch [    8/   30] | d_loss: 1.1279 | g_loss: 0.7253
Epoch [    8/   30] | d_loss: 0.8825 | g_loss: 1.0137
Epoch [    8/   30] | d_loss: 0.9600 | g_loss: 0.8738
Epoch [    8/   30] | d_loss: 0.9759 | g_loss: 0.9528
Epoch [    8/   30] | d_loss: 0.9671 | g_loss: 0.9239
Epoch [    8/   30] | d_loss: 0.8878 | g_loss: 1.8328
Epoch [    8/   30] | d_loss: 1.0598 | g_loss: 1.4567
Epoch [    8/   30] | d_loss: 0.9752 | g_loss: 1.6600
Epoch [    8/   30] | d_loss: 0.9520 | g_loss: 1.1280
Epoch [    9/   30] | d_loss: 0.9573 | g_loss: 1.4715
Epoch [    9/   30] | d_loss: 1.0004 | g_loss: 1.0746
Epoch [    9/   30] | d_loss: 1.0750 | g_loss: 0.9638
Epoch [    9/   30] | d_loss: 1.0842 | g_loss: 0.7764
Epoch [    9/   30] | d_loss: 0.8474 | g_loss: 1.5885
Epoch [    9/   30] | d_loss: 0.9556 | g_loss: 1.6628
Epoch [    9/   30] | d_loss: 0.9930 | g_loss: 1.5318
Epoch [    9/   30] | d_loss: 1.1099 | g_loss: 0.9567
Epoch [    9/   30] | d_loss: 0.7832 | g_loss: 1.4051
Epoch [    9/   30] | d_loss: 1.0117 | g_loss: 0.7172
Epoch [    9/   30] | d_loss: 0.7918 | g_loss: 1.3868
Epoch [    9/   30] | d_loss: 0.9418 | g_loss: 0.7972
Epoch [    9/   30] | d_loss: 0.8983 | g_loss: 1.8124
Epoch [    9/   30] | d_loss: 1.0305 | g_loss: 0.8261
Epoch [    9/   30] | d_loss: 1.1390 | g_loss: 2.5951
Epoch [   10/   30] | d_loss: 1.0445 | g_loss: 2.4838
Epoch [   10/   30] | d_loss: 0.9657 | g_loss: 0.9740
Epoch [   10/   30] | d_loss: 0.9412 | g_loss: 1.1725
Epoch [   10/   30] | d_loss: 0.9643 | g_loss: 1.2174
Epoch [   10/   30] | d_loss: 0.9105 | g_loss: 1.2022
Epoch [   10/   30] | d_loss: 0.8833 | g_loss: 1.0511
Epoch [   10/   30] | d_loss: 0.8727 | g_loss: 1.7644
Epoch [   10/   30] | d_loss: 0.8170 | g_loss: 1.5242
Epoch [   10/   30] | d_loss: 0.9734 | g_loss: 1.0698
```

```
Epoch [   10/   30] | d_loss: 1.1232 | g_loss: 0.6683
Epoch [   10/   30] | d_loss: 1.4119 | g_loss: 0.5632
Epoch [   10/   30] | d_loss: 1.1119 | g_loss: 2.0570
Epoch [   10/   30] | d_loss: 0.6879 | g_loss: 1.7185
Epoch [   10/   30] | d_loss: 0.7621 | g_loss: 1.2193
Epoch [   10/   30] | d_loss: 0.9275 | g_loss: 1.1229
Epoch [   11/   30] | d_loss: 0.9355 | g_loss: 1.3957
Epoch [   11/   30] | d_loss: 1.0182 | g_loss: 1.0376
Epoch [   11/   30] | d_loss: 0.8770 | g_loss: 1.9904
Epoch [   11/   30] | d_loss: 0.8254 | g_loss: 1.7082
Epoch [   11/   30] | d_loss: 1.0714 | g_loss: 1.9779
Epoch [   11/   30] | d_loss: 1.0490 | g_loss: 0.5613
Epoch [   11/   30] | d_loss: 0.9088 | g_loss: 0.9958
Epoch [   11/   30] | d_loss: 1.0301 | g_loss: 2.0042
Epoch [   11/   30] | d_loss: 1.0027 | g_loss: 1.5527
Epoch [   11/   30] | d_loss: 0.8590 | g_loss: 1.6507
Epoch [   11/   30] | d_loss: 0.9610 | g_loss: 1.9664
Epoch [   11/   30] | d_loss: 0.7471 | g_loss: 0.9999
Epoch [   11/   30] | d_loss: 0.8836 | g_loss: 1.2567
Epoch [   11/   30] | d_loss: 1.0980 | g_loss: 1.4988
Epoch [   11/   30] | d_loss: 0.9076 | g_loss: 1.6850
Epoch [   12/   30] | d_loss: 0.8930 | g_loss: 2.1863
Epoch [   12/   30] | d_loss: 0.8165 | g_loss: 1.4062
Epoch [   12/   30] | d_loss: 1.0235 | g_loss: 1.4228
Epoch [   12/   30] | d_loss: 0.7073 | g_loss: 2.5332
Epoch [   12/   30] | d_loss: 0.8823 | g_loss: 1.4247
Epoch [   12/   30] | d_loss: 0.9269 | g_loss: 1.7865
Epoch [   12/   30] | d_loss: 1.2084 | g_loss: 0.7567
Epoch [   12/   30] | d_loss: 0.8401 | g_loss: 1.4239
Epoch [   12/   30] | d_loss: 0.7328 | g_loss: 1.8252
Epoch [   12/   30] | d_loss: 1.3313 | g_loss: 2.0699
Epoch [   12/   30] | d_loss: 0.9600 | g_loss: 0.9757
Epoch [   12/   30] | d_loss: 0.8299 | g_loss: 1.7587
Epoch [   12/   30] | d_loss: 0.9817 | g_loss: 1.1844
Epoch [   12/   30] | d_loss: 0.7147 | g_loss: 1.1693
Epoch [   12/   30] | d_loss: 0.9667 | g_loss: 0.8324
Epoch [   13/   30] | d_loss: 0.8410 | g_loss: 1.8168
Epoch [   13/   30] | d_loss: 0.9037 | g_loss: 1.7368
Epoch [   13/   30] | d_loss: 0.8469 | g_loss: 1.0640
Epoch [   13/   30] | d_loss: 0.9244 | g_loss: 1.2533
Epoch [   13/   30] | d_loss: 0.7585 | g_loss: 0.9491
Epoch [   13/   30] | d_loss: 0.9653 | g_loss: 1.4526
Epoch [   13/   30] | d_loss: 0.7396 | g_loss: 0.9171
Epoch [   13/   30] | d_loss: 0.8475 | g_loss: 2.1268
Epoch [   13/   30] | d_loss: 0.9311 | g_loss: 0.9517
Epoch [   13/   30] | d_loss: 0.9570 | g_loss: 1.8374
Epoch [   13/   30] | d_loss: 1.2707 | g_loss: 2.0883
Epoch [   13/   30] | d_loss: 0.7020 | g_loss: 2.0585
```

```
Epoch [   13/   30] | d_loss: 0.7942 | g_loss: 1.0434
Epoch [   13/   30] | d_loss: 1.0814 | g_loss: 1.7042
Epoch [   13/   30] | d_loss: 0.8405 | g_loss: 1.2040
Epoch [   14/   30] | d_loss: 0.7369 | g_loss: 1.6268
Epoch [   14/   30] | d_loss: 0.9730 | g_loss: 1.0050
Epoch [   14/   30] | d_loss: 0.9393 | g_loss: 1.7210
Epoch [   14/   30] | d_loss: 0.8831 | g_loss: 1.7393
Epoch [   14/   30] | d_loss: 0.9430 | g_loss: 1.3408
Epoch [   14/   30] | d_loss: 0.9776 | g_loss: 1.7534
Epoch [   14/   30] | d_loss: 0.8126 | g_loss: 1.6545
Epoch [   14/   30] | d_loss: 1.0656 | g_loss: 1.9785
Epoch [   14/   30] | d_loss: 0.8181 | g_loss: 1.9103
Epoch [   14/   30] | d_loss: 0.7372 | g_loss: 1.2482
Epoch [   14/   30] | d_loss: 0.8193 | g_loss: 1.0170
Epoch [   14/   30] | d_loss: 1.2548 | g_loss: 1.0523
Epoch [   14/   30] | d_loss: 0.8282 | g_loss: 1.5079
Epoch [   14/   30] | d_loss: 1.0661 | g_loss: 1.0701
Epoch [   14/   30] | d_loss: 0.9380 | g_loss: 1.0469
Epoch [   15/   30] | d_loss: 0.7553 | g_loss: 1.3846
Epoch [   15/   30] | d_loss: 0.8252 | g_loss: 2.0078
Epoch [   15/   30] | d_loss: 0.8559 | g_loss: 1.3390
Epoch [   15/   30] | d_loss: 2.9420 | g_loss: 0.6752
Epoch [   15/   30] | d_loss: 0.4112 | g_loss: 2.0153
Epoch [   15/   30] | d_loss: 0.8470 | g_loss: 0.8915
Epoch [   15/   30] | d_loss: 1.0143 | g_loss: 1.3998
Epoch [   15/   30] | d_loss: 0.8999 | g_loss: 1.2111
Epoch [   15/   30] | d_loss: 0.8036 | g_loss: 1.3478
Epoch [   15/   30] | d_loss: 0.7390 | g_loss: 2.2127
Epoch [   15/   30] | d_loss: 0.7329 | g_loss: 1.0781
Epoch [   15/   30] | d_loss: 0.8222 | g_loss: 1.3319
Epoch [   15/   30] | d_loss: 0.9515 | g_loss: 1.0010
Epoch [   15/   30] | d_loss: 1.1735 | g_loss: 1.3292
Epoch [   15/   30] | d_loss: 0.8312 | g_loss: 1.6143
Epoch [   16/   30] | d_loss: 0.8406 | g_loss: 1.5182
Epoch [   16/   30] | d_loss: 0.9843 | g_loss: 1.1822
Epoch [   16/   30] | d_loss: 0.8924 | g_loss: 1.4062
Epoch [   16/   30] | d_loss: 1.1213 | g_loss: 3.0522
Epoch [   16/   30] | d_loss: 0.7394 | g_loss: 1.5748
Epoch [   16/   30] | d_loss: 0.6981 | g_loss: 1.5899
Epoch [   16/   30] | d_loss: 1.8474 | g_loss: 0.9318
Epoch [   16/   30] | d_loss: 0.8705 | g_loss: 1.4333
Epoch [   16/   30] | d_loss: 0.8593 | g_loss: 1.1292
Epoch [   16/   30] | d_loss: 0.7638 | g_loss: 1.0927
Epoch [   16/   30] | d_loss: 0.7914 | g_loss: 1.6462
Epoch [   16/   30] | d_loss: 0.8485 | g_loss: 1.3579
Epoch [   16/   30] | d_loss: 0.7798 | g_loss: 1.4270
Epoch [   16/   30] | d_loss: 0.8630 | g_loss: 1.5856
Epoch [   16/   30] | d_loss: 1.2502 | g_loss: 2.8124
```

```
Epoch [   17/   30] | d_loss: 1.0557 | g_loss: 0.6708
Epoch [   17/   30] | d_loss: 0.7527 | g_loss: 1.9640
Epoch [   17/   30] | d_loss: 0.7807 | g_loss: 0.9675
Epoch [   17/   30] | d_loss: 0.7390 | g_loss: 2.0912
Epoch [   17/   30] | d_loss: 0.8188 | g_loss: 1.9344
Epoch [   17/   30] | d_loss: 0.7272 | g_loss: 1.7537
Epoch [   17/   30] | d_loss: 0.6949 | g_loss: 1.9098
Epoch [   17/   30] | d_loss: 0.8816 | g_loss: 1.8464
Epoch [   17/   30] | d_loss: 1.0938 | g_loss: 2.2361
Epoch [   17/   30] | d_loss: 0.8781 | g_loss: 1.8971
Epoch [   17/   30] | d_loss: 0.7953 | g_loss: 1.3627
Epoch [   17/   30] | d_loss: 0.9071 | g_loss: 1.6730
Epoch [   17/   30] | d_loss: 0.7925 | g_loss: 2.9614
Epoch [   17/   30] | d_loss: 0.6915 | g_loss: 1.2970
Epoch [   17/   30] | d_loss: 0.8090 | g_loss: 1.4910
Epoch [   18/   30] | d_loss: 0.7438 | g_loss: 1.8823
Epoch [   18/   30] | d_loss: 0.8643 | g_loss: 1.5829
Epoch [   18/   30] | d_loss: 0.7834 | g_loss: 1.3649
Epoch [   18/   30] | d_loss: 0.6895 | g_loss: 1.4193
Epoch [   18/   30] | d_loss: 0.9462 | g_loss: 2.0171
Epoch [   18/   30] | d_loss: 0.7717 | g_loss: 1.6900
Epoch [   18/   30] | d_loss: 0.7387 | g_loss: 1.2878
Epoch [   18/   30] | d_loss: 0.8313 | g_loss: 0.8891
Epoch [   18/   30] | d_loss: 1.2229 | g_loss: 1.8707
Epoch [   18/   30] | d_loss: 0.6659 | g_loss: 1.2277
Epoch [   18/   30] | d_loss: 0.8211 | g_loss: 1.4093
Epoch [   18/   30] | d_loss: 0.7914 | g_loss: 2.1453
Epoch [   18/   30] | d_loss: 0.8608 | g_loss: 2.0498
Epoch [   18/   30] | d_loss: 0.5890 | g_loss: 1.7012
Epoch [   18/   30] | d_loss: 0.8169 | g_loss: 1.4163
Epoch [   19/   30] | d_loss: 0.7063 | g_loss: 1.8563
Epoch [   19/   30] | d_loss: 0.7129 | g_loss: 2.0723
Epoch [   19/   30] | d_loss: 0.8424 | g_loss: 2.1281
Epoch [   19/   30] | d_loss: 0.6663 | g_loss: 1.8409
Epoch [   19/   30] | d_loss: 1.6007 | g_loss: 3.2997
Epoch [   19/   30] | d_loss: 1.0814 | g_loss: 0.6421
Epoch [   19/   30] | d_loss: 0.7579 | g_loss: 1.5471
Epoch [   19/   30] | d_loss: 0.8022 | g_loss: 0.7048
Epoch [   19/   30] | d_loss: 0.5899 | g_loss: 1.7553
Epoch [   19/   30] | d_loss: 0.6062 | g_loss: 2.2893
Epoch [   19/   30] | d_loss: 0.6574 | g_loss: 1.3404
Epoch [   19/   30] | d_loss: 1.0153 | g_loss: 2.6779
Epoch [   19/   30] | d_loss: 0.8630 | g_loss: 1.6249
Epoch [   19/   30] | d_loss: 0.7644 | g_loss: 1.5318
Epoch [   19/   30] | d_loss: 0.7404 | g_loss: 2.0497
Epoch [   20/   30] | d_loss: 0.7213 | g_loss: 0.9650
Epoch [   20/   30] | d_loss: 1.0559 | g_loss: 1.8354
Epoch [   20/   30] | d_loss: 0.9406 | g_loss: 1.4476
```

```
Epoch [   20/   30] | d_loss: 1.7674 | g_loss: 0.0374
Epoch [   20/   30] | d_loss: 0.7372 | g_loss: 1.7874
Epoch [   20/   30] | d_loss: 0.5886 | g_loss: 1.9052
Epoch [   20/   30] | d_loss: 0.5832 | g_loss: 1.7839
Epoch [   20/   30] | d_loss: 0.5874 | g_loss: 1.4571
Epoch [   20/   30] | d_loss: 0.7300 | g_loss: 1.6264
Epoch [   20/   30] | d_loss: 1.5411 | g_loss: 1.3122
Epoch [   20/   30] | d_loss: 0.8762 | g_loss: 1.5310
Epoch [   20/   30] | d_loss: 0.9153 | g_loss: 1.2042
Epoch [   20/   30] | d_loss: 0.6980 | g_loss: 1.5167
Epoch [   20/   30] | d_loss: 0.6858 | g_loss: 1.2834
Epoch [   20/   30] | d_loss: 0.7402 | g_loss: 1.8127
Epoch [   21/   30] | d_loss: 0.6266 | g_loss: 1.5343
Epoch [   21/   30] | d_loss: 0.8162 | g_loss: 1.0780
Epoch [   21/   30] | d_loss: 0.6806 | g_loss: 1.1227
Epoch [   21/   30] | d_loss: 0.6919 | g_loss: 1.8676
Epoch [   21/   30] | d_loss: 0.6622 | g_loss: 2.0813
Epoch [   21/   30] | d_loss: 0.6803 | g_loss: 1.6789
Epoch [   21/   30] | d_loss: 0.8222 | g_loss: 2.0063
Epoch [   21/   30] | d_loss: 0.7691 | g_loss: 2.3662
Epoch [   21/   30] | d_loss: 1.0736 | g_loss: 2.3578
Epoch [   21/   30] | d_loss: 0.8069 | g_loss: 1.2992
Epoch [   21/   30] | d_loss: 0.7797 | g_loss: 1.8984
Epoch [   21/   30] | d_loss: 0.7657 | g_loss: 1.3010
Epoch [   21/   30] | d_loss: 0.5662 | g_loss: 1.4457
Epoch [   21/   30] | d_loss: 0.7514 | g_loss: 1.8506
Epoch [   21/   30] | d_loss: 0.7156 | g_loss: 1.3405
Epoch [   22/   30] | d_loss: 0.7838 | g_loss: 1.9617
Epoch [   22/   30] | d_loss: 0.6883 | g_loss: 2.0339
Epoch [   22/   30] | d_loss: 0.7116 | g_loss: 1.8036
Epoch [   22/   30] | d_loss: 0.9768 | g_loss: 2.7243
Epoch [   22/   30] | d_loss: 0.7674 | g_loss: 1.8694
Epoch [   22/   30] | d_loss: 0.6227 | g_loss: 1.6499
Epoch [   22/   30] | d_loss: 0.5891 | g_loss: 1.8881
Epoch [   22/   30] | d_loss: 0.8212 | g_loss: 1.1283
Epoch [   22/   30] | d_loss: 0.6633 | g_loss: 1.4607
Epoch [   22/   30] | d_loss: 0.8767 | g_loss: 0.8640
Epoch [   22/   30] | d_loss: 0.7280 | g_loss: 2.3279
Epoch [   22/   30] | d_loss: 0.5756 | g_loss: 2.3180
Epoch [   22/   30] | d_loss: 0.6319 | g_loss: 1.7426
Epoch [   22/   30] | d_loss: 0.7054 | g_loss: 2.2985
Epoch [   22/   30] | d_loss: 0.9718 | g_loss: 3.0209
Epoch [   23/   30] | d_loss: 0.9919 | g_loss: 1.1062
Epoch [   23/   30] | d_loss: 0.8103 | g_loss: 1.2822
Epoch [   23/   30] | d_loss: 0.6039 | g_loss: 2.2880
Epoch [   23/   30] | d_loss: 0.5361 | g_loss: 1.7779
Epoch [   23/   30] | d_loss: 0.6497 | g_loss: 1.6547
Epoch [   23/   30] | d_loss: 0.7189 | g_loss: 1.1646
```

```
Epoch [   23/   30] | d_loss: 0.6307 | g_loss: 2.3786
Epoch [   23/   30] | d_loss: 0.7841 | g_loss: 1.8822
Epoch [   23/   30] | d_loss: 0.8378 | g_loss: 1.5357
Epoch [   23/   30] | d_loss: 0.5632 | g_loss: 1.7444
Epoch [   23/   30] | d_loss: 0.6106 | g_loss: 1.7374
Epoch [   23/   30] | d_loss: 0.4952 | g_loss: 2.2534
Epoch [   23/   30] | d_loss: 0.6660 | g_loss: 1.1967
Epoch [   23/   30] | d_loss: 0.8554 | g_loss: 0.8388
Epoch [   23/   30] | d_loss: 0.6894 | g_loss: 1.6579
Epoch [   24/   30] | d_loss: 0.8875 | g_loss: 1.4146
Epoch [   24/   30] | d_loss: 0.5640 | g_loss: 1.3934
Epoch [   24/   30] | d_loss: 0.6407 | g_loss: 2.1742
Epoch [   24/   30] | d_loss: 0.7590 | g_loss: 1.9206
Epoch [   24/   30] | d_loss: 0.5700 | g_loss: 2.1891
Epoch [   24/   30] | d_loss: 0.7891 | g_loss: 2.8970
Epoch [   24/   30] | d_loss: 0.7428 | g_loss: 1.9615
Epoch [   24/   30] | d_loss: 1.9808 | g_loss: 5.0409
Epoch [   24/   30] | d_loss: 0.8951 | g_loss: 1.1036
Epoch [   24/   30] | d_loss: 0.6044 | g_loss: 2.0187
Epoch [   24/   30] | d_loss: 0.5479 | g_loss: 2.0152
Epoch [   24/   30] | d_loss: 0.6989 | g_loss: 2.0119
Epoch [   24/   30] | d_loss: 0.5884 | g_loss: 1.8692
Epoch [   24/   30] | d_loss: 2.8491 | g_loss: 0.2410
Epoch [   24/   30] | d_loss: 0.8735 | g_loss: 1.4435
Epoch [   25/   30] | d_loss: 0.7992 | g_loss: 2.0640
Epoch [   25/   30] | d_loss: 0.6869 | g_loss: 1.2229
Epoch [   25/   30] | d_loss: 1.2111 | g_loss: 2.7793
Epoch [   25/   30] | d_loss: 0.7446 | g_loss: 1.3755
Epoch [   25/   30] | d_loss: 0.7249 | g_loss: 1.4347
Epoch [   25/   30] | d_loss: 0.6013 | g_loss: 1.6394
Epoch [   25/   30] | d_loss: 0.5738 | g_loss: 1.9819
Epoch [   25/   30] | d_loss: 0.9385 | g_loss: 1.0277
Epoch [   25/   30] | d_loss: 0.8581 | g_loss: 1.3279
Epoch [   25/   30] | d_loss: 0.5940 | g_loss: 1.6205
Epoch [   25/   30] | d_loss: 0.7972 | g_loss: 1.4073
Epoch [   25/   30] | d_loss: 0.5287 | g_loss: 1.8171
Epoch [   25/   30] | d_loss: 0.7361 | g_loss: 2.0219
Epoch [   25/   30] | d_loss: 0.5793 | g_loss: 1.7681
Epoch [   25/   30] | d_loss: 0.7788 | g_loss: 1.3458
Epoch [   26/   30] | d_loss: 0.6172 | g_loss: 1.5501
Epoch [   26/   30] | d_loss: 0.5086 | g_loss: 1.4956
Epoch [   26/   30] | d_loss: 0.8458 | g_loss: 2.2607
Epoch [   26/   30] | d_loss: 0.5107 | g_loss: 1.6213
Epoch [   26/   30] | d_loss: 0.6354 | g_loss: 1.7876
Epoch [   26/   30] | d_loss: 0.5740 | g_loss: 1.7442
Epoch [   26/   30] | d_loss: 0.6987 | g_loss: 2.1913
Epoch [   26/   30] | d_loss: 0.6322 | g_loss: 1.8199
Epoch [   26/   30] | d_loss: 0.5900 | g_loss: 2.3430
```

```
Epoch [   26/   30] | d_loss: 0.5922 | g_loss: 1.4733
Epoch [   26/   30] | d_loss: 0.6432 | g_loss: 2.3014
Epoch [   26/   30] | d_loss: 0.5960 | g_loss: 1.8186
Epoch [   26/   30] | d_loss: 0.8216 | g_loss: 2.3206
Epoch [   26/   30] | d_loss: 0.8134 | g_loss: 1.2762
Epoch [   26/   30] | d_loss: 2.6706 | g_loss: 4.9090
Epoch [   27/   30] | d_loss: 0.6675 | g_loss: 1.9890
Epoch [   27/   30] | d_loss: 0.6704 | g_loss: 2.3692
Epoch [   27/   30] | d_loss: 0.8518 | g_loss: 2.7118
Epoch [   27/   30] | d_loss: 0.5528 | g_loss: 1.9336
Epoch [   27/   30] | d_loss: 0.8959 | g_loss: 2.4804
Epoch [   27/   30] | d_loss: 0.7889 | g_loss: 1.3731
Epoch [   27/   30] | d_loss: 0.4805 | g_loss: 2.3434
Epoch [   27/   30] | d_loss: 0.8464 | g_loss: 1.6430
Epoch [   27/   30] | d_loss: 0.6452 | g_loss: 1.5997
Epoch [   27/   30] | d_loss: 1.0544 | g_loss: 3.2579
Epoch [   27/   30] | d_loss: 0.7110 | g_loss: 1.2464
Epoch [   27/   30] | d_loss: 0.4806 | g_loss: 1.4991
Epoch [   27/   30] | d_loss: 0.6859 | g_loss: 2.2045
Epoch [   27/   30] | d_loss: 0.5573 | g_loss: 2.0115
Epoch [   27/   30] | d_loss: 0.7707 | g_loss: 1.4783
Epoch [   28/   30] | d_loss: 0.7285 | g_loss: 1.5722
Epoch [   28/   30] | d_loss: 0.7680 | g_loss: 2.2741
Epoch [   28/   30] | d_loss: 0.6853 | g_loss: 1.9849
Epoch [   28/   30] | d_loss: 0.6786 | g_loss: 1.5908
Epoch [   28/   30] | d_loss: 0.4878 | g_loss: 1.7861
Epoch [   28/   30] | d_loss: 0.6125 | g_loss: 1.4454
Epoch [   28/   30] | d_loss: 0.9369 | g_loss: 2.8347
Epoch [   28/   30] | d_loss: 0.6840 | g_loss: 2.1837
Epoch [   28/   30] | d_loss: 0.6544 | g_loss: 2.6398
Epoch [   28/   30] | d_loss: 0.4905 | g_loss: 1.8847
Epoch [   28/   30] | d_loss: 3.3249 | g_loss: 0.2100
Epoch [   28/   30] | d_loss: 0.7102 | g_loss: 1.7645
Epoch [   28/   30] | d_loss: 0.5382 | g_loss: 2.1360
Epoch [   28/   30] | d_loss: 0.4223 | g_loss: 2.0789
Epoch [   28/   30] | d_loss: 0.4368 | g_loss: 1.1750
Epoch [   29/   30] | d_loss: 0.6060 | g_loss: 1.7995
Epoch [   29/   30] | d_loss: 0.4594 | g_loss: 2.1064
Epoch [   29/   30] | d_loss: 0.7274 | g_loss: 3.3692
Epoch [   29/   30] | d_loss: 1.0243 | g_loss: 3.4545
Epoch [   29/   30] | d_loss: 0.5373 | g_loss: 1.9644
Epoch [   29/   30] | d_loss: 0.5715 | g_loss: 2.1166
Epoch [   29/   30] | d_loss: 0.5822 | g_loss: 1.6182
Epoch [   29/   30] | d_loss: 0.5268 | g_loss: 2.3610
Epoch [   29/   30] | d_loss: 1.1202 | g_loss: 3.6161
Epoch [   29/   30] | d_loss: 0.5686 | g_loss: 1.9527
Epoch [   29/   30] | d_loss: 0.7320 | g_loss: 3.3481
Epoch [   29/   30] | d_loss: 0.5598 | g_loss: 2.2844
```

```
Epoch [   29/   30] | d_loss: 0.7082 | g_loss: 1.9849
Epoch [   29/   30] | d_loss: 0.5180 | g_loss: 2.1528
Epoch [   29/   30] | d_loss: 0.4853 | g_loss: 1.8548
Epoch [   30/   30] | d_loss: 0.5332 | g_loss: 1.9538
Epoch [   30/   30] | d_loss: 0.5418 | g_loss: 2.1027
Epoch [   30/   30] | d_loss: 0.3843 | g_loss: 3.1029
Epoch [   30/   30] | d_loss: 0.5253 | g_loss: 2.1302
Epoch [   30/   30] | d_loss: 1.8483 | g_loss: 2.9472
Epoch [   30/   30] | d_loss: 0.5175 | g_loss: 1.9130
Epoch [   30/   30] | d_loss: 0.6354 | g_loss: 1.7899
Epoch [   30/   30] | d_loss: 0.7201 | g_loss: 1.7050
Epoch [   30/   30] | d_loss: 0.5298 | g_loss: 2.6081
Epoch [   30/   30] | d_loss: 0.7582 | g_loss: 2.2224
Epoch [   30/   30] | d_loss: 0.4712 | g_loss: 2.7727
Epoch [   30/   30] | d_loss: 0.5997 | g_loss: 1.7560
Epoch [   30/   30] | d_loss: 0.5672 | g_loss: 2.5585
Epoch [   30/   30] | d_loss: 0.8396 | g_loss: 1.1063
Epoch [   30/   30] | d_loss: 0.5936 | g_loss: 2.1415
```

2.8 Training loss

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

In [32]:
```
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[32]:
```
<matplotlib.legend.Legend at 0x7fd7c23daa20>
```

2.9 Generator samples from training

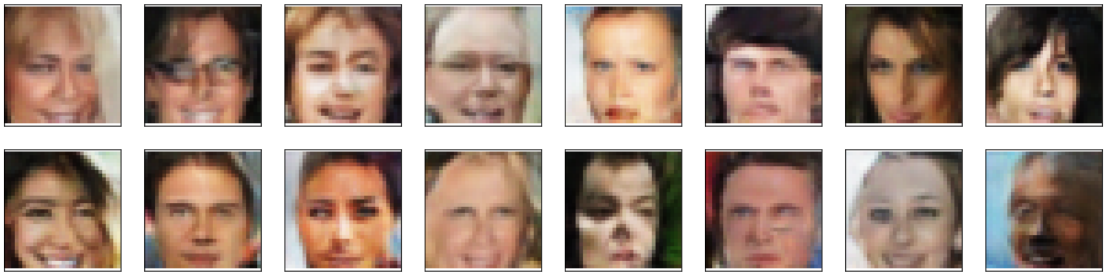
View samples of images from the generator, and answer a question about the strengths and weaknesses of your trained models.

```
In [4]: # 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 [3]: import pickle as pkl
        import matplotlib.pyplot as plt
        import numpy as np

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

```
In [5]: _ = view_samples(-1, samples)
```



2.9.1 Question: What do you notice about your generated samples and how might you improve this model?

When you answer this question, consider the following factors: * The dataset is biased; it is made of "celebrity" faces that are mostly white * Model size; larger models have the opportunity to learn more features in a data feature space * Optimization strategy; optimizers and number of epochs affect your final result

Answer: (Write your answer in this cell)

Obervation

a. The dataset is biased; it is made of "celebrity" faces that are mostly white

   [Ans] Yes. GAN genenerates faces that are mostly white. Because Generator higly biased on training dataset. Definitely, More different type of faces helps to train GAN to made a new type of faces

b. Model size; larger models have the opportunity to learn more features in a data feature space

   [Ans] GAN performs to generate high quality faces well for larger Model. 4 * 4 * 512 model performs better than 4 * 4 * 128. Larger GAN learns more features from input and relatively generates quality images

c. Optimization strategy; optimizers and number of epochs affect your final result

   [Ans] Tried few below hyperparameters and kept optimized values based on recommanded GAN paper. Epochs are critical hyperparameter of GAN. After increasing epochs - 10,20, 30, Generator is getting generated quality images.

   Batch Size - 128 Image Size - 32 * 32 Model - 4 * 4 * 128 GAN Model Epoch - 30 - Tried 10, 20, 50

   Z_size - 100 Weight Initialization - Zero-centered Normal distribution with std 0.02

   Loss Function - BCEWithLogitsLoss Optimizer - Adam Lr = 0.0002 - Tried other option - 0.0001, 0.001 Beta1 = 0.5 - Tried 0.4, 0.3 Beta2 = 0.999

Refernce for hyperparameters
https://arxiv.org/pdf/1511.06434.pdf

Improvement I have analzyed GAN model with few GAN papers and documents, Below stratgies can improves GAN Model

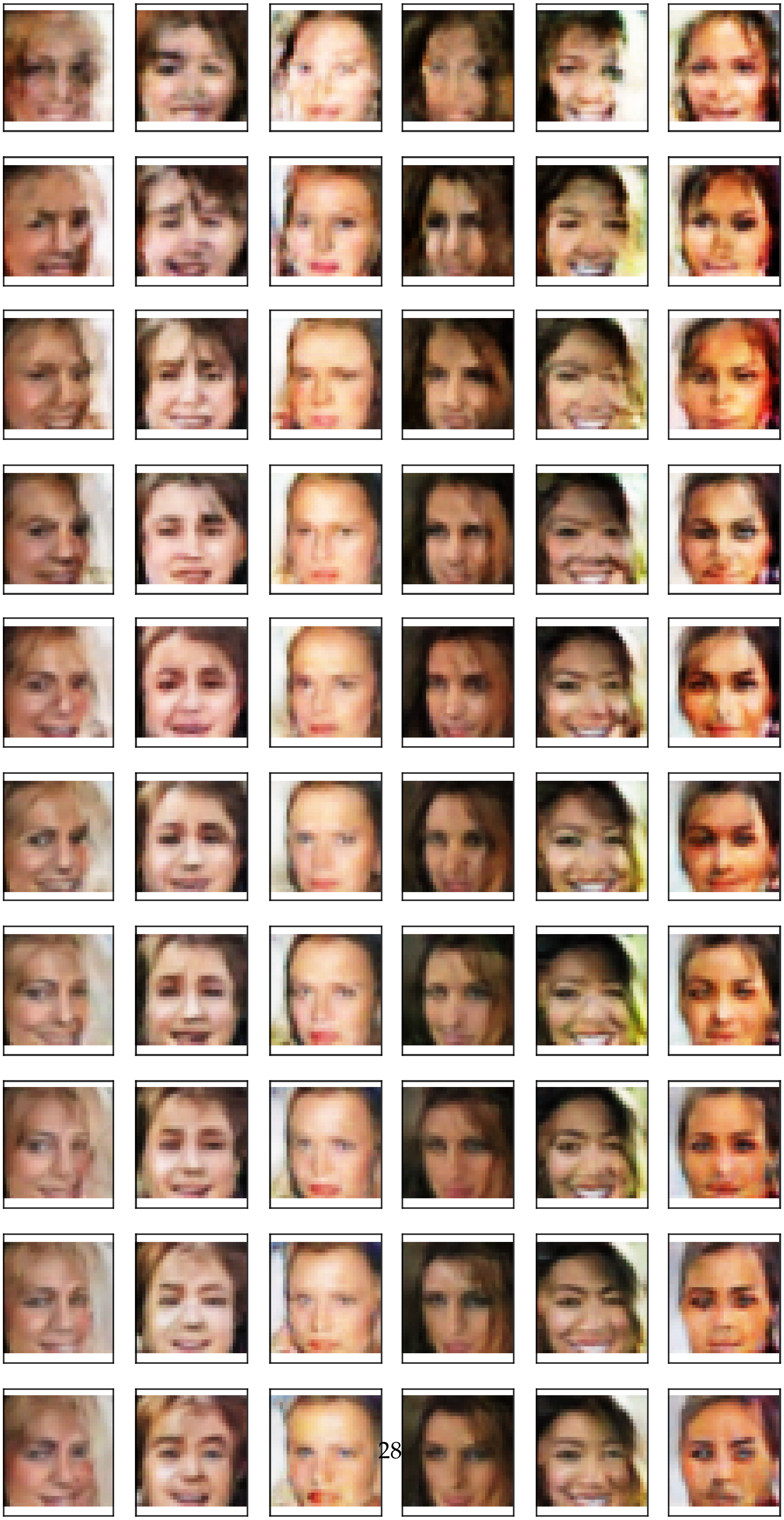
1. Feature Matching
2. Minibatch Discrimination
3. Historical Averaging
4. One-sided Label Smoothing
5. Virtual Batch Normalization
6. Adding Noises
7. Use Better Metric of Distribution Similarity like Wasserstein Distance

Reference https://arxiv.org/pdf/1606.03498.pdf
https://arxiv.org/pdf/1704.00028.pdf
https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b
https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html
https://medium.com/@jonathan_hui/gan-a-comprehensive-review-into-the-gangsters-of-gans-part-2-73233a670d19

In [20]:
```
import cv2

rows = 10 # split epochs into 10, so 100/10 = every 10 epochs
cols = 6
fig, axes = plt.subplots(figsize=(8,16), nrows=rows, ncols=cols, sharex=True, sharey=Tr

for sample, ax_row in zip(samples[::int(len(samples)/rows)], axes):
    for img, ax in zip(sample[::int(len(sample)/cols)], ax_row):
        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)))
```



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.