dlnd_face_generation

July 11, 2021

1 Face Generation

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

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

1.0.1 Get the Data

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

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

1.0.2 Pre-processed Data

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

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

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

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

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

```
In [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
```

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()])
    train_data = datasets.ImageFolder(data_dir, transform=transform)

    return torch.utils.data.DataLoader(train_data, batch_size=batch_size)
```

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 = 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)
```

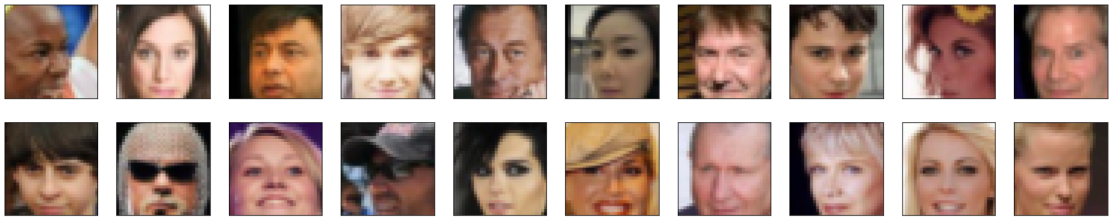
Next, you can view some images! You should seen square images of somewhat-centered faces.

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 [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
    low, high = feature_range
    x = x*(high-low)+low
    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(-0.9529)
Max:  tensor(0.9451)
```

---

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]:
```
class Discriminator(nn.Module):
    #ill be making this architecture:
    #https://raw.githubusercontent.com/udacity/deep-learning-v2-pytorch/b82f18222e46c271
    def __init__(self, conv_dim):
        """
        Initialize the Discriminator Module
        :param conv_dim: The depth of the first convolutional layer
        """
        super(Discriminator, self).__init__()
        self.conv_dim = conv_dim
        # complete init function
        self.conv1 = nn.Conv2d(3, conv_dim, 4, stride=2, padding=1)
        # sees 16x16x..
        self.conv2 = nn.Conv2d(conv_dim, conv_dim*2, 4, stride=2, padding=1)
        # sees 8x8x..
        self.batch1 = nn.BatchNorm2d(conv_dim*2)
        self.conv3 = nn.Conv2d(conv_dim*2, conv_dim*4, 4, stride=2, padding=1)
        self.batch2 = nn.BatchNorm2d(conv_dim*4)
        # sees 4x4x..
        self.fc = nn.Linear(4*4*conv_dim*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 = self.batch1(F.leaky_relu(self.conv2(x), 0.2))
        x = self.batch2(F.leaky_relu(self.conv3(x), 0.2))
        x = x.view(-1, 4*4*self.conv_dim*4)
        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 [10]: class Generator(nn.Module):

             def __init__(self, z_size, conv_dim):
                 #ill be making this network
                 #https://github.com/udacity/deep-learning-v2-pytorch/raw/b82f18222e46c27138fa14
                 """
                 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.z_size = z_size
                 self.conv_dim = conv_dim
                 self.fc = nn.Linear(z_size, 4*4*conv_dim*4)
                 self.conv1 = nn.ConvTranspose2d(conv_dim*4, conv_dim*2, 4, stride=2, padding=1,
                 self.batch1 = nn.BatchNorm2d(conv_dim*2)
                 self.conv2 = nn.ConvTranspose2d(conv_dim*2, conv_dim, 4, stride=2, padding=1, b
                 self.batch2 = nn.BatchNorm2d(conv_dim)
                 self.conv3 = nn.ConvTranspose2d(conv_dim, 3, 4, stride=2, padding=1, bias=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 = self.fc(x)
        x = x.view(-1, self.conv_dim*4, 4, 4) # (batch_size, depth, 4, 4)
        x = self.batch1(F.relu(self.conv1(x)))
        x = self.batch2(F.relu(self.conv2(x)))
        x = F.tanh(self.conv3(x))
        return x

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

```
Tests Passed
```

2.3 Initialize the weights of your networks

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

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

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

Exercise: Complete the weight initialization function

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

```
In [11]: 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
             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)
```

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 [12]: """
         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 [13]: # 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): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (batch1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (batch2): 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)
  (conv1): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fals
  (batch1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
  (batch2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): 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 [14]:
```
"""
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 [15]: def real_loss(D_out):
             '''Calculates how close discriminator outputs are to being real.
                param, D_out: discriminator logits
                return: real loss'''
             labels = torch.ones_like(D_out.squeeze())
             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'''
             labels = torch.zeros_like(D_out.squeeze())
             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 [16]: import torch.optim as optim

         #following https://arxiv.org/pdf/1511.06434.pdf

         # params
         learning_rate = 0.0002
         beta_1 = 0.5
         beta_2 = 0.999

         # Create optimizers for the discriminator D and generator G
         d_optimizer = optim.Adam(D.parameters(), learning_rate, [beta_1, beta_2])
         g_optimizer = optim.Adam(G.parameters(), learning_rate, [beta_1, beta_2])
```

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 [17]: 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_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)
        d_loss = d_real_loss + d_fake_loss
        d_loss.backward()
        d_optimizer.step()

        # 2. Train the generator with an adversarial loss
        g_optimizer.zero_grad()
        z = np.random.uniform(-1, 1, size=(batch_size, z_size))
        z = torch.from_numpy(z).float()
        if train_on_gpu:
            z = z.cuda()
        fake_images = G(z)
        D_fake = D(fake_images)
        g_loss = real_loss(D_fake)
        g_loss.backward()
        g_optimizer.step()


        # ===============================================
        #              END OF YOUR CODE
        # ===============================================

        # Print some loss stats
        if batch_i % print_every == 0:
            # append discriminator loss and generator loss
            losses.append((d_loss.item(), g_loss.item()))
            # print discriminator and generator loss
            print('Epoch [{:5d}/{:5d}] | d_loss: {:6.4f} | g_loss: {:6.4f}'.format(
                    epoch+1, n_epochs, d_loss.item(), g_loss.item()))


    ## AFTER EACH EPOCH##
    # this code assumes your generator is named G, feel free to change the name
    # generate and save sample, fake images
```

```
            G.eval() # for generating samples
            samples_z = G(fixed_z)
            samples.append(samples_z)
            G.train() # back to training mode

        # Save training generator samples
        with open('train_samples.pkl', 'wb') as f:
            pkl.dump(samples, f)

        # finally return losses
        return losses
```

Set your number of training epochs and train your GAN!

```
In [18]: # set number of epochs
         n_epochs = 20


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

```
Epoch [    1/   20] | d_loss: 1.7427 | g_loss: 0.5970
Epoch [    1/   20] | d_loss: 0.0740 | g_loss: 3.6904
Epoch [    1/   20] | d_loss: 0.0449 | g_loss: 4.2517
Epoch [    1/   20] | d_loss: 0.1251 | g_loss: 3.5460
Epoch [    1/   20] | d_loss: 0.5474 | g_loss: 2.5696
Epoch [    1/   20] | d_loss: 0.7517 | g_loss: 1.5036
Epoch [    1/   20] | d_loss: 0.8189 | g_loss: 1.2194
Epoch [    1/   20] | d_loss: 0.7837 | g_loss: 1.9027
Epoch [    1/   20] | d_loss: 0.9164 | g_loss: 1.1374
Epoch [    1/   20] | d_loss: 0.8254 | g_loss: 1.7522
Epoch [    1/   20] | d_loss: 1.0073 | g_loss: 1.3173
Epoch [    1/   20] | d_loss: 1.1840 | g_loss: 2.1046
Epoch [    1/   20] | d_loss: 1.0700 | g_loss: 0.7278
Epoch [    1/   20] | d_loss: 0.9186 | g_loss: 1.5632
Epoch [    1/   20] | d_loss: 0.9053 | g_loss: 1.1757
Epoch [    2/   20] | d_loss: 1.0057 | g_loss: 0.8216
Epoch [    2/   20] | d_loss: 0.9260 | g_loss: 1.2068
Epoch [    2/   20] | d_loss: 1.0479 | g_loss: 1.5992
Epoch [    2/   20] | d_loss: 1.1101 | g_loss: 0.9789
Epoch [    2/   20] | d_loss: 1.1327 | g_loss: 1.0723
Epoch [    2/   20] | d_loss: 1.2674 | g_loss: 0.4947
Epoch [    2/   20] | d_loss: 1.3174 | g_loss: 1.0939
Epoch [    2/   20] | d_loss: 0.7901 | g_loss: 1.2068
Epoch [    2/   20] | d_loss: 1.0003 | g_loss: 1.1316
```

```
Epoch [    2/   20] | d_loss: 1.1252 | g_loss: 2.1196
Epoch [    2/   20] | d_loss: 1.2768 | g_loss: 2.0847
Epoch [    2/   20] | d_loss: 1.2234 | g_loss: 0.9241
Epoch [    2/   20] | d_loss: 1.2168 | g_loss: 0.9574
Epoch [    2/   20] | d_loss: 0.9800 | g_loss: 1.1470
Epoch [    2/   20] | d_loss: 0.9704 | g_loss: 1.0632
Epoch [    3/   20] | d_loss: 1.2513 | g_loss: 1.8542
Epoch [    3/   20] | d_loss: 0.8739 | g_loss: 1.4763
Epoch [    3/   20] | d_loss: 0.9837 | g_loss: 2.2136
Epoch [    3/   20] | d_loss: 1.2459 | g_loss: 1.4557
Epoch [    3/   20] | d_loss: 1.1252 | g_loss: 1.6854
Epoch [    3/   20] | d_loss: 1.0313 | g_loss: 0.8557
Epoch [    3/   20] | d_loss: 1.0825 | g_loss: 1.0836
Epoch [    3/   20] | d_loss: 1.1981 | g_loss: 1.8777
Epoch [    3/   20] | d_loss: 1.0294 | g_loss: 1.4330
Epoch [    3/   20] | d_loss: 1.1590 | g_loss: 1.5991
Epoch [    3/   20] | d_loss: 1.0552 | g_loss: 1.2931
Epoch [    3/   20] | d_loss: 0.9492 | g_loss: 1.1740
Epoch [    3/   20] | d_loss: 0.9858 | g_loss: 1.0084
Epoch [    3/   20] | d_loss: 1.0645 | g_loss: 1.4179
Epoch [    3/   20] | d_loss: 0.9441 | g_loss: 1.0812
Epoch [    4/   20] | d_loss: 1.2621 | g_loss: 1.5526
Epoch [    4/   20] | d_loss: 1.0309 | g_loss: 1.6425
Epoch [    4/   20] | d_loss: 1.1003 | g_loss: 0.8838
Epoch [    4/   20] | d_loss: 1.0158 | g_loss: 1.3945
Epoch [    4/   20] | d_loss: 1.1909 | g_loss: 0.6678
Epoch [    4/   20] | d_loss: 0.9112 | g_loss: 0.9705
Epoch [    4/   20] | d_loss: 1.1023 | g_loss: 1.3225
Epoch [    4/   20] | d_loss: 0.9295 | g_loss: 1.3371
Epoch [    4/   20] | d_loss: 1.3349 | g_loss: 1.2650
Epoch [    4/   20] | d_loss: 1.0429 | g_loss: 0.9364
Epoch [    4/   20] | d_loss: 1.5581 | g_loss: 2.6967
Epoch [    4/   20] | d_loss: 0.8518 | g_loss: 1.4420
Epoch [    4/   20] | d_loss: 1.1658 | g_loss: 0.9626
Epoch [    4/   20] | d_loss: 1.0161 | g_loss: 1.9128
Epoch [    4/   20] | d_loss: 1.2549 | g_loss: 1.6670
Epoch [    5/   20] | d_loss: 1.0224 | g_loss: 0.9585
Epoch [    5/   20] | d_loss: 0.9697 | g_loss: 1.3222
Epoch [    5/   20] | d_loss: 0.9127 | g_loss: 1.6396
Epoch [    5/   20] | d_loss: 0.9980 | g_loss: 0.9969
Epoch [    5/   20] | d_loss: 1.3951 | g_loss: 0.9512
Epoch [    5/   20] | d_loss: 1.1119 | g_loss: 1.2140
Epoch [    5/   20] | d_loss: 1.1610 | g_loss: 0.7973
Epoch [    5/   20] | d_loss: 0.9650 | g_loss: 1.4245
Epoch [    5/   20] | d_loss: 1.3614 | g_loss: 1.5490
Epoch [    5/   20] | d_loss: 0.9901 | g_loss: 1.0379
Epoch [    5/   20] | d_loss: 1.0795 | g_loss: 1.4740
Epoch [    5/   20] | d_loss: 0.9378 | g_loss: 0.9393
```

```
Epoch [    5/   20] | d_loss: 1.0541 | g_loss: 0.9625
Epoch [    5/   20] | d_loss: 1.1215 | g_loss: 1.7283
Epoch [    5/   20] | d_loss: 0.9887 | g_loss: 1.4018
Epoch [    6/   20] | d_loss: 0.8670 | g_loss: 1.1408
Epoch [    6/   20] | d_loss: 1.0011 | g_loss: 1.3362
Epoch [    6/   20] | d_loss: 1.1036 | g_loss: 1.3586
Epoch [    6/   20] | d_loss: 0.8444 | g_loss: 0.6748
Epoch [    6/   20] | d_loss: 1.1064 | g_loss: 0.5956
Epoch [    6/   20] | d_loss: 0.9141 | g_loss: 0.6877
Epoch [    6/   20] | d_loss: 1.1119 | g_loss: 0.9553
Epoch [    6/   20] | d_loss: 1.0663 | g_loss: 1.1732
Epoch [    6/   20] | d_loss: 1.3755 | g_loss: 2.0979
Epoch [    6/   20] | d_loss: 0.8307 | g_loss: 1.6909
Epoch [    6/   20] | d_loss: 0.9496 | g_loss: 1.3523
Epoch [    6/   20] | d_loss: 0.8106 | g_loss: 1.3695
Epoch [    6/   20] | d_loss: 1.0070 | g_loss: 0.8168
Epoch [    6/   20] | d_loss: 0.7614 | g_loss: 1.4107
Epoch [    6/   20] | d_loss: 0.9773 | g_loss: 1.3105
Epoch [    7/   20] | d_loss: 0.8796 | g_loss: 1.7589
Epoch [    7/   20] | d_loss: 0.8442 | g_loss: 1.7696
Epoch [    7/   20] | d_loss: 0.9290 | g_loss: 1.5166
Epoch [    7/   20] | d_loss: 0.7971 | g_loss: 1.2538
Epoch [    7/   20] | d_loss: 1.3050 | g_loss: 0.6180
Epoch [    7/   20] | d_loss: 0.8229 | g_loss: 1.0684
Epoch [    7/   20] | d_loss: 0.8912 | g_loss: 0.9604
Epoch [    7/   20] | d_loss: 0.4972 | g_loss: 1.9471
Epoch [    7/   20] | d_loss: 0.7046 | g_loss: 1.2223
Epoch [    7/   20] | d_loss: 1.0304 | g_loss: 0.9111
Epoch [    7/   20] | d_loss: 0.8530 | g_loss: 1.4095
Epoch [    7/   20] | d_loss: 0.6777 | g_loss: 1.9372
Epoch [    7/   20] | d_loss: 0.8228 | g_loss: 1.6562
Epoch [    7/   20] | d_loss: 0.8129 | g_loss: 2.1938
Epoch [    7/   20] | d_loss: 0.7455 | g_loss: 1.7060
Epoch [    8/   20] | d_loss: 0.8210 | g_loss: 1.1450
Epoch [    8/   20] | d_loss: 0.8405 | g_loss: 1.7259
Epoch [    8/   20] | d_loss: 0.8548 | g_loss: 1.6497
Epoch [    8/   20] | d_loss: 0.6861 | g_loss: 1.2953
Epoch [    8/   20] | d_loss: 1.0859 | g_loss: 0.5681
Epoch [    8/   20] | d_loss: 0.9105 | g_loss: 0.6603
Epoch [    8/   20] | d_loss: 0.7669 | g_loss: 1.0739
Epoch [    8/   20] | d_loss: 0.6335 | g_loss: 1.2467
Epoch [    8/   20] | d_loss: 1.0648 | g_loss: 2.6402
Epoch [    8/   20] | d_loss: 0.6987 | g_loss: 1.5866
Epoch [    8/   20] | d_loss: 0.9201 | g_loss: 1.4020
Epoch [    8/   20] | d_loss: 0.7432 | g_loss: 1.0046
Epoch [    8/   20] | d_loss: 0.5945 | g_loss: 1.4572
Epoch [    8/   20] | d_loss: 0.7987 | g_loss: 2.1379
Epoch [    8/   20] | d_loss: 0.9306 | g_loss: 2.4405
```

```
Epoch [    9/   20] | d_loss: 0.8540 | g_loss: 1.4815
Epoch [    9/   20] | d_loss: 0.8943 | g_loss: 2.1157
Epoch [    9/   20] | d_loss: 0.8691 | g_loss: 1.3261
Epoch [    9/   20] | d_loss: 0.7507 | g_loss: 1.1041
Epoch [    9/   20] | d_loss: 1.8466 | g_loss: 0.4438
Epoch [    9/   20] | d_loss: 0.7156 | g_loss: 1.5514
Epoch [    9/   20] | d_loss: 0.6838 | g_loss: 1.9572
Epoch [    9/   20] | d_loss: 0.4504 | g_loss: 1.9651
Epoch [    9/   20] | d_loss: 0.6725 | g_loss: 0.9480
Epoch [    9/   20] | d_loss: 0.8477 | g_loss: 2.2960
Epoch [    9/   20] | d_loss: 0.7440 | g_loss: 1.1142
Epoch [    9/   20] | d_loss: 0.7324 | g_loss: 2.6842
Epoch [    9/   20] | d_loss: 0.9042 | g_loss: 1.9172
Epoch [    9/   20] | d_loss: 0.8817 | g_loss: 2.0246
Epoch [    9/   20] | d_loss: 0.7863 | g_loss: 1.8130
Epoch [   10/   20] | d_loss: 0.8587 | g_loss: 1.7674
Epoch [   10/   20] | d_loss: 0.9957 | g_loss: 1.8036
Epoch [   10/   20] | d_loss: 0.6021 | g_loss: 1.5647
Epoch [   10/   20] | d_loss: 0.7990 | g_loss: 2.5954
Epoch [   10/   20] | d_loss: 0.9840 | g_loss: 1.2172
Epoch [   10/   20] | d_loss: 0.5301 | g_loss: 1.4668
Epoch [   10/   20] | d_loss: 0.6544 | g_loss: 1.2098
Epoch [   10/   20] | d_loss: 0.3954 | g_loss: 2.7319
Epoch [   10/   20] | d_loss: 0.6500 | g_loss: 1.3433
Epoch [   10/   20] | d_loss: 0.6584 | g_loss: 2.6642
Epoch [   10/   20] | d_loss: 0.8671 | g_loss: 1.7520
Epoch [   10/   20] | d_loss: 0.5751 | g_loss: 2.2429
Epoch [   10/   20] | d_loss: 0.5664 | g_loss: 2.7798
Epoch [   10/   20] | d_loss: 0.6543 | g_loss: 1.5736
Epoch [   10/   20] | d_loss: 0.7166 | g_loss: 0.9169
Epoch [   11/   20] | d_loss: 1.1040 | g_loss: 2.8111
Epoch [   11/   20] | d_loss: 0.5921 | g_loss: 1.5074
Epoch [   11/   20] | d_loss: 0.5983 | g_loss: 1.9776
Epoch [   11/   20] | d_loss: 0.6688 | g_loss: 1.1112
Epoch [   11/   20] | d_loss: 0.9944 | g_loss: 1.3872
Epoch [   11/   20] | d_loss: 0.5391 | g_loss: 2.3324
Epoch [   11/   20] | d_loss: 0.6676 | g_loss: 2.0671
Epoch [   11/   20] | d_loss: 0.6297 | g_loss: 1.7209
Epoch [   11/   20] | d_loss: 1.2555 | g_loss: 3.3177
Epoch [   11/   20] | d_loss: 0.8165 | g_loss: 3.5583
Epoch [   11/   20] | d_loss: 0.5738 | g_loss: 1.3441
Epoch [   11/   20] | d_loss: 0.5576 | g_loss: 2.4064
Epoch [   11/   20] | d_loss: 0.5270 | g_loss: 1.4145
Epoch [   11/   20] | d_loss: 0.5709 | g_loss: 1.8934
Epoch [   11/   20] | d_loss: 0.9731 | g_loss: 2.9229
Epoch [   12/   20] | d_loss: 0.9454 | g_loss: 1.1146
Epoch [   12/   20] | d_loss: 0.5293 | g_loss: 2.3767
Epoch [   12/   20] | d_loss: 0.4577 | g_loss: 1.6173
```

```
Epoch [   12/   20] | d_loss: 0.5427 | g_loss: 1.4374
Epoch [   12/   20] | d_loss: 0.7758 | g_loss: 0.6743
Epoch [   12/   20] | d_loss: 1.1891 | g_loss: 2.6057
Epoch [   12/   20] | d_loss: 0.4601 | g_loss: 1.6892
Epoch [   12/   20] | d_loss: 0.5554 | g_loss: 1.8517
Epoch [   12/   20] | d_loss: 0.5846 | g_loss: 1.7792
Epoch [   12/   20] | d_loss: 0.6275 | g_loss: 2.0653
Epoch [   12/   20] | d_loss: 0.7726 | g_loss: 1.8965
Epoch [   12/   20] | d_loss: 0.5542 | g_loss: 1.9118
Epoch [   12/   20] | d_loss: 0.5655 | g_loss: 1.7891
Epoch [   12/   20] | d_loss: 0.4890 | g_loss: 1.3625
Epoch [   12/   20] | d_loss: 0.6606 | g_loss: 2.1471
Epoch [   13/   20] | d_loss: 0.5435 | g_loss: 1.7101
Epoch [   13/   20] | d_loss: 0.6530 | g_loss: 2.1914
Epoch [   13/   20] | d_loss: 0.6449 | g_loss: 1.5639
Epoch [   13/   20] | d_loss: 0.6654 | g_loss: 2.6487
Epoch [   13/   20] | d_loss: 1.0104 | g_loss: 1.8117
Epoch [   13/   20] | d_loss: 0.4323 | g_loss: 1.5480
Epoch [   13/   20] | d_loss: 0.4679 | g_loss: 2.0748
Epoch [   13/   20] | d_loss: 0.3135 | g_loss: 1.3723
Epoch [   13/   20] | d_loss: 0.4980 | g_loss: 1.5192
Epoch [   13/   20] | d_loss: 1.1467 | g_loss: 3.8900
Epoch [   13/   20] | d_loss: 0.6293 | g_loss: 1.4828
Epoch [   13/   20] | d_loss: 0.9025 | g_loss: 1.0082
Epoch [   13/   20] | d_loss: 0.4315 | g_loss: 1.7448
Epoch [   13/   20] | d_loss: 0.5079 | g_loss: 1.8569
Epoch [   13/   20] | d_loss: 0.5872 | g_loss: 2.3551
Epoch [   14/   20] | d_loss: 0.7500 | g_loss: 1.7908
Epoch [   14/   20] | d_loss: 0.6992 | g_loss: 2.4870
Epoch [   14/   20] | d_loss: 0.4824 | g_loss: 3.0691
Epoch [   14/   20] | d_loss: 0.6089 | g_loss: 2.8987
Epoch [   14/   20] | d_loss: 0.8675 | g_loss: 0.7803
Epoch [   14/   20] | d_loss: 0.5113 | g_loss: 0.8475
Epoch [   14/   20] | d_loss: 0.4796 | g_loss: 2.1828
Epoch [   14/   20] | d_loss: 0.4331 | g_loss: 2.8302
Epoch [   14/   20] | d_loss: 0.4958 | g_loss: 1.2750
Epoch [   14/   20] | d_loss: 0.4831 | g_loss: 1.6834
Epoch [   14/   20] | d_loss: 0.6406 | g_loss: 1.3663
Epoch [   14/   20] | d_loss: 0.4392 | g_loss: 2.4227
Epoch [   14/   20] | d_loss: 0.4749 | g_loss: 2.5893
Epoch [   14/   20] | d_loss: 0.5727 | g_loss: 2.0081
Epoch [   14/   20] | d_loss: 0.6909 | g_loss: 3.3945
Epoch [   15/   20] | d_loss: 0.4974 | g_loss: 2.5697
Epoch [   15/   20] | d_loss: 0.3827 | g_loss: 2.4618
Epoch [   15/   20] | d_loss: 0.8037 | g_loss: 1.9015
Epoch [   15/   20] | d_loss: 0.3503 | g_loss: 0.9107
Epoch [   15/   20] | d_loss: 0.8069 | g_loss: 1.1518
Epoch [   15/   20] | d_loss: 0.4530 | g_loss: 2.5920
```

```
Epoch [   15/   20] | d_loss: 0.4420 | g_loss: 2.5245
Epoch [   15/   20] | d_loss: 0.3417 | g_loss: 3.1737
Epoch [   15/   20] | d_loss: 0.4439 | g_loss: 1.4100
Epoch [   15/   20] | d_loss: 0.3594 | g_loss: 2.0192
Epoch [   15/   20] | d_loss: 0.7240 | g_loss: 1.6177
Epoch [   15/   20] | d_loss: 0.3615 | g_loss: 2.2966
Epoch [   15/   20] | d_loss: 0.2967 | g_loss: 2.4527
Epoch [   15/   20] | d_loss: 0.3108 | g_loss: 2.2035
Epoch [   15/   20] | d_loss: 0.4708 | g_loss: 2.0773
Epoch [   16/   20] | d_loss: 0.4967 | g_loss: 1.6103
Epoch [   16/   20] | d_loss: 0.4905 | g_loss: 2.4851
Epoch [   16/   20] | d_loss: 0.3964 | g_loss: 2.1665
Epoch [   16/   20] | d_loss: 0.2904 | g_loss: 3.0102
Epoch [   16/   20] | d_loss: 1.1440 | g_loss: 0.9409
Epoch [   16/   20] | d_loss: 0.3308 | g_loss: 2.0421
Epoch [   16/   20] | d_loss: 0.3962 | g_loss: 1.9286
Epoch [   16/   20] | d_loss: 0.3437 | g_loss: 2.8996
Epoch [   16/   20] | d_loss: 0.8635 | g_loss: 3.1621
Epoch [   16/   20] | d_loss: 0.3686 | g_loss: 2.2134
Epoch [   16/   20] | d_loss: 0.4420 | g_loss: 2.1367
Epoch [   16/   20] | d_loss: 0.3622 | g_loss: 2.9702
Epoch [   16/   20] | d_loss: 0.4797 | g_loss: 2.1969
Epoch [   16/   20] | d_loss: 0.2184 | g_loss: 2.0611
Epoch [   16/   20] | d_loss: 0.4389 | g_loss: 3.1045
Epoch [   17/   20] | d_loss: 0.4218 | g_loss: 1.6837
Epoch [   17/   20] | d_loss: 0.4157 | g_loss: 2.5578
Epoch [   17/   20] | d_loss: 0.4740 | g_loss: 2.1637
Epoch [   17/   20] | d_loss: 0.2930 | g_loss: 2.5534
Epoch [   17/   20] | d_loss: 0.9577 | g_loss: 1.4192
Epoch [   17/   20] | d_loss: 0.4440 | g_loss: 2.6027
Epoch [   17/   20] | d_loss: 0.5257 | g_loss: 2.0481
Epoch [   17/   20] | d_loss: 0.2598 | g_loss: 2.7104
Epoch [   17/   20] | d_loss: 0.4821 | g_loss: 1.2132
Epoch [   17/   20] | d_loss: 0.2953 | g_loss: 2.2692
Epoch [   17/   20] | d_loss: 0.4920 | g_loss: 0.8810
Epoch [   17/   20] | d_loss: 0.6228 | g_loss: 2.1119
Epoch [   17/   20] | d_loss: 0.3434 | g_loss: 2.0269
Epoch [   17/   20] | d_loss: 0.2329 | g_loss: 2.5180
Epoch [   17/   20] | d_loss: 0.4720 | g_loss: 2.2222
Epoch [   18/   20] | d_loss: 0.4034 | g_loss: 1.6099
Epoch [   18/   20] | d_loss: 0.2509 | g_loss: 3.0507
Epoch [   18/   20] | d_loss: 0.2563 | g_loss: 3.2339
Epoch [   18/   20] | d_loss: 0.3215 | g_loss: 3.6712
Epoch [   18/   20] | d_loss: 0.6989 | g_loss: 1.2042
Epoch [   18/   20] | d_loss: 0.3903 | g_loss: 3.0697
Epoch [   18/   20] | d_loss: 0.4079 | g_loss: 2.5796
Epoch [   18/   20] | d_loss: 0.3742 | g_loss: 3.0958
Epoch [   18/   20] | d_loss: 0.4286 | g_loss: 1.8928
```

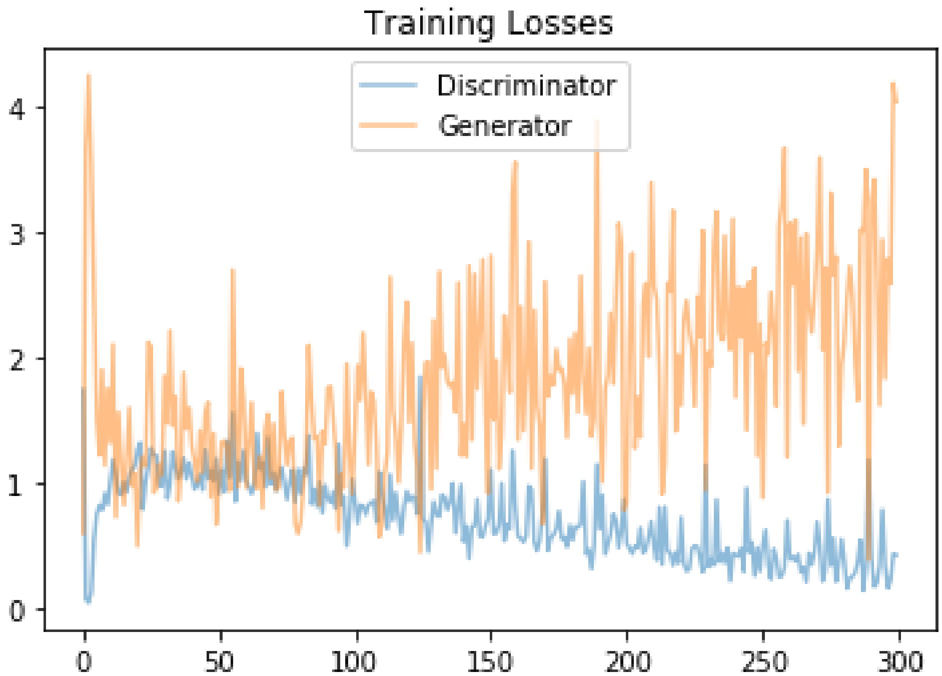
```
Epoch [   18/   20] | d_loss: 0.2898 | g_loss: 2.9517
Epoch [   18/   20] | d_loss: 0.3272 | g_loss: 1.4707
Epoch [   18/   20] | d_loss: 0.2091 | g_loss: 2.9885
Epoch [   18/   20] | d_loss: 0.4379 | g_loss: 2.4095
Epoch [   18/   20] | d_loss: 0.3471 | g_loss: 2.1990
Epoch [   18/   20] | d_loss: 0.4438 | g_loss: 2.4786
Epoch [   19/   20] | d_loss: 0.6878 | g_loss: 2.8637
Epoch [   19/   20] | d_loss: 0.4942 | g_loss: 3.5926
Epoch [   19/   20] | d_loss: 0.2152 | g_loss: 2.0616
Epoch [   19/   20] | d_loss: 0.3006 | g_loss: 2.7155
Epoch [   19/   20] | d_loss: 0.8666 | g_loss: 0.9230
Epoch [   19/   20] | d_loss: 0.3419 | g_loss: 3.3136
Epoch [   19/   20] | d_loss: 0.5593 | g_loss: 2.6553
Epoch [   19/   20] | d_loss: 0.2126 | g_loss: 2.8030
Epoch [   19/   20] | d_loss: 0.3593 | g_loss: 1.2921
Epoch [   19/   20] | d_loss: 0.5482 | g_loss: 1.9387
Epoch [   19/   20] | d_loss: 0.3728 | g_loss: 2.1262
Epoch [   19/   20] | d_loss: 0.1574 | g_loss: 2.4795
Epoch [   19/   20] | d_loss: 0.2342 | g_loss: 2.7301
Epoch [   19/   20] | d_loss: 0.2543 | g_loss: 2.4874
Epoch [   19/   20] | d_loss: 0.2847 | g_loss: 1.9708
Epoch [   20/   20] | d_loss: 0.3749 | g_loss: 1.6547
Epoch [   20/   20] | d_loss: 0.5409 | g_loss: 3.0168
Epoch [   20/   20] | d_loss: 0.1401 | g_loss: 3.0096
Epoch [   20/   20] | d_loss: 0.3265 | g_loss: 3.4995
Epoch [   20/   20] | d_loss: 1.1819 | g_loss: 0.3874
Epoch [   20/   20] | d_loss: 0.4109 | g_loss: 3.2515
Epoch [   20/   20] | d_loss: 0.1746 | g_loss: 3.4196
Epoch [   20/   20] | d_loss: 0.2000 | g_loss: 2.6013
Epoch [   20/   20] | d_loss: 0.3369 | g_loss: 1.6194
Epoch [   20/   20] | d_loss: 0.7840 | g_loss: 2.9401
Epoch [   20/   20] | d_loss: 0.3343 | g_loss: 1.8324
Epoch [   20/   20] | d_loss: 0.1618 | g_loss: 2.7871
Epoch [   20/   20] | d_loss: 0.2252 | g_loss: 2.5882
Epoch [   20/   20] | d_loss: 0.4330 | g_loss: 4.1871
Epoch [   20/   20] | d_loss: 0.4243 | g_loss: 4.0376
```

2.8 Training loss

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

```
In [19]: 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[19]: <matplotlib.legend.Legend at 0x7f43783c55f8>



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 [20]: # 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 [21]: # Load samples from generator, taken while training
         with open('train_samples.pkl', 'rb') as f:
             samples = pkl.load(f)
```

```
In [22]: _ = 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: I noticed that most of the generated images are biased (white faces). The model I made is very simple, I think having more varient data, deeper model, and more number of epochs will improve the result of the model. Furthermore, I noticed that when I train the model the Generator network loss is increasing at some point, while the Descriminator gets better and better, I'm not sure if this is a correct result.

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.