DCGAN_Exercise

May 11, 2020

1 Deep Convolutional GANs

In this notebook, you'll build a GAN using convolutional layers in the generator and discriminator. This is called a Deep Convolutional GAN, or DCGAN for short. The DCGAN architecture was first explored in 2016 and has seen impressive results in generating new images; you can read the original paper, here.

You'll be training DCGAN on the Street View House Numbers (SVHN) dataset. These are color images of house numbers collected from Google street view. SVHN images are in color and much more variable than MNIST.

So, our goal is to create a DCGAN that can generate new, realistic-looking images of house numbers. We'll go through the following steps to do this: * Load in and pre-process the house numbers dataset * Define discriminator and generator networks * Train these adversarial networks * Visualize the loss over time and some sample, generated images

Deeper Convolutional Networks Since this dataset is more complex than our MNIST data, we'll need a deeper network to accurately identify patterns in these images and be able to generate new ones. Specifically, we'll use a series of convolutional or transpose convolutional layers in the discriminator and generator. It's also necessary to use batch normalization to get these convolutional networks to train.

Besides these changes in network structure, training the discriminator and generator networks should be the same as before. That is, the discriminator will alternate training on real and fake (generated) images, and the generator will aim to trick the discriminator into thinking that its generated images are real!

```
In [1]: # import libraries
        import matplotlib.pyplot as plt
        import numpy as np
        import pickle as pkl

        %matplotlib inline
```

1.1 Getting the data

Here you can download the SVHN dataset. It's a dataset built-in to the PyTorch datasets library. We can load in training data, transform it into Tensor datatypes, then create dataloaders to batch our data into a desired size.

```
In [2]: import torch
        from torchvision import datasets
        from torchvision import transforms

        # Tensor transform
        transform = transforms.ToTensor()

        # SVHN training datasets
        svhn_train = datasets.SVHN(root='data/', split='train', download=True, transform=transfo

        batch_size = 128
        num_workers = 0

        # build DataLoaders for SVHN dataset
        train_loader = torch.utils.data.DataLoader(dataset=svhn_train,
                                                   batch_size=batch_size,
                                                   shuffle=True,
                                                   num_workers=num_workers)
```

```
Using downloaded and verified file: data/train_32x32.mat
```

1.1.1 Visualize the Data

Here I'm showing a small sample of the images. Each of these is 32x32 with 3 color channels (RGB). These are the real, training images that we'll pass to the discriminator. Notice that each image has *one* associated, numerical label.

```
In [3]: # obtain one batch of training images
        dataiter = iter(train_loader)
        images, labels = dataiter.next()

        # plot the images in the batch, along with the corresponding labels
        fig = plt.figure(figsize=(25, 4))
        plot_size=20
        for idx in np.arange(plot_size):
            ax = fig.add_subplot(2, plot_size/2, idx+1, xticks=[], yticks=[])
            ax.imshow(np.transpose(images[idx], (1, 2, 0)))
            # print out the correct label for each image
            # .item() gets the value contained in a Tensor
            ax.set_title(str(labels[idx].item()))
```

1.1.2 Pre-processing: scaling from -1 to 1

We need to do a bit of pre-processing; we know that the output of our `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 [4]: # current range
        img = images[0]

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

```
Min:  tensor(1.00000e-02 *
       8.2353)
Max:  tensor(0.3843)
```

```
In [22]: # helper 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 [23]: # scaled range
         scaled_img = scale(img)

         print('Scaled min: ', scaled_img.min())
         print('Scaled max: ', scaled_img.max())
```

```
Scaled min:  tensor(-0.8353)
Scaled max:  tensor(-0.2314)
```

---

2 Define the Model

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

2.1 Discriminator

Here you'll build the discriminator. This is a convolutional classifier like you've built before, only without any maxpooling layers. * The inputs to the discriminator are 32x32x3 tensor images * You'll want a few convolutional, hidden layers * Then a fully connected layer for the output; as before, we want a sigmoid output, but we'll add that in the loss function, BCEWithLogitsLoss, later

For the depths of the convolutional layers I suggest starting with 32 filters in the first layer, then double that depth as you add layers (to 64, 128, etc.). Note that in the DCGAN paper, they did all the downsampling using only strided convolutional layers with no maxpooling layers.

You'll also want to use batch normalization with nn.BatchNorm2d on each layer **except** the first convolutional layer and final, linear output layer.

Helper `conv` function In general, each layer should look something like convolution > batch norm > leaky ReLU, and so we'll define a function to put these layers together. This function will create a sequential series of a convolutional + an optional batch norm layer. We'll create these using PyTorch's Sequential container, which takes in a list of layers and creates layers according to the order that they are passed in to the Sequential constructor.

Note: It is also suggested that you use a **kernel_size of 4** and a **stride of 2** for strided convolutions.

```
In [7]: import torch.nn as nn
        import torch.nn.functional as F

        # create a helper function "conv" that makes use of PyTorch's sequential container

        def conv(in_channels, out_channels, kernel_size=4, stride=2, padding=1, batch_norm=True):
            """Creates a convolutional layer, with optional batch normalization.
            """

            layers = []

            # define conv_layers
            conv = nn.Conv2d(in_channels, out_channels, kernel_size,
                             stride, padding, bias=False)

            # appending conv layers first, in order
            layers.append(conv)

            # define Batch_Norm layers
            if batch_norm == True:
                bn = nn.BatchNorm2d(out_channels)
                layers.append(bn)

            # return layers using the Sequential containter
            return nn.Sequential(*layers)
```

```
In [8]: class Discriminator(nn.Module):
```

```
        def __init__(self, conv_dim=32):
            super(Discriminator, self).__init__()

            self.conv_dim = conv_dim

            # complete init functions, define conv layers
            self.conv1 = conv(3, conv_dim, batch_norm=False)    ## first layer, no batch_nor
            self.conv2 = conv(conv_dim, conv_dim*2)
            self.conv3 = conv(conv_dim*2, conv_dim*4)

            # define fc layer
            self.fc = nn.Linear(conv_dim*4*4*4, 1)

        def forward(self, x):
            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)

            # flatten
            x = x.view(-1, conv_dim*4*4*4)
            output = self.fc(x)

            return output

    conv_dim = 32
    D_DCGANs1 = Discriminator(conv_dim)
    D_DCGANs1
```

Out[8]:
```
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)
)
```

In [9]:
```
## A different approach, using the traditional way to define the Discriminator

import torch.nn as nn
import torch.nn.functional as F
```

```
class Discriminator(nn.Module):

    def __init__(self, conv_dim=32):
        super(Discriminator, self).__init__()

        # complete init function
        # conv layers
        self.conv1 = nn.Conv2d(3, 32, 4, stride=2, padding=1, bias=False)   ### first la
        self.conv2 = nn.Conv2d(32, 64, 4, 2, padding=1, bias=False)
        self.conv3 = nn.Conv2d(64, 128, 4, 2, padding=1, bias=False)

        # batch norm layers
        #self.bn1 = nn.BatchNorm2d(32)
        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)

        # fc
        self.fc = nn.Linear(4*4*128, 1)

        # dropout
        # self.dropout = nn.Dropout(p=0.4)

    def forward(self, x):
        # complete forward function
        x = F.leaky_relu(self.conv1(x), 0.2)      ### Don't forget about the negative sl
        x = F.leaky_relu(self.bn1(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.bn2(self.conv3(x)), 0.2)

        # flatten and feed into the fc layer
        x = x.view(-1, 4*4*128)
        x = self.fc(x)

        return x

D_DCGANs = Discriminator()
D_DCGANs
```

Out[9]:
```
Discriminator(
  (conv1): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (conv2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (conv3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (fc): Linear(in_features=2048, out_features=1, bias=True)
)
```

2.2 Generator

Next, you'll build the generator network. The input will be our noise vector `z`, as before. And, the output will be a *tanh* output, but this time with size 32x32 which is the size of our SVHN images.

What's new here is we'll use transpose convolutional layers to create our new images. * The first layer is a fully connected layer which is reshaped into a deep and narrow layer, something like 4x4x512. * Then, we use batch normalization and a leaky ReLU activation. * Next is a series of transpose convolutional layers, where you typically halve the depth and double the width and height of the previous layer. * And, we'll apply batch normalization and ReLU to all but the last of these hidden layers. Where we will just apply a `tanh` activation.

Helper `deconv` function For each of these layers, the general scheme is transpose convolution > batch norm > ReLU, and so we'll define a function to put these layers together. This function will create a sequential series of a transpose convolutional + an optional batch norm layer. We'll create these using PyTorch's Sequential container, which takes in a list of layers and creates layers according to the order that they are passed in to the Sequential constructor.

Note: It is also suggested that you use a **kernel_size of 4** and a **stride of 2** for transpose convolutions.

```
In [10]: # helper deconv function
         def deconv(in_channels, out_channels, kernel_size=4, stride=2, padding=1, batch_norm=Tr
             """Creates a transposed-convolutional layer, with optional batch normalization.
             """
             ## TODO: Complete this function
             ## create a sequence of transpose + optional batch norm layers

             # initialize layers list
             layers = []

             # define deconv layers
             deconv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
                                         stride, padding, bias=False)
             layers.append(deconv)

             # define batch_norm layers
             if batch_norm:
                 bn = nn.BatchNorm2d(out_channels)
                 layers.append(bn)

             return nn.Sequential(*layers)
```

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

             def __init__(self, z_size, conv_dim=32):
                 super(Generator, self).__init__()

                 self.conv_dim = conv_dim
                 self.z_size = z_size
```

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

        self.tconv1 = deconv(conv_dim*4, conv_dim*2)
        self.tconv2 = deconv(conv_dim*2, conv_dim)
        self.tconv3 = deconv(conv_dim, 3, batch_norm=False)

    def forward(self, x):
        # complete forward function
        x = self.fc(x)
        x = x.view(-1, conv_dim*4, 4, 4)  # shape:(batch_size, depth, width, height)

        x = F.relu(self.tconv1(x))
        x = F.relu(self.tconv2(x))

        output = F.tanh(self.tconv3(x))

        return output
```

2.3 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]: # define hyperparams
         conv_dim = 32
         z_size = 100

         # define discriminator and generator
         D = Discriminator(conv_dim)
         G = Generator(z_size=z_size, conv_dim=conv_dim)

         print(D)
         print()
         print()
         print(G)
```

```
Discriminator(
  (conv1): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (conv2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (conv3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn2): 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)
  (tconv1): 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)
  )
  (tconv2): 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)
  )
  (tconv3): Sequential(
    (0): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
)
```

2.3.1 Training on GPU

Check if you can train on GPU. If you can, set this as a variable and move your models to GPU. > Later, we'll also move any inputs our models and loss functions see (real_images, z, and ground truth labels) to GPU as well.

```
In [13]: train_on_gpu = torch.cuda.is_available()

         if train_on_gpu:
             # move models to GPU
             G.cuda()
             D.cuda()
             print('GPU available for training. Models moved to GPU')
         else:
             print('Training on CPU.')
```

```
GPU available for training. Models moved to GPU
```

2.4 Discriminator and Generator Losses

Now we need to calculate the losses. And this will be exactly the same as before.

2.4.1 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.

The losses will by binary cross entropy loss with logits, which we can get with BCEWithLogitsLoss. This combines a `sigmoid` activation function **and** and binary cross entropy loss in one function.

For the real images, we want `D(real_images) = 1`. That is, we want the discriminator to classify the the real images with a label = 1, indicating that these are real. The discriminator loss for the fake data is similar. We want `D(fake_images) = 0`, where the fake images are the *generator output*, `fake_images = G(z)`.

2.4.2 Generator Loss

The generator loss will look similar only with flipped labels. The generator's goal is to get `D(fake_images) = 1`. In this case, the labels are **flipped** to represent that the generator is trying to fool the discriminator into thinking that the images it generates (fakes) are real!

```
In [14]: def real_loss(D_out, smooth=False):

             batch_size = D_out.size(0)

             # define targets
             if smooth:
                 targets = torch.ones(batch_size)*0.9  ## smooth out real targets, 0.9
             else:
                 targets = torch.ones(batch_size)   ## real targets without smoothing, 1.0

             # moving to GPU if available
             train_on_gpu = torch.cuda.is_available()
             if train_on_gpu:
                 targets = targets.cuda()

             criterion = nn.BCEWithLogitsLoss()
             loss = criterion(D_out.squeeze(), targets)

             return loss

         def fake_loss(D_out):

             batch_size = D_out.size(0)
             targets = torch.zeros(batch_size)

             train_on_gpu = torch.cuda.is_available()
             if train_on_gpu:
                 targets = targets.cuda()

             criterion = nn.BCEWithLogitsLoss()
             loss = criterion(D_out.squeeze(), targets)

             return loss
```

2.5 Optimizers

Not much new here, but notice how I am using a small learning rate and custom parameters for the Adam optimizers, This is based on some research into DCGAN model convergence.

2.5.1 Hyperparameters

GANs are very sensitive to hyperparameters. A lot of experimentation goes into finding the best hyperparameters such that the generator and discriminator don't overpower each other. Try out your own hyperparameters or read the DCGAN paper to see what worked for them.

In [15]:
```
import torch.optim as optim

# params
lr = 0.0002
beta1= 0.5
beta2= 0.1

# Create optimizers for the discriminator and generator
d_optimizer = optim.Adam(D.parameters(), lr, [beta1, beta2])
g_optimizer = optim.Adam(G.parameters(), lr, [beta1, beta2])
```

2.6 Training

Training will involve alternating between training the discriminator and the generator. We'll use our functions `real_loss` and `fake_loss` to help us calculate the discriminator losses in all of the following cases.

2.6.1 Discriminator training

1. Compute the discriminator loss on real, training images

2. Generate fake images
3. Compute the discriminator loss on fake, generated images

4. Add up real and fake loss
5. Perform backpropagation + an optimization step to update the discriminator's weights

2.6.2 Generator training

1. Generate fake images
2. Compute the discriminator loss on fake images, using **flipped** labels!
3. Perform backpropagation + an optimization step to update the generator's weights

Saving Samples As we train, we'll also print out some loss statistics and save some generated "fake" samples.

Evaluation mode

Notice that, when we call our generator to create the samples to display, we set our model to evaluation mode: `G.eval()`. That's so the batch normalization layers will use the population statistics rather than the batch statistics (as they do during training), *and* so dropout layers will operate in eval() mode; not turning off any nodes for generating samples.

```
In [16]: import pickle as pkl

         # training hyperparams
         num_epochs = 10

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

         print_every = 300

         # 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()

         # train the network
         for epoch in range(num_epochs):

             for batch_i, (real_images, _) in enumerate(train_loader):

                 batch_size = real_images.size(0)

                 # important rescaling step
                 real_images = scale(real_images)

                 # ============================================
                 #            TRAIN THE DISCRIMINATOR
                 # ============================================

                 d_optimizer.zero_grad()

                 # 1. Train with real images

                 # Compute the discriminator losses on real images
                 if train_on_gpu:
                     real_images = real_images.cuda()
                 real_images = scale(real_images)
                 D_real = D(real_images)
```

```
d_real_loss = real_loss(D_real)

# 2. Train with fake images

# Generate fake images
z = np.random.uniform(-1, 1, size=(batch_size, z_size))
z = torch.from_numpy(z).float()
# move x to GPU, if available
if train_on_gpu:
    z = z.cuda()
fake_images = G(z)

# Compute the discriminator losses on fake images
D_fake = D(fake_images)
d_fake_loss = fake_loss(D_fake)

# add up loss and perform backprop
d_loss = d_real_loss + d_fake_loss
d_loss.backward()
d_optimizer.step()


# =========================================
#            TRAIN THE GENERATOR
# =========================================
g_optimizer.zero_grad()

# 1. Train with fake images and flipped labels

# Generate fake images
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)

# Compute the discriminator losses on fake images
# using flipped labels!
D_fake = D(fake_images)
g_loss = real_loss(D_fake) # use real loss to flip labels

# perform backprop
g_loss.backward()
g_optimizer.step()

# 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, num_epochs, d_loss.item(), g_loss.item()))


            ## AFTER EACH EPOCH##
            # 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)
```

```
Epoch [    1/   10] | d_loss: 1.4376 | g_loss: 0.9227
Epoch [    1/   10] | d_loss: 0.3895 | g_loss: 3.1244
Epoch [    2/   10] | d_loss: 1.2940 | g_loss: 1.1714
Epoch [    2/   10] | d_loss: 0.7765 | g_loss: 1.1712
Epoch [    3/   10] | d_loss: 1.0162 | g_loss: 1.2236
Epoch [    3/   10] | d_loss: 0.8727 | g_loss: 2.3111
Epoch [    4/   10] | d_loss: 0.9583 | g_loss: 1.1607
Epoch [    4/   10] | d_loss: 1.5968 | g_loss: 0.5267
Epoch [    5/   10] | d_loss: 0.9216 | g_loss: 1.0865
Epoch [    5/   10] | d_loss: 1.3407 | g_loss: 0.7846
Epoch [    6/   10] | d_loss: 0.7632 | g_loss: 1.2868
Epoch [    6/   10] | d_loss: 1.1281 | g_loss: 2.2863
Epoch [    7/   10] | d_loss: 0.7958 | g_loss: 1.6141
Epoch [    7/   10] | d_loss: 1.2145 | g_loss: 1.0058
Epoch [    8/   10] | d_loss: 1.6852 | g_loss: 0.5614
Epoch [    8/   10] | d_loss: 1.1933 | g_loss: 1.1207
Epoch [    9/   10] | d_loss: 0.5435 | g_loss: 0.8994
Epoch [    9/   10] | d_loss: 0.5204 | g_loss: 2.1731
Epoch [   10/   10] | d_loss: 0.8355 | g_loss: 2.1059
Epoch [   10/   10] | d_loss: 0.4389 | g_loss: 1.3256
```

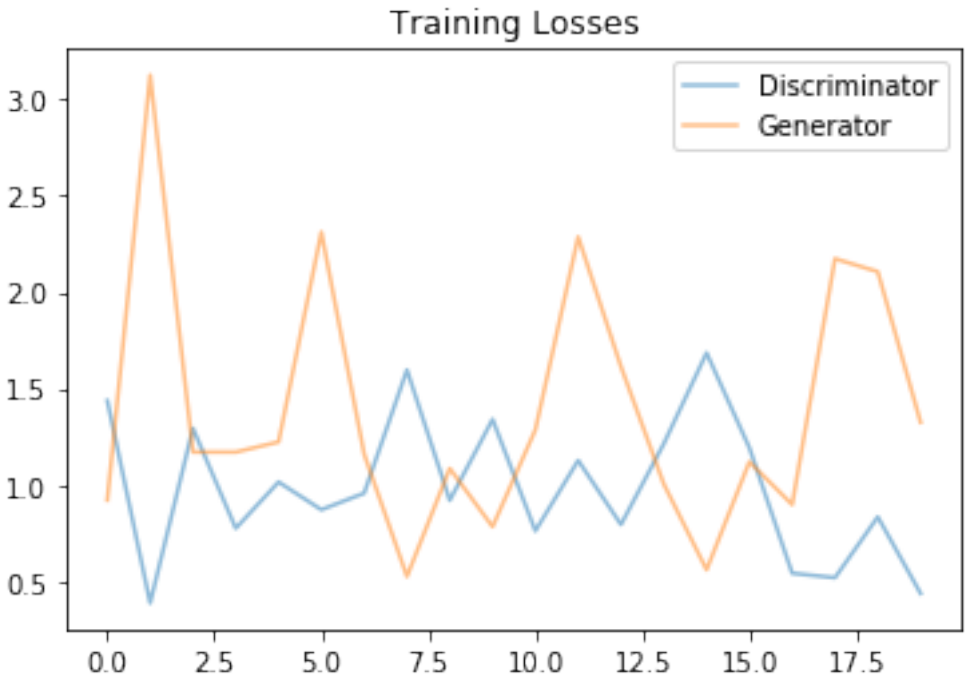
3 Due to time and resource constraint, I have only decided to train for 10 epochs here, but you should try more!

3.1 Training loss

Here we'll plot the training losses for the generator and discriminator, recorded after each epoch.

```
In [17]: 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[17]: <matplotlib.legend.Legend at 0x7f14b03fd8d0>
```



3.2 Generator samples from training

Here we can view samples of images from the generator. We'll look at the images we saved during training.

```
In [18]: # 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) # rescale to pixel range (0-255)
                 ax.xaxis.set_visible(False)
                 ax.yaxis.set_visible(False)
                 im = ax.imshow(img.reshape((32,32,3)))
```

```
In [19]: _ = view_samples(-1, samples)
```

4 If you train for more epochs, I am sure that the images will be more clear

In [ ]: