MNIST_GAN_Exercise

March 2, 2019

1 Generative Adversarial Network

In this notebook, we'll be building a generative adversarial network (GAN) trained on the MNIST dataset. From this, we'll be able to generate new handwritten digits!

GANs were first reported on in 2014 from Ian Goodfellow and others in Yoshua Bengio's lab. Since then, GANs have exploded in popularity. Here are a few examples to check out:

- Pix2Pix
- CycleGAN & Pix2Pix in PyTorch, Jun-Yan Zhu
- A list of generative models

The idea behind GANs is that you have two networks, a generator G and a discriminator D, competing against each other. The generator makes "fake" data to pass to the discriminator. The discriminator also sees real training data and predicts if the data it's received is real or fake. > * The generator is trained to fool the discriminator, it wants to output data that looks *as close as possible* to real, training data. * The discriminator is a classifier that is trained to figure out which data is real and which is fake.

What ends up happening is that the generator learns to make data that is indistinguishable from real data to the discriminator.

The general structure of a GAN is shown in the diagram above, using MNIST images as data. The latent sample is a random vector that the generator uses to construct its fake images. This is often called a **latent vector** and that vector space is called **latent space**. As the generator trains, it figures out how to map latent vectors to recognizable images that can fool the discriminator.

If you're interested in generating only new images, you can throw out the discriminator after training. In this notebook, I'll show you how to define and train these adversarial networks in PyTorch and generate new images!

In [27]:
```
%matplotlib inline

import numpy as np
import torch
import matplotlib.pyplot as plt
```

In [28]:
```
from torchvision import datasets
import torchvision.transforms as transforms

# number of subprocesses to use for data loading
num_workers = 0
```

```
# how many samples per batch to load
batch_size = 64

# convert data to torch.FloatTensor
transform = transforms.ToTensor()

# get the training datasets
train_data = datasets.MNIST(root='data', train=True,
                                   download=True, transform=transform)

# prepare data loader
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers)
```

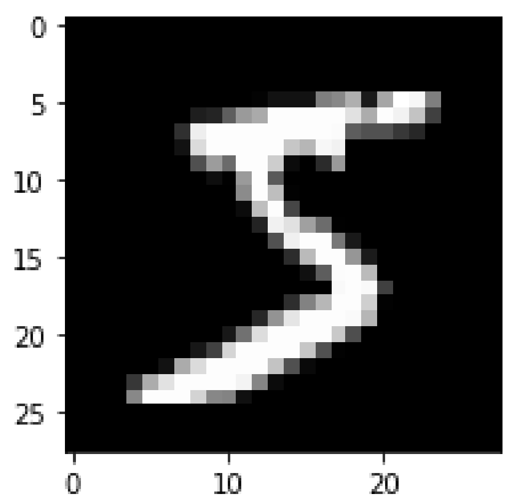
1.0.1 Visualize the data

In [29]:
```
# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy()

# get one image from the batch
img = np.squeeze(images[0])

fig = plt.figure(figsize = (3,3))
ax = fig.add_subplot(111)
ax.imshow(img, cmap='gray')
```

Out[29]: <matplotlib.image.AxesImage at 0x7fa4cafd4a90>



2 Define the Model

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

2.1 Discriminator

The discriminator network is going to be a pretty typical linear classifier. To make this network a universal function approximator, we'll need at least one hidden layer, and these hidden layers should have one key attribute: > All hidden layers will have a Leaky ReLu activation function applied to their outputs.

Leaky ReLu We should use a leaky ReLU to allow gradients to flow backwards through the layer unimpeded. A leaky ReLU is like a normal ReLU, except that there is a small non-zero output for negative input values.

Sigmoid Output We'll also take the approach of using a more numerically stable loss function on the outputs. Recall that we want the discriminator to output a value 0-1 indicating whether an image is *real or fake*. > We will ultimately use BCEWithLogitsLoss, which combines a `sigmoid` activation function **and** and binary cross entropy loss in one function.

So, our final output layer should not have any activation function applied to it.

In [30]:
```
import torch.nn as nn
import torch.nn.functional as F

class Discriminator(nn.Module):

    def __init__(self, input_size, hidden_dim, output_size):
        super(Discriminator, self).__init__()

        self.input_size = input_size

        # define all layers
        self.hidden1 = nn.Linear(input_size, 4*hidden_dim) # dimensions are reduced
        self.hidden2 = nn.Linear(4*hidden_dim, 2*hidden_dim)
        self.hidden3 = nn.Linear(2*hidden_dim, hidden_dim)
        self.output = nn.Linear(hidden_dim, output_size)

        # dropout layer
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        # flatten image
        x = x.view(-1, 28*28)

        # pass x through all layers
        # apply leaky relu activation to all hidden layers
        x = F.leaky_relu(self.hidden1(x), 0.2) # (input, negative_slope=0.2)
        x = self.dropout(x)
```

```
        x = F.leaky_relu(self.hidden2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.hidden3(x), 0.2)
        x = self.dropout(x)
        x = self.output(x) # no activation here (sigmoid will be applied later)

        return x
```

2.2 Generator

The generator network will be almost exactly the same as the discriminator network, except that we're applying a tanh activation function to our output layer.

tanh Output The generator has been found to perform the best with *tanh* for the generator output, which scales the output to be between -1 and 1, instead of 0 and 1.

Recall that we also want these outputs to be comparable to the *real* input pixel values, which are read in as normalized values between 0 and 1. > So, we'll also have to **scale our real input images to have pixel values between -1 and 1** when we train the discriminator.

I'll do this in the training loop, later on.

```
In [31]: class Generator(nn.Module):

             def __init__(self, input_size, hidden_dim, output_size):
                 super(Generator, self).__init__()

                 # define all layers
                 self.hidden1 = nn.Linear(input_size, hidden_dim) # dims are increased (unlike d
                 self.hidden2 = nn.Linear(hidden_dim, 2*hidden_dim)
                 self.hidden3 = nn.Linear(2*hidden_dim, 4*hidden_dim)
                 self.output = nn.Linear(4*hidden_dim, output_size)

                 # dropout layer
                 self.dropout = nn.Dropout(0.3)


             def forward(self, x):
                 # pass x through all layers

                 # final layer should have tanh applied
                 x = F.leaky_relu(self.hidden1(x), 0.2) # (input, negative_slope=0.2)
                 x = self.dropout(x)
                 x = F.leaky_relu(self.hidden2(x), 0.2)
                 x = self.dropout(x)
                 x = F.leaky_relu(self.hidden3(x), 0.2)
                 x = self.dropout(x)
                 x = F.tanh(self.output(x)) # tanh activation: scaling between -1 and 1

                 return x
```

2.3 Model hyperparameters

In [32]:
```
# Discriminator hyperparams

# Size of input image to discriminator (28*28)
input_size = 28*28
# Size of discriminator output (real or fake)
d_output_size = 1
# Size of *last* hidden layer in the discriminator
d_hidden_size = 28

# Generator hyperparams

# Size of latent vector to give to generator
z_size = 100
# Size of discriminator output (generated image)
g_output_size = input_size
# Size of *first* hidden layer in the generator
g_hidden_size = 28
```

2.4 Build complete network

Now we're instantiating the discriminator and generator from the classes defined above. Make sure you've passed in the correct input arguments.

In [33]:
```
# instantiate discriminator and generator
D = Discriminator(input_size, d_hidden_size, d_output_size)
G = Generator(z_size, g_hidden_size, g_output_size)

# check that they are as you expect
print(D)
print()
print(G)
```

```
Discriminator(
  (hidden1): Linear(in_features=784, out_features=112, bias=True)
  (hidden2): Linear(in_features=112, out_features=56, bias=True)
  (hidden3): Linear(in_features=56, out_features=28, bias=True)
  (output): Linear(in_features=28, out_features=1, bias=True)
  (dropout): Dropout(p=0.3)
)

Generator(
  (hidden1): Linear(in_features=100, out_features=28, bias=True)
  (hidden2): Linear(in_features=28, out_features=56, bias=True)
  (hidden3): Linear(in_features=56, out_features=112, bias=True)
  (output): Linear(in_features=112, out_features=784, bias=True)
  (dropout): Dropout(p=0.3)
```

)

2.5 Discriminator and Generator Losses

Now we need to calculate the losses.

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.

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. To help the discriminator generalize better, the labels are **reduced a bit from 1.0 to 0.9**. For this, we'll use the parameter `smooth`; if True, then we should smooth our labels. In PyTorch, this looks like `labels = torch.ones(size) * 0.9`

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)`.

2.5.2 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 [34]: # Calculate losses
         def real_loss(D_out, smooth=False):
             # compare logits to real labels
             # smooth labels if smooth=True
             batch_size = D_out.size(0)
             if(smooth):
                 labels = torch.ones(batch_size) * 0.9
             else:
                 labels = torch.ones(batch_size)

             criterion = nn.BCEWithLogitsLoss()
             # calc loss
             loss = criterion(D_out.squeeze(), labels)

             return loss
```

```
def fake_loss(D_out):
    # compare logits to fake labels
    batch_size = D_out.size(0)
    labels = torch.zeros(batch_size)

    criterion = nn.BCEWithLogitsLoss()
    # calc loss
    loss = criterion(D_out.squeeze(), labels)
    return loss
```

2.6 Optimizers

We want to update the generator and discriminator variables separately. So, we'll define two separate Adam optimizers.

```
In [35]: import torch.optim as optim

         # learning rate for optimizers
         lr = 0.002

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

2.7 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.

2.7.1 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

2.7.2 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.

In [43]:
```
import pickle as pkl

# training hyperparams
num_epochs = 100

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

print_every = 400

# 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
D.train()
G.train()
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 = real_images*2 - 1  # rescale input images from [0,1) to [-1, 1)

        # ============================================
        #            TRAIN THE DISCRIMINATOR
        # ============================================
        d_optimizer.zero_grad()

        # 1. Train with real images

        # Compute the discriminator losses on real images
        # use smoothed labels
        D_out = D.forward(real_images)
        d_real_loss = real_loss(D_out, smooth=True)

        # 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()
        fake_images = G.forward(z)

        # Compute the discriminator losses on fake images
        D_out = D.forward(fake_images)
        d_fake_loss = fake_loss(D_out)

        # add up real and fake losses 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()
        fake_images = G.forward(z)

        # Compute the discriminator losses on fake images
        # using flipped labels!
        D_out = D.forward(fake_images)
        g_loss = real_loss(D_out) # adversarial: real_loss instead of fake_loss

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


        # Print some loss stats
        if batch_i % print_every == 0:
            # 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##
    # append discriminator loss and generator loss
    losses.append((d_loss.item(), g_loss.item()))

    # generate and save sample, fake images
    G.eval() # eval mode for generating samples
    samples_z = G(fixed_z)
```

```python
            samples.append(samples_z)
            G.train() # back to train mode


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

```
Epoch [    1/  100] | d_loss: 1.2913 | g_loss: 0.9683
Epoch [    1/  100] | d_loss: 1.2203 | g_loss: 1.0513
Epoch [    1/  100] | d_loss: 1.3839 | g_loss: 1.0690
Epoch [    2/  100] | d_loss: 1.1739 | g_loss: 1.1843
Epoch [    2/  100] | d_loss: 1.1875 | g_loss: 1.2010
Epoch [    2/  100] | d_loss: 1.2770 | g_loss: 1.0769
Epoch [    3/  100] | d_loss: 1.2870 | g_loss: 1.0315
Epoch [    3/  100] | d_loss: 1.2843 | g_loss: 1.1508
Epoch [    3/  100] | d_loss: 1.3419 | g_loss: 0.9113
Epoch [    4/  100] | d_loss: 1.4190 | g_loss: 0.8892
Epoch [    4/  100] | d_loss: 1.3236 | g_loss: 0.9185
Epoch [    4/  100] | d_loss: 1.2054 | g_loss: 1.0921
Epoch [    5/  100] | d_loss: 1.2896 | g_loss: 0.8734
Epoch [    5/  100] | d_loss: 1.2756 | g_loss: 0.9989
Epoch [    5/  100] | d_loss: 1.3833 | g_loss: 0.9196
Epoch [    6/  100] | d_loss: 1.2616 | g_loss: 0.9206
Epoch [    6/  100] | d_loss: 1.2460 | g_loss: 0.9680
Epoch [    6/  100] | d_loss: 1.3425 | g_loss: 0.8214
Epoch [    7/  100] | d_loss: 1.3468 | g_loss: 0.9200
Epoch [    7/  100] | d_loss: 1.3098 | g_loss: 1.0249
Epoch [    7/  100] | d_loss: 1.2890 | g_loss: 0.9971
Epoch [    8/  100] | d_loss: 1.2636 | g_loss: 0.9520
Epoch [    8/  100] | d_loss: 1.2242 | g_loss: 0.9332
Epoch [    8/  100] | d_loss: 1.3774 | g_loss: 1.0290
Epoch [    9/  100] | d_loss: 1.2166 | g_loss: 1.1847
Epoch [    9/  100] | d_loss: 1.2757 | g_loss: 0.8795
Epoch [    9/  100] | d_loss: 1.3350 | g_loss: 0.9848
Epoch [   10/  100] | d_loss: 1.2258 | g_loss: 1.2294
Epoch [   10/  100] | d_loss: 1.2826 | g_loss: 1.0322
Epoch [   10/  100] | d_loss: 1.3249 | g_loss: 0.7789
Epoch [   11/  100] | d_loss: 1.4087 | g_loss: 0.8040
Epoch [   11/  100] | d_loss: 1.3091 | g_loss: 0.9468
Epoch [   11/  100] | d_loss: 1.3661 | g_loss: 0.8582
Epoch [   12/  100] | d_loss: 1.2972 | g_loss: 1.0706
Epoch [   12/  100] | d_loss: 1.2910 | g_loss: 0.9599
Epoch [   12/  100] | d_loss: 1.4469 | g_loss: 0.9846
Epoch [   13/  100] | d_loss: 1.2522 | g_loss: 0.9096
Epoch [   13/  100] | d_loss: 1.3038 | g_loss: 0.9086
Epoch [   13/  100] | d_loss: 1.3410 | g_loss: 0.9007
Epoch [   14/  100] | d_loss: 1.3972 | g_loss: 0.9263
```

```
Epoch [   14/  100] | d_loss: 1.1457 | g_loss: 0.9268
Epoch [   14/  100] | d_loss: 1.4067 | g_loss: 1.0591
Epoch [   15/  100] | d_loss: 1.3855 | g_loss: 1.0274
Epoch [   15/  100] | d_loss: 1.2194 | g_loss: 1.1702
Epoch [   15/  100] | d_loss: 1.3796 | g_loss: 0.8596
Epoch [   16/  100] | d_loss: 1.2803 | g_loss: 1.0645
Epoch [   16/  100] | d_loss: 1.3096 | g_loss: 0.9516
Epoch [   16/  100] | d_loss: 1.3721 | g_loss: 0.9343
Epoch [   17/  100] | d_loss: 1.2190 | g_loss: 1.2036
Epoch [   17/  100] | d_loss: 1.2734 | g_loss: 0.9071
Epoch [   17/  100] | d_loss: 1.4173 | g_loss: 0.9956
Epoch [   18/  100] | d_loss: 1.2578 | g_loss: 1.1108
Epoch [   18/  100] | d_loss: 1.3411 | g_loss: 0.9701
Epoch [   18/  100] | d_loss: 1.4021 | g_loss: 0.9006
Epoch [   19/  100] | d_loss: 1.3386 | g_loss: 0.9160
Epoch [   19/  100] | d_loss: 1.2963 | g_loss: 0.9935
Epoch [   19/  100] | d_loss: 1.4204 | g_loss: 0.9611
Epoch [   20/  100] | d_loss: 1.3744 | g_loss: 0.9034
Epoch [   20/  100] | d_loss: 1.2739 | g_loss: 0.9600
Epoch [   20/  100] | d_loss: 1.3173 | g_loss: 0.9407
Epoch [   21/  100] | d_loss: 1.3409 | g_loss: 0.9330
Epoch [   21/  100] | d_loss: 1.3033 | g_loss: 0.8945
Epoch [   21/  100] | d_loss: 1.3469 | g_loss: 0.9986
Epoch [   22/  100] | d_loss: 1.3109 | g_loss: 1.1370
Epoch [   22/  100] | d_loss: 1.3431 | g_loss: 1.1827
Epoch [   22/  100] | d_loss: 1.4242 | g_loss: 0.8984
Epoch [   23/  100] | d_loss: 1.3188 | g_loss: 1.0079
Epoch [   23/  100] | d_loss: 1.2875 | g_loss: 0.9586
Epoch [   23/  100] | d_loss: 1.2436 | g_loss: 1.0532
Epoch [   24/  100] | d_loss: 1.3047 | g_loss: 0.8399
Epoch [   24/  100] | d_loss: 1.2922 | g_loss: 1.0547
Epoch [   24/  100] | d_loss: 1.3415 | g_loss: 0.9550
Epoch [   25/  100] | d_loss: 1.3227 | g_loss: 1.0760
Epoch [   25/  100] | d_loss: 1.3114 | g_loss: 1.0052
Epoch [   25/  100] | d_loss: 1.3168 | g_loss: 0.8248
Epoch [   26/  100] | d_loss: 1.3288 | g_loss: 1.0172
Epoch [   26/  100] | d_loss: 1.2819 | g_loss: 0.8772
Epoch [   26/  100] | d_loss: 1.3899 | g_loss: 0.9913
Epoch [   27/  100] | d_loss: 1.3186 | g_loss: 0.9003
Epoch [   27/  100] | d_loss: 1.3118 | g_loss: 0.9462
Epoch [   27/  100] | d_loss: 1.3907 | g_loss: 1.1392
Epoch [   28/  100] | d_loss: 1.2426 | g_loss: 0.9351
Epoch [   28/  100] | d_loss: 1.3180 | g_loss: 0.9030
Epoch [   28/  100] | d_loss: 1.4644 | g_loss: 1.1859
Epoch [   29/  100] | d_loss: 1.3197 | g_loss: 0.7891
Epoch [   29/  100] | d_loss: 1.2137 | g_loss: 1.0703
Epoch [   29/  100] | d_loss: 1.2770 | g_loss: 0.9848
Epoch [   30/  100] | d_loss: 1.2925 | g_loss: 0.8826
```

```
Epoch [   30/  100] | d_loss: 1.2893 | g_loss: 0.8774
Epoch [   30/  100] | d_loss: 1.2956 | g_loss: 1.0330
Epoch [   31/  100] | d_loss: 1.2196 | g_loss: 0.8091
Epoch [   31/  100] | d_loss: 1.3566 | g_loss: 0.9080
Epoch [   31/  100] | d_loss: 1.2753 | g_loss: 1.1016
Epoch [   32/  100] | d_loss: 1.2560 | g_loss: 0.9505
Epoch [   32/  100] | d_loss: 1.2989 | g_loss: 1.0042
Epoch [   32/  100] | d_loss: 1.3643 | g_loss: 0.8927
Epoch [   33/  100] | d_loss: 1.2769 | g_loss: 0.9737
Epoch [   33/  100] | d_loss: 1.3769 | g_loss: 0.8789
Epoch [   33/  100] | d_loss: 1.3109 | g_loss: 1.0953
Epoch [   34/  100] | d_loss: 1.3719 | g_loss: 0.9287
Epoch [   34/  100] | d_loss: 1.3272 | g_loss: 0.9782
Epoch [   34/  100] | d_loss: 1.3579 | g_loss: 0.9943
Epoch [   35/  100] | d_loss: 1.2647 | g_loss: 0.9844
Epoch [   35/  100] | d_loss: 1.3131 | g_loss: 0.9080
Epoch [   35/  100] | d_loss: 1.3689 | g_loss: 0.9350
Epoch [   36/  100] | d_loss: 1.2335 | g_loss: 1.1166
Epoch [   36/  100] | d_loss: 1.1683 | g_loss: 1.1547
Epoch [   36/  100] | d_loss: 1.3894 | g_loss: 0.9600
Epoch [   37/  100] | d_loss: 1.4048 | g_loss: 1.0444
Epoch [   37/  100] | d_loss: 1.2630 | g_loss: 0.8688
Epoch [   37/  100] | d_loss: 1.3267 | g_loss: 0.8951
Epoch [   38/  100] | d_loss: 1.2910 | g_loss: 0.8282
Epoch [   38/  100] | d_loss: 1.2974 | g_loss: 1.0162
Epoch [   38/  100] | d_loss: 1.3941 | g_loss: 0.9535
Epoch [   39/  100] | d_loss: 1.1903 | g_loss: 1.2965
Epoch [   39/  100] | d_loss: 1.2944 | g_loss: 0.8642
Epoch [   39/  100] | d_loss: 1.3383 | g_loss: 0.8559
Epoch [   40/  100] | d_loss: 1.1910 | g_loss: 1.4142
Epoch [   40/  100] | d_loss: 1.2928 | g_loss: 1.0042
Epoch [   40/  100] | d_loss: 1.4476 | g_loss: 0.8785
Epoch [   41/  100] | d_loss: 1.2930 | g_loss: 1.1515
Epoch [   41/  100] | d_loss: 1.2203 | g_loss: 1.0401
Epoch [   41/  100] | d_loss: 1.3833 | g_loss: 0.9253
Epoch [   42/  100] | d_loss: 1.2915 | g_loss: 1.1200
Epoch [   42/  100] | d_loss: 1.2022 | g_loss: 1.2603
Epoch [   42/  100] | d_loss: 1.3558 | g_loss: 1.0367
Epoch [   43/  100] | d_loss: 1.3766 | g_loss: 1.1104
Epoch [   43/  100] | d_loss: 1.2289 | g_loss: 0.9819
Epoch [   43/  100] | d_loss: 1.3489 | g_loss: 0.8969
Epoch [   44/  100] | d_loss: 1.3483 | g_loss: 0.8587
Epoch [   44/  100] | d_loss: 1.2589 | g_loss: 0.8566
Epoch [   44/  100] | d_loss: 1.4368 | g_loss: 0.9920
Epoch [   45/  100] | d_loss: 1.3017 | g_loss: 1.4002
Epoch [   45/  100] | d_loss: 1.2612 | g_loss: 0.8207
Epoch [   45/  100] | d_loss: 1.2975 | g_loss: 0.9893
Epoch [   46/  100] | d_loss: 1.3232 | g_loss: 1.0155
```

```
Epoch [   46/  100] | d_loss: 1.2396 | g_loss: 1.3825
Epoch [   46/  100] | d_loss: 1.3655 | g_loss: 0.9999
Epoch [   47/  100] | d_loss: 1.2516 | g_loss: 0.9896
Epoch [   47/  100] | d_loss: 1.2728 | g_loss: 0.9268
Epoch [   47/  100] | d_loss: 1.4520 | g_loss: 1.0270
Epoch [   48/  100] | d_loss: 1.3270 | g_loss: 1.0549
Epoch [   48/  100] | d_loss: 1.1733 | g_loss: 0.9841
Epoch [   48/  100] | d_loss: 1.3413 | g_loss: 0.8965
Epoch [   49/  100] | d_loss: 1.2776 | g_loss: 0.8877
Epoch [   49/  100] | d_loss: 1.2019 | g_loss: 1.1288
Epoch [   49/  100] | d_loss: 1.3470 | g_loss: 1.0216
Epoch [   50/  100] | d_loss: 1.2505 | g_loss: 0.9932
Epoch [   50/  100] | d_loss: 1.3520 | g_loss: 0.8499
Epoch [   50/  100] | d_loss: 1.2858 | g_loss: 0.9268
Epoch [   51/  100] | d_loss: 1.3898 | g_loss: 1.0027
Epoch [   51/  100] | d_loss: 1.2875 | g_loss: 0.9086
Epoch [   51/  100] | d_loss: 1.2878 | g_loss: 1.0496
Epoch [   52/  100] | d_loss: 1.2401 | g_loss: 0.8165
Epoch [   52/  100] | d_loss: 1.2490 | g_loss: 0.7978
Epoch [   52/  100] | d_loss: 1.2082 | g_loss: 0.9915
Epoch [   53/  100] | d_loss: 1.2312 | g_loss: 1.0420
Epoch [   53/  100] | d_loss: 1.3087 | g_loss: 1.2269
Epoch [   53/  100] | d_loss: 1.2136 | g_loss: 1.1641
Epoch [   54/  100] | d_loss: 1.2707 | g_loss: 0.8842
Epoch [   54/  100] | d_loss: 1.2299 | g_loss: 0.8746
Epoch [   54/  100] | d_loss: 1.3665 | g_loss: 0.9369
Epoch [   55/  100] | d_loss: 1.3157 | g_loss: 0.8254
Epoch [   55/  100] | d_loss: 1.1229 | g_loss: 1.0649
Epoch [   55/  100] | d_loss: 1.2802 | g_loss: 0.9593
Epoch [   56/  100] | d_loss: 1.5371 | g_loss: 1.1198
Epoch [   56/  100] | d_loss: 1.2728 | g_loss: 0.9548
Epoch [   56/  100] | d_loss: 1.3355 | g_loss: 1.1381
Epoch [   57/  100] | d_loss: 1.2697 | g_loss: 0.9037
Epoch [   57/  100] | d_loss: 1.2405 | g_loss: 1.2255
Epoch [   57/  100] | d_loss: 1.3482 | g_loss: 1.1158
Epoch [   58/  100] | d_loss: 1.3005 | g_loss: 1.1478
Epoch [   58/  100] | d_loss: 1.2590 | g_loss: 1.0317
Epoch [   58/  100] | d_loss: 1.2784 | g_loss: 1.0695
Epoch [   59/  100] | d_loss: 1.2271 | g_loss: 0.9009
Epoch [   59/  100] | d_loss: 1.2961 | g_loss: 0.9430
Epoch [   59/  100] | d_loss: 1.2620 | g_loss: 0.9433
Epoch [   60/  100] | d_loss: 1.2619 | g_loss: 1.0115
Epoch [   60/  100] | d_loss: 1.2794 | g_loss: 1.0872
Epoch [   60/  100] | d_loss: 1.4345 | g_loss: 0.9186
Epoch [   61/  100] | d_loss: 1.2988 | g_loss: 0.9782
Epoch [   61/  100] | d_loss: 1.1591 | g_loss: 1.2982
Epoch [   61/  100] | d_loss: 1.4653 | g_loss: 0.9955
Epoch [   62/  100] | d_loss: 1.3233 | g_loss: 0.8551
```

```
Epoch [   62/  100] | d_loss: 1.4645 | g_loss: 0.9856
Epoch [   62/  100] | d_loss: 1.3310 | g_loss: 0.9223
Epoch [   63/  100] | d_loss: 1.2351 | g_loss: 1.6538
Epoch [   63/  100] | d_loss: 1.2489 | g_loss: 1.0497
Epoch [   63/  100] | d_loss: 1.2768 | g_loss: 1.0256
Epoch [   64/  100] | d_loss: 1.3026 | g_loss: 1.1597
Epoch [   64/  100] | d_loss: 1.3820 | g_loss: 1.1499
Epoch [   64/  100] | d_loss: 1.3198 | g_loss: 1.0013
Epoch [   65/  100] | d_loss: 1.2573 | g_loss: 1.2529
Epoch [   65/  100] | d_loss: 1.2858 | g_loss: 1.0339
Epoch [   65/  100] | d_loss: 1.3242 | g_loss: 0.9070
Epoch [   66/  100] | d_loss: 1.3821 | g_loss: 1.1427
Epoch [   66/  100] | d_loss: 1.3039 | g_loss: 0.9871
Epoch [   66/  100] | d_loss: 1.3826 | g_loss: 1.0136
Epoch [   67/  100] | d_loss: 1.2751 | g_loss: 0.7706
Epoch [   67/  100] | d_loss: 1.1972 | g_loss: 1.0919
Epoch [   67/  100] | d_loss: 1.4427 | g_loss: 0.9262
Epoch [   68/  100] | d_loss: 1.2587 | g_loss: 0.9697
Epoch [   68/  100] | d_loss: 1.1228 | g_loss: 1.2282
Epoch [   68/  100] | d_loss: 1.4037 | g_loss: 0.9404
Epoch [   69/  100] | d_loss: 1.3538 | g_loss: 0.8863
Epoch [   69/  100] | d_loss: 1.2865 | g_loss: 0.8829
Epoch [   69/  100] | d_loss: 1.3817 | g_loss: 1.0767
Epoch [   70/  100] | d_loss: 1.3271 | g_loss: 0.9144
Epoch [   70/  100] | d_loss: 1.2499 | g_loss: 1.0674
Epoch [   70/  100] | d_loss: 1.2603 | g_loss: 0.9379
Epoch [   71/  100] | d_loss: 1.2161 | g_loss: 1.0424
Epoch [   71/  100] | d_loss: 1.3128 | g_loss: 0.9293
Epoch [   71/  100] | d_loss: 1.4150 | g_loss: 1.0259
Epoch [   72/  100] | d_loss: 1.3225 | g_loss: 0.8122
Epoch [   72/  100] | d_loss: 1.3034 | g_loss: 1.2024
Epoch [   72/  100] | d_loss: 1.4149 | g_loss: 1.3195
Epoch [   73/  100] | d_loss: 1.3639 | g_loss: 0.9724
Epoch [   73/  100] | d_loss: 1.1890 | g_loss: 1.1638
Epoch [   73/  100] | d_loss: 1.3886 | g_loss: 0.9566
Epoch [   74/  100] | d_loss: 1.3317 | g_loss: 0.8894
Epoch [   74/  100] | d_loss: 1.0961 | g_loss: 1.2001
Epoch [   74/  100] | d_loss: 1.3188 | g_loss: 0.9089
Epoch [   75/  100] | d_loss: 1.2584 | g_loss: 1.0603
Epoch [   75/  100] | d_loss: 1.2356 | g_loss: 1.0055
Epoch [   75/  100] | d_loss: 1.2381 | g_loss: 0.9942
Epoch [   76/  100] | d_loss: 1.3108 | g_loss: 0.8810
Epoch [   76/  100] | d_loss: 1.2471 | g_loss: 0.9351
Epoch [   76/  100] | d_loss: 1.3371 | g_loss: 1.0227
Epoch [   77/  100] | d_loss: 1.3044 | g_loss: 1.0448
Epoch [   77/  100] | d_loss: 1.3171 | g_loss: 1.0235
Epoch [   77/  100] | d_loss: 1.4401 | g_loss: 0.9617
Epoch [   78/  100] | d_loss: 1.2551 | g_loss: 1.0200
```

```
Epoch [   78/  100] | d_loss: 1.2172 | g_loss: 0.9833
Epoch [   78/  100] | d_loss: 1.4484 | g_loss: 1.0597
Epoch [   79/  100] | d_loss: 1.2945 | g_loss: 1.0672
Epoch [   79/  100] | d_loss: 1.2892 | g_loss: 1.0825
Epoch [   79/  100] | d_loss: 1.3873 | g_loss: 1.1598
Epoch [   80/  100] | d_loss: 1.3170 | g_loss: 1.0550
Epoch [   80/  100] | d_loss: 1.2631 | g_loss: 0.9427
Epoch [   80/  100] | d_loss: 1.3912 | g_loss: 1.0001
Epoch [   81/  100] | d_loss: 1.2396 | g_loss: 1.2220
Epoch [   81/  100] | d_loss: 1.2506 | g_loss: 0.9819
Epoch [   81/  100] | d_loss: 1.3788 | g_loss: 0.9197
Epoch [   82/  100] | d_loss: 1.3445 | g_loss: 0.8363
Epoch [   82/  100] | d_loss: 1.2379 | g_loss: 1.1038
Epoch [   82/  100] | d_loss: 1.3953 | g_loss: 1.3281
Epoch [   83/  100] | d_loss: 1.2572 | g_loss: 1.1708
Epoch [   83/  100] | d_loss: 1.2816 | g_loss: 0.9025
Epoch [   83/  100] | d_loss: 1.3750 | g_loss: 0.8586
Epoch [   84/  100] | d_loss: 1.3142 | g_loss: 0.8887
Epoch [   84/  100] | d_loss: 1.1706 | g_loss: 1.0228
Epoch [   84/  100] | d_loss: 1.2826 | g_loss: 1.4092
Epoch [   85/  100] | d_loss: 1.2179 | g_loss: 1.1822
Epoch [   85/  100] | d_loss: 1.2414 | g_loss: 1.0184
Epoch [   85/  100] | d_loss: 1.2499 | g_loss: 0.9643
Epoch [   86/  100] | d_loss: 1.4081 | g_loss: 1.0215
Epoch [   86/  100] | d_loss: 1.0672 | g_loss: 1.2303
Epoch [   86/  100] | d_loss: 1.3982 | g_loss: 0.9861
Epoch [   87/  100] | d_loss: 1.2966 | g_loss: 0.9585
Epoch [   87/  100] | d_loss: 1.2863 | g_loss: 0.9498
Epoch [   87/  100] | d_loss: 1.3952 | g_loss: 0.9766
Epoch [   88/  100] | d_loss: 1.3173 | g_loss: 1.2159
Epoch [   88/  100] | d_loss: 1.2444 | g_loss: 1.1231
Epoch [   88/  100] | d_loss: 1.3617 | g_loss: 0.9537
Epoch [   89/  100] | d_loss: 1.2070 | g_loss: 1.0308
Epoch [   89/  100] | d_loss: 1.3292 | g_loss: 1.0234
Epoch [   89/  100] | d_loss: 1.2550 | g_loss: 1.0707
Epoch [   90/  100] | d_loss: 1.2781 | g_loss: 0.9172
Epoch [   90/  100] | d_loss: 1.2674 | g_loss: 1.0166
Epoch [   90/  100] | d_loss: 1.3347 | g_loss: 1.1140
Epoch [   91/  100] | d_loss: 1.3387 | g_loss: 0.9324
Epoch [   91/  100] | d_loss: 1.2619 | g_loss: 0.9601
Epoch [   91/  100] | d_loss: 1.4142 | g_loss: 1.0002
Epoch [   92/  100] | d_loss: 1.3123 | g_loss: 0.8796
Epoch [   92/  100] | d_loss: 1.2449 | g_loss: 0.9165
Epoch [   92/  100] | d_loss: 1.2785 | g_loss: 0.8928
Epoch [   93/  100] | d_loss: 1.2613 | g_loss: 1.4571
Epoch [   93/  100] | d_loss: 1.2500 | g_loss: 0.9886
Epoch [   93/  100] | d_loss: 1.3941 | g_loss: 0.9039
Epoch [   94/  100] | d_loss: 1.2658 | g_loss: 0.9556
```

```
Epoch [   94/  100] | d_loss: 1.2096 | g_loss: 1.0293
Epoch [   94/  100] | d_loss: 1.2934 | g_loss: 0.9862
Epoch [   95/  100] | d_loss: 1.3609 | g_loss: 1.4349
Epoch [   95/  100] | d_loss: 1.2245 | g_loss: 1.1067
Epoch [   95/  100] | d_loss: 1.3066 | g_loss: 0.9759
Epoch [   96/  100] | d_loss: 1.2949 | g_loss: 0.8319
Epoch [   96/  100] | d_loss: 1.3333 | g_loss: 0.9025
Epoch [   96/  100] | d_loss: 1.2014 | g_loss: 1.1053
Epoch [   97/  100] | d_loss: 1.2985 | g_loss: 1.0141
Epoch [   97/  100] | d_loss: 1.2627 | g_loss: 1.0140
Epoch [   97/  100] | d_loss: 1.3135 | g_loss: 0.9844
Epoch [   98/  100] | d_loss: 1.3431 | g_loss: 0.9857
Epoch [   98/  100] | d_loss: 1.1359 | g_loss: 0.8790
Epoch [   98/  100] | d_loss: 1.2968 | g_loss: 1.0595
Epoch [   99/  100] | d_loss: 1.2918 | g_loss: 1.0832
Epoch [   99/  100] | d_loss: 1.2491 | g_loss: 1.1004
Epoch [   99/  100] | d_loss: 1.3484 | g_loss: 0.9297
Epoch [  100/  100] | d_loss: 1.2533 | g_loss: 0.9658
Epoch [  100/  100] | d_loss: 1.2838 | g_loss: 1.0807
Epoch [  100/  100] | d_loss: 1.3661 | g_loss: 1.1609
```

2.8 Training loss

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

In [44]:
```
fig, ax = plt.subplots()
losses = np.array(losses)
plt.plot(losses.T[0], label='Discriminator')
plt.plot(losses.T[1], label='Generator')
plt.title("Training Losses")
plt.legend()
```

Out[44]:
```
<matplotlib.legend.Legend at 0x7fa4c8702160>
```

2.9 Generator samples from training

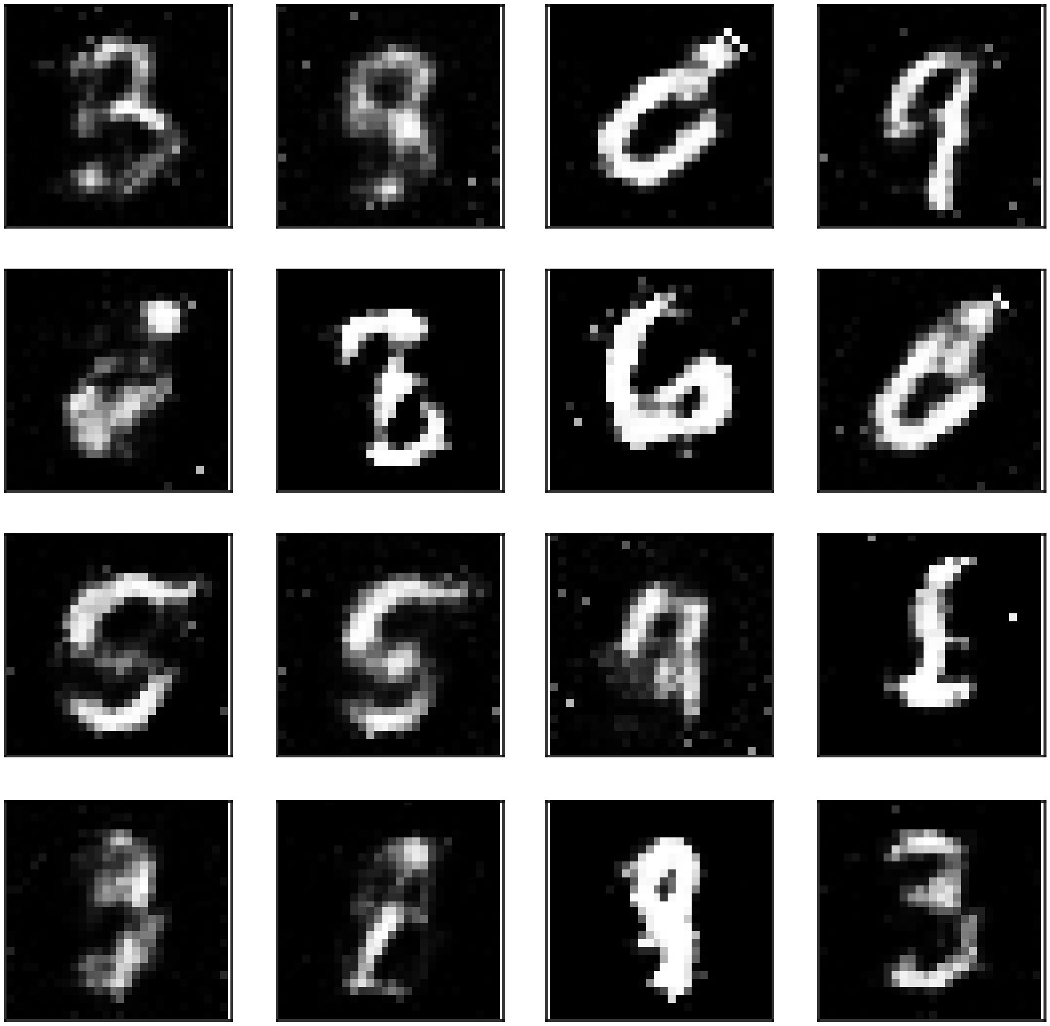
Here we can view samples of images from the generator. First we'll look at the images we saved during training.

```
In [45]: # helper function for viewing a list of passed in sample images
         def view_samples(epoch, samples):
             fig, axes = plt.subplots(figsize=(7,7), nrows=4, ncols=4, sharey=True, sharex=True)
             for ax, img in zip(axes.flatten(), samples[epoch]):
                 img = img.detach()
                 ax.xaxis.set_visible(False)
                 ax.yaxis.set_visible(False)
                 im = ax.imshow(img.reshape((28,28)), cmap='Greys_r')
```

```
In [46]: # Load samples from generator, taken while training
         with open('train_samples.pkl', 'rb') as f:
             samples = pkl.load(f)
```

These are samples from the final training epoch. You can see the generator is able to reproduce numbers like 1, 7, 3, 2. Since this is just a sample, it isn't representative of the full range of images this generator can make.

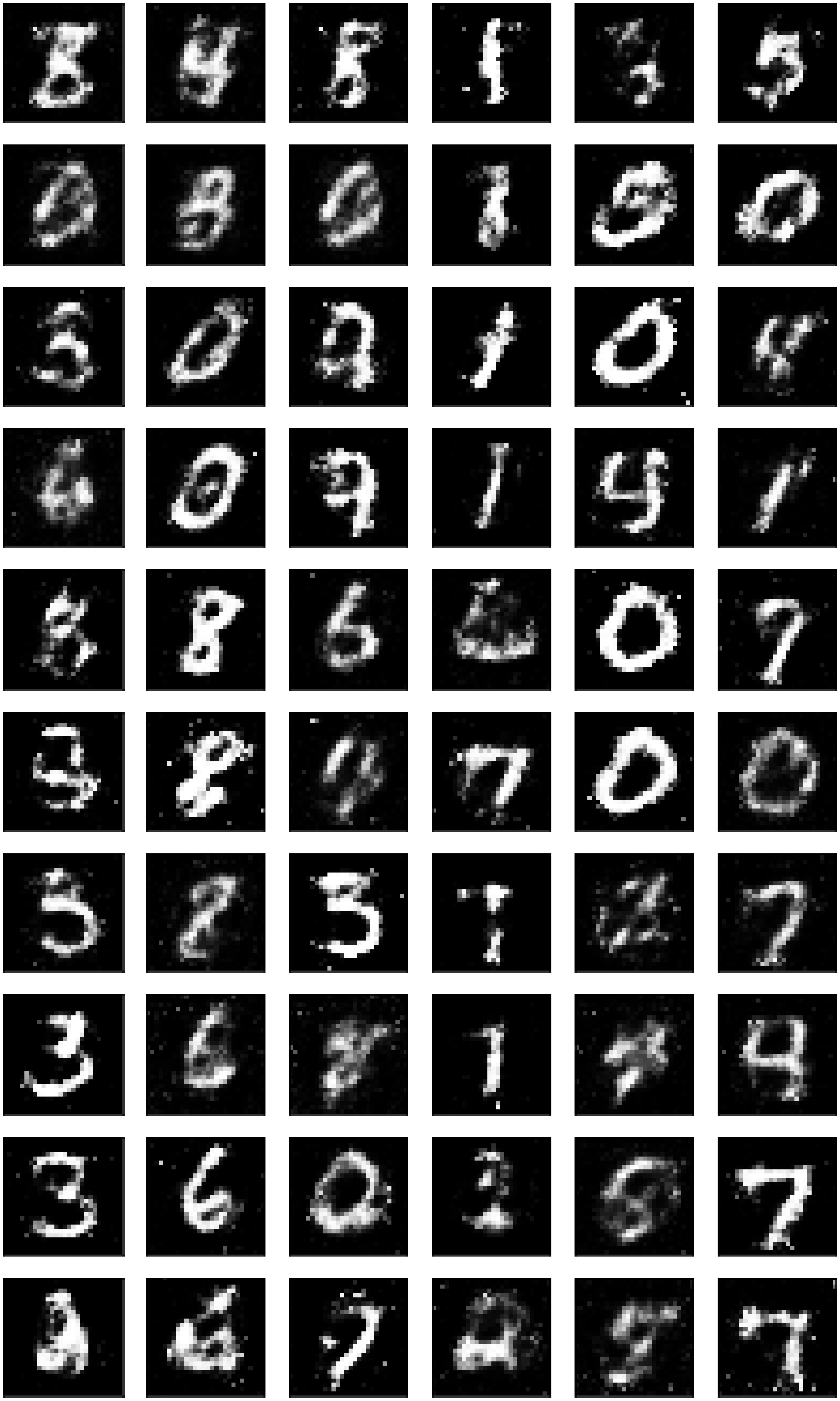
```
In [47]: # -1 indicates final epoch's samples (the last in the list)
         view_samples(-1, samples)
```



Below I'm showing the generated images as the network was training, every 10 epochs.

```
In [48]: rows = 10 # split epochs into 10, so 100/10 = every 10 epochs
         cols = 6
         fig, axes = plt.subplots(figsize=(7,12), 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()
                 ax.imshow(img.reshape((28,28)), cmap='Greys_r')
                 ax.xaxis.set_visible(False)
                 ax.yaxis.set_visible(False)
```



It starts out as all noise. Then it learns to make only the center white and the rest black. You can start to see some number like structures appear out of the noise like 1s and 9s.

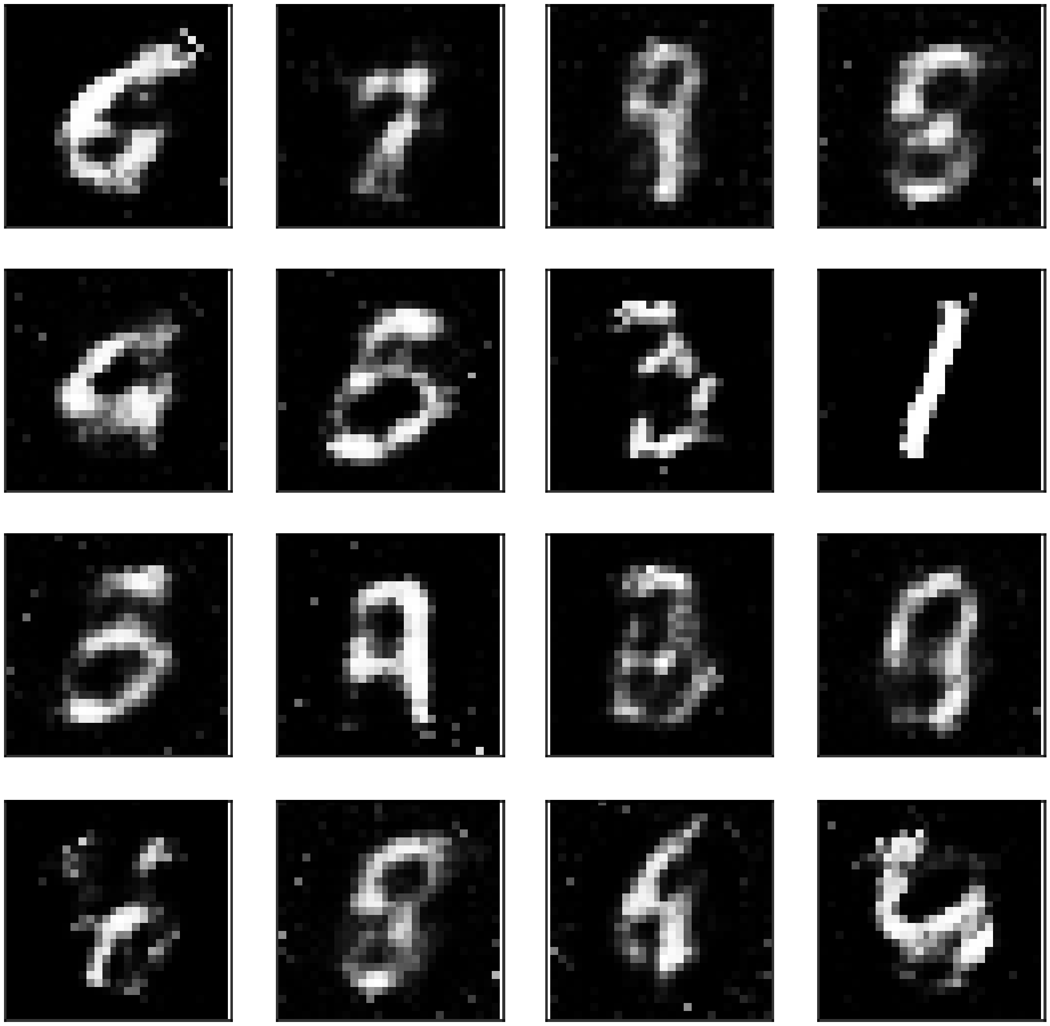
2.10 Sampling from the generator

We can also get completely new images from the generator by using the checkpoint we saved after training. **We just need to pass in a new latent vector z and we'll get new samples**!

```
In [49]: # randomly generated, new latent vectors
         sample_size=16
         rand_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
         rand_z = torch.from_numpy(rand_z).float()

         G.eval() # eval mode
         # generated samples
         rand_images = G(rand_z)

         # 0 indicates the first set of samples in the passed in list
         # and we only have one batch of samples, here
         view_samples(0, [rand_images])
```



```
In [ ]:
```

```
In [ ]:
```