MNIST_GAN_Exercise

May 8, 2020

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 [1]:
```
%matplotlib inline

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

In [2]:
```
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)
```

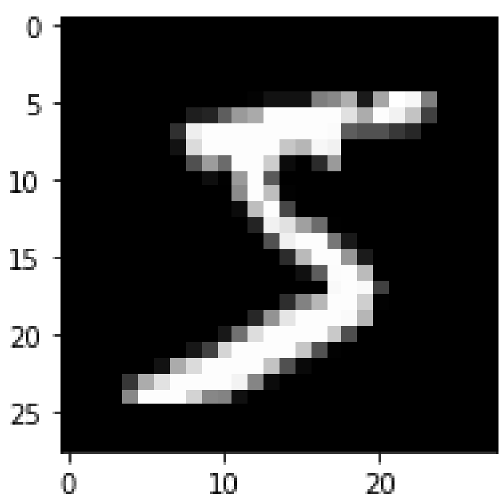
1.0.1 Visualize the data

In [3]:
```
# 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[3]: <matplotlib.image.AxesImage at 0x7fb7d72f8588>



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 [4]: 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__()

                # define hidden linear layers
                self.fc1 = nn.Linear(input_size, hidden_dim*4)
                self.fc2 = nn.Linear(hidden_dim*4, hidden_dim*2)
                self.fc3 = nn.Linear(hidden_dim*2, hidden_dim)

                # final fully-connected layer
                self.fc4 = 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)
                # all hidden layers
                x = F.leaky_relu(self.fc1(x), 0.2) # (input, negative_slope=0.2)
                x = self.dropout(x)
                x = F.leaky_relu(self.fc2(x), 0.2)
```

```
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)
        # final layer
        out = self.fc4(x)

        return out
```

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 [5]: class Generator(nn.Module):

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

                # define hidden linear layers
                self.fc1 = nn.Linear(input_size, hidden_dim)
                self.fc2 = nn.Linear(hidden_dim, hidden_dim*2)
                self.fc3 = nn.Linear(hidden_dim*2, hidden_dim*4)

                # final fully-connected layer
                self.fc4 = nn.Linear(hidden_dim*4, output_size)

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

            def forward(self, x):
                # all hidden layers
                x = F.leaky_relu(self.fc1(x), 0.2) # (input, negative_slope=0.2)
                x = self.dropout(x)
                x = F.leaky_relu(self.fc2(x), 0.2)
                x = self.dropout(x)
                x = F.leaky_relu(self.fc3(x), 0.2)
                x = self.dropout(x)
                # final layer with tanh applied
                out = F.tanh(self.fc4(x))

                return out
```

2.3 Model hyperparameters

In [6]:
```
# Discriminator hyperparams

# Size of input image to discriminator (28*28)
input_size = 784
# Size of discriminator output (real or fake)
d_output_size = 1
# Size of last hidden layer in the discriminator
d_hidden_size = 32

# Generator hyperparams

# Size of latent vector to give to generator
z_size = 100
# Size of discriminator output (generated image)
g_output_size = 784
# Size of first hidden layer in the generator
g_hidden_size = 32
```

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 [7]:
```
# 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(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=32, bias=True)
  (fc4): Linear(in_features=32, out_features=1, bias=True)
  (dropout): Dropout(p=0.3)
)

Generator(
  (fc1): Linear(in_features=100, out_features=32, bias=True)
  (fc2): Linear(in_features=32, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=128, bias=True)
  (fc4): Linear(in_features=128, 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 [8]: # Calculate losses
        def real_loss(D_out, smooth=False):
            batch_size = D_out.size(0)
            # label smoothing
            if smooth:
                # smooth, real labels = 0.9
                labels = torch.ones(batch_size)*0.9
            else:
                labels = torch.ones(batch_size) # real labels = 1

            # numerically stable loss
            criterion = nn.BCEWithLogitsLoss()
            # calculate loss
            loss = criterion(D_out.squeeze(), labels)
            return loss
```

```
def fake_loss(D_out):
    batch_size = D_out.size(0)
    labels = torch.zeros(batch_size) # fake labels = 0
    criterion = nn.BCEWithLogitsLoss()
    # calculate 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 [9]: import torch.optim as optim

        # 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 [10]: import pickle as pkl

         # training hyperparams
         num_epochs = 60

         # 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
                 # smooth the real labels
                 D_real = D(real_images)
                 d_real_loss = real_loss(D_real, 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(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()
        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:
            # 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
```

```python
            samples_z = G(fixed_z)
            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/   60] | d_loss: 1.3884 | g_loss: 0.7077
Epoch [    1/   60] | d_loss: 1.2285 | g_loss: 1.3171
Epoch [    1/   60] | d_loss: 1.3043 | g_loss: 0.7438
Epoch [    2/   60] | d_loss: 1.2051 | g_loss: 0.9774
Epoch [    2/   60] | d_loss: 0.9847 | g_loss: 1.7905
Epoch [    2/   60] | d_loss: 1.4263 | g_loss: 0.8025
Epoch [    3/   60] | d_loss: 1.2751 | g_loss: 0.8635
Epoch [    3/   60] | d_loss: 0.8772 | g_loss: 1.4054
Epoch [    3/   60] | d_loss: 1.0519 | g_loss: 1.1982
Epoch [    4/   60] | d_loss: 1.4932 | g_loss: 2.1488
Epoch [    4/   60] | d_loss: 1.0803 | g_loss: 1.3460
Epoch [    4/   60] | d_loss: 1.0941 | g_loss: 1.8086
Epoch [    5/   60] | d_loss: 1.0234 | g_loss: 1.6869
Epoch [    5/   60] | d_loss: 1.1167 | g_loss: 1.6820
Epoch [    5/   60] | d_loss: 1.4299 | g_loss: 0.8281
Epoch [    6/   60] | d_loss: 1.0371 | g_loss: 1.5915
Epoch [    6/   60] | d_loss: 1.1193 | g_loss: 1.3167
Epoch [    6/   60] | d_loss: 1.1302 | g_loss: 1.1695
Epoch [    7/   60] | d_loss: 1.3306 | g_loss: 1.0796
Epoch [    7/   60] | d_loss: 1.2251 | g_loss: 1.5026
Epoch [    7/   60] | d_loss: 1.0923 | g_loss: 1.2386
Epoch [    8/   60] | d_loss: 1.2108 | g_loss: 1.2196
Epoch [    8/   60] | d_loss: 1.2018 | g_loss: 1.0554
Epoch [    8/   60] | d_loss: 1.3157 | g_loss: 1.1538
Epoch [    9/   60] | d_loss: 1.2553 | g_loss: 0.9547
Epoch [    9/   60] | d_loss: 1.2158 | g_loss: 1.0619
Epoch [    9/   60] | d_loss: 1.1163 | g_loss: 1.1443
Epoch [   10/   60] | d_loss: 1.3878 | g_loss: 1.2848
Epoch [   10/   60] | d_loss: 1.4674 | g_loss: 0.8704
Epoch [   10/   60] | d_loss: 1.1789 | g_loss: 1.1959
Epoch [   11/   60] | d_loss: 1.1506 | g_loss: 1.1854
Epoch [   11/   60] | d_loss: 1.2284 | g_loss: 0.8909
Epoch [   11/   60] | d_loss: 1.2638 | g_loss: 0.9715
Epoch [   12/   60] | d_loss: 1.3462 | g_loss: 1.0427
Epoch [   12/   60] | d_loss: 1.2359 | g_loss: 1.2030
Epoch [   12/   60] | d_loss: 1.2173 | g_loss: 0.8969
Epoch [   13/   60] | d_loss: 1.1959 | g_loss: 1.4167
Epoch [   13/   60] | d_loss: 1.3602 | g_loss: 1.1672
Epoch [   13/   60] | d_loss: 1.3264 | g_loss: 0.9296
```

```
Epoch [   14/   60] | d_loss: 1.2738 | g_loss: 0.9522
Epoch [   14/   60] | d_loss: 1.3238 | g_loss: 1.0279
Epoch [   14/   60] | d_loss: 1.4493 | g_loss: 0.8698
Epoch [   15/   60] | d_loss: 1.3362 | g_loss: 0.9259
Epoch [   15/   60] | d_loss: 1.3352 | g_loss: 1.0147
Epoch [   15/   60] | d_loss: 1.2065 | g_loss: 1.2479
Epoch [   16/   60] | d_loss: 1.2397 | g_loss: 1.6607
Epoch [   16/   60] | d_loss: 1.3028 | g_loss: 1.0501
Epoch [   16/   60] | d_loss: 1.3963 | g_loss: 1.0288
Epoch [   17/   60] | d_loss: 1.4047 | g_loss: 0.8405
Epoch [   17/   60] | d_loss: 1.3119 | g_loss: 0.8246
Epoch [   17/   60] | d_loss: 1.2459 | g_loss: 0.9801
Epoch [   18/   60] | d_loss: 1.2128 | g_loss: 1.1596
Epoch [   18/   60] | d_loss: 1.3366 | g_loss: 0.8877
Epoch [   18/   60] | d_loss: 1.3857 | g_loss: 0.9105
Epoch [   19/   60] | d_loss: 1.1782 | g_loss: 1.4711
Epoch [   19/   60] | d_loss: 1.1180 | g_loss: 1.7423
Epoch [   19/   60] | d_loss: 1.2733 | g_loss: 0.8968
Epoch [   20/   60] | d_loss: 1.1236 | g_loss: 1.2875
Epoch [   20/   60] | d_loss: 1.2618 | g_loss: 1.0150
Epoch [   20/   60] | d_loss: 1.3100 | g_loss: 0.8772
Epoch [   21/   60] | d_loss: 1.3610 | g_loss: 0.8693
Epoch [   21/   60] | d_loss: 1.2332 | g_loss: 0.9685
Epoch [   21/   60] | d_loss: 1.2803 | g_loss: 1.3458
Epoch [   22/   60] | d_loss: 1.3346 | g_loss: 0.9456
Epoch [   22/   60] | d_loss: 1.2718 | g_loss: 1.0138
Epoch [   22/   60] | d_loss: 1.3024 | g_loss: 0.9630
Epoch [   23/   60] | d_loss: 1.3203 | g_loss: 1.2992
Epoch [   23/   60] | d_loss: 1.3566 | g_loss: 0.9330
Epoch [   23/   60] | d_loss: 1.2482 | g_loss: 1.1637
Epoch [   24/   60] | d_loss: 1.4943 | g_loss: 1.0669
Epoch [   24/   60] | d_loss: 1.2290 | g_loss: 1.0777
Epoch [   24/   60] | d_loss: 1.3320 | g_loss: 0.9260
Epoch [   25/   60] | d_loss: 1.2518 | g_loss: 1.0554
Epoch [   25/   60] | d_loss: 1.3245 | g_loss: 0.8181
Epoch [   25/   60] | d_loss: 1.3363 | g_loss: 1.2943
Epoch [   26/   60] | d_loss: 1.1900 | g_loss: 1.0788
Epoch [   26/   60] | d_loss: 1.2416 | g_loss: 0.9704
Epoch [   26/   60] | d_loss: 1.4097 | g_loss: 0.9517
Epoch [   27/   60] | d_loss: 1.5551 | g_loss: 0.8930
Epoch [   27/   60] | d_loss: 1.2183 | g_loss: 1.0185
Epoch [   27/   60] | d_loss: 1.3388 | g_loss: 1.0019
Epoch [   28/   60] | d_loss: 1.3018 | g_loss: 1.3341
Epoch [   28/   60] | d_loss: 1.2652 | g_loss: 1.0707
Epoch [   28/   60] | d_loss: 1.3204 | g_loss: 0.8523
Epoch [   29/   60] | d_loss: 1.2488 | g_loss: 1.1985
Epoch [   29/   60] | d_loss: 1.2772 | g_loss: 0.9125
Epoch [   29/   60] | d_loss: 1.2434 | g_loss: 0.9989
```

```
Epoch [   30/   60] | d_loss: 1.3071 | g_loss: 1.1866
Epoch [   30/   60] | d_loss: 1.2758 | g_loss: 0.9508
Epoch [   30/   60] | d_loss: 1.3255 | g_loss: 1.0869
Epoch [   31/   60] | d_loss: 1.2181 | g_loss: 0.9239
Epoch [   31/   60] | d_loss: 1.2393 | g_loss: 0.9209
Epoch [   31/   60] | d_loss: 1.3457 | g_loss: 1.0523
Epoch [   32/   60] | d_loss: 1.2711 | g_loss: 1.1656
Epoch [   32/   60] | d_loss: 1.3096 | g_loss: 1.0965
Epoch [   32/   60] | d_loss: 1.3004 | g_loss: 1.0638
Epoch [   33/   60] | d_loss: 1.2859 | g_loss: 1.0291
Epoch [   33/   60] | d_loss: 1.2190 | g_loss: 1.0418
Epoch [   33/   60] | d_loss: 1.3476 | g_loss: 0.9261
Epoch [   34/   60] | d_loss: 1.3933 | g_loss: 0.9802
Epoch [   34/   60] | d_loss: 1.2783 | g_loss: 0.9696
Epoch [   34/   60] | d_loss: 1.4638 | g_loss: 1.3835
Epoch [   35/   60] | d_loss: 1.4258 | g_loss: 2.0603
Epoch [   35/   60] | d_loss: 1.2850 | g_loss: 0.9085
Epoch [   35/   60] | d_loss: 1.4509 | g_loss: 0.9419
Epoch [   36/   60] | d_loss: 1.3304 | g_loss: 0.9733
Epoch [   36/   60] | d_loss: 1.2524 | g_loss: 0.8699
Epoch [   36/   60] | d_loss: 1.3029 | g_loss: 0.9594
Epoch [   37/   60] | d_loss: 1.3028 | g_loss: 0.8500
Epoch [   37/   60] | d_loss: 1.3430 | g_loss: 0.8029
Epoch [   37/   60] | d_loss: 1.3396 | g_loss: 0.9733
Epoch [   38/   60] | d_loss: 1.3225 | g_loss: 1.0202
Epoch [   38/   60] | d_loss: 1.3669 | g_loss: 0.8398
Epoch [   38/   60] | d_loss: 1.4268 | g_loss: 0.9517
Epoch [   39/   60] | d_loss: 1.3312 | g_loss: 0.9041
Epoch [   39/   60] | d_loss: 1.2339 | g_loss: 0.9726
Epoch [   39/   60] | d_loss: 1.3242 | g_loss: 0.9406
Epoch [   40/   60] | d_loss: 1.3856 | g_loss: 0.8984
Epoch [   40/   60] | d_loss: 1.2799 | g_loss: 0.9118
Epoch [   40/   60] | d_loss: 1.2946 | g_loss: 1.2216
Epoch [   41/   60] | d_loss: 1.3181 | g_loss: 0.9175
Epoch [   41/   60] | d_loss: 1.2278 | g_loss: 0.8886
Epoch [   41/   60] | d_loss: 1.2813 | g_loss: 0.9294
Epoch [   42/   60] | d_loss: 1.2732 | g_loss: 0.9880
Epoch [   42/   60] | d_loss: 1.3195 | g_loss: 0.9519
Epoch [   42/   60] | d_loss: 1.3454 | g_loss: 1.0715
Epoch [   43/   60] | d_loss: 1.2020 | g_loss: 1.6028
Epoch [   43/   60] | d_loss: 1.3432 | g_loss: 0.8819
Epoch [   43/   60] | d_loss: 1.3206 | g_loss: 0.9768
Epoch [   44/   60] | d_loss: 1.3043 | g_loss: 0.8567
Epoch [   44/   60] | d_loss: 1.3472 | g_loss: 0.9475
Epoch [   44/   60] | d_loss: 1.3753 | g_loss: 0.8114
Epoch [   45/   60] | d_loss: 1.2313 | g_loss: 1.1159
Epoch [   45/   60] | d_loss: 1.2942 | g_loss: 0.8438
Epoch [   45/   60] | d_loss: 1.2299 | g_loss: 0.9296
```

```
Epoch [   46/   60] | d_loss: 1.2955 | g_loss: 1.0312
Epoch [   46/   60] | d_loss: 1.2527 | g_loss: 1.2390
Epoch [   46/   60] | d_loss: 1.3621 | g_loss: 0.8361
Epoch [   47/   60] | d_loss: 1.2230 | g_loss: 1.0662
Epoch [   47/   60] | d_loss: 1.2319 | g_loss: 0.9217
Epoch [   47/   60] | d_loss: 1.3416 | g_loss: 0.9799
Epoch [   48/   60] | d_loss: 1.2978 | g_loss: 1.0461
Epoch [   48/   60] | d_loss: 1.2277 | g_loss: 1.1099
Epoch [   48/   60] | d_loss: 1.2239 | g_loss: 1.4972
Epoch [   49/   60] | d_loss: 1.3298 | g_loss: 0.9697
Epoch [   49/   60] | d_loss: 1.2926 | g_loss: 1.0085
Epoch [   49/   60] | d_loss: 1.3520 | g_loss: 0.8952
Epoch [   50/   60] | d_loss: 1.1682 | g_loss: 0.9983
Epoch [   50/   60] | d_loss: 1.1878 | g_loss: 0.9267
Epoch [   50/   60] | d_loss: 1.4014 | g_loss: 0.9846
Epoch [   51/   60] | d_loss: 1.3085 | g_loss: 0.9878
Epoch [   51/   60] | d_loss: 1.2962 | g_loss: 0.9604
Epoch [   51/   60] | d_loss: 1.3530 | g_loss: 0.9212
Epoch [   52/   60] | d_loss: 1.3382 | g_loss: 0.8317
Epoch [   52/   60] | d_loss: 1.2786 | g_loss: 0.9968
Epoch [   52/   60] | d_loss: 1.3102 | g_loss: 0.9271
Epoch [   53/   60] | d_loss: 1.2568 | g_loss: 0.8155
Epoch [   53/   60] | d_loss: 1.2866 | g_loss: 1.0641
Epoch [   53/   60] | d_loss: 1.3421 | g_loss: 0.9050
Epoch [   54/   60] | d_loss: 1.3491 | g_loss: 1.0781
Epoch [   54/   60] | d_loss: 1.2876 | g_loss: 0.8498
Epoch [   54/   60] | d_loss: 1.2902 | g_loss: 0.9837
Epoch [   55/   60] | d_loss: 1.2502 | g_loss: 1.1737
Epoch [   55/   60] | d_loss: 1.2667 | g_loss: 1.0283
Epoch [   55/   60] | d_loss: 1.3517 | g_loss: 0.8530
Epoch [   56/   60] | d_loss: 1.2095 | g_loss: 0.9883
Epoch [   56/   60] | d_loss: 1.2448 | g_loss: 0.9131
Epoch [   56/   60] | d_loss: 1.2924 | g_loss: 0.9068
Epoch [   57/   60] | d_loss: 1.2963 | g_loss: 1.1111
Epoch [   57/   60] | d_loss: 1.2914 | g_loss: 0.8530
Epoch [   57/   60] | d_loss: 1.4525 | g_loss: 0.8508
Epoch [   58/   60] | d_loss: 1.3299 | g_loss: 0.8455
Epoch [   58/   60] | d_loss: 1.2705 | g_loss: 1.1248
Epoch [   58/   60] | d_loss: 1.3951 | g_loss: 1.0257
Epoch [   59/   60] | d_loss: 1.3116 | g_loss: 0.9647
Epoch [   59/   60] | d_loss: 1.2430 | g_loss: 0.9689
Epoch [   59/   60] | d_loss: 1.2870 | g_loss: 1.1637
Epoch [   60/   60] | d_loss: 1.3487 | g_loss: 1.2861
Epoch [   60/   60] | d_loss: 1.2684 | g_loss: 1.0935
Epoch [   60/   60] | d_loss: 1.2567 | g_loss: 1.0948
```

2.8 Training loss

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

```
In [11]: 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[11]: <matplotlib.legend.Legend at 0x7fb7cda36f28>
```

2.9 Generator samples from training

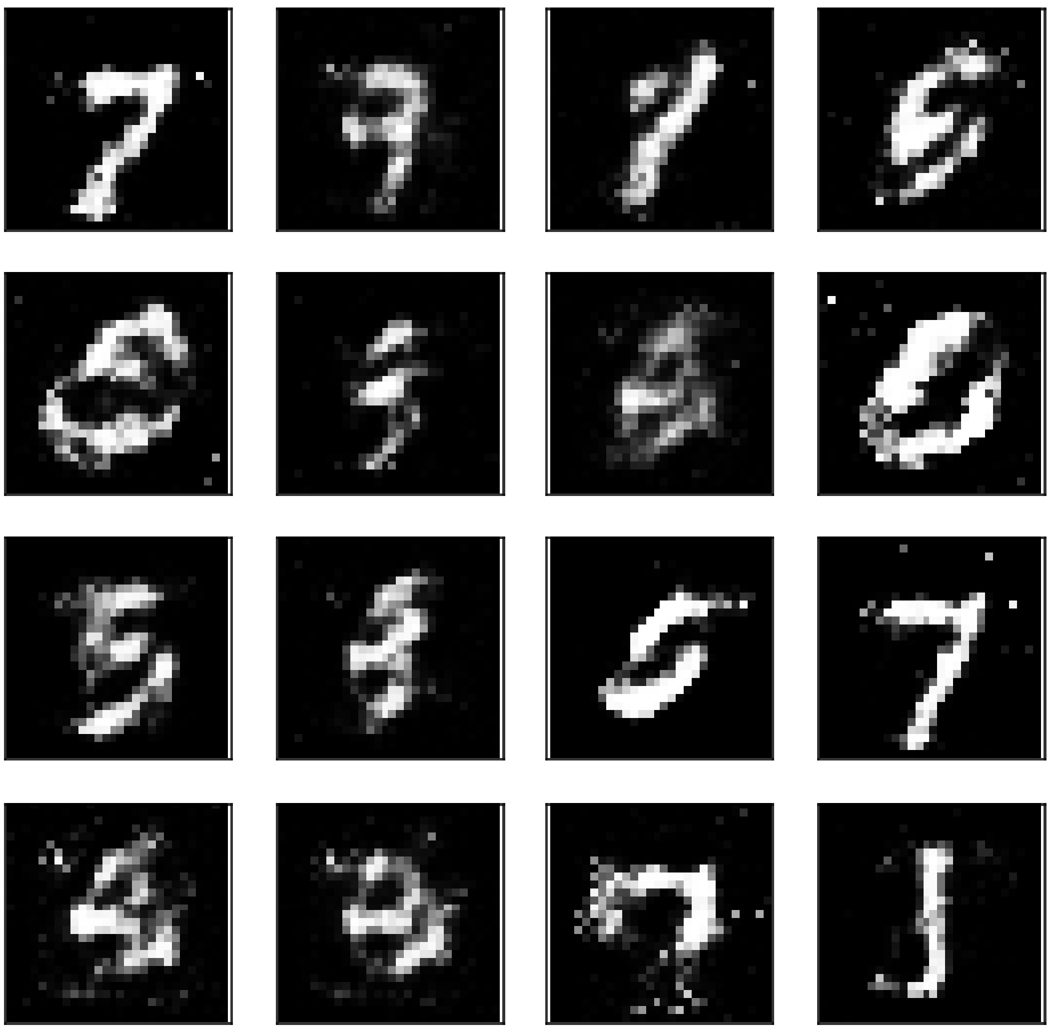
Here we can view samples of images from the generator. First we'll look at the images we saved during training.

```
In [12]: # 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 [13]: # 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.

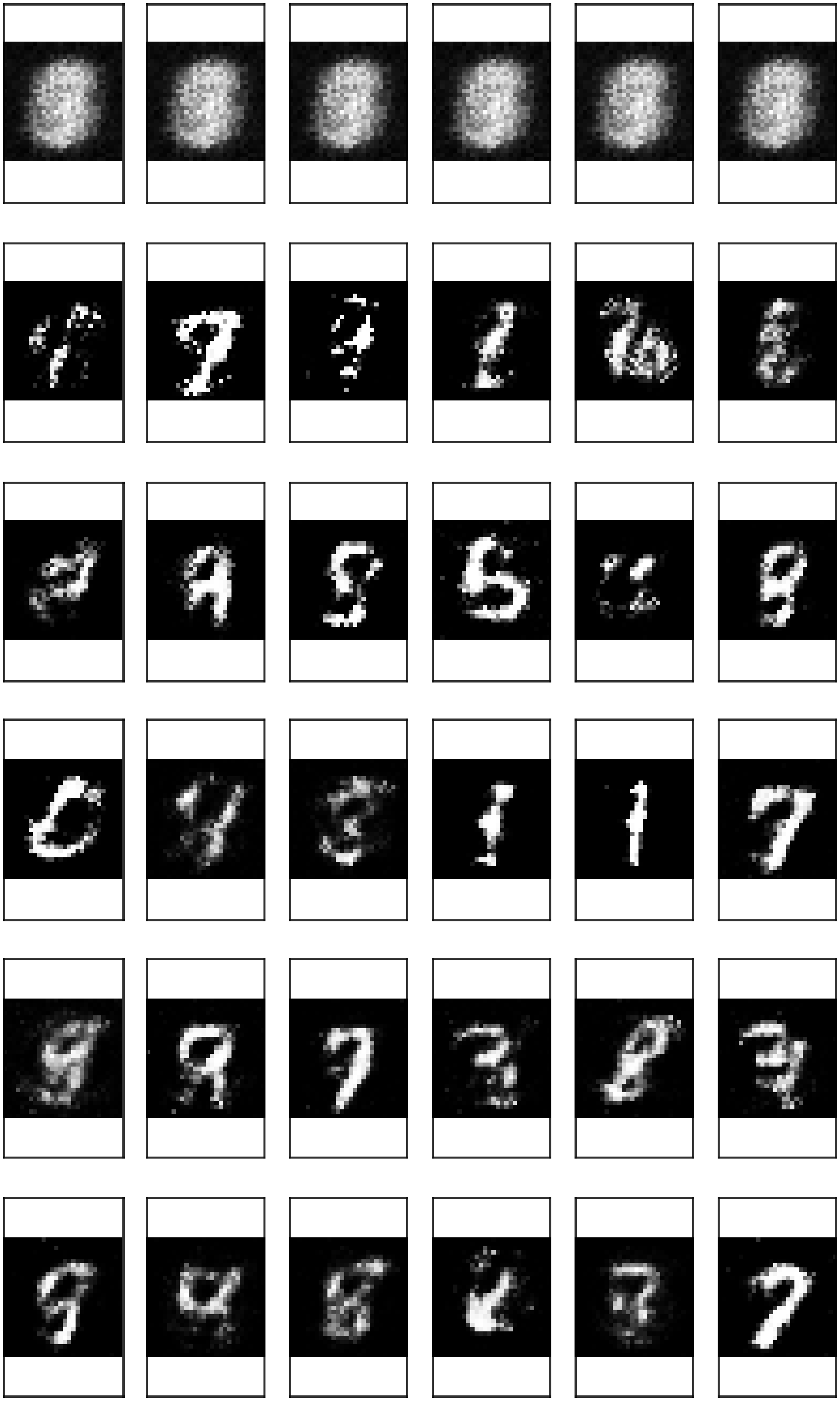
```
In [14]: # -1 indicates final epoch's samples (the last in the list)
         view_samples(-1, samples)
```



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

```
In [15]: rows = 6 # 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
```

```python
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.

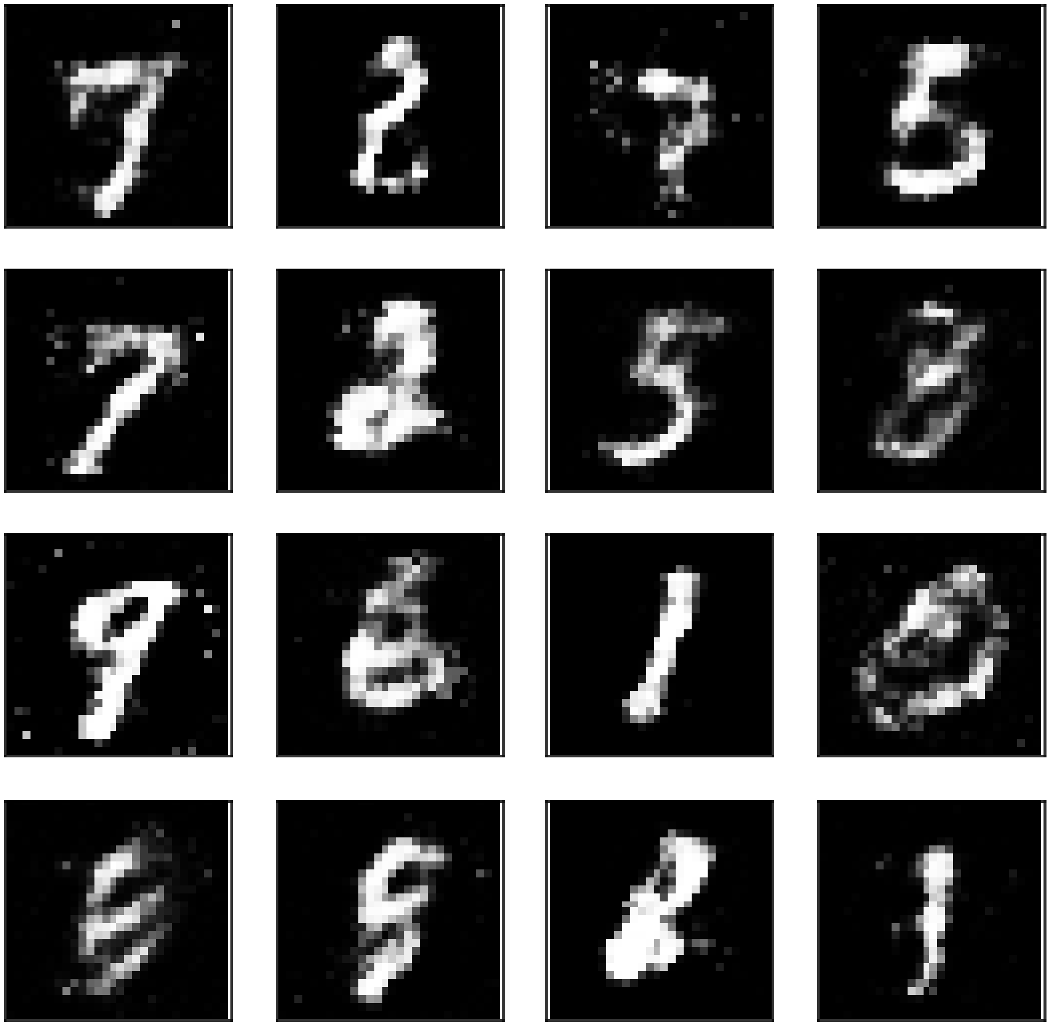
2.10 Sampling from the generator

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 [16]:
```
# 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 [ ]:
```