```python
import numpy as np

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import sampler

import PIL

NOISE_DIM = 96

dtype = torch.cuda.FloatTensor if torch.cuda.is_available() else torch.FloatTensor

def sample_noise(batch_size, dim, seed=None):
    """
    Generate a PyTorch Tensor of uniform random noise.

    Input:
    - batch_size: Integer giving the batch size of noise to generate.
    - dim: Integer giving the dimension of noise to generate.

    Output:
    - A PyTorch Tensor of shape (batch_size, dim) containing uniform
      random noise in the range (-1, 1).
    """
    if seed is not None:
        torch.manual_seed(seed)

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    noise =  torch.rand(batch_size, dim)*2 - 1

    return noise

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

def discriminator(seed=None):
    """
    Build and return a PyTorch model implementing the architecture above.
    """

    if seed is not None:
        torch.manual_seed(seed)
```

```python
    model = None

    ###############################################################################
    # TODO: Implement architecture                                                #
    #                                                                             #
    # HINT: nn.Sequential might be helpful. You'll start by calling Flatten().   #
    ###############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    model = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(256, 256),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(256, 1)
    )

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###############################################################################
    #                                END OF YOUR CODE                             #
    ###############################################################################
    return model

def generator(noise_dim=NOISE_DIM, seed=None):
    """
    Build and return a PyTorch model implementing the architecture above.
    """

    if seed is not None:
        torch.manual_seed(seed)

    model = None

    ###############################################################################
    # TODO: Implement architecture                                                #
    #                                                                             #
    # HINT: nn.Sequential might be helpful.                                       #
    ###############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    model = nn.Sequential(
        nn.Linear(noise_dim, 1024),
        nn.ReLU(),
        nn.Linear(1024, 1024),
```

```python
        nn.ReLU(),
        nn.Linear(1024, 784),
        nn.Tanh()
    )

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return model

def bce_loss(input, target):
    """
    Numerically stable version of the binary cross-entropy loss function in PyTorch.

    Inputs:
    - input: PyTorch Tensor of shape (N, ) giving scores.
    - target: PyTorch Tensor of shape (N,) containing 0 and 1 giving targets.

    Returns:
    - A PyTorch Tensor containing the mean BCE loss over the minibatch of input data.
    """
    bce = nn.BCEWithLogitsLoss()
    return bce(input.squeeze(), target)

def discriminator_loss(logits_real, logits_fake):
    """
    Computes the discriminator loss described above.

    Inputs:
    - logits_real: PyTorch Tensor of shape (N,) giving scores for the real data.
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Returns:
    - loss: PyTorch Tensor containing (scalar) the loss for the discriminator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    # device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    label_one = torch.ones(logits_real.shape[0]).type(dtype)
    label_zero = torch.zeros(logits_fake.shape[0]).type(dtype)
    loss = bce_loss(logits_real, label_one) + bce_loss(logits_fake, label_zero)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss
```

```python
def generator_loss(logits_fake):
    """
    Computes the generator loss described above.

    Inputs:
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Returns:
    - loss: PyTorch Tensor containing the (scalar) loss for the generator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    # Generator wants to fool Discriminator, so the loss is calculated w.r.t label 1
    # More 1 from D, less loss for G
    # device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    label_one = torch.ones(logits_fake.shape[0]).type(dtype)
    loss = bce_loss(logits_fake, label_one)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss

def get_optimizer(model):
    """
    Construct and return an Adam optimizer for the model with learning rate 1e-3,
    beta1=0.5, and beta2=0.999.

    Input:
    - model: A PyTorch model that we want to optimize.

    Returns:
    - An Adam optimizer for the model with the desired hyperparameters.
    """
    optimizer = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    optimizer = optim.Adam(model.parameters(), lr=1e-3, betas=(0.5, 0.999))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return optimizer

def ls_discriminator_loss(scores_real, scores_fake):
    """
    Compute the Least-Squares GAN loss for the discriminator.

    Inputs:
    - scores_real: PyTorch Tensor of shape (N,) giving scores for the real data.
```

```
    - scores_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Outputs:
    - loss: A PyTorch Tensor containing the loss.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    loss = 0.5*torch.mean((scores_real-1)**2) + 0.5*torch.mean((scores_fake)**2)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss

def ls_generator_loss(scores_fake):
    """
    Computes the Least-Squares GAN loss for the generator.

    Inputs:
    - scores_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Outputs:
    - loss: A PyTorch Tensor containing the loss.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    loss = 0.5*torch.mean((scores_fake-1)**2)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss

def build_dc_classifier(batch_size):
    """
    Build and return a PyTorch model for the DCGAN discriminator implementing
    the architecture above.
    """

    ############################################################################
    # TODO: Implement architecture                                             #
    #                                                                          #
    # HINT: nn.Sequential might be helpful.                                    #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    d_model = nn.Sequential(nn.Conv2d(1, 32, 5, 1),
                            nn.LeakyReLU(negative_slope=0.01),
```

```python
                      nn.MaxPool2d(2,2),
                      nn.Conv2d(32, 64, 5, 1),
                      nn.LeakyReLU(negative_slope=0.01),
                      nn.MaxPool2d(2,2),
                      nn.Flatten(),
                      nn.Linear(1024, 1024),
                      nn.LeakyReLU(negative_slope=0.01),
                      nn.Linear(1024, 1))

    return d_model

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################


def build_dc_generator(noise_dim=NOISE_DIM):
    """
    Build and return a PyTorch model implementing the DCGAN generator using
    the architecture described above.
    """

    ##############################################################################
    # TODO: Implement architecture                                               #
    #                                                                            #
    # HINT: nn.Sequential might be helpful.                                      #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # class GenNet(nn.Module):
    #     def __init__(self):
    #         super(GenNet, self).__init__()
    #         layers=[]
    #         layers.append(nn.Linear(NOISE_DIM, 1024))
    #         layers.append(nn.ReLU())
    #         layers.append(nn.BatchNorm1d(1024))
    #         layers.append(nn.Linear(1024, 7*7*128))
    #         layers.append(nn.ReLU())
    #         layers.append(nn.BatchNorm1d(7*7*128))
    #         layers.append(nn.Unflatten(1, [128,7,7]))
    #         layers.append(nn.ConvTranspose2d(128, 64, 4, 2, 1))
    #         layers.append(nn.ReLU())
    #         layers.append(nn.BatchNorm2d(64))
    #         layers.append(nn.ConvTranspose2d(64, 1, 4, 2, 1))
```

```
    #         layers.append(nn.Tanh())
    #         layers.append(nn.Flatten())

    #         self.model = nn.ModuleList(layers)

    #     def forward(self, x):

    #         for layer in self.model:
    #             x = layer(x)
    #             print(layer)
    #             print(x.shape)

    #         return x

    # return GenNet()

    g_model = nn.Sequential(nn.Linear(noise_dim, 1024),
                          nn.ReLU(),
                          nn.BatchNorm1d(1024),
                          nn.Linear(1024, 7*7*128),
                          nn.ReLU(),
                          nn.BatchNorm1d(7*7*128),
                          nn.Unflatten(1, [128,7,7]),
                          nn.ConvTranspose2d(in_channels=128, out_channels=64, \
                                             kernel_size=4, stride=2, padding=1),
                          nn.ReLU(),
                          nn.BatchNorm2d(64),
                          nn.ConvTranspose2d(in_channels=64, out_channels=1, \
                                             kernel_size=4, stride=2, padding=1),
                          nn.Tanh(),
                          nn.Flatten())

    return g_model

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################

def run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss, loader_train, sh
              batch_size=128, noise_size=96, num_epochs=10):
    """
    Train a GAN!

    Inputs:
    - D, G: PyTorch models for the discriminator and generator
```

```
- D_solver, G_solver: torch.optim Optimizers to use for training the
  discriminator and generator.
- discriminator_loss, generator_loss: Functions to use for computing the generator and
  discriminator loss, respectively.
- show_every: Show samples after every show_every iterations.
- batch_size: Batch size to use for training.
- noise_size: Dimension of the noise to use as input to the generator.
- num_epochs: Number of epochs over the training dataset to use for training.
"""

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

images = []
iter_count = 0
for epoch in range(num_epochs):
    for x, _ in loader_train:
        if len(x) != batch_size:
            continue
        D_solver.zero_grad()
        real_data = x.type(dtype)
        logits_real = D(2* (real_data - 0.5)).type(dtype)

        g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
        fake_images = G(g_fake_seed).detach()
        logits_fake = D(fake_images.view(batch_size, 1, 28, 28))

        d_total_error = discriminator_loss(logits_real, logits_fake)
        d_total_error.backward()
        D_solver.step()

        G_solver.zero_grad()
        g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
        fake_images = G(g_fake_seed)

        gen_logits_fake = D(fake_images.view(batch_size, 1, 28, 28))
        g_error = generator_loss(gen_logits_fake)
        g_error.backward()
        G_solver.step()

        if (iter_count % show_every == 0):
            print('Iter: {}, D: {:.4}, G:{:.4}'.format(iter_count,d_total_error.item(),g
            imgs_numpy = fake_images.data.cpu().numpy()
            images.append(imgs_numpy[0:16])

        iter_count += 1
```

```
    return images


class ChunkSampler(sampler.Sampler):
    """Samples elements sequentially from some offset.
    Arguments:
        num_samples: # of desired datapoints
        start: offset where we should start selecting from
    """
    def __init__(self, num_samples, start=0):
        self.num_samples = num_samples
        self.start = start

    def __iter__(self):
        return iter(range(self.start, self.start + self.num_samples))

    def __len__(self):
        return self.num_samples


class Flatten(nn.Module):
    def forward(self, x):
        N, C, H, W = x.size() # read in N, C, H, W
        return x.view(N, -1)  # "flatten" the C * H * W values into a single vector per imag

class Unflatten(nn.Module):
    """
    An Unflatten module receives an input of shape (N, C*H*W) and reshapes it
    to produce an output of shape (N, C, H, W).
    """
    def __init__(self, N=-1, C=128, H=7, W=7):
        super(Unflatten, self).__init__()
        self.N = N
        self.C = C
        self.H = H
        self.W = W
    def forward(self, x):
        return x.view(self.N, self.C, self.H, self.W)

def initialize_weights(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.ConvTranspose2d):
        nn.init.xavier_uniform_(m.weight.data)

def preprocess_img(x):
    return 2 * x - 1.0
```

```python
def deprocess_img(x):
    return (x + 1.0) / 2.0

def rel_error(x,y):
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def count_params(model):
    """Count the number of parameters in the model. """
    param_count = np.sum([np.prod(p.size()) for p in model.parameters()])
    return param_count
```

gan

April 15, 2024

We would like to acknowledge Stanford University's CS231n on which we based the development of this Jupyter Notebook.

[ ]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# project folder, e.g. '239AS.3/project1/gan'
FOLDERNAME = None
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

%cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Generative Adversarial Networks (GANs)

In C147/C247, all the applications of neural networks that we have explored have been **discriminative models** that take an input and are trained to produce a labeled output. In this notebook, we will expand our repetoire, and build **generative models** using neural networks. Specifically, we will learn how to build models which generate novel images that resemble a set of training images.

1.0.1 What is a GAN?

In 2014, Goodfellow et al. presented a method for training generative models called Generative Adversarial Networks (GANs for short). In a GAN, we build two different neural networks. Our first network is a traditional classification network, called the **discriminator**. We will train the discriminator to take images and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the **generator**, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

We can think of this back and forth process of the generator (G) trying to fool the discriminator (D) and the discriminator trying to correctly classify real vs. fake as a minimax game:

$$\underset{G}{\text{minimize}}\;\underset{D}{\text{maximize}}\;\mathbb{E}_{x\sim p_\text{data}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log\left(1-D(G(z))\right)\right]$$

where $z\sim p(z)$ are the random noise samples, $G(z)$ are the generated images using the neural network generator G, and D is the output of the discriminator, specifying the probability of an input being real. In Goodfellow et al., they analyze this minimax game and show how it relates to minimizing the Jensen-Shannon divergence between the training data distribution and the generated samples from G.

To optimize this minimax game, we will alternate between taking gradient *descent* steps on the objective for G and gradient *ascent* steps on the objective for D: 1. update the **generator** (G) to minimize the probability of the **discriminator making the correct choice**. 2. update the **discriminator** (D) to maximize the probability of the **discriminator making the correct choice**.

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator gradient vanishing when the discriminator is confident. This is the standard update used in most GAN papers and was used in the original paper from Goodfellow et al..

In this assignment, we will alternate the following updates: 1. Update the generator (G) to maximize the probability of the discriminator making the incorrect choice on generated data:

$$\underset{G}{\text{maximize}}\;\mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

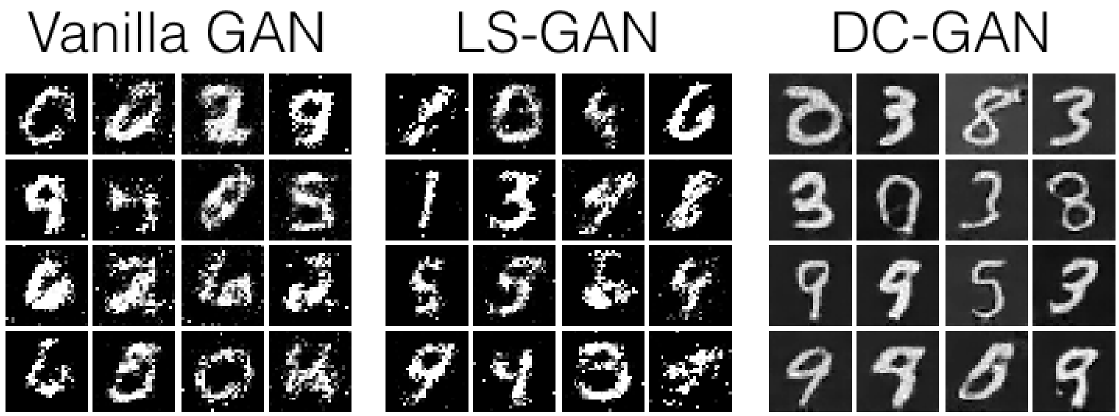
2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:

$$\underset{D}{\text{maximize}}\;\mathbb{E}_{x\sim p_\text{data}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log\left(1-D(G(z))\right)\right]$$

Here's an example of what your outputs from the 3 different models you're going to train should look like. Note that GANs are sometimes finicky, so your outputs might not look exactly like this. This is just meant to be a *rough* guideline of the kind of quality you can expect:

[ ]:
```
# Run this cell to see sample outputs.
from IPython.display import Image
Image('nndl2/gan_outputs.png')
```

[ ]:



[ ]:
```
# Setup cell.
import numpy as np
import torch
import torch.nn as nn
from torch.nn import init
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
import torchvision.datasets as dset
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from gan import preprocess_img, deprocess_img, rel_error, count_params,␣
↪ChunkSampler

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # Set default size of plots.
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

def show_images(images):
    images = np.reshape(images, [images.shape[0], -1]) # Images reshape to␣
↪(batch_size, D).
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))
    sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))

    fig = plt.figure(figsize=(sqrtn, sqrtn))
    gs = gridspec.GridSpec(sqrtn, sqrtn)
```

```
    gs.update(wspace=0.05, hspace=0.05)

    for i, img in enumerate(images):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(img.reshape([sqrtimg,sqrtimg]))
    return

answers = dict(np.load('nndl2/gan-checks.npz'))
dtype = torch.cuda.FloatTensor if torch.cuda.is_available() else torch.
 ↪FloatTensor
```

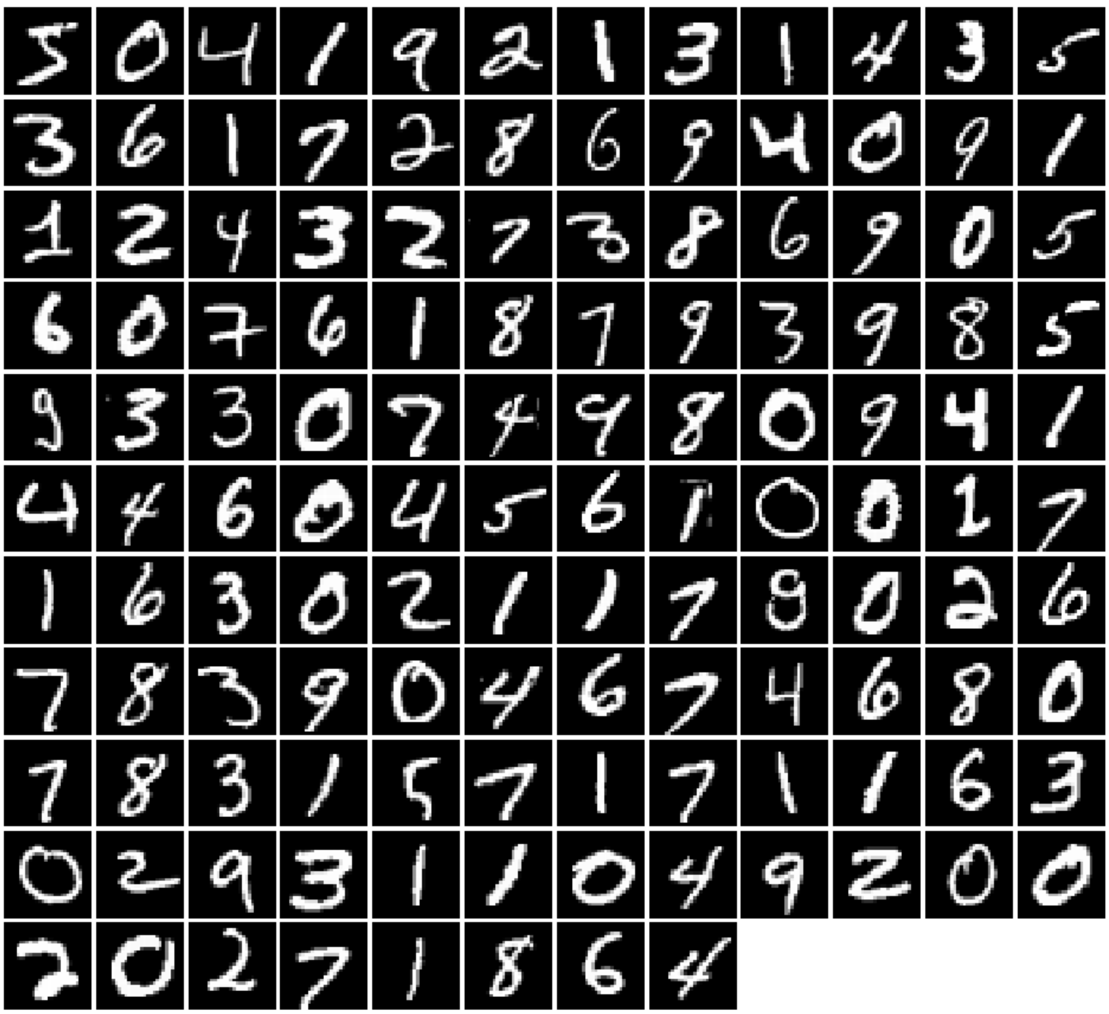
1.1 Dataset

GANs are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable, we will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each picture contains a centered image of white digit on black background (0 through 9). This was one of the first datasets used to train convolutional neural networks and it is fairly easy – a standard CNN model can easily exceed 99% accuracy.

To simplify our code here, we will use the PyTorch MNIST wrapper, which downloads and loads the MNIST dataset. See the documentation for more information about the interface. The default parameters will take 5,000 of the training examples and place them into a validation dataset. The data will be saved into a folder called `MNIST`.

[ ]:
```
NUM_TRAIN = 50000
NUM_VAL = 5000

NOISE_DIM = 96
batch_size = 128

mnist_train = dset.MNIST(
    './nndl2',
    train=True,
    download=True,
    transform=T.ToTensor()
)
loader_train = DataLoader(
    mnist_train,
    batch_size=batch_size,
    sampler=ChunkSampler(NUM_TRAIN, 0)
)

mnist_val = dset.MNIST(
    './nndl2',
```

```
    train=True,
    download=True,
    transform=T.ToTensor()
)
loader_val = DataLoader(
    mnist_val,
    batch_size=batch_size,
    sampler=ChunkSampler(NUM_VAL, NUM_TRAIN)
)

iterator = iter(loader_train)
imgs, labels = next(iterator)
imgs = imgs.view(batch_size, 784).numpy().squeeze()
show_images(imgs)
```



1.2 Random Noise (1 point)

Generate uniform noise from -1 to 1 with shape `[batch_size, dim]`.

Implement `sample_noise` in `gan.py`.

Hint: use `torch.rand`.

Make sure noise is the correct shape and type:

[ ]:
```
from gan import sample_noise

def test_sample_noise():
    batch_size = 3
    dim = 4
    torch.manual_seed(231)
    z = sample_noise(batch_size, dim)
    np_z = z.cpu().numpy()
    assert np_z.shape == (batch_size, dim)
    assert torch.is_tensor(z)
    assert np.all(np_z >= -1.0) and np.all(np_z <= 1.0)
    assert np.any(np_z < 0.0) and np.any(np_z > 0.0)
    print('All tests passed!')

test_sample_noise()
```

```
All tests passed!
```

1.3 Flatten

We provide an Unflatten, which you might want to use when implementing the convolutional generator. We also provide a weight initializer (and call it for you) that uses Xavier initialization instead of PyTorch's uniform default.

[ ]:
```
from gan import Flatten, Unflatten, initialize_weights
```

2 Discriminator (1 point)

Our first step is to build a discriminator. Fill in the architecture as part of the `nn.Sequential` constructor in the function below. All fully connected layers should include bias terms. The architecture is: * Fully connected layer with input size 784 and output size 256 * LeakyReLU with alpha 0.01 * Fully connected layer with input_size 256 and output size 256 * LeakyReLU with alpha 0.01 * Fully connected layer with input size 256 and output size 1

Recall that the Leaky ReLU nonlinearity computes $f(x) = \max(\alpha x, x)$ for some fixed constant α; for the LeakyReLU nonlinearities in the architecture above we set $\alpha = 0.01$.

The output of the discriminator should have shape `[batch_size, 1]`, and contain real numbers corresponding to the scores that each of the `batch_size` inputs is a real image.

Implement `discriminator` in `gan.py`

Test to make sure the number of parameters in the discriminator is correct:

```
[ ]: from gan import discriminator

def test_discriminator(true_count=267009):
    model = discriminator()
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in discriminator. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in discriminator.')

test_discriminator()
```

```
Correct number of parameters in discriminator.
```

3 Generator (1 point)

Now to build the generator network: * Fully connected layer from noise_dim to 1024 * `ReLU` * Fully connected layer with size 1024 * `ReLU` * Fully connected layer with size 784 * `TanH` (to clip the image to be in the range of [-1,1])

Implement `generator` in `gan.py`

Test to make sure the number of parameters in the generator is correct:

```
[ ]: from gan import generator

def test_generator(true_count=1858320):
    model = generator(4)
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in generator. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in generator.')

test_generator()
```

```
Correct number of parameters in generator.
```

4 GAN Loss (2 points)

Compute the generator and discriminator loss. The generator loss is:

$$\ell_G = -\mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

and the discriminator loss is:

$$\ell_D = -\mathbb{E}_{x\sim p_\text{data}}\left[\log D(x)\right] - \mathbb{E}_{z\sim p(z)}\left[\log\left(1-D(G(z))\right)\right]$$

Note that these are negated from the equations presented earlier as we will be *minimizing* these losses.

HINTS: You should use the `bce_loss` function defined below to compute the binary cross entropy loss which is needed to compute the log probability of the true label given the logits output from the discriminator. Given a score $s \in \mathbb{R}$ and a label $y \in \{0, 1\}$, the binary cross entropy loss is

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

A naive implementation of this formula can be numerically unstable, so we have provided a numerically stable implementation that relies on PyTorch's `nn.BCEWithLogitsLoss`.

You will also need to compute labels corresponding to real or fake and use the logit arguments to determine their size. Make sure you cast these labels to the correct data type using the global `dtype` variable, for example:

`true_labels = torch.ones(size).type(dtype)`

Instead of computing the expectation of $\log D(G(z))$, $\log D(x)$ and $\log (1 - D(G(z)))$, we will be averaging over elements of the minibatch. This is taken care of in `bce_loss` which combines the loss by averaging.

Implement `discriminator_loss` and `generator_loss` in `gan.py`

Test your generator and discriminator loss. You should see errors < 1e-7.

[ ]:
```
from gan import bce_loss, discriminator_loss, generator_loss

def test_discriminator_loss(logits_real, logits_fake, d_loss_true):
    d_loss = discriminator_loss(torch.Tensor(logits_real).type(dtype),
                                torch.Tensor(logits_fake).type(dtype)).cpu().
 ↪numpy()
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))

test_discriminator_loss(
    answers['logits_real'],
    answers['logits_fake'],
    answers['d_loss_true']
)
```

```
Maximum error in d_loss: 3.97058e-09
```

[ ]:
```
def test_generator_loss(logits_fake, g_loss_true):
    g_loss = generator_loss(torch.Tensor(logits_fake).type(dtype)).cpu().numpy()
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_generator_loss(
    answers['logits_fake'],
    answers['g_loss_true']
)
```

```
Maximum error in g_loss: 4.4518e-09
```

5 Optimizing Our Loss (1 point)

Make a function that returns an `optim.Adam` optimizer for the given model with a 1e-3 learning rate, beta1=0.5, beta2=0.999. You'll use this to construct optimizers for the generators and discriminators for the rest of the notebook.

Implement `get_optimizer` in `gan.py`

6 Training a GAN!

We provide you the main training loop. You won't need to change `run_a_gan` in `gan.py`, but we encourage you to read through it for your own understanding. If you train with the CPU, it takes about 7 minutes. If you train with the T4 GPU, it takes about 1 minute and 30 seconds.

[ ]:
```python
from gan import get_optimizer, run_a_gan
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')



# Make the discriminator
D = discriminator().type(dtype)
D.to(device)

# Make the generator
G = generator().type(dtype)
G.to(device)

print(device)

# Use the function you wrote earlier to get optimizers for the Discriminator
 ↪and the Generator
D_solver = get_optimizer(D)
G_solver = get_optimizer(G)

# Run it!
images = run_a_gan(
    D,
    G,
    D_solver,
    G_solver,
    discriminator_loss,
    generator_loss,
    loader_train
)
```

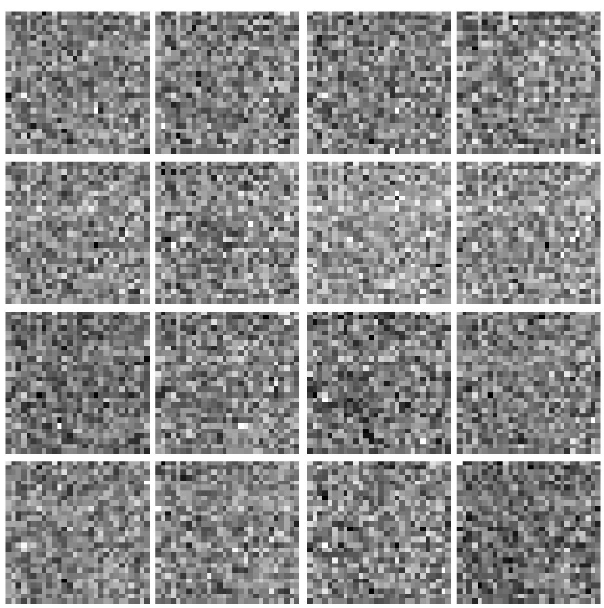
```
cuda:0
Iter: 0, D: 1.328, G:0.7202
```

```
Iter: 250, D: 1.079, G:0.8139
Iter: 500, D: 1.152, G:1.114
Iter: 750, D: 1.121, G:1.194
Iter: 1000, D: 1.197, G:1.019
Iter: 1250, D: 1.344, G:1.546
Iter: 1500, D: 1.171, G:0.8669
Iter: 1750, D: 1.243, G:0.9491
Iter: 2000, D: 1.289, G:0.8889
Iter: 2250, D: 1.263, G:0.8899
Iter: 2500, D: 1.323, G:0.859
Iter: 2750, D: 1.337, G:0.8549
Iter: 3000, D: 1.381, G:0.8909
Iter: 3250, D: 1.334, G:0.7981
Iter: 3500, D: 1.331, G:0.829
Iter: 3750, D: 1.356, G:0.8002
```

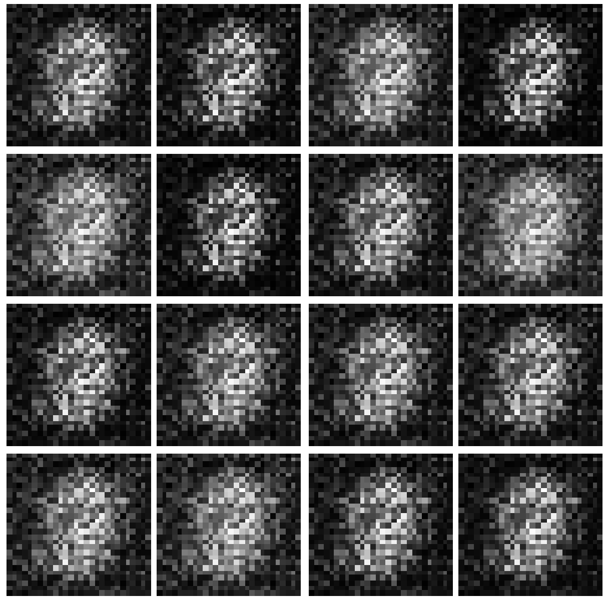
Run the cell below to show the generated images.

[ ]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

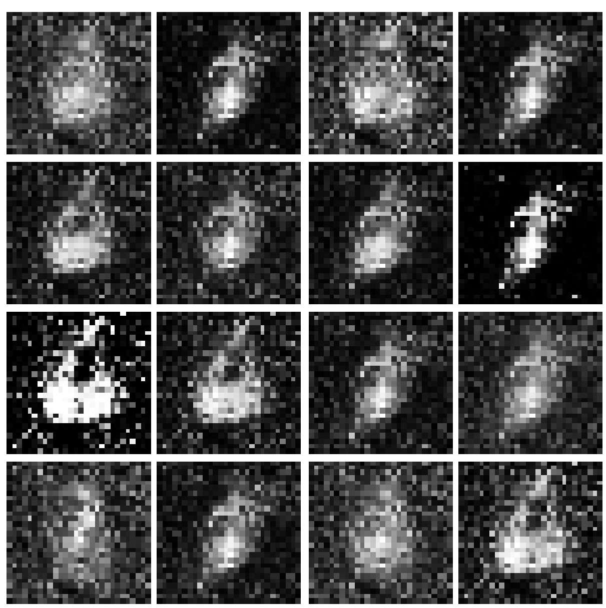
```
Iter: 0
```



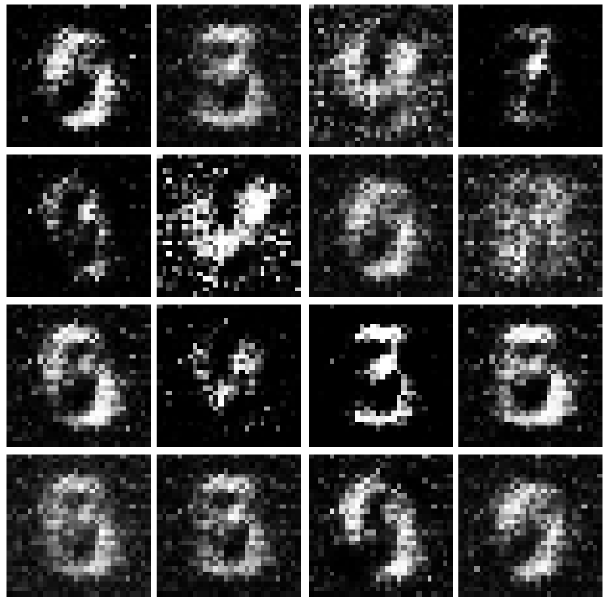
Iter: 250



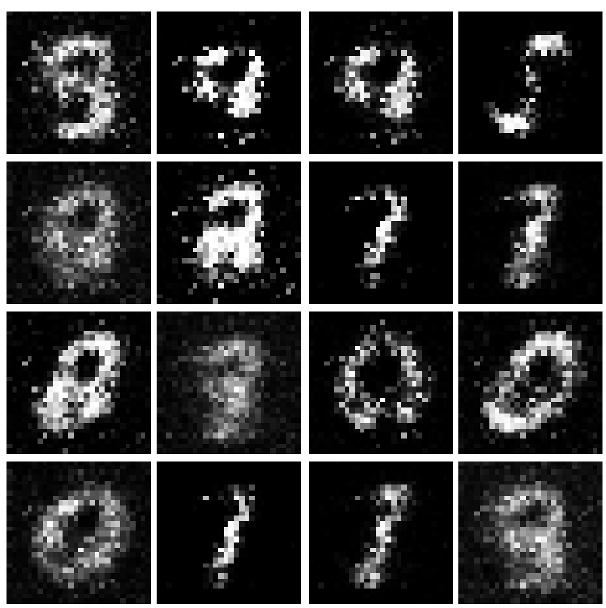
Iter: 500



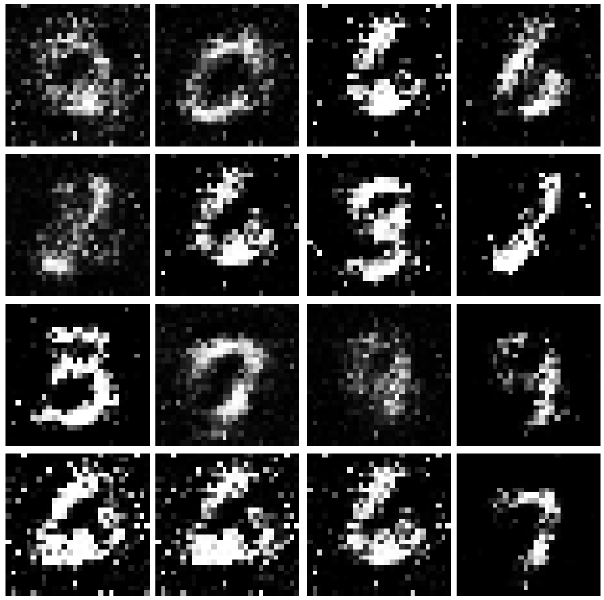
Iter: 750



Iter: 1000

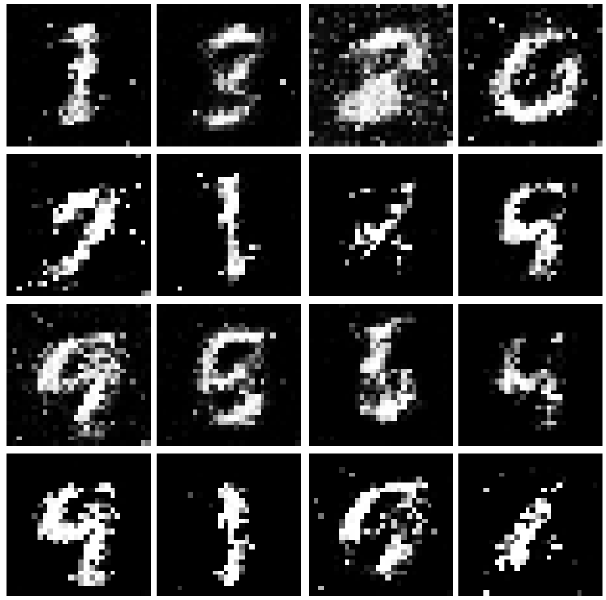


Iter: 1250

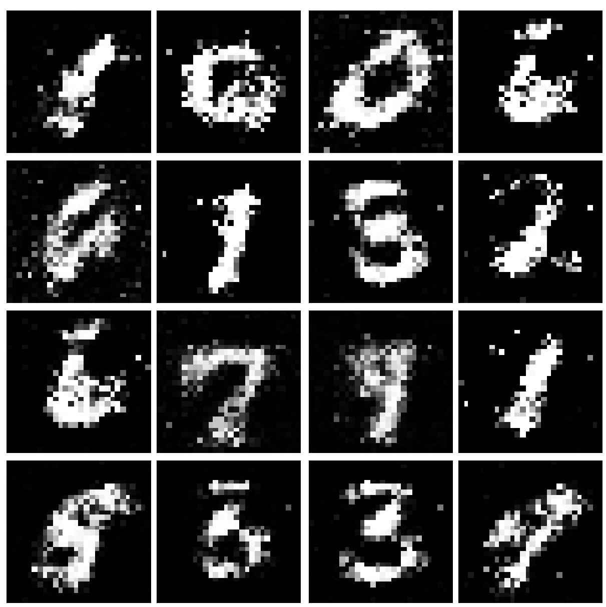


Iter: 1500

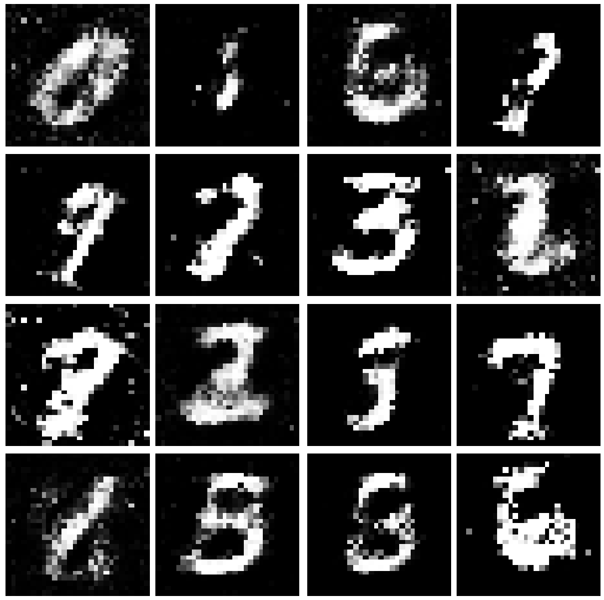
Iter: 1750



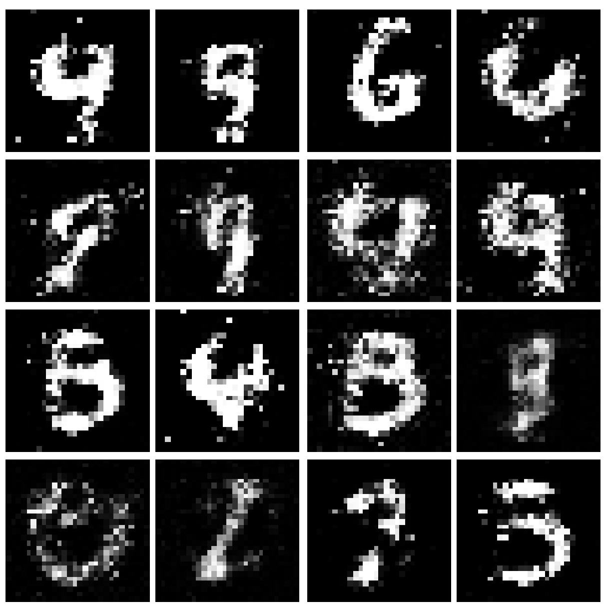
Iter: 2000



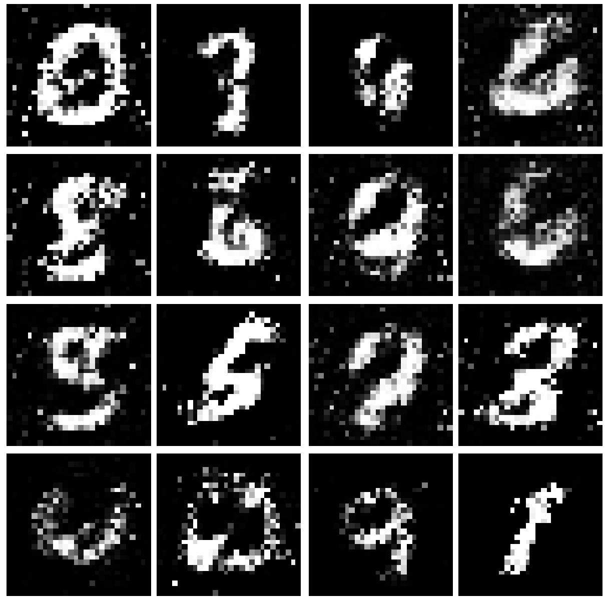
Iter: 2250



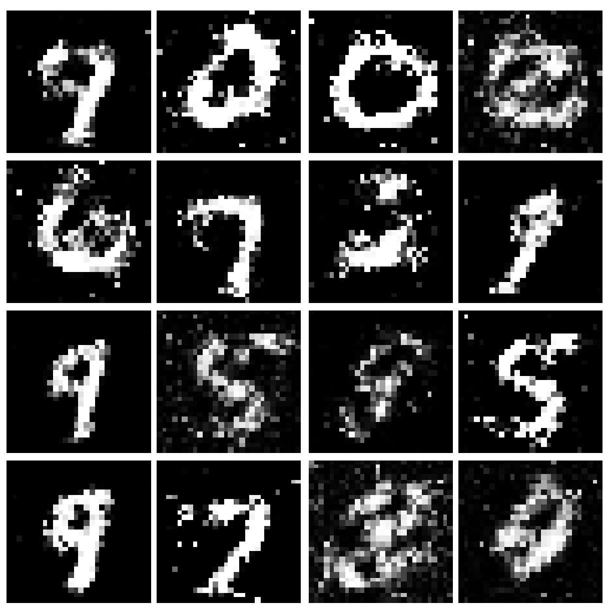
Iter: 2500



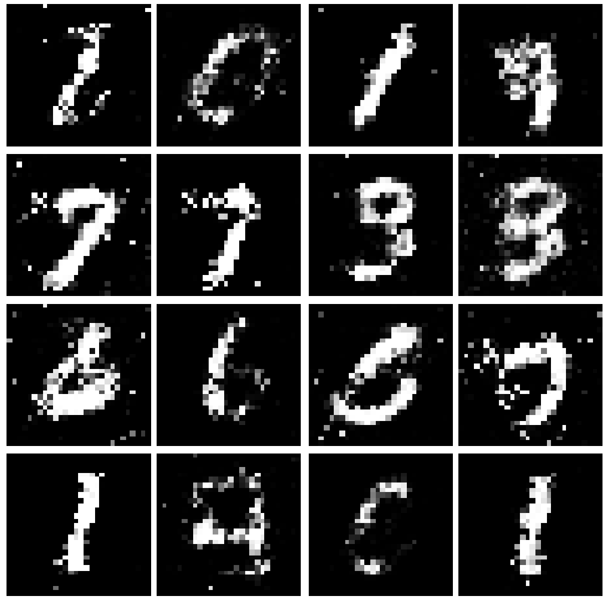
Iter: 2750



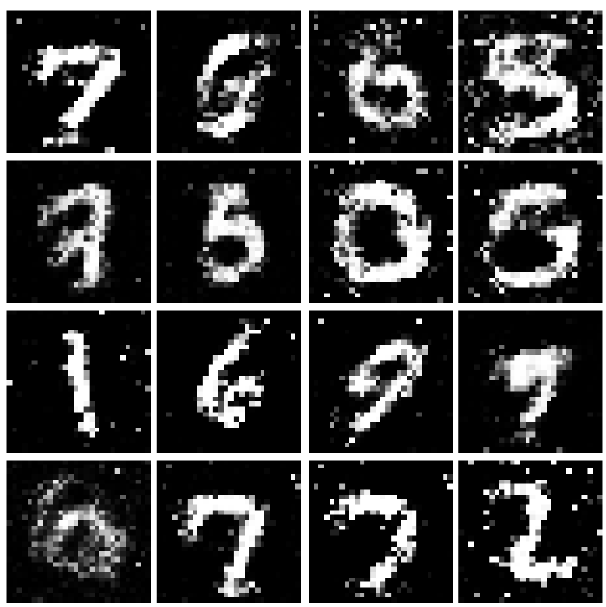
Iter: 3000



Iter: 3250



Iter: 3500



Iter: 3750

6.1 Inline Question 1

What does your final vanilla GAN image look like?

[ ]:
```
# This output is your answer.
print("Vanilla GAN final image:")
show_images(images[-1])
plt.show()
```

```
Vanilla GAN final image:
```

Well that wasn't so hard, was it? In the iterations in the low 100s you should see black backgrounds, fuzzy shapes as you approach iteration 1000, and decent shapes, about half of which will be sharp and clearly recognizable as we pass 3000.

7 Least Squares GAN (2 points)

We'll now look at Least Squares GAN, a newer, more stable alernative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement equation (9) in the paper, with the generator loss:

$$\ell_G = \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[\left(D(G(z))-1\right)^2\right]$$

and the discriminator loss:

$$\ell_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[\left(D(x)-1\right)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[\left(D(G(z))\right)^2\right]$$

HINTS: Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. When plugging in for $D(x)$ and $D(G(z))$ use the direct output from the discriminator (`scores_real` and `scores_fake`).

Implement `ls_discriminator_loss`, `ls_generator_loss` in `gan.py`

Before running a GAN with our new loss function, let's check it:

[ ]:
```
from gan import ls_discriminator_loss, ls_generator_loss

def test_lsgan_loss(score_real, score_fake, d_loss_true, g_loss_true):
    score_real = torch.Tensor(score_real).type(dtype)
```

```
    score_fake = torch.Tensor(score_fake).type(dtype)
    d_loss = ls_discriminator_loss(score_real, score_fake).cpu().numpy()
    g_loss = ls_generator_loss(score_fake).cpu().numpy()
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_lsgan_loss(
    answers['logits_real'],
    answers['logits_fake'],
    answers['d_loss_lsgan_true'],
    answers['g_loss_lsgan_true']
)
```

```
Maximum error in d_loss: 1.53171e-08
Maximum error in g_loss: 2.7837e-09
```

Run the following cell to train your model! If you train with the CPU, it takes about 7 minutes. If you train with the T4 GPU, it takes about 1 minute and 30 seconds.

[ ]:
```
D_LS = discriminator().type(dtype)
G_LS = generator().type(dtype)

D_LS_solver = get_optimizer(D_LS)
G_LS_solver = get_optimizer(G_LS)

images = run_a_gan(
    D_LS,
    G_LS,
    D_LS_solver,
    G_LS_solver,
    ls_discriminator_loss,
    ls_generator_loss,
    loader_train
)
```

```
Iter: 0, D: 0.5689, G:0.51
Iter: 250, D: 0.09176, G:0.4335
Iter: 500, D: 0.19, G:0.3312
Iter: 750, D: 0.104, G:0.4333
Iter: 1000, D: 0.1734, G:0.5379
Iter: 1250, D: 0.125, G:0.3622
Iter: 1500, D: 0.1723, G:0.227
Iter: 1750, D: 0.174, G:0.2973
Iter: 2000, D: 0.2118, G:0.1722
Iter: 2250, D: 0.2111, G:0.1855
Iter: 2500, D: 0.209, G:0.1902
Iter: 2750, D: 0.2307, G:0.1276
Iter: 3000, D: 0.2272, G:0.1803
Iter: 3250, D: 0.2191, G:0.177
```

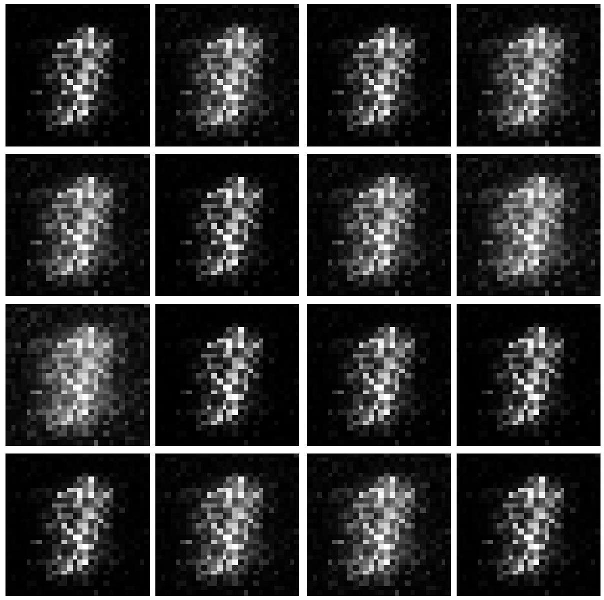
Iter: 3500, D: 0.2395, G:0.1628
Iter: 3750, D: 0.2322, G:0.1787

Run the cell below to show generated images.

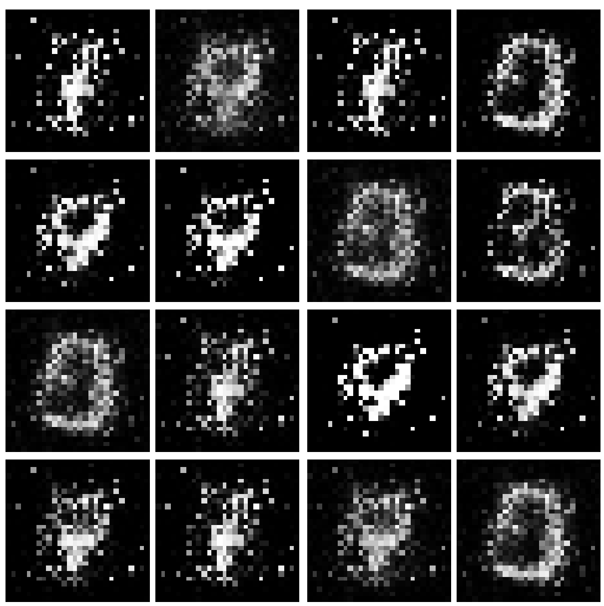
[ ]:
```python
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

Iter: 0

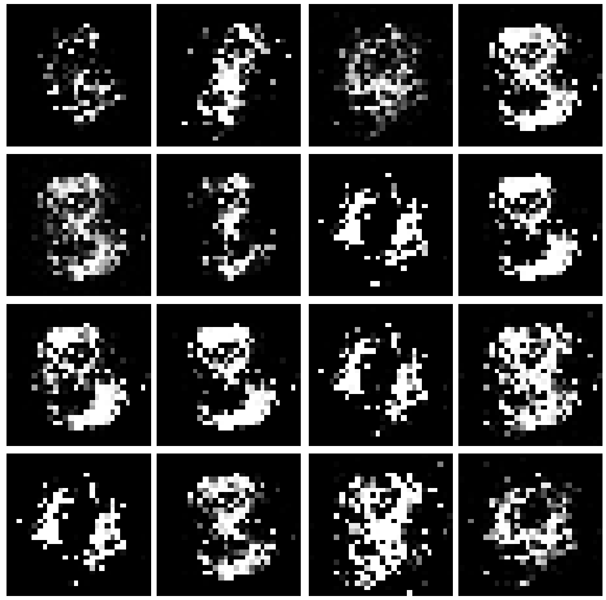
Iter: 250



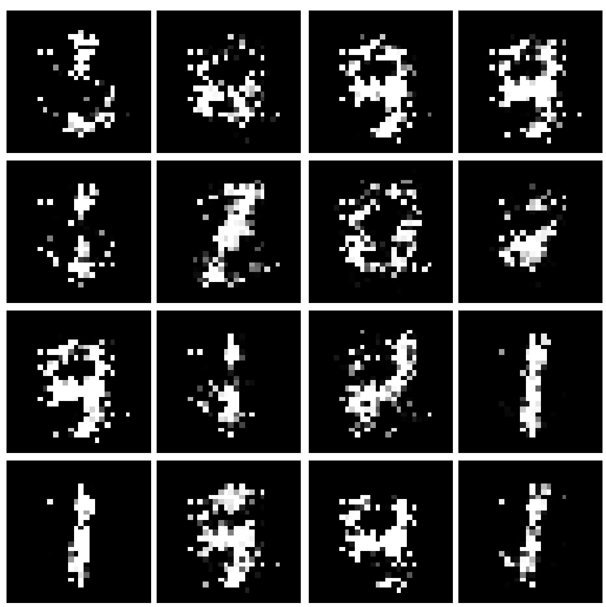
Iter: 500



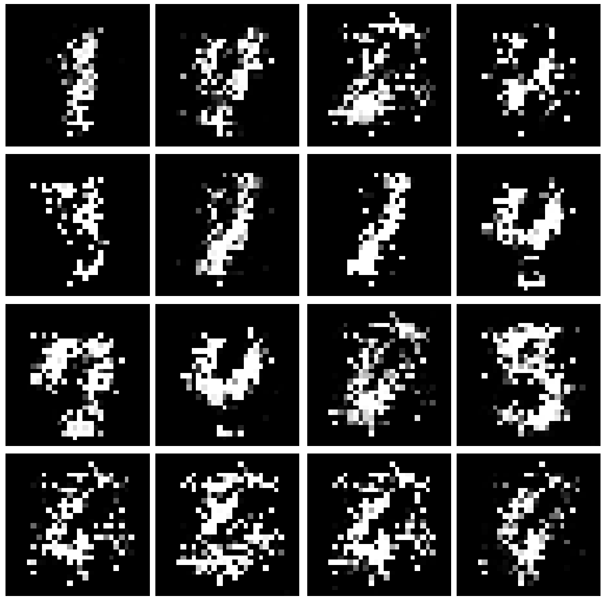
Iter: 750



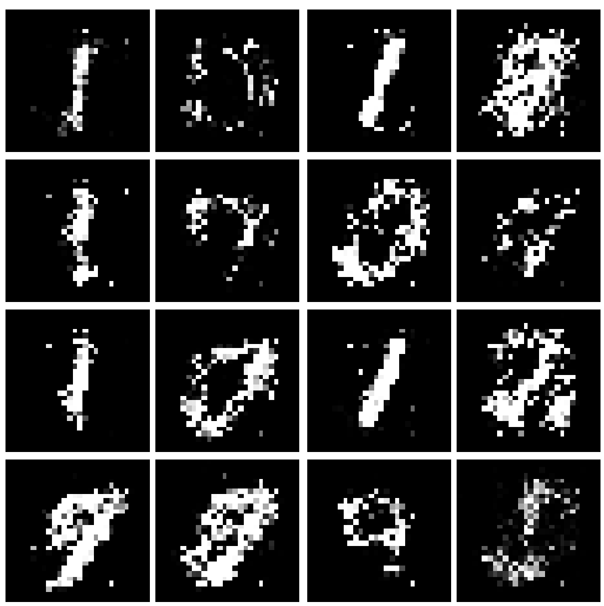
Iter: 1000



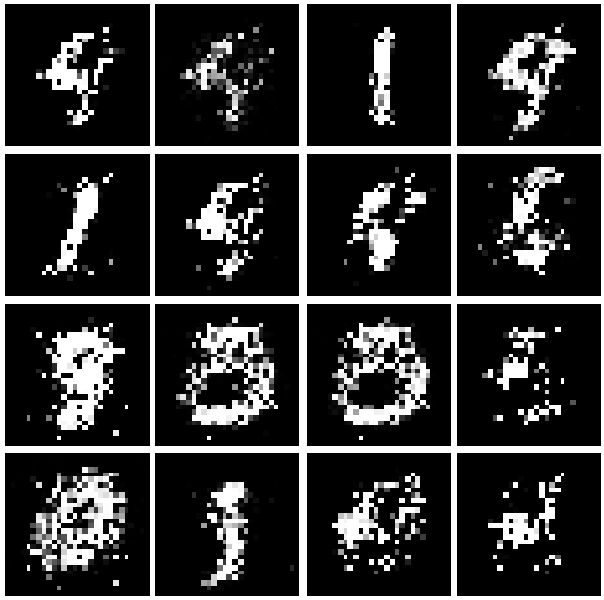
Iter: 1250



Iter: 1500



Iter: 1750



Iter: 2000

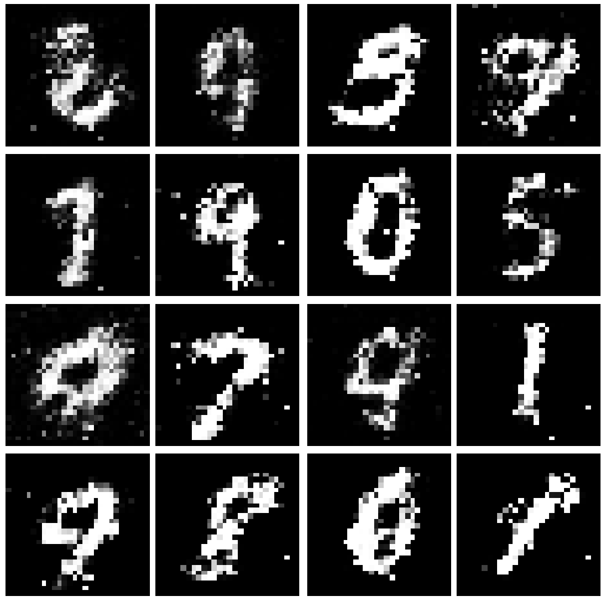
Iter: 2250

Iter: 2500

Iter: 2750

Iter: 3000

Iter: 3250



```
Iter: 3500
```

```
Iter: 3750
```

7.1 Inline Question 2

What does your final LSGAN image look like?

[ ]:
```python
# This output is your answer.
print("LSGAN final image:")
show_images(images[-1])
plt.show()
```

```
LSGAN final image:
```

8 Deeply Convolutional GANs (2 points)

In the first part of the notebook, we implemented an almost direct copy of the original GAN network from Ian Goodfellow. However, this network architecture allows no real spatial reasoning. It is unable to reason about things like “sharp edges” in general because it lacks any convolutional layers. Thus, in this section, we will implement some of the ideas from DCGAN, where we use convolutional networks

Discriminator We will use a discriminator inspired by the TensorFlow MNIST classification tutorial, which is able to get above 99% accuracy on the MNIST dataset fairly quickly. * Conv2D: 32 Filters, 5x5, Stride 1 * Leaky ReLU(alpha=0.01) * Max Pool 2x2, Stride 2 * Conv2D: 64 Filters, 5x5, Stride 1 * Leaky ReLU(alpha=0.01) * Max Pool 2x2, Stride 2 * Flatten * Fully Connected with output size 4 x 4 x 64 * Leaky ReLU(alpha=0.01) * Fully Connected with output size 1

Implement `build_dc_classifier` in `gan.py`

[ ]:
```
from gan import build_dc_classifier

data = next(enumerate(loader_train))[-1][0].type(dtype)
b = build_dc_classifier(batch_size).type(dtype)
out = b(data)
print(out.size())
```

```
torch.Size([128, 1])
```

Check the number of parameters in your classifier as a sanity check:

[ ]:
```
def test_dc_classifer(true_count=1102721):
    model = build_dc_classifier(batch_size)
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in classifier. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in classifier.')

test_dc_classifer()
```

```
Correct number of parameters in classifier.
```

Generator For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. See the documentation for nn.ConvTranspose2d. We are always "training" in GAN mode. * Fully connected with output size 1024 * `ReLU` * BatchNorm * Fully connected with output size 7 x 7 x 128 * ReLU * BatchNorm * Use `Unflatten()` to reshape into Image Tensor of shape 7, 7, 128 * ConvTranspose2d: 64 filters of 4x4, stride 2, 'same' padding (use `padding=1`) * `ReLU` * BatchNorm * ConvTranspose2d: 1 filter of 4x4, stride 2, 'same' padding (use `padding=1`) * `TanH` * Should have a 28x28x1 image, reshape back into 784 vector (using `Flatten()`)

Implement `build_dc_generator` in `gan.py`

[ ]:
```
from gan import build_dc_generator

test_g_gan = build_dc_generator().type(dtype)
test_g_gan.apply(initialize_weights)

fake_seed = torch.randn(batch_size, NOISE_DIM).type(dtype)
fake_images = test_g_gan.forward(fake_seed)
fake_images.size()
```

[ ]:
```
torch.Size([128, 784])
```

Check the number of parameters in your generator as a sanity check:

[ ]:
```
from torchinfo import summary
model = build_dc_generator(4)
summary(model, input_size=[128,4])
```

[ ]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
Sequential                                [128, 784]                --
 Linear: 1-1                             [128, 1024]               5,120
 ReLU: 1-2                               [128, 1024]               --
 BatchNorm1d: 1-3                        [128, 1024]               2,048
```

```
 Linear: 1-4                           [128, 6272]              6,428,800
 ReLU: 1-5                             [128, 6272]              --
 BatchNorm1d: 1-6                      [128, 6272]              12,544
 Unflatten: 1-7                        [128, 128, 7, 7]         --
 ConvTranspose2d: 1-8                  [128, 64, 14, 14]        131,136
 ReLU: 1-9                             [128, 64, 14, 14]        --
 BatchNorm2d: 1-10                     [128, 64, 14, 14]        128
 ConvTranspose2d: 1-11                 [128, 1, 28, 28]         1,025
 Tanh: 1-12                            [128, 1, 28, 28]         --
 Flatten: 1-13                         [128, 784]               --
================================================================================
==========
Total params: 6,580,801
Trainable params: 6,580,801
Non-trainable params: 0
Total mult-adds (G): 4.22
================================================================================
==========
Input size (MB): 0.00
Forward/backward pass size (MB): 41.44
Params size (MB): 26.32
Estimated Total Size (MB): 67.76
================================================================================
==========
```

[ ]:
```python
def test_dc_generator(true_count=6580801):
    model = build_dc_generator(4)
    cur_count = count_params(model)
    print(cur_count)
    if cur_count != true_count:
        print('Incorrect number of parameters in generator. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in generator.')

test_dc_generator()
```

```
6580801
Correct number of parameters in generator.
```

Run the following cell to train your DCGAN. If you train with the CPU, it takes about 35 minutes. If you train with the T4 GPU, it takes about 1 minute.

[ ]:
```python
D_DC = build_dc_classifier(batch_size).type(dtype)
D_DC.apply(initialize_weights)
G_DC = build_dc_generator().type(dtype)
G_DC.apply(initialize_weights)
```

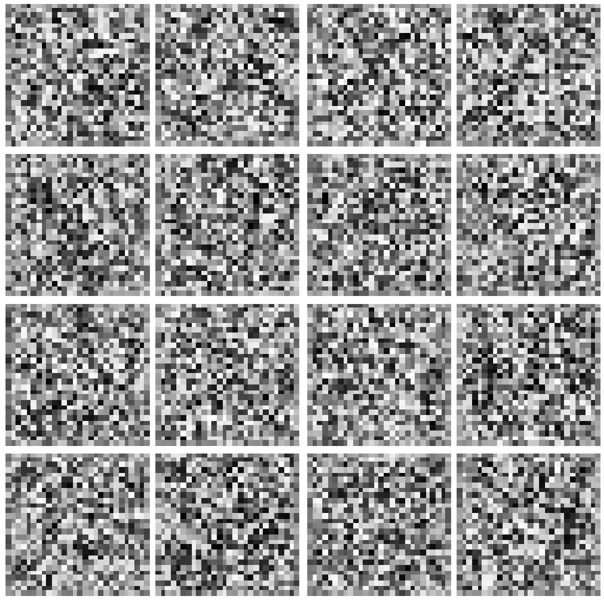
```python
D_DC_solver = get_optimizer(D_DC)
G_DC_solver = get_optimizer(G_DC)

images = run_a_gan(
    D_DC,
    G_DC,
    D_DC_solver,
    G_DC_solver,
    discriminator_loss,
    generator_loss,
    loader_train,
    num_epochs=5
)
```

```
Iter: 0, D: 1.413, G:0.3349
Iter: 250, D: 1.227, G:0.8495
Iter: 500, D: 1.257, G:0.9724
Iter: 750, D: 1.226, G:1.121
Iter: 1000, D: 1.2, G:0.9422
Iter: 1250, D: 1.258, G:0.8282
Iter: 1500, D: 1.205, G:0.8817
Iter: 1750, D: 1.259, G:0.7328
```

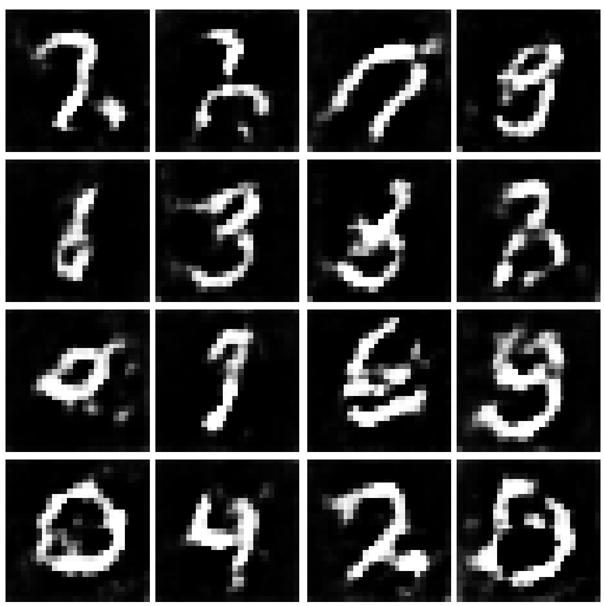
Run the cell below to show generated images.

[ ]:
```python
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

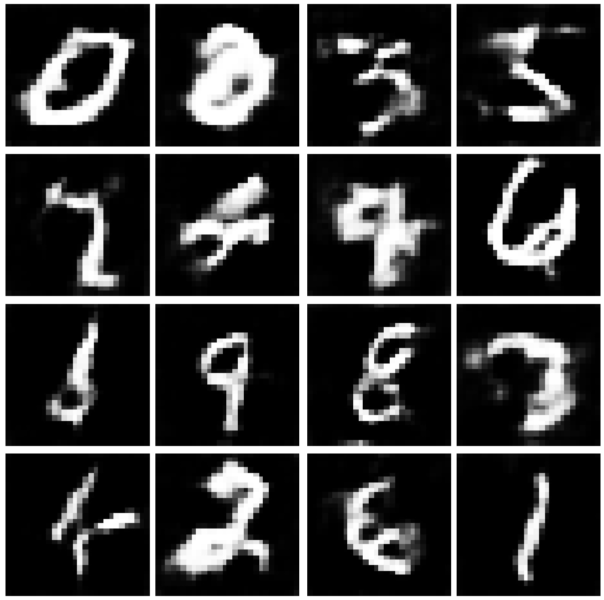
```
Iter: 0
```



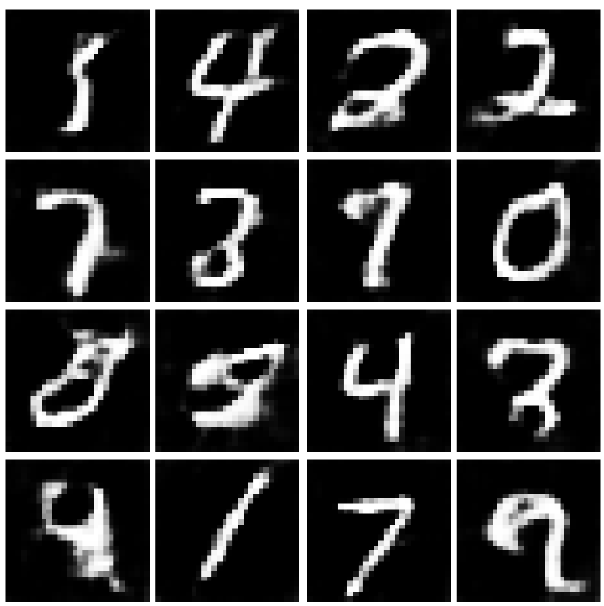
Iter: 250



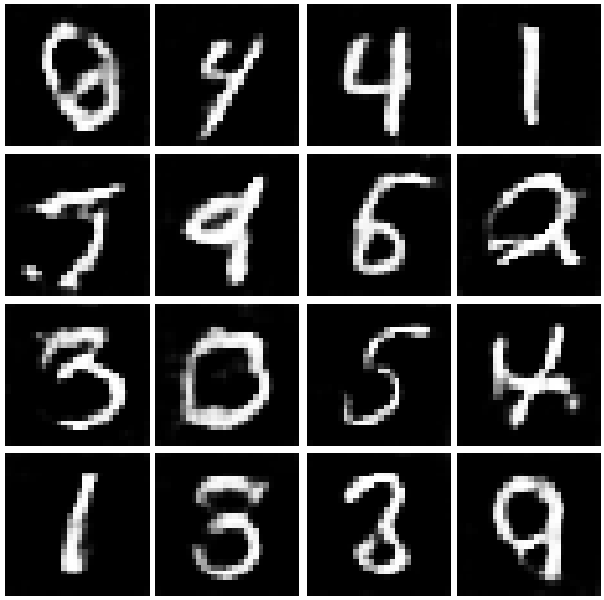
Iter: 500



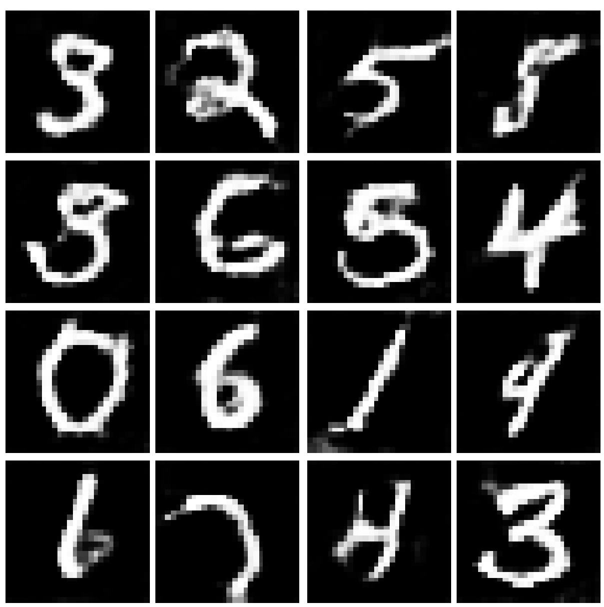
Iter: 750



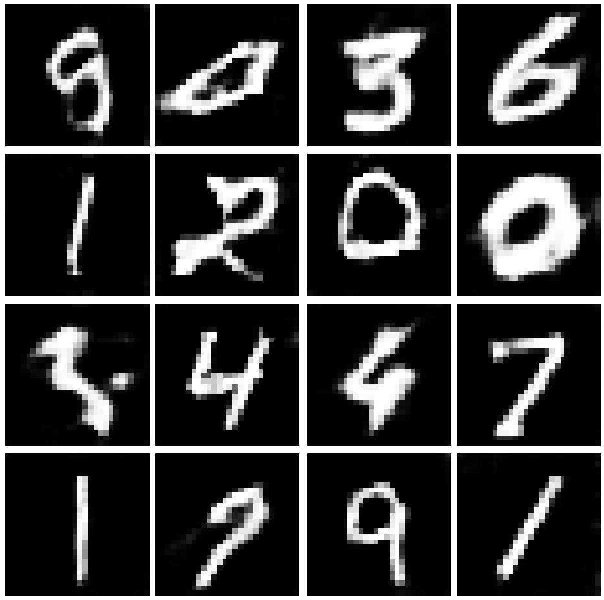
Iter: 1000



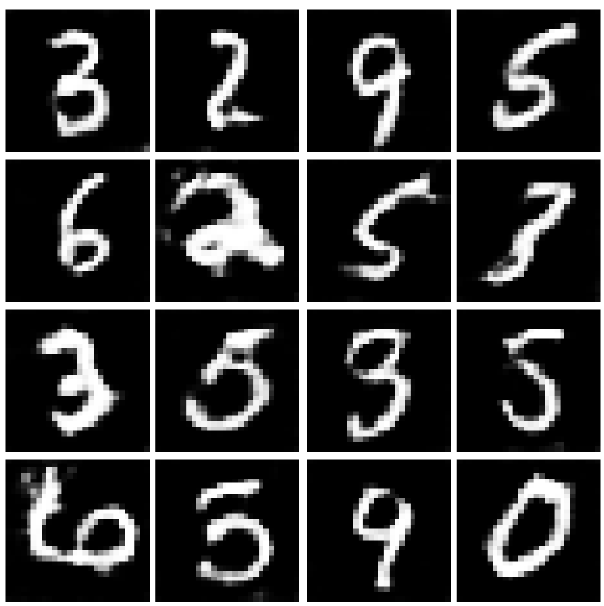
Iter: 1250



Iter: 1500



Iter: 1750

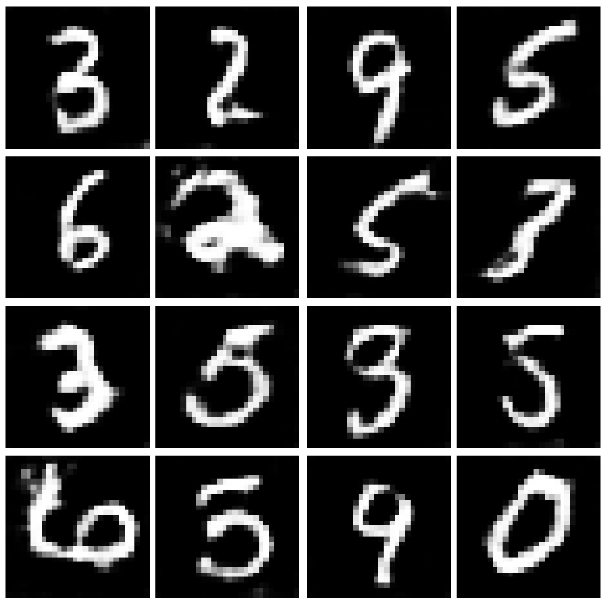


8.1 Inline Question 3

What does your final DCGAN image look like?

```
[ ]: # This output is your answer.
print("DCGAN final image:")
show_images(images[-1])
plt.show()
```

DCGAN final image:



8.2 Inline Question 4 (1 point)

We will look at an example to see why alternating minimization of the same objective (like in a GAN) can be tricky business.

Consider $f(x, y) = xy$. What does $\min_x \max_y f(x, y)$ evaluate to? (Hint: minmax tries to minimize the maximum value achievable.)

Now try to evaluate this function numerically for 6 steps, starting at the point $(1, 1)$, by using alternating gradient (first updating y, then updating x using that updated y) with step size 1. **Here step size is the learning_rate, and steps will be learning_rate * gradient.** You'll find that writing out the update step in terms of $x_t, y_t, x_{t+1}, y_{t+1}$ will be useful.

Breifly explain what $\min_x \max_y f(x, y)$ evaluates to and record the six pairs of explicit values for (x_t, y_t) in the table below.

8.2.1 Your answer:

y_0	y_1	y_2	y_3	y_4	y_5	y_6
1	2	1	-1	-2	-1	1
x_0	x_1	x_2	x_3	x_4	x_5	x_6
1	-1	-2	-1	1	2	1

Similar to a GAN, it is a zero-sum game between y and x that y tends to maximize the objective function $f(x, y) = xy$ (corresponds with the the Generator aiming to maximize the loss function) whereas x tends to minimize the objective function (corresponds with the the Discriminator aiming to minimize the loss function)

```
[ ]: x_val = 1
     y_val = 1

     for i in range(6):
         y_val = y_val + 1*x_val
         x_val = x_val - 1*y_val
         print("n:", i+1, "y_n:", y_val, "x_n:", x_val)
```

```
n: 1 y_n: 2 x_n: -1
n: 2 y_n: 1 x_n: -2
n: 3 y_n: -1 x_n: -1
n: 4 y_n: -2 x_n: 1
n: 5 y_n: -1 x_n: 2
n: 6 y_n: 1 x_n: 1
```

8.3 Inline Question 5 (1 point)

Using this method, will we ever reach the optimal value? Why or why not?

8.3.1 Your answer:

The function value $x * y$ oscillates between 1 and -2. As a result, we will never reach the optimal value. Such an observation sheds light on why it is hard to train a GAN, which is also a zero-sum game. The loss function tend to change dynamically and cannot reach optima as we update the Generator and the Discriminator.

8.4 Inline Question 6 (1 point)

If the generator loss decreases during training while the discriminator loss stays at a constant high value from the start, is this a good sign? Why or why not? A qualitative answer is sufficient.

8.4.1 Your answer:

It is not a good sign. If a Discriminator is bad from the start, the Generator wouldn't be penalized even if it dishes out really fake-looking images. The Generator can easily fool the Discriminator with its synthetic image as the Discriminator is doing a coin flip on whether the image is fake or not. Without the correct feedback from the Discriminator, the Generator is hardly penalized and it is hard to train the Generator to generate realistic-looking synthetic images.