GANs-PyTorch

December 12, 2024

[ ]:
```
#COMMENT IF NOT USING COLAB VM

# 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
# assignment folder, e.g. 'DeepLearning/assignments/assignment5/'
FOLDERNAME = 'cs6353/assignments/assignment5/'
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))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

[ ]:
```
# #UNCOMMENT IF USING CADE
# import os
# ##### Request a GPU #####
# ## This function locates an available gpu for usage. In addition, this␣
 ↪function reserves a specificed
# ## memory space exclusively for your account. The memory reservation prevents␣
 ↪the decrement in computational
# ## speed when other users try to allocate memory on the same gpu in the shared␣
 ↪systems, i.e., CADE machines.
# ## Note: If you use your own system which has a GPU with less than 4GB of␣
 ↪memory, remember to change the
# ## specified mimimum memory.
# def define_gpu_to_use(minimum_memory_mb = 3500):
#     thres_memory = 600 #
```

```
#     gpu_to_use = None
#     try:
#         os.environ['CUDA_VISIBLE_DEVICES']
#         print('GPU already assigned before: ' + str(os.
 ↪environ['CUDA_VISIBLE_DEVICES']))
#         return
#     except:
#         pass

#     for i in range(16):
#         free_memory = !nvidia-smi --query-gpu=memory.free -i $i␣
 ↪--format=csv,nounits,noheader
#         if free_memory[0] == 'No devices were found':
#             break
#         free_memory = int(free_memory[0])

#         if free_memory>minimum_memory_mb-thres_memory:
#             gpu_to_use = i
#             break

#     if gpu_to_use is None:
#         print('Could not find any GPU available with the required free memory␣
 ↪of ' + str(minimum_memory_mb) \
#               + 'MB. Please use a different system for this assignment.')
#     else:
#         os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_to_use)
#         print('Chosen GPU: ' + str(gpu_to_use))

# ## Request a gpu and reserve the memory space
# define_gpu_to_use(4000)
```

1 Generative Adversarial Networks (GANs)

So far in cs6353, 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. This has ranged from straightforward classification of image categories to sentence generation (which was still phrased as a classification problem, our labels were in vocabulary space and we'd learned a recurrence to capture multi-word labels). 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 aternate 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:

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

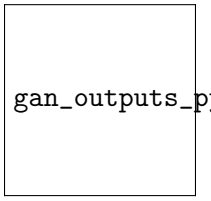
1.0.2 What else is there?

Since 2014, GANs have exploded into a huge research area, with massive workshops, and hundreds of new papers. Compared to other approaches for generative models, they often produce the highest quality samples but are some of the most difficult and finicky models to train (see this github repo that contains a set of 17 hacks that are useful for getting models working). Improving the stabiilty and robustness of GAN training is an open research question, with new papers coming out every day! For a more recent tutorial on GANs, see here. There is also some even more recent exciting work that changes the objective function to Wasserstein distance and yields much more stable results across model architectures: WGAN, WGAN-GP.

GANs are not the only way to train a generative model! For other approaches to generative modeling check out the deep generative model chapter of the Deep Learning book. Another popular way of training neural networks as generative models is Variational Autoencoders (co-discovered here and

here). Variatonal autoencoders combine neural networks with variationl inference to train deep generative models. These models tend to be far more stable and easier to train but currently don't produce samples that are as pretty as GANs.

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:



1.1 Setup

[ ]:
```
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 numpy as np

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

%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'

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

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

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"""
    param_count = np.sum([np.prod(p.size()) for p in model.parameters()])
    return param_count

answers = dict(np.load('gan-checks-tf.npz'))
```

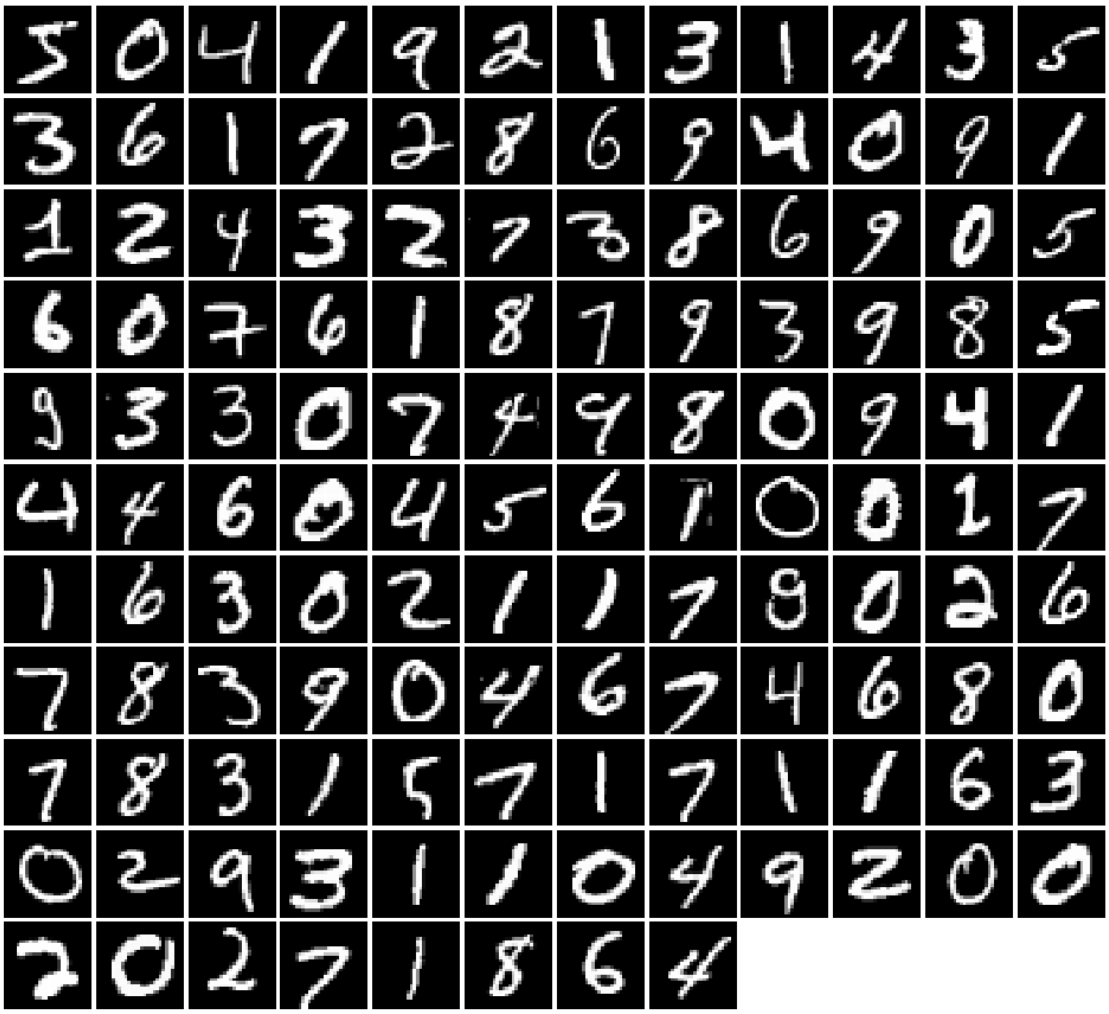
1.2 Dataset

GANs are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable without a GPU, 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_data`.

[ ]:
```
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

NUM_TRAIN = 50000
NUM_VAL = 5000

NOISE_DIM = 96
batch_size = 128

mnist_train = dset.MNIST('./cs6353/datasets/MNIST_data', 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('./cs6353/datasets/MNIST_data', train=True, download=True,
                           transform=T.ToTensor())
loader_val = DataLoader(mnist_val, batch_size=batch_size,
                        sampler=ChunkSampler(NUM_VAL, NUM_TRAIN))


imgs = next(loader_train.__iter__())[0].view(batch_size, 784).numpy().squeeze()
show_images(imgs)
```



1.3 Random Noise

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

Hint: use `torch.rand`.

[ ]:
```
def sample_noise(batch_size, dim):
    """
    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).
    """
    #Add your code here
    return 2 * torch.rand(batch_size, dim) - 1
```

Make sure noise is the correct shape and type:

[ ]:
```
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!
```

1.4 Flatten

Recall our Flatten operation from previous notebooks... this time we also 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.

[ ]:
```
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 image

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):
        init.xavier_uniform_(m.weight.data)
```

1.5 CPU / GPU

By default all code will run on CPU. GPUs are not needed for this assignment, but will help you to train your models faster. If you do want to run the code on a GPU, then change the `dtype` variable in the following cell.

[ ]:
```
dtype = torch.FloatTensor
#dtype = torch.cuda.FloatTensor ## UNCOMMENT THIS LINE IF YOU'RE ON A GPU!
```

2 Discriminator

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.

[ ]:
```
def discriminator():
    """
    Build and return a PyTorch model implementing the architecture above.
    """
    model = nn.Sequential(
        #Add your code here
        nn.Linear(784, 256),
        nn.LeakyReLU(0.01),
        nn.Linear(256, 256),
        nn.LeakyReLU(0.01),
        nn.Linear(256, 1)
    )
    return model
```

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

[ ]:
```
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

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])

[ ]:
```
def generator(noise_dim=NOISE_DIM):
    """
    Build and return a PyTorch model implementing the architecture above.
    """
    model = nn.Sequential(
        #Add your code here
        nn.Linear(noise_dim, 1024),
        nn.ReLU(),
        nn.Linear(1024, 1024),
        nn.ReLU(),
        nn.Linear(1024, 784),
        nn.Tanh()

    )
    return model
```

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

[ ]:
```
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

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 for you below.

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 $\log\left(1 - D(G(z))\right)$, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing.

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

    As per https://github.com/pytorch/pytorch/issues/751

    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.
    """
    neg_abs = - input.abs()
    loss = input.clamp(min=0) - input * target + (1 + neg_abs.exp()).log()
    return loss.mean()
```

[ ]:
```
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
    #Add your code here
    true_labels = torch.ones_like(logits_real).type(logits_real.dtype)
    false_labels = torch.zeros_like(logits_fake).type(logits_fake.dtype)

    loss_real = bce_loss(logits_real, true_labels)
    loss_fake = bce_loss(logits_fake, false_labels)

    return loss_real + loss_fake


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
    #Add your code here
    true_labels = torch.ones_like(logits_fake).type(logits_fake.dtype)
    return bce_loss(logits_fake, true_labels)
```

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

[ ]:
```
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: 2.83811e-08
```

[ ]:
```python
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

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.

[ ]:
```python
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
    #Add your code here
    optimizer = optim.Adam(model.parameters(), lr=1e-3, betas=(0.5, 0.999))
    return optimizer
```

6 Training a GAN!

We provide you the main training loop... you won't need to change this function, but we encourage you to read through and understand it.

[ ]:
```python
def run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss,␣
↪show_every=250,
              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.
```

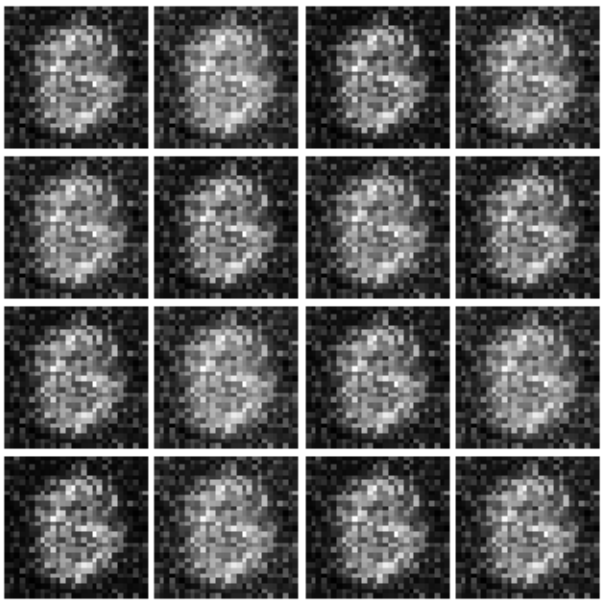
```
    - discriminator_loss, generator_loss: Functions to use for computing the␣
↪generator and
      discriminator loss, respectively.
    - show_every: Show samples after every show_every iterations.
    - batch_size: Batch size to use for training.
    - noise_size: Dimension of the noise to use as input to the generator.
    - num_epochs: Number of epochs over the training dataset to use for training.
    """
    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.view(batch_size, -1).type(dtype)  # Flatten the input
            logits_real = D(2 * (real_data - 0.5))

            g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
            fake_images = G(g_fake_seed)
            logits_fake = D(fake_images)

            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)
            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_error.item()))
                imgs_numpy = fake_images.data.cpu().numpy()
                show_images(imgs_numpy[0:16])
                plt.show()
                print()
            iter_count += 1
```

[ ]:
```
# Make the discriminator
D = discriminator().type(dtype)

# Make the generator
```

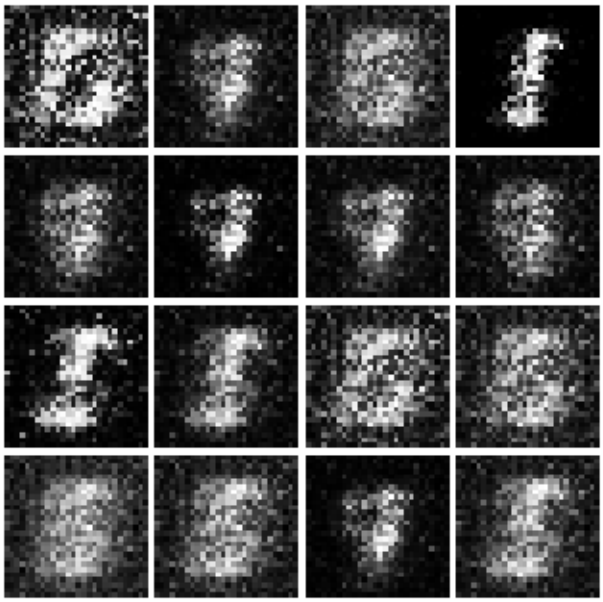
```
G = generator().type(dtype)

# 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!
run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss)
```

```
Iter: 0, D: 1.374, G:0.7046
```

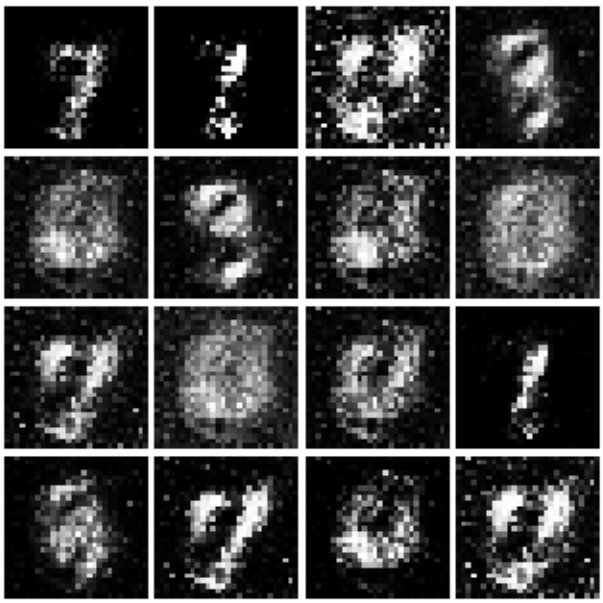
```
Iter: 250, D: 1.126, G:0.872
```



Iter: 500, D: 1.287, G:0.8172

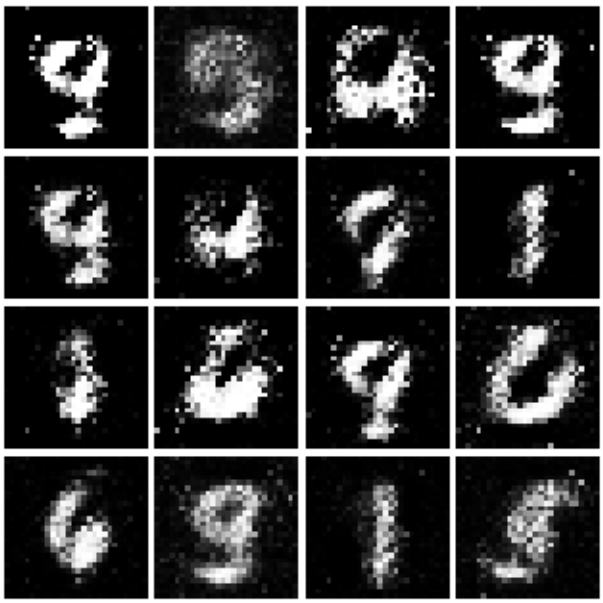


Iter: 750, D: 1.274, G:1.092

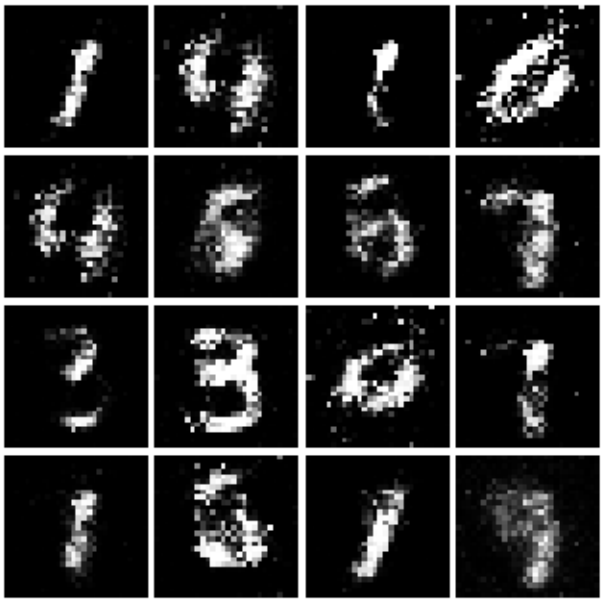


Iter: 1000, D: 1.206, G:1.0

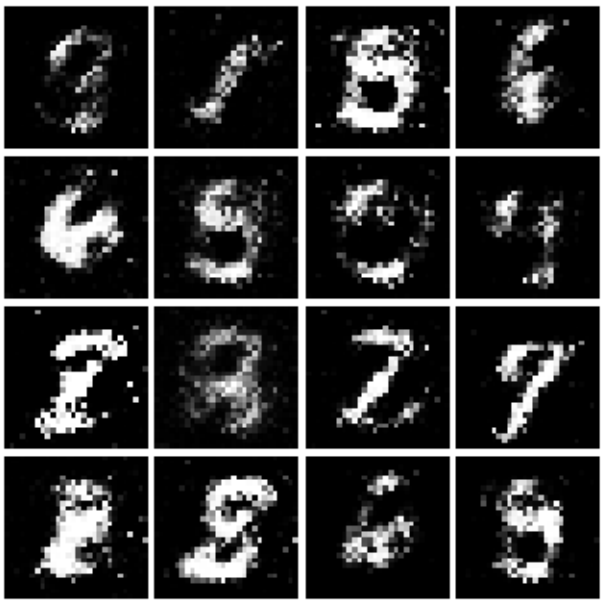
Iter: 1250, D: 1.294, G:0.9384



Iter: 1500, D: 1.324, G:0.9245

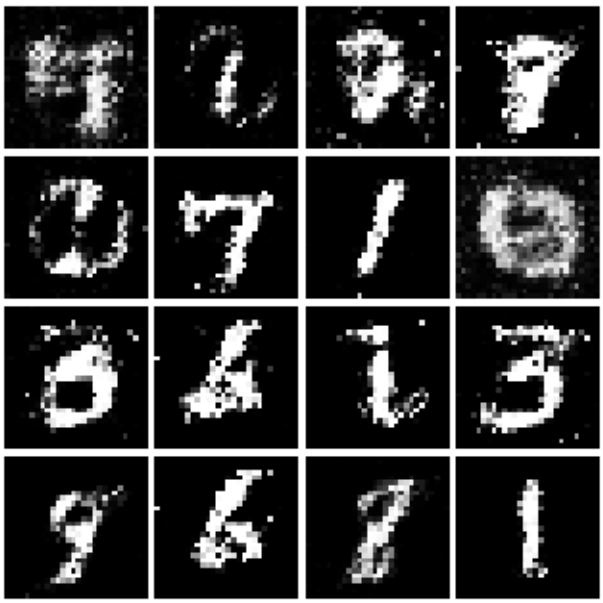


Iter: 1750, D: 1.319, G:0.8597



Iter: 2000, D: 1.313, G:0.7069

Iter: 2250, D: 1.344, G:0.816

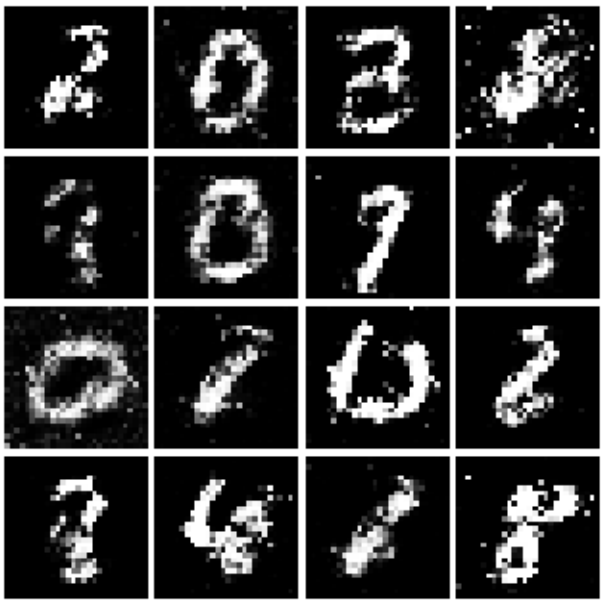


Iter: 2500, D: 1.262, G:0.8367

Iter: 2750, D: 1.279, G:0.8162

Iter: 3000, D: 1.468, G:0.7639

Iter: 3250, D: 1.311, G:0.799



Iter: 3500, D: 1.294, G:0.9003

Iter: 3750, D: 1.345, G:0.7502

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

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:

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

and the discriminator loss:

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

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`).

[ ]:
```
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
    #Add your code here

    # Compute the discriminator loss for real and fake scores
    loss_real = 0.5 * ((scores_real - 1) ** 2).mean()
    loss_fake = 0.5 * (scores_fake ** 2).mean()

    # Combine the two losses
    loss = loss_real + loss_fake
    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
    #Add your code here

    # Compute the generator loss
    loss = 0.5 * ((scores_fake - 1) ** 2).mean()

    return loss
```

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

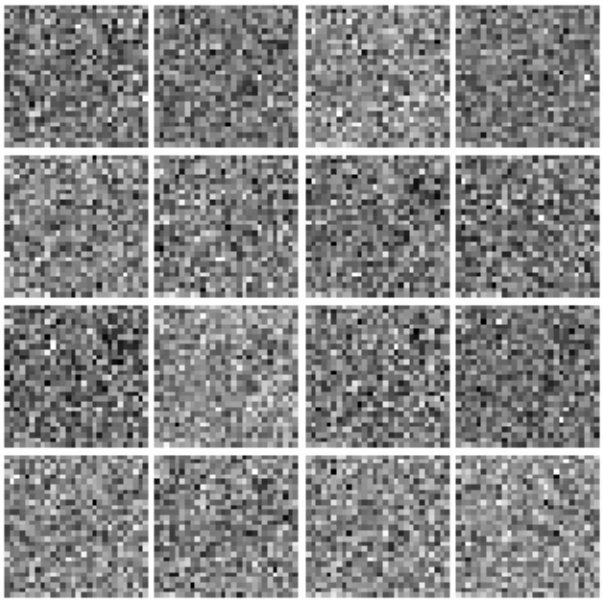
[ ]:
```
def test_lsgan_loss(score_real, score_fake, d_loss_true, g_loss_true):
    score_real = torch.Tensor(score_real).type(dtype)
    score_fake = torch.Tensor(score_fake).type(dtype)
    d_loss = ls_discriminator_loss(score_real, score_fake).cpu().numpy()
    g_loss = ls_generator_loss(score_fake).cpu().numpy()
    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: 3.36961e-08
```

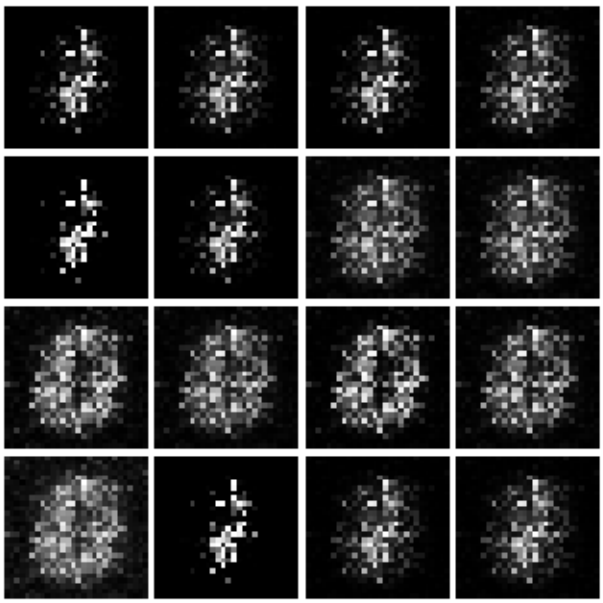
Run the following cell to train your model!

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

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

run_a_gan(D_LS, G_LS, D_LS_solver, G_LS_solver, ls_discriminator_loss,␣
↪ls_generator_loss)
```

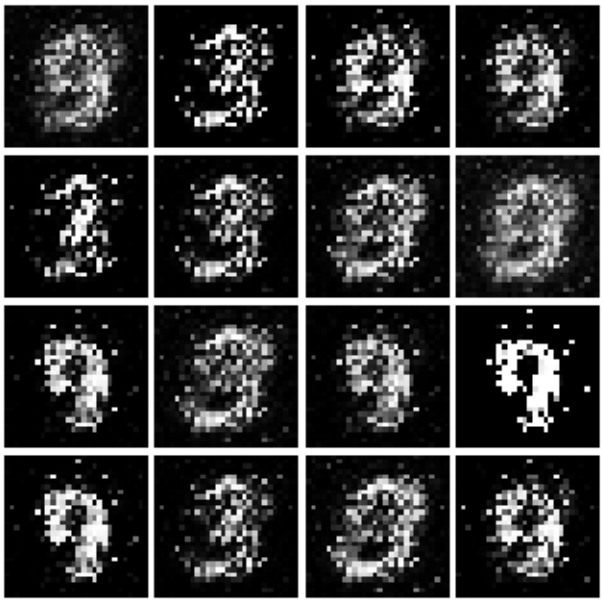
```
Iter: 0, D: 0.4947, G:0.4745
```



```
Iter: 250, D: 0.09093, G:0.2866
```

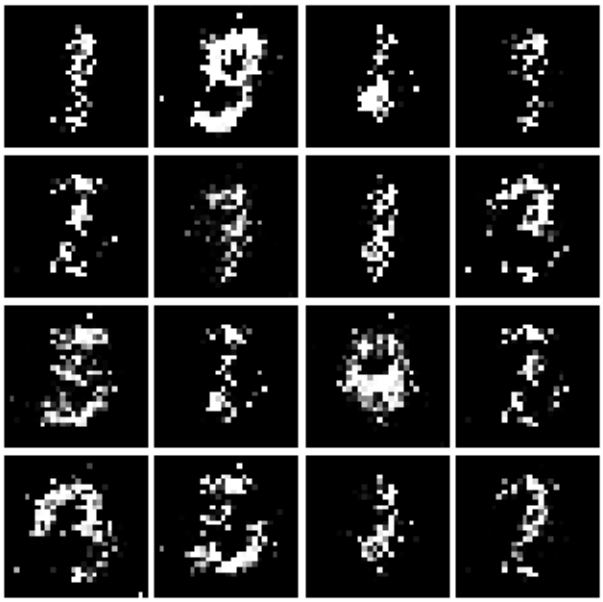


```
Iter: 500, D: 0.1713, G:0.4508
```



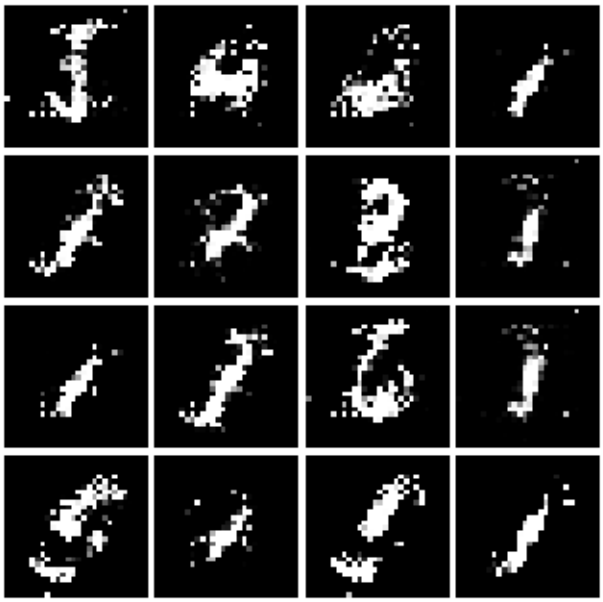
```
Iter: 750, D: 0.08827, G:0.3807
```

Iter: 1000, D: 0.1558, G:0.2658



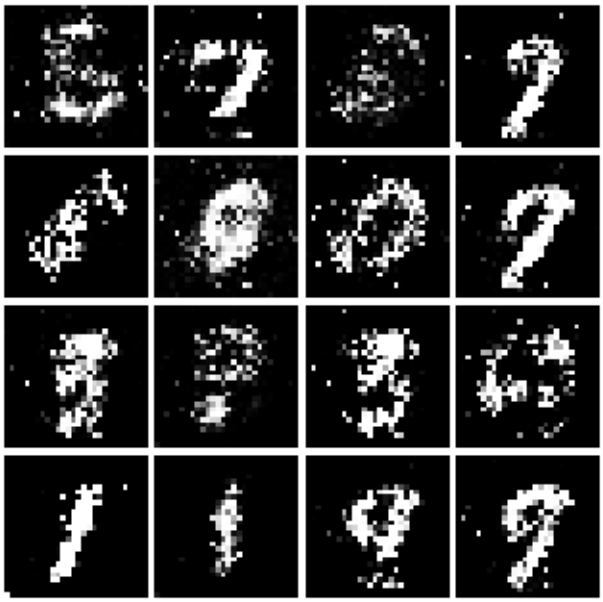
Iter: 1250, D: 0.1541, G:0.2381

Iter: 1500, D: 0.1442, G:0.3187

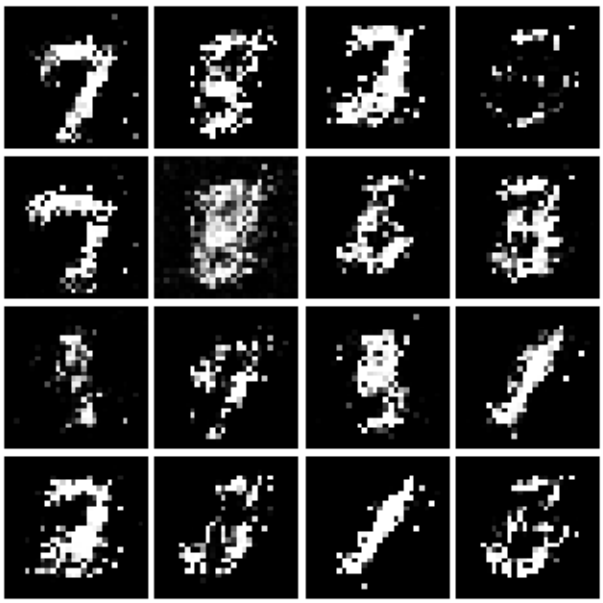


Iter: 1750, D: 0.1717, G:0.2602

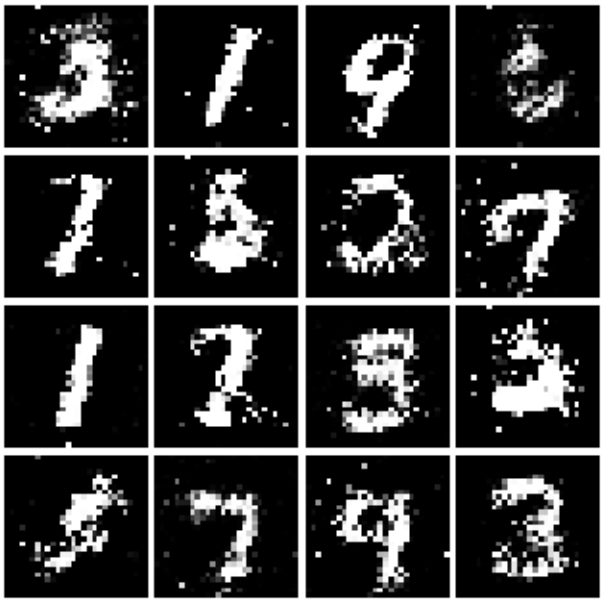
Iter: 2000, D: 0.2018, G:0.2315



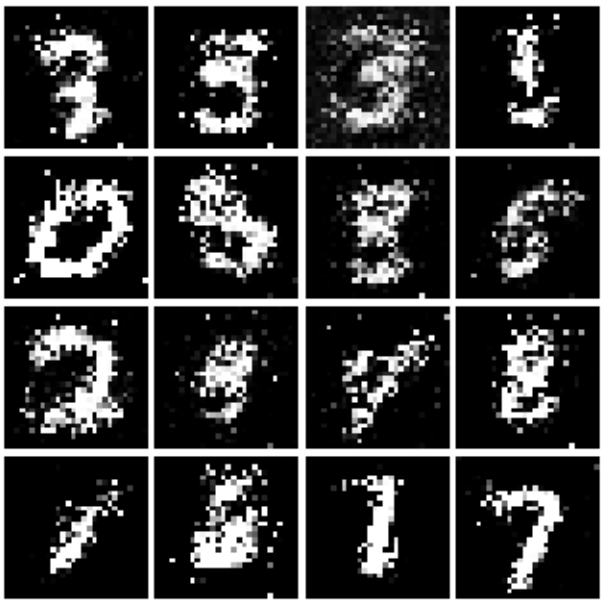
Iter: 2250, D: 0.2049, G:0.2581



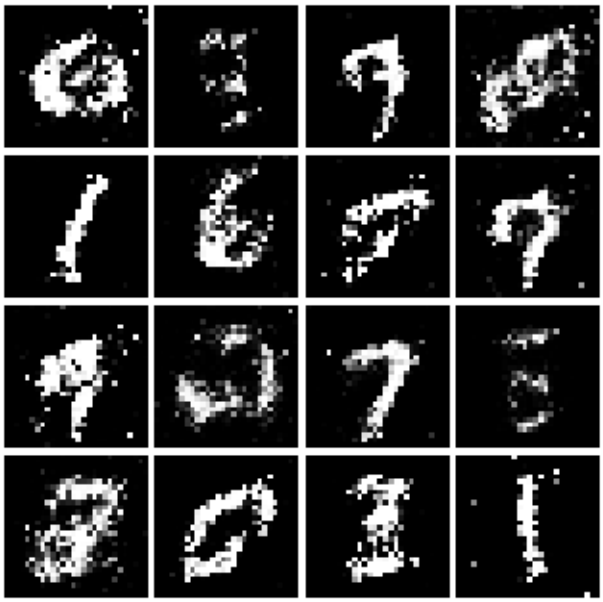
Iter: 2500, D: 0.2623, G:0.2015



Iter: 2750, D: 0.2425, G:0.1772



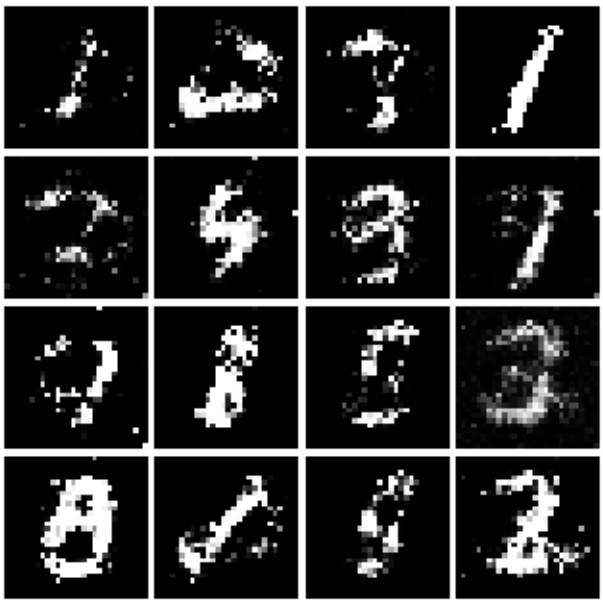
Iter: 3000, D: 0.2474, G:0.168



Iter: 3250, D: 0.2278, G:0.1595

Iter: 3500, D: 0.2286, G:0.1647

Iter: 3750, D: 0.2353, G:0.1682



8 Deeply Convolutional GANs

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

- Reshape into image tensor (Use Unflatten!)
- 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

[ ]:
```
def build_dc_classifier():
    """
    Build and return a PyTorch model for the DCGAN discriminator implementing
```

```
    the architecture above.
    """
    return nn.Sequential(
        ############################
        ######### TO DO ###########
        ############################
        Unflatten(N=-1, C=1, H=28, W=28),  # Reshape into image tensor
        nn.Conv2d(1, 32, kernel_size=5, stride=1),  # Conv2D: 32 Filters, 5x5,␣
↪Stride 1
        nn.LeakyReLU(0.01),  # Leaky ReLU with alpha=0.01
        nn.MaxPool2d(kernel_size=2, stride=2),  # Max Pool 2x2, Stride 2
        nn.Conv2d(32, 64, kernel_size=5, stride=1),  # Conv2D: 64 Filters, 5x5,␣
↪Stride 1
        nn.LeakyReLU(0.01),  # Leaky ReLU with alpha=0.01
        nn.MaxPool2d(kernel_size=2, stride=2),  # Max Pool 2x2, Stride 2
        Flatten(),  # Flatten the tensor
        nn.Linear(64 * 4 * 4, 1024),  # Fully Connected with output size 4 x 4 x␣
↪64
        nn.LeakyReLU(0.01),  # Leaky ReLU with alpha=0.01
        nn.Linear(1024, 1)  # Fully Connected with output size 1
    )

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

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

[ ]:
```
def test_dc_classifer(true_count=1102721):
    model = build_dc_classifier()
    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_dc_classifer()
```

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

Generator For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. See the documentation for tf.nn.conv2d_transpose. 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 * Reshape into Image Tensor of shape 7, 7, 128 * Conv2D^T (Transpose): 64 filters of 4x4, stride 2, 'same' padding * `ReLU` * BatchNorm * Conv2D^T (Transpose): 1 filter of 4x4, stride 2, 'same' padding * `TanH` * Should have a 28x28x1

image, reshape back into 784 vector

[ ]:
```
def build_dc_generator(noise_dim=NOISE_DIM):
    """
    Build and return a PyTorch model implementing the DCGAN generator using
    the architecture described above.
    """
    return nn.Sequential(
        ############################
        ######### TO DO ############
        ############################
        nn.Linear(noise_dim, 1024),  # Fully connected with output size 1024
        nn.ReLU(),
        nn.BatchNorm1d(1024),  # BatchNorm
        nn.Linear(1024, 7 * 7 * 128),  # Fully connected with output size 7 x 7␣
↪x 128
        nn.ReLU(),
        nn.BatchNorm1d(7 * 7 * 128),  # BatchNorm
        Unflatten(N=-1, C=128, H=7, W=7),  # Reshape into image tensor
        nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),  #␣
↪Conv2D^T: 64 filters
        nn.ReLU(),
        nn.BatchNorm2d(64),  # BatchNorm
        nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2, padding=1),  #␣
↪Conv2D^T: 1 filter
        nn.Tanh(),  # TanH activation
        Flatten()  # Reshape back into 784 vector
    )

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:

[ ]:
```
def test_dc_generator(true_count=6580801):
    model = build_dc_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.')
```

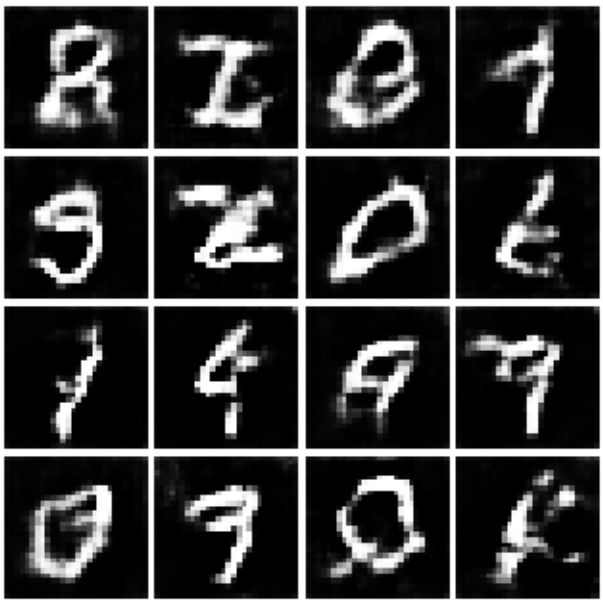
```
test_dc_generator()
```

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

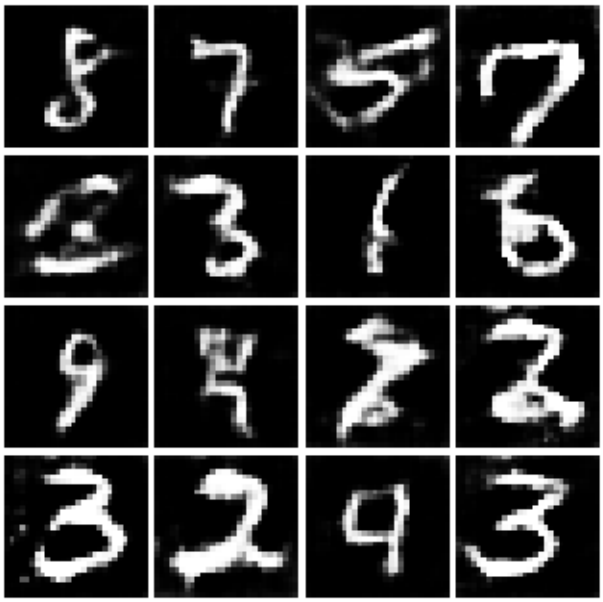
[36]:
```
D_DC = build_dc_classifier().type(dtype)
D_DC.apply(initialize_weights)
G_DC = build_dc_generator().type(dtype)
G_DC.apply(initialize_weights)

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

run_a_gan(D_DC, G_DC, D_DC_solver, G_DC_solver, discriminator_loss,␣
 ↪generator_loss, num_epochs=5)
```

```
Iter: 0, D: 1.433, G:1.441
```

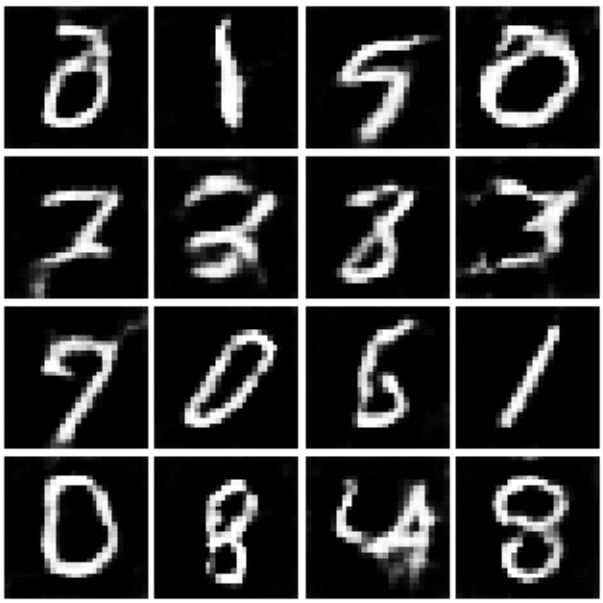
```
Iter: 250, D: 1.272, G:0.695
```



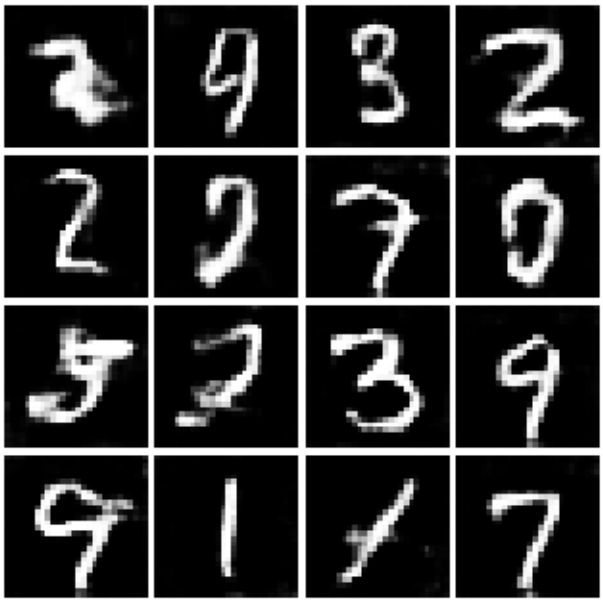
Iter: 500, D: 1.144, G:1.266



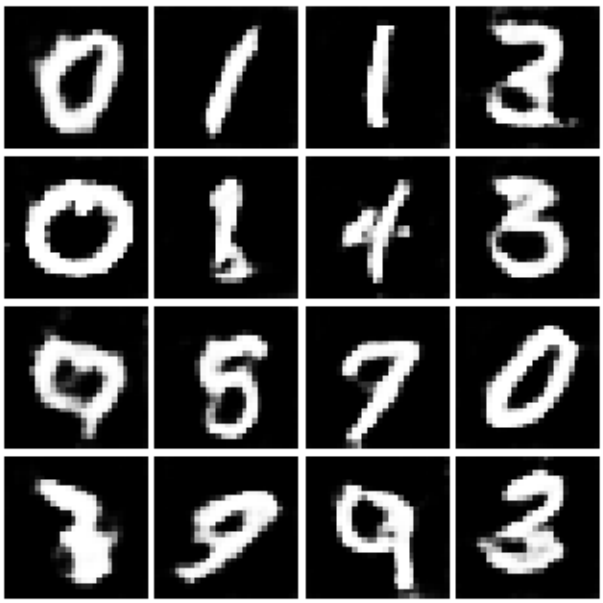
Iter: 750, D: 1.135, G:1.086



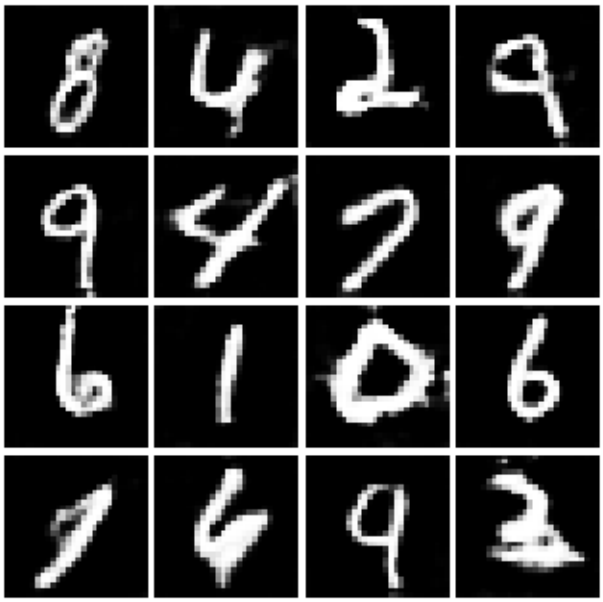
Iter: 1000, D: 1.186, G:1.019



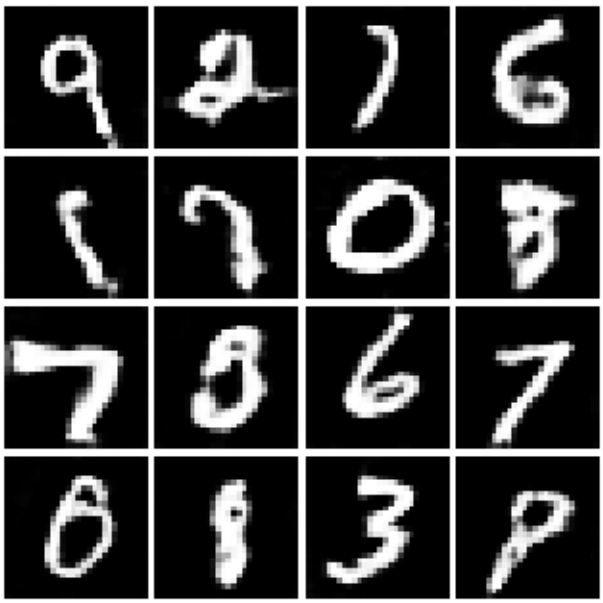
Iter: 1250, D: 1.206, G:1.006



Iter: 1500, D: 1.204, G:0.7908



Iter: 1750, D: 1.172, G:1.022



8.1 INLINE QUESTION 1

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

8.1.1 Your answer:

If the generator loss decreases while the discriminator loss stays consistently high, this is not a good sign because it indicates an imbalance in the GAN's training dynamics:

1. Generator Loss Decrease: A lower generator loss suggests that the generator is improving, but this improvement might not be meaningful if the discriminator is not providing effective feedback.
2. Constant High Discriminator Loss: If the discriminator cannot differentiate between real and fake images (indicated by a high loss), it fails to push the generator toward generating high-quality outputs.

Problem:

1. Imbalance: GANs require the generator and discriminator to learn together. A weak discriminator leads to unchecked generator progress, often resulting in poor or repetitive outputs (e.g., mode collapse).
2. Lack of Feedback: Without a learning discriminator, the generator doesn't get meaningful gradients to improve effectively.

Prossible Solutions:

1. Improve discriminator capacity (e.g., deeper networks).
2. Balance training by adjusting learning rates or training the discriminator more frequently.
3. Use stabilization techniques like gradient penalties (e.g., Wasserstein GAN).