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

```python
# 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('/data/zilai/self/gan_239')

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

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.

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:

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

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**.

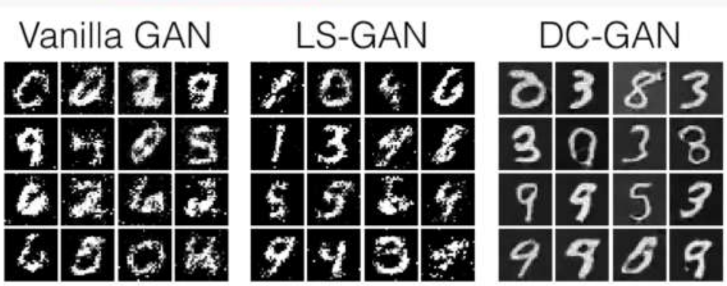
While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator 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:
$$\underset{D}{\text{maximize}}\; \mathbb{E}_{x \sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z \sim p(z)}\left[\log(1 - D(G(z)))\right]$$

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:

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




```python
# Setup cell.
import numpy as np
import torch
```

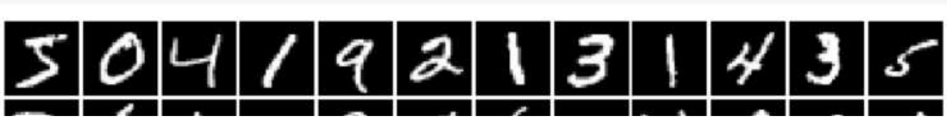
```python
import torch.nn as nn
from torch.nn import init
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
import torchvision.datasets as dset
import matplotlib.pyplot as plt
import matplotlib.gridspec 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
```

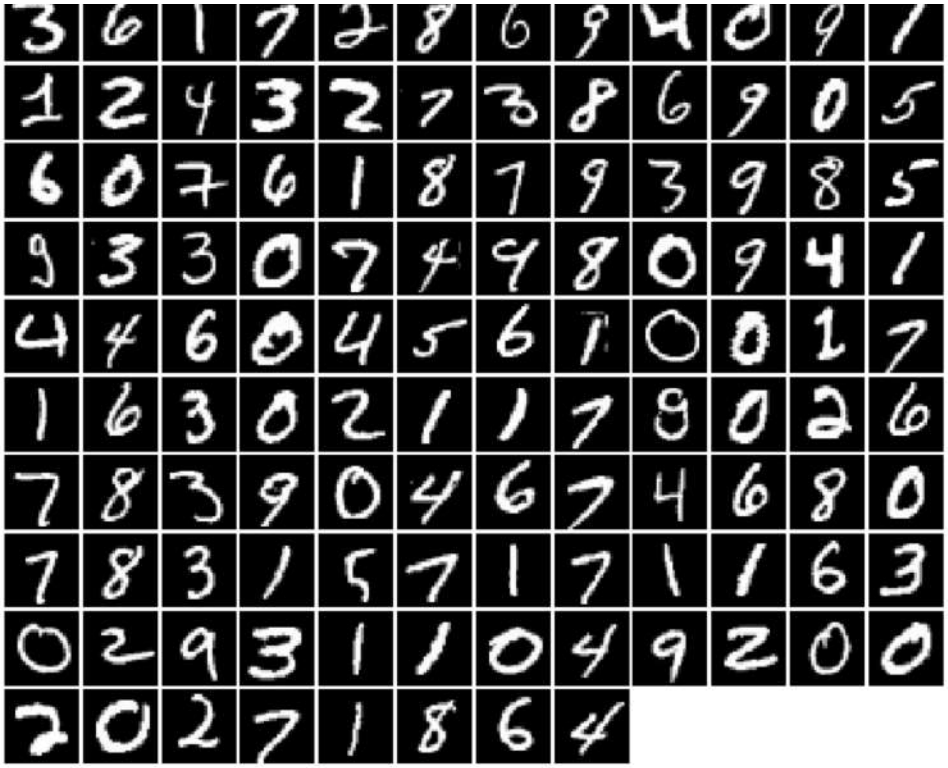
Dataset

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`.

```python
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)
```





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!
```

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
```

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.
```

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.
```

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:

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

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
```

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`

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.

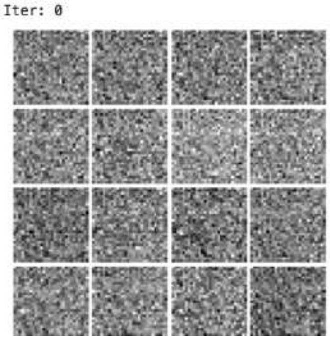
```
from gan import get_optimizer, run_a_gan

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

```
Iter: 0, D: 1.328, G:0.7202
Iter: 250, D: 1.269, G:0.9942
Iter: 500, D: 0.9579, G:1.302
Iter: 750, D: 1.042, G:1.02
Iter: 1000, D: 1.225, G:1.241
Iter: 1250, D: 1.107, G:1.271
Iter: 1500, D: 1.299, G:1.014
Iter: 1750, D: 1.215, G:0.9923
Iter: 2000, D: 1.328, G:0.8528
Iter: 2250, D: 1.536, G:0.6775
Iter: 2500, D: 1.383, G:0.7431
Iter: 2750, D: 1.335, G:0.8688
Iter: 3000, D: 1.315, G:0.8493
Iter: 3250, D: 1.326, G:0.8126
Iter: 3500, D: 1.319, G:0.8526
Iter: 3750, D: 1.338, G:0.7657
```

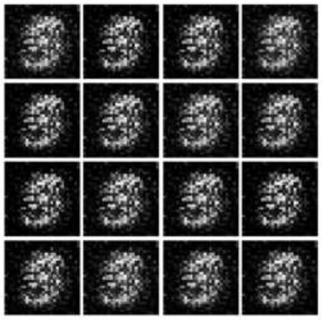
Run the cell below to show the generated images.

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

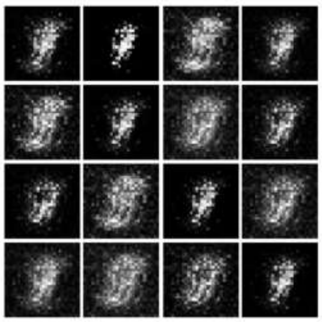
```
Iter: 0
```



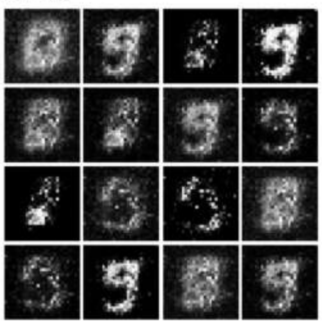
Iter: 250



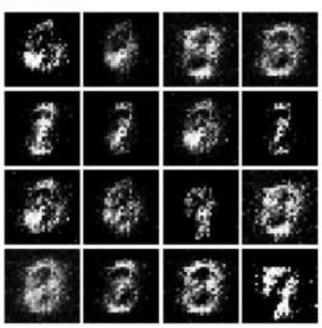
Iter: 500



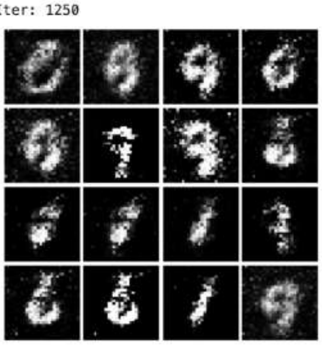
Iter: 750



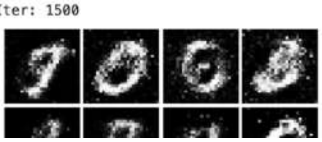
Iter: 1000



Iter: 1250



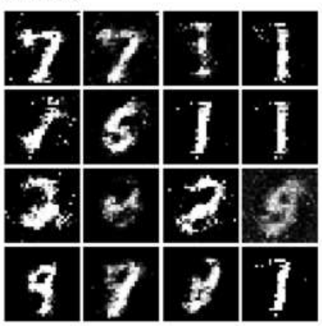
Iter: 1500



Iter: 1750

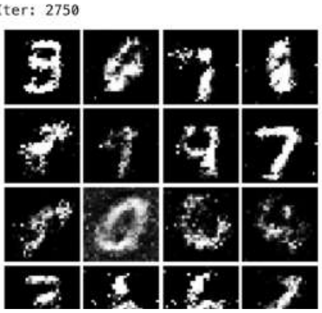
Iter: 2000

Iter: 2250



Iter: 2500

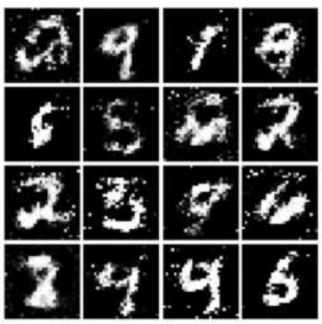
Iter: 2750



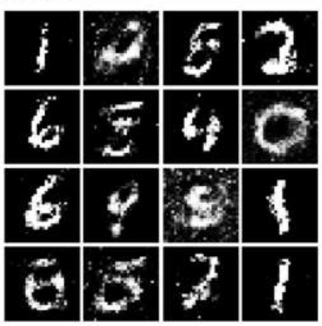
Iter: 3000

Iter: 3250

Iter: 3500



Iter: 3750

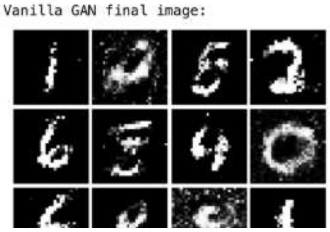


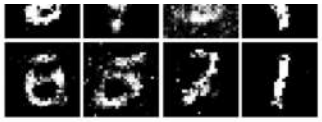
Inline Question 1

What does your final vanilla GAN image look like?

```python
# 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.

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:

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

and the discriminator loss:

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

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`).

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: 0.1674, G:0.9122
Iter: 500, D: 0.139, G:0.2987
Iter: 750, D: 0.1501, G:0.2977
Iter: 1000, D: 0.1335, G:0.357
Iter: 1250, D: 0.139, G:0.2768
Iter: 1500, D: 0.1558, G:0.367
Iter: 1750, D: 0.2487, G:0.1797
Iter: 2000, D: 0.2291, G:0.2039
Iter: 2250, D: 0.2082, G:0.2018
Iter: 2500, D: 0.2397, G:0.154
Iter: 2750, D: 0.2427, G:0.2019
Iter: 3000, D: 0.216, G:0.1657
Iter: 3250, D: 0.2246, G:0.1628
Iter: 3500, D: 0.2189, G:0.1618
Iter: 3750, D: 0.2256, G:0.1719
```

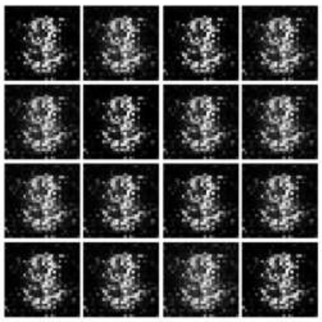
Run the cell below to show generated images.

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

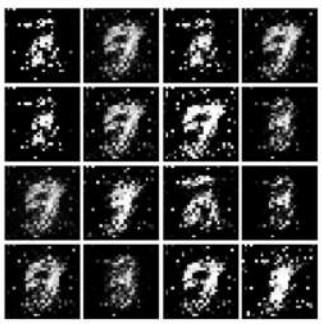
```
print()
```

Iter: 0

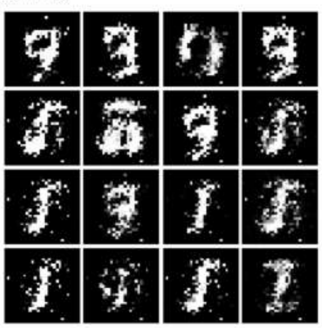
Iter: 250



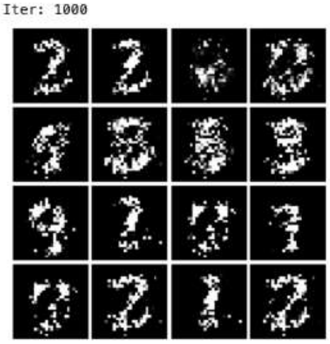
Iter: 500



Iter: 750

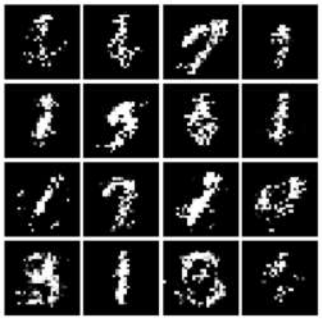


Iter: 1000



Iter: 1250

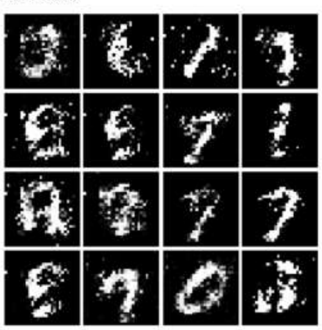
Iter: 1500



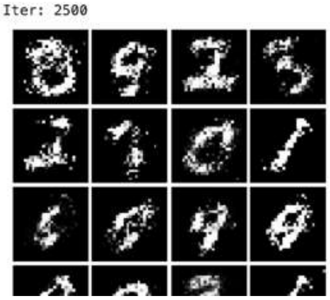
Iter: 1750

Iter: 2000

Iter: 2250



Iter: 2500

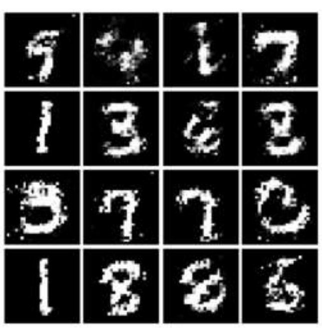


Iter: 2750

Iter: 3000

Iter: 3250

Iter: 3500



Iter: 3750

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:
```

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`

```python
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:

```python
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:

```
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.
```

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.

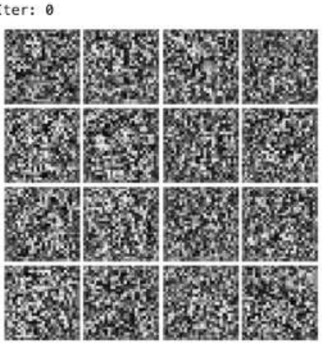
```
D_DC = build_dc_classifier(batch_size).type(dtype)
D_DC.apply(initialize_weights)
G_DC = build_dc_generator().type(dtype)
G_DC.apply(initialize_weights)

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: 1.508, G:0.1981
Iter: 250, D: 1.355, G:1.055
Iter: 500, D: 1.272, G:1.157
Iter: 750, D: 1.138, G:1.143
Iter: 1000, D: 1.292, G:1.002
Iter: 1250, D: 1.246, G:1.193
Iter: 1500, D: 1.131, G:1.118
Iter: 1750, D: 1.024, G:1.405
```

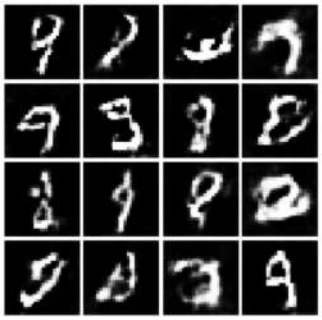
Run the cell below to show generated images.

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

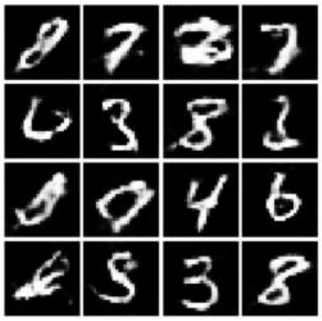
```
Iter: 0
```



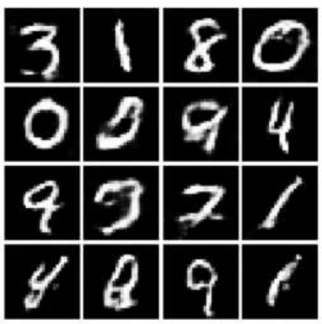
```
Iter: 250
```



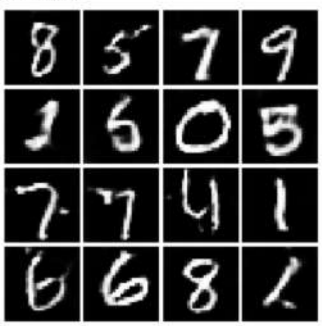
Iter: 500



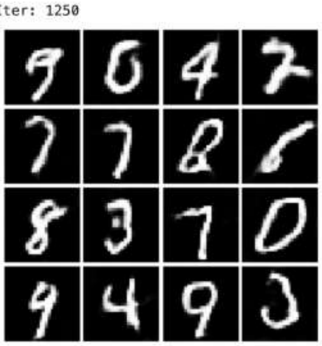
Iter: 750



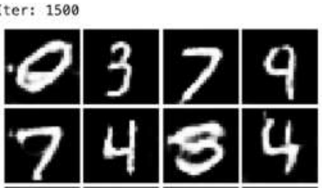
Iter: 1000

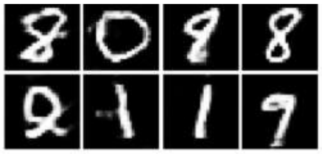


Iter: 1250

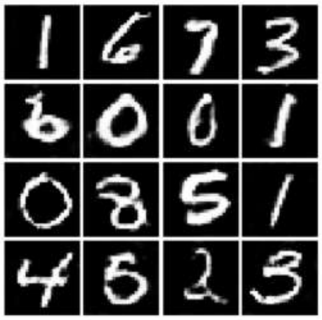


Iter: 1500





Iter: 1750

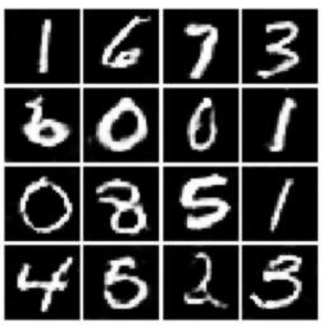


Inline Question 3

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:



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.

Your answer:

The gradients needed for the updates are simple. The gradient of $f(x, y) = xy$ with respect to y is x (holding x constant), and the gradient with respect to x is y (holding y constant). With a learning rate of 1, the update steps are as follows:

Update y: $y_{t+1} = y_t + 1 * x_t$ Update x: $x_{t+1} = x_t - 1 * y_{t+1}$

Initial: $(x_0, y_0) = (1, 1)$ Step 1: Update y: $y_1 = 1 + 1 * 1 = 2$, then update x: $x_1 = 1 - 2 = -1$ Step 2: Update y: $y_2 = 2 + 1 * (-1) = 1$, then update x: $x_2 = -1 - 1 = -2$ Step 3: Update y: $y_3 = 1 + 1 * (-2) = -1$, then update x: $x_3 = -2 - (-1) = -1$ Step 4: Update y: $y_4 = -1 + 1 * (-1) = -2$, then update x: $x_4 = -1 - (-2) = 1$ Step 5: Update y: $y_5 = -2 + 1 * 1 = -1$, then update x: $x_5 = 1 - (-1) = 2$ Step 6: Update y: $y_6 = -1 + 1 * 2 = 1$, then update x: $x_6 = 2 - 1 = 1$

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

Inline Question 5 (1 point)

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

Your answer:

using this alternating gradient method, we will not reach an optimal value for $\min_x \max_y f(x, y)$ because the method leads to divergent oscillatory behavior rather than converging to a specific value.

The function $f(x, y) = xy$ is unbounded in both positive and negative directions. This means there is no single minimum or maximum value:

The function $f(x, y) = xy$ is unbounded in both positive and negative directions. This means there is no single minimum or maximum value; the function's value can grow infinitely large or small depending on the signs and magnitudes of x and y.

The process of alternating updates between x and y based on the gradient of the function leads to an oscillatory path rather than converging to a stationary point. As observed from the numerical evaluation, the updates lead to a cycle where the absolute values of x and y grow without bound, but alternate in sign. This pattern does not converge to a point but rather demonstrates divergent behavior.

This illustrates a fundamental challenge in training GANs and similar models where alternating optimization is used without additional constraints or regularization techniques to ensure convergence.

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.

Your answer:

this is generally not a good sign.

(1)The discriminator in a GAN has the task of distinguishing between real data and fake data generated by the generator. A well-functioning discriminator should improve over time, becoming better at telling real from fake. This, in turn, forces the generator to improve its ability to generate data that looks real.

(2)If the discriminator loss is high and constant, it suggests that the discriminator is not improving its ability to distinguish real from fake data. A high loss implies that it's making incorrect classifications frequently. If this happens from the start and does not change, it indicates that the discriminator is not learning effectively.

(3) The generator loss decreasing implies that the generator is getting better at fooling the discriminator. However, if the discriminator is not improving (as indicated by its constant high loss), the generator's improvements might not necessarily mean it's generating high-quality data. Instead, it could simply mean that the generator is exploiting the weaknesses of a poorly performing discriminator.

(4) A poorly performing discriminator can lead to a generator that produces data that deviates significantly from the real data distribution, as it is not being correctly penalized for producing non-realistic outputs. The generator might converge to producing outputs that consistently fool the discriminator but are not genuinely realistic or diverse.

ga_py.ipynb ☆

文件 修改 视图 插入 代码执行程序 工具 帮助 已保存所有更改

评论 分享

+ 代码 + 文本 连接 Colab AI

```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)

                      OUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # Generate uniform random noise from -1 to 1
    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)

    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(
        # Flatten the input
        Flatten(),
        # Fully connected layer
        nn.Linear(784, 256),
        # LeakyReLU activation
        nn.LeakyReLU(0.01),
        # Fully connected layer
        nn.Linear(256, 256),
        # LeakyReLU activation
        nn.LeakyReLU(0.01),
        # Fully connected layer
        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                                               #
```

已成功保存！

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

    model = nn.Sequential(
    # Fully connected layer
    nn.Linear(noise_dim, 1024),
    # ReLU activation
    nn.ReLU(),
    # Fully connected layer
    nn.Linear(1024, 1024),
    # ReLU activation
    nn.ReLU(),
    # Fully connected layer
    nn.Linear(1024, 784),
    # Tanh activation
    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)*****

     # target label for real images is 1, for fake images is 0
    labels_real = torch.ones_like(logits_real)
    labels_fake = torch.zeros_like(logits_fake)

    # compute loss for real images
    loss_real = nn.BCEWithLogitsLoss()(logits_real, labels_real)
    # compute loss for fake images
    loss_fake = nn.BCEWithLogitsLoss()(logits_fake, labels_fake)

    # total loss is the sum of these two losses
    loss = loss_real + loss_fake

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

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)*****

    # target label for fake images is 1
    labels_fake = torch.ones_like(logits_fake)

    # compute loss for fake images
    loss = nn.BCEWithLogitsLoss()(logits_fake, labels_fake)

    # *****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.
```

```python

    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 = torch.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)*****

     # target label for real images is 1, for fake images is 0
    loss_real = ((scores_real - 1)**2).mean()
    loss_fake = (scores_fake**2).mean()

    # total loss is the sum of these two losses
    loss = 0.5 * (loss_real + loss_fake)

    # *****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)*****

     # target label for fake images is 1
    loss = 0.5 * ((scores_fake - 1)**2).mean()

    # *****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)*****

    return nn.Sequential(
    # Conv2D: 32 Filters, 5x5, Stride 1
    nn.Conv2d(1, 32, 5, stride=1),
    nn.LeakyReLU(0.01),
    # Max Pool 2x2, Stride 2
    nn.MaxPool2d(2, stride=2),
    # Conv2D: 64 Filters, 5x5, Stride 1
    nn.Conv2d(32, 64, 5, stride=1),
    nn.LeakyReLU(0.01),
    # Max Pool 2x2, Stride 2
    nn.MaxPool2d(2, stride=2),
    # Flatten
    nn.Flatten(),
    # Fully Connected with output size 4 x 4 x 64
    nn.Linear(64*4*4, 4*4*64),
    nn.LeakyReLU(0.01),
    # Fully Connected with output size 1
    nn.Linear(4*4*64, 1)
)

    # *****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)*****

    return nn.Sequential(
    # Fully connected with output size 1024
    nn.Linear(noise_dim, 1024),
    nn.ReLU(),
    nn.BatchNorm1d(1024),
    # Fully connected with output size 7 x 7 x 128
    nn.Linear(1024, 7*7*128),
    nn.ReLU(),
    nn.BatchNorm1d(7*7*128),
    # Use Unflatten() to reshape into Image Tensor of shape 7, 7, 128
    Unflatten(N=-1, C=128, H=7, W=7),
    # ConvTranspose2d: 64 filters of 4x4, stride 2, 'same' padding (use padding=1)
    nn.ConvTranspose2d(128, 64, 4, stride=2, padding=1),
    nn.ReLU(),
    nn.BatchNorm2d(64),
    # ConvTranspose2d: 1 filter of 4x4, stride 2, 'same' padding (use padding=1)
    nn.ConvTranspose2d(64, 1, 4, stride=2, padding=1),
    nn.Tanh(),
    # Should have a 28x28x1 image, reshape back into 784 vector (using Flatten())
    nn.Flatten()
)

    # *****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, 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.
    """
    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_error.item()))
                imgs_numpy = fake_images.data.cpu().numpy()
                images.append(imgs_numpy[0:16])

            iter_count += 1

    return images
```

```python


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

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

```