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

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

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

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/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**.

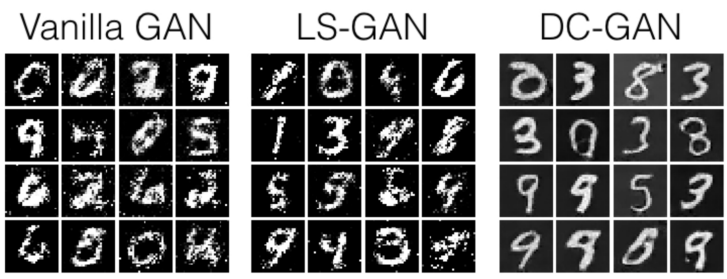
While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator 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:

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




```
[ ]
# Setup cell.
import numpy as np
import torch
```

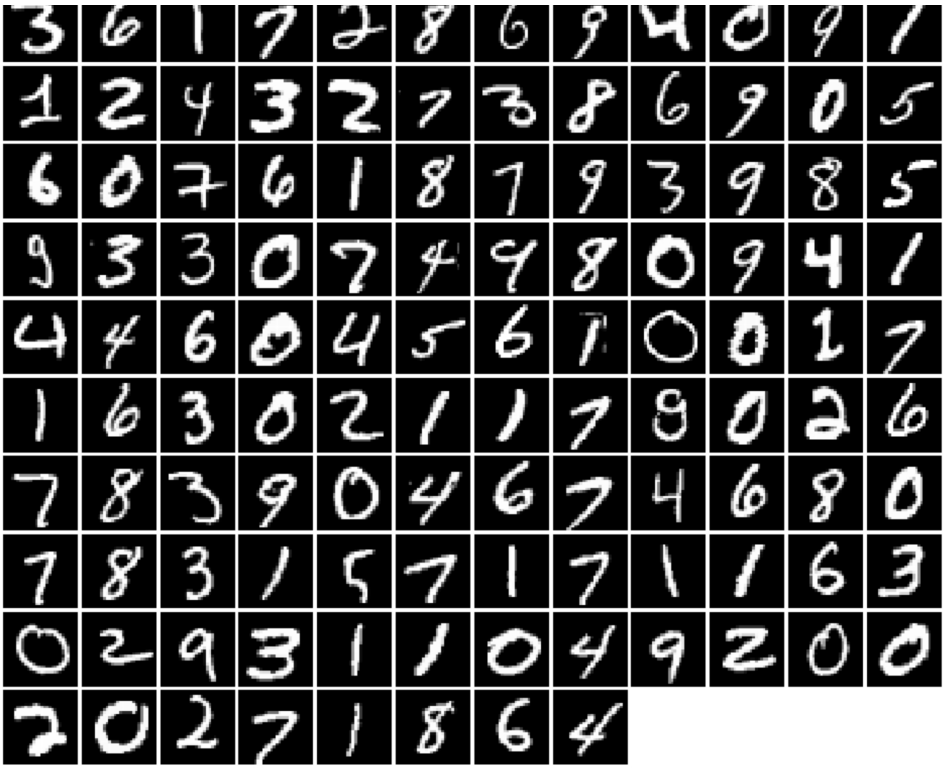
```
import torch.nn as nn
from torch.nn import init
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
import torchvision.datasets as dset
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from gan import preprocess_img, deprocess_img, rel_error, count_params, ChunkSampler

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

%load_ext autoreload
%autoreload 2

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

    fig = plt.figure(figsize=(sqrtn, sqrtn))
    gs = gridspec.GridSpec(sqrtn, sqrtn)
    gs.update(wspace=0.05, hspace=0.05)

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

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

## Dataset

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

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

[ ]
```
NUM_TRAIN = 50000
NUM_VAL = 5000

NOISE_DIM = 96
batch_size = 128

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

mnist_val = dset.MNIST(
    './nndl2',
    train=True,
    download=True,
    transform=T.ToTensor()
)
loader_val = DataLoader(
    mnist_val,
    batch_size=batch_size,
    sampler=ChunkSampler(NUM_VAL, NUM_TRAIN)
)

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



## 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:

```
[ ]
```

```python
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.

```
[ ]
```

```python
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 $\alpha$; 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:

```
[ ]
```

```python
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:

```
[ ]
```

```python
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.

```
[ ]
```

```python
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.

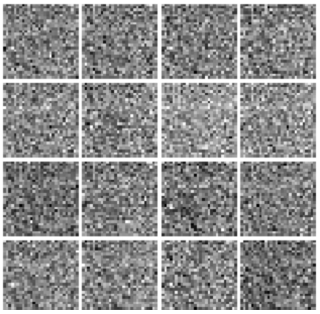
```
from gan import get_optimizer, run_a_gan

# 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
```

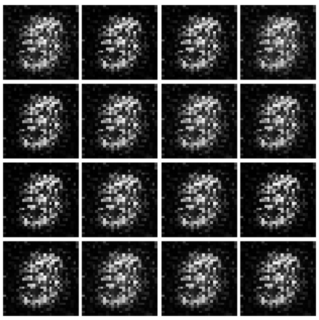
Run the cell below to show the generated images.

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

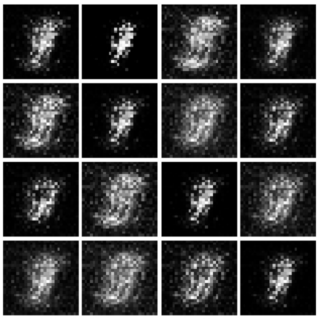
```
Iter: 0
```



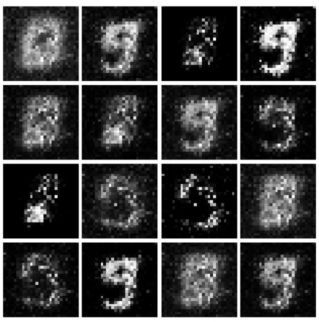
Iter: 250



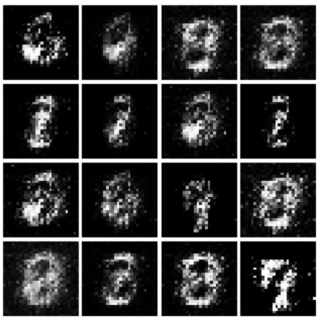
Iter: 500



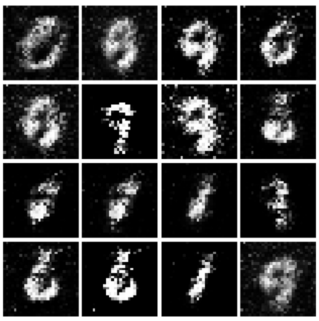
Iter: 750



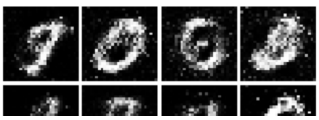
Iter: 1000



Iter: 1250



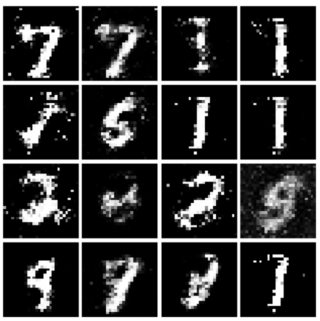
Iter: 1500



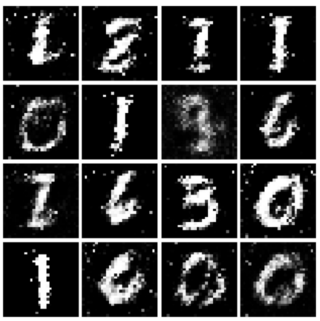
Iter: 1750

Iter: 2000

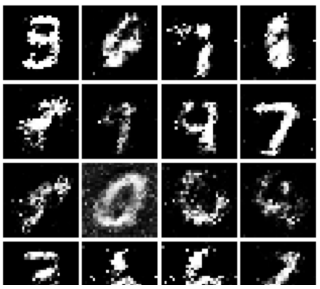
Iter: 2250



Iter: 2500



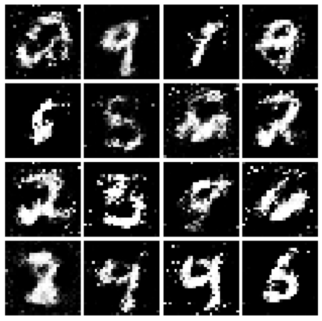
Iter: 2750



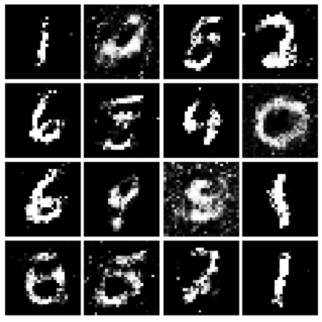
Iter: 3000

Iter: 3250

Iter: 3500



Iter: 3750

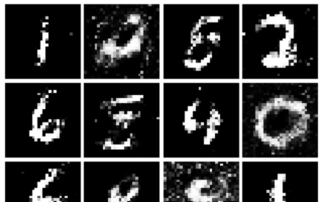


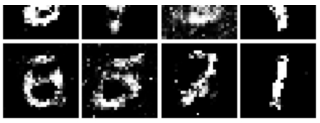
## Inline Question 1

What does your final vanilla GAN image look like?

```
# 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:

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

**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`).

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
```