Lab Session 2

Generative Adversarial Networks (GANs)

In this lab session, we will explore **generative models** and in particular, we will build a Generative Adversarial Network. GANs are basically neural networks that learn to generate samples (synthetic data) that are very similar to some input data. In this exercise, we build and train a simple GAN that can generate new images (handwritten digits) that resemble a set of training images (taken from MNIST dataset). In a way, we are teaching a neural network how to write.

Recap on GANs

GANs consist of two different neural networks (models). The first network is called the **discriminator** and is a standard binary classifier that determines whether an image looks like a real one (taken from the dataset) or a fake one (not present in the training set). The second network is called the **generator**. It takes random noise as input and transforms it into images using a neural network. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

The continuous process of the generator (G) trying to fool the discriminator (D), and the discriminator trying to correctly classify real vs. fake can be described as a minimax game:

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

where $x \sim p_{\text{data}}$ are samples from the input data, $z \sim p(z)$ 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. This minimax game is shown to be related to minimizing the Jensen-Shannon divergence between the training data distribution and the generated samples from G.

Figure 1: image.png

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

As we discussed during the lecture, we update the generator as maximizing the probability of the **discriminator making the incorrect choice**. That way, we alleviate the vanishing gradient problem of the generator. Therefore, in this exercise, we alternate the following updates: 1. Update G to maximize the probability of the discriminator making the incorrect choice on generated data:

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

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

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

Lab Session Objective and Output:

You will need to complete the notebooks by performing all tasks (7 implementation tasks and 2 questions related to a qualitative assessment of your results).

Initial Setup and Imports

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
#Tensorflow is imported as tf using that way to avoid any compability issues
between TF versions

import numpy as np

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

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'inferno'

# Functions for plotting and training
def show_images(images):
    images = np.reshape(images, [images.shape[0], -1])
    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 img_show():
    config = tf.ConfigProto()
    show = tf.Session(config=config)
    return show
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Loading Dataset (MNIST data)

For simplicity and computational convenience, we use the MNIST dataset. This dataset contains real handwritten digits and 70,000 images of handwritten digits compiled by the U.S. National Institute of Standards and Technology from Census Bureau employees and high school students. Each picture contains a centered image of white digit on black background (0 through 9).

```
class MNIST(object):
    def __init__(self, batch_size, shuffle=False):
        """
        Inputs:
        - batch_size: Integer giving number of elements per minibatch
        - shuffle: (optional) Boolean, whether to shuffle the data on each epoch
        """
        train, _ = tf.keras.datasets.mnist.load_data()
        X, y = train
        X = X.astype(np.float32)/255
        X = X.reshape((X.shape[0], -1))
        self.X, self.y = X, y
        self.batch_size, self.shuffle = batch_size, shuffle

    def __iter__(self):
        N, B = self.X.shape[0], self.batch_size
        idxs = np.arange(N)
        if self.shuffle:
            np.random.shuffle(idxs)
        return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))

# show a batch
```

```
mnist = MNIST(batch_size=16)
show_images(mnist.X[:16])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Activation Function

We implement below the neural network function. To avoid potential issues from ReLU, we will implement Leaky ReLU.

```
def leaky_relu(x, alpha=0.01):
    """Compute the leaky ReLU activation function.
    Inputs:
    - x: TensorFlow Tensor with arbitrary shape
    - alpha: leak parameter for leaky ReLU
    Output:
    - y: a TensorFlow Tensor with the same shape as x
    """
    # TASK 1 TODO: Implement a leaky ReLU activation function
    # Your code here
    y = tf.nn.leaky_relu(x,alpha)
    return y
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Random Noise

In the cell below, we will generate a random uniform noise from -1 to 1 with shape `[batch_size, dim]`.

```
def random_noise(batch_size, dim):
    """
    Inputs:
    - batch_size: integer giving the batch size of noise to generate
    - dim: integer giving the dimension of the the noise to generate
    Output:
    - noise: a TensorFlow Tensor containing uniform noise in [-1, 1] with shape
[batch_size, dim]
    """
    # Task 2 TODO: sample from and return random noise (uniform)
    noise = tf.random.uniform([batch_size,dim],minval = -1,maxval = 1)
    return noise
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Discriminator

First, we will implement the first neural network, the discriminator. To build the model, we will use the layers in `tf.layers`. The architecture we adopt is as follows: * Fully connected layer from size 784 to 256 * LeakyReLU with alpha 0.01 * Fully connected layer from 256 to 256 * LeakyReLU with alpha 0.01 * Fully connected layer from 256 to 1

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

```
def discriminator(x):
    """Compute discriminator score for a batch of input images.
    Input:
    - x: TensorFlow Tensor of flattened input images, shape [batch_size, 784]
    Output:
    - logits: a TensorFlow Tensor with shape [batch_size, 1], containing the
score
    for an image being real for each input image.
    """
    with tf.variable_scope("discriminator"):
        # Task 3 TODO: implement the discriminator NN architecture as described
        above
        # Layer 1: Fully connected from 784 to 256
        hidden1 = tf.layers.dense(x, 256, name='fc1')
        # LeakyReLU Activation with alpha = 0.01
```

```
        act1 = tf.nn.leaky_relu(hidden1, alpha=0.01, name='leaky_relu1')
        # Layer 2: Fully connected from 256 to 256
        hidden2 = tf.layers.dense(act1, 256, name='fc2')
        # LeakyReLU Activation with alpha = 0.01
        act2 = tf.nn.leaky_relu(hidden2, alpha=0.01, name='leaky_relu2')
        # Output Layer: Fully connected from 256 to 1
        logits = tf.layers.dense(act2, 1, name='output')
    return logits
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Generator

Second, we will build the generator in a similar way as the discriminator. The architecure we adopt is: * Fully connected layer from tf.shape(z)[1] (the number of noise dimensions) to 1024 * ReLU * Fully connected layer from 1024 to 1024 * ReLU * Fully connected layer from 1024 to 784 * Tanh

```
def generator(z):
    """Generate images from a random noise vector.
    Input:
    - z: TensorFlow Tensor of random noise with shape [batch_size, noise_dim]
    Output:
    - img: a TensorFlow Tensor of generated images, with shape [batch_size, 784].
    """
    with tf.variable_scope("generator"):
        # Task 4 TODO: implement the generator NN architecture as described above
        # Layer 1: Fully connected from noise_dim to 1024
        hidden1 = tf.layers.dense(z, 1024, name='fc1')
        # ReLU Activation
        act1 = tf.nn.relu(hidden1, name='relu1')
        # Layer 2: Fully connected from 1024 to 1024
        hidden2 = tf.layers.dense(act1, 1024, name='fc2')
        # ReLU Activation
        act2 = tf.nn.relu(hidden2, name='relu2')
        # Output Layer: Fully connected from 1024 to 784 (image size)
        logits = tf.layers.dense(act2, 784, name='output')
        # Tanh Activation to scale output to [-1, 1]
        img = tf.nn.tanh(logits, name='tanh')
    return img
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

GAN Loss

We will implement below the loss of the GAN.

The generator loss is:

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

The discriminator loss is:

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

For computing the loss function, we will use the sigmoid cross entropy loss.

```
def gan_loss(logits_real, logits_fake):
    """
    Inputs:
    - logits_real: Tensor, shape [batch_size, 1], output of discriminator
        Log probability that the image is real for each real image
    - logits_fake: Tensor, shape[batch_size, 1], output of discriminator
        Log probability that the image is real for each fake image

    Outputs:
    - D_loss: discriminator loss [scalar]
    - G_loss: generator loss [scalar]
    """
    # Task 5 TODO: Compute the loss functions: D_loss and G_loss
    # Discriminator Loss
    D_loss_real = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=logits_real, labels=tf.ones_like(logits_real)
        )
    )
    D_loss_fake = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=logits_fake, labels=tf.zeros_like(logits_fake)
        )
    )
    D_loss = D_loss_real + D_loss_fake
    # Generator Loss
    G_loss = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=logits_fake, labels=tf.ones_like(logits_fake)
        )
    )
    return D_loss, G_loss
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Optimizer

For the loss optimization, we will implement an `AdamOptimizer` with a 1e-3 learning rate, beta1=0.5 to mininize G_loss and D_loss separately. If you want to see a 'pathological' mode in GANs, you can beta1=0.9. That way, the discriminator loss might go to zero (i.e., learns too fast) and the generator might fail completely to learn. You can also experiment with other optimizers (e.g., SGD with Momentum or RMSProp).

```
# Task 6 TODO: Implement an AdamOptimizer for D_opt and G_opt
def gan_optimizers(learning_rate=1e-3, beta1=0.5):
    """Create optimizers for GAN training.
    Inputs:
    - learning_rate: learning rate to use for both optimizers
    - beta1: beta parameter for both optimizers (first moment decay)

    Outputs:
    - D_opt: instance of tf.train.AdamOptimizer with correct learning_rate and
beta1
    - G_opt: instance of tf.train.AdamOptimizer with correct learning_rate and
beta1
    """
    D_opt = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=beta1,
name='Adam_D')
    G_opt = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=beta1,
name='Adam_G')
    return D_opt, G_opt
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

GAN Composition

In the below cell, we compose the generator and discriminator by using the previous functions

```
tf.reset_default_graph()

# number of images for each batch
batch_size = 128
# random noise dimension
noise_dim = 96
```

```
# placeholder for images from the training dataset
x = tf.placeholder(tf.float32, [None, 784])
# random noise fed into our generator
z = random_noise(batch_size, noise_dim)
# generated images
G_sample = generator(z)

with tf.variable_scope("") as scope:
    #scale images to be -1 to 1
    logits_real = discriminator(2 * x - 1.0)
    # Re-use discriminator weights on new inputs
    scope.reuse_variables()
    logits_fake = discriminator(G_sample)

# Get the list of variables for the discriminator and generator
D_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'discriminator')
G_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'generator')

# get our solver
D_opt, G_opt = gan_optimizers()

# get our loss
D_loss, G_loss = gan_loss(logits_real, logits_fake)

# setup training steps
D_train_step = D_opt.minimize(D_loss, var_list=D_vars)
G_train_step = G_opt.minimize(G_loss, var_list=G_vars)
D_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS, 'discriminator')
G_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS, 'generator')
```

```
UserWarning: `tf.layers.dense` is deprecated and will be removed in a future
version. Please use `tf.keras.layers.Dense` instead.
  hidden1 = tf.layers.dense(z, 1024, name='fc1')
<ipython-input-24-7f0a16f43c23>:15: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  hidden2 = tf.layers.dense(act1, 1024, name='fc2')
<ipython-input-24-7f0a16f43c23>:19: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  logits = tf.layers.dense(act2, 784, name='output')
<ipython-input-23-d2129b935812>:12: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  hidden1 = tf.layers.dense(x, 256, name='fc1')
```

```
<ipython-input-23-d2129b935812>:16: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  hidden2 = tf.layers.dense(act1, 256, name='fc2')
<ipython-input-23-d2129b935812>:20: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  logits = tf.layers.dense(act2, 1, name='output')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Time to train our GAN!

It is not time to start training our GAN. We use a simple procedure. We train D(x) and G(z) with one batch each every iteration. Training could take several minutes if run on a CPU.

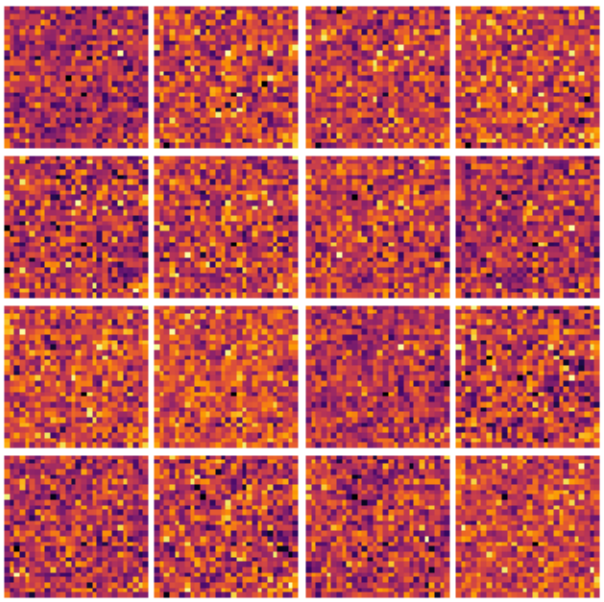
```
# the main function to train a GAN

def run_a_gan(sess, G_train_step, G_loss, D_train_step, D_loss, G_extra_step,
D_extra_step,
              show_every=2, print_every=1, batch_size=128, num_epoch=10):
    """Train a GAN for a certain number of epochs.
    Inputs:
    - sess: A tf.Session that we want to use to run our data
    - G_train_step: A training step for the Generator
    - G_loss: Generator loss
    - D_train_step: A training step for the Generator
    - D_loss: Discriminator loss
    - G_extra_step: A collection of tf.GraphKeys.UPDATE_OPS for generator
    - D_extra_step: A collection of tf.GraphKeys.UPDATE_OPS for discriminator
    """
    # compute the number of iterations needed
    mnist = MNIST(batch_size=batch_size, shuffle=True)
    for epoch in range(num_epoch):
        # show a sample result
        if epoch % show_every == 0:
            samples = sess.run(G_sample)
            fig = show_images(samples[:16])
            plt.show()
            print()
        for (minibatch, minbatch_y) in mnist:
            # run a batch of data through the network
            _, D_loss_curr = sess.run(
                [D_train_step, D_loss], feed_dict={x: minibatch})
            _, G_loss_curr = sess.run([G_train_step, G_loss])
```

```
        # We want to make sure D_loss doesn't go to 0
        if epoch % print_every == 0:
            print('Epoch: {}, D: {:.4}, G:{:.4}'.format(
                epoch, D_loss_curr, G_loss_curr))
    print('Final images')
    samples = sess.run(G_sample)

    fig = show_images(samples[:16])
    plt.show()
```

```
<IPython.core.display.Javascript object>
```

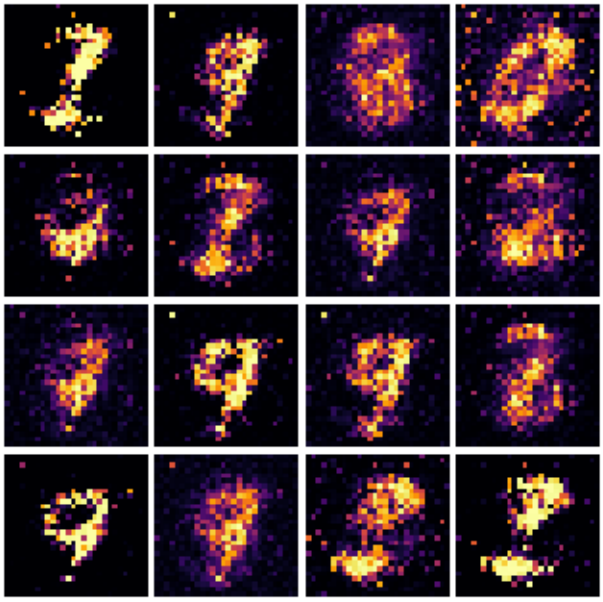
```
<IPython.core.display.Javascript object>
```

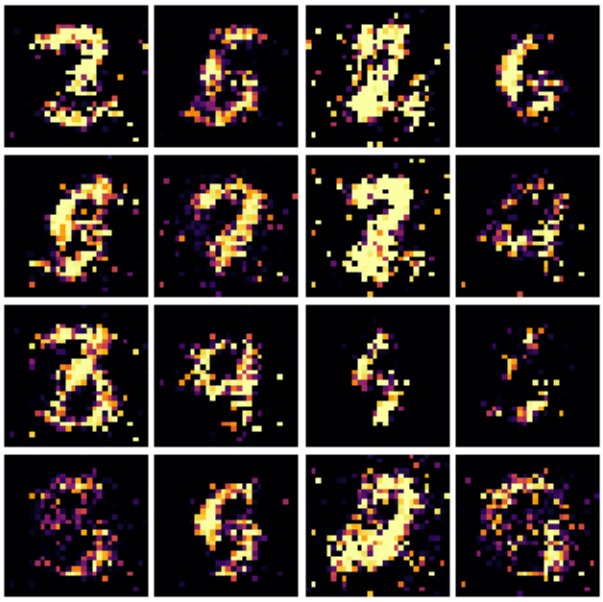
```
with img_show() as sess:
    sess.run(tf.global_variables_initializer())
    run_a_gan(sess, G_train_step, G_loss, D_train_step,
              D_loss, G_extra_step, D_extra_step)
```

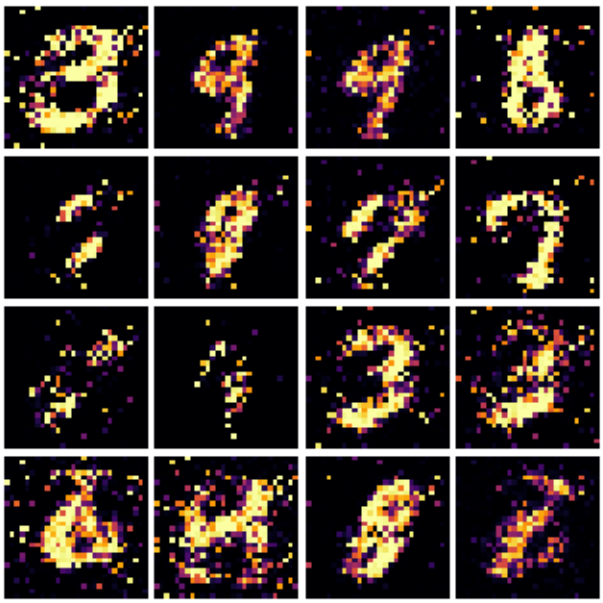


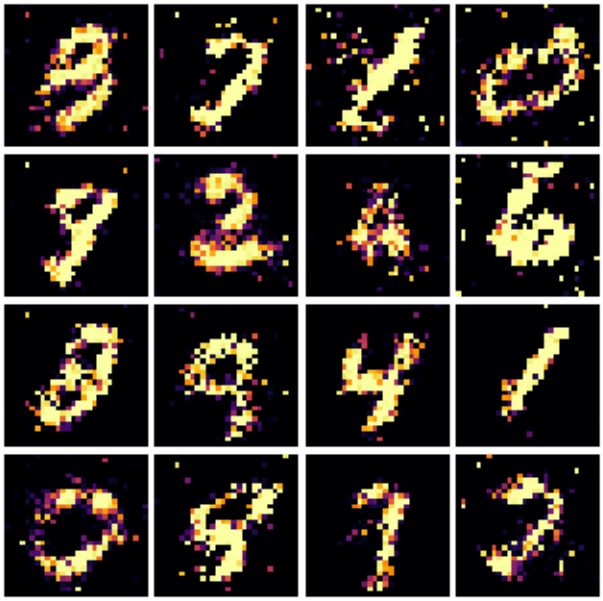
```
Epoch: 0, D: 2.117, G:1.037
Epoch: 1, D: 1.492, G:3.241

Epoch: 2, D: 1.219, G:1.266
Epoch: 3, D: 1.101, G:1.045

Epoch: 4, D: 1.227, G:0.9846
```

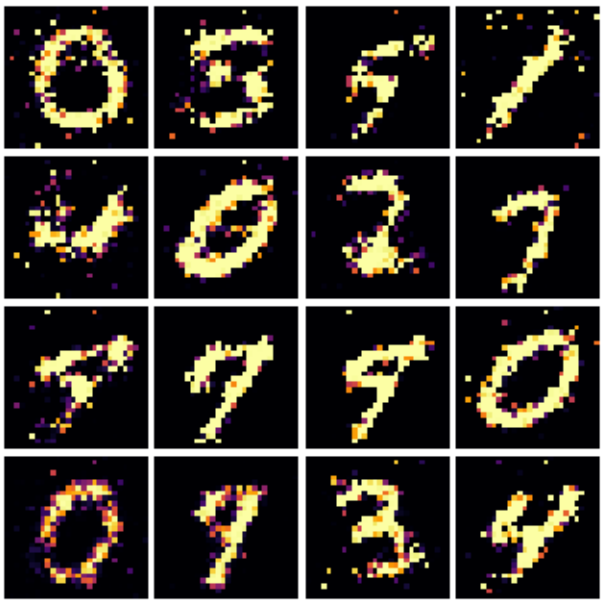
```
Epoch: 5, D: 1.266, G:0.841

Epoch: 6, D: 1.328, G:0.8536
Epoch: 7, D: 1.139, G:0.841

Epoch: 8, D: 1.201, G:0.8322
Epoch: 9, D: 1.302, G:0.7117
Final images
```











```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

GAN2: Changing the loss function and the divergence

We will now implement a GAN with a different loss function. Specifically, we will implement the following objective functions for the generator loss:

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

and the discriminator loss:

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

This GAN variant is known as Least Squares GAN, which substitutes the binary cross entropy loss with a least square (ℓ_2) loss, which has better properties for optimization and is less likely to saturate. It has been shown that minimizing the above objective functions yields minimizing the Pearson χ^2 divergence.

```
def gan2_loss(score_real, score_fake):
    """
    Inputs:
    - score_real: Tensor, shape [batch_size, 1], output of discriminator
        score for each real image
    - score_fake: Tensor, shape[batch_size, 1], output of discriminator
        score for each fake image
```

```
    Outputs:
    - D_loss: discriminator loss scalar
    - G_loss: generator loss scalar
    """
    # Task 7 TODO: Compute the loss function D_loss and G_loss for the second GAN
    architecture
    # Discriminator Loss
    D_loss_real = tf.reduce_mean(0.5 * tf.square(score_real - 1))
    D_loss_fake = tf.reduce_mean(0.5 * tf.square(score_fake))
    D_loss = D_loss_real + D_loss_fake
    # Generator Loss
    G_loss = tf.reduce_mean(0.5 * tf.square(score_fake - 1))
    return D_loss, G_loss
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Create new training steps to minimize the new GAN loss:

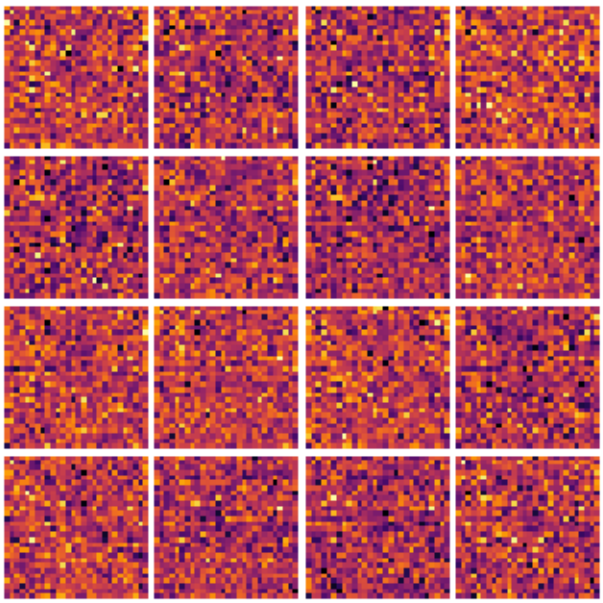
```
D_loss, G_loss = gan2_loss(logits_real, logits_fake)
D_train_step = D_opt.minimize(D_loss, var_list=D_vars)
G_train_step = G_opt.minimize(G_loss, var_list=G_vars)
```

```
<IPython.core.display.Javascript object>
```

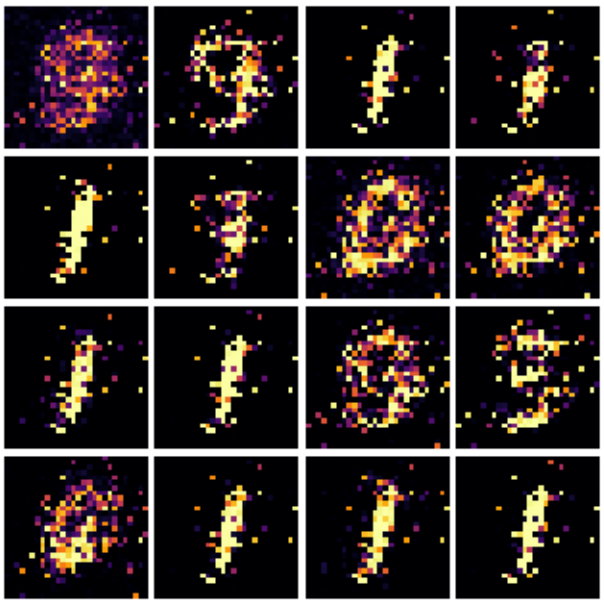
```
<IPython.core.display.Javascript object>
```

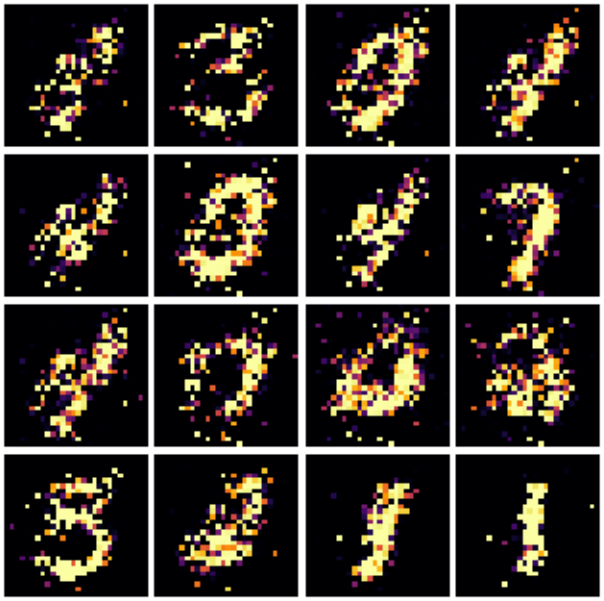
Run the below cell to train the model:

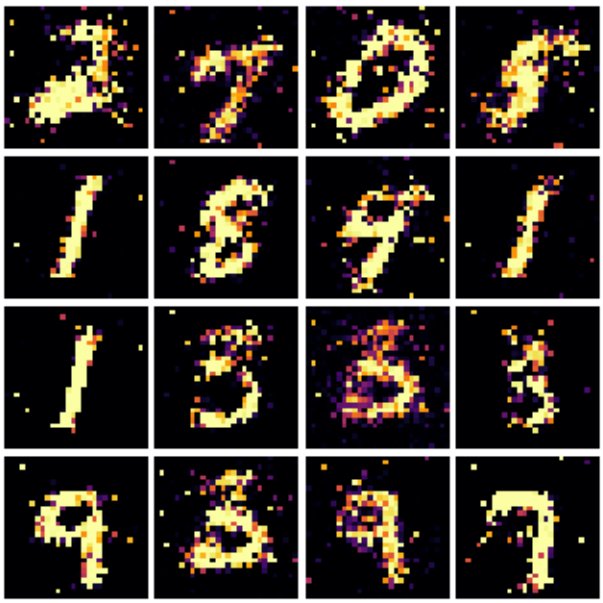
```
with img_show() as ses:
    ses.run(tf.global_variables_initializer())
    run_a_gan(ses, G_train_step, G_loss, D_train_step, D_loss, G_extra_step,
D_extra_step)
```

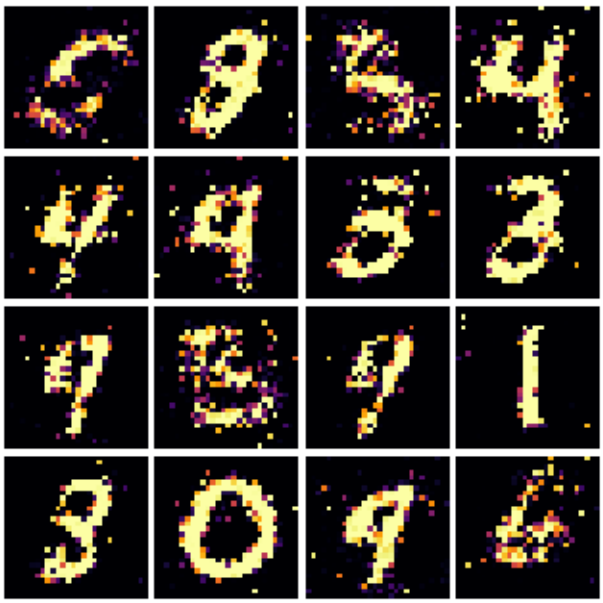


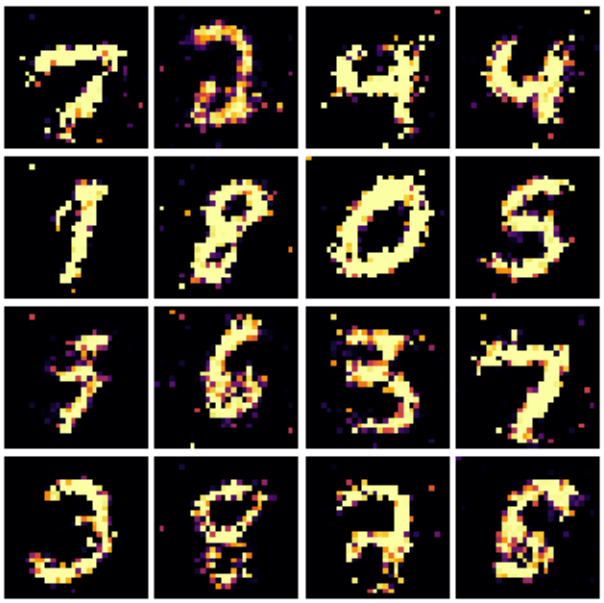
```
Epoch: 0, D: 0.2133, G:0.199
Epoch: 1, D: 0.1757, G:0.3375

Epoch: 2, D: 0.4489, G:0.4187
Epoch: 3, D: 0.2118, G:0.2342

Epoch: 4, D: 0.1765, G:0.2402
Epoch: 5, D: 0.2372, G:0.1965

Epoch: 6, D: 0.2118, G:0.1855
Epoch: 7, D: 0.2284, G:0.1737

Epoch: 8, D: 0.1999, G:0.1867
Epoch: 9, D: 0.2353, G:0.182
Final images
```











```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Qualitative Results:

Task 8 TODO: Please comment on the visual quality of the samples and how the results change over different training runs.

Task 9 TODO: Re-train the first GAN architecture with random Gaussian noise instead of uniform. Please comment on the performance of the GAN and on the visual quality of the samples (during the course of the training).

Task 8: Please comment on the visual quality of the samples and how the results change over different training runs.

The visual quality of the generated images in both GAN implementations is poor overall, not yet resembling the original digits in the MNIST dataset. The visual quality does show improvement over the training epochs, suggesting that further training could lead to even better results.

The quality tends to fluctuate during training. Sometimes, the generator seems to make slight progress, producing images that look almost like digits. However, this progress is often followed by periods where the images become more noise-like. This instability is a common challenge in GAN training, as the generator and discriminator try to outsmart each other in a dynamic process.

The Least Squares GAN generally produces visually sharper and more distinct images compared to the original GAN with the standard cross-entropy loss. The images generated by the LSGAN appear less noisy and have more defined features.

The limited architecture of the generator and discriminator, along with the relatively small number of training epochs, contribute to the poor quality of the results.

Task 9: Re-train the first GAN architecture with random Gaussian noise instead of uniform. Please comment on the performance of the GAN and on the visual quality of the samples (during the course of the training).

```
def random_noise_gaussian(batch_size, dim):
    """
    Generate random Gaussian noise.

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

    Output:
    - noise: A TensorFlow Tensor containing Gaussian noise with shape
[batch_size, dim].
    """
    noise = tf.random.normal([batch_size, dim])
    return noise
```

```
<IPython.core.display.Javascript object>
```

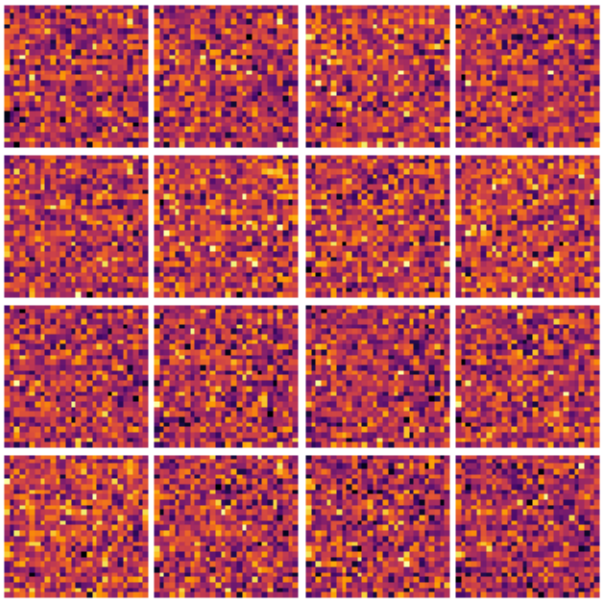
```
<IPython.core.display.Javascript object>
```

```
tf.reset_default_graph()

# number of images for each batch
batch_size = 128
# random noise dimension
noise_dim = 96

# placeholder for images from the training dataset
x = tf.placeholder(tf.float32, [None, 784])
# random noise fed into our generator
z = random_noise_gaussian(batch_size, noise_dim)
# generated images
G_sample = generator(z)

with tf.variable_scope("") as scope:
    #scale images to be -1 to 1
    logits_real = discriminator(2 * x - 1.0)
    # Re-use discriminator weights on new inputs
    scope.reuse_variables()
    logits_fake = discriminator(G_sample)
```

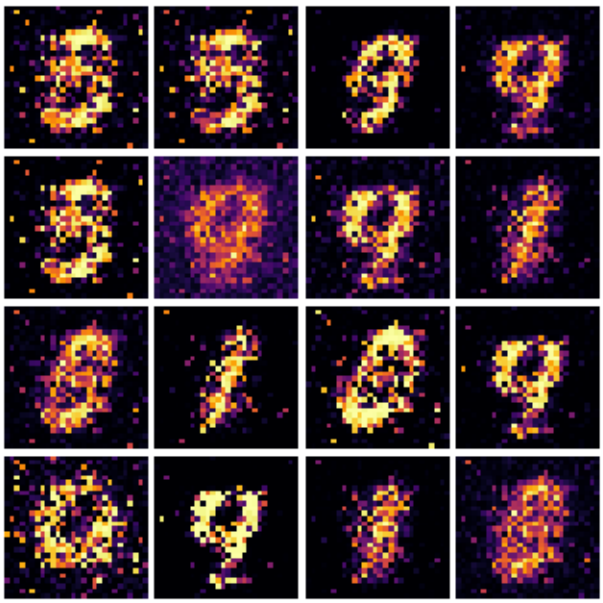
```
# Get the list of variables for the discriminator and generator
D_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'discriminator')
G_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'generator')

# get our solver
D_opt, G_opt = gan_optimizers()

# get our loss
D_loss, G_loss = gan_loss(logits_real, logits_fake)

# setup training steps
D_train_step = D_opt.minimize(D_loss, var_list=D_vars)
G_train_step = G_opt.minimize(G_loss, var_list=G_vars)
D_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS, 'discriminator')
G_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS, 'generator')
```

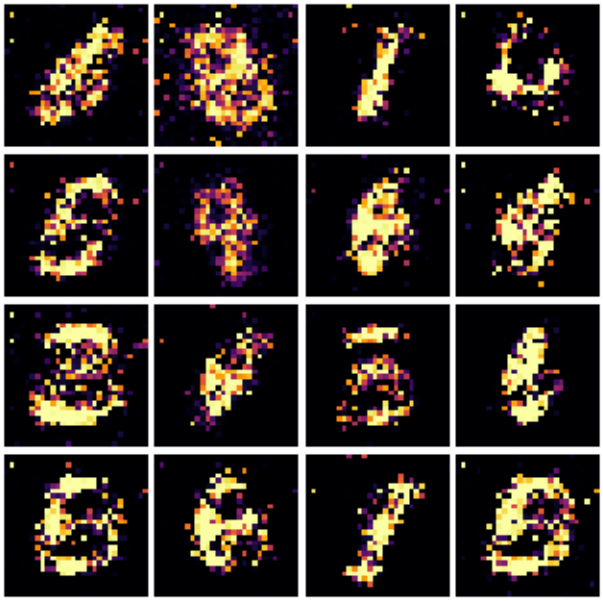
```
UserWarning: `tf.layers.dense` is deprecated and will be removed in a future
version. Please use `tf.keras.layers.Dense` instead.
  hidden1 = tf.layers.dense(z, 1024, name='fc1')
<ipython-input-24-7f0a16f43c23>:15: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  hidden2 = tf.layers.dense(act1, 1024, name='fc2')
<ipython-input-24-7f0a16f43c23>:19: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  logits = tf.layers.dense(act2, 784, name='output')
<ipython-input-23-d2129b935812>:12: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  hidden1 = tf.layers.dense(x, 256, name='fc1')
<ipython-input-23-d2129b935812>:16: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  hidden2 = tf.layers.dense(act1, 256, name='fc2')
<ipython-input-23-d2129b935812>:20: UserWarning: `tf.layers.dense` is deprecated
and will be removed in a future version. Please use `tf.keras.layers.Dense`
instead.
  logits = tf.layers.dense(act2, 1, name='output')

<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>
```

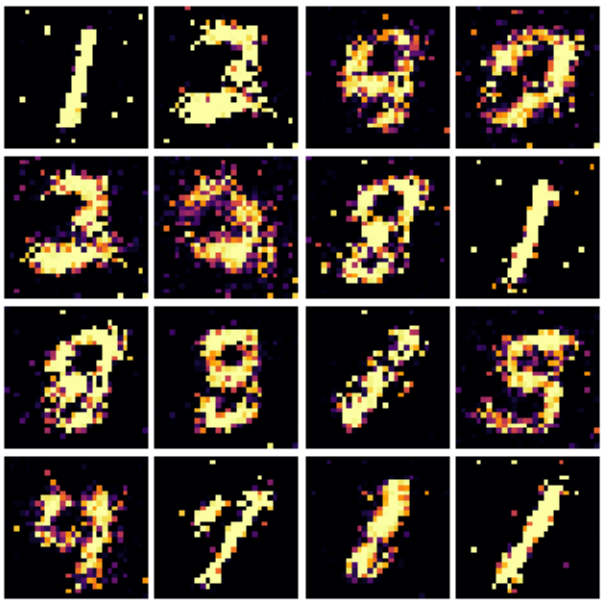
```
with img_show() as se:
    se.run(tf.global_variables_initializer())
    run_a_gan(se, G_train_step, G_loss, D_train_step, D_loss, G_extra_step,
D_extra_step)
```

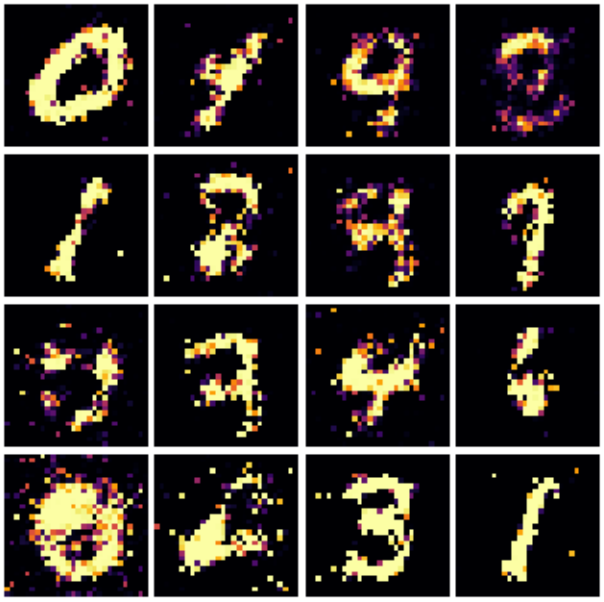


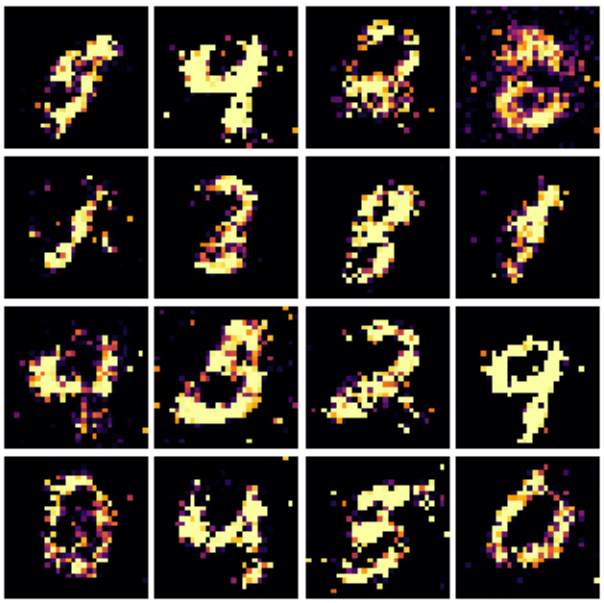
```
Epoch: 0, D: 1.59, G:0.658
Epoch: 1, D: 1.346, G:0.873

Epoch: 2, D: 1.15, G:1.462
Epoch: 3, D: 1.376, G:1.056

Epoch: 4, D: 1.291, G:0.9986
Epoch: 5, D: 1.487, G:0.7303

Epoch: 6, D: 1.487, G:0.7889
Epoch: 7, D: 1.335, G:0.7094

Epoch: 8, D: 1.31, G:0.7508
Epoch: 9, D: 1.391, G:0.8199
Final images
```











```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```