dlnd_face_generation

July 26, 2017

1 Face Generation

In this project, you'll use generative adversarial networks to generate new images of faces. ### Get the Data You'll be using two datasets in this project: - MNIST - CelebA

Since the celebA dataset is complex and you're doing GANs in a project for the first time, we want you to test your neural network on MNIST before CelebA. Running the GANs on MNIST will allow you to see how well your model trains sooner.

If you're using FloydHub, set `data_dir` to "/input" and use the FloydHub data ID "R5KrjnANiKVhLWAkpXhNBe".

In [1]:
```
data_dir = './data'

# FloydHub - Use with data ID "R5KrjnANiKVhLWAkpXhNBe"
#data_dir = '/input'


"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import helper

helper.download_extract('mnist', data_dir)
helper.download_extract('celeba', data_dir)
```

```
Found mnist Data
Found celeba Data
```

1.1 Explore the Data

1.1.1 MNIST

As you're aware, the MNIST dataset contains images of handwritten digits. You can view the first number of examples by changing `show_n_images`.

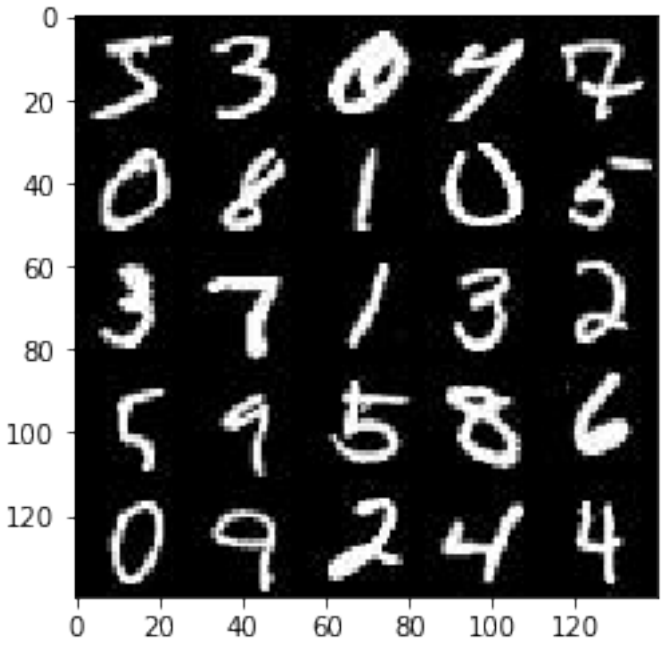
In [2]:
```
show_n_images = 25

"""
```

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
%matplotlib inline
import os
from glob import glob
from matplotlib import pyplot

mnist_images = helper.get_batch(glob(os.path.join(data_dir, 'mnist/*.jpg'))[:show_n_ima
pyplot.imshow(helper.images_square_grid(mnist_images, 'L'), cmap='gray')
```

Out[2]: <matplotlib.image.AxesImage at 0x7ebd828>



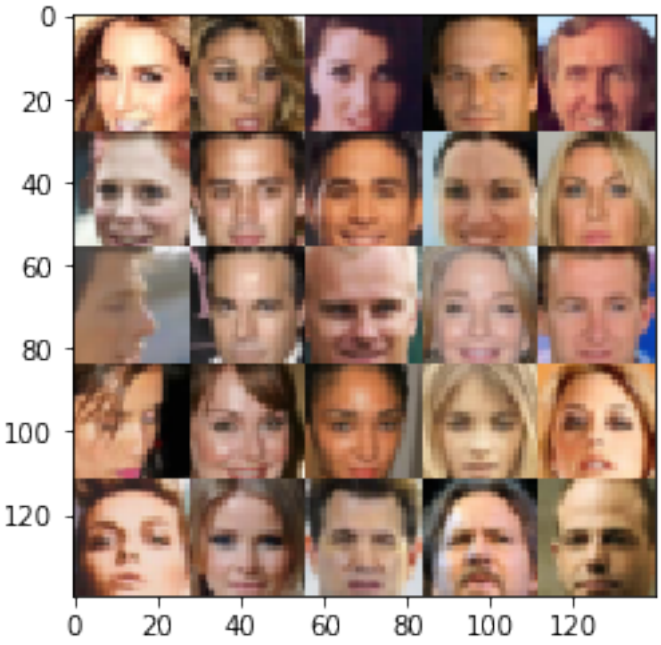
1.1.2 CelebA

The CelebFaces Attributes Dataset (CelebA) dataset contains over 200,000 celebrity images with annotations. Since you're going to be generating faces, you won't need the annotations. You can view the first number of examples by changing `show_n_images`.

In [3]:
```
show_n_images = 25

"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
mnist_images = helper.get_batch(glob(os.path.join(data_dir, 'img_align_celeba/*.jpg'))
pyplot.imshow(helper.images_square_grid(mnist_images, 'RGB'))
```

```
Out[3]: <matplotlib.image.AxesImage at 0x8f44f28>
```



1.2 Preprocess the Data

Since the project's main focus is on building the GANs, we'll preprocess the data for you. The values of the MNIST and CelebA dataset will be in the range of -0.5 to 0.5 of 28x28 dimensional images. The CelebA images will be cropped to remove parts of the image that don't include a face, then resized down to 28x28.

The MNIST images are black and white images with a single [color channel](https://en.wikipedia.org/wiki/Channel_(digital_image%29) while the CelebA images have [3 color channels (RGB color channel)](https://en.wikipedia.org/wiki/Channel_(digital_image%29#RGB_Images). ## Build the Neural Network You'll build the components necessary to build a GANs by implementing the following functions below: - `model_inputs` - `discriminator` - `generator` - `model_loss` - `model_opt` - `train`

1.2.1 Check the Version of TensorFlow and Access to GPU

This will check to make sure you have the correct version of TensorFlow and access to a GPU

```
In [4]: """
        DON'T MODIFY ANYTHING IN THIS CELL
        """
        from distutils.version import LooseVersion
        import warnings
        import tensorflow as tf
```

```
    # Check TensorFlow Version
    assert LooseVersion(tf.__version__) >= LooseVersion('1.0'), 'Please use TensorFlow vers
    print('TensorFlow Version: {}'.format(tf.__version__))

    # Check for a GPU
    if not tf.test.gpu_device_name():
        warnings.warn('No GPU found. Please use a GPU to train your neural network.')
    else:
        print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))
```

```
TensorFlow Version: 1.1.0
Default GPU Device: /gpu:0
```

1.2.2 Input

Implement the `model_inputs` function to create TF Placeholders for the Neural Network. It should create the following placeholders: - Real input images placeholder with rank 4 using `image_width`, `image_height`, and `image_channels`. - Z input placeholder with rank 2 using `z_dim`. - Learning rate placeholder with rank 0.

Return the placeholders in the following the tuple (tensor of real input images, tensor of z data)

```
In [5]: import problem_unittests as tests

    def model_inputs(image_width, image_height, image_channels, z_dim):
        """
        Create the model inputs
        :param image_width: The input image width
        :param image_height: The input image height
        :param image_channels: The number of image channels
        :param z_dim: The dimension of Z
        :return: Tuple of (tensor of real input images, tensor of z data, learning rate)
        """
        inputs_real = tf.placeholder(tf.float32, (None, image_width, image_height, image_c
        inputs_z = tf.placeholder(tf.float32, (None, z_dim), name='input_z')
        learning_rate = tf.placeholder(tf.float32, (None))
        return inputs_real, inputs_z, learning_rate



    """
    DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
    """
    tests.test_model_inputs(model_inputs)
```

```
Tests Passed
```

1.2.3 Discriminator

Implement `discriminator` to create a discriminator neural network that discriminates on `images`. This function should be able to reuse the variables in the neural network. Use `tf.variable_scope` with a scope name of "discriminator" to allow the variables to be reused. The function should return a tuple of (tensor output of the discriminator, tensor logits of the discriminator).

```
In [6]: def discriminator(images, reuse=False):
            """
            Create the discriminator network
            :param images: Tensor of input image(s)
            :param reuse: Boolean if the weights should be reused
            :return: Tuple of (tensor output of the discriminator, tensor logits of the discri
            """
            alpha=0.2
            x = images
            with tf.variable_scope('discriminator', reuse=reuse):
                x = tf.layers.conv2d(x, 64, 4, strides=2, padding="same")
                x = tf.layers.batch_normalization(x, training=True)
                x = tf.maximum(alpha * x, x)
                #x = tf.layers.dropout(x, 0.5)

                x = tf.layers.conv2d(x, 128, 4, strides=2, padding="same")
                x = tf.layers.batch_normalization(x, training=True)
                x = tf.maximum(alpha * x, x)
                #x = tf.layers.dropout(x, 0.5)

                x = tf.layers.conv2d(x, 256, 4, strides=2, padding="same")
                x = tf.layers.batch_normalization(x, training=True)
                x = tf.maximum(alpha * x, x)
                #x = tf.layers.dropout(x, 0.5)

                x = tf.reshape(x, (-1, 4 * 4 * 256))
                logits = tf.layers.dense(x, 1)
                out = tf.sigmoid(logits)

            return out, logits

        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        tests.test_discriminator(discriminator, tf)

Tests Passed
```

1.2.4 Generator

Implement `generator` to generate an image using `z`. This function should be able to reuse the variables in the neural network. Use `tf.variable_scope` with a scope name of "generator" to allow the variables to be reused. The function should return the generated 28 x 28 x `out_channel_dim` images.

```
In [7]: def generator(z, out_channel_dim, is_train=True):
            """
            Create the generator network
            :param z: Input z
            :param out_channel_dim: The number of channels in the output image
            :param is_train: Boolean if generator is being used for training
            :return: The tensor output of the generator
            """
            reuse = not is_train
            alpha= 0.2
            with tf.variable_scope('generator', reuse=reuse):
                x = tf.layers.dense(z, 4 * 4 * 512)

                x = tf.reshape(x, (-1, 4, 4, 512))
                x = tf.layers.batch_normalization(x, training=is_train)
                #x = tf.layers.dropout(x, 0.5)
                x = tf.maximum(alpha * x, x)
                #print(x.shape)
                x = tf.layers.conv2d_transpose(x, 256, 4, strides=1, padding="valid")
                x = tf.layers.batch_normalization(x,training=is_train)
                x = tf.maximum(alpha * x, x)
                #print(x.shape)
                x = tf.layers.conv2d_transpose(x, 128, 4, strides=2, padding="same")
                x = tf.layers.batch_normalization(x,training=is_train)
                x = tf.maximum(alpha * x, x)
                #print(x.shape)
                x = tf.layers.conv2d_transpose(x, out_channel_dim, 4, strides=2, padding="same"
                #x = tf.maximum(alpha * x, x)

                logits = x
                out = tf.tanh(logits)

            return out



        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        tests.test_generator(generator, tf)

Tests Passed
```

1.2.5 Loss

Implement `model_loss` to build the GANs for training and calculate the loss. The function should return a tuple of (discriminator loss, generator loss). Use the following functions you implemented: - `discriminator(images, reuse=False)` - `generator(z, out_channel_dim, is_train=True)`

```
In [8]: def model_loss(input_real, input_z, out_channel_dim):
            """
            Get the loss for the discriminator and generator
            :param input_real: Images from the real dataset
            :param input_z: Z input
            :param out_channel_dim: The number of channels in the output image
            :return: A tuple of (discriminator loss, generator loss)
            """
            smooth = 0.1
            _, d_logits_real = discriminator(input_real, reuse=False)
            fake = generator(input_z, out_channel_dim, is_train=True)
            d_logits_fake = discriminator(fake, reuse=True)
            # Calculate losses
            d_loss_real = tf.reduce_mean(
                                tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,
                                                                        labels=tf.ones_like(d_log
            d_loss_fake = tf.reduce_mean(
                                tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                                                        labels=tf.zeros_like(d_l
            d_loss = d_loss_real + d_loss_fake

            g_loss = tf.reduce_mean(
                            tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                                                    labels=tf.ones_like(d_logits_
            return d_loss, g_loss


        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        tests.test_model_loss(model_loss)
```

```
Tests Passed
```

1.2.6 Optimization

Implement `model_opt` to create the optimization operations for the GANs. Use `tf.trainable_variables` to get all the trainable variables. Filter the variables with names

that are in the discriminator and generator scope names. The function should return a tuple of (discriminator training operation, generator training operation).

```
In [9]: def model_opt(d_loss, g_loss, learning_rate, beta1):
            """
            Get optimization operations
            :param d_loss: Discriminator loss Tensor
            :param g_loss: Generator loss Tensor
            :param learning_rate: Learning Rate Placeholder
            :param beta1: The exponential decay rate for the 1st moment in the optimizer
            :return: A tuple of (discriminator training operation, generator training operatio
            """
            t_vars = tf.trainable_variables()
            g_vars = [var for var in t_vars if var.name.startswith('generator')]
            d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
            all_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

            g_update_ops = [var for var in all_update_ops if var.name.startswith('generator')]
            d_update_ops = [var for var in all_update_ops if var.name.startswith('discriminato

            with tf.control_dependencies(d_update_ops):
                d_train_opt = tf.train.AdamOptimizer(learning_rate, beta1=beta1).minimize(d_los
            with tf.control_dependencies(g_update_ops):
                g_train_opt = tf.train.AdamOptimizer(learning_rate, beta1=beta1).minimize(g_los
            return d_train_opt, g_train_opt


        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        tests.test_model_opt(model_opt, tf)
```

```
Tests Passed
```

1.3 Neural Network Training

1.3.1 Show Output

Use this function to show the current output of the generator during training. It will help you determine how well the GANs is training.

```
In [10]: """
         DON'T MODIFY ANYTHING IN THIS CELL
         """
         import numpy as np

         def show_generator_output(sess, n_images, input_z, out_channel_dim, image_mode):
             """
```

```
    Show example output for the generator
    :param sess: TensorFlow session
    :param n_images: Number of Images to display
    :param input_z: Input Z Tensor
    :param out_channel_dim: The number of channels in the output image
    :param image_mode: The mode to use for images ("RGB" or "L")
    """
    cmap = None if image_mode == 'RGB' else 'gray'
    z_dim = input_z.get_shape().as_list()[-1]
    example_z = np.random.uniform(-1, 1, size=[n_images, z_dim])

    samples = sess.run(
        generator(input_z, out_channel_dim, False),
        feed_dict={input_z: example_z})

    images_grid = helper.images_square_grid(samples, image_mode)
    pyplot.imshow(images_grid, cmap=cmap)
    pyplot.show()
```

1.3.2 Train

Implement `train` to build and train the GANs. Use the following functions you implemented: - `model_inputs(image_width, image_height, image_channels, z_dim)` - `model_loss(input_real, input_z, out_channel_dim)` - `model_opt(d_loss, g_loss, learning_rate, beta1)`

Use the `show_generator_output` to show `generator` output while you train. Running `show_generator_output` for every batch will drastically increase training time and increase the size of the notebook. It's recommended to print the `generator` output every 100 batches.

```
In [11]: def train(epoch_count, batch_size, z_dim, learning_rate, beta1, get_batches, data_sha
    """
    Train the GAN
    :param epoch_count: Number of epochs
    :param batch_size: Batch Size
    :param z_dim: Z dimension
    :param learning_rate: Learning Rate
    :param beta1: The exponential decay rate for the 1st moment in the optimizer
    :param get_batches: Function to get batches
    :param data_shape: Shape of the data
    :param data_image_mode: The image mode to use for images ("RGB" or "L")
    """
    inputs_real, inputs_z, lr = model_inputs(data_shape[1], data_shape[2], data_shape
    d_loss, g_loss = model_loss(inputs_real, inputs_z, data_shape[-1])
    d_train_opt, g_train_opt = model_opt(d_loss, g_loss, learning_rate, beta1)
    batch_num = 0

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
```

```
            for epoch_i in range(epoch_count):
                for batch_images in get_batches(batch_size):
                    batch_num = batch_num+1
                    batch_images = batch_images * 2
                    batch_z = np.random.uniform(-1, 1, size=(batch_size, z_dim))
                    _ = sess.run(d_train_opt, feed_dict={inputs_real: batch_images, inputs
                    _ = sess.run(g_train_opt, feed_dict={inputs_z: batch_z, lr:learning_ra

                    if batch_num % 100 == 0:
                        train_loss_d = d_loss.eval({inputs_z:batch_z, inputs_real: batch_i
                        train_loss_g = g_loss.eval({inputs_z:batch_z})
                        print("Epoch {}/{} batch {}...".format(epoch_i+1, epoch_count, bat
                          "Discriminator Loss: {:.4f}...".format(train_loss_d),
                          "Generator Loss: {:.4f}".format(train_loss_g))
```

1.3.3 MNIST

Test your GANs architecture on MNIST. After 2 epochs, the GANs should be able to generate images that look like handwritten digits. Make sure the loss of the generator is lower than the loss of the discriminator or close to 0.

In [12]:
```
batch_size = 64
z_dim = 100
learning_rate = 0.001
beta1 = 0.6


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
epochs = 2

mnist_dataset = helper.Dataset('mnist', glob(os.path.join(data_dir, 'mnist/*.jpg')))
with tf.Graph().as_default():
    train(epochs, batch_size, z_dim, learning_rate, beta1, mnist_dataset.get_batches,
          mnist_dataset.shape, mnist_dataset.image_mode)
```

```
Epoch 1/2 batch 100... Discriminator Loss: 0.7630... Generator Loss: 1.9291
Epoch 1/2 batch 200... Discriminator Loss: 1.0593... Generator Loss: 1.0565
Epoch 1/2 batch 300... Discriminator Loss: 1.0405... Generator Loss: 1.5586
Epoch 1/2 batch 400... Discriminator Loss: 1.4328... Generator Loss: 0.4926
Epoch 1/2 batch 500... Discriminator Loss: 1.0879... Generator Loss: 0.7879
Epoch 1/2 batch 600... Discriminator Loss: 1.1272... Generator Loss: 0.9465
Epoch 1/2 batch 700... Discriminator Loss: 1.4522... Generator Loss: 0.4740
Epoch 1/2 batch 800... Discriminator Loss: 1.2347... Generator Loss: 1.0109
Epoch 1/2 batch 900... Discriminator Loss: 1.1933... Generator Loss: 0.7435
Epoch 2/2 batch 1000... Discriminator Loss: 1.2134... Generator Loss: 1.1011
Epoch 2/2 batch 1100... Discriminator Loss: 1.8529... Generator Loss: 0.3481
```

```
Epoch 2/2 batch 1200... Discriminator Loss: 1.0882... Generator Loss: 0.8587
Epoch 2/2 batch 1300... Discriminator Loss: 1.3286... Generator Loss: 1.9796
Epoch 2/2 batch 1400... Discriminator Loss: 1.4972... Generator Loss: 0.4602
Epoch 2/2 batch 1500... Discriminator Loss: 1.1960... Generator Loss: 0.7907
Epoch 2/2 batch 1600... Discriminator Loss: 1.1038... Generator Loss: 0.7785
Epoch 2/2 batch 1700... Discriminator Loss: 1.6359... Generator Loss: 0.4039
Epoch 2/2 batch 1800... Discriminator Loss: 1.4698... Generator Loss: 0.5001
```

1.3.4 CelebA

Run your GANs on CelebA. It will take around 20 minutes on the average GPU to run one epoch. You can run the whole epoch or stop when it starts to generate realistic faces.

```
In [13]: batch_size = 64
         z_dim = 100
         learning_rate = 0.001
         beta1 = 0.6


         """
         DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
         """
         epochs = 1

         celeba_dataset = helper.Dataset('celeba', glob(os.path.join(data_dir, 'img_align_celeb
         with tf.Graph().as_default():
             train(epochs, batch_size, z_dim, learning_rate, beta1, celeba_dataset.get_batches
                   celeba_dataset.shape, celeba_dataset.image_mode)
```

```
Epoch 1/1 batch 100... Discriminator Loss: 1.2866... Generator Loss: 0.7389
Epoch 1/1 batch 200... Discriminator Loss: 1.4933... Generator Loss: 2.0067
Epoch 1/1 batch 300... Discriminator Loss: 1.1822... Generator Loss: 1.3739
Epoch 1/1 batch 400... Discriminator Loss: 1.0543... Generator Loss: 0.9299
Epoch 1/1 batch 500... Discriminator Loss: 1.0040... Generator Loss: 1.0078
Epoch 1/1 batch 600... Discriminator Loss: 1.1295... Generator Loss: 1.1926
Epoch 1/1 batch 700... Discriminator Loss: 1.1912... Generator Loss: 0.8660
Epoch 1/1 batch 800... Discriminator Loss: 1.1241... Generator Loss: 0.9590
Epoch 1/1 batch 900... Discriminator Loss: 1.1821... Generator Loss: 0.8174
Epoch 1/1 batch 1000... Discriminator Loss: 1.2113... Generator Loss: 0.8655
Epoch 1/1 batch 1100... Discriminator Loss: 1.2301... Generator Loss: 0.9278
Epoch 1/1 batch 1200... Discriminator Loss: 1.1780... Generator Loss: 1.0185
Epoch 1/1 batch 1300... Discriminator Loss: 1.1357... Generator Loss: 0.9080
Epoch 1/1 batch 1400... Discriminator Loss: 1.1534... Generator Loss: 0.9177
Epoch 1/1 batch 1500... Discriminator Loss: 1.2248... Generator Loss: 0.7144
Epoch 1/1 batch 1600... Discriminator Loss: 1.2484... Generator Loss: 0.6340
Epoch 1/1 batch 1700... Discriminator Loss: 1.2000... Generator Loss: 1.1183
Epoch 1/1 batch 1800... Discriminator Loss: 1.2814... Generator Loss: 0.6454
```

```
Epoch 1/1 batch 1900... Discriminator Loss: 1.2000... Generator Loss: 0.7506
Epoch 1/1 batch 2000... Discriminator Loss: 1.2678... Generator Loss: 0.6986
Epoch 1/1 batch 2100... Discriminator Loss: 1.2978... Generator Loss: 0.6892
Epoch 1/1 batch 2200... Discriminator Loss: 1.2396... Generator Loss: 0.9381
Epoch 1/1 batch 2300... Discriminator Loss: 1.2827... Generator Loss: 1.1751
Epoch 1/1 batch 2400... Discriminator Loss: 1.2388... Generator Loss: 0.6809
Epoch 1/1 batch 2500... Discriminator Loss: 1.3669... Generator Loss: 0.5396
Epoch 1/1 batch 2600... Discriminator Loss: 1.3611... Generator Loss: 0.7077
Epoch 1/1 batch 2700... Discriminator Loss: 1.3802... Generator Loss: 0.5204
Epoch 1/1 batch 2800... Discriminator Loss: 1.2944... Generator Loss: 1.0672
Epoch 1/1 batch 2900... Discriminator Loss: 1.3774... Generator Loss: 0.5583
Epoch 1/1 batch 3000... Discriminator Loss: 1.5517... Generator Loss: 0.4169
Epoch 1/1 batch 3100... Discriminator Loss: 1.3932... Generator Loss: 0.4996
```

1.3.5 Submitting This Project

When submitting this project, make sure to run all the cells before saving the notebook. Save the notebook file as "dlnd_face_generation.ipynb" and save it as a HTML file under "File" -> "Download as". Include the "helper.py" and "problem_unittests.py" files in your submission.