```
import os
import time
import tensorflow.compat.v1 as tf
import numpy as np
from glob import glob
import datetime
import random
from PIL import Image
import matplotlib.pyplot as plt
%matplotlib inline

from google.colab import drive
drive.mount('/content/gdrive')
def generator(z, output_channel_dim, training):
    with tf.variable_scope("generator", reuse= not training):

        # 8x8x1024
        fully_connected_layers = tf.layers.dense(z, 8*8*1024)
        fully_connected_layers = tf.reshape(fully_connected_layers, (-1, 8, 8,
1024))
        fully_connected_layers = tf.nn.leaky_relu(fully_connected_layers)

        # 8x8x1024 -> 16x16x512
        transverse_convolutional1 =
tf.layers.conv2d_transpose(inputs=fully_connected_layers,
                                                 filters=512,
                                                 kernel_size=[5,5],
                                                 strides=[2,2],
                                                 padding="SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),

name="transverse_convolutional1")
        batch_transverse_convolutional1 = tf.layers.batch_normalization(inputs =
transverse_convolutional1,
                                                          training=training,
                                                          epsilon=EPSILON,

name="batch_transverse_convolutional1")
        transverse_convolutional1_out =
tf.nn.leaky_relu(batch_transverse_convolutional1,
                                           name="transverse_convolutional1_out")

        # 16x16x512 -> 32x32x256
        transverse_convolutional2 =
tf.layers.conv2d_transpose(inputs=transverse_convolutional1_out,
                                                 filters=256,
                                                 kernel_size=[5,5],
                                                 strides=[2,2],
                                                 padding="SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),

name="transverse_convolutional2")
        batch_transverse_convolutional2 = tf.layers.batch_normalization(inputs =
transverse_convolutional2,
                                                          training=training,
                                                          epsilon=EPSILON,

name="batch_transverse_convolutional2")
        transverse_convolutional2_out =
tf.nn.leaky_relu(batch_transverse_convolutional2,
                                           name="transverse_convolutional2_out")
        # 32x32x256 -> 64x64x128
        transverse_convolutional3 =
tf.layers.conv2d_transpose(inputs=transverse_convolutional2_out,
                                                 filters=128,
                                                 kernel_size=[5,5],
                                                 strides=[2,2],
                                                 padding="SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),

name="transverse_convolutional3")
        batch_transverse_convolutional3 = tf.layers.batch_normalization(inputs =
transverse_convolutional3,
                                                          training=training,
                                                          epsilon=EPSILON,

name="batch_transverse_convolutional3")
        transverse_convolutional3_out =
tf.nn.leaky_relu(batch_transverse_convolutional3,
                                           name="transverse_convolutional3_out")

        # 64x64x128 -> 128x128x64
        transverse_convolutional4 =
tf.layers.conv2d_transpose(inputs=transverse_convolutional3_out,
                                                 filters=64,
                                                 kernel_size=[5,5],
                                                 strides=[2,2],
                                                 padding="SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),

name="transverse_convolutional4")
        batch_transverse_convolutional4 = tf.layers.batch_normalization(inputs =
transverse_convolutional4,
                                                          training=training,
                                                          epsilon=EPSILON,

name="batch_transverse_convolutional4")
        transverse_convolutional4_out =
tf.nn.leaky_relu(batch_transverse_convolutional4,
                                           name="transverse_convolutional4_out")

        # 128x128x64 -> 128x128x3
        logits =
tf.layers.conv2d_transpose(inputs=transverse_convolutional4_out,
                                            filters=3,
                                            kernel_size=[5,5],
                                            strides=[1,1],
                                            padding="SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                            name="logits")
        out = tf.tanh(logits, name="out")
        return out
def discriminator(x, reuse):
    with tf.variable_scope("discriminator", reuse=reuse):

        # 128*128*3 -> 64x64x64
        convolutional_layer1 = tf.layers.conv2d(inputs=x,
                                 filters=64,
                                 kernel_size=[5,5],
                                 strides=[2,2],
                                 padding="SAME",
kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                 name='convolutional_layer1')
        batch_norm1 = tf.layers.batch_normalization(convolutional_layer1,
                                                    training=True,
                                                    epsilon=EPSILON,
                                                    name='batch_norm1')
        convolutional_layer1_out = tf.nn.leaky_relu(batch_norm1,
                                     name="convolutional_layer1_out")

        # 64x64x64-> 32x32x128
        convolutional_layer2 = tf.layers.conv2d(inputs=convolutional_layer1_out,
                                 filters=128,
                                 kernel_size=[5, 5],
                                 strides=[2, 2],
                                 padding="SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                 name='convolutional_layer2')
        batch_norm2 = tf.layers.batch_normalization(convolutional_layer2,
                                                    training=True,
                                                    epsilon=EPSILON,
                                                    name='batch_norm2')
        convolutional_layer2_out = tf.nn.leaky_relu(batch_norm2,
                                     name="convolutional_layer2_out")

        # 32x32x128 -> 16x16x256
        convolutional_layer3 = tf.layers.conv2d(inputs=conv2_out,
                                 filters=256,
                                 kernel_size=[5, 5],
                                 strides=[2, 2],
                                 padding="SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                 name='convolutional_layer3')
        batch_norm3 = tf.layers.batch_normalization(convolutional_layer3,
                                                    training=True,
                                                    epsilon=EPSILON,
                                                    name='batch_norm3')
        convolutional_layer3_out = tf.nn.leaky_relu(batch_norm3,
                                     name="convolutional_layer3_out")

        # 16x16x256 -> 16x16x512
        convolutional_layer4 = tf.layers.conv2d(inputs=conv3_out,
                                 filters=512,
                                 kernel_size=[5, 5],
                                 strides=[1, 1],
                                 padding="SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                 name='convolutional_layer4')
        batch_norm4 = tf.layers.batch_normalization(convolutional_layer4,
                                                    training=True,
                                                    epsilon=EPSILON,
                                                    name='batch_norm4')
        convolutional_layer4_out = tf.nn.leaky_relu(batch_norm4,
                                     name="convolutional_layer4_out")

        # 16x16x512 -> 8x8x1024
        convolutional_layer5 = tf.layers.conv2d(inputs=convolutional_layer4_out,
                                filters=1024,
                                kernel_size=[5, 5],
                                strides=[2, 2],
                                padding="SAME",
kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                name='convolutional_layer5')
        batch_norm5 = tf.layers.batch_normalization(convolutional_layer5,
                                                    training=True,
                                                    epsilon=EPSILON,
                                                    name='batch_norm5')
        convolutional_layer5_out = tf.nn.leaky_relu(batch_norm5,
                                     name="convolutional_layer5_out")

        flatten = tf.reshape(convolutional_layer5_out, (-1, 8*8*1024))
        logits = tf.layers.dense(inputs=flatten,
                                 units=1,
                                 activation=None)
        out = tf.sigmoid(logits)
        return out, logits
def model_loss_calulations(input_real, input_z, output_channel_dim):
    generator_model = generator(input_z, output_channel_dim, True)

    noisy_input_real = input_real + tf.random_normal(shape=tf.shape(input_real),
                                                     mean=0.0,
                                                     stddev=random.uniform(0.0,
0.1),
                                                     dtype=tf.float32)

    discriminator_model_real, d_logits_real = discriminator(noisy_input_real,
reuse=False)
    discriminator_model_fake, d_logits_fake = discriminator(generator_model,
reuse=True)

    discriminator_loss_real =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,

labels=tf.ones_like(discriminator_model_real)*random.uniform(0.9, 1.0)))
    discriminator_loss_fake =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,

labels=tf.zeros_like(discriminator_model_fake)))
    discriminator_loss = tf.reduce_mean(0.5 * (discriminator_loss_real +
discriminator_loss_fake))
    generator_loss =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,

labels=tf.ones_like(discriminator_model_fake)))
    return discriminator_loss, generator_loss
def model_optimizers(discriminator_loss, generator_loss):
    t_vars = tf.trainable_variables()
    generator_vars = [var for var in t_vars if var.name.startswith("generator")]
    discriminator_vars = [var for var in t_vars if
var.name.startswith("discriminator")]

    update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    gen_updates = [op for op in update_ops if op.name.startswith('generator')]

    with tf.control_dependencies(gen_updates):
        discriminator_train_optimizer =
tf.train.AdamOptimizer(learning_rate=LR_D,
beta1=BETA1).minimize(discriminator_loss, var_list=discriminator_vars)
        generator_train_optimizer = tf.train.AdamOptimizer(learning_rate=LR_G,
beta1=BETA1).minimize(generator_loss, var_list=generator_vars)
    return discriminator_train_optimizer, generator_train_optimizer
def model_inputs(real_dim, z_dim):
    inputs_real = tf.placeholder(tf.float32, (None, *real_dim),
name='inputs_real')
    inputs_z = tf.placeholder(tf.float32, (None, z_dim), name="input_z")
    learning_rate_generator = tf.placeholder(tf.float32, name="lr_g")
    learning_rate_discriminator = tf.placeholder(tf.float32, name="lr_d")
    return inputs_real, inputs_z, learning_rate_generator,
learning_rate_discriminator
def show_samples(sample_images, name, epoch):
    figure, axes = plt.subplots(1, len(sample_images), figsize = (IMAGE_SIZE,
IMAGE_SIZE))
    for index, axis in enumerate(axes):
        axis.axis('off')
        image_array = sample_images[index]
        axis.imshow(image_array)
        image = Image.fromarray(image_array)
        image.save(name+"_"+str(epoch)+"_"+str(index)+".png")
    plt.savefig(name+"_"+str(epoch)+".png", bbox_inches='tight', pad_inches=0)
    plt.show()
    plt.close()
def test(sess, input_z, out_channel_dim, epoch):
    example_z = np.random.uniform(-1, 1, size=[SAMPLES_TO_SHOW,
input_z.get_shape().as_list()[-1]])
    samples = sess.run(generator(input_z, out_channel_dim, False),
feed_dict={input_z: example_z})
    sample_images = [((sample + 1.0) * 127.5).astype(np.uint8) for sample in
samples]
    show_samples(sample_images, OUTPUT_DIR + "samples", epoch)
def summarize_epoch(epoch, duration, sess, discriminator_losses,
generator_losses, input_z, data_shape):
    minibatch_size = int(data_shape[0]//BATCH_SIZE)
    print("Epoch {}/{}".format(epoch, EPOCHS),
          "\nDuration: {:.5f}".format(duration),
          "\nD Loss: {:.5f}".format(np.mean(discriminator_losses[-
minibatch_size:])),
          "\nG Loss: {:.5f}".format(np.mean(generator_losses[-
minibatch_size:])))
    fig, ax = plt.subplots()
    plt.plot(discriminator_losses, label='Discriminator', alpha=0.6)
    plt.plot(generator_losses, label='Generator', alpha=0.6)
    plt.title("Losses")
    plt.legend()
    plt.savefig(OUTPUT_DIR + "losses_" + str(epoch) + ".png")
    plt.show()
    plt.close()
    test(sess, input_z, data_shape[3], epoch)
def get_batches(data):
    batches = []
    for i in range(int(data.shape[0]//BATCH_SIZE)):
        batch = data[i * BATCH_SIZE:(i + 1) * BATCH_SIZE]
        augmented_images = []
        for img in batch:
            image = Image.fromarray(img)
            if random.choice([True, False]):
                image = image.transpose(Image.FLIP_LEFT_RIGHT)
            augmented_images.append(np.asarray(image))
        batch = np.asarray(augmented_images)
        normalized_batch = (batch / 127.5) - 1.0
        batches.append(normalized_batch)
    return batches
def train(get_batches, data_shape, checkpoint_to_load=None):
    input_images, input_z, lr_G, lr_D = model_inputs(data_shape[1:], NOISE_SIZE)
    discriminator_loss, generator_loss = model_loss_calculations(input_images,
input_z, data_shape[3])
    discriminator_optimizer, generator_optimizer =
model_optimizers(discriminator_loss, generator_loss)

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        epoch = 0
        iteration = 0
        discriminator_losses = []
        generator_losses = []

        for epoch in range(EPOCHS):
            epoch += 1
            start_time = time.time()

            for batch_images in get_batches:
                iteration += 1
                batch_z = np.random.uniform(-1, 1, size=(BATCH_SIZE,
NOISE_SIZE))
                _ = sess.run(discriminator_optimizer, feed_dict={input_images:
batch_images, input_z: batch_z, lr_D: LR_D})
                _ = sess.run(generator_optimizer, feed_dict={input_images:
batch_images, input_z: batch_z, lr_G: LR_G})
                discriminator_losses.append(discriminator_loss.eval({input_z:
batch_z, input_images: batch_images}))
                generator_losses.append(generator_loss.eval({input_z: batch_z}))

            summarize_epoch(epoch, time.time()-start_time, sess,
discriminator_losses, generator_losses, input_z, data_shape)
# Paths
INPUT_DATA_DIR = "/content/gdrive/My Drive/cropped_mini/"
OUTPUT_DIR = '/content/gdrive/My Drive/hiper/{date:%Y-%m-%d_%H:%M:
%S}/'.format(date=datetime.datetime.now())
#!ls "/content/gdrive/My Drive/cropped_mini/"


# INPUT_DATA_DIR = "/content/gdrive/My Drive/celebs_dataset/celeb_dataset/"
# OUTPUT_DIR = '/content/gdrive/My Drive/celebs_dataset/{date:%Y-%m-%d_%H:%M:
%S}/'.format(date=datetime.datetime.now())

#INPUT_DATA_DIR = "/content/gdrive/My Drive/Fish_Dataset/Black Sea Sprat/"
#OUTPUT_DIR = '/content/gdrive/My Drive/Fish_Dataset/{date:%Y-%m-%d_%H:%M:
%S}/'.format(date=datetime.datetime.now())


if not os.path.exists(OUTPUT_DIR):
    os.makedirs(OUTPUT_DIR)
# Hyperparameters
IMAGE_SIZE = 128
NOISE_SIZE = 100
LR_D = 0.0003
LR_G = 0.003
BATCH_SIZE = 100
EPOCHS = 5
BETA1 = 0.4
WEIGHT_INIT_STDDEV = 0.05
EPSILON = 0.00008
SAMPLES_TO_SHOW = 5

# Training
input_images = np.asarray([np.asarray(Image.open(file).resize((IMAGE_SIZE,
IMAGE_SIZE)).convert('RGB')) for file in glob(INPUT_DATA_DIR + '*')])
print ("Input: " + str(input_images.shape))

np.random.shuffle(input_images)

sample_images = random.sample(list(input_images), SAMPLES_TO_SHOW)
show_samples(sample_images, OUTPUT_DIR + "inputs", 0)
```

```
with tf.Graph().as_default():
    train(get_batches(input_images), input_images.shape)
```