SOURCE CODE:

Created by:

A.KASTHOORI

RegNo:912321104013

Naanmudhalvan ID:au912321104013

III - year, CSE

SACS MAVMM ENGINEERING COLLEGE,

MADURAI.

```
from flask import Flask, render_template, request, jsonify, send_file
import numpy as np
import scipy.misc
import base64
from io import BytesIO
from test import *
import time

app = Flask(__name__)

@app.route('/')
def index():
    return render_template("index.html")


@app.route('/denoisify', methods=['GET', 'POST'])
def denoisify():
    if request.method == "POST":
        inputImg = request.files['file']
        outputImg = denoise(inputImg)
        scipy.misc.imsave('static/output.png', outputImg)
        return jsonify(result="Success")


if __name__=="__main__"
```

```
app.run(host="0.0.0.0",port="80")
```

```
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models

# Load your dataset of noisy images
# Assuming you have a function to load your dataset, e.g., load_dataset()
def load_dataset():
    # Load and preprocess your dataset here
    pass

# Define the generator model
def build_generator(input_shape):
    model = models.Sequential([
        layers.Dense(8 * 8 * 128, input_shape=input_shape),
        layers.Reshape((8, 8, 128)),
        layers.Conv2DTranspose(64, kernel_size=3, strides=2, padding='same'),
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding='same', activation='tanh')
    ])
    return model

# Define the discriminator model
def build_discriminator(input_shape):
    model = models.Sequential([
        layers.Conv2D(64, kernel_size=3, strides=2, padding='same', input_shape=input_shape),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.4),
        layers.Conv2D(128, kernel_size=3, strides=2, padding='same'),
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.4),
        layers.Flatten(),
        layers.Dense(1, activation='sigmoid')
    ])
    return model

# Define the GAN model
def build_gan(generator, discriminator):
    discriminator.trainable = False
    model = models.Sequential([
        generator,
        discriminator
    ])
    return model

# Define the training loop
def train_gan(generator, discriminator, gan, dataset, epochs=100, batch_size=64):
    for epoch in range(epochs):
        for i in range(0, len(dataset), batch_size):
            # Train discriminator
            real_images = dataset[i:i+batch_size]
            noise = np.random.normal(0, 1, (len(real_images), 100))
            generated_images = generator.predict(noise)
            labels_real = np.ones((len(real_images), 1))
            labels_fake = np.zeros((len(real_images), 1))
            discriminator_loss_real = discriminator.train_on_batch(real_images, labels_real)
            discriminator_loss_fake = discriminator.train_on_batch(generated_images, labels_fake)
            discriminator_loss = 0.5 * np.add(discriminator_loss_real, discriminator_loss_fake)

            # Train generator
            noise = np.random.normal(0, 1, (batch_size, 100))
            labels_gen = np.ones((batch_size, 1))
            generator_loss = gan.train_on_batch(noise, labels_gen)

            print(f"Epoch {epoch + 1}/{epochs}, Batch {i + 1}/{len(dataset)}, Discriminator Loss: {discriminator_loss}, Generator Loss: {generator_loss}")

# Load and preprocess dataset
dataset = load_dataset()

# Define input shape
input_shape = (64, 64, 1)  # Assuming grayscale images of size 64x64

# Build and compile models
```

```
generator = build_generator(input_shape=(100,))

discriminator = build_discriminator(input_shape=input_shape)

discriminator.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']
import numpy as np
import tensorflow as tf
import tensorflow.contrib.slim as slim


from utils import *
from conv_helper import *


def generator(input):
    conv1, conv1_weights = conv_layer(input, 9, 3, 32, 1, "g_conv1")
    conv2, conv2_weights = conv_layer(conv1, 3, 32, 64, 1, "g_conv2")
    conv3, conv3_weights = conv_layer(conv2, 3, 64, 128, 1, "g_conv3")

    res1, res1_weights = residual_layer(conv3, 3, 128, 128, 1, "g_res1")
    res2, res2_weights = residual_layer(res1, 3, 128, 128, 1, "g_res2")
    res3, res3_weights = residual_layer(res2, 3, 128, 128, 1, "g_res3")

    deconv1 = deconvolution_layer(res3, [BATCH_SIZE, 128, 128, 64], 'g_deconv1')
    deconv2 = deconvolution_layer(deconv1, [BATCH_SIZE, 256, 256, 32], "g_deconv2")

    deconv2 = deconv2 + conv1

    conv4, conv4_weights = conv_layer(deconv2, 9, 32, 3, 1, "g_conv5", activation_function=tf.nn.tanh)
    conv4 = conv4 + input
    output = output_between_zero_and_one(conv4)

    return output


def discriminator(input, reuse=False):
    conv1, conv1_weights = conv_layer(input, 4, 3, 48, 2, "d_conv1", reuse=reuse)
    conv2, conv2_weights = conv_layer(conv1, 4, 48, 96, 2, "d_conv2", reuse=reuse)
    conv3, conv3_weights = conv_layer(conv2, 4, 96, 192, 2, "d_conv3", reuse=reuse)
    conv4, conv4_weights = conv_layer(conv3, 4, 192, 384, 1, "d_conv4", reuse=reuse)
    conv5, conv5_weights = conv_layer(conv4, 4, 384, 1, 1, "d_conv5", activation_function=tf.nn.sigmoid, reuse=reuse)

    return conv5
```

```
)
generator.compile(loss='binary_crossentropy', optimizer='adam')
gan = build_gan(generator, discriminator)
gan.compile(loss='binary_crossentropy', optimizer='adam')

# Train the GAN
train_gan(gen
import time


import tensorflow as tf
import numpy as np


from utils import *
from model import *


from skimage import measure




def test(image):
    tf.reset_default_graph()


    global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name='global_step')


    gen_in = tf.placeholder(shape=[None, BATCH_SHAPE[1], BATCH_SHAPE[2], BATCH_SHAPE[3]], dtype=tf.float32, name='generated_image')
    real_in = tf.placeholder(shape=[None, BATCH_SHAPE[1], BATCH_SHAPE[2], BATCH_SHAPE[3]], dtype=tf.float32, name='groundtruth_image')


    Gz = generator(gen_in)
    init = tf.global_variables_initializer()
    with tf.Session() as sess:
        sess.run(init)

        saver = initialize(sess)
        initial_step = global_step.eval()

        start_time = time.time()
        n_batches = 200
        total_iteration = n_batches * N_EPOCHS

        image = sess.run(tf.map_fn(lambda img: tf.image.per_image_standardization(img), image))
        image = sess.run(Gz, feed_dict={gen_in: image})
        image = np.resize(image[0][56:, :, :], [144, 256, 3])
        imsave('output', image)
        return image

def denoise(image):
    image = scipy.misc.imread(image, mode='RGB').astype('float32')
    npad = ((56, 56), (0, 0), (0, 0))
    image = np.pad(image, pad_width=npad, mode='constant', constant_values=0)
    image = np.expand_dims(image, axis=0)
    print(image[0].shape)
    output = test(image)
    return output
```

```
if __name__=='__main__':
    image = scipy.misc.imread(sys.argv[-1], mode='RGB').astype('float32')
    npad = ((56, 56), (0, 0), (0, 0))
    image = np.pad(image, pad_width=npad, mode='constant', constant_values=0)
    image = np.expand_dims(image, axis=0)
    print(image[0].shape)
    test(image)
```

```
import time


import tensorflow as tf
import numpy as np


from utils import *
from model import *


from skimage import measure




def train():
    tf.reset_default_graph()


    global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name='global_step')


    gen_in = tf.placeholder(shape=[None, BATCH_SHAPE[1], BATCH_SHAPE[2], BATCH_SHAPE[3]], dtype=tf.float32, name='generated_image')
    real_in = tf.placeholder(shape=[None, BATCH_SHAPE[1], BATCH_SHAPE[2], BATCH_SHAPE[3]], dtype=tf.float32, name='groundtruth_image')


    Gz = generator(gen_in)
    Dx = discriminator(real_in)
    Dg = discriminator(Gz, reuse=True)
    real_in_bgr = tf.map_fn(lambda img: RGB_TO_BGR(img), real_in)
    Gz_bgr = tf.map_fn(lambda img: RGB_TO_BGR(img), Gz)

    psnr=0
    ssim=0

    d_loss = -tf.reduce_mean(tf.log(Dx) + tf.log(1.-Dg))
    g_loss = ADVERSARIAL_LOSS_FACTOR * -tf.reduce_mean(tf.log(Dg)) + PIXEL_LOSS_FACTOR * get_pixel_loss(real_in, Gz) \
            + STYLE_LOSS_FACTOR * get_style_loss(real_in_bgr, Gz_bgr) + SMOOTH_LOSS_FACTOR * get_smooth_loss(Gz)

    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if 'd_' in var.name]
    g_vars = [var for var in t_vars if 'g_' in var.name]

    d_solver = tf.train.AdamOptimizer(LEARNING_RATE).minimize(d_loss, var_list=d_vars, global_step=global_step)
    g_solver = tf.train.AdamOptimizer(LEARNING_RATE).minimize(g_loss, var_list=g_vars)


    init = tf.global_variables_initializer()
    with tf.Session() as sess:
        sess.run(init)

        saver = initialize(sess)
        initial_step = global_step.eval()
        start_time = time.time()

        n_batches = 200

        total_iteration = n_batches * N_EPOCHS


        validation_batch = sess.run(tf.map_fn(lambda img: tf.image.per_image_standardization(img), validation))



        for index in range(initial_step, total_iteration):

            input_batch = load_next_training_batch()

            training_batch, groundtruth_batch = np.split(input_batch, 2, axis=2)


            training_batch = sess.run(tf.map_fn(lambda img: tf.image.per_image_standardization(img), training_batch))

            groundtruth_batch = sess.run(tf.map_fn(lambda img: tf.image.per_image_standardization(img), groundtruth_batch))



            _, d_loss_cur = sess.run([d_solver, d_loss], feed_dict={gen_in: training_batch, real_in: groundtruth_batch})

            _, g_loss_cur = sess.run([g_solver, g_loss], feed_dict={gen_in: training_batch, real_in: groundtruth_batch})
```

```
            if(index + 1) % SKIP_STEP == 0:

                saver.save(sess, CKPT_DIR, index)
                image = sess.run(Gz, feed_dict={gen_in: validation_batch})
                image = np.resize(image[7][56:, :, :], [144, 256, 3])

                imsave('val_%d' % (index+1), image)
                image = scipy.misc.imread(IMG_DIR+'val_%d.png' % (index+1), mode='RGB').astype('float32')
                psnr = measure.compare_psnr(metrics_image, image, data_range=255)
                ssim = measure.compare_ssim(metrics_image, image, multichannel=True, data_range=255,
win_size=11)

                print(
                    "Step {}/{} Gen Loss: ".format(index + 1, total_iteration) + str(g_loss_cur) + " Disc Loss: " + str(
                        d_loss_cur)+ " PSNR: "+str(psnr)+" SSIM: "+str(ssim))


if __name__=='__main__':
    training_dir_list = training_dataset_init()
    validation = load_validation()
    train()
```

```
import os
import re
import sys
import glob
import scipy.misc
from itertools import cycle

import numpy as np
import tensorflow as tf


from libs import vgg16

from PIL import Image


LEARNING_RATE = 0.002
BATCH_SIZE = 5
BATCH_SHAPE = [BATCH_SIZE, 256, 256, 3]
SKIP_STEP = 10
N_EPOCHS = 500
CKPT_DIR = './Checkpoints/'
IMG_DIR = './Images/'
GRAPH_DIR = './Graphs/'
TRAINING_SET_DIR= './dataset/training/'
```

```
# GROUNDTRUTH_SET_DIR='./dataset/groundtruth/'
VALIDATION_SET_DIR='./dataset/validation/'
METRICS_SET_DIR='./dataset/metrics/'
TRAINING_DIR_LIST = []
ADVERSARIAL_LOSS_FACTOR = 0.5
PIXEL_LOSS_FACTOR = 1.0
STYLE_LOSS_FACTOR = 1.0
SMOOTH_LOSS_FACTOR = 1.0
metrics_image = scipy.misc.imread(METRICS_SET_DIR+'gt.png', mode='RGB').astype('float32')



def initialize(sess):
    saver = tf.train.Saver()
    writer = tf.summary.FileWriter(GRAPH_DIR, sess.graph)

    if not os.path.exists(CKPT_DIR):
        os.makedirs(CKPT_DIR)
    if not os.path.exists(IMG_DIR):
        os.makedirs(IMG_DIR)

    ckpt = tf.train.get_checkpoint_state(os.path.dirname(CKPT_DIR))
    if ckpt and ckpt.model_checkpoint_path:
        saver.restore(sess, ckpt.model_checkpoint_path)
    return saver
```

```
def get_training_dir_list():
    training_list = [d[1] for d in os.walk(TRAINING_SET_DIR)]
    global TRAINING_DIR_LIST
    TRAINING_DIR_LIST = training_list[0]
    return TRAINING_DIR_LIST


def load_next_training_batch():
    batch = next(pool)


    # filelist = sorted(glob.glob(TRAINING_SET_DIR+ batch +'/*.png'), key=alphanum_key)
    # batch = np.array([np.array(scipy.misc.imread(fname, mode='RGB').astype('float32')) for fname in filelist])
    # npad =((0, 0), (56, 56), (0, 0), (0, 0))
    # batch = np.pad(batch, pad_width=npad, mode='constant', constant_values=0)
    return batch


# def load_groundtruth():
#     filelist = sorted(glob.glob(GROUNDTRUTH_SET_DIR + '/*.png'), key=alphanum_key)
#     groundtruth = np.array([np.array(scipy.misc.imread(fname, mode='RGB').astype('float32')) for fname in filelist])
#     # npad = ((0, 0), (56, 56), (0, 0), (0, 0))
#     # groundtruth = np.pad(groundtruth, pad_width=npad, mode='constant', constant_values=0)
#     return groundtruth


def load_validation():
    filelist = sorted(glob.glob(VALIDATION_SET_DIR + '/*.png'), key=alphanum_key)
    validation = np.array([np.array(scipy.misc.imread(fname, mode='RGB').astype('float32')) for fname in filelist])
    npad = ((0, 0), (56, 56), (0, 0), (0, 0))
    validation = np.pad(validation, pad_width=npad, mode='constant', constant_values=0)
    return validation


def training_dataset_init():
    filelist = sorted(glob.glob(TRAINING_SET_DIR + '/*.png'), key=alphanum_key)
    batch = np.array([np.array(scipy.misc.imread(fname, mode='RGB').astype('float32')) for fname in filelist])
    batch = split(batch, BATCH_SIZE)
    training_dir_list = get_training_dir_list()
    global pool
    pool = cycle(batch)
    # return training_dir_list



def imsave(filename, image):
    scipy.misc.imsave(IMG_DIR+filename+'.png', image)


def merge_images(file1, file2):
    """Merge two images into one, displayed side by side
    :param file1: path to first image file
    :param file2: path to second image file
    :return: the merged Image object
    """
    image1 = Image.fromarray(np.uint8(file1))
    image2 = Image.fromarray(np.uint8(file2))

    (width1, height1) = image1.size
    (width2, height2) = image2.size

    result_width = width1 + width2
    result_height = max(height1, height2)

    result = Image.new('RGB', (result_width, result_height))
    result.paste(im=image1, box=(0, 0))
    result.paste(im=image2, box=(width1, 0))
    return result


def tryint(s):
    try:
        return int(s)
    except:
        return s

def alphanum_key(s):
    """ Turn a string into a list of string and number chunks.
        "z23a" -> ["z", 23, "a"]
    """
    return [ tryint(c) for c in re.split('([0-9]+)', s) ]


def split(arr, size):
    arrs = []
    while len(arr) > size:
        pice = arr[:size]
        arrs.append(pice)
        arr = arr[size:]
    arrs.append(arr)
    return arrs


def lrelu(x, leak=0.2, name='lrelu'):
    with tf.variable_scope(name):
        f1 = 0.5 * (1 + leak)
        f2 = 0.5 * (1 - leak)
        return f1 * x + f2 * abs(x)

def RGB_TO_BGR(img):
    img_channel_swap = img[..., ::-1]
    # img_channel_swap_1 = tf.reverse(img, axis=[-1])
    return img_channel_swap
```

```
def get_pixel_loss(target,prediction):

    pixel_difference = target - prediction

    pixel_loss = tf.nn.l2_loss(pixel_difference)

    return pixel_loss


def get_style_layer_vgg16(image):

    net = vgg16.get_vgg_model()

    style_layer = 'conv2_2/conv2_2:0'

    feature_transformed_image = tf.import_graph_def(

        net['graph_def'],

        name='vgg',

        input_map={'images:0': image},return_elements=[style_layer])

    feature_transformed_image = (feature_transformed_image[0])

    return feature_transformed_image


def get_style_loss(target,prediction):

    feature_transformed_target = get_style_layer_vgg16(target)

    feature_transformed_prediction = get_style_layer_vgg16(prediction)

    feature_count = tf.shape(feature_transformed_target)[3]

    style_loss = tf.reduce_sum(tf.square(feature_transformed_target-feature_transformed_prediction))

    style_loss = style_loss/tf.cast(feature_count, tf.float32)

    return style_loss


def get_smooth_loss(image):

    batch_count = tf.shape(image)[0]
    image_height = tf.shape(image)[1]
    image_width = tf.shape(image)[2]

    horizontal_normal = tf.slice(image, [0, 0, 0,0], [batch_count, image_height, image_width-1,3])
    horizontal_one_right = tf.slice(image, [0, 0, 1,0], [batch_count, image_height, image_width-1,3])
    vertical_normal = tf.slice(image, [0, 0, 0,0], [batch_count, image_height-1, image_width,3])
    vertical_one_right = tf.slice(image, [0, 1, 0,0], [batch_count, image_height-1, image_width,3])
    smooth_loss = tf.nn.l2_loss(horizontal_normal-horizontal_one_right)+tf.nn.l2_loss(vertical_normal-vertical_one_right)
    return smooth_loss
```

```
erator, discriminator, gan, dataset)
```