import os

import time

import math

import pprint

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

from skimage.metrics import structural\_similarity as ssim

import configargparse

import data\_handling

import render

import models

def\_dtype=np.float32

COMPLETENESS\_THRESHOLD=1.0 #In meters. Completeness is the ratio of pixels in the estimated DEM where the error is lower than the threshold.

def config\_parser():

    parser = configargparse.ArgumentParser(description='Train and test a Shadow Neural Radiance Field on a set of posed, forward-facing images. Produces the model weights and a summary of the evaluation on the test set.')

    parser.add\_argument('--config', is\_config\_file=True, help='Config file path.')

    # Dataset arguments

    parser.add\_argument('--data.image.path', type=str, default='./data/', help='Path that contains all images.')

    parser.add\_argument('--data.image.df', type=int, default=1, help='Image downsample factor.')

    parser.add\_argument('--data.image.sd', type=np.float32, default=1.0, help='Image sampling distance.')

    parser.add\_argument('--data.depth.path', type=str, default='./data/depth.tif', help='Depth map path.')

    parser.add\_argument('--data.depth.df', type=int, default=1, help='Depth map downsample factor.')

    parser.add\_argument('--data.md.path', type=str, default='./data/md.txt', help='Metadata file path.')

    parser.add\_argument('--data.train\_id', type=str, nargs='+', help='ID of train images.')

    parser.add\_argument('--data.test\_id', type=str, nargs='+', help='ID of test images.')

    # Model arguments

    parser.add\_argument('--model.ins.light', type=bool, default=False, help='Use light directions as network inputs.')

    parser.add\_argument('--model.ins.views', type=bool, default=False, help='Use view directions as inputs.')

    parser.add\_argument('--model.outs.shad', type=bool, default=False, help='Directional light source visibility function as network output.')

    parser.add\_argument('--model.outs.sky', type=bool, default=False, help='Diffuse light color as network output.')

    parser.add\_argument('--model.act', type=str, default='relu', help='Neuron activation function [relu, sin].')

    parser.add\_argument('--model.act.sin.w0', type=np.float32, default=30.0, help='Initial wavelength for SIREN.')

    parser.add\_argument('--model.sigma.depth', type=int, default=8, help='Number of fully-connected layers for sigma function.')

    parser.add\_argument('--model.sigma.width', type=int, default=256, help='Width of layers for sigma function.')

    parser.add\_argument('--model.sigma.skips', type=int, nargs='+', default=[], help='Skip connections.')

    parser.add\_argument('--model.c.depth', type=int, default=1, help='Number of fully-connected layers for color function.')

    parser.add\_argument('--model.c.width', type=int, default=128, help='Width of layers for color function.')

    parser.add\_argument('--model.shad.depth', type=int, default=4, help='Number of fully-connected layers for shadow function.')

    parser.add\_argument('--model.shad.width', type=int, default=128, help='Width of layers for shadow function.')

    parser.add\_argument('--model.emb.pos', type=int, default=0, help='Length of on-axis positional encoding. 0 to disable.')

    parser.add\_argument('--model.emb.dir', type=int, default=0, help='Length of on-axis directional encoding. 0 to disable.')

    # Rendering arguments

    parser.add\_argument('--rend.nsamples', type=int, default=64, help='Number of samples for coarse rendering.')

    parser.add\_argument('--rend.nimportance', type=int, default=64, help='Number of samples for fine rendering, 0 to disable.')

    parser.add\_argument('--rend.mode', type=str, default='nf', help='Rendering mode : near-far or altitude sampling [nf, alt].')

    parser.add\_argument('--rend.mode.nf.near', type=np.float32, default=3.0, help='Near point (px).')

    parser.add\_argument('--rend.mode.nf.far', type=np.float32, default=10.0, help='Far point (px).')

    parser.add\_argument('--rend.mode.alt.max', type=np.float32, default=30.0, help='Max alt (px).')

    parser.add\_argument('--rend.mode.alt.min', type=np.float32, default=-30.0, help='Min alt(px).')

    parser.add\_argument('--rend.unzoom', type=bool, default=False, help='Special unzoom mode for off-nadir EO images.')

    parser.add\_argument('--rend.rescale', type=np.float32, default=None, help='Largest scene extent in pixel units. Calculated based on image sizes if not provided.')

    # Training arguments

    parser.add\_argument('--train.n\_epoch', type=int, default=200, help='Number of iterations for training.')

    parser.add\_argument('--train.n\_rand', type=int, default=1024, help='Number of random rays at each iteration.')

    parser.add\_argument('--train.lr.init', type=np.float32, default=1e-4, help='Initial learning rate.')

    parser.add\_argument('--train.lr.decay', type=np.float32, default=0.2, help='Learning rate decay over entire training.')

    parser.add\_argument('--train.noise.sigma', type=np.float32, default=10.0, help='Standard deviation of sigma pre-activation noise.')

    parser.add\_argument('--train.noise.shad', type=np.float32, default=1.0, help='Standard deviation of shadow function pre-activation noise.')

    parser.add\_argument('--train.shad', type=bool, default=False, help='Use solar correction rays.')

    parser.add\_argument('--train.shad.lambda', type=np.float32, default=0.1, help='Weight of solar correction loss.')

    parser.add\_argument('--train.shad.df', type=int, default=1, help='Downsample factor of solar correction rays compared to image rays, 1 to sample at same resolution.')

    parser.add\_argument('--train.shad.custom', type=str, default='none', help='Type of custon solar correction rays [linear, rectangle].')

    parser.add\_argument('--train.shad.custom.bounds.start', type=np.float32, nargs='+', default=[160.0, 40.0], help='Start point (az, el) in deg.')

    parser.add\_argument('--train.shad.custom.bounds.end', type=np.float32, nargs='+', default=[100.0, 80.0], help='End point (az, el) in deg.')

    parser.add\_argument('--train.shad.custom.bounds.samp', type=int, nargs='+', default=[10, 1], help='Sampling scheme for solar correction rays. If linear, 1st dimension is number of samples.')

    # Output arguments

    parser.add\_argument('--out.iplot', type=int, default=0, help='Frequency of test evaluation for output, 0 to disable.')

    parser.add\_argument('--out.path', type=str, default='./results/', help='Path to save outputs.')

    #Hardware options

    parser.add\_argument('--gpu', type=str, default='1', help='GPU to use.')

    return parser

def read\_config(path):

    """Read config file from a path and return the corresponding argument dictionary"""

    parser = config\_parser()

    args = parser.parse\_args(f'--config {path}')

    arg\_dict = vars(args)

    return arg\_dict

def init\_exp\_decay\_adam(init\_lr, N\_iters, decay):

    """

    Intialize Adam optimizer with learning rate (lr) following an exponential decay.

    The leaning rate of init\_lr\*decay is reached after N\_iters.

    Parameters:

    init\_lr (float): Initial learning rate.

    N\_iters (int): Number of iterations for decay.

    decay (float): Decay factor.

    Outputs:

    optimizer: Keras optimizer

    """

    lrate = tf.keras.optimizers.schedules.ExponentialDecay(init\_lr, N\_iters, decay\_rate=decay)

    optimizer = tf.keras.optimizers.Adam(lrate)

    return optimizer

def train\_model(model, optimizer, N\_iterations, arg\_dict, train\_rays, sc\_train\_rays=None, decrease\_noise=True, eval\_dataset=None):

    """

    Main training function, optimize a S-NeRF model to a set of pre-extracted training rays.

    Parameters:

    model (dict): Initial model with embeddings and dimensions (models.py).

    optimizer (keras.optimizer): optimizer with learning rate schedule (output of init\_exp\_decay\_adam).

    N\_iterations (int): Number of epochs to train the model. Each epoch corresponds to one batch of rays.

    arg\_dict (dict): Global config variables.

    train\_rays (dict): Pre-extracted training rays from generate\_train\_rays (render.py).

    sc\_train\_rays (dict): Pre-extracted solar correction rays (render.py)

    decrease\_noise (bool): Activate to linearly decrease the strength of the regularization noise.

    eval\_dataset (dict): Dataset of test images to evalue the test scores during training.

    Outputs:

    model (dict): Trained model.

    loss\_log (list(string)): Log of the different components of the loss function during training.

    scores (list(...)): Log of test scores achieved during training.

    """

    N\_rand = arg\_dict['train.n\_rand']

    N\_train\_rays = train\_rays['rays\_o'].shape[0]

    i\_batch = 0

    raw\_noise\_std\_init = tf.convert\_to\_tensor((arg\_dict["train.noise.sigma"], arg\_dict["train.noise.shad"]), dtype=def\_dtype)

    raw\_noise\_std = raw\_noise\_std\_init

    loss\_log = [] # For logging the training losses

    scores = [] # For logging the train + test scores

    # Shuffle train rays

    train\_rays = render.shuffle\_rays(train\_rays)

    if sc\_train\_rays is not None:

        i\_batch\_sc = 0

        N\_sc\_train\_rays = sc\_train\_rays['rays\_o'].shape[0]

        sc\_train\_rays = render.shuffle\_rays(sc\_train\_rays)

        lambda\_sc = arg\_dict["train.shad.lambda"]

        # No importance sampling for shadow correction : all samples in first pass

        sc\_arg\_dict = arg\_dict.copy()

        sc\_arg\_dict['rend.nsamples'] = arg\_dict['rend.nsamples'] + arg\_dict['rend.nimportance']

        sc\_arg\_dict['rend.nimportance'] = 0

    print("Begin training")

    # Extract gradient variables from the network

    grad\_vars = model['model'].trainable\_variables

    for i in range(N\_iterations):

        if i\_batch >= N\_train\_rays:

            # Once all rays have been sampled shuffle and reset batch index

            train\_rays = render.shuffle\_rays(train\_rays)

            i\_batch = 0

        # Extract N\_rand rays from the batch

        train\_ray\_batch = render.get\_ray\_batch(train\_rays, i\_batch, i\_batch+N\_rand)

        i\_batch+=N\_rand

        if sc\_train\_rays is not None:

            if i\_batch\_sc > N\_sc\_train\_rays:

                i\_batch\_sc = 0

                sc\_train\_rays = render.shuffle\_rays(sc\_train\_rays)

            sc\_train\_ray\_batch = render.get\_ray\_batch(sc\_train\_rays, i\_batch\_sc, i\_batch\_sc+N\_rand)

            i\_batch\_sc+=N\_rand

        if decrease\_noise:

            raw\_noise\_std = raw\_noise\_std\_init\*(1-i/N\_iterations)

        # Render train ray batch and sc train batch

        with tf.GradientTape() as tape:

            ret\_dict\_c = render.render\_rays(model, arg\_dict, train\_ray\_batch, rand=True, raw\_noise\_std=raw\_noise\_std, rets=['rgb'])

            # Compute rgb loss

            rgb\_loss = tf.reduce\_mean(tf.square(ret\_dict\_c['rgb'] - train\_ray\_batch['values']))

            loss = rgb\_loss

            if sc\_train\_rays is not None:

                # Render shadow correction rays without perturbation on opacity

                ret\_dict\_sc = render.render\_rays(model, sc\_arg\_dict, sc\_train\_ray\_batch, rand=True,

                                                       raw\_noise\_std=(0.0, raw\_noise\_std[1]), rets=['ret\_sun', 'ret\_shadow\_loss'])

                # Compute shadow loss

                s\_loss = (tf.reduce\_mean(ret\_dict\_sc['ret\_shadow\_loss']) + tf.reduce\_mean(1.0-ret\_dict\_sc['ret\_sun']))\*lambda\_sc

                loss += s\_loss

        # Propagate gradients

        gradients = tape.gradient(loss, grad\_vars)

        optimizer.apply\_gradients(zip(gradients, grad\_vars))

        # Log loss values

        loss\_log.append(f"{i} {rgb\_loss} {s\_loss if sc\_train\_rays is not None else ''}\n")

        if (i < 10) or (i % 25 == 0):

            rgb\_psnr = -10. \* tf.math.log(rgb\_loss) / tf.math.log(10.)

            print(f"{i} {rgb\_psnr} {s\_loss if sc\_train\_rays is not None else ''}")

        if (eval\_dataset is not None) and (arg\_dict['out.iplot'] > 0) and (i % arg\_dict['out.iplot'] == 0):

            dataset\_rend = render.render\_dataset(eval\_dataset, model, ['rgb'], arg\_dict)

            train, test, alt = test\_model(model, eval\_dataset, dataset\_rend, arg\_dict)

            scores.append((i, (train, test, alt)))

            print(f"Test {i}")

            print(f"{test} {alt}")

    return model, loss\_log, scores

def compute\_image\_scores(ref\_img, rend\_img):

    """

    Evaluate the two image quality metrics between a rendered image and a reference image.

    1. Peak Signal to Noise Ratio (PSNR)

    2. Structural SIMilarity (SSIM) fromm scikit-image.

    Parameters:

    ref\_img (array[H, W, 3]): Reference image.

    rend\_img  (array[H, W, 3]): Rendered image.

    Outputs:

    psnr (float): PSNR value

    ssim (float): SSIM value

    """

    mse = tf.reduce\_mean(tf.square(rend\_img - ref\_img))

    psnr = -10. \* tf.math.log(mse) / tf.math.log(10.)

    struct\_sim = ssim(rend\_img.numpy(), ref\_img.numpy(), data\_range=1, multichannel=True)

    return [psnr, struct\_sim]

def render\_dsm(model, arg\_dict, dsm):

    """

    Render the depth map from nadir angle and subtract from orbital radius to extract Digital Surface Model.

    Parameters:

    model (dict): Model with embeddings and dimensions (models.py).

    arg\_dict (dict): Global configuration variables.

    dsm (Tensor[N, M]): Ground truth DSM

    Outputs:

    dsm\_predict (Tensor[N, M]): Predicted DSM

    """

    SR = 0.5 \* arg\_dict['data.depth.df']

    radius = 617000.0/SR

    arg\_dict\_temp = arg\_dict.copy()

    arg\_dict\_temp['data.image.sd'] = SR

    arg\_dict\_temp['data.image.df'] = 1

    az, el = np.pi, np.pi/2

    pose = data\_handling.pose\_spherical(az, -el, radius)

    hwf = dsm.shape[0], dsm.shape[1], radius

    light\_dir=tf.reshape(tf.convert\_to\_tensor([np.deg2rad(100), np.deg2rad(80)], dtype=def\_dtype), [1,2])

    view\_dir=tf.reshape(tf.convert\_to\_tensor([az, el], dtype=def\_dtype), [1,2])

    ret\_dict = render.render\_image(model, arg\_dict\_temp, hwf, pose, 1.0, light\_dir, view\_dir, rets=['depth'])

    dsm\_predict = (radius - ret\_dict['depth'])\*SR

    return dsm\_predict

def test\_model(model, dataset, dataset\_rend, arg\_dict):

    """

    Evaluate the image scores of a set of rendered images (PSNR, SSIM), and compute the altitude scores.

    1. Global Mean Average Error

    2. Completeness : ratio of pixels with error < 1m (between 0 and 1).

    3. Accuracy : Mean Average Error of those pixels with an error < 1m.

    Parameters:

    model (dict): Model with embeddings and dimensions (models.py).

    dataset (dict): Training and test images (generate\_dataset from data\_handling.py).

    dataset\_rend (dict): Rendered training and test images (render\_dataset from render.py)

    arg\_dict (dict): Global configuration variables.

    Outputs:

    train\_scores: Train scores, see compute\_image\_scores

    test\_scores: Test scores, see compute\_image\_scores

    alt\_scores: Altitude scores (MAE, Completeness, Accuracy).

    """

    train\_scores, test\_scores, alt\_scores=[], [], []

    for i in range(len(dataset['train\_imgs'])):

        train\_scores.append(compute\_image\_scores(dataset['train\_imgs'][i], dataset\_rend['train\_rend'][i]['rgb']))

    for i in range(len(dataset['test\_imgs'])):

        test\_scores.append(compute\_image\_scores(dataset['test\_imgs'][i], dataset\_rend['test\_rend'][i]['rgb']))

    # Evaluate loss from one given depth map

    dsm = render\_dsm(model, arg\_dict, dataset['depth\_map'])

    alt\_abs\_diff = tf.abs(dataset['depth\_map'] - dsm)

    alt\_mae = tf.reduce\_mean(alt\_abs\_diff)

    alt\_comp = tf.reduce\_mean(tf.where(alt\_abs\_diff < COMPLETENESS\_THRESHOLD, 1.0, 0.0))

    alt\_acc = tf.reduce\_mean(tf.gather\_nd(alt\_abs\_diff, tf.where(alt\_abs\_diff < COMPLETENESS\_THRESHOLD)))

    alt\_scores.append(alt\_mae)

    alt\_scores.append(alt\_comp)

    alt\_scores.append(alt\_acc)

    return np.array(train\_scores), np.array(test\_scores), np.array(alt\_scores)

def score\_overview(score\_list, train\_loss, path=None):

    """

    Write the overview of losses and scores computed during training to two text files

    1. scores.txt for image and altitude quality during training.

    2. train\_loss.txt for different loss functions during training.

    Parameters:

    score\_list: List of scores, see test\_model.

    train\_loss (list(string)): List of losses memorized during training.

    path (string): Path to save scores

    Outputs:

    psnr (float): PSNR value

    ssim (float): SSIM value

    """

    lines=["It, Train\_PSNR, Train\_PSNR\_std, Train\_SSIM, Train\_SSIM\_std, Test\_PSNR, Test\_PSNR\_std, Test\_SSIM, Test\_SSIM\_std, Alt\_MAE, Alt\_Comp, Alt\_Acc\n"]

    for i, scores in score\_list:

        tr\_p = np.mean(scores[0][:,0]), np.std(scores[0][:,0])

        tr\_s = np.mean(scores[0][:,1]), np.std(scores[0][:,1])

        te\_p = np.mean(scores[1][:,0]), np.std(scores[1][:,0])

        te\_s = np.mean(scores[1][:,1]), np.std(scores[1][:,1])

        lines.append(f"{i},"+",".join([f"{x[0]:.4},{x[1]:.3}" for x in [tr\_p, tr\_s, te\_p, te\_s]])+f",{scores[2][0]},{scores[2][1]},{scores[2][2]}\n")

    if path is not None:

        with open(f"{path}scores.txt", 'w') as f:

            f.writelines(lines)

        if train\_loss is not None:

            with open(f"{path}train\_loss.txt", 'w') as f:

                f.writelines(train\_loss)

    return lines

def prepare\_train\_rays(dataset, arg\_dict):

    """

    Prepare the training and solar correction rays for a given dataset.

    Parameters:

    dataset (dict): Dataset of training images, see generate\_dataset (data\_handling.py).

    arg\_dict (dict): Config variables containing shading options.

    Outputs:

    train\_rays (dict): Training rays, see generate\_train\_rays (render.py).

    sc\_train\_rays (dict): Solar correction rays (render.py).

    """

    train\_rays = render.generate\_train\_rays(dataset, arg\_dict)

    if arg\_dict['train.shad']:

        sc\_train\_rays = render.generate\_train\_light\_correction\_rays(dataset, arg\_dict)

        if arg\_dict['train.shad.custom'] in ['linear', 'rectangle']:

            custom\_sc\_rays = render.generate\_custom\_light\_correction\_rays(dataset, arg\_dict)

            sc\_train\_rays = render.concat\_rays(sc\_train\_rays, custom\_sc\_rays)

    else:

        sc\_train\_rays=None

    return train\_rays, sc\_train\_rays

if \_\_name\_\_ == "\_\_main\_\_":

    # Parse input arguments to arg\_dict

    parser = config\_parser()

    args = parser.parse\_args()

    arg\_dict = vars(args)

    pprint.pprint(arg\_dict)

    # Setup GPU

    os.environ['TF\_FORCE\_GPU\_ALLOW\_GROWTH'] = 'true'

    os.environ['CUDA\_VISIBLE\_DEVICES'] = arg\_dict['gpu']

    # Create dataset of test and train images + metadata

    dataset = data\_handling.generate\_dataset(arg\_dict)

    # Setup output folder

    if not os.path.exists(os.path.dirname(arg\_dict['out.path'])):

        os.makedirs(os.path.dirname(arg\_dict['out.path']))

    # Compute rescale factor if not given

    if arg\_dict['rend.rescale'] is None:

        arg\_dict['rend.rescale'] = render.calculate\_rescale\_factor(dataset)

    # Model initialization

    model = models.generate\_model(arg\_dict)

    print(model['model'].summary())

    # Early ray generation

    print("Generating rays")

    train\_rays, sc\_train\_rays= prepare\_train\_rays(dataset, arg\_dict)

    # Train model

    optimizer = init\_exp\_decay\_adam(arg\_dict['train.lr.init'], arg\_dict['train.n\_epoch'], arg\_dict['train.lr.decay'])

    model, loss, scores = train\_model(model, optimizer, arg\_dict['train.n\_epoch'], arg\_dict, train\_rays, sc\_train\_rays=sc\_train\_rays,  decrease\_noise=True, eval\_dataset=dataset)

    # Write model and training scores

    models.save\_model(arg\_dict['out.path'], model)

    dataset\_rend = render.render\_dataset(dataset, model, ['rgb'], arg\_dict)

    train, test, alt = test\_model(model, dataset, dataset\_rend, arg\_dict)

    scores.append((arg\_dict['train.n\_epoch'], (train, test, alt)))

    score\_overview(scores, loss, arg\_dict['out.path'])