import tensorflow as tf

import numpy as np

from tf\_siren import SinusodialRepresentationDense

from tf\_siren import SIRENModel

# Positional encoding

def posenc(x, L\_embed):

    """

    Basic positional encoding with log-sampling of frequencies. Note that the same function can be used for directional encoding.

    Parameters:

    x (Tensor[N]): variables to apply positional encoding

    L\_embed (int): length of positional encoding

    Outputs:

    y (Tensor[N, L\_embed\*2+1]): positional encoding

    """

    rets = [x]

    for i in range(L\_embed):

        for fn in [tf.sin, tf.cos]:

            rets.append(fn(2.\*\*i \* x))

    return tf.concat(rets, -1)

def posenc\_no\_x(x, L\_embed):

    """Basic positional encoding with log-sampling of frequencies without the initial vector itself. See posenc."""

    rets = []

    for i in range(L\_embed):

        for fn in [tf.sin, tf.cos]:

            rets.append(fn(2.\*\*i \* x))

    return tf.concat(rets, -1)

# Layers

def dense\_siren(W, w0, ker='siren\_uniform'):

    """

    Dense layer of sine activated neurons, based on the library tf\_siren.

    Parameters:

    W (int): Width of layer.

    w0 (float): Base wavelength of sine neurons.

    ker (string): Initializer, set to 'siren\_first\_uniform' for first layer special initialization

    """

    return SinusodialRepresentationDense(W, activation='sine', w0=w0, use\_bias=True, kernel\_initializer=ker)

def dense\_no\_act(W):

    """Dense layer of width W, with no activation function"""

    return tf.keras.layers.Dense(W, activation=None)

def dense\_relu(W):

    """Dense layer width W with ReLU activation function"""

    return tf.keras.layers.Dense(W, activation=tf.keras.layers.ReLU())

def generate\_model(arg\_dict):

    """

    Create and initialize a S-NeRF model according to the model parameters in the config file.

    Parameters:

    arg\_dict (dict): Contains model size, activation functions, inputs and outputs.

    Ouputs:

    model (dict): Model, encodings and dimensions for ease of use.

        model (function): keras model

        emb (function, function): positional and directional encodings

        dim (dict): input and output sizes of different inputs and outputs of the model

    """

    siren\_model = (arg\_dict['model.act'] == 'sin')

    relu\_model = (arg\_dict['model.act'] == 'relu')

    if not (siren\_model or relu\_model):

        print("Unrecognized activation function")

        return None

    # Compute size of input and output dimensions of the network

    input\_ch = 3

    input\_ch\_views = 2 if arg\_dict['model.ins.views'] else 0

    input\_ch\_light = 2 if arg\_dict['model.ins.light'] else 0

    output\_ch\_sigma = 1

    output\_ch\_rgb = 3

    output\_ch\_sh = 1 if arg\_dict['model.outs.shad'] else 0

    output\_ch\_sky = 3 if arg\_dict['model.outs.sky'] else 0

    model\_dims = {'in':[input\_ch, input\_ch\_views, input\_ch\_light],

                  'out':[output\_ch\_sigma, output\_ch\_rgb, output\_ch\_sh, output\_ch\_sky]}

    # Positional encoding

    if (arg\_dict['model.emb.pos'] == 0) or siren\_model:

        embed\_fn\_pos=(lambda x: tf.identity(x))

    else:

        embed\_fn\_pos=(lambda x: posenc(x, arg\_dict['model.emb.pos']))

        input\_ch += input\_ch\*2\*arg\_dict['model.emb.pos']

    # Directional encoding

    if (arg\_dict['model.emb.dir'] == 0):

        embed\_fn\_dir=(lambda x: tf.identity(x))

    else:

        embed\_fn\_dir=(lambda x: posenc\_no\_x(x, arg\_dict['model.emb.dir']))

        input\_ch\_views = input\_ch\_views\*2\*arg\_dict['model.emb.dir']

        input\_ch\_light = input\_ch\_light\*2\*arg\_dict['model.emb.dir']

    # Setup input layer

    inputs = tf.keras.Input(shape=(input\_ch + input\_ch\_light + input\_ch\_views))

    inputs\_pts, inputs\_light, inputs\_views = tf.split(inputs, [input\_ch, input\_ch\_light, input\_ch\_views], -1)

    inputs\_pts.set\_shape([None, input\_ch])

    if input\_ch\_light > 0:

        inputs\_light.set\_shape([None, input\_ch\_light])

    if input\_ch\_views > 0:

        inputs\_views.set\_shape([None, input\_ch\_views])

    # Initial layer

    W = arg\_dict['model.sigma.width']

    if siren\_model:

        D\_sigma = arg\_dict['model.sigma.depth'] - 1

        layer\_fn = lambda x : dense\_siren(x, w0=arg\_dict['model.act.sin.w0'])

        init\_layer\_fn = lambda x : dense\_siren(x, w0=arg\_dict['model.act.sin.w0'], ker='siren\_first\_uniform')

        outputs = init\_layer\_fn(W)(inputs\_pts)

    if relu\_model:

        D\_sigma = arg\_dict['model.sigma.depth']

        layer\_fn = dense\_relu

        outputs = inputs\_pts

    # All other sigma layers

    for i in range(D\_sigma):

        outputs = layer\_fn(W)(outputs)

        if i in arg\_dict['model.sigma.skips']:

            outputs = tf.concat([inputs\_pts, outputs], -1)

    bottleneck = dense\_no\_act(W)(outputs)

    alpha\_out = dense\_no\_act(1)(bottleneck)

    # Color layers

    outputs\_rgb = bottleneck

    if input\_ch\_views > 0:

        outputs\_rgb = tf.concat([outputs\_rgb, inputs\_views])

    for i in range(arg\_dict['model.c.depth']):

        outputs\_rgb = layer\_fn(arg\_dict['model.c.width'])(outputs\_rgb)

    outputs\_rgb = dense\_no\_act(3)(outputs\_rgb)

    # Standard NeRF outputs

    outputs = tf.concat([outputs\_rgb, alpha\_out], -1)

    # Shadow function layers

    if arg\_dict['model.outs.shad']:

        W\_shad = arg\_dict['model.shad.width']

        if siren\_model:

            inputs\_light\_init = init\_layer\_fn(W\_shad)(inputs\_light)

        else:

            inputs\_light\_init = inputs\_light

        outputs\_shad = tf.concat([bottleneck, inputs\_light\_init], -1)

        for i in range(arg\_dict['model.shad.depth']):

            outputs\_shad = dense\_relu(W\_shad)(outputs\_shad)

        outputs\_shad = dense\_no\_act(1)(outputs\_shad)

        outputs = tf.concat([outputs, outputs\_shad], -1)

    # Sky color

    if arg\_dict['model.outs.sky']:

        outputs\_light = dense\_relu(arg\_dict['model.c.width'])(inputs\_light)

        outputs\_light = dense\_no\_act(3)(outputs\_light)

        outputs = tf.concat([outputs, outputs\_light], -1)

    model = tf.keras.Model(inputs=inputs, outputs=outputs)

    return {'model' : model,

            'emb' : (embed\_fn\_pos, embed\_fn\_dir),

            'dim' : model\_dims}

def save\_model(path, model):

    """Save model weights to path"""

    np.save(f"{path}model.npy", model['model'].get\_weights())

def load\_model(path, arg\_dict):

    """Initialize model and load weights from path"""

    model = generate\_model(arg\_dict)

    model['model'].set\_weights(np.load(path, allow\_pickle=True))

    return model