```
#!/bin/env python3.8
"""
Simon Yoon
ECE471 Deep Learning
Professor Curro

I had originally informed by activation function, learning rate, regularization choices by using
different configurations on the TensorFlow NN playground by Carnegie Mellon students, but it did
not work as intended. I had my first few runs with the layers set up as [(2,100),(100,100),(100,1)]
with ELU. tf.nn.sigmoid_cross_entropy_with_logits did not work as intended. It improved when
changed to RELU and when more neurons were added to the hidden layers 100-->200. Sigmoid as final
activation function to keep output between [0,1]. My model was stuck at NaN for loss for nonzero
lambda. Running when lambda = 0, fixed the NaN loss, but changing the lambda and learning rate to
much smaller numbers helped it run with nonzero lambda. Loss was blowing up with more variables and
had to divide by len(model.trainable_variables) to suppress loss.

"""

import os
import logging
import matplotlib.pyplot as plt
import numpy as np
from numpy import pi
import tensorflow as tf

from absl import app
from absl import flags
from tqdm import trange

from dataclasses import dataclass, field, InitVar

script_path = os.path.dirname(os.path.realpath(__file__))

FLAGS = flags.FLAGS
flags.DEFINE_integer("num_features", 2, "Number of features in record")  # spiral count
flags.DEFINE_integer("num_samples", 1000, "Number of samples in dataset")
flags.DEFINE_integer("batch_size", 32, "Number of samples in batch")
flags.DEFINE_integer("num_iters", 1000, "Number of iterations")
flags.DEFINE_float("learning_rate", 0.003, "Learning rate")
flags.DEFINE_integer("random_seed", 31415, "Random seed")
flags.DEFINE_float("sigma_noise", 0.1, "Standard deviation of noise random variable")
flags.DEFINE_float("LAMBDA", 0.001, "Lambda")
flags.DEFINE_list("layers", [(2, 200), (200, 100), (100, 1)], "Layers")
flags.DEFINE_bool("debug", False, "Set logging level to debug")

THETA_MAX = 4 * pi


@dataclass
class SpiralModel:
    x: np.ndarray
    y: np.ndarray
    theta: np.ndarray
    true: np.ndarray


@dataclass
class Data:
    model: SpiralModel
    rng: InitVar[np.random.Generator]
    num_features: int
    num_samples: int
```



```
    sigma: float

    x: np.ndarray = field(init=False)
    y: np.ndarray = field(init=False)
    theta: np.ndarray = field(init=False)
    # true: np.ndarray = field(init=False)

    def __post_init__(self, rng):
        self.index = np.arange(self.num_samples * self.num_features)
        self.theta = rng.uniform(0, THETA_MAX, size=(self.num_samples, 1))
        self.x = np.zeros((self.num_samples * self.num_features, 2), dtype="float32")
        self.y = np.zeros(self.num_samples * self.num_features, dtype="float32")
        # spirals
        # formatting for clean output
        for c in range(self.num_features):
            i = range(self.num_samples * c, self.num_samples * (c + 1))
            r = np.linspace(1, 15, self.num_samples)
            self.theta = (
                np.linspace(c * 3, (c + 4) * 3, self.num_samples)
                + np.random.randn(self.num_samples) * self.sigma
            )
            self.x[i] = np.c_[r * np.sin(self.theta), r * np.cos(self.theta)]
            self.y[i] = c

    def get_batch(self, rng, batch_size):
        """
    Select random subset of examples for training batch
    """
        choices = rng.choice(self.index, size=batch_size)

        return self.x[choices], self.y[choices]


# https://tensorflow.org/guide/core/mlp_core#multilayer_perceptron_mlp_overview
def xavier_init(shape):
    # Computes the xavier initialization values for a weight matrix
    in_dim, out_dim = shape
    xavier_lim = tf.sqrt(6.0) / tf.sqrt(tf.cast(in_dim + out_dim, tf.float32))
    weight_vals = tf.random.uniform(
        shape=(in_dim, out_dim), minval=-xavier_lim, maxval=xavier_lim, seed=22
    )
    return weight_vals


# https://www.tensorflow.org/api_docs/python/tf/Module
class DenseLayer(tf.Module):
    def __init__(self, out_dim, weight_init=xavier_init, activation=tf.identity):
        # Initialize the dimensions and activation functions
        self.out_dim = out_dim
        self.weight_init = weight_init
        self.activation = activation
        self.built = False

    def __call__(self, x):
        if not self.built:
            # Infer the input dimension based on first call
            self.in_dim = x.shape[1]
            # Initialize the weights and biases using Xavier scheme
            self.w = tf.Variable(xavier_init(shape=(self.in_dim, self.out_dim)))
            self.b = tf.Variable(tf.zeros(shape=(self.out_dim,)))
```

```
            self.built = True
        # Compute the forward pass
        # z = tf.add(tf.matmul(x, self.w), self.b)
        z = x @ self.w + self.b
        return self.activation(z)


# https://www.tensorflow.org/api_docs/python/tf/Module
class MLP(tf.Module):
    def __init__(self, layers):
        self.layers = layers

    @tf.function
    def __call__(self, x, rng):
        # Execute the model's layers sequentially
        for layer in self.layers:
            x = layer(x)
        return tf.squeeze(x)

    @property
    def model(self):
        return self.layers


# https://www.tensorflow.org/guide/keras/custom_layers_and_models
def main(a):
    logging.basicConfig()

    if FLAGS.debug:
        logging.getLogger().setLevel(logging.DEBUG)

    # Safe np and tf PRNG
    seed_sequence = np.random.SeedSequence(FLAGS.random_seed)
    np_seed, tf_seed = seed_sequence.spawn(2)
    np_rng = np.random.default_rng(np_seed)
    tf_rng = tf.random.Generator.from_seed(tf_seed.entropy)

    data_generating_model = SpiralModel(
        theta=np_rng.integers(low=0, high=THETA_MAX, size=(FLAGS.num_samples, 1)
),
        x=np_rng.integers(low=0, high=10, size=(FLAGS.num_samples, 1)),
        y=np_rng.integers(low=0, high=10, size=(FLAGS.num_samples, 1)),
        true=np_rng.integers(low=0, high=1, size=(FLAGS.num_samples, 1)),
    )

    data = Data(
        data_generating_model,
        np_rng,
        FLAGS.num_features,
        FLAGS.num_samples,
        FLAGS.sigma_noise,
    )

    model = MLP(
        [
            DenseLayer(200, activation=tf.nn.relu),
            DenseLayer(100, activation=tf.nn.relu),
            DenseLayer(1, activation=tf.math.sigmoid),
        ]
    )

    # https://ai.stackexchange.com/questions/18206/what-kind-of-optimizer-is-sug
```



```
gested-to-use-for-binary-classification-of-similar
    optimizer = tf.keras.optimizers.Adam(learning_rate=FLAGS.learning_rate)

    def cross_entropy(y, y_hat):
        loss = -y * tf.math.log(y_hat) - (1 - y) * tf.math.log(1 - y_hat)
        return tf.reduce_mean(loss)

    bar = trange(FLAGS.num_iters)
    for i in bar:
        with tf.GradientTape() as tape:
            x, y = data.get_batch(np_rng, FLAGS.batch_size)
            y_hat = model(x, tf_rng)
            # loss = tf.reduce_mean(
            #     tf.nn.sigmoid_cross_entropy_with_logits(logits=y_hat, labels=y
)
            # )
            loss = cross_entropy(y, y_hat)
            l2_reg = 0
            for var in model.trainable_variables:
                l2_reg += tf.nn.l2_loss(var)
            loss = tf.reduce_mean(
                loss + FLAGS.LAMBDA * l2_reg / len(model.trainable_variables)
            )

        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
        bar.set_description(f"Loss @ {i} => {loss.numpy():0.6f}")
        bar.refresh()

    # logging.debug(model.model)

    n = 100
    x = np.linspace(-15, 15, n, dtype="float32")
    xx, yy = np.meshgrid(x, x)

    # reshape to fit mat dim of Model
    decision = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            xy = np.reshape([x[j], x[i]], (1, 2))
            p = model(xy, tf_rng)
            if p >= 0.5:
                decision[i, j] = 1
    cm = plt.cm.RdBu

    plt.contourf(xx, yy, decision, cmap=cm, alpha=0.5)
    plt.scatter(
        data.x[:, 0], data.x[:, 1], c=data.y, cmap=cm, alpha=1, edgecolors="black
"
    )
    plt.show()
    plt.tight_layout()
    plt.savefig(f"{script_path}/spiral.pdf")


if __name__ == "__main__":
    app.run(main)
```