```python
#!/usr/bin/env python

# ECE472-Samuel Maltz
# Assignment 2: Classification of spirals using multilayered perceptron (MLP)

# In attempting to find the right MLP for this task I first tried using layers
# (3-4) with smaller widths, then increased the layer widths and found that
# only 2 layers were necessary with sizable widths. I also adjusted the
# learning rate from 0.1 down to 0.001. Finally, an important step which
# ensured stability of the weights turned out to be their initialization. I
# originally sampled from a normal distribution with mean 0, stddev 1 but that
# caused many NaNs when computing the gradients. By decreasing the stddev to
# 0.5 I was able to overcome this problem.

import tensorflow as tf
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

from absl import app
from absl import flags

font = {"size": 10}
matplotlib.rc("font", **font)

FLAGS = flags.FLAGS
flags.DEFINE_integer("num_samples", 500, "Number of samples in dataset")
flags.DEFINE_integer("batch_size", 16, "Number of samples per batch")
flags.DEFINE_integer("num_iter", 3000, "Number of training iterations")
flags.DEFINE_float("learning_rate", 0.001, "Learning rate")
flags.DEFINE_integer("random_seed", 12345, "Random seed")
flags.DEFINE_list("hidden_widths", [100, 200], "Widths of hidden layers")


class Data(object):
    def __init__(self, num_samp, random_seed):
        np.random.seed(random_seed)

        self.num_samp = num_samp
        spiral_samp = num_samp // 2

        # Spiral 1 starts at pi/2 to ensure spirals do not overlap.
        data0 = self.create_data(np.random.uniform(0, 4 * np.pi, spiral_samp), t
ype=-1)
        data1 = self.create_data(
            np.random.uniform(0.5 * np.pi, 4 * np.pi, spiral_samp), type=1
        )

        self.data = np.vstack((data0, data1))
        self.type = np.concatenate((np.zeros((spiral_samp,)), np.ones((spiral_sa
mp,))))

    def create_data(self, t, type):
        x = type * t * np.sin(t)
        y = type * t * np.cos(t)

        return np.transpose(np.vstack((x, y))) + np.random.normal(
            scale=0.2, size=(len(t), 2)
        )

    def get_batch(self, batch_size):
        choices = np.random.choice(self.num_samp, batch_size)
```



```python
        return self.data[choices], self.type[choices]


class Model(tf.Module):
    def __init__(self, widths, random_seed):
        tf.random.set_seed(random_seed)

        self.index = np.arange(len(widths) - 1)

        # List of transition parameters.
        self.W = [
            tf.Variable(tf.random.normal(shape=[widths[i], widths[i + 1]], stdde
v=0.05))
            for i in self.index
        ]

        # List of bias parameters
        self.b = [tf.Variable(tf.zeros(shape=[widths[i + 1]])) for i in self.ind
ex]

    def __call__(self, z, test):
        for i in self.index:
            z = z @ self.W[i] + self.b[i]
            if i == len(self.index) - 1:
                break

            z = tf.maximum(z, 0)  # ReLU

        if test:
            z = tf.math.sigmoid(z)  # tf sigmoid when loss is not computed

        return tf.squeeze(z)

    def loss(self, t, z):
        # Sigmoid final layer and binary cross-entropy loss simplified to
        # maintain stability.
        return tf.reduce_mean(tf.math.log(tf.math.exp(z) + 1) - t * z)


def main(a):
    data = Data(FLAGS.num_samples, FLAGS.random_seed)

    widths = np.concatenate(([2], FLAGS.hidden_widths, [1]))
    model = Model(widths, FLAGS.random_seed)
    optimizer = tf.optimizers.Adam(learning_rate=FLAGS.learning_rate)

    for i in range(FLAGS.num_iter):
        z, t = data.get_batch(FLAGS.batch_size)
        with tf.GradientTape() as tape:
            z = model(z, test=False)
            loss = model.loss(t, z)

        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    # Testing on full training set
    t_hat = model(data.data, test=True)
    print(np.sum(np.around(t_hat.numpy()) != data.type))  # number incorrect

    # Full grid to show boundary of function
    y_grid, x_grid = np.mgrid[15:-15:-0.1, -15:15:0.1]
```

```
    t_grid = model(
        np.transpose(np.vstack((x_grid.flatten(), y_grid.flatten()))), test=True
    )
    t_grid = np.reshape(t_grid, (300, 300))
    spiral_samps = FLAGS.num_samples // 2

    fig, ax = plt.subplots(1, 1)
    ax.plot(data.data[:spiral_samps, 0], data.data[:spiral_samps, 1], "bo")
    ax.plot(data.data[spiral_samps:, 0], data.data[spiral_samps:, 1], "ro")
    c = ax.pcolormesh(x_grid, y_grid, t_grid, shading="nearest")
    fig.colorbar(c)
    ax.set_xlabel("x")
    y_label = ax.set_ylabel("y")
    y_label.set_rotation(0)
    ax.set_title("Spirals classification")
    fig.savefig("spiral_class.pdf")


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