```python
#!/usr/bin/python3

import random

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

from matplotlib.colors import ListedColormap
from tqdm import trange

NUM_SAMPLES = 1000
INPUT_NOISE = 0.5

BATCH_SIZE = 1000
NUM_BATCHES = 3000
NUM_FEATURES = 2
BETA = 0.001

FIRST_LAYER = 1024
SECOND_LAYER = 32
THIRD_LAYER = 16
FOURTH_LAYER = 4
OUTPUT_LAYER = 1

NUM_LAYERS = 5

random.seed(42)

"""
I have learned that the choice of initialization and activation functions drastically
changes the performance of overall function. For the initialization, I initially tried
with normal, but the model failed randomly with extreme case of normal initialization.
Secondly, I tried truncated_normal to avoid samples over and below two standard deviation,
to avoid the saturation region when I tried sigmoid or tanh activation functions.
And then, I researched about popular initialization methods and found out glorot_uniform
that is default initializer for keras models. Glorot_uniform depends on the the number
of input units in the weight tensor and the number of output units in the weight distribution
to find the limit of uniform distribution U(-limit, limit) where limit = sqrt(6/(#in + #out)).
With the Glorot_uniform, the model converged more stably.

For the activation functions, Sigmoid was not considered to be a part of f because of
the gradient vanishing. There are only 5 layers, so the gradient is at least reduced by
(1/4)^5 at the output. Tanh seemed to work fine but the output boundary line seemed to
be easily overfitting. Lastly, relu and its variations (elu, relu6, leaky_relu) were
considered to avoid the gradient vanishing, and there is not much of difference, but elu
seems to be the one that best generalize.

Also, the number of layers was started with 4 layers because spiral dataset is a non-linear
function that does not look similar to activation functions that we commonly use. Therefore,
multiple layers were considered initially. And, I read in one of the textbooks that latter
layers learn higher-level information compared to earlier layers, so I tried to increase the number
of layer instead of increasing the size of each layer. And later on, I tried to increase the size of
earlier layers and decrease the size of latter layer in an assumption that the higher-level information
requires more lower-level information to have a firm foundation. The layer sizes were selected with power
of two for optimization for matrix operations.
"""


class Data(object):
    def __init__(self):
        # spiral generation code snippets inspiration https://gist.github.com/ld86/497e2bcb917d828f3ccd6922345571bd
        half_samples = NUM_SAMPLES // 2
        theta = (1 + 1.75 * np.random.rand(NUM_SAMPLES)) * 2 * np.pi
        x = (
```

```
            np.concatenate(
                (
                    -theta[:half_samples] * np.cos(theta[:half_samples]),
                    theta[half_samples:] * np.cos(theta[half_samples:]),
                )
            )
            + np.random.rand(NUM_SAMPLES) * INPUT_NOISE
        )
        y = (
            np.concatenate(
                (
                    -theta[:half_samples] * np.sin(theta[:half_samples]),
                    theta[half_samples:] * np.sin(theta[half_samples:]),
                )
            )
            + np.random.rand(NUM_SAMPLES) * INPUT_NOISE
        )

        self.X = np.vstack((x, y)).T
        self.X = self.X.astype("float32")
        self.label = (
            np.concatenate((np.zeros(half_samples), np.ones(half_samples)))
            .astype("float32")
            .reshape(NUM_SAMPLES, 1)
        )
        self.idx = np.arange(NUM_SAMPLES)

    def get_batch(self, batch_size=BATCH_SIZE):
        choices = np.random.choice(self.idx, size=batch_size)
        return self.X[choices, :], self.label[choices]


class Model(tf.Module):
    def __init__(self):
        self.initializer = tf.initializers.GlorotUniform()

        w1, b1 = self.generate_layer(NUM_FEATURES, FIRST_LAYER, "w1", "b1", False)
        w2, b2 = self.generate_layer(FIRST_LAYER, SECOND_LAYER, "w2", "b2", False)
        w3, b3 = self.generate_layer(SECOND_LAYER, THIRD_LAYER, "w3", "b3", False)
        w4, b4 = self.generate_layer(THIRD_LAYER, FOURTH_LAYER, "w4", "b4", False)
        w5, b5 = self.generate_layer(FOURTH_LAYER, OUTPUT_LAYER, "w5", "b5", True)

        self.weights = [w1, w2, w3, w4, w5]
        self.biases = [b1, b2, b3, b4, b5]

    def generate_layer(
        self, input_layer_size, out_layer_size, weight_name, bias_name, out
    ):
        w = tf.Variable(
            self.initializer(shape=(input_layer_size, out_layer_size)), name=weight_name
        )
        if out:
            b = tf.Variable(tf.zeros(shape=(1, 1)), name=bias_name)
        else:
            b = tf.Variable(tf.zeros(shape=(1, out_layer_size)), name=bias_name)
        return w, b

    def __call__(self, x):
        for idx, (w, b) in enumerate(zip(self.weights, self.biases)):
            x = x @ w + b
            if idx + 1 == NUM_LAYERS:
                x = tf.math.sigmoid(x)
            else:
                x = tf.nn.relu(x)
        return x


if __name__ == "__main__":
    data = Data()
    model = Model()

    optimizer = tf.optimizers.Adam()
```

```python

    bar = trange(NUM_BATCHES)
    for i in bar:
        with tf.GradientTape() as tape:
            X, y = data.get_batch()
            y_hat = model(X)
            loss = tf.reduce_mean(tf.losses.binary_crossentropy(y_true=y, y_pred=y_hat))
            for w in model.weights:
                loss += tf.nn.l2_loss(w) * BETA
        grads = tape.gradient(loss, model.variables)
        optimizer.apply_gradients(zip(grads, model.variables))
        bar.set_description(f"Loss @ {i} => {loss.numpy():0.6f}")
        bar.refresh()

    plt.subplot(2, 1, 1)
    plt.scatter(
        data.X[:, 0],
        data.X[:, 1],
        c=np.squeeze(data.label),
        cmap=ListedColormap(["#FF0000", "#0000FF"]),
    )
    class_zero = mpatches.Patch(color="#FF0000", label="class 0")
    class_one = mpatches.Patch(color="#0000FF", label="class 1")

    plt.title("ground truth")
    plt.xlabel("x")
    plt.ylabel("y").set_rotation(0)
    plt.legend(handles=[class_zero, class_one])

    plt.subplot(2, 1, 2)
    plt.scatter(
        data.X[:, 0],
        data.X[:, 1],
        c=np.squeeze(model(data.X) > 0.5),
        cmap=ListedColormap(["#FF0000", "#0000FF"]),
    )
    class_zero = mpatches.Patch(color="#FF0000", label="class 0")
    class_one = mpatches.Patch(color="#0000FF", label="class 1")

    plt.title("predicted classes")
    plt.xlabel("x")
    plt.ylabel("y").set_rotation(0)
    plt.legend(handles=[class_zero, class_one])

    x, y = np.meshgrid(
        np.linspace(np.min(data.X[:, 0]), np.max(data.X[:, 0]), 200),
        np.linspace(np.min(data.X[:, 1]), np.max(data.X[:, 1]), 200),
    )

    z = np.vstack((x.flatten(), y.flatten())).T
    z = tf.reshape(model(z), x.shape)

    plt.contour(x, y, np.squeeze(z), levels=1, colors="k")

    plt.show()

```