```
#!/bin/env python3.8

"""
Simon Yoon
ECE470 Deep Learning
Professor Curro

Took inspiration from Professor Curro's example submission

Husam Almanakly and Michael Bentivegna helped debug;
There were issues in methods that I used both tf and np simultaneously in.
Changing it so that it was one or the either locally fixed it
-believed it to be a datatype handling issue

i.e tf.zeros vs np.zeros, tf.exp vs np.exp

./tf.sh  gs -sDEVICE=pdfwrite -dNOPAUSE -dBATCH -dSAFER -sOutputFile=main.py.pdf hw1/main.py.pdf hw1/
fit.pdf
"""

import os
import logging
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
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__))


@dataclass
class LinearModel:
    weights: np.ndarray
    bias: float

    m: np.ndarray  # mu
    s: np.ndarray  # sigma


@dataclass
class Data:
    model: LinearModel
    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)

    def __post_init__(self, rng):
        self.index = np.arange(self.num_samples)
        self.x = rng.uniform(
            0.1, 0.9, size=(self.num_samples, 1)
        )  # self.num_features -> 1
        clean_y = np.sin(2 * np.pi * self.x)  # + epsilon
        e = rng.normal(
```



```
            loc=0, scale=0.2, size=(self.num_samples, 1)
        )  # e_i is drawn from N(O,sigma_noise) something close to 0.1
        self.y = clean_y + e  # y_i = sin(2*pi*x_i)+e

    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].flatten()


"""
def compare_linear_models(a: LinearModel, b: LinearModel):
    for w_a, w_b in zip(a.weights, b.weights):
        print(f"{w_a:0.2f}, {w_b:0.2f}")

    print(f"{a.bias:0.2f}, {b.bias:0.2f}")
"""

font = {
    # "family": "Adobe Caslon Pro",
    "size": 10,
}

matplotlib.style.use("classic")
matplotlib.rc("font", **font)

FLAGS = flags.FLAGS
flags.DEFINE_integer("num_features", 6, "Number of bases in record")  # M = 6
flags.DEFINE_integer("num_samples", 50, "Number of samples in dataset")  # N = 50
flags.DEFINE_integer("batch_size", 16, "Number of samples in batch")
flags.DEFINE_integer("num_iters", 3000, "Number of SGD iterations")
flags.DEFINE_float(
    "learning_rate", 0.05, "Learning rate / step size for SGD"
)  # 0.1 -> 0.05 for oscillation on smaller magnitude iters
flags.DEFINE_integer("random_seed", 31415, "Random seed")
flags.DEFINE_float(
    "sigma_noise", 0.1, "Standard deviation of noise random variable"
)  # 0.5 -> 0.1
flags.DEFINE_bool("debug", False, "Set logging level to debug")


class Model(tf.Module):
    def __init__(self, rng, num_features):
        """
        A plain linear regression model with a bias term
        altered to be a regression model to estimate sinusoid
        with weights (w), bias (b), mu (m), sigma (s)
        """
        self.num_features = num_features
        # trainable params
        self.w = tf.Variable(rng.normal(shape=[self.num_features]), name="weights
")
        self.b = tf.Variable(tf.zeros(shape=[1, 1]), name="bias")
        self.m = tf.Variable(rng.normal(shape=[self.num_features]), name="mean")
        # mu
        self.s = tf.Variable(
            rng.normal(shape=[self.num_features]), name="sigma"
        )  # stdev
```

```python
    def __call__(self, x):
        """
        Functional form with expanded phi
        Gaussian basis functions
        """
        gau = tf.zeros(shape=(x.shape[0], 1))
        """
        for i in range(self.num_features):
            gau += tf.math.exp(-1 * (((x - self.m[i]) ** 2) / (self.s[i] ** 2)))
        gau += self.b
        """
        gau = self.w * tf.math.exp(-1 * (((x - self.m) ** 2) / (self.s**2)))
        return tf.squeeze(tf.reduce_sum(gau, 1) + self.b)  # incl mu, sig est

    @property
    def model(self):
        return LinearModel(
            self.m.numpy().reshape([self.num_features]),
            self.s.numpy().reshape([self.num_features]),
            self.w.numpy().reshape([self.num_features]),
            self.b.numpy().squeeze(),
        )


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 = LinearModel(
        weights=np_rng.integers(low=0, high=5, size=(FLAGS.num_features)),
        bias=2,
        m=np_rng.integers(low=0, high=1, size=(FLAGS.num_features)),
        s=np_rng.integers(low=0, high=1, size=(FLAGS.num_features)),
    )
    logging.debug(data_generating_model)

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

    model = Model(tf_rng, FLAGS.num_features)
    logging.debug(model.model)

    optimizer = tf.optimizers.SGD(learning_rate=FLAGS.learning_rate)

    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)
```



```python
            loss = 0.5 * tf.reduce_mean(
                (y_hat - y) ** 2
            )  # implement loss function to minimize

        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)

    # print out true values versus estimates
    print("w,   w_hat")
    """
    compare_linear_models(data.model, model.model)
    """

    logging.info(f"Mu: {model.m}")

    fig, ax = plt.subplots(1, 2, figsize=(10, 3), dpi=200)
    ax[0].set_title("Linear Regression on Sinewave")
    ax[0].set_xlabel("x")
    h = ax[0].set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    # plot with noise
    xs = np.linspace(-1.0, 2, 1000)
    xs = xs[:, np.newaxis]
    yhat = model(xs)
    ax[0].plot(xs, np.squeeze(yhat), "--", color="green")
    ax[0].plot(np.squeeze(data.x), data.y, "o", color="blue")
    ax[0].set_ylim(np.amin(data.y) * 1.5, np.amax(data.y) * 1.5)
    ax[0].set_xlim(0.1, 0.9)

    # plot sine
    true_y = np.sin(2 * np.pi * xs)
    ax[0].plot(xs, true_y, color="red")
    ax[0].legend(["Regression Estimate", "Data", "True Sine"], fontsize=9)

    # plot bases
    ax[1].set_title("Basis Functions")
    ax[1].set_xlabel("x")
    h = ax[1].set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    # plot gaussians
    gaussians = np.zeros((1000, model.num_features))
    for j in range(model.num_features):
        gaussians[:, j] = tf.math.exp(
            -1 * ((xs.T - model.m[j]) ** 2) / (model.s[j] ** 2)
        )
    ax[1].plot(xs, gaussians)

    plt.tight_layout()
    plt.savefig(f"{script_path}/fit.pdf")


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