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

"""
Example assignment. Author: Chris Curro
"""
import os
import logging

# from turtle import shape
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import math

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


@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, 2, size=(self.num_samples, self.num_features))
        clean_y = tf.math.sin(2 * math.pi * self.x)
        self.y = rng.normal(loc=clean_y, scale=self.sigma)

    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", 1, "Number of features in record")
flags.DEFINE_integer("num_samples", 120, "Number of samples in dataset")
flags.DEFINE_integer("batch_size", 3, "Number of samples in batch")
flags.DEFINE_integer("num_iters", 100, "Number of SGD iterations")
flags.DEFINE_float("learning_rate", 0.1, "Learning rate / step size for SGD")
flags.DEFINE_integer("random_seed", 31415, "Random seed")
flags.DEFINE_float("sigma_noise", 0.1, "Standard deviation of noise random variable")
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
    """
        self.m = 3
        self.num_features = num_features
        self.w = tf.constant(
            tf.ones([self.m, self.num_features], dtype=tf.dtypes.float32, name=N
one)
        )
        # self.sigma = tf.constant(tf.fill([self.num_features, self.m], 0.5))

        # self.w = tf.Variable(rng.normal(shape=[self.m, self.num_features]))
        self.b = tf.Variable(tf.zeros(shape=[1, 1]))
        self.mu = tf.Variable(rng.normal(shape=[self.num_features, self.m]))
        self.sigma = tf.Variable(rng.normal(shape=[self.num_features, self.m]))

    def __call__(self, x):
        print(x)
        leni = tf.shape(x).numpy()[0]
        step_1 = tf.cast(tf.broadcast_to(x, [leni, self.m]), tf.float32)
        print("STEP 1: " + str(tf.shape(step_1)))
        print("STEP MU: " + str(tf.shape(self.mu)))
        print("MU: " + str(self.mu.numpy()))
        step_2 = tf.cast(tf.broadcast_to(self.mu, [leni, self.m]), tf.float32)
        print("STEP 2: " + str(tf.shape(step_2)))
        print("STEP 2: " + str(step_2.numpy()))

        step_3 = step_1 - step_2
        print("STEP 3: " + str(step_3.numpy()))

        step_4 = -((step_3) ** 2) / tf.broadcast_to(self.sigma, [leni, self.m])
        print("sigma" + str(self.sigma.numpy()))
        print("STEP 4: " + str(step_4.numpy()))

        print("WEIGHTS: " + str(self.w.numpy()))
        step_5 = step_4 @ self.w + self.b
        print("STEP 5: " + str(step_5.numpy()))
        # print(step_5)
        # print(tf.shape(tf.squeeze(x @ self.w + self.b)))
        return tf.squeeze(step_5)
```

```
    @property
    def model(self):
        return LinearModel(self.w.numpy().reshape([self.m]), self.b.numpy().sque
eze())


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
    )
    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)
            loss = 0.5 * tf.reduce_mean((y_hat - y) ** 2)
            print(loss)

        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)

    if FLAGS.num_features > 1:
        # Only continue to plotting if x is a scalar
        exit(0)

    fig, ax = plt.subplots(1, 1, figsize=(11, 8), dpi=200)

    ax.set_title("Linear fit")
    ax.set_xlabel("x")
```



```
    ax.set_ylim(np.amin(data.y) * 10.5, np.amax(data.y) * 10.5)
    h = ax.set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    xs = np.linspace(0, 2, 10)
    xs = xs[:, np.newaxis]
    ax.plot(xs, np.squeeze(model(xs)), "-", np.squeeze(data.x), data.y, "o")

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


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