TF_test_gradient.py

```python
# Import necessary libraries:
import tensorflow as tf
import tensorflow.keras.backend as K
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Generate random example data
'''
This code generates random input data x_train with a shape of (100, 10) and random output data y_train with a shape of (100, 1). The output dimension is modified to 2 in the comment, but the code still generates a 1-dimensional output.
'''
x_train = np.random.rand(100, 10)
y_train = np.random.rand(100, 1)  # Modify output dimension to 2

# Create a TensorFlow variable for input data
'''
The inputdata variable is created as a TensorFlow variable using tf.Variable(). It holds the x_train data and allows it to be used in the custom loss function.
'''
inputdata = tf.Variable(x_train)

# Define a custom loss function
'''
The custom loss function is defined to calculate both the mean squared error (MSE) loss and the mean absolute error (MAE) loss.

The y_true argument represents the true labels, and y_pred represents the predicted labels by the model. The MSE loss is calculated between the first column of y_true and y_pred.

The code then computes the output of the model (out) by passing the inputdata through the model. It then computes the gradient of the first column of out with respect to the inputdata using tf.gradients(). The resulting gradient is cast to float32.

Finally, the MAE loss is computed by taking the absolute difference between the second column of the gradient (grad1[:, 1]) and the second column of the output (out[:, 1]).

The function returns the sum of the MSE and MAE losses.
'''
def custom_loss(y_true, y_pred):
    mse_loss = tf.reduce_mean(tf.square(y_true[:, 0] - y_pred[:, 0]))  # Mean squared error

    out = model(inputdata)
    grad1 = tf.gradients(out[:, 0], inputdata)[0][:, 1]
    grad1 = tf.cast(grad1,tf.float32)
    mae_loss = tf.reduce_mean(tf.abs(grad1 - out[:, 1]))  # Mean absolute error with derivative

    return mse_loss + mae_loss  # Combine the two losses

# Define your network architecture
'''
The model is defined as a sequential model using tf.keras.models.Sequential(). It consists of two dense layers with ReLU activation and 64 units each. The input shape is (10,), and the output dimension is modified to 2.
'''
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(10,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(2)  # Modify output dimension to 2
])

# Compile the model with custom loss function
'''
The model is compiled with the Adam optimizer and the custom loss function defined earlier.
'''
model.compile(optimizer='adam', loss=custom_loss)

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32)

# Plot the model's fitting history
losses = pd.DataFrame(model.history.history)
losses.plot()
plt.show()
```