Start coding or generate with AI.

Estimating λ and μ, Inverse Problem using PINN

```
#  PINNs for Inverse Problem: Estimating λ and μ
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Set seed for reproducibility
tf.random.set_seed(0)
np.random.seed(0)

# 📌 Domain
N = 10000  # number of collocation points
x = np.random.rand(N, 1)
y = np.random.rand(N, 1)
X = np.hstack([x, y])

# 🎯 Exact displacements (ground truth for inverse learning)
Q = 4.0
u_exact = np.cos(2 * np.pi * x) * np.sin(np.pi * y)
v_exact = (Q / 4) * np.sin(np.pi * x) * y**4

# Convert to tensors
X_tf = tf.convert_to_tensor(X, dtype=tf.float32)
u_data_tf = tf.convert_to_tensor(u_exact, dtype=tf.float32)
v_data_tf = tf.convert_to_tensor(v_exact, dtype=tf.float32)

# 🧠 Neural Net model for displacement field
def create_model():
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape=(2,)))
    for _ in range(4):
        model.add(tf.keras.layers.Dense(50, activation='tanh'))
    model.add(tf.keras.layers.Dense(2))  # output: u, v
    return model

model = create_model()

# 🎯 Trainable Lamé parameters
lambda_param = tf.Variable(1.0, dtype=tf.float32, trainable=True)
mu_param = tf.Variable(0.5, dtype=tf.float32, trainable=True)

# 📐 Automatic differentiation to compute physics loss
def compute_loss(X):

    with tf.GradientTape(persistent=True) as tape2:
        tape2.watch(X)
        with tf.GradientTape(persistent=True) as tape1:
            tape1.watch(X)
            uv = model(X)
            u = uv[:, 0:1]
            v = uv[:, 1:2]

        u_x = tape1.gradient(u, X)[:, 0:1]
        u_y = tape1.gradient(u, X)[:, 1:2]
        v_x = tape1.gradient(v, X)[:, 0:1]
        v_y = tape1.gradient(v, X)[:, 1:2]

        # Strain components (plane strain)
        ε_xx = u_x
        ε_yy = v_y
        ε_xy = 0.5 * (u_y + v_x)

        # Stress components using constitutive model
        σ_xx = (lambda_param + 2 * mu_param) * ε_xx + lambda_param * ε_yy
        σ_yy = lambda_param * ε_xx + (lambda_param + 2 * mu_param) * ε_yy
        σ_xy = 2 * mu_param * ε_xy

    σ_xx_x = tape2.gradient(σ_xx, X)[:, 0:1]
    σ_xy_y = tape2.gradient(σ_xy, X)[:, 1:2]
    σ_xy_x = tape2.gradient(σ_xy, X)[:, 0:1]
    σ_yy_y = tape2.gradient(σ_yy, X)[:, 1:2]

    # Equilibrium equations: residuals
    fx = λ_fn(X) + μ_fn(X)  # known from analytical expression (optional)
    fy = λ_fn2(X) + μ_fn2(X)
```

```python
    res_x = σ_xx_x + σ_xy_y - fx
    res_y = σ_xy_x + σ_yy_y - fy

    # Physics-informed loss
    physics_loss = tf.reduce_mean(tf.square(res_x)) + tf.reduce_mean(tf.square(res_y))

    # Data loss
    u_pred = u
    v_pred = v
    data_loss = tf.reduce_mean(tf.square(u_pred - u_data_tf)) + tf.reduce_mean(tf.square(v_pred - v_data_tf))

    total_loss = physics_loss + data_loss
    return total_loss

# Body force expressions (based on exact solution)
# def λ_fn(X):
#     x, y = X[:, 0:1], X[:, 1:2]
#     return (4 * np.pi**2 * np.cos(2*np.pi*x) * np.sin(np.pi*y)
#             - np.pi * np.cos(np.pi*x) * (Q*y**3))

# def μ_fn(X):
#     x, y = X[:, 0:1], X[:, 1:2]
#     return (9 * np.pi**2 * np.cos(2*np.pi*x) * np.sin(np.pi*y)
#             - np.pi * np.cos(np.pi*x) * (Q*y**3))

# def λ_fn2(X):
#     x, y = X[:, 0:1], X[:, 1:2]
#     return (-3 * np.sin(np.pi*x) * Q * y**2 +
#             2 * np.pi**2 * np.sin(2*np.pi*x) * np.cos(np.pi*y))

# def μ_fn2(X):
#     x, y = X[:, 0:1], X[:, 1:2]
#     return (-6 * np.sin(np.pi*x) * Q * y**2 +
#             2 * np.pi**2 * np.sin(2*np.pi*x) * np.cos(np.pi*y) +
#             np.pi**2 * np.sin(np.pi*x) * (Q * y**4 / 4))
def λ_fn(X):
    x, y = X[:, 0:1], X[:, 1:2]
    pi = tf.constant(np.pi, dtype=tf.float32)
    return (4 * pi**2 * tf.cos(2 * pi * x) * tf.sin(pi * y)
            - pi * tf.cos(pi * x) * (Q * y**3))

def μ_fn(X):
    x, y = X[:, 0:1], X[:, 1:2]
    pi = tf.constant(np.pi, dtype=tf.float32)
    return (9 * pi**2 * tf.cos(2 * pi * x) * tf.sin(pi * y)
            - pi * tf.cos(pi * x) * (Q * y**3))

def λ_fn2(X):
    x, y = X[:, 0:1], X[:, 1:2]
    pi = tf.constant(np.pi, dtype=tf.float32)
    return (-3 * tf.sin(pi * x) * Q * y**2 +
            2 * pi**2 * tf.sin(2 * pi * x) * tf.cos(pi * y))

def μ_fn2(X):
    x, y = X[:, 0:1], X[:, 1:2]
    pi = tf.constant(np.pi, dtype=tf.float32)
    return (-6 * tf.sin(pi * x) * Q * y**2 +
            2 * pi**2 * tf.sin(2 * pi * x) * tf.cos(pi * y) +
            pi**2 * tf.sin(pi * x) * (Q * y**4 / 4))

# 🛠️ Optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

# 🔁 Training loop
@tf.function
def train_step():
    with tf.GradientTape() as tape:
        loss = compute_loss(X_tf)
    grads = tape.gradient(loss, model.trainable_variables + [lambda_param, mu_param])
    optimizer.apply_gradients(zip(grads, model.trainable_variables + [lambda_param, mu_param]))
    return loss

# 🔁 Training
epochs = 5000
for epoch in range(epochs):
    loss = train_step()
    if epoch % 500 == 0:
        print(f"Epoch {epoch}: Loss = {loss.numpy():.5e}, λ = {lambda_param.numpy():.4f}, μ = {mu_param.numpy():.4f}")

print("\nEstimated Parameters:")
print(f"λ (lambda): {lambda_param.numpy():.4f}")
print(f"μ (mu): {mu_param.numpy():.4f}")
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/input_layer.py:27: UserWarning: Argument `input_shape` is deprecated.
  warnings.warn(
Epoch 0: Loss = 4.59806e+03, λ = 0.9990, μ = 0.4990
Epoch 500: Loss = 7.12189e+00, λ = 1.1414, μ = 0.6482
Epoch 1000: Loss = 4.50520e+00, λ = 1.1581, μ = 0.6661
Epoch 1500: Loss = 3.25345e+00, λ = 1.1744, μ = 0.6830
Epoch 2000: Loss = 1.47946e+01, λ = 1.1906, μ = 0.7018
Epoch 2500: Loss = 1.89479e+00, λ = 1.2048, μ = 0.7210
Epoch 3000: Loss = 1.56633e+00, λ = 1.2176, μ = 0.7412
Epoch 3500: Loss = 1.41599e+00, λ = 1.2306, μ = 0.7634
Epoch 4000: Loss = 1.26862e+00, λ = 1.2420, μ = 0.7840
Epoch 4500: Loss = 1.19976e+00, λ = 1.2529, μ = 0.8035

Estimated Parameters:
λ (lambda): 1.2640
μ (mu): 0.8226
```