import torch  
import pandas as pd  
import numpy as np  
from torch.autograd import grad  
from torch.utils.data import TensorDataset, random\_split, DataLoader  
  
  
# 读取CSV文件  
device=torch.device("cuda"if torch.cuda.is\_available() else"cpu")  
df = pd.read\_csv(r"C:\Users\亓宇鹏\Documents\Tencent Files\1134944482\FileRecv\数据集929.csv")  
  
  
# 假设 df 是你的原始数据  
# 初始化一个空的DataFrame，用于存储重塑后的数据  
reshaped\_df = pd.DataFrame()  
  
# 提取前3列  
first\_three\_columns = df.iloc[:, :3]  
first\_14\_columns = df.iloc[:, :14]  
# 保存第一组的列名  
first\_14\_column\_names = first\_14\_columns.columns.tolist()  
  
# 按照每11列为一组进行重塑，从第14列开始  
num\_columns = df.shape[1]  
num\_rows = df.shape[0]  
first\_group = True  
for i in range(14, num\_columns, 11):  
 # 提取当前组的列  
 group\_df1 = df.iloc[:, i:i + 11]  
  
 # 将前3列添加到当前组的前面  
 group\_df = pd.concat([first\_three\_columns, group\_df1], axis=1)  
  
 # 对于除第一组之外的其他组，去掉表头  
  
 group\_df.columns = range(group\_df.shape[1])  
  
  
 # 将当前组的列堆叠到第一组的下面  
 reshaped\_df = pd.concat([reshaped\_df, group\_df], ignore\_index=True)  
# 将第一组的列名应用到整个DataFrame上  
reshaped\_df.columns = first\_14\_column\_names + reshaped\_df.columns[len(first\_14\_column\_names):].tolist()  
  
reshaped\_df=pd.concat([first\_14\_columns, reshaped\_df]).reset\_index(drop=True)  
  
  
# 重塑后的数据行数应该是原始数据行数的倍数  
num\_groups = (num\_columns - 14) // 11 + 1  
reshaped\_df = reshaped\_df.iloc[:num\_rows \* num\_groups, :]  
  
# 保存重塑后的数据到一个新的CSV文件  
reshaped\_df.to\_csv('reshaped\_data.csv', index=False)  
print(reshaped\_df)  
  
# 假设前4列是输入数据，第5列是标签  
qiansilie = torch.tensor(reshaped\_df.iloc[:, :4].values, dtype=torch.float32) # 使用适当的数据类型  
diwulie = torch.tensor(reshaped\_df.iloc[:, 4].values, dtype=torch.long) # 使用适当的数据类型  
  
# 创建一个TensorDataset  
dataset = TensorDataset(qiansilie, diwulie)  
  
# 定义训练集和验证集的大小  
train\_size = int(0.8 \* len(dataset)) # 假设训练集是数据集的80%  
val\_size = len(dataset) - train\_size  
  
# 使用random\_split分割数据集  
train\_dataset, val\_dataset = random\_split(dataset, [train\_size, val\_size])  
  
# 创建数据加载器  
train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)  
val\_loader = DataLoader(val\_dataset, batch\_size=64)  
  
  
# 定义芯片的几何形状  
pcb = {'x': [-0.08, 0.08], 'y': [-0.08, 0.08], 'z': [0, 0.0065]} # pcb板  
adhesive = {'x': [-0.04, 0.04], 'y': [-0.04, 0.04], 'z': [0.0065, 0.007]} # 粘合剂层  
chip = {'x': [-0.04, 0.04], 'y': [-0.04,0.04], 'z': [0.07, 0.0135]} # 芯片  
  
# 定义热交换系数和空气温度  
h = 20  
T\_air = 293.15  
  
# 定义热源  
def heat\_source(x, y, z, t):  
 condition\_chip = (chip['x'][0] <= x) & (x <= chip['x'][1]) & (chip['y'][0] <= y) & (y <= chip['y'][1]) & (chip['z'][0] <= z) & (z <= chip['z'][1])  
  
 source = torch.where(condition\_chip, 2.404e7, 0)  
 return source  
  
  
# 定义热导率函数  
def get\_k(x, y, z, k\_pcb, k\_adhesive, k\_chip):  
 condition\_pcb = (pcb['x'][0] <= x) & (x <= pcb['x'][1]) & (pcb['y'][0] <= y) & (y <= pcb['y'][1]) & (pcb['z'][0] <= z) & (z <= pcb['z'][1])  
 condition\_adhesive = (adhesive['x'][0] <= x) & (x <= adhesive['x'][1]) & (adhesive['y'][0] <= y) & (y <= adhesive['y'][1]) & (adhesive['z'][0] <= z) & (z <= adhesive['z'][1])  
 condition\_chip = (chip['x'][0] <= x) & (x <= chip['x'][1]) & (chip['y'][0] <= y) & (y <= chip['y'][1]) & (chip['z'][0] <= z) & (z <= chip['z'][1])  
  
 k = torch.where(condition\_pcb, k\_pcb, torch.where(condition\_adhesive, k\_adhesive, torch.where(condition\_chip, k\_chip, 0)))  
 return k  
 # 如果不在任何定义的区域内，我们可以假设 k = 0  
  
# 定义密度和比热容  
def get\_rho\_and\_c(x, y, z, rho\_pcb, rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive, c\_chip):  
 condition\_pcb = (pcb['x'][0] <= x) & (x <= pcb['x'][1]) & (pcb['y'][0] <= y) & (y <= pcb['y'][1]) & (pcb['z'][0] <= z) & (z <= pcb['z'][1])  
 condition\_adhesive = (adhesive['x'][0] <= x) & (x <= adhesive['x'][1]) & (adhesive['y'][0] <= y) & (y <= adhesive['y'][1]) & (adhesive['z'][0] <= z) & (z <= adhesive['z'][1])  
 condition\_chip = (chip['x'][0] <= x) & (x <= chip['x'][1]) & (chip['y'][0] <= y) & (y <= chip['y'][1]) & (chip['z'][0] <= z) & (z <= chip['z'][1])  
  
 rho = torch.where(condition\_pcb, rho\_pcb, torch.where(condition\_adhesive, rho\_adhesive, torch.where(condition\_chip, rho\_chip, 0)))  
 c = torch.where(condition\_pcb, c\_pcb, torch.where(condition\_adhesive, c\_adhesive, torch.where(condition\_chip, c\_chip, 0)))  
  
 return rho, c  
 # 如果不在任何定义的区域内，我们可以假设rho = 0, c = 0  
#定义  
rho\_pcb=1900  
rho\_adhesive=1673  
rho\_chip=2329  
c\_pcb=1369  
c\_adhesive=1000  
c\_chip=700  
# 热传导系数  
k\_pcb = 0.3  
k\_adhesive = 2.5  
k\_chip = 80  
# 这是温度输入模型  
class TemperatureModel(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super(TemperatureModel, self).\_\_init\_\_()  
 self.net = torch.nn.Sequential(  
 torch.nn.Linear(4, 50),  
 torch.nn.ReLU(),  
 torch.nn.Linear(50, 50),  
 torch.nn.ReLU(),  
 torch.nn.Linear(50, 1)  
 )  
  
 def forward(self, x, create\_graph=True):  
 return self.net(x)  
  
  
# 创建模型实例并移动到GPU  
model = TemperatureModel().to(device)  
  
  
# 定义温度函数  
def Tmodel(x, y, z, t, model,create\_graph=False):  
 x.requires\_grad\_(True)  
 y.requires\_grad\_(True)  
 z.requires\_grad\_(True)  
 t.requires\_grad\_(True)  
 T = model(torch.stack([x, y, z, t], dim=-1).to(device), create\_graph=True) # 使用模型预测温度  
  
 grad\_outputs = torch.ones\_like(T)  
 T\_x = grad(T, x, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0] # 计算T对x的导数  
 T\_y = grad(T, y, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0] # 计算T对y的导数  
 T\_z = grad(T, z, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0] # 计算T对z的导数  
 T\_t = grad(T, t, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0] # 计算T对z的导数  
  
 return T, T\_x, T\_y, T\_z, T\_t  
  
  
  
# 定义热传导方程  
def heat\_equation(x, y, z, t, model, k\_pcb, k\_adhesive, k\_chip, rho\_pcb, rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive,  
 c\_chip):  
 k = get\_k(x, y, z, k\_pcb, k\_adhesive, k\_chip)  
 rho, c = get\_rho\_and\_c(x, y, z, rho\_pcb, rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive, c\_chip)  
 T, T\_x, T\_y, T\_z,T\_t = Tmodel(x, y, z, t, model)  
  
 grad\_outputs = torch.ones\_like(T).squeeze()  
  
 T\_xx = grad(T\_x, x, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
 T\_yy = grad(T\_y, y, grad\_outputs=grad\_outputs, create\_graph=True,allow\_unused=True)[0]  
 T\_zz = grad(T\_z, z, grad\_outputs=grad\_outputs, create\_graph=True,allow\_unused=True)[0]  
  
 f = rho \* c \* T\_t - (k \* T\_xx + k \* T\_yy + k \* T\_zz) - heat\_source(x, y, z, t) # 热传导方程  
 return f  
  
  
# 定义边界条件  
def boundary\_conditions(x, y, z, t, model):  
 k = get\_k(x, y, z, k\_pcb, k\_adhesive, k\_chip) # 获取相应的热导率  
 # 上表面与空气进行热交换  
 z\_top = torch.full\_like(x, chip['z'][1]).requires\_grad\_(True)  
 T, \_, \_, T\_z, \_ = Tmodel(x, y, z\_top, t, model)  
 conditions = [h \* (T - T\_air) - k \* T\_z]  
  
 # 下表面是绝热的  
 z\_bottom = torch.full\_like(x, chip['z'][0]).requires\_grad\_(True)  
 T, \_, \_, T\_z, \_ = Tmodel(x, y, z\_bottom, t, model)  
  
 conditions.append(h \* (T - T\_air) - k \* T\_z)  
  
 for boundary in [pcb, adhesive, chip]:  
 for i in range(2):  
 x\_temp = torch.full\_like(x,boundary['x'][i],requires\_grad=True )  
 T, T\_x, \_, \_ , \_ = Tmodel(x\_temp, y, z, t, model)  
 conditions.append(h \* (T - T\_air) - k \* T\_x)  
 y\_temp = torch.full\_like(x,boundary['y'][i], requires\_grad=True )  
 T, \_, T\_y, \_, \_ = Tmodel(x, y\_temp, z, t, model)  
 conditions.append(h \* (T - T\_air) - k \* T\_y)  
  
 return conditions  
  
  
# 定义温度初始条件  
def initial\_conditions(x, y, z, t, model):  
 T, \_, \_, \_ ,\_= Tmodel(x, y, z, torch.zeros\_like(t).to(device), model)  
 return T - 293.15  
  
  
# 定义损失函数  
def losstem(x, y, z, t, T\_actual, Tmodel, k\_pcb, k\_adhesive, k\_chip, rho\_pcb, rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive, c\_chip,create\_graph=True):  
 f = heat\_equation(x, y, z, t, model, k\_pcb, k\_adhesive, k\_chip, rho\_pcb, rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive, c\_chip)  
 conditions = boundary\_conditions(x, y, z, t, model)  
 ic = initial\_conditions(x, y, z, t,model)  
 T, \_, \_, \_ , \_= Tmodel(x, y, z, t, model, create\_graph=True)  
 loss = torch.mean(f \*\* 2) + torch.mean(torch.stack(conditions) \*\* 2) + torch.mean(ic \*\* 2) + torch.mean(  
 (T - T\_actual) \*\* 2)#包含了方程 边界条件 初始调价 data条件  
 return loss  
  
  
  
  
  
  
  
  
 # 在这一步，你的 boundary\_conditions 函数应该在一批数据上进行计算，并将结果保存下来。  
 # 例如，你可以将结果添加到一个列表中，或者直接写入文件，以便后续处理。  
 # 注意，你可能需要修改 boundary\_conditions 函数，使其能够处理批量数据。  
  
######################################################################################  
  
  
class DisplacementModel(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super(DisplacementModel, self).\_\_init\_\_()  
 self.net = torch.nn.Sequential(  
 torch.nn.Linear(4, 50),#位移的输入应该是x y z T  
 torch.nn.ReLU(),  
 torch.nn.Linear(50, 50),  
 torch.nn.ReLU(),  
 torch.nn.Linear(50, 3)  
 )  
  
 def forward(self, x):  
 return self.net(x)  
  
# 创建位移模型实例并移动到GPU  
displacement\_model = DisplacementModel().to(device)  
  
# 热膨胀系数  
alpha\_chip = 2.6E-6  
alpha\_adhesive = 22E-6  
alpha\_pcb = 18E-6  
  
# 弹性模量和泊松比  
E\_pcb = 22000  
mu\_pcb = 0.15  
E\_adhesive = 7789  
mu\_adhesive = 0.311  
E\_chip = 106600  
mu\_chip = 0.25  
  
  
  
# 定义热膨胀系数函数  
def get\_alpha(x, y, z, alpha\_pcb, alpha\_adhesive, alpha\_chip):  
 condition\_pcb = (pcb['x'][0] <= x) & (x <= pcb['x'][1]) & (pcb['y'][0] <= y) & (y <= pcb['y'][1]) & (pcb['z'][0] <= z) & (z <= pcb['z'][1])  
 condition\_adhesive = (adhesive['x'][0] <= x) & (x <= adhesive['x'][1]) & (adhesive['y'][0] <= y) & (y <= adhesive['y'][1]) & (adhesive['z'][0] <= z) & (z <= adhesive['z'][1])  
 condition\_chip = (chip['x'][0] <= x) & (x <= chip['x'][1]) & (chip['y'][0] <= y) & (y <= chip['y'][1]) & (chip['z'][0] <= z) & (z <= chip['z'][1])  
  
 alpha = torch.where(condition\_pcb, alpha\_pcb, torch.where(condition\_adhesive, alpha\_adhesive, torch.where(condition\_chip, alpha\_chip, 0)))  
  
 return alpha  
 # 如果不在任何定义的区域内，我们可以假设 alpha = 0  
  
  
  
  
  
# 定义弹性模量和泊松比函数  
def get\_E\_and\_mu(x, y, z, E\_pcb, E\_adhesive, E\_chip, mu\_pcb, mu\_adhesive, mu\_chip):  
 condition\_pcb = (pcb['x'][0] <= x) & (x <= pcb['x'][1]) & (pcb['y'][0] <= y) & (y <= pcb['y'][1]) & (pcb['z'][0] <= z) & (z <= pcb['z'][1])  
 condition\_adhesive = (adhesive['x'][0] <= x) & (x <= adhesive['x'][1]) & (adhesive['y'][0] <= y) & (y <= adhesive['y'][1]) & (adhesive['z'][0] <= z) & (z <= adhesive['z'][1])  
 condition\_chip = (chip['x'][0] <= x) & (x <= chip['x'][1]) & (chip['y'][0] <= y) & (y <= chip['y'][1]) & (chip['z'][0] <= z) & (z <= chip['z'][1])  
  
 E = torch.where(condition\_pcb, E\_pcb, torch.where(condition\_adhesive, E\_adhesive, torch.where(condition\_chip, E\_chip, 0)))  
 mu = torch.where(condition\_pcb, mu\_pcb, torch.where(condition\_adhesive, mu\_adhesive, torch.where(condition\_chip, mu\_chip, 0)))  
  
 return E, mu  
 # 如果不在任何定义的区域内，我们可以假设 E = 0, mu = 0  
  
  
  
class DisplacementModel(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super(DisplacementModel, self).\_\_init\_\_()  
 self.net = torch.nn.Sequential(  
 torch.nn.Linear(5, 50),  
 torch.nn.ReLU(),  
 torch.nn.Linear(50, 50),  
 torch.nn.ReLU(),  
 torch.nn.Linear(50, 3)  
 )  
  
 def forward(self, x):  
 return self.net(x)  
  
model\_u = DisplacementModel().to(device)  
  
# 定义位移函数  
# 定义位移函数  
def displacement(x, y, z, t, model\_u, model): # 注意这里添加了 model 作为参数  
  
 # 在这里调用 Tmodel 函数来获取 T\_value  
 T\_value, \_, \_, \_, \_ = Tmodel(x, y, z, t, model) # 假设 Tmodel 返回四个值，我们只取第一个  
  
 # 确保 T\_value 与其他变量具有相同的形状  
 T\_value = T\_value.squeeze(-1)  
  
 # 将 x, y, z, t, T\_value 组合成一个张量  
 inputs = torch.cat((x.unsqueeze(-1), y.unsqueeze(-1), z.unsqueeze(-1), t.unsqueeze(-1), T\_value.unsqueeze(-1)), dim=-1)  
  
 # 然后直接将这个张量传递给 model\_u  
 disp = model\_u(inputs)  
 u, v, w = disp[:, 0], disp[:, 1], disp[:, 2]  
 # 假设 u, v, w 已经被计算出来了  
 grad\_outputs = torch.ones\_like(u)  
  
 # 计算 u, v, w 关于 x, y, z 的梯度  
 u\_x = grad(u, x, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
 u\_y = grad(u, y, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
 u\_z = grad(u, z, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs = torch.ones\_like(v)  
 v\_x = grad(v, x, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
 v\_y = grad(v, y, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
 v\_z = grad(v, z, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs = torch.ones\_like(w)  
 w\_x = grad(w, x, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
 w\_y = grad(w, y, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
 w\_z = grad(w, z, grad\_outputs=grad\_outputs, create\_graph=True, allow\_unused=True)[0]  
  
 # 现在你有了 u, v, w 关于 x, y, z 的梯度  
  
 return u, u\_x, u\_y, u\_z, v, v\_x, v\_y, v\_z, w, w\_x, w\_y, w\_z  
  
  
  
  
  
def displaceloss(x, y, z, t, model\_u, mu, alpha, model):  
 ###定义了二阶导  
 u, u\_x, u\_y, u\_z, v, v\_x, v\_y, v\_z, w, w\_x, w\_y, w\_z = displacement(x, y, z, t, model\_u, model)  
 grad\_outputs\_u\_x = torch.ones\_like(u\_x)  
 u\_xx = grad(u\_x, x, grad\_outputs=grad\_outputs\_u\_x, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs\_u\_y = torch.ones\_like(u\_y)  
 u\_yy = grad(u\_y, y, grad\_outputs=grad\_outputs\_u\_y, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs\_u\_z = torch.ones\_like(u\_z)  
 u\_zz = grad(u\_z, z, grad\_outputs=grad\_outputs\_u\_z, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs\_v\_x = torch.ones\_like(v\_x)  
 v\_xx = grad(v\_x, x, grad\_outputs=grad\_outputs\_v\_x, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs\_v\_y = torch.ones\_like(v\_y)  
 v\_yy = grad(v\_y, y, grad\_outputs=grad\_outputs\_v\_y, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs\_v\_z = torch.ones\_like(v\_z)  
 v\_zz = grad(v\_z, z, grad\_outputs=grad\_outputs\_v\_z, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs\_w\_x = torch.ones\_like(w\_x)  
 w\_xx = grad(w\_x, x, grad\_outputs=grad\_outputs\_w\_x, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs\_w\_y = torch.ones\_like(w\_y)  
 w\_yy = grad(w\_y, y, grad\_outputs=grad\_outputs\_w\_y, create\_graph=True, allow\_unused=True)[0]  
  
 grad\_outputs\_w\_z = torch.ones\_like(w\_z)  
 w\_zz = grad(w\_z, z, grad\_outputs=grad\_outputs\_w\_z, create\_graph=True, allow\_unused=True)[0]  
  
 #前面已经定义过一次T的导数了 这里应该不用定义了  
 T, T\_x, T\_y, T\_z ,T\_t= Tmodel(x, y, z, t, model)  
  
 pde\_u = u\_xx + (1 - mu) / 2 \* (u\_yy + u\_zz) + (1 + mu) / 2 \* (v\_x \* v\_y + w\_x \* w\_z) + (1 + mu) \* alpha \* T\_x  
 pde\_v = v\_yy + (1 - mu) / 2 \* (v\_xx + v\_zz) + (1 + mu) / 2 \* (u\_x \* u\_y + w\_y \* w\_z) +(1 + mu) \* alpha \* T\_y  
 pde\_w = w\_zz + (1 - mu) / 2 \* (w\_xx + w\_yy) + (1 + mu) / 2 \* (u\_x \* u\_z + v\_y \* v\_z) + (1 + mu) \* alpha \* T\_z  
  
 return torch.mean(pde\_u \*\* 2) + torch.mean(pde\_v \*\* 2) + torch.mean(pde\_w \*\* 2)#所有的方程loss  
  
  
# 定义边界条件损失函数  
def boundary\_loss(x, y, z, t, model\_u):  
 u, \*\_ = displacement(x, y, z, torch.zeros\_like(t), model\_u,model)  
 return torch.mean(u \*\* 2)  
T\_ref = 293.15  
  
# 在损失函数中添加一个新的项  
def ref\_temp\_loss(x, y, z, t, model\_u, Tmodel):  
 u,\*\_ = displacement(x, y, z, torch.full\_like(t, T\_ref), model\_u,model)  
 T,\*\_ = Tmodel(x, y, z, torch.full\_like(t, T\_ref), model)  
 return torch.mean(u \*\* 2) + torch.mean((T - T\_ref) \*\* 2)  
  
def compute\_stress(x, y, z, t, T,model\_u):  
 u, u\_x, u\_y, u\_z, v, v\_x, v\_y, v\_z, w, w\_x, w\_y, w\_z = displacement(x, y, z, t, model\_u,model)  
 E, mu = get\_E\_and\_mu(x, y, z)  
 alpha = get\_alpha(x, y, z)  
  
 # 计算应力分量  
 common\_factor = E / (1 - mu \*\* 2)  
 thermal\_factor = E \* alpha \* T \* (1 - mu) / (1 - mu \*\* 2)  
  
 sigma\_xx = common\_factor \* (u\_x + mu \* v\_y + mu \* w\_z) - thermal\_factor  
 sigma\_yy = common\_factor \* (v\_y + mu \* u\_x + mu \* w\_z) - thermal\_factor  
 sigma\_zz = common\_factor \* (w\_z + mu \* u\_x + mu \* v\_y) - thermal\_factor  
  
 sigma\_xy = E / (2 \* (1 + mu)) \* (u\_y + v\_x)  
 sigma\_xz = E / (2 \* (1 + mu)) \* (u\_z + w\_x)  
 sigma\_yz = E / (2 \* (1 + mu)) \* (v\_z + w\_y)  
  
  
  
 return x, y, z, sigma\_xx, sigma\_yy, sigma\_zz, sigma\_xy, sigma\_xz, sigma\_yz  
  
  
# 加载实际的热应力数据  
sxx\_actual = torch.from\_numpy(data['sxx'].values.astype(np.float32)).to(device)  
  
# 加载数据  
data = pd.read\_csv('reshaped\_data.csv')  
print(data.columns)  
  
x = torch.from\_numpy(data['x'].values.astype(np.float32)).to(device)  
y = torch.from\_numpy(data['y'].values.astype(np.float32)).to(device)  
z = torch.from\_numpy(data['z'].values.astype(np.float32)).to(device)  
t = torch.from\_numpy(data['t'].values.astype(np.float32)).to(device)  
T\_actual = torch.from\_numpy(data['T'].values.astype(np.float32)).to(device) # 加载实际的温度数据  
  
  
batch\_size = 100  
  
# 计算批次数量  
num\_batches = len(x) // batch\_size  
# 假设 x, y, z, t, T\_actual 已经是 Tensor 类型并且已经加载到了设备上  
# 如果不是，您可以使用 torch.from\_numpy 来转换它们  
  
# 将数据合并到一个 TensorDataset 中  
dataset = TensorDataset(x, y, z, t, T\_actual)  
  
# 定义批次大小  
batch\_size = 64  
  
# 使用 random\_split 将数据集分为训练集和验证集  
train\_size = int(0.8 \* len(dataset)) # 计算训练集的大小  
val\_size = len(dataset) - train\_size # 计算验证集的大小  
train\_dataset, val\_dataset = random\_split(dataset, [train\_size, val\_size])  
  
# 创建 DataLoader  
train\_loader = DataLoader(dataset=train\_dataset, batch\_size=batch\_size, shuffle=True)  
val\_loader = DataLoader(dataset=val\_dataset, batch\_size=batch\_size, shuffle=False)  
  
# 计算批次数量  
num\_batches = len(x) // batch\_size  
# 创建一个新的优化器，包含热应力模型的参数  
optimizer = torch.optim.Adam(list(model.parameters()) + list(model\_u.parameters()))  
  
# 训练模型  
# 对每个批次进行处理  
num\_epochs = 10 # 设置你想要的epoch数量  
for epoch in range(num\_epochs):  
 model.train()  
 total\_loss = 0  
 # 训练阶段  
 for x\_batch, y\_batch, z\_batch, t\_batch, T\_actual\_batch in train\_loader:  
 x\_batch, y\_batch, z\_batch, t\_batch, T\_actual\_batch = (  
 x\_batch.to(device),  
 y\_batch.to(device),  
 z\_batch.to(device),  
 t\_batch.to(device),  
 T\_actual\_batch.to(device),  
 )  
  
 optimizer.zero\_grad()  
 E, mu = get\_E\_and\_mu(x\_batch, y\_batch, z\_batch, E\_pcb, E\_adhesive, E\_chip, mu\_pcb, mu\_adhesive, mu\_chip)  
 alpha = get\_alpha(x\_batch, y\_batch, z\_batch, alpha\_pcb, alpha\_adhesive, alpha\_chip)  
 rho, c = get\_rho\_and\_c(x\_batch, y\_batch, z\_batch, rho\_pcb, rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive, c\_chip)  
  
 loss\_value = losstem(x\_batch, y\_batch, z\_batch, t\_batch, T\_actual\_batch, Tmodel, k\_pcb, k\_adhesive, k\_chip, rho\_pcb,  
 rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive, c\_chip)  
 loss\_value += displaceloss(x\_batch, y\_batch, z\_batch, t\_batch, model\_u, mu, alpha, model)  
 loss\_value += boundary\_loss(x\_batch, y\_batch, z\_batch, t\_batch, model\_u)  
 loss\_value += ref\_temp\_loss(x\_batch, y\_batch, z\_batch, t\_batch, model\_u, Tmodel)  
  
 loss\_value.backward()  
 optimizer.step()  
 total\_loss += loss\_value.item()  
  
 average\_loss = total\_loss / len(train\_loader)  
 print(f"Epoch {epoch + 1}/{num\_epochs}, Average Training Loss: {average\_loss:.4f}")  
  
 # 验证阶段  
 model.eval()  
 val\_loss = 0  
 with torch.no\_grad():  
 for x\_batch, y\_batch, z\_batch, t\_batch, T\_actual\_batch in val\_loader:  
 x\_batch, y\_batch, z\_batch, t\_batch, T\_actual\_batch = (  
 x\_batch.to(device),  
 y\_batch.to(device),  
 z\_batch.to(device),  
 t\_batch.to(device),  
 T\_actual\_batch.to(device),  
 )  
 E, mu = get\_E\_and\_mu(x\_batch, y\_batch, z\_batch, E\_pcb, E\_adhesive, E\_chip, mu\_pcb, mu\_adhesive, mu\_chip)  
 alpha = get\_alpha(x\_batch, y\_batch, z\_batch, alpha\_pcb, alpha\_adhesive, alpha\_chip)  
 rho, c = get\_rho\_and\_c(x\_batch, y\_batch, z\_batch, rho\_pcb, rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive, c\_chip)  
  
 loss\_value = losstem(x\_batch, y\_batch, z\_batch, t\_batch, T\_actual\_batch, Tmodel, k\_pcb, k\_adhesive, k\_chip, rho\_pcb,  
 rho\_adhesive, rho\_chip, c\_pcb, c\_adhesive, c\_chip)  
 loss\_value += displaceloss(x\_batch, y\_batch, z\_batch, t\_batch, model\_u, mu, alpha, model)  
 loss\_value += boundary\_loss(x\_batch, y\_batch, z\_batch, t\_batch, model\_u)  
 loss\_value += ref\_temp\_loss(x\_batch, y\_batch, z\_batch, t\_batch, model\_u, Tmodel)  
  
 val\_loss += loss\_value.item()  
  
 val\_loss /= len(val\_loader)  
 print(f'Epoch [{epoch+1}/{num\_epochs}], Validation Loss: {val\_loss:.4f}')  
  
# 在训练结束后保存模型  
model\_save\_path = 'path/to/save/model.pth' # 指定模型保存的路径  
torch.save(model.state\_dict(), model\_save\_path)  
print(f'Model saved to {model\_save\_path}')  
  
# 将预测结果存储到DataFrame中  
results = pd.DataFrame({  
  
 'x': x.cpu().numpy(),  
 'y': y.cpu().numpy(),  
 'z': z.cpu().numpy(),  
 't': t.cpu().numpy(),  
 'T': T.cpu().detach().numpy().flatten(),  
 'u': u.cpu().detach().numpy().flatten(),  
 'v': v.cpu().detach().numpy().flatten(),  
 'w': w.cpu().detach().numpy().flatten(),  
 'sigma\_xx': sigma\_xx.cpu().detach().numpy().flatten(),  
 'sigma\_yy': sigma\_yy.cpu().detach().numpy().flatten(),  
 'sigma\_zz': sigma\_zz.cpu().detach().numpy().flatten(),  
 'sigma\_xy': sigma\_xy.cpu().detach().numpy().flatten(),  
 'sigma\_xz': sigma\_xz.cpu().detach().numpy().flatten(),  
 'sigma\_yz': sigma\_yz.cpu().detach().numpy().flatten()  
})  
  
# 将结果保存到CSV文件中  
results.to\_csv('results.csv', index=False)