

《机器学习基础理论及其在工程科学中的应用》

课程实践手册（二）

案例 5：动力学系统响应预测.....	1
案例 6：材料聚类.....	8
案例 7：材料本构建模.....	15
案例 8：基于 PINN 的 Burgers 方程求解.....	21

案例 5：动力学系统响应预测

一、问题描述

考虑如图 1 所示的三自由度质量-弹簧-阻尼振动系统。系统在质量 m_3 处受到一随时间变化的水平力 $F(t)$ 的作用，系统的初始位移和初始速度分别为 $\mathbf{x}_0 = \mathbf{0}$ 和 $\dot{\mathbf{x}}_0 = \mathbf{0}$ ，系统输出为质量 m_3 处的水平位移 $x_3(t)$ 。系统的质量、刚度和阻尼分别为 $m_1 = 1.2 \text{ kg}$ ， $m_2 = 1.0 \text{ kg}$ ， $m_3 = 1.6 \text{ kg}$ ； $k_1 = 350 \text{ Nm}^{-1}$ ， $k_2 = 300 \text{ Nm}^{-1}$ ， $k_3 = 320 \text{ Nm}^{-1}$ ； $c_1 = 80 \text{ Nsm}^{-1}$ ， $c_2 = 60 \text{ Nsm}^{-1}$ ， $c_3 = 64 \text{ Nsm}^{-1}$ 。受阻尼的影响，系统的输入和输出之间具有复杂的非线性关系，我们将利用循环神经网络（RNN）方法建立由输入载荷（即力-时间历程 $F(t)$ ）预测输出响应（即位移-时间历程 $x_3(t)$ ）的机器学习模型。

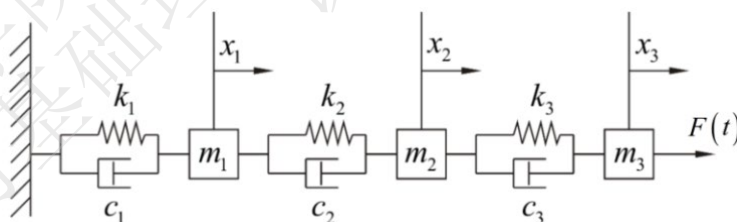


图 1 三自由度质量-弹簧-阻尼振动系统

二、学习方法

本例中，我们将使用 RNN 的一种变体——门控循环单元（GRU）神经网络模型来建立输入力-时间历程与输出位移-时间历程之间的非线性关系，并使用训练好的学习模型代替真实的结构动力学系统，在给定输入下预测系统的位移响应。

三、数据准备

我们借助 matlab 建立了振动系统的动力学模型，并使用高斯信号作为输入 $F(t)$ ，求解得到了 m_3 处的位移响应 $x_3(t)$ 。计算中，时间步长取 0.02 s ，计算时长为 $100,100$ 个时间步，输入（力-时间历程）数据文件 `input_x.csv` 和输出（位移-时间历程）数据文件 `output_y.csv` 均存储在目标路径下。图 2 截取了系统输入和输出的某个历程片段作为示意。整个历程数据被等分成 100 段，每段包含连续的 1001 个时间步，其中，前 80 段数据构成训练集，后 20 段数据构成测试集。

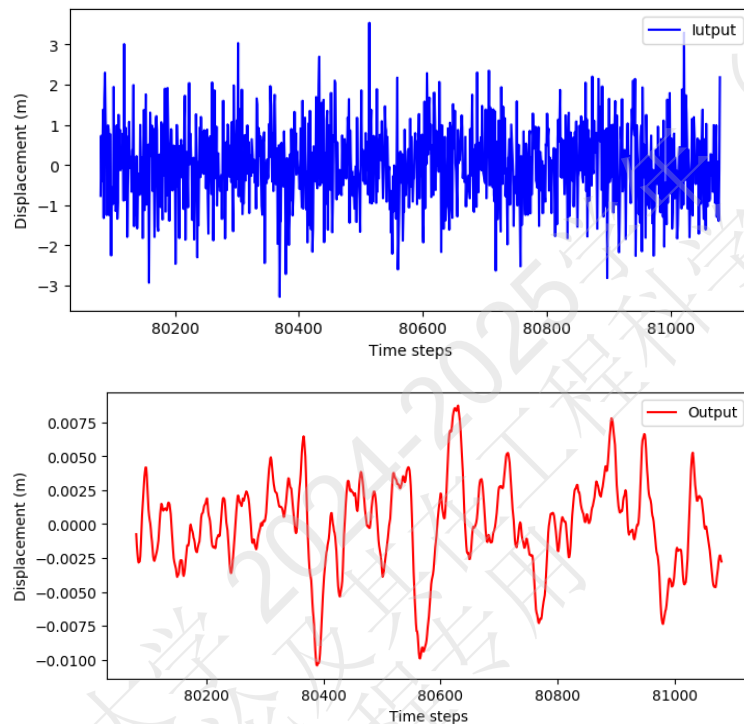


图 2 系统输入（力-时间历程）和对应的输出（位移-时间历程）片段示意图

四、编程过程

（1）导入相关函数库

```
[1] import time
[2] import pandas as pd
[3] import matplotlib.pyplot as plt
```

（2）读取输入和输出数据

```
[4] time_start=time.time() # 计时
[5] file_path = 'C:/Users/dell/Desktop/dynamic_system' # 目标路径
[6] input_data = pd.read_csv(file_path+'/input_x.csv',error_bad_lines=False) # 读取
训练数据
[7] output_data = pd.read_csv(file_path+'/output_y.csv',error_bad_lines=False) # 读
```

取训练数据

```
[8] time_end=time.time()
[9] print('totally cost',time_end-time_start) # 输出读取数据的用时
[10] x=input_data.values # 输入力 (100100×1)
[11] y=output_data.values # 输出位移 (100100×1)
```

(3) 输入和输出数据缩放处理

```
[12] from sklearn.preprocessing import MinMaxScaler
[13] scalerX=MinMaxScaler(feature_range=(-2,2)) # 输入缩放至-2 至 2
[14] XX = scalerX.fit_transform(x)
[15] scalerY=MinMaxScaler(feature_range=(0,1)) # 输出缩放至 0 至 1
[16] YY = scalerY.fit_transform(y)
```

(4) 训练集和测试集划分

```
[17] sample = 100 # 样本总数 (即将长时间历程数据分成的段数)
[18] nn = 80 # 训练集样本数
[19] mm = sample-nn # 测试集样本数
[20] deltlength=1001 # 每个样本的序列长度
[21] bs=10 # 批处理大小
[22] X = XX[:sample*deltlength] # 输入
[23] Y = YY[:sample*deltlength] # 输出
[24] n_train = deltlength*nn
[25] n_test = deltlength*sample
[26] # 训练数据
[27] trainX = X[0:n_train]
[28] trainY = Y[0:n_train]
[29] # 测试数据
[30] testX = X[n_train:n_test]
[31] testY = Y[n_train:n_test]
[32] # 输入和输出 3D 化
[33] train3DX = trainX.reshape((nn,deltlength,trainX.shape[1])) # 训练集输入
[34] test3DX = testX.reshape((mm,deltlength,testX.shape[1])) # 测试集输入
[35] train3DY = trainY.reshape((nn,deltlength,trainY.shape[1])) # 训练集输出
```

```
[36] test3DY = testY.reshape((mm,deltlength,testY.shape[1])) # 测试集输出
```

(5) 定义 GRU 神经网络模型

```
[37] from keras.models import Sequential
[38] from keras.layers import Dense
[39] from keras.layers.recurrent import GRU
[40] model = Sequential()
[41] # 第一层 GRU 网络设置
[42] model.add(GRU(units=20,input_shape=(train3DX.shape[1],train3DX.shape[2]),return_sequences=True))
[43] # 第二层 GRU 网络设置
[44] model.add(GRU(units=20,return_sequences=True))
[45] # 输出全链接层设置
[46] model.add(Dense(units=1,kernel_initializer='normal',activation='sigmoid'))
[47] model.summary()
```

【结果】运行本部分代码后将得到如图 3 所示的模型参数信息。

```
In [138]: model.summary()
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
gru_1 (GRU)	(None, 1001, 20)	1320
gru_2 (GRU)	(None, 1001, 20)	2460
dense_1 (Dense)	(None, 1001, 1)	21

```
Total params: 3,801
Trainable params: 3,801
Non-trainable params: 0
```

图 3 模型参数信息

(6) 编译模型

```
[48] from keras import optimizers
[49] adma=optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-7,
decay=0, amsgrad=False)
[50] # 自定义损失函数 (这里采用均方误差作为损失函数)
[51] import keras.backend as K
[52] def myloss(y_true, y_pred):
[53]     return K.mean(K.square(y_pred[:, :] - y_true[:, :]), axis=-1)
```

```
[54] model.compile(loss=myloss,optimizer='adam') # 选择误差评价准则、参数优化方法
```

(7) 模型训练

```
[55] lr_new=0.005 # 调整学习率
```

```
[56] K.set_value(model.optimizer.lr,lr_new)
```

```
[57] model.optimizer.get_config()
```

```
[58] # 设置 checkpoint
```

```
[59] from keras.callbacks import ModelCheckpoint
```

```
[60] filepath = file_path+'/model_n20n20_size1001_lr0.005_epoch100_best.h5' # 模
```

型存储路径及文件名称

```
[61] checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=1,  
save_best_only=True, mode='min') # 实时保存截止当前训练轮 (epoch) 时在测试集上预测误差  
最小的模型
```

```
[62] callbacks_list = [checkpoint]
```

```
[63] import time
```

```
[64] time_start=time.time() # 计时
```

```
[65] history =model.fit(train3DX, train3DY, epochs=100, batch_size=bs,  
validation_data=(test3DX, test3DY), verbose=2, shuffle=False,  
callbacks=callbacks_list)
```

```
[66] time_end=time.time()
```

```
[67] print('totally cost',time_end-time_start) # 输出训练用时
```

```
[68] # 记录训练集和测试集上的均方误差历程
```

```
[69] aa=history.history['loss']
```

```
[70] bb=history.history['val_loss']
```

(8) 绘制损失函数收敛历程图

```
[71] t=range(100)
```

```
[72] fig = plt.figure(dpi=100,figsize=(8,8))
```

```
[73] ax = fig.add_subplot(2,1,1)
```

```
[74] ax.plot(t,aa,c='blue',label='loss')
```

```
[75] ax.plot(t,bb,c='red',label='va_loss')
```

```
[76] ax.set_xlabel('Epoches')
```

```
[77] ax.set_ylabel('MSE')
```

```
[78] ax.set_yscale('log')
[79] plt.legend(loc=1)
[80] plt.show()
```

【结果】运行本部分代码后将得到如图 4 所示的模型在训练集和测试集上的均方误差随训练轮（epoch）的变化情况。

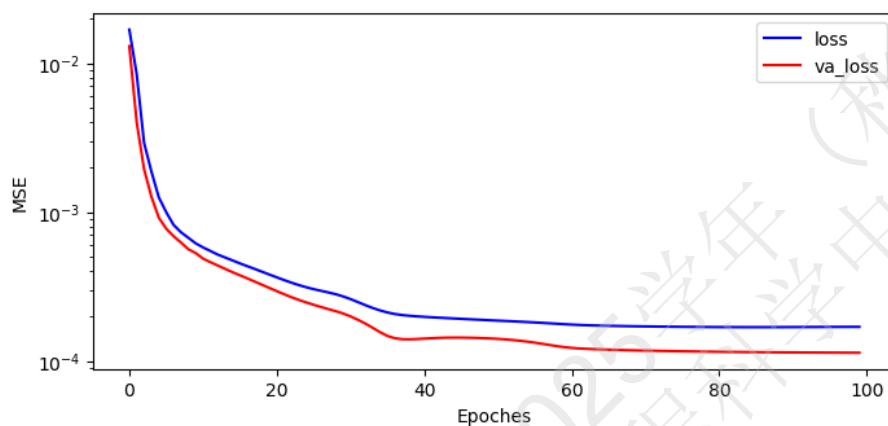


图 4 模型在训练集和测试集上的均方误差收敛曲线

(9) 利用学习模型预测测试集输出

```
[81] # 加载模型
[82] from keras.models import load_model
[83] model=load_model(file_path+'/model_n20n20_size1001_lr0.005_epoch100_best.h5',
,custom_objects={'myloss':myloss})
[84] # 将测试集中的输入数据作为输入，利用学得模型预测输出
[85] forecasttest3DY0 =model.predict(test3DX)
[86] forecasttest2DY0 = forecasttest3DY0.reshape((deltlength*mm,1))
[87] # 将输出结果反归一化，得到测试集输出的预测值
[88] YP =scalerY.inverse_transform(forecasttest2DY0)
```

(10) 绘制测试集上的输出预测值与实际值

```
[89] t=range(100100)
[90] index = 10 # 测试集上共含有 20 段位移历程数据（每段包含连续 1001 个时间步的数据），
index=10 表示提取其中的第 10 段进行绘图
[91] fig = plt.figure(dpi=100,figsize=(8,8))
[92] ax = fig.add_subplot(2,1,1)
[93] ax.plot(t[(nn+index-1)*1001:(nn+index)*1001],y[(nn+index-
```

```

1)*1001:(nn+index)*1001],c='blue',label='Actual') # 蓝色实线表示实际值
[94] ax.plot(t[(nn+index-1)*1001:(nn+index)*1001],YP[(index-
1)*1001:index*1001],c='red',label='Predicted') # 红色实线表示预测值
[95] ax.set_xlabel('Time steps')
[96] ax.set_ylabel('Displacement (m)')
[97] plt.legend(loc=1)
[98] plt.show()

```

【结果】运行本部分代码后将得到如图 5 所示的模型在测试集上预测的位移-时间历程及其与实际值的比较情况。

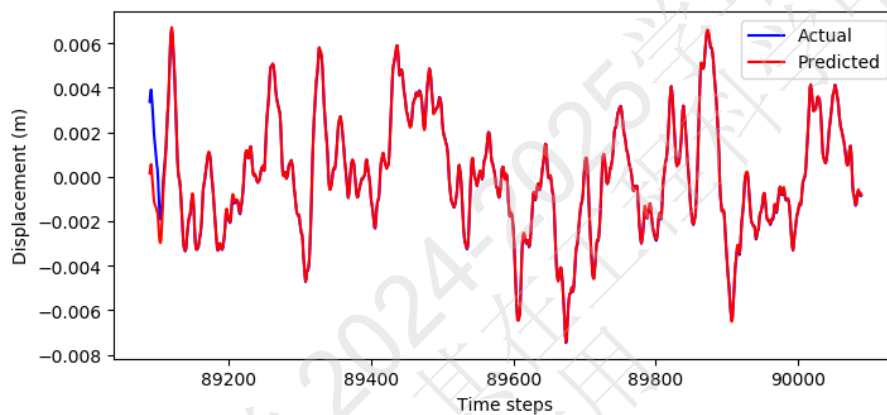


图 5 模型在测试集上的位移响应预测值与实际值的比较（选取第 10 段历程数据作为示例）

五、结果

本例的结果详见编程过程中各个步骤的结果。

六、拓展

（1）图 5 中初始阶段的预测值与实际值存在较大偏差，试调整本例程序第 90 行中的 index 取值，看看测试集中的其他段是否有类似情况？若有请思考如何改进算法来避免这些误差。

（2）尝试采用其他的时序模型，如经典的循环神经网络（RNN）模型、长短期记忆网络（LSTM）模型等，来学习本例中结构动力学系统的输入和输出关系，并比较不同方法的精度和效率。

案例 6：材料聚类

一、问题描述

聚类是针对给定的样本，依据它们特征的相似度或距离，将其归并到若干个“类”或“簇”的过程。相似度高或距离近的样本聚集在相同的类，相似度低或距离远的样本则分散在不同的类中。本例中，我们将根据弹性模量、断裂伸长率、极限强度等特征，利用 K -均值聚类方法对不同的材料进行聚类分析。

二、学习方法

本例中，我们将材料的基本属性作为特征，使用 K -均值聚类方法将不同合金材料归并到不同的类中，从而实现聚类分析。

三、数据准备

我们从网上收集了 108 种 1 系铝合金、2 系铝合金和钛合金材料的 3 种基本材料属性参数，包括弹性模量 E 、断裂伸长率 e 和极限强度 σ_b 。我们将这些数据整理到“材料聚类数据集.xlsx”文件中，并存储在目标路径下。图 6 给出了材料数据集的示意图，其中包括材料名称、弹性模量、断裂伸长率、极限强度、标签等数据信息。需要注意的是，最后一列标签只是作为判断聚类结果是否准确的依据，在聚类过程中不需要用到标签信息。

	A	B	C	D	E
1	材料名称	弹性模量 (GPa)	断裂伸长率 (%)	极限强度 (MPa)	标签
2	1050-H112Aluminum	70	20	83	0
3	1050-H12Aluminum	70	10	96	0
4	1050-H14Aluminum	70	8	110	0
5	1050-H16Aluminum	70	6	130	0
6	1050-H18Aluminum	70	5	140	0
7	1050-H22Aluminum	70	10	96	0
8	1050-H24Aluminum	70	7	110	0
9	1050-H26Aluminum	70	5	130	0
10	1050A-H12Aluminum	70	2	100	0
11	1050A-H14Aluminum	70	5	120	0
12	1050A-H16Aluminum	70	3	140	0

图 6 合金材料数据集示意图

四、编程过程

(1) 导入相关函数库

```
[1] import pandas as pd
[2] import numpy as np
[3] import matplotlib.pyplot as plt
[4] from sklearn.cluster import KMeans
```



```
[5] from sklearn.preprocessing import MinMaxScaler
```

(2) 数据读取和预处理

```
[6] file_path = 'C:/Users/dell/Desktop/clustering' # 目标路径
```

```
[7] sample_data = pd.read_excel(file_path+'/材料聚类数据集.xlsx',  
                                error_bad_lines=False)
```

```
[8] XX = sample_data.loc[:, '弹性模量 (GPa) ':'极限强度 (MPa) '] # 根据列标签选择输入数据，这里选取标签为弹性模量、断裂伸长率和极限强度的 3 列数据作为输入样本
```

```
[9] X = XX.values # 将 DataFrame 格式的数据转化为 float 64 格式
```

```
[10] YY = sample_data.loc[:, '标签']
```

```
[11] Y = YY.values # 将标签信息存储到列向量 Y 中，注意：该信息只用于对比作图，不用于聚类学习的过程
```

(3) 样本数据归一化

```
[12] scalerX = MinMaxScaler(feature_range=(0,1)) # 将样本数据缩放至 0~1
```

```
[13] X_scaler = scalerX.fit_transform(X)
```

(4) K -均值聚类模型构建和训练 ($K = 2$)

首先将材料归并为两类，即 $K = 2$ 。

```
[14] n = 2
```

```
[15] km = KMeans(algorithm='auto', init='k-means++', max_iter=300, n_clusters=n,  
                n_init=10) # 聚类模型设置：类别数  $n = 2$ ，最大迭代步数设为 300，从 10 个不同的初始值开始聚类分析，选取聚类效果最好的模型作为最终的聚类模型
```

```
[16] km.fit(X_scaler) # 模型训练
```

(5) 聚类结果获取

```
[17] # 获取簇心
```

```
[18] cc = km.cluster_centers_
```

```
[19] cc = scalerX.inverse_transform(cc) # 反归一化
```

```
[20] Y_pred = km.predict(X_scaler) # 获取聚类后样本所属簇的对应值
```

(6) 绘制聚类结果图

① 绘制原始分类图

```
[21] x0 = X[Y!=2] # 将 1 系和 2 系铝合金作为一类
```

```
[22] x1 = X[Y==2] # 钛合金
```

```
[23] fig = plt.figure(dpi=100, figsize=(8,8))
```

```

[24] ax = fig.gca(projection='3d')
[25] ax.set_xlim(0, 30) # 设置坐标轴取值范围
[26] ax.set_ylim(70, 120)
[27] ax.set_zlim(0, 1400)
[28] ax.scatter(x0[:,1],x0[:,0],x0[:,2],c='red',marker='o',label='Al')
[29] ax.scatter(x1[:,1],x1[:,0],x1[:,2],c='blue',marker='^',label='Ti')
[30] ax.set_xlabel('e (%)') # 设置坐标轴名称
[31] ax.set_ylabel('E (GPa)')
[32] ax.set_zlabel('sigma_b (MPa)')
[33] plt.legend(loc=2)
[34] plt.show()

```

【结果】运行本部分代码后将得到如图 7 所示的铝合金和钛合金两类材料在属性空间中分布的示意图。

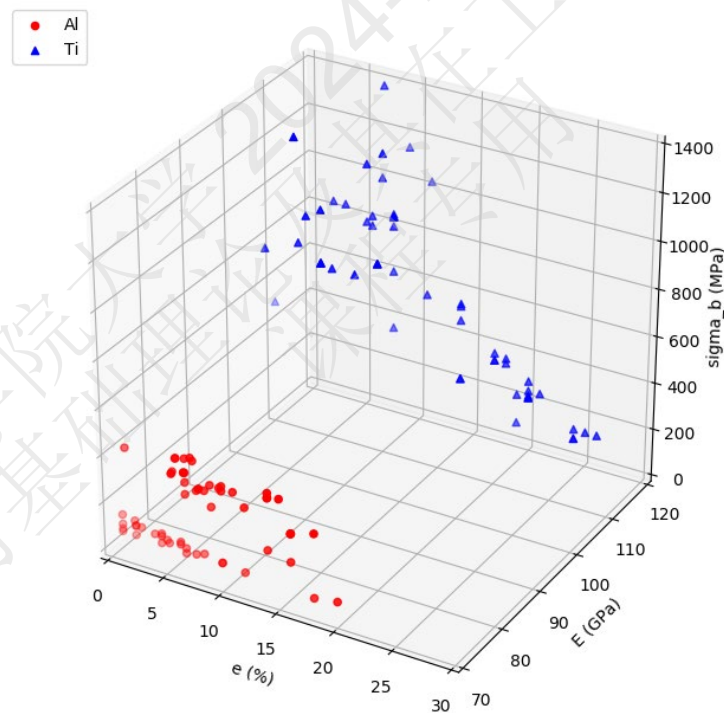


图 7 两类材料在属性空间中的分布：红色圆点为铝合金，蓝色三角形为钛合金

② 绘制聚类结果图

```

[35] x0 = X[Y_pred==0]
[36] x1 = X[Y_pred==1]
[37] fig = plt.figure(dpi=100,figsize=(8,8))
[38] ax = fig.gca(projection='3d')

```

```

[39] ax.set_xlim(0, 30)
[40] ax.set_ylim(70, 120)
[41] ax.set_zlim(0, 1400)
[42] ax.scatter(x0[:,1],x0[:,0],x0[:,2],c='red',marker='o',label='C-1')
[43] ax.scatter(x1[:,1],x1[:,0],x1[:,2],c='blue',marker='^',label='C-2')
[44] ax.scatter(cc[0,1],cc[0,0],cc[0,2],c='black',marker='o')
[45] ax.scatter(cc[1,1],cc[1,0],cc[1,2],c='black',marker='^')
[46] ax.set_xlabel('e (%)')
[47] ax.set_ylabel('E (GPa)')
[48] ax.set_zlabel('sigma_b (MPa)')
[49] plt.legend(loc=2)
[50] plt.show()

```

【结果】运行本部分代码后将得到如图 8 所示的聚类结果示意图，此时 $K = 2$ 。从图 8 可以看到，聚类算法将所有材料归并为了两类，聚类结果与实际材料类别一致。需要指出的是，聚类算法只是将材料根据给定的属性特征区分为了两类，并不知道哪一类是哪一种材料。

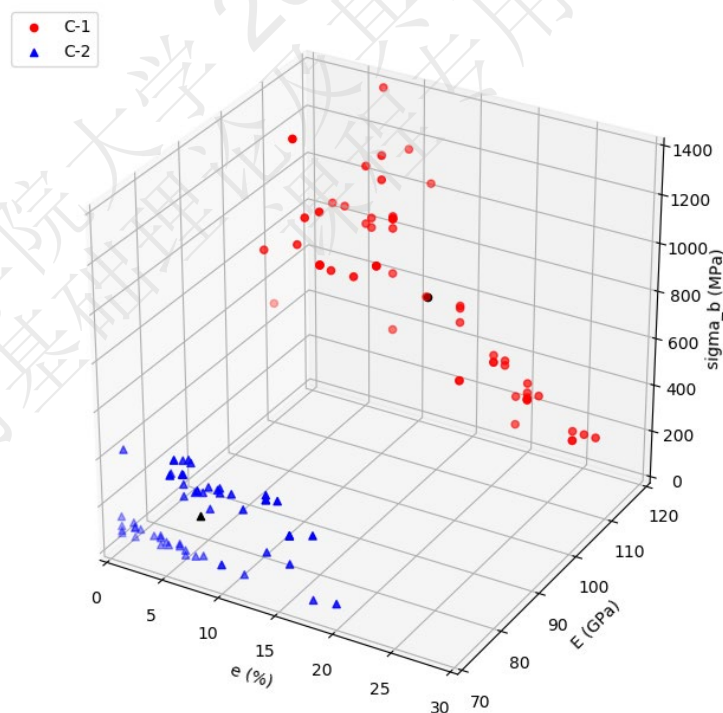


图 8 材料聚类结果示意图 ($K = 2$): 红色圆点和蓝色三角形各代表一类

如果我们将材料聚类成三类，又会得到怎样的结果呢？

当取 $K = 3$ 时，代码如下（前 13 行保持不变，从第 14 行开始调整）：

```

[14] n = len(np.unique(Y)) # 取  $K = 3$ 
[15] km = KMeans(algorithm='auto', init='k-means++', max_iter=300, n_clusters=n,
             n_init=10)
[16] km.fit(X_scaler)
[17] # 获取簇心
[18] cc = km.cluster_centers_
[19] cc = scalerX.inverse_transform(cc)
[20] # 获取聚类后样本所属簇的对应值
[21] Y_pred = km.predict(X_scaler)
[22] # 绘制原始分类图
[23] x0 = X[Y==0]
[24] x1 = X[Y==1]
[25] x2 = X[Y==2]
[26] fig = plt.figure(dpi=100, figsize=(8,8))
[27] ax = fig.gca(projection='3d')
[28] ax.set_xlim(0, 30)
[29] ax.set_ylim(70, 120)
[30] ax.set_zlim(0, 1400)
[31] ax.scatter(x0[:,1], x0[:,0], x0[:,2], c='red', marker='o', label='Al-1')
[32] ax.scatter(x1[:,1], x1[:,0], x1[:,2], c='green', marker='^', label='Al-2')
[33] ax.scatter(x2[:,1], x2[:,0], x2[:,2], c='blue', marker='s', label='Ti')
[34] ax.set_xlabel('e (%)')
[35] ax.set_ylabel('E (GPa)')
[36] ax.set_zlabel('sigma_b (MPa)')
[37] plt.legend(loc=2)
[38] plt.show()
[39] # 绘制聚类结果图
[40] x0 = X[Y_pred==0]
[41] x1 = X[Y_pred==1]
[42] x2 = X[Y_pred==2]
[43] fig = plt.figure(dpi=100, figsize=(8,8))

```

```

[44] ax = fig.gca(projection='3d')
[45] ax.set_xlim(0, 30)
[46] ax.set_ylim(70, 120)
[47] ax.set_zlim(0, 1400)
[48] ax.scatter(x0[:,1],x0[:,0],x0[:,2],c='red',marker='o',label='C-1')
[49] ax.scatter(x1[:,1],x1[:,0],x1[:,2],c='green',marker='^',label='C-2')
[50] ax.scatter(x2[:,1],x2[:,0],x2[:,2],c='blue',marker='s',label='C-3')
[51] ax.scatter(cc[0,1],cc[0,0],cc[0,2],c='black',marker='o')
[52] ax.scatter(cc[1,1],cc[1,0],cc[1,2],c='black',marker='^')
[53] ax.scatter(cc[2,1],cc[2,0],cc[2,2],c='black',marker='s')
[54] ax.set_xlabel('e (%)')
[55] ax.set_ylabel('E (GPa)')
[56] ax.set_zlabel('sigma_b (MPa)')
[57] plt.legend(loc=2)
[58] plt.show()

```

【结果】运行本部分代码后将得到如图 9 所示的原始材料分类示意图和图 10 所示的聚类结果示意图。从图 10 可以看到，当簇的数量选为 3 时，聚类结果与实际材料类别存在较大差异，表现为无法区分 1 系和 2 系铝合金，将钛合金错误分为两类。这是由于本例所使用的数据在特征空间中的分布规律和 K -均值聚类所要求的高斯分布存在较大差异造成的。

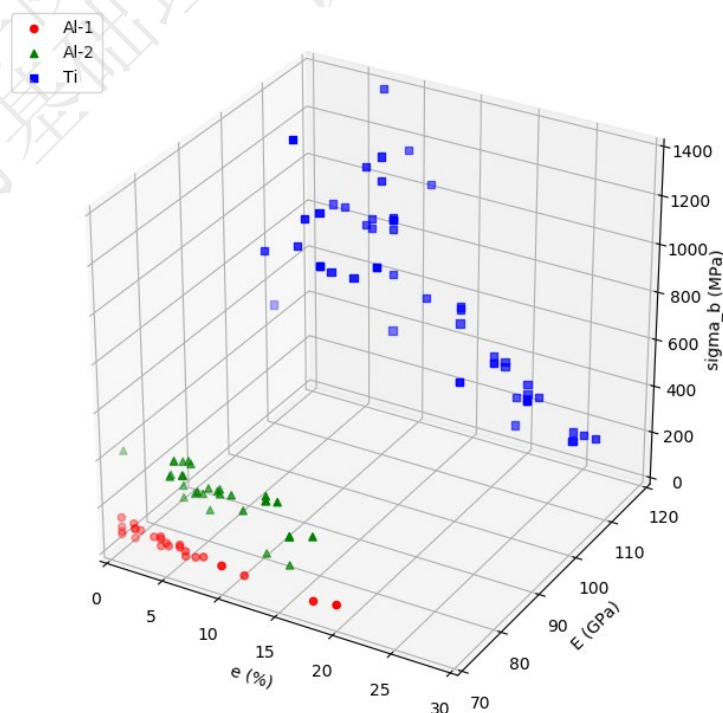


图 9 三类材料在属性空间中的分布：红色圆点为 1 系铝合金，绿色三角形为 2 系铝合金，蓝色方块为钛合金

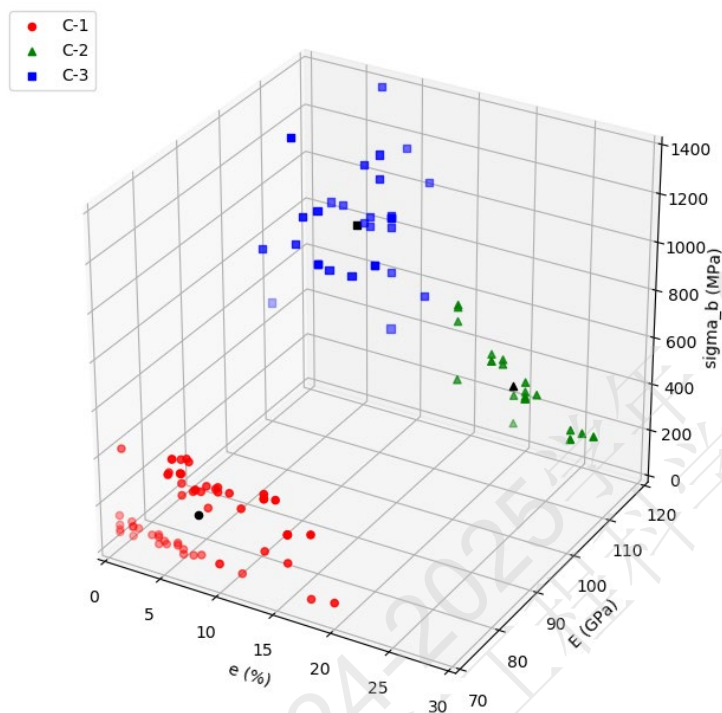


图 10 聚类结果示意图 ($K = 3$): 红色圆点、绿色三角形、蓝色方块各代表一类

五、结果

本例的结果详见编程过程中相关步骤的结果。

六、拓展

本例中, 当 $K = 3$ 时, K -均值聚类方法的分类效果明显不佳, 请尝试通过改变距离定义、引入核函数或使用其他聚类方法等途径实现本例中材料数据的正确聚类 (要求簇数量为 3)。

案例 7：材料本构建模

一、问题描述

锂金属电极由于高理论容量（3860 mAh/g）、低密度和低电势（约-3.04 V），是最理想的锂电池负极材料。准确地认识和表征锂金属负极温度、应力和率相关的变形行为是实现锂电池寿命和可靠性提升的关键。然而，由于涉及到温度场、力场、率效应等多物理场多因素之间的相互作用，以及非常有限的实验数据，目前仍缺乏可靠的物理模型来描述锂金属温度-应力-率-变形行为。我们将机器学习方法与物理机理相结合，利用有限的实验数据，采用梯度提升回归树（GBRT）模型建立锂金属的塑性应变率与应力、应变、温度、应变率之间的非线性关系，即 $\dot{\gamma} = f(\sigma_e, \bar{\epsilon}, T, \dot{\epsilon})$ 。本例中，我们重点关注锂金属变形行为的温度相关性，在保持应变率不变的情况下，尝试构建预测锂金属塑性应变率与应力、应变、温度关系的机器学习模型。图 11 给出了不同温度下锂金属的应力-应变曲线（ $\dot{\epsilon} = 3 \times 10^{-5}$ ），可以看到，温度对于锂金属的应力-应变行为影响显著。

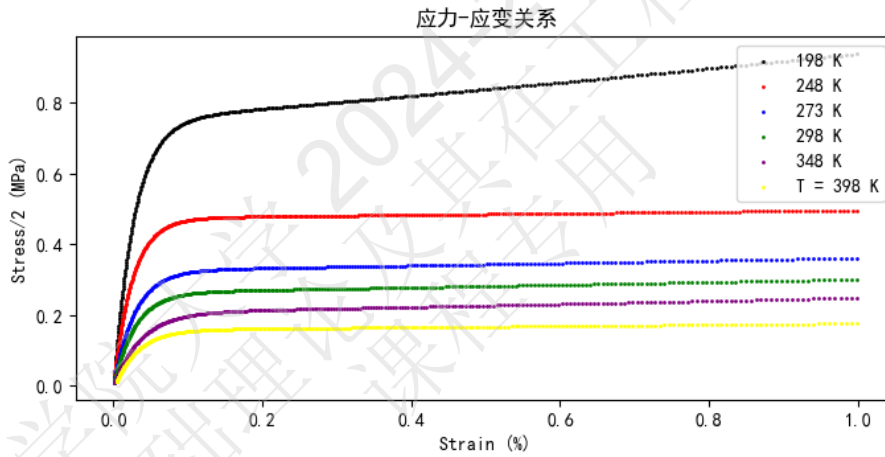


图 11 不同温度下锂金属材料的应力-应变曲线（ $\dot{\epsilon} = 3 \times 10^{-5}$ ）

二、学习方法

本例中，我们将使用梯度提升回归树（GBRT）方法来建立锂金属塑性应变率与应力、应变、温度之间的非线性关系，并利用训练好的模型预测不同温度下锂金属材料的变形行为。

三、数据准备

我们从文献（LePage, W.S. et al., *J. Electrochem. Soc.*, 2019）中提取了如图 11 所示的不同温度下锂金属材料的应力-应变实验数据（ $\dot{\epsilon} = 3 \times 10^{-5}$ ）。我们对实验数据进行分析，获得了应力、温度、总应变、总应变率、塑性应变率的样本数据，并将这些数据存储到目标路径下。

的“Data_temp.xlsx”文件中。图 12 给出了根据实验结果得到的反映锂金属本构关系的数据样本。需要注意的是，相较于原始数据，图 12 中的数据都进行了标准化处理，如应力数据的大小是实际值的 1/2，温度标准化为了 0~1 的参数，总应变为百分量，总应变率统一为 1，塑性应变率也做了相应的变换。

应力 (MPa)	温度(K)	总应变	总应变率(1/s)	塑性应变率(1/s)
0.010559893	1	0.003932497	1	2.88219E-09
0.011473542	1	0.004122747	1	0.006132493
0.012410321	1	0.004319036	1	0.012420095
0.013370166	1	0.004521459	1	0.018862362
0.01435299	1	0.00473011	1	0.0254587
0.015358684	1	0.004945082	1	0.032208363
0.016387116	1	0.005166472	1	0.039110459

图 12 描述锂金属本构关系的关键数据样本示意图

四、编程过程

(1) 导入相关函数库

```
[1] import pandas as pd
[2] from sklearn import ensemble
[3] from sklearn.metrics import r2_score, mean_squared_error
[4] import joblib
[5] import matplotlib.pyplot as plt
```

(2) 读取样本数据

```
[6] xcol=range(4)
[7] file_path = 'C:/Users/dell/Desktop/Material constitutive relation' # 目标路径
[8] dd=pd.read_excel(file_path+"/Data_temp.xlsx") # 样本数据，其中，第 1~4 列为输入，
依次为应力 (MPa)、温度 (K)、总应变 (%)、总应变率 (1/s)；第 5 列为输出，即塑性应变率
[9] x=dd.values[:,xcol] # 样本输入，第 1~4 列依次为应力 (MPa)、温度 (K)、总应变 (%)、总应变率 (1/s)
```

(3) 绘制不同温度下锂金属的应力-塑性应变率关系图（实验数据）

```
[10] fig = plt.figure(dpi=100,figsize=(8,8))
[11] ax = fig.add_subplot(2,1,1)
[12] ax.scatter(dd.values[1883:2248,0],dd.values[1883:2248,4], s=1, c='black',
marker='o',label='198 K')
[13] ax.scatter(dd.values[1418:1883,0],dd.values[1418:1883,4], s=1, c='red',
marker='o',label='248 K')
```



```

[14] ax.scatter(dd.values[1050:1418,0],dd.values[1050:1418,4], s=1, c='blue',
marker='o',label='273 K')

[15] ax.scatter(dd.values[689:1050,0],dd.values[689:1050,4], s=1, c='green',
marker='o',label='298 K')

[16] ax.scatter(dd.values[337:689,0],dd.values[337:689,4], s=1, c='purple',
marker='o',label='348 K')

[17] ax.scatter(dd.values[0:337,0],dd.values[0:337,4], s=1, c='yellow',
marker='o',label='T = 398 K')

[18] ax.set_xlim(0.02, 1)

[19] ax.set_ylim(0.2, 1)

[20] ax.set_yscale('log')

[21] ax.set_xlabel('Stress/2 (MPa)')

[22] ax.set_ylabel('dgamma/epsilon')

[23] plt.title('应力-塑性应变率关系')

[24] plt.legend(loc=4)

[25] plt.show()

```

【结果】运行本部分代码后将得到如图 13 所示的根据实验数据获得的锂金属应力-塑性应变率关系示意图。我们的目标就是利用机器学习方法来刻画这类温度相关的非线性关系。

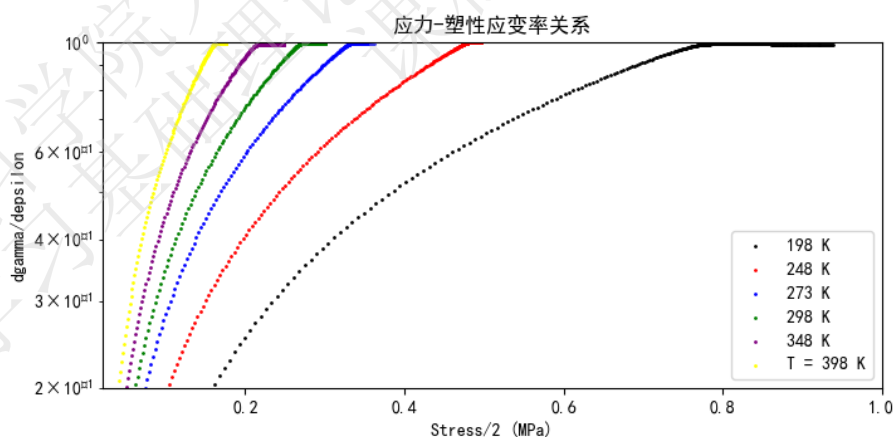


图 13 不同温度下根据实验数据获得的锂金属应力-塑性应变率关系 ($\dot{\epsilon} = 3 \times 10^{-5}$)

(4) 梯度提升回归树 (GBRT) 模型参数设置

```

[26] params = {'n_estimators': 3000, 'max_depth': 6, 'min_samples_split': 2,
[27]           'learning_rate': 0.01, 'loss': 'squared_error'}
[28] clf_1= ensemble.GradientBoostingRegressor(**params)

```

(4) 模型训练和预测

```
[29] x_train=dd.values[689:2248:1,xcol] # 训练集输入, 这里 198 K、248 K、273 K、298
K 四个温度下的数据被用作训练

[30] x_test=dd.values[:,xcol] # 训练集 + 测试集的输入, 这里 348 K 和 398 K 两个温度下的
数据被用作测试

[31] y_train=dd.values[689:2248:1,4] # 训练集输出

[32] y_test=dd.values[:,4] # 训练集 + 测试集的输出

[33] clf_1.fit(x_train, y_train) # 模型训练

[34] joblib.dump(clf_1, "clf1_full.pkl") # 模型保存

[35] mse = mean_squared_error(y_test, clf_1.predict(x_test)) # 计算模型在训练集和测
试集上的预测均方误差

[36] r2 = r2_score(y_test, clf_1.predict(x_test)) # 计算模型在训练集和测试集上的预测
值相较于实际值的决定系数 (越接近 1 表示一致性越好)

[37] print('塑性应变率预测均方误差: ',mse) # 显示模型预测结果的均方误差

[38] print('塑性应变率预测值与实际值的决定系数: ',r2) # 显示模型预测结果相较于实际值的
决定系数

[39] YP1=clf_1.predict(x_test) # 模型在训练集和测试集上的预测输出
```

【结果】运行本部分代码后将显示如图 14 所示的模型在训练集和预测集上的预测均方误差、预测值相较于实际值的决定系数。

```
塑性应变率预测均方误差: 0.0005939591586329733
塑性应变率预测值与实际值的决定系数: 0.9941850108135688
```

图 14 模型在训练集和预测集上的预测均方误差、预测值相较于实际值的决定系数

(5) 绘制不同温度下锂金属的应力-塑性应变率关系预测值与实际值的比较图

```
[40] fig = plt.figure(dpi=100,figsize=(8,8))

[41] ax = fig.add_subplot(2,1,1)

[42] ax.plot(dd.values[1883:2248,0],dd.values[1883:2248,4],
c='gray',label='Actual')

[43] ax.plot(dd.values[1418:1883,0],dd.values[1418:1883,4], c='gray')

[44] ax.plot(dd.values[1050:1418,0],dd.values[1050:1418,4], c='gray')

[45] ax.plot(dd.values[689:1050,0],dd.values[689:1050,4], c='gray')

[46] ax.plot(dd.values[337:689,0],dd.values[337:689,4], c='gray')

[47] ax.plot(dd.values[0:337,0],dd.values[0:337,4], c='gray')
```

```

[48] ax.scatter(dd.values[1883:2248,0],YP1[1883:2248], s=4, c='black',
marker='o',label='T = 198 K, Pred')

[49] ax.scatter(dd.values[1418:1883,0],YP1[1418:1883], s=4, c='red',
marker='o',label='248 K, Pred')

[50] ax.scatter(dd.values[1050:1418,0],YP1[1050:1418], s=4, c='blue',
marker='o',label='273 K, Pred')

[51] ax.scatter(dd.values[689:1050,0],YP1[689:1050], s=4, c='green',
marker='o',label='298 K, Pred')

[52] ax.scatter(dd.values[337:689,0],YP1[337:689], s=4, c='purple',
marker='o',label='348 K, Pred')

[53] ax.scatter(dd.values[0:337,0],YP1[0:337], s=4, c='yellow',
marker='o',label='T = 398 K, Pred')

[54] ax.set_xlim(0.02, 1)
[55] ax.set_ylim(0.2, 1)
[56] ax.set_yscale('log')
[57] ax.set_xlabel('Stress/2 (MPa)')
[58] ax.set_ylabel('dgamma/depsilon')
[59] plt.title('应力-塑性应变率关系')
[60] plt.legend(loc=4)
[61] plt.show()

```

【结果】运行本部分代码后将得到如图 15 所示的不同温度下锂金属的应力-塑性应变率关系预测值与实际值的比较。从图 15 可以看到，训练好的模型在训练集上具有很高的预测精度，能够准确刻画不同温度下锂金属的应力-塑性应变率关系。模型在测试集（T=348 K、398 K）上虽然能够捕捉到上述关系的变化趋势，但在预测精度上依然有进一步提升的空间。

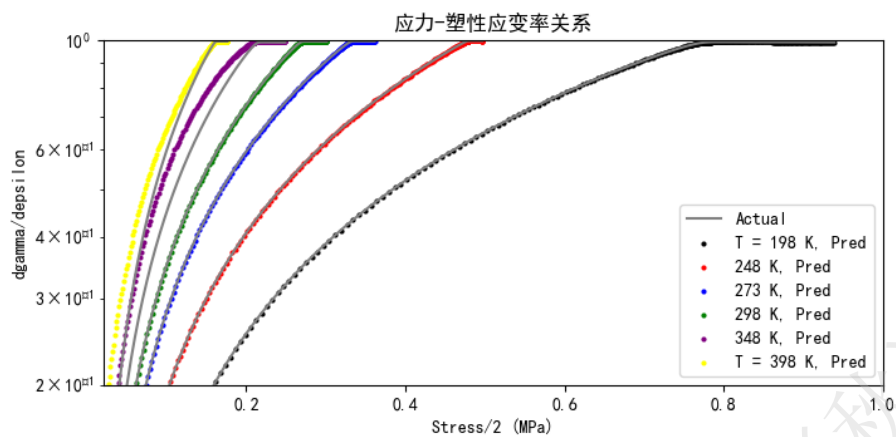


图 15 不同温度下锂金属的应力-塑性应变率关系预测值与实际值的比较 ($\dot{\epsilon} = 3 \times 10^{-5}$)

五、结果

本例的结果详见编程过程中各个步骤的结果。

六、拓展

(1) 如何改善梯度提升回归树 (GBRT) 模型在测试集上的预测精度?

(2) 尝试采用其他的学习模型, 如神经网络模型等, 来学习本例中锂金属的温度-应力-率-变形行为, 并比较不同方法在精度和效率上的优劣。

案例 8：基于 PINN 的 Burgers 方程求解

一、问题描述

在求解偏微分方程时，解析解在多数情况下难以获得，通常可以采用数值方法来求解。传统数值方法将连续偏微分方程离散为一组代数方程，通过迭代的方式进行求解。基于物理信息神经网络（PINN）的求解方法近年来受到广泛关注，为偏微分方程的求解提供了新的思路。PINN 利用神经网络逼近方程的解，通过优化损失函数，将满足初边界条件的神经网络解拟合到偏微分方程的解上。相较于传统的数值方法，PINN 具有更好的精度和效率，可以处理任意复杂的几何形状和边界条件，而且可以在不需要显式网格的情况下进行计算，有效提高解的准确性和模型泛化能力。

在本例中，我们将求解一个一维的对流-扩散方程 —— Burgers 方程，它是一种非线性偏微分方程，广泛应用于流体力学、交通流、湍流建模等多个领域。Burgers 方程的具体形式如下：

$$\begin{aligned}\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} &= v \frac{\partial^2 u}{\partial x^2}, x \in [-1, 1], t \in [0, 1] \\ u(x, 0) &= -\sin(\pi x) \\ u(-1, t) &= u(1, t) = 0\end{aligned}$$

其中， u 为流体的速度， x 为空间坐标， t 为时间， $u \frac{\partial u}{\partial x}$ 为非线性对流项， $v \frac{\partial^2 u}{\partial x^2}$ 为扩散项， v 为粘性系数。本例中，取 $v = \frac{0.01}{\pi}$ 。

二、学习方法

本例将利用 PINN 进行偏微分方程求解的学习。针对 Burgers 方程，以一维空间位置 x 和时间 t 作为输入，以 Burgers 方程解 $u(x, t)$ 作为输出。利用 PINN 模型，通过将初始条件、边界条件以及 Burgers 方程的物理信息嵌入到损失函数中，来训练模型并获得 Burgers 方程在指定区域的数值解。损失函数包括两部分，数据损失和 PDE 损失。其中数据损失为网络模型对边界条件和初始条件的输出值和实际值的均方误差：

$$\text{Loss}_{\text{data}} = \frac{1}{N} \sum_{i=1}^N \left(u_{\text{pred}}(x_i, t_i) - u_{\text{true}}(x_i, t_i) \right)^2$$

其中， N 为边界条件和初始条件的采样点数目。这部分损失相当于边界和初始条件的约束，使得网络学习到符合这些条件的解。PDE 损失是确保网络输出满足 Burgers 方程，其表达式为：

$$\text{Loss}_{\text{PDE}} = \frac{1}{M} \sum_{j=1}^M \left(\frac{\partial \hat{u}(x_j, t_j)}{\partial t} + \hat{u}(x_j, t_j) \frac{\partial \hat{u}(x_j, t_j)}{\partial x} - v \frac{\partial^2 \hat{u}(x_j, t_j)}{\partial x^2} \right)^2$$

其中, $\hat{u}(x_j, t_j)$ 为网络的预测值, M 为在 (x, t) 内的采样点数目。通过最小化总损失, 即 $\text{Loss}_{\text{total}} = \text{Loss}_{\text{PDE}} + \text{Loss}_{\text{data}}$, PINN 的网络参数不断调整, 最终使得网络输出 $u(x, t)$ 满足初始条件、边界条件和 Burgers 方程的约束, 接近真实解。

三、前期准备

可按如下方式安装 PyTorch 深度学习框架:

```
pip install torch==1.13.0 torchvision==0.14.0
```

四、编程过程

打开 python 开发工具 (IDE), 新建 PINN_Burgers.py 文件, 利用 PINN 求解 Burgers 方程的参考代码如下 ([·] 为行号标记):

(1) 导入相关库

```
[1] import math
[2] import torch
[3] import torch.nn as nn
[4] import numpy as np
[5] import matplotlib.pyplot as plt
[6] from collections import OrderedDict
```

(2) 多层神经网络类定义

神经网络包括输入层 (有两个输入, 分别为坐标 x , 时间 t), 4 层隐藏层 (每个隐藏层含 20 个神经元), 以及输出层 (只有一个输出, 方程的解 u)。

```
[7] class NN(nn.Module):
[8]     def __init__(self, input_size, hidden_size, output_size, depth,
act=torch.nn.Tanh):
[9]         super(NN, self).__init__()
[10]         layers = [('input', torch.nn.Linear(input_size, hidden_size))] # 输入层
[11]         layers.append(('input_activation', act())) # 输入激活函数
[12]         for i in range(depth): # 隐藏层
[13]             layers.append(('hidden_%d' % i, torch.nn.Linear(hidden_size,
```

```

hidden_size)))

[14]         layers.append(('activation_%d' % i, act())) # 隐藏层激活函数层
[15]         layers.append(('output', torch.nn.Linear(hidden_size, output_size)))
# 输出层

[99]         layerDict = OrderedDict(layers) # 将层存储在有序字典中，以便顺序执行
[16]         self.layers = torch.nn.Sequential(layerDict)
[17]         def forward(self, x): # 前向传播函数
[18]             out = self.layers(x)
[19]             return out

```

(3) 定义神经网络训练类，用于 PINN 训练过程

```

[20] class Net:
[21]     def __init__(self):
[22]         device = torch.device("cuda") if torch.cuda.is_available() else
torch.device("cpu") # 确定设备 (CPU 或 GPU)
[23]         self.model = NN(input_size=2, hidden_size=20, output_size=1,
depth=4, act=torch.nn.Tanh).to(device) # 定义神经网络模型参数

```

(4) 生成训练数据

包括生成初始条件和边界条件上的采样点并给定标签值；生成 (x, t) 内的采样点（注意：这部分采样点无需标签值）。

```

[24]         self.h = 0.1 # 定义空间步长
[25]         self.k = 0.1 # 定义时间步长
[26]         x = torch.arange(-1 + self.h, 1, self.h) # 空间坐标离散
[27]         t = torch.arange(0 + self.k, 1 + self.k, self.k) # 时间坐标离散
[28]         self.X = torch.stack(torch.meshgrid(x, t)).reshape(2, -1).T # 计算域
内的采样点
[29]         bc1 = torch.stack(torch.meshgrid(torch.tensor([-1]), dtype=t.dtype),
t)).reshape(2, -1).T # 边界条件 1 的采样点
[30]         bc2 = torch.stack(torch.meshgrid(torch.tensor([1]), dtype=t.dtype),
t)).reshape(2, -1).T # 边界条件 2 的采样点
[31]         x0 = torch.cat([torch.tensor([-1]), x, torch.tensor([1])])
[32]         ic = torch.stack(torch.meshgrid(x0, torch.tensor([0]),

```

```

dtype=x.dtype))).reshape(2, -1).T    # 初始条件的采样点
[33]         self.X_train = torch.cat([bc1, bc2, ic]) # 合并初边界条件采样点
[34]         y_bc1 = torch.zeros(len(bc1)) # 边界条件 1 处的值, 即 x=-1 时, y=0
[35]         y_bc2 = torch.zeros(len(bc2)) # 边界条件 2 处的值, 即 x=1 时, y=0
[36]         y_ic = -torch.sin(math.pi * ic[:, 0]) # 初始条件的值
[37]         self.y_train = torch.cat([y_bc1, y_bc2, y_ic]).unsqueeze(1) # 合并初
边值条件处采样点的标签值, 并增加维度
[38]         self.X = self.X.to(device) # 将数据移到设备 (CPU 或 GPU)
[39]         self.X_train = self.X_train.to(device)
[40]         self.y_train = self.y_train.to(device)
[41]         self.X.requires_grad = True # 设置为计算梯度

```

(5) 定义损失函数和优化器

```

[42]         self.criterion = torch.nn.MSELoss() # 使用均方误差损失
[43]         self.iter = 1
[44]         self.optimizer = torch.optim.LBFGS( # 定义 LBFGS 优化器 (收敛速度快, 适
用于小批量数据), 参数有学习率 lr、最大迭代步数 max_iter 等
[45]             self.model.parameters(),
[46]             lr=1.0,
[47]             max_iter=50000,
[48]             max_eval=50000,
[49]             history_size=50,
[50]             tolerance_grad=1e-7,
[51]             tolerance_change=1.0 * np.finfo(float).eps,
[52]             line_search_fn="strong_wolfe",
[53]         )
[54]         self.adam = torch.optim.Adam(self.model.parameters()) # 定义 Adam 优
化器 (适用于非线性优化问题)
[55]         def loss_func(self): # 定义损失函数
[56]             self.adam.zero_grad() # 重置 Adam 优化器梯度
[57]             self.optimizer.zero_grad() # 重置 LBFGS 优化器梯度
[58]             y_pred = self.model(self.X_train)

```



```

[59]         loss_data = self.criterion(y_pred, self.y_train) # 计算数据损失：出边
界条件采样点处网络预测值与标签值的均方误差

[60]         u = self.model(self.X)
[61]         du_dX = torch.autograd.grad( # 计算 u 对 X 的偏导
[62]             inputs=self.X,
[63]             outputs=u,
[64]             grad_outputs=torch.ones_like(u),
[65]             retain_graph=True,
[66]             create_graph=True
[67]         )[0]
[68]         du_dt = du_dX[:, 1] # 计算 u 对 t 的偏导
[69]         du_dx = du_dX[:, 0] # 计算 u 对 x 的偏导
[70]         du_dxx = torch.autograd.grad( # 计算 u 对 x 的二阶偏导
[71]             inputs=self.X,
[72]             outputs=du_dX,
[73]             grad_outputs=torch.ones_like(du_dX),
[74]             retain_graph=True,
[75]             create_graph=True
[76]         )[0][:, 0]
[77]         loss_pde = self.criterion(du_dt + u.squeeze() * du_dx, 0.01 /
math.pi * du_dxx) # 计算 PDE 方程损失（基于物理方程）
[78]         loss = loss_pde + loss_data # 计算总损失
[79]         loss.backward() # 总损失反向传播
[80]         if self.iter % 100 == 0:
[81]             print(self.iter, loss.item()) # 输出迭代次数和总损失值
[82]             self.iter += 1
[83]             return loss
[84]     def train(self): # 训练模型
[85]         self.model.train()
[86]         for i in range(1000): # 初始训练阶段使用 Adam 优化器
[87]             self.adam.step(self.loss_func)

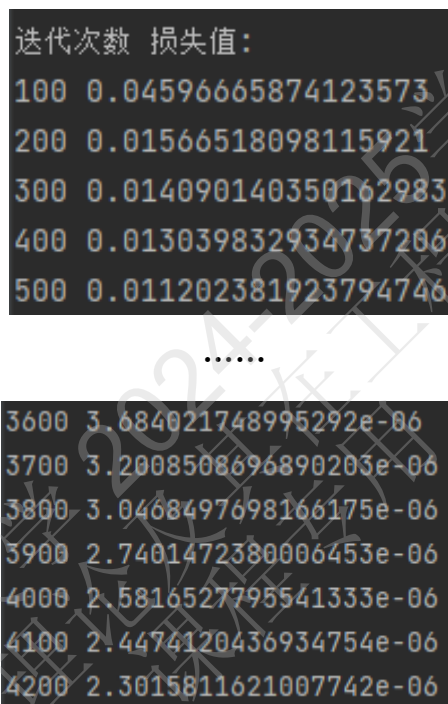
```

```

[88]         self.optimizer.step(self.loss_func) # 使用 LBFGS 优化器进行进一步优化
[89]     def eval_(self):
[90]         self.model.eval()
[91]     net = Net() # 初始化并训练模型
[92]     net.train()

```

执行本段代码将完成神经网络模型的训练，每训练 100 次后将显示迭代次数和当前损失函数的值，随着迭代次数增加，损失函数值在不断减小，说明网络输出值逐渐满足控制方程以及边界条件和初始条件的约束。



```

迭代次数 损失值:
100 0.04596665874123573
200 0.01566518098115921
300 0.014090140350162983
400 0.013039832934737206
500 0.011202381923794746
.....
3600 3.684021748995292e-06
3700 3.2008508696890203e-06
3800 3.0468497698166175e-06
3900 2.7401472380006453e-06
4000 2.5816527795541333e-06
4100 2.4474120436934754e-06
4200 2.3015811621007742e-06

```

图 16 训练结果

(6) 模型预测（偏微分方程求解）

```

[93]     h = 0.01
[94]     k = 0.01
[95]     xx = torch.arange(-1, 1 + h, h)
[96]     tt = torch.arange(0, 1 + k, k)
[97]     X = torch.stack(torch.meshgrid(xx, tt)).reshape(2, -1).T # 在时空域内进行采样，用于评估模型预测能力
[98]     X = X.to(net.X.device)
[99]     model = net.model
[100]    model.eval()

```

```
[101] with torch.no_grad():
[102]     u_pred = model(X).reshape(len(xx), len(tt)).cpu().numpy() # 获取预测结果
并转换为 numpy 数组
```

(7) 绘制解的云图

```
[103] plt.figure(figsize=(12, 6))
[104] plt.contourf(X[:, 1].reshape(len(xx), len(tt)), X[:, 0].reshape(len(xx),
    len(tt)), u_pred, levels=200, cmap='jet')
[105] plt.colorbar()
[106] plt.title('u(x,t)')
[107] plt.xlabel('t (s)')
[108] plt.ylabel('x (m)')
[109] plt.show()
```

执行本段代码将输出如下偏微分方程解 $u(x, t)$ 的云图。

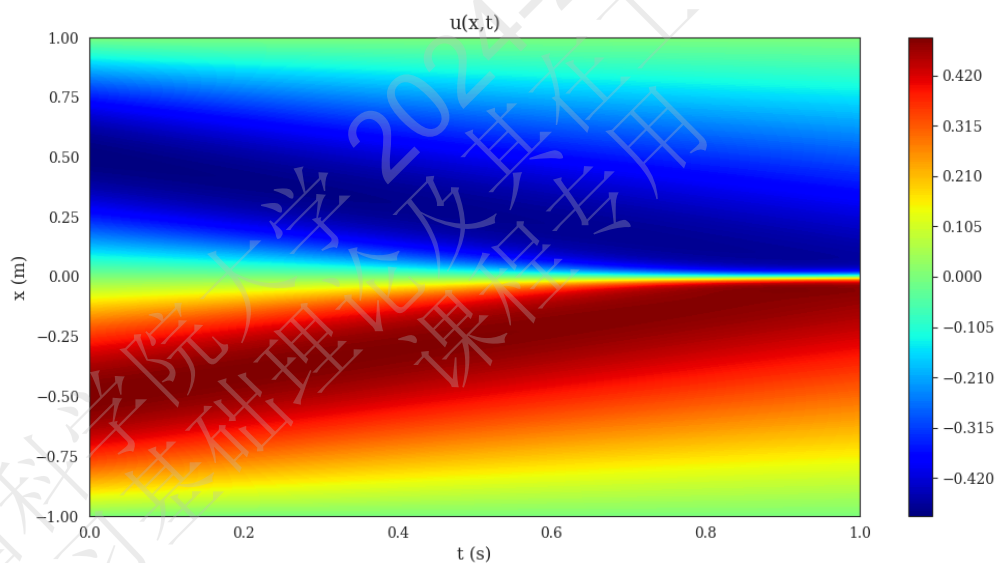


图 17 求解结果

五、结果

本例的结果详见编程过程中各个步骤的结果。

六、拓展

- (1) 尝试求解具有不同扩散系数、边界条件与初始条件的 Burgers 方程；
- (2) 尝试将此问题推广至二维或更高维的情况；
- (3) 使用该方法，解决其他偏微分方程求解问题，尤其是高维或者复杂边界的情形；
- (4) 基于 PINN 求解反问题。