1. 加载和预览数据

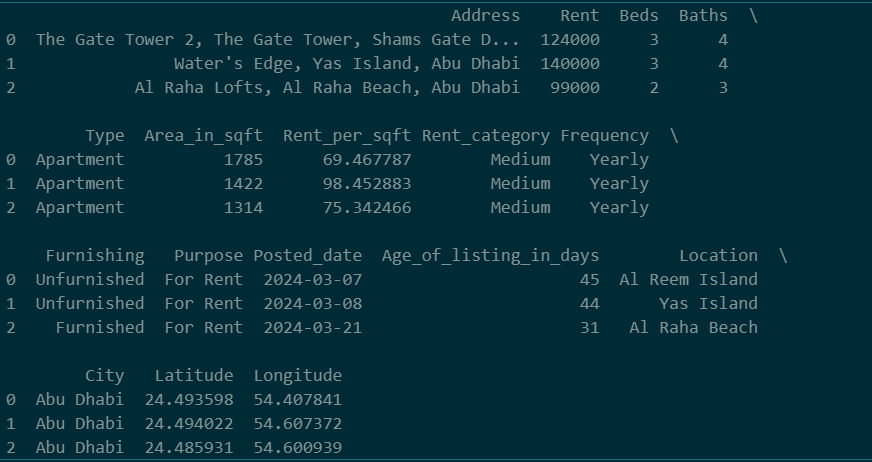
data=pd.read\_csv("dubai\_properties.csv")

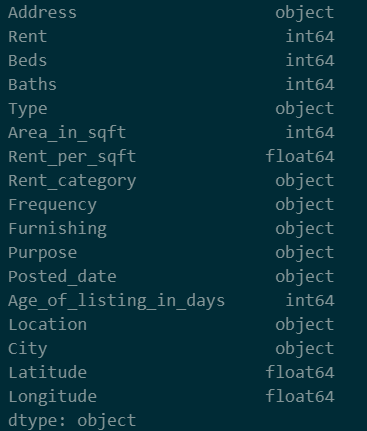
data.dropna(inplace=True)

print(data.head(3))

*# data*

简单查看数据内容head()



查看数据类型data.dtypes, 准备筛选非数值化

1. 特征工程-数据预处理
2. 数据清洗

data.dropna(inplace=True)

dropna()方法将不会返回一个新的DataFrame，而是直接修改原始的DataFrame，删除掉所有含有至少一个NaN值的行。

1. 特征编码

标签编码：对于每个类别 c*c* 在特征 F*F* 中，标签编码分配一个唯一的整数 e(c)*e*(*c*)

*# 创建一个字典来存储每个特征的编码器*

encoders = {}

*# 遍历每列*

for column in data.columns:

*# 如果是对象类型（通常是字符串或混合类型）*

    if data[column].dtype == object:

*# 创建一个 LabelEncoder*

        encoder = LabelEncoder()

*# 对当前列进行编码*

        data[column] = encoder.fit\_transform(data[column])

*# 保存编码器以便后续使用*

        encoders[column] = encoder

    elif data[column].dtype == 'int64' or data[column].dtype == 'float64':

*# 数值类型，不需要编码*

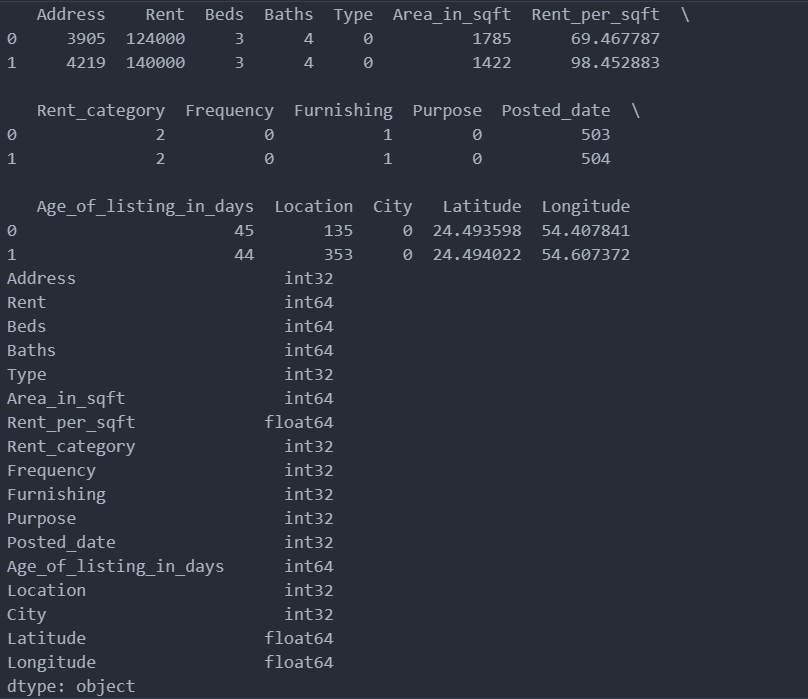
        pass

    else:

        print(**f**"Warning: Unhandled data type for column '{column}'")

1. **数学公式**：
   * 对于每个类别 c*c* 在特征 F*F* 中，标签编码分配一个唯一的整数 e(c)*e*(*c*)。
   * 这个整数是通过 LabelEncoder 的 fit\_transform 方法自动计算得出的，它基于 fit 阶段看到的类别集合。
2. **特征**：
   * **唯一性**：每个类别映射到一个唯一的整数。
   * **顺序无关**：标签编码假设类别之间没有自然的顺序，尽管分配的整数可能会暗示一个顺序。因此，在使用标签编码时，应该小心，确保模型不会错误地解释这些整数作为有序数据。
   * **可逆性**：可以通过相同的 LabelEncoder 实例使用 inverse\_transform 方法将整数转换回原始的类别标签， 便于验证模型时使用。

再次查看编码



Object(String 等类型)类型特征被编码为数值特征

1. 解释：**算法兼容性**：许多机器学习算法（如回归、神经网络、梯度提升机等）都要求输入特征是数值类型的。因此，需要对类别特征进行编码，以便算法能够处理这些特征。
2. **保持信息**：类别特征中包含的信息不能简单地通过数字顺序（如1、2、3）来表示，因为这会给特征赋予一个不存在的数值意义（例如，一个类别为2的值并不意味着它比类别为1的值大）。标签编码通过为每个类别分配一个唯一的整数来保持类别信息，而不引入任何数值上的比较。
3. **模型性能**：正确地编码类别特征可以显著提高模型的性能。如果类别特征没有被适当编码，模型可能无法学习到数据中的模式，从而导致次优的性能。
4. 特征缩放

使用sklearn中的归一化器进行归一化处理

from sklearn.preprocessing import StandardScaler

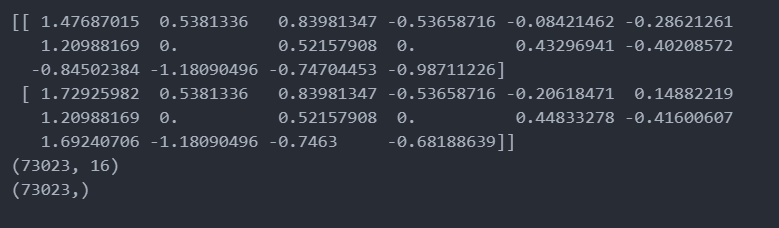
*# 使用标准归一化器*

standard\_scaler = StandardScaler()

X\_scaled = standard\_scaler.fit\_transform(X)

print(X\_scaled[:2])

print(X\_scaled.shape)

print(y.shape) 

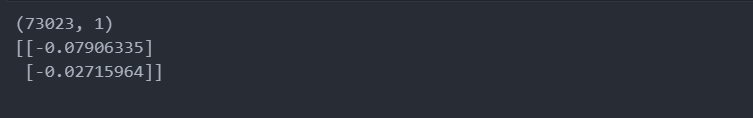
*# 归一化正确，将y也归一化处理*

y\_from\_to\_frame = y.to\_frame()

print(y\_from\_to\_frame.shape)

y\_scaled = standard\_scaler.fit\_transform(y\_from\_to\_frame)

print(y\_scaled[:2])



解释：

1. **避免梯度消失或爆炸**：在深度学习中，如果输入特征的范围差异很大，可能会导致梯度在反向传播过程中消失或爆炸，影响模型的训练效率和效果。
2. **加速学习过程**：特征标准化可以加快梯度下降的收敛速度，因为梯度下降在特征具有相似的范围时更有效。
3. **提高模型泛化能力**：标准化可以减少某些特征对模型的影响过大，从而提高模型的泛化能力。

不进归一化处理，可能导致梯度爆炸等问题，直观的训练错误如每轮batch损失值递增，超出float64精度范围溢出等问题。

使用标准化归一化的好处:

* 不受异常值的影响：由于它考虑了特征的方差，所以异常值对标准化过程的影响较小。
* 保留了特征之间的相关性：标准化不会改变特征之间的比例关系，这对于保持特征之间的相关性很重要。
* 适用于大多数回归模型：许多回归模型（如线性回归、岭回归、Lasso回归等）都假设特征是标准化的，或者对标准化数据表现更好。

import torch

import torch.nn as nn

import torch.optim as optim

from sklearn.model\_selection import train\_test\_split

*# 划分数据集*

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y\_scaled, test\_size=0.1, random\_state=42) *# 批量归一化*

X\_train\_tensor = torch.FloatTensor(X\_train)

y\_train\_tensor = torch.FloatTensor(y\_train)

X\_test\_tensor = torch.FloatTensor(X\_test)

y\_test\_tensor = torch.FloatTensor(y\_test)

*# X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.1, random\_state=42) # 非批量归一化*

*# X\_train\_tensor = torch.FloatTensor(X\_train.values)*

*# y\_train\_tensor = torch.FloatTensor(y\_train.values)*

*# X\_test\_tensor = torch.FloatTensor(X\_test.values)*

*# y\_test\_tensor = torch.FloatTensor(y\_test.values)*

print(X\_train\_tensor[:3].numpy())

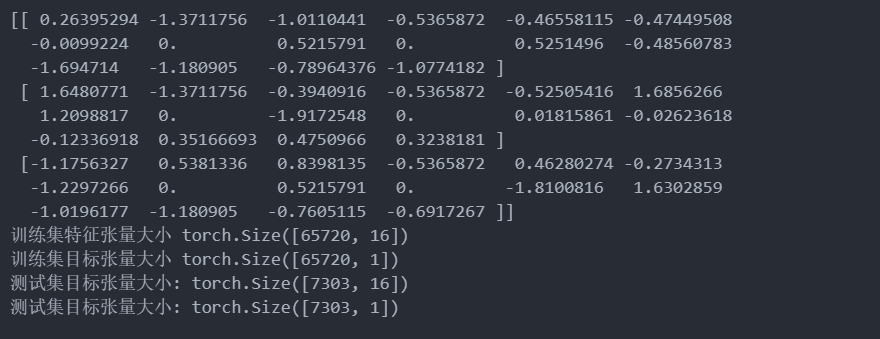
print("训练集特征张量大小", X\_train\_tensor.size())

print("训练集目标张量大小", y\_train\_tensor.size())

print("测试集目标张量大小:", X\_test\_tensor.size())

print("测试集目标张量大小:", y\_test\_tensor.size())

代码运行时采用了批量归一化的结果， 使用from sklearn.model\_selection import train\_test\_split 进行数据集划分，设置random\_state=42进行数据混洗，设置test\_size=0.1 将测试集与输入数据集比例设置为0.1，训练集自然得到0.9的比例， 同时将得到的数据集转换为神经网络(torch.nn)能进行计算的张量(tensor)，打印查看数据预处理后的数据集。



训练集样本量在65720，选取特征维度为16， 测试集样本量为7303

1. 模型训练

1. 构建神经网络

# 构建神经网络

class RentPredictor(nn.Module):

def \_\_init\_\_(self, input\_dim):

super(RentPredictor, self).\_\_init\_\_()

self.layer1 = nn.Linear(input\_dim, 64)

self.layer2 = nn.Linear(64, 128)

self.layer3 = nn.Linear(128, 256)

self.layer4 = nn.Linear(256, 64)

self.layer5 = nn.Linear(64, 1)

self.swish = nn.SiLU()

def forward(self, x):

x = self.swish(self.layer1(x))

x = self.swish(self.layer2(x))

x = self.swish(self.layer3(x))

x = self.swish(self.layer4(x))

x = self.layer5(x)

return x

继承torch.nn.Module定义Rent\_Prediction神经网络模型，输入的张量大小为[batch, input\_dim],

1. **初始化**：
   * \_\_init\_\_(self, input\_dim)：构造函数接收一个参数input\_dim，表示输入特征的数量。
   * super(RentPredictor, self).\_\_init\_\_()：调用基类的构造函数。
   * self.layer1到self.layer5：定义了五个全连接层（nn.Linear），分别是输入层到第一个隐藏层、第一个隐藏层到第二个隐藏层、第二个隐藏层到第三个隐藏层、第三个隐藏层到第四个隐藏层、第四个隐藏层到输出层的连接。每个全连接层后面都跟着一个Swish激活函数（nn.SiLU）。
2. **前向传播**：
   * forward(self, x)：定义了数据在模型中的前向传播路径。
   * x = self.swish(self.layer1(x))：输入数据x首先通过第一个全连接层，然后通过Swish激活函数。
   * 接下来的几行代码重复了这个过程，数据通过一系列的全连接层和Swish激活函数。
   * x = self.layer5(x)：最后，数据通过第五个全连接层，这个层没有激活函数，因为它是一个回归任务，需要直接输出一个数值（租金预测）。

Swish激活函数是一种平滑的非线性函数，定义为f(x) = x \* sigmoid(x)，它在前向传播中提供了非线性变换，使得神经网络能够学习更复杂的函数。

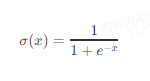
这个模型的结构是一个深层的前馈神经网络，它接受一个维度为input\_dim的向量作为输入，并通过多层全连接层进行变换，最终输出一个单一的数值，这个数值是租金的预测值。由于模型的最后一层没有激活函数，这是一个回归任务，而不是分类任务。

SiLU() 激活函数:

SiLU（Sigmoid-weighted Linear Unit）激活函数，也称为Swish激活函数，是一种在深度学习中使用的新型激活函数。它的数学表达式为：

f(x) = x \times \sigma(x)*f*(*x*)=*x*×*σ*(*x*) 

其中，\sigma(x)*σ*(*x*) 是Sigmoid函数，定义为：

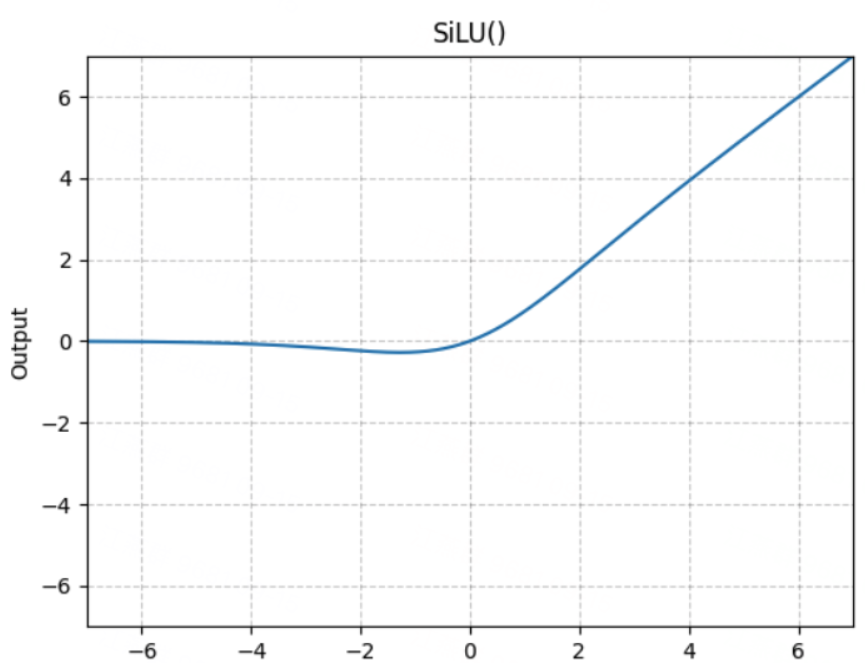
\sigma(x) = \frac{1}{1 + e^{-x}}*σ*(*x*)=1+*e*−*x*1​ 

SiLU激活函数结合了Sigmoid函数和线性函数的性质，具有以下特点：

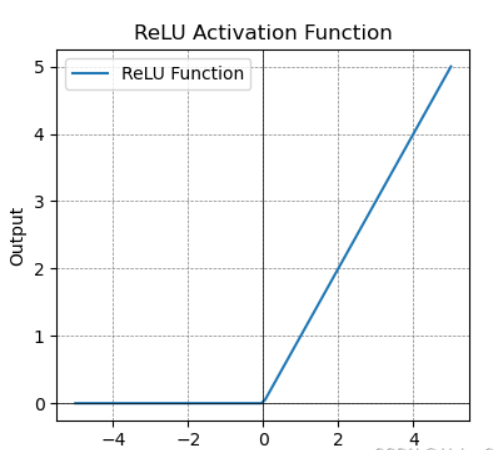
1. **平滑性**：SiLU激活函数在其定义域内是平滑的，这意味着它具有连续的导数，有助于在训练过程中使用梯度下降算法。
2. **无上界有下界**：SiLU激活函数没有上界，但有一个下界（即f(x)*f*(*x*)的最小值是负数），这有助于模型学习更复杂的函数。
3. **自门控特性**：SiLU激活函数具有自门控特性，即它的输出依赖于输入本身通过Sigmoid函数调整的比例。当输入为正时，Sigmoid函数接近1，SiLU接近线性；当输入为负时，Sigmoid函数接近0，SiLU接近0。
4. **避免梯度消失**：在输入值非常大或非常小的情况下，SiLU的导数不会接近0，这有助于避免在训练深层网络时出现的梯度消失问题。

实验表明，SiLU激活函数在某些情况下可以提供比传统的ReLU激活函数更好的性能。然而，激活函数的选择通常依赖于具体的应用和任务，没有一种通用的最佳激活函数。因此，在设计和训练深度学习模型时，尝试不同的激活函数并选择性能最佳的那个是一个常见的做法。

SiLU()激活函数图像：



而ReLU函数图像为:



为什么用SiLU而不用ReLU：

* **ReLU**：在 x < 0*x*<0 时，梯度为0；在 x > 0*x*>0 时，梯度为1。这可能导致梯度消失问题，尤其是在深层网络中。
* **SiLU**：梯度在 x < 0*x*<0 时不为0，且在整个定义域内平滑变化，有助于避免梯度消失问题。
* **ReLU**：由于ReLU的梯度可能为0，这可能导致某些神经元的权重无法更新，从而影响模型的训练。
* **SiLU**：由于其梯度在整个定义域内不为0，SiLU有助于保持训练过程中的梯度流动，这可能有助于提高模型的训练效率和性能。

2. 硬件选取

*# 将模型放到GPU上运行*

if torch.cuda.is\_available():

    device = torch.device("cuda")

else:

    device = torch.device("cpu")

*# 将数据也送到GPU*

X\_train\_tensor\_gpu = X\_train\_tensor.to(device)

X\_test\_tensor\_gpu = X\_test\_tensor.to(device)

y\_train\_tensor\_gpu = y\_train\_tensor.to(device)

y\_test\_tensor\_gpu = y\_test\_tensor.to(device)

选择图形显卡作为训练硬件，因为GPU具有高度并行化的架构，能够提供高性能计算，具有优化的内存带宽，支持专门的库和框架

同时将数据集从内存送到显存中.to(device)

1. 创建模型实例，定义加载器，损失函数，优化器

*# 创建数据加载器，指定批量大小*

batch\_size = 64

train\_dataset = torch.utils.data.TensorDataset(X\_train\_tensor\_gpu, y\_train\_tensor\_gpu)

test\_dataset = torch.utils.data.TensorDataset(X\_test\_tensor\_gpu, y\_test\_tensor\_gpu)

train\_data\_loader = DataLoader(dataset=train\_dataset, batch\_size=batch\_size, shuffle=True)

test\_data\_loader = DataLoader(dataset=test\_dataset, batch\_size=batch\_size, shuffle=False)  *# 关闭打乱*

*# 实例化模型*

print(**f**"shape X\_train\_tensor\_gpu: {X\_train\_tensor\_gpu.shape}")

*# print(f"shape[0] y\_test\_tensor\_gpu: {y\_test\_tensor\_gpu.shape[0]}")*

model = RentPredictor(X\_train\_tensor.shape[1]).to(device)

*# 定义损失函数*

criterion = nn.MSELoss()

*# criterion = nn.SmoothL1Loss()*

*# 定义优化器*

*# optimizer = optim.SGD(model.parameters(), lr=0.0001, momentum=0.9)*

*# optimizer = optim.Adam(model.parameters(), lr=0.0001, betas=(0.9, 0.999))*

optimizer = optim.AdamW(model.parameters(), lr=0.0001, betas=(0.9, 0.999), weight\_decay=0.01)

*# optimizer = optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99)*

*# optimizer = optim.Adagrad(model.parameters(), lr=0.01)*

*# optimizer = optim.NAdam(model.parameters(), lr=0.002, betas=(0.9, 0.999))*

epochs = 20

用DataLoader中加载数据，定义一个较小的batch size，给训练提供更多的梯度更新机会，定义训练循环次数epochs  
**回归任务**：MSELoss适用于回归任务，特别是当输出是一个连续值时。它计算预测值和真实值之间的平方差的平均值，这使得模型能够专注于减少预测误差。

1. **稳定性**：MSELoss对于异常值相对稳定，因为它考虑了误差的平方，放大了较大的误差。这使得模型对于异常值的影响不那么敏感。
2. **凸性**：MSELoss是一个凸函数，这意味着它有一个全局最小值，这使得优化过程更加可靠和稳定。
3. **易于解释**：MSELoss的输出是一个标量值，代表预测误差的大小，这使得损失函数的结果易于理解和解释。
4. **与梯度下降兼容**：MSELoss的梯度是连续的，这使得它能够与梯度下降和其他优化算法兼容，从而可以有效地训练模型。

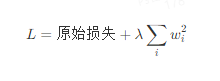
使用AdamW()作为训练优化器的原因:

1. **自适应学习率**：AdamW结合了Adam优化器的自适应学习率特性，能够根据每个参数的梯度的一阶矩估计和二阶矩估计来调整学习率，这有助于快速收敛。
2. **权重衰减**：AdamW在Adam的基础上增加了权重衰减（L2正则化）的功能，这有助于防止过拟合，并通过惩罚大权重来提高模型的泛化能力。
3. **稳定性**：AdamW在训练过程中提供了稳定的学习率调整，这有助于模型在训练过程中避免震荡和发散。
4. **适用于大规模和复杂模型**：AdamW适用于大规模和复杂的深度学习模型，因为它能够高效地处理大量的参数和复杂的梯度动态。
5. **社区认可**：AdamW在深度学习社区中得到了广泛的应用和认可，许多成功的模型和论文都使用了AdamW作为优化器。

optim.AdamW是torch.optim.Adam的变体 它的参数：

1. lr=0.0001：
   * 学习率（learning rate）是优化器调整参数值的大小。这里学习率被设置为0.0001，意味着每次迭代时，参数的更新将是非常小的。
2. betas=(0.9, 0.999)：
   * 这是两个超参数，它们用于AdamW算法中的动量（momentum）计算。动量是一种技术，它帮助优化器更快地收敛。这些参数控制了梯度的一阶矩估计和二阶矩估计的指数衰减速率。在配置中第一个参数（0.9）用于一阶矩估计，第二个参数（0.999）用于二阶矩估计。
3. weight\_decay=0.01：
   * 权重衰减（weight decay）是一种正则化技术，它通过将每个参数的权重值乘以一个小的衰减因子来防止过拟合。这里权重衰减被设置为0.01，这意味着每个参数的权重值在每次迭代时将减少其原始值的0.01。权重衰减有助于模型在训练过程中保持泛化能力。

，L2正则化的损失函数可以表示为：

L = \text{原始损失} + \lambda \sum\_{i} w\_i^2*L*=原始损失+*λi*∑​*wi*2​ 

其中，w\_i*wi*​ 是模型的权重参数，\lambda*λ* 是正则化强度的一个超参数，用于控制正则化项相对于原始损失的重要性

4.训练模型

*# 训练模型*

for epoch in range(epochs):

*# 训练模式*

    model.train()

    for batch in train\_data\_loader:

        X\_batch, y\_batch = batch

        optimizer.zero\_grad()

        y\_pred = model(X\_batch)

        loss = criterion(y\_pred, y\_batch)

        loss.backward()

        optimizer.step()

*# 评估模式*

    model.eval()

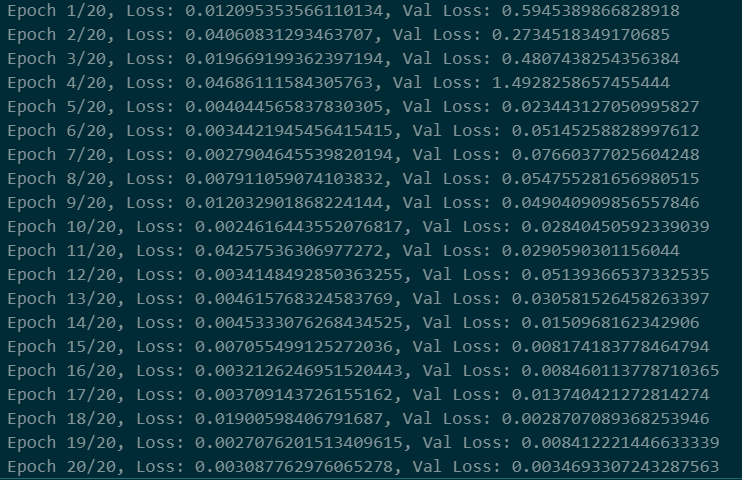
    with torch.no\_grad():

        for X\_val\_batch, y\_val\_batch in test\_data\_loader:

            y\_val\_pred = model(X\_val\_batch)

            val\_loss = criterion(y\_val\_pred, y\_val\_batch)

        print(**f**'Epoch {epoch+1}/{epochs}, Loss: {loss.item()}, Val Loss: {val\_loss.item()}')



明显看出模型损失震荡下降

Loss 为模型一轮训练后在训练数据上的损失值， val\_loss为模型一轮训练后在测试集上的损失值，val\_loss反映了模型在未见过的数据上的泛化能力，它是评估模型性能和检测过拟合的重要指标。

5.指标计算，模型验证

*# 5. 计算指标*

model.eval()

with torch.no\_grad():

    y\_test\_pred = model(X\_test\_tensor\_gpu)

*# print(f"shape: {y\_test\_pred.shape}")*

    test\_loss = criterion(y\_test\_pred, y\_test\_tensor\_gpu.unsqueeze(1))

*# print(f"shape: {y\_test\_tensor\_gpu.shape}")*

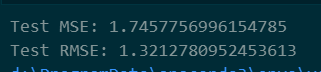
    mse = test\_loss.item()

    rmse = torch.sqrt(test\_loss).item()

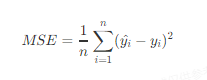
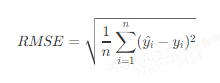
*# 打印指标*

print(**f**'Test MSE: {mse}')

print(**f**'Test RMSE: {rmse}')



直接调用损失函数得到MSE和RMSE，

* + **MSE**：
    - MSE 是预测值与真实值差的平方的平均值。数学上，如果 \hat{y}*y*^​ 是预测值，y*y* 是真实值，则 MSE 为：MSE = \frac{1}{n} \sum\_{i=1}^{n} (\hat{y\_i} - y\_i)^2*MSE*=*n*1​*i*=1∑*n*​(*yi*​^​−*yi*​)2
    - **解读**：MSE 越小，表示预测值与真实值之间的差异越小，模型的预测能力越强。
  + **RMSE**：
    - RMSE 是 MSE 的平方根。数学上，RMSE 为：RMSE = \sqrt{\frac{1}{n} \sum\_{i=1}^{n} (\hat{y\_i} - y\_i)^2}*RMSE*=*n*1​*i*=1∑*n*​(*yi*​^​−*yi*​)2​
    - 

交叉验证分数：调用from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.metrics import mean\_squared\_error, r2\_score

model.eval()

with torch.no\_grad():

    y\_val\_pred = model(X\_test\_tensor.to(device))

*# 将预测结果从GPU复制到CPU*

y\_val\_pred\_cpu = y\_val\_pred.to('cpu')

*# 将预测结果转换为Numpy数组*

y\_val\_pred\_numpy = y\_val\_pred\_cpu.numpy()

*# 安全地使用y\_val\_pred\_numpy作为mean\_squared\_error的输入*

mse\_lr = mean\_squared\_error(y\_test, y\_val\_pred\_numpy)

rmse\_lr = np.sqrt(mse\_lr)

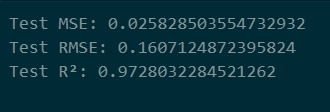
r2\_lr = r2\_score(y\_test, y\_val\_pred\_numpy)

*# 打印指标*

print(**f**'Test MSE: {mse\_lr}')

print(**f**'Test RMSE: {rmse\_lr}')

print(**f**'Test R²: {r2\_lr}')



调用sklearn包和模型损失函数得到的MSE和RMSE略微不同，但RMSE同处于一个数量级

6.模型参数量

*# 获取模型的所有参数*

parameters = list(model.parameters())

*# 将所有参数打包成一个向量*

parameters\_vector = torch.cat([p.view(-1) for p in parameters])

*# 计算参数量*

num\_parameters = parameters\_vector.numel()

print(**f**"The model has {num\_parameters} parameters.")



进行简单的测试

*# 选择测试集中的几个样本*

X\_test\_sample = X\_test\_tensor\_gpu[:5]  *# 选择前5个样本*

y\_test\_sample = y\_test\_tensor\_gpu[:5]

*# 使用模型进行预测*

with torch.no\_grad():  *# 确保不计算梯度*

    y\_pred\_sample = model(X\_test\_sample)

*# 将预测结果和实际目标值转换为 NumPy 数组*

y\_pred\_sample\_np = y\_pred\_sample.cpu().numpy()

y\_test\_sample\_np = y\_test\_sample.cpu().numpy()

*# 反归一化*

y\_pred\_original = standard\_scaler.inverse\_transform(y\_pred\_sample\_np)

y\_test\_original = standard\_scaler.inverse\_transform(y\_test\_sample\_np)

*# 打印反归一化后的预测结果和实际目标值*

print('反归一化后的预测结果:', y\_pred\_original)

print('反归一化后的实际目标值:', y\_test\_original)

