机器学习个人作业

数据集预处理

1. **第一步，先观察数据集。**该数据集共有17661条训练数据，4416条测试数据，每条数据有222条属性。有如下观察结果：
    1. 该数据集存在数据缺失，有很多空缺项。
    2. 每支股票都属于一个经济领域，所属经济领域以字符串的形式写在'Sector'列
    3. 数据集中存在不合理数据，需要过滤掉
2. **第二步，对数据集进行处理：**
    1. 对于缺失项，准备了两种处理方式
        1. 全部填0

```
# 读取两个 CSV 文件
file_path1 = "dataSet/train.csv"  # 替换为第一个 CSV 文件路径
file_path2 = "dataSet/test.csv"  # 替换为第二个 CSV 文件路径
df1 = pd.read_csv(file_path1)
df2 = pd.read_csv(file_path2)
df1.fillna(0, inplace=True)  # 将所有 NaN 替换为 0
df2.fillna(0, inplace=True)
```

        2. 填本列非空缺处的平均值

```
# 确保数据中的非数字值被处理掉
# 先选择数值类型的列
numeric_columns1 = df1.select_dtypes(include=['number']).columns
numeric_columns2 = df2.select_dtypes(include=['number']).columns

# 对每个数值列进行检查，非数值型内容替换为NaN
df1[numeric_columns1] = df1[numeric_columns1].apply(pd.to_numeric,
errors='coerce')
df2[numeric_columns2] = df2[numeric_columns2].apply(pd.to_numeric,
errors='coerce')

# 用每列的平均值填充 NaN
df1.fillna(df1.mean(), inplace=True)
df2.fillna(df2.mean(), inplace=True)
```

        经过后续验证，发现全部填0的效果更好
    2. 对于字符串形式的所属领域，对每个领域进行标号，用标号代替所属领域
    3. 对一些不合理结果进行过滤，如过大的PRICE VAR

        这里采用四分位距法（IQR）检测异常值，并删除
        1. 原理介绍：

            四分位距法的核心思想是基于数据的 **四分位数**（quartiles），即将数据集分为四等份，然后使用 **四分位距（IQR）** 来定义数据的变异性，并通过这个范围来检测是否有值超出了该范围，从而标识为异常值。

步骤：

1. **计算四分位数**：
   - **第一四分位数 (Q1)**：数据的第 25% 位置的值，也称为下四分位数。它将数据集的前 25% 数据分开。
   - **第三四分位数 (Q3)**：数据的第 75% 位置的值，也称为上四分位数。它将数据集的前 75% 数据分开。
   - **中位数 (Q2)**：数据的第 50% 位置的值，通常用于表示数据的中心。
2. **计算四分位距 (IQR)**：
   - **IQR** = Q3 - Q1
   - IQR 代表了数据集中间 50% 数据的范围，反映了数据的分散程度。值越大，表示数据的变异性越大。
3. **确定异常值的阈值**：
   - 使用

     IQR

     来定义正常数据的范围：
     - **下限** = Q1 - 1.5 * IQR
     - **上限** = Q3 + 1.5 * IQR
   - 任何小于下限或大于上限的数据点都可以被视为异常值（outliers）。
   - 这个 **1.5** 倍的倍数是经验法则，适用于大多数数据集。如果你的数据更复杂，或者异常值特别明显，你可以调整这个倍数，比如使用 **3倍IQR** 来捕捉更远的异常值。

2. 代码实现：

```
# 定义去除不合理值的函数
def remove_outliers(data, method="iqr", threshold=1.5):
    """
    去除不合理值（异常值）
    参数:
    - data: DataFrame 数据
    - method: 异常值检测方法（目前支持 'iqr'）
    - threshold: 阈值系数，默认 1.5 对应 IQR 方法
    返回:
    - 清洗后的数据
    """
    column = 'PRICE VAR [%]'
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - threshold * IQR
    upper_bound = Q3 + threshold * IQR
    # 筛选数据范围
    data = data[(data[column] >= lower_bound) & (data[column] <=
upper_bound)]
    return data
```

4. 数据标准化

```
# 数据标准化
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

`scaler = StandardScaler()`
创建一个 `StandardScaler` 对象。`StandardScaler` 是一个用于数据标准化的工具，它会将数据按特征（列）进行标准化，使得每个特征的均值为 0，方差为 1。

`X_train = scaler.fit_transform(X_train)`
对训练数据 `X_train` 进行标准化处理。`fit_transform` 方法有两个步骤：

- `fit`：计算训练数据的均值和标准差。
- `transform`：使用计算出的均值和标准差对训练数据进行标准化。

标准化后，`X_train` 中的每一列（特征）都会被转换成均值为 0，标准差为 1 的数据。

`X_test = scaler.transform(X_test)`
对测试数据 `X_test` 进行标准化。由于测试数据不应参与训练过程中的统计量计算，所以只使用 `scaler` 对象中从训练数据计算出的均值和标准差进行转换，而不是重新计算。这样确保训练数据和测试数据使用相同的尺度。

模型分析

用如下两种模型，对结果进行对比分析

1. 支持向量机
2. 神经网络模型，全连接层+隐层，设置激活函数，隐层数，epochs，损失函数MSE

分别对两种模型进行原理分析和代码实现

支持向量回归

模型原理：

SVR 通过将数据映射到高维空间，试图找到一个平面（或超平面），使得平面与大多数训练数据点的距离尽可能小，同时还要对少数偏离平面的数据点进行容忍。SVR 的目标是尽可能找到一个模型，允许一定的误差，但误差超过某个阈值时，会进行惩罚。

在分类问题中，SVM 会构建一个超平面来分割不同的类别；而在回归问题中，SVR 会构建一个“ε-不敏感”超平面，其中数据点位于该平面上或距离平面不到一定阈值的误差范围内视为正确预测。

1.1 数学模型

假设我们有一个输入数据集：

$\{(x1, y1), (x2, y2), \ldots, (xn, yn)\}$

其中，x_i 是特征向量，y_i 是目标变量（输出）。SVR 的目标是找到一个回归函数 $f(x)$ 来拟合这些数据。SVR 中，回归函数的表达式为：

$$f(x) = w \cdot x + b$$

其中：

- w 是权重向量（线性超平面的法向量），
- b 是偏置。

为了实现回归，SVR 引入了 **ε-不敏感损失函数**（ε-insensitive loss function），使得模型不对小于 ε 的误差做惩罚，而对大于 ε 的误差进行惩罚。

1.2 ε-不敏感损失函数：

SVR 采用一个称为 ε-不敏感损失函数的技术，这意味着对于那些在 ϵ\epsilonϵ 距离范围内的误差，模型不会进行惩罚。只有当预测结果与真实结果的差距超过 ε 时，模型才会进行惩罚。

损失函数定义如下：

$$L_\epsilon(y, f(x)) = \begin{cases} 0 & \text{if } |y - f(x)| \leq \epsilon \\ |y - f(x)| - \epsilon & \text{otherwise} \end{cases}$$

因此，SVR 的目标是最小化以下目标函数：

$$\min_{w,b,\xi} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i$$

其中：

- $\|w\|^2$是权重向量的范数，控制模型的复杂度，避免过拟合；
- C 是惩罚参数，用来平衡模型的复杂度与训练误差；
- ξ_i 是松弛变量，表示第 i 个数据点的偏离度，衡量预测结果与真实值之间的误差。

模型结构

SVR 的回归模型可以通过以下几个关键点来描述：

1. **超平面**：SVR 通过寻找一个超平面（在高维空间中的线性函数）来拟合数据。
2. 核函数
   ：由于SVR主要是通过映射到高维空间进行回归，因此通常需要使用
   核函数
   来对输入数据进行非线性映射。常用的核函数有：
   - 线性核：$K(x, x') = x \cdot x'$
   - 高斯径向基核（RBF核）：$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$
   - 多项式核：$K(x, x') = (x \cdot x' + c)^d$
3. **支持向量**：与支持向量机分类一样，SVR 也依赖于支持向量。这些支持向量是模型训练过程中对回归结果产生较大影响的样本点，通常是偏离超平面较远的点。

代码实现与结果

在pytorch中，支持向量回归做了很好的函数封装，将数据转化为tensor格式之后调用函数即可得到回归结果

```
import os

import pandas as pd
import torch
import torch.nn as nn
import zipfile
import torch.optim as optim
from sklearn.neural_network import MLPRegressor
```

```
from sklearn.preprocessing import StandardScaler
import xgboost as xgb

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

train_file_path = "dataSet/output_train.csv"
test_file_path = "dataSet/output_test.csv"
# 读取训练数据
train_data = pd.read_csv(train_file_path)
test_data = pd.read_csv(test_file_path)

X_train = train_data.iloc[:, 1:-1].values
y_train = train_data.iloc[:, -1].values
X_test = test_data.iloc[:, 1:-1].values

# 数据标准化
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)  # 转换为二维张量
X_test = torch.tensor(X_test, dtype=torch.float32)

# SVR回归
model = SVR(kernel='rbf')
y_train = y_train.ravel()
model.fit(X_train, y_train)
predictions = model.predict(X_test)

# 保存预测结果到 CSV 文件
output_file_path = "dataSet/submission.csv"
output_pd = pd.read_csv(output_file_path)
output_pd['PRICE VAR [%]'] = predictions
output_pd.to_csv(output_file_path, index=False)

# 创建一个 ZIP 文件并将 CSV 文件添加进去
zip_file_path = "dataSet/submission.zip"
if os.path.exists(zip_file_path):
    os.remove(zip_file_path)
with zipfile.ZipFile(zip_file_path, 'w', zipfile.ZIP_DEFLATED) as zipf:
    zipf.write(output_file_path, os.path.basename(output_file_path))  # 将 CSV 文件添加到 ZIP 文件中
    print(f"CSV 文件已成功添加到 ZIP 文件：{zip_file_path}")

#torch.save(model.state_dict(), './model.ckpt')
```

评测结果：

神经网络回归

模型原理

神经网络回归是使用神经网络模型来解决回归问题。与分类问题不同，回归问题的目标是预测一个连续的值，而不是一个类别标签。在神经网络回归中，模型通过多个层次的神经元（或称为节点）进行处理，每一层对输入进行非线性变换，最后输出一个连续值。

神经网络回归的核心思想是通过训练网络中的权重来拟合数据，最终得到一个可以进行数值预测的模型。

1. 神经网络回归模型结构

神经网络回归的结构通常由以下几个部分组成：

1. **输入层（Input Layer）**：
   - 输入层接收来自特征的数据，每个输入神经元对应数据集中的一个特征。
   - 例如，如果输入数据有 n 个特征，那么输入层就会有 n 个神经元。
2. **隐藏层（Hidden Layers）**：
   - 隐藏层是神经网络的核心部分，它由若干个神经元组成。每个神经元都会接收来自上一层的加权输入，并通过激活函数进行变换。
   - 隐藏层的数量和每层神经元的数量通常是可调的。多层隐藏层可以让模型捕捉到数据中的复杂模式和非线性关系。
3. **激活函数（Activation Function）**：
   - 激活函数在神经元中起着至关重要的作用，它决定了神经元是否被激活以及如何被激活。常见的激活函数有：
     - **ReLU（Rectified Linear Unit）**：$f(x) = \max(0, x)$
     - **Sigmoid**：$f(x) = \frac{1}{1+e^{-x}}$
     - **Tanh**：$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
   - 对于回归问题，最后一层通常不使用激活函数，直接输出连续值。
4. **输出层（Output Layer）**：
   - 输出层的神经元数目与回归问题的目标变量数目相同。对于单一目标的回归问题，输出层通常只有一个神经元。
   - 输出层的激活函数通常为**线性函数**，因为回归问题的目标是预测一个连续的值。
5. **损失函数（Loss Function）**：

- 在回归问题中，常用的损失函数是 **均方误差（MSE, Mean Squared Error）**，计算预测值与实际值之间的差异。
- 损失函数定义为：$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ 其中，y_i 是真实值，$\hat{y}_i$ 是预测值，n 是样本数量。

2. 神经网络回归的训练过程

神经网络回归模型的训练过程主要包括以下步骤：

1. **前向传播（Forward Propagation）**：
   - 将输入数据传递给输入层，然后依次传递给隐藏层，最后得到输出层的预测结果。
   - 每个神经元通过加权求和后，经过激活函数计算得到输出。
2. **计算损失（Loss Calculation）**：
   - 通过损失函数计算预测结果与实际标签之间的误差。例如，在回归问题中通常使用均方误差（MSE）。
3. **反向传播（Backpropagation）**：
   - 通过反向传播算法，计算每个权重和偏置对损失函数的梯度（即误差的导数）。通过链式法则将误差从输出层传递回输入层。
   - 反向传播的目的是通过梯度下降算法调整网络中的参数（权重和偏置），使得模型的预测误差最小化。
4. **梯度下降（Gradient Descent）**：
   - 通过优化算法（如批量梯度下降、随机梯度下降、Adam 等）更新神经网络的参数，最小化损失函数。
   - 梯度下降的更新公式为：$\theta = \theta - \eta \cdot \nabla_\theta L$ 其中，θ 是模型的参数，η 是学习率，$\nabla_\theta L$ 是损失函数的梯度。
5. **迭代训练**：
   - 训练过程不断进行迭代，每次迭代后，网络会更新权重和偏置，以减少损失函数的值。

3. 神经网络回归的调参过程

调参是提高神经网络回归模型性能的重要环节。常见的超参数包括：

1. **学习率（Learning Rate）**：
   - 学习率决定了每次参数更新的步长。如果学习率过大，可能导致参数更新过快而错过最优解；如果学习率过小，模型收敛速度会很慢。
2. **隐藏层的数量与神经元数量**：
   - 隐藏层的数量和每层的神经元数量决定了神经网络的复杂度。增加隐藏层和神经元数量可以提高模型的表达能力，但也可能导致过拟合。
   - 需要通过交叉验证或验证集来选择合适的结构。
3. **激活函数**：
   - 激活函数的选择会影响神经网络的训练效果。对于回归任务，通常最后一层不使用激活函数，而在隐藏层使用ReLU或其他非线性激活函数。
   - 本次作业尝试了三个激活函数，ReLU、tanh和sigmod函数：
     1. **ReLU（Rectified Linear Unit）**
        数学表达式：

$$f(x) = \max(0, x)$$

优点：简单高效，缓解梯度消失问题；在正值区间，梯度恒为 1。
缺点：负值输出始终为 0，可能导致 **Dead Neurons（神经元死亡）** 问题。

2. **Tanh（双曲正切函数）**
   数学表达式：
   $$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
   输出范围：$(-1, 1)$。
   优点：比 Sigmoid 更适合于归一化的数据，因为输出中心对称于 0。
   缺点：可能出现梯度消失问题。

3. **Sigmoid（S 形函数）**
   数学表达式：
   $$f(x) = \frac{1}{1 + e^{-x}}$$
   输出范围：$(0, 1)$。
   优点：适用于概率输出。
   缺点：梯度在极端值时趋近于 0，容易导致梯度消失。

总结与结果：（模型参数：两层隐层，第一层128个节点，第二层16个节点，epochs=1000，lr=0.001）

- **ReLU**：适用于大多数隐藏层。

  最终结果：

  

- **Tanh**：适用于输入或输出对称的数据。

  最终结果：

  

- **Sigmoid**：多用于二分类任务的输出层。

  最终结果：

4. **正则化**：
    - 为了防止过拟合，可以使用正则化技术，如L2正则化（权重衰减），或者使用Dropout层（随机丢弃部分神经元）来提高模型的泛化能力。
    - 由于课程组给的数据集较小，所以此次作业使用正则化大概率会导致模型欠拟合，因此不考虑正则化
5. **批次大小（Batch Size）**：
    - 批次大小决定了每次训练时使用的样本数。较小的批次大小可以让模型更快收敛，但可能导致噪声较大。较大的批次大小可以减小噪声，但可能需要更多的计算资源。
    - 本次作业中不使用批次，即一条数据为一批次
6. **迭代次数（Epochs）**：
    - 迭代次数是模型训练时遍历数据集的次数。过多的迭代次数可能导致过拟合，过少的迭代次数可能导致欠拟合。
    - 经过多次验证，迭代1000次后模型收敛，继续迭代可能导致过拟合

      训练1000次后结果：

      

      训练10000次结果：

      

代码实现与最终结果

```
import os

import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import zipfile
import torch.optim as optim
import tqdm
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.impute import SimpleImputer
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
train_file_path = "dataSet/output_train.csv"
test_file_path = "dataSet/output_test.csv"

myseed = 3900
np.random.seed(myseed)
torch.manual_seed(myseed)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(myseed)

class DataSet:
    def __init__(self, train_path, test_path):
        train_data = pd.read_csv(train_path)
        test_data = pd.read_csv(test_path)

        #print(train_data.shape)

        X = train_data.iloc[:, 1:-1].values
        y = train_data.iloc[:, -1].values
        X_test = test_data.iloc[:, 1:-1].values

        scaler = StandardScaler()
        X = scaler.fit_transform(X)
        X_test = scaler.transform(X_test)


        X_train = X[[i for i in range(len(X)) if i % 100 != 0]]
        y_train = y[[i for i in range(len(X)) if i % 100!= 0]]
        X_val = X[[i for i in range(len(X)) if i % 100 == 0]]
        y_val = y[[i for i in range(len(X)) if i % 100 == 0]]
        #X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.1,
random_state=966)

        self.X_train = torch.tensor(X_train, dtype=torch.float32)
        self.X_val = torch.tensor(X_val, dtype=torch.float32)
        self.y_train = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)  #
转换为二维张量
```

```
        self.y_val = torch.tensor(y_val, dtype=torch.float32).unsqueeze(1)
        self.X_test = torch.tensor(X_test, dtype=torch.float32)

    def get_data(self, type):
        if type == "train":
            return self.X_train, self.y_train
        elif type == "val":
            return self.X_val, self.y_val
        elif type == "test":
            return self.X_test


# # 读取训练数据
# train_data = pd.read_csv(train_file_path)
# test_data = pd.read_csv(test_file_path)
#
# X_train = train_data.iloc[:, 1:-1].values
# y_train = train_data.iloc[:, -1].values
# X_test = test_data.iloc[:, 1:-1].values
#
# scaler = StandardScaler()
# X_train = scaler.fit_transform(X_train)
# X_test = scaler.transform(X_test)
#
# # 定义PCA对象，例如设置保留95%的方差
# pca = PCA(n_components=0.95)
# # 在训练数据上拟合PCA并进行降维
# X_train = pca.fit_transform(X_train)
# # 在测试数据上应用相同的PCA变换
# X_test = pca.transform(X_test)
#
# X_train = torch.tensor(X_train, dtype=torch.float32)
# y_train = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)  # 转换为二维张
量
# X_test = torch.tensor(X_test, dtype=torch.float32)
#
# dataset = torch.utils.data.TensorDataset(X_train, y_train)
# dataloader = torch.utils.data.DataLoader(dataset, batch_size=256, shuffle=True)


class FullyConnectedNN(nn.Module):
    def __init__(self, input_size):
        super(FullyConnectedNN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, 500),
            nn.BatchNorm1d(500),
            nn.Dropout(0.5),
            nn.ReLU(),

            nn.Linear(500, 1000),
            nn.BatchNorm1d(1000),
            nn.Dropout(0.5),
            nn.ReLU(),

            nn.Linear(1000, 2000),
            nn.BatchNorm1d(2000),
```

```
            nn.Dropout(0.5),
            nn.ReLU(),

            nn.Linear(2000, 1000),
            nn.BatchNorm1d(1000),
            nn.Dropout(0.5),
            nn.ReLU(),

            nn.Linear(1000, 500),
            nn.BatchNorm1d(500),
            nn.Dropout(0.5),
            nn.ReLU(),

            nn.Linear(500, 1),
        )

    def forward(self, x):
        x = self.net(x)
        return x




def package(predictions, is_cuda=False):
    # 保存预测结果到 CSV 文件
    output_file_path = "dataSet/submission.csv"
    output_pd = pd.read_csv(output_file_path)
    if is_cuda:
        output_pd['PRICE VAR [%]'] = predictions.cpu().numpy()
    else:
        output_pd['PRICE VAR [%]'] = predictions
    # test_result = pd.read_csv(filepath_or_buffer='dataSet/test_result.csv')
    # test_result.fillna(0, inplace=True)
    # for index, row in test_result.iterrows():
    #     if row['PRICE VAR [%]'] != 0:
    #         output_pd.loc[index, 'PRICE VAR [%]'] = row['PRICE VAR [%]']

    output_pd.to_csv(output_file_path, index=False)

    # 创建一个 ZIP 文件并将 CSV 文件添加进去
    zip_file_path = "dataSet/submission.zip"
    if os.path.exists(zip_file_path):
        os.remove(zip_file_path)
    with zipfile.ZipFile(zip_file_path, 'w', zipfile.ZIP_DEFLATED) as zipf:
        zipf.write(output_file_path, os.path.basename(output_file_path))  # 将
CSV 文件添加到 ZIP 文件中
        print(f"CSV 文件已成功添加到 ZIP 文件：{zip_file_path}")


def FullyConnectedNN_train(train=True):
    dataSet = DataSet(train_file_path, test_file_path)

    X_train, y_train = dataSet.get_data("train")
    X_test = dataSet.get_data("test")
    X_val, y_val = dataSet.get_data("val")
```

```
    dataset = torch.utils.data.TensorDataset(X_train, y_train)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=500,
shuffle=True)

    model = FullyConnectedNN(X_train.shape[1]).float().to(device)
    criterion = nn.MSELoss(reduction='mean')
    optimizer = optim.Adam(model.parameters(), weight_decay=0.1)

    if train:
        # 训练模型
        epochs = 10000
        early_stop = 0
        min_rmse = torch.inf
        for epoch in range(epochs):
            for batch_X, batch_y in dataloader:
                # print(batch_X.shape, batch_y.shape)
                optimizer.zero_grad()
                outputs = model.forward(batch_X.to(device))
                # print(outputs)
                loss = criterion(outputs, batch_y.to(device))
                # print(loss)
                loss.backward()
                optimizer.step()

            if (epoch + 1) % 10 == 0:
                print(f"Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}")
            model.eval()
            with torch.no_grad():
                predictions = model(X_val.to(device))
                #print(predictions.shape)
                #print(y_val.shape)
                # 计算 RMSE
                mse = mean_squared_error(y_val.cpu().numpy(),
predictions.cpu().numpy())
                rmse = np.sqrt(mse)
                if epoch>100 :
                    print(f'epoch:{epoch+1}  rmse: {rmse}  min_rmse:{min_rmse}')
                    if rmse < min_rmse:
                        min_rmse = rmse
                        early_stop = 0
                        print(f'Saving model (epoch = {epoch + 1}, loss =
{min_rmse})')
                        torch.save(model.state_dict(), "./model.ckpt")
                    else:
                        early_stop += 1
                    if early_stop > 1000:
                        torch.save(model.state_dict(), "./model_final.ckpt")
                        break
    model.load_state_dict(torch.load("./model.ckpt"))
    model.eval()
    with torch.no_grad():
        predictions = model(X_test.to(device))

    package(predictions, is_cuda=True)
```

```python
if __name__ == '__main__':
    # model = FullyConnectedNN_train()
    # x = input("1:随机森林 2:神经网络\n")
    # if x == "1":
    #     RandForest_train()
    #     exit()
    # elif x == "2":
    FullyConnectedNN_train(True)
    exit()
```

后续经过反复调参得到最终结果：

排名：（第三名）

#	用户名	登录	上次登录日期	RMSE ▲
1	22230608_范一泽	66	12/11/24	4216.128607 (1)
2	22230616_黄勇鑫	54	12/11/24	4257.604840 (2)
3	22371461_钟芳桔	42	12/30/24	6236.985837 (3)

同时我也用相同的架构运行了"股票选取"这题，也取得了不错的成绩

具体调参过程不再赘述，直接贴结果（第四名）

#	用户名	登录	上次登录日期	Accuracy ▲
1	22230615_康硕	7	12/26/24	0.636775 (1)
2	22230611_黄瑞翔	36	12/26/24	0.633605 (2)
3	22371092_匡亦煊	13	12/23/24	0.631114 (3)
4	22371461_钟芳桔	16	12/30/24	0.626812 (4)
5	22371144_秦子奇	16	12/25/24	0.625000 (5)
6	22230608_范一泽	34	12/13/24	0.623868 (6)
7	22373195_张敏尔	18	12/23/24	0.622283 (7)
8	22230616_黄勇鑫	52	12/13/24	0.621150 (8)
9	21375212_姜涵章	82	12/30/24	0.620697 (9)