HW-1: Regression

姓名：罗威　学号：SA24218095

1 概述

本次作业旨在构建一个全连接神经网络模型，对波士顿房价进行预测。通过对波士顿房价数据集进行特征筛选、数据预处理，利用PyTorch搭建并训练神经网络模型，最终评估模型在测试集上的性能。

1.1 训练结果总结

- 首先，对13组数据进行了标签相关性计算，并在代码中对相关性较低的数据进行筛选，以此降低不相关数据产生的噪声影响。
- 对相关性较低的数据组合进行剔除过后，训练结果在测试集上的均方误差为11.9592。
- 对先前的代码加入早停策略，使模型在验证集上均方误差不再下降时保存模型并停止训练。通过早停策略训练得到的模型在测试集上的均方误差进一步降低到6.0042。

1.2 软硬件环境

（1）系统环境：**Ubuntu 22.04 LTS**

（2）语言及框架版本：

- `Python`: 3.11.11
- `Pytorch`: 2.5.0+cu124
- `Pandas`: 2.2.3
- `Sklearn`: 1.6.0

2 标签相关性分析

利用`pandas`库自带的`corr()`函数进行相关性分析。该函数可以计算出多种相关性系数（pearson、kendall、spearman），默认计算的是皮尔逊系数。

```python
import pandas as pd

data = pd.read_excel('./BostonHousingData.xlsx', sheet_name='Sheet1')

correlation_matrix = data.corr()
correlation = correlation_matrix['MEDV'].sort_values(ascending=False).drop('MEDV')

print("The correlation coefficients with MEDV:")
print(correlation)
```

```
The correlation coefficients with MEDV:
RM          0.695360
ZN          0.360445
B           0.333461
DIS         0.249929
CHAS        0.175260
AGE        -0.376955
```

```
RAD        -0.381626
CRIM       -0.388305
NOX        -0.427321
TAX        -0.468536
INDUS      -0.483725
PTRATIO    -0.507787
LSTAT      -0.737663
Name: MEDV, dtype: float64
```

相关系数的取值范围为[-1,1]，其中系数越接近-1和1的标签，与房价的相关性越高。可以看到，计算出的相关系数中，DIS、CHAS与房价的相关性较低；同时，ZN、B、AGE、RAD、CRIM等的相关性相对也较低。因此，可以逐渐剔除这些相关性低下的标签，让模型获得更好的拟合效果。

3 全连接神经网络模型设计

构建了一个全连接神经网络模型`Regression`，具体结构如下：

（1）包含5个有参层，分别为3个全连接层（`nn.Linear`）和2个批量归一化层（`nn.BatchNorm1d`）其中的2个隐藏层均使用`nn.Linear`进行线性变换，接着使用`nn.BatchNorm1d`进行批量归一化，`nn.ReLU`作为激活函数，`nn.Dropout(0.2)`进行正则化，防止过拟合；

（2）使用常用的`Adam`优化器来进行参数更新，学习率设置为0.001，权重衰减设置为0.001，有助于模型在训练过程中更快地收敛并防止过拟合；

（3）使用均方误差损失函数`nn.MSELoss()`，用于衡量模型预测值与真实值之间的差异。

如下结果所示，最佳训练结果在测试集上的均方误差为11.9592。

```python
import torch
import pandas as pd
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import StandardScaler

# 1.从.xlsx文件中读取数据集
data = pd.read_excel('BostonHousingData.xlsx', sheet_name='Sheet1')

# 2.计算相关系数并通过设置的阈值来筛选数据
threshold = 0.5
corre = data.corr()
correlation = corre['MEDV'].sort_values(ascending=False)
selected_features = correlation[abs(correlation) >=
threshold].index.tolist()

selected_data = data[selected_features]
X = selected_data.drop('MEDV', axis=1).values
y = selected_data['MEDV'].values.reshape(-1, 1)

# 3.划分训练集和测试集
X_train, X_test = X[:450], X[450:]
y_train, y_test = y[:450], y[450:]

# 4.数据标准化处理、转换格式为pytorch张量
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

```

```python
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# 5.使用库函数加载数据集
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# 定义输入数据维度，与阈值设置有关
input_size = selected_data.shape[1] - 1

# 6.全连接神经网络模型的定义
class Regression(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(64, 1)
        )

    def forward(self, x):
        return self.fc(x)

model = Regression()

# 7. 定义损失函数和优化器
criterion = torch.nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

# 8. 训练模型：自定义训练次数
for epoch in range(100):
    model.train()
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{100}], Loss: {loss.item():.4f}')

# 9. 模型评估
model.eval()
total_loss = 0
with torch.no_grad():
```

```python
    for inputs, labels in test_loader:
        outputs = model(inputs)
        total_loss += criterion(outputs, labels).item() * inputs.size(0)

mse = total_loss / len(test_dataset)

print(f'\nMSE on test dataset: {mse:.4f}')
```

```
Epoch [10/100], Loss: 95.0033
Epoch [20/100], Loss: 25.6044
Epoch [30/100], Loss: 28.5615
Epoch [40/100], Loss: 13.7481
Epoch [50/100], Loss: 125.9341
Epoch [60/100], Loss: 72.2138
Epoch [70/100], Loss: 29.9736
Epoch [80/100], Loss: 95.5638
Epoch [90/100], Loss: 20.1719
Epoch [100/100], Loss: 176.2024

MSE on test dataset: 11.9592
```

4 使用早停与模型保存的训练方法

对于模型训练，固定的`epoch`使得最终训练得到的模型可能并不是最佳模型。`epoch`过多可能导致模型记忆噪声，造成过拟合。`epoch`不足时模型未充分学习，造成欠拟合。因此，采用较大的`epoch`结合早停策略，可以在模型性能表现最佳时将模型保存下来。

早停策略的参数设置如下：

`best_val_loss`：初始化为正无穷大，用于记录验证集上的最小损失。

`patience`：设定为20，表示当验证集损失在连续20个`epoch`中没有下降时，触发早停机制。

`counter`：用于记录验证集损失没有下降的连续`epoch`数量，初始化为0。

如下结果所示，最佳训练结果在测试集上的均方误差进一步降低到6.0042。

```python
import torch
import pandas as pd
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# 1.从.xlsx文件中读取数据集
data = pd.read_excel('BostonHousingData.xlsx', sheet_name='Sheet1')

# 2.计算相关系数并通过设置的阈值来筛选数据
threshold = 0
corre = data.corr()
correlation = corre['MEDV'].sort_values(ascending=False)
selected_features = correlation[abs(correlation) >= threshold].index.tolist()

selected_data = data[selected_features]
X = selected_data.drop('MEDV', axis=1).values
y = selected_data['MEDV'].values.reshape(-1, 1)
```

```python

# 3.划分训练集和验证集、测试集
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.1, random_state=42)

# 4.数据标准化处理、转换格式为pytorch张量
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val_scaled, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# 5.使用库函数加载数据集
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# 定义输入数据维度，与阈值设置有关
input_size = selected_data.shape[1] - 1

# 6.全连接神经网络模型的定义
class Regression(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(64, 1)
        )

    def forward(self, x):
        return self.fc(x)

model = Regression()

# 7. 定义损失函数和优化器
criterion = torch.nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0005, weight_decay=0.0001)
```

```python

# 8. 早停策略参数
best_val_loss = float('inf')
patience = 20
counter = 0

# 9. 训练模型：自定义训练次数
num_epochs = 500
for epoch in range(num_epochs):
    model.train()
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    # 验证集评估
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            outputs = model(inputs)
            val_loss += criterion(outputs, labels).item() * inputs.size(0)
    val_loss /= len(val_dataset)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), 'model.pth')
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print(f'Early stopping at epoch {epoch+1}')
            break

    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {loss.item():.4f}, Val Loss: {val_loss:.4f}')

# 10. 加载最佳模型
model.load_state_dict(torch.load('model.pth', weights_only = True))

# 11. 模型评估
model.eval()
total_loss = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        total_loss += criterion(outputs, labels).item() * inputs.size(0)

mse = total_loss / len(test_dataset)

print(f'\nMSE on test dataset: {mse:.4f}')
```

```
Epoch [10/500], Train Loss: 417.8493, Val Loss: 545.7247
Epoch [20/500], Train Loss: 348.2777, Val Loss: 461.9928
Epoch [30/500], Train Loss: 365.9896, Val Loss: 386.1222
Epoch [40/500], Train Loss: 420.8316, Val Loss: 315.0311
Epoch [50/500], Train Loss: 207.3795, Val Loss: 231.4717
Epoch [60/500], Train Loss: 244.2944, Val Loss: 179.0083
Epoch [70/500], Train Loss: 193.6992, Val Loss: 133.3739
Epoch [80/500], Train Loss: 136.5826, Val Loss: 88.3605
Epoch [90/500], Train Loss: 106.8433, Val Loss: 60.4388
Epoch [100/500], Train Loss: 29.5513, Val Loss: 36.1021
Epoch [110/500], Train Loss: 33.9615, Val Loss: 20.6541
Epoch [120/500], Train Loss: 21.0119, Val Loss: 16.6938
Epoch [130/500], Train Loss: 15.2447, Val Loss: 10.4564
Epoch [140/500], Train Loss: 23.1641, Val Loss: 9.5385
Epoch [150/500], Train Loss: 61.7226, Val Loss: 8.8601
Epoch [160/500], Train Loss: 14.9008, Val Loss: 8.6332
Epoch [170/500], Train Loss: 15.9442, Val Loss: 8.9074
Epoch [180/500], Train Loss: 34.8720, Val Loss: 7.9747
Early stopping at epoch 186

MSE on test dataset: 6.0042
```