基于神经网络的房价预测建模与优化分析

课程名称： 最优化理论与方法　　**实验日期：** 2025-09-18

班级： placeholder　　**姓名：** placeholder　　**学号：** placeholder

目录

1 问题描述

1.1 任务目标

构建全连接神经网络，预测房价，重点设计基于反向传播算法的连接权优化方法，并分析不同优化算法效果。

1.2 具体要求

1.使用包含 **1 个隐藏层**的神经网络：

```
输入层(根据特征确定) → 隐藏层(确定隐层神经元数, Sigmoid或者ReLU隐函数) → 输出层(1, Linear)
```

2. 实现基于梯度下降及其改进的**反向传播算法**训练网络
3. **禁止使用 TensorFlow/PyTorch 等框架**（仅允许使用 `numpy` ）

2 实验任务

2.1 模型构建与理论推导

首先加载波士顿房价数据集并进行观察：

```python
# 加载数据
boston = fetch_openml(name="boston", version=1)
X = boston.data.to_numpy()
y = boston.target.to_numpy().reshape(-1, 1)

print(X.shape)
print(y.shape)
```

发现共拥有 13 个输入的特征，1 个输出的特征：

```
(506, 13)
(506, 1)
```

于是可以搭建一个有 **13 个输入，1 个输出的神经网络，要求只有一层隐藏层，于是我设定了隐藏层拥有 10 个神经元**。

2.1.1 前向计算

设输入特征为 $X \in \mathbb{R}^{n\times d}$，权重矩阵 $W_1 \in \mathbb{R}^{d\times h}$，偏置 $b_1 \in \mathbb{R}^{1\times h}$，隐藏层激活函数为 $f(\cdot)$，输出层权重 $W_2 \in \mathbb{R}^{h\times 10}$，偏置 $b_2 \in \mathbb{R}$。

前向计算过程如下：

1. 隐藏层线性变换：$Z_1 = XW_1 + b_1$
2. 激活：$A_1 = f(Z_1)$
3. 输出层线性变换：$\hat{y} = A_1W_2 + b_2$

其中，$\hat{y}$ 为模型预测的房价。

2.1.2 损失函数

本实验采用均方误差（Mean Squared Error, MSE）作为损失函数，其定义为：

$$L = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2-1}$$

其中，$\hat{y}_i$ 表示模型预测值，y_i 表示真实值，n 为样本数量。MSE 衡量预测值与真实值之间的平均平方差，值越小表示模型拟合效果越好。

2.1.3 反向传播

1. 损失函数为 $L = \frac{1}{n}\sum(\hat{y}_i - y_i)^2$，对输出层的梯度为

$$\frac{\partial L}{\partial \hat{y}} = \frac{2}{n}(\hat{y} - y) \tag{2-2}$$

2. 输出层权重和偏置的梯度：

$$\frac{\partial L}{\partial W_2} = A_1^T \frac{\partial L}{\partial \hat{y}} \tag{2-3}$$

$$\frac{\partial L}{\partial b_2} = \sum \frac{\partial L}{\partial \hat{y}} \tag{2-4}$$

3. 反向传播到隐藏层：

$$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial \hat{y}} W_2^T \tag{2-5}$$

$$\frac{\partial L}{\partial Z_1} = \frac{\partial L}{\partial A_1} \cdot f'(Z_1) \tag{2-6}$$

4. 隐藏层权重和偏置的梯度：

$$\frac{\partial L}{\partial W_1} = X^T \frac{\partial L}{\partial Z_1} \tag{2-7}$$

$$\frac{\partial L}{\partial b_1} = \sum \frac{\partial L}{\partial Z_1} \tag{2-8}$$

5. 按照学习率更新参数。

2.2 Python 编程代码实现

完整代码实现放在了报告最后（节 4）。

```python
# 前向计算
def forward(self, X):
```

```python
    self.Z1 = X @ self.W1 + self.b1
    self.A1 = self.activation_func(self.Z1)
    self.Z2 = self.A1 @ self.W2 + self.b2
    return self.Z2

# 计算损失
def compute_loss(self, y_pred, y_true):
        return np.mean((y_pred - y_true) ** 2)

# 反向传播，使用简单梯度下降
def backward_with_gd(self, X, y_true, y_pred):
    m = y_true.shape[0]

    dZ2 = (y_pred - y_true) / m  # dLoss/dZ2
    dW2 = self.A1.T @ dZ2
    db2 = np.sum(dZ2, axis=0, keepdims=True)

    dA1 = dZ2 @ self.W2.T
    dZ1 = dA1 * self.activation_func_derivative(self.Z1)
    dW1 = X.T @ dZ1
    db1 = np.sum(dZ1, axis=0, keepdims=True)

    # 更新权重
    self.W2 -= self.lr * dW2
    self.b2 -= self.lr * db2
    self.W1 -= self.lr * dW1
    self.b1 -= self.lr * db1
```

2.3 使用 Adam 算法优化反向传播中的梯度下降

Adam（Adaptive Moment Estimation）是一种自适应学习率优化算法，结合了动量法和 RMSProp 的思想。其核心思想是对每个参数分别维护一阶矩（梯度的指数加权平均）和二阶矩（梯度平方的指数加权平均），并进行偏差校正。简单推导如下：

1. 初始化一阶矩 $m_0 = 0$，二阶矩 $v_0 = 0$，设学习率为 α，一阶和二阶衰减率分别为 β_1, β_2（如 0.9, 0.999），以及极小常数 ε（如10^{-8}）防止除零。
2. 每次迭代 t，计算当前梯度 g_t。
3. 更新一阶矩估计：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2-9}$$

4. 更新二阶矩估计：

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{2-10}$$

5. 进行偏差修正：

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2-11}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2-12}$$

6. 更新参数：

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \tag{2-13}$$

2.3.1 Adam 算法 Python 代码实现

```python
# 反向传播，使用 Adam 优化
def backward_with_adam(self, X, y_true, y_pred, beta1=0.9, beta2=0.999, epsilon=1e-8):
    m = y_true.shape[0]
    t = self.iterations

    # 计算梯度
    dZ2 = (y_pred - y_true) / m
    dW2 = self.A1.T @ dZ2
    db2 = np.sum(dZ2, axis=0, keepdims=True)

    dA1 = dZ2 @ self.W2.T
    dZ1 = dA1 * self.activation_func_derivative(self.Z1)
    dW1 = X.T @ dZ1
    db1 = np.sum(dZ1, axis=0, keepdims=True)

    # 为每个 weight 和 bias 更新动量和参数
    for param, dparam, m_key, v_key in [
        (self.W1, dW1, "mW1", "vW1"),
        (self.b1, db1, "mb1", "vb1"),
        (self.W2, dW2, "mW2", "vW2"),
        (self.b2, db2, "mb2", "vb2"),
    ]:
        self.adam_params[m_key] = beta1 * self.adam_params[m_key] + (1 - beta1) * dparam
        self.adam_params[v_key] = beta2 * self.adam_params[v_key] + (1 - beta2) * (dparam**2)

```

```
        # 更正 bias
        m_corrected = self.adam_params[m_key] / (1 - beta1**t)
        v_corrected = self.adam_params[v_key] / (1 - beta2**t)

        # 更新参数
        param -= self.lr * m_corrected / (np.sqrt(v_corrected) + epsilon)
```

3 结果分析与报告

3.1 结果可视化

3.1.1 损失函数曲线

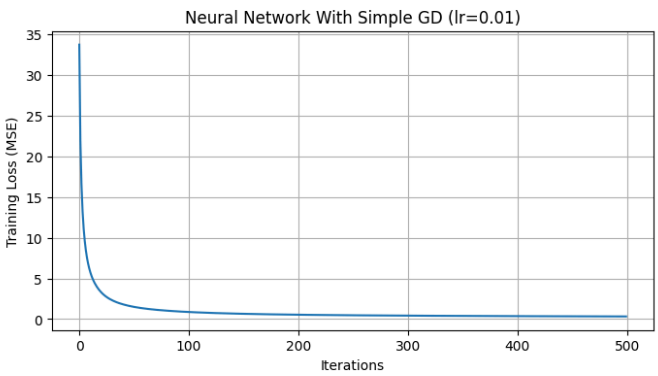

图 3-1 简单梯度下降法

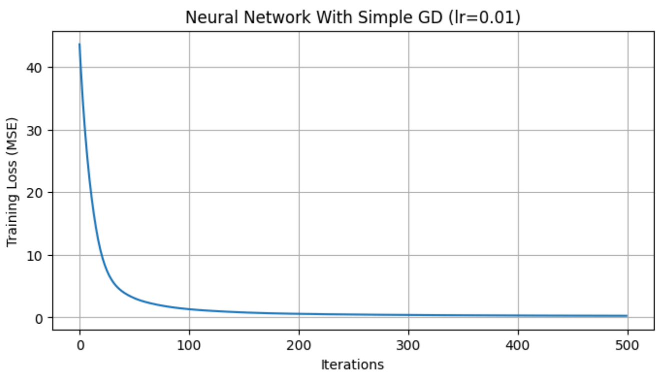

图 3-2 Adam 算法

3.1.2 与实验一至实验六的相关结果的对比分析

在保证 `lr=0.01, iterations = 500` 的情况下：

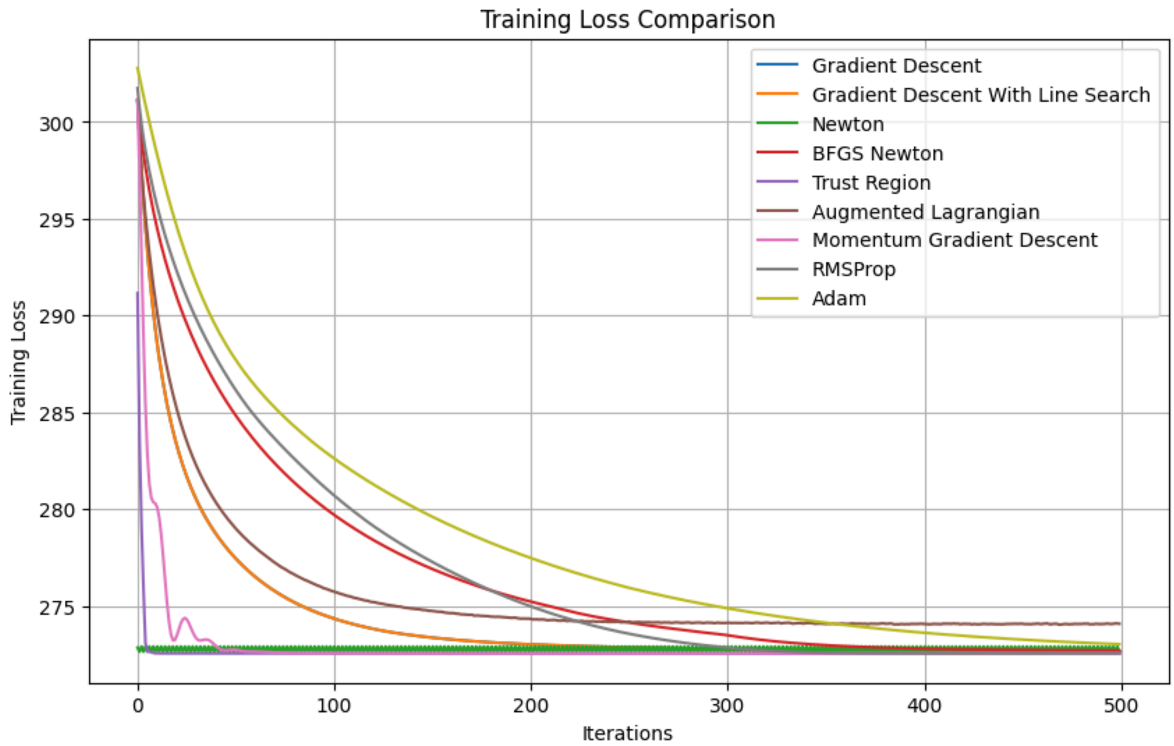

图 3-3 实验一至实验六 Lasso 模型优化

通过对比发现，由于神经网络中考虑了每个神经元具有偏置参数（bias），所以最终神经网络对于波士顿房价问题的 Loss **更低**，同时，使用神经网络可以获得比普通 Lasso 模型**更好的收敛速度**。

3.2 优化分析

3.2.1 ReLU 和 Sigmoid 激活函数的比较

```python
# ReLU 激活函数
def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return np.sign(z)

# Sigmoid 激活函数
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(z):
    s = sigmoid(z)
    return s * (1 - s)
```

3.2.2 学习率设置合理性

当 `lr=0.01` 时:

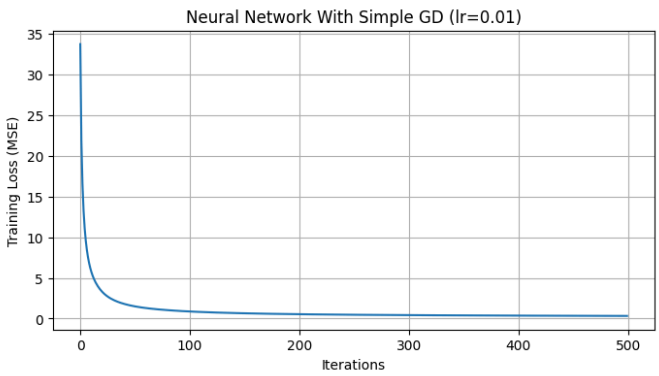

图 3-4 Test Loss (MSE): 39.0493

当 `lr=0.1` 时：

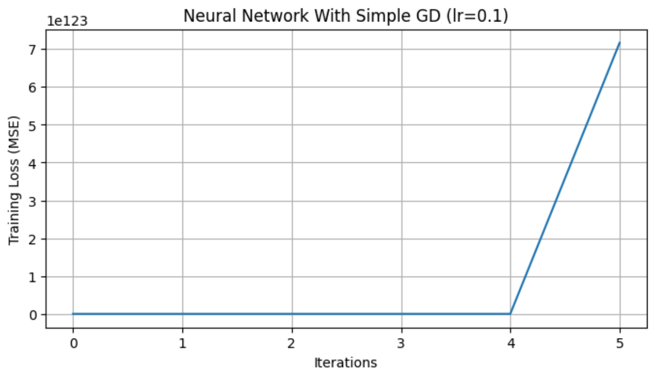

图 3-5 训练出错，模型无法正常收敛

当 `lr=0.001` 时：

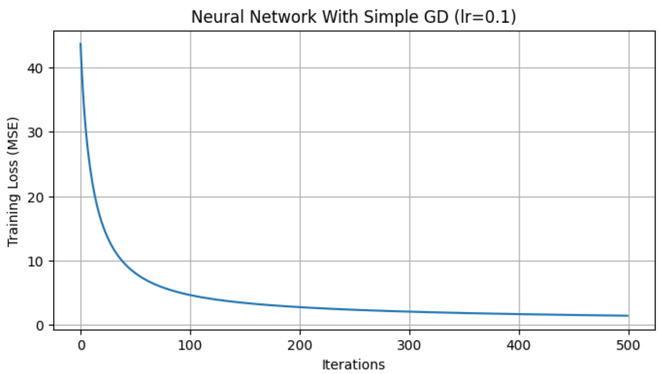

图 3-6 Test Loss (MSE): 132.5201

由此得出，当 `lr` 设置的过大的时候，**模型无法正常收敛**，当 `lr` 设置的过小，会**降低模型收敛速度**，而且**在训练轮数过低的情况下会影响模型性能**。

3.2.3 其他改进方案

可以考虑继续增加隐藏层数量，进一步优化模型的拟合能力。

4 完整代码实现

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 固定随机数种子
np.random.seed(24)

# 加载数据
boston = fetch_openml(name="boston", version=1)
X = boston.data.to_numpy()
y = boston.target.to_numpy().reshape(-1, 1)

print(X.shape)
print(y.shape)

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 数据标准化
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_train = scaler_X.fit_transform(X_train)
X_test = scaler_X.transform(X_test)
y_train = scaler_y.fit_transform(y_train)
y_test = scaler_y.transform(y_test)


# ReLU 激活函数
def relu(z):
    return np.maximum(0, z)


def relu_derivative(z):
    return np.sign(z)
```

```python


# Sigmoid 激活函数
def sigmoid(z):
    return 1 / (1 + np.exp(-z))


def sigmoid_derivative(z):
    s = sigmoid(z)
    return s * (1 - s)


class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size,
    activation_func, activation_func_derivative, lr=0.01):
        # 权重初始化
        self.W1 = np.random.randn(input_size, hidden_size)
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size)
        self.b2 = np.zeros((1, output_size))
        self.lr = lr

        # 设定激活函数及其导数
        self.activation_func = activation_func
        self.activation_func_derivative = activation_func_derivative

        # 设定反向传播方法
        self.backward = self.backward_with_gd

        # 用于 Adam 的参数
        self.adam_params = {
            "mW1": np.zeros_like(self.W1),
            "vW1": np.zeros_like(self.W1),
            "mb1": np.zeros_like(self.b1),
            "vb1": np.zeros_like(self.b1),
            "mW2": np.zeros_like(self.W2),
            "vW2": np.zeros_like(self.W2),
            "mb2": np.zeros_like(self.b2),
            "vb2": np.zeros_like(self.b2),
        }

        # 记录训练轮数
        self.iterations = 0
```

```python

    # 计算损失
    def compute_loss(self, y_pred, y_true):
        return np.mean((y_pred - y_true) ** 2)

    # 前向计算
    def forward(self, X):
        self.Z1 = X @ self.W1 + self.b1
        self.A1 = self.activation_func(self.Z1)
        self.Z2 = self.A1 @ self.W2 + self.b2
        return self.Z2

    # 反向传播，使用简单梯度下降
    def backward_with_gd(self, X, y_true, y_pred):
        m = y_true.shape[0]

        dZ2 = (y_pred - y_true) / m  # dLoss/dZ2
        dW2 = self.A1.T @ dZ2
        db2 = np.sum(dZ2, axis=0, keepdims=True)

        dA1 = dZ2 @ self.W2.T
        dZ1 = dA1 * self.activation_func_derivative(self.Z1)
        dW1 = X.T @ dZ1
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # 更新权重
        self.W2 -= self.lr * dW2
        self.b2 -= self.lr * db2
        self.W1 -= self.lr * dW1
        self.b1 -= self.lr * db1

    # 反向传播，使用 Adam 优化
    def backward_with_adam(self, X, y_true, y_pred, beta1=0.9, beta2=0.999,
    epsilon=1e-8):
        m = y_true.shape[0]
        t = self.iterations

        # 计算梯度
        dZ2 = (y_pred - y_true) / m
        dW2 = self.A1.T @ dZ2
        db2 = np.sum(dZ2, axis=0, keepdims=True)

        dA1 = dZ2 @ self.W2.T
```

```
        dZ1 = dA1 * self.activation_func_derivative(self.Z1)
        dW1 = X.T @ dZ1
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # 为每个 weight 和 bias 更新动量和参数
        for param, dparam, m_key, v_key in [
            (self.W1, dW1, "mW1", "vW1"),
            (self.b1, db1, "mb1", "vb1"),
            (self.W2, dW2, "mW2", "vW2"),
            (self.b2, db2, "mb2", "vb2"),
        ]:
            self.adam_params[m_key] = beta1 * self.adam_params[m_key] + (1 -
            beta1) * dparam
            self.adam_params[v_key] = beta2 * self.adam_params[v_key] + (1 -
            beta2) * (dparam**2)

            # 更正 bias
            m_corrected = self.adam_params[m_key] / (1 - beta1**t)
            v_corrected = self.adam_params[v_key] / (1 - beta2**t)

            # 更新参数
            param -= self.lr * m_corrected / (np.sqrt(v_corrected) +
            epsilon)

    # 训练
    def train(self, X, y, epochs=1000):
        losses = []
        for _ in range(epochs):
            self.iterations += 1
            y_pred = self.forward(X)
            loss = self.compute_loss(y_pred, y)
            losses.append(loss)
            self.backward(X, y, y_pred)
        return losses


# 初始化网络
nn = NeuralNetwork(
    input_size=13,
    hidden_size=10,
    output_size=1,
    activation_func=relu,
    activation_func_derivative=relu,
```

```python
    lr=0.01,
)

# 训练模型
losses = nn.train(X_train, y_train, epochs=500)

# 预测
y_pred = nn.forward(X_test)

# 反标准化并评估
y_pred_inv = scaler_y.inverse_transform(y_pred)
y_test_inv = scaler_y.inverse_transform(y_test)

# 计算均方误差
mse = np.mean((y_pred_inv - y_test_inv) ** 2)
print(f"\nTest Loss (MSE): {mse:.4f}")

plt.figure(figsize=(8, 4))
plt.title("Neural Network With Simple GD (lr=0.1)")
plt.xlabel("Iterations")
plt.ylabel("Training Loss (MSE)")
plt.plot(range(len(losses)), losses)
plt.grid()
plt.show()
```