

# 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: 2025-07-11班级: placeholder姓名: placeholder学号: placeholder

## 目录

1 问题描述 .....	1
1.1 任务目标 .....	1
1.2 具体要求 .....	2
2 实验任务 .....	2
2.1 模型构建与理论推导 .....	2
2.1.1 前向计算 .....	2
2.1.2 损失函数 .....	3
2.1.3 反向传播 .....	3
2.2 Python 编程代码实现 .....	3
2.3 使用 Adam 算法优化反向传播中的梯度下降 .....	4
2.3.1 Adam 算法 Python 代码实现 .....	5
3 结果分析与报告 .....	6
3.1 结果可视化 .....	6
3.1.1 损失函数曲线 .....	6
3.1.2 与实验一至实验六的相关结果的对比分析 .....	6
3.2 优化分析 .....	7
3.2.1 ReLU 和 Sigmoid 激活函数的比较 .....	7
3.2.2 学习率设置合理性 .....	8
3.2.3 其他改进方案 .....	9
4 完整代码实现 .....	9

## 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：

```
1 # 加载数据
2 boston = fetch_openml(name="boston", version=1)
3 X = boston.data.to_numpy()
4 y = boston.target.to_numpy().reshape(-1, 1)
5
6 print(X.shape)
7 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：

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

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

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

2. 输出层权重和偏置的梯度：

$$\frac{\partial L}{\partial W_2} = A_1^T \frac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial b_2} = \sum \frac{\partial L}{\partial \hat{y}}$$

3. 反向传播到隐藏层：

$$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial \hat{y}} W_2^T$$

$$\frac{\partial L}{\partial Z_1} = \frac{\partial L}{\partial A_1} \cdot f'(Z_1)$$

4. 隐藏层权重和偏置的梯度：

$$\frac{\partial L}{\partial W_1} = X^T \frac{\partial L}{\partial Z_1}$$

$$\frac{\partial L}{\partial b_1} = \sum \frac{\partial L}{\partial Z_1}$$

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）。

```
1 # 前向计算
2 def forward(self, x):
```

```

3     self.Z1 = X @ self.W1 + self.b1
4     self.A1 = self.activation_func(self.Z1)
5     self.Z2 = self.A1 @ self.W2 + self.b2
6     return self.Z2
7
8 # 计算损失
9 def compute_loss(self, y_pred, y_true):
10     return np.mean((y_pred - y_true) ** 2)
11
12 # 反向传播, 使用简单梯度下降
13 def backward_with_gd(self, X, y_true, y_pred):
14     m = y_true.shape[0]
15
16     dZ2 = (y_pred - y_true) / m # dLoss/dZ2
17     dW2 = self.A1.T @ dZ2
18     db2 = np.sum(dZ2, axis=0, keepdims=True)
19
20     dA1 = dZ2 @ self.W2.T
21     dZ1 = dA1 * self.activation_func_derivative(self.Z1)
22     dW1 = X.T @ dZ1
23     db1 = np.sum(dZ1, axis=0, keepdims=True)
24
25     # 更新权重
26     self.W2 -= self.lr * dW2
27     self.b2 -= self.lr * db2
28     self.W1 -= self.lr * dW1
29     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  $\alpha$ , 一阶和二阶衰减率分别为  $\beta_1, \beta_2$  (如 0.9, 0.999), 以及极小常数  $\epsilon$  (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:

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

4. 更新二阶矩估计:

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

5. 进行偏差修正:

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

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

6. 更新参数:

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

### 2.3.1 Adam 算法 Python 代码实现

```

1  # 反向传播, 使用 Adam 优化
2  def backward_with_adam(self, X, y_true, y_pred, beta1=0.9, beta2=0.999,
    epsilon=1e-8):
3      m = y_true.shape[0]
4      t = self.iterations
5
6      # 计算梯度
7      dZ2 = (y_pred - y_true) / m
8      dW2 = self.A1.T @ dZ2
9      db2 = np.sum(dZ2, axis=0, keepdims=True)
10
11     dA1 = dZ2 @ self.W2.T
12     dZ1 = dA1 * self.activation_func_derivative(self.Z1)
13     dW1 = X.T @ dZ1
14     db1 = np.sum(dZ1, axis=0, keepdims=True)
15
16     # 为每个 weight 和 bias 更新动量和参数
17     for param, dparam, m_key, v_key in [
18         (self.W1, dW1, "mW1", "vW1"),
19         (self.b1, db1, "mb1", "vb1"),
20         (self.W2, dW2, "mW2", "vW2"),
21         (self.b2, db2, "mb2", "vb2"),
22     ]:
23         self.adam_params[m_key] = beta1 * self.adam_params[m_key] + (1 -
            beta1) * dparam
24         self.adam_params[v_key] = beta2 * self.adam_params[v_key] + (1 -
            beta2) * (dparam**2)
25

```

```

26         # 更正 bias
27         m_corrected = self.adam_params[m_key] / (1 - beta1**t)
28         v_corrected = self.adam_params[v_key] / (1 - beta2**t)
29
30         # 更新参数
31         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

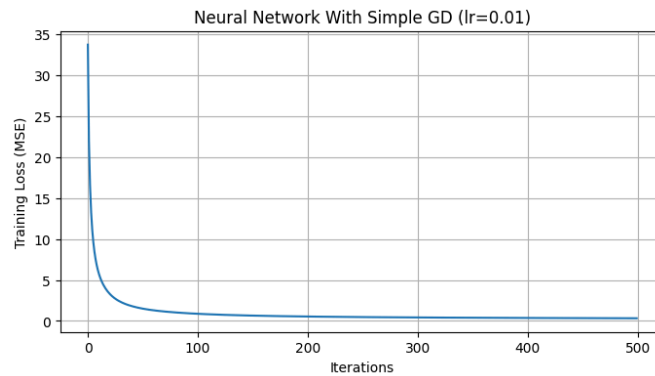


Figure 1: 简单梯度下降法

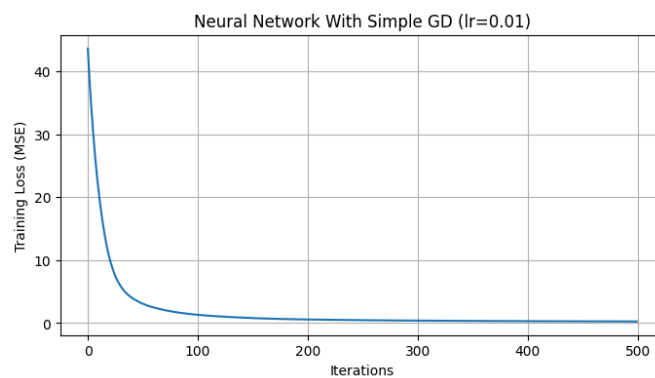


Figure 2: Adam 算法

#### 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:

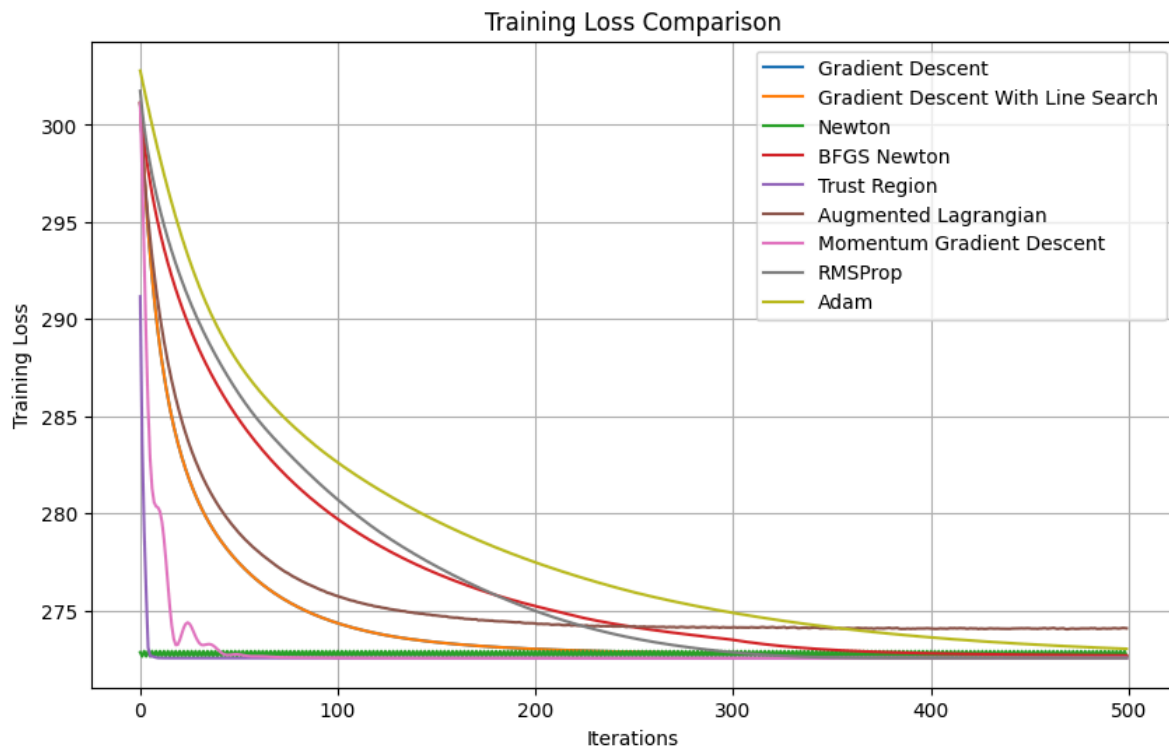


Figure 3: 实验一至实验六 Lasso 模型优化

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

```

1 # ReLU 激活函数
2 def relu(z):
3     return np.maximum(0, z)
4
5 def relu_derivative(z):
6     return np.sign(z)
7
8 # Sigmoid 激活函数
9 def sigmoid(z):
10    return 1 / (1 + np.exp(-z))
11
12 def sigmoid_derivative(z):
13    s = sigmoid(z)
14    return s * (1 - s)

```

### 3.2.2 学习率设置合理性

当  $lr=0.01$  时:

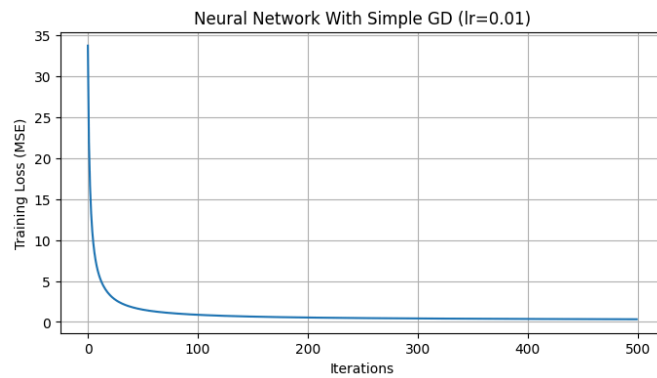


Figure 4: Test Loss (MSE): 39.0493

当  $lr=0.1$  时:

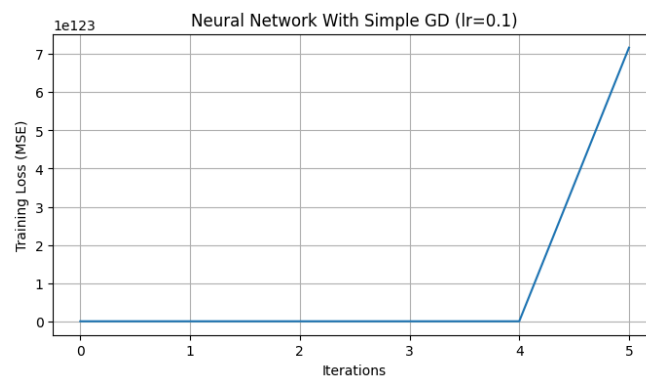


Figure 5: 训练出错，模型无法正常收敛

当  $lr=0.001$  时:

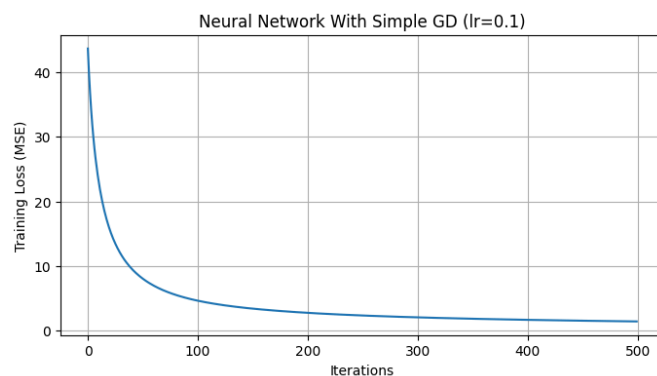


Figure 6: Test Loss (MSE): 132.5201

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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.datasets import fetch_openml
4 from sklearn.model_selection import train_test_split
5 from sklearn.preprocessing import StandardScaler
6
7 # 固定随机数种子
8 np.random.seed(24)
9
10 # 加载数据
11 boston = fetch_openml(name="boston", version=1)
12 X = boston.data.to_numpy()
13 y = boston.target.to_numpy().reshape(-1, 1)
14
15 print(X.shape)
16 print(y.shape)
17
18 # 划分训练集和测试集
19 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
20 random_state=42)
21
22 # 数据标准化
23 scaler_X = StandardScaler()
24 scaler_y = StandardScaler()
25 X_train = scaler_X.fit_transform(X_train)
26 X_test = scaler_X.transform(X_test)
27 y_train = scaler_y.fit_transform(y_train)
28 y_test = scaler_y.transform(y_test)
29
30 # ReLU 激活函数
31 def relu(z):
32     return np.maximum(0, z)
33
34
35 def relu_derivative(z):
36     return np.sign(z)
```

```

37
38
39 # Sigmoid 激活函数
40 def sigmoid(z):
41     return 1 / (1 + np.exp(-z))
42
43
44 def sigmoid_derivative(z):
45     s = sigmoid(z)
46     return s * (1 - s)
47
48
49 class NeuralNetwork:
50     def __init__(self, input_size, hidden_size, output_size,
51         activation_func, activation_func_derivative, lr=0.01):
52         # 权重初始化
53         self.W1 = np.random.randn(input_size, hidden_size)
54         self.b1 = np.zeros((1, hidden_size))
55         self.W2 = np.random.randn(hidden_size, output_size)
56         self.b2 = np.zeros((1, output_size))
57         self.lr = lr
58
59         # 设定激活函数及其导数
60         self.activation_func = activation_func
61         self.activation_func_derivative = activation_func_derivative
62
63         # 设定反向传播方法
64         self.backward = self.backward_with_gd
65
66         # 用于 Adam 的参数
67         self.adam_params = {
68             "mW1": np.zeros_like(self.W1),
69             "vW1": np.zeros_like(self.W1),
70             "mb1": np.zeros_like(self.b1),
71             "vb1": np.zeros_like(self.b1),
72             "mW2": np.zeros_like(self.W2),
73             "vW2": np.zeros_like(self.W2),
74             "mb2": np.zeros_like(self.b2),
75             "vb2": np.zeros_like(self.b2),
76         }
77
78         # 记录训练轮数
79         self.iterations = 0

```

```
79
80 # 计算损失
81 def compute_loss(self, y_pred, y_true):
82     return np.mean((y_pred - y_true) ** 2)
83
84 # 前向计算
85 def forward(self, X):
86     self.Z1 = X @ self.W1 + self.b1
87     self.A1 = self.activation_func(self.Z1)
88     self.Z2 = self.A1 @ self.W2 + self.b2
89     return self.Z2
90
91 # 反向传播, 使用简单梯度下降
92 def backward_with_gd(self, X, y_true, y_pred):
93     m = y_true.shape[0]
94
95     dZ2 = (y_pred - y_true) / m # dLoss/dZ2
96     dW2 = self.A1.T @ dZ2
97     db2 = np.sum(dZ2, axis=0, keepdims=True)
98
99     dA1 = dZ2 @ self.W2.T
100    dZ1 = dA1 * self.activation_func_derivative(self.Z1)
101    dW1 = X.T @ dZ1
102    db1 = np.sum(dZ1, axis=0, keepdims=True)
103
104    # 更新权重
105    self.W2 -= self.lr * dW2
106    self.b2 -= self.lr * db2
107    self.W1 -= self.lr * dW1
108    self.b1 -= self.lr * db1
109
110 # 反向传播, 使用 Adam 优化
111 def backward_with_adam(self, X, y_true, y_pred, beta1=0.9, beta2=0.999,
112     epsilon=1e-8):
113     m = y_true.shape[0]
114     t = self.iterations
115
116     # 计算梯度
117     dZ2 = (y_pred - y_true) / m
118     dW2 = self.A1.T @ dZ2
119     db2 = np.sum(dZ2, axis=0, keepdims=True)
120
121     dA1 = dZ2 @ self.W2.T
```

```

121     dZ1 = dA1 * self.activation_func_derivative(self.Z1)
122     dW1 = X.T @ dZ1
123     db1 = np.sum(dZ1, axis=0, keepdims=True)
124
125     # 为每个 weight 和 bias 更新动量和参数
126     for param, dparam, m_key, v_key in [
127         (self.W1, dW1, "mW1", "vW1"),
128         (self.b1, db1, "mb1", "vb1"),
129         (self.W2, dW2, "mW2", "vW2"),
130         (self.b2, db2, "mb2", "vb2"),
131     ]:
132         self.adam_params[m_key] = beta1 * self.adam_params[m_key] + (1 -
133             beta1) * dparam
134         self.adam_params[v_key] = beta2 * self.adam_params[v_key] + (1 -
135             beta2) * (dparam**2)
136
137         # 更正 bias
138         m_corrected = self.adam_params[m_key] / (1 - beta1**t)
139         v_corrected = self.adam_params[v_key] / (1 - beta2**t)
140
141         # 更新参数
142         param -= self.lr * m_corrected / (np.sqrt(v_corrected) +
143             epsilon)
144
145     # 训练
146     def train(self, X, y, epochs=1000):
147         losses = []
148         for _ in range(epochs):
149             self.iterations += 1
150             y_pred = self.forward(X)
151             loss = self.compute_loss(y_pred, y)
152             losses.append(loss)
153             self.backward(X, y, y_pred)
154         return losses
155
156     # 初始化网络
157     nn = NeuralNetwork(
158         input_size=13,
159         hidden_size=10,
160         output_size=1,
161         activation_func=relu,
162         activation_func_derivative=relu,

```

```
161     lr=0.01,
162 )
163
164 # 训练模型
165 losses = nn.train(X_train, y_train, epochs=500)
166
167 # 预测
168 y_pred = nn.forward(X_test)
169
170 # 反标准化并评估
171 y_pred_inv = scaler_y.inverse_transform(y_pred)
172 y_test_inv = scaler_y.inverse_transform(y_test)
173
174 # 计算均方误差
175 mse = np.mean((y_pred_inv - y_test_inv) ** 2)
176 print(f"\nTest Loss (MSE): {mse:.4f}")
177
178 plt.figure(figsize=(8, 4))
179 plt.title("Neural Network With Simple GD (lr=0.1)")
180 plt.xlabel("Iterations")
181 plt.ylabel("Training Loss (MSE)")
182 plt.plot(range(len(losses)), losses)
183 plt.grid()
184 plt.show()
```