NumPy 实现 MLP 实验

1. 实验基础

1.1 实验目标

- 深入理解神经网络核心机制：**前向传播、反向传播、梯度下降**的数学原理与工程实现
- 掌握纯 NumPy 构建多层感知机（MLP）的完整流程（层设计、激活函数、损失函数、优化器）
- 学会数据预处理、模型训练、性能评估的标准化方法
- 理解超参数（学习率、批次大小、网络深度/宽度）对模型性能的影响机制
- 具备超参数调优、模型改进的实践能力

1.2 实验原理（补充）

核心概念快速梳理

- **前向传播**：输入数据通过网络层（权重矩阵乘法+偏置加法+激活函数）计算输出预测值的过程，公式为：$z = Wx + b$，$a = \sigma(z)$（σ为激活函数）
- **反向传播**：基于链式法则，从损失函数出发，反向计算各层参数（W, b）的梯度，为参数更新提供依据
- **梯度下降**：通过优化器（如 Adam、SGD）利用梯度调整参数，最小化损失函数，公式为：$W = W - \eta \cdot \nabla_W L$（$\eta$为学习率，$\nabla_W L$为损失对$W$的梯度）
- **MLP 适用于回归任务**：输出层使用 Linear 激活函数，损失函数采用均方误差（MSE），评估指标包括 MSE、MAE、R^2

1.3 实验内容

使用纯 NumPy 实现端到端的 MLP，解决 Boston Housing 房价回归问题，核心任务：

1. 实现神经网络核心组件（全连接层、激活函数、损失函数、优化器）
2. 构建可灵活配置的 MLP 模型（支持动态添加层、切换损失/优化器）
3. 解决 Boston Housing 数据集加载（替代方案）与标准化预处理

4. 完成模型训练、验证、测试全流程
5. 可视化训练过程与预测结果，深入分析模型性能
6. 超参数调优与模型改进实践

1.4 技术栈

- Python 3.8~3.10（兼容性最佳）
- NumPy 1.21~1.26（数组运算核心）
- Pandas 1.3+（数据读取与预处理辅助）
- Scikit-learn 0.24~1.3（数据集加载、分割、标准化、指标计算）
- Matplotlib 3.4+（训练曲线、预测结果可视化）

注意：Boston Housing 数据集因伦理问题已从 Scikit-learn 1.2+中移除，下文提供替代加载方案

2. 环境搭建

2.1 安装依赖

```
# 1. 创建虚拟环境（推荐，避免版本冲突）
python -m venv mlp-env
# Windows 激活：mlp-env\Scripts\activate
# Mac/Linux 激活：source mlp-env/bin/activate

# 2. 安装依赖（指定兼容版本）
pip install numpy==1.24.3 pandas==1.5.3 scikit-learn==1.2.2
matplotlib==3.7.1
```

2.2 项目结构

```
Plain Text

lab1-mlp/
├── mlp/                          # 核心模块（可直接复用）
│   ├── __init__.py              # 模块暴露接口（如 from mlp import MLP,
```

```
Dense）
│   ├── activations.py      # 激活函数（ReLU/Tanh/Sigmoid/Linear）
│   ├── datasets.py         # 数据集加载（Boston 替代方案+预处理）
│   ├── layers.py           # 神经网络层（全连接层 Dense）
│   ├── losses.py           # 损失函数（MSE）
│   ├── model.py            # MLP 模型核心（训练/预测/评估）
│   └── optimizers.py       # 优化器（SGD/Adam）
├── demo.py                 # 基础演示脚本（快速运行）
├── demo_advanced.py        # 高级演示（超参数调优+正则化）
├── requirements.txt        # 依赖列表（复制上述安装命令）
├── README.md               # 项目说明
└── results/                # 结果保存目录（自动生成，存储可视化图）
```

2.3 TRAE IDE 开发环境配置

TRAE IDE 是轻量高效的 Python 开发工具，支持代码编辑、环境管理、脚本运行等全流程操作，以下是适配本实验的完整配置步骤：

2.3.1 TRAE IDE 安装与启动

1. 下载与安装：访问 TRAE IDE 官方网站（根据操作系统选择对应版本，Windows/macOS/Linux 均支持），按照向导完成安装（建议勾选“添加到系统 PATH”选项，方便快速启动）。

2. 首次启动配置：打开 TRAE IDE 后，在欢迎界面选择“自定义工作区”，建议选择空文件夹作为工作区（如 `D:/trae-workspace` 或`~/trae-workspace`），避免中文路径导致兼容性问题。

2.3.2 在 TRAE IDE 中创建实验项目

1. 创建项目：点击顶部菜单栏「File」→「New」→「Project」，在弹出的窗口中选择「Python Project」，项目名称填写 `lab1-mlp`，点击「Create」完成创建（默认会生成基础项目结构）。

2. 创建目录结构：在项目根目录（lab1-mlp）上右键，选择「New」→「Directory」，依次创建 `mlp` 和 `results` 文件夹；然后在 `mlp` 文件夹上右键，创建 `__init__.py`、`activations.py` 等 7 个核心模块文件（参考 2.2 节项目结构）。

3. 创建脚本文件：在项目根目录右键，创建 `demo.py`、`demo_advanced.py` 和 `requirements.txt` 文件，完成后项目结构应与 2.2 节完全一致。

2.3.3 配置 Python 虚拟环境

1. 打开终端：点击 TRAE IDE 底部「Terminal」标签，打开内置终端（默认关联项目根目录）。

2. 创建虚拟环境：在终端中执行以下命令（与 2.1 节命令一致，TRAE IDE 终端支持跨系统命令适配）：
```
# 创建虚拟环境
python -m venv mlp-env
# Windows 激活
mlp-env\Scripts\activate
# Mac/Linux 激活
source mlp-env/bin/activate
```

3. 验证环境激活：激活成功后，终端提示符前会显示`(mlp-env)`，表示当前已进入虚拟环境。

4. 安装依赖：在虚拟环境中执行依赖安装命令，TRAE IDE 会自动识别 pip 包管理器并完成安装：`pip install numpy==1.24.3 pandas==1.5.3 scikit-learn==1.2.2 matplotlib==3.7.1`

5. （可选）通过 requirements.txt 安装：将 2.1 节的依赖命令复制到`requirements.txt`中，在终端执行`pip install -r requirements.txt`，可快速完成依赖批量安装。

2.3.4 配置 Python 解释器（确保环境关联）

1. 打开解释器配置：点击顶部菜单栏「File」→「Settings」（Windows）/「Preferences」（macOS），在左侧导航栏选择「Project: lab1-mlp」→「Python Interpreter」。

2. 添加虚拟环境解释器：点击右上角「Add Interpreter」→「Existing environment」，在弹出的窗口中点击「...」，导航到项目根目录下的`mlp-env/Scripts/python.exe`（Windows）或`mlp-env/bin/python`（Mac/Linux），选中后点击「OK」完成关联。

3. 验证配置：配置完成后，在「Python Interpreter」页面会显示当前使用的解释器路径为虚拟环境下的 Python，确保无报错提示。

2.3.5 在 TRAE IDE 中运行代码与查看结果

1. 运行 demo.py 脚本：打开`demo.py`文件，点击代码编辑区右上角的「Run」按钮（绿色三角形图标），或右键选择「Run 'demo'」。

2. 查看运行日志：TRAE IDE 底部会弹出「Run」标签页，显示脚本运行的完整日志（与 2.1 节命令行输出一致），包括数据集加载信息、训练过程日志、模型评估结果等。

3. 查看可视化结果：脚本运行完成后，会自动生成`results`目录并保存训练曲线和

预测结果图，在 TRAE IDE 左侧「Project」面板中展开 `results` 文件夹，右键点击图片文件选择「Open With」→「Image Viewer」，可直接在 IDE 中查看图片。

4. 调试功能使用（可选）：若需调试代码，可在关键行（如模型训练、前向传播等步骤）左侧点击设置断点，然后点击「Debug」按钮（红色甲壳虫图标），TRAE IDE 会进入调试模式，支持单步执行、变量查看等调试操作，方便定位代码问题。

2.3.6 TRAE IDE 实用技巧（提升开发效率）

- 代码补全：TRAE IDE 支持 Python 代码智能补全，输入代码时会自动提示模块、函数、变量等，按「Tab」键可快速补全，减少代码输入错误。
- 语法检查：实时检测代码语法错误，错误行下方会显示红色波浪线，鼠标悬停可查看错误原因及修复建议（如缺少导入、缩进错误等）。
- 文件导航：左侧「Project」面板支持快速定位文件，双击文件可在编辑区打开；使用「Ctrl+P」（Windows）/「Cmd+P」（macOS）快捷键，输入文件名可快速搜索并打开目标文件。
- 终端快捷操作：在编辑区右键选择「Open in Terminal」，可快速打开当前文件所在目录的终端；终端支持复制、粘贴、清屏（输入 `cls`（Windows）/`clear`（Mac/Linux））等操作。

常见问题解决

- 虚拟环境激活失败：检查 Python 是否添加到系统 PATH，或重新创建虚拟环境（删除 `mlp-env` 文件夹后重新执行创建命令）。
- 解释器配置失败：确保选择的是虚拟环境下的 Python 解释器，而非系统默认 Python；避免路径包含中文或特殊字符。
- 脚本运行无输出：检查是否已激活虚拟环境并配置正确的解释器，或重启 TRAE IDE 重试。
- 图片无法查看：确保脚本已成功运行并生成 `results` 目录，若未生成，查看「Run」标签页的错误日志，定位数据集加载或代码运行问题。

```
lab1-mlp/
├── mlp/                          # 核心模块（可直接复用）
│   ├── __init__.py           # 模块暴露接口（如 from mlp import MLP, Dense）
│   ├── activations.py        # 激活函数（ReLU/Tanh/Sigmoid/Linear）
│   ├── datasets.py           # 数据集加载（Boston 替代方案+预处理）
```

```
│   ├── layers.py            # 神经网络层（全连接层 Dense）
│   ├── losses.py            # 损失函数（MSE）
│   ├── model.py             # MLP 模型核心（训练/预测/评估）
│   └── optimizers.py        # 优化器（SGD/Adam）
├── demo.py                  # 基础演示脚本（快速运行）
├── demo_advanced.py         # 高级演示（超参数调优+正则化）
├── requirements.txt         # 依赖列表（复制上述安装命令）
├── README.md                # 项目说明
└── results/                 # 结果保存目录（自动生成，存储可视化图）
```

3. 核心代码实现

3.1 mlp/layers.py

```
import numpy as np

class Layer:
    """层基类：所有层需实现 forward/backward 方法"""
    def forward(self, x):
        raise NotImplementedError("子类必须实现 forward 方法")

    def backward(self, grad_output):
        raise NotImplementedError("子类必须实现 backward 方法")

    def update_params(self, learning_rate):
        """无参数层（如激活函数）无需实现"""
        pass

class Dense(Layer):
    """全连接层：y = W @ x + b"""
    def __init__(self, input_dim, output_dim,
```

```
weight_initializer='he'):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.weight_initializer = weight_initializer

        # 权重初始化（关键：影响模型收敛速度）
        if weight_initializer == 'he':
            # He 初始化（适合 ReLU 激活）：W ~ N(0, 2/input_dim)
            self.weights = np.random.randn(input_dim, output_dim)
* np.sqrt(2 / input_dim)
        elif weight_initializer == 'xavier':
            # Xavier 初始化（适合 Tanh/Sigmoid）：W ~ N(0,
1/(input_dim+output_dim))
            self.weights = np.random.randn(input_dim, output_dim)
* np.sqrt(1 / (input_dim + output_dim))
        else:
            raise ValueError("仅支持 he/xavier 初始化")

        # 偏置初始化（通常为 0）
        self.biases = np.zeros((1, output_dim))

        # 反向传播需用到的中间变量
        self.input = None
        self.grad_weights = None  # 权重梯度
        self.grad_biases = None   # 偏置梯度

    def forward(self, x):
        """前向传播：x.shape=(batch_size, input_dim) →
output.shape=(batch_size, output_dim)"""
        self.input = x  # 保存输入，用于反向传播
        return np.dot(x, self.weights) + self.biases
```

```
    def backward(self, grad_output):
        """反向传播：计算梯度并返回输入的梯度"""
        # grad_output.shape=(batch_size, output_dim)
        self.grad_weights = np.dot(self.input.T, grad_output) / self.input.shape[0]  # 平均梯度（避免批次大小影响）
        self.grad_biases = np.mean(grad_output, axis=0, keepdims=True)  # 偏置梯度（按批次平均）
        grad_input = np.dot(grad_output, self.weights.T)  # 输入的梯度（传给前一层）
        return grad_input

    def update_params(self, learning_rate):
        """直接用学习率更新参数（适用于 SGD）"""
        self.weights -= learning_rate * self.grad_weights
        self.biases -= learning_rate * self.grad_biases
```

3.2 mlp/activations.py

```
import numpy as np
from .layers import Layer

class Activation(Layer):
    """激活函数基类：保存输入用于反向传播"""
    def __init__(self):
        self.input = None

class ReLU(Activation):
    """ReLU 激活函数：f(x) = max(0, x)（隐藏层首选）"""
    def forward(self, x):
        self.input = x
        return np.maximum(0, x)

    def backward(self, grad_output):
```

```
        """梯度：x>0 时为 1，否则为 0"""
        grad_x = np.where(self.input > 0, 1, 0)
        return grad_x * grad_output

class Tanh(Activation):
    """Tanh 激活函数：f(x) = (e^x - e^(-x))/(e^x + e^(-x))"""
    def forward(self, x):
        self.input = x
        return np.tanh(x)

    def backward(self, grad_output):
        """梯度：1 - tanh²(x)"""
        return grad_output * (1 - np.tanh(self.input)**2)

class Sigmoid(Activation):
    """Sigmoid 激活函数：f(x) = 1/(1+e^(-x))（二分类输出层）"""
    def forward(self, x):
        self.input = x
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))  # 避免指数
溢出

    def backward(self, grad_output):
        """梯度：sigmoid(x) * (1 - sigmoid(x))"""
        sig = self.forward(self.input)
        return grad_output * sig * (1 - sig)

class Linear(Activation):
    """线性激活函数：f(x) = x（回归任务输出层）"""
    def forward(self, x):
        self.input = x
        return x
```

```
    def backward(self, grad_output):
        """梯度：1（直接传递梯度）"""
        return grad_output
```

3.3 mlp/losses.py

```
import numpy as np

class Loss:
    """损失函数基类"""
    def forward(self, y_pred, y_true):
        raise NotImplementedError("子类必须实现 forward 方法")

    def backward(self, y_pred, y_true):
        raise NotImplementedError("子类必须实现 backward 方法")

class MSE(Loss):
    """均方误差损失：L = (1/N) * Σ(y_pred - y_true)²（回归任务）"""
    def forward(self, y_pred, y_true):
        # 确保 y_pred 和 y_true 形状一致（(batch_size, 1)）
        y_pred = y_pred.reshape(-1, 1)
        y_true = y_true.reshape(-1, 1)
        return np.mean((y_pred - y_true)**2)

    def backward(self, y_pred, y_true):
        # 梯度：2*(y_pred - y_true)/N
        y_pred = y_pred.reshape(-1, 1)
        y_true = y_true.reshape(-1, 1)
        return 2 * (y_pred - y_true) / y_pred.shape[0]
```

3.4 mlp/optimizers.py

```
import numpy as np


class Optimizer:
    """优化器基类：更新层参数"""
    def update(self, layer):
        raise NotImplementedError("子类必须实现 update 方法")


class SGD(Optimizer):
    """随机梯度下降：W = W - lr * grad + momentum"""
    def __init__(self, learning_rate=0.01, momentum=0.0):
        self.learning_rate = learning_rate
        self.momentum = momentum
        # 保存动量（避免每次创建新变量）
        self.v_w = {}  # 权重动量：key=layer_id, value=动量值
        self.v_b = {}  # 偏置动量


    def update(self, layer):
        # 仅更新有权重和偏置的层（如 Dense 层）
        if not hasattr(layer, 'weights') or not hasattr(layer,
'biases'):
            return


        layer_id = id(layer)  # 唯一标识层
        # 初始化动量（首次更新时）
        if layer_id not in self.v_w:
            self.v_w[layer_id] = np.zeros_like(layer.grad_weights)
            self.v_b[layer_id] = np.zeros_like(layer.grad_biases)


        # 计算动量：v = momentum * v_prev - lr * grad
        self.v_w[layer_id] = self.momentum * self.v_w[layer_id] -
self.learning_rate * layer.grad_weights
        self.v_b[layer_id] = self.momentum * self.v_b[layer_id] -
```

```
self.learning_rate * layer.grad_biases


        # 更新参数
        layer.weights += self.v_w[layer_id]
        layer.biases += self.v_b[layer_id]


class Adam(Optimizer):
    """Adam 优化器：结合动量和自适应学习率（默认首选）"""
    def __init__(self, learning_rate=0.001, beta1=0.9,
beta2=0.999, epsilon=1e-8):
        self.lr = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps = epsilon
        self.t = 0  # 迭代次数（用于偏差修正）
        # 保存一阶动量（m）和二阶动量（v）
        self.m_w = {}
        self.m_b = {}
        self.v_w = {}
        self.v_b = {}


    def update(self, layer):
        if not hasattr(layer, 'weights') or not hasattr(layer,
'biases'):
            return


        layer_id = id(layer)
        self.t += 1  # 每次更新迭代次数+1


        # 初始化动量
        if layer_id not in self.m_w:
            self.m_w[layer_id] = np.zeros_like(layer.grad_weights)
```

```
            self.m_b[layer_id] = np.zeros_like(layer.grad_biases)
            self.v_w[layer_id] = np.zeros_like(layer.grad_weights)
            self.v_b[layer_id] = np.zeros_like(layer.grad_biases)

        # 一阶动量更新：m = beta1*m_prev + (1-beta1)*grad
        self.m_w[layer_id] = self.beta1 * self.m_w[layer_id] + (1 - self.beta1) * layer.grad_weights
        self.m_b[layer_id] = self.beta1 * self.m_b[layer_id] + (1 - self.beta1) * layer.grad_biases

        # 二阶动量更新：v = beta2*v_prev + (1-beta2)*grad²
        self.v_w[layer_id] = self.beta2 * self.v_w[layer_id] + (1 - self.beta2) * (layer.grad_weights**2)
        self.v_b[layer_id] = self.beta2 * self.v_b[layer_id] + (1 - self.beta2) * (layer.grad_biases**2)

        # 偏差修正（初期 m 和 v 接近 0，修正后更稳定）
        m_w_hat = self.m_w[layer_id] / (1 - self.beta1**self.t)
        m_b_hat = self.m_b[layer_id] / (1 - self.beta1**self.t)
        v_w_hat = self.v_w[layer_id] / (1 - self.beta2**self.t)
        v_b_hat = self.v_b[layer_id] / (1 - self.beta2**self.t)

        # 更新参数：W = W - lr * m_hat / (sqrt(v_hat) + eps)
        layer.weights -= self.lr * m_w_hat / (np.sqrt(v_w_hat) + self.eps)
        layer.biases -= self.lr * m_b_hat / (np.sqrt(v_b_hat) + self.eps)
```

3.5 mlp/model.py

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, r2_score
```

```
import os


class MLP:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.optimizer = None
        # 训练历史（用于可视化）
        self.history = {
            'train_loss': [],
            'val_loss': []
        }

    def add_layer(self, layer):
        """添加层（按顺序添加，输入层→隐藏层→输出层）"""
        self.layers.append(layer)

    def set_loss(self, loss):
        """设置损失函数"""
        self.loss = loss

    def set_optimizer(self, optimizer):
        """设置优化器"""
        self.optimizer = optimizer

    def forward(self, x):
        """前向传播：计算模型输出"""
        output = x
        for layer in self.layers:
            output = layer.forward(output)
        return output
```

```
    def backward(self, y_pred, y_true):
        """反向传播：计算梯度"""
        if self.loss is None:
            raise ValueError("请先调用 set_loss 设置损失函数")
        grad = self.loss.backward(y_pred, y_true)
        # 反向遍历层（从输出层到输入层）
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

    def update(self):
        """更新模型参数"""
        if self.optimizer is None:
            raise ValueError("请先调用 set_optimizer 设置优化器")
        for layer in self.layers:
            self.optimizer.update(layer)

    def train(self, X_train, y_train, epochs=100, batch_size=32,
validation_data=None):
        """
        训练模型
        :param X_train: 训练特征 (n_samples, n_features)
        :param y_train: 训练标签 (n_samples,) 或 (n_samples, 1)
        :param epochs: 迭代次数
        :param batch_size: 批次大小
        :param validation_data: 验证集 (X_val, y_val)
        :return: 训练历史
        """
        n_samples = X_train.shape[0]
        y_train = y_train.reshape(-1, 1)  # 统一形状为(n_samples,
1)
```

```
        for epoch in range(1, epochs + 1):
            # 打乱训练数据（避免顺序依赖）
            indices = np.random.permutation(n_samples)
            X_shuffled = X_train[indices]
            y_shuffled = y_train[indices]

            train_loss = 0.0
            n_batches = n_samples // batch_size

            # 批次训练
            for i in range(n_batches):
                # 取批次数据
                start = i * batch_size
                end = start + batch_size
                X_batch = X_shuffled[start:end]
                y_batch = y_shuffled[start:end]

                # 前向传播
                y_pred = self.forward(X_batch)
                # 计算损失
                batch_loss = self.loss.forward(y_pred, y_batch)
                train_loss += batch_loss
                # 反向传播
                self.backward(y_pred, y_batch)
                # 更新参数
                self.update()

            # 计算平均训练损失
            avg_train_loss = train_loss / n_batches
            self.history['train_loss'].append(avg_train_loss)
```

```
            # 计算验证损失（如有验证集）
            val_loss = None
            if validation_data is not None:
                X_val, y_val = validation_data
                y_val = y_val.reshape(-1, 1)
                y_val_pred = self.forward(X_val)
                val_loss = self.loss.forward(y_val_pred, y_val)
                self.history['val_loss'].append(val_loss)

            # 每 10 个 epoch 打印一次日志
            if epoch % 10 == 0:
                if val_loss is not None:
                    print(f"Epoch {epoch:3d}/{epochs} | Train
Loss: {avg_train_loss:.4f} | Val Loss: {val_loss:.4f}")
                else:
                    print(f"Epoch {epoch:3d}/{epochs} | Train
Loss: {avg_train_loss:.4f}")

        return self.history

    def evaluate(self, X_test, y_test):
        """
        评估模型性能
        :return: (test_loss, metrics) → metrics 包含 MAE、R²
        """
        X_test = X_test.reshape(-1, X_test.shape[-1])
        y_test = y_test.reshape(-1, 1)
        y_pred = self.forward(X_test).reshape(-1, 1)

        # 计算损失和指标
        test_loss = self.loss.forward(y_pred, y_test)
```

```
        mae = mean_absolute_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)

        metrics = {
            'mae': mae,
            'r2': r2
        }
        return test_loss, metrics

    def predict(self, X):
        """预测新数据"""
        return self.forward(X).reshape(-1)  # 输出形状为
(n_samples,)

    def plot_history(self,
save_path='results/training_curve.png'):
        """绘制训练曲线（训练损失 vs 验证损失）"""
        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        plt.figure(figsize=(8, 5))
        plt.plot(self.history['train_loss'], label='Train Loss',
linewidth=2)
        if 'val_loss' in self.history and
self.history['val_loss']:
            plt.plot(self.history['val_loss'], label='Val Loss',
linewidth=2, linestyle='--')
        plt.xlabel('Epochs', fontsize=12)
        plt.ylabel('MSE Loss', fontsize=12)
        plt.title('Training & Validation Loss Curve', fontsize=14)
        plt.legend()
        plt.grid(alpha=0.3)
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
        plt.show()
```

```
    def plot_predictions(self, X_test, y_test,
save_path='results/predictions.png'):
        """绘制真实值 vs 预测值散点图"""
        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        y_pred = self.predict(X_test)
        plt.figure(figsize=(8, 6))
        plt.scatter(y_test, y_pred, alpha=0.6, s=50)
        # 绘制理想预测线（y=x）
        min_val = min(min(y_test), min(y_pred))
        max_val = max(max(y_test), max(y_pred))
        plt.plot([min_val, max_val], [min_val, max_val], 'r--',
linewidth=2, label='Ideal Prediction (y=x)')
        plt.xlabel('True Housing Price', fontsize=12)
        plt.ylabel('Predicted Housing Price', fontsize=12)
        plt.title('True vs Predicted Housing Prices', fontsize=14)
        plt.legend()
        plt.grid(alpha=0.3)
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
        plt.show()
```

3.6 mlp/datasets.py

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

class BostonHousingLoader:
    """Boston Housing 数据集加载器（使用替代方案）"""
    def __init__(self, test_size=0.2, random_state=42):
        self.test_size = test_size
        self.random_state = random_state
```

```
        self.scaler = StandardScaler()  # 特征标准化器


    def load_data(self):
        """
        加载数据（替代方案：从 UCI 或 openml 获取）
        :return: X_train, y_train, X_test, y_test（均已标准化）
        """
        try:
            # 方案 1：从 openml 加载（需 sklearn≥0.24）
            from sklearn.datasets import fetch_openml
            data = fetch_openml(name="boston", version=1,
as_frame=True, parser="pandas")
            X = data.data.values
            y = data.target.values.astype(np.float32)
        except:
            # 方案 2：从 UCI 下载（备用）
            url = "https://archive.ics.uci.edu/ml/machine-
learning-databases/housing/housing.data"
            columns = [
                'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
                'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT',
'MEDV'
            ]
            data = pd.read_csv(url, sep='\s+', names=columns)
            X = data.drop('MEDV', axis=1).values
            y = data['MEDV'].values.astype(np.float32)


        # 分割训练集和测试集
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=self.test_size,
random_state=self.random_state, shuffle=True
        )
```

```
        # 特征标准化（仅对训练集拟合，避免数据泄露）
        X_train = self.scaler.fit_transform(X_train)
        X_test = self.scaler.transform(X_test)

        return X_train, y_train, X_test, y_test

    def preprocess_new_data(self, X):
        """预处理新数据（使用训练集的标准化参数）"""
        return self.scaler.transform(X.reshape(-1, X.shape[-1]))
```

3.7 demo.py（基础演示脚本）

```
import numpy as np
from mlp.model import MLP
from mlp.layers import Dense
from mlp.activations import ReLU, Linear
from mlp.losses import MSE
from mlp.optimizers import Adam
from mlp.datasets import BostonHousingLoader

def main():
    # 1. 加载并预处理数据集
    print("=" * 50)
    print("1. 加载 Boston Housing 数据集...")
    data_loader = BostonHousingLoader(test_size=0.2,
random_state=42)
    X_train, y_train, X_test, y_test = data_loader.load_data()
    print(f"   训练集：{X_train.shape[0]} 样本，{X_train.shape[1]}
特征")
    print(f"   测试集：{X_test.shape[0]} 样本，{X_test.shape[1]} 特
征")
```

```
    print(f"   房价范围：{y_train.min():.1f} ~ {y_train.max():.1f}（千美元）")
    print("=" * 50)


    # 2. 构建 MLP 模型
    print("\n2. 构建 MLP 模型...")
    model = MLP()
    # 网络结构：输入层(13) → 隐藏层 1(64, ReLU) → 隐藏层 2(32, ReLU) → 输出层(1, Linear)
    model.add_layer(Dense(input_dim=13, output_dim=64, weight_initializer='he'))
    model.add_layer(ReLU())
    model.add_layer(Dense(input_dim=64, output_dim=32, weight_initializer='he'))
    model.add_layer(ReLU())
    model.add_layer(Dense(input_dim=32, output_dim=1, weight_initializer='he'))
    model.add_layer(Linear())
    # 设置损失函数和优化器
    model.set_loss(MSE())
    model.set_optimizer(Adam(learning_rate=0.001))
    print("   模型结构：13 → 64(ReLU) → 32(ReLU) → 1(Linear)")
    print("   损失函数：MSE")
    print("   优化器：Adam (lr=0.001)")
    print("=" * 50)


    # 3. 训练模型
    print("\n3. 开始训练模型...")
    history = model.train(
        X_train=X_train,
        y_train=y_train,
        epochs=100,
```

```
        batch_size=32,
        validation_data=(X_test, y_test)
    )
    print("=" * 50)

    # 4. 评估模型性能
    print("\n4. 评估模型性能...")
    test_loss, metrics = model.evaluate(X_test, y_test)
    print(f"   测试集 MSE 损失：{test_loss:.4f}")
    print(f"   平均绝对误差(MAE)：{metrics['mae']:.2f}（千美元）")
    print(f"   R²评分：{metrics['r2']:.4f}（越接近 1 越好）")
    print("=" * 50)

    # 5. 可视化结果
    print("\n5. 可视化训练结果...")
    model.plot_history(save_path='results/training_curve.png')
    model.plot_predictions(X_test, y_test,
save_path='results/predictions.png')
    print("   训练曲线已保存至：results/training_curve.png")
    print("   预测结果图已保存至：results/predictions.png")
    print("=" * 50)

    # 6. 示例预测
    print("\n6. 示例预测结果...")
    sample_indices = [0, 10, 20, 30, 40]
    X_sample = X_test[sample_indices]
    y_true_sample = y_test[sample_indices]
    y_pred_sample = model.predict(X_sample)
    for i, (true, pred) in enumerate(zip(y_true_sample,
y_pred_sample)):
        error = abs(true - pred)
        print(f"   样本{i+1}：真实房价={true:.2f}, 预测房价
```

```
={pred:.2f}, 误差={error:.2f}（千美元）")
    print("=" * 50)

if __name__ == "__main__":
    main()
```

4. 实验步骤

4.1 步骤 1：准备环境与代码（支持命令行/TRAE IDE 双方式）

1. 命令行方式：按照 2.1 节创建虚拟环境并安装依赖（避免版本冲突）；TRAE IDE 方式：按照 2.3 节完成 IDE 配置、项目创建及环境关联。
2. 按照 2.2 节创建项目目录结构，复制 3.1~3.7 节的代码到对应文件（TRAE IDE 可直接在编辑区复制粘贴，支持代码自动格式化）。
3. 创建 `results/` 目录（用于保存可视化结果）：命令行可通过 `mkdir results` 创建；TRAE IDE 可右键项目根目录选择「New」→「Directory」创建。

1. 按照 2.1 节创建虚拟环境并安装依赖（避免版本冲突）
2. 按照 2.2 节创建项目目录结构，复制 3.1~3.7 节的代码到对应文件
3. 创建 `results/` 目录（用于保存可视化结果）

4.2 步骤 2：运行基础演示脚本（命令行/TRAE IDE 双方式）

```
# 命令行方式（需激活虚拟环境）
python demo.py

# TRAE IDE 方式
# 1. 打开 demo.py 文件，点击右上角「Run」按钮或右键选择「Run 'demo'」
# 2. 运行日志在底部「Run」标签页查看，可视化结果在 results 目录中查看
```

```
python demo.py
```

预期输出（示例）

```
==================================================
1. 加载 Boston Housing 数据集...
   训练集：404 样本，13 特征
   测试集：102 样本，13 特征
   房价范围：5.0 ~ 50.0（千美元）
==================================================

2. 构建 MLP 模型...
   模型结构：13 → 64(ReLU) → 32(ReLU) → 1(Linear)
   损失函数：MSE
   优化器：Adam (lr=0.001)
==================================================

3. 开始训练模型...
Epoch  10/100 | Train Loss: 8.6315 | Val Loss: 35.2813
Epoch  20/100 | Train Loss: 7.4026 | Val Loss: 22.6025
...
Epoch 100/100 | Train Loss: 2.9939 | Val Loss: 10.5702
==================================================

4. 评估模型性能...
   测试集 MSE 损失：10.5702
   平均绝对误差(MAE)：2.35（千美元）
   R²评分：0.8642（越接近 1 越好）
==================================================
```

关键检查点

- 若出现“数据集加载失败”：检查网络连接，或手动下载 `housing.data` 文件到本地，修改 `datasets.py` 的加载路径
- 若出现“维度不匹配”：确保 y 的形状已 reshape 为(样本数,1)，代码中已包含该处

理

4.3 步骤 3：理解核心模块

3.1 全连接层（Dense）

- 核心参数：`input_dim`（输入特征数）、`output_dim`（神经元数）
- 权重初始化：He 初始化适合 ReLU，Xavier 适合 Tanh，避免梯度消失/爆炸
- 反向传播：梯度计算依赖前向传播保存的 `input`，公式推导如下：
  - 权重梯度：$\nabla_W L = X^T \cdot \nabla_z L / N$（N 为批次大小）
  - 偏置梯度：\ nabla_b L ＝ \ text{mean}(\ nabla_z L, axis ＝ 0)
  - 输入梯度：$\nabla_X L = \nabla_z L \cdot W^T$（传给前一层）

3.2 激活函数

- ReLU：隐藏层首选，解决梯度消失，但注意死亡 ReLU（输入恒负）
- Linear：回归输出层，无激活效果，直接传递梯度
- Sigmoid：仅用于二分类输出层，需注意梯度消失（输入绝对值过大）

3.3 模型训练流程（`model.train`）

1. 打乱数据 → 分批次 → 前向传播（计算预测值）
2. 计算损失 → 反向传播（计算各层梯度）
3. 优化器更新参数 → 重复 epochs 次
4. 关键：验证集用于监控过拟合，避免训练过度

4.4 步骤 4：分析实验结果

4.4.1 训练曲线分析

- 理想情况：训练损失和验证损失均下降，且最终差距较小（无过拟合）
- 过拟合：训练损失持续下降，验证损失先降后升 → 解决方案：减小模型复杂度、早停、正则化
- 欠拟合：训练损失和验证损失均很高 → 解决方案：增加网络深度/宽度、增大学习率、增加训练轮数

4.4.2 评估指标分析

- MSE：反映误差的平方（对大误差敏感），越小越好
- MAE：反映平均绝对误差（更直观），示例中 2.35 表示平均误差 2350 美元
- R^2：模型解释方差的比例，0.86 表示模型能解释 86%的房价变化（优秀）

4.4.3 预测结果散点图

- 点越靠近红色线（y=x），预测越准确
- 若某些区域点偏离较大：分析该区域的特征分布（如极端房价样本）

4.5 步骤 5：超参数调优实践

修改 `demo.py` 的模型配置，对比性能变化，示例如下：

超参数调整	测试 MSE	MAE	R^2	结论
基准（64→32, lr=0.001）	10.57	2.35	0.86	基础性能
隐藏层增大（128→64）	9.82	2.18	0.87	增大模型容量提升性能
学习率增大（0.01）	15.63	3.12	0.79	学习率过大导致震荡
学习率减小（0.0001）	18.72	3.45	0.75	学习率过小收敛缓慢
批次大小增大（64）	11.23	2.48	0.85	批次过大收敛变慢
激活函数换 Tanh	12.45	2.67	0.83	ReLU 优于 Tanh
优化器换 SGD（momentum=0.9）	13.18	2.79	0.82	Adam 优于 SGD

调优技巧

- 学习率：优先调优，建议范围 0.0001~0.01，用 Adam 时默认 0.001 通常可行
- 模型容量：隐藏层神经元数从 32/64 开始，逐步增大，避免一开始用过大模型
- 批次大小：2 的幂次（16/32/64），兼顾速度和稳定性

5. 扩展实验

5.1 实现 L2 正则化（解决过拟合）

修改 `Dense` 层的 `backward` 方法，添加权重衰减：

```
def backward(self, grad_output, l2_lambda=0.001):
    """添加 L2 正则化：梯度 += l2_lambda * weights"""
    self.grad_weights = np.dot(self.input.T, grad_output) /
self.input.shape[0] + l2_lambda * self.weights
    self.grad_biases = np.mean(grad_output, axis=0, keepdims=True)
    grad_input = np.dot(grad_output, self.weights.T)
    return grad_input
```

在 `model.train` 的 `backward` 调用中传入 `l2_lambda`：

```
self.backward(y_pred, y_batch, l2_lambda=0.001)
```

5.2 实现 Dropout 层（正则化）

在 `layers.py` 中添加 Dropout 层：

```
class Dropout(Activation):
    def __init__(self, rate=0.5):
        super().__init__()
        self.rate = rate  # 丢弃概率
        self.mask = None  # 掩码（训练时生成，测试时不用）

    def forward(self, x, training=True):
        if not training:
```

```
            return x  # 测试时不丢弃，直接返回
        # 生成掩码（保留概率=1-rate）
        self.mask = np.random.binomial(1, 1-self.rate, size=x.shape) / (1-self.rate)
        return x * self.mask

    def backward(self, grad_output):
        return grad_output * self.mask  # 反向传播时仅保留掩码为 1 的梯度
```

修改 `model.forward` 支持训练模式：

```
def forward(self, x, training=True):
    output = x
    for layer in self.layers:
        if isinstance(layer, Dropout):
            output = layer.forward(output, training=training)
        else:
            output = layer.forward(output)
    return output
```

在模型中添加 Dropout 层：

```
model.add_layer(Dense(13, 64))
model.add_layer(ReLU())
model.add_layer(Dropout(rate=0.2))  # 添加 Dropout（丢弃 20%神经元）
model.add_layer(Dense(64, 32))
model.add_layer(ReLU())
model.add_layer(Dropout(rate=0.2))
model.add_layer(Dense(32, 1))
model.add_layer(Linear())
```

5.3 网格搜索超参数（demo_advanced.py）

```Python
import itertools
from mlp.model import MLP
from mlp.layers import Dense
from mlp.activations import ReLU, Linear
from mlp.losses import MSE
from mlp.optimizers import Adam, SGD
from mlp.datasets import BostonHousingLoader

def grid_search():
    # 定义超参数网格
    param_grid = {
        'hidden_sizes': [(64, 32), (128, 64), (32, 16)],
        'learning_rate': [0.001, 0.0005],
        'batch_size': [32, 64],
        'optimizer': [Adam, lambda lr: SGD(lr=lr, momentum=0.9)]
    }

    # 加载数据
    data_loader = BostonHousingLoader(random_state=42)
    X_train, y_train, X_test, y_test = data_loader.load_data()

    best_r2 = 0
    best_params = None
    best_model = None

    # 遍历所有超参数组合
    for params in itertools.product(*param_grid.values()):
        hidden_sizes, lr, batch_size, optimizer_cls = params
        optimizer = optimizer_cls(lr) if callable(optimizer_cls)
else optimizer_cls(learning_rate=lr)

        # 构建模型
        model = MLP()
        model.add_layer(Dense(13, hidden_sizes[0]))
        model.add_layer(ReLU())
        model.add_layer(Dense(hidden_sizes[0], hidden_sizes[1]))
        model.add_layer(ReLU())
        model.add_layer(Dense(hidden_sizes[1], 1))
        model.add_layer(Linear())
```

```
        model.set_loss(MSE())
        model.set_optimizer(optimizer)

        # 训练模型
        print(f"训练参数：hidden_sizes={hidden_sizes}, lr={lr},
batch_size={batch_size},
optimizer={optimizer.__class__.__name__}")
        model.train(X_train, y_train, epochs=80,
batch_size=batch_size, validation_data=(X_test, y_test))

        # 评估模型
        _, metrics = model.evaluate(X_test, y_test)
        print(f"R²评分：{metrics['r2']:.4f}\n")

        # 更新最佳模型
        if metrics['r2'] > best_r2:
            best_r2 = metrics['r2']
            best_params = {
                'hidden_sizes': hidden_sizes,
                'learning_rate': lr,
                'batch_size': batch_size,
                'optimizer': optimizer.__class__.__name__
            }
            best_model = model

    # 输出最佳结果
    print("=" * 50)
    print(f"最佳超参数：{best_params}")
    print(f"最佳 R²评分：{best_r2:.4f}")
    best_model.plot
```

6. 项目提示语

```
# 项目生成提示语：基于 NumPy 的 MLP 神经网络实现


## 1. 项目概述

请创建一个完整的基于 NumPy 的多层感知机(MLP)神经网络项目，用于解决 Boston
Housing 房价预测问题。要求不使用任何深度学习框架（如 TensorFlow、PyTorch），完全
基于 NumPy 实现神经网络的所有核心组件。
```

2. 技术栈要求

- 核心语言：Python 3.8+
- 核心库：NumPy 1.21+（用于矩阵运算）
- 数据处理：Pandas、Scikit-learn（用于数据加载、分割和标准化）
- 可视化：Matplotlib（用于结果展示）

3. 项目结构设计

请创建以下项目结构：

```
lab1-mlp/
├── mlp/                    # 核心模块目录
│   ├── __init__.py         # 模块初始化文件，导出主要组件
│   ├── activations.py      # 激活函数实现
│   ├── datasets.py         # 数据集加载器
│   ├── layers.py           # 神经网络层实现
│   ├── losses.py           # 损失函数实现
│   ├── model.py            # MLP 模型实现
│   └── optimizers.py       # 优化器实现
├── demo.py                 # 演示脚本，完整的训练和评估流程
├── demo_advanced.py        # 高级 API 使用演示
├── requirements.txt        # 依赖列表
├── README.md               # 项目说明文档
├── EXPERIMENT_GUIDE.md     # 详细的实验指南
└── boston_housing_results.png # 示例结果图（运行 demo.py 生成）
```

4. 核心组件实现要求

4.1 神经网络层 (`mlp/layers.py`)

- 实现`Layer`基类，包含`forward()`、`backward()`和`update_params()`方法
- 实现`Dense`全连接层类：

- 支持权重初始化（He 初始化和 Xavier 初始化）
- 正确实现前向传播和反向传播
- 支持参数更新

4.2 激活函数 (`mlp/activations.py`)
- 实现`Activation`基类，继承自`Layer`
- 实现以下激活函数：
  - ReLU
  - Sigmoid
  - Tanh
  - Linear（恒等映射，用于回归输出层）
- 每个激活函数需要正确实现前向传播和反向传播的梯度计算

4.3 损失函数 (`mlp/losses.py`)
- 实现`Loss`基类，包含`forward()`和`backward()`方法
- 实现 MSE（均方误差）损失函数，适用于回归任务
- 正确实现损失计算和梯度计算

4.4 优化器 (`mlp/optimizers.py`)
- 实现`Optimizer`基类，包含`update()`方法
- 实现以下优化器：
  - SGD（随机梯度下降），支持动量
  - Adam（自适应矩估计）
- 优化器需要能够更新 Dense 层的权重和偏置

4.5 MLP 模型 (`mlp/model.py`)
- 实现`MLP`类，包含以下方法：
  - `__init__()`：初始化模型，支持指定损失函数和优化器
  - `add_layer()`和`add()`：添加层到模型
  - `set_loss()`：设置损失函数
  - `set_optimizer()`：设置优化器
  - `forward()`：前向传播

- `backward()`：反向传播
- `update()`：更新参数
- `train()`：完整的训练循环，支持批次训练和验证
- `evaluate()`：评估模型性能，计算 MSE、MAE 和 R^2 指标
- `predict()`：预测新数据

4.6 数据集加载器 (`mlp/datasets.py`)
- 实现`BostonHousingLoader`类：
- 从官方 URL 加载 Boston Housing 数据集
- 分割训练集和测试集
- 特征标准化
- 提供预处理新数据的方法

5. 演示脚本要求

5.1 基础演示 (`demo.py`)
创建完整的演示脚本，包含以下步骤：
1. 加载和预处理数据集
2. 构建 MLP 模型（建议结构：13→64(ReLU)→32(ReLU)→1(Linear)）
3. 设置损失函数（MSE）和优化器（Adam，学习率 0.001）
4. 训练模型（100 轮，批次大小 32，使用测试集作为验证集）
5. 评估模型性能
6. 进行示例预测
7. 可视化训练曲线和预测结果
8. 保存结果图像

5.2 高级 API 演示 (`demo_advanced.py`)
创建演示脚本，展示两种模型构建方式：
1. 方式 1：初始化时指定损失函数和优化器
2. 方式 2：分别设置各组件
- 两种方式都要包含完整的训练和评估流程

- 训练轮数设为 10 轮，用于快速演示

6. 文档要求

6.1 README.md

包含以下内容：

- 项目简介
- 安装说明
- 快速开始示例
- 模型构建与训练指南
- API 参考
- 常见问题

6.2 EXPERIMENT_GUIDE.md

创建详细的实验指南，包含：

- 实验概述（目标、内容、技术栈）
- 环境搭建
- 代码结构分析
- 详细的实验步骤
- 结果分析方法
- 扩展实验建议
- 常见问题与解决方案
- 实验报告要求

7. 性能要求

- 模型应能在合理时间内完成训练
- 测试集 MSE 损失应低于 15
- 代码应具有良好的可读性和可维护性
- 提供充分的注释说明

8. 注意事项

- 确保所有组件正确实现反向传播和梯度计算

- 处理好参数更新逻辑，避免更新无参数的层（如激活函数层）
- 数据预处理要正确，包括特征标准化
- 提供清晰的错误处理和用户提示
- 确保代码兼容 Python 3.8+

9. 验收标准

1. 项目结构完整，所有文件正确生成
2. 核心组件实现正确，能够正常进行前向传播和反向传播
3. 演示脚本能够成功运行，生成预期的输出和结果图
4. 模型能够正确训练和评估，达到预期的性能指标
5. 文档完整，包含项目说明和详细的实验指南