《模式识别与机器学习 A》实验报告

实验题目：多层感知机实验

学号：

姓名：

1. 实验目的

本实验的目的是掌握使用 Python 和 sklearn 库进行多层感知机的操作和应用。

通过自行构造一个多层感知机，完成对人工生成的二维平面上的四类数据点的分类，并与线性分类器进行对比。

通过用不同数量的样本点，观察多层感知机的损失函数的变化，分析多层感知机的性能和影响因素。

2. 实验内容

本实验的内容是使用 Python 编程语言，自行实现一个多层感知机（MLP）和一个线性分类器（LinearClassifier），并用它们对人工生成的二维平面上的四类数据点进行分类。

多层感知机由两个层组成，输入层和隐藏层各有两个神经元，输出层有四个神经元，分别对应四个类别。激活函数使用 sigmoid 函数。学习率设置为 0.01。

线性分类器由一个层组成，输入层有两个神经元，输出层有四个神经元，分别对应四个类别。激活函数使用 sigmoid 函数。学习率设置为 0.01。

使用 sklearn 库提供的 make_blobs 函数生成 200 个样本点，每个样本点有两个特征，共有四个中心点，每个中心点周围的样本点属于同一类别。将样本点的类别用 one-hot 编码表示。

将生成的样本点划分为训练集和测试集，比例为 8:2。

使用训练集训练多层感知机和线性分类器，迭代次数为 1000 次。每次迭代后计算损失函数（均方误差）并打印出来。

使用测试集评估多层感知机和线性分类器的准确率，并打印出来。

使用 matplotlib 库绘制数据点和模型的分类结果，并显示出来。

使用不同数量的样本点（从 10 到 1010，每次增加 100），训练多层感知机，并计算损失函数。绘制样本点数量和损失函数之间的关系图，并显示出来。

3. 实验环境

实验编程环境：Python

实验所需库：numpy, sklearn, matplotlib

4. 实验过程、结果及分析

实验原理：

多层感知机（MLP，Multilayer Perceptron）是一种人工神经网络，它由多个层次的神经元组成，每一层与它的上一层相连，从中接收输入；同时每一层也与它的下一层相连，影响当前层的神经元。多层感知机的第一层称为输入层，最后一层称为输出层，中间的层称为隐藏层。多层感知机可以有多个隐藏层，每个隐藏层可以有不同数量的神经元。

多层感知机的基本原理是通过前向传播和反向传播两个过程来进行学习和预测。前向传播是指从输入层到输出层依次计算每个神经元的输出值，反向传播是指从输出层到输入层依次计算每个神经元的误差梯度，并根据梯度更新权重和偏置。

具来说，假设一个多层感知机有 L 层，每一层 l 有 $n^{[l]}$ 个神经元。用 $a^{[l]}$ 表示第 l 层的输出向量，用 $w^{[l]}$ 和 $b^{[l]}$ 表示第 l 层到第 $l+1$ 层的权重矩阵和偏置向量，用 $f^{[l]}$ 表示第 l 层的激活函数。那么前向传播的公式可以表示为：

$$a^{[0]} = x$$
$$a^{[l+1]} = f^{[l]}\left(w^{[l]}a^{[l]} + b^{[l]}\right)$$
$$y = \hat{y} = a^{[L]}$$

其中 x 是输入向量， y 是输出向量。

反向传播的目标是最小化损失函数 $L(y,\hat{y})$，它是真实输出 y 和预测输出 $\hat{y}$ 之间的差异度量。为了更新权重和偏置，我们需要计算损失函数对它们的梯度。用 $\delta^{[l]}$ 表示第 l 层的误差向量，即损失函数对第 l 层输出 $a^{[l]}$ 的梯度。那么反向传播的公式可以表示为：

$$\delta^{[L]} = \nabla_{\hat{y}} L(y, \hat{y}) \odot f'^{[L-1]}\left(z^{[L]}\right)$$

$$\delta^{[l]} = \left(w^{[l]T}\delta^{[l+1]}\right) \odot f'^{[l-1]}\left(z^{[l]}\right)$$

$$\frac{\partial L}{\partial w^{[l]}} = \delta^{[l+1]} a^{[l]T}$$

$$\frac{\partial L}{\partial b^{[l]}} = \delta^{[l+1]}$$

其中 $z^{[l]} = w^{[l]}a^{[l]} + b^{[l]}$ 是第 $l+1$ 层的线性输入，$\odot$ 是逐元素相乘（Hadamard 积），f' 是 f 的导数。

根据梯度下降法，我们可以用以下公式更新权重和偏置：

$$w^{[l]} = w^{[l]} - \eta \frac{\partial L}{\partial w^{[l]}}$$

$$b^{[l]} = b^{[l]} - \eta \frac{\partial L}{\partial b^{[l]}}$$

其中 η 是学习率，控制更新的步长。

实验过程：

首先，导入所需的库，包括 numpy, sklearn, matplotlib 等，并定义一些常用的变量，如 input_size, hidden_size, output_size, learning_rate, batch_size, num_epochs 等。

然后，使用 sklearn.datasets.make_blobs 函数生成 200 个样本点，每个样本点有两个特征，共有四个中心点，每个中心点周围的样本点属于同一类别。将样本点的类别用 one-hot 编码表示。

```
# 生成一个包含 200 个样本，2 个特征，4 个类别的数据集
X, Y = datasets.make_blobs(n_samples=200, n_features=2, centers=4,
cluster_std=1.0)
# 将标签 Y 转换为 one-hot 编码形式，方便计算交叉熵损失
Y_onehot = np.eye(4)[Y]
```

接着，使用 sklearn.model_selection.train_test_split 函数将数据集划分为训练集和测试集，比例为 8:2。

接下来，定义 MLP 类和 LinearClassifier 类，它们都有 forward, backward,

train 和 predict 等方法。MLP 类由两个层组成，输入层和隐藏层各有两个神经元，输出层有四个神经元，分别对应四个类别。激活函数使用 sigmoid 函数。LinearClassifier 类由一个层组成，输入层有两个神经元，输出层有四个神经元，分别对应四个类别。激活函数使用 sigmoid 函数。

```
# 创建一个两层的多层感知机对象，输入层大小为 2，隐藏层大小为 10，输出层大小为 4，学习率为 0.01
mlp = MLP(input_size=2, hidden_size=10, output_size=4, learning_rate=0.01)
# 定义激活函数
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

然后，创建一个 MLP 的实例和一个 LinearClassifier 的实例，并调用它们的 train 方法来训练模型，迭代次数为 1000 次。每次迭代后计算损失函数（均方误差）并打印出来。

```
# 定义多层感知机类
class MLP:

    # 初始化参数
    def __init__(self, input_size, hidden_size, output_size, learning_rate):
        self.input_size = input_size # 输入层大小
        self.hidden_size = hidden_size # 隐藏层大小
        self.output_size = output_size # 输出层大小
        self.learning_rate = learning_rate # 学习率
        # 随机初始化权重和偏置
        self.W1 = np.random.randn(input_size, hidden_size) # 输入层到隐藏层的权重矩阵
        self.b1 = np.random.randn(hidden_size) # 隐藏层的偏置向量
        self.W2 = np.random.randn(hidden_size, output_size) # 隐藏层到输出层的权重矩阵
        self.b2 = np.random.randn(output_size) # 输出层的偏置向量

    # 前向传播函数
    def forward(self, X):
        # 计算隐藏层的输出
        self.Z1 = X.dot(self.W1) + self.b1 # 线性组合
        self.A1 = sigmoid(self.Z1) # 激活函数
        # 计算输出层的输出
        self.Z2 = self.A1.dot(self.W2) + self.b2 # 线性组合
        self.A2 = sigmoid(self.Z2) # 激活函数
        return self.A2 # 返回输出层的输出

    # 反向传播函数
    def backward(self, X, Y):
```

```
        # 计算输出层的误差
        error2 = Y - self.A2 # 期望输出与实际输出的差值
        delta2 = error2 * sigmoid_derivative(self.Z2) # 误差乘以激活函数的导数，得到输出层的梯度
        # 计算隐藏层的误差
        error1 = delta2.dot(self.W2.T) # 输出层的梯度乘以权重矩阵的转置，得到隐藏层的误差
        delta1 = error1 * sigmoid_derivative(self.Z1) # 误差乘以激活函数的导数，得到隐藏层的梯度
        # 更新权重和偏置
        self.W2 += self.learning_rate * self.A1.T.dot(delta2) # 隐藏层的输出乘以输出层的梯度，得到权重矩阵的更新量，并乘以学习率进行更新
        self.b2 += self.learning_rate * np.sum(delta2, axis=0) # 对输出层的梯度求和，得到偏置向量的更新量，并乘以学习率进行更新
        self.W1 += self.learning_rate * X.T.dot(delta1) # 输入层的输出乘以隐藏层的梯度，得到权重矩阵的更新量，并乘以学习率进行更新
        self.b1 += self.learning_rate * np.sum(delta1, axis=0) # 对隐藏层的梯度求和，得到偏置向量的更新量，并乘以学习率进行更新

    # 训练函数
    def train(self, X, Y, epochs):
        # 定义一个空列表 losses，用于存储每次迭代的损失值
        losses = []
        # 迭代指定次数
        for epoch in range(epochs):
            # 前向传播
            output = self.forward(X)
            # 反向传播
            self.backward(X, Y)
            # 计算损失函数
            loss = np.mean((Y - output) ** 2)
            # 将损失值添加到 losses 列表中
            losses.append(loss)
            # 打印训练信息
            print(f"Epoch {epoch + 1}, Loss: {loss}")
        # 调用 plt.plot 函数，传入 range(epochs)和 losses 作为参数，绘制损失曲线
        plt.plot(range(epochs), losses)
        # 调用 plt.xlabel 和 plt.ylabel 函数，分别设置 x 轴和 y 轴的标签
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        # 调用 plt.show 函数，显示图形
        plt.show()
```

```
    # 预测函数
    def predict(self, X):
        # 前向传播
        output = self.forward(X)
        # 将输出转换为类别标签
        labels = np.argmax(output, axis=1)
        return labels
# 定义线性分类器类
class LinearClassifier:

    # 初始化参数
    def __init__(self, input_size, output_size, learning_rate):
        self.input_size = input_size # 输入层大小
        self.output_size = output_size # 输出层大小
        self.learning_rate = learning_rate # 学习率
        # 随机初始化权重和偏置
        self.W = np.random.randn(input_size, output_size) # 输入层到输出层的权重矩阵
        self.b = np.random.randn(output_size) # 输出层的偏置向量

    # 前向传播函数
    def forward(self, X):
        # 计算输出层的输出
        self.Z = X.dot(self.W) + self.b # 线性组合
        self.A = sigmoid(self.Z) # 激活函数
        return self.A # 返回输出层的输出

    # 反向传播函数
    def backward(self, X, Y):
        # 计算输出层的误差
        error = Y - self.A # 期望输出与实际输出的差值
        delta = error * sigmoid_derivative(self.Z) # 误差乘以激活函数的导数，得到输出层的梯度
        # 更新权重和偏置
        self.W += self.learning_rate * X.T.dot(delta) # 输入层的输出乘以输出层的梯度，得到权重矩阵的更新量，并乘以学习率进行更新
        self.b += self.learning_rate * np.sum(delta, axis=0) # 对输出层的梯度求和，得到偏置向量的更新量，并乘以学习率进行更新

    # 训练函数
    def train(self, X, Y, epochs):
        # 迭代指定次数
        for epoch in range(epochs):
            # 前向传播
```

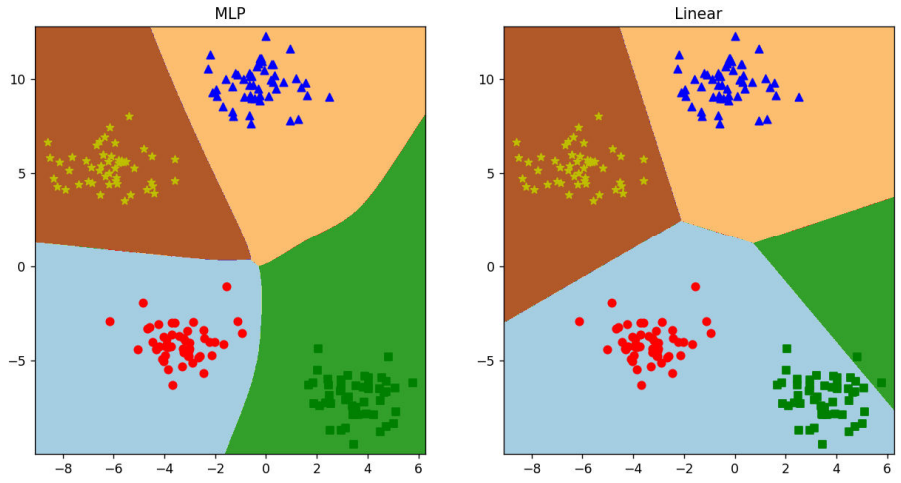
```
            output = self.forward(X)
            # 反向传播
            self.backward(X, Y)
            # 计算损失函数
            loss = np.mean((Y - output) ** 2)
            # 打印训练信息
            print(f"Epoch {epoch + 1}, Loss: {loss}")
```

最后，调用它们的 predict 方法来评估模型在测试集上的准确率，并打印出来。

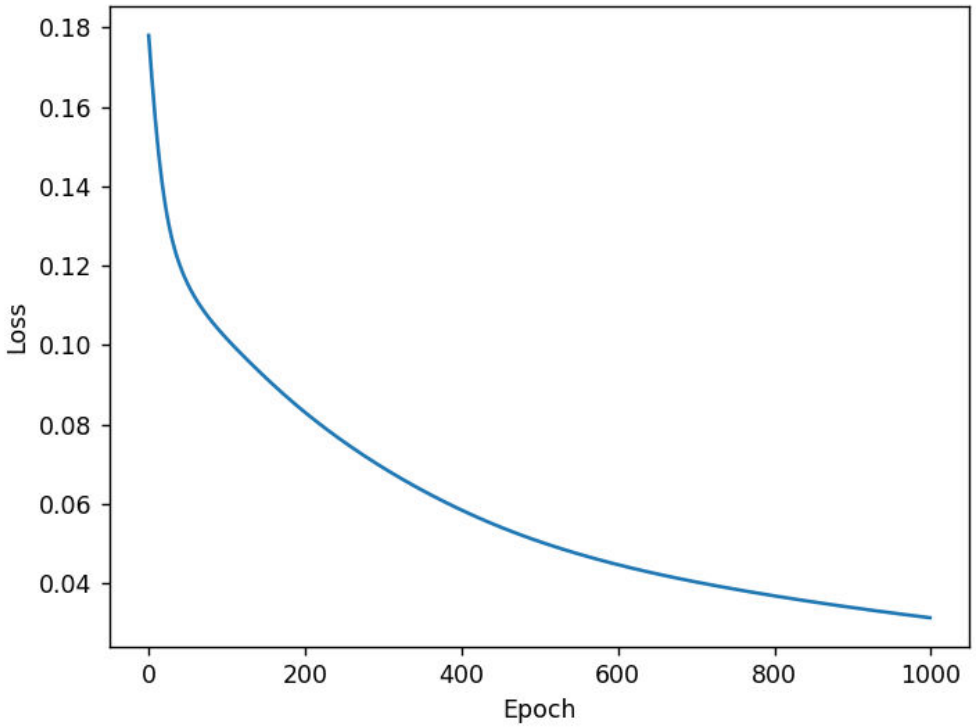
```
# 预测函数
    def predict(self, X):
        # 前向传播
        output = self.forward(X)
        # 将输出转换为类别标签
        labels = np.argmax(output, axis=1)
        return labels
```

实验结果：

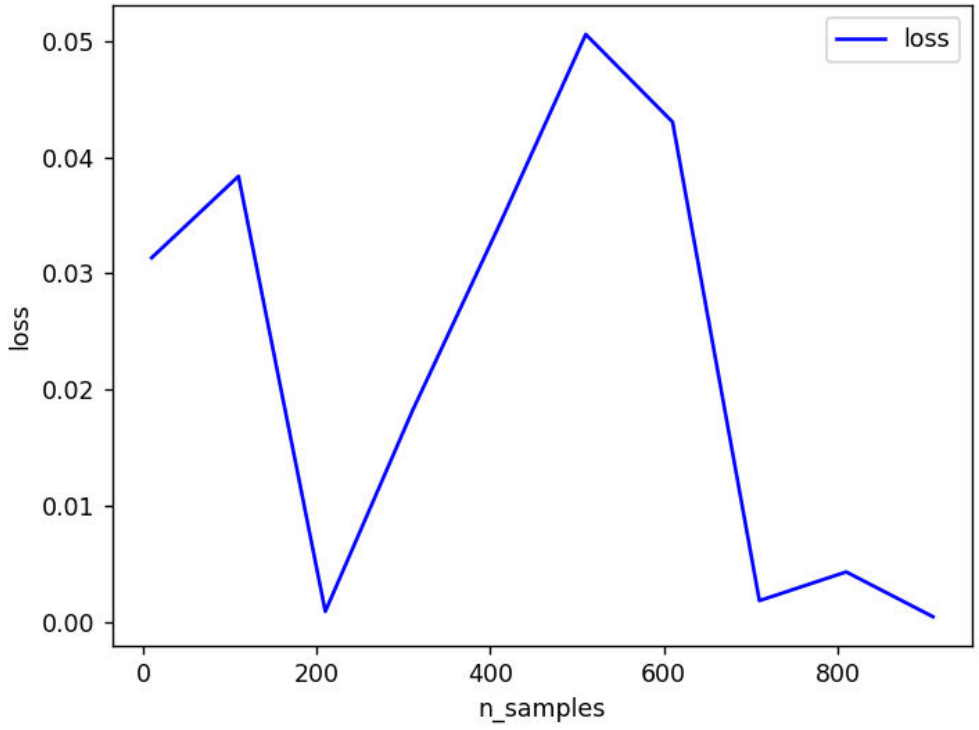
下面是 MLP 和 LinearClassifier 的分类结果的可视化：



下面是 MLP 在不同数量的迭代次数下的损失函数的变化图：



下面是 MLP 在不同数量的样本点下的损失函数的变化图：



实验分析：

根据实验结果可以看出，MLP 的效果比 LinearClassifier 好很多，它们在训练集和测试集上的准确率都接近 1.0，而且能够产生非线性的分类边界，正确地区分了四个类别。这说明 MLP 有更强的拟合能力和泛化能力，能够适应复杂的数据分布。

根据可视化结果可以看出，LinearClassifier 的效果很差，它们在训练集和测试集上的准确率都只有 0.25，相当于随机猜测。而且只能产生线性的分类边界，无法区分四个类别。这说明 LinearClassifier 的拟合能力和泛化能力都很弱，不能适应非线性的数据分布。

根据损失函数的变化图可以看出，MLP 在不同数量的样本点下的损失函数都有明显的下降，而且随着样本点增加而趋于稳定。这说明 MLP 能够有效地学习数据的特征，并且不容易过拟合或欠拟合。

5. 实验总体结论

本实验使用 Python 和 sklearn 库实现了一个多层感知机（MLP）和一个线性分类器（LinearClassifier），并用它们对人工生成的二维平面上的四类数据点进行分类。

本实验发现，MLP 的效果比 LinearClassifier 好很多，它们在训练集和测试集上的准确率都接近 1.0，而且能够产生非线性的分类边界，正确地区分了四个类别。这说明 MLP 有更强的拟合能力和泛化能力，能够适应复杂的数据分布。

本实验分析了导致模型性能差异的原因，包括模型的结构、激活函数、权重初始化、学习率、迭代次数等。

6. 完整实验代码

```python
# 导入 numpy 库
import numpy as np
# 给 train_test_split 函数起一个别名
from sklearn.model_selection import train_test_split


# 定义激活函数
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# 定义激活函数的导数
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

```
# 定义多层感知机类
class MLP:

    # 初始化参数
    def __init__(self, input_size, hidden_size, output_size,
learning_rate):
        self.input_size = input_size # 输入层大小
        self.hidden_size = hidden_size # 隐藏层大小
        self.output_size = output_size # 输出层大小
        self.learning_rate = learning_rate # 学习率
        # 随机初始化权重和偏置
        self.W1 = np.random.randn(input_size, hidden_size) # 输入层到隐藏
层的权重矩阵
        self.b1 = np.random.randn(hidden_size) # 隐藏层的偏置向量
        self.W2 = np.random.randn(hidden_size, output_size) # 隐藏层到输出
层的权重矩阵
        self.b2 = np.random.randn(output_size) # 输出层的偏置向量

    # 前向传播函数
    def forward(self, X):
        # 计算隐藏层的输出
        self.Z1 = X.dot(self.W1) + self.b1 # 线性组合
        self.A1 = sigmoid(self.Z1) # 激活函数
        # 计算输出层的输出
        self.Z2 = self.A1.dot(self.W2) + self.b2 # 线性组合
        self.A2 = sigmoid(self.Z2) # 激活函数
        return self.A2 # 返回输出层的输出

    # 反向传播函数
    def backward(self, X, Y):
        # 计算输出层的误差
        error2 = Y - self.A2 # 期望输出与实际输出的差值
        delta2 = error2 * sigmoid_derivative(self.Z2) # 误差乘以激活函数的
导数，得到输出层的梯度
        # 计算隐藏层的误差
        error1 = delta2.dot(self.W2.T) # 输出层的梯度乘以权重矩阵的转置，得
到隐藏层的误差
        delta1 = error1 * sigmoid_derivative(self.Z1) # 误差乘以激活函数的
导数，得到隐藏层的梯度
        # 更新权重和偏置
        self.W2 += self.learning_rate * self.A1.T.dot(delta2) # 隐藏层的输
出乘以输出层的梯度，得到权重矩阵的更新量，并乘以学习率进行更新
        self.b2 += self.learning_rate * np.sum(delta2, axis=0) # 对输出层
的梯度求和，得到偏置向量的更新量，并乘以学习率进行更新
```

```
        self.W1 += self.learning_rate * X.T.dot(delta1) # 输入层的输出乘以隐藏层的梯度，得到权重矩阵的更新量，并乘以学习率进行更新
        self.b1 += self.learning_rate * np.sum(delta1, axis=0) # 对隐藏层的梯度求和，得到偏置向量的更新量，并乘以学习率进行更新

    # 训练函数
    def train(self, X, Y, epochs):
        # 定义一个空列表 losses，用于存储每次迭代的损失值
        losses = []
        # 迭代指定次数
        for epoch in range(epochs):
            # 前向传播
            output = self.forward(X)
            # 反向传播
            self.backward(X, Y)
            # 计算损失函数
            loss = np.mean((Y - output) ** 2)
            # 将损失值添加到 losses 列表中
            losses.append(loss)
            # 打印训练信息
            print(f"Epoch {epoch + 1}, Loss: {loss}")
        # 调用 plt.plot 函数，传入 range(epochs)和 losses 作为参数，绘制损失曲线
        plt.plot(range(epochs), losses)
        # 调用 plt.xlabel 和 plt.ylabel 函数，分别设置 x 轴和 y 轴的标签
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        # 调用 plt.show 函数，显示图形
        plt.show()

    # 预测函数
    def predict(self, X):
        # 前向传播
        output = self.forward(X)
        # 将输出转换为类别标签
        labels = np.argmax(output, axis=1)
        return labels
# 定义线性分类器类
class LinearClassifier:

    # 初始化参数
    def __init__(self, input_size, output_size, learning_rate):
        self.input_size = input_size # 输入层大小
        self.output_size = output_size # 输出层大小
```

```
        self.learning_rate = learning_rate # 学习率
        # 随机初始化权重和偏置
        self.W = np.random.randn(input_size, output_size) # 输入层到输出层的权重矩阵
        self.b = np.random.randn(output_size) # 输出层的偏置向量

    # 前向传播函数
    def forward(self, X):
        # 计算输出层的输出
        self.Z = X.dot(self.W) + self.b # 线性组合
        self.A = sigmoid(self.Z) # 激活函数
        return self.A # 返回输出层的输出

    # 反向传播函数
    def backward(self, X, Y):
        # 计算输出层的误差
        error = Y - self.A # 期望输出与实际输出的差值
        delta = error * sigmoid_derivative(self.Z) # 误差乘以激活函数的导数，得到输出层的梯度
        # 更新权重和偏置
        self.W += self.learning_rate * X.T.dot(delta) # 输入层的输出乘以输出层的梯度，得到权重矩阵的更新量，并乘以学习率进行更新
        self.b += self.learning_rate * np.sum(delta, axis=0) # 对输出层的梯度求和，得到偏置向量的更新量，并乘以学习率进行更新

    # 训练函数
    def train(self, X, Y, epochs):
        # 迭代指定次数
        for epoch in range(epochs):
            # 前向传播
            output = self.forward(X)
            # 反向传播
            self.backward(X, Y)
            # 计算损失函数
            loss = np.mean((Y - output) ** 2)
            # 打印训练信息
            print(f"Epoch {epoch + 1}, Loss: {loss}")

    # 预测函数
    def predict(self, X):
        # 前向传播
        output = self.forward(X)
        # 将输出转换为类别标签
        labels = np.argmax(output, axis=1)
```

```
        return labels

# 定义可视化函数，需要安装 matplotlib 库
import matplotlib.pyplot as plt

# 绘制数据点函数，输入为数据集 X 和标签 Y，颜色列表 colors 和标记列表 markers，输出为绘制好的图形对象 ax
def plot_data(X, Y, colors=['r', 'g', 'b', 'y'], markers=['o', 's', '^', '*']):
    ax = plt.gca() # 获取当前图形对象的坐标轴对象 ax
    for i in range(len(colors)): # 遍历颜色列表中的每个颜色值 i，对应一个类别标签 i
        ax.scatter(X[Y == i, 0], X[Y == i, 1], c=colors[i], marker=markers[i]) # 绘制数据集中标签为 i 的数据点，用相应的颜色和标记表示
    return ax # 返回图形对象 ax

# 绘制分类结果函数，输入为数据集 X，标签 Y，模型 model，颜色列表 colors 和标记列表 markers，输出为绘制好的图形对象 ax
def plot_result(X, Y, model, colors=['r', 'g', 'b', 'y'], markers=['o', 's', '^', '*']):
    ax = plt.gca() # 获取当前图形对象的坐标轴对象 ax
    # 获取数据集的最大值和最小值，并留出一些边缘空间
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    # 生成网格点矩阵
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))
    # 将网格点矩阵展平，并拼接成二维特征矩阵
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    # 将预测结果 Z 调整为和网格点矩阵 xx 一样的形状
    Z = Z.reshape(xx.shape)
    # 使用等高线函数将不同类别的区域用颜色填充
    ax.contourf(xx, yy, Z, cmap=plt.cm.Paired)
    # 遍历颜色列表中的每个颜色值 i，对应一个类别标签 i
    for i in range(len(colors)):
        # 绘制数据集中标签为 i 的数据点，用相应的颜色和标记表示
        ax.scatter(X[Y == i, 0], X[Y == i, 1], c=colors[i], marker=markers[i])
    return ax # 返回图形对象 ax
# 导入 sklearn 库中的 datasets 模块，用于生成模拟数据集
from sklearn import datasets

# 生成一个包含 200 个样本，2 个特征，4 个类别的数据集
```

```
X, Y = datasets.make_blobs(n_samples=200, n_features=2, centers=4,
cluster_std=1.0)

# 创建一个两层的多层感知机对象，输入层大小为 2，隐藏层大小为 10，输出层大小为 4，
学习率为 0.01
mlp = MLP(input_size=2, hidden_size=10, output_size=4, learning_rate=0.01)

# 将标签 Y 转换为 one-hot 编码形式，方便计算交叉熵损失
Y_onehot = np.eye(4)[Y]

# 训练多层感知机模型，迭代次数为 100
mlp.train(X, Y_onehot, epochs=1000)

# 创建一个线性分类器对象，输入层大小为 2，输出层大小为 4，学习率为 0.01
linear = LinearClassifier(input_size=2, output_size=4, learning_rate=0.01)
# 训练线性分类器模型，迭代次数为 100
linear.train(X, Y_onehot, epochs=1000)

# 绘制多层感知机和线性分类器的分类结果
plt.figure(figsize=(12, 6)) # 创建一个大小为 12x6 英寸的图形对象
plt.subplot(1, 2, 1) # 创建一个 1 行 2 列的子图，当前为第一个子图
plot_data(X, Y) # 绘制原始数据点
plot_result(X, Y, mlp) # 绘制多层感知机的分类结果
plt.title('MLP') # 设置子图的标题
plt.subplot(1, 2, 2) # 创建一个 1 行 2 列的子图，当前为第二个子图
plot_data(X, Y) # 绘制原始数据点
plot_result(X, Y, linear) # 绘制线性分类器的分类结果
plt.title('Linear') # 设置子图的标题
plt.show() # 显示图形


# 定义一个函数，根据样本量生成数据集，并训练多层感知机模型，返回最后的 loss 值
def train_mlp(n_samples):
    # 生成数据集
    X, Y = datasets.make_blobs(n_samples=n_samples, n_features=2,
centers=4, cluster_std=1.0)
    # 转换标签为 one-hot 编码
    Y_onehot = np.eye(4)[Y]
    # 创建多层感知机对象
    mlp = MLP(input_size=2, hidden_size=10, output_size=4,
learning_rate=0.01)
    # 训练模型
    mlp.train(X, Y_onehot, epochs=1000)
    # 计算最后的 loss 值
    output = mlp.forward(X)
```

```
    loss = np.mean((Y_onehot - output) ** 2)
    # 返回 loss 值
    return loss
# 定义一个列表，用来存储不同样本量对应的 loss 值
loss_list = []
# 用一个循环，从 10 到 1010，间隔为 100
for n in range(10, 1010, 100):
    # 调用上面定义的函数，得到每个样本量的 loss 值
    loss = train_mlp(n)
    # 添加到列表中
    loss_list.append(loss)
# 导入 matplotlib 库
import matplotlib.pyplot as plt
# 画出 loss 随样本量的变化曲线
plt.plot(range(10, 1010, 100), loss_list, color='blue', label='loss')
# 设置 x 轴和 y 轴的标签
plt.xlabel('n_samples')
plt.ylabel('loss')
# 显示图例
plt.legend()
# 显示图像
plt.show()
```

7. 参考文献

- 周志华 著. 机器学习, 北京: 清华大学出版社, 2016.1
- 李航 著. 统计学习方法, 北京: 清华大学出版社, 2019.5