《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

班级：2203601

学号：2022113416

姓名：刘子康

实验报告内容

一、实验目的

• 掌握多层感知机的原理和构造方法，理解前向传播与反向传播过程；

• 构造一个多层感知机，完成对某种类型的样本数据的分类（如图像、文本等），也可以对人工自行构造的二维平面超过 3 类数据点（或者其它标准数据集）进行分类。

二、实验内容

使用 Python 编程，实现一个多层感知机模型（MLP）和一个线性分类器（Linear Classifier），对人工生成的 4 类别的二维数据点进行分类。

比较多层感知机与线性分类器的性能，并分析原因；采用不同数据量，不同超参数，比较实验效果。实现实验结果的可视化。

三、实验环境

• 操作系统：Windows 11

• 编程语言：Python 3.10

• 第三方库：Numpy 1.23.4，Matplotlib 3.8.2

• IDE：Pycharm 2022 社区版

四、实验过程、结果及分析

4.1 实验原理

感知机（PLA）是一种由输入层和输出层组成的单个神经元模型，也是较大神经网络的前身，对于输入 X，输出 y 的计算公式为$y = g(\sum_{i=1}^{n} w_i x_i + b) = g(W^T X)$，其中$W = [w_1, w_2, \dots, w_n, b]^T, X = [x_1, x_2, \dots, x_n, 1]^T, g(z) = \begin{cases} +1, z \geq 0 \\ -1, z < 0 \end{cases}$，g(z)是激活函数，常使用阶跃函数，因此可以模拟逻辑电路。感知机是一个线性的二分类器，对于非线性的数据不能进行有效的分类，因此可以加深神经元的网络层次。

多层感知机（MLP）便是由感知机推广而来，它最主要的特点就是有多个神经元层，通常包含一个输入层、一个或多个隐藏层以及一个输出层，是一种特定类型的人工神经网络。理论上，利用多层感知机构成的神经网络可模拟任意逻辑函数。多层感知机通过前向传播和反向传播的迭代过程对数据特征进行学习，并更新权重和偏置。

前向传播是指从输入层输入样本，逐步经各隐藏层计算，最后在输出层经过运算得到样本的类别预测结果的过程。设：

• w_{jk}^l为第l层的第j个神经元与其前一层，即第l-1 层的第k个神经元之间连接的权重

• b_j^l为第l层的第j个神经元上的偏置

• a_j^l为第l层的第j个神经元上的激活值

那么第l层的第j个神经元上的激活值a_j^l与其前一层，即第l-1 层的所有神经元k的激活值具有如下关系：$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$，其中$\sigma$为 Sigmoid 函数。考虑到第$l$层所有神经元的整体，则可写成矩阵形式：$a^l = \sigma(w^l a^{l-1} + b^l) = \sigma(z^l)$，其中$z^l$为第$l$层的神经元的加权输入。

多层感知机通常以误差平方和作为目标函数，即$C = \frac{1}{2n}\sum_x \|y(x) - \alpha^L(x)\|^2$，其中$y$(x)

是输入x的真实值输出，n是训练样本个数，L是神经网络的层数，$\alpha^L(x)$是对输入x经过网络最后一层处理得到的激活输出。

反向传播是指按照从输出层到输入层的方向，逐层求出上述目标函数对各神经元权值参数的偏导，进而求出梯度，并对权重矩阵和偏置向量进行更新。然后前向传播利用更新后的参数取值再次进行预测并计算目标函数。上述“前向传播+反向传播”的过程将多次重复，直到网络收敛或者完成了预定的迭代次数。

- 输出层误差：$\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$
- 隐藏层误差：$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$

其中⊙表示哈达玛乘积，即矩阵对应元素相乘。

4.2 实验过程

4.2.1 数据预处理

首先使用 Sklearn 库的 datasets.make_blobs 函数，生成 n_samples 个数据样本，每个样本有 2 个特征值，一共 K 个中心点（K 默认为 4），每个中心点附近的样本点为同一类别。对标签进行 one-hot 编码，如 0, 1, 2, 3 编码为[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]，与模型输出对齐，便于计算误差和准确率。使用 model_selection.train_test_split 函数随机划分数据集，80%为训练集，20%为测试集。

```
# 生成K个类别二维高斯分布数据
单元测试 | 注释生成 | 代码解释 | 缺陷检测
def generate_data(n_samples, K=4):
    X, Y = make_blobs(n_samples=n_samples, n_features=2, centers=K, cluster_std=2.56, random_state=42)
    # 将标签进行 one-hot 编码
    Y = np.eye(K)[Y]
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)

    return X_train, X_test, y_train, y_test
```

4.2.2 MLP 模型构建

（1）模型参数初始化

定义 MLP 类，接收参数为输入输出层维度及各层神经元数量，以及学习率，包含 1 个输入层、2 个隐藏层和 1 个输出层。初始化权重和偏置，权重初始为上一层×当前层维度的(0, 1)随机值矩阵，偏置初始为 0 向量。

```
# 多层感知机模型
class MLP():
    单元测试 | 注释生成 | 代码解释 | 缺陷检测
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size, learning_rate=1e-2):
        self.input_size = input_size
        self.hidden_size1 = hidden_size1
        self.hidden_size2 = hidden_size2
        self.output_size = output_size
        self.learning_rate = learning_rate

        # 初始化权重和偏置
        self.W1 = np.random.randn(self.input_size, self.hidden_size1)
        self.W2 = np.random.randn(self.hidden_size1, self.hidden_size2)
        self.W3 = np.random.randn(self.hidden_size2, self.output_size)
        self.b1 = np.zeros((1, self.hidden_size1))
        self.b2 = np.zeros((1, self.hidden_size2))
        self.b3 = np.zeros((1, self.output_size))
```

（2）前向传播

从输入层到输出层，逐层计算各神经元的激活值，最后得到输出结果。除输入层外，每层首先利用权重矩阵和偏置向量计算神经元的加权输入，然后通过激活函数 Sigmoid 得到激活值。$Sigmoid(z) = \frac{1}{1+e^{-z}}$。

```
43          # 前向传播
            单元测试 | 注释生成 | 代码解释 | 缺陷检测
44          def forward(self, x):
45              self.x = x
46              self.h1 = Sigmoid(np.dot(x, self.W1) + self.b1)
47              self.h2 = Sigmoid(np.dot(self.h1, self.W2) + self.b2)
48              self.y = Sigmoid(np.dot(self.h2, self.W3) + self.b3)
49              return self.y
```

（3）反向传播

从输出层到输入层，逐层计算误差，最后使用梯度下降法，更新和优化各层权重和偏置参数。反向传播的目标是最小化损失函数$L(y,\hat{y})$，它是真实值 y 和预测值$\hat{y}$的差异度量。

```
51          # 反向传播 更新参数
            单元测试 | 注释生成 | 代码解释 | 缺陷检测
52          def backward(self, y):
53              # 输出层误差
54              output_error = y - self.y
55              output_delta = output_error * Sigmoid_derivative(self.y)
56
57              # 第二个隐藏层误差
58              hidden_error2 = np.dot(output_delta, self.W3.T)
59              hidden_delta2 = hidden_error2 * Sigmoid_derivative(self.h2)
60
61              # 第一个隐藏层误差
62              hidden_error1 = np.dot(hidden_delta2, self.W2.T)
63              hidden_delta1 = hidden_error1 * Sigmoid_derivative(self.h1)
64
65              # 更新权重和偏置
66              self.W3 += self.learning_rate * np.dot(self.h2.T, output_delta)
67              self.b3 += self.learning_rate * np.sum(output_delta, axis=0)
68              self.W2 += self.learning_rate * np.dot(self.h1.T, hidden_delta2)
69              self.b2 += self.learning_rate * np.sum(hidden_delta2, axis=0)
70              self.W1 += self.learning_rate * np.dot(self.x.T, hidden_delta1)
71              self.b1 += self.learning_rate * np.sum(hidden_delta1, axis=0)
```

4.2.3 线性分类器模型构建

与实现 MLP 模型过程基本相同，定义 LinearClassifier 类，初始化模型参数，定义前向传播和反向传播过程，不同之处在于线性分类器只有一个输入层和一个输出层，将输入 X 经过一次简单线性变化和激活函数后即得到输出结果 y（各类别概率值）。

4.2.4 训练和测试

设置不同数据量和学习率，分别训练模型，设置迭代次数，每次迭代过程分别进行前向传播和反向传播，每完成 1/10 输出准确率和误差。模型训练完成后在测试集上进行测试，将预测结果与真实值比较，计算准确率。

```python
# 训练模型
单元测试 | 注释生成 | 代码解释 | 缺陷检测
def train(model, X, y, epochs=1000):
    acc, loss = [], []
    for epoch in range(epochs):
        # 前向传播
        y_pred = model.forward(X)
        # 反向传播
        model.backward(y)
        # 输出误差及准确率
        if (epoch + 1) % (epochs // 10) == 0:
            acc.append(test(model, X, y))
            loss.append(np.mean(np.abs(y_pred - y)))
            print(f'Epoch {epoch + 1}, Accuracy: {acc[-1] * 100:.2f}%, Loss: {loss[-1]:.6f}')
    return acc, loss

# 测试模型
单元测试 | 注释生成 | 代码解释 | 缺陷检测
def test(model, X, y):
    y_pred = model.forward(X)
    correct = np.sum(np.argmax(y_pred, axis=1) == np.argmax(y, axis=1))
    return correct / len(y)
```

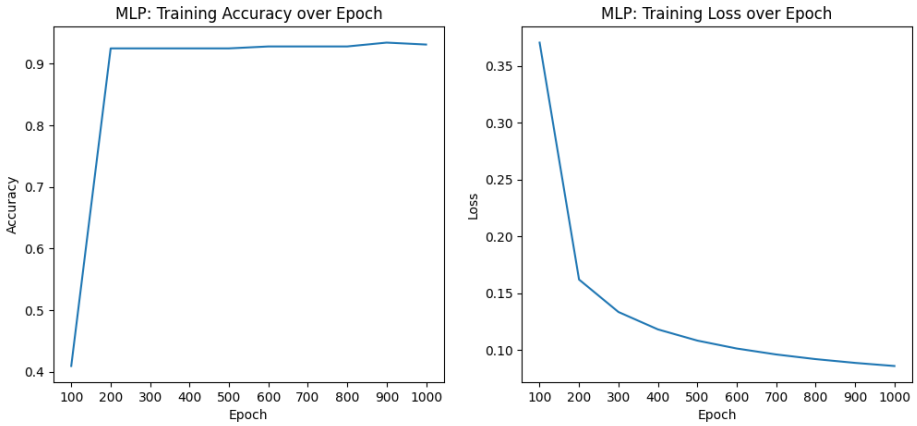
4.2.5 输出结果并绘图

比较多层感知机和线性分类器的性能，分析可能原因。比较不同数据量和不同学习率下模型的分类效果，分析可能原因。绘制数据点分布图，准确率和误差随迭代次数增大的变化图像，分析实验结果。

4.3 实验结果及分析

4.3.1 准确率、误差变化曲线

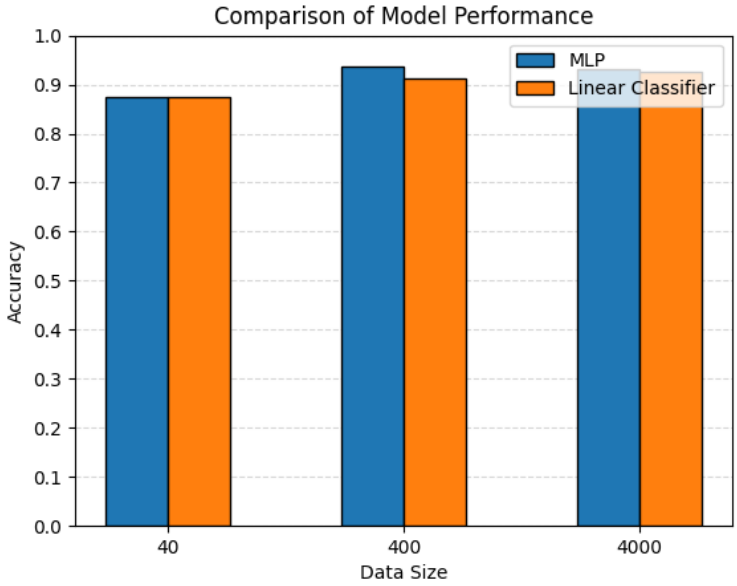
数据量为 400，学习率为 1e-3 时，MLP 训练过程准确率和误差变化如下图所示。模型在 200 个 epoch 时已达到较高准确率，之后缓慢上升并稳定在 93%左右波动；loss 值先是迅速下降，在 epoch=200 后缓慢下降。



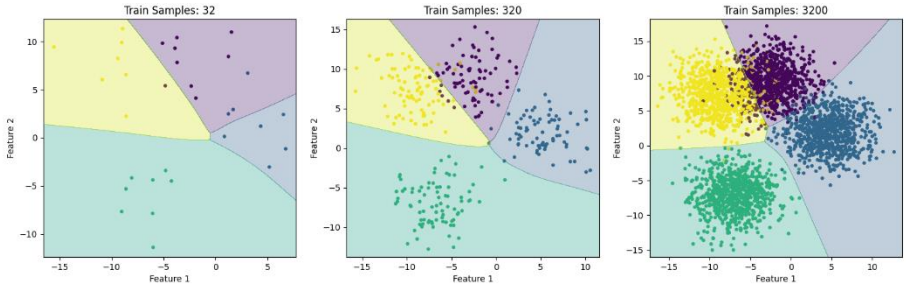
4.3.2 不同数据量

学习率 1e-3，数据量分别为 40、400、4000 时，MLP 和线性分类器的分类效果如下图所示。由图可以看出三个数据量时 MLP 的分类准确率均比线性分类器高。

数据量为 40 时，由于数据量较少，模型可能欠拟合，没有很好的学习到数据特征，且误分类点对计算准确率影响较大，因此分类准确率不高；数据量为 4000 时，由于数据量较多，由样本点分布可以看到，部分类别的样本点重叠度变高，边界模糊，边界附近样本点分类难度较高，因此分类准确率略有下降。



不同数据量时，样本点分布情况及 MLP 分类决策边界如下图所示：

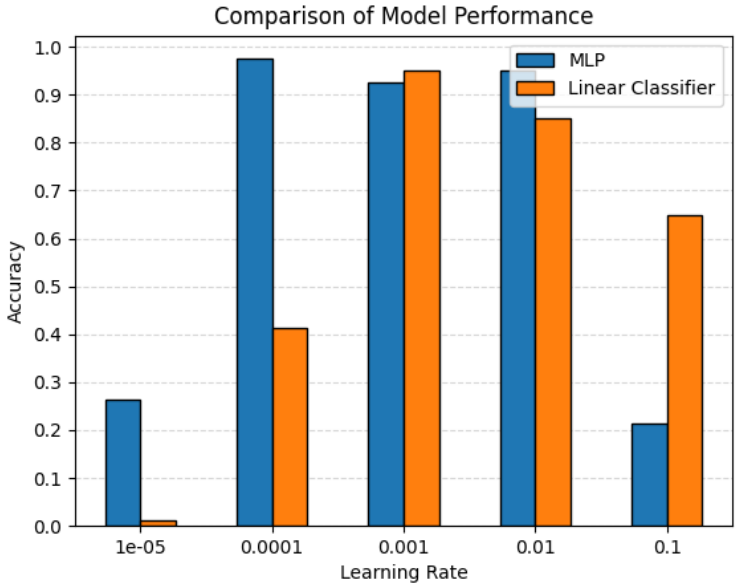


4.3.3 不同学习率

数据量为 400，学习率分别为 1e-5、1e-4、1e-3、1e-2、1e-1 时，MLP 和线性分类器的分类效果如下图所示。

由图可以看出学习率较小时(lr≤0.01)，MLP 的分类准确率高于线性分类器，学习率较大时(lr=0.1)，MLP 的分类准确率低于线性分类器。这可能是因为相较于线性分类器，MLP 网络层次更深，对学习率更敏感。在较小学习率时，MLP 能通过多层神经元学习到有效特征，而线性分类器较为简单，收敛速度很慢；而在学习率较大时，参数更新波动过大，模型无法收敛，且 MLP 权重和偏置参数较多，影响程度更大。

总体来说，学习率过小会导致模型收敛速度过慢，而学习率过大会导致模型参数更新波动过大，从而无法收敛。MLP 和线性分类器分别在 lr=1e-4 和 lr=1e-3 时达到最高准确率。



五、实验总体结论

此次试验构建了一个多层感知机模型和一个线性分类器，并对手动生成的 4 类别二维数据点进行分类，比较了两个模型的性能，以及不同数据量、不同学习率下的分类效果。掌握了感知机和多层感知机模型的基本原理，前向传播和反向传播的过程。

在多数情况下，MLP 性能好于线性分类器，分类准确率更高，其优势在于具有多层神经元网络，每一层都能学习不同层次的特征，具有较强的非线性映射能力，能够逼近任何连续的非线性函数和构建复杂的决策边界，在面对复杂的数据样本分布时能达到更好的分类效果。但 MLP 也存在着容易过拟合、对初始权重敏感以及计算量大等缺点，需要进一步改进。

较大的数据量可以提供更多的信息，有助于模型学习到更丰富的特征，从而提高分类的准确性和泛化能力，同时减少过拟合风险。但数据量的增大也可能使得样本点分布重叠度变高，边界附近的样本点分类难度较高，从而使得准确率下降。

学习率过大可能导致模型在最优解附近震荡而无法收敛，甚至会发散；学习率过小则可能导致模型收敛速度过慢，或陷入局部极小点，而不是全局最优解。

六、完整实验代码

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  from sklearn.datasets import make_blobs
4.  from sklearn.model_selection import train_test_split
5.
6.
7.  np.random.seed(42)
8.
9.  # 生成K 个类别二维高斯分布数据
10. def generate_data(n_samples, K=4):
```

```
    X, Y = make_blobs(n_samples=n_samples, n_features=2, centers=K, cluster_std=2.56, random_state=42)
    # 将标签进行 one-hot 编码
    Y = np.eye(K)[Y]
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)

    return X_train, X_test, y_train, y_test

# 激活函数及其导数
def Sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def Sigmoid_derivative(x):
    return x * (1.0 - x)

# 多层感知机模型
class MLP():
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size, learning_rate=1e-3):
        self.input_size = input_size
        self.hidden_size1 = hidden_size1
        self.hidden_size2 = hidden_size2
        self.output_size = output_size
        self.learning_rate = learning_rate

        # 初始化权重和偏置
        self.W1 = np.random.randn(self.input_size, self.hidden_size1)
        self.W2 = np.random.randn(self.hidden_size1, self.hidden_size2)
        self.W3 = np.random.randn(self.hidden_size2, self.output_size)
        self.b1 = np.zeros((1, self.hidden_size1))
        self.b2 = np.zeros((1, self.hidden_size2))
        self.b3 = np.zeros((1, self.output_size))

    # 前向传播
    def forward(self, x):
        self.x = x
        self.h1 = Sigmoid(np.dot(x, self.W1) + self.b1)
        self.h2 = Sigmoid(np.dot(self.h1, self.W2) + self.b2)
        self.y = Sigmoid(np.dot(self.h2, self.W3) + self.b3)
        return self.y

    # 反向传播 更新参数
    def backward(self, y):
```

```
53.         # 输出层误差
54.         output_error = y - self.y
55.         output_delta = output_error * Sigmoid_derivative(self.y)
56.
57.         # 第二个隐藏层误差
58.         hidden_error2 = np.dot(output_delta, self.W3.T)
59.         hidden_delta2 = hidden_error2 * Sigmoid_derivative(self.h2)
60.
61.         # 第一个隐藏层误差
62.         hidden_error1 = np.dot(hidden_delta2, self.W2.T)
63.         hidden_delta1 = hidden_error1 * Sigmoid_derivative(self.h1)
64.
65.         # 更新权重和偏置
66.         self.W3 += self.learning_rate * np.dot(self.h2.T, output_delta)
67.         self.b3 += self.learning_rate * np.sum(output_delta, axis=0)
68.         self.W2 += self.learning_rate * np.dot(self.h1.T, hidden_delta2)
69.         self.b2 += self.learning_rate * np.sum(hidden_delta2, axis=0)
70.         self.W1 += self.learning_rate * np.dot(self.x.T, hidden_delta1)
71.         self.b1 += self.learning_rate * np.sum(hidden_delta1, axis=0)
72.
73. # 线性分类器
74. class LinearClassifier():
75.     def __init__(self, input_size, output_size, learning_rate=1e-3):
76.         self.input_size = input_size
77.         self.output_size = output_size
78.         self.learning_rate = learning_rate
79.
80.         # 初始化权重和偏置
81.         self.W = np.random.randn(self.input_size, self.output_size)
82.         self.b = np.zeros((1, self.output_size))
83.
84.     # 前向传播
85.     def forward(self, x):
86.         self.x = x
87.         self.y = Sigmoid(np.dot(x, self.W) + self.b)
88.         return self.y
89.
90.     # 反向传播 更新参数
91.     def backward(self, y):
92.         # 输出层误差
93.         error = y - self.y
94.         delta = error * Sigmoid_derivative(self.y)
95.
96.         # 更新权重和偏置
```

```
97.          self.W += self.learning_rate * np.dot(self.x.T, delta)
98.          self.b += self.learning_rate * np.sum(delta, axis=0)
99.
100.# 训练模型
101.def train(model, X, y, epochs=1000):
102.    acc, loss = [], []
103.    for epoch in range(epochs):
104.        # 前向传播
105.        y_pred = model.forward(X)
106.        # 反向传播
107.        model.backward(y)
108.        # 输出误差及准确率
109.        if (epoch + 1) % (epochs // 10) == 0:
110.            acc.append(test(model, X, y))
111.            loss.append(np.mean(np.abs(y_pred - y)))
112.            print(f'Epoch {epoch + 1}, Accuracy: {acc[-1] * 100:.2f}%, Loss: {loss[-
    1]:.6f}')
113.    return acc, loss
114.
115.# 测试模型
116.def test(model, X, y):
117.    y_pred = model.forward(X)
118.    correct = np.sum(np.argmax(y_pred, axis=1) == np.argmax(y, axis=1))
119.    return correct / len(y)
120.
121.# 预测值
122.def pred(model, X):
123.    y_pred = model.forward(X)
124.    return np.argmax(y_pred, axis=1)
125.
126.# 绘制数据分布及决策边界
127.def plot_decision_boundary(model, X, y):
128.    h = 0.01  # 网格步长
129.    color = [np.array(np.where(y[i][:] != 0))[1] for i in range(len(y))]
130.    fig, axes =  plt.subplots(1, 3, figsize=(18, 5))
131.
132.    for i in range(3):
133.        axes[i].scatter(X[i][:, 0], X[i][:, 1], c=color[i], s=10)
134.        axes[i].set_title('Train Samples: %d' % (X[i].shape[0]))
135.        axes[i].set_xlabel('Feature 1')
136.        axes[i].set_ylabel('Feature 2')
137.
138.        x_min, x_max = X[i][:, 0].min() - 1, X[i][:, 0].max() + 1
139.        y_min, y_max = X[i][:, 1].min() - 1, X[i][:, 1].max() + 1
```

```
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

        Z = pred(model[i], np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)

        axes[i].contourf(xx, yy, Z, alpha=0.3)

    plt.show()

# 绘制条形图
def plot_bar(acc, x_label, x_ticks):
    bar_width = 0.2 # 条形图宽度
    index = np.arange(len(x_ticks)) * 0.75   # 数据的索引位置
    # MLP 模型
    plt.bar(index - bar_width / 2, acc[0], bar_width, label='MLP', edgecolor='black', zorder=2)
    # 线性分类器
    plt.bar(index + bar_width / 2, acc[1], bar_width, label='Linear Classifier', edgecolor='black', zorder=2)

    plt.xlabel(x_label)
    plt.ylabel('Accuracy')
    plt.title('Comparison of Model Performance')
    plt.xticks(index, x_ticks)
    plt.yticks(np.arange(0, 1.1, 0.1))
    plt.legend()
    plt.grid(axis='y', linestyle='--', alpha=0.5, zorder=0)
    plt.show()

if __name__ == '__main__':
    data_sizes = [40, 400, 4000]
    lrs = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]
    model, X, y = [], [], []

    '''======================================== MLP 训练过程 ========================================'''
    # 不同数据量时样本点分布及决策边界
    for n_samples in data_sizes:
        print("=" * 15 + f'n_samples: {n_samples}' + "=" * 15)
        # 生成数据集
        X_train, X_test, y_train, y_test = generate_data(n_samples=n_samples)
        X.append(X_train)
        y.append(y_train)

```

```
        # 多层感知机
        model_MLP = MLP(input_size=X_train.shape[1], hidden_size1=8, hidden_size2=32, output_size=y_train.shape[1])
        acc, loss = train(model_MLP, X_train, y_train)
        model.append(model_MLP)
        acc_MLP = test(model_MLP, X_test, y_test)
        print("Final Accuracy: %.2f%%\n" % (acc_MLP * 100))

        if n_samples == 400:
            epoch = np.linspace(0, 1000, 11)[1:]
            fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharex=True)

            axes[0].plot(epoch, acc)
            axes[0].set_title('MLP: Training Accuracy over Epoch')
            axes[0].set_xlabel('Epoch')
            axes[0].set_ylabel('Accuracy')
            axes[0].set_xticks(epoch)

            axes[1].plot(epoch, loss)
            axes[1].set_xlabel('Epoch')
            axes[1].set_ylabel('Loss')
            axes[1].set_title('MLP: Training Loss over Epoch')
            axes[1].set_xticks(epoch)

            plt.show()

    plot_decision_boundary(model, X, y)

    '''======================================== 不同数据量 ========================================'''
    acc = [[], []]
    for n_samples in data_sizes:
        print("=" * 15 + f'n_samples: {n_samples}' + "=" * 15)
        # 生成数据集
        X_train, X_test, y_train, y_test = generate_data(n_samples=n_samples)
        # print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)

        # 多层感知机
        print("MLP:")
        model_MLP = MLP(input_size=X_train.shape[1], hidden_size1=8, hidden_size2=32, output_size=y_train.shape[1])
        train(model_MLP, X_train, y_train)
        acc_MLP = test(model_MLP, X_test, y_test)
        acc[0].append(acc_MLP)
```

```
        print("Final Accuracy: %.2f%%\n" % (acc_MLP * 100))

        # 线性分类器
        print("Linear:")
        model_Linear = LinearClassifier(input_size=X_train.shape[1], output_size=y_train.shape[1])
        train(model_Linear, X_train, y_train)
        acc_Linear = test(model_Linear, X_test, y_test)
        acc[1].append(acc_Linear)
        print("Final Accuracy: %.2f%%\n" % (acc_Linear * 100))

    plot_bar(acc, 'Data Size', data_sizes)

    '''======================================== 不同学习率 ========================================'''
    acc = [[], []]
    for lr in lrs:
        print("=" * 12 + f'learning rate: {lr}' + "=" * 12)
        # 生成数据集
        X_train, X_test, y_train, y_test = generate_data(n_samples=400)
        # print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)

        # 多层感知机
        print("MLP:")
        model_MLP = MLP(input_size=X_train.shape[1], hidden_size1=8, hidden_size2=32, output_size=y_train.shape[1], learning_rate=lr)
        train(model_MLP, X_train, y_train)
        acc_MLP = test(model_MLP, X_test, y_test)
        acc[0].append(acc_MLP)
        print("Final Accuracy: %.2f%%\n" % (acc_MLP * 100))

        # 线性分类器
        print("Linear:")
        model_Linear = LinearClassifier(input_size=X_train.shape[1], output_size=y_train.shape[1], learning_rate=lr)
        train(model_Linear, X_train, y_train)
        acc_Linear = test(model_Linear, X_test, y_test)
        acc[1].append(acc_Linear)
        print("Final Accuracy: %.2f%%\n" % (acc_Linear * 100))

    plot_bar(acc, 'Learning Rate', lr)
```

七、参考文献

[1] 刘远超. 深度学习基础: 高等教育出版社, 2023.