華中科技大學

# 人工智能导论课程论文

## 基于 PyTorch 的多层感知器模型对噪声同心圆数据进行分类的研究与可视化


院　　系　软件学院

专业班级　软件 2102 班

姓　　名

学　　号

指导教师　GANG SHEN


2023 年 11 月 21 日

# 学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名：　　年　　月　　日

# 学位论文版权使用授权书

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保密　□，在　　年解密后适用本授权书。

2、不保密 ☑。

（请在以上相应方框内打“✓”）

作者签名：　　年　　月　　日

导师签名：　　年　　月　　日

# 摘　　要

随着机器学习技术的不断进步，多层感知器（MLP）模型已被广泛应用于解决非线性可分问题。本论文利用 PyTorch 框架实现了一个简单的 MLP 模型，旨在探究其对于具有非线性边界的同心圆数据集的分类性能。通过引入噪声，我们增加了分类任务的复杂度，更贴近实际应用中的数据分布情况。实验过程中，模型结构设计、训练及优化策略被详细记录和分析。我们采用了 Sigmoid 激活函数，通过二维和三维的数据可视化方法，深入探讨了隐层表示和权重更新过程。结果表明，MLP 模型在测试集上达到了较高的准确率，并展示出良好的收敛性质。本论文为深入理解 MLP 模型的内部机制和改进提供了实验基础和视觉证据。

**关键词**：多层感知器；非线性分类；PyTorch；同心圆数据集；数据可视化；Sigmoid 激活函数

# Abstract

Abstract: With the continuous advancement of machine learning technologies, Multilayer Perceptrons (MLPs) have been extensively applied to tackle non-linearly separable problems. This paper utilizes the PyTorch framework to implement a straightforward MLP model, aimed at investigating its classification performance on a concentric circles dataset with nonlinear boundaries. By introducing noise, the complexity of the classification task is increased to better resemble the data distribution found in practical applications. The model structure design, training, and optimization strategies are meticulously recorded and analyzed throughout the experimental process. Employing the Sigmoid activation function, the study delves into the hidden layer representations and the weight update process through two-dimensional and three-dimensional data visualization techniques. The results indicate that the MLP model achieves a high accuracy rate on the test set and demonstrates good convergence properties. This research provides an empirical foundation and visual evidence for a deeper understanding of the MLP model's internal mechanisms and potential improvements.

**Key Words:** Multilayer Perceptron; Nonlinear Classification; PyTorch; Concentric Circles Dataset; Data Visualization; Sigmoid Activation Function

# 目　　录

# 1 引言

## 1.1 问题陈述

在机器学习领域，对平面不可分数据进行准确分类一直是一个极具挑战性的问题。这一问题在数据展现出复杂的非线性分布特征时尤为显著，因为传统的线性分类器在这种情况下往往力不从心。为了克服这一障碍，深度学习模型，尤其是多层感知器（Multilayer Perceptron, MLP），被引入作为解决方案。MLP 通过嵌入多个隐藏层，具备学习数据中非线性关系的能力，从而有效地对复杂结构的数据进行分类。

## 1.2 目标

本项研究的核心目标在于构建并分析一个基于 PyTorch 实现的多层感知器模型，该模型将处理一个注入噪声的二维同心圆数据集，并以 Sigmoid 激活函数作为非线性变换的工具。通过深入的可视化分析，本研究旨在揭示隐层数据分布的内在特征以及权重在训练过程中的动态变化，以此来评估模型在非线性分类任务中的表现和学习效率。

## 1.3 结构

本报告的结构安排如下：首先，将对使用的数据集进行介绍，并阐述其预处理步骤；继而，详细说明所构建的多层感知器模型的结构与组件；随后，利用数据可视化技术详细展示隐层数据的分布和权重的迭代变化；进一步，基于实验结果对模型的分类准确性和学习能力进行深入分析；最终，讨论研究成果的学术及应用意义，并对模型的潜在优化方向提出建设性见解。报告的最后将总结本研究的关键发现，并提供完整的参考文献列表。

# 2 数据集介绍

## 2.1 数据来源

本研究所使用的数据集是基于 `sklearn.datasets` 库中的 `make_circles` 函数生成的，此函数专门用于产生一个二维平面上包含两个同心圆的数据分布。为了使得分类任务的难度增加，我们引入了随机噪声，通过调整 `noise` 参数的值来控制噪声的强度。此外，`factor` 参数的设定用于调节两个同心圆之间的间隔，以此来模拟真实世界数据的非线性特征和复杂性。如图 2-1 所示。

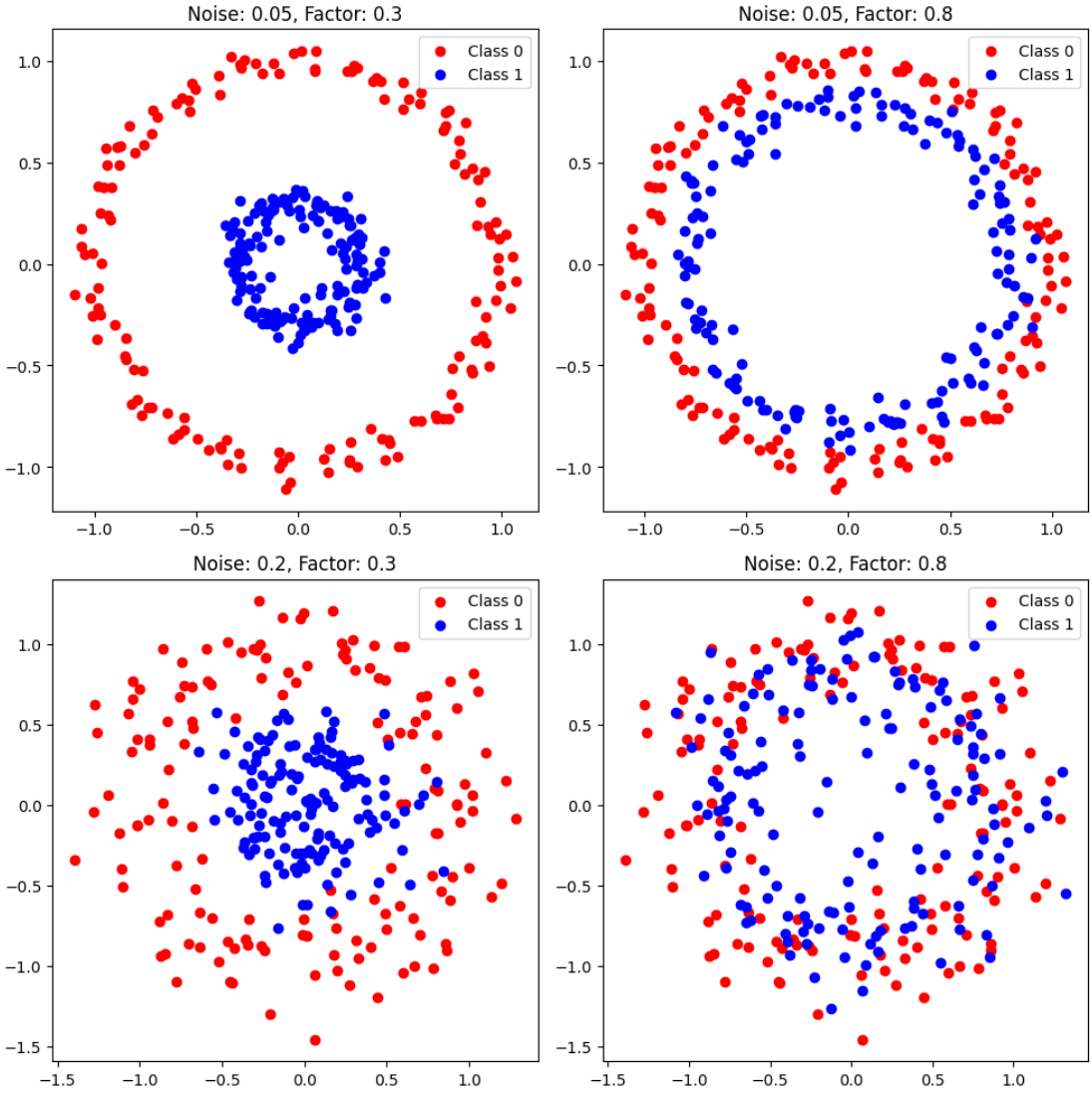


图 2-1 make_circles 函数示例

## 2.2 数据预处理

在数据预处理阶段，我们首先需要将数据转换为适合 PyTorch 框架处理的格式，即将数据集中的特征和标签转换为 PyTorch 张量格式。接着，我们按照 80% 和 20% 的比例将数据集分割为训练集和测试集，这样的划分策略确保了模型在训练过程中能够接触到充足的数据样本，并为后续的模型评估保留了一定数量的未知数据，以测试模型的泛化性能。

以下是数据生成与预处理的代码实现：

```
# Generate the dataset with two concentric circles
X, y = make_circles(n_samples=500, noise=0.075, factor=0.5, random_state=2023)

# Convert the dataset to PyTorch tensors
X = torch.tensor(X, dtype=torch.float)
y = torch.tensor(y.reshape(-1, 1), dtype=torch.float)

# Split the dataset into training and testing sets
split = int(0.8 * len(X))
X_train, y_train = X[:split], y[:split]
X_test, y_test = X[split:], y[split:]

```

经过预处理，我们得到了以下数据集的基本统计信息：

- 整个数据集包含 500 个样本点。
- 每个样本点有两个特征，分别对应于二维平面上的 $x$ 和 $y$ 坐标。
- 数据集中的标签分为 0 和 1 两类，代表两个不同的类别。
- 训练集和测试集的划分比例分别为 80% 和 20%，其中训练集包含 400 个样本点，测试集包含 100 个样本点。

# 3 模型架构

## 3.1 多层感知器模型

在深度学习的范畴内，多层感知器（MLP）模型是最早期和最基础的架构之一，它通过堆叠多个层次的神经元来构建一个能够捕捉数据复杂特征的网络。在本项研究中，所采用的多层感知器模型是由一个输入层、一个隐层及一个输出层构成的简单但强大的网络结构。具体而言，输入层负责接收原始的二维数据点，隐层由四个神经元组成，并负责进行特征转换和非线性组合。最后，输出层由单个神经元构成，其任务是综合隐层的信息并输出预测的分类结果。此模型的特点在于其隐层神经元能够接收前一层所有神经元的加权输入，并通过非线性激活函数进行转换，从而实现从数据到特征的有效映射。如图 3-1 所示。

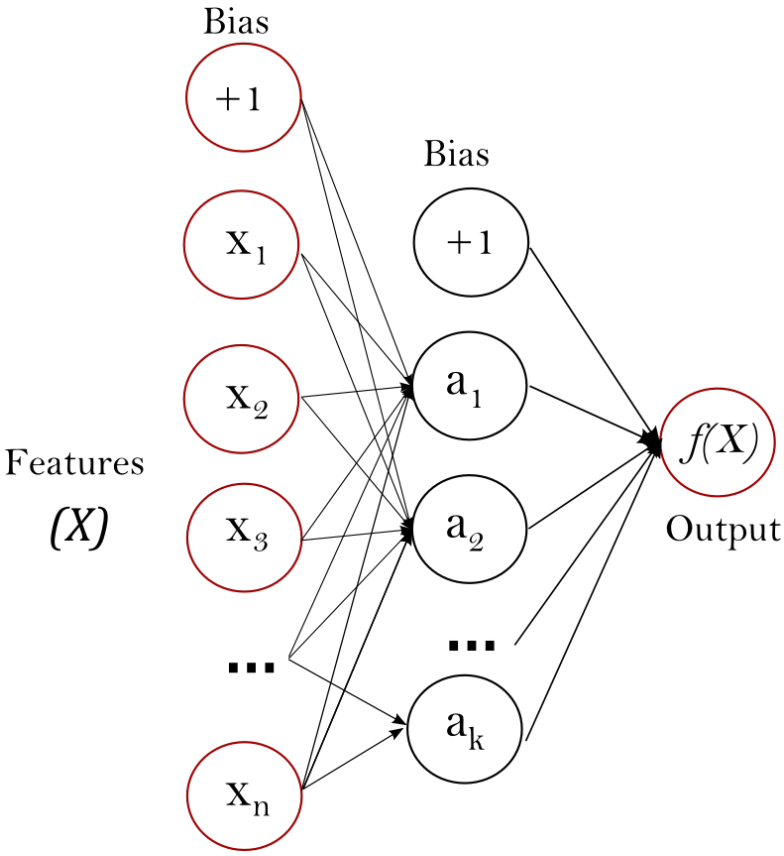


图 3-1 单隐藏层 MLP 模型

## 3.2 激活函数选择

在本模型中，我们采用了 Sigmoid 函数作为神经元的激活函数，其定义如式 (3.1) 所示：

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.1}$$

Sigmoid 函数，形似 S 形的逻辑函数，它将任意实值映射到区间（0,1）内，并且其输出具有平滑连续的特性。这使得 Sigmoid 函数在二分类任务中尤为有价值，因为它可以将模型输出解释为概率。此外，Sigmoid 函数在其导数中具有自洽性，即导数可以用函数本身表示，这对于利用梯度下降算法在训练过程中的误差反向传播是有益的。尽管在更深层的网络结构中，Sigmoid 函数可能会导致梯度消失的问题，但在我们的浅层网络中，这一风险较小。因此，选择 Sigmoid 函数作为激活函数是恰当的，既能够实现非线性变换，又能够保持网络的训练稳定性。如图 3-2 所示。

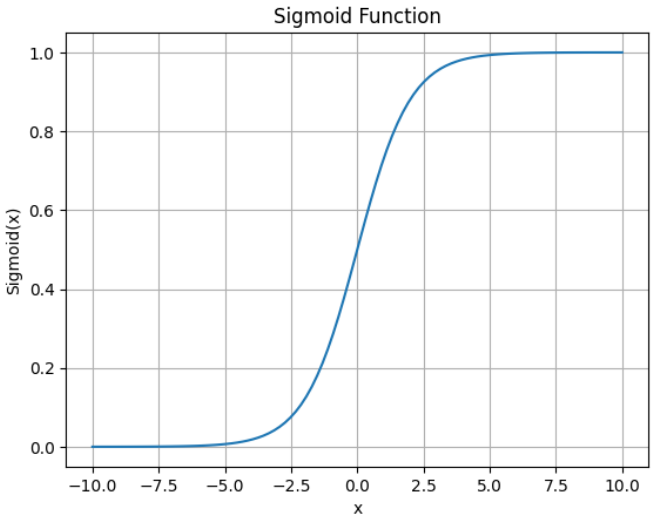


**图 3-2 Sigmoid 函数示意图**

以下是用于构建和初始化我们的多层感知器模型的代码，包括了模型定义、损失函数的选择以及优化器的配置。

```
# Create the neural network model
class BinaryClassifier(nn.Module):
  def __init__(self):
    super(BinaryClassifier, self).__init__()
    self.fc1 = nn.Linear(2, 4)
    self.fc2 = nn.Linear(4, 1)
```

```
7
8       def forward(self, x):
9           x = torch.sigmoid(self.fc1(x))
10          x = torch.sigmoid(self.fc2(x))
11          return x
12
13  # Create an instance of the model
14  model = BinaryClassifier()
15
16  # Define the loss function and optimizer
17  criterion = nn.MSELoss()
18  optimizer = optim.Adam(model.parameters(), lr=0.02)
19
```

# 4 数据可视化分析

## 4.1 隐层数据分布可视化

为了深入理解多层感知器模型的内部机制，本研究采用了隐层数据分布的可视化方法。通过这种方式，我们能够直观地观察模型如何处理和变换输入数据，以及隐层如何对不同类别的数据进行区分。

### 4.1.1 数据处理

我们的数据处理步骤包括将输入数据传递至模型的隐层并记录其输出值。具体来说，我们使用模型的前向传播过程仅计算到隐层的输出，忽略最后的输出层。这允许我们观察隐层如何在不受输出层影响的情况下表示数据。为此，我们先修改 BinaryClassifier 这个类，保存保存隐层输出以便可视化：

```
class BinaryClassifier(nn.Module):
  def __init__(self):
    super(BinaryClassifier, self).__init__()
    self.fc1 = nn.Linear(2, 4)
    self.fc2 = nn.Linear(4, 1)

  def forward(self, x):
    self.hidden_output = torch.sigmoid(self.fc1(x))   # 保存隐层输出以便可视化
    x = torch.sigmoid(self.fc2(self.hidden_output))
    return x

```

### 4.1.2 图表展示

为了可视化隐层数据，我们绘制了隐层数据在二维平面和三维空间中的可视化图形。通过将隐层输出使用降维技术映射到二维或三维空间，展示模型是如何区分不同类别的数据。这种可视化揭示了模型内部的学习动态，并帮助我们理解其非线性映射能力。数据经 Sigmoid 函数处理前后，可视化结果如图 4-1 与图 4-2 所示：

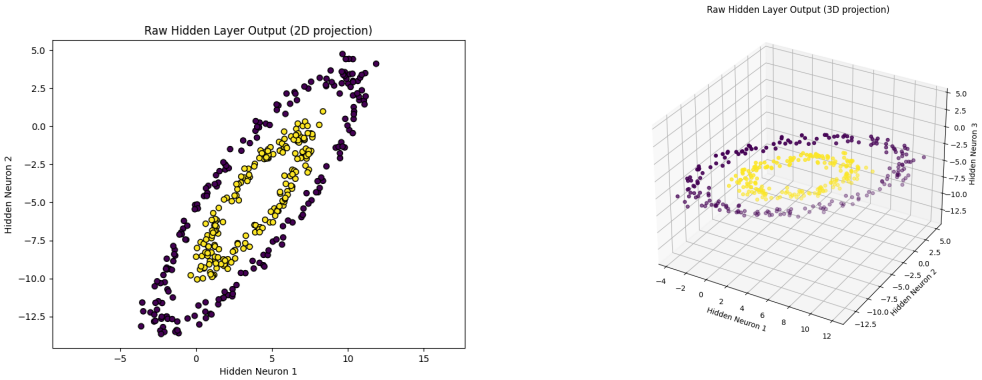


图 4-1　经 sigmoid 处理之前的数据可视化结果

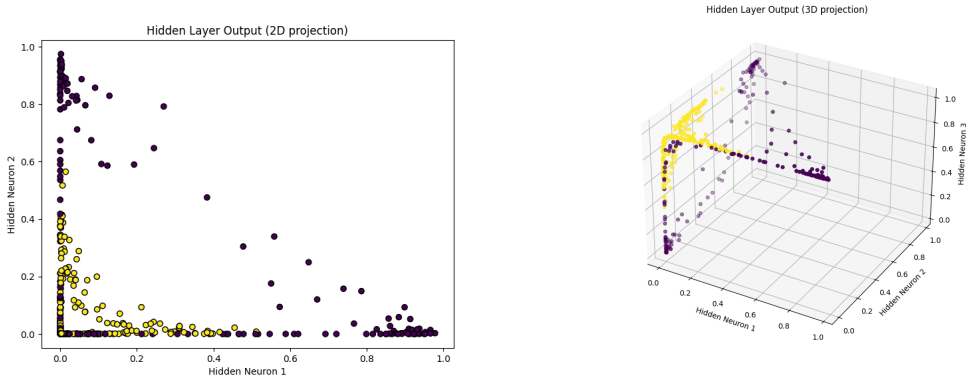


图 4-2　经 sigmoid 处理之后的数据可视化结果

## 4.2　权重变化过程可视化

权重变化的可视化是理解神经网络训练过程的关键。它展示了模型在学习过程中是如何调整其参数以适应数据的。

### 4.2.1　训练过程记录

我们在每个训练周期（epoch）后记录了隐层神经元的权重值。这一记录过程允许我们追踪模型在整个训练期间权重的变化情况，从而了解模型学习的动态。代码如下：

```
def visualize_weights(model, epochs=250):
    weights = []
    for epoch in range(epochs):
    # 记录权重
        weights.append(model.fc1.weight.data.numpy().copy())
```

```
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch {epoch+1}/{120}, Loss: {loss.item():.4f}')

    # 将权重变化绘制成图
    weights = np.array(weights)
    for i in range(weights.shape[1]):
        plt.plot(weights[:, i], label=f'Weight{i+1}')
    plt.title('Weight Change Curve')
    plt.xlabel('Epoch')
    plt.ylabel('Weights')
    plt.legend()
    plt.show()

# 可视化权重变化
visualize_weights(model)

```

### 4.2.2 图表展示

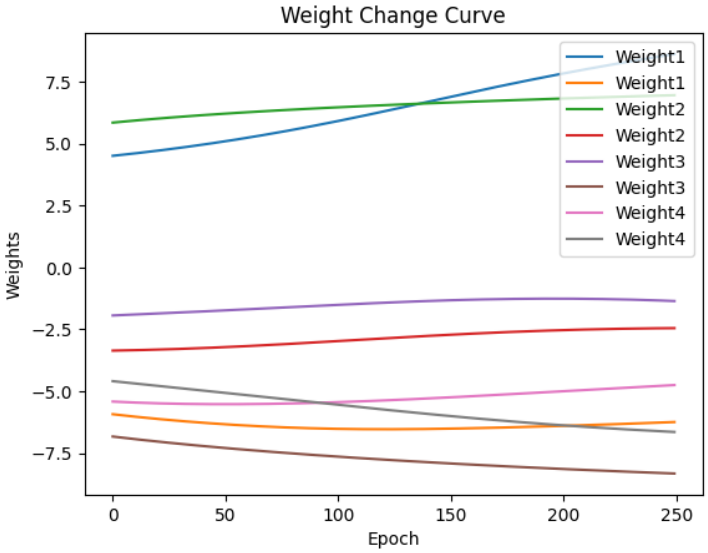


图 4-3 权重变化曲线

如图 4-3 所示。我们观察到，在训练的初始阶段，权重的变化较为显著，这反映了模型在学习过程中对数据特征的快速适应。随着训练周期的增加，权重调整逐渐减小，显示出模型对数据集的特征逐渐达到了较稳定的理解。这种权重变化的趋势提供了对神经网络训练动态的直观理解，并证明了所采用的训练方法能有效促进模型学习。通过这种方法，我们不仅能够评估模型的最终性能，还能深入了解训练过程中权重调整的内在机制，这对于理解和优化神经网络模型具有重要意义。

# 5 实验结果与分析

## 5.1 隐层数据分布分析结果

通过对隐层数据分布的定量分析，本研究旨在评估模型在分类任务中的效能。隐层的主要作用是提取输入数据的特征，并进行必要的非线性转换，从而使得模型能够区分原本在特征空间中不可分的数据。

### 5.1.1 数据分布特点

在本实验中，我们观察到隐层输出数据经过降维处理后在二维或三维空间中表现出明显的类别分离。这表明隐层神经元能够有效地学习并映射数据的内在结构，从而使得不同类别的数据在新的特征空间中被有效区分，如图 5-1 与图 5-2 所示。

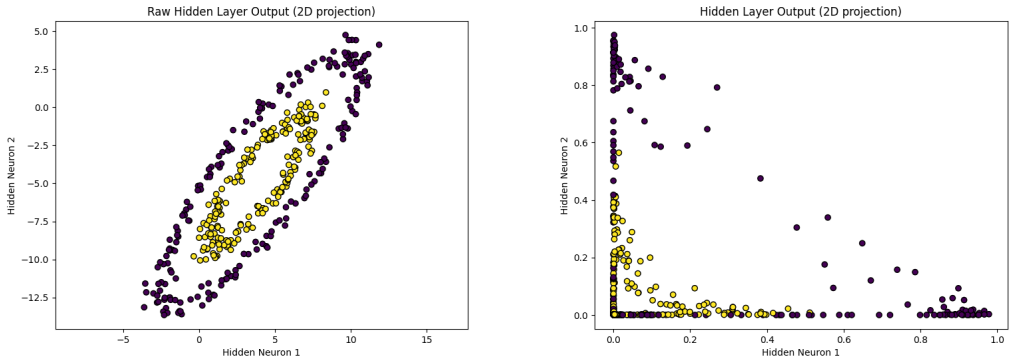


图 5-1 经 sigmoid 处理前后的二维数据可视化结果

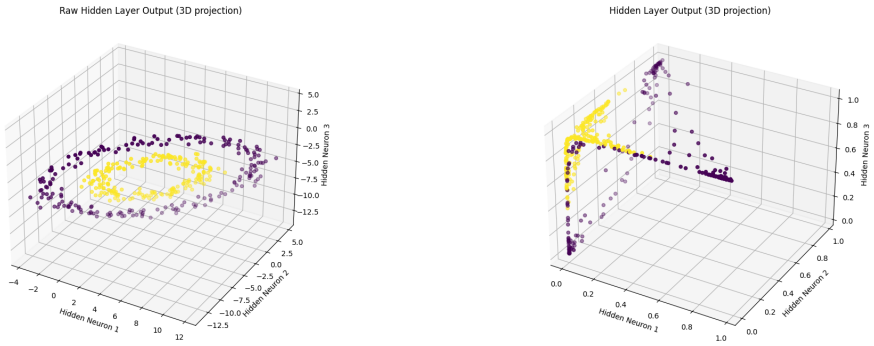


图 5-2 经 sigmoid 处理前后的三维数据可视化结果

### 5.1.2 分类效果观察

通过对隐层数据分布的观察，我们发现模型能够将具有复杂非线性关系的数据有效分类。隐层神经元通过对二维的数据进行升维处理（升维到四维空间），使得在二维空间中稠密在四维空间中变得稀疏。由于我们无法直观到了解到四维空间中的数据分布，我们将四维空间的数据投影到三维空间进行观察：我们可以发现将数据进行升维后，可以十分轻松的用超平面（Hyperplane）将两类数据分割开来。在三维空间中，隐层输出展现出清晰的边界线，可以用二维平面将其分割开（四维空间中同理）；随后将高维空间中的数据投影到二维空间，我们可以发现超平面成功的将两类数据分割开来，说明了模型对于这种非线性数据具有较好的处理能力和分类准确性，如图 5-3 所示。[1]

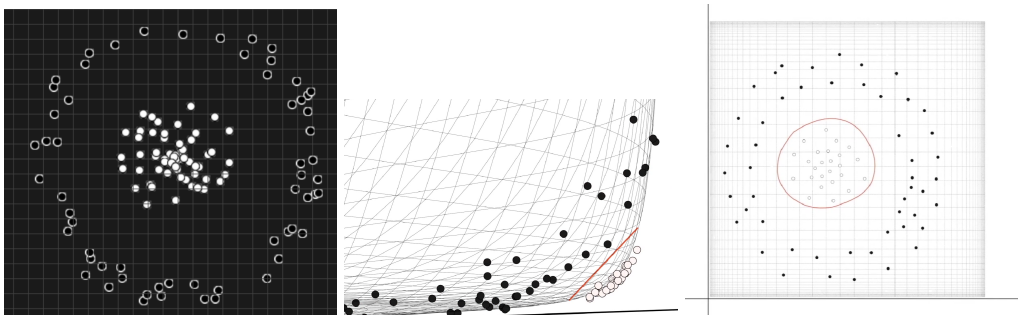

图 5-3 数据升维与投影

## 5.2 权重变化分析结果

权重变化的分析揭示了模型在学习过程中的动态变化，特别是其如何适应数据特征和整体的收敛情况。

### 5.2.1 权重变化趋势

在训练的初期，我们观察到权重的变化比较显著。这表明模型正在积极学习并适应数据集中的特征。随着训练的进行，权重变化逐渐减小并趋于稳定，这不仅显示了模型对数据特征的深入和稳定理解，也反映了其优化策略的有效性。权重变化曲线如图 5-4 所示，清晰地展示了这一过程。

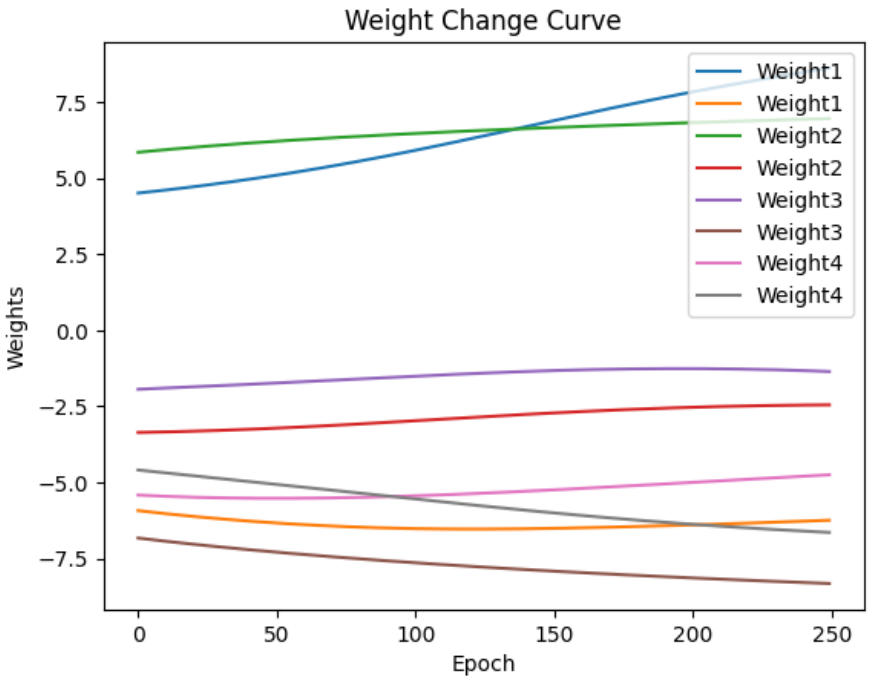


**图 5-4　权重变化曲线**

### 5.2.2　模型学习能力和收敛情况

权重变化的趋势不仅展示了模型在训练过程中的学习能力，还揭示了其收敛情况。初始阶段权重的快速变化是模型对数据特征的初步学习和探索，而后期权重的稳定则表明模型已经找到了最优或接近最优的参数配置来最小化损失函数。这一过程不仅体现了模型优化算法的效率，也暗示了其良好的泛化能力。一个有效收敛的模型能够更好地泛化到未见数据上，从而在实际应用中展现出更高的可靠性和鲁棒性。

此外，权重变化的趋势还可以反映模型训练过程中可能遇到的挑战，如梯度消失或爆炸。在本研究中，权重的平稳收敛表明这些常见问题得到了有效的控制和缓解。这可能归因于模型结构设计的合理性以及优化算法的选择，如使用了适应性学习率的优化器等。总的来说，这些发现对于深入理解和改进神经网络模型具有重要的意义。

# 6 讨论与结论

## 6.1 结果分析

### 6.1.1 模型适应性

多层感知器模型显示出对具有复杂非线性特征数据的高效处理能力。实验结果表明，模型能够通过其隐层神经元有效地对输入数据进行非线性转换，使得原本在输入空间中不可分的数据在隐层空间得到有效分离。

### 6.1.2 非线性表示能力

隐层神经元的表示能力对于提高分类性能至关重要。在本研究中，隐层神经元能够学习和提取输入数据中的复杂特征，这在隐层数据分布的可视化中得到了明确的证明。

### 6.1.3 权重收敛性

权重变化过程展示了模型的学习能力和收敛情况。在训练初期，权重的迅速变化反映了模型对数据特征的积极学习；随着训练的进行，权重变化逐渐减小并趋于稳定，显示了模型的收敛和泛化能力。

### 6.1.4 对三同心圆数据分类的考虑

如果研究问题扩展到对三个同心圆的数据分类，可能需要对现有的神经网络结构进行调整。具体来说，可能需要增加更多的隐层神经元，甚至引入额外的隐层，以增强模型捕捉更复杂数据结构的能力。此外，也可以考虑采用更复杂的激活函数，如 ReLU 或 LeakyReLU，以提高模型对于更复杂数据特征的学习能力。

## 6.2 结果总结

本研究的主要发现强调了多层感知器模型在处理具有复杂非线性特征数据时的有效性。以下是实验结果的关键总结及其对研究问题的启示和意义：

### 6.2.1 隐层数据分布的重要性

本研究明确展示了隐层数据分布在分类任务中的重要性。隐层神经元能够对原始数据进行有效的非线性变换，将在原始特征空间中难以区分的类别，在隐层空间中进行有效分离。这一发现不仅证实了多层感知器在处理非线性可分问题上的能力，同时也强调了隐层结构设计的重要性。

### 6.2.2 权重变化与模型学习能力

权重变化的分析揭示了模型的学习过程和收敛性。权重的初期快速变化显示了模型对数据特征的迅速学习，而随着训练的进行，权重变化的减缓和稳定则表明了模型的收敛和泛化能力。这一点对于理解和优化神经网络模型的训练过程至关重要。

### 6.2.3 对更复杂数据结构的适应能力

本研究还探讨了多层感知器模型在处理更复杂数据结构，如三个同心圆数据集时的潜在适应能力。这一探索为未来的研究提供了方向，即通过增加隐层神经元数量或引入更复杂的激活函数来处理更高维度和更复杂的数据结构。

综上，这些结果不仅证实了多层感知器模型在处理非线性分类问题上的有效性，而且为未来在更复杂数据集上应用深度学习提供了重要的理论和实践基础。

# 7 结果总结

为了提供一个清晰的概览，本部分通过表格形式总结了实验的关键结果。这些结果概括了多层感知器模型在处理二维同心圆数据集时的表现，包括训练损失、测试损失以及准确率，反映了模型在非线性分类任务上的有效性。

| 指标 | 值 |
| --- | --- |
| 训练损失 | 0.0488 |
| 测试损失 | 0.0482 |
| 准确率 | 100.00% |

**表 7-1 实验结果总结**

在表 7-1 中，我们展示了模型在训练和测试阶段的性能。训练损失和测试损失指标反映了模型在拟合训练数据和泛化到新数据上的能力。准确率则直接显示了模型在分类任务上的效能。这些指标共同揭示了模型的总体表现，并为进一步的分析和讨论提供了基础。

通过对这些关键指标的观察，我们可以得出模型在处理复杂非线性数据集时具有良好的性能和稳定的泛化能力的结论。未来的研究可以在这一基础上进一步探索不同模型结构和训练策略对结果的影响。

# 8 结果图示

为了更直观地展示模型的学习过程及其效果，报告中将包含以下图表：

1. **图 8-1**：经 sigmoid 处理之前的数据可视化结果，数据呈现出明显的同心圆分布。

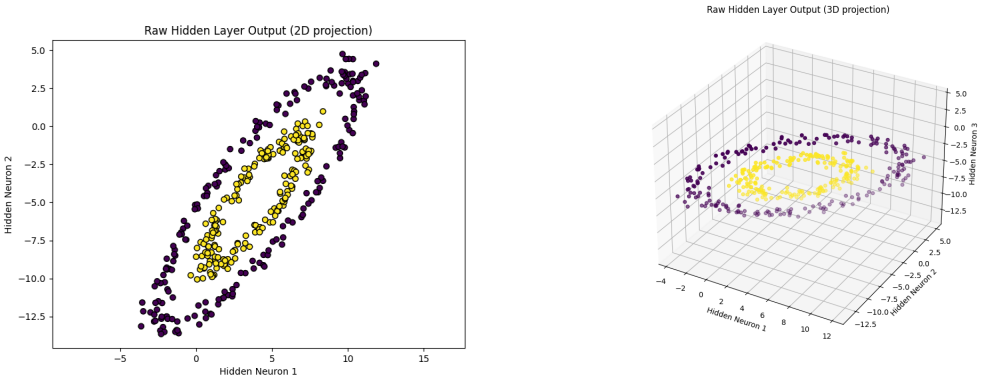


**图 8-1　经 sigmoid 处理之前的数据可视化结果**

2. **图 8-2：**经 sigmoid 处理之后的数据可视化结果，数据呈现出明显的可分割性。

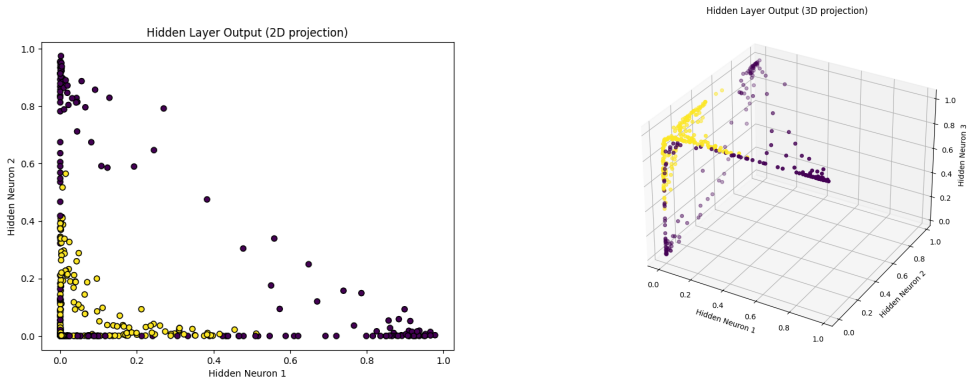


**图 8-2　经 sigmoid 处理之后的数据可视化结果**

3. **图 8-3：**最终的分类结果可视化。

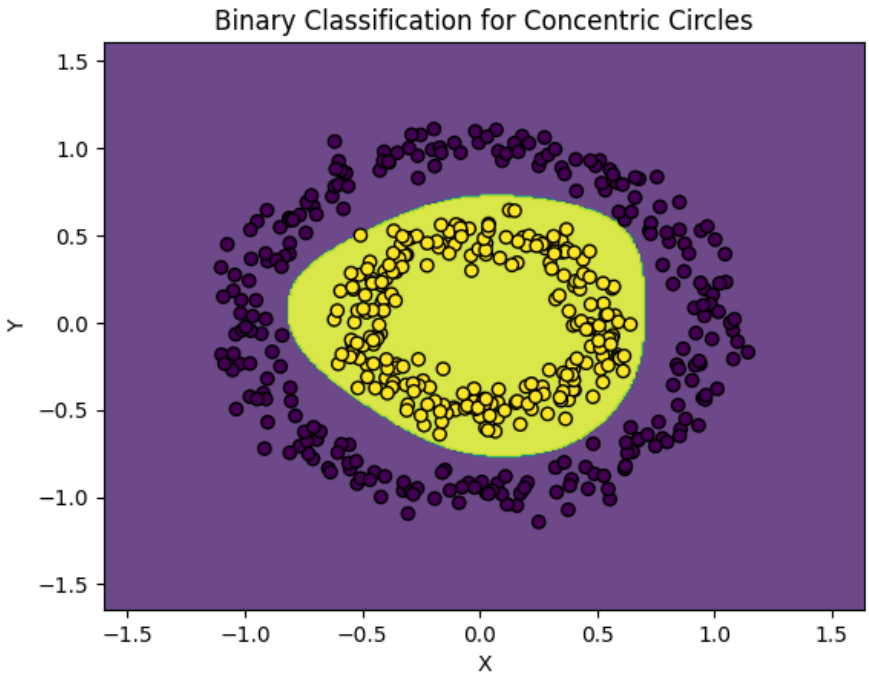


**图 8-3　分类结果**

4. **图 8-4** 模型权重随训练周期变化的趋势图，揭示了学习过程中权重调整的动态。

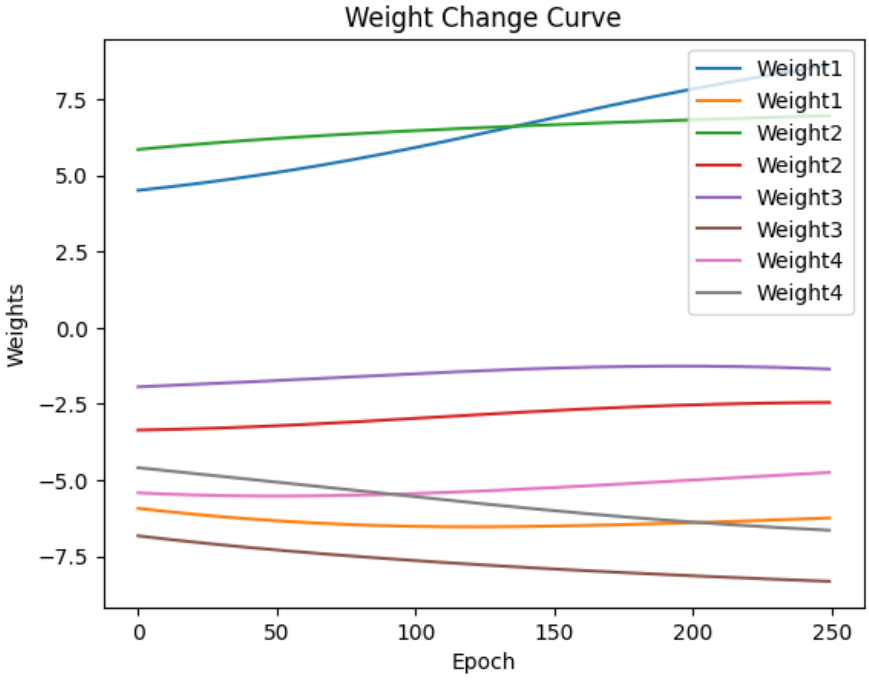


**图 8-4　权重变化曲线**

这些图表直观地展现了模型训练过程中隐层数据的演变以及权重调整的轨迹，有助于读者深入理解模型的学习机制和决策过程。

# 致谢

首先，我要感谢我的导师沈刚教授，感谢他的专业指导和无私帮助。在整个研究过程中，沈刚教授不仅提供了宝贵的建议，而且给予了我持续的鼓励和支持。

我还要特别感谢软件学院的所有老师。在研究期间，他们提供了无数的帮助、建议和灵感。他们的专业知识和热情对我完成这项研究至关重要。

最后，我要感谢我的家人和朋友们。他们的爱和支持是我在研究过程中不可或缺的力量来源。特别是在遇到困难和挑战时，他们给予了我巨大的鼓励和安慰。

再次感谢所有帮助和支持我的人。

# 参考文献

[1] DEKRT'S BLOG. "AI Learning Notes." Retrieved from https://dekrt.cn/note/ai1-3/

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

[3] Haykin, S. O. (1998). Neural Networks: A Comprehensive Foundation. Prentice Hall.

[4] Tufte, E. R. (2001). The Visual Display of Quantitative Information. Graphics Press.

[5] Paszke, A., Gross, S., Massa, F., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems.

[6] Jolliffe, I. T. (2002). Principal Component Analysis. Springer.

[7] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.

# 附录 A　代码文件

```python
# import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

# Generate the dataset with two concentric circles
X, y = make_circles(n_samples=500, noise=0.075, factor=0.5, random_state
  =2023)

# Convert the dataset to PyTorch tensors
X = torch.tensor(X, dtype=torch.float)
y = torch.tensor(y.reshape(-1, 1), dtype=torch.float)

# Split the dataset into training and testing sets
split = int(0.8 * len(X))
X_train, y_train = X[:split], y[:split]
X_test, y_test = X[split:], y[split:]

# Create the neural network model
class BinaryClassifier(nn.Module):
  def __init__(self):
    super(BinaryClassifier, self).__init__()
    self.fc1 = nn.Linear(2, 4)
    self.fc2 = nn.Linear(4, 1)

  def forward(self, x):
    x = torch.sigmoid(self.fc1(x))
    x = torch.sigmoid(self.fc2(x))
    return x

# Create an instance of the model
model = BinaryClassifier()

# Define the loss function and optimizer
criterion = nn.MSELoss()
```

```
optimizer = optim.Adam(model.parameters(), lr=0.02)

# Train the model
for epoch in range(250):
  optimizer.zero_grad()
  outputs = model(X_train)
  loss = criterion(outputs, y_train)
  loss.backward()
  optimizer.step()

  if (epoch + 1) % 10 == 0:
    print(f'Epoch {epoch+1}/{120}, Loss: {loss.item():.4f}')

# Evaluate the model on the testing set
with torch.no_grad():
  test_outputs = model(X_test)
  test_loss = criterion(test_outputs, y_test)
  predicted_classes = (test_outputs > 0.5).float()
  accuracy = (predicted_classes == y_test).float().mean()

print(f'Test Loss: {test_loss.item():.4f}')
print(f'Test Accuracy: {accuracy.item()*100:.2f}%')

# Plot the decision boundary
x1_min, x1_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
x2_min, x2_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, 0.01), np.arange(x2_min,
  x2_max, 0.01))
Z = model(torch.tensor(np.c_[xx1.ravel(), xx2.ravel()], dtype=torch.float)
  )
Z = (Z > 0.5).float().reshape(xx1.shape)

plt.contourf(xx1, xx2, Z, alpha=0.8)
plt.scatter(X[:, 0], X[:, 1], c=y.squeeze(), edgecolors='k')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Binary Classification for Concentric Circles')
plt.show()

# def visualize_hidden_layer(model, X, y):
```

```
    # 获取隐层的输出
    with torch.no_grad():
        hidden_layer_output = model.fc1(X).detach().numpy()

    # 使用PCA降维到二维空间
    from sklearn.decomposition import PCA
    pca = PCA(n_components=2)
    reduced_data = pca.fit_transform(hidden_layer_output)

    # 绘制降维后的数据点
    plt.scatter(reduced_data[:, 0], reduced_data[:, 1], c=y.squeeze(),
    edgecolors='k', alpha=0.5)
    plt.title('Hidden Layer Visualization')
    plt.xlabel('Main Ingredient-1')
    plt.ylabel('Main Ingredient-2')
    plt.show()

# 可视化训练集的隐层数据分布
visualize_hidden_layer(model, X_train, y_train)


# def visualize_weights(model, epochs=250):
    weights = []
    for epoch in range(epochs):
        # 记录权重
        weights.append(model.fc1.weight.data.numpy().copy())
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch {epoch+1}/{120}, Loss: {loss.item():.4f}')

    # 将权重变化绘制成图
    weights = np.array(weights)
    for i in range(weights.shape[1]):
        plt.plot(weights[:, i], label=f'Weight{i+1}')
    plt.title('Weight Change Curve')
```

```
    plt.xlabel('Epoch')
    plt.ylabel('Weights')
    plt.legend()
    plt.show()

# 可视化权重变化
visualize_weights(model)


# from sklearn.datasets import make_circles
import matplotlib.pyplot as plt

# Function to create and plot circles with noise
def plot_circles(noise_level, factor, ax):
    # Generate the dataset with specified noise and factor
    X, y = make_circles(n_samples=300, factor=factor, noise=noise_level,
    random_state=42)

    # Scatter plot for class 0 and class 1
    ax.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='red', label='Class 0
    ')
    ax.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='blue', label='Class
    1')
    ax.set_title(f"Noise: {noise_level}, Factor: {factor}")
    ax.legend()

# Create figure with subplots for different noise levels and factors
fig, axs = plt.subplots(2, 2, figsize=(10, 10))

# Plot 1: Low noise, small factor
plot_circles(0.05, 0.3, axs[0, 0])

# Plot 2: Low noise, large factor
plot_circles(0.05, 0.8, axs[0, 1])

# Plot 3: High noise, small factor
plot_circles(0.2, 0.3, axs[1, 0])

# Plot 4: High noise, large factor
plot_circles(0.2, 0.8, axs[1, 1])
```

```
151
152 # Adjust layout
153 plt.tight_layout()
154
155 # Save the figure
156 plt.savefig('circles_dataset_examples.png')
157
158 # Show the plots
159 plt.show()
160
161
162 # import matplotlib.pyplot as plt
163 import numpy as np
164
165 # Sigmoid函数定义
166 def sigmoid(x):
167     return 1 / (1 + np.exp(-x))
168
169 # 创建一个数值范围
170 x = np.linspace(-10, 10, 100)
171 # 计算Sigmoid函数值
172 y = sigmoid(x)
173
174 # 绘制Sigmoid函数
175 plt.plot(x, y)
176 plt.title('Sigmoid Function')
177 plt.xlabel('x')
178 plt.ylabel('Sigmoid(x)')
179 plt.grid(True)
180 plt.show()
181
182
183 # 通过隐层传递输入数据并记录输出
184 def extract_hidden_layer_output(model, input_data):
185     with torch.no_grad():
186         hidden_layer_output = torch.sigmoid(model.fc1(input_data))
187     return hidden_layer_output.numpy()
188
189 hidden_layer_output_train = extract_hidden_layer_output(model, X_train)
190 hidden_layer_output_test = extract_hidden_layer_output(model, X_test)
```

```
191
192 # 二维可视化
193 plt.figure(figsize=(10, 5))
194 plt.scatter(hidden_layer_output_train[:, 0], hidden_layer_output_train[:,
      1], c=y_train.squeeze(), cmap='viridis', alpha=0.5)
195 plt.title('Hidden Layer Output (2D Visualization)')
196 plt.xlabel('Hidden Unit 1')
197 plt.ylabel('Hidden Unit 2')
198 plt.colorbar()
199 plt.show()
200
201
202 # from mpl_toolkits.mplot3d import Axes3D
203
204 # 仅在隐层超过2个神经元时三维可视化才有意义
205 if hidden_layer_output_train.shape[1] > 2:
206   fig = plt.figure(figsize=(10, 7))
207   ax = fig.add_subplot(111, projection='3d')
208   ax.scatter(hidden_layer_output_train[:, 0], hidden_layer_output_train[:,
       1], hidden_layer_output_train[:, 2], c=y_train.squeeze(), cmap='viridis'
      , alpha=0.5)
209   ax.set_title('Hidden Layer Output (3D Visualization)')
210   ax.set_xlabel('Hidden Unit 1')
211   ax.set_ylabel('Hidden Unit 2')
212   ax.set_zlabel('Hidden Unit 3')
213   plt.show()
214
215
```