 **手写数字识别**

Python语言设计

**分**

**析**

**报**

**告**

**专业班级：自卓2201班**

**姓名：杨欣怡**

**学号：U202215067**

**指导老师：卢仁智**

**上交时间：2023.10.19**

目录

**[第一部分 前言 2](#_Toc14812)**

[一、 编写背景与算法功能 2](#_Toc31570)

[二、 训练数据与测试数据 2](#_Toc22933)

**[第二部分 算法设计 3](#_Toc27469)**

[一、 神经网络结构 3](#_Toc22976)

[二、 神经网络的学习 7](#_Toc22846)

**[第三部分 运行结果 8](#_Toc29999)**

**[第四部分 源代码 9](#_Toc27920)**

# 前言

## 一、 **编写背景与算法功能**

近年来，深度学习的浪潮席卷全球，渗透与人们生活息息相关的各个领域，与深度学习相关的研究成果也层出不穷。为了解深度学习，探求相关知识，我决定从图像识别角度出发，完成一个手写数字识别算法。算法的目标功能为对手写数字图像的识别精度达到99%。

1. **训练数据与测试数据**

训练数据与测试数据来自MNIST数据集，本程序中利用资料中找到的脚本导入mnist数据，该脚本支持从下载MNIST数据集到将这些数据转换成NumPy数组等处理，具体位置在dataset目录下的mnist.py。load\_mnist函数以“(训练图像 ,训练标签 )，(测试图像，测试标签 )”的 形式返回读入的MNIST数据。

# 算法设计

1. **神经网络结构**

这段代码实现了一个简单的多层感知机（MLP）神经网络结构。具体而言，它包含一个输入层、一个隐藏层和一个输出层。

输入层的大小由训练数据集的特征数量决定，即`input\_size = X\_train.shape[1]`。

隐藏层的大小由`hidden\_size`参数指定，这里设置为64。

输出层的大小由训练数据集的标签类别数量决定，即`output\_size = len(np.unique(y\_train))`。

因此，这个神经网络结构的层数为两层（不包括输入层），其中隐藏层的大小为64。

1、定义

多层感知机（MLP）神经网络是一种前馈神经网络，由输入层、隐藏层和输出层组成。

- 输入层：接受来自外部的输入数据，每个输入节点代表一个特征。

- 隐藏层：负责处理输入层的数据，提取特征并进行非线性变换。可以有多个隐藏层，每个隐藏层包含多个神经元。

- 输出层：根据隐藏层的输出，进行最终的预测或分类。

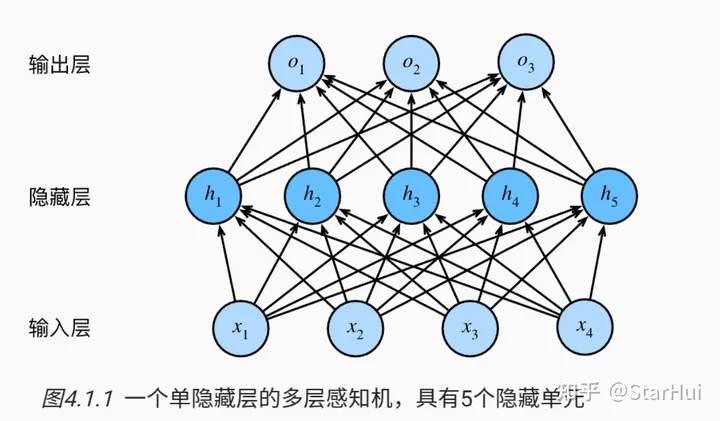
每个神经元都有一组权重和偏置，用于计算输入数据的加权和，并经过激活函数进行非线性变换。常用的激活函数包括Sigmoid、ReLU、Tanh等。

MLP的训练过程主要包括前向传播和反向传播。在前向传播中，输入数据通过网络的各个层，得到最终的预测结果。在反向传播中，根据预测结果与真实标签之间的误差，调整网络中的权重和偏置，以最小化损失函数。

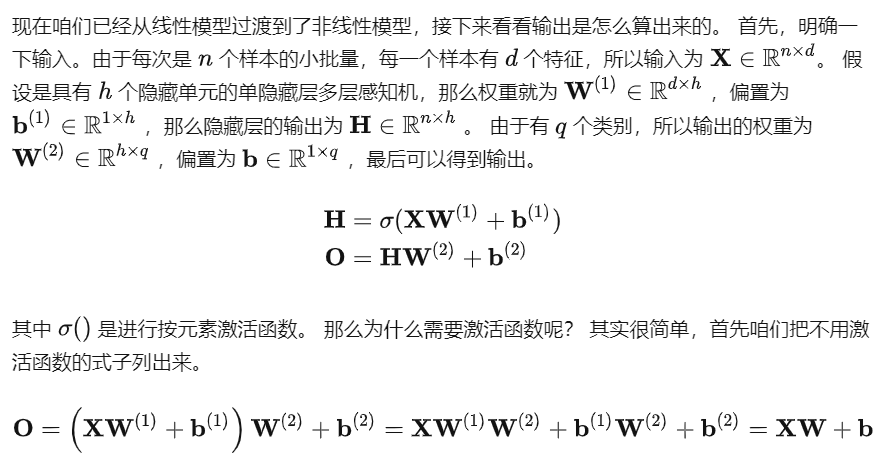
通过反复迭代训练样本集，不断更新网络参数，直到达到预定的训练停止条件，如达到最大迭代次数或损失函数收敛。训练完成后，MLP可以用于对新的未知数据进行预测或分类。

我们可以通过在网络中 加入一个或多个隐藏层 来克服线性模型的限制， 使其能处理更普遍的函数关系类型。 要做到这一点，最简单的方法是将许多全连接层堆叠在一起。 每一层都输出到上面的层，直到生成最后的输出。 我们可以把前

层看作表示，把最后一层看作线性预测器。 这种架构通常称为多层感知机（multilayer perceptron），通常缩写为MLP。



1. 从线性到非线性

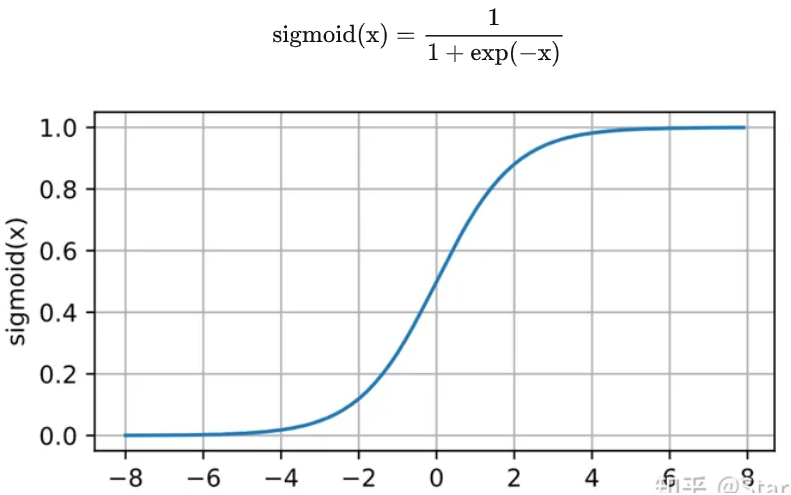


这里不难看出，O与 X还是线性关系，这和单层感知机没有什么区别，就是一个简单的线性模型而已。所以需要加上激活函数，接下来咱们看看有哪些激活函数可以使用。

3、激活函数

（1）sigmod函数

Sigmoid函数将输入值映射到一个范围为(0, 1)的区间，具有平滑的S形曲线。



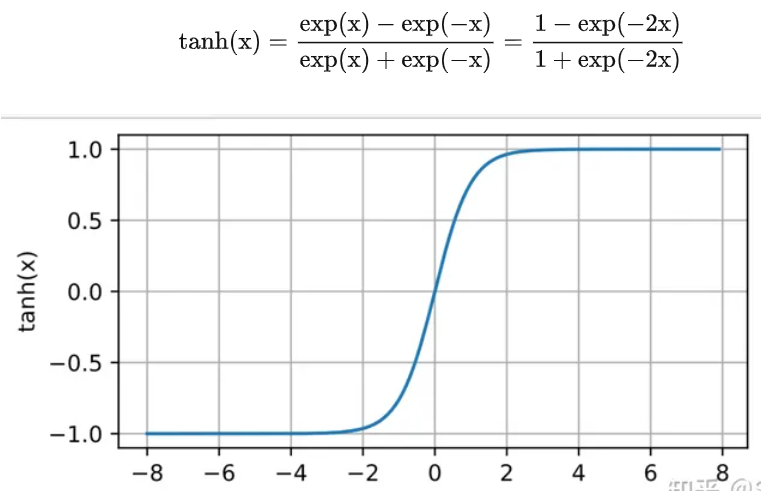
但是是有弊端的。

1）梯度饱和：在Sigmoid函数的两端，梯度接近于0，导致梯度消失问题。这会使得在反向传播时梯度逐渐变小，从而在深度网络中影响参数的更新和收敛速度。

2）输出非零均值：由于Sigmoid函数输出在(0, 1)之间，对于大的负输入，输出接近于0，导致神经元的平均输出值不为0。这可能会对下一层的神经元带来偏置，影响整体网络的学习能力。

（2）tanh函数

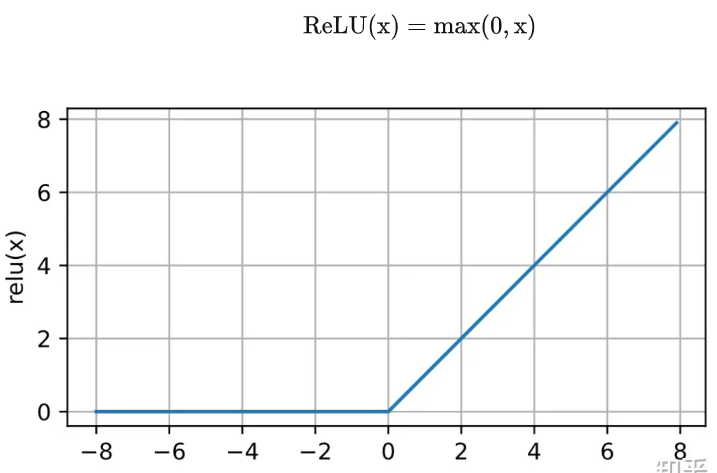
tanh函数将输入值映射到一个范围为(-1, 1)的区间，也具有S形曲线。



弊端：与Sigmoid函数类似，Tanh函数在两端也存在梯度饱和和输出非零均值的问题，会影响神经网络的训练效果。

（3）ReLU函数

ReLU（Rectified Linear Unit）函数在输入大于0时返回该输入值，小于等于0时返回0



ReLU函数有以下两个特点。

1）ReLU函数计算简单高效，不涉及复杂的指数运算。

2）ReLU函数解决了Sigmoid和Tanh函数存在的梯度饱和问题，激活函数的导数在正数区域始终为常数1，可以有效传递梯度。

1. **神经网络的学习**

神经网络的学习方法有很多种，其中最常见的是反向传播算法。下面是一个简单的神经网络学习的步骤：

1. 初始化神经网络的权重和偏置：首先需要随机初始化神经网络的权重和偏置，这些参数将在训练过程中被优化。

2. 前向传播：通过将输入数据输入到神经网络中，计算每个神经元的输出。这个过程中，可以使用激活函数对神经元的输出进行非线性变换。

3. 计算损失函数：将神经网络的输出与真实的标签进行比较，计算损失函数。常见的损失函数包括均方误差（Mean Square Error）和交叉熵（Cross Entropy）。

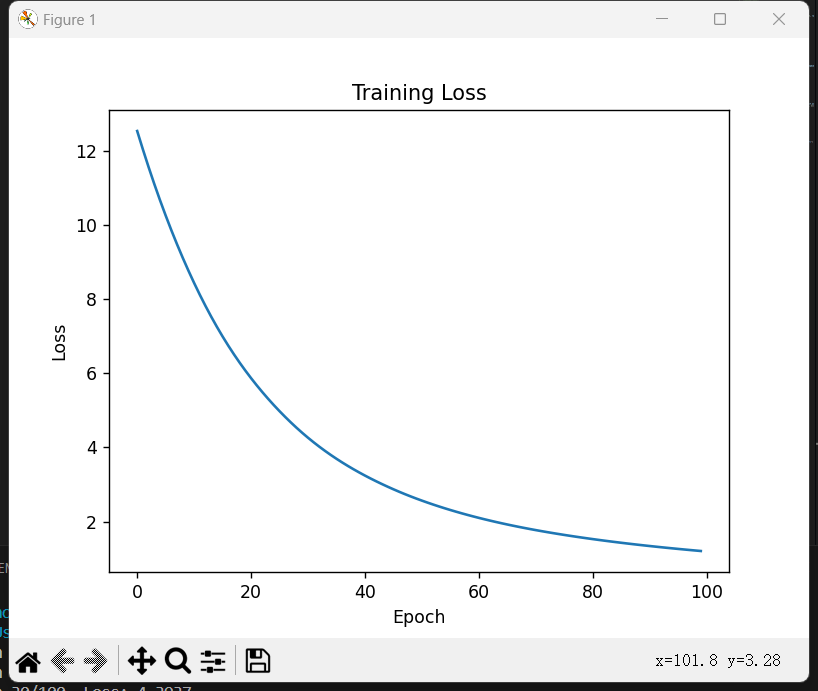
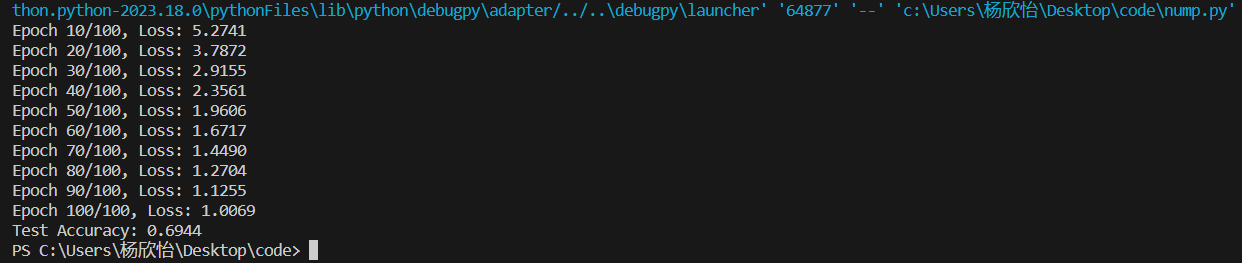
4. 反向传播：根据损失函数的值，计算每个神经元对损失函数的贡献，并将这些梯度信息传播回网络中。在这个过程中，可以使用梯度下降算法来更新权重和偏置。

5. 更新参数：根据反向传播计算得到的梯度信息，使用梯度下降算法来更新神经网络的权重和偏置。

6. 重复步骤2到步骤5：重复进行前向传播和反向传播的过程，直到达到预设的停止条件，例如达到一定的训练轮数或者损失函数的值足够小。

通过反复迭代这些步骤，神经网络可以逐渐调整权重和偏置，从而不断优化模型的性能。需要注意的是，神经网络的学习过程中可能会存在过拟合和欠拟合的问题，需要进行适当的调参和模型选择来解决这些问题。

# 运行结果



1. 源代码

首先，我们需要导入所需的库和数据集：

import numpy as np

from sklearn.datasets import load\_digits

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

接下来，我们加载手写数字数据集并进行预处理：

# 加载手写数字数据集

digits = load\_digits()

# 获取特征和标签

X = digits.data

y = digits.target

# 将数据集分为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

接下来，我们定义一个全连接层类（NeuralNetwork），其中包括初始化方法、前向传播方法和反向传播方法：

# 数据标准化

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# 定义神经网络类

class NeuralNetwork:

    def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

        self.input\_size = input\_size

        self.hidden\_size = hidden\_size

        self.output\_size = output\_size

        # 初始化权重和偏置

        self.W1 = np.random.randn(self.input\_size, self.hidden\_size)

        self.b1 = np.zeros(self.hidden\_size)

        self.W2 = np.random.randn(self.hidden\_size, self.output\_size)

        self.b2 = np.zeros(self.output\_size)

    # 定义前向传播方法

    def forward(self, X):

        self.z1 = np.dot(X, self.W1) + self.b1

        self.a1 = np.tanh(self.z1)

        self.z2 = np.dot(self.a1, self.W2) + self.b2

        self.a2 = np.exp(self.z2) / np.sum(np.exp(self.z2), axis=1, keepdims=True)

        return self.a2

    # 定义交叉熵损失函数

    def compute\_loss(self, X, y):

        m = y.shape[0]

        probs = self.forward(X)

        loss = -np.log(probs[range(m), y])

        return np.sum(loss) / m

    # 定义反向传播方法

    def backward(self, X, y, learning\_rate):

        m = y.shape[0]

        deltas = self.forward(X)

        deltas[range(m), y] -= 1

        deltas /= m

        dW2 = np.dot(self.a1.T, deltas)

        db2 = np.sum(deltas, axis=0)

        dW1 = np.dot(X.T, np.dot(deltas, self.W2.T) \* (1 - np.power(self.a1, 2)))

        db1 = np.sum(np.dot(deltas, self.W2.T) \* (1 - np.power(self.a1, 2)), axis=0)

  # 更新权重和偏置

        self.W2 -= learning\_rate \* dW2

        self.b2 -= learning\_rate \* db2

        self.W1 -= learning\_rate \* dW1

        self.b1 -= learning\_rate \* db1

最后，使用定义的类来训练和测试模型，并绘制损失曲线：

# 定义模型训练方法

    def train(self, X, y, learning\_rate, num\_epochs):

        self.loss\_history = []

        for epoch in range(num\_epochs):

            self.backward(X, y, learning\_rate)

            loss = self.compute\_loss(X, y)

            self.loss\_history.append(loss)

            if (epoch + 1) % 10 == 0:

                print(f"Epoch {epoch + 1}/{num\_epochs}, Loss: {loss:.4f}")

    # 定义模型预测方法

    def predict(self, X):

        return np.argmax(self.forward(X), axis=1)

# 定义模型参数

input\_size = X\_train.shape[1]

hidden\_size = 64

output\_size = len(np.unique(y\_train))

# 创建神经网络模型

model = NeuralNetwork(input\_size, hidden\_size, output\_size)

# 训练模型

learning\_rate = 0.1

num\_epochs = 100

model.train(X\_train, y\_train, learning\_rate, num\_epochs)

# 绘制损失曲线

plt.plot(range(num\_epochs), model.loss\_history)

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.title('Training Loss')

plt.show()

# 在测试集上评估模型

y\_pred = model.predict(X\_test)

accuracy = np.mean(y\_pred == y\_test)

print(f"Test Accuracy: {accuracy:.4f}")

完整代码：

import numpy as np

from sklearn.datasets import load\_digits

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

# 加载手写数字数据集

digits = load\_digits()

# 获取特征和标签

X = digits.data

y = digits.target

# 将数据集分为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 数据标准化

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# 定义神经网络类

class NeuralNetwork:

    def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

        self.input\_size = input\_size

        self.hidden\_size = hidden\_size

        self.output\_size = output\_size

        # 初始化权重和偏置

        self.W1 = np.random.randn(self.input\_size, self.hidden\_size)

        self.b1 = np.zeros(self.hidden\_size)

        self.W2 = np.random.randn(self.hidden\_size, self.output\_size)

        self.b2 = np.zeros(self.output\_size)

    # 定义前向传播方法

    def forward(self, X):

        self.z1 = np.dot(X, self.W1) + self.b1

        self.a1 = np.tanh(self.z1)

        self.z2 = np.dot(self.a1, self.W2) + self.b2

        self.a2 = np.exp(self.z2) / np.sum(np.exp(self.z2), axis=1, keepdims=True)

        return self.a2

    # 定义交叉熵损失函数

    def compute\_loss(self, X, y):

        m = y.shape[0]

        probs = self.forward(X)

        loss = -np.log(probs[range(m), y])

        return np.sum(loss) / m

    # 定义反向传播方法

    def backward(self, X, y, learning\_rate):

        m = y.shape[0]

        deltas = self.forward(X)

        deltas[range(m), y] -= 1

        deltas /= m

        dW2 = np.dot(self.a1.T, deltas)

        db2 = np.sum(deltas, axis=0)

        dW1 = np.dot(X.T, np.dot(deltas, self.W2.T) \* (1 - np.power(self.a1, 2)))

        db1 = np.sum(np.dot(deltas, self.W2.T) \* (1 - np.power(self.a1, 2)), axis=0)

        # 更新权重和偏置

        self.W2 -= learning\_rate \* dW2

        self.b2 -= learning\_rate \* db2

        self.W1 -= learning\_rate \* dW1

        self.b1 -= learning\_rate \* db1

    # 定义模型训练方法

    def train(self, X, y, learning\_rate, num\_epochs):

        self.loss\_history = []

        for epoch in range(num\_epochs):

            self.backward(X, y, learning\_rate)

            loss = self.compute\_loss(X, y)

            self.loss\_history.append(loss)

            if (epoch + 1) % 10 == 0:

                print(f"Epoch {epoch + 1}/{num\_epochs}, Loss: {loss:.4f}")

    # 定义模型预测方法

    def predict(self, X):

        return np.argmax(self.forward(X), axis=1)

# 定义模型参数

input\_size = X\_train.shape[1]

hidden\_size = 64

output\_size = len(np.unique(y\_train))

# 创建神经网络模型

model = NeuralNetwork(input\_size, hidden\_size, output\_size)

# 训练模型

learning\_rate = 0.1

num\_epochs = 100

model.train(X\_train, y\_train, learning\_rate, num\_epochs)

# 绘制损失曲线

plt.plot(range(num\_epochs), model.loss\_history)

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.title('Training Loss')

plt.show()

# 在测试集上评估模型

y\_pred = model.predict(X\_test)

accuracy = np.mean(y\_pred == y\_test)

print(f"Test Accuracy: {accuracy:.4f}")