

**基于全链接网络的手写数字体识别**

**神经网络与深度学习课程实验报告**

班级：105

姓名：耿翊中

学号：2021213382

2023年10月

目录

**[1. 实验要求 3](#_Toc5923)**

**[2. 代码成果及实现思路 4](#_Toc19301)**

[2.1源码文件 4](#_Toc11671)

[2.2数据集导入 4](#_Toc30413)

[2.3Softmax分类器 4](#_Toc30331)

[2.4全连接神经网络分类器 9](#_Toc9587)

[2.4.3输出结果 13](#_Toc27590)

# 实验要求

数据：[MNIST data set](http://yann.lecun.com/exdb/mnist/" \t "https://ucloud.bupt.edu.cn/uclass/course.html" \l "/student/_blank)

• 本题目考察如何设计并实现一个简单的图像分类器。设置本题目的目的如下：

– 理解基本的图像识别流程及数据驱动的方法（训练、预测等阶段）

– 理解训练集/验证集/测试集的数据划分，以及如何使用验证数据调整模型的超参数

– 实现一个**Softmax**分类器

– 实现一个**全连接神经网络**分类器

– 理解不同的分类器之间的区别，以及使用不同的更新方法优化神经网络

**• 附加题：**

– 尝试使用不同的损失函数和正则化方法，观察并分析其对实验结果的影响 (+5 points)

– 尝试使用不同的优化算法，观察并分析其对训练过程和实验结果的影响 (如batch GD, online GD, mini-batch GD, SGD, 或其它的优化算法，如Momentum, Adsgrad, Adam, Admax) (+5 points)

**• 补充：**[MINST](http://yann.lecun.com/exdb/mnist/" \t "https://ucloud.bupt.edu.cn/uclass/course.html" \l "/student/_blank)是一个手写数字数据集，包括了若干手写数字体及其对应的数字，共60000个训练样本，10000个测试样本。每个手写数字被表示为一个28\*28的向量。

# 代码成果及实现思路

**2.1源码文件**

Softmax分类器：soft\_main.py

全连接神经网络分类器：Accuracy.py,Main.py,Method.py,Model.py,Optim.py，GUI.py

使用的数据集：mnist-original.mat

来源网站：<https://www.kaggle.com/avnishnish/mnist-original/download>

**2.2数据集导入**

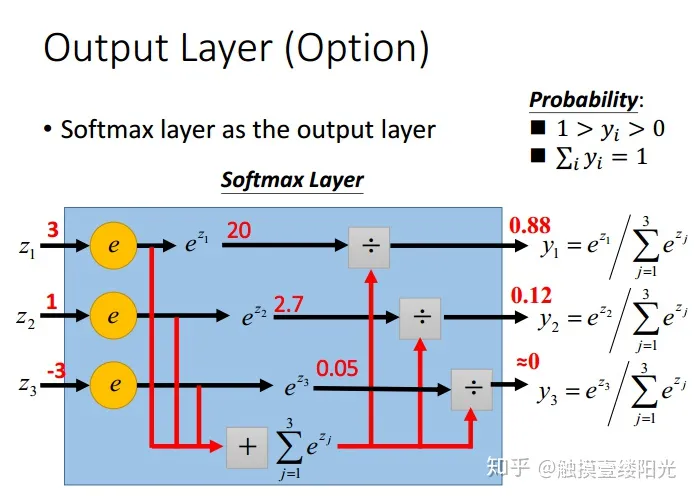
在Softmax和全连接神经网络分类器中都使用了相同的数据集导入方式，数据集来源于网站：<https://www.kaggle.com/avnishnish/mnist-original/download>

，下载到本地后为mnist-original.mat文件，使用from scipy.io import loadmat 函数将其导入，并划分为60000个训练集和10000个测试集。其中x中每个元素为28\*28的矩阵，代表数字图像像素；y为数字图像像素对应的正确数字标签。

1. **from** scipy.io **import** loadmat
2. **def** loadFile():
3. # 加载数据文件
4. data = loadmat('mnist-original.mat')
5. # 提取数据的特征矩阵，并进行转置，将二维数组转换为一维数组的形式
6. x = data['data']
7. x = x.transpose()
8. # 然后将特征除以255.0，以浮点数的形式重新缩放到[0,1]的范围内，以避免在计算过程中溢出
9. x = x / 255.0
10. # 从数据中提取labels，即x数字像素对应的数字类别y
11. y = data['label']
12. y = y.flatten()
13. # 将数据分割为60,000个训练集
14. x\_train = x[:60000, :]
15. y\_train = y[:60000]
16. # 和10,000个测试集
17. x\_test = x[60000:, :]
18. y\_test = y[60000:]
20. **return** x\_train, y\_train, x\_test, y\_test

**2.3Softmax分类器**

**2.3.1Softmax实现分类原理**



Softmax分类器通过将输入特征与权重相乘，然后通过Softmax函数将结果转化为类别概率分布，从而实现多类别分类。模型的训练目标是最小化损失函数，以便在给定输入时能够准确预测类别。

实现思路如下：

准备数据：首先，准备包含训练数据和相应标签的数据集。每个样本都有一个特征向量，通常表示为 x，以及一个标签（类别） y。

特征加权：对于每个输入特征向量 x，Softmax分类器执行一个线性变换，将每个特征与相应的权重相乘，然后将这些加权和进行求和，同时加上一个偏置项。这可以表示为：z = Wx + b

其中，z 是一个包含每个类别的分数向量，W 是权重矩阵，b 是偏置向量。

计算Softmax分布：Softmax函数用于将这些分数转化为类别概率分布。对于每个分数 $z\_i$，Softmax函数计算相应类别的概率 $p(y=i|x)$。Softmax函数的定义如下：IMG_256

这里， $K$ 是类别的总数， $e^{z\_i}$ 表示 $z\_i$ 的指数形式，分母是所有类别的分数的指数和，这确保了所有概率之和为1。

预测：Softmax分类器选择具有最高概率的类别作为预测结果。即，对于输入特征向量 x，选择具有最大概率的类别 $i$ 作为预测的类别。

训练：训练Softmax分类器的目标是调整权重矩阵 W 和偏置项 b 以最小化损失函数。通常，交叉熵损失函数用于衡量模型的预测与实际标签之间的差异。训练的过程通常使用梯度下降或其变种，根据损失的梯度来更新模型参数。

评估性能：模型的性能通常通过准确率等指标来评估。准确率表示正确分类的样本数与总样本数的比例。

**2.3.2代码实现解释**

1. 设置超参数及加载数据集
2. # 设置超参数
3. learning\_rate = 0.1
4. num\_epochs = 100
5. batch\_size = 128
6. # 加载数据集
7. x\_train, y\_train, x\_test, y\_test = loadFile()
8. # 训练 softmax 分类器
9. train\_softmax\_classifier(x\_train, y\_train, x\_test, y\_test, learning\_rate, num\_epochs, batch\_size)
10. 定义函数 train\_softmax\_classifier(x\_train, y\_train, x\_test, y\_test, learning\_rate, num\_epochs, batch\_size)；该函数执行Softmax分类器的训练过程。初始化模型参数（权重矩阵 W 和偏置向量 b）。在每个训练周期（epoch）内，使用小批量（mini-batch）梯度下降来更新模型参数。在每个 epoch 结束后，计算测试集上的损失和准确率，并打印出来。
11. **def** train\_softmax\_classifier(x\_train, y\_train, x\_test, y\_test, learning\_rate, num\_epochs, batch\_size):
12. input\_size = x\_train.shape[1]
13. num\_classes = len(np.unique(y\_train))
14. W, b = initialize\_parameters(input\_size, num\_classes)
16. m = x\_train.shape[0]
17. **for** epoch **in** range(num\_epochs):
18. **for** i **in** range(0, m, batch\_size):
19. x\_batch = x\_train[i:i + batch\_size]
20. y\_batch = y\_train[i:i + batch\_size]
21. y\_batch = y\_batch.astype(int)  # 强制转换为整数类型
22. # 计算预测值
23. scores = np.dot(x\_batch, W) + b
24. y\_pred = softmax(scores)
25. # 计算损失
26. loss = cross\_entropy\_loss(y\_batch, y\_pred)
27. # 计算梯度
28. grad = y\_pred
29. grad[np.arange(x\_batch.shape[0]), y\_batch] -= 1
30. grad /= batch\_size
32. dW = np.dot(x\_batch.T, grad)
33. db = np.sum(grad, axis=0)
35. # 更新权重和偏置
36. W -= learning\_rate \* dW
37. b -= learning\_rate \* db
39. # 在每个 epoch 结束后打印损失和准确率
40. y\_test\_pred = softmax(np.dot(x\_test, W) + b)
41. test\_loss = cross\_entropy\_loss(y\_test, y\_test\_pred)
42. test\_accuracy = accuracy(y\_test, y\_test\_pred)
43. **print**(
44. f"Epoch {epoch + 1}/{num\_epochs}, Loss: {loss:.4f}, Test Loss: {test\_loss:.4f}, Test Accuracy: {test\_accuracy:.2%}")
45. 其它辅助函数

定义函数 softmax(x)：该函数实现了Softmax函数，用于将输入向量转换为概率分布。

1. # 定义 softmax 函数
2. **def** softmax(x):
3. e\_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
4. **return** e\_x / e\_x.sum(axis=-1, keepdims=True)

定义函数 initialize\_parameters(input\_size, output\_size)：该函数用于初始化权重矩阵 W 和偏置向量 b，这些参数将在Softmax分类器的训练中学习。

W 被初始化为小的随机值，b 被初始化为全零。

1. # 初始化权重矩阵和偏置向量
2. **def** initialize\_parameters(input\_size, output\_size):
3. W = np.random.randn(input\_size, output\_size) \* 0.001
4. b = np.zeros(output\_size)
5. **return** W, b

定义函数 cross\_entropy\_loss(y\_true, y\_pred)：该函数计算交叉熵损失，用于度量模型的性能。计算交叉熵损失时，将真实标签 y\_true 与模型的预测概率分布 y\_pred 进行比较。

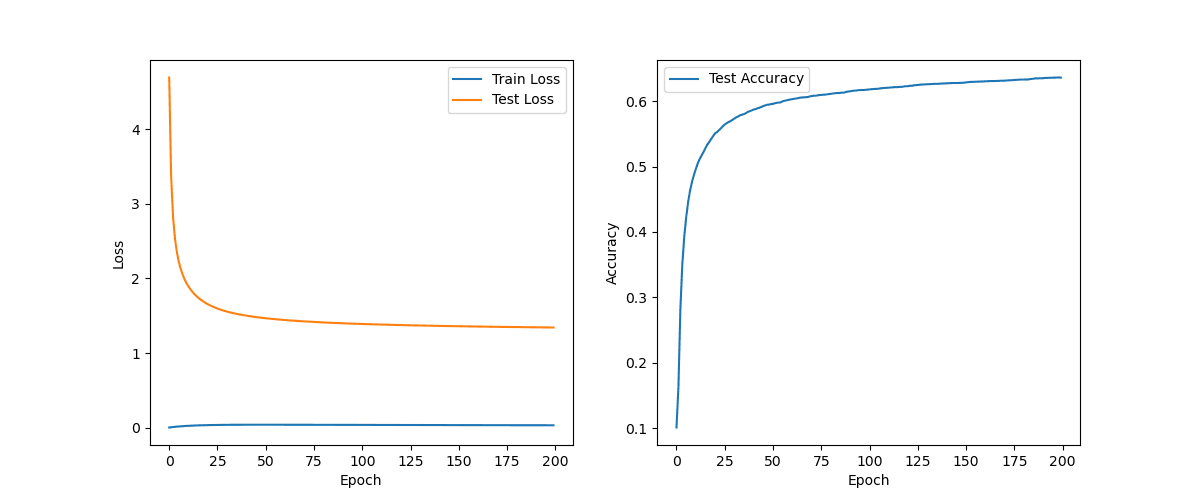
1. # 定义交叉熵损失函数
2. **def** cross\_entropy\_loss(y\_true, y\_pred):
3. m = y\_true.shape[0]
4. y\_true = y\_true.astype(int)  # 强制转换为整数类型
5. **return** -np.sum(np.log(y\_pred[np.arange(m), y\_true])) / m

**2.3.3输出结果**

以学习率为0.1，迭代次数为200，批次为128得到的最终结果是：

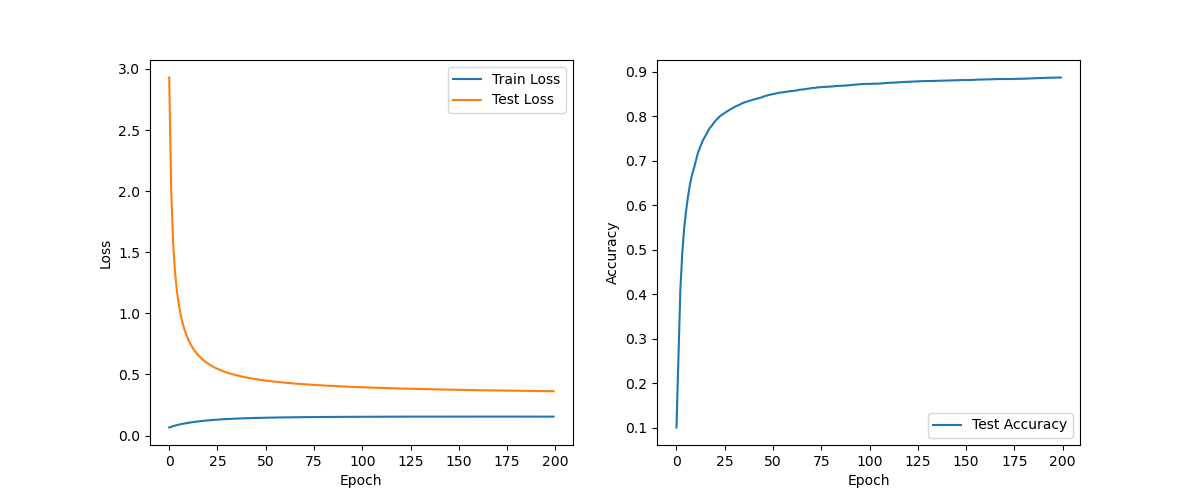
Epoch 200/200, Loss: 0.0343, Test Loss: 1.3429, Test Accuracy: 63.59%

绘出图像为：



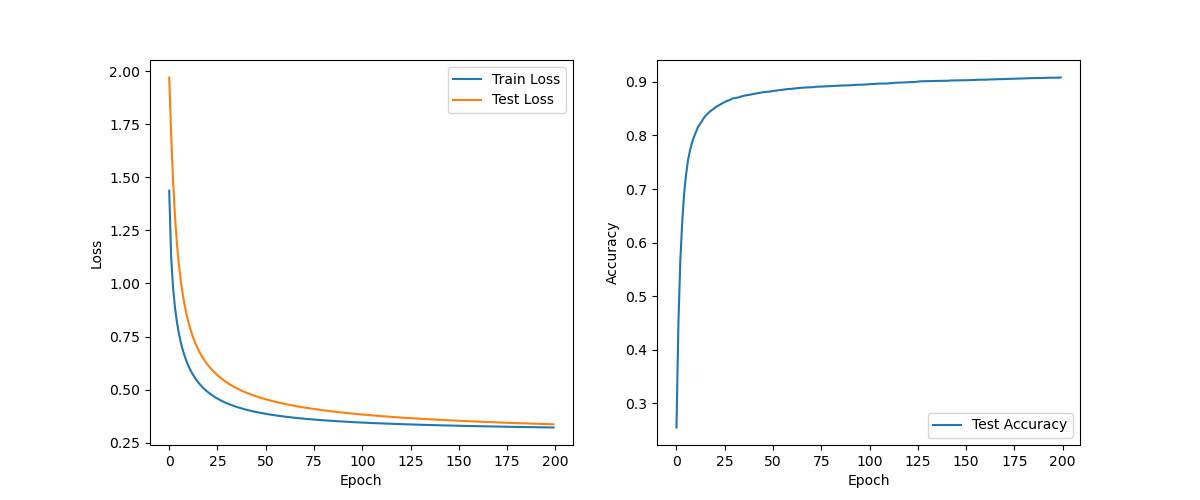
以学习率为0.01，迭代次数为200，批次为128得到的最终结果是：

Epoch 200/200, Loss: 0.1552, Test Loss: 0.3632, Test Accuracy: 88.69%绘出图像为：



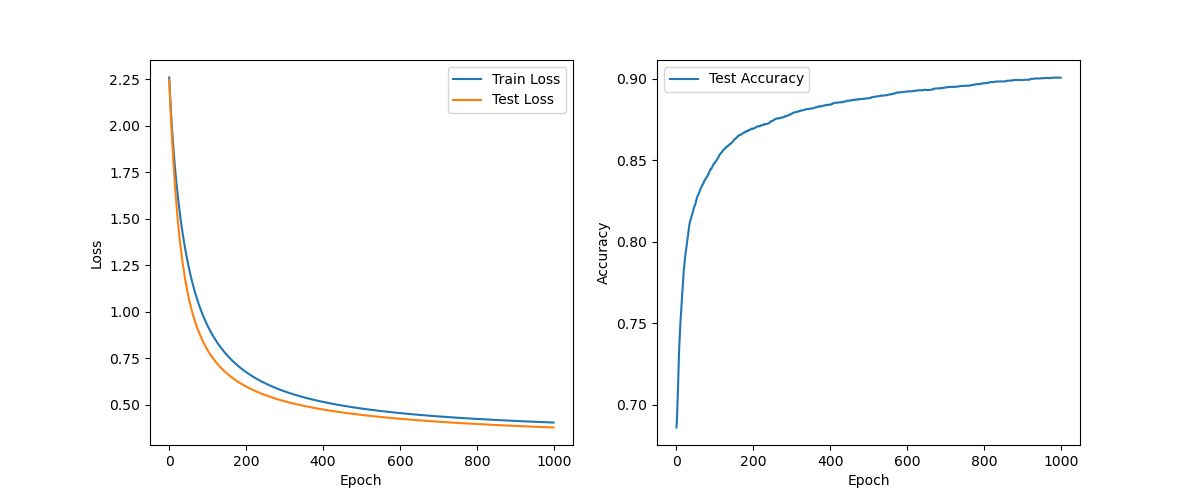
以学习率为0.01，迭代次数为200，批次为128得到的最终结果是：

Epoch 200/200, Loss: 0.3221, Test Loss: 0.3371, Test Accuracy: 90.82%绘出图像为：



以学习率为0.001，迭代次数为1000，批次为128得到的最终结果是：

Epoch 1000/1000, Loss: 0.4052, Test Loss: 0.3785, Test Accuracy: 90.07%绘出图像为：



通过调整学习率可以发现，学习率越低，训练的速度越低，准确率上升越慢，但最终达到的准确率会更高。

**2.4全连接神经网络分类器**

**2.4.1整体框架**

数据准备：使用MNIST数据集的一部分作为训练和测试数据。构建一个具有输入层、隐藏层和输出层的三层全连接神经网络。

神经网络的参数初始化：初始化神经网络的权重参数Theta1和Theta2。设置学习率、正则化参数和迭代次数。

神经网络训练：可以使用多种优化算法训练神经网络，根据随机小批次的数据更新参数。

记录每次迭代的损失和准确度。

计算准确度：使用训练好的神经网络参数计算训练集和测试集的准确度。并将训练过程中的准确度结果绘制为直观的图像。

GUI界面：使用Tkinter库创建一个GUI界面，允许用户手写数字。提供清除界面和进行识别的按钮。绘制区域用于手写数字输入。便于用户直观感受自己训练出来的网络在实用中准确率如何。

**2.4.2具体结构和流程**

神经网络结构：

输入层（Layer 1）：784个节点，对应28x28像素的手写数字图像。这些节点用来接受输入特征。

隐藏层（Layer 2）：16个节点，采用Sigmoid函数作为激活函数。这是一个中间层，用于学习特征表示。

输出层（Layer 3）：10个节点，对应0到9的数字类别。这是分类问题的输出层，由于之前已经实现了softmax分类器，故这里采用Sigmoid函数作为激活函数。

1. # 输入层，隐藏层，输出层节点个数
2. input\_layer\_size = 784  # 图片大小为 (28 X 28) px 所以设置784个特征
3. hidden\_layer\_size = 16
4. num\_labels = 10  # 拥有十个标准为 [0, 9] 十个数字
5. # 初始化层之间的权重 Thetas
6. initial\_Theta1 = initialise(hidden\_layer\_size, input\_layer\_size)  # 输入层和隐藏层之间的权重
7. initial\_Theta2 = initialise(num\_labels, hidden\_layer\_size)  # 隐藏层和输出层之间的权重
8. # 设置神经网络的参数
9. initial\_nn\_params = np.concatenate((initial\_Theta1.flatten(), initial\_Theta2.flatten()))

向前传播（Forward Propagation）：

输入层（Layer 1）接受手写数字图像作为输入。

向前传播计算从输入层到隐藏层（Layer 2）的权重加权和，并应用Sigmoid激活函数。

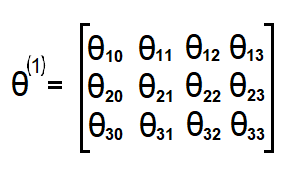
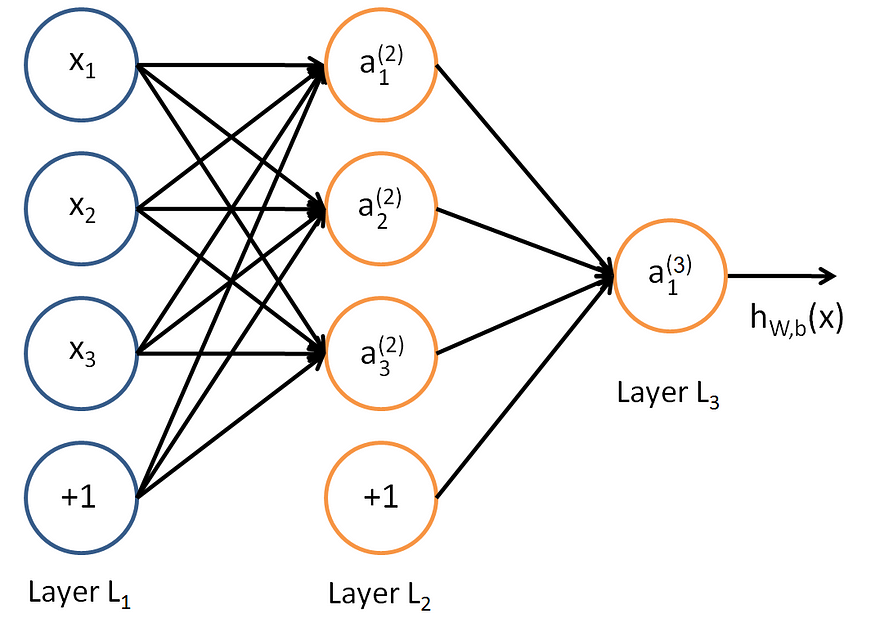
隐藏层的输出再次计算从隐藏层到输出层（Layer 3）的权重加权和，并再次应用Sigmoid激活函数。

输出层的激活值表示每个数字类别的预测概率。

1. # 向前传播
2. m = X.shape[0]
3. one\_matrix = np.ones((m, 1))
4. X = np.append(one\_matrix, X, axis=1)  # 向输入层添加偏置单元，使之成为偏差节点
5. a1 = X
6. z2 = np.dot(X, Theta1.transpose())
7. a2 = 1 / (1 + np.exp(-z2))  # 采用Sigmoid函数对隐藏层进行激活
8. one\_matrix = np.ones((m, 1))
9. a2 = np.append(one\_matrix, a2, axis=1)  # 向隐藏层添加偏置单元，使之成为偏差节点
10. z3 = np.dot(a2, Theta2.transpose())
11. a3 = 1 / (1 + np.exp(-z3))  # 采用Sigmoid函数对输出层进行激活
12. # 将标签改为一个长度为10的布尔向量，在向量的10个布尔数值里，哪个数等于1，它就代表着几
13. y\_vect = np.zeros((m, 10))
14. **for** i **in** range(m):
15. y\_vect[i, int(y[i])] = 1

偏置节点处理：

不同于将偏置b分离出来的方式，此神经网络采取了另外一种将偏置放入层次中的方法，在原本的输入层和隐藏层传播时多加入一个偏置节点。如该模型将有 4 个输入节点（3 + 1 个“偏差”）。 一个隐藏层，具有 4 个节点（3 + 1“偏置”）和一个输出节点。如果网络在 j 层中有单位，在 j+1 层中有 b 单位，则 θj 的维度为 b×（a+1）。



反向传播（Backpropagation）：

计算损失函数，这里采用的是损失函数 loss。

通过反向传播算法，计算输出层（Layer 3）到隐藏层（Layer 2）和隐藏层（Layer 2）到输入层（Layer 1）的梯度，即误差对权重的偏导数。

使用梯度下降法更新权重参数（Theta1和Theta2）以减小损失函数。

1. # 计算损失值
2. J = loss(Theta1, Theta2, y\_vect, a3, lamb, m)
3. # 向后传播
4. Delta3 = a3 - y\_vect
5. Delta2 = np.dot(Delta3, Theta2) \* a2 \* (1 - a2)
6. Delta2 = Delta2[:, 1:]
7. # 计算梯度
8. Theta1[:, 0] = 0
9. Theta1\_grad = (1 / m) \* np.dot(Delta2.transpose(), a1) + (lamb / m) \* Theta1
10. Theta2[:, 0] = 0
11. Theta2\_grad = (1 / m) \* np.dot(Delta3.transpose(), a2) + (lamb / m) \* Theta2
12. grad = np.concatenate((Theta1\_grad.flatten(), Theta2\_grad.flatten()))

损失函数 (Loss Function):

损失函数是用来度量模型预测与实际标签之间的差异的函数。此网络使用了两种不同的损失函数计算方法：交叉熵损失和均方误差损失。

交叉熵损失（Cross-Entropy Loss）：通常用于分类问题，特别是多类别分类。它度量了模型的预测概率与实际标签之间的差异。交叉熵损失越低，模型的预测越接近实际标签。该损失函数在神经网络中常用。

均方误差损失（Mean Squared Error Loss）：通常用于回归问题，度量模型的预测值与实际值之间的平方差异。这个损失函数也在神经网络中使用，特别是在回归任务中。

1. # 交叉熵损失（Cross-Entropy Loss）
2. J = (1 / m) \* (np.sum(np.sum(-y\_vect \* np.log(a3) - (1 - y\_vect) \* np.log(1 - a3)))) + Reg
3. 均方误差损失（Mean Squared Error Loss）
4. J = (1 / (2 \* m)) \* np.sum(np.square(a3 - y\_vect)) + Reg

正则化 (Regularization):

正则化是用来防止模型过拟合的技术。此神经网络提供了L2和L1两种正则化方式给损失函数，同时在梯度计算中也添加了正则化项。

L2正则化：L2正则化通过向损失函数添加权重的平方和来防止权重值过大，从而降低过拟合的风险。在损失函数中，L2\_Reg 表示L2正则化项，包括输入层到隐藏层的权重 Theta1 和隐藏层到输出层的权重 Theta2。

L1正则化：L1正则化通过向损失函数添加权重的绝对值和来降低一些权重为零的参数，这对于稀疏模型有用。在代码中，提供了L1正则化的计算方法，但实际使用的是L2正则化。

正则化参数 lamb 控制了正则化的强度。通过调整 lamb 的值，可以平衡模型的拟合能力和泛化能力。更大的 lamb 值会增强正则化效果，减小权重的大小，从而降低过拟合的风险。

1. # 正则化方法选择
2. # L2正则化
3. L2\_Reg = (lamb / (2 \* m)) \* (
4. np.sum(np.square(Theta1[:, 1:])) + np.sum(np.square(Theta2[:, 1:])))
5. # L1正则化
6. L1\_Reg = (lamb / (2 \* m)) \* (
7. np.sum(np.abs(Theta1[:, 1:])) + np.sum(np.abs(Theta2[:, 1:]))
8. )
9. Reg = L2\_Reg

优化方法：

Batch Gradient Descent (BGD):

BGD 是最基本的梯度下降算法，它在每个迭代步骤中使用整个训练集来计算梯度并更新模型参数。iter\_num 控制了迭代的次数，alpha\_rate 是学习率，控制了参数更新的步长。

训练过程中，记录了损失和训练集/测试集的准确度，并可视化损失和准确度的历史数据。

**def** BGD(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lambda\_reg, iter\_num, alpha\_rate, X\_test,  y\_test):

Stochastic Gradient Descent (SGD):

SGD 与 BGD 相似，但每次迭代仅随机选择一个样本来计算梯度并更新参数。这对于大型数据集可以加速训练。

**def** SGD(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lambda\_reg, iter\_num, alpha\_rate, X\_test, y\_test):

Online Gradient Descent (OGD):

OGD 是 SGD 的变种，但每次迭代使用更大的 batch\_size，适用于在线学习场景，其中数据逐步到达。alpha\_rate 是学习率，iter\_num 控制了迭代的次数。

**def** OGD(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lambda\_reg, iter\_num, alpha\_rate, X\_test,  y\_test):

Mini-Batch Gradient Descent:

Mini-Batch Gradient Descent 是介于 BGD 和 SGD 之间的一种方法，它在每次迭代中随机使用一小批样本来计算梯度并更新参数。

**def** MiniBGD(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lambda\_reg, iter\_num, alpha\_rate, X\_test,  y\_test):

Momentum:

Momentum 添加了动量项，有助于加速收敛，特别是在存在局部极小值的情况下。

beta 控制了动量项的权重，通常设置为 0.9。

**def** Momentum(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lambda\_reg, iter\_num, alpha\_rate, X\_test,  y\_test):

Adagrad:

Adagrad 自适应地调整学习率，对于不同参数有不同的学习率，有助于快速收敛。

epsilon 是用于避免除零错误的小常数。

**def** Adagrad(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lambda\_reg, iter\_num, alpha\_rate, X\_test,

Adam:

Adam 结合了动量和 Adagrad，具有良好的性能和鲁棒性。

beta1 和 beta2 控制了动量项和学习率自适应项的权重。

**def** Adam(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lambda\_reg, iter\_num, alpha\_rate, X\_test, y\_test):

Adamax:

Adamax 是 Adam 的变种，用于解决 Adam 在某些情况下可能出现的问题。

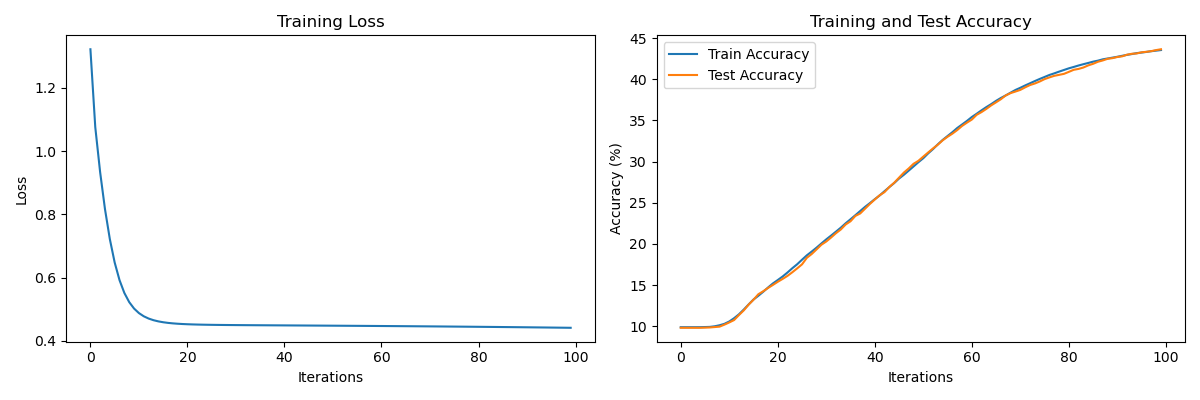
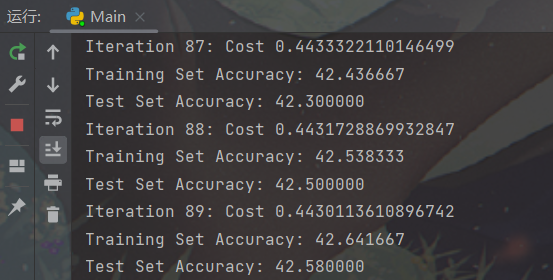
beta1 和 beta2 控制了动量项和学习率自适应项的权重。

**def** Adamax(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lambda\_reg, iter\_num, alpha\_rate, X\_test,  y\_test):

**2.4.3输出结果**

在每一次迭代中打印出当前迭代后的损失函数和关于训练集和测试集的准确率

在所有迭代完成后绘制训练过程中的损失函数变化曲线和测试准确度



之后可以调用GUI.py进行手写数字识别的交互

