

**基基于卷积神经网络图像分类**

**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11月

目录

[1. 实验要求 3](#_Toc8326)

[2. 代码实现过程 4](#_Toc10658)

[2.1代码结构 4](#_Toc21202)

[2.2数据集处理 4](#_Toc27304)

[2.3卷积，池化，全连接层实现 5](#_Toc1572)

[2.3.1卷积层 5](#_Toc29475)

[2.3.3全连接 9](#_Toc29823)

[2.4全连接神经网络分类器 11](#_Toc17229)

[2.4.1整体框架 11](#_Toc6768)

[2.4.2具体结构和流程 11](#_Toc21972)

[4.     Delta3 = a3 - y\_vect   13](#_Toc32500)

[6.     Delta2 = Delta2[:, 1:]   13](#_Toc22683)

[8.     Theta1[:, 0] = 0   13](#_Toc22459)

[10.     Theta2[:, 0] = 0   13](#_Toc21644)

[9.     Reg = L2\_Reg   14](#_Toc21557)

[2.4.3输出结果 15](#_Toc1240)

[3. 题目问题回答与深入研究 16](#_Toc28234)

[3.1基本的图像识别流程及数据驱动的方法 16](#_Toc24918)

[3.2使用验证数据调整模型的超参数 19](#_Toc21944)

[3.3使用不同的更新方法优化神经网络 23](#_Toc15600)

[3.3.1传统的四种方法 23](#_Toc29538)

[3.3.2新的四种改进方法 24](#_Toc8237)

[3.4使用不同的损失函数和正则化方法 25](#_Toc11288)

[3.4.1正则化方法 25](#_Toc2399)

[3.4.2损失函数 26](#_Toc1698)

[4.经验总结 28](#_Toc18180)

# 实验要求

Task: 基于卷积神经网络图像分类 (15 Points)

• Due Date:  Nov.9, 2023 (Tuesday)

• 数据集：[MNIST data set](http://yann.lecun.com/exdb/mnist/" \\t "_blank" \t "https://ucloud.bupt.edu.cn/uclass/course.html" \l "/student/_blank)，或其它数据集，如[CIFAR-10](https://www.cs.toronto.edu/~kriz/cifar.html" \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基于卷积神经网络的图像分类器。设置本题目的目的如下：

– 理解卷积神经网络的基本结构、代码实现及训练过程

– 应用dropout和多种normalization方法，理解它们对模型泛化能力的影响

– 理解如何通过交叉验证，为神经网络找到最好的hyperparameters

• 附加题

– 在训练网络的过程中，可根据需要自由尝试其它提升性能的方法，例如通过增加模型层数、使用不同的正则化方法、使用模型集成等 (+5 points)

# 代码实现过程

## 2.1代码结构

Main.py：主函数

model.py:卷积层，全连接层，池化层的实现

utils.py：加载数据等工具类和函数

Method.py：提升模型性能的方法类，Batch Normalization和dropout

## 2.2数据集处理

使用Minist手写数字识别数据集，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，将数据集分为训练集和测试集，根据构造函数中指定的train\_size和test\_size参数。

类中提供了两个方法来加载训练集和测试集：

load\_train()方法：返回训练集的图像数据 X\_train 和相应的标签数据 y\_train。在这个方法内，图像数据被重新形状为28x28的图像，并且像素值被除以255，以将其缩放到0到1的范围内。标签数据也经过了one-hot编码，其中onehot()函数用于将标签转化为one-hot向量。

load\_test()方法：返回测试集的图像数据 X\_test 和相应的标签数据 y\_test。同样，图像数据也被重新形状和缩放，标签数据也被one-hot编码。

1. **class** Load\_Data:
2. **def** \_\_init\_\_(self, train\_size=60000, test\_size=10000):
3. '''''
4. 导入数据集，划分为 60,000 个训练样本，10,000 个测试样本
5. '''
6. # 加载数据文件
7. data = loadmat('mnist-original.mat')
8. # 提取数据的特征矩阵，并进行转置
9. X = data['data']
10. self.X = X.transpose()
11. # 然后将特征除以 255，重新缩放到 [0,1] 的范围内，以避免在计算过程中溢出
12. # 从数据中提取 labels
13. y = data['label']
14. self.y = y.flatten()
15. # 将数据分割为 60,000 个训练集
16. self.train\_size = train\_size
17. # 和 10,000 个测试集
18. self.test\_size = test\_size
19. self.indices = np.arange(70000)
20. np.random.shuffle(self.indices)
21. self.X = self.X[self.indices]
22. self.y = self.y[self.indices]
24. **def** load\_train(self):
25. X\_train = self.X[:self.train\_size, :].reshape(self.train\_size, 28, 28)
26. y\_train = self.y[:self.train\_size]
27. X\_train = X\_train.reshape(self.train\_size, 28, 28, 1) / 255.  # 输入向量处理
28. y\_train = onehot(y\_train, self.train\_size)  # 标签one-hot处理 (60000, 10)
29. **return** X\_train, y\_train
31. **def** load\_test(self):
32. X\_test = self.X[self.train\_size:self.train\_size + self.test\_size, :].reshape(self.test\_size, 28, 28)
33. y\_test = self.y[self.train\_size:self.train\_size + self.test\_size]
34. X\_test = X\_test.reshape(self.test\_size, 28, 28, 1) / 255.
35. **return** X\_test, y\_test
37. **def** onehot(targets, num):
38. result = np.zeros((num, 10))
39. **for** i **in** range(num):
40. result[i][int(targets[i])] = 1
41. **return** result

## 2.3卷积，池化，全连接层实现

### 2.3.1卷积层

卷积层构造Conv 类通过几个方法来实现卷积层的功能，包括初始化层、执行前向传播和执行反向传播（用于学习）。

\_\_init\_\_

当一个 Conv 类的对象被创建时，\_\_init\_\_ 方法首先被调用。这个方法负责初始化卷积层的权重和偏置。

kernel\_shape 是一个四元组，包含了卷积核的宽度、高度、输入通道数和输出通道数。

Xavier初始化方法被用来初始化权重，这是一种常见的权重初始化方法，目的是为了保持输入和输出的方差一致，防止梯度消失或爆炸。

self.k 是卷积核的权重，self.b 是偏置。

self.k\_gradient 和 self.b\_gradient 用于存储权重和偏置的梯度，这些梯度在反向传播时更新权重和偏置。

1. **def** \_\_init\_\_(self, kernel\_shape, path=1):
2. width, height, in\_channel, out\_channel = kernel\_shape
3. self.path = path
4. scale = np.sqrt(3 \* in\_channel \* width \* height / out\_channel)  # 使用Xavier初始化缩放因子
5. self.k = np.random.standard\_normal(kernel\_shape) / scale  # 初始化k，为一个四维向量
6. self.b = np.random.standard\_normal(out\_channel) / scale  # 初始化b，为一个一维向量
7. self.k\_gradient = np.zeros(kernel\_shape)  # 相应存储k的梯度
8. self.b\_gradient = np.zeros(out\_channel)  # 相应存储b的梯度
9. pad = 0
10. self.pad = pad

forward

forward 方法实现了前向传播，这是神经网络计算其输出的过程。其中关键步骤是利用矩阵乘法实现了卷积的操作。

输入数据 x 可能会根据 pad 参数被填充，以确保输出尺寸的正确性。

方法中使用了 img2col 函数，这个函数会将输入图像展开为列，以便能够使用矩阵乘法来实现卷积操作。首先，函数接收一个三维的输入数据 x，这个数据的形状是 [width, height, channel]，代表图像的宽度、高度和通道数。函数计算特征图的尺寸 feature\_w，这是经过卷积后的输出宽度和高度，根据卷积核的大小 ksize 和步长 stride 来确定。然后创建一个用于存放“展开”后图像块的矩阵 image\_col。这个矩阵的每一行都将对应输入数据中的一个局部区域，该区域的大小和卷积核相匹配。矩阵的列数是卷积核大小乘以输入通道数，即 ksize \* ksize \* cx。接下来，通过两个嵌套循环遍历特征图的宽度和高度，对于特征图中的每一个位置，都会从输入数据 x 中提取出对应的局部区域。这些局部区域会被“展开”成一维数组，并被赋值到 image\_col 矩阵的相应行。展开是通过 reshape(-1) 实现的，这个操作会保持数据的顺序，但是把它变成一维数组。num 变量用于跟踪 image\_col 矩阵的行索引，每处理一个局部区域后，num 会增加，以便下一个局部区域可以放在下一行。

最终，输出的维度是 (batch\_size, bkimg\_size, bkimg\_size, kernel\_num)。

1. **def** img2col(x, ksize, stride):
2. wx, hx, cx = x.shape  # [width,height,channel]
3. feature\_w = (wx - ksize) // stride + 1  # 返回的特征图尺寸
4. image\_col = np.zeros((feature\_w \* feature\_w, ksize \* ksize \* cx))
5. num = 0
6. **for** i **in** range(feature\_w):
7. **for** j **in** range(feature\_w):
8. image\_col[num] = x[i \* stride:i \* stride + ksize, j \* stride:j \* stride + ksize, :].reshape(-1)
9. num += 1
10. **return** image\_col
11. **def** forward(self, x):
12. self.x = x
13. **if** self.pad != 0:
14. self.x = np.pad(self.x, ((0, 0), (self.pad, self.pad), (self.pad, self.pad), (0, 0)), 'constant')
15. # 批量，宽度，高度，输入通道数
16. batch\_size, in\_width, in\_height, in\_channel = self.x.shape
17. # 卷积核宽度，高度，卷积核输入通道数,卷积核个数
18. kernel\_width, kernel\_height, kin\_channel, kernel\_num = self.k.shape
19. # 返回图像的参数，批量，高，宽，通道
20. bkimg\_size = (in\_width - kernel\_width) // self.path + 1
21. bk\_img = np.zeros((batch\_size, bkimg\_size, bkimg\_size, kernel\_num))
23. # 进行卷积运算
24. self.image\_col = []
25. kernel = self.k.reshape(-1, kernel\_num)
26. **for** i **in** range(batch\_size):
27. image\_col = img2col(self.x[i], kernel\_width, self.path)
28. bk\_img[i] = (np.dot(image\_col, kernel) + self.b).reshape(bkimg\_size, bkimg\_size, kernel\_num)
29. self.image\_col.append(image\_col)
30. **return** bk\_img

backward 方法

backward 方法执行反向传播，这是训练神经网络时用于计算梯度并更新权重的过程。

方法计算从下一层回传的梯度 delta，并根据这些梯度来更新卷积核的权重 self.k 和偏置 self.b。

在权重更新之前，权重梯度 self.k\_gradient 和偏置梯度 self.b\_gradient 会被累积和平均。

最后，计算了传播到前一层的梯度 delta\_backward，这是通过对卷积核进行180度旋转并执行卷积操作实现的。

1. **def** backward(self, delta, learning\_rate):
2. # 批量，宽度，高度，输入通道数
3. batch\_size, in\_width, in\_height, in\_channel = self.x.shape
4. # 卷积核宽度，高度，卷积核输入通道数,卷积核个数
5. kernel\_width, kernel\_height, kin\_channel, kernel\_num = self.k.shape
6. # 从上一层传回来的梯度,批量,宽度,高度,卷积核个数
7. delta\_batch, delta\_width, delta\_height, delta\_channel = delta.shape
9. delta\_col = delta.reshape(delta\_batch, -1, delta\_channel)
10. **for** i **in** range(batch\_size):
11. self.k\_gradient += np.dot(self.image\_col[i].T, delta\_col[i]).reshape(self.k.shape)
12. self.k\_gradient /= batch\_size
13. self.b\_gradient += np.sum(delta\_col, axis=(0, 1))
14. self.b\_gradient /= batch\_size
16. # 计算delta\_backward
17. delta\_backward = np.zeros(self.x.shape)
18. k\_180 = np.rot90(self.k, 2, (0, 1))  # numpy矩阵旋转180度
19. k\_180 = k\_180.swapaxes(2, 3)
20. k\_180\_col = k\_180.reshape(-1, kin\_channel)
22. **if** delta\_height - kernel\_height + 1 != in\_height:
23. pad = (in\_height - delta\_height + kernel\_height - 1) // 2
24. pad\_delta = np.pad(delta, ((0, 0), (pad, pad), (pad, pad), (0, 0)), 'constant')
25. **else**:
26. pad\_delta = delta
28. **for** i **in** range(batch\_size):
29. pad\_delta\_col = img2col(pad\_delta[i], kernel\_width, self.path)
30. delta\_backward[i] = np.dot(pad\_delta\_col, k\_180\_col).reshape(in\_width, in\_height, in\_channel)
32. # 反向传播
33. self.k -= self.k\_gradient \* learning\_rate
34. self.b -= self.b\_gradient \* learning\_rate
36. **return** delta\_backward

### **2.3.2池化层**

Pool 类实现了池化层的操作，用来减少特征图的空间尺寸，从而减少参数的数量和计算的复杂度，并且帮助提取特征。在这个类中，实现了最大池化（max pooling）操作。

forward

forward 方法实现了前向传播过程中的池化操作。

方法首先接收输入数据 x，其形状为 (batch\_size, in\_width, in\_height, in\_channel)，代表批次大小、输入宽度、输入高度和通道数。

创建输出特征图 bk\_img，其尺寸是输入的一半，因为这里实现的是步长为2的2x2最大池化。

self.max\_address 是一个与输入 x 形状相同的数组，用于在池化过程中记录每个通道的最大值的位置，这在反向传播中非常有用。

在四个嵌套循环中，对于输入数据的每个批次、每个通道，以及在宽度和高度上的每个2x2的区域：

找到当前2x2区域中的最大值，并将其赋值给输出特征图 bk\_img 相应的位置。

使用 np.argmax 找到这个最大值在2x2区域中的索引，并将该位置在 self.max\_address 中标记为1。

最后，返回池化后的输出特征图 bk\_img。

1. **def** forward(self, x):
2. batch\_size, in\_width, in\_height, in\_channel = x.shape
3. bkimg\_size = in\_width // 2
4. bk\_img = np.zeros((batch\_size, bkimg\_size, bkimg\_size, in\_channel))
5. # 记录池化位置,在反向传播中使用
6. self.max\_address = np.zeros((batch\_size, in\_width, in\_height, in\_channel))
7. **for** b **in** range(batch\_size):
8. **for** c **in** range(in\_channel):
9. **for** w **in** range(bkimg\_size):
10. **for** h **in** range(bkimg\_size):
11. bk\_img[b, w, h, c] = np.max(x[b, w \* 2:w \* 2 + 2, h \* 2: h \* 2 + 2, c])
12. index = np.argmax(x[b, w \* 2:w \* 2 + 2, h \* 2: h \* 2 + 2, c])
13. self.max\_address[b, w \* 2 + index // 2, h \* 2 + index // 2, c] = 1
14. **return** bk\_img

backward

backward 方法实现了池化层的反向传播。

方法接收来自后一层的梯度 delta，其形状与池化后的输出特征图相同。

由于最大池化操作不是一个有参数的操作，我们不需要计算参数的梯度，只需要将错误梯度从池化层传播回卷积层。

使用 np.repeat 方法将 delta 在每个轴上扩展两倍，这样就恢复到了池化前的尺寸。

最后，将扩展后的梯度与 self.max\_address 相乘，以确保梯度只回传到每个2x2区域中最大值的位置，其他位置的梯度为0。

1. **def** backward(self, delta):
2. **return** np.repeat(np.repeat(delta, 2, axis=1), 2, axis=2) \* self.max\_address

### 2.3.3全连接

位于神经网络的末端，对特征进行最后的决策分类处理。

当 Linear 类的一个实例被创建时，\_\_init\_\_ 方法首先被调用。这个方法负责初始化全连接层的权重 W 和偏置 b。

inChannel 表示输入特征的数量，outChannel 表示输出特征的数量。

权重和偏置使用标准正态分布随机初始化，并且使用 Xavier 初始化的变种（使用输入通道数量的平方根除以2作为分母）来进行缩放，以保持激活值在训练开始时的合理范围。

self.W\_gradient 和 self.b\_gradient 用于存储权重和偏置的梯度。

forward 方法实现了全连接层的前向传播。

backward 方法实现了全连接层的反向传播。

这里的处理和普通神经网络区别不大，不再多加赘述。

## 2.4神经网络的训练和测试

### 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。

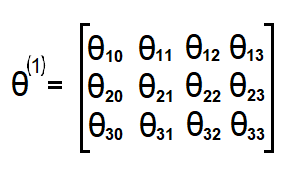
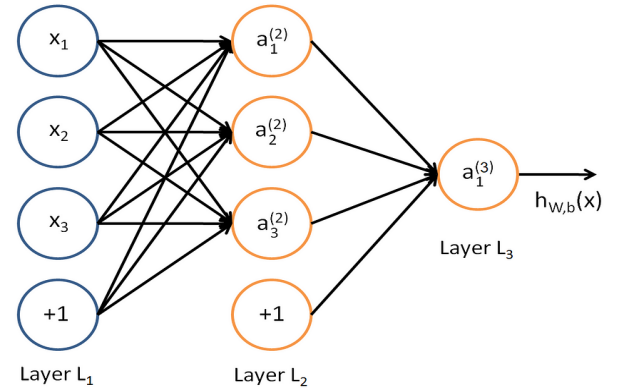
隐藏层的输出再次计算从隐藏层到输出层（Layer 3）的权重加权和，并再次应用Sigmoid激活函数。

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

更新方法：

此次实验使用了八种神经网络的更新方法对神经网络进行优化。分别是以下几种，具体使用后的结果见后面部分。

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。

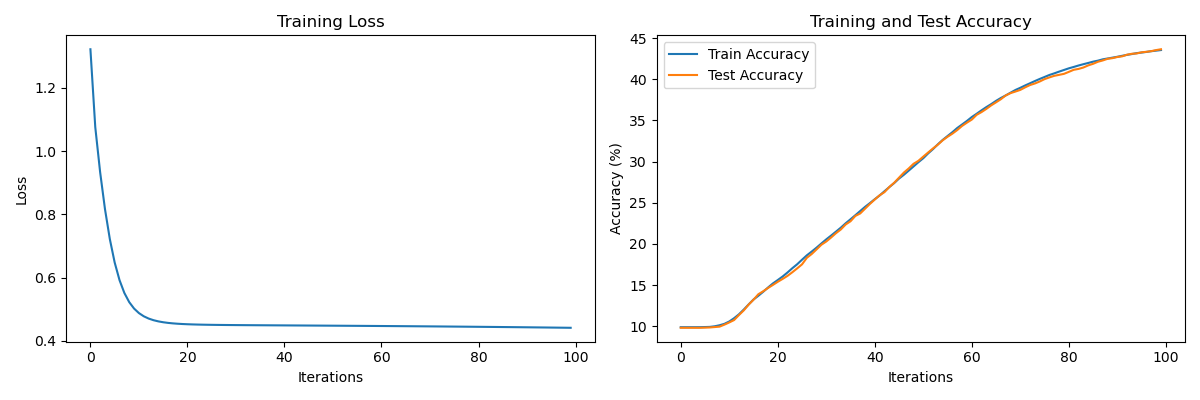
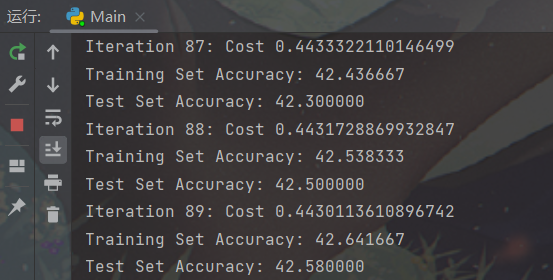
beta1 和 beta2 控制了动量项和学习率自适应项的权重。

**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



# 题目问题回答与深入研究

## 3.1基本的图像识别流程及数据驱动的方法

数据收集和准备阶段：

数据收集：首先，需要收集包含手写数字的图像数据集。这些数据通常由手写数字图像和相应的标签组成，其中标签指示每个图像中的数字是什么。

1. # 加载数据文件
2. data = loadmat('mnist-original.mat')

数据预处理：对数据进行预处理是很重要的。这包括图像大小标准化、去噪、灰度化和数据增强等操作，以确保输入数据对模型的训练有利。

1. # 提取数据的特征矩阵，并进行转置
2. X = data['data']
3. X = X.transpose()
4. # 然后将特征除以255，重新缩放到[0,1]的范围内，以避免在计算过程中溢出
5. X = X / 255
6. # 从数据中提取labels
7. y = data['label']
8. y = y.flatten()

数据分割：通常，将数据集分为训练集、验证集和测试集。训练集用于训练模型，验证集用于调整超参数和监控模型性能，测试集用于最终模型评估。

1. # 将数据分割为60,000个训练集
2. train\_size = 60000
3. X\_train = X[:train\_size, :]
4. y\_train = y[:train\_size]
5. # 和10,000个测试集
6. test\_size = 10000
7. X\_test = X[train\_size:train\_size + test\_size, :]
8. y\_test = y[train\_size:train\_size + test\_size]

训练阶段：

特征提取：对于手写数字识别，通常不需要手动提取特征，而是使用深度学习模型来自动学习特征。

模型选择：选择一个适当的深度学习模型，本例选择了全连接神经网络。

1. **def** neural\_network(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, lamb):

损失函数和正则化：选择适当的损失函数，通常对于多类别分类问题，使用交叉熵损失函数。正则化可以提高模型的泛化能力，可以使用正则化方法，如 L2 正则化，以减少过拟合风险。

1. **def** loss(Theta1, Theta2, y\_vect, a3, lamb, m):
2. # 计算损失值
3. # 正则化方法选择
4. # L2正则化
5. L2\_Reg = (lamb / (2 \* m)) \* (
6. np.sum(np.square(Theta1[:, 1:])) + np.sum(np.square(Theta2[:, 1:])))
7. # L1正则化
8. L1\_Reg = (lamb / (2 \* m)) \* (
9. np.sum(np.abs(Theta1[:, 1:])) + np.sum(np.abs(Theta2[:, 1:]))
10. )
11. Reg = L2\_Reg
12. # 交叉熵损失（Cross-Entropy Loss）
13. # J = (1 / m) \* (np.sum(np.sum(-y\_vect \* np.log(a3) - (1 - y\_vect) \* np.log(1 - a3)))) + Reg
14. # 均方误差损失（Mean Squared Error Loss）
15. J = (1 / (2 \* m)) \* np.sum(np.square(a3 - y\_vect)) + Reg
16. **return** J

优化算法：选择一个梯度下降的优化算法，本次实验选择了batch GD, online GD, mini-batch GD, SGD,，Momentum, Adsgrad, Adam, Admax八种优化方法用于训练模型。设置学习率和其他超参数。

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

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

模型保存：一旦训练完成并在验证集上达到满意的性能，可以保存模型以备后续的预测。

1. # 将Theta参数保存在txt文件中，用作后续程序识别
2. np.savetxt('Theta1.txt', Theta1, delimiter=' ')
3. np.savetxt('Theta2.txt', Theta2, delimiter=' ')

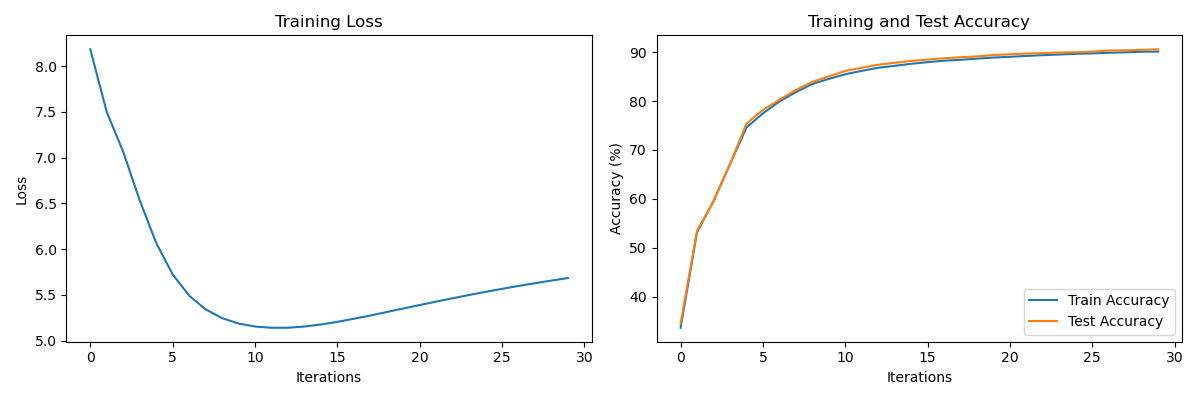
预测阶段：

模型加载：加载之前训练好的模型。

1. # 进行一次正向传播来得到结果，用于预测准确率
2. **def** predict(Theta1, Theta2, X):
3. m = X.shape[0]
4. one\_matrix = np.ones((m, 1))
5. X = np.append(one\_matrix, X, axis=1)  # 给第一层加入偏置参数
6. z2 = np.dot(X, Theta1.transpose())
7. a2 = 1 / (1 + np.exp(-z2))  # 使用Sigmoid函数激活第二层
8. one\_matrix = np.ones((m, 1))
9. a2 = np.append(one\_matrix, a2, axis=1)  # 给第二层加入偏置参数
10. z3 = np.dot(a2, Theta2.transpose())
11. a3 = 1 / (1 + np.exp(-z3))  # 激活第三层
12. p = (np.argmax(a3, axis=1))  # 输出预测的分类
13. **return** p

结果输出：将最终的识别结果呈现给用户或应用程序。

1. **def** plot\_loss\_and\_accuracy(loss\_history, train\_accuracy\_history, test\_accuracy\_history):
2. # 绘制损失曲线
3. plt.figure(figsize=(12, 4))
4. plt.subplot(1, 2, 1)
5. plt.plot(range(len(loss\_history)), loss\_history, label='Loss')
6. plt.xlabel('Iterations')
7. plt.ylabel('Loss')
8. plt.title('Training Loss')
10. # 绘制准确度曲线
11. plt.subplot(1, 2, 2)
12. plt.plot(range(len(train\_accuracy\_history)), train\_accuracy\_history, label='Train Accuracy')
13. plt.plot(range(len(test\_accuracy\_history)), test\_accuracy\_history, label='Test Accuracy')
14. plt.xlabel('Iterations')
15. plt.ylabel('Accuracy (%)')
16. plt.title('Training and Test Accuracy')
17. plt.legend()  # 添加图例
19. plt.tight\_layout()
20. plt.show()



## **3.2使用验证数据调整模型的超参数**

准备数据集：

将数据集分为三个部分：训练数据、验证数据和测试数据。通常，70-80%的数据用于训练，10-15%用于验证，10-15%用于测试。

选择并调整超参数：

确定需要调整的超参数，例如学习率、批量大小、神经网络结构、正则化参数等。逐渐改变超参数，并观察模型在验证数据上的性能。这里采用的三个超参数分别是学习率，正则化参数，迭代次数。

1. # 寻找最佳超参数
2. best\_asem = []
3. **for** alpha **in** [0.5, 0.1, 0.01, 0.001]:
4. **for** lambda\_reg **in** [0.01, 0.1, 1.0]:
5. **for** max\_iter **in** [10, 50, 100, 200]:

建立模型：

使用训练数据来训练初始模型，使用默认超参数设置或者你认为合适的初始超参数。

initial\_nn\_params = MiniBGD(initial\_nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X\_train, y\_train, lambda\_reg, max\_iter, alpha, X\_test, y\_test)

验证模型性能：

使用验证数据来评估模型的性能。这可以包括计算准确率、损失函数值等。这里我们选用准确度来验证

1. # 验证集的准确度
2. pred = predict(Theta1, Theta2, X\_test)
3. **print**(f'alpha={alpha},lambda\_reg={lambda\_reg},max\_iter={max\_iter}')
4. **print**('Accuracy: {:f}'.format((np.mean(pred == y\_test) \* 100)))

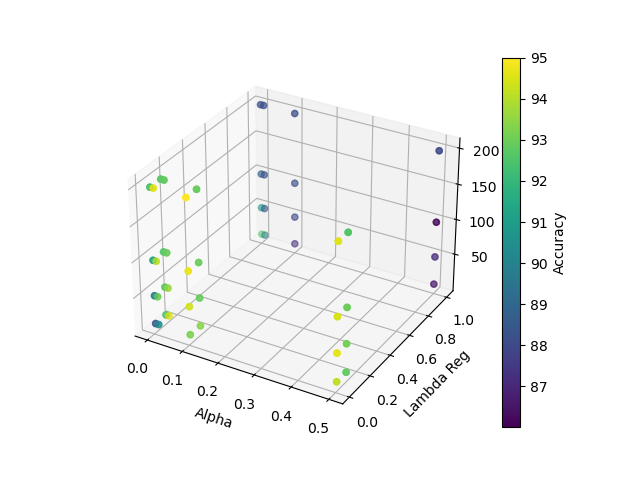
实践结果：

选用小批量梯度下降作为优化方法，依次改变学习率为：0.5，0.1，0.01，0.001；改变正则化参数为：0.01，0.1，1.0；改变迭代次数为：10，50，100，200。记录每一次得到的验证集准确率，得到以下结果：

1. alpha=0.5,lambda\_reg=0.01,max\_iter=10
2. Accuracy: 94.090000
3. alpha=0.5,lambda\_reg=0.01,max\_iter=50
4. Accuracy: 94.630000
5. alpha=0.5,lambda\_reg=0.01,max\_iter=100
6. Accuracy: 94.430000
7. alpha=0.5,lambda\_reg=0.01,max\_iter=200
8. Accuracy: 94.560000
9. alpha=0.5,lambda\_reg=0.1,max\_iter=10
10. Accuracy: 92.850000
11. alpha=0.5,lambda\_reg=0.1,max\_iter=50
12. Accuracy: 93.100000
13. alpha=0.5,lambda\_reg=0.1,max\_iter=100
14. Accuracy: 92.970000
15. alpha=0.5,lambda\_reg=0.1,max\_iter=200
16. Accuracy: 92.600000
17. alpha=0.5,lambda\_reg=1.0,max\_iter=10
18. Accuracy: 86.640000
19. alpha=0.5,lambda\_reg=1.0,max\_iter=50
20. Accuracy: 87.190000
21. alpha=0.5,lambda\_reg=1.0,max\_iter=100
22. Accuracy: 86.010000
23. alpha=0.5,lambda\_reg=1.0,max\_iter=200
24. Accuracy: 87.960000
25. alpha=0.1,lambda\_reg=0.01,max\_iter=10
26. Accuracy: 93.220000
27. alpha=0.1,lambda\_reg=0.01,max\_iter=50
28. Accuracy: 94.390000
29. alpha=0.1,lambda\_reg=0.01,max\_iter=100
30. Accuracy: 94.710000
31. alpha=0.1,lambda\_reg=0.01,max\_iter=200
32. Accuracy: 95.000000
33. alpha=0.1,lambda\_reg=0.1,max\_iter=10
34. Accuracy: 93.370000
35. alpha=0.1,lambda\_reg=0.1,max\_iter=50
36. Accuracy: 92.910000
37. alpha=0.1,lambda\_reg=0.1,max\_iter=100
38. Accuracy: 92.960000
39. alpha=0.1,lambda\_reg=0.1,max\_iter=200
40. Accuracy: 92.900000
41. alpha=0.1,lambda\_reg=1.0,max\_iter=10
42. Accuracy: 87.170000
43. alpha=0.1,lambda\_reg=1.0,max\_iter=50
44. Accuracy: 87.920000
45. alpha=0.1,lambda\_reg=1.0,max\_iter=100
46. Accuracy: 88.190000
47. alpha=0.1,lambda\_reg=1.0,max\_iter=200
48. Accuracy: 88.090000
49. alpha=0.01,lambda\_reg=0.01,max\_iter=10
50. Accuracy: 90.600000
51. alpha=0.01,lambda\_reg=0.01,max\_iter=50
52. Accuracy: 93.020000
53. alpha=0.01,lambda\_reg=0.01,max\_iter=100
54. Accuracy: 94.110000
55. alpha=0.01,lambda\_reg=0.01,max\_iter=200
56. Accuracy: 94.660000
57. alpha=0.01,lambda\_reg=0.1,max\_iter=10
58. Accuracy: 94.600000
59. alpha=0.01,lambda\_reg=0.1,max\_iter=50
60. Accuracy: 93.700000
61. alpha=0.01,lambda\_reg=0.1,max\_iter=100
62. Accuracy: 92.980000
63. alpha=0.01,lambda\_reg=0.1,max\_iter=200
64. Accuracy: 92.820000
65. alpha=0.01,lambda\_reg=1.0,max\_iter=10
66. Accuracy: 89.760000
67. alpha=0.01,lambda\_reg=1.0,max\_iter=50
68. Accuracy: 88.190000
69. alpha=0.01,lambda\_reg=1.0,max\_iter=100
70. Accuracy: 88.140000
71. alpha=0.01,lambda\_reg=1.0,max\_iter=200
72. Accuracy: 88.170000
73. alpha=0.001,lambda\_reg=0.01,max\_iter=10
74. Accuracy: 88.500000
75. alpha=0.001,lambda\_reg=0.01,max\_iter=50
76. Accuracy: 89.950000
77. alpha=0.001,lambda\_reg=0.01,max\_iter=100
78. Accuracy: 91.360000
79. alpha=0.001,lambda\_reg=0.01,max\_iter=200
80. Accuracy: 92.430000
81. alpha=0.001,lambda\_reg=0.1,max\_iter=10
82. Accuracy: 92.480000
83. alpha=0.001,lambda\_reg=0.1,max\_iter=50
84. Accuracy: 92.650000
85. alpha=0.001,lambda\_reg=0.1,max\_iter=100
86. Accuracy: 92.630000
87. alpha=0.001,lambda\_reg=0.1,max\_iter=200
88. Accuracy: 92.710000
89. alpha=0.001,lambda\_reg=1.0,max\_iter=10
90. Accuracy: 92.640000
91. alpha=0.001,lambda\_reg=1.0,max\_iter=50
92. Accuracy: 90.940000
93. alpha=0.001,lambda\_reg=1.0,max\_iter=100
94. Accuracy: 88.850000
95. alpha=0.001,lambda\_reg=1.0,max\_iter=200
96. Accuracy: 88.260000

使用更直观的绘图方式可以得到，代码和图如下：

1. alphas = [item[0] **for** item **in** data]
2. lambda\_regs = [item[1] **for** item **in** data]
3. max\_iters = [item[2] **for** item **in** data]
4. accuracies = [item[3] **for** item **in** data]
5. # 创建一个三维散点图
6. fig = plt.figure()
7. ax = fig.add\_subplot(111, projection='3d')
8. ax.scatter(alphas, lambda\_regs, max\_iters, c=accuracies, cmap='viridis', marker='o')
9. # 设置坐标轴标签
10. ax.set\_xlabel('Alpha')
11. ax.set\_ylabel('Lambda Reg')
12. ax.set\_zlabel('Max Iter')
13. # 添加颜色条
14. cbar = plt.colorbar(ax.scatter(alphas, lambda\_regs, max\_iters, c=accuracies, cmap='viridis', marker='o'))
15. cbar.set\_label('Accuracy')
16. # 显示图表
17. plt.show()



经过图标观察，最终发现，当学习率为0.01，正则化参数为0.01时，迭代次数越高，得到的验证准确率越高。所以最佳的超参数应该是学习率=0.01，正则化参数=0.01，迭代次数=200。

## 3.3使用不同的更新方法优化神经网络

### 3.3.1传统的四种方法

我倾向于将BGD,SGD,OGD,MiniBGD四种方法归于一类，它们改变的是每一次迭代中，每一次进行梯度计算的数据量和数据顺序。

从数据量上来说，最基础的方法BGD采用每次迭代中，计算整个训练数据集的损失和梯度的方式，OGD则在迭代中，每一个样本都用于更新参数，MiniBGD则是每次迭代只计算一个小批量的损失和梯度，用于更新参数。

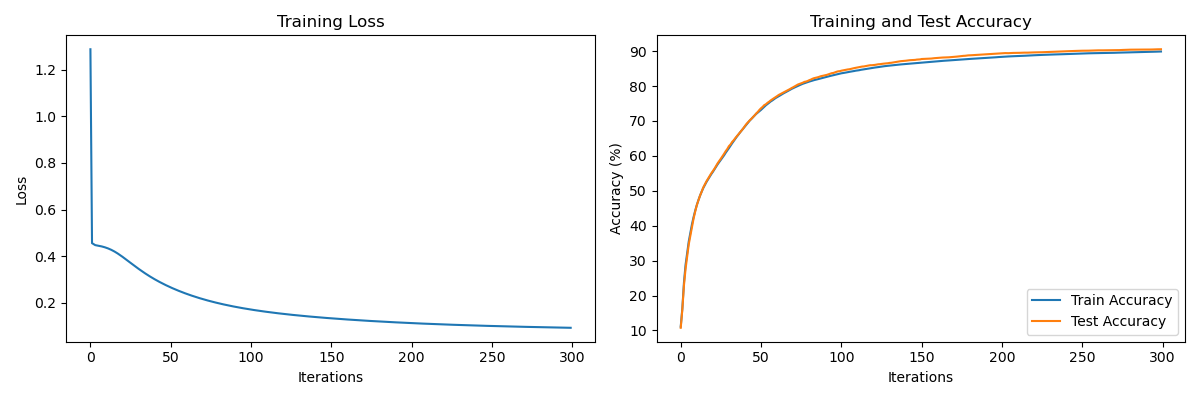
顺序而言，BGD由于每次迭代是对整个数据集进行操作，不存在顺序。OGD则是按照顺序更新，SGD是将顺序打乱，而后进行更新。

查阅资料发现在实际使用中，已无特别严格的定义限制，例如MiniBGD的样本顺序是否能打乱，SGD是否要对每一个样本更新，还是可以按批次更新。

我通过实验，分别得到了四种更新方法相对优秀的超参数，和在优秀超参数下，他们的准确率和损失变化曲线。（因为时间问题，来不及跑每一种方法的最优超参数了）

BGD（Batch Gradient Descent）:

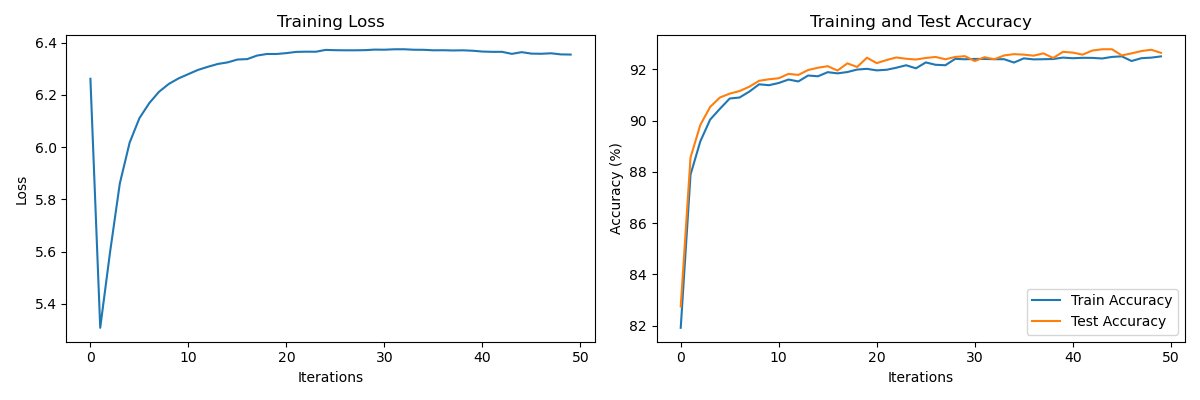
学习率=0.9，正则化参数=0.1，迭代次数=300



需要较大的学习率，在大学习率下效果好但低学习率下效果一般。这可能是因为其更新数据集较大的缘故。

MiniBGD (Mini-Batch Gradient Descent):

学习率=0.1，正则化参数=0.1，迭代次数=50



相较于BGD，MiniBGD能以比BGD快得多的速度迅速到达很高的正确率，通过损失函数的变化发现，在前几次迭代之后，就已经达到了损失函数的最低值，之后反而损失函数在增大。

### 3.3.2新的四种改进方法

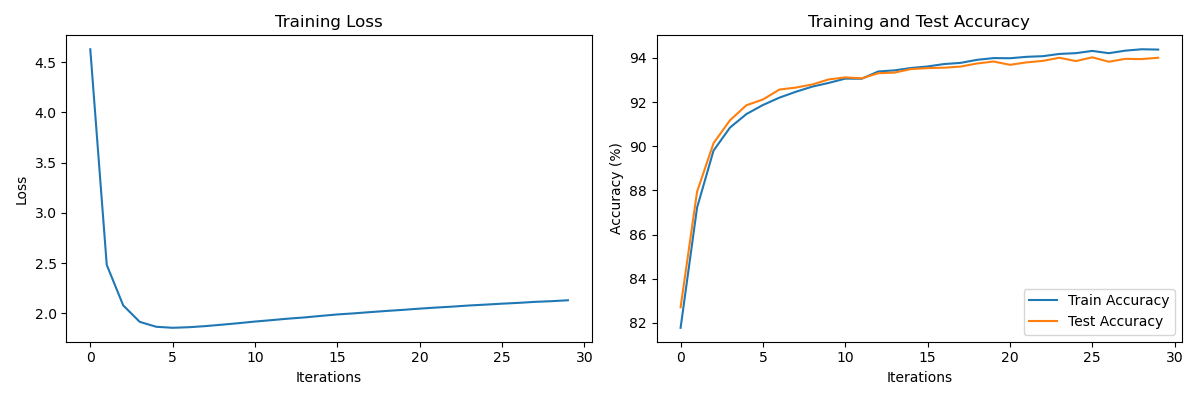
新的四种方法通过引入新的参数和技巧，对更新方式进行了优化，因此我在与原本小批量梯度下降方法的基础上，进行更改，得到了这四种新的方法。

Momentum:

参数更新方式：动量优化

引入动量概念，通过加速梯度更新的方向，减小了收敛过程中的抖动。

有助于加速收敛，特别是在曲率变化较大的情况下。



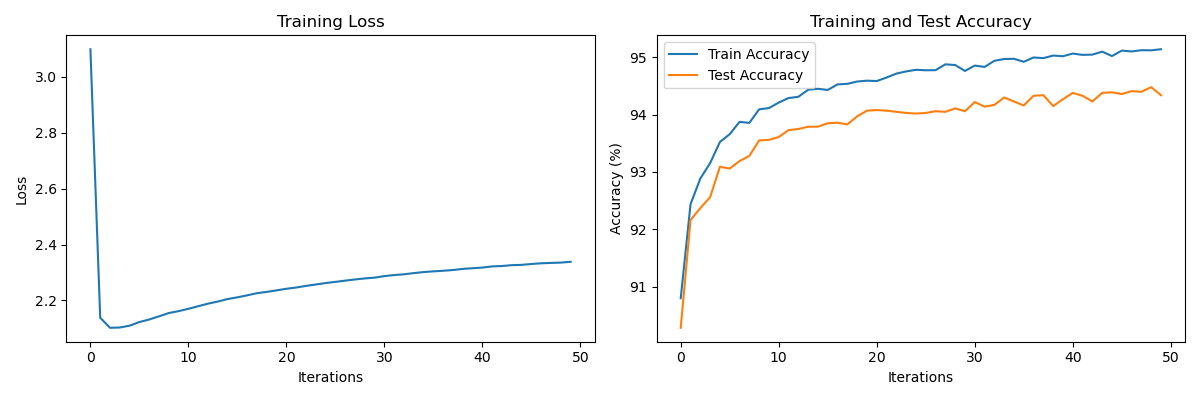
通过实验结果可以发现其收敛速度确实有了较大的提升，仅在30此循环以内就明显看到收敛趋势。

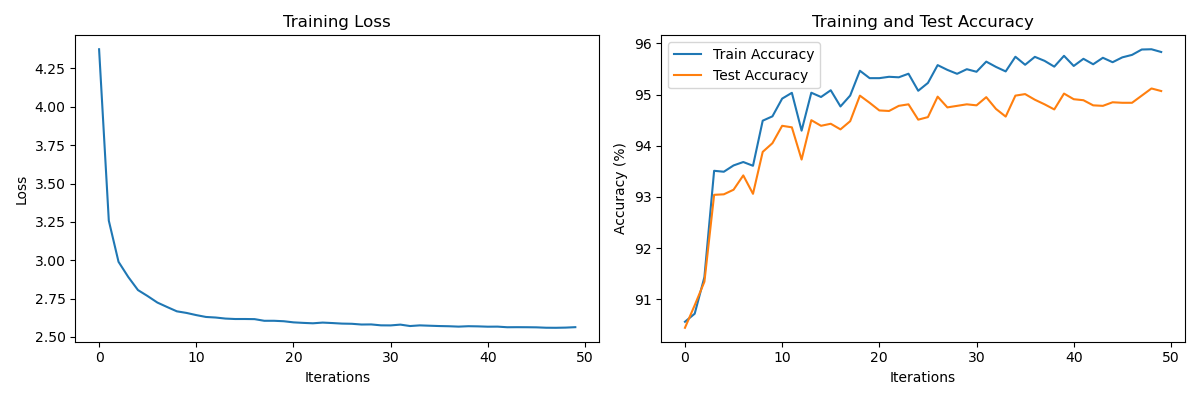
Adagrad (Adaptive Gradient Descent):

参数更新方式：自适应梯度下降

自适应学习率，每个参数都有自己的学习率，适应了不同参数的重要性。

适用于非常稀疏的数据集，但可能随时间降低学习率过快。





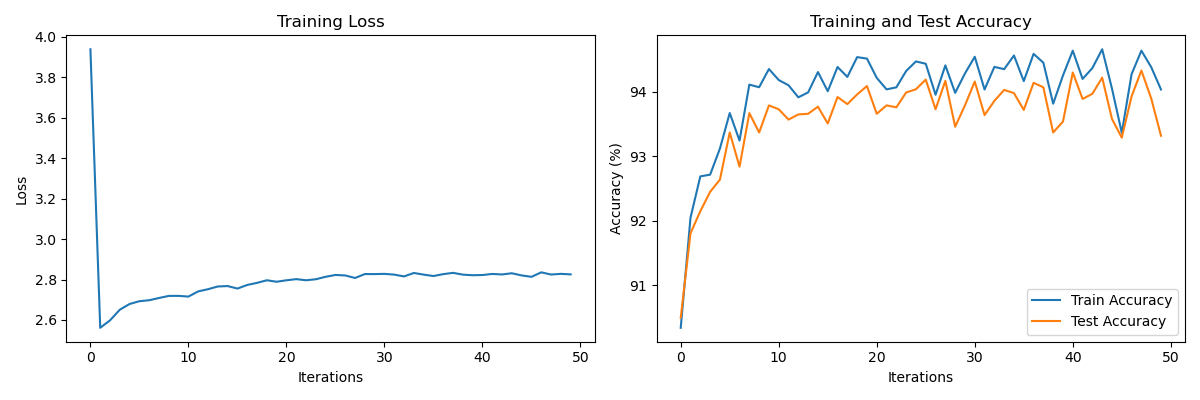
这种方法训练后测试集准确度明显低于训练集，可能是适用于较为稀疏的数据集，而本数据集相对密集，过拟合较为明显。

Adam (Adaptive Moment Estimation):

参数更新方式：自适应矩估计

结合了动量和自适应学习率的优点，适应不同参数的梯度。

通常表现良好，是常用的优化算法之一。

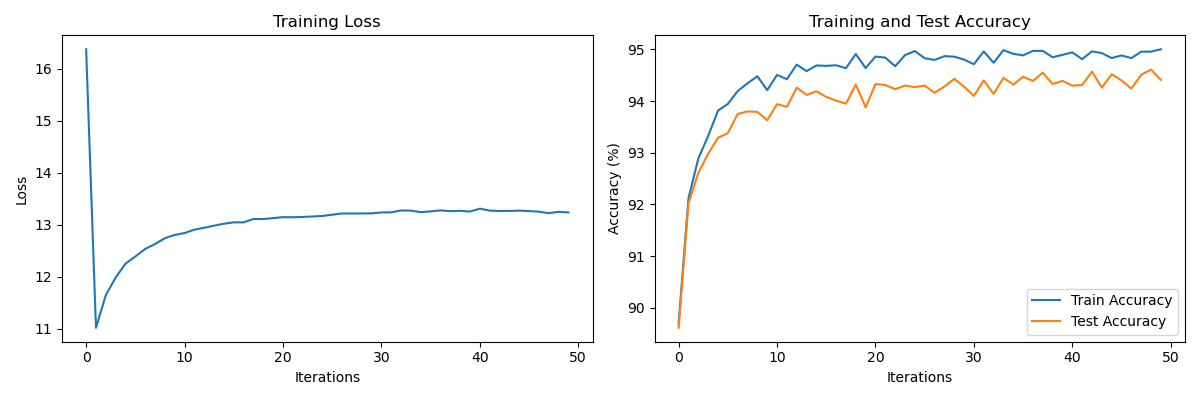


Adamax:

参数更新方式：自适应最大值

类似于Adam，但使用无穷范数（绝对值最大）来进行梯度估计。

适用于具有稀疏梯度的问题。



## 3.4使用不同的损失函数和正则化方法

### 3.4.1正则化方法

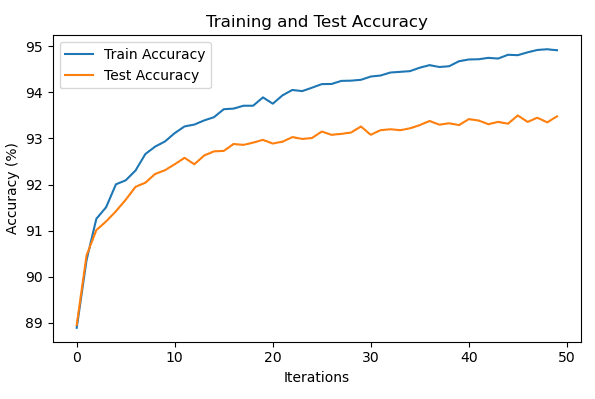
正则化主要用于改善模型的泛化性能和防止过拟合。过拟合是指模型在训练数据上表现得非常好，但在未见过的数据上表现较差的问题。正则化的主要目的是控制模型的复杂性，使其更好地适应未见过的数据，同时减少对训练数据的过度拟合。

由此我希望通过加入正则化，改变正则化参数，从而提高测试集的准确度，为此，我尝试通过改变正则化参数，观察在不同参数下，测试机准确度是否得到提升。

在使用Adagrad更新方法时，过拟合较为严重，故选用其作为实验样本。

学习率=0.1，迭代次数=50的超参数下，我们来改变正则化参数。

正则化参数=0时（没有正则化）



正则化参数=0.1时：



可以发现，进行正则化后，测试集的准确率明显提升，并高出了训练集的准确率，这体现了正则化对泛化性优秀的提升能力，减少了对训练数据的过拟合。

### 3.4.2损失函数

本次实验选用了常见的交叉熵损失和均方误差损失两种损失函数。

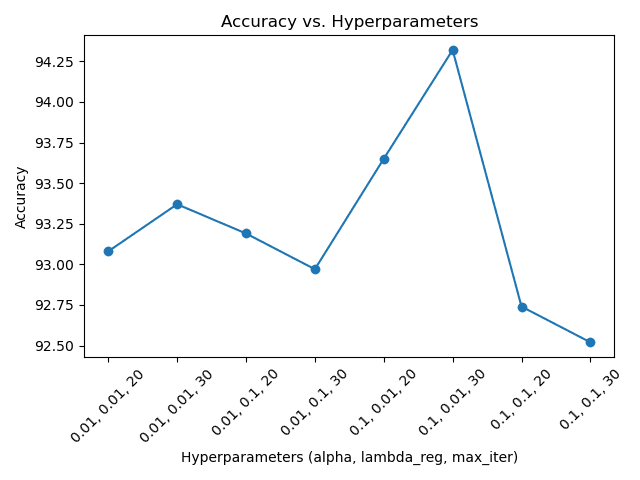
交叉熵损失（Cross-Entropy Loss）：通常用于分类问题，对于多类别分类问题表现良好。它惩罚分类错误的程度较大，因此在分类任务中通常能够更好地推动神经网络学习，特别是当输出类别是one-hot编码时。

均方误差损失（Mean Squared Error Loss）：通常用于回归问题，对于连续数值的输出预测较为适用。它衡量了预测值与真实值之间的平方差异，因此更适合回归任务。

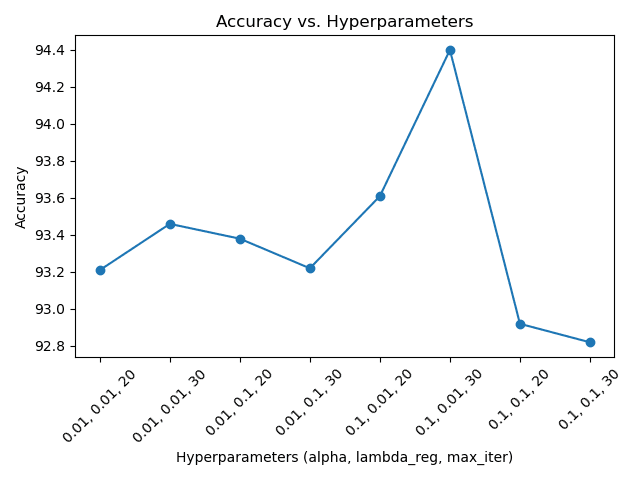
选择哪种损失函数通常取决于问题的性质。如果问题是分类问题，交叉熵损失通常更适合。如果是回归问题，均方误差损失可能更适合。

为了对比两种损失函数，我选择了MiniBatch作为优化方法，控制相同的超参数，对比两种损失函数训练的效果。

使用交叉熵损失函数：



使用均方误差损失函数：

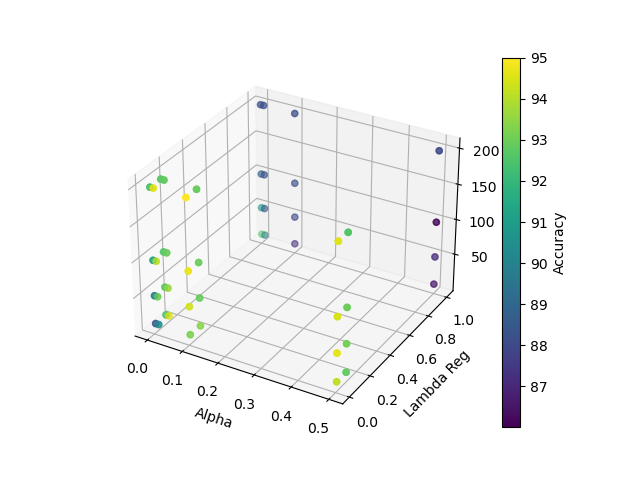


观察发现，使用两种损失函数得到的实验结果区别并不大，均方误差损失得到的准确度比使用交叉熵损失函数得到的准确度略高。

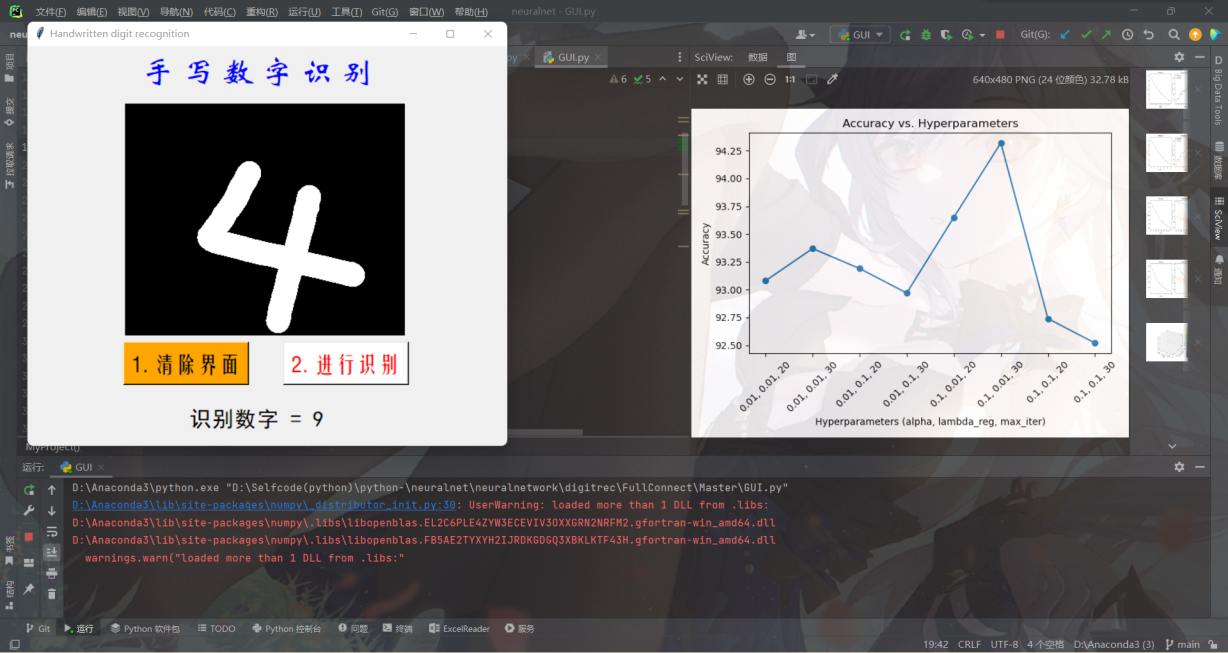
# 4.经验总结

本次实验从基础开始搭建神经网络结构，对神经网络结构上有了更清晰的认识，神经网络工作流程，更新方式有了清晰的了解。不过具体数学公式推导上仍然有待加强。对于通过调整超参数来提高神经网络能力上有了更多的体会，当使用SGD时准确率总是很低，通过调整学习率来使得10%左右的准确率一下子上升起来令人十分兴奋。同时学习到通过画散点图直接寻找到最优超参数的方法十分的便捷。

实验中最令我开心的两个方面，一个是寻找最优参数的三维散点图：



另一个就是通过保存的数据实现一个GUI，能切实感受到自己训练结果的准确与否：



如图这个结果就不大准确...